



Bilgisayar Mimarisi

RISC-V Tabanlı Yaklaşım

Prof. Dr. Oğuz Ergin

Sürüm 0.4.5 · 2026

Bilgisayar Mimarisi — RISC-V Tabanlı Yaklaşım

Prof. Dr. Oğuz Ergin

Sürüm 0.4.5, 2026

ISBN: 978-625-00-6081-0

Bu kitap, lisans düzeyinde bilgisayar mimarisi derslerinde ders kitabı olarak kullanılmak üzere hazırlanmıştır.

Kitaptaki tüm mimari eser fotoğrafları Wikimedia Commons'tan Creative Commons (CC BY-SA) lisansı altında alınmıştır.

Tüm hakları saklıdır.



*Vatan uğruna can veren, bu ülkenin
gerçek mimarlarının anısına ...*

Önsöz

Bu kitap, lisans düzeyindeki bilgisayar mimarisi derslerinde kaynak kitap olarak kullanılmak üzere hazırlanmıştır. Kitabın amacı, bilgisayar mühendisliği ve ilgili mühendislik dallarındaki öğrencilere, çağdaş bilgisayar sistemlerinin temel yapısını ve çalışma prensiplerini Türkçe olarak sunmaktır.

Kitap boyunca RISC-V buyruk kümesi mimarisi temel alınmıştır. RISC-V, açık kaynaklı ve modüler yapısıyla hem akademik hem de endüstriyel alanda giderek yaygınlaşan çağdaş bir mimaridir. Bu tercih, öğrencilerin güncel ve geleceğe yönelik bir mimari üzerinden kavramları öğrenmesini sağlamaktadır.

İlk bölümde bilgisayarların tarihsel gelişimi ve temel kavramlar ele alınmaktadır. İkinci bölümde başımlı değerlendirilmesi, üçüncü bölümde buyruk kümesi mimarisi, dördüncü bölümde RISC-V ile programlama, beşinci bölümde aritmetik işlemler, altıncı bölümde işlemci tasarımı ve yedinci bölümde boru hattı yöntemleri anlatılmaktadır. Sekizinci bölüm bellek hiyerarşisine, dokuzuncu bölüm giriş/çıkış sistemlerine ayrılmıştır. Onuncu bölümde çok çekirdekli işlemciler, on birinci bölümde GPU ve hızlandırıcılar, on ikinci bölümde ise güncel mimari eğilimleri tartışılmaktadır. Ek A'da sayı sistemleri ve kodlama, Ek B'de sayısal tasarım temelleri, Ek C'de RISC-V referans kartı, Ek D'de ise uygulamalı çalışma rehberi ve mini projeler sunulmaktadır.

Türkiye'de bilgisayar mimarisi alanında Türkçe kaynak üretme çabası uzun bir geçmişe sahiptir. Eşref Adalı'nın *Mikroişlemciler Mikrobilgisayarlar* [1] kitabı, M. Morris Mano'nun Türkçeye çevrilen *Bilgisayar Sistemleri Mimarisi* [23] eseri, Mehmet Bodur'un EMO yayını olan *Bilgisayar Organizasyonu: RISC Donanımına Giriş* [4] kitabı, Şirzat Kahramanlı'nın *Bilgisayar Mimarisi* [19] eseri, Patterson ve Hennessy'nin Türkçeye kazandırılan *Bilgisayar Mimarisi ve Tasarım* [34] çevirisi ve Cengiz Uğurkaya ile Osman Aliefendioğlu'nun *Çağdaş Bilgisayar Mimarisi* [42] kitabı bu alandaki öncü çalışmalardır. Bu kitap, söz konusu değerli çabaların bir sonraki adımı olarak RISC-V buyruk kümesi mimarisi üzerine odaklanmakta ve açık erişim felsefesiyle Türkçe teknik içerik birikimine katkı sunmayı amaçlamaktadır.

Her bölümün başında, Türkiye'nin mimari mirasından bir eser fotoğrafı yer almaktadır. Bu eserler, bilgisayar mimarisi kavramlarıyla tematik bir bağ kurularak seçilmiştir.

Bu kitabın hazırlanmasında destek ve katkılarından dolayı Kasırğa araştırma takımındaki öğrencilerime teşekkür ederim.

Oğuz Ergin
Ankara, 2026

İçindekiler

Önsöz	v
Şekil Listesi	xviii
Tablo Listesi	xxi
Bilgi Notları Listesi	xxiii
1 Bilgisayar Mimarisine Giriş	1
1.1 Giriş	2
1.2 Bilgisayarların Gelişimi	2
1.2.1 Moore Yasası ve Transistör Artışı	5
1.3 Bilgisayar Nedir?	7
1.3.1 Bilgisayarın Parçaları	8
1.3.2 Gerçek Dünyayla Etkileşen Bilgisayarlar	9
1.4 İşlemci ve Yonga Üretimi	10
1.4.1 İşlemcinin Yapısı	11
1.4.2 Tümüleşik Devreler	12
1.4.3 Yonga Üretim Süreci	12
1.4.4 Teknoloji Düşümleri ve Gelişimi	14
1.4.5 Üretim Maliyeti ve Verim	15
1.5 Programların Çalıştırılması	17
1.5.1 Makine Dili ve Çevirici Dil	17
1.5.2 Von Neumann ve Harvard Mimarileri	18
1.5.3 RISC ve CISC Tasarım Felsefeleri	19
1.5.4 Üst Düzey Diller ve Derleme Zinciri	20
1.5.5 Yazılım Katmanları	21
1.5.6 Buyruk Yürütüm Döngüsü	22
1.6 Bilgisayar Mimarisini Şekillendiren Etkenler	23
1.6.1 Dört Tasarım İlkesi	24
1.7 Güç Tüketimi Eğilimleri	25
1.7.1 Devingen Güç Tüketimi	26
1.7.2 Durağan Güç Tüketimi	28
1.7.3 Güç Tüketimini Azaltma Yöntemleri	28
1.8 Güvenilirlik	29
1.8.1 Arıza, Hata ve Bozukluk	30

1.8.2	Bozukluk Türleri	30
1.8.3	Güvenilirlik Ölçütleri	30
1.8.4	Güvenilirlik, Dayanıklılık ve Gürbüzlük	32
1.9	Neden RISC-V?	33
1.10	Bilgisayar Mimarisinin Yeni Ufku: Yapay Zekâ	35
1.10.1	Genel Amaçlıdan Alana Özgü Mimarilere	36
1.10.2	Yapay Zekâ Hızlandırıcıları: GPU, TPU, NPU	36
1.10.3	Mimari Eğitimine Etkisi	37
1.11	Yanılığlar ve Tuzaklar	37
1.12	Özet	38
	Alıştırmalar	39
2	Bilgisayar Başarımı	45
2.1	Gereksinim Belirleme	47
2.1.1	Gereksinimden Ölçüme Giden Yol	47
2.1.2	Uygulama Alanlarına Göre Gereksinimler	48
2.2	Başarım Ölçütleri	49
2.2.1	Yürütme Zamanı	49
2.2.2	Saat Sıklığı ve Çevrim Zamanı	51
2.2.3	Buyruk Sayısı	53
2.2.4	Buyruk Başına Çevrim (BBC)	56
2.2.5	Hepsini Bir Araya Getirelim: Yürütme Zamanı Denklemi	57
2.2.6	İşlem Hacmi ve Gecikme	59
2.2.7	Başarımı Etkileyen Etkenler	60
2.2.8	MIPS ve MFLOPS	61
2.2.9	FLOPS ve Türevleri	63
2.2.10	Yapay Zeka İş Yüklerinde Başarım	64
2.2.11	Karşılaştırmalı Başarım Ölçümü	65
2.3	Standart Sınama Programı Kümeleri	69
2.3.1	SPEC	69
2.3.2	SPEC Puanı Nasıl Hesaplanır?	71
2.3.3	Diğer Sınama Programı Kümeleri	73
2.4	Başarım Eğilimleri: Geçmişten Günümüze	73
2.4.1	Tek Çekirdek Başarımının Altın Çağı (1980–2005)	73
2.4.2	Güç Duvarı ve Başarım Duraksaması (2005–günümüz)	74
2.4.3	Günümüz: Verimlilik ve Uzmanlaşma Çağı	75
2.5	Amdahl Yasası	76
2.5.1	Gustafson Yasası	79
2.5.2	Heterojen Sistemlerde Başarım	80
2.6	Bellek Duvarı	83
2.7	Güç-Başarım Dengesi	84
2.8	Gerçek Dünya: İşlemci Karşılaştırmaları	87
2.9	Yanılığlar ve Tuzaklar	88
2.10	Özet	89
	Alıştırmalar	90

3	Buyruk Kümesi Mimarisi	101
3.1	Giriş	102
3.2	Buyruk Kümesi Tasarımı	103
3.2.1	BKM'nin Kısa Tarihçesi	104
3.2.2	Buyruk Sayısı ve İşlem Kodu Uzayı	105
3.3	RISC-V İşlemcisi ve Buyruk Kümesi	106
3.3.1	Yazılım Açısından Buyruk Kümesi	108
3.3.2	Donanım Açısından Buyruk Kümesi	109
3.4	Yazmaçlar	110
3.4.1	Yazmaç Türleri	111
3.4.2	RISC-V'te Kullanılan Yazmaçlar	111
3.4.3	Özel Amaçlı Yazmaçlar	112
3.4.4	Sıfır Yazmacı: x0 Neden Hep Sıfırdır	116
3.4.5	RISC-V'te Neden Bayrak Yazmacı Yoktur?	116
3.4.6	Yazmaç Sayısı ve Tasarım Dengeleri	117
3.4.7	Yazmaçlarda Veri Gösterimi	123
3.5	Bellek Düzeni	124
3.5.1	Sözcük Kavramı	124
3.5.2	Belleğe İki Tür Erişim	126
3.5.3	Bellek Hizalama	126
3.5.4	Verilerin Bellekte Yerleşimi: Bayt Sırası	127
3.6	Buyruklar	129
3.6.1	Buyrukların Yapısı	129
3.6.2	Aynı İşlem, Farklı Biçim: add ve addi Neden Ayrı Buyruklardır	130
3.6.3	Buyruk Türleri	130
3.7	Veri Aktarma Buyrukları	132
3.8	Aritmetik İşlem Buyrukları	134
3.8.1	Çarpma ve Bölme: Geniş Sonuç Sorunu	134
3.9	Mantık Buyrukları	136
3.9.1	DEĞİL İşlemi ve Mimariler Arası Farklılıklar	137
3.10	Bit Düzeyinde Kaydırma Buyrukları	138
3.11	Dallanma ve Atlama Buyrukları	138
3.11.1	Koşullu Dallanma Buyrukları	139
3.11.2	Koşulsuz Atlama Buyrukları	141
3.12	Sözde Buyruklar	143
3.13	Adresleme Kipleri	144
3.13.1	Yazmaç Adresleme Kipi	145
3.13.2	Anlık Adresleme Kipi	146
3.13.3	Eklemeli Adresleme Kipi (Taban Adresleme)	146
3.13.4	PS'ye Göreceli Adresleme Kipi	146
3.13.5	Üst Anlık Adresleme Kipi	147
3.14	Altyordam ve Yığıtlar	150
3.14.1	RISC-V'te Altyordamların Kullanılması	150
3.14.2	İç İçe Geçmiş Çağrılar	150
3.14.3	Çağırın ve Çağrılan Tarafın Sorumlulukları	151
3.14.4	Yığıt Çerçevesi: Giriş ve Çıkış Kodu	151

3.15	Programın Yürütülmesi	154
3.15.1	Derleyici	154
3.15.2	Çevirici	154
3.15.3	Bağlayıcı	155
3.15.4	Yükleyici	155
3.15.5	Bellek Düzeni ve Bölümler	155
3.15.6	Sistem Çağrılar ve ecall	156
3.16	Diğer BKM'lerde Buyruk Kümeleri	157
3.16.1	MIPS Buyruk Kümesi	157
3.16.2	ARM Buyruk Kümesi	157
3.16.3	x86 Buyruk Kümesi	158
3.16.4	Dört Mimarinin Genel Karşılaştırması	160
3.17	Diğer Bazı Buyruk Kümesi Mimarileri	160
3.18	Gerçek Dünya: Buyruk Kümesi Tasarım Kararları	161
3.19	Yanılığlar ve Tuzaklar	163
3.20	Özet	165
	Alıştırmalar	166
4	RISC-V ile Programlama	171
4.1	İlk Adımlar	172
4.1.1	Yazmaçlarla Aritmetik	172
4.1.2	Bellekten Okuma ve Yazma	174
4.1.3	Yığıt Kullanımı	176
4.1.4	Döngüler	179
4.2	Program Yapısı ve Geliştirme Ortamı	183
4.2.1	Bir RISC-V Programının Anatomisi	183
4.2.2	Geliştirme Ortamı	184
4.3	Algoritmalar ve Veri Yapıları	185
4.3.1	Dizilerde Doğrusal Arama	185
4.3.2	Kabarcık Sıralama	186
4.3.3	Karakter Dizisi İşleme	188
4.3.4	Yapılar ve Bellek Düzeni	190
4.3.5	Bağlı Liste	191
4.3.6	Özyineleme	192
4.3.7	Çok Fonksiyonlu Programlar	194
4.4	C'den RISC-V'e: Derleme Çıktısı Okuma	197
4.4.1	Derleyici Çıktısı Elde Etme	197
4.4.2	Eniyileme Seviyelerinin Etkisi	197
4.4.3	Yapısal Dönüşümler	198
4.5	Gömülü Sistem Programlama	200
4.5.1	Bellek Eşlemeli Giriş/Çıkış	200
4.5.2	UART ile Seri İletişim	201
4.6	Başarım ve Eniyileme	202
4.6.1	Buyruk Sayımı ve Döngü Gövdesi Çözümlemesi	202
4.6.2	Döngü Açma	202
4.6.3	Matris İşlemleri ve Bellek Erişim Kalıpları	203

4.7	Hata Ayıklama	204
4.7.1	Yaygın Hatalar	205
4.7.2	RARS ile Adım Adım Yürütme	206
4.8	Gerçek Dünya: RISC-V Geliştirme Ekosistemi	207
4.8.1	Geliştirme Kartları	207
4.8.2	Araç Zinciri	207
4.9	Yanılıgılar ve Tuzaklar	208
4.10	Özet	209
	Alıştırmalar	210
5	Aritmetik İşlemler	215
5.1	Toplama ve Çıkarma	216
5.1.1	Tek Bitlik Toplama	217
5.1.2	Çıkarma	217
5.1.3	32 Bit ile Çıkarma	219
5.1.4	Taşma	219
5.1.5	RISC-V Aritmetik Buyrukları ve Taşma	220
5.2	Mantık İşlemleri	220
5.2.1	Kaydırma (Shift) İşlemi	221
5.3	AMB (Aritmetik Mantık Birimi)	223
5.3.1	AMB ile Toplama	223
5.3.2	AMB ile Çıkarma	225
5.3.3	AMB ile Mantık İşlemleri	227
5.3.4	Bir Bitlik AMB	227
5.3.5	32 Bitlik AMB	228
5.3.6	Dallanma Buyruklarının Desteklenmesi	229
5.3.7	Eldenin Önceden Üretilmesi (Carry Lookahead)	230
5.3.8	Elde ile Seçmeli Toplayıcı (Carry Select Adder)	233
5.4	Çarpma	235
5.4.1	Kaydır-Topla Çarpma (İşaretsiz Sayılar)	237
5.4.2	Booth Algoritması (İşaretili Sayılar)	243
5.4.3	Değiştirilmiş Booth Kodlaması (Radix-4)	245
5.4.4	Koşut Çarpıcılar	246
5.4.5	İşlenenlerden Birinin Sıfır Olması	251
5.5	Bölme	251
5.5.1	Geri Yükleme Bölme	252
5.5.2	Geri Yükleme Bölme	259
5.5.3	SRT Bölme	260
5.5.4	Bölme Yöntemlerinin Karşılaştırması	261
5.5.5	RISC-V Bölme Buyrukları	262
5.5.6	Sıfıra Bölme ve Taşma	263
5.6	Kayan Nokta İşlemleri	264
5.6.1	Sabit Nokta ve Kayan Nokta	265
5.6.2	Bilimsel Gösterim ve Olağanlaştırma	266
5.6.3	IEEE 754 Kayan Nokta Standardı	266
5.6.4	Kayan Nokta Özel Değerleri	272

5.6.5	Yuvarlama	274
5.6.6	Kayan Nokta İşlemlerinde Toplama, Çarpma ve Bölme	277
5.7	RISC-V Kayan Nokta Buyrukları	282
5.8	Yapay Zeka Çağında Kayan Nokta Biçimleri	284
5.8.1	FP16: IEEE 754 Yarım Duyarlı	285
5.8.2	BF16: Google Brain'den Doğan Biçim	285
5.8.3	TF32: NVIDIA'nın Uzlaşısı	286
5.8.4	FP8: Çıkarım İçin En Dar Biçim	286
5.8.5	Karma Duyarlık Eğitimi	288
5.8.6	Karşılaştırma	288
5.9	Yanılığlar ve Tuzaklar	289
5.10	Gerçek Dünyada Aritmetik	291
5.10.1	Yardımcı İşlemci Dönemi (1980'ler)	291
5.10.2	Tümleşik Kayan Nokta Birimi (1989–)	291
5.10.3	Günümüz: SIMD, Vektör ve Matris Birimleri	291
5.11	Özet	292
	Alıştırmalar	293
A	Sayı Sistemleri ve Kodlama	603
A.1	İkilik Tabanda İşlemler	604
A.1.1	Taban Değiştirme İşlemleri	604
A.1.2	İkilik Tabanda Dört İşlem	606
A.1.3	8'lik ve 16'lık Tabanda İşlemler	607
A.2	İşaretli Sayılar	608
A.2.1	Ayrı İşaret Biti	609
A.2.2	Artık N	609
A.2.3	Bire Tümleyen Gösterimi	609
A.2.4	İkiye Tümleyen Gösterimi	610
A.2.5	Taşma Tespiti	612
A.3	Gray Kodu	614
A.4	İkiye Kodlanmış Onluk Taban Gösterimi	615
A.5	Karakter Kodlaması	616
A.5.1	ASCII	616
A.5.2	Unicode ve UTF-8	616
A.6	IEEE 754 Kayan Nokta Gösterimi	617
A.6.1	Tek Duyarlı Gösterim (32 bit)	618
A.6.2	IEEE 754 Duyarlık Düzeyleri	618
A.6.3	Özel Değerler	619
A.7	Yanılığlar ve Tuzaklar	621
	Özet	623
	Alıştırmalar	623
B	Sayısal Tasarım Temelleri	625
B.1	Giriş	626
B.2	Mantık Kapıları ve Doğruluk Tabloları	627
B.2.1	İkilik Sistem ve Mantık	627
B.2.2	Temel Kapılar ve Devre Sembolleri	628

B.2.3	Evrensel Kapılar: VE-DEĞİL ve VEYA-DEĞİL	629
B.3	Boole Cebri	629
B.3.1	De Morgan Kuramları	631
B.3.2	Mini Terimler ve Maksi Terimler	633
B.4	Boole Fonksiyonlarının Sadeleştirilmesi	634
B.4.1	Karno Haritası Yöntemi	635
B.4.2	Önemsenmeyen Durumlar	640
B.4.3	Quine–McCluskey Algoritması	640
B.5	Birleşik Yapı Taşları	643
B.5.1	Toplayıcılar	643
B.5.2	Kod Çözücü	645
B.5.3	Kodlayıcı	648
B.5.4	Çoklayıcı	650
B.5.5	Tekleyici	654
B.6	Flip-Flop	656
B.6.1	RS Flip-flop	657
B.6.2	D Flip-flop	657
B.6.3	T Flip-flop	658
B.6.4	JK Flip-flop	659
B.7	Yazmaçlar	659
B.7.1	Koşut Yükleme Yazmaç	659
B.7.2	Kaydırmalı Yazmaç	660
B.8	Sayaçlar	661
B.8.1	Eşzamanlı İkili Sayaç	661
B.8.2	Eşzamansız (Dalgalı) Sayaç	662
B.9	Mealy ve Moore Makineleri	663
B.9.1	İki Model Arasındaki Farklar	663
B.9.2	Blok Diyagramları	664
B.9.3	Durum Diyagramı Gösterimi	664
B.9.4	Kullanım Alanları ve Tercih Kriterleri	665
B.9.5	Ardışıl Devre Tasarımı Adımları	665
B.10	Yanılıgılar ve Tuzaklar	672
	Özet	674
	Alıştırmalar	675
C	RISC-V Referans Kartı	677
C.1	RISC-V'in Modüler Yapısı	678
C.1.1	Taban Buyruk Kümeleri	678
C.1.2	Standart Uzantılar	678
C.1.3	Yaygın Yapılandırmalar	679
C.1.4	Ayrıcalık Düzeyleri	679
C.2	Bir RISC-V Buyruğunun Anatomisi	680
C.2.1	Genel Yazım Kuralı	680
C.2.2	İşlenen Türleri	680
C.2.3	Bellek Adresleme: Parantez Gösterimi	680
C.2.4	Buyruk Kategorileri	681

C.3	RV32I Temel Buyruk Kümesi	681
C.3.1	Buyruk Kodlama Örneği	683
C.4	Yazmaç Tablosu	683
C.4.1	Çağrı Kuralları ve Yığıt Çerçevesi	684
C.5	Buyruk Biçimleri	685
C.6	Sözde Buyruklar	686
C.7	Kodlama Örnekleri	687
	Özet	689
	Alıştırmalar	690
D	RISC-V Uygulama Çalışmaları ve Projeler	693
D.1	RISC-V Araç Zinciri	694
D.1.1	Benzetimlikler	694
D.1.2	GNU Araç Zinciri	695
D.1.3	Derleme–Bağlama–Çalıştırma Akışı	696
D.2	Verilog ile Donanım Tasarımı	696
D.2.1	Modül Yapısı	697
D.2.2	Veri Tipleri: wire ve reg	697
D.2.3	Sürekli Atama ve always Blokları	697
D.2.4	Doğrulama Ortamı Yazma	698
D.3	Yazmaç Öbeği Tasarımı	699
D.4	Mini Projeler	700
D.4.1	Proje A: RV32I Yorumlayıcı	700
D.4.2	Proje B: Boru Hattı Benzetimliği	701
D.4.3	Proje C: Önbellek ve Dallanma Öngörücüsü	702
D.5	Proje Değerlendirme Ölçütleri	702
	Alıştırmalar	703
	Terimler Sözlüğü	705
	Türkçe – İngilizce	705
	İngilizce – Türkçe	714
	Yazar Hakkında	728

Şekil Listesi

Bölüm 1 – Bilgisayar Mimarisine Giriş

1.1	Blaise Pascal’ın 1642’de yaptığı mekanik hesap makinesi (Pascaline)	3
1.2	Intel işlemcilerdeki transistör sayısının artışı	6
1.3	Bilgisayarın 5 ana bileşeni	7
1.4	Farklı bilgisayar türleri	9
1.5	Gerçek dünyayla etkileşen bilgisayar örnekleri	10
1.6	Bilgisayar sistemindeki soyutlama katmanları	11
1.7	Yonga yapılma süreci	12
1.8	Bir silisyum plakadan yongaların kesilmesi	13
1.9	Von Neumann ve Harvard mimarileri	19
1.10	Bir üst düzey programın alt düzey denetim işaretlerine çevrim aşamaları . . .	21
1.11	Yazılım katmanları ve donanım arasındaki ilişki	22
1.12	Buyruk yürütüm döngüsü	23
1.13	Bilgisayar Mimarisini Şekillendiren Etkenler	23
1.14	İşlemcilerin TDP değerlerinin tarihsel gelişimi	26
1.15	x86, ARM ve RISC-V buyruk kümesi mimarilerinin karşılaştırması	34
1.16	RISC-V çekirdek sayısının yıllara göre büyümesi	35
1.17	Bilgisayar mimarisinde paradigma değişimi	36

Bölüm 2 – Bilgisayar Başarımı

2.1	Gereksinimden ölçüme giden üç aşamalı döngü	47
2.2	İşlemci saat darbeleri: çevrim zamanı (T) ve saat sıklığı ($f = 1/T$)	51
2.3	Gecikme ve işlem hacmi karşılaştırması	60
2.4	Aritmetik ve geometrik ortalama karşılaştırması	68
2.5	İşlemci başarımlar eğilimleri	75
2.6	Amdahl Yasası’nın etkisi	77
2.7	Amdahl Yasası hızlanma eğrileri	79
2.8	Günümüz tek yonga blok diyagramı	81
2.9	İşlemci ve bellek başarımlar artış hızları arasındaki açılan makas (Bellek Duvarı)	83
2.10	Farklı işlemci türlerinin güç-başarım düzlemindeki konumu	86

Bölüm 3 – Buyruk Kümesi Mimarisi

3.1	Buyruk Kümesi Mimarisi (BKM) ile Yazılım ve Donanımın İlişkisi	103
3.2	R-tipi buyruğun kodlanması	108
3.3	RISC-V’te veri yapıları	109
3.4	D türü flip-flop	110

3.5	8 bitlik veri tutan yazmaç (D flip-floplardan oluşan)	111
3.6	RISC-V yazmaç türleri	116
3.7	16 bitlik buyruklarda yazmaç sayısının etkisi (Y türü buyruklar)	122
3.8	Bellek yapısının iki gösterimi	125
3.9	Bellekte hizalı ve hizasız veri erişimi	127
3.10	Büyüğü başta ve küçüğü başta bayt sırası	128
3.11	RISC-V'te Buyruk Biçimleri (R, I, S, B, U, J türleri)	131
3.12	RISC-V'te J Türü Buyruk yapısı	132
3.13	Programda Dallanma	139
3.14	RISC-V'te Yazmaç Adresleme Kipi	145
3.15	RISC-V'te Anlık Adresleme Kipi	146
3.16	RISC-V'te Eklemeli Adresleme Kipi	146
3.17	RISC-V'te PS'ye Göreceli Adresleme Kipi	147
3.18	RISC-V'te üst anlık adresleme kipi	147
3.19	İç içe altyordam çağrılarında yığıt durumu	151
3.20	Altyordam çağrısı sırasında yığıt çerçevesi	152
3.21	İç içe altyordam çağrısında yığıt çerçeveleri	152
3.22	Programın işlenmesi	155
3.23	Tipik bir RISC-V programının bellek yerleşimi	156
3.24	x86'da Değişik Buyruk Biçimleri	160

Bölüm 4 – RISC-V ile Programlama

4.1	Yığıt işaretçisinin yazmaç saklama sırasındaki değişimi	177
4.2	Altyordam çağrısında kontrol akışı	182
4.3	RISC-V programının bellek düzeni	184
4.4	Bağlı liste veri yapısının bellek görünümü	192
4.5	Özyinelemeli Fibonacci çağrı ağacı	194
4.6	Özyinelemeli Fibonacci yığıt büyümesi	194

Bölüm 5 – Aritmetik İşlemler

5.1	Yarım toplayıcı devresi	224
5.2	Tam toplayıcı devre şeması	224
5.3	1 bitlik toplayıcı (blok gösterimi)	225
5.4	4 bitlik dalga eldeli toplayıcı	225
5.5	Toplama ve çıkarma yapabilen 1 bitlik AMB	226
5.6	4 bitlik toplayıcı-çıkarcı	226
5.7	VE ve VEYA işlemleri yapabilen AMB (1 bitlik)	227
5.8	1 bitlik AMB	227
5.9	“Küçük” girişi eklenmiş 1 bitlik AMB	229
5.10	32 Bitlik AMB	230
5.11	Önceden elde üreten devre	232
5.12	4 Bitlik önceden elde üretici	233
5.13	8 bitlik elde ile seçmeli toplayıcı	234
5.14	Kaydır-Topla Çarpma (1. Yol) akış şeması	237
5.15	Kaydır-Topla Çarpma (1. Yol) donanımı	238
5.16	Kaydır-Topla Çarpma (2. Yol) donanımı	239

5.17	Kaydır-Topla Çarpma (2. Yol) akış şeması	240
5.18	Kaydır-Topla Çarpma (3. Yol) donanımı	241
5.19	Kaydır-Topla Çarpma (3. Yol) akış şeması	241
5.20	Booth Çarpım Algoritması akış şeması	244
5.21	Dizi çarpıcı: 4 bit	247
5.22	Wallace ağacı çarpıcı: 4 kısmi çarpım	249
5.23	Geri Yükleme Bölme (1. Yol) akış şeması	253
5.24	Geri Yükleme Bölme (1. Yol) donanımı	254
5.25	Geri Yükleme Bölme (2. Yol) akış şeması	255
5.26	Geri Yükleme Bölme (2. Yol) donanımı	255
5.27	Geri Yükleme Bölme (3. Yol) donanımı	257
5.28	Geri Yükleme Bölme (3. Yol) akış şeması	257
5.29	SRT bölme donanımı	261
5.30	Kayan nokta toplama/çıkarma donanımının blok şeması	278
5.31	Kayan nokta çarpma işleminin akış şeması	280
5.32	FP32, BF16 ve FP16 bit düzenleri	285
5.33	FP8 bit düzenleri: E4M3 ve E5M2	286
5.34	Yapay zeka modellerinin kayan nokta biçimine göre bellek gereksinimleri	287
5.35	Farklı yapay zeka modellerinin ağırlık dağılımları	288
5.36	Karma duyarlık eğitim döngüsü	289

Ek A — Sayı Sistemleri ve Kodlama

A.1	ASCII Karakter Tablosu (yazdırılabilir 96 karakter)	616
A.2	IEEE 754 tek duyarlı kayan nokta biçimi	618
A.3	IEEE 754 kayan nokta biçimlerinin karşılaştırması	619

Ek B — Sayısal Tasarım Temelleri

B.1	Temel ve türetilmiş mantık kapılarının devre sembolleri	629
B.2	İki değişkenli Karno haritası (a, b) ve mini terim numaraları	636
B.3	Üç değişkenli Karno haritası ve mini terim numaraları (sütunlar Gray kodu sırasıyla dizilir: 00, 01, 11, 10)	637
B.4	Dört değişkenli Karno haritası ve mini terim numaraları	638
B.5	Beş değişkenli Karno haritası (sol: 3D görünüm, sağ: 2D açılım ve gruplama örneği, $F = \Sigma(2, 6, 7, 14, 15, 18, 22, 23, 30, 31)$)	639
B.6	Altı değişkenli Karno haritası (sol: 3D görünüm, sağ: 2D açılım, 2×2 düzen). Komşu haritalar arasındaki aynı konumdaki hücreler de komşudur.	639
B.7	Yarım toplayıcı devresi: bir Dışlayan VEYA ve bir VE kapısından oluşur	644
B.8	Tam toplayıcı devresi: iki yarım toplayıcı ve bir VEYA kapısından oluşur	644
B.9	4 bitlik dalgali elde toplayıcı: elde sinyali en az anlamlı bitten (sağ) en anlamlı bite (sol) doğru yayılır	645
B.10	3×8 kod çözücü: (a) blok diyagram, (b) doğruluk tablosu, (c) kapı düzeyinde gerçekleştirme	646
B.11	3×8 kod çözücünün iki 2×4 kod çözücü ve bir DEĞİL kapısıyla oluşturulması; x etkinleştirme sinyali hangi alt kod çözücünün etkin olduğunu seçer	647
B.12	4×2 kodlayıcının blok diyagramı: dört girişten 1 olan birinin indeksi iki bitlik $Y_1 Y_0$ kodunda üretilir	649

B.13	4×1 çoklayıcı: kapı düzeyi, doğruluk tablosu, blok diyagram	651
B.14	8×1 çoklayıcının iki 4×1 ve bir 2×1 çoklayıcı ile oluşturulması; s_1s_0 alt çoklayıcıları, s_2 son katmandaki 2×1 çoklayıcıyı sürer	652
B.15	1×4 tekleyicinin blok diyagramı: G verisi s_1s_0 seçim sinyaline göre Y_0-Y_3 çıkışlarından birine yönlendirilir	655
B.16	RS Flip-flop	657
B.17	D Flip-flop	658
B.18	D flip-flop zamanlama diyagramı: saat sinyalinin her yükselen kenarında D değeri Q 'ya aktarılır	658
B.19	T Flip-flop	658
B.20	JK Flip-flop	659
B.21	4 bitlik koşut yüklemeli yazmaç	660
B.22	4 bitlik sağa kaydırmalı yazmaç: seri giriş en soldaki D FF'nin girişine bağlanır; her saat vuruşunda her bit bir konum sağa kayar ve seri çıkıştan çıkar. Q_3-Q_0 koşut çıkışlardır.	660
B.23	3 bitlik eşzamansız (dalgalı) sayaç: her flip-flopun saat girişi bir önceki flip-flopun çıkışına bağlıdır	662
B.24	3-bit eşzamansız sayacın dalga formu	662
B.25	Mealy ve Moore makinelerinin zamanlama farkı: Mealy çıkışı Y_M giriş X değiştiği anda hemen güncellenir; Moore çıkışı Y_O ise bir sonraki saat yükselen kenarında değişir. Bu nedenle Mealy daha hızlı tepki verirken Moore titreşime karşı daha karardır.	664
B.26	Mealy ve Moore makinelerinin blok diyagramları. Saat ortak senkronizasyon sinylidir; her saat vuruşunda durum yazmaçları yenilenir. Mealy makinesinde giriş sinyali (kesikli kırmızı ok) çıkış mantığını doğrudan etkiler; Moore'da ise çıkışlar yalnızca duruma bağlıdır.	664
B.27	Ardışık iki '1' algılama: Mealy (2 durum, solda) ve Moore (3 durum, sağda) karşılaştırması	665

Ek C – RISC-V Referans Kartı

C.1	RISC-V alt yordam yığıt çerçevesinin yapısı	685
C.2	R türü buyruk biçimi	685
C.3	I türü buyruk biçimi	685
C.4	S türü buyruk biçimi	685
C.5	B türü buyruk biçimi	685
C.6	U türü buyruk biçimi	685
C.7	J türü buyruk biçimi	686

Ek D – RISC-V Uygulama Çalışmaları ve Projeler

D.1	Derleme-bağlama-çalıştırma akışı	696
D.2	Verilog modül hiyerarşisi	697
D.3	Yazmaç öbeği blok diyagramı	700

Tablo Listesi

Bölüm 1 – Bilgisayar Mimarisine Giriş

1.1	Intel işlemcilerin gelişimi	6
1.2	Teknoloji düğümlerinin gelişimi	14
1.3	Teknoloji düğümlerine göre üretim maliyetleri	15
1.4	RISC ve CISC tasarım felsefelerinin karşılaştırması	20
1.5	Soğutma çözümlerinin TDP kapasitesi ve maliyeti	25
1.6	Durağan güç oranının teknoloji düğümlerine göre değişimi	28

Bölüm 2 – Bilgisayar Başarımı

2.1	Uygulama alanlarına göre başarımların gereksinimleri	48
2.2	Aynı C kodunun üç farklı BKM’de derlenmesi	54
2.3	Başarımların bileşenlerini etkileyen tasarım katmanları	60
2.4	SPEC CPU sinama programı kümelerinin evrimi	70
2.5	SPECint 2017 Tamsayı Sinama Programları	70
2.6	SPECfp 2017 Kayan Nokta Sinama Programları	71
2.7	Hesaplama başarımında kilometre taşları	75
2.8	Güncel işlemci karşılaştırması (2023–2024)	87

Bölüm 3 – Buyruk Kümesi Mimarisi

3.1	Farklı BKM’lerde yaklaşık buyruk sayıları	106
3.2	RISC-V uzantıları	107
3.3	RISC-V’te kullanılan yazmaçlar	111
3.4	Temel CSR yazmaçları	113
3.5	Dört mimaride yazmaç karşılaştırması	118
3.6	Yazmaç sayısının artı ve eksileri	119
3.7	İşlemci ailelerinin bayt sırası tercihleri	128
3.8	RISC-V’te R türü buyruğun anlamlı parçalara ayrılması	129
3.9	RISC-V’te bir R türü buyruğun alanları	130
3.10	RISC-V buyruk biçimlerinin karşılaştırması	132
3.11	RISC-V’te veri aktarma buyrukları	133
3.12	RISC-V’te aritmetik işlem buyrukları	134
3.13	Çarpma sonucunun saklanması: mimari yaklaşımlar	135
3.14	Aynı çarpma işleminin dört mimarideki gerçekleştirimi	136
3.15	RISC-V’te mantık buyrukları	137
3.16	RISC-V’te bit düzeyinde kaydırma buyrukları	138
3.17	RISC-V’te koşullu dallanma buyrukları	139

3.18	RISC-V'te karşılaştırma buyrukları	139
3.19	RISC-V'te koşulsuz atlama buyrukları	141
3.20	RISC-V'te yaygın sözde buyruklar ve gerçek buyruk karşılıkları	144
3.21	Adresleme kipleri ve üst düzey dil karşılıkları	144
3.22	İşlemcilerde kullanılan adresleme kipleri	149
3.23	RISC-V ile MIPS'in karşılaştırması	157
3.24	Aynı işlemin RISC-V ve MIPS'teki karşılıkları	157
3.25	RISC-V ile ARM (AArch64) karşılaştırması	158
3.26	Aynı işlemin RISC-V ve ARM (AArch64)'teki karşılıkları	158
3.27	x86'daki bazı yazmaçlar	159
3.28	RISC-V, MIPS, ARM ve x86 mimarilerinin karşılaştırması	160
3.29	Bilgisayar tarihinde öne çıkan diğer buyruk kümesi mimarileri	161

Bölüm 4 – RISC-V ile Programlama

4.1	Veri tanımlama yönergeleri	184
4.2	Yaygın derleyici eniyilemeleri	199
4.3	Çevirici dilde sık yapılan hatalar	205

Bölüm 5 – Aritmetik İşlemler

5.1	Aritmetik İşlem Buyrukları	220
5.2	Bit Düzeyinde Kaydırma Buyrukları	222
5.3	Mantık Buyrukları	222
5.4	Yarım toplayıcı doğruluk tablosu	223
5.5	Tam toplayıcı doğruluk tablosu	224
5.6	32 Bitlik AMB İşlem Tablosu	229
5.7	Toplayıcı türlerinin karşılaştırması	235
5.8	Yazılım ve donanım çarpma karşılaştırması	236
5.9	Örnek 5.4 – 1. yol çarpma algoritması	239
5.10	Örnek 5.5 – 2. yol çarpma algoritması	240
5.11	Örnek 5.6 – 3. yol çarpma algoritması	242
5.12	Kaydır-topla çarpma yollarının karşılaştırması (32 bitlik RISC-V)	242
5.13	Booth Algoritması Kuralları	244
5.14	Radix-4 Booth kodlama tablosu (M = çarpılan)	245
5.15	Dizi çarpıcı ve Wallace ağacı karşılaştırması (n bitlik çarpma)	250
5.16	Yazılım ve donanım bölme karşılaştırması	252
5.17	Örnek 5.10 – 1. yol geri yüklemeli bölme	254
5.18	Örnek 5.11 – 2. yol geri yüklemeli bölme	256
5.19	Örnek 5.12 – 3. yol geri yüklemeli bölme	258
5.20	Bölme yöntemlerinin karşılaştırması	262
5.21	RISC-V Bölme Buyrukları (M Uzantısı)	262
5.22	Sıfıra bölme ve taşma sonuçları	264
5.23	IEEE 754 özel değerlerinin kodlanması (tek duyarlı)	272
5.24	Özel değerlerle yapılan işlemler	274
5.25	RISC-V Kayan Nokta Buyrukları (F/D Uzantıları)	283
5.26	Kayan nokta biçimlerinin karşılaştırması	289
5.27	Güncel yongalardaki aritmetik birim sayıları	292

Ek A – Sayı Sistemleri ve Kodlama

A.1	16’lık tabandaki sayıların diğer tabanlardaki gösterimleri	608
A.2	Değişik yöntemlerde 4 bitle gösterilen sayılar	612
A.3	3 Bitlik Gray Kodu	614
A.4	10’luk tabandaki sayıların BCD yöntemi ile gösterimleri	615
A.5	UTF-8 kodlama kuralları	617
A.6	IEEE 754 duyarlık düzeylerinin karşılaştırması	619
A.7	IEEE 754 özel değerler (tek duyarlı)	620

Ek B – Sayısal Tasarım Temelleri

B.1	Boole cebrinin temel özellikleri	630
B.2	$F_1 = xyz$, $F_2 = x + \bar{y} + \bar{z}$, $F_3 = \bar{x}(y + z)$ fonksiyonlarının doğruluk tablosu . . .	631
B.3	Standart Mantık İşlemleri	633
B.4	Mini ve maksî terimlerin gösterimi	634
B.5	4×2 kodlayıcının doğruluk tablosu (aynı anda yalnızca bir girişin 1 olduğu ka- bulü altında)	649
B.6	RS Flip-flop doğruluk tablosu	657
B.7	D Flip-flop doğruluk tablosu – tam (sol) ve özet (sağ)	658
B.8	T Flip-flop doğruluk tablosu – tam (sol) ve özet (sağ)	659
B.9	JK Flip-flop doğruluk tablosu	659
B.10	Mealy ve Moore makinelerinin karşılaştırması	663

Ek C – RISC-V Referans Kartı

C.1	RISC-V standart uzantıları	679
C.2	RISC-V ayrıcalık düzeyleri	680
C.3	RISC-V buyruk kategorileri ve yazım örnekleri	681
C.4	RV32I temel buyruk kümesi	681
C.5	RISC-V RV32I yazmaç tablosu	684
C.6	Buyruk biçimleri ve kullanım alanları	686
C.7	RISC-V sözde buyrukları	686

Ek D – RISC-V Uygulama Çalışmaları ve Projeler

D.1	RISC-V benzetimliklerinin karşılaştırması	694
D.2	RISC-V GNU araç zinciri bileşenleri	696
D.3	Mini proje özet tablosu	700
D.4	Önbellek parametre taraması örneği	702
D.5	Proje değerlendirme ölçütleri	703

Bilgi Notları Listesi

Bölüm 1 – Bilgisayar Mimarisine Giriş

1.1	Robert Dennard ve Dennard Ölçeklemesi	4
1.2	Moore Yasası’nın Geleceği	7
1.3	Silisyum mu, Silikon mu?	12
1.4	Silisyum plaka neden yuvarlak?	13
1.5	RISC-V: Bu Kitabın Buyruk Kümesi	20
1.6	Getir–Çöz–Yürüt ve Bulaşık Makinesi Benzetmesi	23
1.7	Tasarım İlkeleri Yalnızca Donanım İçin mi?	24
1.8	Kaynak Geriliminin Tarihsel Evrimi	26
1.9	GHz Yarışından Verimlilik Yarışına	28
1.10	Mars’ta Güvenilirlik	33
1.11	Meta ve RISC-V: Veri Merkezine Giden Yol	35
1.12	Büyük Dil Modelleri ve Donanım	37

Bölüm 2 – Bilgisayar Başarımı

2.1	Yanlış Ölçütlerle Ölçmek: Goodhart Yasası	49
2.2	“2 kat daha hızlı” mı, “2 katı kadar hızlı” mı?	50
2.3	Üç Farklı Buyruk Sayısı	54
2.4	Pazarlama Savaşlarında Başarım Ölçütleri	62
2.5	Tepe Başarım ve Gerçek Başarım	64
2.6	Skynet: 60 Teraflop Yeter mi?	64
2.7	MLPerf: Yapay Zekanın SPEC’i	65
2.8	SPEC 2006’dan SPEC 2017’ye Geçiş	71
2.9	Amdahl Yasası ve Koşut Hesaplama	78
2.10	Amdahl ve Gustafson: Hangisi Doğru?	80
2.11	Doğru İş Yükünü Doğru Birime Yönlendirmek	83
2.12	Green500: Süper Bilgisayarların Enerji Verimliliği Sıralaması	87

Bölüm 3 – Buyruk Kümesi Mimarisi

3.1	Sayı Sistemleri ve Kodlama	103
3.2	Bu Bölümde Ele Alınmayan Uzantılar	107
3.3	PC Neden Genel Amaçlı Yazmaç Değildir?	112
3.4	Başarım Sayaçları ve BBÇ Ölçümü	114
3.5	Yığıt Göstergesi: Donanım mı, Uzlaşım mı?	115
3.6	RV64I’de 64 Bit Yazmaçlar	123
3.7	RISC-V Sıkıştırılmış Buyruklar	126

3.8	Güliver'in Yumurta Savaşı	127
3.9	Ortası Başta	128
3.10	Anlık Değerlerde İşaret Genişletme Düzeltmesi	133
3.11	RV64I'de 32 Bitlik Çarpma ve Bölme	135
3.12	RISC-V'te Geciktirilmiş Dallanma Yoktur	141
3.13	Yaprak Altyordam Eniyilemesi	154
3.14	ecall ve Ayrıcalık Düzeyleri	156
3.15	Neden Elde Taşınır Cihazlarda x86 Yok?	161

Bölüm 4 – RISC-V ile Programlama

4.1	auipc ve Program Sayacı	176
4.2	RARS'ta Adım Adım Yürütme	182
4.3	Programlama ve Başarım İlişkisi	182
4.4	Sistem Çağrıları	184
4.5	Yığın ve Yığıt: İki Farklı Kavram	192
4.6	Kuyruk Özyineleme Eniyilemesi	194
4.7	Derleyiciye Güvenmek	198
4.8	Uçucu Bellek Erişimi ve Derleyici	201
4.9	Alanda Yerellik ve Zamanda Yerellik	204

Bölüm 5 – Aritmetik İşlemler

5.1	RISC-V'te DEĞİL İşlemi	222
5.2	Bit Numaralandırması: 0'dan mı 1'den mi?	225
5.3	Gerçek İşlemcilerde VEYA-DEĞİL ve VE-DEĞİL Kapıları	228
5.4	Taşma Denetiminde Elde Kuralı	230
5.5	Güncel İşlemcilerde ve Yapay Zeka Hızlandırıcılarında Hızlı Toplayıcılar	233
5.6	Booth Algoritmasının Tarihçesi	243
5.7	Dadda Ağacı	249
5.8	Çağdaş İşlemcilerde Çarpma Birimi	250
5.9	İşaretili Bölme İşlemi	258
5.10	Bölmeyi Çarpmaya Dönüştürmek	261
5.11	Gizli 1 Her Yerde Var mı?	267
5.12	IEEE 754 Üsleri Neden Eklentili Saklar?	268
5.13	RISC-V'te Dörtlü Duyarlık: Q Uzantısı	270
5.14	NaN ile Programlama Uygulaması	274
5.15	Kayan Nokta Duyarlığı ve Günlük Etkisi	277
5.16	Intel Pentium FDIV Hatası	284
5.17	Patriot Füzesi Kazası: Yuvarlama Hatasının Bedeli	284

Ek A – Sayı Sistemleri ve Kodlama

A.1	Ariane 5: Taşmanın Bedeli	613
A.2	Unicode'un Doğuşu	616
A.3	IEEE 754 ve William Kahan	618

Ek B – Sayısal Tasarım Temelleri

B.1	Sayısal Tasarım ve Teknoloji	627
B.2	George Boole ve Mantığın Cebirleşmesi	629

B.3	Boole Cebrinde Gösterim ve Öncelik	630
B.4	Maurice Karnaugh ve Karno Haritası	635
B.5	Değişken Sırası ve Mini Terim Numarası	635

Ek C – RISC-V Referans Kartı

C.1	Bu kitapta kullanılan yapılandırma	679
C.2	Yükleme ve saklama buyruklarındaki fark	681

Ek D – RISC-V Uygulama Çalışmaları ve Projeler

D.1	Hızlı Başlangıç	694
D.2	Tek Adımda Derleme	696
D.3	Bloklı ve Bloksuz Atama	698
D.4	FPGA ile Gerçekleme	699
D.5	Buyruk Çözme İpucu	701
D.6	İz Dosyası Kaynakları	702

Bilgisayar Mimarisine Giriş



Ayasofya – İstanbul

Yaklaşık 1500 yıldır ayakta duran Ayasofya, çağının en ileri mühendislik bilgisiyle inşa edilmiştir. Bilgisayar mimarisi de benzer bir mühendislik disiplini: sınırlı kaynaklarla en yüksek başarıyı elde etmeye çalışır.

Öğrenme Hedefleri

Bu bölümü tamamladığınızda aşağıdakileri yapabileceksiniz:

- Bilgisayar tarihinin beş neslini ve her neslin ayırt edici teknolojisini açıklamak.
- Moore Yasası ve Dennard ölçeklemesinin ne olduğunu ve neden sona erdiğini tartışmak.
- Bir bilgisayarın beş ana bileşenini sıralamak ve her birinin işlevini tanımlamak.
- Yonga üretim sürecini (fotolitografi, dağlama, kaplama) ana hatlarıyla betimlemek.
- Üst düzey dilden makine koduna uzanan derleme zincirini açıklamak.
- RISC ile CISC tasarım felsefelerini karşılaştırmak.
- RISC-V buyruk kümesinin neden tercih edildiğini gerekçelendirmek.
- Güç duvarının mimari tasarıma etkisini ve heterojen hesaplama kavramını tartışmak.

1.1 Giriş

Cebinizdeki akıllı telefon, 1969'da insanı Ay'a götüren Apollo rehberlik bilgisayarından milyonlarca kat daha güçlü. Bugün bilgisayarlar yalnızca masaüstünde değil, saatimizde, buzdolabımızda, otomobilimizde ve veri merkezlerindeki devasa sunucu çiftliklerinde karşımıza çıkıyor. Bu yaygınlığın arkasında yarım yüzyılı aşan sürekli bir mimari yenilenme var.

Bilgisayar mimarisi, donanım ile yazılım arasındaki sınırı tanımlayan ve işlemci, bellek ve giriş/çıkış birimlerinin nasıl bir araya geleceğini belirleyen mühendislik dalıdır. Verimli yazılım geliştirmek isteyen bir programcı için donanımın çalışma ilkelerini anlamak, yüksek başarımlı donanım tasarlamak isteyen bir mühendis için de yazılım gereksinimlerini kavramak vazgeçilmez. Üstelik alan, tarihsel bir dönüm noktasında: transistörler küçüldükçe güç yoğunluğunun sabit kalacağını öngören Dennard ölçeklemesinin sona ermesi, yapay zekânın yükselişi ve açık kaynak donanım hareketleri bilgisayar mimarisini her zamankinden daha heyecanlı bir araştırma alanı yapıyor.

Bu bölümde önce bilgisayarların tarihsel gelişimini, ardından bir bilgisayarın temel bileşenlerini ve işlemcinin nasıl üretildiğini göreceğiz. Sonra programların bu donanım üzerinde nasıl çalıştırıldığını, mimariyi şekillendiren etkenleri, güç tüketimi sorununu, RISC-V buyruk kümesinin neden tercih edildiğini ve yapay zekânın bilgisayar mimarisine getirdiği yeni ufukları ele alacağız.

1.2 Bilgisayarların Gelişimi

Abaküsle başlayan hesap yapma gereksinimi, önce mekanik, sonra da elektronik hesap makineleriyle karşılanmaya çalışıldı. Bilinen anlamda ilk mekanik hesap makinesini Blaise Pascal 1642'de tasarladı. Pascal, toplama ve çıkarma işlemlerini yapabilen bu makineyi vergi tahsilatı babasına yardım etmek amacıyla geliştirdi. Şekil 1.1'de Pascal'ın yaptığı bu ilk makinenin bir resmi görülüyor.



Şekil 1.1: Blaise Pascal'ın 1642'de yaptığı mekanik hesap makinesi (Pascaline)

19. yüzyılın sonlarından itibaren hesaplama makineleri hızla gelişti. 1890'da Herman Holerith delikli kart sistemiyle çalışan bir hesaplama düzeneği geliştirdi; bu sistem ABD nüfus sayımında başarıyla kullanıldı.

Bu gelişmelere koşut olarak Alan Turing 1936'da yayımladığı makalesinde [41], sonradan *Turing makinesi* olarak anılacak soyut hesaplama modelini tanımlayarak hesaplanabilirliğin teorik temelini attı. Bu model, bir algoritma ile çözülebilecek her problemin basit bir bant, okuma/yazma kafası ve sonlu durum tablosundan oluşan bir makineyle çözülebileceğini gösteriyordu. Turing makinesi kavramı, bilgisayar biliminin kurucu taşlarından biri olarak kabul edilir ve günümüzde bile algoritma karmaşıklığı ve hesaplanabilirlik kuramının temelini oluşturur.

2. Dünya Savaşı döneminde hesaplama ihtiyacı iyice arttı ve birçok öncü makine üretildi. Turing, İngiltere'de Bletchley Park'ta Alman Enigma şifre makinesinin kırılması için elektromekanik Bombe makinesini tasarladı. İngilizler daha sonra daha gelişmiş şifrelerin çözümü için Colossus'u geliştirdi. Aynı dönemde IBM ile çalışan Howard H. Aiken 1944'te rölelerden oluşan Mark I'i tasarladı. Bilgisayar tarihinin sonraki dönüm noktaları aşağıda nesiller halinde ele alınıyor.

Birinci Nesil: Vakum Tüpleri (1940'lar–1950'ler)

Günümüz bilgisayarlarının atası sayılan ENIAC (Electronic Numerical Integrator And Computer), 1945'te Pennsylvania Üniversitesi'nde tamamlandı. Yaklaşık 18.000 vakum tüpü kullanan, 30 tonluk ve yaklaşık 150 kW güç tüketen bu dev makine yalnızca dört işlem ve basit karşılaştırma yapabiliyordu. Kısa süre sonra Von Neumann'ın teorik katkılarıyla tasarlanan EDVAC, programı ve veriyi aynı bellekte saklayarak çıkış açtı. Birinci nesil bilgisayarlar makine dili ile programlanıyordu; vakum tüplerinin kısa ömrü, büyük boyutu ve yüksek ısı üretimi en önemli sınırlılıkları oluştuyordu.

İkinci Nesil: Transistörler (1950'ler–1960'lar)

1947'de Bell Laboratuvarları'nda William Shockley, John Bardeen ve Walter Brattain tarafından icat edilen transistör, vakum tüplerinin yerini aldı. Transistörler çok daha küçük, güvenilir ve ucuzdu; güç tüketimleri de düşüktü. Bu dönemde çevirici dil (Assembly) kullanılmaya başlandı ve programlama önemli ölçüde kolaylaştı. Fortran ve COBOL gibi ilk üst düzey diller de bu nesilde ortaya çıktı.

Üçüncü Nesil: Tümleşik Devreler (1960'lar–1970'ler)

Texas Instruments'ta Jack Kilby [21] ve Fairchild Semiconductor'da Robert Noyce birbirinden bağımsız olarak tümleşik devreyi geliştirdi. Tüm bileşenlerin ve bağlantıların aynı yarıiletken malzeme üzerinde üretilmesi, devrelerin boyutunu ve maliyetini büyük ölçüde düşürdü. Bu dönemde işletim sistemleri yaygınlaştı ve birden çok programın aynı belleği paylaşması mümkün hale geldi.

Dördüncü Nesil: Mikroişlemciler (1970'ler–2000'ler)

1971'de Intel'in 4004 yongası, işlemcinin tüm bileşenlerini tek bir tümleşik devre üzerinde toplayan ilk *mikroişlemci* oldu. Moore yasasının öngördüğü transistör artışı sayesinde işlemciler hızla güçlendi; 1981'de kişisel bilgisayar (PC) kavramı doğdu ve bilgisayarlar geniş kitlelere ulaştı. Bu nesil boyunca saat sıklığı sürekli artırılarak başarımların kazanımı sağlandı.

Beşinci Nesil: Çok Çekirdekli ve Heterojen Mimariler (2000'ler–günümüz)

2000'li yılların ortalarından itibaren güç duvarı ve Dennard ölçeklemesinin sona ermesi saat sıklığını artırma stratejisini sürdürülemez kıldı. Tasarımcılar, tek bir yonga üzerine birden çok çekirdek yerleştirerek başarımların artışı için hesaplamaya kaydıldı. Günümüzde MİB çekirdeklerinin yanına GPU, NPU ve diğer özelleşmiş hızlandırıcılar eklenerek *heterojen* mimariler oluşturuluyor. Açık kaynaklı RISC-V buyruk kümesi mimarisi de bu dönemin önemli gelişmeleri arasında yer alıyor.

Bilgi Notu 1.1: Robert Dennard ve Dennard Ölçeklemesi

IBM araştırmacısı Robert Dennard, 1974'te yayımladığı makalede [9] transistör boyutlarının küçültülmesinin etkilerini inceledi. Dennard'ın ortaya koyduğu ölçekleme kuralına göre transistörler küçüldükçe çalışma gerilimleri ve akımları da orantılı biçimde azalır; dolayısıyla birim alan başına güç tüketimi yaklaşık sabit kalır. Bu ilke sayesinde 1970'lerden 2000'lerin başına kadar her yeni üretim teknolojisiyle hem daha fazla transistör sığdırılabilirdi hem de saat sıklığı artırılabilirdi; çünkü ek transistörler fazladan güç tüketmiyordu. Ancak 2006 civarında transistör boyutları 65 nm'nin altına indiğinde sızıntı akımları baskın hale geldi ve Dennard ölçeklemesi geçerliliğini yitirdi. Bu dönüm noktası, işlemci tasarımında saat sıklığı yarışının sona ermesine ve çok çekirdekli mimarilere geçişe yol açtı.

Terim Kökenleri

donanım (*hardware*): *Donanmak* (to equip) fiilinden *-ım* ekiyle. Bilgisayarın fiziksel “donanımı”, tıpkı bir geminin donanması gibi.

yazılım (*software*): *Yazılmak* (to be written) fiilinin edilgen biçiminden *-ım* ekiyle. Donanımın üzerinde çalışan “yazılmış” programlar.

yonga (*chip*): *Yontmak* (to carve, to whittle) fiilinin kökü *yon-* ve *-ga* ekiyle; ağaçtan yontularak çıkarılan ince parça demektir. İngilizce *chip* de “koparılan küçük parça” anlamındadır. Paketlenmiş tümleşik devreyi ifade eder.

kırmık (*die*): *Kırmak* (to break) fiilinden *-ık* ekiyle; silisyum dilimden (wafer) kesilerek “kırılıp” ayrılan, henüz paketlenmemiş yarıiletken parçasıdır.

bellek (*memory*): *Bellemek* (to memorize) fiilinden *-ek* araç yapım ekiyle; “belleme aracı” demektir. Yazar Nurullah Ataç’ın önerisiyle Türkçeye kazandırılmış, Arapça kökenli *hafıza* sözcüğünün yerini almıştır.

tümleşik devre (*integrated circuit*): *Tümleşmek* (to integrate, to merge into a whole) fiilinden; tüm bileşenlerin tek bir “bütün” haline getirildiği devre.

merkezi işlem birimi (MİB) (*CPU, Central Processing Unit*): Bilgisayarın hesaplama ve denetim işlevlerini gerçekleştiren ana birim. *Merkez* (center) ve *işlemek* (to process) köklerinden; İngilizce *CPU* kısaltması da yaygın biçimde kullanılır.

Nesiller Boyunca: Boyut, Güç ve Başarım

Her nesil, önceki neslin kısıtlarını aşarak yeni olanaklar açmıştır:

Nesil	Anahtarlama hızı	Tipik güç	Tipik boyut
Vakum tüpü	$\sim \mu s$	$\sim 150 \text{ kW}$	Oda
Transistör	$\sim 100 \text{ ns}$	$\sim 10 \text{ kW}$	Dolap
Tümleşik devre	$\sim 10 \text{ ns}$	$\sim 1 \text{ kW}$	Masa
Mikroişlemci	$\sim 1 \text{ ns}$	$\sim 100 \text{ W}$	Yonga
Çok çekirdekli	$< 1 \text{ ns}$	$5\text{--}300 \text{ W}$	Yonga

Bu tabloda çarpıcı bir örüntü görülür: anahtarlama hızı her nesilde yaklaşık 10 kat artarken, sistem boyutu odadan tek bir yongaya küçülmüştür. Güç tüketimi ise beşinci nesilde artık düşürülemez hale gelmiştir; bu durum, “güç duvarı” kavramının ortaya çıkmasına yol açmıştır.

1.2.1 Moore Yasası ve Transistör Artışı

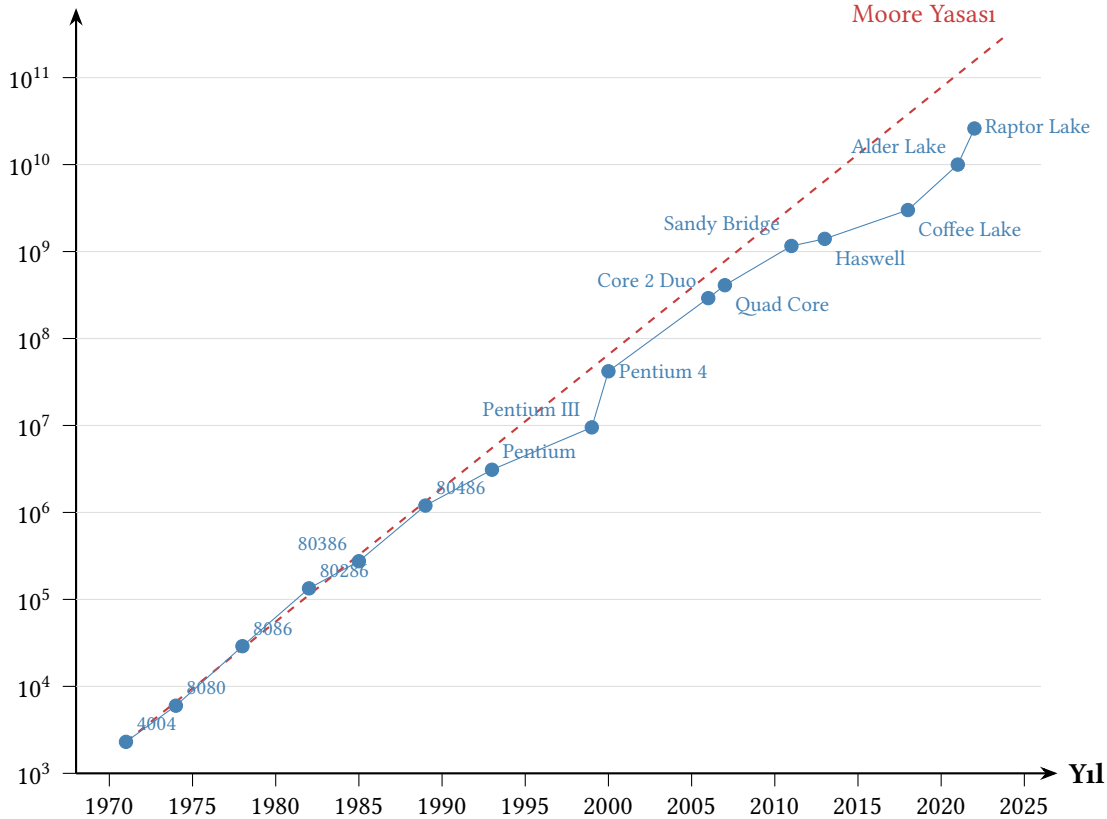
Bu beş nesil boyunca yaşanan hızlı gelişimin arkasında, transistör teknolojisindeki üstel ilerleme yatıyor. Intel marka işlemcilerin tarihte gelişimi Tablo 1.1’de veriliyor.

Intel’in kurucularından Gordon Moore 1965 yılında bir gözlem ortaya koydu [29]: Bir yonga üzerindeki transistör sayısı yaklaşık her iki yılda bir ikiye katlanıyor. Moore bu öngörüü ilk olarak her yıl ikiye katlanma biçiminde ifade etmiş, 1975’te ise süreyi yaklaşık iki yıla revize etmiştir. “Moore Yasası” olarak bilinen bu kural, Newton yasaları gibi değişmez bir fizik yasası değil, gözleme dayalı ampirik bir eğilimdir. Buna karşın yarıiletken endüstrisi on yıllar boyunca bu eğilimi bir geliştirme hedefi olarak benimsemiş ve tasarım yol haritalarını buna göre şekillendirmiştir. Tablodaki veriler incelendiğinde, 1971’den 2022’ye kadar transistör sayısının yaklaşık 10 milyon kat arttığı görülür. Bu büyüme hızı, insan uygarlığının başka hiçbir teknoloji alanında gözlemlenmemiştir. Şekil 1.2’de bu büyümenin logaritmik ölçekteki grafiği verilmiştir.

Tablo 1.1: Intel işlemcilerin gelişimi

Yonga	Tarih	Saat Sıklığı	Transistör Sayısı	Açıklama
4004	1971	0,74 MHz	2.300	Bir yongadaki ilk mikroişlemci
8008	1972	0,5–0,8 MHz	3.500	İlk 8 bitlik mikroişlemci
8080	1974	2 MHz	6.000	İlk genel amaçlı işlemci
8086	1978	5–10 MHz	29.000	İlk 16 bitlik mikroişlemci
8088	1979	5–8 MHz	29.000	IBM PC’de kullanılan işlemci
80286	1982	8–12 MHz	134.000	Bellek koruması bulunur
80386	1985	16–33 MHz	275.000	İlk 32 bitlik mikroişlemci
80486	1989	25–100 MHz	1,2M	8 KB önbelleğe sahip
Pentium	1993	60–200 MHz	3,1M	İlk süperskaler x86 (iki boru hattı)
Pentium MMX	1997	166–233 MHz	4,5M	SIMD buyruk desteği (MMX)
Pentium Pro	1995	150–200 MHz	5,5M	İki düzey önbellek
Pentium II	1997	233–400 MHz	7,5M	Pentium Pro mimarisi + MMX
Pentium III	1999	500 MHz	9,5M	Güç tasarrufu seviyeleri
Pentium 4	2000	1,5 GHz	42M	NetBurst mikromimarisi; derin boru hattı
Pentium D	2005	3,2 GHz	230M	İlk 2 çekirdekli masaüstü
Core 2 Duo	2006	2,93 GHz	291M	Pentium M’in gelişmiş hali
Quad Core	2007	>3 GHz	410M	İki tane Core 2 Duo
Xeon	2009	2,26 GHz	781M	Ölçeklenebilirlik odaklı
Sandy Bridge	2011	3,4 GHz	1,16Mr	AVX, tümleşik GPU
Haswell	2013	3,5 GHz	1,4Mr	AVX2, FMA3
Skylake	2015	4,0 GHz	1,75Mr	DDR4 desteği
Coffee Lake	2018	3,6 GHz	~3Mr	İlk 6 çekirdekli masaüstü
Alder Lake	2021	5,2 GHz	~10Mr	İlk karma mimari (P+E çekirdek)
Raptor Lake	2022	5,8 GHz	~26Mr	8P+16E çekirdek

M = milyon; Mr = milyar; “~” tahmini değer.

Transistör Sayısı**Şekil 1.2:** Yıllara göre Intel işlemcilerdeki transistör sayısının artışı (grafik logaritmik ölçektedir). Kesik çizgi Moore Yasası’nın öngördüğü eğilimi göstermektedir.

Bilgi Notu 1.2: Moore Yasası'nın Geleceği

Moore Yasası, yarım yüzyılı aşkın bir süre endüstrinin yol haritasını belirledi. Ancak transistör boyutları atom ölçeğine yaklaştıkça (3 nm ve altı) fiziksel sınırlar belirginleşiyor. Dennard ölçeklemesinin de sona ermesiyle birlikte saat sıklığını artırma stratejisi sürdürülemez hale geldi ve tasarımcılar çok çekirdekli ve heterojen mimarilere yöneldi. Günümüzde başarımların artışı, tek çekirdeğin hızlandırılması yerine koşut hesaplama ve özel amaçlı hızlandırıcılarla (GPU, TPU, NPU) sağlanıyor.

Örnek 1.1

Moore yasasına göre transistör sayısı her 2 yılda ikiye katlanmaktadır.

- (a) 1971'de 2300 transistörlü Intel 4004'ten başlayarak, 2024'te beklenen transistör sayısını hesaplayın.
- (b) Gerçek değer (Apple M4 Ultra: ~46 milyar) ile karşılaştırın.

Çözüm:

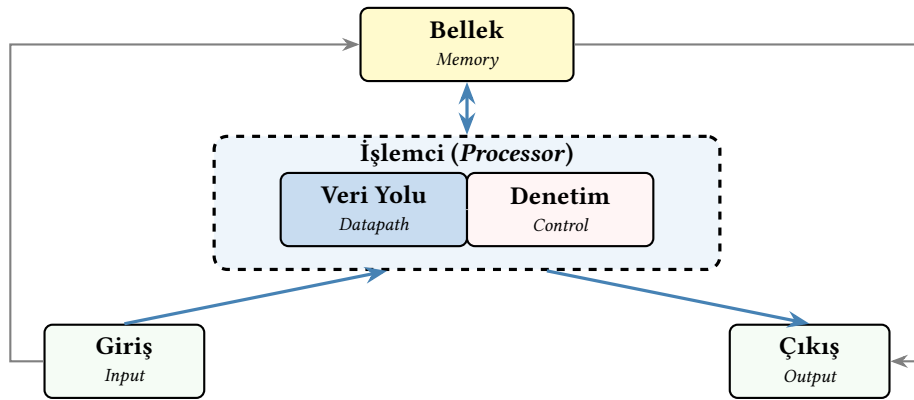
(a) 1971'den 2024'e kadar 53 yıl = 26,5 ikiye katlama dönemi geçmiştir:

$$N = 2300 \times 2^{26,5} \approx 2300 \times 94,9 \times 10^6 \approx 218 \times 10^9 \approx 218 \text{ milyar}$$

(b) Gerçek değer (~46 milyar), Moore yasasının öngörüsünün (~218 milyar) yaklaşık beşte biri kadardır. Bu sapmanın başlıca nedenleri: Dennard ölçeklemesinin 2006 civarında sona ermesi, saat sıklığının sabitlenmesi ve güç duvarının transistör sayısı artışını yavaşlatmasıdır. Yine de 53 yılda ~20 milyon katlık artış, insanlık tarihinin en hızlı teknolojik ilerlemesidir.

1.3 Bilgisayar Nedir?

Bilgisayar veri alabilen, aldığı veriler üzerinde işlemler yapan ve sonuç üreten bir makinedir. Bilgisayarlar veri yolu, denetim, bellek, giriş ve çıkış olmak üzere beş ana bileşenden oluşur. Şekil 1.3'te gösterildiği üzere veri yolu ve denetim birimleri işlemciyi oluşturur.



Şekil 1.3: Bilgisayarın 5 ana bileşeni

1.3.1 Bilgisayarın Parçaları

Günümüzde “bilgisayar” kavramı yalnızca masaüstü ya da dizüstü bilgisayarlarla sınırlı değil. Akıllı telefonlar, tabletler, akıllı saatler, sunucular ve araç içi gömülü sistemler de birer bilgisayar. Boyutları ve kullanım alanları farklı olsa da tüm bu aygıtlar Şekil 1.3’te gösterilen beş temel bileşeni içerir.

İşlemci

Bilgisayarın beyni olan *işlemci* (*processor*), programdaki buyrukları yürüten birimdir. Şekil 1.3’te görüldüğü gibi işlemci iki alt birimden oluşur: **veri yolu** (*datapath*) ve **denetim** (*control*). Veri yolu, aritmetik ve mantık işlemlerini gerçekleştiren donanım yollarını (toplayıcılar, yazmaçlar, çoklayıcılar gibi birimleri) barındırır. Denetim birimi ise her buyruğun hangi adımda ne yapacağını belirleyen sinyalleri üretir; bir orkestra şefi gibi veri yolundaki birimlerin doğru zamanda doğru işi yapmasını sağlar. İşlemci tasarımı Bölüm 6’da, boru hattıyla başarımlı artırma teknikleri ise Bölüm 7’de ayrıntılı olarak ele alınacaktır.

Bellek

Programların ve verilerin saklandığı bileşene *bellek* denir. Bilgisayarlarda iki temel bellek türü bulunur: uçucu (*volatile*) ve uçucu olmayan (*non-volatile*). Uçucu bellekler (örneğin DRAM) güç kesildiğinde içeriklerini kaybeder ve program çalışırken veriye hızlı erişim sağlamak için kullanılır. Uçucu olmayan bellekler ise (SSD, sabit disk gibi) güç olmadan da veriyi korur ve kalıcı depolama için tercih edilir. Bellek hiyerarşisi, hız ve maliyet dengesini sağlamak amacıyla farklı türdeki belleklerin katmanlı biçimde düzenlenmesiyle oluşturulur; bu konu Bölüm 8’de ayrıntılı olarak ele alınacaktır.

Giriş aygıtları

Bilgisayara veri sağlayan birimlere *giriş aygıtı* denir. Klavye ve fare onlarca yıldır en yaygın giriş aygıtları olmuştur; akıllı telefonlarda ise dokunmatik ekran hem giriş hem çıkış işlevi görür. Bunların yanı sıra ivmeölçer, GPS alıcısı ve parmak izi okuyucu gibi çeşitli algılayıcılar da birer giriş aygıtıdır.

Yapay zekâ çağıyla birlikte *mikrofon* ve *kamera* giderek daha önemli giriş aygıtları hâline geliyor. Büyük dil modelleri ve konuşma tanıma sistemleri sayesinde kullanıcılar bilgisayarla doğal dilde sesli iletişim kurabiliyor. Kameralar ise yalnızca fotoğraf çekmekle kalmayıp, yapay zekâ destekli görüntü anlama, yüz tanıma, hareket algılama ve artırılmış gerçeklik gibi uygulamalara girdi sağlıyor. Bu eğilim, mikrofon ve kameranın yakın gelecekte klavye ve farenin önüne geçeceğine işaret ediyor.

Giriş aygıtlarının hızları birbirinden çok farklıdır. Bir klavye saniyede birkaç bayt üretirken, bir mikrofon saniyede on binlerce ses örneği (*sample*) üretir; örneğin 16 bit, 48 kHz örnekleme ile saniyede yaklaşık 96 KB. Yüksek çözünürlüklü bir kamera ise saniyede gigabaytlar düzeyinde ham veri üretebilir. Bu büyük hız farkı, giriş/çıkış alt sisteminin tasarımında can alıcı bir etkidir ve Bölüm 9’da ayrıntılı olarak ele alınacaktır.

Çıkış aygıtları

İşlenen verilerin kullanıcıya ya da başka bir sisteme iletilmesini sağlayan birimlere *çıkış aygıtı* denir. Monitör (LCD, OLED) ve yazıcı en bilinen görsel çıkış aygıtlarıdır. Bir benneğin (*pixel*)

rengini göstermek için kırmızı, yeşil ve mavi kanalların her birine 8 bit ayrılır; dolayısıyla bir beneğin rengi 24 bitle ifade edilir. 1920×1080 çözünürlükteki bir ekranda yaklaşık 2 milyon benek bulunur ve bunların tamamının saniyede en az 60 kez yenilenmesi gerekir; bu da grafik alt sisteminden yüksek bir veri aktarım hızı talep eder.

Yapay zekânın yaygınlaşmasıyla birlikte *hoparlör* de en az ekran kadar önemli bir çıkış aygıtı hâline gelmiştir. Yapay zekâ destekli konuşma sentezi (*text-to-speech*) teknolojileri artık insandan ayırt edilemeyecek kadar doğal ses üretebiliyor. Akıllı hoparlörler, araç içi asistanlar ve giyilebilir aygıtlar gibi ekranı küçük ya da hiç olmayan sistemlerde ses, birincil çıkış kanalı olarak öne çıkıyor. Böylece giriş tarafında mikrofon ve kamera, çıkış tarafında ise hoparlör ve ekran birlikte çalışarak insanla bilgisayar arasında çok kipli (*multimodal*) bir etkileşim ortamı oluşturuyor.

Ana kart

Bilgisayarın bileşenlerini birbirine bağlayan ve uyumlu çalışmalarını sağlayan donanıma *ana kart* denir. Ana kart üzerinde işlemci yuvası, bellek yuvaları, giriş/çıkış denetleyicileri, genişleme yuvaları ve çeşitli bağlantı arabirimleri bulunur. Bileşenler arasındaki veri akışı, ana kart üzerindeki *yol (bus)* adı verilen iletişim hatları aracılığıyla sağlanır. Çağdaş ana kartlarda yüksek hızlı bileşenler için ayrı bir kuzey köprüsü, düşük hızlı bileşenler için ise güney köprüsü kullanılırdı; günümüzde bu işlevlerin çoğu doğrudan işlemci içine entegre edilmiştir.

Bilgisayar türleri

Yukarıda tanımlanan beş temel bileşen (işlemci, bellek, giriş aygıtları, çıkış aygıtları ve ana kart) farklı biçim ve boyutlarda bir araya gelebilir. Masaüstü bilgisayarlar bu bileşenleri ayrı parçalar hâlinde büyük bir kasada barındırırken, dizüstü bilgisayarlar aynı bileşenleri katlanabilir ince bir gövdeye sığdırır. Akıllı telefonlarda ise işlemci, bellek ve birçok denetleyici tek bir yonga üzerinde (*System on Chip*, SoC) bütünleştirilir. Sunucularda yüksek başarımlı ve güvenilirlik ön plandadır; gömülü sistemlerde ise düşük güç tüketimi ve küçük boyut öne çıkar. Şekil 1.4’te bu farklı bilgisayar türlerinin örnekleri gösterilmektedir.



Şekil 1.4: Farklı bilgisayar türleri

1.3.2 Gerçek Dünyayla Etkileşen Bilgisayarlar

Önceki alt bölümde bilgisayarın beş temel bileşenini tanıdık ve bunların masaüstünden akıllı telefona kadar çeşitli biçimlerde bir araya geldiğini gördük. Bu bileşenler yalnızca insan kullanıcıya hizmet eden aygıtlarla sınırlı değil: günümüzde giderek artan sayıda bilgisayar, gerçek dünyayı doğrudan algılayıp ona müdahale edebiliyor. İnsansı robotlar, sürücüsüz araçlar (otonom), insansız hava araçları (İHA) ve endüstriyel robot kolları bunun en çarpıcı örnekleridir (Şekil 1.5).



Şekil 1.5: Gerçek dünyayla etkileşen bilgisayar örnekleri

Beş bileşen aynı, giriş/çıkış farklı

Sayılan sistemlerin her biri özünde bir bilgisayardır ve Şekil 1.3'te gösterilen beş bileşeni eksiksiz barındırır. Onları farklı kılan şey giriş ve çıkış aygıtlarıdır. Geleneksel bir bilgisayarda giriş aygıtı klavye ve fare, çıkış aygıtı ise monitör ve hoparlördür. Oysa bir insansı robotta giriş aygıtları stereo kameralar (gözler), mikrofönler (kulaklar), LiDAR, kuvvet/tork algılayıcıları ve eklem konum algılayıcılarıdır. Çıkış aygıtları ise bacak ve kol eklemlerini hareket ettiren elektrik motorları ve sürücüleridir (*actuator*). Bir sürücüsüz araçta da durum farklı değildir: kameralar, radar, LiDAR ve ultrasonik algılayıcılar giriş sağlarken; direksiyon motoru, gaz ve fren denetleyicileri çıkış aygıtlarını oluşturur.

Gerçek zamanlılık gereksinimi

Çevreyle etkileşen sistemlerde en önemli mimari kısıt *gerçek zamanlılık (real-time)* gereksinimidir. Bir masaüstü bilgisayarda fare tıklamasına verilen yanıtın birkaç milisaniye gecikmesi fark edilmeyebilir; ancak saatte 100 km hızla giden bir sürücüsüz aracın çevreyi algılayıp karar vermesi için yalnızca birkaç on milisaniye süresi vardır. Bu nedenle bu tür sistemlerdeki işlemciler, saniyede milyonlarca algılayıcı verisini birleştirmeli (*sensor fusion*), yapay zekâ modelleriyle çevreyi yorumlamalı ve eyleyicilere komut göndermeli; tüm bunları kesin zaman sınırları içinde yapmalıdır. Bu gereksinim, özel yapay zekâ hızlandırıcılarının (GPU, NPU, TPU gibi) bu platformlara entegre edilmesinin temel nedenidir ve Bölüm 11'de ayrıntılı olarak ele alınacaktır.

Bilgisayar mimarisi açısından önemi

Gerçek dünyayla etkileşen bilgisayarlar, mimari tasarımı birçok yönden zorlamaktadır: algılayıcılardan gelen yüksek bant genişlikli veri akışları verimli giriş/çıkış alt sistemleri gerektirir; gerçek zamanlı yapay zekâ çıkarımı güçlü koşturucu işlem kapasitesi talep eder; enerji kaynağının sınırlı olması (pil) düşük güç tüketimini zorunlu kılar; güvenlik açısından can alıcı kararların deterministik zamanda tamamlanması gerekir. Bu gereksinimler, bilgisayar mimarisinin geleceğini şekillendiren en önemli itici güçlerden biri hâline gelmiştir.

1.4 İşlemci ve Yonga Üretimi

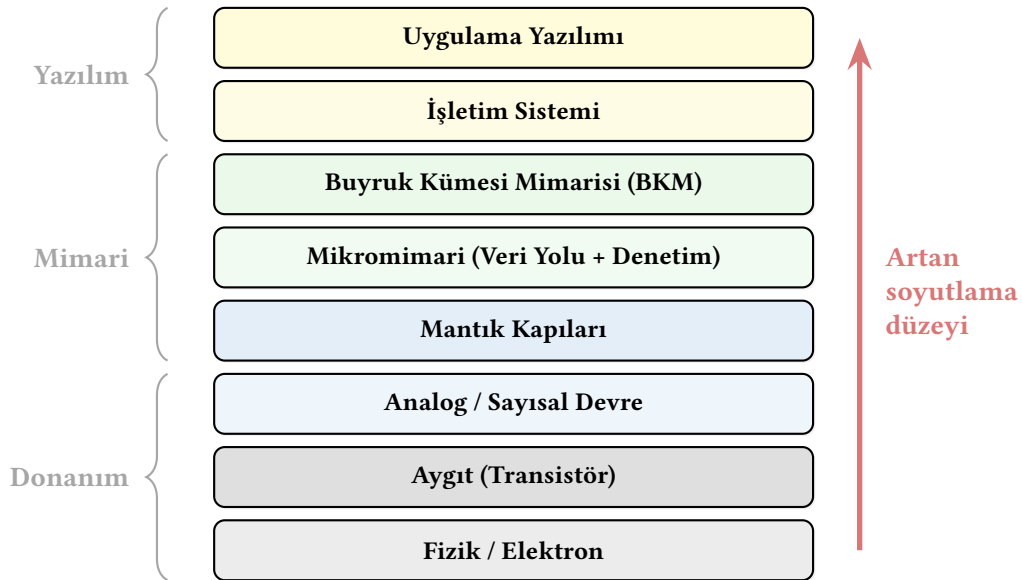
Bilgisayarın beş ana bileşeni arasında bilgisayarın kalbi sayılan **işlemci** özel bir yere sahiptir. İşlemci, bellekten okunan buyrukları yorumlayıp yürüten, aritmetik ve mantık işlemlerini gerçekleştiren ve tüm sistemi denetleyen merkezi birimdir. Bir bilgisayarın ne kadar hızlı çalışa-

cağı, ne kadar enerji tüketeceği ve hangi iş yüklerini kaldırabileceği büyük ölçüde işlemcisinin tasarımına bağlıdır. Bu nedenle bilgisayar mimarisinin en temel konusu işlemci tasarımıdır ve bu kitabın büyük bölümü işlemcinin iç yapısını, çalışma ilkelerini ve başarımını artırma yöntemlerini ele almaktadır. Şimdi bu can alıcı bileşenin nasıl üretildiğini inceleyelim.

1.4.1 İşlemcinin Yapılaşması

İlk olarak vakum tüpleri kullanılarak yapılan işlemci daha sonra transistörlerle yapılmaya başlandı. Yapısı oldukça karmaşık olan işlemcinin yapımında milyonlarca, hatta milyarlarca transistör kullanılabilir. Örneğin Apple M4 yongasında yaklaşık 28 milyar, AMD EPYC işlemcisi ise 50 milyardan fazla transistör bulunur. Bu denli büyük bir karmaşıklık yönetmek için işlemci tasarımının değişik aşamaları birbirinden soyutlanır. *Soyutlama*, her katmandaki ayrıntıları bir üst katmandan gizleyerek karmaşıklığın denetlenmesini kolaylaştırır. Örneğin bir derleyici, işlemcinin içindeki transistör bağlantılarını bilmek zorunda değildir; yalnızca buyruk kümesi düzeyindeki soyutlamayla çalışır. Benzer biçimde bir işlemci tasarımcısı, transistörlerin fiziksel üretim sürecini değil, mantık kapısı düzeyindeki davranışları temel alır. Şekil 1.6'da bilgisayar sistemindeki soyutlama katmanları gösterilmektedir.

Bu katmanlı yapı yalnızca karmaşıklığı yönetmekle kalmaz, aynı zamanda farklı uzmanlık alanlarının bağımsız çalışmasını sağlar. Bir yazılım geliştiricisi işlemcinin iç yapısını bilmeden program yazabilir; bir donanım mühendisi de üzerinde hangi yazılımların çalışacağını önceden kestirmek zorunda kalmadan işlemci tasarlayabilir. Bu bağımsızlık, bilgisayar endüstrisinin hızlı gelişiminin temel taşlarından biridir. Örneğin aynı x86 buyruk kümesi mimarisi, 1978'deki 8086 işlemcisinden günümüzdeki milyarlarca transistörlük işlemcilere kadar korunmuş; yazılımlar her yeni nesil donanımda yeniden yazılmadan çalışmaya devam etmiştir. Soyutlama katmanları arasındaki bu *arayüz sözleşmesi*, ilerleyen bölümlerde ayrıntılı olarak ele alınacak olan buyruk kümesi mimarisi kavramının ta kendisidir.



Şekil 1.6: Bilgisayar sistemindeki soyutlama katmanları. Her katman yalnızca bir altındaki katmanın sunduğu arayüzü kullanır.

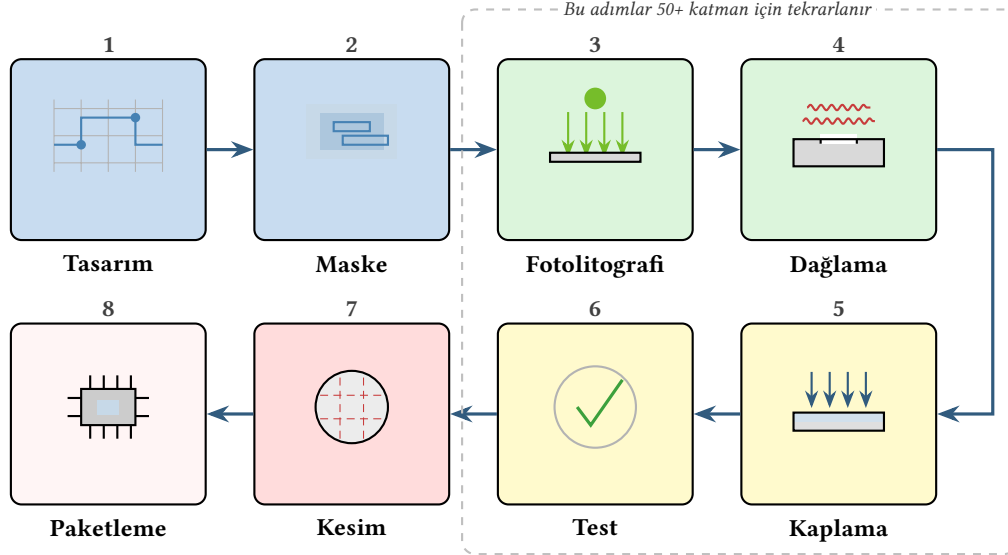
1.4.2 Tümleşik Devreler

Bölüm 1.2’de anlatıldığı gibi, tümleşik devrelerin keşfi bilgisayar tarihinin dönüm noktalarından biridir. Tümleşik devredeki ana fikir, transistörler, dirençler, kapasitörler ve bunlar arasındaki bağlantıların tamamının aynı yarıiletken malzeme üzerine tek bir süreçle üretilmesidir. Böylece ayrı bileşenlerden oluşan devrelere kıyasla çok daha küçük, hızlı ve güvenilir yongalar elde ediliyor.

Tümleşik devreler, içerdikleri transistör sayısına göre sınıflandırılır: birkaç düzine transistör barındıran *SSI* (Small Scale Integration), birkaç yüz transistörlük *MSI*, on binlerce transistörlük *LSI* ve yüz binlerin üzerine çıkan *VLSI* (Very Large Scale Integration: Çok Büyük Ölçekli Tümleştirme). Günümüzdeki işlemciler milyarlarca transistör içermekte olup artık “ultra-VLSI” olarak nitelendirilebilecek bir ölçekte. Transistör boyutlarının küçülmesi, aynı alana daha fazla transistör sığdırılmasını sağlamış; ancak beraberinde güç tüketimi, ısı yönetimi ve üretim maliyeti gibi yeni mühendislik zorlukları da getirmiştir.

Tümleşik devrelerin etkisi yalnızca boyut ve hız kazanımıyla sınırlı kalmamıştır. Ayrı bileşenlerle kurulan devrelerde her lehim noktası olası bir arıza kaynağıyken, tümleşik devrede bağlantılar fabrikada otomatik olarak oluşturulduğundan güvenilirlik büyük ölçüde artmıştır. Üstelik üretim miktarı arttıkça birim maliyet düşer: günümüzde birkaç milyar transistör içeren bir işlemci yongası, enflasyon etkisi çıkarıldığında 1960’ların birkaç yüz transistörlük devresinden daha ucuza üretilmektedir. Bu maliyet-başarımlı ilişkisi, bilgisayarların yalnızca laboratuvar ve iş dünyasında değil, cep telefonlarından bulaşık makinelerine kadar günlük yaşamın her köşesinde yer almasını olanaklı kılmıştır.

1.4.3 Yonga Üretim Süreci



Şekil 1.7: Yonga yapılma süreci. Çağdaş bir işlemcinin üretiminde fotolitografi–dağlama–kaplama adımları onlarca katman için tekrarlanır.

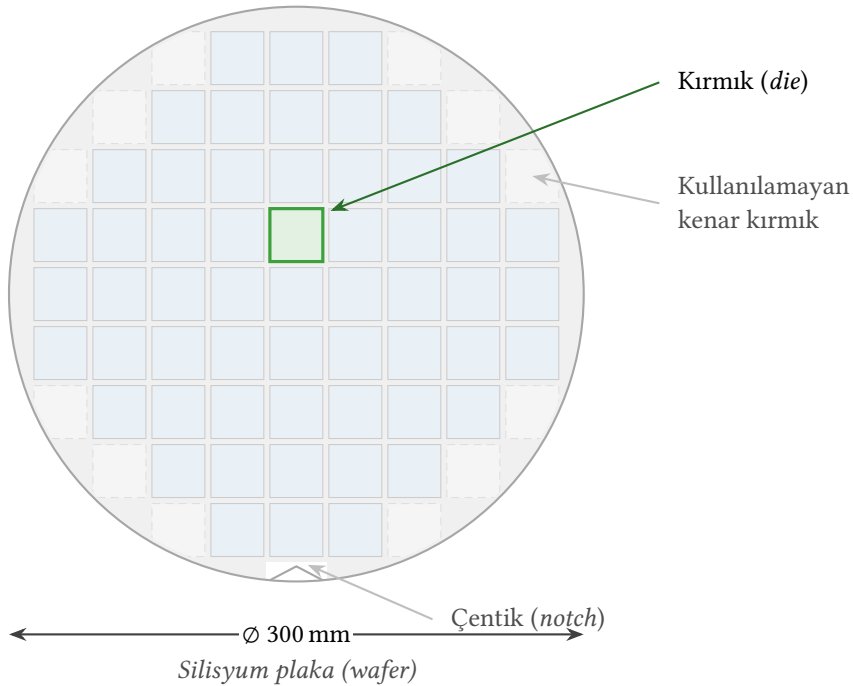
Yapılması planlanan yonga öncelikle bilgisayar üzerinde plan olarak çizilir ve denir. Işıklı baskı (photolithography) tekniğiyle tüm yongaların basılması için bir kalıp hazırlanır. Silisyum silindirden kristal bir testereyle kesilen ince katmanların üzerine katmanları birbirinden yalıtacak şekilde silisyum oksit kaplanır ve kalıbın ana hatlarının geçilebilmesi için ışığa duyarlı bir tabaka kaplanır. Yonga yapım süreci Şekil 1.7’de gösterilmiştir.

Bilgi Notu 1.3: Silisyum mu, Silikon mu?

Yarıiletken yongaların temel malzemesi olan kimyasal element İngilizcede *silicon* (Si, atom numarası 14), Türkçede ise **silisyum** olarak adlandırılır. Günlük dilde sıkça karışılan “silikon” sözcüğü ise İngilizce *silicone* kelimesinin karşılığıdır ve silisyum ile oksijenden türetilmiş sentetik bir polimeri ifade eder (mutfak kalıpları, conta malzemeleri vb.). Dolayısıyla yonga üretiminde kullanılan malzeme *silisyum*, ABD’deki ünlü teknoloji vadisinin adı ise doğru çevirisiyle *Silisyum Vadisi*’dir.

Işıklı baskı işlemi son derece hassas bir ortamda gerçekleştirilir. Üretim tesislerindeki *temiz oda* (*cleanroom*) koşullarında havadaki toz partikül sayısı dış ortamdan on binlerce kat azdır; çünkü nanometre ölçeğindeki devrelerde tek bir toz tanesi bile yongayı kullanılamaz hâle getirebilir. Bu nedenle yonga üretim tesisleri (*fab*), dünyada inşa edilen en pahalı endüstriyel yapılar arasında yer alır; günümüzde ileri düğüm teknolojisine sahip bir fabrikanın maliyeti 20 milyar doları aşabilmektedir.

Üretim sürecinin en kritik adımlarından biri, plaka üzerindeki deseni oluşturan fotolitografi basamağıdır. Günümüzde 5 nm ve altı düğüm teknolojilerinde *aşırı morötesi ışıklı baskı* (EUV: Extreme Ultraviolet Lithography) kullanılmaktadır. EUV, 13,5 nm dalga boylu ışık kullanılarak geleneksel 193 nm ArF lazerlerinin çözünürlük sınırlarını aşar ve daha ince hatların çizilmesini olanaklı kılar. Ancak EUV makineleri, içinde plazma kaynağı, vakum odaları ve çok katmanlı aynalar barındıran son derece karmaşık sistemlerdir; dünyada yalnızca tek bir firma (ASML) bu makineleri üretebilmektedir.



Şekil 1.8: Bir silisyum plakadan yongaların kesilmesi

Bilgi Notu 1.4: Silisyum plaka neden yuvarlak?

Öğrenciler sıklıkla “Plaka neden kare değil, kare olsa daha fazla kırmık sığmaz mı?” diye sorar. Yanıt üretim sürecinde yatmaktadır. Silisyum plakalar *Czochralski yöntemi* ile üretilir: çok saf silisyum bir potada eritilir, dönen bir tohum kristal erimiş silisyuma dokundurulur yavaşça yukarı çekilir. Dönerek çekilme hareketi doğal olarak silindirik bir tek kristal kütle (*ingot*) oluşturur; bu silindirin enine kesitleri daire biçimindedir. Kare plaka üretmek istendiğinde silindirik kütleden kesim sırasında büyük miktarda malzeme israf olurdu. Ayrıca yüksek sıcaklıklı üretim aşamalarında (900–1100 °C) kare plakanın köşelerinde gerilim yoğunlaşması oluşarak kırılma riski artardı. Yuvarlak geometri ise termal genleşme sırasında gerilimi eşit biçimde dağıtır.

1.4.4 Teknoloji Düzümleri ve Gelişimi

Bir yonga üretim sürecinin en temel parametresi *teknoloji düğümü* (*technology node*) olarak adlandırılan ve nanometre (nm) cinsinden ifade edilen boyuttur. Tarihsel olarak bu değer transistörün kapı uzunluğuna karşılık gelirdi; ancak günümüzde gerçek fiziksel boyutlarla doğrudan ilişkisi zayıflamış ve daha çok bir nesil göstergesi hâline gelmiştir. Her yeni teknoloji düğümüne geçişte transistör yoğunluğu yaklaşık iki katına çıkar, bu da aynı alana daha fazla transistör sığdırılmasını sağlar.

Tablo 1.2: Yıllar içinde teknoloji düğümlerinin gelişimi. Transistör yoğunlukları yaklaşık değerler olup üretim tesisine ve tasarıma göre değişir.

Düğüm	Yıl	Yoğunluk (MTr/mm ²)	Öncü Üretici	Örnek Yonga
90 nm	2004	~15	Intel	Pentium 4 Prescott
65 nm	2006	~30	Intel	Core 2 Duo
45 nm	2008	~60	Intel	Core i7 (Nehalem)
32 nm	2010	~90	Intel	Sandy Bridge
22 nm	2012	~160	Intel	Ivy Bridge (FinFET)
14 nm	2014	~310	Intel, Samsung	Skylake, Exynos 7
10 nm	2017	~530	TSMC	Apple A11
7 nm	2018	~910	TSMC	Apple A12, AMD Zen 2
5 nm	2020	~1 710	TSMC	Apple M1, A14 Bionic
3 nm	2022	~3 000	TSMC, Samsung	Apple M3, A17 Pro

Tablo 1.2’den görüldüğü üzere, 2004’ten 2022’ye kadar transistör yoğunluğu yaklaşık 200 kat artmıştır. Ancak bu kazanımlar bedelsiz değildir: teknoloji düğümü küçüldükçe hem plaka maliyeti hem de *maske seti* (*mask set*) maliyeti dramatik biçimde artar. Maske seti, yonga üzerindeki her bir katmanın desenini plakaya aktarmak için kullanılan fotolitografi kalıplarının tamamıdır; çağdaş bir işlemcide 80’den fazla maske katmanı bulunabilir ve her maske yüksek duyarlılıkta üretilmek zorundadır. Tablo 1.3 bu maliyet artışını göstermektedir.

3 nm düğümünde bir maske setinin maliyeti 75 milyon doları aşabilmektedir. Bu durum, ileri düğümlerde üretimi yalnızca çok yüksek hacimli ürünler (akıllı telefon işlemcileri, veri merkezi yongaları) için ekonomik kılar. Düşük hacimli uygulamalar (IoT algılayıcıları, otomotiv denetleyicileri gibi) maliyet açısından 28 nm veya 16 nm gibi olgun düğümlerde üretilmeye devam eder. Bu ekonomik gerçeklik, farklı uygulama alanlarının farklı teknoloji düğümlerinde kalmasına neden olur ve “en yeni teknoloji her zaman en iyi seçimdir” yanılgısını çürütür.

Tablo 1.3: Farklı teknoloji düğümlerinde 300 mm silisyum plakanın ve maske setinin yaklaşık maliyeti. Maliyetler yonga üretim tesisine ve sipariş hacmine göre değişkenlik gösterir.

Düğüm	Plaka Maliyeti (\$)	Maske Seti (\$)	Tipik Kullanım Alanı
28 nm	3 000–4 000	~5 M	IoT, otomotiv, gömülü
16/14 nm	5 000–6 000	~15 M	Orta seviye taşınabilir, ağ
7 nm	9 000–10 000	~30 M	Üst seviye taşınabilir, GPU
5 nm	16 000–17 000	~50 M	En üst düzey işlemciler
3 nm	20 000+	~75 M	Son nesil yongalar

M = milyon; maliyetler tahmini olup üretim tesisine ve hacme göre değişir.

1.4.5 Üretim Maliyeti ve Verim

Yonga üretiminin ekonomisini anlamak, mimari kararlar için can alıcıdır. Büyük alanlı yongalar daha fazla transistör barındırabilir ancak verim düşüşü nedeniyle orantısız biçimde pahalılaşır. Tasarımcılar yonga alanı, üretim maliyeti ve verim arasındaki dengeyi gözetmek zorundadır.

Bir yonganın üretim maliyeti aşağıdaki denklemle gösterilebilir:

$$\text{Yonga maliyeti} = \frac{\text{Plaka maliyeti}}{\text{Plakadaki yonga sayısı} \times \text{Verim}} \quad (1.1)$$

Plakanın alacağı yonga sayısı yaklaşık olarak aşağıdaki eşitlikle gösterilebilir:

$$\text{Plakadaki yonga sayısı} = \frac{\text{Plaka alanı}}{\text{Yonga alanı}} \quad (1.2)$$

Verim yaklaşık olarak aşağıdaki denklemle modellenir:

$$\text{Verim} = \left(\frac{1}{1 + \text{Birim alan başına hata sayısı} \times \text{Yonga alanı}} \right)^N \quad (1.3)$$

Bu denklemdeki kesirin paydasındaki değerin üssü (N) üretim sürecindeki kritik aşama sayısı ile ilişkilidir.

Verim Öğrenme Eğrisi

Yeni bir teknoloji düğümü üretime girdiğinde verim genellikle düşüktür; tipik olarak %30–50 arasında başlar. Üretim süreci olgunlaştıkça ve hata kaynakları belirlendikçe verim kademeli olarak %80–90 ve üzerine çıkar. Bu iyileşme sürecine *verim öğrenme eğrisi* (*yield learning curve*) denir ve bir yonga üretim tesisinin teknolojik olgunluğunun en önemli göstergesidir.

Verim verisi şirketler tarafından ticari sır olarak korunduğundan kesin rakamlar kamuya açıklanmaz; ancak genel eğilimler endüstri analistleri tarafından raporlanır. Örneğin TSMC'nin 7 nm süreci (N7) 2018'de üretime girdiğinde verimlerin birkaç çeyrek dönem içinde %80'in üzerine çıktığı tahmin edilmektedir. Denklem 1.1'den görüldüğü üzere verim yarıya düşerse yonga maliyeti iki katına çıkar; bu nedenle yonga üreticileri yeni düğümlerde verimi artırmak için büyük mühendislik yatırımları yapar.

Örnek 1.2

Bir yonga tasarımı şirketi, akıllı telefon işlemcisi için 3 nm teknolojisi kullanmayı düşünüyor. Yonganın kırmık alanı 100 mm^2 olacak. 300 mm çaplı bir plakanın maliyeti 20 000 \$, maske setinin maliyeti 75 M\$, olgun süreçteki verim %75 olsun. Bir kırmığın üretim maliyetini ve maske seti payını iki farklı üretim hacmi için hesaplayın: (a) 100 milyon adet, (b) 1 milyon adet.

Çözüm:

Plaka alanı: $A_{\text{plaka}} = \pi \times 150^2 \approx 70\,686 \text{ mm}^2$.

Plakadaki kırmık sayısı (yaklaşık): $70\,686 \div 100 \approx 706$ kırmık. Kenar kayıpları düşüldüğünde kullanılabilir kırmık sayısı yaklaşık 600'dür.

Verimli kırmık sayısı: $600 \times 0,75 = 450$ iyi kırmık/plaka.

Plaka başına kırmık maliyeti (Denklem 1.1): $20\,000 \div 450 \approx 44,4 \text{ $/kırmık}$.

Maske seti payı:

100 milyon adet: $75\,000\,000 \div 100\,000\,000 = 0,75 \text{ $/kırmık} \Rightarrow \text{toplam} \approx 45 \text{ $/kırmık}$

1 milyon adet: $75\,000\,000 \div 1\,000\,000 = 75 \text{ $/kırmık} \Rightarrow \text{toplam} \approx 119 \text{ $/kırmık}$

Karşılaştırma: Aynı yonga 28 nm'de üretilseydi plaka maliyeti ~3 500 \$, maske seti ~5 M\$, verim ~%90 olacaktı. Plaka başına maliyet ~6,5 \$/kırmık, 1 milyon adet için maske payı 5 \$/kırmık, toplam ~11,5 \$/kırmık; bu 3 nm'dekinin onda biridir. Ancak 28 nm'de aynı alana sığan transistör sayısı 3 nm'nin yüzde biri kadar olduğundan, aynı işlevsellik çok daha büyük bir kırmık alanı gerektirir.

Bu örnek, ileri teknoloji düğümlerinin neden yalnızca yüksek hacimli ürünlerde ekonomik olduğunu ve maske seti maliyetinin düşük hacimlerde baskın hâle geldiğini göstermektedir.

Örnek 1.3

Çapı 300 mm olan bir silisyum plakanın maliyeti 12 000 \$ olsun. Üretilcek yonganın alanı 200 mm^2 , birim alan başına hata sayısı 0,03 hata/ mm^2 ve kritik aşama sayısı $N = 12$ ise bir yonganın maliyetini hesaplayın.

Çözüm:

Plaka alanı: $A_{\text{plaka}} = \pi \times 150^2 \approx 70\,686 \text{ mm}^2$.

Plakadaki yonga sayısı: $\frac{70\,686}{200} \approx 353$ yonga.

Verim: $\left(\frac{1}{1 + 0,03 \times 200} \right)^{12} = \left(\frac{1}{7} \right)^{12} \approx 7,3 \times 10^{-11}$; bu son derece düşük bir verimdir!

Gerçek üretimde birim alan başına hata sayısı çok daha düşüktür. Eğer hata yoğunluğu 0,001 hata/ mm^2 olsaydı:

$$\text{Verim} = \left(\frac{1}{1 + 0,001 \times 200} \right)^{12} = \left(\frac{1}{1,2} \right)^{12} \approx 0,112$$

Bu durumda yonga maliyeti: $\frac{12\,000}{353 \times 0,112} \approx 303 \text{ $}$.

Bu örnek, yonga alanının ve hata yoğunluğunun maliyet üzerindeki dramatik etkisini açıkça gösteriyor.

Örnek 1.4

300 mm çapında bir silisyum plakadan 100 mm^2 büyüklüğünde kırmıklar (*die*) kesilmektedir. Plaka başına hata yoğunluğu $d = 0,04$ hata/ cm^2 ve üretim karmaşıklığı parametresi $\alpha = 4$ ise:

- (a) Plakadan elde edilen yaklaşık kırmık sayısını hesaplayın.
- (b) Kırmık verimini hesaplayın.
- (c) Kırmık boyutu 50 mm^2 'ye küçültülürse verim nasıl değişir?

Çözüm:

(a) Plaka alanı $A_{\text{plaka}} = \pi \times (150)^2 \approx 70\,686 \text{ mm}^2$. Kırmık sayısı yaklaşık:

$$N_{\text{kırmık}} \approx \frac{A_{\text{plaka}}}{A_{\text{kırmık}}} = \frac{70\,686}{100} \approx 707 \text{ kırmık}$$

(Kenar kayıpları düşüldüğünde uygulamada bu sayı ~ 600 civarına düşer.)

(b) Verim formülü:

$$V = \left(1 + \frac{d \times A_{\text{kırmık}}}{\alpha}\right)^{-\alpha} = \left(1 + \frac{0,04 \times 1}{4}\right)^{-4} = (1,01)^{-4} \approx 0,961 = \%96,1$$

(Burada $A_{\text{kırmık}} = 100 \text{ mm}^2 = 1 \text{ cm}^2$.)

(c) $A_{\text{kırmık}} = 50 \text{ mm}^2 = 0,5 \text{ cm}^2$ için:

$$V = \left(1 + \frac{0,04 \times 0,5}{4}\right)^{-4} = (1,005)^{-4} \approx 0,980 = \%98,0$$

Kırmık boyutu yarıya düşünce verim $\%96,1$ 'den $\%98,0$ 'a yükselir ve plaka başına çalışan kırmık sayısı iki kattan fazla artar. Bu, Moore yasasını destekleyen ekonomik motivasyonun önemli bir parçasıdır: daha küçük transistör $\rightarrow$ daha küçük kırmık $\rightarrow$ daha yüksek verim $\rightarrow$ daha düşük birim maliyet.

1.5 Programların Çalıştırılması

Şimdiye kadar bilgisayarın fiziksel bileşenlerini tanıdık: işlemci, bellek, giriş/çıkış aygıtları ve bunların bir yonga üzerinde nasıl üretildiği. Ancak donanım tek başına anlamsızdır; onu yararlı kılan şey üzerinde çalışan programlardır. Bu bölümde donanımla yazılım arasındaki köprüyü inceleyeceğiz: programlar hangi dilde yazılır, belleğe nasıl yerleştirilir, işlemci tarafından nasıl yorumlanır ve bir bütün olarak nasıl yürütülür?

1.5.1 Makine Dili ve Çevirici Dil

Bir programı makine üzerinde çalıştırmak için öncelikle makineyle aynı dilin konuşulması gerekir. Bilgisayarların alfabesi 1'ler ve 0'lardan oluşur; bu alfabedeki harflerden oluşan sözcükler ise ikilik tabanda yazılmış sayılardır. Sözcüklerdeki 1 ya da 0 rakamlarından oluşan her bir basamağa "Binary Digit" veya kısaca "Bit" denir. Sekiz bitin bir araya gelmesiyle bir *bayt* (*byte*) oluşur; bayt, bilgisayarların en temel adreslenebilir veri birimidir. Bilgisayarların anladığı kelimeler 1 ve 0'lardan oluştuğu için üst düzey dilde yazılanların bilgisayar tarafından

anlaşılması bunların ikilik düzene çevrilmesiyle mümkün olur.

Makine dili, işlemcinin doğrudan anlayıp yürütebildiği tek dildir. Her makine dili buyruğu üç temel bilgi taşır: (1) yapılacak işlemi belirleyen *işlem kodu* (*opcode*), (2) işlemin üzerinde gerçekleştirileceği verilerin konumunu gösteren *işlenen* (*operand*) alanları ve (3) buyruğun biçimini tanımlayan ek denetim bitleri. Örneğin RISC-V mimarisinde bir toplama buyruğu 32 bit genişliğindedir ve bu bitler arasında işlem kodu, kaynak yazmaçlarını belirten alanlar ve hedef yazmacı belirten alan yer alır. İlk bilgisayarlarda programcılar bu ikilik kodları elle yazıyordu; yüzlerce satırlık bir programda tek bir bitin yanlış girilmesi saatlerce hata aramayı gerektiriyordu.

Bu zorlukları aşmak için 1950’lerde *çevirici dil* (*Assembly*) geliştirildi. Çevirici dil, ikilik kodlar yerine insanların okuyabileceği kısa anımsatıcılar (*mnemonic*) kullanır: add toplama, sub çıkarma, lw bellekten yükleme anlamına gelir. Her anımsatıcı, tam olarak bir makine buyruğuna karşılık düşer. Çevirici dilde yazılmış kaynak kodu, *çevirici* (*Assembler*) adı verilen bir program tarafından makine diline dönüştürülür. Bu dönüşüm bire bir olduğu için çevirici dil, makine diline en yakın programlama düzeyidir. Aşağıdaki örnek bu ilişkiyi somutlaştırır:

Düzye	Gösterim	Açıklama
Çevirici dil	add x5, x6, x7	x6 ile x7’yi topla, sonucu x5’e yaz
Makine dili (ikilik)	0000000 00111 00110 000 00101 0110011	32 bitlik R-tipi RISC-V buyruğu
Makine dili (onaltılık)	0x007302B3	Aynı buyruğun onaltılık gösterimi

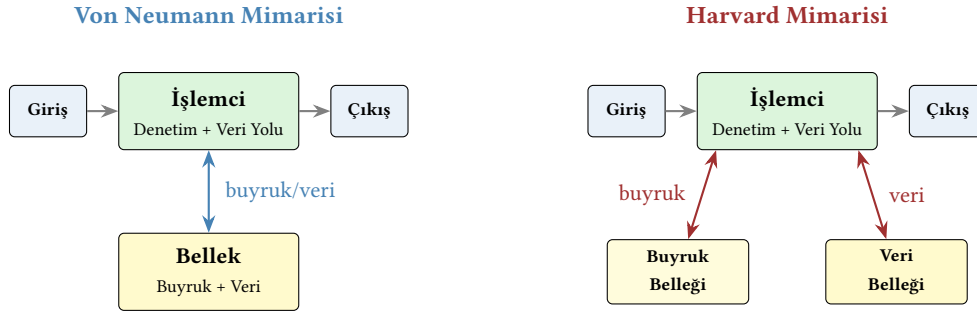
Çevirici dilin en önemli avantajı, programcının kaynak ve hedef yazmaçlarını (x5, x6, x7) isimle görmesini sağlaması ve işlem kodlarını anlamlı kısaltmalarla temsil etmesidir. Bununla birlikte çevirici dil, mimariye özgüdür: bir RISC-V işlemcisi için yazılmış çevirici dil kodu, x86 ya da ARM işlemcisinde çalışmaz. Bu sınırlama, ilerleyen alt bölümlerde ele alınacak üst düzey dillerin ve derleyicilerin geliştirilmesinin temel motivasyonlarından biri olmuştur.

1.5.2 Von Neumann ve Harvard Mimarileri

Buyruklar ve veriler bellekte nasıl saklanmalı? Bu soru, bilgisayar mimarisinin en temel tasarım kararlarından birini oluşturur ve tarihsel olarak iki farklı yaklaşım geliştirilmiştir.

Von Neumann mimarisi, John Von Neumann’ın 1945’te EDVAC tasarımı için kaleme aldığı raporda [43] ortaya koyduğu ilkelere dayanır. Bu mimaride buyruklar ve veriler tek bir ortak bellekte saklanır; işlemci aynı yol üzerinden hem buyruğu çeker hem de veriyi okur/yazar. Ortak bellek kullanımının en büyük avantajı esnekliktir: programlar veriyle aynı biçimde depolanabilir, kopyalanabilir ve değiştirilebilir. Bu sayede bir program başka bir programı belleğe yükleyebilir; işletim sistemlerinin ve derleyicilerin çalışma ilkesi tam da budur. Ancak tek yoldan hem buyruk hem veri geçmesi kaçınılmaz bir yarışma yaratır: işlemci aynı anda buyruğu çekip veriyi okuyamaz. Bu sınırlama *Von Neumann darboğazı* olarak bilinir ve günümüzde bile başarımı etkileyen temel sorunlardan biridir.

Harvard mimarisi bu darboğazı ortadan kaldırmayı hedefler. İsmi 1944’te Howard Aiken’in Harvard Üniversitesi’nde tasarladığı Mark I bilgisayarından alan bu yaklaşımda buyruklar ve veriler fiziksel olarak ayrı belleklerde tutulur ve her birinin kendi yolu vardır. İşlemci, aynı anda buyruk belleğinden bir sonraki buyruğu çekerken veri belleğinden de ihtiyaç duyduğu veriyi okuyabilir. Bu koşut erişim, özellikle gerçek zamanlı sinyal işleme gibi bant genişliğine duyarlı uygulamalarda belirgin bir başarımlı artışı sağlar. Dezavantajı ise esneklik kaybıdır: buyruk ve veri belleklerinin boyutları tasarım zamanında belirlenmek zorundadır ve biri boşken diğeri doluysa kapasite israf edilir.



Şekil 1.9: Von Neumann mimarisinde buyruk ve veri aynı belleği paylaşır (tek yol); Harvard mimarisinde ise ayrı belleklerde tutulur (iki bağımsız yol).

Günümüzde çoğu genel amaçlı işlemci, her iki yaklaşımın avantajlarını birleştiren *değiştirilmiş Harvard mimarisi* kullanır: ana bellekte buyruklar ve veriler birlikte saklanır (Von Neumann esnekliği), ancak işlemcinin içindeki ön belleklerde buyruk ön belleği ile veri ön belleği ayrı tutulur (Harvard bant genişliği). Bu tasarım, Bölüm 8’de ele alınacak bellek hiyerarşisinin temel yapı taşlarından biridir.

1.5.3 RISC ve CISC Tasarım Felsefeleri

Bir buyruk kümesi mimarisi tasarlanırken karşılaşılan temel soru şudur: buyruklar ne kadar karmaşık olmalı? Tarihsel süreçte bu soruya iki farklı felsefe yanıt vermiştir.

CISC (Complex Instruction Set Computer: Karmaşık Buyruk Kümeli Bilgisayar) yaklaşımı, az sayıda buyrukla çok iş yapmayı hedefler. CISC mimarisinde tek bir buyruk, bellekten veri okuma, üzerinde işlem yapma ve sonucu belleğe yazma gibi birden fazla adımı bir arada gerçekleştirebilir. Bu yaklaşım, belleğin pahalı ve yavaş olduğu dönemlerde programları küçük tutmak amacıyla geliştirildi: daha az buyruk demek daha az bellek kullanımı demekti. Intel’in x86 ailesi, CISC felsefesinin en yaygın temsilcisidir.

RISC (Reduced Instruction Set Computer: İndirgenmiş Buyruk Kümeli Bilgisayar) yaklaşımı ise 1980’lerde Berkeley ve Stanford üniversitelerindeki araştırmalardan doğdu. RISC felsefesinde her buyruk yalnızca bir temel işlem yapar, tüm buyruklar sabit uzunluktadır ve bellek erişimi yalnızca özel yükleme (lw) ve saklama (sw) buyruklarıyla gerçekleştirilir; aritmetik işlemler doğrudan bellekteki veriler üzerinde değil, yazmaçlardaki değerler üzerinde yapılır. Bu ayırım *yükle-sakla mimarisi* (load-store architecture) olarak bilinir. ARM ve RISC-V, RISC felsefesinin günümüzdeki önemli temsilcileridir.

Farkı somut bir örnekle görelim. “Bellekteki bir değeri bir yazmacın üstüne ekle” işlemini düşünelim (bu işlem C dilinde $A = A + B[i]$ gibi bir ifadeye karşılık gelir):

	CISC (x86 benzeri)	RISC (RISC-V benzeri)
Buyruk dizisi	add eax, [ebx+ecx*4] (Tek buyruk: bellek oku + topla)	slli t1, t0, 2 add t1, s1, t1 lw t2, 0(t1) add s0, s0, t2 (Dört buyruk: kaydır + adres hesapla + yükle + topla)
Buyruk sayısı	1	4
Buyruk uzunluğu	Değişken (1–15 bayt)	Sabit (4 bayt)
Donanım karmaşıklığı	Yüksek (karmaşık çözücü)	Düşük (yalın boru hattı)

Bu örnekte görüldüğü gibi CISC, aynı işi daha az buyrukla yapabilir; ancak her buyruğun uzunluğu ve yürütüm süresi farklı olduğundan donanım tasarımı karmaşılaşır. RISC ise daha fazla buyruk kullanır; buna karşılık her buyruğun sabit uzunlukta ve öngörülebilir sürede tamamlanması, boru hattı (*pipeline*) tasarımını kolaylaştırır ve saat sıklığının artırılmasına olanak tanır.

Tablo 1.4: RISC ve CISC tasarım felsefelerinin karşılaştırması

Özellik	RISC	CISC
Buyruk sayısı	Az (tipik 50–150)	Çok (tipik 500–1500+)
Buyruk uzunluğu	Sabit (32 bit)	Değişken (8–120+ bit)
Bellek erişim modeli	Yalnızca yükle/sakla	Doğrudan bellekle işlem
Yazmaç sayısı	Çok (32+)	Az (8–16)
Adresleme kipleri	Az ve yalın	Çok ve çeşitli
Donanım karmaşıklığı	Düşük	Yüksek
Derleyici karmaşıklığı	Yüksek	Düşük
Program boyutu	Büyük	Küçük
Örnekler	RISC-V, ARM, MIPS	x86, VAX, IBM z/Architecture

Günümüzde bu iki felsefe arasındaki sınır eskisi kadar keskin değildir. Çağdaş x86 işlemcileri, karmaşık CISC buyruklarını donanım içinde basit *mikro-işlemlere* (*micro-op*) böler ve bu mikro-işlemleri RISC benzeri bir boru hattında yürütür. Öte yandan RISC mimarileri de özelleşmiş buyruklar (vektör uzantıları, atomik işlemler gibi) ekleyerek buyruk kümelerini genişletmiştir. Sonuç olarak bugünün başarımları farkları mimari felsefeden çok mikromimari tasarım kararlarına (önbellek boyutuna, boru hattı derinliğine, dal öngörücüsü kalitesine) bağlıdır.

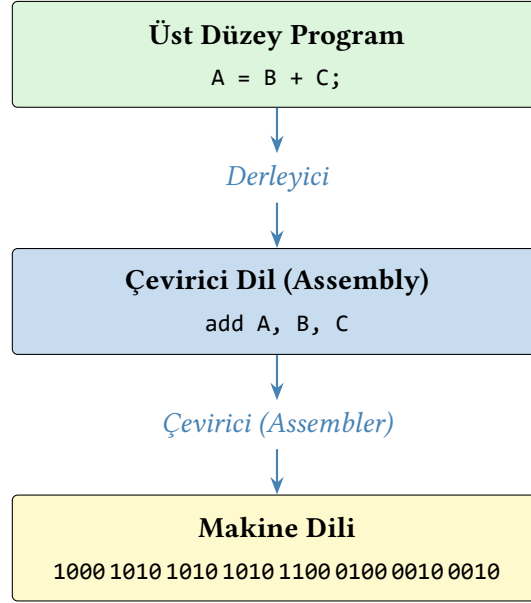
Bilgi Notu 1.5: RISC-V: Bu Kitabın Buyruk Kümesi

Bu kitapta buyruk kümesi mimarisi olarak açık kaynaklı **RISC-V** kullanılıyor. RISC, *Reduced Instruction Set Computer* (İndirgenmiş Buyruk Kümeli Bilgisayar) kısaltmasıdır ve 1980’lerde Berkeley ile Stanford’da geliştirilen tasarım felsefesini ifade eder. Berkeley’de bu felsefe çerçevesinde art arda beş nesil işlemci tasarlandı: RISC-I (1981), RISC-II (1983), SOAR (1984), SPUR (1988) ve son olarak 2010’da başlayan proje. Beşinci nesil olduğu için adındaki “V” Romen rakamı 5’tir.

RISC-V “**risk beş**” diye okunur (*risk five*); “risk-vee” okuyuşu yanlıştır. Türkçe ek alırken son ses /ş/ sert ünsüz olduğundan sert ünsüz uyumu uygulanır: RISC-V’te, RISC-V’ten, RISC-V’in, RISC-V’e. RISC-V’in neden tercih edildiği, tescilli mimarilerle karşılaştırması ve endüstrideki yeri Bölüm 1.9’da ayrıntılı olarak ele alınıyor.

1.5.4 Üst Düzey Diller ve Derleme Zinciri

Mimari tasarımındaki gelişmeler ve buyrukların çeviricilerle makine koduna çevrilmesi kullanıcılar için büyük kolaylık sağladı. Ancak bilimsel çalışmalar, modellemeler ve karmaşık problemler için daha karmaşık programlama yöntemlerine gereksinim duyuldu ve bu gereksinimi karşılamak için üst düzey diller oluşturuldu. Üst düzey dillerde neredeyse metin gibi okunup anlaşılabilir kadar anlamlı yapılar kullanılır. Üst düzey bir dilde yazılmış kodları çevirici dile dönüştüren *derleyici* (*Compiler*) programlar yazıldı. Programcı $A = B + C$; yazdığında, derleyici bu kodu `add A, B, C` haline getirir ve çevirici de bu kodu `100010101010101100010000100010` olarak makine diline çevirir. Şekil 1.10’da üst düzey bir dilde yazılmış bir programın nasıl en alt düzeye çevrildiği bir RISC-V işlemcisi örneğiyle gösteriliyor.



Şekil 1.10: Bir üst düzey programın alt düzey denetim işaretlerine çevrim aşamaları

1.5.5 Yazılım Katmanları

Üst düzey dillerin ortaya çıkması, programların yazılmasını ve okunmasını büyük ölçüde hızlandırdı. Farklı ihtiyaçlara uygun diller geliştirildi: bilimsel hesaplamalar için Fortran (1957), ticari uygulamalar için COBOL (1959), sistem programlama için C (1972), nesne yönelimli geliştirme için Java (1995) ve C++ (1985). Ancak tek başına bir programlama dili yeterli değildir; dil yalnızca programları ifade etmenin bir yoludur. Programların çalışması için altlarında katmanlı bir yazılım altyapısı gerekir.

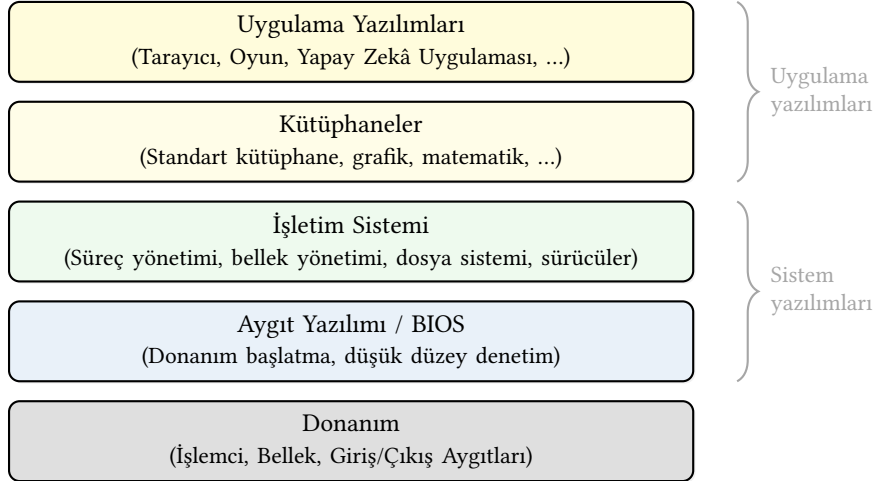
Yazılımlar kullanılma amaçlarına göre *sistem yazılımları* ve *uygulama yazılımları* olmak üzere iki gruba ayrılır:

- *Uygulama yazılımları*, kullanıcının doğrudan etkileştiği programlardır: tarayıcılar, metin düzenleyiciler, oyunlar, hesap çizelgeleri, yapay zekâ uygulamaları gibi. Kullanıcı bir uygulamayı çalıştırdığında, aslında alttaki tüm katmanları dolaylı olarak kullanır.
- *Sistem yazılımları*, donanım ile uygulama yazılımları arasında köprü kurar ve donanım kaynaklarını yönetir. Bu gruptaki en önemli iki bileşen *işletim sistemi* ve *derleyicidir*.

İşletim sistemi (Operating System), bilgisayarın kaynaklarını (işlemci zamanı, bellek alanı, giriş/çıkış aygıtları) uygulamalar arasında paylaştırır ve düzenler. Birden fazla programın aynı anda çalışabilmesi (çoklu görev), dosya sisteminin yönetilmesi ve kullanıcıyla etkileşim sağlanması işletim sisteminin sorumluluğundadır. Linux, Windows, macOS, Android ve iOS günümüzün en yaygın işletim sistemleridir.

Derleyici (Compiler), üst düzey dilde yazılmış kaynak kodu önce çevirici dile, ardından çevirici aracılığıyla makine diline dönüştüren programdır. Derleyicinin görevi yalnızca çeviri yapmak değildir; aynı zamanda programı eniyileyerek (döngü açma, ölü kod eleme, yazmaç ataması gibi tekniklerle) donanımdan en yüksek başarıyı elde etmeye çalışır. Bu nedenle derleyici kalitesi, aynı kaynak kodunun farklı donanımlarda ne kadar hızlı çalışacağını doğrudan etkiler.

Bu katmanlar arasındaki ilişki Şekil 1.11’de gösterilmektedir. Her katman yalnızca bir altındaki katmanın sunduğu *arayüzü* kullanır; altındaki katmanın iç ayrıntılarını bilmek zorunda değildir. Bu yapı, Şekil 1.6’da ele alınan soyutlama ilkesinin yazılım tarafındaki yansımasıdır.



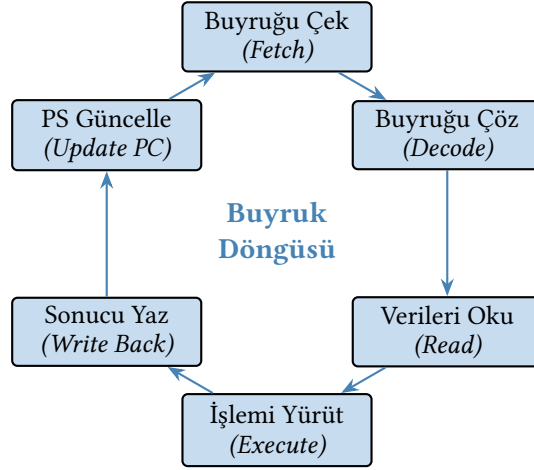
Şekil 1.11: Yazılım katmanları ve donanım arasındaki ilişki

Kütüphaneler (library), sık kullanılan işlevlerin bir araya getirildiği hazır kod koleksiyonlarıdır. Bir programcı matematiksel bir fonksiyon hesaplamak ya da ekrana görüntü çizmek istediğinde ilgili kütüphaneyi çağırır; aynı kodu sıfırdan yazmak zorunda kalmaz. Kütüphaneler, yazılım geliştirme hızını artırmanın yanı sıra test edilmiş ve eniyilenmiş kodun tekrar tekrar kullanılmasını sağlar.

Bu katmanlı yapı, bilgisayar mimarisi açısından can alıcı bir sonuç doğurur: bir programın başarımı yalnızca donanıma değil, derleyicinin eniyileme kalitesine, işletim sisteminin kaynak yönetim politikalarına ve kütüphanelerin verimliliğine de bağlıdır. Mimari tasarımcılar, buyruk kümesini ve donanımı bu yazılım katmanlarıyla uyumlu biçimde geliştirmek zorundadır. Bu bölümde tanıttığımız komut-derleme-çalıştırma zincirini RISC-V buyruklarıyla ayrıntılı biçimde Bölüm 3’te, çevirici dil programlamayı ise Bölüm 4’te ele alacağız.

1.5.6 Buyruk Yürütüm Döngüsü

Derleyici ve çevirici yoluyla makine diline çevrilen buyrukların yürütüm döngüsü Şekil 1.12’de veriliyor. İlk olarak programın saklandığı yerden sıradaki buyruk çekilir ve kullanılan buyruk kümesinin tanımlamalarına göre yapılacak işlem ve buyruğun büyüklüğü belirlenir. İşlenecek veriler bulundukları konumlardan okunur, verilerin üzerinde işlem yapılır, sonuçlar ya da durumlar daha sonra kullanılmak üzere saklanır ve bir sonraki buyruk bellekten çekilmek üzere, işlenen buyruğun bellekteki konumunun adresini tutan Program Sayacı yazmacının değeri güncellenir. Programın son buyruğuna kadar bu döngü sürer.



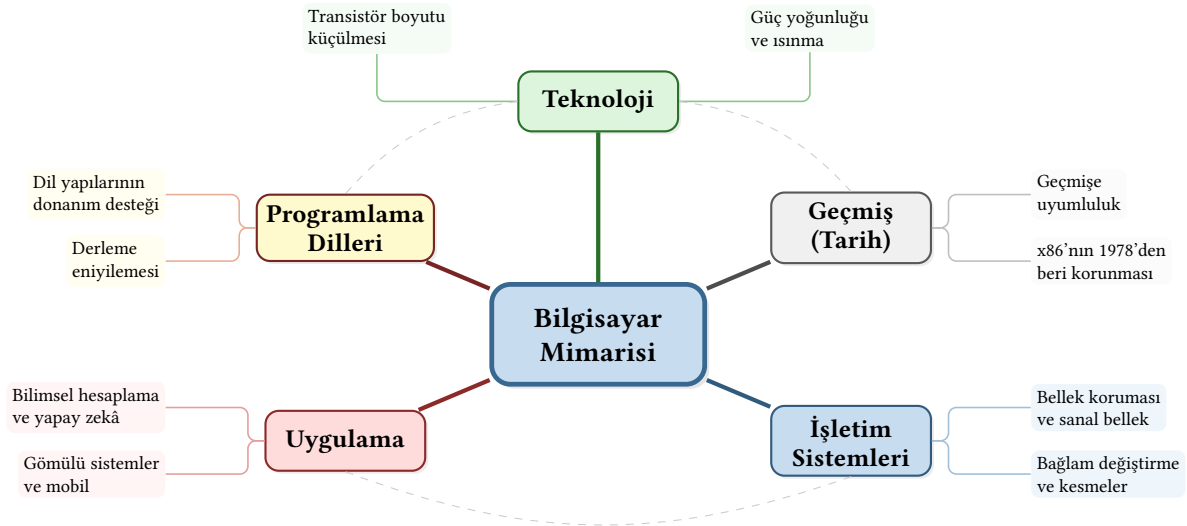
Şekil 1.12: Buyruk yürütüm döngüsü

Bilgi Notu 1.6: Getir-Çöz-Yürüt ve Bulaşık Makinesi Benzetmesi

Bir bulaşık makinesini düşünün: kirli tabaklar konulur (getir), makine programını seçer (çöz) ve yıkama başlar (yürüt). Her seferinde tek bir tabak yıkamak yerine, birden çok tabağı sırayla makineye vermek verimliliği artırır; tıpkı boru hattının (*pipeline*) birden çok buyruğu çakıştırarak (*overlap*) işlemesi gibi. Boru hattı kavramı Bölüm 7’de ayrıntılı olarak ele alınacaktır.

1.6 Bilgisayar Mimarisini Şekillendiren Etkenler

Bilgisayar, birbirini tamamlayan pek çok ayrı parçadan oluşur. Bu parçaların birlikte verimli ve hatasız çalışması için birimlerin organizasyonu oldukça önemli. Bir bilgisayarın mimarisini şekillendiren beş temel etken Şekil 1.13’de gösteriliyor.



Şekil 1.13: Bilgisayar Mimarisini Şekillendiren Etkenler

- **Teknoloji:** Yeni üretim süreçleri ve malzeme bilimi ilerlemeleri, transistör boyutunu

küçültüp işlem gücünü artırırken aynı zamanda güç tüketimi ve ısınma gibi yeni tasarım kısıtları getiriyor. Teknolojideki her sıçrama mimarinin yeniden düşünülmesini gerektiriyor.

- **Programlama dilleri:** Bilgisayarın programlama dillerini desteklemesi için işlemcinin, dilin gereksinim duyduğu işlemleri verimli biçimde gerçekleştiren buyrukları içerecek şekilde tasarlanması gerekiyor.
- **Uygulama:** Farklı kullanım alanları (bilimsel hesaplama, yapay zekâ, gömülü sistemler, taşınabilir uygulamalar) farklı mimari gereksinimlere yol açar. Alana özgü hızlandırıcıların ortaya çıkması bu etkinin somut bir sonucu.
- **İşletim sistemleri:** Donanımın, işletim sisteminin gereksinim duyduğu bellek koruması, bağlam değiştirme ve kesme mekanizmalarını desteklemesi gerekiyor. İşletim sistemi ile donanım birlikte evrilir.
- **Geçmiş (Tarih):** *Geçmiş uyumluluk (backward compatibility)* ilkesi, daha önce yazılmış yazılımların yeni donanımda da çalışmasını zorunlu kılar. Bu ilke mimariyi tarihe bağlar; x86 buyruk kümesinin 1978’den bu yana korunması bunun en bilinen örneği.

Bu etkenler arasında *geçmiş uyumluluk* özellikle güçlü bir kısıt oluşturur. Intel, 8086’dan bu yana her yeni işlemcide önceki neslin tüm yazılımlarını çalıştırma taahhüdünü sürdürmüştür. Devasa bir yazılım ekosistemine sahip olmanın getirdiği bu rekabet avantajı, şirket içinde *altın kelepçe (golden handcuffs)* olarak adlandırılır: milyarlarca dolarlık mevcut yazılım yatırımı müşterileri x86 ekosistemine bağlarken, aynı zamanda mimarların eski buyrukları ve davranışları sonsuza dek desteklemesini zorunlu kılar. Bu durum, teknik açıdan daha verimli tasarım seçeneklerini engellese bile ticari açıdan vazgeçilmez görülmektedir. Geçmiş uyumluluğun buyruk kümesi tasarımına etkileri Bölüm 3’te ayrıntılı olarak ele alınacaktır.

1.6.1 Dört Tasarım İlkesi

Bilgisayar mimarisi alanında onlarca yıllık deneyimden süzölmüş dört temel tasarım ilkesi vardır. Bu ilkeleri kitap boyunca tekrar tekrar göreceğiz; burada kısaca tanıyalım.

1. *Küçük olan hızlıdır.* Bir birim ne kadar küçükse içindeki teller o kadar kısa, sinyal gecikmesi o kadar düşük olur. Yazmaç öbeği, önbellek, çoklayıcı: hepsi gereksinimi karşılayacak en küçük boyutta tutulmalıdır.
2. *Olağan durumu hızlandır.* Nadir gerçekleşen durumlar için karmaşık donanım eklemek yerine, en sık karşılaşılan durumu hızlı kılmak toplam başarıma çok daha fazla katkı sağlar. Bu ilke, Bölüm 2’de ele alacağımız Amdahl Yasası ile doğrudan bağlantılıdır.
3. *İyi tasarım ödünleşme gerektirir.* Her tasarım kararının bir getirisi, bir de bedeli vardır. Daha geniş bir veri yolu aktarım hızını artırır ama daha fazla silikon alanı tüketir; daha büyük bir önbellek bulma oranını yükseltir ama erişim süresini uzatır. Burada “en iyi” diye mutlak bir seçenek yoktur; önemli olan gereksinime göre getiri–bedel çözümlemesi yaparak artıda kalmaktır.
4. *Yalınlık düzenden gelir.* Aynı boyutta buyruklar, tutarlı alan düzenleri, az sayıda kural dışı durum: düzenli bir yapı donanımı yalınlaştırır. Yalın donanım daha küçük, daha hızlı ve daha az hata eğilimlidir.

Bilgi Notu 1.7: Tasarım İlkeleri Yalnızca Donanım İçin mi?

Bu dört ilke yalnızca silikon üzerinde değil, aslında herhangi bir sistem tasarımında geçerlidir. Bir takım küçükse daha çevik hareket eder (küçük olan hızlıdır). Her gün yaptığınız bir işe yatırım yapmak, yılda bir yaptığınız işe yatırım yapmaktan çok daha verimlidir (olağan durumu hızlandır). Aldığınız her kararın bir artısı, bir de eksisi vardır; hiçbir seçenek mutlak iyi değildir (ödünleşme). İstisnası az olan düzenli bir sistem, yönetmesi ve uygulaması kolay bir sistemdir (yalnızlık düzenden gelir).

Bunların hepsinden önce ise sıfırıncı adım gelir: *gereksinim belirleme*. Bir işi neden yaptığınızı ortaya koymadan, ne kadar başarılı olduğunuzu ölçemezsiniz. “En iyi uçak hangisidir?” sorusunu düşünelim: F-16 mı, Boeing 747 mi? Yanıt, o uçakla ne yapmak istediğinize bağlıdır. Bomba atıp hızla kaçacaksınız F-16, yüzlerce yolcuyu ucuza uzak mesafeye taşıyacaksanız Boeing 747. Mühendislikte de hayatta da her şey gereksinimle başlar.^a

^aBu konunun daha geniş bir tartışması için “Mühendisliği Hayata Uygulamak” başlıklı canlı yayını izleyebilirsiniz: <https://www.youtube.com/live/S85FnepzrGo>

1.7 Güç Tüketimi Eğilimleri

Sayısal devrelerde güç tüketimi giderek daha fazla önem kazanan bir tasarım ölçütü. Taşınabilir bilgisayarların kullanımının artmasıyla önem kazanan pil ömrü, sayısal devrelerin daha az güç tüketecek biçimde tasarlanmasını gerektiriyor.

Bir işlemcinin güç tüketimini ifade etmek için yaygın olarak kullanılan ölçüt **TDP** (*Thermal Design Power*: Soğutma Tasarım Gücü) değeridir. TDP, işlemcinin tipik iş yükü altında ürettiği en yüksek ısı miktarını watt cinsinden belirtir ve soğutma sisteminin bu ısıyı uzaklaştırabilecek kapasitede tasarlanması gerektiğini gösterir. TDP değeri işlemcinin tüketebileceği mutlak en yüksek güç değil, soğutma tasarımı için referans alınan üst sınırdır. Tablo 1.5 farklı soğutma çözümlerinin kapasitesini ve yaklaşık maliyetini karşılaştırmaktadır.

Tablo 1.5: İşlemci soğutma çözümlerinin yaklaşık ısı uzaklaştırma kapasitesi, maliyeti ve tipik kullanım alanları. Değerler tipik ürünlere göre verilmiştir; gerçek başarımlar kasaya, hava akışına ve ortam sıcaklığına bağlı olarak değişir.

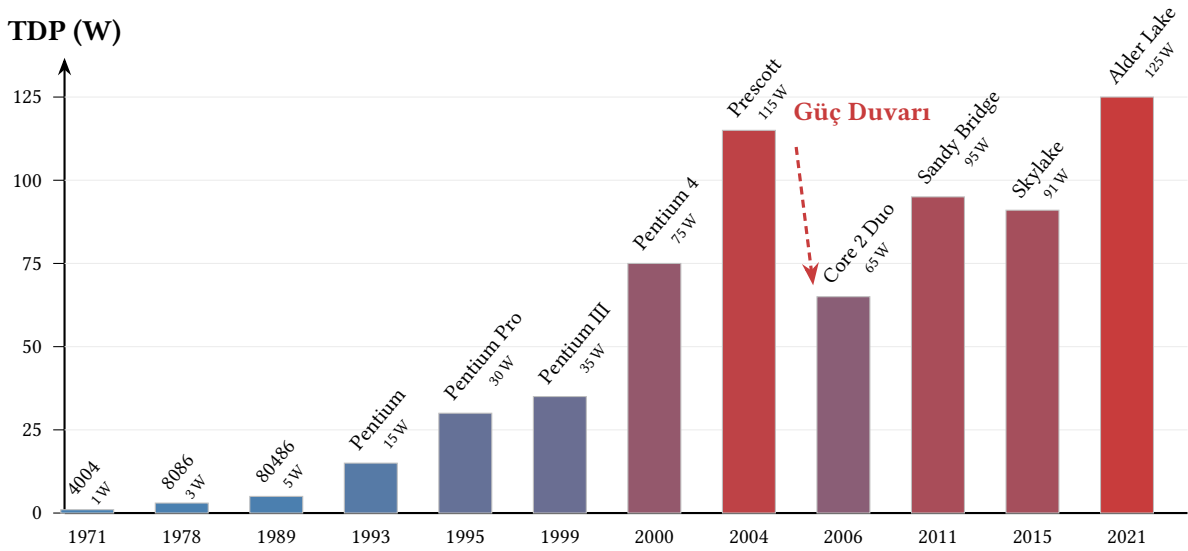
Soğutma Türü	Kapasite (W)	Maliyet (\$)	Açıklama	Tipik Kullanım
Stok soğutucu	45–65	0 (dahil)	Alüminyum blok + pervane	Ofis, web, hafif kullanım
Kule tipi hava (giriş)	120–150	25–40	Tek kule, 4 ısı borusu, pervane	Oyun, yazılım geliştirme
Kule tipi hava (üst)	200–250	80–110	Çift kule, 6+ ısı borusu, 2 pervane	Üst seviye oyun, içerik üretimi
HBA sıvı (240 mm)	200–250	70–120	Kapalı döngü, 240 mm radyatör	Oyun, iş istasyonu
HBA sıvı (360 mm)	250–300	100–180	Kapalı döngü, 360 mm radyatör	Hız aşırma, çok çekirdekli
Özel döngü sıvı	300+	300–600+	Açık döngü, kullanıcı kurulumu	Aşırı hız aşırma, çoklu GPU

Tabloda geçen **HBA** (Hepsi Bir Arada) terimi, pompası, radyatörü ve sıvı bağlantıları fab-

rikada birleştirilmiş kapalı döngü sıvı soğutma sistemlerini ifade eder. Kullanıcının sıvı doldurmasına veya bağlantı yapmasına gerek kalmaz; kutudan çıktığı gibi takılır. Hava soğutmalı çözümlerde ise ısı, metal kanatçıklar (*heatsink*) aracılığıyla yayılır ve bir veya birden fazla *per-vane* (*fan*) ile üflenmiş hava akışıyla uzaklaştırılır.

TDP'si 65 W olan bir işlemci stok soğutucuyla çalışabilirken, 125 W ve üzeri TDP'ye sahip işlemciler kule tipi veya sıvı soğutma gerektirir. Günümüzde üst seviye masaüstü işlemcilerin TDP değerleri 200 W'ı aşabildiğinden, soğutma sistemi tasarımı ve maliyeti önemli bir mühendislik kısıtı hâline gelmiştir.

Şekil 1.14 Intel işlemcilerinin yıllara göre TDP değerlerini göstermektedir.



Şekil 1.14: Intel işlemcilerin yıllara göre TDP değerleri. Çubuk rengi TDP değeriyle orantılıdır: mavi serin (düşük güç), kırmızı sıcak (yüksek güç). 2004'teki Prescott'un 115 W'a ulaşması güç duvarını somutlaştırmış, Core 2 Duo ile verimlilik odaklı tasarıma geçilmiştir.

Şekilden görüldüğü üzere, işlemcilerin güç tüketimi 1971'deki 4004'ün 1 W altındaki değerinden 2004'te Pentium 4 Prescott ile 115 W'a yükselmiştir. Intel'in 2006'da Core 2 Duo ile mimari değişikliğe gitmesi, saat sıklığı yerine verimliliğe odaklanıldığının açık göstergesidir.

1.7.1 Devingen Güç Tüketimi

Devingen güç tüketimi, mantık devrelerini oluşturan transistörlerin açılıp kapanarak işlem gerçekleştirdiği sırada ortaya çıkan güç tüketimidir:

$$P_{\text{devingen}} = C \times V_{dd}^2 \times f \quad (1.4)$$

Burada C sürülen yükü, V_{dd} kaynak gerilimini, f ise açılıp kapanma sıklığını gösterir. Denklemden görüldüğü üzere güç tüketimi kaynak gerilimiyle karesel orantılı olduğundan, V_{dd} 'nin düşürülmesi güç tüketimini azaltmanın en etkili yoludur.

Bilgi Notu 1.8: Kaynak Geriliminin Tarihsel Evrimi

İlk mikroişlemcilerde (1970'ler–1980'ler) kaynak gerilimi 5,0 V idi. Üretim teknolojileri geliştikçe transistör boyutları küçüldü ve kaynak gerilimi de buna koşut olarak düşürüldü:

- 1,0 μm teknoloji (1980'ler): $V_{dd} = 5,0 \text{ V}$
- 0,35 μm teknoloji (1990'lar): $V_{dd} = 3,3 \text{ V}$
- 0,18 μm teknoloji (2000'ler): $V_{dd} = 1,8 \text{ V}$
- 65 nm teknoloji (2006): $V_{dd} = 1,2 \text{ V}$
- 22 nm teknoloji (2012): $V_{dd} = 0,8 \text{ V}$
- 5 nm teknoloji (2020'ler): $V_{dd} \approx 0,7 \text{ V}$

Kaynak gerilimi 5,0 V'tan 0,7 V'a düştüğünde, yalnızca bu değişikliğin etkisiyle devingen güç tüketimi yaklaşık 50 kat azalır ($5^2/0,7^2 \approx 51$). Ancak düşen gerilimle birlikte transistör eşik gerilimine yaklaşıldığından sızıntı akımları artar ve devrelerin gürültüye karşı dayanıklılığı azalır.

Güç (P) birim zamanda harcanan enerji miktarını, yani anlık tüketimi, ifade ederken, *enerji* (E) belirli bir süre boyunca harcanan toplam miktarı gösterir. Bir işlemcinin güç tüketimi yüksek olsa bile görevi kısa sürede bitirirse toplam enerji tüketimi düşük kalabilir; tam tersine, düşük güçlü bir işlemci uzun süre çalışarak daha fazla enerji harcayabilir. Taşınabilir aygıtlarda pil ömrünü belirleyen güç değil enerjidir; benzer biçimde veri merkezlerinin elektrik faturasını da toplam enerji tüketimi belirler.

Enerji tüketimi şu denklemle ifade edilir:

$$E = P \times t = C \times V_{dd}^2 \times f \times t \quad (1.5)$$

Burada t işlemin süresini gösterir. Enerji tasarrufu hem güç tüketimini düşürerek hem de işlemi daha kısa sürede tamamlayarak sağlanabilir. Bu nedenle “yavaş ama düşük güçlü” bir işlemci her zaman “hızlı ama yüksek güçlü” bir işlemciden daha verimli değildir; enerji açısından her iki boyutun da birlikte değerlendirilmesi gerekir.

Örnek 1.5

Bir işlemcinin kapasitif yükü $C = 20 \text{ nF}$, besleme gerilimi $V = 1,0 \text{ V}$ ve saat sıklığı $f = 3 \text{ GHz}$ ise:

- Devingen güç tüketimini hesaplayın.
- Gerilim %10 azaltılırsa ($V = 0,9 \text{ V}$) güç tüketimi ne kadar düşer?
- Hem gerilim %10 hem de saat sıklığı %10 azaltılırsa güç tasarrufu ne olur?

Çözüm:

(a) Devingen güç: $P = C \times V^2 \times f = 20 \times 10^{-9} \times 1,0^2 \times 3 \times 10^9 = 60 \text{ W}$.

(b) $V = 0,9 \text{ V}$ için: $P = 20 \times 10^{-9} \times 0,81 \times 3 \times 10^9 = 48,6 \text{ W}$. Güç %19 azalmıştır. Gerilim yalnızca %10 düşürülmesine karşın güç %19 azalmıştır; bunun nedeni güç tüketiminin gerilimin *karesiyle* orantılı olmasıdır.

(c) $V = 0,9 \text{ V}$, $f = 2,7 \text{ GHz}$: $P = 20 \times 10^{-9} \times 0,81 \times 2,7 \times 10^9 = 43,74 \text{ W}$. Toplam güç tasarrufu: $(60 - 43,74)/60 = \%27,1$.

Bu örnek, gerilim düşürmenin (*voltage scaling*) neden en etkili güç azaltma yöntemi olduğunu açıkça göstermektedir. Dennard ölçeklemesi sona erince transistör boyutu küçülse bile gerilim artık orantılı biçimde düşürülemez hale geldi; bu durum güç duvarını ortaya çıkardı.

1.7.2 Durağan Güç Tüketimi

Mantık devrelerini oluşturan transistörlerin hiçbir anlamlı iş yapmadıkları halde, kaynak geriliminden toprağa akım sızdırmaları nedeniyle oluşan güç tüketimine *durağan güç tüketimi* denir:

$$P_{\text{durağan}} = I_{\text{sızıntı}} \times V_{dd} \quad (1.6)$$

Transistör boyutları küçüldükçe sızıntı akımı artar. Yükselen transistör sayıları ve düşen transistör boyutları durağan güç tüketimini giderek artırıyor. Tablo 1.6 durağan güç tüketiminin toplam güç içindeki payının teknoloji düğümleriyle nasıl değiştiğini göstermektedir.

Tablo 1.6: Durağan (sızıntı) güç tüketiminin toplam işlemci güç tüketimi içindeki yaklaşık payı.

Teknoloji Düzümü	Sızıntı Payı (%)	Dönem
250 nm	2–5	1990’ların sonu
130 nm	~10	2001
90 nm	20–25	2004
65 nm	30–35	2006
45 nm	35–40	2008
32 nm	40–50	2010
22 nm	35–40	2012

22 nm’deki düşüş FinFET teknolojisinin etkisini yansıtır.

Tablodan görüldüğü üzere, 250 nm’den 32 nm’ye geçişte sızıntı akımının toplam güç içindeki payı yaklaşık 10 kat artmıştır. Ancak Intel’in 22 nm düğümünde **FinFET** (üç boyutlu kapılı, *Fin Field-Effect Transistor*) yapısına geçmesiyle sızıntı akımı kısmen kontrol altına alınmıştır. Günümüzdeki ileri düğümlerde (5 nm ve altı) **GAA** (*Gate-All-Around*) yapılar sızıntıyı daha da azaltmayı hedeflemektedir.

1.7.3 Güç Tüketimini Azaltma Yöntemleri

Güç tüketiminin birimi Watt’tır. Saat sıklığı giderek yükselen işlemcilerdeki güç tüketimini azaltmanın en kolay yolu kaynak gerilimini düşürmek. Ancak kaynak geriliminin düşmesi devre gecikmelerini ve saklanan elektrik sinyallerinin gürültüden etkilenme olasılığını artırır.

Güç tüketimini azaltmak için kullanılan bir diğer yöntem ise kullanılmayan birimlerin kaynak gerilimini kesmek ya da saat sıklığını düşürmek. Güç tüketimi işlemci tasarımcıları tarafından tasarımın en önemli kısıtlarından biri olarak görüldüğü için aşılması gereken bu engele *Güç Duvarı* adı verildi. Tek bir çekirdek çok yüksek oranda işlem yapılmasını kısıtlayan bu güç duvarı, tasarımcıları aynı anda daha fazla işlem yapılmasını sağlayan çok çekirdekli tasarımlara yönlendirdi. Güç tüketimi kısıtının boru hattı derinliğini ve işlemci mikromimarisini nasıl biçimlendirdiğini Bölüm 6’da ve Bölüm 7’de ayrıntılı olarak göreceğiz.

Bilgi Notu 1.9: GHz Yarışından Verimlilik Yarışına

2000’li yılların başında işlemci üreticileri saat sıklığını artırarak başarımlarını kazanımı sağlama yarışına girmişti. Ancak Pentium 4 serisiyle saat sıklığı 3,8 GHz’e ulaştığında güç tüketimi ve ısınma sorunları bu yaklaşımın sürdürülemez olduğunu gösterdi. Günümüzde başarımlar ölçütü olarak “performans/watt” birimi öne çıkıyor. ARM tabanlı işlemcilerin taşınabilir aygıtlarda ve hatta sunucularda (ör. AWS Graviton) yaygınlaşması, güç verimli tasarımın önemini somut bir göstergesi. RISC-V’in modüler yapısı da tasarımcılara güç bütçesine göre özelleştirilmiş çekirdekler oluşturma esnekliği sunuyor.

Örnek 1.6

Dennard ölçeklemesi döneminde transistör boyutu küçüldükçe kaynak gerilimi ve saat sıklığı da buna uyumlu biçimde değişiyordu. Aşağıdaki iki kuşak işlemciyi karşılaştırarak güç duvarının kökenini nicel olarak gösterin.

	Kuşak 1 (130 nm)	Kuşak 2 (65 nm)
Transistör boyutu (s)	1 (referans)	0,5×
Kaynak gerilimi (V_{dd})	1,2 V	?
Saat sıklığı (f)	2 GHz	?
Kapasitif yük (C)	20 nF	?

- (a) Dennard ölçeklemesi geçerli olsaydı Kuşak 2'nin V_{dd} , f ve C değerleri ne olurdu? Devingen güç tüketimi nasıl değişirdi?
- (b) Uygulamada Kuşak 2 işlemcisinin gerilimi yalnızca 1,0 V'a düşürülebildi (0,83×), saat sıklığı ise 3 GHz'e (1,5×) çıkarıldı. Kapasitif yük 0,5× oranında küçüldü. Gerçek güç tüketimini hesaplayın ve ideal durumla karşılaştırın.

Çözüm:

(a) Dennard ölçeklemesinde boyut s kat küçüldüğünde: gerilim s kat düşer, saat sıklığı $1/s$ kat artar, kapasitif yük s kat azalır. $s = 0,5$ için:

$$V_{dd} = 1,2 \times 0,5 = 0,6 \text{ V}, \quad f = 2 \times 2 = 4 \text{ GHz}, \quad C = 20 \times 0,5 = 10 \text{ nF}.$$

$$\text{İdeal güç: } P = C \times V_{dd}^2 \times f = 10 \times 10^{-9} \times 0,6^2 \times 4 \times 10^9 = 14,4 \text{ W}.$$

$$\text{Kuşak 1'in gücü: } P_1 = 20 \times 10^{-9} \times 1,2^2 \times 2 \times 10^9 = 57,6 \text{ W}.$$

İdeal ölçeklemede güç $14,4/57,6 = 0,25 = s^2$ oranında düşer. Transistör sayısı 4 kat artmasına karşın güç aynı kalır; Dennard ölçeklemesinin büyüğü budur.

$$(b) \text{ Gerçek değerlerle: } P_2 = 10 \times 10^{-9} \times 1,0^2 \times 3 \times 10^9 = 30 \text{ W}.$$

	İdeal (Dennard)	Gerçek	Fark
Gerilim	0,6 V	1,0 V	1,67× yüksek
Saat sıklığı	4 GHz	3 GHz	0,75× düşük
Güç tüketimi	14,4 W	30 W	2,08× yüksek

Gerilimin ideal değer 1,67 katında kalması, güç tüketimini $1,67^2 \approx 2,8$ kat artırır. Saat sıklığının düşük kalması bunu kısmen telafi etse de sonuçta güç tüketimi idealin 2 katını aşar. Her yeni kuşakta bu fark birikir ve sonunda güç duvarına ulaşılır. 65 nm sonrasında Dennard ölçeklemesinin çöküşü, endüstriyi tek çekirdekli hız artışından çok çekirdekli tasarımlara yönelmeye zorlamıştır.

1.8 Güvenilirlik

Bir bilgisayar sisteminden beklenen yalnızca hızlı çalışması değil, *doğru* ve *kesintisiz* çalışmasıdır. Bir süper bilgisayar saniyede katrilyon işlem yapabilir ama hesaplama sonuçları yanlışsa ya da sistem saatte bir çökerse hiçbir kullanıcıya faydası yoktur. Güç tüketimi gibi güvenilirlik de günümüzde başarımlar kadar önemli bir tasarım ölçütüdür; özellikle sunucu, veri merkezi,

uzay ve otomotiv uygulamalarında.

1.8.1 Arıza, Hata ve Bozukluk

Güvenilirlik çözümlemesinde üç kavram dikkatli biçimde birbirinden ayrılmalıdır:

1. **Bozukluk** (*fault*): Sistemin bir bileşenindeki fiziksel veya mantıksal kusur. Örneğin bir bellek hücresine kozmik ışının çarpması ya da bir lehim noktasının çatlaması bir bozukluktur.
2. **Hata** (*error*): Bozukluğun sisteme yansması. Kozmik ışının çarptığı bellek hücresindeki bitin 0'dan 1'e dönmesi bir hatadır.
3. **Arıza** (*failure*): Hatanın kullanıcı tarafından gözlemlenebilir bir yanlış sonuca veya hizmet kesintisine dönüşmesi. Bellekteki hatalı bit bir program çökmesine neden olursa bu bir arızadır.

Bu zincir şöyle özetlenebilir:

$$\text{Bozukluk} \longrightarrow \text{Hata} \longrightarrow \text{Arıza}$$

Her bozukluk hataya, her hata arızaya dönüşmek zorunda değildir. *Hata düzeltme kodlaması* (ECC: Error Correcting Code) hatayı arızaya dönüşmeden yakalayabilir; yedekli hesaplama ise bir birimin bozukluğunu diğerleriyle maskeleyebilir. Güvenilirlik mühendisliğinin amacı bu zinciri mümkün olan en erken noktada kırmaktır.

ECC'nin temel çalışma ilkesi şudur: veri saklanırken veya iletilirken sözcüğe fazladan *denetim bitleri* eklenir. Bu bitler, belirli matematiksel kodlama kurallarına göre üretilir ve okuma sırasında denetlenir. Tek bitlik bir hata oluşmuşsa ECC devresi hatayı hem *saptayıp* hem de *düzeltebilir*; çift bitlik hatalar ise saptanabilir ama düzeltilemez (SEDED: *Single Error Correction, Double Error Detection*). Bu mekanizma özellikle sunucu belleklerinde yaygın olarak kullanılır ve kozmik ışın kaynaklı geçici bozuklukların büyük çoğunluğunu kullanıcı farkına bile varmadan giderir. Hata düzeltme kodlamasının ayrıntılı matematiksel temeli ve farklı kodlama yöntemleri Bölüm 8'de ele alınacaktır.

1.8.2 Bozukluk Türleri

Bozukluklar süresine göre üç gruba ayrılır:

- **Geçici bozukluk** (*transient fault*): Anlık bir olay sonucu oluşur ve kendiliğinden kaybolur. Kozmik ışının neden olduğu bit hataları, gerilim dalgalanmaları ve elektromanyetik girişim başlıca örnekleridir. Uzay uygulamalarında radyasyon kaynaklı geçici bozukluklar özellikle sık görülür.
- **Aralıklı bozukluk** (*intermittent fault*): Belirli koşullar altında tekrar tekrar ortaya çıkan ama koşullar değişince kaybolan bozukluktur. Yüksek sıcaklıkta temas kaybeden bir lehim noktası veya belirli bir veri örüntüsünde hatalı davranan bir bellek hücresi bu türe örnektir. Tanınması en zor bozukluk türüdür.
- **Kalıcı bozukluk** (*permanent fault*): Bir bileşenin geri dönüşü olmayan biçimde hasar görmesi. Yanmış bir transistör, aşınmış bir disk yüzeyi veya kopmuş bir bağlantı kalıcı bozukluklara örnektir.

1.8.3 Güvenilirlik Ölçütleri

Bir sistemin güvenilirliğini nicel olarak ifade etmek için üç temel ölçüt kullanılır:

- **MTTF** (*Mean Time To Failure*: Ortalama Arızaya Kadar Geçen Süre): Onarılamayan bir bileşenin çalışmaya başlamasından ilk arızaya kadar geçen ortalama süredir.
- **MTTR** (*Mean Time To Repair*: Ortalama Onarım Süresi): Arıza gerçekleşikten sonra sistemin onarılıp yeniden çalışır duruma getirilmesi için geçen ortalama süredir.
- **MTBF** (*Mean Time Between Failures*: Ortalama Arızalar Arası Süre): Onarılabilir bir sistemde iki ardışık arıza arasında geçen ortalama süredir:

$$MTBF = MTTF + MTTR \quad (1.7)$$

Bu ölçütlerden türetilen iki önemli kavram daha vardır. Birincisi **kullanılabilirlik** (*availability*), sistemin çalışır durumda olduğu zamanın toplam zamana oranıdır:

$$\text{Kullanılabilirlik} = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF} \quad (1.8)$$

Örneğin MTTF'si 999 saat ve MTTR'si 1 saat olan bir sistemin kullanılabilirliği $999/1000 = \%99,9$ olur. Veri merkezlerinde kullanılabilirlik hedefi genellikle “beş dokuz” (%99,999) olarak ifade edilir; bu, yılda en fazla yaklaşık 5 dakika kesinti anlamına gelir.

İkincisi **FIT** (*Failures In Time*), bir milyar aygıt-saatinde beklenen arıza sayısıdır:

$$FIT = \frac{10^9}{MTTF \text{ (saat cinsinden)}} \quad (1.9)$$

FIT birimi özellikle yarıiletken endüstrisinde yaygın olarak kullanılır. Bir bellek yongasının FIT değeri 100 ise, bu yonga bir milyar saat çalışmada ortalama 100 kez arıza verir; eşdeğer olarak, 10 milyon saat çalışmada 1 kez arıza beklenir.

Birden fazla bileşenden oluşan bir sistemde, herhangi bir bileşenin arızası tüm sistemi durduruyorsa (seri bağlantı), sistemin MTTF'si her bir bileşenin arıza oranlarının toplamının ter-sidir:

$$MTTF_{\text{sistem}} = \frac{1}{\sum_{i=1}^N \frac{1}{MTTF_i}} = \frac{1}{\sum_{i=1}^N \lambda_i} \quad (1.10)$$

Burada $\lambda_i = 1/MTTF_i$ bileşen i 'nin arıza oranıdır. N özdeş bileşenden oluşan bir sistemde bu ifade $MTTF_{\text{sistem}} = MTTF_{\text{bileşen}}/N$ biçimine sadeleşir. Bu denklem can alıcı bir sonuç doğurur: bir veri merkezinde 10 000 sunucu varsa ve her birinin MTTF'si 100 000 saat ise, merkezin MTTF'si yalnızca 10 saat olur; ortalama her 10 saatte bir sunucu arızası beklenir.

Örnek 1.7

Bir sunucu üç temel bileşenden oluşuyor: işlemci, bellek ve disk. Her bileşenin MTTF değeri şöyledir:

Bileşen	MTTF (saat)	FIT
İşlemci	500 000	2 000
Bellek	200 000	5 000
Disk	100 000	10 000

Herhangi bir bileşenin arızası sunucuyu durduruyor (seri bağlantı). Ortalama onarım süresi (MTTR) 2 saattir.

- (a) Sunucunun MTTF'sini hesaplayın.
 (b) FIT değerini bulun.
 (c) Kullanılabilirliği (%) hesaplayın.
 (d) Aynı sunucudan 500 tane bulunan bir veri merkezinin MTTF'sini bulun.

Çözüm:

- (a) Seri bağlantıda toplam arıza oranı:

$$\lambda_{\text{toplam}} = \frac{1}{500\,000} + \frac{1}{200\,000} + \frac{1}{100\,000} = 2 + 5 + 10 = 17 \times 10^{-6} \text{ arıza/saat}$$

$$\text{MTTF}_{\text{sunucu}} = \frac{1}{\lambda_{\text{toplam}}} = \frac{1}{17 \times 10^{-6}} \approx 58\,824 \text{ saat} \approx 6,7 \text{ yıl}$$

Dikkat: en zayıf halka olan disk (100 000 saat) sunucunun MTTF'sini büyük ölçüde belirliyor.

- (b) FIT değeri:

$$\text{FIT} = \frac{10^9}{\text{MTTF}} = \frac{10^9}{58\,824} \approx 17\,000$$

Bu değer, bileşenlerin FIT toplamına ($2\,000 + 5\,000 + 10\,000 = 17\,000$) eşittir; seri bağlantıda FIT değerleri doğrudan toplanır.

- (c) Kullanılabilirlik:

$$\text{MTBF} = \text{MTTF} + \text{MTTR} = 58\,824 + 2 = 58\,826 \text{ saat}$$

$$\text{Kullanılabilirlik} = \frac{\text{MTTF}}{\text{MTBF}} = \frac{58\,824}{58\,826} \approx \%99,9966$$

Bu “dört dokuz”un üzerinde ama “beş dokuz” (%99,999) hedefinin altında bir değerdir.

- (d) Veri merkezinin MTTF'si:

$$\text{MTTF}_{\text{merkez}} = \frac{\text{MTTF}_{\text{sunucu}}}{500} = \frac{58\,824}{500} \approx 117,6 \text{ saat} \approx 4,9 \text{ gün}$$

Veri merkezinde ortalama her 5 günde bir sunucu arızası beklenir. Bu nedenle büyük veri merkezleri yedekleme ve otomatik yük aktarma mekanizmaları olmadan çalışamaz.

1.8.4 Güvenilirlik, Dayanıklılık ve Gürbüzlük

Günlük dilde birbiriyle karıştırılan üç kavramı kesin biçimde ayırmak gerekir:

- **Güvenilirlik** (*reliability*): Sistemin belirli bir süre boyunca doğru çalışma olasılığı. MTTF ve MTBF ile ölçülür. “Bu sunucu iki yıl arızasız çalışır mı?” sorusunun yanıtıdır.
- **Dayanıklılık** (*durability*): Fiziksel aşınma ve çevresel koşullara direnme kapasitesi. Bir SSD'nin yazma dayanıklılığı (ör. 600 TBW: Terabytes Written) veya bir askeri bilgisayara-
nın titreşim dayanımı bu kavramla ifade edilir.
- **Gürbüzlük** (*robustness*): Beklenmedik koşullar altında kabul edilebilir biçimde çalışmaya devam edebilme yeteneği. Gerilim dalgalanmalarında veri kaybetmeyen bir dosya sistemi veya geçersiz girdide çökmek yerine hata iletili veren bir yazılım gürbüzüdür.

Bu üç kavram birbirini tamamlar: güvenilir bir sistem uzun süre doğru çalışır; dayanıklı

bir sistem fiziksel koşullara direnir; gürbüz bir sistem beklenmedik durumlarla başa çıkar. İyi bir tasarım üçünü birden gözetir. Güvenilirliği donanım düzeyinde destekleyen önbellek ECC bitleri ve hata düzeltme mekanizmalarını Bölüm 8’de, giriş/çıkış sistemlerinin güvenilirlik tasarımını ise Bölüm 9’da ele alacağız.

Bilgi Notu 1.10: Mars’ta Güvenilirlik

NASA’nın Mars keşif aracı *Curiosity*, radyasyona dayanıklı (*radiation-hardened*) bir RAD750 işlemcisi kullanır. Bu işlemci yalnızca 200 MHz’de çalışır; dünya üzerindeki herhangi bir akıllı telefondan yüzlerce kat yavaştır. Ancak Mars yüzeyinde kozmik ışınlara ve aşırı sıcaklık değişimlerine (-130°C ile $+70^{\circ}\text{C}$ arası) yıllarca dayanabilir. Uzun mühendisliğinde başarımlı ölçütü “saniye başına buyruk” değil, “milyonlarca kilometre uzakta on yıl boyunca hatasız çalışma”dır.

1.9 Neden RISC-V?

Günümüzde masaüstü ve sunucu pazarına Intel ile AMD’nin **x86** mimarisi, taşınabilir ve gömülü pazara ise ARM Holdings’in **ARM** mimarisi hâkim. Her iki buyruk kümesi mimarisi de tescilli (proprietary) olup kullanımı lisans anlaşmalarına ve ücretlere bağlı. Bu durum yeni işlemci tasarlamak isteyen girişimciler, araştırmacılar ve ülkeler için yüksek bir giriş bariyeri oluşturuyor.

2010 yılında Kaliforniya Üniversitesi Berkeley’de Krste Asanović ve David Patterson öncülüğünde geliştirilen **RISC-V**, bu soruna köklü bir çözüm sunuyor. RISC-V, herhangi bir lisans ücreti gerektirmeyen, tamamen açık kaynaklı bir buyruk kümesi mimarisi. Başlangıçta RISC-V Foundation tarafından yönetilen proje, 2020’de *RISC-V International* çatısı altında küresel bir standart haline geldi.

RISC-V’in öne çıkan özellikleri şöyle sıralanabilir:

- **Açık kaynak ve ücretsiz:** Bir üniversite öğrencisinden çok uluslu bir şirkete kadar herkes RISC-V tabanlı işlemci tasarlayıp üretebilir.
- **Modüler tasarım:** Temel tamsayı buyruk kümesi (RV32I/RV64I) üzerine çarpma (M), atomik (A), kayan nokta (F, D) ve vektör (V) gibi isteğe bağlı uzantılar eklenebilir. Bu modülerlik, gömülü bir denetleyiciden süper bilgisayar çekirdeğine kadar geniş bir yelpazede özelleştirme olanağı tanır.
- **Temiz ve çağdaş:** 1970’lerin geriye uyumluluk yükünden arınmış, günümüz derleyici ve işletim sistemi gereksinimlerine göre tasarlanmış yalın bir mimari.
- **Eğitim dostu:** Karmaşık tarihsel eklentiler içermediği için öğrencilerin bir işlemciyi sıfırdan tasarlayıp benzetmesi oldukça kolaylaşıyor.

	x86	ARM	RISC-V
Lisanslama	Tescilli	Tescilli	Açık Kaynak
Kullanım Alanı	Masaüstü, Sunucu	Mobil, Gömülü	Giderek genişleyen
Genişletilebilirlik	Sabit (eski yük)	Kısmi	Tam modüler
Tasarım Maliyeti	Yüksek	Orta	Ücretsiz

Şekil 1.15: x86, ARM ve RISC-V buyruk kümesi mimarilerinin karşılaştırması

BKM Karşılaştırması: Kullanım Alanları

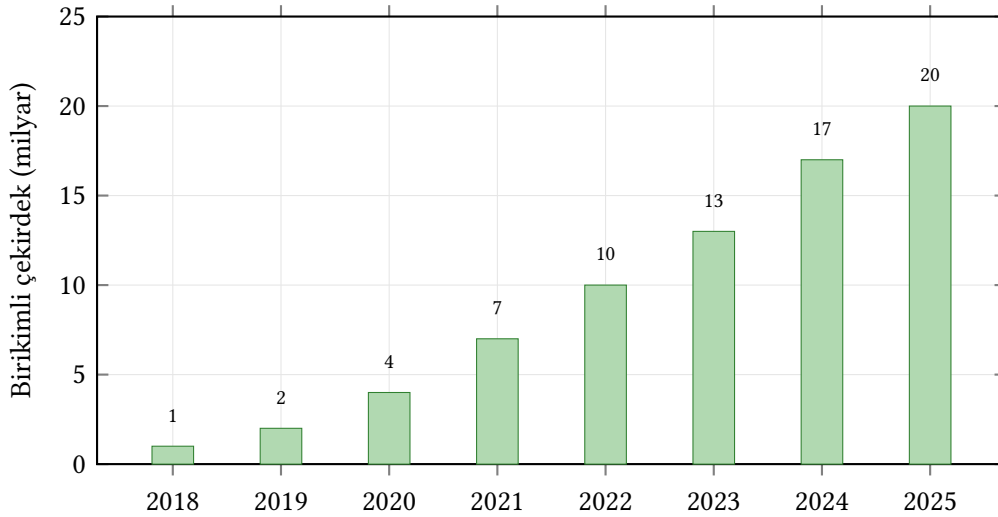
Günümüzde üç büyük buyruk kümesi mimarisi farklı alanlarda baskın konumdadır:

	x86-64	ARM	RISC-V
Baskın alan	Masaüstü, sunucu	Mobil, gömülü	Gömülü, araştırma
Lisans	Kapalı (Intel/AMD)	Ücretli lisans	Açık ve ücretsiz
Buyruk boyutu	Değişken (1–15 B)	Sabit (32 bit)	Sabit (32 bit)
Geriye uyumluluk	1978'den beri	AArch32/AArch64	Yük yok
Pazar payı (2025)	~%85 masaüstü	~%95 mobil	Hızla büyüyor

RISC-V'in en büyük avantajı, tarihsel yükten arınmış olarak sıfırdan tasarlanmış olmasıdır. x86'nın 1978'den bu yana sürdürmek zorunda olduğu geriye uyumluluk, mimariyi karmaşıklaştırmış ve güç tüketimini artırmıştır. ARM, taşınabilir pazarda hegemonya kurmuş olmakla birlikte lisans modeli küçük girişimler için engel oluşturabilmektedir. RISC-V ise açık kaynak modeli sayesinde hem üniversiteler hem de şirketler için erişilebilir bir seçenektir.

Şekil 1.15'te üç önemli buyruk kümesi mimarisinin temel boyutlarda karşılaştırması veriliyor. Günümüzde SiFive, Alibaba T-Head, Andes ve NVIDIA gibi firmalar RISC-V tabanlı ticari işlemciler üretiyor. Avrupa İşlemci Girişimi (European Processor Initiative) de yüksek başarılı hesaplama alanında RISC-V'e yatırım yapıyor. Akademik dünyada ise BOOM, Rocket ve CVA6 gibi açık kaynaklı RISC-V çekirdekleri araştırmacılara gerçek bir işlemci tasarımı üzerinde deney yapma olanağı sunuyor.

RISC-V'in benimsenmesi yalnızca akademik dünyayla sınırlı kalmadı; endüstride de hızla ivme kazandı. Şekil 1.16'te birikimli RISC-V çekirdek sayısının yıllara göre artışı gösterilmektedir. ARM mimarisinin 10 milyar çekirdeğe ulaşması 17 yıl sürmüştü, RISC-V bu eşiğe yalnızca 12 yılda ulaştı. Dikkat çekici bir nokta: NVIDIA tek başına 2024'te 1 milyar RISC-V çekirdeğini piyasaya sürdü; bunlar GPU denetleyicileri, güç yönetimi ve güvenlik alt sistemleri gibi görevlerde kullanılmaktadır.



Kaynak: RISC-V International, Semico Research (2022–2025 verileri; 2025 tahmini)

Şekil 1.16: Piyasaya sürülen birikimli RISC-V çekirdek sayısı (milyar). 2022’de 10 milyar eşiği aşıldı; 2025 itibarıyla tahmini 20 milyar çekirdeğe ulaşıldı.

Bilgi Notu 1.11: Meta ve RISC-V: Veri Merkezine Giden Yol

RISC-V’in yalnızca gömülü sistemler ve mikrodenetleyiciler için bir mimari olduğu algısı, son yıllarda hızla değişmektedir. Meta (Facebook), Eylül 2025’te RISC-V tabanlı sunucu ve yapay zekâ yongaları geliştiren *Rivos* girişimini satın aldı. Rivos, 3,1 GHz’lik 64-bit RISC-V işlemci çekirdekleri ve büyük dil modelleri için eniyilenmiş özel bir GPU tasarlıyordu. Meta’nın bu hamlesi, daha önce Amazon’un Annapurna Labs’ı alarak ARM tabanlı Graviton işlemcilerini geliştirmesine benzetilmektedir; ancak burada tercih edilen mimari ARM değil, RISC-V’tir. Meta’nın yapay zekâ çıkarım hızlandırıcısı MTIA zaten RISC-V tabanlı işlem elemanları kullanmaktaydı; Rivos satın almasıyla şirket, veri merkezlerinde RISC-V’i çok daha geniş ölçekte konumlandırmayı hedeflemektedir. Bu gelişme, açık kaynaklı bir buyruk kümesi mimarisinin en yüksek başarımlı gerektiren iş yüklerinde bile rekabetçi olabileceğini gösteren güçlü bir işarettir.

Bu kitapta RISC-V’in tercih edilmesinin başlıca nedeni, öğrencilerin buyruk kümesinden işlemci tasarımına, boru hattından bellek hiyerarşisine kadar tüm kavramları açık ve erişilebilir bir mimari üzerinde uygulayabilmeleridir. RISC-V buyruk kümesinin ayrıntıları Bölüm 3’te ele alınacaktır.

1.10 Bilgisayar Mimarisinin Yeni Ufku: Yapay Zekâ

Bilgisayar mimarisi alanı, 2010’lardan itibaren yapay zekâ (YZ) uygulamalarının muazzam hesaplama gereksinimleriyle yeni bir dönüşüm sürecine girdi. Uzun yıllar boyunca genel amaçlı işlemciler (MİB) hemen her iş yükü için yeterli görülürken, günümüzde derin öğrenme modelleri gibi hesaplama yoğun uygulamalar alana özgü hızlandırıcılar olmaksızın verimli biçimde çalıştırılmaz hale geldi.

1.10.1 Genel Amaçlıdan Alana Özgü Mimarilere

Geleneksel bilgisayar mimarisinde tek bir işlemci tüm görevleri (metin işleme, bilimsel hesaplama, grafik oluşturma) üstlenirdi. Ancak transistör bütçesinin artması ve güç duvarının ortaya çıkması, tasarımcıları farklı bir soruya yönlendirdi: “Her şeyi biraz iyi yapan bir yonga mı, belirli bir şeyi çok iyi yapan bir yonga mı tasarlamalıyız?”

Bu sorunun yanıtı *alana özgü mimari* (Domain-Specific Architecture: DSA) kavramını doğurdu. Günümüzde yapay zekâ iş yükleri bu dönüşümün en güçlü itici gücü; çünkü derin öğrenme modelleri birkaç temel özellik sergiliyor:

- **Yoğun matris aritmetiği:** Sinir ağlarının her katmanı büyük matris çarpımları ve toplama işlemleri gerektirir.
- **Yüksek koşutluk:** Milyonlarca bağımsız çarpma–toplama işlemi eş zamanlı yürütülebilir.
- **Düşük duyarlılıkta yeterli doğruluk:** Eğitim 16 bitle, çıkarım ise 8 hatta 4 bitle yapılabilir; 64 bitlik kayan nokta gerektiren bilimsel hesaplamalardan çok farklıdır.

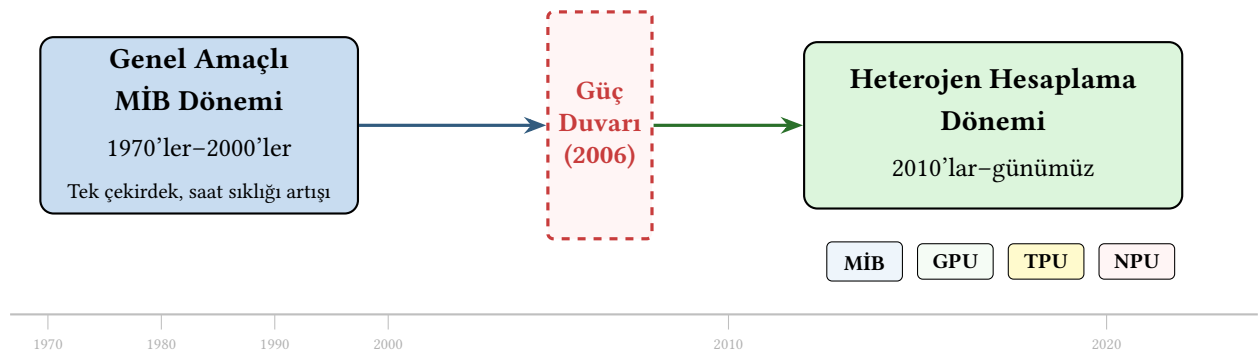
Bu özellikler, genel amaçlı bir MİB yerine bu işlemlere göre özelleştirilmiş bir yonganın çok daha verimli olabileceğini gösteriyor.

1.10.2 Yapay Zekâ Hızlandırıcıları: GPU, TPU, NPU

Yapay zekâ çağının üç temel donanım yapı taşı şöyle özetlenebilir:

- **GPU (Grafik İşlem Birimi):** Aslen üç boyutlu grafik işleme için tasarlanmış olan GPU’lar, binlerce küçük çekirdeğiyle yoğun koşut hesaplamalara olanak tanır. NVIDIA’nın CUDA platformu, GPU’ları yapay zekâ araştırmacıları için birincil hesaplama aracına dönüştürdü. Günümüzün büyük dil modellerinin eğitimi binlerce GPU içeren kümeler üzerinde haftalarca sürebiliyor.
- **TPU (Tensör İşlem Birimi):** Google tarafından geliştirilen TPU’lar, *sistolik dizi* mimarisiyle matris çarpımını doğrudan donanımda gerçekleştirir. Bu yaklaşım, genel amaçlı bir çekirdeğe kıyasla çok daha yüksek enerji verimliliği sağlıyor.
- **NPU (Sinir Ağı İşlem Birimi):** Apple Neural Engine, Qualcomm Hexagon ve Google Tensor yongalarında bulunan NPU’lar, taşınabilir ve uç aygıtlarda düşük güçle yapay zekâ çıkarımı gerçekleştirmek için tasarlandı. Fotoğraf iyileştirmeden gerçek zamanlı çeviriye kadar pek çok özellik bu birimlere dayanıyor.

Bu hızlandırıcıların ayrıntılı mimarisi Bölüm 11’de kapsamlı olarak incelenecek.



Şekil 1.17: Bilgisayar mimarisinde paradigma değişimi: güç duvarı genel amaçlı MİB dönemini sonlandırmış, yerini heterojen hesaplamaya bırakmıştır.

1.10.3 Mimari Eğitime Etkisi

Alana özgü hızlandırıcılar ne kadar farklı görünürse görünsün, hepsi bu kitapta öğreneceğiniz temel kavramlar üzerine inşa edilmiş. Bir GPU'nun çekirdek yapısı, boru hattı ve önbellek ilkeleriyle anlaşılır. Bir TPU'nun sistolik dizisi, sayısal aritmetik temelleri olmadan kavranamaz. Bellek hiyerarşisi ve bant genişliği hesapları her mimari için geçerli.

Bu nedenle yapay zekâ çağı, bilgisayar mimarisi eğitimini önemsizleştirmek bir yana, daha da *can alıcı* hale getirdi. Buyruk kümesi tasarımı (Bölüm 3), işlemci mikromimarisi (Bölüm 6), boru hattı (Bölüm 7) ve bellek hiyerarşisi (Bölüm 8) gibi konular, hem geleneksel MİB'leri hem de yeni nesil hızlandırıcıları anlamanın temelini oluşturuyor.

RISC-V'in modüler genişletilebilirlik yapısı bu bağlamda özel bir avantaj sunuyor: Öğrenciler, temel RV32I çekirdeğine özel matris çarpımı buyrukları ekleyerek basit bir yapay zekâ hızlandırıcısını prototipleyebilir. Bu deneyim, alana özgü mimari tasarımının özünü kavramak için eşsiz bir fırsat sağlıyor.

Bilgi Notu 1.12: Büyük Dil Modelleri ve Donanım

ChatGPT ve benzeri büyük dil modelleri (Large Language Models: LLM), on milyarlarca parametreden oluşan yapay sinir ağları. GPT-4 düzeyinde bir modelin eğitimi, on bin GPU'dan oluşan bir küme üzerinde aylarca sürebiliyor ve milyonlarca dolar enerji maliyeti gerektiriyor. Modelin kullanıcılara sunulması (çıkartım) ise daha düşük güçle mümkün, ancak milyonlarca eş zamanlı istek düşünüldüğünde bu da büyük ölçekli bir donanım altyapısı gerektiriyor. Transformer mimarisindeki *dikkat mekanizması* (attention), bellek bant genişliği açısından son derece yoğun bir işlem olup yeni nesil donanım tasarımlarını doğrudan şekillendiriyor.

1.11 Yanılgılar ve Tuzaklar

Bilgisayar mimarisine giriş aşamasında bazı kavramlar sezgisel olarak doğru görünse de gerçekte yanıltıcıdır. Bu bölümde en yaygın yanlış kanıları ele alacağız.

Yanılgı: İşlemci hızı bilgisayarın hızını tek başına belirler

Bir bilgisayarın başarımı yalnızca işlemciye bağlı değildir. Bellek hızı, depolama erişim süresi, veri yolu bant genişliği ve yazılımın kalitesi de en az işlemci kadar belirleyicidir. Örneğin yavaş bir sabit disk (HDD) kullanan güçlü bir işlemci, hızlı bir SSD'ye sahip daha düşük saat sıklıklı bir sistemden uygulamada daha yavaş çalışabilir.

Yanılgı: Daha fazla transistör her zaman daha iyi başarımla demektir

Moore Yasası transistör sayısının artacağını öngörür, ancak bu artış doğrudan başarıma dönüşmez. Dennard ölçeklemesinin sona ermesiyle birlikte eklenen transistörler daha fazla güç tüketir ve ısınır. Bu nedenle çağdaş işlemciler transistörleri saat sıklığını artırmak yerine daha çok çekirdeğe, önbelleğe ve özelleşmiş birimlere ayırır.

Yanılğı: CISC mimarileri RISC'ten yavaştır

Alt Bölüm 1.5.3'te ele alındığı gibi, 1980'lerdeki RISC devrimiyle birlikte “basit buyruklar hızlıdır” sloganı yerleşmiş olsa da günümüzdeki tablo çok daha karmaşıktır. Çağdaş x86 işlemcileri CISC buyrukları donanım içinde mikro-işlemlere (RISC benzeri) çevirir ve boru hattından geçirir. Apple M serisi (ARM/RISC) ile Intel Core serisi (x86/CISC) arasındaki başarımların farkları artık mimari felsefeden çok mikromimari tasarım kararlarına bağlıdır.

Tuzak: Tek bir ölçütle bilgisayar seçmek

GHz, çekirdek sayısı, RAM miktarı gibi tek bir özelliğe bakarak bilgisayar karşılaştırmak yanıltıcıdır. Bir bilgisayarın gerçek başarımlı iş yüküne bağlıdır ve birden fazla ölçütün birlikte değerlendirilmesini gerektirir. Pazarlama kampanyaları genellikle tek bir sayıyı öne çıkarır; mühendislik ise bütünsel bir bakış açısı gerektirir.

Tuzak: “Açık kaynak” ile “ücretsiz” kavramlarını karıştırmak

RISC-V açık kaynaklı bir buyruk kümesi mimarisidir; bu, spesifikasyonun serbestçe kullanılabileceği anlamına gelir. Ancak bir RISC-V işlemcisinin tasarlanması, üretilmesi ve doğrulanması ciddi mühendislik yatırımı gerektirir. Açık kaynak, giriş engelini düşürür ama geliştirme maliyetini sıfırlamaz.

1.12 Özet

Bilgisayar mimarisi çok hızlı gelişen bir mühendislik alanıdır. Elektrik-elektronik ve bilgisayar mühendislerinin bir eşgüdüm içinde oluşturduğu değişik soyutlama katmanlarından oluşur. Bir bilgisayarın veri yolu, denetim, bellek, giriş ve çıkıştan oluşan beş ana bileşeni vardır; veri yolu ve denetimi içinde barındıran işlemci en önemli parçasıdır. Programlar üst düzey dillerden derleyiciler aracılığıyla makine diline dönüştürülür ve işletim sistemi, kütüphaneler ile aygıt yazılımından oluşan katmanlı bir yazılım altyapısı üzerinde çalışır. Buyrukların ve verilerin bellekte saklanma biçimine göre Von Neumann (ortak bellek) ve Harvard (ayrı bellekler) olmak üzere iki temel mimari model bulunur; günümüzde çoğu işlemci her ikisinin avantajlarını birleştiren değiştirilmiş Harvard modelini kullanır. Buyrukların tasarımında ise RISC ve CISC olmak üzere iki temel yaklaşım vardır; bu kitapta açık kaynaklı bir RISC mimarisi olan RISC-V kullanılmaktadır.

Bilgisayar tarihinde vakum tüplerinden transistörlere, tümleşik devrelerden mikroişlemcilere ve günümüzün çok çekirdekli heterojen sistemlerine uzanan beş nesil ayırt edilebilir. Moore Yasası'nın öngördüğü transistör artışı onlarca yıl boyunca endüstrinin yol haritasını belirledi, ancak Dennard ölçeklemesinin sona ermesiyle güç duvarı ortaya çıktı. Güç tüketimi, günümüzde işlemci tasarımının en temel kısıtlarından biri. Bilgisayar başarımının nasıl ölçüldüğü ve değerlendirildiği Bölüm 2'de ayrıntılı olarak ele alınıyor.

RISC-V'in açık kaynak, modüler ve eğitim dostu yapısı, onu hem bu kitabın hem de dünya genelinde giderek artan sayıda mimari projenin tercih ettiği buyruk kümesi haline getirdi. Yapay zekâ ise bilgisayar mimarisinin yeni ufku: GPU, TPU ve NPU gibi alana özgü hızlandırıcılar, genel amaçlı MİB'lerin tek başına karşılayamadığı devasa hesaplama gereksinimlerini verimli biçimde karşılamak için tasarlanıyor. Bu hızlandırıcıların mimarisi Bölüm 11'de kapsamlı olarak incelenecek.

İlerleyen bölümlerde başarımlar (Bölüm 2), buyruk kümesi mimarisi (Bölüm 3), sayısal aritmetik (Bölüm 5), işlemci tasarımı (Bölüm 6), boru hattı (Bölüm 7), bellek hiyerarşisi (Bölüm 8), giriş/çıkış (Bölüm 9), çok çekirdekli sistemler (Bölüm 10) ve GPU ile hızlandırıcılar (Bölüm 11) ele alınacak.

Alıştırmalar

Alıştırma 1.1. ★ Aşağıdaki sözcüklerin anlamları nedir? (a) Von Neumann Mimarisi, (b) Harvard Mimarisi, (c) CISC, (d) RISC.

Alıştırma 1.2. ★ Aşağıdaki kavramları açıklayın: (a) Makine Dili, (b) Derleyici, (c) Çevirici, (d) Programlama Dili, (e) İşletim Sistemi.

Alıştırma 1.3. ★ Buyruk yürütüm döngüsünü açıklayın.

Alıştırma 1.4. ★ Bir bilgisayarı oluşturan beş ana bileşen nedir?

Alıştırma 1.5. ★★ Geçmişin bilgisayar mimarisine nasıl bir etkisi vardır? Daha önce tasarlanmış işlemcilere uyum mutlaka gerekli midir?

Alıştırma 1.6. ★ Aşağıdaki kavramları açıklayın: (a) Vakum tüpü, (b) Transistör, (c) Tümlşik devre, (d) Yonga, (e) Silisyum ve plaka, (f) Moore yasası.

Alıştırma 1.7. ★★ Moore Yasası nedir? Moore Yasası diğer bilim dallarındaki gibi mutlak bir gerçeği mi tanımlar? Açıklayın.

Alıştırma 1.8. ★★ Eğer 2001 yılının Ekim ayında üretilen bir yongada 100 transistör varsa ve yongayı üreten şirket Moore yasasını izliyorsa, şirketin 2007 yılının Ekim ayında ürettiği yeni yongada kaç transistör olur?

Alıştırma 1.9. ★★ Bir üniversitedeki öğrenci sayısı şu anda yaklaşık 2000'dir. Eğer bu sayı Moore yasasında belirlenmiş olan yonga başına düşen transistör sayısının artış hızıyla artarsa üniversitenin mevcudunun 32000 kişi olması için ne kadar zaman geçmesi gerekir?

Alıştırma 1.10. ★★ Plaka maliyetiyle bir yonganın alanı arasındaki bağıntıyı gösterin ve buna göre yonga alanının tasarımındaki önemini yorumlayın.

Alıştırma 1.11. ★★★ Günümüz bilgisayar endüstrisinde Moore Yasası'nın yavaşlamasının yarattığı zorlukları ve bu zorluklara karşı geliştirilen mimari çözümleri tartışın. Güç duvarı kavramını açıklayın.

Alıştırma 1.12. ★★ Devingen ve durağan güç tüketimi arasındaki farkları açıklayın. Aynı işlemcinin daha düşük çalışma gerilimiyle çalıştırılması devingen güç tüketimini nasıl etkiler?

Alıştırma 1.13. ★★★ RISC ve CISC mimarilerinin avantaj ve dezavantajlarını karşılaştırın. RISC-V'in açık kaynak olmasının eğitim ve endüstri açısından sağladığı faydaları tartışın.

Alıştırma 1.14. ★★★ Moore yasası fiilen sona erdi mi, yoksa yalnızca yavaşladı mı? Aşağıdaki senaryoları değerlendirin ve her birinin avantaj ile sınırlılıklarını tartışın:

- (a) Transistör boyutlarını küçültmeye devam etmek (EUV litografi, GAA transistörler).
- (b) Üçüncü boyuta genişlemek (3B yonga istifleme, HBM bellek).
- (c) Alana özgü hızlandırıcılar tasarlamak (GPU, TPU, FPGA).
- (d) Yeni hesaplama paradigmaları geliştirmek (kuantum bilgisayarlar, nöromorfik yongalar).

Sizce gelecek on yılda bilgisayar başarımı artışının birincil kaynağı hangisi olacaktır? Yanıtınızı gerekçelendirin.

Alıştırma 1.15. ★★★ Yapay zekâ iş yüklerinin (özellikle derin öğrenme eğitimi ve çıkarımı) geleneksel MİB mimarisi üzerindeki taleplerini açıklayın. GPU, TPU ve NPU gibi hızlandırıcıların neden geliştirildiğini mimari tasarım perspektifinden tartışın.

Alıştırma 1.16. ★★ Genel amaçlı işlemci (MİB) ile alana özgü hızlandırıcı (GPU, TPU) arasındaki temel tasarım felsefesi farkını “gecikme odaklı” (latency-oriented) ve “işlem hacmi odaklı” (throughput-oriented) kavramlarıyla açıklayın.

Alıştırma 1.17. ★★★ x86, ARM ve RISC-V buyruk kümesi mimarilerini lisanslama modeli, kullanım alanları ve genişletilebilirlik açısından karşılaştırın. Açık kaynak bir buyruk kümesi mimarisinin ülkelerin teknolojik bağımsızlığı için neden önemli olabileceğini tartışın.

Alıştırma 1.18. ★ Aşağıdaki cümlelerde boş bırakılan yerleri doldurun ya da doğru/yanlış (D/Y) olarak yanıtlayın.

- (a) Saat vuruş sıklığı, saatin iki yükselen darbesi arasındaki geçen süreye denir. (D/Y)
- (b) 2 bayt adresleme yapılan bir mimaride buyruklar 128 bit genişliğindeyse, sonraki buyruğa erişmek için program sayacını _____ kadar artırmalıyız.
- (c) _____ mimarisinde buyruk ve veri aynı bellekte bulunur.
- (d) İşlemcilerde saat sıklığını devredeki _____ belirler.
- (e) İşlemcideki bellek gecikmeleri her zaman sabittir. (D/Y)
- (f) Bilgisayarınızın anabelleği _____ teknolojisini kullanır.
- (g) Von Neumann mimarisinde veri ve buyruk bellekleri ayrı iken, Harvard mimarisinde ortaktır. (D/Y)

Alıştırma 1.19. ★★ Aşağıdaki ifadelerin her biri için doğruysa D, yanlışsa Y yazın. Yanlış olanlarda gerekçenizi belirtin.

- (a) Moore Yasası fiziksel bir yasa değil, gözleme dayalı bir eğilimdir.
- (b) RISC mimarisinde buyruklar değişken uzunluktadır.
- (c) Harvard mimarisinde buyruk ve veri bellekleri birbirinden ayrıdır.
- (d) Aynı buyruk kümesini kullanan iki farklı işlemci, aynı programı farklı sürelerde çalıştırabilir.

Alıştırma 1.20. ★ Aşağıdaki boşlukları doldurun.

- (a) Bir bilgisayarı oluşturan beş ana bileşen: _____, _____, _____, giriş ve çıkıştır.
- (b) Programcı ile donanım arasında bir sözleşme niteliği taşıyan soyutlama katmanına _____ denir.

- (c) Von Neumann mimarisinde buyruklar ve veriler _____ bellekte saklanır.
 (d) Tüm bileşenleri bir arada tutan bağlantıya _____ denir.

Alıştırma 1.21. ★★★ Yeni bir buyruk kümesi tasarlıyorsanız, aşağıdaki tasarım seçeneklerinin her birinin olası artı ve eksilerini tartışın:

- (a) Yazmaç sayısının artırılması
 (b) Buyruk uzunluğunun sabit mi yoksa değişken mi olması
 (c) Anlık değer alanının büyütülmesi
 (d) Buyruk türü sayısının artırılması

Alıştırma 1.22. ★★★ Çapı 300 mm olan bir silisyum plakasının maliyeti 16 000 \$ olsun. Üretilecek yonganın kırmık alanı 120 mm^2 , birim alan başına hata sayısı $0,002 \text{ hata/mm}^2$ ve kritik aşama sayısı $N = 10$ ise:

- (a) Plakadan elde edilebilecek yaklaşık kırmık sayısını hesaplayın.
 (b) Verimi hesaplayın.
 (c) Bir kırmığın üretim maliyetini bulun.
 (d) Kırmık alanı iki katına (240 mm^2) çıkarılsaydı, maliyet nasıl değişirdi? Hesaplayın ve yorumlayın.

Alıştırma 1.23. ★★★ Bir yonga üreticisi aynı tasarımı iki farklı teknoloji düğümünde üretmeyi karşılaştırıyor:

	28 nm	5 nm
Plaka maliyeti	3 500 \$	17 000 \$
Kırmık alanı	250 mm^2	80 mm^2
Verim	%88	%72
Maske seti maliyeti	5 M \$	60 M \$

300 mm çaplı plaka kullanıldığını varsayın.

- (a) Her iki düğüm için plaka başına düşen kırmık maliyetini hesaplayın.
 (b) Her iki düğüm için maske seti payını 10 milyon adetlik ve 500 bin adetlik üretim hacimleri için ayrı ayrı hesaplayın.
 (c) Hangi üretim hacminde 5 nm ekonomik olarak anlamlı hâle gelir? Tartışın.

Alıştırma 1.24. ★★★ Yeni bir 3 nm üretim hattı başlatılıyor. Verim öğrenme eğrisi aşağıdaki gibi ilerliyor:

Çeyrek dönem	Ç1	Ç2	Ç3	Ç4	Ç5
Verim (%)	35	48	62	74	82

Plaka maliyeti 20 000 \$, kırmık alanı 100 mm^2 ve plaka çapı 300 mm ise:

- (a) Her çeyrek dönem için kırmık maliyetini hesaplayın.
 (b) Kırmık maliyetinin Ç1'den Ç5'e kadar yüzde kaç düştüğünü bulun.
 (c) Bir şirket ürününü Ç1'de piyasaya sürmek ile Ç3'ü beklemek arasında karar veriyor. Her iki senaryonun avantaj ve dezavantajlarını maliyet ve rekabet açısından tartışın.

Alıştırma 1.25. ★★ Aşağıdaki iki işlemcinin kırmık alanı ve plaka bilgileri verilmiştir:

	İşlemci A	İşlemci B
Kırmık alanı	50 mm ²	400 mm ²
Hata yoğunluğu	0,001 hata/mm ²	
Kritik aşama sayısı (N)	12	

- Her iki işlemcinin verimini hesaplayın.
- 300 mm çaplı plakadan her işlemciden kaç tane sağlam kırmık elde edilir?
- Verim farkının büyük kırmıklar için neden daha dramatik olduğunu açıklayın.

Alıştırma 1.26. ★★ Bir işlemci 1,0 V kaynak geriliminde ve 3 GHz saat sıklığında çalışırken devingen güç tüketimi 80 W'tır.

- Kaynak gerilimi 0,8 V'a ve saat sıklığı 2,4 GHz'e düşürülürse devingen güç tüketimi kaç watt olur?
- Bu değişiklikle güç tüketimi yüzde kaç azalır?
- Saat sıklığının düşürülmesi başarımı nasıl etkiler? Güç-başarım ödünleşimini tartışın.

Alıştırma 1.27. ★★★ Aynı görevi çalıştıran iki işlemciyi karşılaştırın:

	İşlemci X	İşlemci Y
Güç tüketimi	120 W	40 W
Görev süresi	5 s	18 s

- Her iki işlemci için toplam enerji tüketimini ($E = P \times t$) hesaplayın.
- Hangi işlemci enerji açısından daha verimlidir?
- Bir veri merkezinde bu görev günde 10 000 kez çalıştırılıyorsa, yıllık enerji farkı kaç kWh olur?
- "Düşük güçlü işlemci her zaman daha az enerji harcar" ifadesinin neden yanıltıcı olduğunu bu örnekle açıklayın.

Alıştırma 1.28. ★★ Bir taşınabilir işlemcinin durağan (sızıntı) güç tüketimi $P_{\text{durağan}} = I_{\text{sızıntı}} \times V_{dd}$ formülüyle hesaplanır. İşlemci 0,75 V kaynak geriliminde çalışmaktadır ve toplam sızıntı akımı 2,5 A'dır.

- Durağan güç tüketimini hesaplayın.
- Devingen güç tüketimi 4 W ise durağan gücün toplam güç içindeki yüzdesini bulun.
- Bu işlemci bir akıllı telefonda kullanılıyor ve telefon bekleme modundayken (devingen güç ≈ 0) yalnızca durağan güç tüketiyor. 5 000 mAh kapasiteli, 3,8 V'luk bir pilde saklanan enerji kaç Wh'tır? Telefon yalnızca sızıntı gücüyle kaç saat dayanır?

Alıştırma 1.29. ★★ Moore Yasası'na göre bir yongadaki transistör sayısı yaklaşık her 2 yılda ikiye katlanır.

- 2010 yılında 1 milyar transistör içeren bir işlemci ailesi Moore Yasası'nı izlerse, 2024'te kaç transistör içerir?

- (b) Gerçekte Apple M4 (2024) yaklaşık 28 milyar transistör içermektedir. Bu değer Moore Yasası tahminiyle uyumlu mu? Tartışın.
- (c) Moore Yasası'nın “yasa” olarak adlandırılması fizik yasalarıyla karşılaştırıldığında neden yanıltıcıdır?

Alıştırma 1.30. ★★ Bozukluk (*fault*), hata (*error*) ve arıza (*failure*) kavramları arasındaki farkı bir bellek modülü örneği üzerinden açıklayın. Hata düzeltme kodlarının (ECC) bu zincirdeki rolünü tartışın.

Alıştırma 1.31. ★ Geçici, aralıklı ve kalıcı bozuklukları tanımlayın. Her biri için birer gerçek dünya örneği verin. Hangisinin tanınması en zordur? Nedenini açıklayın.

Alıştırma 1.32. ★★★ Bir gömülü sistemde dört özdeş bileşen seri bağlı olarak çalışmaktadır. Her bileşenin MTTF'si 250 000 saattir.

- (a) Sistemin MTTF'sini hesaplayın.
- (b) Her bileşenin FIT değerini ve sistemin toplam FIT değerini bulun.
- (c) MTTR 4 saat ise sistemin kullanılabilirliğini hesaplayın.
- (d) Bu kullanılabilirlik “beş dokuz” (%99,999) hedefini karşılıyor mu? Karşılamıyorsa, bileşen MTTF'si en az ne olmalıdır?

Alıştırma 1.33. ★★★ Bir veri merkezi 2 000 sunucudan oluşmaktadır. Her sunucunun FIT değeri 20 000'dir ve ortalama onarım süresi 1,5 saattir.

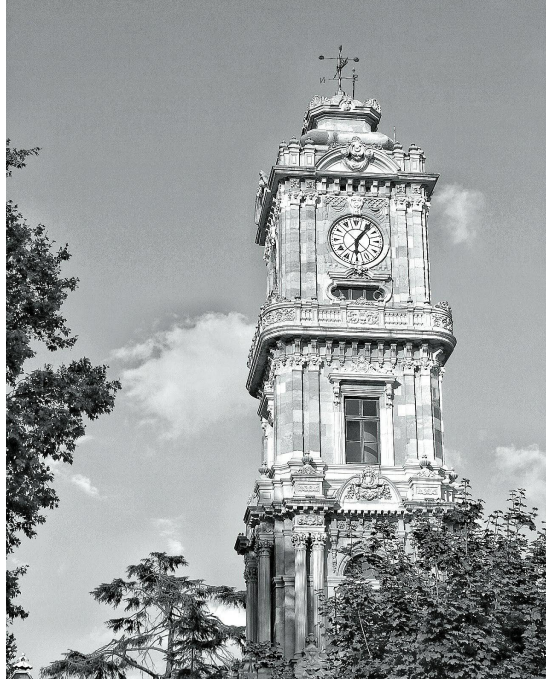
- (a) Tek bir sunucunun MTTF'sini saat ve yıl cinsinden bulun.
- (b) Veri merkezinin tamamı için MTTF'yi hesaplayın. Ortalama ne sıklıkla bir sunucu arızası beklenir?
- (c) Veri merkezinin kullanılabilirliğini hesaplayın.
- (d) Sunucu sayısı 10 000'e çıkarılırsa MTTF nasıl değişir? Büyük ölçekli sistemlerde yedeklemenin neden zorunlu olduğunu tartışın.

Alıştırma 1.34. ★★★ Bir otonom araç işlemcisi için MTTF hedefi en az 1 000 000 saat (yaklaşık 114 yıl) olarak belirlenmiştir.

- (a) Bu hedefe karşılık gelen FIT değerini hesaplayın.
- (b) İşlemci üç alt birimden oluşuyor: hesaplama birimi (FIT = 200), bellek denetleyicisi (FIT = 150) ve giriş/çıkış birimi (FIT = 100). Sistem seri bağlıysa toplam FIT ve MTTF değerlerini bulun. Hedef karşılanıyor mu?
- (c) Uzay uygulamalarındaki güvenilirlik gereksinimleri otomotivden nasıl farklıdır? Mars'taki Curiosity aracını örnek vererek tartışın.

Alıştırma 1.35. ★★ Güvenilirlik (*reliability*), dayanıklılık (*durability*) ve gürbüzlük (*robustness*) kavramlarını bir askeri dizüstü bilgisayar tasarımı bağlamında karşılaştırın. Her kavram için bu bilgisayarda alınması gereken bir tasarım kararı örneği verin.

Bilgisayar Başarımı



Dolmabahçe Sarayı Saat Kulesi – İstanbul

Dolmabahçe'nin saat kulesi 1890'dan bu yana zamanı ölçer. Bilgisayar başarımını ölçmek de sonuçta bir zamanlama sorusudur: aynı işi daha kısa sürede bitiren bilgisayar daha hızlıdır.

Öğrenme Hedefleri

Bu bölümü tamamladığınızda aşağıdakileri yapabileceksiniz:

- Başarım kavramını tanımlamak ve farklı iş yükleri için başarım ölçütlerini ayırt etmek.
- Yürütme zamanı denklemini (buyruk sayısı, BBÇ, saat sıklığı) kurmak ve hesaplama yapmak.
- İşlem hacmi ile gecikme arasındaki farkı açıklamak.
- MIPS, MFLOPS ve FLOPS ölçütlerinin güçlü ve zayıf yönlerini tartışmak.
- SPEC sinama programlarının nasıl çalıştığını ve SPEC puanının nasıl hesaplandığını açıklamak.
- Amdahl Yasası'nı uygulamak ve hızlanma üst sınırını hesaplamak.
- Gustafson Yasası'nın Amdahl'dan farkını gerekçelendirmek.
- Heterojen SoC mimarilerinde genişletilmiş Amdahl Yasası'nı uygulamak.
- Bellek duvarı ve güç-başarım dengesinin mimari tasarıma etkisini tartışmak.

“Hangi bilgisayar daha hızlı?” Basit görünen bu soru, aslında basit bir yanıtı olmayan bir sorudur. Bir süper bilgisayar hava tahmin modelini saniyeler içinde çözebilir; ancak aynı makineyi bir akıllı telefon uygulaması çalıştırmak için kullanmak anlamsız olur. *Başarım* (performance), bir bilgisayarın belirli bir iş yükünü ne kadar verimli gerçekleştirdiğini ölçen kavramdır ve “hızlı” kelimesinin ne anlama geldiği iş yüküne göre değişir.

Aslında “Hangi bilgisayar daha hızlı?” sorusundan önce sorulması gereken çok daha temel bir soru vardır: “*Sen bu bilgisayarla ne yapacaksın?*” Bu soruya verilen yanıt, kişiden kişiye kökten değişir:

- Bir **öğrenci** “Ödev yapacağım, kod yazacağım, ara sıra oyun oynayacağım” der; ona yeterli tepki süresi ve makul bir fiyat gerekir.
- Bir **oyun geliştiricisi** “Saniyede 120 kare ile gerçek zamanlı 3B sahne çizeceğim” der; düşük gecikme ve güçlü bir grafik işlemcisi ister.
- Bir **veri merkezi yöneticisi** “Saniyede yüz bin örün isteğine yanıt vereceğim” der; yüksek işlem hacmi ve enerji verimliliği arar.
- Bir **gömülü sistem mühendisi** “Sensör verisini on yıl boyunca pille işleyeceğim” der; en düşük güç tüketimi birincil hedeftir.
- Bir **yapay zekâ araştırmacısı** “Milyarlarca parametrelili bir modeli eğiteceğim” der; maksimum hesaplama gücü ve bellek bant genişliği arar.

Görüldüğü gibi herkes “hızlı bir bilgisayar” istemektedir; ama “hızlı” kelimesinin anlamı herkes için farklıdır. Gereksinimler tanımlanmadan ölçüt seçilemez; ölçüt seçilmeden ölçüm yapılamaz. Bu yüzden bu bölüme “Hangi bilgisayar daha hızlı?” sorusunun ötesine geçerek, “İyi başarım *kimin için ve neye göre* tanımlanır?” sorusuyla başlıyoruz.

Terim Kökenleri

başarım (*performance*): *Başarmak* (to accomplish) fiilinden *-ım* ekiyle. Bir sistemin bir işi ne kadar “başarıyla” gerçekleştirdiğinin ölçüsüdür. Fransızca kökenli *performans* sözcüğünün Türkçe karşılığı olarak türetilmiştir.

sinama programı (*benchmark*): *Sınamak* (to test) fiilinden *-ma* ekiyle oluşan *sinama* ve *program* sözcüklerinin birleşimi. Bir sistemin başarımını ölçmek için kullanılan standart test programıdır. İngilizce *benchmark* “referans noktası” anlamına gelir. Bazı kaynaklarda *denektaş* (deneme taşı: kuyumcuların altın ayarını sınıadığı taş) terimi de kullanılır; bu kitapta *sinama prog-*

ramı tercih edilmiştir.

Bir otomobil alırken yalnızca en yüksek hıza bakılmaz; yakıt tüketimi, güvenlik donanımı, konfor düzeyi ve fiyatı da değerlendirilir. Bilgisayar başarımı da öyle: tek bir sayıya indirgemek yerine iş yükünün niteliğine uygun ölçütleri birlikte ele almak gerekir. Bu bölümde önce gereksinimlerin nasıl belirlendiğini, ardından başarımın nasıl tanımlanıp ölçüldüğünü, FLOPS ve SPEC gibi karşılaştırma ölçütlerini, tarihsel başarımlar eğilimlerini ve Amdahl Yasası'nın tasarım kararlarına etkisini ele alacağız.

2.1 Gereksinim Belirleme

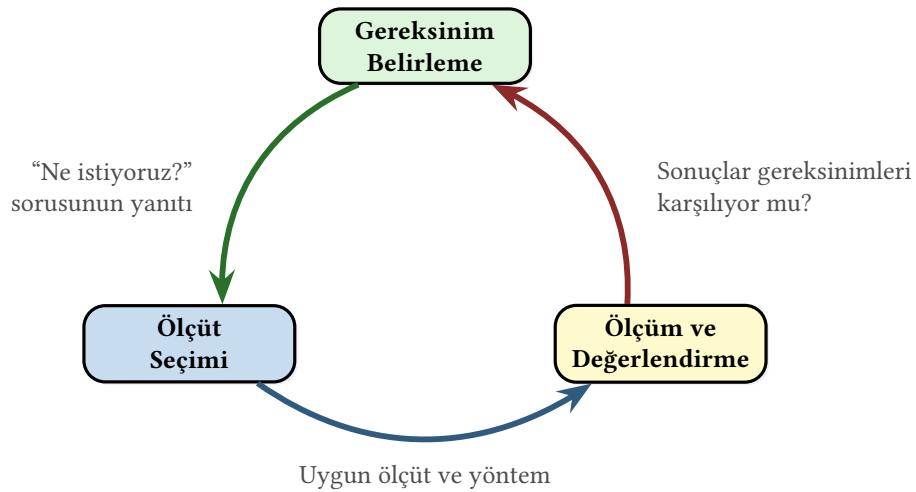
Bir bilgisayar mimarı, tasarıma başlamadan önce sistemin hangi koşullarda, hangi iş yüklerini, hangi kısıtlar altında çalıştırması gerektiğini tanımlamalıdır. Bu süreç *gereksinim belirleme* olarak adlandırılır ve başarımlar ölçümünün mantıksal ilk adımıdır.

Tasarım İlkesi 0: Önce gereksinimleri belirle

Başarımı ölçmek için önce neyin “başarılı” sayılacağını tanımlamak gerekir. Gereksinimler tanımlanmadan seçilen herhangi bir ölçüt (saat sıklığı, MIPS ya da FLOPS) yanıltıcı sonuçlar verebilir. Doğru soru “Bu bilgisayar hızlı mı?” değil, “Bu bilgisayar *hedeflenen iş yükünü kabul edilebilir kısıtlar altında çalıştırabiliyor mu?*” sorusudur.

2.1.1 Gereksinimden Ölçüme Giden Yol

Başarım değerlendirmesi üç aşamalı bir süreçtir. Şekil 2.1’de gösterildiği gibi bu aşamalar döngüsel bir ilişki içindedir: ölçüm sonuçları gereksinimlerin yeniden gözden geçirilmesine yol açabilir.



Şekil 2.1: Gereksinimden ölçüme giden üç aşamalı döngü

1. **Gereksinim belirleme:** Sistemin hangi iş yükünü çalıştıracığı, kabul edilebilir gecikme ve işlem hacmi değerleri, güç ve maliyet bütçesi tanımlanır.
2. **Ölçüt seçimi:** Gereksinimlere uygun başarımlar ölçütleri (yürütme zamanı, işlem hacmi, FLOPS/watt vb.) belirlenir.

3. **Ölçüm ve değerlendirme:** Seçilen ölçütler standart iş yükleri üzerinde ölçülür ve sonuçlar gereksinimlerle karşılaştırılır. Gereksinimler karşılanmıyorsa döngü yeniden başlar.

2.1.2 Uygulama Alanlarına Göre Gereksinimler

Farklı uygulama alanları temelden farklı başarımlar gereksinimleri ortaya koyar. Tablo 2.1, altı temel uygulama alanını birincil gereksinim, kısıt ve tipik ölçüt açısından karşılaştırmaktadır.

Tablo 2.1: Uygulama alanlarına göre başarımlar gereksinimleri

Uygulama alanı	Birincil gereksinim	Temel kısıt	Tipik ölçüt
Gömülü / IoT	Düşük güç, düşük maliyet	Sınırlı alan ve pil	mW/işlem
Masaüstü / oyun	Düşük gecikme	Maliyet, gürültü	Yürütme zamanı (ms)
Sunucu / bulut	Yüksek işlem hacmi	Enerji, soğutma	İşlem/sn/watt
Taşınabilir	Pil ömrü, tepki süresi	Termal kısıt	FLOPS/watt
YZ eğitimi	Maksimum hesaplama gücü	Bellek bant genişliği	TFLOPS (FP16/BF16)
HPC / bilimsel	Çift duyarlılık başarımı	Ölçeklenebilirlik	FP64 PFLOPS

Bu tablodan önemli bir sonuç çıkar: “en iyi bilgisayar” diye evrensel bir kavram yoktur. Bir süper bilgisayar, bilimsel hesaplama için en iyisi olabilir ama bir akıllı saatin içine sığmaz; bir IoT sensör düğümü on yıl pille çalışabilir ama bir derin öğrenme modeli eğitemez. Dolayısıyla başarımlar her zaman *bağlama göre* tanımlanmalıdır.

Örnek 2.1

Bir video sıkıştırma yazılımı üç farklı platformda çalıştırılacaktır. Her platformun gereksinimlerini ve uygun başarımlar ölçütünü belirleyin.

Platform	Senaryo	Uygun ölçüt
Akıllı telefon	Kullanıcı kısa video kaydediyor; pil ömrü can alıcı	Kare/joule
Masaüstü	Video editörü gerçek zamanlı önizleme ister	Kare/saniye
Yayın sunucusu	Binlerce kullanıcıya eş zamanlı yayın	Eş zamanlı akış/sunucu

Çözüm: Aynı yazılım, üç farklı platformda üç farklı ölçütlerle değerlendirilmelidir. Akıllı telefonda enerji başına kare sayısı (kare/joule), masaüstünde saniye başına kare sayısı (kare/saniye, bir gecikme ölçütü), yayın sunucusunda ise eş zamanlı akış sayısı (işlem hacmi ölçütü) birincil başarımlar göstergesidir. Yalnızca “saniye başına kare” ölçerek üç platformu karşılaştırmak yanıltıcı sonuçlar verir.

Bilgi Notu 2.1: Yanlış Ölçütle Ölçmek: Goodhart Yasası

İngiliz iktisatçı Charles Goodhart 1975'te şu gözlemi formüle etmiştir: “Bir ölçüt hedefe dönüştüğünde, iyi bir ölçüt olmaktan çıkar.” Bu ilke bilgisayar mimarisinde de geçerlidir. 2000’li yılların başında işlemci üreticileri saat sıklığını (GHz) pazarlama aracı olarak kullandığında, tasarımlar yalnızca saat sıklığını artırmaya yöneldi: daha derin boru hatları, daha yüksek güç tüketimi ve bazı iş yüklerinde düşen gerçek başarımlar. Intel’in Pentium 4 serisi bu tuzağın somut bir örneğidir: 3,8 GHz’lik Pentium 4, birçok iş yükünde 2 GHz’lik rakiplerinden yavaş çalışıyordu. Ölçüt seçimi, gereksinimlerden bağımsız yapıldığında benzer tuzaklara düşme riski her zaman vardır.

2.2 Başarım Ölçütleri

Gereksinimler belirlendikten sonra sıra, bu gereksinimleri somut sayılara dönüştürecek ölçütlerin tanımlanmasına gelir. Sayısal devrelerde hiçbir işlem sıfır zamanda gerçekleşmez; yapılan her işlemin devre düzeyinde bir gecikmesi vardır. Bu yüzden başarımın en temel ölçüsü, bir işin tamamlanması için geçen süredir. Peki bu süreyi neler belirler? Bu alt bölümde yürütme zamanını oluşturan bileşenleri teker teker inceleyelim.

2.2.1 Yürütme Zamanı

Sayısal devrelerin başarımı her zaman *zaman* ile ölçülür. Bir mantık devresine giriş verildiği andan doğru çıkışın elde edildiği ana kadar geçen süre, o devrenin hızını belirler. Bir toplayıcı devresi iki sayıyı 2 nanosaniyede topluyorsa, aynı işi 5 nanosaniyede yapan bir toplayıcıdan daha hızlıdır. Aynı ilke bilgisayar düzeyinde de geçerlidir: bir programın başlatılmasından tamamlanmasına kadar geçen süre, o bilgisayarın ilgili iş yükündeki başarımının en doğrudan göstergesidir.

Bir işin başlangıcı ile bitişi arasında geçen bu süreye *yürütme zamanı* ya da tepki zamanı (*execution time*) denir. Yürütme zamanı ile başarımlar arasındaki ilişki ters orantılıdır: süre ne kadar kısaysa başarımlar o kadar yüksektir. Tıpkı bir maratoncu gibi: İstanbul Maratonu’nu 2 saatte koşan atlet, 3 saatte koşandan daha başarılıdır çünkü aynı işi daha kısa sürede tamamlamıştır. Bu ters orantıyı şöyle ifade ederiz:

$$\text{Başarım} \propto \frac{1}{\text{Yürütme zamanı}} \quad (2.1)$$

Burada $\propto$ simgesi “ile orantılıdır” anlamına gelir. Bu ifadeyi bir *eşitlik* olarak yazmak (Başarım = $1/\text{Yürütme zamanı}$) cazip görünse de o zaman başarımın birimi 1/saniye, yani hertz (Hz), olurdu; bu fiziksel olarak anlamsızdır. “Bu bilgisayarın başarımı 0,1 Hz’dir” demek ne bir mühendise ne de bir kullanıcıya bir şey ifade eder.

Gerçekte başarımlar her zaman *karşılaştırmalı* olarak ölçülür. “Bu bilgisayar hızlıdır” cümlesi tek başına eksiktir; “Bu bilgisayar *şu bilgisayardan* hızlıdır” ya da “Bu bilgisayar *geçen yılki modelden* hızlıdır” demek gerekir. Günlük hayatta da böyle yaparız: yeni çıkan bir akıllı telefonun tanıtımında “önceki modelden %30 daha hızlı” denir, “başarımı 0,05 Hz’dir” denmez. Yeni bir otomobil tanıtılırken “0–100 km/sa hızlanma 4,5 saniye” dendiğinde bile zihinsel olarak bunu bildiğimiz başka otomobillerle karşılaştırırız.

İşte bu nedenle başarımın doğal ifade biçimi iki sistemin *oranıdır*:

$$\frac{\text{Başarım}_A}{\text{Başarım}_B} = \frac{\text{Yürütme zamanı}_B}{\text{Yürütme zamanı}_A} = n \quad (2.2)$$

“A, B’den n kat hızlıdır” demek, A’nın aynı programı B’den n kat daha kısa sürede çalıştırdığı anlamına gelir. Oran hesabında orantı sabiti sadeleştiği için birim sorunu ortadan kalkar. Örneğin bir program A bilgisayarında 20 saniyede, B bilgisayarında 60 saniyede tamamlanıyorsa A, B’den $60/20 = 3$ kat hızlıdır. Önemli olan 20 saniyenin mutlak değeri değil, B’ye göre 3 kat daha kısa olmasıdır.

Bilgi Notu 2.2: “2 kat daha hızlı” mı, “2 katı kadar hızlı” mı?

Günlük dilde “A, B’den 2 kat *daha* hızlı” ile “A, B’nin 2 *katı kadar* hızlı” ifadeleri sıklıkla birbirinin yerine kullanılır; ancak dikkatli okunduğunda farklı anlamlara gelebilir. “2 katı kadar hızlı” açıkça hızlanma = 2 demektir: A’nın başarımı B’nin başarımının 2 katıdır. “2 kat *daha* hızlı” ise B’nin hızına ek olarak 2 kat fazlasını (toplam 3 katı) ima edebilir; tıpkı “100 TL’den 50 TL *daha* fazla” dendiğinde 150 TL anlaşılması gibi. Mühendislik ve bilimsel yazında bu belirsizlikten kaçınmak gerekir. Bu kitapta “ n kat hızlı” ve “ n kat daha hızlı” ifadelerinin her ikisi de Denklem 2.2’deki oranı, başka deyişle $\text{hızlanma} = n$ anlamını taşır. Belirsizlik riski olan yerlerde oran doğrudan verilmelidir: “A’nın yürütme zamanı B’nin yarısıdır” ya da “hızlanma $2\times$ ” gibi.

Başarımın bu karşılaştırmalı doğası önemli bir sonuç doğurur: *mutlak başarımlar* diye bir kavram yoktur. Hiçbir bilgisayar için “bu bilgisayarın başarımı şu kadardır” gibi tek başına anlamlı bir sayı verilemez; başarımlar her zaman bir referansa göre ölçülür. Dahası, her sistem daha iyisini yapabilecek bir tasarımla karşılaştırılabilir; her zaman daha hızlı bir çözüm üretilir. Bugünün en hızlı süper bilgisayarı, birkaç yıl sonra orta düzey bir sistemin gerisinde kalabilir. Bu gerçek, bilgisayar mimarisini sürekli gelişen bir mühendislik alanı kılar: bugünün en iyi tasarımı, yarının başlangıç noktasıdır.

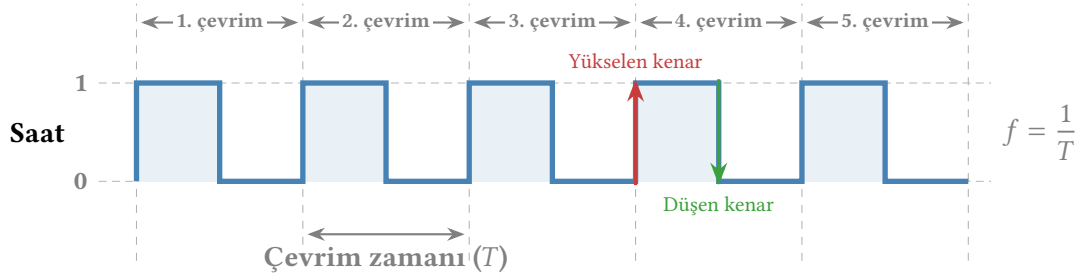
Yürütme zamanı iki farklı biçimde ölçülebilir:

- **Geçen zaman** (elapsed time / wall-clock time): Programın başlatılmasından tamamlanmasına kadar duvardaki saatte geçen toplam süre. Disk erişimi, bellek erişimi, giriş/çıkış (G/Ç) bekleme süreleri ve işletim sisteminin diğer programlara ayırdığı zaman dilimlerini de kapsar.
- **MİB zamanı** (CPU time): Yalnızca işlemcinin ilgili program için harcadığı süre. G/Ç bekleme süreleri ve başka programların yürütülmesine ayrılan zaman dahil değildir.

MİB zamanı kendi içinde de ikiye ayrılır: *kullanıcı MİB zamanı* (programın kendi buyruklarının yürütülmesi) ve *sistem MİB zamanı* (program adına çalışan işletim sistemi çağrıları). Başarım karşılaştırmalarında genellikle *kullanıcı MİB zamanı* tercih edilir çünkü işletim sistemi ek yükü, G/Ç süreleri ve sistemdeki diğer programlar donanımdan bağımsız olarak değişkenlik gösterir. Bir programın geçen zamanı 10 saniye olabilir ama bunun yalnızca 3 saniyesi gerçekten işlemcide harcanmış, geri kalanı diskten veri okunması beklenilerek geçirilmiş olabilir. Bu kitapta aksi belirtilmedikçe “yürütme zamanı” ile *MİB zamanını* kastediyoruz. Denklem 2.1’deki ters orantı ve Denklem 2.2’deki karşılaştırma da bu MİB zamanı üzerine kuruludur.

2.2.2 Saat Sıklığı ve Çevrim Zamanı

Bir kadirgadaki davulcuyu düşünün: davulcu düzenli aralıklarla davula vurur ve her vuruşta tüm kürekçiler aynı anda kürek çeker. Davulcu hızlanırsa kadirga hızlanır; ama kürekçiler yetiştiremeyecek kadar hızlı vurursa düzen bozulur. İşlemci *saat sinyali* tam olarak bu davulcunun rolünü üstlenir: düzenli aralıklarla yükselen ve düşen bir kare dalga üreterek işlemcinin tüm birimlerinin (yazmaçların, toplayıcıların, bellek denetleyicisinin) aynı anda adım atmasını sağlar. Şekil 2.2’de bu kare dalga sinyali gösterilmektedir.



Şekil 2.2: İşlemci saat darbeleri: çevrim zamanı (T) ve saat sıklığı ($f = 1/T$)

Saatın iki yükselen ya da iki düşen darbesi arasında geçen süreye *çevrim zamanı* denir. 1 saniyelik zaman aralığındaki çevrim sayısı ise *saat vuruşu sıklığı* (frekans) olarak tanımlanır. Peki işlemci hangi kenarı kullanır? Büyük çoğunluk *yükselen kenarı* temel alır: flip-floplar yalnızca saat sinyali 0’dan 1’e geçtiğinde yeni değerlerini yakalar, geri kalan sürede içeriklerini korur. Düşen kenar kullanılarak da aynı sonuç elde edilebilir; önemli olan tasarım boyunca tutarlı bir seçim yapılmasıdır.¹

$$\text{Çevrim zamanı} = \frac{1}{\text{Saat sıklığı}} \quad (2.3)$$

Sıklık birimi olan Hertz, bir saniyede bir olayın kaç kere gerçekleştiğini gösterir. Kadirga benzetmemize dönersek: davulcu davula saniyede 2 kez vuruyorsa vuruş sıklığı 2 Hz, çevrim zamanı ise $1/2 = 0,5$ saniyedir. Günümüz işlemcilerinde saat sıklığı milyar Hertz (GHz) mertebesinde; 4 GHz’lik bir işlemcide davulcu saniyede 4 milyar kez vurmakta, her vuruş yalnızca 0,25 ns sürmektedir.

İşlemcinin içinde zaman kavramını anlamasının tek yolu saat vuruşlarını, başka bir deyişle çevrimleri, saymasıdır. İşlemci “10 nanosaniye geçti” diye düşünmez; onun bildiği tek şey “5 çevrim geçti”dir. Bu nedenle işlemci içindeki tüm zamanlar *çevrim sayısı* cinsinden ifade edilir: bir bellek erişimi kaç çevrim sürer, bir çarpma kaç çevrimde tamamlanır, bir program baştan sona kaç çevrimde biter; hepsi çevrim sayısı ile ölçülür. Saat sıklığı bilindiği için çevrim sayısını gerçek zamana dönüştürmek kolaydır: 5 GHz’lik bir işlemcide 10 çevrim, $10 \times 0,2 \text{ ns} = 2 \text{ ns}$ demektir.

Bir programın yürütülmesi sırasında işlemcinin harcadığı toplam saat çevrimi sayısına *çevrim sayısı* (cycle count) denir. Çevrim sayısı programdan programa büyük farklılık gösterir: basit bir toplama işlemi birkaç düzine çevrimde tamamlanabilirken, karmaşık bir sıralama algoritması milyarlarca çevrim gerektirebilir. Çevrim sayısını belirleyen iki temel etken vardır: programda kaç buyruk yürütüldüğü ve her bir buyruğun kaç çevrimde tamamlandığı. Bu ayrıntılı çözümlemeyi bir sonraki alt bölümde BBÇ kavramıyla ele alacağız; şimdilik çevrim sa-

¹Bazı bellek teknolojileri her iki kenarı da kullanır. DDR (Double Data Rate) bellek adını tam da buradan alır: hem yükselen hem düşen kenarda veri aktararak aynı saat sıklığında iki kat bant genişliği sağlar.

yısını bilinen bir değer olarak kabul edelim. Saat sıklığını belirleyen kritik yol analizini ve saat dağıtım ağlarını Bölüm 6’da inceleyeceğiz.

Çevrim sayısı ile çevrim zamanını çarparak programın toplam yürütme zamanını elde ederiz:

$$\text{Yürütme zamanı} = \text{Çevrim sayısı} \times \text{Çevrim zamanı} = \frac{\text{Çevrim sayısı}}{\text{Saat sıklığı}} \quad (2.4)$$

Bu formülü küçük bir örneklerle pekiştirelim. İki firmayı düşünelim: her ikisi de RISC-V BKM’sini kullanan bir işlemci üretmektedir ve aynı programı çalıştıracaklardır. Aynı BKM, aynı program; dolayısıyla her iki işlemci de bu programı aynı çevrim sayısında (diyelim 12 milyar çevrimde) tamamlar.² Fark, üretim teknolojisindedir: İşlemci A eski 7 nm süreçle üretilmiş ve 3 GHz’de çalışmaktadır; İşlemci B ise daha yeni 5 nm süreçle üretilmiş ve transistörleri daha hızlı anahtarlama yapabildiği için 4 GHz’e ulaşmaktadır. Denklem 2.4’i uygularsak:

$$T_A = \frac{12 \times 10^9}{3 \times 10^9} = 4 \text{ s}, \quad T_B = \frac{12 \times 10^9}{4 \times 10^9} = 3 \text{ s}$$

Her iki işlemci aynı BKM’yi kullanıp aynı programı çalıştırdığı hâlde, yalnızca üretim teknolojisindeki fark saat sıklığını değiştirmiş ve İşlemci B programı %25 daha kısa sürede tamamlamıştır. Başarımı etkileyen etkenlerin yalnızca yazılım ya da mimari tasarımıyla sınırlı olmadığı, yarı-iletken üretim teknolojisinin de doğrudan başarımı belirlediği burada açıkça görülmektedir.

Saat sıklığını fiziksel olarak belirleyen etken, işlemciye *kritik yol gecikmesidir*. Kritik yol, bir saat çevrimi içinde sinyalin geçmesi gereken en uzun mantık kapısı zinciridir. Çevrim zamanı bu gecikmenin altına düşürülemez; aksi halde sinyaller hedef noktaya ulaşmadan yeni çevrim başlar ve devre yanlış sonuçlar üretir. Saat sıklığını artırmanın bir yolu kritik yoldaki kapı sayısını azaltmaktır (örneğin boru hattı aşamalarını daha ince dilimlere bölmek gibi); ancak bu genellikle her çevrimde yapılan iş miktarını düşürür ve BBÇ’yi artırır. İşte saat sıklığı ile BBÇ arasındaki bu ödünleşme, başarımlar mühendisliğinin temel zorluklarından biridir.

Terim Kökenleri

ödünleşme (*trade-off*): Ödün “taviz, bir şeyden vazgeçme” demektir; -leşme eki karşılıklılık bildirir. Bir özelliği kazanmak için başka birinden vazgeçmeyi, yani İngilizce *trade-off* (takas) kavramını, ifade eder. Bilgisayar mimarisinde neredeyse her tasarım kararı bir ödünleşme barındırır.

Kritik yol gecikmesini belirleyen yalnızca mantık kapıları değildir; kapılar arasındaki *tel gecikmeleri* de toplam gecikmenin önemli bir bölümünü oluşturur. Bunu anlamak için evrendeki en hızlı şeyden, ışık hızından, başlayalım. Işık boşlukta saniyede yaklaşık 300 000 km yol alır; bu, Dünya’nın çevresini bir saniyede yedi buçuk kez dolanabilecek, akıl almaz bir hızdır. Şimdi 4 GHz’lik bir işlemciyi düşünelim: bir saat çevrimi yalnızca 250 ps (pikosaniye, 250×10^{-12} s) sürer. Bu sürede ışık bile yalnızca

$$3 \times 10^8 \text{ m/s} \times 250 \times 10^{-12} \text{ s} = 0,075 \text{ m} = 7,5 \text{ cm}$$

yol alabilir; bu bir ana kartın uzun kenarından bile kısadır! Üstelik işlemcinin içindeki sinyaller boşlukta değil, bakır teller üzerinde ilerler; bakırda yayılma hızı ışık hızının yaklaşık üçte

²Gerçekte aynı BKM’yi kullanan iki farklı işlemci, mikromimari farklılıkları nedeniyle aynı programı farklı çevrim sayılarında tamamlayabilir. Burada karşılaştırmayı yalınlaştırmak için çevrim sayısını sabit tutuyoruz.

ikisine ($\sim 2 \times 10^8$ m/s) düşer ve bir çevrimdeki menzil yalnızca ~ 5 cm'ye iner. Mantık kapılarının kendi anahtarlama gecikmeleri de bu bütçeden pay aldığına, kritik yol üzerindeki fiziksel tel uzunluğu birkaç santimetreyi geçemez.

Bu fiziksel gerçek doğrudan bir tasarım ilkesine yol açar: bir birimi küçültmek, içindeki telleri kısaltır ve sinyal gecikmesini azaltır; dolayısıyla daha yüksek saat sıklığına olanak tanır. Tersine, büyük bir yazmaç öbeği veya geniş bir çoklayıcı daha uzun teller ve daha fazla kapı gecikmesi demektir; fiziksel boyut doğrudan hızı sınırlar.

Tasarım İlkesi 1: Küçük olan hızlıdır

Donanımda küçük yapılar büyük yapılardan daha hızlı çalışır. Daha az sayıda eleman, daha kısa bağlantı hatları ve daha az kapı gecikmesi anlamına gelir. Bu ilke, yazmaç öbeğinin boyutundan önbellek büyüklüğüne, boru hattı aşamalarının karmaşıklığından veri yolu genişliğine kadar pek çok mimari kararı doğrudan etkiler. Tabii ki “küçük” her zaman “daha iyi” demek değildir: az sayıda yazmaç derleyicilerin elini bağlar, küçük bir önbellek bulma oranını düşürür. İyi bir tasarım, küçüklüğün hız avantajı ile büyüklüğün işlevsel avantajı arasında doğru dengeyi bulmaktır.

Denklem 2.4'nin önemli bir iletişi vardır: saat sıklığını iki katına çıkarmak, çevrim sayısı aynı kalırsa yürütme zamanını yarıya düşürür. Ancak yürütme zamanını belirleyen yalnızca saat sıklığı değildir; programın kaç buyruktan oluştuğu ve her buyruğun kaç çevrimde tamamlandığı da denkleme girer. Sonraki iki alt bölümde bu kavramları ele alacağız.

2.2.3 Buyruk Sayısı

Bir programın yürütme zamanını etkileyen ikinci büyük bileşen, programın işlemci tarafından yürütülmesi gereken toplam **buyruk sayısıdır** (*instruction count*). Aynı üst düzey program (örneğin C++ ile yazılmış bir oyun) farklı BKM'ler için derlendiğinde farklı sayıda makine buyruğu üretir.

Bunu somut bir örnek ile düşünelim. *Kızgın Kuşlar* (🔥) oyununu hem bir iPhone'da (ARM işlemci) hem de Intel tabanlı bir dizüstü bilgisayarda (x86 işlemci) oynayan kişi, oyun deneyiminde bir fark görmez. Oysa derleyici aynı kaynak kodundan iki çok farklı buyruk dizisi üretmiştir: x86'nın karmaşık buyrukları az sayıda buyrukla çok iş yapabilirken, ARM'ın daha basit buyrukları aynı işi daha çok buyrukla ifade eder. RISC-V'e derlendiğinde ise buyruk sayısı yine farklı olacaktır. Kullanıcı aynı kuşu aynı hedefe fırlatsa da perde arkasında yürütülen buyruk sayısı platformdan platforma değişir.

Tablo 2.2 bu farkı somut biçimde gösterir. Oyundaki hasar hesabını gerçekleyen tek satırlık bir C ifadesi üç farklı BKM'de derlendiğinde birbirinden çok farklı makine buyruklarına dönüşür:

Tablo 2.2: Tek satırlık bir C ifadesinin ($\text{puan} = \text{hasar}[\text{tur}] * \text{carpan}$) üç farklı BKM’de derlenmiş hâli. Buyruk ayrıntılarını Bölüm 3’te öğreneceğiz; burada önemli olan buyruk sayısı ve türlerindeki farktır.

x86 (CISC)	ARM (RISC)	RISC-V (RISC)
1 mov eax, [tur]	ldr r0, [r4]	lw t0, 0(a4)
2 mov eax, [hasar+eax*4]	ldr r1, [r5, r0, lsl #2]	slli t0, t0, 2
3 imul eax, [carpan]	ldr r2, [r6]	add t0, a5, t0
4 mov [puan], eax	mul r0, r1, r2	lw t1, 0(t0)
5	str r0, [r7]	lw t2, 0(a6)
6		mul t0, t1, t2
7		sw t0, 0(a7)
4 buyruk	5 buyruk	7 buyruk

x86, `mov eax, [hasar+eax*4]` gibi tek buyrukta ölçekli dizin adresleme yapabildiği ve `imul eax, [carpan]` ile bellekten doğrudan çarpma gerçekleştirebildiği için aynı işi yalnızca 4 buyrukla bitirir. ARM’ın kaydırma birimi (*barrel shifter*) adres hesabını yükleme buyruğunun içine gömer; buyruk sayısı 5’e düşer. RISC-V ise en yalın yapıdadır: dizin hesabı (`slli + add`) ve veri yükleme ayrı buyruklarda yapılır, toplam 7 buyruk gerekir. Ancak “az buyruk = hızlı” demek değildir; karmaşık bir buyruk daha çok çevrimde tamamlanabilir. Bu dengeyi BBÇ kavramıyla ele alacağız.

Buyruk sayısını belirleyen başlıca etkenler şunlardır:

- **BKM:** CISC mimariler (x86 gibi) karmaşık buyruklar sunarak daha az buyrukla aynı işi yapabilir; RISC mimariler (ARM, RISC-V gibi) basit buyruklar kullanır ve genellikle daha fazla buyruk üretir.
- **Derleyici:** Aynı BKM için bile farklı eniyileme düzeyleri (-O0, -O2, -O3) buyruk sayısını önemli ölçüde değiştirir. Eniyileyici bir derleyici gereksiz buyrukları eler, döngüleri açar veya buyrukları birleştirir.
- **Programlama dili ve algoritma:** Üst düzey dildeki bir satır, seçilen algoritmaya ve veri yapısına bağlı olarak onlarca ya da yüzlerce makine buyruğuna dönüşebilir.

Buyruk sayısı tek başına başarıımı belirlemez; az buyruk her zaman daha hızlı demek değildir. Her buyruğun kaç çevrimde tamamlandığı da en az o kadar önemlidir. İşte bir sonraki kavram, tam da bu soruyu yanıtlar.

Durağan ve Devingen Buyruk Sayısı

Buyruk sayısından söz ederken iki farklı kavramı birbirinden ayırt etmek gerekir:

- *Durağan buyruk sayısı (static instruction count):* Programın makine kodunda yer alan farklı buyruk satırlarının sayısıdır. Bu değer derleme tamamlandığında sabittir ve programın bellekte kapladığı alanı belirler.
- *Devingen buyruk sayısı (dynamic instruction count):* Programın çalışması sırasında gerçekte yürütülen toplam buyruk sayısıdır. Döngüler aynı buyrukları tekrar tekrar yürüttüğü için devingen sayı, durağan sayıdan çok daha büyük olabilir.

Başarım denklemindeki buyruk sayısı her zaman *devingen* buyruk sayısıdır; çünkü yürütme zamanını belirleyen, programda yazılı olan değil gerçekte yürütülen buyruklardır.

Bilgi Notu 2.3: Üç Farklı Buyruk Sayısı

Çevirici dilde programcının yazdığı buyruklar arasında *sözde buyruklar* (*pseudo-instructions*) da bulunabilir. Sözde buyruklar çevirici tarafından bir veya daha fazla gerçek buyruğa dönüştürülür. Bu durumda üç farklı buyruk sayısından söz edebiliriz:

1. Programcının yazdığı buyruk sayısı (sözde buyruklar dahil),
2. Durağan buyruk sayısı: çevirici sözde buyrukları açtıktan sonra bellekte yer alan gerçek buyruk sayısı,
3. Devingen buyruk sayısı: çalışma sırasında yürütülen toplam buyruk sayısı.

Örneğin programcının yazdığı 10 sözde buyruk, çevirici tarafından 12 gerçek buyruğa açılabilir (durağan sayı = 12); bu 12 buyruğun bir kısmı döngü içinde 100 kez çalışırsa devingen sayı çok daha büyük olur. Başarım hesaplamalarında her zaman gerçek (durağan veya devingen) buyruk sayılarını kullanmalıyız. Sözde buyrukların ayrıntıları Bölüm 3'te ele alınmaktadır.

Bu ayrımı basit bir örnekle görelim. Aşağıdaki C kodu bir dizinin her elemanını 1 artırır:

```
1 for (int i = 0; i < 100; i++)
2     A[i] = A[i] + 1;
```

Derleyici bu döngüyü kabaca şu şekilde çevirir (buyruk ayrıntılarını Bölüm 3'te öğreneceğiz):

```
1     addi t0, zero, 0      # i = 0
2 don:
3     slli t1, t0, 2        # t1 = i * 4
4     add  t1, a0, t1       # t1 = adres(A[i])
5     lw   t2, 0(t1)       # t2 = A[i]
6     addi t2, t2, 1       # t2 = A[i] + 1
7     sw   t2, 0(t1)       # A[i] = t2
8     addi t0, t0, 1       # i++
9     blt  t0, a1, don     # i < 100 ise tekrarla
```

Bu programda 8 durağan buyruk vardır. Ancak döngü 100 kez yinelendiğinde yürütülen toplam buyruk sayısı $1 + 7 \times 100 = 701$ olur (ilk addi bir kez, döngü gövdesindeki 7 buyruk 100'er kez). Başarım denkleminde kullanılacak değer 701'dir, 8 değil.

Derleyicilerin sıkça başvurduğu en iyileme yöntemlerinden biri *döngünün açılması* (*loop unrolling*). Bu yöntemde döngünün gövdesi birden fazla kopyalanarak yineleme sayısı azaltılır. Yukarıdaki döngü tamamen açıldığında döngü yapısı ortadan kalkar:

```
1 A[0] = A[0] + 1;
2 A[1] = A[1] + 1;
3 A[2] = A[2] + 1;
4     ...           // 100 satır
5 A[99] = A[99] + 1;
```

Açılmış döngüde her satır yaklaşık 4 buyruğa derlenir (yükle, artır, sakla, adres hesapla); toplam $4 \times 100 = 400$ durağan buyruk ortaya çıkar. Devingen buyruk sayısı da 400'dür çünkü artık döngü yoktur ve her buyruk yalnızca bir kez yürütülür. Döngü yapısındaki dallanma ve sayaç güncelleme buyrukları tamamen ortadan kalktığı için devingen buyruk sayısı 701'den 400'e düşer. Buna karşılık durağan buyruk sayısı 8'den 400'e çıkar ve program bellekte çok

daha fazla yer kaplar. Bu ödünleşmeyi ilerideki bölümlerde (boru hattı ve önbellek) daha ayrıntılı ele alacağız.

2.2.4 Buyruk Başına Çevrim (BBÇ)

Saat sıklığı, davulcunun ne kadar sık vurduğunu söyler; buyruk sayısı, yürütülecek adım sayısını verir. Ancak bir işlemcinin birim zamanda ne kadar iş bitirdiğini anlamak için üçüncü bir bileşen daha gerekir: her buyruğun ortalama kaç çevrimde tamamlandığı. Aynı saat sıklığında çalışan iki işlemciden biri her buyruğu 2 çevrimde, diğeri 4 çevrimde bitiriyorsa, birincisi iki kat daha fazla iş yapar. İşte bu “buyruk başına çevrim” kavramı, saat sıklığının ve buyruk sayısının tek başına anlatamadığı gerçeği ortaya koyar.

Terim Kökenleri

buyruk başına çevrim (*cycles per instruction, CPI*): Bir buyruğun yürütülmesi için harcanan ortalama saat çevrimi sayısını veren ölçüttür. Kitapta kısaca *BBÇ* olarak kullanılmaktadır.

Buyruk başına çevrim (BBÇ), programın çalıştırılması için gereken buyruklardan her birinin yürütülmesi için geçen ortalama çevrim sayısıdır:

$$BBÇ = \frac{\text{Toplam çevrim sayısı}}{\text{Buyruk sayısı}} \quad (2.5)$$

Farklı buyruk türleri farklı sayıda çevrime ihtiyaç duyar. Örneğin bir toplama buyruğu 1 çevrimde, bir bellek erişim buyruğu 3–5 çevrimde tamamlanabilir. Bir programın toplam çevrim sayısı, her buyruk türünün gerektirdiği çevrim sayısı ile o buyruktan kaç kere yürütüldüğünün çarpımlarının toplamıdır:

$$\text{Toplam çevrim sayısı} = \sum_{i=1}^n \text{Çevrim}_i \times S_i \quad (2.6)$$

Burada n programdaki buyruk türlerinin sayısını, Çevrim_i ilgili buyruğun kaç çevrimde işlendiğini, S_i ise buyruğun programın işletilmesi sırasında kaç kere yürütüldüğünü gösterir.

Tablo 2.2’teki Kızgın Kuşlar örneğiyle BBÇ’yi somutlaştıralım. RISC-V kodundaki 7 buyruğu türlerine göre sınıflandırıp her türe basit bir çevrim maliyeti atayalım:³

RISC-V: 7 buyruk				x86 4 buyruk			
Buyruk türü	Çvr.	Adet	Top.	Buyruk türü	Çvr.	Adet	Top.
Aritmetik (add, slli)	1	2	2	Yükleme (mov)	2	1	2
Yükleme (lw)	2	3	6	Dizinli yükleme (mov)	3	1	3
Saklama (sw)	2	1	2	Bellek iş. çarpma (imul)	4	1	4
Çarpma (mul)	3	1	3	Saklama (mov)	2	1	2
Toplam		7	13	Toplam		4	11

$$BBÇ = 13 / 7 \approx 1,86$$

$$BBÇ = 11 / 4 = 2,75$$

RISC-V daha çok buyruk kullanıyor ama her biri daha az çevrime mal oluyor ($BBÇ \approx 1,86$); x86 ise daha az buyrukla işi yapıyor ama her buyruğu daha çok çevrim gerektiriyor ($BBÇ$

³Buradaki çevrim sayıları kavramsal anlaşılabilirlik için seçilmiş yalınlaştırılmış değerlerdir. Gerçek çevrim maliyetleri mikromimari tasarıma bağlıdır ve boru hattı, önbellek yapısı gibi etkenlere göre değişir.

= 2,75). Dikkat çekici nokta şudur: buyruk sayıları arasında neredeyse iki kat fark varken (4’e karşı 7), toplam çevrim sayıları çok daha yakındır (11’e karşı 13). İşte BBÇ’nin buyruk sayısıyla birlikte değerlendirilmesinin gerekliliği burada ortaya çıkar.

Çevrim Başına Buyruk (ÇBB)

BBÇ’nin tersi olan **Çevrim Başına Buyruk** (*Instructions Per Cycle*, ÇBB), her saat çevriminde tamamlanan ortalama buyruk sayısını verir:

$$\text{ÇBB} = \frac{1}{\text{BBÇ}} \quad (2.7)$$

BBÇ’nin düşük olması istenirken ÇBB’nin yüksek olması istenir; ikisi aynı bilgiyi ters yönden ifade eder. Yukarıdaki örnekte RISC-V’in ÇBB’si $1/1,86 \approx 0,54$, x86’nın ÇBB’si $1/2,75 \approx 0,36$ ’dır: RISC-V her çevrimde ortalama yarım buyruk bitirirken x86 üçte birden biraz fazlasını bitirir. Çağdaş çok yollu işlemciler ise her çevrimde birden fazla buyruk başlatıp tamamlayabildiği için $\text{ÇBB} > 1$ değerlerine ulaşabilir. İşlemci üreticileri başarımlarında genellikle ÇBB’yi tercih eder çünkü “daha yüksek = daha iyi” sezgisi daha kolay kavranır.

2.2.5 Hepsini Bir Araya Getirelim: Yürütme Zamanı Denklemi

Önceki üç alt bölümde başarımın üç temel bileşenini ayrı ayrı tanıdık: saat sıklığı (davulcunun vuruş hızı), buyruk sayısı (yürütülecek adım sayısı) ve BBÇ (her adımın kaç vuruşta tamamlandığı). Şimdi bu üçünü tek bir denklemde bir araya getiriyoruz:

$$\text{Yürütme zamanı} = \text{Buyruk sayısı} \times \text{BBÇ} \times \text{Çevrim zamanı} = \frac{\text{Buyruk sayısı} \times \text{BBÇ}}{\text{Saat sıklığı}} \quad (2.8)$$

Denklemin birim çözümlemesini yaparak doğruluğunu sınayabiliriz; her çarpan bir öncekinin birimini götürür ve geriye yalnızca “saniye” kalır:

$$\text{Yürütme zamanı} = \frac{\text{Buyruk sayısı}}{\text{program}} \times \frac{\text{Çevrim sayısı}}{\text{buyruk}} \times \frac{\text{saniye}}{\text{çevrim}} \quad (2.9)$$

Bu denklemin en can alıcı iletisi şudur: **üç bileşen birbirinden bağımsız değildir**. Birini iyileştirmeye çalışmak, diğerlerini kötüleştirebilir:

- Saat sıklığını artırmak için boru hattı derinleştirilir → BBÇ artar (her buyruk daha çok aşamadan geçer).
- Buyruk sayısını azaltmak için karmaşık buyruklar tanımlanır (CISC yaklaşımı) → BBÇ artar (karmaşık buyruklar daha çok çevrim ister).
- BBÇ’yi düşürmek için sırasız yürütüm kullanılır → devre karmaşıklığı artar ve saat sıklığı düşebilir.

Dolayısıyla herhangi bir bileşeni tek başına eniyilemeye çalışmak yanıltıcıdır. Bir bileşendeki kazanım, diğerindeki kayıpla silinebilir. İyi bir tasarımcı her zaman *toplam yürütme zamanına* bakar, tek bir bileşene değil. Bu etkileşimlerin ayrıntılı dökümünü Bölüm 2.2.7’de inceleyeceğiz.

Tasarım İlkesi 2: İyi tasarım, iyi ödünleşme gerektirir

Mühendislikte bedava kazanım yoktur: bir özelliği iyileştirmek, neredeyse her zaman bir başka özellikten ödün vermeyi gerektirir. Saat sıklığını artırmak güç tüketimini yükseltir; işlem hacmini artırmak gecikmeyi büyütebilir; güç tüketimini düşürmek başarımlı sınırlar; yonga alanını büyütmek maliyeti artırır. İyi bir mimar, bu ödünleşmelerin farkında olarak hedef iş yüküne en uygun dengeyi kurar. Örneğin masaüstü bilgisayar tasarımcısı düşük gecikmeye, sunucu tasarımcısı yüksek işlem hacmine, gömülü sistem tasarımcısı ise düşük güç tüketimine öncelik verir; her biri farklı bir ödünleşme tercihidir.

Örnek 2.2

100 buyruktan oluşan `Persone1.java` programının kullandığı buyruk türleri ve bu buyruk türlerinin program içindeki oranı tabloda verilmiştir. Bu program işlemcisi 200 MHz olan bir bilgisayarda kaç saniyede yürütülür? Tüm program için BBÇ nedir?

Buyruk türü	Kullanım yüzdesi	Gereken Çevrim Sayısı
K	%38	1
L	%15	3
M	%42	4
N	%5	5

Çözüm: Programda K türü buyruklardan 38, L'den 15, M'den 42, N'den 5 adet bulunmaktadır.

$$BBÇ = \frac{38 \times 1 + 15 \times 3 + 42 \times 4 + 5 \times 5}{100} = \frac{276}{100} = 2,76$$

$$\text{Yürütme zamanı} = \frac{100 \times 2,76}{200 \times 10^6} = 1,38 \times 10^{-6} \text{ sn} = 1,38 \mu\text{sn}$$

Örnek 2.3

Aynı program bu kez farklı bir mimariye sahip, 300 MHz'lik saat sıklığı olan işlemcili bir bilgisayarda çalıştırılmak istenmiştir.

Buyruk türü	Kullanım yüzdesi	Gereken Çevrim Sayısı
K	%38	2
L	%15	3
M	%42	8
N	%5	4

Çözüm:

$$\text{Yürütme zamanı} = \frac{38 \times 2 + 15 \times 3 + 42 \times 8 + 5 \times 4}{300 \times 10^6} = \frac{477}{300 \times 10^6} = 1,59 \mu\text{sn}$$

Personel.java programı Örnek 2.2'deki bilgisayarda $1,38 \mu\text{sn}$ 'de, bu bilgisayarda ise $1,59 \mu\text{sn}$ 'de çalışmıştır. Saat sıklığının artması tek başına değerlendirilerek bilgisayar başarımının artacağı öngörülemez.

2.2.6 İşlem Hacmi ve Gecikme

Bilgisayar başarımı iki farklı açıdan değerlendirilir:

- **Gecikme** (latency): Tek bir görevin başlangıcından bitişine kadar geçen süre. “Bu program ne kadar sürede çalışır?” sorusunun yanıtıdır. Yürütme zamanı gecikmedir.
- **İşlem hacmi** (throughput): Birim zamanda tamamlanan görev sayısı. “Bu sunucu saniyede kaç isteğe yanıt verir?” sorusunun yanıtıdır.

Terim Kökenleri

işlem hacmi (*throughput*): İngilizce *throughput* sözcüğü “birim zamanda sisteme giren ve çıkan iş miktarı” anlamına gelir. Türkçe kaynaklarda bu kavram zaman zaman *verim* olarak çevrilse de “verim” sözcüğü (*yield*) yonga üretiminde hatasız ürün oranını, günlük dilde ise *verimlilik* (*efficiency*) kavramını çağrıştırdığından karışıklığa yol açabilir. Bu kitapta throughput karşılığı olarak *işlem hacmi* tercih edilmiştir.

Bu iki kavram her zaman koşut hareket etmez. Bir boru hattı tasarımı, tek bir buyruğun gecikmesini artırabilir ama birim zamanda tamamlanan buyruk sayısını (işlem hacmini) yükseltir. Benzer şekilde, çok çekirdekli bir işlemci tek bir programı hızlandıramayabilir ama birden fazla programı eş zamanlı çalıştırarak işlem hacmini artırır.

Gecikme ve işlem hacmini somutlaştırmak için bir taşımacılık benzetmesi düşünelim. İstanbul'dan Ankara'ya giden bir otomobil 4 saatte varabilir (düşük gecikme) ama yalnızca 4 yolcu taşır. Bir otobüs 5 saat sürebilir (yüksek gecikme) ama 50 yolcu taşır. Saatlik yolcu işlem hacmi açısından otobüs çok daha üstündür ($50/5 = 10$ yolcu/saat karşısında $4/4 = 1$ yolcu/saat). İşlemci tasarımında da benzer bir ödünleşme yaşanır: bir boru hattı buyruğun toplam geçiş süresini artırabilir ama birim zamanda tamamlanan buyruk sayısını önemli ölçüde yükseltir. Hangi ölçütün önemli olduğu (gecikme mi, işlem hacmi mi) kullanım senaryosuna göre değişir.

Terim Kökenleri

örün (*web*): İngilizce *web* sözcüğü “ağ, örülmüş yapı” anlamına gelir. Türkçe karşılığı olan *örün*, *örmek* fiilinden türetilmiştir ve ODTÜ Bilişim Terimleri Sözlüğü'nde yer almaktadır. Tıpkı İngilizcedeki *web* gibi “birbirine bağlı düğümlerden oluşan örüntü” kavramını yansıtır. Bu kitapta *web sunucusu* yerine *örün sunucusu*, *web tarayıcısı* yerine *örün tarayıcısı* kullanılmıştır.

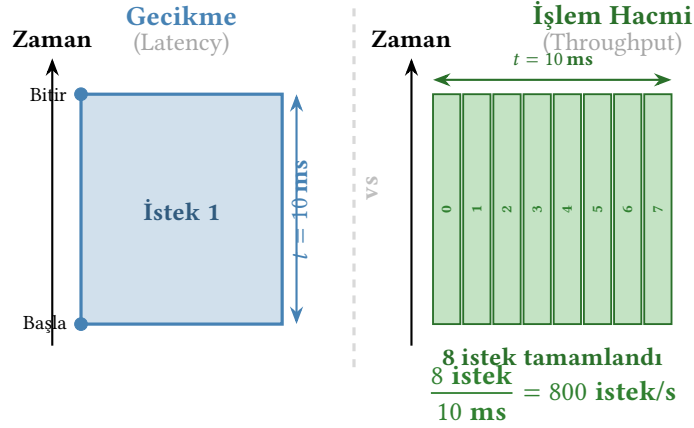
Örnek 2.4

Bir örün sunucusu her isteği 10 ms 'de yanıtlar (gecikme = 10 ms). Aynı anda en fazla 8 istek işleyebilir.

- Sunucunun işlem hacmi nedir?
- Her isteğin gecikmesini 20 ms 'ye çıkaran ama 32 eş zamanlı isteği destekleyen yeni bir sunucu tasarlanırsa işlem hacmi ne olur?

Çözüm:

- (a) İşlem hacmi = $\frac{8}{10 \text{ ms}} = 800 \text{ istek/s}$
- (b) İşlem hacmi = $\frac{32}{20 \text{ ms}} = 1600 \text{ istek/s}$. Gecikme 2 kat kötüleşse de işlem hacmi 2 kat artmıştır.



Gecikme: tek bir isteğin uçtan uca süresi. İşlem hacmi: birim zamanda tamamlanan istek sayısı.

Şekil 2.3: Gecikme ve işlem hacmi karşılaştırması. Gecikme tek bir isteğin ne kadar sürdüğünü, işlem hacmi ise birim zamanda kaç isteğin tamamlandığını ölçer. Aynı süre içinde tek bir isteği izlemek gecikmeyi, tüm istekleri saymak işlem hacmini verir.

2.2.7 Başarımı Etkileyen Etkenler

Başarım denklemini oluşturan üç temel bileşen (buyruk sayısı, BBÇ ve saat sıklığı) birbirinden bağımsız değildir ve her biri farklı tasarım katmanlarından etkilenir:

Tablo 2.3: Başarım bileşenlerini etkileyen tasarım katmanları

Tasarım katmanı	Buyruk sayısı	BBÇ	Saat sıklığı
Algoritma	Doğrudan	Dolaylı	—
Programlama dili	Dolaylı	Dolaylı	—
Derleyici	Doğrudan	Dolaylı	—
Buyruk kümesi mimarisi (BKM)	Doğrudan	Doğrudan	—
Mikromimari	—	Doğrudan	Doğrudan
Devre tasarımı	—	—	Doğrudan
Üretim teknolojisi	—	—	Doğrudan

Bu tablo önemli bir gerçeği vurgular: buyruk sayısını azaltma çabası (örneğin karmaşık buyruklar tanımlama) BBÇ'yi artırabilir; saat sıklığını yükseltme çabası ise boru hattı derinleştirme yoluyla BBÇ'yi artırabilir. Dolayısıyla herhangi bir bileşeni tek başına eniyilemeye çalışmak yerine *toplam yürütme zamanına* odaklanmak gerekir.

Bu etkileşimleri daha somut olarak inceleyelim:

- **Algoritma → buyruk sayısı:** Kabarcık sıralaması $O(n^2)$ buyruk üretirken hızlı sıralama $O(n \log n)$ buyruk üretir. Doğru algoritma seçimi, donanım eniyilemesinden çok daha büyük bir etki yaratabilir. 10 000 öğeyi sıralamak için kabarcık sıralaması ~100 milyon

karşılaştırma yaparken hızlı sıralama ~130 bin karşılaştırma yapar; yaklaşık 770 kat fark. Saat sıklığını 770 kat artırmak olanaksızdır, oysa algoritmayı değiştirmek bir satırlık iştir.

- **Derleyici → buyruk sayısı ve BBÇ:** Eniyileyici bir derleyici döngü açma (*loop unrolling*) ile buyruk sayısını artırabilir ama dallanma buyruklarını azaltarak BBÇ'yi düşürebilir. Aynı kaynak kodu farklı eniyileme düzeyleriyle derlenmesi (-00 ile -03 gibi) %30–300 başarımlık farkı yaratabilir.
- **BKM → buyruk sayısı ve BBÇ:** CISC mimariler az sayıda karmaşık buyrukla (düşük buyruk sayısı, yüksek BBÇ), RISC mimariler ise çok sayıda basit buyrukla (yüksek buyruk sayısı, düşük BBÇ) aynı işi yapar. Hangisinin daha hızlı olduğu *toplam çevrim sayısına* bağlıdır.
- **Mikromimari → BBÇ ve saat sıklığı:** Sırasız yürütüm mikromimarisi veri bağımlılıkları olan buyrukları yeniden sıralayarak BBÇ'yi düşürür ama gerektirdiği karmaşık kontrol mantığı kritik yolu uzatabilir ve saat sıklığını sınırlayabilir. Tersine, sıralı (*in-order*) bir tasarım daha yüksek saat sıklığına ulaşabilir ama BBÇ'si daha yüksektir.

Yürütme zamanı denklemi başarımların yazılım ve donanım tasarımı boyutunu özetler. Ancak gerçek dünyada başarımları etkileyen *fiziksel ve çevresel* etkenler de göz ardı edilemez:

- **Sıcaklık:** İşlemci sıcaklığı yükseldikçe transistörlerin anahtarlama hızı düşer ve sızıntı akımları artar. Çağdaş işlemciler belirli bir sıcaklık eşiğini aştığında saat sıklığını otomatik olarak düşürür (*termal kısma, thermal throttling*); bu mekanizma işlemciyi korur ama doğrudan yürütme zamanını uzatır. Yoğun iş yükünde dizüstü bilgisayarın masaüstüne göre yavaş kalmasının başlıca nedeni budur.
- **Gerilim dalgalanmaları ve geçici hatalar:** Güç kaynağındaki anlık gerilim düşüşleri veya kozmik ışınların neden olduğu bit hataları (*soft error*) hesaplama sonuçlarını bozabilir. Uzay uygulamalarında radyasyon kaynaklı hatalar özellikle can alıcıdır; bu nedenle uydu ve uzay aracı işlemcileri hata düzeltme kodları ve yedekli hesaplama ile donatılır; genellikle dünya üzerindeki muadillerinden çok daha düşük saat sıklığında çalışsalar da.
- **Güvenilirlik ve dayanıklılık:** Bir sistemin başarımlı yalnızca hızıyla değil, ne kadar süre kesintisiz çalışabildiğiyle de ölçülür. Bir sunucu saatte milyarlarca buyruk yürütebilir ama haftada bir çökerse gerçek başarımlı sıfıra yakındır. Bölüm 1.8'te tanımlanan MTTF, MTBF ve kullanılabilirlik kavramları, bu etkeni nicel olarak değerlendirmeyi sağlar.

Bu nedenle bir bilgisayar mimarı, her tasarım kararının yürütme zamanı denkleminin üç bileşeni üzerindeki net etkisini değerlendirmenin yanı sıra güç tüketimi, sıcaklık sınırları ve güvenilirlik gereksinimlerini de gözlemelidir.

2.2.8 MIPS ve MFLOPS

Yürütme zamanı güvenilir bir başarımlı ölçütüdür, ancak hesaplanması buyruk sayısı, BBÇ ve saat sıklığı bilgisini gerektirir. Endüstri tarihinde bu karmaşıklıktan kaçınarak tek bir rakamla başarımlı özetleme isteği, zaman zaman yanıltıcı pazarlama uygulamalarına yol açmıştır. Bu basitleştirilmiş ölçütlerin en bilineni MIPS'tir; öyle ki mühendisler arasında kısaltma zamanla *Meaningless Indicator of Processor Speed* ("İşlemci Hızının Anlamsız Göstergesi") şeklinde alaycı bir ters açılıma dönüşmüştür.

Saniye Başına İşlenen Milyon Buyruk (*Million Instructions Per Second*, MIPS):

$$\text{MIPS} = \frac{\text{Buyruk sayısı}}{\text{Yürütme zamanı} \times 10^6} \quad (2.10)$$

Terim Kökenleri

MIPS (*million instructions per second*): MIPS kısaltması bilgisayar dünyasında iki farklı anlam taşır. Bu bölümde ele aldığımız *Million Instructions Per Second* (Saniye Başına Milyon Buyruk) bir başarımlı ölçütüdür. Ancak MIPS aynı zamanda Stanford Üniversitesi'nde John Hennessy'nin 1981'de tasarladığı RISC işlemci mimarisinin de adıdır: *Microprocessor without Interlocked Pipeline Stages* (Kilitlenmesiz Boru Hattı Aşamalarına Sahip Mikroişlemci). Hennessy ve ekibinin bu adı seçmesi bir rastlantı değildir: ölçüt olarak MIPS'in "anlamsız" olduğu tartışmaları zaten sürmekteydi ve bu isim hem teknik bir özelliği (boru hattı kilitlemelerinin yazılıma bırakılmasını) hem de alaycı bir göndermeyi barındırıyordu. MIPS mimarisi ilerleyen yıllarda gömülü sistemlerde ve oyun konsollarında (PlayStation 1 ve 2 gibi) yaygın biçimde kullanıldı. Bu kitapta bağlamdan anlaşılmadığı durumlarda ölçüt için "MIPS değeri", mimari için "MIPS mimarisi" ifadesi tercih edilecektir.

Bir saniyede işlenen buyruk sayısı ilk bakışta mantıklı bir başarımlı ölçütü gibi görünür; ancak farklı buyruk kümeleri farklı karmaşıklıkta buyruklar içerdiğinden MIPS değeri yanıltıcı sonuçlar verebilir. Aşağıdaki örnekte bu durum somut olarak gösterilmektedir.

Örnek 2.5

Ayşegül ve Demre'nin bilgisayarlarındaki işlemciler aynıdır ve saat sıklıkları 200 MHz'dir. MIPS değerlerini karşılaştırın.

Ayşegül'ün bilgisayarı			Demre'nin bilgisayarı		
Buyruk	Sayı (M)	BBC	Buyruk	Sayı (M)	BBC
K	8	2	K	10	2
L	4	3	L	8	3
M	2	8	M	2	8
N	4	4	N	4	4

Çözüm:

Ayşegül: toplam buyruk: $8+4+2+4 = 18$ M; toplam çevrim: $8 \times 2 + 4 \times 3 + 2 \times 8 + 4 \times 4 = 60$ M çevrim.

$$t_A = \frac{60 \times 10^6}{200 \times 10^6} = 0,3 \text{ s}, \quad \text{MIPS}_A = \frac{18}{0,3} = 60,0$$

Demre: toplam buyruk: $10+8+2+4 = 24$ M; toplam çevrim: $10 \times 2 + 8 \times 3 + 2 \times 8 + 4 \times 4 = 76$ M çevrim.

$$t_D = \frac{76 \times 10^6}{200 \times 10^6} = 0,38 \text{ s}, \quad \text{MIPS}_D = \frac{24}{0,38} = 63,16$$

Demre'nin yürütme zamanı daha uzundur ($0,38 \text{ s} > 0,30 \text{ s}$); dolayısıyla **başarımı daha düşüktür**. Ancak MIPS değeri daha yüksektir ($63,16 > 60,0$) çünkü Demre'nin programı daha fazla buyruk içermektedir. Bu örnek, MIPS'in güvenilir bir başarımlı ölçütü olmadığını gösterir.

Bilgi Notu 2.4: Pazarlama Savaşlarında Başarım Ölçütleri

Başarım ölçütlerinin pazarlama aracına dönüşmesinin uzun bir tarihi vardır. DEC, 1977’de VAX-11/780 işlemcisini “1 MIPS makine” diye tanıttı; oysa gerçekte saniyede yaklaşık 500 000 buyruk yürütüyordu. Tartışma öyle büyüdü ki DEC sonunda kendi “VAX MIPS” birimini tanımlamak zorunda kaldı.

Benzer bir pazarlama savaşı kayan nokta başarımında da yaşandı. Apple, 1999’da Power Mac G4’ü piyasaya sürerken PowerPC G4 işlemcisinin 1 gigaflop’u aşmasını öne çıkardı ve bilgisayarı “kişisel süper bilgisayar” olarak pazarladı. Steve Jobs, Soğuk Savaş döneminden kalma ABD ihracat yasalarına göre bu eşiği aşan bilgisayarların “mühimmat” sayıldığını hatırlattı; bir televizyon reklamında bilgisayarın etrafı tanklarla sarıldı, Intel tabanlı PC’ler ise “zararsız” ilan edildi. Gerçekte aynı dönemde Intel işlemciler pek çok iş yükünde benzer ya da daha iyi başarım gösteriyordu; fark, hangi ölçütün seçildiğindeydi. Nitekim Apple, 2005’te Intel’e geçerek bu başarım yarışını uygulamada kaybettiğini kabul etti.

Saniye başına milyon kayan nokta işlemi, *MFLOPS* (Million Floating Point Operations per Second) olarak kısaltılır ve yalnızca kayan nokta (floating point) işlemleri yapan buyrukların sayısını hesaba katar. Değişik bilgisayarlarda kayan nokta işlemleri farklılık gösterdiği için MFLOPS da tutarlı bir başarım ölçütü değildir.

Terim Kökenleri

kayan nokta (*floating point*): İngilizce *floating point* terimi, ondalık ayırıcının sayı içinde “kayabildiğini” ifade eder. Türkçede ondalık ayırıcı olarak nokta değil *virgül* kullanıldığından (3,14 gibi), aslında bu terime “kayan virgül” demek dilimize daha uygun olurdu. Nitekim bazı Türkçe kaynaklarda *kayan virgüllü sayı* ifadesine rastlanır. Ne var ki *floating point* karşılığı olarak *kayan nokta* kullanımı hem uluslararası literatürle tutarlılık hem de yaygınlık açısından bir öz ad haline gelmiştir. Bu kitapta biz de *kayan nokta* ifadesini kullanıyoruz.

2.2.9 FLOPS ve Türevleri

Bir önceki alt bölümde MFLOPS’un belirli bir programın başarımını ölçmekte güvenilir olmadığını gördük: farklı mimarilerdeki kayan nokta buyrukları farklı karmaşıklıkta olduğundan, program düzeyinde sayılan “işlem sayısı” yanıltıcı oluyordu. Ancak FLOPS kavramı burada farklı bir bağlamda karşımıza çıkar: belirli bir programın kaç kayan nokta işlemi yürüttüğünü ölçmek yerine, *donanımın kuramsal üst kapasitesini* tanımlamak amacıyla kullanılır. Bu ayırım can alıcıdır.

FLOPS (Floating Point Operations Per Second, Saniye Başına Kayan Nokta İşlemi), bir sistemin birim zamanda gerçekleştirebildiği en yüksek kayan nokta aritmetik işlem sayısını ifade eder:

$$\text{FLOPS} = \text{Çekirdek sayısı} \times \text{Saat sıklığı} \times \text{Çekirdek başına çevrim başına KN işlemi} \quad (2.11)$$

Günümüz sistemlerinin kapasitesi milyonlarca FLOPS’un çok ötesinde olduğu için SI ön ekleriyle ifade edilir:

Kısaltma	Değer	Büyüklik	Örnek Sistem
MFLOPS	10^6	Milyon	1980'ler süper bilgisayarları
GFLOPS	10^9	Milyar	Masaüstü MİB
TFLOPS	10^{12}	Trilyon	Oyun GPU'su (NVIDIA RTX 4090: ~83 TFLOPS FP32)
PFLOPS	10^{15}	Katrilyon	Süper bilgisayar (Summit, 2018: ~149 PFLOPS FP64)
EFLOPS	10^{18}	Kentilyon	Ekzaölçek süper bilgisayarlar

Kayan nokta işlemleri farklı duyarlık düzeylerinde gerçekleştirilebilir. *Çift duyarlık* (FP64, 64 bit), bilimsel hesaplamada altın standart iken yapay zeka eğitiminde genellikle *tek duyarlık* (FP32) veya *yarı duyarlık* (FP16/BF16) kullanılır. Bazı çıkarım iş yükleri INT8 hatta INT4 gibi düşük duyarlıklı tam sayı biçimlerini bile tercih eder. Bu nedenle bir FLOPS değeri karşılaştırılırken hangi duyarlık düzeyinde ölçüldüğü mutlaka belirtilmeli.

Bilgi Notu 2.5: Tepe Başarım ve Gerçek Başarım

Bir işlemcinin katalog değeri olarak verilen FLOPS rakamı *tepe başarım* (peak performance) olup, tüm birimlerin her çevrimde tam kapasiteyle çalıştığı kuram düzeyinde bir üst sınırdır. Gerçek uygulamalarda bellek darboğazları, dal yanlış kestirimleri ve veri bağımlılıkları nedeniyle genellikle tepe başarımın %10–60'ı arasında bir *sürdürülebilir başarım* (sustained performance) elde edilir. Bir sistemin gerçek kapasitesini anlamak için LINPACK gibi karşılaştırma ölçütlerindeki sürdürülebilir değerlere bakmak gerekir.

Örnek 2.6

Bir GPU'nun teknik özelliklerinde 5120 CUDA çekirdeği, 1,5 GHz saat sıklığı ve her çekirdeğin çevrim başına 2 tek duyarlıklı (FP32) kayan nokta işlemi yapabildiği belirtilmiştir. Bu GPU'nun FP32 tepe başarımını TFLOPS cinsinden hesaplayın.

Çözüm:

$$\text{FLOPS}_{\text{FP32}} = 5120 \times 1,5 \times 10^9 \times 2 = 15,36 \times 10^{12} = 15,36 \text{ TFLOPS}$$

Bilgi Notu 2.6: Skynet: 60 Teraflop Yeter mi?

2003 yapımı *Terminator 3: Rise of the Machines* filminde insanlığı tehdit eden yapay zekâ Skynet'in işlem gücü 60 TFLOPS olarak belirtilir; o dönem için korku verici bir rakamdır. Bugünün gözüyle bakıldığında tablo oldukça farklıdır: tek bir NVIDIA H100 GPU, FP8 duyarlığında yaklaşık 4 PFLOPS tepe başarımına ulaşır; Skynet'in yaklaşık 67 katı. Hatta bir oyun konsolu olan PlayStation 5 bile ~10 TFLOPS FP32 başarımıyla Skynet'in altıda birini tek başına karşılar. Bu karşılaştırma, Moore Yasası'nın 20 yılda hesaplama kapasitesini ne denli ileriye taşıdığının çarpıcı bir göstergesidir.

2.2.10 Yapay Zeka İş Yüklerinde Başarım

Yapay zeka ve özellikle derin öğrenme iş yükleri, geleneksel MİB uygulamalarından temelden farklı başarım gereksinimleri ortaya koyar. Bir sinir ağının eğitimi sırasında milyarlarca çarpma-toplama işlemi tekrar tekrar gerçekleştirilir; bu işlemler büyük ölçüde koşut ve düzenli veri erişim örüntüsüne sahiptir.

Eğitim ve çıkarım başarımı

Yapay zeka iş yüklerinde iki ayrı aşama bulunur. *Eğitim* (training) aşamasında model parametreleri büyük veri kümeleri üzerinde yinelenerek güncellenir; bu aşamada yüksek FLOPS kapasitesi ve geniş bellek bant genişliği gereklidir. *Çıkarım* (inference) aşamasında ise eğitilmiş model yeni girdilere yanıt üretir; burada gecikme süresi (latency) ve enerji verimliliği (işlem/watt) öne çıkar.

Hesaplama yoğunluğu

Yapay zeka modellerinin hesaplama gereksinimi astronomik boyutlara ulaşmıştır. GPT-3 düzeyinde bir büyük dil modelinin eğitimi yaklaşık $3,14 \times 10^{23}$ kayan nokta işlemi (FLOP) gerektirir. Bu büyüklük, tek bir GPU'nun bile yıllar boyunca hesaplama yapmasını gerektireceğinden binlerce hızlandırıcıdan oluşan kümeler kullanılır.

Terim Kökenleri

belirteç (*token*): *Belirtmek* fiilinden *-eç* ekiyle. Yapay zeka ve doğal dil işleme alanlarında bir metnin işlenebilir en küçük parçasına verilen addır. Bir belirteç bir sözcük, sözcüğün bir parçası (alt sözcük) veya bir noktalama işareti olabilir. Büyük dil modelleri girdi metnini belirteçlere ayırarak işler ve çıktıyı yine belirteç belirteç üretir. Modelin kapasitesi ve hızı genellikle *belirteç/saniye* birimiyle ölçülür. İngilizce *token* sözcüğü “simge, jeton” anlamlarına gelir; bazı Türkçe kaynaklarda *jeton* çevirisi de kullanılır. Bu kitapta Türkçeye daha uygun olan **belirteç** tercih edilmiştir.

Başarım ölçütleri

Yapay zeka alanında FLOPS'un yanı sıra şu ölçütler de yaygın biçimde kullanılır:

- **Belirteç/saniye** (tokens/second): Büyük dil modellerinin çıkarım hızı.
- **Görüntü/saniye** (images/second): Görüntü sınıflandırma modellerinin işlem hacmi ölçüsü.
- **FLOPS/watt**: Enerji verimliliği; veri merkezleri için can alıcı.
- **MLPerf**: SPEC'in yapay zeka karşılığı olarak geliştirilen standart karşılaştırma ölçütü kümesi.

Bilgi Notu 2.7: MLPerf: Yapay Zekanın SPEC'i

MLCommons tarafından yönetilen *MLPerf*, yapay zeka donanımlarını standart iş yükleriyle karşılaştırmak amacıyla oluşturulmuştur. Eğitim (Training) ve çıkarım (Inference) olmak üzere iki ayrı sınama kümesi bulunur. Görüntü sınıflandırma (ResNet-50), nesne algılama (RetinaNet), doğal dil işleme (BERT) ve büyük dil modeli (GPT) gibi çeşitli iş yüklerini kapsar. Sonuçlar hem kapalı (closed: sabit model ve hiperparametre) hem de açık (open: serbest eniyileme) kategorilerde yayımlanır.

2.2.11 Karşılaştırmalı Başarım Ölçümü

Bir bilgisayarın başarımını ölçmek isteyen kişinin ilk sorusu şudur: *neyi ölçeceğim?* Bu sorunun yanıtı bakış açısına göre değişir.

Kullanıcı açısından en güvenilir ölçüt, kendi gerçek iş yüküdür. Amacınız bir bilgisayar oyunu oynamaksa, SPEC skorlarına ya da MIPS değerlerine bakmak sizi yanıltabilir; tek güvenilir yol, aday bilgisayarlarda o oyunu doğrudan çalıştırıp kare hızını karşılaştırmaktır. Hiçbir sınama programı sizin programınızın yerine geçemez; çünkü her programın bellek erişim örüntüsü, dal davranışı ve hesaplama yoğunluğu kendine özgüdür.

Tasarımcı açısından ise durum farklıdır. Bir işlemci tasarımcısı hangi kullanıcının hangi programı çalıştıracağını önceden bilemez; yeni bir mikromimari önerisinin geniş bir uygulama yelpazesinde nasıl davranacağını anlaması gerekir. Bu nedenle tasarımcılar çok sayıda temsili programdan oluşan *sınama kümeleri* kullanır. Büyük işlemci firmalarında yüzlerce gerçek uygulamadan alınan program kesitleri (*trace*) ile testler yapılır; yalnızca standart sınama kümeleriyle sınırlı kalınmaz. Yine de farklı uygulamalar birbirinden çok farklı sonuçlar üretir; tek bir programa ya da tek bir ölçüte bakarak “bu işlemci daha hızlı” demek yanıltıcıdır.

İster kullanıcı ister tasarımcı olalım, birden fazla programın sonuçlarını tek bir sayıda özetlemek gerektiğinde iki temel yöntem kullanılır: *ağırlıklı aritmetik ortalama* ve *geometrik ortalama*.

Ağırlıklı aritmetik ortalama

n adet sınama programının yürütme zamanları $T_1, T_2, \dots, T_n$ ve her programın iş yükündeki önem derecesini yansıtan ağırlıkları $a_1, a_2, \dots, a_n$ olmak üzere ağırlıklı aritmetik ortalama şöyle tanımlanır:

$$\bar{T}_{\text{aritmetik}} = \frac{\sum_{i=1}^n a_i \times T_i}{\sum_{i=1}^n a_i} \quad (2.12)$$

Tüm ağırlıklar eşitse ($a_i = 1$) bu formül basit aritmetik ortalamaya dönüşür. Ağırlıklı aritmetik ortalama, yürütme zamanı gibi *mutlak büyüklükleri* özetlemek için uygundur; ancak hızlanma oranlarını özetlerken yanıltıcı sonuçlar verebilir.

Geometrik ortalama

n adet sınama programının hızlanma oranları $H_1, H_2, \dots, H_n$ olmak üzere geometrik ortalama şöyle hesaplanır:

$$\bar{H}_{\text{geometrik}} = \left(\prod_{i=1}^n H_i \right)^{1/n} = (H_1 \times H_2 \times \dots \times H_n)^{1/n} \quad (2.13)$$

Geometrik ortalama, *oran* ve *hızlanma* gibi çarpımsal büyüklükleri özetlemek için daha uygundur. En önemli avantajı, aşırı yüksek veya düşük tek bir değer sonucunu baskılamamasıdır. Bu nedenle SPEC gibi endüstri standartları geometrik ortalamayı tercih eder.

Aritmetik mi, geometrik mi?

Aşağıdaki basit örnek iki yöntem arasındaki farkı somutlaştırır.

Örnek 2.7

Bir referans işlemciye göre B işlemcisinin beş sınam programındaki hızlanma oranları aşağıda verilmiştir. İki durum için aritmetik ve geometrik ortalamaları hesaplayın ve karşılaştırın.

	Durum 1	Durum 2
Program	B'nin hızlanması	B'nin hızlanması
Program 1	1×	1×
Program 2	1×	1×
Program 3	1×	1×
Program 4	1×	1×
Program 5	10×	100×

Çözüm:

Durum 1: Beş programın dördünde hiç hızlanma yok; yalnızca birinde 10× hızlanma var.

$$\text{Aritmetik} = \frac{1 + 1 + 1 + 1 + 10}{5} = \frac{14}{5} = 2,8\times$$

$$\text{Geometrik} = (1 \times 1 \times 1 \times 1 \times 10)^{1/5} = 10^{0,2} \approx 1,58\times$$

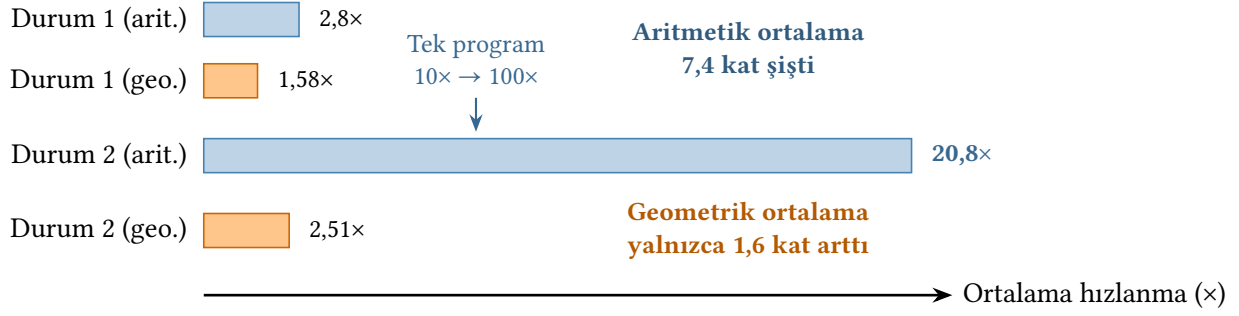
Durum 2: Tek fark, Program 5'teki hızlanmanın 100×'e çıkmasıdır.

$$\text{Aritmetik} = \frac{1 + 1 + 1 + 1 + 100}{5} = \frac{104}{5} = 20,8\times$$

$$\text{Geometrik} = (1 \times 1 \times 1 \times 1 \times 100)^{1/5} = 100^{0,2} \approx 2,51\times$$

Tek bir programdaki hızlanma 10×'ten 100×'e çıktığında aritmetik ortalama 2,8'den 20,8'e fırladı; yaklaşık 7,4 kat artış. Oysa geometrik ortalama yalnızca 1,58'den 2,51'e yükseldi (1,6 kat artış). İşlemci gerçekte beş programın dördünde hiçbir iyileşme sağlamıyor; aritmetik ortalama bu gerçeği gizlerken geometrik ortalama daha dürüst bir tablo sunuyor.

Bu örneğin gösterdiği ilke şudur: aritmetik ortalama, tek bir aşırı değer genel sonucu baskılamasına izin verir; geometrik ortalama ise tüm programlardaki başarıyı daha dengeli biçimde yansıtır. Bu nedenle başarımların karşılaştırmalarında, özellikle hızlanma oranları söz konusu olduğunda, geometrik ortalama tercih edilir.



Şekil 2.4: Aritmetik ve geometrik ortalama karşılaştırması: beş programın yalnızca birindeki uç hızlanma değeri aritmetik ortalamayı orantısız biçimde yükseltirken geometrik ortalama dengeli kalır.

Örnek 2.8

Kasırğa 5 işlemcisiyle Kasırğa 3 işlemcisinin başarımı 9 yapay sına programıyla karşılaştırılmıştır. Her programın iki işlemcideki yürütme zamanı aşağıda verilmiştir.

Program	Kasırğa 5 (sn)	Kasırğa 3 (sn)
perlbench	640	660
sjeng	120	200
gobmk	500	508
gcc	90	1200
libquantum	48	920
hmm	94	1400
art	860	320
milc	302	230
facerec	164	48

- Her program için Kasırğa 5'in Kasırğa 3'e göre hızlanmasını hesaplayın.
- Hızlanma değerlerinin aritmetik ve geometrik ortalamasını bulun.
- İki ortalama arasındaki farkın nedenini açıklayın. Kasırğa 5 gerçekten daha iyi bir işlemci midir?

Çözüm:

(a) Her program için hızlanma = $T_{\text{Kasırğa 3}} / T_{\text{Kasırğa 5}}$:

Program	Kasırğa 5 (sn)	Kasırğa 3 (sn)	Hızlanma
perlbench	640	660	$660/640 = 1,03$
sjeng	120	200	$200/120 = 1,67$
gobmk	500	508	$508/500 = 1,02$
gcc	90	1200	$1200/90 = 13,33$
libquantum	48	920	$920/48 = 19,17$
hmm	94	1400	$1400/94 = 14,89$
art	860	320	$320/860 = 0,37$
milc	302	230	$230/302 = 0,76$
facerec	164	48	$48/164 = 0,29$

Dikkat çekici bir nokta: art, milc ve facerac programlarında hızlanma 1'in altındadır; bu, Kasırğa 5'in bu programlarda Kasırğa 3'ten *daha yavaş* olduğu anlamına gelir.

(b) Aritmetik ve geometrik ortalamalar:

$$\bar{H}_{\text{aritmetik}} = \frac{1,03 + 1,67 + 1,02 + 13,33 + 19,17 + 14,89 + 0,37 + 0,76 + 0,29}{9} = \frac{52,53}{9} \approx 5,84\times$$

$$\bar{H}_{\text{geometrik}} = (1,03 \times 1,67 \times 1,02 \times 13,33 \times 19,17 \times 14,89 \times 0,37 \times 0,76 \times 0,29)^{1/9} \approx 2,02\times$$

(c) Aritmetik ortalama (5,84×) iyimser bir tablo çizer çünkü gcc, libquantum ve hmer'daki uç değerler (13–19×) toplamı yukarı çeker. Geometrik ortalama (2,02×) ise hem çok yüksek hem çok düşük değerlerin etkisini dengeler ve daha gerçekçi bir resim sunar. Kasırğa 5 “kesinlikle daha iyi” denilebilir mi? Hayır; üç programda daha yavaştır. Aritmetik ortalamaya bakarak “6 kat hızlı” demek yanıltıcıdır. Geometrik ortalamanın gösterdiği 2× bile tüm iş yüklerinde tutarlı bir üstünlük anlamına gelmez; yalnızca *ortalama eğilimi* yansıtır. İş yüküne bağlı olarak Kasırğa 3 daha iyi bir seçim bile olabilir.

2.3 Standart Sınama Programı Kümeleri

Bir önceki bölümde gördüğümüz gibi, işlemci tasarımcıları kendi önerilerini geniş bir uygulama yelpazesinde değerlendirmek zorundadır. Ancak 1980'lerin sonuna kadar her üretici kendi seçtiği programlarla başarımlarını ölçüyor ve sonuçları kendi lehine yorumluyordu. Bir firma “işlemcimiz rakipten 3 kat hızlı” derken başka bir firma tam tersi bir iddiada bulunabiliyordu; çünkü her biri farklı programlar, farklı derleyici ayarları ve farklı ölçütler kullanıyordu. Bu durum hem tüketicilerin adil karşılaştırma yapmasını hem de araştırmacıların sonuçları tekrarlamasını olanaksız kılıyordu.

Bu sorunu çözmek için endüstri ortak bir standart oluşturma yoluna gitti. *Yapay sınama programları (benchmark)*, gerçek dünya uygulamalarını temsil eden ve herkesin aynı koşullarda çalıştırabileceği standart iş yükleridir; farklı işlemcilerin başarımlarını tekrarlanabilir ve adil koşullar altında karşılaştırmaya olanak tanır. Günümüzde farklı uygulama alanlarına yönelik birçok sınama programı kümesi bulunur; bunların en eskisi ve işlemci başarımları için en yaygın kabul göreni **SPEC**'tir.

2.3.1 SPEC

SPEC (Standard Performance Evaluation Corporation), 1988 yılında Apollo Computer, Hewlett-Packard, MIPS Computer Systems ve Sun Microsystems tarafından kâr amacı gütmeyen bir konsorsiyum olarak kurulmuştur. Bu dört firma, dönemin önde gelen iş istasyonu üreticileriydi ve özellikle RISC tabanlı sistemlerin başarımlarını nesnel biçimde ölçmek istiyorlardı. Bugün 120'den fazla donanım ve yazılım firması, üniversite ve araştırma kuruluşunu bünyesinde barındıran SPEC, kuruluşundan bu yana altı nesil işlemci sınama programı kümesi yayımlamıştır (Tablo 2.4).

Tablo 2.4: *SPEC CPU sinama programı kümelerinin evrimi*

Küme	Yıl	Tamsayı	KN	Öne çıkan yenilik
SPEC89	1989	4	6	İlk sürüm; referans makine: VAX 11/780
SPEC92	1992	6	14	<i>SPECrate</i> ve “base/peak” ölçümleri
SPEC95	1995	8	10	Daha büyük veri kümeleri; L1 önbelleğe sığmayan iş yükleri
SPEC CPU2000	2000	12	14	Referans skorları 100 olarak belirlendi
SPEC CPU2006	2006	12	17	Daha kapsamlı ve çeşitli iş yükleri
SPEC CPU 2017	2017	10	13	OpenMP çoklu iş parçacığı desteği

Güncel sürüm olan **SPEC CPU 2017** [6], dört ayrı sinama kümesinden oluşur: *SPECspeed* (tek görev tamamlama süresi) ve *SPECrate* (birim zamanda tamamlanan görev sayısı) ayrımı hem tamsayı hem de kayan nokta için yapılmaktadır. SPEC 2006’ya kıyasla ortalama iş yükü büyüklüğü yaklaşık 10 kat artmış, yapay zeka, video sıkıştırma, 3B görüntüleme ve okyanus modelleme gibi yeni uygulama alanları eklenmiştir.

Tablo 2.5: *SPECint 2017 Tamsayı Sinama Programları*

Program	Dil	Açıklama
perlbench_r	C	Perl yorumlayıcısı; metin işleme
gcc_r	C	GNU C derleyicisi
mcf_r	C	Minimum maliyetli ağ akışı; araç rotalama
omnetpp_r	C++	Ayrık olay ağ benzetimi
xalancbmk_r	C++	XML/XSLT dönüşümü
x264_r	C	H.264/AVC video sıkıştırma
deepsjeng_r	C++	Yapay zeka; satranç motoru
leela_r	C++	Yapay zeka; Go oyunu (Monte Carlo ağaç arama)
exchange2_r	Fortran	Yapay zeka; Sudoku çözücüsü
xz_r	C	LZMA veri sıkıştırma

Tablo 2.6: SPECfp 2017 Kayan Nokta Sinama Programları

Program	Dil	Açıklama
bwaves_r	Fortran	Patlama dalgası benzetimi
cactuBSSN_r	C++/C/Fortran	Genel görelilik (Einstein denklemleri)
namd_r	C++	Moleküler dinamik benzetimi
parest_r	C++	Biyomedikal görüntüleme; sonlu elemanlar
povray_r	C++/C	Işın izleme (ray tracing)
lbm_r	C	Kafes Boltzmann akışkanlar dinamiği
wrf_r	Fortran/C	Hava tahmini modeli
blender_r	C++/C	3B görüntüleme ve canlandırma
cam4_r	Fortran/C	Atmosfer modelleme (iklim)
imagick_r	C	Görüntü işleme (ImageMagick)
nab_r	C	Moleküler dinamik; nükleik asit modelleme
fotonik3d_r	Fortran	Hesaplamalı elektromanyetik (FDTD)
roms_r	Fortran	Bölgesel okyanus modelleme

Bilgi Notu 2.8: SPEC 2006'dan SPEC 2017'ye Geçiş

SPEC CPU 2017 [6], sefine [16] göre önemli değişiklikler içerir. *SPECspeed* kümesinde iş parçacığı düzeyinde koşutluk (OpenMP) ilk kez desteklenerek çok çekirdekli işlemcilerin avantajı ölçülebilir hale gelmiştir. Bellek gereksinimi ortalama 4–6 kat artmıştır. Yapay zeka ve oyun motoru gibi yeni program türleri eklenmiştir. SPEC 2006 ile SPEC 2017 sonuçları doğrudan karşılaştırılamaz; farklı iş yükleri ve ölçüm yöntemleri kullanıldığından her biri kendi referans çerçevesinde değerlendirilmelidir.

2.3.2 SPEC Puanı Nasıl Hesaplanır?

Bir SPEC kümesinde 10–23 arasında program bulunur ve her biri farklı bir uygulama alanını temsil eder. Peki bir işlemcinin tüm bu programlardaki başarımını nasıl tek bir sayıda özetleyebiliriz? Çiğ yürütme zamanlarının ortalamasını almak işe yaramaz: programlar arasında devasa süre farkları vardır; biri 10 saniyede biterken diğeri 3 saat sürebilir. Böyle bir ortalama en uzun süren program sonucu baskılar ve kısa programlardaki farklılıklar görünmez olur.

SPEC bu sorunu iki adımda çözer. Birincisi, her programın yürütme zamanını bir *referans makinenin* süresine bölerek *normalize* eder. Böylece tüm programlar “referans makineye göre kaç kat hızlı?” sorusuna yanıt veren birimsiz oranlara dönüşür ve saniye–saat ölçek farkı ortadan kalkar. İkincisi, bu oranları Bölüm 2.2.11’de tartıştığımız *geometrik ortalama* ile birleştirir; böylece tek bir uç değerin sonucu baskılaması önlenir.

Formülleştirelim: SPEC CPU 2017’de referans makine (sabit bir eski sistem) her program i için bir referans süre $T_{\text{ref},i}$ tanımlar. Sinanan makinenin aynı program için ölçülen süresi $T_{\text{test},i}$ olsun. Her programın bireysel SPEC oranı:

$$r_i = \frac{T_{\text{ref},i}}{T_{\text{test},i}} \quad (2.14)$$

$r_i > 1$ ise sınanan makine referanstan hızlı, $r_i < 1$ ise yavaş demektir. Nihai SPEC puanı, tüm bireysel oranların geometrik ortalamasıdır:

$$\text{SPEC puanı} = \left(\prod_{i=1}^n r_i \right)^{1/n} \quad (2.15)$$

Geometrik ortalamanın burada tercih edilmesinin bir nedeni daha vardır: iki işlemcinin geometrik ortalamalarının oranı, her zaman bireysel oranların geometrik ortalamasına eşittir. Başka bir deyişle, hangi makine referans seçilirse seçilsin, iki işlemci arasındaki göreceli sıralama değişmez.

Örnek 2.9

Fırtına ve Bora adlı iki işlemci, SPECint 2017 kümesinden seçilmiş beş programla sınanmıştır. Referans makinenin ve iki işlemcinin yürütme zamanları (saniye) aşağıda verilmiştir.

Program	Referans (sn)	Fırtına (sn)	Bora (sn)
gcc	8200	1640	2050
mcf	6800	1943	1360
deepsjeng	7400	1480	1850
leela	7000	2000	1167
xz	5600	800	1120

- Her program için iki işlemcinin SPEC oranlarını (r_i) hesaplayın.
- Her iki işlemcinin SPEC puanını bulun. Hangisi daha yüksek skora sahiptir?
- Sonucu yorumlayın: yüksek skorlu işlemci her programda daha mı hızlıdır?

Çözüm:

- (a) Her program için $r_i = T_{\text{ref},i} / T_{\text{test},i}$:

Program	Referans	Fırtına	r_i (Fırtına)	Bora	r_i (Bora)
gcc	8200	1640	$8200/1640 = 5,00$	2050	$8200/2050 = 4,00$
mcf	6800	1943	$6800/1943 = 3,50$	1360	$6800/1360 = 5,00$
deepsjeng	7400	1480	$7400/1480 = 5,00$	1850	$7400/1850 = 4,00$
leela	7000	2000	$7000/2000 = 3,50$	1167	$7000/1167 = 6,00$
xz	5600	800	$5600/800 = 7,00$	1120	$5600/1120 = 5,00$

- (b) Geometrik ortalamalar:

$$\text{Fırtına} = (5,00 \times 3,50 \times 5,00 \times 3,50 \times 7,00)^{1/5} = (2143,75)^{0,2} \approx 4,56$$

$$\text{Bora} = (4,00 \times 5,00 \times 4,00 \times 6,00 \times 5,00)^{1/5} = (2400)^{0,2} \approx 4,73$$

Bora'nın SPEC puanı (4,73) Fırtına'ninkinden (4,56) yaklaşık %4 daha yüksektir.

(c) Bora genel SPEC puanında önde olmasına karşın, gcc ve deepsjeng programlarında Fırtına daha hızlıdır; xz'de ise Fırtına belirgin biçimde üstündür (7,00'e karşı 5,00). SPEC puanı tüm programları eşit ağırlıkla birleştirir; dolayısıyla tek bir sayı, her programdaki davranışı gizler. Kullanıcının iş yükü gcc ve xz ağırlıklıysa Fırtına daha iyi bir seçim olabilir. Bu örnek, SPEC puanının yararlı bir özet olduğunu ama her durumda yeterli olmadığını göstermektedir; bireysel program oranlarına da bakılmalıdır.

2.3.3 Diğer Sınama Programı Kümeleri

SPEC, işlemci başarımı için en yaygın kabul görmüş standart olsa da tek sınama programı kümesi değildir. Farklı uygulama alanları, kendine özgü sınama standartları geliştirmiştir:

- **EEMBC** (Embedded Microprocessor Benchmark Consortium): Gömülü sistemler ve mikrodenetleyiciler için tasarlanmış sınama kümeleri sunar. *CoreMark* [12] programı bu konsorsiyumun en bilinen ürünüdür ve gömülü işlemci karşılaştırmalarında uygulamada standart hâline gelmiştir.
- **PARSEC** [3] ve **SPLASH-2** [45]: Çok çekirdekli işlemcilerin koşut hesaplama başarımını ölçmek için geliştirilmiş akademik sınama kümeleridir. Görüntü işleme, akışkan dinamiği ve veri madenciliği gibi koşutluğu yüksek iş yükleri içerir.
- **TPC** (Transaction Processing Performance Council) [40]: Veri tabanı ve işlem işleme başarımını ölçer. TPC-C (çevrimiçi işlem), TPC-H (karar destek) ve TPC-DS (veri ambarı) gibi farklı senaryolara yönelik sınama kümeleri sunar.
- **LINPACK** [10]: Yoğun doğrusal cebir hesaplamalarına dayanan bu sınama programı, dünyanın en hızlı 500 süper bilgisayarını sıralayan *Top500* listesinin temelini oluşturur.
- **Geekbench** [35], **Cinebench**, **3DMark**: Tüketici odaklı sınama programlarıdır. Özellikle bilgisayar ve akıllı telefon incelemelerinde yaygın biçimde kullanılır; ancak endüstri ve akademi çevrelerinde SPEC kadar ağırlık taşımaz.

Bu çeşitlilik, Bölüm 2.2.11’de vurguladığımız ilkeyi pekiştirir: başarımlar tek boyutlu bir kavram değildir. Gömülü bir sistem tasarımcısı *CoreMark* sonuçlarına, süper bilgisayar araştırmacısı *LINPACK*’e, yapay zeka mühendisi *MLPerf*’e [24] (Bilgi Notu 2.7) bakarken, bir oyuncu yalnızca kendi oyunundaki kare hızına güvenir. Doğru sınama programı, ölçülmek istenen iş yüküne bağlıdır.

2.4 Başarım Eğilimleri: Geçmişten Günümüze

Bilgisayar başarımının tarihsel gelişimi, mimari kararları derinden şekillendirmiştir. Bu eğilimleri anlamak, bugünün tasarım tercihlerinin nedenini kavramak için önemlidir.

2.4.1 Tek Çekirdek Başarımının Altın Çağı (1980–2005)

1980’lerin başından 2000’lerin ortasına kadar uzanan yaklaşık 25 yıllık dönemde tek çekirdekli işlemci başarımı her yıl ortalama %40–50 oranında artmıştır. Bu büyümenin üç temel kaynağı vardı:

1. **Saat sıklığının artması:** Transistörlerin küçülmesi, devrelerin daha hızlı anahtarlama-sına olanak tanıdı. 1985’te 16 MHz olan saat sıklığı 2005’te 3,8 GHz’e ulaştı; yaklaşık 240 kat artış.
2. **Buyruk düzeyinde koşutluk:** Boru hattı derinleştirme, çoklu yürütme birimi ve sırasız yürütüm (out-of-order execution) gibi mikromimari yenilikler BBÇ değerini düşürdü. Tek çevrimlik işlemcilerden süperskaler işlemcilere geçiş, aynı saat sıklığında bile başarımı birkaç kat artırdı.
3. **Önbellek hiyerarşisinin gelişmesi:** Artan transistör bütçesi daha büyük ve çok katmanlı önbelleklere ayrıldı; bu sayede bellek erişim gecikmelerinin etkisi azaltıldı.

Bu dönemde ortaya çıkan en önemli mimari gelişmelerden biri 1980’lerde başlayan *RISC devrimi*’dir. Berkeley ve Stanford üniversitelerinde geliştirilen RISC (Reduced Instruction Set Computer – İndirgenmiş Buyruk Kümeli Bilgisayar) felsefesi, az sayıda basit buyruğun boru hattında verimli bir şekilde yürütülmesini öneriyordu. MIPS, SPARC ve ARM gibi BKM’ler bu dönemde doğmuş, RISC-V ise 2010’larda bu geleneğin açık kaynak devamı olarak ortaya çıkmıştır. RISC yaklaşımı, saat sıklığı artışı ve buyruk düzeyinde koşutluk yenilikleriyle birleşerek altın çağın itici gücü olmuştur.

Bu 25 yıllık dönemde tek çekirdek başarımı yaklaşık 10 000 kat artmıştır: 1985’teki bir iş istasyonu yaklaşık 1 MIPS başarıma sahipken, 2005’teki bir masaüstü bilgisayar 10 000 MIPS’i aşıyordu. Programcılar açısından bu dönem son derece verimli bir çağdı: yazılımı değiştirmeden her 18–24 ayda bir donanımın yenilenmesiyle program kendiliğinden hızlanıyordu. Herb Sutter’ın deyişiyle “bedava öğle yemeği” dönemi yaşanıyordu. Ne var ki bu dönem 2005’te güç duvarıyla birlikte sona erecekti.

2.4.2 Güç Duvarı ve Başarım Duraksaması (2005–günümüz)

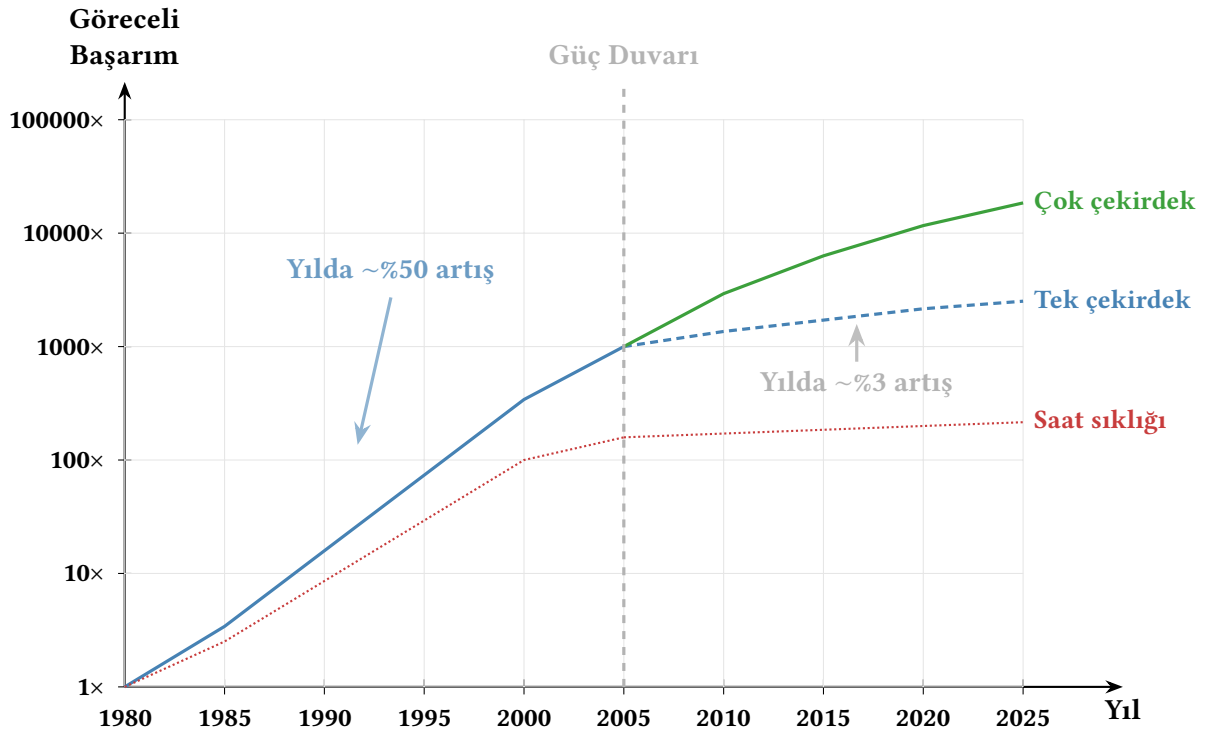
2005 civarında Dennard ölçeklemesinin sona ermesiyle güç duvarı ortaya çıktı. Transistörler küçülmeye devam etse de güç yoğunluğu artık düşmüyordu; aksine sızıntı akımları nedeniyle yükseliyordu. Saat sıklığını artırmak sürdürülemez bir güç tüketimine ve ısınma sorununa yol açtığı için endüstri paradigma değiştirmek zorunda kaldı.

Bu dönüm noktasından sonra tek çekirdek başarımındaki yıllık artış %3–5 düzeyine geriledi. Başarım artışı için üç yeni strateji öne çıktı:

- **Çok çekirdekli işlemciler:** Transistör bütçesi tek bir hızlı çekirdeğe değil, birden fazla çekirdeğe harcanmaya başladı. Başarım artışı, yazılımın koşut olarak yazılmasını gerektirdiği için Amdahl Yasası’nın kısıtları devreye girdi.
- **Alana özgü hızlandırıcılar:** GPU, TPU ve NPU gibi özel amaçlı birimlerin işlemci ile aynı yonga üzerine veya aynı sisteme entegre edilmesi. Genel amaçlı esneklikten ödün vererek belirli iş yüklerinde büyük başarım kazanımları sağlar.
- **Gelişmiş bellek hiyerarşisi:** HBM (High Bandwidth Memory) gibi yüksek bant genişlikli bellek teknolojileri ve 3B yonga istifleme, bellek darboğazını hafifletmek için geliştirildi.

Terim Kökenleri

koşut (*parallel*): İngilizce *parallel* sözcüğü Latince *parallelus* “yan yana giden” kökünden gelir. Türkçe karşılığı olan *koşut*, *koşmak* fiilinden türetilmiştir: yan yana aynı yönde koşan iki şerit gibi, birbirinden bağımsız ama eş zamanlı ilerleyen süreçleri ifade eder. Bu kitapta *koşut hesaplama* (parallel computing), *koşutluk* (parallelism) ve *koşutlaştırma* (parallelization) biçimlerinde kullanılmıştır.



Şekil 2.5: İşlemci başarımları eğilimleri: tek çekirdek başarımları 2005'ten sonra duraklama dönemine girmiş, çok çekirdekli ve heterojen mimariler büyümeye devam ettirmektedir.

2.4.3 Günümüz: Verimlilik ve Uzmanlaşma Çağı

Transistör sayısı Moore Yasası doğrultusunda artmaya devam etmektedir; ancak bu transistörler artık saat sıklığını yükseltmek yerine daha fazla çekirdek, daha büyük önbellek veya alana özgü hızlandırıcı birimler olarak kullanılmaktadır.

Yapay zeka iş yüklerindeki patlayıcı büyüme, başarımları talebini yeni boyutlara taşımıştır. Tablo 2.7 bazı tarihi dönüm noktalarını özetlemektedir.

Tablo 2.7: Hesaplama başarımlarında kilometre taşları

Yıl	Sistem	Başarım	Dönüm Noktası
1946	ENIAC	~5 000 toplama/s	İlk elektronik bilgisayar
1964	CDC 6600	3 MFLOPS	İlk süper bilgisayar
1985	Cray-2	1,9 GFLOPS	Gigaflops sınırı
1997	ASCI Red	1,07 TFLOPS	İlk teraflops sistemi
2008	IBM Roadrunner	1,0 PFLOPS	İlk petaflops sistemi
2022	Frontier	1,2 EFLOPS	İlk ekzaflops sistemi

Bu tablo, hesaplama başarımlarının kabaca her 10–15 yılda 1000 kat arttığını göstermektedir. Ancak bu artış artık tek bir işlemcinin hızlanmasıyla değil, binlerce işlemcinin ve hızlandırıcının koşut çalışmasıyla elde edilmektedir.

Günümüzün en belirgin eğilimlerinden biri *Tek Yongada Sistem* (System on Chip – SoC) yaklaşımıdır. MİB çekirdekleri, GPU, sinir ağı motoru, medya işleme birimi ve bellek denetleyicisi gibi farklı bileşenler tek bir yongada tümleştirilir. Apple'ın 2020'de Mac bilgisayarlarını Intel x86 mimarisinden kendi tasarladığı ARM tabanlı M serisi yongalara geçirmesi bu eğili-

min en çarpıcı örneklerinden biridir. M1'den M5'e uzanan bu yonga ailesi, her kuşakta artan çekirdek sayısı ve genişleyen bellek bant genişliğiyle yüksek başarımlar ve düşük güç tüketimini bir arada sunarak alana özgü hızlandırıcıların potansiyelini somut olarak göstermiştir.

Bir diğer önemli eğilim ise *yongacık mimarisidir* (chiplet architecture). Tek bir büyük yonga yerine, küçük yonga parçacıklarının (*chiplet*) gelişmiş paketleme teknolojisiyle birbirine bağlanması hem üretim verimini artırır hem de farklı üretim süreçlerinin (örneğin MİB çekirdekleri için 5 nm, G/Ç birimleri için 14 nm) bir arada kullanılmasına olanak tanır. AMD'nin Zen mimarisi ve Intel'in Ponte Vecchio gibi tasarımları bu yaklaşımın öncüleridir. Yongacık mimarisi, Moore Yasası'nın yavaşladığı bir dönemde maliyet ve üretim verimliliği açısından yeni bir ölçekleme yolu sunmaktadır.

2.5 Amdahl Yasası

1960'lı yıllarda bilgisayar tasarımcılarını meşgul eden temel sorulardan biri şuydu: *Tek bir güçlü işlemci mi yapmak, yoksa birden çok işlemciyi bir arada çalıştırmak mı daha etkilidir?* Dönemin işlemcileri bugünkü ölçütlerle son derece yavaştı ve birçok araştırmacı çok işlemcili sistemlerin bu darboğazı aşacağını umuyordu.

IBM'in System/360 ana bilgisayar ailesinin baş mimarı olan Gene Amdahl (1922–2015) bu iyimser beklentiyi sorguladı. 1967'de AFIPS Bilgisayar Konferansı'nda sunduğu kısa ama son derece etkili bildirisinde [2], programların doğası gereği sıralı kalan bölümlerinin, sisteme ne kadar işlemci eklenirse eklensin toplam hızlanmayı sınırlayacağını gösterdi. Bu düşüncüyü somutlaştırmak için günlük hayattan bir benzetme yapalım: bir kişi Artvin'e yürüyerek 100 günde gidiyorsa, 100 kişinin yola çıkması bu süreyi 1 güne indirmez; çünkü yürüme işini herkesin kendi adına kendisinin yapması gerekir. Programlarda da durum benzerdir: bir kısmı doğası gereği sıralıdır ve koşutlaştırılmaz.

Amdahl'ın bu gözlemi tek işlemcili mimarilerin önemini yeniden vurgulasa da, daha geniş bir ilke olarak her türlü *kısmi iyileştirmenin* toplam başarıma etkisini nicel olarak açıklar. Bir mühendis işlemcinin kayan nokta birimini 10 kat hızlandırmak için aylarca çalıştığında toplam başarımlar ne kadar artar? Yanıt, o birimin programın ne kadarını etkilediğine bağlıdır. Bu ilişki *Amdahl Yasası* olarak bilinir.

Amdahl Yasası'nın temel düşüncesi şudur: yapılan bir iyileştirme programın tamamını değil, yalnızca belirli bir bölümünü hızlandırır. Programın geri kalan kısmı bu iyileştirmeden etkilenmez ve eski hızında çalışmaya devam eder. Bu yüzden, iyileştirme ne kadar büyük olursa olsun, etkilenmeden kalan bölüm toplam hızlanmaya bir üst sınır koyar.

Bu ilişkiyi nicel olarak ifade edelim. Önce hızlanmayı tanımlayalım:

$$\text{Hızlanma} = \frac{\text{Başarıml}_{\text{yeni}}}{\text{Başarıml}_{\text{eski}}} = \frac{\text{Yürütme zamanı}_{\text{eski}}}{\text{Yürütme zamanı}_{\text{yeni}}} \quad (2.16)$$

Programın toplam yürütme zamanını iki parçaya ayıralım: iyileştirmeden *etkilenen* bölüm ve *etkilenmeyen* bölüm. Etkilenen bölümün programın toplam yürütme zamanı içindeki oranını P ile gösterelim ($0 \leq P \leq 1$). Bu durumda etkilenmeyen bölümün oranı $(1 - P)$ olur. İyileştirme, etkilenen bölümü H kat hızlandırıyorsa yeni yürütme zamanının her bir parçası şöyle hesaplanır:

- **Etkilenmeyen bölüm:** Bu kısım aynen kalır: $(1 - P) \times \text{Yürütme zamanı}_{\text{eski}}$
- **Etkilenen bölüm:** H kat hızlandığı için süresi H 'ye bölünür: $\frac{P}{H} \times \text{Yürütme zamanı}_{\text{eski}}$

İki parçayı toplarsak yeni yürütme zamanını elde ederiz:

$$\text{Yürütme zamanı}_{\text{yeni}} = (1 - P) \times \text{Yürütme zamanı}_{\text{eski}} + \frac{P}{H} \times \text{Yürütme zamanı}_{\text{eski}} \quad (2.17)$$

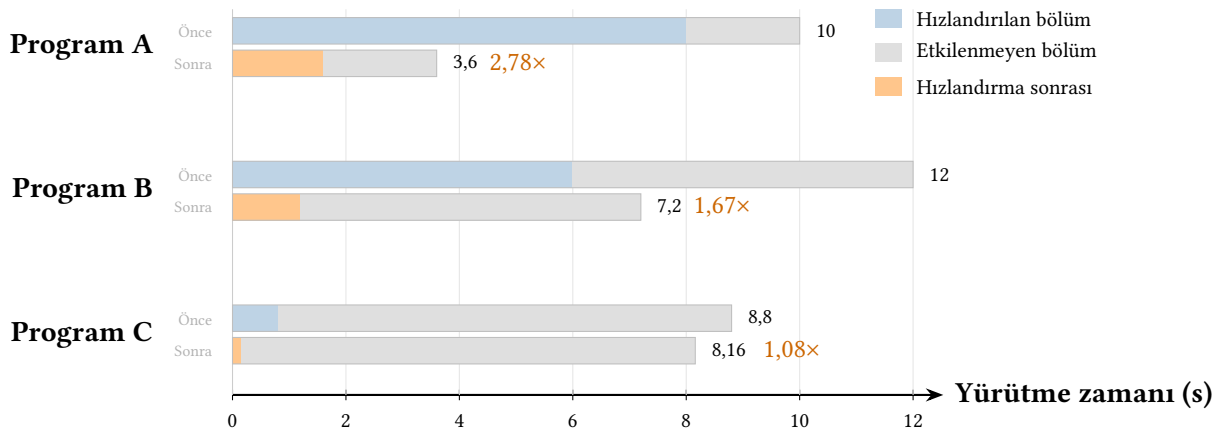
Eski yürütme zamanını paranteze alırsak:

$$\text{Yürütme zamanı}_{\text{yeni}} = \text{Yürütme zamanı}_{\text{eski}} \times \left((1 - P) + \frac{P}{H} \right) \quad (2.18)$$

Bu ifadeyi Denklem 2.16'daki hızlanma tanımında yerine koyarsak Amdahl Yasası'nın genel formülünü elde ederiz:

$$\text{Hızlanma} = \frac{1}{(1 - P) + \frac{P}{H}} \quad (2.19)$$

Formülün iki uç durumunu inceleyelim. Eğer $P = 0$ ise (iyileştirme programın hiçbir bölümünü etkilemiyorsa) hızlanma = 1 olur; başka bir deyişle program hiç hızlanmaz. Eğer $P = 1$ ise (programın tamamı etkileniyorsa) hızlanma = H olur ve iyileştirmenin tüm etkisi sisteme yansır. Uygulamada P bu iki uç arasında bir yerdedir ve iyileştirmenin anlamlı bir hızlanma sağlaması için programın yeterince büyük bir bölümünü etkilemesi gerekir. Şekil 2.6'te üç farklı etki oranı için aynı hızlandırmanın ($H = 5$) toplam başarıma etkisi gösterilmektedir. Şekilde Program C'de görüldüğü gibi, yapılan iyileştirme 5 kat gibi yüksek bir düzeyde olsa bile programın çok küçük bir bölümünü ($P = 0,1$) etkiliyorsa genel hızlanma yalnızca $1,08\times$ düzeyinde kalır.



Üç programda da aynı bileşen $5\times$ hızlandırılmış; ancak P oranı farklıdır

Şekil 2.6: Amdahl Yasası'nın etkisi: aynı hızlandırma ($H = 5$), farklı etki oranları. Program A'da $P = 0,8$, B'de $P = 0,5$, C'de $P = 0,1$.

Tasarım İlkesi 3: Olağan durumu hızlandır

Olağan durumu hızlandırmak bilgisayar mimarisinin en önemli ilkelerinden biridir. Sonuçta getirisi sınırlı olacak, çok büyük emeğin israf edilmesine yol açacak geliştirme çabalarının yerine sistem başarımına en fazla etki edecek bileşenlerin geliştirilmesi gerektiğini vurgular.

Örnek 2.10

Bir görüntü işleme programının toplam yürütme zamanı 10 saniyedir. Programın profil analizi, bu sürenin %50'sinin kayan nokta (ondalıklı sayı) işlemlerinde harcandığını göstermektedir. Mühendislik ekibi, kayan nokta birimini yeniden tasarlayarak bu işlemleri 5 kat hızlandırmayı planlamaktadır.

- (a) Bu iyileştirmeden sonra programın yeni yürütme zamanı ve toplam hızlanma ne olur?
- (b) Ekip toplam başarımı 3 katına çıkarmak isteseydi, kayan nokta işlemlerinin program içindeki oranının en az ne kadar olması gerekirdi?
- (c) Kayan nokta birimi sonsuz hıza ulaşsa bile ($H \rightarrow \infty$) elde edilebilecek en yüksek hızlanma nedir?

Çözüm:

- (a) $P = 0,5$ ve $H = 5$ değerlerini Amdahl Yasası'na (Denklem 2.19) yerleştirelim:

$$\text{Hızlanma} = \frac{1}{(1 - 0,5) + \frac{0,5}{5}} = \frac{1}{0,5 + 0,1} = \frac{1}{0,6} \approx 1,67$$

Yeni yürütme zamanı = $10 \times 0,6 = 6$ saniye olur. Kayan nokta birimi 5 kat hızlansa da programın yarısı bu birimden bağımsız olduğu için toplam hızlanma yalnızca $1,67 \times$ düzeyinde kalır.

- (b) Toplam hızlanmanın 3 olması için $H = 5$ ile gereken P oranını bulalım:

$$3 = \frac{1}{(1 - P) + \frac{P}{5}} \implies (1 - P) + \frac{P}{5} = \frac{1}{3}$$

$$\implies 1 - \frac{4P}{5} = \frac{1}{3} \implies P = \frac{5}{6} \approx 0,833$$

Programın en az %83,3'ünün kayan nokta işlemlerinden oluşması gerekir. Bu oldukça yüksek bir oran olup, Amdahl Yasası'nın neden "iyileştirme oranı kadar etki oranı da önemlidir" dediğini somut olarak gösterir.

- (c) $H \rightarrow \infty$ durumunda $P/H \rightarrow 0$ olur ve formül sadeleşir:

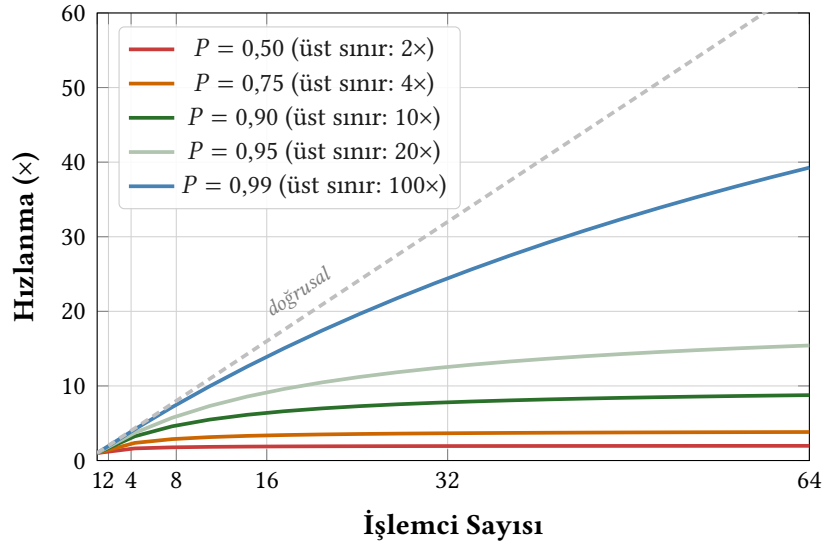
$$\text{Hızlanma}_{\max} = \frac{1}{1 - P} = \frac{1}{1 - 0,5} = 2$$

Kayan nokta birimi ne kadar hızlandırılırsa hızlandırılınsın, programın kayan nokta dışında kalan yarısı toplam hızlanmayı en fazla $2 \times$ ile sınırlar. Bu sonuç Amdahl Yasası'nın en çarpıcı mesajıdır: *etkilenmeyen kısım, iyileştirmenin üst sınırını belirler.*

Bilgi Notu 2.9: Amdahl Yasası ve Koşut Hesaplama

Amdahl Yasası, bir programın seri kalan kısmının koşut hızlanmayı ciddi biçimde sınırlandığını gösterir. Örneğin programın %10'u seri ise, sonsuz sayıda çekirdek kullansak bile en fazla 10 kat hızlanma elde edebiliriz. Bu durum çok çekirdekli tasarımlarda yazılımın da koşut olarak yazılması gerekliliğini vurgular. Amdahl Yasası'nın boru hattı tasarımında verimliliği nasıl sınırladığını Bölüm 7'de somut örneklerle inceleyeceğiz.

Şekil 2.7, Amdahl Yasası'nın koşut oran (P) ve işlemci sayısına göre hızlanma üst sınırını nasıl belirlediğini göstermektedir. İşlemci sayısı arttıkça her eğri yataylaşmakta; azalan getiri yasası devreye girmektedir. $P = 0,99$ (%99 koşut) olan programda bile 64 işlemciyle yalnızca $\sim 39\times$ hızlanma elde edilebilmektedir; bunun nedeni, seri kalan %1'lik kısmın $H \rightarrow \infty$ durumunda toplam hızlanmayı $1/0,01 = 100\times$ ile sınırlamasıdır.



Şekil 2.7: Amdahl Yasası hızlanma eğrileri: farklı koşut oranlar (P) için işlemci sayısına göre elde edilebilecek hızlanma. Her eğri, işlemci sayısı arttıkça $1/(1 - P)$ değerine yaklaşır ve yataylaşır. Kesik çizgi ideal (doğrusal) hızlanmayı gösterir.

2.5.1 Gustafson Yasası

Amdahl Yasası önemli bir gerçeği ortaya koyar; ancak karamsar bir varsayıma dayanır: *problem büyüklüğü sabittir*. Daha fazla çekirdek eklediğimizde aynı programı, aynı veriyle, sadece daha hızlı çözeceğimizi varsayar. Peki gerçek hayatta kullanıcılar böyle mi davranır?

Bir hava tahmini merkezini düşünelim. Merkez tek işlemcili bir sistemle 10 km çözünürlükte tahmin yapabiliyorken 64 çekirdekli bir sisteme geçtiğinde genellikle aynı 10 km tahmini 64 kat hızlı yapmaz; bunun yerine çözünürlüğü 1 km'ye çıkararak *daha ayrıntılı* bir tahmin üretir. Benzer şekilde, bir yapay zeka araştırmacısı daha fazla GPU aldığı anda aynı modeli daha hızlı eğitmek yerine *daha büyük* bir model eğitmeye yönelir. Kısacası, uygulamada hesaplama kapasitesi arttığında insanlar doğal olarak problem büyüklüğünü de artırır.

John Gustafson 1988'de tam da bu gözlemi formüle dökmüştür [13]. Gustafson'ın bakış açısı şöyledir: çekirdek sayısı arttıkça programın koşut kısmı büyür (daha fazla veri işlenir, daha ince çözünürlük hesaplanır), ama seri kısım (başlatma, yapılandırma, sonuçları birleştirme gibi işlemler) hemen hemen aynı kalır. Dolayısıyla seri kısmın *mutlak süresi* değişmese

bile programın geneline oranla *payı* düşer. Amdahl “elimdeki program sabit, kaç çekirdek gerekir?” diye sorarken, Gustafson “elimde N çekirdek var, ne büyüklükte problem çözebilirim?” diye sorar.

Gustafson Yasası ölçeklenmiş hızlanmayı (*scaled speedup*) şöyle tanımlar:

$$S_{\text{Gustafson}}(N) = N - s \times (N - 1) \quad (2.20)$$

Burada N çekirdek sayısı, s ise N çekirdekli koşut yürütme sırasında ölçülen seri kısmın oranıdır ($0 \leq s \leq 1$). Formülün iki uç durumuna bakalım: $s = 0$ (tamamen koşut program) ise $S = N$, başka bir deyişle çekirdek sayısı ile doğru orantılı, *doğrusal ölçeklenme* elde edilir. $s = 1$ (tamamen seri program) ise $S = 1$ olup hiç hızlanma yoktur. s küçüldükçe hızlanma N 'ye yaklaşır; bu da Amdahl'ın koyduğu üst sınırın çok ötesinde bir ölçeklenme vaat eder.

Örnek 2.11

Bir hava tahmini programı 64 çekirdekli bir sistemde 1 km çözünürlükte çalıştırılıyor. Koşut yürütme sırasında yapılan ölçümde seri kısmın oranı $s = 0,04$ (%) olarak belirlenmiştir.

- (a) Gustafson Yasası'na göre ölçeklenmiş hızlanma nedir?
- (b) Amdahl Yasası aynı seri oran için ne söyler?

Çözüm:

- (a) Gustafson formülünü uygularsak:

$$S = 64 - 0,04 \times (64 - 1) = 64 - 2,52 = 61,48\times$$

Problem büyüklüğünün ölçeklenmesiyle birlikte 64 çekirdek neredeyse 61,5 kat hızlanma sağlamaktadır.

- (b) Amdahl Yasası aynı seri oran için sabit problem büyüklüğü varsayar:

$$S = \frac{1}{0,04 + \frac{0,96}{64}} = \frac{1}{0,055} \approx 18,2\times$$

Aynı %4 seri oranla Amdahl yalnızca 18,2× öngörür. Aradaki fark, Gustafson'ın “çekirdek sayısı artınca problem de büyür” varsayımından kaynaklanır.

Bilgi Notu 2.10: Amdahl ve Gustafson: Hangisi Doğru?

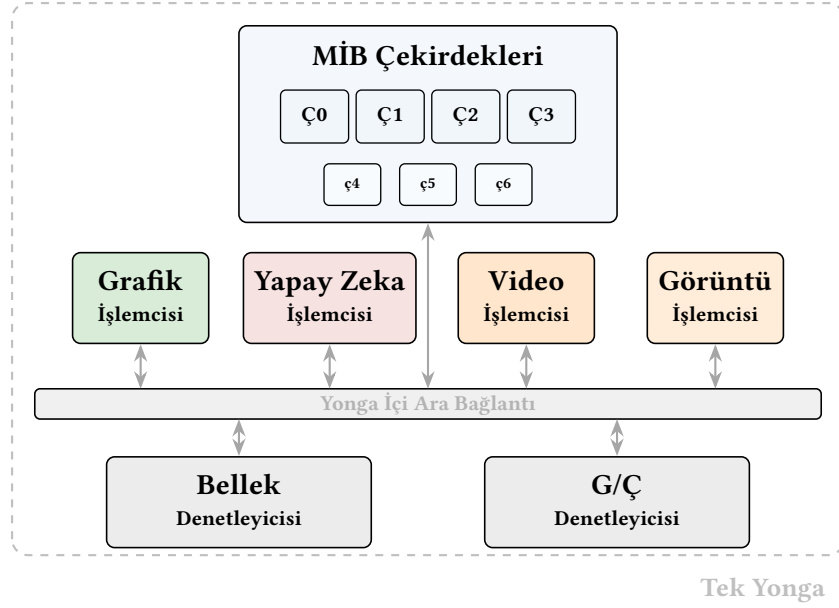
Her iki yasa da doğrudur; farklı senaryoları modellerler. Eğer problem büyüklüğü *sabit* ise (örneğin belirli bir matris çarpımı), Amdahl Yasası geçerlidir ve daha fazla çekirdek eklemek azalan getiri sağlar. Eğer çekirdek sayısı arttıkça problem de büyütülüyorsa (örneğin daha yüksek çözünürlüklü tahmin, daha büyük yapay zeka modeli), Gustafson Yasası daha gerçekçi bir tablo çizer.

2.5.2 Heterojen Sistemlerde Başarım

Amdahl ve Gustafson yasaları bir programın seri ve koşut kısımlarını ele alır; ancak her iki yasa da tüm çekirdeklerin *aynı türde* olduğunu varsayar. Günümüzün tek yonga (*System on*

Chip – SoC) tasarımlarında bu varsayım artık geçerli değildir. Bir akıllı telefon ya da dizüstü bilgisayar yongası, genel amaçlı MİB çekirdeklerinin yanı sıra grafik işlemcisi (*GPU*), yapay zeka işlemcisi (*NPU*), görüntü işlemcisi (*ISP*), video işlemcisi ve ses işleme birimi gibi düzinelere özelleşmiş birim barındırır. Her birim, belirli bir iş yükünde MİB’den onlarca hatta yüzlerce kat daha hızlı ve enerji verimli çalışabilir.

Şekil 2.8 günümüzde yaygın bir tek yonganın basitleştirilmiş blok diyagramını göstermektedir. Bu yongada her birim kendi uzmanlık alanındaki hesaplamayı üstlenir; MİB ise genel amaçlı denetim ve hesaplama görevlerini yürütür.



Şekil 2.8: Günümüzde yaygın bir tek yonga mimarisi. Büyük (Ç0–Ç3) ve küçük (ç4–ç6) MİB çekirdekleri genel amaçlı hesaplama yaparken grafik işlemcisi, yapay zeka işlemcisi, video işlemcisi ve görüntü işlemcisi gibi birimler kendi uzmanlık alanlarında çok daha yüksek başarımlar ve enerji verimliliği sunar.

Bu mimari yapıda başarımları yalnızca MİB hızıyla ölçmek yanıltıcı olur. Bir video düzenleme uygulamasının süresini belirleyen asıl etken MİB değil video işlemcisidir; bir yapay zeka uygulamasının çıkarım hızını belirleyen yapay zeka işlemcisidir. Kullanıcı deneyimini şekillendiren, her iş yükünün doğru birime yönlendirilmesidir.

Genişletilmiş Amdahl Yasası

Heterojen bir sistemde iş yükünün farklı kısımları farklı birimlerde yürütülür. Bir programın toplam yürütme zamanı T olsun. Bu programın K adet hızlandırıcıya dağıtılabilecek kısımları olsun: k 'nci hızlandırıcı, programın f_k oranındaki kısmını MİB'e göre S_k kat hızlandırsın. Geri kalan $(1 - \sum_{k=1}^K f_k)$ oranındaki kısım yalnızca MİB üzerinde çalışabilir. Bu durumda toplam hızlanma, Amdahl Yasası'nın doğal bir genelleştirmesi olarak şöyle yazılır [17]:

$$\text{Hızlanma}_{\text{heterojen}} = \frac{1}{\left(1 - \sum_{k=1}^K f_k\right) + \sum_{k=1}^K \frac{f_k}{S_k}} \quad (2.21)$$

Denklem 2.21, klasik Amdahl formülünün (Denklem 2.19) özel bir hâlidir: tek bir hızlandırıcı ($K = 1$) ve $f_1 = P$, $S_1 = H$ alınırsa özgün biçime dönüşür.

Bu formül iki önemli gerçeği açığa çıkarır. Birincisi, her hızlandırıcının katkısı yalnızca kendi kapsadığı iş yükü oranıyla (f_k) sınırlıdır; tıpkı klasik Amdahl'da olduğu gibi. İkincisi, iş yükünün MİB'de kalan kısmı azaldıkça toplam hızlanma çarpıcı biçimde artabilir. Günümüzün tek yonga tasarımlarında grafik işlemcisi, yapay zeka işlemcisi ve video işlemcisi gibi birimlerin bir arada bulunmasının nedeni budur: her biri iş yükünün farklı bir dilimini hızlandırarak MİB'de kalan oranı en aza indirir.

Örnek 2.12

Bir akıllı telefon tek yongasında video düzenleme uygulaması çalıştırılıyor. Uygulamanın iş yükü dağılımı ve her birimin MİB'e göre hızlanma katsayısı şöyledir:

İş yükü bileşeni	Oran (f_k)	Hızlanma (S_k)
Video kodlama/kod çözme (video işlemcisi)	0,45	50×
Yapay zeka süzgeçleri (yapay zeka işlemcisi)	0,20	30×
Grafik önizleme (grafik işlemcisi)	0,15	20×
Genel işlemler (yalnızca MİB)	0,20	1×

- Tüm hızlandırıcılar etkinken toplam hızlanmayı hesaplayın.
- Hızlandırıcılar devre dışı bırakılıp her şey MİB üzerinde çalıştırılıyorsa uygulama ne kadar yavaşlardı?
- Video işlemcisi 50× yerine 200× hızlandırırsa toplam hızlanma ne olurdu?

Çözüm:

- Denklem 2.21'i uygulayalım. MİB'de kalan oran = 0,20; hızlandırıcılara dağıtılan oranlar toplamı = 0,45 + 0,20 + 0,15 = 0,80. O hâlde:

$$\begin{aligned} \text{Hızlanma} &= \frac{1}{0,20 + \frac{0,45}{50} + \frac{0,20}{30} + \frac{0,15}{20}} \\ &= \frac{1}{0,20 + 0,009 + 0,0067 + 0,0075} = \frac{1}{0,2232} \approx 4,48\times \end{aligned}$$

Hızlandırıcılar sayesinde uygulama MİB-yalnızca duruma göre yaklaşık 4,5 kat hızlıdır.

- Hızlandırıcılar olmadan her şey MİB'de çalışır ($S_k = 1$ hepsi için), dolayısıyla hızlanma = 1×. Bu, hızlandırıcılı sisteme göre uygulamanın 4,48 kat daha yavaş olacağı anlamına gelir. Bir dakikada tamamlanan düzenleme, MİB-yalnızca sistemde yaklaşık 4,5 dakika sürerdi.
- Video işlemcisini 50×'ten 200×'e çıkaralım:

$$\begin{aligned} \text{Hızlanma} &= \frac{1}{0,20 + \frac{0,45}{200} + \frac{0,20}{30} + \frac{0,15}{20}} \\ &= \frac{1}{0,20 + 0,00225 + 0,0067 + 0,0075} = \frac{1}{0,2165} \approx 4,62\times \end{aligned}$$

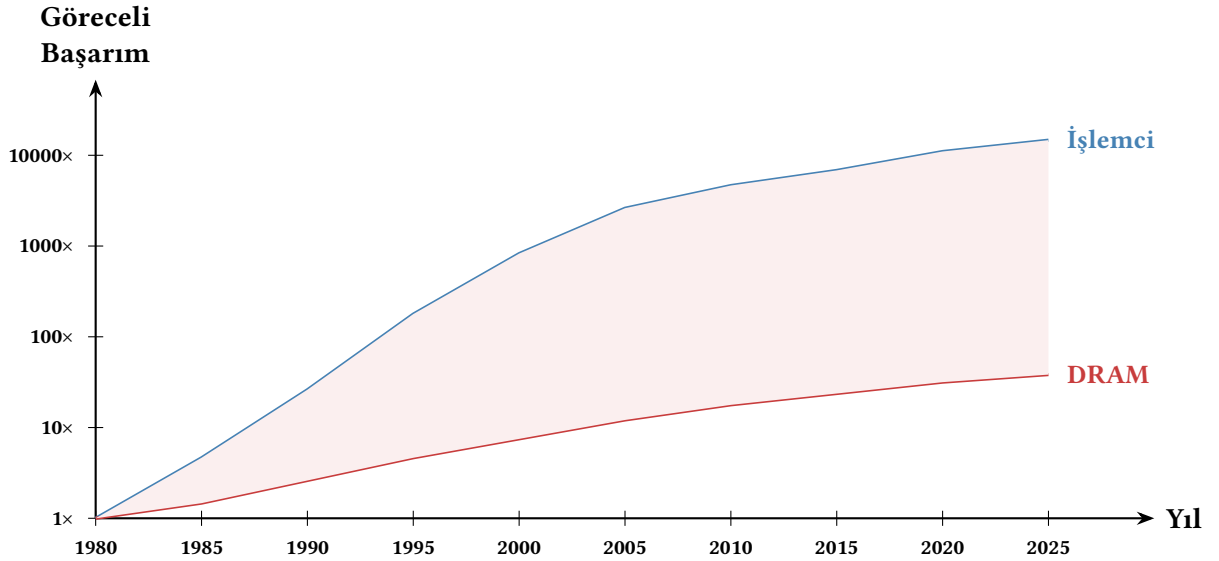
Video işlemcisi 4 kat hızlanmasına rağmen toplam hızlanma yalnızca $4,48\times$ 'ten $4,62\times$ 'e çıkmıştır. Bunun nedeni tanındıktır: Amdahl Yasası burada da geçerlidir; MİB'de kalan %20'lik kısım, toplam hızlanmayı en fazla $1/0,20 = 5\times$ ile sınırlar. Bu üst sınıra zaten yaklaşıldığı için tek bir hızlandırıcıyı daha da güçlendirmek azalan getiri sağlar.

Bilgi Notu 2.11: Doğru İş Yükünü Doğru Birime Yönlendirmek

Heterojen sistemlerde en büyük başarımlı kazanımı, mümkün olan her iş yükü parçasını uygun hızlandırıcıya yönlendirmekten gelir. Yazılımın MİB üzerinde kalan oranını azaltmak, tek bir hızlandırıcıyı daha da güçlendirmekten çok daha etkilidir. Bu ilke, işletim sistemi ve derleyici tasarımında giderek daha önemli bir rol oynamaktadır: çağdaş işletim sistemleri iş yükü karakterizasyonuna göre görevleri büyük veya küçük çekirdeklerle, grafik işlemcisine ya da yapay zeka işlemcisine otomatik olarak yönlendirebilir.

2.6 Bellek Duvarı

İşlemci başarımlı yılda %50 artarken DRAM erişim süreleri yılda yalnızca %7–9 iyileşmiştir. Bu farklı büyüme hızları, işlemci ile bellek arasında giderek genişleyen bir başarımlı uçurumu yaratmıştır; bu olguya *Bellek Duvarı* (Memory Wall) adı verilir.



Şekil 2.9: İşlemci ve bellek başarımlı artış hızları arasındaki açılan makas (Bellek Duvarı)

Bu sorunu hafifletmek için birçok yöntem geliştirilmiştir:

- **Çok katmanlı ön bellek hiyerarşisi** (L1, L2, L3): Sık erişilen veriler işlemciye yakın, küçük ama hızlı belleklerde tutulur.
- **Donanımsal önceden getirme** (hardware prefetching): İşlemci, yakında ihtiyaç duyulacak veriyi tahmin ederek önceden belleğe yükler.

- **Sırasız yürütüm** (out-of-order execution): Bellek erişimi beklerken diğer bağımsız buyrukları yürütür.
- **HBM ve yığılmış bellek**: 3B yonga istifleme ile bant genişliğini artırır.

Bu uçurumun büyüklüğünü somut sayılarla görelim. 1990 yılında tipik bir işlemcinin saat sıklığı 50 MHz (çevrim süresi 20 ns), DRAM erişim süresi ise 100 ns idi; dolayısıyla bir bellek erişimi $100/20 = 5$ çevrim sürerdi. 2020'ye gelindiğinde saat sıklığı 4 GHz'e (çevrim süresi 0,25 ns) çıkmış, DRAM erişim süresi ise ancak 50 ns'ye düşmüştür. Bir bellek erişimi artık $50/0,25 = 200$ çevrim bekletmektedir. İşlemci hızı 80 kat artarken bellek bekleme süresi çevrim cinsinden 40 kat artmıştır. İşte önbellek hiyerarşisini zorunlu kılan temel neden budur.

Bellek duvarı sorunu Bölüm 8'de ayrıntılı olarak incelenecektir. Burada vurgulanması gereken nokta, işlemci hızını artırmanın tek başına yeterli olmadığı; bellek hiyerarşisinin de buna uygun şekilde tasarlanması gerektiğidir.

2.7 Güç-Başarım Dengesi

Günümüzde bir bilgisayarın başarımını değerlendirirken yalnızca hızına değil, harcadığı enerjiye de bakmak gerekir. Veri merkezlerinde elektrik maliyeti, dizüstü bilgisayarlarda pil ömrü ve taşınabilir aygıtlarda ısı kısıtları, *güç-başarım oranını* (performance per watt) önemli bir tasarım ölçütü haline getirmiştir.

Bölüm 1.7.1'te gördüğümüz gibi devingen güç tüketimi $P_{\text{devingen}} = C \times V_{dd}^2 \times f$ formülüyle hesaplanır ve besleme geriliminin *karesiyle* orantılıdır (Denklem 1.4). Gerilimin karesiyle doğru orantılı olan bu ilişki önemli bir sonuç doğurur: saat sıklığını (f) artırmak devingen gücü doğru orantılı olarak artırırken, bunu desteklemek için gerilimin de yükseltilmesi gerektiğinde güç tüketimi çok daha hızlı yükselir. Buna ek olarak, Bölüm 1.7.2'te ele alınan durağan güç (sızıntı akımları) da transistörler küçüldükçe artmaktadır. İşte bu iki etken, 2005'ten sonra endüstrinin saat sıklığını artırmak yerine çok çekirdekli ve heterojen tasarımlara yönelmesinin temel nedenidir.

Enerji verimliliği artık başarım kadar önemli bir tasarım ölçütüdür:

$$\text{Enerji verimliliği} = \frac{\text{Başarım}}{\text{Güç tüketimi}} = \frac{\text{İşlem/saniye}}{\text{Watt}} \quad (2.22)$$

Örnek 2.13

Bir veri merkezi yöneticisi yapay zeka eğitimi için 1000 sunucudan oluşan bir küme kuracaktır. İki aday işlemci vardır:

	İşlemci A	İşlemci B
Başarım	100 GFLOPS	60 GFLOPS
Güç tüketimi	200 W	45 W

- Her iki işlemcinin enerji verimliliğini hesaplayın.
- 100 GFLOPS toplam başarıma ulaşmak için her işlemciden kaç tane gerekir ve toplam güç tüketimi ne olur?

- (c) Elektrik birim fiyatı 0,10 \$/kWh ise her iki seçeneğin yıllık enerji maliyetini karşılaştırın.

Çözüm:

- (a) Denklem 2.22'i uygularsak:

$$A: \frac{100}{200} = 0,5 \text{ GFLOPS/W}, \quad B: \frac{60}{45} \approx 1,33 \text{ GFLOPS/W}.$$

İşlemci B, watt başına 2,67 kat daha fazla işlem yapabilmektedir.

- (b) İşlemci A zaten 100 GFLOPS sağladığından 1 adet yeterlidir: $1 \times 200 = 200 \text{ W}$. İşlemci B ile aynı başarıma ulaşmak için $\lceil 100/60 \rceil = 2$ adet gerekir: $2 \times 45 = 90 \text{ W}$. Aynı başarımlı düzey için İşlemci B seçeneği yarıdan az güç tüketir.

- (c) 1000 sunucu, yılda $365 \times 24 = 8760$ saat çalışacaktır.

$$A \text{ ile: } 1000 \times 200 \text{ W} = 200 \text{ kW} \Rightarrow 200 \times 8760 = 1\,752\,000 \text{ kWh} \Rightarrow 175\,200 \text{ $/yıl}.$$

$$B \text{ ile (sunucu başına 2 işlemci): } 1000 \times 90 \text{ W} = 90 \text{ kW} \Rightarrow 90 \times 8760 = 788\,400 \text{ kWh} \Rightarrow 78\,840 \text{ $/yıl}.$$

B seçeneği yılda yaklaşık 96 000 \$ tasarruf sağlar; buna ek olarak soğutma maliyeti de güç tüketimiyle orantılı düştüğü için gerçek fark daha da büyük olacaktır.

Not: Bu hesap yalnızca işletme maliyetini kapsar. B seçeneğinde sunucu başına 2 işlemci gerektiğinden ilk yatırım maliyeti daha yüksek olabilir. Gerçek kararda bu iki maliyet birlikte değerlendirilir ve yatırımın kendini ne kadar sürede geri ödediğine (*payback period*) bakılır.

Örnek 2.14

Enerji-gecikme çarpımı (*Energy-Delay Product*, EDP), hem enerji verimliliğini hem de başarımlı tek bir sayıda birleştiren bir ölçüttür:

$$EDP = E \times t = P \times t^2$$

Burada E harcanan enerji, t ise görevin tamamlanma süresidir. Düşük EDP değeri, aynı işi hem az enerjiyle hem de kısa sürede yapabilmeyi gösterir. Bir taşınabilir aygıt tasarımcısı iki işlemci arasında seçim yapacaktır:

	İşlemci K	İşlemci L
Güç tüketimi	2 W	8 W
Görev süresi	400 ms	120 ms

- (a) Her iki işlemcinin enerji tüketimini ve EDP değerini hesaplayın.
 (b) Hangi işlemci taşınabilir aygıt için daha uygundur?
 (c) İşlemci L'nin gerilimi %20 düşürülürse saat sıklığı %15 azalır. Yeni güç tüketimini, görev süresini ve EDP değerini hesaplayın.

Çözüm:

- (a) Enerji = güç $\times$ süre:

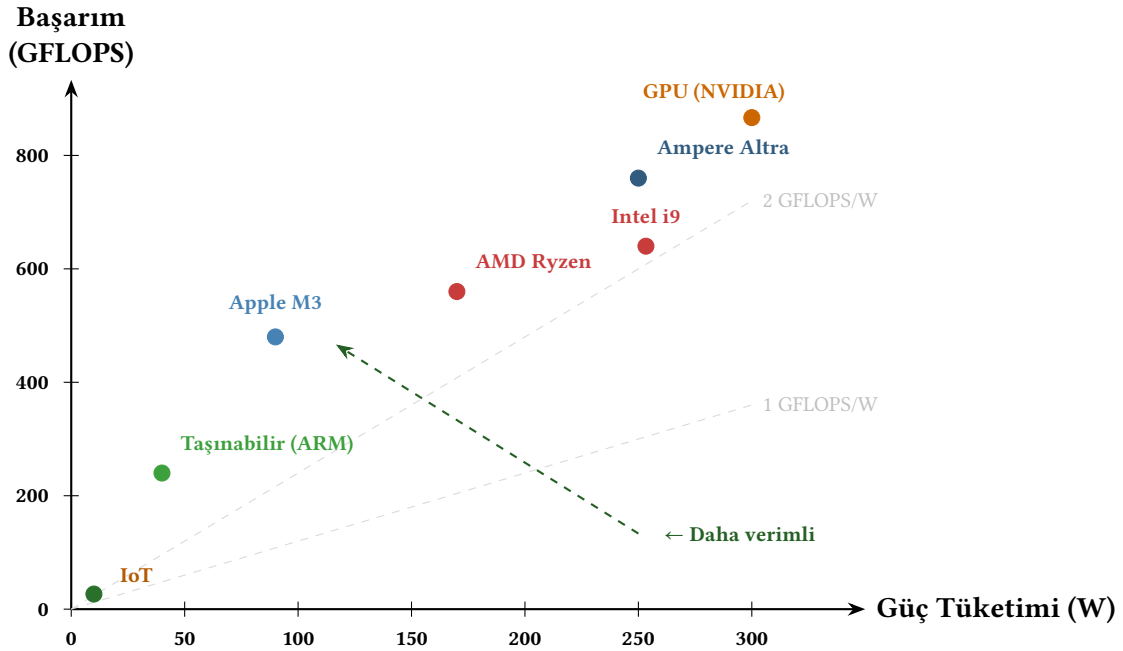
$$K: E_K = 2 \times 0,4 = 0,8 \text{ J}. \quad L: E_L = 8 \times 0,12 = 0,96 \text{ J}.$$

$$EDP = \text{enerji} \times \text{süre:}$$

$$K: EDP_K = 0,8 \times 0,4 = 0,320 \text{ J}\cdot\text{s}. \quad L: EDP_L = 0,96 \times 0,12 = 0,1152 \text{ J}\cdot\text{s}.$$

- (b) İşlemci K daha az enerji harcar ($0,8 < 0,96 \text{ J}$) ama EDP'si daha yüksektir ($0,320 > 0,1152$). İşlemci L ise görevi 3,3 kat daha hızlı bitirerek EDP açısından 2,8 kat daha iyidir. Taşınabilir aygıtta yalnızca pil ömrü değil, kullanıcı deneyimi de önemlidir; L'nin düşük EDP'si onu daha dengeli bir seçim yapar.
- (c) Gerilim $0,8\times$ olduğunda güç, gerilimin karesiyle orantılı düşer: $P_{\text{yeni}} = 8 \times 0,8^2 \times 0,85 = 8 \times 0,544 = 4,35 \text{ W}$. (Saat sıklığı %15 azaldığından f çarpanı 0,85.)
 Görev süresi saat sıklığıyla ters orantılı: $t_{\text{yeni}} = 0,12/0,85 \approx 0,141 \text{ s}$.
 Yeni enerji: $4,35 \times 0,141 \approx 0,614 \text{ J}$; enerji %36 düştü.
 Yeni EDP: $0,614 \times 0,141 \approx 0,0866 \text{ J-s}$; EDP %25 iyileşti.
 Gerilim düşürme hem enerjiyi hem de EDP'yi iyileştirdi; küçük bir başarımla önemli enerji tasarrufu sağlandı. Bu ödünleşme, taşınabilir işlemci tasarımında sıklıkla kullanılan *gerilim-sıklık ölçekleme* (DVFS, Dynamic Voltage and Frequency Scaling) tekniğinin temelidir.

Bu ödünleşme, ARM tabanlı sunucu işlemcilerinin (AWS Graviton, Ampere Altra) ve RISC-V tabanlı tasarımların yükselişinin temel nedenidir. Aynı toplam başarımla daha az enerjiyle elde etmek, hem ekonomi hem çevre açısından önemlidir. Şekil 2.10'de farklı işlemci türlerinin güç-başarımlar düzlemindeki konumu görülmektedir.



Şekil 2.10: Farklı işlemci türlerinin güç-başarımlar düzlemindeki konumu. Sol üst köşeye yakın noktalar daha yüksek enerji verimliliğine sahiptir.

Yapay zeka ve veri merkezlerinde güç-başarımlar ilişkisi

Güç-başarımlar dengesi, yapay zeka çağında daha da can alıcı bir hale gelmiştir. Büyük bir dil modelinin (örneğin GPT-4 düzeyinde) eğitimi, binlerce hızlandırıcının haftalarca çalışmasını gerektirir ve tek bir eğitim döngüsü milyonlarca dolar değerinde elektrik tüketebilir. Bir yapay zeka veri merkezinin işletme maliyetinin en büyük kalemi artık donanım satın alma değil,

elektrik faturasıdır. Yüzlerce megawatt güç çeken bir veri merkezi için yılda yüz milyonlarca dolarlık enerji gideri söz konusudur; buna bir de soğutma sistemlerinin enerji tüketimi eklenir.

Bu gerçek, “harcanan her bir watt için en yüksek başarıyı almak” ilkesini (FLOPS/watt ölçütünü) mimari tasarımın merkezine taşımıştır. Hızlandırıcı üreticileri (NVIDIA, AMD, Google TPU, Intel) yeni nesil ürünlerini tanıtırken artık yalnızca mutlak FLOPS değerini değil, watt başına FLOPS değerini de ön plana çıkarmaktadır. Benzer şekilde bulut servis sağlayıcıları, sunucu donanımı seçerken toplam sahip olma maliyetinde (TCO, *total cost of ownership*) enerji payını birincil parametre olarak değerlendirmektedir.

Bilgi Notu 2.12: Green500: Süper Bilgisayarların Enerji Verimliliği Sıralaması

Top500 listesi en hızlı süper bilgisayarları sıralarken, *Green500* listesi en enerji verimli süper bilgisayarları sıralar. Ölçüt, LINPACK karşılaştırma sınavındaki GFLOPS/watt değeridir. 2024 yılı itibarıyla en verimli sistemler 60–70 GFLOPS/watt düzeyine ulaşmıştır; bu rakam on yıl öncesinin en hızlı süper bilgisayarının *toplam* başarımından daha yüksek bir değerdir!

2.8 Gerçek Dünya: İşlemci Karşılaştırmaları

Bu bölümde öğrendiğimiz kavramları gerçek işlemciler üzerinde uygulayalım. Tablo 2.8, 2023–2024 yıllarında piyasadaki dört farklı işlemciyi başarımlar parametreleri açısından karşılaştırmaktadır.

Tablo 2.8: Güncel işlemci karşılaştırması (2023–2024)

Özellik	Intel Core i9-14900K	Apple M3 Max	AMD Ryzen 9 7950X	Ampere Altra Max
BKM	x86-64	ARMv8	x86-64	ARMv8
Çekirdek sayısı	8P+16E	12P+4E	16	128
Maks. saat sıklığı	6,0 GHz	4,05 GHz	5,7 GHz	3,0 GHz
TDP	253 W	~92 W	170 W	250 W
SPECint rate (est.)	~310	~280	~290	~370
Kullanım alanı	Masaüstü	Dizüstü	Masaüstü	Sunucu

Bu tablodan çarpıcı gözlemler çıkarmak mümkündür:

- **Saat sıklığı yanıltır:** Intel’in 6 GHz saat sıklığı Apple’ın 4 GHz’inden çok daha yüksek olmasına rağmen, tek çekirdek başarımları yakın düzeydedir. Bunun nedeni Apple M3’ün çevrim başına daha fazla iş yapmasıdır (düşük BBÇ).
- **İşlem hacmi için çekirdek sayısı:** Ampere Altra Max 128 çekirdekle en düşük saat sıklığına sahip olmasına rağmen, çok iş parçacıklı iş yüklerinde en yüksek toplam işlem hacmini sağlar.
- **Enerji verimliliği:** Apple M3 Max, 92 W ile diğerlerinin yarısı kadar güç harcayarak yakın başarımlar sunar. Bu, ARM mimarisinin enerji verimliliği avantajını somut olarak gösterir. Sonraki kuşak M4 ve M5 yongaları bu verimliliği daha da ileriye taşımıştır.
- **Heterojen tasarım:** Hem Intel hem Apple, büyük (performans) ve küçük (verimlilik) çekirdekleri bir arada kullanarak farklı iş yüklerine uyum sağlar.

2.9 Yanılgılar ve Tuzaklar

Başarım kavramı ilk bakışta basit görünse de sezgiye dayanan yaklaşımlar çoğu zaman yanıltıcı sonuçlara yol açar. Tek bir ölçüte odaklanmak, korelasyonu nedensellikte karıştırmak ya da pazarlama verilerini mühendislik gerçeği gibi kabul etmek; bunların hepsi hem öğrencilerin hem de deneyimli mühendislerin düşebileceği tuzaklardır. Bu bölümde başarımla ilgili en yaygın yanılgıları ve tasarım hatalarını somut örneklerle ele alacağız.

Yanılgı: Saat sıklığı yüksek olan işlemci her zaman daha hızlıdır

Örnek 2.2’de gördüğümüz gibi, 300 MHz’lik işlemci 200 MHz’lik olandan daha yavaş çalışabilir. Başarım üç bileşenin çarpımına bağlıdır (Denklem 2.8); saat sıklığı bunlardan yalnızca biridir. Intel Pentium 4 işlemcisi 3,8 GHz’e ulaşmıştı ancak aynı dönemdeki 2 GHz’lik AMD Athlon 64 birçok iş yükünde daha hızlıydı; çünkü Athlon 64’ün daha kısa boru hattı ve daha verimli mikromimarisi sayesinde BBÇ değeri çok daha düşüktü. Günümüzde de Apple M serisi (3–4 GHz) ile Intel i9 (6 GHz) arasında benzer bir tablo mevcuttur: düşük saat sıklığına rağmen Apple çevrimi başına daha fazla iş yaparak yakın tek çekirdek başarımları elde etmektedir.

Yanılgı: MIPS güvenilir bir başarımlar ölçüsüdür

MIPS değeri, farklı buyruk kümelerine sahip işlemciler arasında anlamlı bir karşılaştırma yapılmasına izin vermez. Aynı işi yapan iki farklı BKM, biri 100 basit buyrukla diğeri 40 karmaşık buyrukla tamamlayabilir; basit buyruklu olan MIPS’te daha yüksek skor alır ama toplam çevrim sayısı daha fazla olabilir. Hatta aynı işlemci üzerinde bile farklı programlar farklı MIPS değerleri verir. Daha da kötüsü, bazı derleyici eniyilemeleri buyruk sayısını artırarak MIPS’i yükseltebilir ama programı yavaşlatabilir; Örnek 2.5’te bunu somut olarak gördük. Bu nedenle MIPS, işlemcilerin başarımlarını karşılaştırmak için tek başına güvenilir bir ölçüt değildir.

Tuzak: Tepe başarımlar gerçek başarımlar sanmak

Bir GPU’nun 80 TFLOPS tepe başarımlarına sahip olması, gerçek uygulamalarda bu değere ulaşacağı anlamına gelmez. Tepe başarımlar, tüm hesaplama birimlerinin her saat çevriminde tam kapasiteyle ve kesintisiz çalıştığı kuram düzeyinde bir üst sınırdır. Ancak gerçek uygulamalarda bellek bant genişliği darboğazları, veri bağımlılıkları ve kontrol akışı düzensizlikleri nedeniyle tepe başarımların genellikle %10–60’ı arasında bir sürdürülebilir başarımlarla sonuçlanır. Bir işlemciyi tepe başarımlarına bakarak seçmek, bir otomobili yalnızca en yüksek hızına bakarak seçmek gibidir: şehir içi trafikte 300 km/sa yapabilen spor otomobil, 180 km/sa yapabilen aile otomobilinden daha yavaş kalabilir.

Tuzak: Amdahl Yasası’nı göz ardı etmek

Bir mühendis, programın yalnızca %5’ini oluşturan bir birimi 100 kat hızlandırmak için aylarca çalışabilir. Amdahl Yasası’na göre bu çabanın karşılığı en fazla $\frac{1}{0,95+0,05/100} \approx 1,05$ kat hızlanma; yalnızca %5’lik bir iyileşme. Aynı sürede programın %60’ını oluşturan bileşeni 2 kat hızlandırmak $\frac{1}{0,4+0,6/2} \approx 1,43$ kat hızlanma sağlardı. Bu tuzak uygulamada sıkça karşılaşılar: mühendisler çoğu zaman en “ilginç” veya en “yavaş” bileşeni eniyileme eğilimindedir, oysa

Amdahl Yasası bize bileşenin mutlak yavaşlığının değil *toplam yürütme zamanı içindeki payının* önemli olduğunu söyler.

Yanılığ: Transistör sayısı arttıkça başarımlar aynı oranda artar

Moore Yasası transistör sayısını her iki yılda ikiye katlamıştır; ancak 2005'ten sonra eklenen transistörler tek çekirdek başarımlarına doğrudan katkı sağlamamaktadır. Ek transistörler çekirdek sayısını artırmak, önbellek boyutunu büyütmek veya GPU, sinir ağı motoru gibi özel amaçlı hızlandırıcılar eklemek için kullanılır. Bu birimlerin başarımlarına dönüşmesi, yazılımın koştürleştirilmesine veya ilgili hızlandırıcıyı kullanacak şekilde yazılmasına bağlıdır. Dolayısıyla “transistör sayısı iki katına çıktı, bilgisayar iki kat hızlandı” çıkarımı artık geçerli değildir.

Yanılığ: Enerji verimli işlemci her zaman yavaştır

Yüksek başarımlar ile düşük güç tüketiminin bir arada olamayacağı yaygın bir önyargıdır. Oysa Apple M serisi yongalar, ARM tabanlı sunucu işlemcileri (AWS Graviton, Ampere Altra) ve RISC-V tabanlı tasarımlar, daha az güç harcayarak rakipleriyle yarışan veya onları geçen başarımlar sunmaktadır. Enerji verimliliği, yalnızca pil ömrü için değil, veri merkezlerinin soğutma ve elektrik maliyeti açısından da doğrudan ekonomik bir kazanımdır.

2.10 Özet

Bu bölümde bilgisayar başarımlarını tanımlayan, ölçen ve sınırlayan temel kavramları inceledik. Başarımlar, sezgisel bir kavram gibi görünse de onu sayısal olarak ele almak için birden fazla farklı ölçüt gerekir; bu ölçütlerin hangisinin öne çıktığı, hangi iş yükü için değerlendirme yapıldığına bağlıdır.

Yürütme zamanı, başarımların en doğrudan ölçütüdür ve üç bileşenin çarpımından oluşur: buyruk sayısı, buyruk başına çevrim (BBC) ve çevrim zamanı. Bu üç bileşen birbirine bağlıdır; saat sıklığını artırmak çevrim zamanını düşürür, ancak derinleşen boru hattı BBC'yi yükseltebilir. Benzer biçimde, derleme eniyilemesi buyruk sayısını azaltırken BBC'yi artırabilir. Başarımlar artırmak, bu bileşenler arasındaki dengeyi bilerek kurmayı gerektirir.

İşlem hacmi ile gecikme, birbirini tamamlayan iki ayrı başarımlar boyutudur. Gecikme tek bir görevin ne kadar sürede tamamlandığını, işlem hacmi ise birim zamanda kaç görevin bitirildiğini ölçer. Etkileşimli uygulamalar, örneğin bir web tarayıcısı veya oyun, gecikmeye duyarlıdır; kullanıcı her tuş vuruşunda hızlı yanıt bekler. Sunucu iş yükleri ise genellikle işlem hacmini önceliklendirir; tek bir isteğin birkaç milisaniye geç yanıtlanması, günde milyonlarca isteği doğru biçimde karşılayabilmek kadar önemli değildir.

MIPS ve MFLOPS gibi ölçütler yanıltıcı olabilir, çünkü farklı buyruk kümesi mimarilerindeki buyruklar eşdeğer iş yapmaz. İki işlemci arasında MIPS değerleri üzerinden doğrudan karşılaştırma yapmak, birinin kilometreyle diğerinin mil ile hız ölçtüğü iki aracı karşılaştırmaya benzer. Kayan nokta işlemi sayısını esas alan FLOPS ölçütü, bilimsel hesaplama ve yapay zeka iş yüklerinde daha anlamlıdır; ancak burada da tepe başarımlar ile sürdürülebilir başarımlar arasındaki farka ve hangi sayısal duyarlılık düzeyinde çalışıldığına dikkat etmek gerekir. SPEC ve MLPerf gibi standart kıyaslama paketleri, gerçekçi iş yükleri üzerinde tekrarlanabilir ve adil karşılaştırma imkânı sunar.

Amdahl Yasası, bir iyileştirmenin toplam başarıma katkısını nicel olarak ortaya koyar: yalnızca programın P oranındaki kısmını H kat hızlandırırsanız toplam hızlanma $\frac{1}{(1-P)+P/H}$ ile sınırlıdır. Bu formülün temel çıkarımı “olağan durumu hızlandır” ilkesidir: programın büyük bölümünü oluşturan kısmı hızlandırmak, küçük bir bölümü büyük ölçüde hızlandırmaktan çok daha etkilidir. Gustafson Yasası ise farklı bir bakış açısı sunar: çekirdek sayısı arttıkça problem büyüklüğü de artırılırsa Amdahl’ın belirlediği karamsar üst sınır aşılabılır. Gerçek uygulamalarda büyük ölçekli bilimsel simülasyonlar bu modele yakın davranır.

Çağdaş işlemciler ve sistemler heterojen yapıdadır: tek bir yonga üzerinde grafik işlemciler, sinir ağı hızlandırıcıları ve video kod çözücüler bir arada bulunur. Bu hızlandırıcılar, iş yükünün belirli dilimlerini genel amaçlı işlemciden çok daha hızlı işler. Genişletilmiş Amdahl formülü bu durumu modellemek için kullanılır; MİB’de kalan iş oranı, tüm sistemin ulaşabileceği hızlanmayı sınırlar. Dolayısıyla heterojen sistemlerde iş yükünü farklı işlem birimlerine ne ölçüde aktarabildiğiniz, genel başarıma belirleyen en önemli etkenlerden biridir.

Bellek duvarı, işlemci hızının bellek hızından çok daha hızlı ilerlemesinin yarattığı darboğazı ifade eder. Önbellek hiyerarşisi ve yüksek bant genişlikli bellek (HBM) bu farkı hafifletmek için kullanılan başlıca araçlardır. Güç tüketimi de başarıma doğrudan bağlantılıdır: devingen güç, gerilimin karesiyle orantılıdır. Bu ilişki, saat sıklığı yerine koşutluğa yönelmenin temel fiziksel nedenidir ve Green500 listesindeki başarıma/watt ölçütünün neden bu denli önem kazandığını açıklar.

Başarıma eğilimleri 2005’ten bu yana köklü bir dönüşüm geçirmiştir. O güne kadar yılda yaklaşık %50 oranında seyreden tek çekirdek başarıma artışı, güç duvarının devreye girmesiyle %3–5 düzeyine gerilemiştir. Günümüzde başarıma artışı çok çekirdekli işlemciler, alana özgü hızlandırıcılar ve gelişmiş bellek hiyerarşileriyle sağlanmaktadır. Saat sıklığının tek başına belirleyici olduğu, MIPS’in güvenilir bir karşılaştırma ölçütü olduğu ve tepe başarımanın sürdürülebilir başarıma eşit sayılabileceği gibi yaygın yanılgılar, hem tasarım kararlarında hem de sınav sorularında sıkça karşılaşılan tuzaklardır.

Başarıma, yazılım ile donanım arasındaki arayüzde şekillenir. Bir programın kaç buyruk çalıştırdığı, bu buyrukların hangi kaynakları ne kadar süre kullandığı ve donanımın bu kaynakları nasıl organize ettiği birlikte başarıma belirler. Bir sonraki bölümde bu arayüzün temel sözleşmesini, buyruk kümesi mimarisini ele alacağız.

Alıştırmalar

Alıştırma 2.1. ★ Aşağıda verilen saat vuruşu sıklığı değerleri için çevrim zamanını pikosaniye ve saniye cinsinden hesaplayın: (a) 1 MHz, (b) 10 GHz, (c) 0,5 Hz.

Alıştırma 2.2. ★ Bir bilgisayarın başarıma hangi ölçüt (ya da ölçütler) kullanılarak ölçülür?

Alıştırma 2.3. ★ 500 MHz’lik işlemcisi olan X bilgisayarının üzerinde Calculator.java programı 10 sn’de çalışıyor. Eğer aynı programın daha hızlı çalışması için alınan Y bilgisayarı programı 5 sn’de çalıştırıyorsa Y’nin saat sıklığı nedir? (Her iki bilgisayarın aynı buyruk kümesini ve aynı BBÇ değerlerini kullandığını varsayın.)

Alıştırma 2.4. ★★ Elektrik-elektronik mühendisliği bölümünden bilgisayar mühendisliği bölümüne yan dal yapmaya gelen öğrencilerin temsilcisi Çorumlu Orhan ile ev sahibi bilgisayarlıların temsilcisi Edirneli Orhan koşu yarışı yapacaklardır. Koşulacak mesafe 400 metredir. Ço-

rumlunun her koşu adımı 1,6 metre, Edirnelinin her koşu adımı 1 metredir. Çorumlu dakikada 60, Edirneli ise dakikada 120 adım atıyorsa:

- Hangi Orhan yarışı kazanır? Kazanan Orhan diğerinden ne kadar daha hızlıdır?
- Eğer yenilen Orhan dakikada %25 daha fazla adım atmaya başlarsa yarışın sonucu değişir mi?
- Bu sorudaki yarış mesafesi, koşu adımı ve dakikada atılan adım sayısı değişkenlerinin bilgisayar mimarisindeki karşılıkları nelerdir?

Alıştırma 2.5. ★★ Duygu'nun bilgisayarı A buyruk kümesini, Gülşah'ın bilgisayarı ise B buyruk kümesini kullanıyor. 100 buyruklu programın %50'si X, %10'u Y, %18'i Z, %22'si W buyruğundan oluşmaktadır. Her iki buyruk kümesinin BBÇ değerleri aşağıdaki tabloda verilmiştir.

Buyruk türü	A kümesi BBÇ	B kümesi BBÇ
X	1	2
Y	4	3
Z	2	3
W	5	4

- İkisi de aynı programı aynı sürede çalıştırmak istediklerine göre Duygu'nun bilgisayarının saat sıklığı Gülşah'ınkinin kaç katı olmalıdır?
- Her iki bilgisayar da A buyruk kümesini kullansaydı kimin bilgisayarının başarıımı yüksek olacaktı?

Alıştırma 2.6. ★★ Amdahl Yasası'na göre, bir programın %40'ını oluşturan kayan nokta işlemlerinin 8 kat hızlandırılması toplam başarıımı kaç kat artırır? Aynı hızlanmayı bütün programa yaymak için kayan nokta oranının ne olması gerekirdi?

Alıştırma 2.7. ★★ Bir yapay zeka hızlandırıcısı FP16 duyarlığında 312 TFLOPS tepe başarıma sahip. Ancak bellek bant genişliği 2 TB/s ile sınırlıdır. Eğer bir matris çarpımı işlemi her 2 baytlık veri okuması için 128 kayan nokta işlemi gerçekleştiriyorsa bu iş yükü hesaplama sınırlı mı yoksa bellek sınırlı mı? Aritmetik yoğunluk kavramını açıklayarak yanıtlayın.

Alıştırma 2.8. ★★ Bir GPU'nun 10 240 çekirdeği, 2 GHz saat sıklığı var ve her çekirdek çevrim başına 2 FP32 işlemi gerçekleştirebiliyor.

- FP32 tepe başarıımını TFLOPS cinsinden hesaplayın.
- Aynı GPU FP16 işlemlerinde çekirdek başına çevrim başına 4 işlem yapabiliyorsa FP16 tepe başarıımı nedir?
- LINPACK sınavında sürdürülebilir başarıım tepe değerinin %55'i olarak ölçülmüştür. Sürdürülebilir FP32 başarıımı kaç TFLOPS'tur?

Alıştırma 2.9. ★★ Bir web tarayıcısının çalışma zamanı üç bileşene ayrılmaktadır: %30 JavaScript yürütme, %50 sayfa oluşturma (rendering) ve %20 ağ bekleme. Aşağıdaki üç iyileştirme senaryosunun her biri için Amdahl Yasası'nı kullanarak toplam hızlanmayı hesaplayın:

- JavaScript motoru 4 kat hızlandırıldı.

- (b) Sayfa oluşturma GPU ile 10 kat hızlandırıldı.
 (c) Her üç bileşen de aynı anda iyileştirildi: JavaScript 4×, sayfa oluşturma 10×, ağ bekleme 2×.

Her senaryo için darboğazın nereden kaynaklandığını yorumlayın.

Alıştırma 2.10. ★★★ SPEC CPU2017 ölçüt paketinde aşağıdaki üç tamsayı programı için iki işlemcinin puanları verilmiştir:

Program	Referans süresi (s)	İşlemci X süresi (s)	İşlemci Y süresi (s)
500.perlbench	1 600	200	160
505.mcf	1 800	360	300
520.omnetpp	1 000	250	200

- (a) Her program için her iki işlemcinin SPEC oranını hesaplayın.
 (b) Her işlemcinin geometrik ortalama SPEC puanını bulun.
 (c) Hangisi daha hızlıdır ve ne kadar?

Alıştırma 2.11. ★ Aşağıdaki cümlelerde boş bırakılan yerleri doldurun ya da doğru/yanlış (D/Y) olarak yanıtlayın.

- (a) Saat vuruş sıklığı, saatin iki yükselen darbesi arasındaki geçen süreye denir. (D/Y)
 (b) İşlemcilerde saat sıklığını devredeki _____ belirler.
 (c) Tek vuruşluk bir işlemcinin başarımı, boru hattı kullanan bir işlemciden her zaman düşüktür. (D/Y)
 (d) Çevrim zamanı, işlemcideki kritik yol gecikmesi tarafından belirlenir. (D/Y)
 (e) MIPS ölçütü, farklı buyruk kümelerine sahip iki işlemciyi karşılaştırmak için güvenilir bir birimdir. (D/Y)

Alıştırma 2.12. ★ A işlemcisinin saat vuruş sıklığı 4 GHz ve ortalama buyruk başına çevrimi (BBÇ) 4, B işlemcisinin saat vuruş sıklığı 2 GHz ve ortalama BBÇ değeri 2 ise aynı programı hangi işlemci daha hızlı çalıştırır? Yürütme zamanlarını hesaplayarak karşılaştırın.

Alıştırma 2.13. ★★ Aşağıdaki analogide, bir üniversitenin ders programı ile işlemci başarımları parametreleri arasındaki benzerlikleri bulun.

ODTÜ’de dönemler 14 hafta sürmekte ve her haftada 5 gün ders yapılmaktadır. TOBB ETÜ’de ise dönemler yaklaşık 12 hafta sürmekte ve her haftada 6 gün ders yapılmaktadır. Aynı müfredatı uygulayan her iki okul da 70 ders gününü tamamlamayı hedefliyorsa, 3 kredilik bir ders ODTÜ’de haftada 3 saat, TOBB ETÜ’de ise ortalama 3,5 saat olmak üzere toplam 42 saatte yapılmaktadır.

Bilgisayar Mimarisi Kavramı	Sorudaki Benzeri
Buyruk Kümesi Mimarisi	_____
Programdaki Buyruk Sayısı	_____
Programın Yürütme Zamanı	_____
Çevrim	_____
Çevrim Zamanı	_____

Alıştırma 2.14. ★★★ Farklı buyruk türlerinden oluşan bir P programı, 1 GHz saat sıklığına sahip bir işlemcide çalıştırıldığında yürütme zamanının 17 saniye olduğu ölçülüyor.

- Program üzerinde eniyileme çalışması yapan bir ekip, her biri 5 çevrim süren A türü buyrukların yerine, her biri 1 çevrim süren B türü 3'er buyruk koyarak yürütme zamanını 16 saniyeye düşürüyor. Kaç adet A türü buyruk, B türü buyruklarla değiştirilmiştir?
- Araştırmalarına devam eden ekip, eniyilenmiş programı aynı saat sıklığına sahip ancak bölme işlemlerini beş kat hızlı yapabilen yeni bir sistemde çalıştırdığında yürütme zamanının 15 saniyeye düştüğünü gözlemliyor. Programda n adet bölme işlemi varsa, yeni sistem bir bölme işlemini kaç çevrimde tamamlamaktadır?

Alıştırma 2.15. ★★★ Yusuf yeni bir bilgisayar almak istiyor. Araştırmasının sonucunda Y, C ve T adında üç bilgisayar buluyor:

- Y bilgisayarı:** 5 MHz saat sıklığı, tek çevrimlik işlemci, fiyatı 20 000 TL.
 - C bilgisayarı:** 25 MHz saat sıklığı, çok çevrimlik işlemci (çarpma buyruğu 5 çevrim, bellek buyrukları 6 çevrim, diğer buyruklar 4 çevrim), fiyatı 25 000 TL.
 - T bilgisayarı:** 20 MHz saat sıklığı, 4 aşamalı boru hattı (çarpma 2 çevrim, bellek 3 çevrim, diğerleri 1 çevrim), fiyatı 30 000 TL.
- $N = 1000$ yinelemeli bir döngü çalıştırıldığında her bilgisayarda program kaç saniyede tamamlanır? (Döngüde 2 bellek buyruğu, 1 çarpma buyruğu ve 4 aritmetik buyruk olduğunu varsayın.)
 - Yusuf fiyat-başarım oranına göre karar vermek istiyor. "TL başına başarımlar" ölçütü nasıl hesaplanır? Bu ölçüte göre bilgisayarları sıralayın.

Alıştırma 2.16. ★★ Bir programın yürütme süresinin %70'i tek çekirdekte, %30'u ise koşut bölümde harcanmaktadır. Amdahl Yasası'nı kullanarak:

- 4 çekirdekli bir işlemcideki hızlanmayı hesaplayın.
- 64 çekirdekli bir işlemcideki hızlanmayı hesaplayın.
- Sonsuz sayıda çekirdek kullanılsa bile elde edilebilecek en yüksek hızlanma nedir?
- Gustafson Yasası bu sonucu nasıl farklı yorumlar? Kısaca açıklayın.

Alıştırma 2.17. ★★ İki bilgisayarın SPEC CPU 2017 SPECrate sonuçları şöyledir: A bilgisayar 320, B bilgisayar 280. SPECspeed sonuçları ise A için 85, B için 110'dur. Hangi bilgisayar "daha hızlı" denir? Yanıtınızı SPECrate ve SPECspeed'in ölçtüğü farklı başarımlar boyutlarıyla açıklayın.

Alıştırma 2.18. ★★ Bir iklim modelleme programı 128 çekirdekli bir sistemde çalıştırılmaktadır. Koşut yürütme sırasında ölçülen seri kısmın oranı $s = 0,02$.

- Gustafson Yasası'na (Denklem 2.20) göre ölçeklenmiş hızlanmayı hesaplayın.
- Aynı s değerini Amdahl Yasası'na uygularsanız hızlanma ne olur?
- İki sonuç arasındaki farkı açıklayın. Hangi yasa bu senaryo için daha gerçekçidir?

Alıştırma 2.19. ★ İşlem hacmi ve gecikme kavramlarını bir restoran benzetmesiyle açıklayın. Bir restoran mutfağının yemek başına hazırlık süresi 15 dakikadır (gecikme). Mutfakta 4 aşçı çalışmaktadır.

- (a) Restoranın saatlik işlem hacmi (tabak/saat) nedir?
- (b) 8 aşçıya çıkılırsa yemek başına gecikme değişir mi? İşlem hacmi ne olur?
- (c) Bu benzetmede “aşçı sayısı” bilgisayar mimarisindeki hangi kavrama karşılık gelir?

Alıştırma 2.20. ★★ 1995 yılında bir işlemcinin saat sıklığı 200 MHz, DRAM erişim süresi 120 ns idi.

- (a) İşlemci bir saat çevriminde kaç nanosaniye harcar? Bellek erişimi kaç çevrime karşılık gelir?
- (b) 2025’te saat sıklığı 5 GHz, DRAM erişim süresi 40 ns olmuştur. Aynı hesabı tekrarlayın.
- (c) Sonuçları karşılaştırarak “bellek duvarı” kavramının zaman içinde nasıl derinleştiğini sayısal olarak açıklayın.
- (d) L1 önbellek erişim süresi 1 ns ise 2025 için önbellekte bulma oranının en az ne kadar olması gerekir ki ortalama bellek erişim süresi 10 çevrimden az olsun?

Alıştırma 2.21. ★★★ Üç sunucu işlemcisi karşılaştırılıyor:

	İşlemci P	İşlemci Q	İşlemci R
Saat sıklığı	3,5 GHz	2,8 GHz	3,0 GHz
Çekirdek sayısı	8	32	16
SPEC rate skoru	180	420	310
TDP (Watt)	125	250	150

- (a) Hangi işlemci tek çekirdek başarımında en iyidir? (İpucu: SPECrate / çekirdek sayısı)
- (b) Hangi işlemci en yüksek enerji verimliliğine (SPECrate/Watt) sahiptir?
- (c) Bir veri merkezi yöneticisi toplam 1000 SPECrate skoru elde etmek istiyor. Her işlemci-den kaç tane alması gerekir ve hangi seçenek en az güç harcar?

Alıştırma 2.22. ★★ Bir yazılım ekibi, bir uygulamanın başarımını artırmak için iki farklı eniyileme stratejisi değerlendiriyor:

- **Strateji A:** Programın %30’unu oluşturan bellek erişim bölümünü önbellekleme ile 4 kat hızlandırmak.
- **Strateji B:** Programın %70’ini oluşturan hesaplama bölümünü algoritmik iyileştirme ile 2 kat hızlandırmak.

- (a) Her iki strateji için Amdahl Yasası’nı kullanarak hızlanmayı hesaplayın.
- (b) Her iki strateji aynı anda uygulanabilseydi toplam hızlanma ne olurdu?
- (c) “Olağan durumu hızlandır” ilkesi açısından hangi strateji daha değerli? Neden?

Alıştırma 2.23. ★★ Dört farklı yapay sına programındaki hızlanma değerleri şöyledir: 0,8×, 2,5×, 1,1× ve 12,0×.

- (a) Aritmetik ortalama hızlanmayı hesaplayın.
- (b) Geometrik ortalama hızlanmayı hesaplayın.
- (c) Hangi ortalama daha anlamlıdır? Neden?

- (d) $0,8 \times$ “hızlanma” ne anlama gelir? Bu durumda yeni tasarım eski tasarımdan daha iyi mi?

Alıştırma 2.24. ★★ Aşağıdaki ifadelerin her birinin bir yanlış mı yoksa doğru bir gözlem mi olduğunu belirleyin ve kısaca açıklayın:

- (a) “6 GHz işlemci, 3 GHz işlemciden her zaman 2 kat hızlıdır.”
- (b) “MIPS değeri yüksek olan bilgisayar her iş yükünde daha hızlıdır.”
- (c) “Tepe başarımı 100 TFLOPS olan GPU, gerçek uygulamada 100 TFLOPS verir.”
- (d) “Amdahl Yasası’na göre sonsuz çekirdek kullanılsa bile hızlanmanın bir üst sınırı vardır.”
- (e) “Transistör sayısı iki katına çıkarsa başarımlar da iki katına çıkar.”

Alıştırma 2.25. ★★★ Intel Pentium 4 (2004, 3,8 GHz) ile Intel Core 2 (2006, 2,4 GHz) işlemcilerini düşünün. Core 2, Pentium 4’ten daha düşük saat sıklığına sahip olmasına rağmen birçok iş yükünde daha hızlıydı.

- (a) Bu nasıl mümkün olabilir? Başarım denkleminin hangi bileşeni bunu açıklar?
- (b) Pentium 4’ün 31 aşamalı, Core 2’nin 14 aşamalı boru hattı olduğunu biliyorsanız derin boru hattının saat sıklığına ve BBÇ’ye etkisini tartışın.
- (c) Bu örnek, güç tüketimi açısından ne gösterir? (İpucu: Pentium 4 115 W, Core 2 65 W harcamaktadır.)

Alıştırma 2.26. ★★★ Aşağıdaki tabloda C0 ve C1 işlemcilerinin özellikleri listelenmektedir.

Özellik	C0 İşlemcisi	C1 İşlemcisi
Saat vuruş sıklığı	2,1 GHz	2,8 GHz
Toplama BBÇ	2	3
Çarpma BBÇ	3	5
Dallanma BBÇ	7	6
Bellek BBÇ	15	10

- (a) 700 toplama, 200 çarpma, 300 dallanma ve 250 bellek buyruğundan oluşan DOTA-1500 programının C0 ve C1 işlemcilerinde yürütülmesi kaç nanosaniye sürer? Hangisi daha hızlıdır ve ne kadar?
- (b) Uzun bir incelemeden sonra C0 ve C1 işlemcisinde çalışan programların ortalama %35 toplama, %20 çarpma, %15 dallanma ve %30 bellek buyruklarından oluştuğunu gözlemlediniz. Buna göre aynı programları yürüten C0 işlemcisinin başarımının C1 ile aynı olması için C0’ın dallanma buyruklarını ortalama kaç çevrimde yürütmesi gerekmektedir?
- (c) C0 işlemcisi saat vuruş sıklığını devingen olarak 2,1 GHz’den 3,0 GHz’ye artırabilmektedir. Ancak bu işlemi yalnızca bir buyruk türü için gerçekleştirebilmektedir. Eşit sayıda toplama, çarpma, dallanma ve bellek buyruğu içeren bir program için C0 işlemcisi hangi buyruk tipinde bu işlemi uygularsa en yüksek başarımların gözlemlenir?

Alıştırma 2.27. ★★★ Aşağıdaki tabloda K0 ve K1 işlemcilerinin özellikleri listelenmektedir.

Özellik	K0 İşlemcisi	K1 İşlemcisi
Saat vuruş sıklığı	2 GHz	1 GHz
Toplama BBÇ	3	4
Bölme BBÇ	3	6
Bellek BBÇ	X	15
Dallanma BBÇ	8	7

- (a) $X = 19$ için; 400 toplama, 700 bölme, 100 dallanma ve 650 bellek buyruğundan oluşan Ka-Fa programının K0 ve K1 işlemcilerinde yürütülmesi kaç nanosaniye sürer? Hangi işlemci diğerinden kaç kat hızlıdır?
- (b) K0 ve K1 işlemcisinde çalışan programların ortalama %5 toplama, %60 bölme, %20 dallanma ve %15 bellek buyruklarından oluştuğu biliniyor. Buna göre aynı programları yürüten bu işlemcilerden K0 işlemcisinin başarımı K1 ile aynı olabilir mi? Olabiliyorsa K1 ile aynı olması için K0'ın bellek buyruğu başına çevrim sayısı (X) kaç olmalıdır?

Alıştırma 2.28. ★★★ Aşağıdaki tabloda Break0 ve Break1 işlemcilerinin özellikleri listelenmektedir. Break1 işlemcisi çarpma işlemi yapamamaktadır.

Özellik	Break0 İşlemcisi	Break1 İşlemcisi
Saat vuruş sıklığı	1 GHz	200 MHz
Toplama BBÇ	5	1
Çarpma BBÇ	8	–
Dallanma BBÇ	12	15
Bellek BBÇ	20	5

PaCMAN programı Break0 işlemcisi için derlendiğinde 100 toplama, 300 çarpma, 400 dallanma ve 50 bellek buyruğundan oluşmaktadır. Break1 işlemcisi çarpma yapamadığından, derleyici çarpma işlemlerini toplama buyrukları ile gerçekleştirmektedir; bu durumda program 900 toplama, 200 dallanma ve 50 bellek buyruğundan oluşmaktadır.

- (a) Break0 ve Break1 işlemcileri PaCMAN programını kaç nanosaniyede yürütür?
- (b) PaCMAN programının iki işlemcide de aynı sürede bitmesi için Break1 işlemcisinin saat vuruş sıklığı kaç MHz olmalıdır?

Alıştırma 2.29. ★★ Aşağıdaki tabloda Chronus ve PaCRAM işlemcilerinin özellikleri verilmiştir.

Özellik	Chronus	PaCRAM
Saat vuruş sıklığı	1 GHz	500 MHz
Aritmetik BBÇ	10	3
Dallanma BBÇ	8	3
Bellek BBÇ	16	12

OUS programı bu iki işlemcide çalıştırılmak istenmektedir. OUS programı %45 aritmetik, %35 dallanma ve %20 bellek buyruklarından oluşmaktadır.

Bu iki işlemcinin OUS programını eşit sürede çalıştırabilmesi için Chronus işlemcisinin aritmetik işlemlerini yüzde kaç hızlandırmalıyız ya da yavaşlatmalıyız?

Alıştırma 2.30. ★★★ Yusuf yeni bir bilgisayar almak istiyor. Araştırdıktan sonra Y, C ve T adında üç farklı bilgisayar buluyor. Aşağıdaki kod parçacığını bu üç bilgisayarda çalıştırıp sonuçları karşılaştırmak istiyor.

```
addi x1, x0, #N
addi x2, x0, #0x8000
addi x4, x0, #1
addi x5, x0, #0
dongu:
    ld    x8, 0(x2)
    ld    x9, 8(x2)
    add   x2, x0, x9
    mul   x4, x4, x8
    addi  x5, x5, #1
    blt   x5, x1, <dongu>
    add   x6, x0, x4
```

Y bilgisayarı: 5 MHz saat sıklığı, tek çevrimlik işlemci, fiyatı 20 bin TL.

C bilgisayarı: 25 MHz saat sıklığı, çok çevrimlik işlemci (çarpma 5 çevrim, bellek 6 çevrim, diğer buyruklar 4 çevrim), fiyatı 25 bin TL.

T bilgisayarı: 20 MHz saat sıklığı, boru hattı olan işlemci. Boru hattı Getir (G), Çöz (Ç), yürüt (Ü), geri Yaz (Y) olmak üzere 4 aşamadan oluşur. Yürüt aşaması buyruk tiplerine göre farklı çevrimlerde gerçekleştirilir: çarpma 2 çevrim, bellek 3 çevrim, diğer buyruklar 1 çevrim. Yürüt aşamasında tek bir buyruk olabilir. Dallanmalar her zaman doğru öngörülür. Veri yönlendirmesi yoktur. Yazmaçlar yazıldığı çevrimde okunabilir. Fiyatı 30 bin TL.

- $N = 1$ için kod parçacığını dongu etiketinden itibaren T bilgisayarıda çalıştırdığınızda oluşan boru hattı şemasını çizin.
- $N = 1000$ için kod parçacığını Y, C ve T bilgisayarlarında yürüttüğünüzde kaç saniyede tamamlanır? (Yalnızca döngüyü kullanarak hesaplayın.)
- Yusuf fiyat–performans oranını maksimize etmek istiyor. “TL başına performans” ölçütünü nasıl hesaplayabileceğinizi gösterin ve bilgisayarları sıralayın.
- Bu kod parçacığı ne yapıyor?

Alıştırma 2.31. ★★ Verilen bir programdaki buyruklar ve kullandıkları işlemci zamanı oranları aşağıdaki gibidir:

Buyruk türü	İşlemci zamanı oranı
Yükle	%20
Sakla	%15
Dallan, Atla	%20
Kayan Nokta	%5
Aritmetik	%40

Eğer programdaki buyruk sayısı ve işlemcinin saat sıklığı değişmez ise:

- (a) Kayan nokta işlemlerinin hızını 5 katına çıkarmak işlemciyi ne kadar hızlandırır?
- (b) İşlemcinin %20 daha hızlı çalışması için aritmetik işlemlerinin kaç kat hızlanması gerekir?
- (c) Hızı öncekinin 7 katı olan bir bellek birimi alınırsa program ne kadar daha hızlı çalışır? (Bellek birimi yalnızca bellekle işi olan “Yükle” ve “Sakla” buyruklarını etkiler.)

Alıştırma 2.32. ★ Eğer bir program bilgisayar üzerinde 100 saniyede çalışıyorsa ve bu zamanın 80 saniyesi çarpma işlemi için harcanıyorsa, tüm programın 4 kat hızlı çalışması için çarpma işleminin ne kadar hızlandırılması gerekir?

Alıştırma 2.33. ★ “Kasırğa” marka bir işlemcinin saat sıklığı 4 GHz, “Fırtına” marka işlemcinin saat sıklığı 2 GHz ise ve bu işlemcilerin üzerinde aynı program çalışırsa hangisi daha hızlı çalışır? Verilen saat sıklığı bilgileri bu iki işlemciyi birbiriyle karşılaştırmak için yeterli midir? Eğer yeterli ise Kasırğa, Fırtına’dan ne kadar daha hızlıdır? Eğer yeterli değilse Kasırğa ile Fırtına’nın aynı hızda çalışması için ne olması gerekir? Yanıtınızı gerekçeleriyle birlikte açıklayın.

Alıştırma 2.34. ★★★ Hintel firmasının yeni çıkardığı “Canavar” model işlemci 10 GHz hızında çalışıyor ve Cintel firmasının yeni çıkardığı “Fırtına” model işlemci 2 GHz hızında çalışıyor ise:

- (a) Her iki firmanın işlemcilerinin kullandığı saatlerin iki vuruşu arasında kaç pikosaniye geçer?
- (b) Canavar, Fırtına’dan ne kadar hızlıdır?
- (c) Eğer aynı programı Fırtına’nın Canavar’dan 2 kat hızlı çalıştırması istenirse Cintel firmasının nasıl bir cinlik düşünmüş olması gerekir?
- (d) Eğer Canavar her buyruk türünü aynı sayıda çevrimde tamamlarken Fırtına “önkayıt” türündeki buyrukların dışındaki buyrukları Canavar ile aynı sayıda çevrimde tamamlıyorsa ve Fırtına önkayıt buyruklarını Canavar’dan 10 kat daha hızlı çalıştırabiliyorsa, Fırtına’nın verilen bir sinama programını Canavar ile aynı zamanda bitirebilmesi için programın içindeki önkayıt buyruklarının oranı ne olmalıdır?
- (e) Cintel önkayıt buyruklarıyla ilgili hızlandırmayı yaparken hangi tasarım ilkesini kullanmıştır?

Alıştırma 2.35. ★★★ Tasarlanan bir bilgisayarda Aritmetik (A), Bellek (B), Canavar (C) ve Dallenma (D) olmak üzere 4 tür buyruk çalışmaktadır. Bilgisayarın saat sıklığı 1 GHz olarak belirlenmiştir. Bilgisayarın üzerinde Tulpar ve Umay derleyicileri ile derlenen kodlar denenecektir. Tulpar %20 A, %30 B, %35 C ve %15 D türünde buyruklar üretirken Umay %40 A, %10 B, %25 C ve %25 D türünde buyruk üretmektedir.

- (a) Bilgisayarda tek vuruşluk bir işlemci kullanılırsa ve derleyiciler tarafından “Karşı Saldırı” adlı program derlendiğinde Tulpar 1000, Umay ise 800 buyruk üretiyorsa Tulpar’ın ürettiği kod Umay’ın ürettiği koddan ne kadar daha yavaştır?
- (b) Bilgisayarda çok vuruşluk bir işlemci kullanılırsa ve A, B, C ve D türü buyruklar sırasıyla 4, 5, 3 ve 4 vuruşta tamamlanabilirse “Karşı Saldırı” adlı oyunu hangi derleyici daha hızlı çalıştırır? Bu durumda Tulpar, Umay’dan ne kadar daha hızlıdır?

- (c) Bilgisayarda 15 aşamalı boru hattı bulunan bir işlemci kullanılırsa ve D türü (dallanma) buyrukların koşulları 11'inci aşamada çözülüyorsa, "Karşı Saldırı" adlı program için hangi derleyicinin ürettiği kod bilgisayarda daha hızlı çalışır? Bu durumda hızlı olan kod yavaş olandan ne kadar daha hızlıdır?

Alıştırma 2.36. ★ Bir bilgisayarın kullandığı saat sıklığı sabitse, "Karşı Saldırı" isimli oyunun bu bilgisayarda 5 kat hızlı çalışması için derleyici aşamasında hangi önlemler alınabilir?

Alıştırma 2.37. ★★★ Aşağıda bir işlemcinin çalıştırabildiği buyruk türleri, bu buyrukların kaç çevrimde çalıştığı ile A ve B derleyicilerinin aynı programı derlediklerinde ortaya çıkan kodlarda bu buyrukların ne oranda bulunduğu gösterilmiştir:

Buyruk türü	Çevrim sayısı	A (%)	B (%)
Yükle	7	10	15
Kayan Nokta	10	5	10
Sakla	6	10	5
Aritmetik	4	25	35
Dallan, Atla	3	20	5
Ninja	12	10	15
Ulubatlı Hasan	15	20	15

- (a) Eğer B'nin çıkardığı buyruk sayısı A'nın çıkardığından %20 fazlaysa A'nın kodu B'nin kodundan ne kadar daha hızlı çalışır?
- (b) A derleyicisi ile B derleyicisinin ürettiği kodların işlemcide aynı zamanda çalışması için ürettikleri buyrukların sayısının oranı ne olmalıdır?
- (c) Ulubatlı Hasan ve Ninja buyruklarının çevrim sayısı birbirine eşitlenir ve A ve B derleyicisinin ürettiği kodların aynı zamanda koşması istenirse, birbirine eşit çevrim sayısında çalışan Ulubatlı Hasan ve Ninja kaç çevrim tutmalıdır?

Buyruk Kümesi Mimarisi



Süleymaniye Camisi – İstanbul

Mimar Sinan, Süleymaniye'yi sınırlı sayıda yapı elemanının ustaca birleşimiyle inşa etti. Buyruk kümesi mimarisi de aynı ilkeye dayanır: az sayıda basit buyrukla her türlü hesaplamayı ifade edebilmek.

Öğrenme Hedefleri

Bu bölümü tamamladığınızda aşağıdakileri yapabileceksiniz:

- Buyruk kümesi mimarisinin (BKM) ne olduğunu ve yazılım ile donanım arasındaki sözleşme rolünü açıklamak.
- RISC-V'in modüler uzantı yapısını ve temel buyruk kümesi RV32I'nin kapsamını tanımlamak.
- Yazmaç sayısı, buyruk uzunluğu ve adresleme kipi sayısı gibi tasarım kararlarındaki ödünleşmeleri tartışmak.
- RISC-V'in altı buyruk biçimini (R, I, S, B, U, J) tanıyıp verilen bir buyruğun ikili kodlamasını çözmek.
- Aritmetik, mantık, kaydırma, bellek erişim ve dallanma buyruklarını kullanarak basit programlar yazmak.
- Adresleme kiplerini (yazmaç, anlık, eklemeli, PS'ye göreceli, üst anlık) ayırt etmek ve her birinin hangi üst düzey dil yapısına karşılık geldiğini göstermek.
- Sözde buyrukların nasıl gerçek buyruklara dönüştürüldüğünü ve işlem kodu uzayını neden verimli kullandığını açıklamak.
- Çağırma alışkanlıklarını (çağırıcı/çağırılan taraf sorumlulukları, giriş/çıkış kodu) uygulayarak altyordam yazmak.
- Bellek düzenini (.text, .data, .bss, yığın, yığıt) ve programın derleme zincirini (derleyici → çevirici → bağlayıcı → yükleyici) açıklamak.
- RISC-V, MIPS, ARM ve x86 mimarilerini buyruk sayısı, adresleme kipleri ve tasarım felsefesi açısından karşılaştırmak.

3.1 Giriş

Bir bilgisayara “iki sayıyı topla” ya da “bellekteki şu veriyi oku” dediğimizde aslında ne olur? İşlemci bu emirleri nasıl anlar? İşte tam bu noktada karşımıza **buyruk** (*instruction*) kavramı çıkar. Yazılım ile donanım arasındaki iletişimi sağlayan bu emirlerin tümüne **buyruk kümesi** (*instruction set*) denir. Nasıl ki iki insan arasındaki iletişim ortak bir dile dayanıyorsa, yazılım ile donanım arasındaki iletişim de buyruk kümesine dayanır. Bilgisayar yalnızca 0 ve 1'lerle çalıştığı için her buyruk da sonuçta bir bit dizisinden ibarettir.

Buyrukların yapısı, yazmaçların sayısı ve kullanım kuralları, bellek erişim yöntemleri ve adresleme kipleri bir arada düşünüldüğünde, yazılım ile donanım arasında bir sözleşme oluşur. Bu sözleşmeye *buyruk kümesi mimarisi* (BKM; İng. *Instruction Set Architecture, ISA*) adı verilir. BKM, bir işlemcinin hangi buyrukları tanıdığını, verileri nasıl sakladığını ve programcıya hangi kaynakları sunduğunu tanımlar; kısacası donanımın yazılıma sunduğu arayüzdür.

Bu kitapta buyruk kümesi mimarisi olarak açık kaynaklı **RISC-V** kullanılmaktadır. RISC-V'in temiz, modüler ve geriye uyumluluk yükünden arınmış yapısı, buyruk kümesi kavramlarını öğrenmek için ideal bir zemin oluşturur.

Bölüm boyunca RISC-V ana eksen olarak ele alınacak; ancak her önemli kavram, x86, ARM ve MIPS mimarileriyle karşılaştırılarak sunulacaktır. Karşılaştırmalar, ilgili konunun hemen ardından ayrı kutular içinde verilmiştir. Böylece okuyucu, bir kavramı RISC-V üzerinden öğrenirken diğer mimarilerin aynı soruna nasıl yaklaştığını da görebilecektir.

Terim Kökenleri

buyruk (*instruction*): *Buyurmak* fiilinden *-uk* ekiyle türetilmiştir; tarihte padişah fermanları ve dini emirler için kullanılan bu sözcük, işlemcinin doğrudan yürüttüğü makine düzeyindeki her emri ifade eder. Bazı Türkçe kaynaklarda *instruction* karşılığı olarak *komut* kullanılır; ancak *komut*, İngilizcedeki *command* sözcüğünün karşılığıdır ve kullanıcının sisteme verdiği üst düzey emirleri (kabuk komutu, menü komutu vb.) tanımlar. Bu kitapta makine düzeyindeki emirler için tutarlı biçimde *buyruk* tercih edilmiştir.

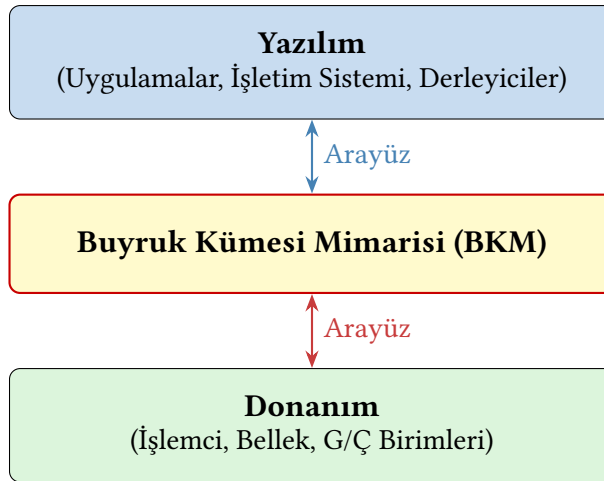
yazmaç (*register*): *Yazmak* fiiline eski Türkçe araç yapım eki *-maç* eklenerek türetilmiştir. Verinin geçici olarak “yazıldığı” küçük, hızlı saklama birimidir.

Bilgi Notu 3.1: Sayı Sistemleri ve Kodlama

İşlemciler yalnızca 0 ve 1’lerle çalıştığı için buyruk kümesi mimarisinin her yönü (buyruk kodlama biçimleri, anlık değerlerin gösterimi, bellek adresleme, işaretli ve işaret-siz aritmetik) sayı sistemlerine dayanır. Bir buyruğun bit alanlarını çözümlemek ikilik tabana, bellek adreslerini okumak onaltılık tabana, işaretli tamsayıları anlamak ikiye tümleyen gösterimine hâkim olmayı gerektirir. Bu kavramlar Ek A’da ayrıntılı olarak ele alınmaktadır; bu bölüme başlamadan önce gözden geçirilmesi önerilir.

3.2 Buyruk Kümesi Tasarımı

Bir bilgisayarı katmanlar halinde düşünebiliriz: üstte yazılım (uygulamalar, işletim sistemi), altta ise donanım (işlemci, bellek, giriş/çıkış birimleri) yer alır. Bu iki katmanın birbirleriyle “konuşabilmesi” için ortak bir dile ihtiyaçları vardır; bu ortak dil, **buyruk kümesi mimarisi** (BKM) arayüzüdür.



Şekil 3.1: Buyruk Kümesi Mimarisi (BKM) ile Yazılım ve Donanımın İlişkisi

Şekil 3.1’de görüldüğü gibi BKM, yazılım ile donanım arasında arayüz görevi üstlenir. BKM’yi bir *sözleşme* olarak düşünmek yararlıdır. Bu sözleşme; hangi buyrukların var olduğunu, her buyruğun hangi biçimde kodlandığını, yazmaç sayısını ve adreslenebilir bellek düzenini tanımlar. Sözleşmenin iki tarafı vardır: bir yanda derleyiciler ve işletim sistemleri, öte

yanda işlemciyi tasarlayan donanım mühendisleri. Her iki taraf da yalnızca bu sözleşmeye uyarak, birbirlerinin iç ayrıntılarını bilmeden bağımsız olarak çalışabilir. GCC derleyicisini yazan bir programcı Intel'in boru hattı tasarımını bilmek zorunda değildir; Intel'in donanım mühendisi de GCC'nin eniyileme algoritmalarını bilmek zorunda değildir; her ikisi de x86 BKM sözleşmesine uyduğu sürece uyumluluk sağlanır.

Bu soyutlama katmanı güçlü bir esneklik sunar: aynı BKM sözleşmesini farklı donanımlar gerçekleyebilir. Örneğin RISC-V BKM'sini hem düşük güçlü bir gömülü denetleyici hem de yüksek başarılı, çok çekirdekli bir sunucu işlemcisi gerçekleyebilir; her iki durumda da aynı derlenmiş program sorunsuz çalışır. Ancak bu esnekliğin bir bedeli vardır: bir BKM sözleşmesi geniş çapta benimsendikten sonra onu değiştirmek son derece güçtür. x86 mimarisi 1978'den bu yana geriye uyumluluğunu sürdürmek zorundadır; o tarihte yazılmış bir programın bugünkü işlemcide çalışabilmesi gerekir. Bu geriye uyumluluk yükü, BKM tasarımının neden bu denli özenli yapılması gerektiğini açıkça gösterir.

Bölüm 1'de gördüğümüz gibi, işlemci her buyruğu üç temel aşamadan geçirir: önce bellekten *getirir*, sonra bit alanlarına ayırarak *çözer*, en sonunda da ilgili işlemi *yürütür*. Bu sürekli tekrar eden döngüye kısaca **Getir – Çöz – Yürüt** (*Fetch – Decode – Execute*) döngüsü denir.

3.2.1 BKM'nin Kısa Tarihçesi

Bilgisayar tarihi boyunca buyruk kümesi mimarileri üç temel yaklaşım etrafında şekillenmiştir. Her yaklaşım, işlenenlerin (*operand*) nerede saklandığı sorusuna farklı bir yanıt verir.

Yığıt mimarileri (1960'lar)

İlk yaklaşımda işlenenler bir yığıt (*stack*) yapısında tutulur. Buyruklar işlenen adresi taşımaz; her zaman yığıtın tepesindeki değerler üzerinde çalışır. Örneğin $a + b$ işlemi için önce a ve b yığıta itilir (PUSH), ardından ADD buyruğu ikisini çekip toplar ve sonucu yığıta geri koyar. Buyruklar kısa ve basit olduğu için donanım ucuzdur; ancak yığıtın tepesinden başka bir konuma doğrudan erişilemediğinden derleyicinin işi zorlaşır ve başarımlar düşer. Burroughs B5000 (1961) bu yaklaşımın en bilinen örneğidir. Buradaki “yığıt” kavramı, bu bölümün ilerleyen kısımlarında göreceğimiz altyordam çağrılarında kullanılan veri yapısı ile aynı ilkeye dayanır; ancak yığıt mimarisinde yığıt, altyordamlar için değil tüm hesaplamalar için kullanılır.

Biriktireç mimarileri (1970'ler)

Terim Kökenleri

biriktireç (*accumulator*): *Biriktirmek* fiilinin geçişli kökü *biriktir-* üzerine araç yapım eki *-eç* eklenerek türetilmiştir. Aynı kalıp: *saymak* → *sayaç* (*counter*), *süzme* → *süzgeç* (*filter*). “Biriktiren aygıt” anlamına gelir; ardışık aritmetik işlemlerin sonucunu üzerinde toplayarak biriktirdiği için bu adı almıştır. Bazı Türkçe kaynaklarda *accumulator* karşılığı olarak *toplayıcı* kullanılır; ancak *toplayıcı*, İngilizcedeki *adder* sözcüğünün doğal karşılığı olduğundan karışıklık yaratır. Bu kitapta tutarlı biçimde *biriktireç* tercih edilmiştir.

İkinci yaklaşımda *biriktireç* (*accumulator*) adı verilen tek bir özel yazmaç vardır. Her aritmetik buyruk, bir işleneni bellekten alır ve sonucu biriktirece yazar. Örneğin ADD x buyruğu “biriktirece x bellek adresindeki değeri ekle” anlamına gelir. Bu tasarım yığıt mimarisinden daha esnek olmakla birlikte, her ara sonucun biriktireçten belleğe yazılıp tekrar okunması ge-

rektiğinden bellek trafiği yoğundur. PDP-8 (1965), MOS 6502 (1975) ve Intel 8080 (1974) bu yaklaşımı benimsemiştir.

Yazmaç-yazmaç mimarileri (1980'ler–günümüz)

Üçüncü ve günümüzde baskın olan yaklaşımda birden çok genel amaçlı yazmaç bulunur. Aritmetik buyruklar yalnızca yazmaçlar arasında çalışır; bellek erişimi ayrı yükle/sakla (*load/store*) buyruklarıyla yapılır. Bu ayırım, derleyiciye sık kullanılan değerleri yazmaçlarda tutma olanağı tanır ve bellek trafiğini önemli ölçüde azaltır. MIPS (1985), SPARC (1987), ARM (1985) ve RISC-V (2010) bu yaklaşımın temsilcileridir. Bu kitapta ele alacağımız RISC-V de bir yazmaç-yazmaç mimarisidir.

CISC ve RISC tartışması

1980'e kadar geliştirilen işlemcilerin buyruk kümeleri giderek büyümüş, buyruklar karmaşıklaşmıştı. Bu yaklaşıma sonradan *Karmaşık Buyruk Kümesi Bilgisayarı* (CISC – *Complex Instruction Set Computer*) adı verildi. 1980'de David Patterson ve David Ditzel, "The Case for the Reduced Instruction Set Computer" başlıklı çalışmalarında [32] tam tersi bir görüş ileri sürdüler: az sayıda, basit ve düzenli buyruk kullanan bir mimari, karmaşık buyrukları olan bir mimariden daha hızlı olabilir; çünkü basit buyruklar daha kolay boru hattına alınır ve saat döngüsü kısaltılabilir. Bu ilkeye *İndirgenmiş Buyruk Kümesi Bilgisayarı* (RISC – *Reduced Instruction Set Computer*) adı verildi.

Tarih her iki tarafa da kısmen hak verdi. RISC ilkeleri galip geldi; bugünkü tüm yüksek başarılı işlemciler boru hattı, yazmaç penceresi ve yükle/sakla mimarisi gibi RISC özelliklerini kullanır. Ancak x86 gibi CISC mimarileri ortadan kalkmadı; bunun yerine karmaşık x86 buyruklarını iç düzende basit mikro-işlemlere (*micro-ops*) çeviren bir yaklaşım benimsediler (1995, Intel Pentium Pro). Böylece dışarıdan CISC görünen işlemci, içeride RISC gibi çalışır. RISC-V ise RISC felsefesinin en güncel ve en sade temsilcisi olarak bu bölümün ana konusunu oluşturmaktadır.

CISC ve RISC: Tarihsel Ayrışma

1980'lerin RISC/CISC tartışması, günümüz mimarilerini doğrudan şekillendirmiştir. x86, CISC çizgisini koruyarak geriye uyumluluğu ön planda tuttu; ticari başarısı bu kararlılığın ürünüdür. Buna karşın MIPS (1981), ARM (1985) ve RISC-V (2010) RISC ilkelerini benimsedi: az ve düzenli buyruk, sabit buyruk uzunluğu, yükle/sakla mimarisi.

Gerçekte iki taraf birbirinden tamamen ayrıışmadı. Çağdaş x86 işlemciler (Intel Pentium Pro'dan itibaren, 1995) karmaşık CISC buyrukları içeride basit mikro-işlemlere (*micro-ops*) dönüştürür; boru hattı bu mikro-işlemler üzerinde çalışır. Dışarıdan CISC görünen işlemci içeride RISC gibi davranmaktadır.

ARM ise farklı bir yol izleyerek AArch32'den AArch64'e geçişte kökten yeniden tasarım yaptı ve 64 bit mimarisinde pek çok RISC-V ile ortak ilkeye yer verdi. RISC-V, bu çizginin en sade ve en temiz halkasıdır; tarihsel birikimden bağımsız, sıfırdan tasarlanmıştır.

3.2.2 Buyruk Sayısı ve İşlem Kodu Uzaı

Bir buyruk kümesindeki işlem sayısı (*opcode count*), mimarinin karmaşıklığını ve donanım maliyetini doğrudan etkiler. Az işlemli bir BKM çözme donanımını basit tutar ama her işi

birden fazla buyrukla yapmak zorunda kalır; çok işlemli bir BKM ise tek buyrukla karmaşık görevleri tamamlar ancak çözücü donanımını büyütür.

Tablo 3.1: Farklı BKM’lerde yaklaşık buyruk sayıları

BKM	Yaklaşık buyruk sayısı	Not
RISC-V RV32I	~47	Yalnızca temel küme; M uzantısıyla ~55
MIPS R2000	~110	İlk nesil MIPS
ARM AArch64	~400	SIMD (NEON) dahil
x86-64	~1500+	SSE/AVX/AMX dahil; sürekli büyüyor

RISC-V’in yalnızca 47 temel buyruğa sahip olması tesadüf değildir. Tasarımcılar, mevcut buyruklarla elde edilebilen işlemler için yeni bir işlem kodu tanımlamak yerine sözde buyruklar kullanmayı bilinçli olarak tercih etmiştir. Örneğin RISC-V’te ve MIPS’te ayrı bir `mov` (kopyala) buyruğu yoktur: yazmaçtan yazmaca kopyalama, `addi rd, rs, 0` (sıfır ekle) ile gerçekleştirilir ve çevirici bunu `mv rd, rs` sözde buyruğu olarak sunar. Bu yaklaşımın iki avantajı vardır:

1. İşlem kodu uzayında yer tasarrufu sağlar. 32 bitlik sabit genişlikli bir buyrukta işlem koduna ayrılan bit sayısı sınırlıdır; her yeni gerçek buyruk bu sınırlı alanı tüketir. Sözde buyruklar ise bu alandan hiç pay almaz.
2. Çözme donanımı basit kalır. Daha az gerçek buyruk demek, daha az çözme yolu ve daha küçük çözücü devre demektir.

Bunun karşılığında, mikrodenetleyici gibi çok kısıtlı aygıtlarda RISC-V RV32I yeterli olabilirken, yoğun bilimsel hesaplama gerektiren sunucularda V (vektör) ve F/D (kayan nokta) uzantıları eklenerek buyruk sayısı artırılır. RISC-V’in modüler yapısı bu esnekliği sağlar: tasarımcı uygulamanın gereksinimine göre yalnızca ihtiyaç duyduğu uzantıları ekler.

3.3 RISC-V İşlemcisi ve Buyruk Kümesi

Buyruk kümesi mimarisinin ne olduğunu anladığımıza göre, artık somut bir örnek üzerinden ilerleyebiliriz. Bu kitapta kullandığımız RISC-V, 2010 yılında California Üniversitesi, Berkeley’de (UC Berkeley) Krste Asanovic ve David Patterson öncülüğünde tasarlanmış açık kaynaklı bir buyruk kümesi mimarisidir. Adındaki “V” harfi, Berkeley’de tasarlanan beşinci nesil RISC (*Reduced Instruction Set Computer*) mimarisini ifade eder.

RISC-V’i özel kılan en önemli özelliği açık kaynak ve ücretsiz bir BKM olmasıdır. ARM ya da x86 gibi ticari mimarilerde işlemci tasarlamak için lisans ücreti ödenirken, RISC-V’i herkes (hem üniversiteler hem de şirketler) serbestçe kullanabilir. Bu özellik, RISC-V’i gömülü sistemlerden yüksek başarılı bilgisayarlara kadar geniş bir yelpazede tercih edilen bir mimari haline getirmiştir.

RISC-V, başlangıçta RISC-V Foundation tarafından yönetilirken, daha sonra **RISC-V International** adını alan kuruluş tarafından geliştirilmeye devam etmektedir. Bu bölümde ele alınan buyrukların tam listesi, yazmaç tablosu, buyruk biçimleri ve sözde buyruklar Ek C’de özet hâlinde sunulmaktadır.

RISC-V’in bir diğer ayırt edici özelliği **modüler tasarımıdır**. Temel buyruk kümesi olan RV32I’nin yanına gereksinim duyulan uzantılar eklenerek işlemci özelleştirilebilir:

Tablo 3.2: RISC-V uzantıları

Uzantı	Açıklama
I	Temel tam sayı buyrukları (RV32I / RV64I)
M	Çarpma ve bölme buyrukları (<i>Multiply/Divide</i>)
A	Atomik bellek işlemleri (<i>Atomic</i>)
F	Tek duyarlıklı kayan nokta (<i>Single-precision Float</i>)
D	Çift duyarlıklı kayan nokta (<i>Double-precision Float</i>)
C	Sıkıştırılmış buyruklar (16 bit, yalnızca 8 yazmaç) (<i>Compressed</i>)
G	IMAFD birleşimi (genel amaçlı kısayol)

Tablo 3.2’de listelenen uzantılar, RISC-V’in modüler tasarımının temelini oluşturur. Bu modüler yapı, RISC-V’i ticari mimarilerden ayıran en önemli özelliklerden biridir. x86 ya da ARM gibi mimarilerde tüm buyruklar tek bir monolitik küme halindedir ve işlemci tasarımcısının seçim yapma imkânı kısıtlıdır. RISC-V’te ise tasarımcı yalnızca ihtiyaç duyduğu uzantıları ekler:

- Basit bir gömülü denetleyici için yalnızca **RV32I** yeterli olabilir; çarpma ve kayan nokta donanımına gerek yoktur.
- Genel amaçlı bir uygulama işlemcisi için **RV64GC** (= RV64IMAFDC) birleşimi tercih edilir; bu küme Linux çekirdeğini çalıştırabilecek kadar zengindir.
- Yapay zekâ hızlandırıcıları için **V** (vektör) uzantısı eklenerek SIMD benzeri koştur veri işleme olanağı sağlanır.
- Şirketler, standart uzantılara ek olarak kendi **özel buyruk uzantılarını** tanımlayabilir; bu esneklik, RISC-V’i özelleştirilmiş hızlandırıcılar (kripto, sinyal işleme vb.) için çekici kılar.

Bu kitapta aksi belirtilmedikçe RV32I temel buyruk kümesi ele alınacak, gerekli yerlerde RV64I farklılıkları ve M uzantısı ayrıca belirtilecektir.

Bilgi Notu 3.2: Bu Bölümde Ele Alınmayan Uzantılar

Kayan nokta uzantıları (F/D), ayrı bir yazmaç öbeği ($f0-f31$), durum yazmacı ($fcsr$) ve özel yuvarlama kipleri gerektirir; bu konular kitabın ileriki bölümlerinde kayan nokta aritmetiğiyle birlikte ele alınacaktır. Atomik bellek işlemleri (A uzantısı: $lr.w/sc.w$ ve AMO buyrukları) ise çok çekirdekli eşzamanlılık mekanizmalarıyla birlikte ayrıca incelenecektir.

Uzantı Modelleri

Mimariler, buyruk kümesini genişletme konusunda birbirinden farklı yaklaşımlar benimsemiştir. x86, tek parça (monolitik) bir model izler: her yeni işlemci kuşağı önceki tüm buyruklara ek olarak yeni uzantılar (MMX, SSE, AVX, AVX-512) getirir. Tüm uzantılar tek bir dev küme oluşturur; tasarımcının seçim hakkı yoktur.

ARM, hedef kullanım senaryosuna göre üç profil tanımlar: A serisi (uygulama işlemcileri), R serisi (gerçek zamanlı sistemler) ve M serisi (mikrodenetleyiciler). Her profil zorunlu ve isteğe bağlı uzantılardan oluşur.

MIPS sabit bir buyruk kümesi sunar; mimariye özgü seçenek sayısı sınırlıdır.

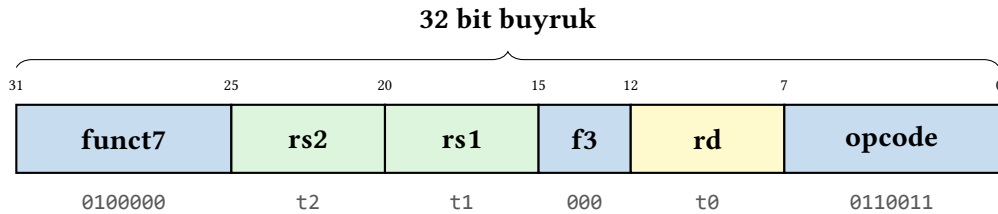
RISC-V bu tabloda ayrıştır: I temel kümesi üzerine M (çarpma/bölme), A (atomik), F/D (kayan

nokta), C (sıkıştırılmış) gibi uzantılar bağımsız olarak eklenir. Şirketler kendi özel uzantılarını da tanımlayabilir. Bu modüler yaklaşım, gömülü denetleyiciden yüksek başarımli sunucuya kadar tek bir BKM sözleşmesini farklı donanım maliyetiyle gerçeklemeye olanak tanır.

3.3.1 Yazılım Açısından Buyruk Kümesi

Farklı mimarilerin buyruk kümeleri birbirinden farklı olsa da temel işlemler her yerde aynıdır: topla, çıkar, bellekten oku, belleğe yaz, karşılaştır, dallan... Bu, tüm buyruk kümelerinin özünde birbirine benzemesinin nedenidir.

Bir programcı C ya da Java gibi üst düzey bir dilde kod yazar; ancak işlemci bu dili doğrudan anlamaz. Yazılan kodun önce **çevirici dile** (assembly), oradan da **makine diline** (0 ve 1'ler) dönüştürülmesi gerekir. Çevirici dilde her satır tek bir buyruğu ifade eder. Örneğin “sub t0, t1, t2” buyruğu t1 ve t2 yazmaçlarındaki değerleri çıkararak sonucu t0 yazmacına yazar (t0 = t1 - t2); bu buyruğun 32 bitlik kodlama biçimi Şekil 3.2’de gösterilmiştir.



Şekil 3.2: sub t0, t1, t2 buyruğunun RISC-V R-tipi biçiminde kodlanması. İşlem kodu (opcode) ve işlev alanları (funct7, funct3) birlikte hangi işlemin yapılacağını belirler; rd hedef yazmacı, rs1 ve rs2 ise kaynak yazmaçları gösterir.

Örnek 3.1

$a = b + c + d$ eşitliğini RISC-V çevirici dil buyruklarıyla yazın. b , c ve d değerlerinin sırasıyla s1, s2 ve s3 yazmaçlarında tutulduğunu, sonucun s0 yazmacına yazılacağını kabul edin.

Çözüm:

```
1 add t0, s1, s2    # b + c -> t0
2 add s0, t0, s3    # t0 + d -> a (s0)
```

Buyruk Alanları ve Kodlama

RISC-V ve MIPS, tüm buyrukları sabit 32 bit uzunluğunda kodlar; alan konumları (opcode, rs1, rs2, rd) biçimler arasında büyük ölçüde tutarlıdır. Bu düzenlilik, çözme (*decode*) donanımını basitleştirir: alan sınırları değişmez, bu yüzden koşut okuma kolaylaşır.

ARM (AArch64) da 32 bit sabit uzunluğu korur; ancak kodlama şeması RISC-V’ten farklıdır ve bazı alan pozisyonları biçime göre değişir.

x86 değişken uzunluklu buyruklara sahiptir: bir buyruk 1 ile 15 bayt arasında herhangi bir boyutta olabilir. Ön ek baytları (*prefix*), işlem kodu, ModRM, SIB, yer değiştirme (*displacement*) ve anlık değer (*immediate*) gibi isteğe bağlı alanlar birbiri ardına eklenir. Çözme aşamasında işlemci önce buyruk sınırlarını bulmak zorundadır; bu durum boru hattı tasarımını önemli ölçüde karmaşıktırır ve yüksek başarımli x86 işlemcilerinin büyük bir silikon alanını çözme devrelerine

ayırmasına neden olur.

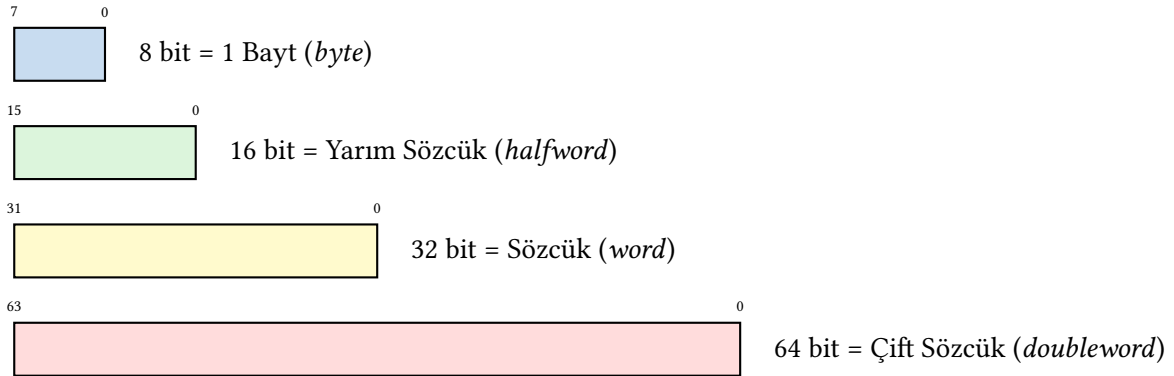
Bir buyruk kümesi mimarisinde buyruk boyutu tasarım öğelerinden biridir. Sabit boyutlu buyrukları olan bir BKM’de işlemci bellekten sürekli sabit boyutta veri okuyacağı için donanım çok daha yalın olur.

Tasarım İlkesi 4: Yalınlık düzenden gelir

Bölüm 1.6.1’de tanıttığımız bu ilke burada ilk kez somut biçimde karşımıza çıkar. Herhangi bir işlem ya da veri boyutu ne kadar düzenliyse ve kural dışı durumlardan ne kadar arındırılmışsa, donanımdaki uygulaması o kadar yalındır. Daha yalın olan donanımın karmaşıklığı düşüktür ve tasarımı kolaydır.

3.3.2 Donanım Açısından Buyruk Kümesi

Bölüm 1’de gördüğümüz gibi bir bilgisayarı oluşturan beş temel bileşenden biri olan bellek, buyrukların ve verilerin tutulduğu birimdir. Geçici değişkenlerin programlarda çok kullanılması ve bellek işlemlerinin işlemcinin içindeki işlemlerden daha yavaş olması nedeniyle çağdaş işlemcilerin tamamı **yazmaç** denilen ve işlemcinin içinde bulunan saklama alanlarına sahiptir. RISC-V’teki temel veri boyutları (bayt, yarım sözcük, sözcük, çift sözcük) Şekil 3.3’de özetlenmiştir.



Şekil 3.3: RISC-V’te veri yapıları

Tasarım İlkesi 1: Küçük olan hızlıdır (*devam*)

Bölüm 1.6.1 ve 2’de tanıttığımız bu ilke burada somut biçimde karşımıza çıkar. RISC-V tasarımcıları, programcıya çok sayıda yazmaç sunarken yazmaç birimini tek çevrimde erişilebilecek kadar küçük tutmayı hedeflemiştir. 32 yazmaç bu dengeyin sonucudur: 16 yazmaç derleyiciler için yetersiz kalır; 64 yazmaç ise yazmaç öbeğini büyütür erişim süresini uzatır ve buyruk içindeki adres alanını genişleterek buyruk boyutunu artırır.

RISC-V’te genel amaçlı yazmaçlar ABI (Application Binary Interface) adlarıyla gösterilir. Saklanan değerleri tutan yazmaçlar $s0, s1, s2, \dots$ şeklinde, geçici sonuçları tutan yazmaçlar da $t0, t1, t2, \dots$ şeklinde gösterilmektedir. MIPS’ten farklı olarak, RISC-V yazmaç adlarının başında $\$$ karakteri bulunmaz.

Örnek 3.2

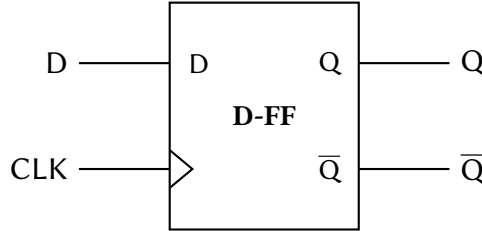
$e = (a + b) - (d + c)$ matematiksel ifadesini RISC-V buyruklarıyla yazın. a, b, c, d, e değerlerinin sırasıyla $s1, s2, s3, s4, s5$ yazmaçlarında tutulduğu kabul edilecektir.

Çözüm:

```
1 add t0, s1, s2 # (a + b) -> t0
2 add t1, s3, s4 # (c + d) -> t1
3 sub s5, t0, t1 # e = (a+b) - (c+d)
```

3.4 Yazmaçlar

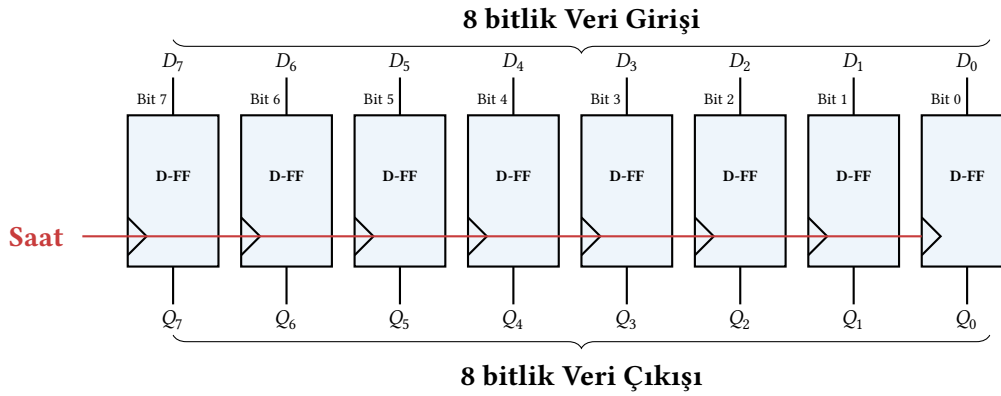
Bilgisayar yalnızca 0 ve 1'lerle çalıştığı için sayısal devrelerin doğal dili ikilik tabandır. İkilik tabandaki tek bir basamağı (0 ya da 1) saklayan en temel donanım birimi **D türü flip-flop**'tur (Şekil 3.4). Yazmaçlar bu flip-flopların yan yana dizilmesiyle oluşur.



Şekil 3.4: D türü flip-flop

İşlemcinin verilerle çalışabilmesi için onları çok hızlı erişebildiği bir yerde tutması gerekir. Bellek bu iş için fazla yavaş kalır. İşte yazmaçlar, doğrudan işlemci içinde yer alan ve en hızlı erişilen veri saklama birimleridir. Örneğin RISC-V RV32I'de her biri 32 flip-floptan oluşan toplam 32 adet yazmaç bulunur. Sekiz flip-floptan oluşan 8 bitlik bir yazmaç örneği Şekil 3.5'te gösterilmiştir. D flip-flopların çalışma ilkesi ve diğer ardışıl devre elemanları Ek B'de (Bölüm B.7) ayrıntılı olarak ele alınmıştır.

Buradaki D flip-flop gösterimi, yazmacın kavramsal yapısını anlatmak içindir. Günümüz yüksek başarılı işlemcilerinde yazmaç öbekleri genellikle SRAM tabanlı bit hücrelerinden oluşturulur; bu yapılar flip-floplara göre daha az alan kaplar ve çok kapılı (çok bağlantı noktalı) okuma/yazma erişimine olanak tanır. SRAM bit hücreleri bellek bölümünde ayrıntılı olarak incelenecektir.



Şekil 3.5: 8 bitlik veri tutan yazmaç (D flip-floplardan oluşan)

3.4.1 Yazmaç Türleri

- **Veri Yazmaçları:** Tam sayı değerleri saklamak için kullanılırlar.
- **Adres Yazmaçları:** Bellekteki veriye ulaşmak için gerekli adresleri tutan yazmaçlardır.
- **Genel Amaçlı Yazmaçlar:** Her tür veriyi tutabildikleri için derleyici bu yazmaçları hem değişkenleri hem de adresleri saklamak için kullanabilir.
- **Kayan Nokta Yazmaçları:** Ondalıkli sayıları tutmak için özel olarak tasarlanan yazmaçlardır.
- **Sabit Değerli Yazmaçlar:** Yalnızca sabit değerleri tutar; üzerlerine yazılamaz.
- **Vektör Yazmaçları:** Çoklu ortam buyruklarının yaptığı vektör işlemleri için kullanılır.
- **Özel Amaçlı Yazmaçlar:** Program sayacı yazmacı gibi özel bir amaç için kullanılan yazmaçlardır.

3.4.2 RISC-V’te Kullanılan Yazmaçlar

RISC-V BKM’sinde 32 adet tam sayı yazmacı bulunur. RISC-V’te yazmaçlar hem donanım adlarıyla (x0–x31) hem de ABI adlarıyla gösterilebilir; tüm yazmaçların adları ve görevleri Tablo 3.3’de verilmiştir (ayrıca bkz. Ek C, Bölüm C.4).

Tablo 3.3: RISC-V’te kullanılan yazmaçlar

ABI Adı	Donanım Adı	No	Kullanımı	Korunur?
zero	x0	0	Mutlak sıfır (her zaman 0)	–
ra	x1	1	Dönüş adresi	Hayır
sp	x2	2	Yığıt göstergesi	Evet
gp	x3	3	Genel gösterge	–
tp	x4	4	İş parçacığı göstergesi	–
t0–t2	x5–x7	5–7	Geçici yazmaçlar	Hayır
s0/fp	x8	8	Saklanan / Çerçeve göstergesi	Evet
s1	x9	9	Saklanan yazmaç	Evet
a0–a1	x10–x11	10–11	Terim / Sonuç yazmaçları	Hayır
a2–a7	x12–x17	12–17	Terim yazmaçları	Hayır
s2–s11	x18–x27	18–27	Saklanan yazmaçlar	Evet
t3–t6	x28–x31	28–31	Geçici yazmaçlar	Hayır

t ile başlayan yazmaçlar (geçici) ve s ile başlayan yazmaçlar (saklanan) RISC-V’te genel

amaçlı olarak kullanılır. Aralarındaki tek fark altyordam çağrılması sırasında s yazmaçlarının değerlerinin korunması, t yazmaçlarının ise değerlerinin korunmamasıdır.

3.4.3 Özel Amaçlı Yazmaçlar

Tablo 3.3'daki 32 genel amaçlı yazmacın yanı sıra, RISC-V mimarisinde programcının doğrudan erişemediği veya özel buyruklarla erişilen başka yazmaçlar da vardır. Bu yazmaçlar işlemcinin çalışma durumunu yönetir.

Program Sayacı (PC)

Program sayacı (*program counter*, PC), o anda yürütülmekte olan buyruğun bellek adresini tutar. Her buyruk yürütüldükten sonra PC otomatik olarak 4 artar (çünkü RV32I'de her buyruk 4 bayt uzunluğundadır). Dallanma ve atlama buyrukları PC'yi farklı bir adrese yönlendirerek bu doğrusal akışı değiştirir.

“Program sayacı” adı ilk bakışta yanıltıcı olabilir: bu yazmaç programları saymaz, *program içinde* sayar; bellekteki buyrukları sırasıyla adresleyerek programın neresinde olduğumuzu takip eder. İlk bilgisayarlarda buyruklar ardışık bellek adreslerinde yer aldığından, bir sonraki buyruğa geçmek için bir sayaç yeterliydi ve bu yazmaç “program sayacı” adını aldı. Günümüzde PC bir sayaçtan çok bir *adres göstergesidir*; ancak tarihsel ad yerleşmiştir.

PC, genel amaçlı yazmaç öbeğinin (x0–x31) bir parçası değildir; ayrı bir donanım yazmacıdır. Programcı PC'ye doğrudan bir değer yazamaz; PC yalnızca dallanma (beq, bne, ...), atlama (jal, jalr) ve kural dışı durum mekanizmaları aracılığıyla değişir. Ancak auipc buyruğu PC'nin o anki değerini *okuyarak* bir yazmaçta kullanılabilir hale getirir; bu sayede konumdan bağımsız kod (*position-independent code*) yazılabilir.

Program Sayacı: Farklı Mimarilerde Farklı Yaklaşımlar

Her mimari programın o anki konumunu izleyen bir yazmaca sahiptir; ancak adı, yapısı ve erişim biçimi farklılık gösterir.

RISC-V: Ayrı bir donanım yazmacı olan PC, genel amaçlı yazmaç öbeğinin dışındadır. Doğrudan okunamaz ve yazılamaz; yalnızca auipc ile dolaylı okunabilir, dallanma/atlama buyrukları ile değiştirilebilir. Sabit 4 baytlık buyruklar nedeniyle her adımda +4 artar.

x86: Intel bu yazmaca “program sayacı” değil, *buyruk işaretçisi* (*instruction pointer*) adını verir; 32 bitte EIP, 64 bitte RIP. Değişken uzunluklu buyruklar (1–15 bayt) nedeniyle artış miktarı her buyrukta farklıdır. Eski 16 bit modunda (8086) adres uzayı 1 MB ile sınırlıydı; bu sınırı aşmak için CS (*Code Segment*) yazmacı ile IP birlikte kullanılırdı: fiziksel adres = CS × 16 + IP. Çağdaş 64 bit modunda bölütleme (*segmentation*) işlevini yitirmiştir; RIP tek başına 64 bitlik sanal adresi tutar.

ARM AArch32: Program sayacı, genel amaçlı yazmaç öbeğinin bir parçasıydı (R15 = PC). Bu tasarım, ADD PC, PC, R0 gibi buyrukların doğrudan PC'ye yazmasına olanak tanıyordu; son derece esnekti ancak güvenlik açıklarına ve dallanma öngörücüsü karmaşıklığına yol açtı. ARM bu kararı AArch64'te düzeltti: PC artık ayrı bir yazmaçtır ve genel amaçlı buyruklar ile erişilemez.

MIPS: RISC-V'e çok benzer biçimde PC ayrı bir yazmaçtır. Ancak MIPS'te bir ek karmaşıklık vardır: *gecikmeli dallanma* (*delayed branch*) nedeniyle dallanma buyruğundan *sonraki* buyruk da yürütülür. RISC-V bu karmaşıklığı ortadan kaldırmıştır.

Bilgi Notu 3.3: PC Neden Genel Amaçlı Yazmaç Değildir?

ARM AArch32’de program sayacının R15 olarak genel amaçlı yazmaç öbeğine dahil edilmesi, başlangıçta zarif bir tasarım kararı gibi görünüyordu: herhangi bir buyruk PC’yi okuyabilir veya değiştirebilirdi. Ancak bu esneklik ciddi sorunlara yol açtı: (1) güvenlik açıkları; bir saldırgan yazmaç üzerinden PC’yi keyfi bir adrese yönlendirebilirdi, (2) dallanma öngörüsü karmaşıklığı; herhangi bir veri işleme buyruğu gizli bir dallanma olabilirdi, (3) boru hattı tasarımı; PC’nin ne zaman güncelleneceği belirsizleşiyordu. Bu deneyimden ders çıkaran çağdaş mimariler (RISC-V, ARM AArch64, x86-64) PC’yi ayrı tutmayı tercih eder.

Denetim ve Durum Yazmaçları (CSR)

RISC-V, işlemcinin çalışma durumunu ve ayrıcalık düzeyini yöneten özel yazmaçlara sahiptir. Bunlara *denetim ve durum yazmaçları* (*Control and Status Registers*, CSR) denir. CSR’lere genel amaçlı buyruklar (add, lw vb.) ile erişilemez; bunun yerine csrrw, csrrs, csrrc gibi özel CSR buyrukları kullanılır.

En sık karşılaşılan CSR yazmaçları Tablo 3.4’de listelenmiştir.

Tablo 3.4: Temel denetim ve durum yazmaçları (CSR)

CSR Adı	Açılımı	Görevi
mstatus	Machine Status	İşlemcinin çalışma durumunu tutar: kesmelerin açık/kapalı olduğu, önceki ayrıcalık düzeyi vb.
mepc	Machine Exception PC	Kural dışı durum oluştuğunda PC’nin o anki değerini saklar. İşleyiciden dönüşte (mret) bu adrese geri dönülür.
mcause	Machine Cause	Kural dışı durumun nedenini belirten bir kod tutar (örneğin: geçersiz buyruk, sayfa hatası, zamanlayıcı kesmesi).
mtvec	Machine Trap Vector	Kural dışı durum işleyicisinin başlangıç adresini tutar. Bir kesme veya kural dışı durum oluştuğunda PC bu adrese yönlendirilir.
mscratch	Machine Scratch	İşletim sistemi çekirdeğinin geçici veri saklayabileceği bir yazmaçtır.
mhartid	Machine Hart ID	Çok çekirdekli sistemlerde çekirdeğin kimlik numarasını tutar.
<i>Başarım sayaçları</i>		
mcycle	Machine Cycle	İşlemcinin başlatılmasından bu yana geçen saat çevrimi sayısını tutar.
minstret	Machine Instructions Retired	İşlemcinin başlatılmasından bu yana tamamlanan buyruk sayısını tutar.

CSR yazmaçları *ayrıcalıklı* erişim gerektirir: m öneki “makine” (*machine*) ayrıcalık düzeyini belirtir; bu, en yüksek yetki düzeyidir. Kullanıcı programları bu yazmaçlara doğrudan erişemez; erişim yalnızca işletim sistemi çekirdeği veya önyükleme yazılımı tarafından yapılır.

RISC-V’te üç ayrıcalık düzeyi tanımlanmıştır:

Düzy	Ad	Kullanım alanı
0	Kullanıcı (<i>User</i> , U)	Uygulama programları
1	Gözetmen (<i>Supervisor</i> , S)	İşletim sistemi çekirdeği
3	Makine (<i>Machine</i> , M)	Donanım yazılımı, önyükleyici

ecall buyruğu, bir alt düzeyden bir üst düzeye geçiş isteği göndermek için kullanılır. Örneğin kullanıcı düzeyindeki bir program ecall çalıştırdığında işlemci gözetmen veya makine düzeyine geçer; bu geçiş sırasında PC, mtvec'teki adrese yönlendirilir ve eski PC değeri mepc'ye kaydedilir. Bu mekanizmanın ayrıntıları Bölüm 9'da ele alınacaktır.

Bilgi Notu 3.4: Başarım Sayaçları ve BBÇ Ölçümü

Tablo 3.4'deki mcycle ve minstret yazmaçları, Bölüm 2'deki başarım formülüyle doğru-
dan bağlantılıdır. Bu iki yazmacı okuyarak bir programın gerçek BBÇ değerini ölçebiliriz:

$$BBÇ = \frac{mcycle}{minstret}$$

Örneğin bir program çalıştırıldıktan sonra mcycle = 4200 ve minstret = 3500 okunursa BBÇ = 4200/3500 = 1,20 olur; bu, ortalama olarak her buyruğun 1,2 çevrimde tamamlandığı anlamına gelir.

RISC-V, kullanıcı düzeyinden bu sayaçlara salt okunur erişim sağlayan cycle ve instret adlı gölge CSR'ler de tanımlar. Bu sayede ayrıcalıklı kipe geçmeden başarım ölçümü yapılabilir; ancak bu erişimin açık olup olmadığı mcounteren yazmacı ile denetlenir.

Donanım başarım sayaçları fikri RISC-V ile ortaya çıkmış değildir. Intel, 1993'te Pentium işlemcisiyle birlikte zaman damgası sayacı (*Time Stamp Counter*, TSC) adını verdiği 64 bitlik bir çevrim sayacını tanıttı; bu sayaç RDTSC buyruğuyla okunabiliyordu. Pentium Pro (1995) ile birlikte başarım izleme sayaçları (*Performance Monitoring Counters*, PMC) eklendi: önbellek bulma/bulamama oranları, dallanma öngörü başarısı, boru hattı duraklamaları gibi yüzlerce farklı olay sayılabilir hale geldi. Günümüzde Intel işlemcileri binlerce farklı donanım olayını izleyebilen programlanabilir sayaçlara sahiptir; Linux'taki perf aracı bu sayaçları kullanarak program başarımını ayrıntılı biçimde çözümler.

ARM da benzer bir altyapı sunar: ARMv7'den itibaren Başarım İzleyici Birimi (*Performance Monitor Unit*, PMU) standart bir bileşen olarak tanımlanmıştır. RISC-V'in farkı, en temel iki sayacı (mcycle ve minstret) zorunlu CSR olarak tanımlamasıdır; en basit RISC-V gerçeklemesinde bile bu sayaçlar bulunmalıdır.

Özel Yazmaçlar: RISC-V, ARM ve x86

Her mimari, genel amaçlı yazmaçların yanında özel amaçlı yazmaçlar tanımlar; ancak erişim yöntemleri farklıdır:

RISC-V: Özel yazmaçlar (CSR) genel amaçlı yazmaç öbeğinden tamamen ayrıdır. Erişim yalnızca csrrw/csrrs/csrrc buyrukları ile yapılır. Bu ayırım tasarımı basitleştirir ve güvenliği artırır.

ARM AArch64: Sistem yazmaçlarına MSR/MRS buyrukları ile erişilir. Yapı RISC-V'e benzer; ancak AArch32'de PC'nin genel amaçlı yazmaç (R15) olarak erişilebilmesi güvenlik sorunlarına yol açmıştır ve AArch64'te bu kaldırılmıştır.

x86-64: Bayrak yazmacı (RFLAGS) aritmetik işlemlerin yan etkisi olarak otomatik güncellenir; ayrı bir buyrukla yazılmaz. Denetim yazmaçları (CR0–CR4) ve modele özgü yazmaçlar (MSR) ayrıca-

lıklı erişim gerektirir. Bayrak yazmacının varlığı, RISC-V'in bayrak yazmacı kullanmama kararıyla ilginç bir karşılık oluşturur.

Kayan Nokta Yazmaçları

RISC-V'in F (tek duyarlık) ve D (çift duyarlık) uzantıları etkinleştirildiğinde, tam sayı yazmaçlarından bağımsız 32 adet kayan nokta yazmacı ($f0-f31$) kullanılabilir hale gelir. Bu yazmaçlar ayrı bir yazmaç öbeğinde yer alır ve yalnızca kayan nokta buyrukları ($fadd.s$, $f1w$, fsw vb.) ile erişilebilir.

Ayrıca kayan nokta işlemlerinin durumunu tutan bir denetim yazmacı ($fcsr$) bulunur. Bu yazmaç yuvarlama kipini ve kural dışı durum bayraklarını (taşma, sıfıra bölme vb.) içerir. Kayan nokta aritmetiğinin ayrıntıları Bölüm 5'te ele alınacaktır.

Bilgi Notu 3.5: Yığıt Göstergesi: Donanım mı, Uzlaşım mı?

Tablo 3.3'da $x2$ yazmacının ABI adı sp (*stack pointer*, yığıt göstergesi) olarak belirtilmişti. Ancak RISC-V donanımı açısından $x2$ ile diğer yazmaçlar arasında hiçbir fark yoktur; $x2$ sıradan bir genel amaçlı yazmaçtır. Yığıt göstergesi rolü tamamen yazılım uzlaşısına (ABI) dayanır; donanım $x2$ 'yi özel olarak tanımaz.

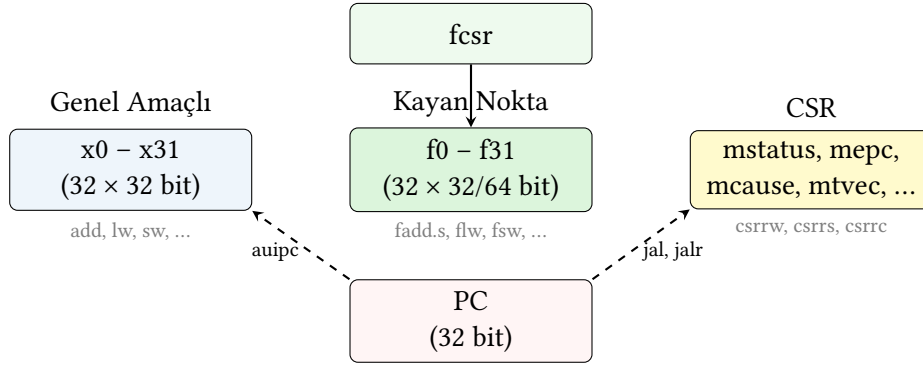
Bu, $x86$ ile temel bir farktır. $x86$ 'da RSP donanım düzeyinde özel bir yazmaçtır: **PUSH** buyruğu RSP 'yi otomatik olarak azaltıp veriyi belleğe yazar; **POP** ise bellekten okuyup RSP 'yi artırır. **CALL** dönüş adresini RSP 'nin gösterdiği yere yazar; **RET** oradan okur. Programcı RSP 'yi başka amaçla kullansa bile bu buyruklar hâlâ RSP 'yi yığıt göstergesi olarak varsayar.

RISC-V'te ise **PUSH/POP** buyrukları yoktur. Yığıta yazmak için programcı sp 'yi açıkça güncelleyip sw/lw kullanır:

```
1  addi sp, sp, -4    # yer aç (PUSH'un birinci yarısı)
2  sw    t0, 0(sp)    # yığıta yaz (PUSH'un ikinci yarısı)
```

Bu yaklaşım daha fazla buyruk gerektirir; ancak donanımı sadeleştirir ve yazmaç öbeğinde özel durum mantığı gerektirmez. Aynı tasarım kararı ra ($x1$), gp ($x3$) ve tp ($x4$) için de geçerlidir; bunların hepsi donanım açısından sıradan yazmaçlardır, özel rolleri ABI tarafından atanmıştır.

Şekil 3.6, RISC-V'teki tüm yazmaç türlerini ve aralarındaki ilişkiyi özetlemektedir.



Şekil 3.6: RISC-V’teki yazmaç türleri ve erişim yöntemleri. Genel amaçlı, kayan nokta ve CSR yazmaçları birbirinden bağımsız dosyalarda yer alır. Program sayacı (PC) ayrı bir donanım yazmacıdır; dolaylı olarak okunabilir (aupc) ancak doğrudan yazılamaz.

3.4.4 Sıfır Yazmacı: x0 Neden Hep Sıfırdır

Tablo 3.3’de dikkat çeken bir ayrıntı, x0 (zero) yazmacının değerinin her zaman sıfır olması ve üzerine yazılamamasıdır. Bu tasarım kararının ardında güçlü bir neden yatar: sıfır değeri programlarda son derece sık kullanılır ve ayrı bir sıfır yazmacı, pek çok işlemi ek buyruk gerektirmeden gerçekleştirmeyi sağlar. Örneğin:

- Yazmaçtan yazmaca kopyalama: `add t0, t1, zero` $\equiv$ `mv t0, t1`
- Koşulsuz atlama: `jal zero, etiket` $\equiv$ `j etiket` (dönüş adresi atılır)
- Sonucu yok sayma: `jalr zero, rs1, 0` dönüş adresini kaydetmeden atlar
- İşaret değiştirme: `sub t0, zero, t1` $\equiv$ `neg t0, t1`
- Sıfırla karşılaştırma: `beq t0, zero, etiket` (t0 sıfır mı?)

Bu kullanımlar sayesinde ayrı `mov`, `nop`, `neg`, `j` buyrukları tanımlamaya gerek kalmaz; hepsi x0 sayesinde mevcut buyrukların özel durumları olarak elde edilir. MIPS de aynı yaklaşımı benimser (\$zero yazmacı). ARM AArch64’te ise XZR/WZR yazmaçları benzer role sahiptir ancak aynı zamanda yığıt göstergesi (SP) olarak da kullanılabilir. x86’da böyle bir sıfır yazmacı yoktur; sıfır değeri gereken yerlerde anlık değer olarak belirtilmek zorundadır.

RISC-V’te terim yazmaçları a0–a7 olarak toplam 8 adettir. Bu, MIPS’teki 4 terim yazmacından (\$a0–\$a3) daha fazladır ve altyordam çağrılarında daha fazla verinin yazmaçlar aracılığıyla geçirilmesine olanak tanır.

3.4.5 RISC-V’te Neden Bayrak Yazmacı Yoktur?

x86 ve ARM gibi mimarilerde aritmetik ve mantık buyrukları bir *bayrak yazmacını* (flags register) otomatik olarak güncellerler. Bu yazmaçtaki tek bitlik bayraklar, işlemin sonucuyla ilgili bilgi taşır:

Bayrak	Ad	Anlamı
ZF	Zero Flag (Sıfır Bayrağı)	Sonuç sıfır mı?
CF	Carry Flag (Elde Bayrağı)	İşaretsiz taşma oluştu mu?
SF	Sign Flag (İşaret Bayrağı)	Sonuç negatif mi?
OF	Overflow Flag (Taşma Bayrağı)	İşaretli taşma oluştu mu?

x86’da bir koşullu dallanma iki adımda gerçekleşir: önce `CMP` buyruğu bayrakları günceller, ardından `JE`, `JL` gibi bir koşullu atlama buyruğu bayrakları okuyarak karar verir:


```
CMP EAX, EBX ; EAX - EBX hesapla, bayrakları güncelle
JGE buyuk    ; SF=OF ise (>= durumu) atla
```

RISC-V ise bayrak yazmacı kullanmaz. Bunun yerine karşılaştırma ve dallanma tek bir buyrukta birleştirilir:

```
1 bge t0, t1, buyuk # t0 >= t1 ise dallan
```

Bu tasarım kararının arkasında üç önemli neden vardır:

1. *Bağımlılık zincirini kırmak*: Bayrak yazmacı tek bir ortak kaynaktır. Ardışık buyrukların hepsi aynı bayrak yazmacını okuyup yazarsa, bu buyruklar birbirine bağımlı hale gelir ve koşut yürütülmeleri zorlaşır. Bayrak yazmacı olmadığına her buyruk yalnızca kendi hedef yazmacını yazar; bağımsız buyruklar koşut biçimde yürütülebilir.
2. *Sırasız yürütümü kolaylaştırmak*: Çağdaş yüksek başarılı işlemciler buyrukları program sırasından farklı bir düzende yürütebilir (*out-of-order execution*). Bayrak yazmacı bu esnekliği kısıtlar çünkü bayrakları güncelleyen ve okuyan buyruklar arasındaki sıra korunmalıdır. RISC-V’te bu sorun yoktur; karşılaştırma ve dallanma aynı buyrukta yapılır.
3. *Tasarımı sadeleştirmek*: Bayrak yazmacının güncelleme ve okuma mantığı, boru hattı tasarımına ek karmaşıklık getirir. Hangi buyruğun bayrakları güncellediği, hangisinin yalnızca okuduğu izlenmelidir. RISC-V bu karmaşıklığı ortadan kaldırarak donanım tasarımını sadeleştirir.

Dallanma Mekanizması: Bayraklı ve Bayraksız Mimariler

x86 (bayraklı): Karşılaştırma (CMP) ve dallanma (JGE) iki ayrı buyruktur. Arada başka buyruklar bayrakları bozabilir; derleyicinin bu durumu dikkatle yönetmesi gerekir. Avantajı: tek bir karşılaştırma sonucunu birden çok dallanma buyruğu kullanabilir.

ARM (bayraklı, seçimli): AArch32’de yalnızca sonuna S eklenen buyruklar (ADDS, SUBS) bayrakları günceller; ADD, SUB güncellemez. Bu, x86’nın “her buyruk bayrak günceller” sorununu kısmen çözer. AArch64’te ayrıca CBZ/CBNZ gibi bayraksız dallanma buyrukları da eklenmiştir.

RISC-V (bayraksız): beq, bne, blt, bge, bltu, bgeu buyrukları iki yazmacı doğrudan karşılaştırıp dallanır. Bayrak yazmacı olmadığı için ardışık buyruklar arasında gizli bağımlılık oluşmaz. Dezavantajı: aynı karşılaştırma sonucunu birden çok dallanma kullanmak istiyorsa karşılaştırma tekrarlanmalıdır; ancak bu durum uygulamada nadirdir.

MIPS (bayraksız): RISC-V’e benzer biçimde beq/bne buyrukları kullanır; ancak “küçükse” karşılaştırması için ayrı bir slt (set on less than) buyruğu ile sonucu bir yazmaca yazıp ardından bne ile dallanma gerekir: iki buyruk. RISC-V’in blt buyruğu bu işi tek buyrukta yapar.

3.4.6 Yazmaç Sayısı ve Tasarım Dengeleri

Farklı mimarilerin yazmaç sayıları arasında büyük farklar vardır. Bu fark rastlantısal değildir; yazmaç sayısı hem başarımlı hem de buyruk kodlama biçimini doğrudan etkiler.

Tablo 3.5: Dört mimaride yazmaç karşılaştırması

Mimari	Genel Amaçlı	Genişlik	Adlandırma	Özel Yazmaçlar
RISC-V (RV32I)	32	32 bit	x0–x31	x0 = sıfır
RISC-V (RV64I)	32	64 bit	x0–x31	x0 = sıfır
MIPS	32	32 bit	\$0–\$31	\$0 = sıfır, HI/LO
ARM (AArch64)	31	64 bit	X0–X30	XZR/SP (bağlama göre)
x86 (IA-32)	8	32 bit	EAX–EDI	Örtük roller
x86-64	16	64 bit	RAX–R15	Örtük roller

Tablo 3.5’de görüldüğü gibi mimari tasarımlar arasında yazmaç sayısı önemli ölçüde farklılık gösterir. Her iki uçtaki sorunları anlamak, mimari tasarım kararlarının ardındaki mantığı kavramak açısından önemlidir.

Yazmaç sayısı az olduğunda derleyici, hesaplama sırasında kullanılan tüm değişkenleri yazmaçlarda tutamaz. Bu durumda bazı değerleri geçici olarak belleğe (yığıta) yazmak ve daha sonra geri okumak zorunda kalır. Bu işleme *yazmaçtan belleğe taşıma (register spilling)* denir; bu duruma yol açan yetersizliğe de *yazmaç yetersizliği* adı verilir. Derleyici, yazmaca sığdıramadığı her değer için fazladan bir saklama (store) ve bir yükleme (load) buyruğu üretmek zorundadır; bu da hem buyruk sayısını artırır hem de yavaş bellek erişimleri nedeniyle başarıyı düşürür. Örneğin x86 IA-32’nin yalnızca 8 genel amaçlı yazmacı olması, derleyicileri sık sık yazmaçtan belleğe taşıma yapmaya zorlamış ve bu durum x86-64’te yazmaç sayısının 16’ya çıkarılmasının başlıca nedenlerinden biri olmuştur.

Öte yandan yazmaç sayısını artırmanın da bir bedeli vardır. Her buyrukta kaynak ve hedef yazmaçlarını belirtmek için daha fazla bit gerekir: 32 yazmaç, buyrukta yazmaç numarasını kodlamak için 5 bit gerektirirken, 16 yazmaç 4 bit, 8 yazmaç ise 3 bit gerektirir. Üç yazmaçlı bir buyrukta (kaynak 1, kaynak 2, hedef) bu fark $3 \times (5 - 3) = 6$ bite ulaşır. Sabit 32 bitlik buyruk genişliğinde bu fazladan bitler, anlık değerler ve işlem kodları için ayrılacak alandan çalınır. Aynı programın makine kodu daha fazla yer kaplar ve buyruk ön belleğine (*instruction cache*) daha fazla baskı uygular; bu duruma *kod şişmesi (code bloat)* denir.

Bağlam değiştirme (*context switch*) maliyeti de yazmaç sayısı ile doğru orantılıdır. İşletim sistemi bir süreçten diğerine geçerken tüm yazmaçların belleğe kaydedilip geri yüklenmesi gerekir; 32 yazmaçlı bir mimaride bu işlem, 8 yazmaçlı bir mimariye göre dört kat daha fazla veri aktarımı gerektirir.

Tablo 3.6: Yazmaç sayısının artı ve eksileri

	Az yazmaç (ör. x86 IA-32: 8)	Çok yazmaç (ör. RISC-V: 32)
Buyruk genişliği	✓ Yazmaç numarası az bit tutar → anlık değerlere ve işlem koduna daha fazla alan kalır	✗ Yazmaç numarası çok bit tutar → sabit genişlikli buyrukta alan daralır
Kod boyutu	✓ Daha küçük buyruklar → buyruk önbellegi verimli	✗ Daha geniş buyruklar → kod şişmesi riski
Bellek erişimi	✗ Yazmaç yetmez → yazmaçtan belleğe taşıma sık → yavaş	✓ Değerler yazmaçlarda kalır → az bellek erişimi → hızlı
Yazmaç erişim gecikmesi	✓ Yazmaç dosyası küçük → okuma/yazma süresi kısa; yüksek saat hızını destekler	✗ Yazmaç dosyası büyük → okuma/yazma bağlantı noktaları artar, fiziksel erişim süresi uzar; saat döngüsünü sınırlayabilir
Derleyici karmaşıklığı	✗ Yazmaç ataması zor; belleğe taşıma yönetimi gerekli	✓ Yazmaç ataması kolay; boş yazmaç bulmak daha olası
Bağlam değiştirme	✓ Az yazmaç kaydedilir → hızlı	✗ Çok yazmaç kaydedilir → yavaş
Donanım maliyeti	✓ Yazmaç dosyası küçük → az alan ve güç	✗ Yazmaç dosyası büyük → daha fazla alan ve güç

✓ = o seçenek için avantaj ✗ = o seçenek için dezavantaj

Tablo 3.6’de özetlendiği gibi, az ve çok yazmaç arasında bir denge noktası vardır. Aşağıdaki iki örnek bu dengeyi somut olarak göstermektedir.

Terim Kökenleri

yazmaçtan belleğe taşıma (*register spilling*): İngilizcede *register spilling* terimi, “taşma” ve “dökülme” çağrışımı taşır; tıpkı dolan bir tabaktan sulu yemeğin taşması gibi, yazmaçlara sığmayan değerler belleğe “dökülür.” Türkçede bu mecaz doğal karşılanmadığından, kavramı daha açık ifade eden *yazmaçtan belleğe taşıma* veya kısaca *yazmaç boşaltma* karşılıkları tercih edilmektedir. Derleyici, hesaplama sırasında kullanılan tüm değişkenleri yazmaçlarda tutamadığında bazı değerleri geçici olarak belleğe (genellikle yığta) yazar ve ihtiyaç duyulduğunda geri yükler. Bu işlem fazladan saklama (store) ve yükleme (load) buyrukları üretir; hem buyruk sayısını artırır hem de yavaş bellek erişimleri nedeniyle başarımı düşürür.

Örnek 3.3

Aşağıdaki C ifadesini ele alalım:

```
1 int a = b + c - d * e + f;
```

Bu ifade derlendiğinde beş farklı değişken (b, c, d, e, f) eş zamanlı olarak kullanımdadır. Derleyicinin bunları yazmaçlarda nasıl yönettiği, mimarının sunduğu yazmaç sayısına bağlıdır.

Çözüm:

(a) 8 yazmaçlı mimari (x86 IA-32 benzeri):

x86 IA-32’de yalnızca 8 genel amaçlı yazmaç bulunur; ancak bazıları örtük rollere sahip olduğundan derleyiciye en fazla 5–6 yazmaç kalır. Beş değişken aynı anda gerektiğinde derleyici bir ya da iki değeri yığta taşımak zorunda kalır. Bunun sonucunda üretilen

kod şuna benzer:

```

1 ; x86 IA-32 benzeri (basitleştirilmiş)
2 ; Yazmaçlar: EAX, ECX, EDX, EBX (4 yazmaç mevcut)
3 ; d ve e yığıta alınıyor (yazmaçtan belleğe taşıma kodu):
4     mov [esp-4], d_val ; store: d'yi yığıta yaz
5     mov [esp-8], e_val ; store: e'yi yığıta yaz
6     mov EAX, b ; EAX = b
7     add EAX, c ; EAX = b + c
8     mov ECX, [esp-4] ; load: d'yi yığıttan geri oku
9     imul ECX, [esp-8] ; load: ECX = d * e (e yığıttan)
10    sub EAX, ECX ; EAX = b + c - d*e
11    add EAX, f ; EAX = b + c - d*e + f
12    mov [a], EAX ; a'yı belleğe yaz

```

Toplam: 2 fazladan saklama (store) + 2 fazladan yükleme (load) = 4 fazladan bellek erişim buyruğu.

(b) 32 yazmaçlı mimari (RISC-V):

RISC-V'te 32 genel amaçlı yazmaç bulunur. Beş değişken kolayca ayrı yazmaçlara atanır; yığıta taşıma gerekmez:

```

1 # t0=b, t1=c, t2=d, t3=e, t4=f; s0 a değişkeninin adresi
2 add t5, t0, t1 ; t5 = b + c
3 mul t6, t2, t3 ; t6 = d * e
4 sub t5, t5, t6 ; t5 = b + c - d*e
5 add t5, t5, t4 ; t5 = b + c - d*e + f
6 sw t5, 0(s0) ; sonucu belleğe yaz

```

Toplam: 0 fazladan saklama/yükleme buyruğu (yalnızca son sonuç belleğe yazılır).

Sonuç: 8 yazmaçlı mimaride toplam 9 buyruk (4'ü belleğe taşıma kodu), 32 yazmaçlı mimaride ise 5 buyruk yeterlidir. Bu fark yalnızca buyruk sayısını değil, Bölüm 2'deki yürütme zamanı denkleminin (Denklem 2.8) her üç bileşenini de etkiler:

- *Buyruk sayısı azalır:* 32 yazmaçlı mimaride belleğe taşıma buyrukları ortadan kalktığından toplam buyruk sayısı düşer.
- *BBÇ düşer:* Eklenen her 1w/sw buyruğu belleğe erişir; önbellekte bulamama durumunda tek bir yükleme onlarca saat döngüsü sürebilir. Yazmaç erişimi ise tek döngüde tamamlanır. Bellek buyruklarının oranı azaldıkça ortalama BBÇ de düşer.
- *Çevrim zamanı artabilir:* Yazmaç dosyası büyüdükçe fiziksel erişim gecikmesi uzar; bu durum saat döngüsünü sınırlayabilir. Ancak günümüz teknolojisinde 32 yazmaçlı bir dosyanın erişim süresi genellikle tek döngüde kalır.

Gerçek derleyicilerde bu fark, özellikle döngü gövdelerinde sürekli yinelenerek toplam başarımlar üzerinde belirgin bir etki bırakır. Bir sonraki örnekte bu etkiyi sayısal olarak hesaplayacağız.

Örnek 3.4

Örnek 3.3'teki iki mimariyi karşılaştıralım. Aşağıdaki varsayımları kullanarak her iki mimarideki yürütme zamanını hesaplayın ve hangisinin daha hızlı olduğunu bulun.

- 8 yazmaçlı mimari: 9 buyruk (4'ü bellek erişimi), saat sıklığı 4,0 GHz.
- 32 yazmaçlı mimari: 5 buyruk (yalnızca 1 bellek erişimi), saat sıklığı 3,8 GHz (daha büyük yazmaç öbeği nedeniyle biraz düşük).
- Aritmetik buyruklar 1 çevrim, bellek buyrukları (1w/sw) ortalama 3 çevrim sürüyor (önbellek etkisi dahil).

Çözüm:

Denklem 2.8'yi kullanıyoruz: Yürütme zamanı = Buyruk sayısı $\times$ BBÇ $\times$ Çevrim zamanı.

(a) 8 yazmaçlı mimari:

9 buyruktan 5'i aritmetik (1 çevrim), 4'ü bellek (3 çevrim):

$$\text{Toplam çevrim} = 5 \times 1 + 4 \times 3 = 17 \Rightarrow \text{BBÇ} = \frac{17}{9} \approx 1,89$$

$$\text{Yürütme zamanı} = \frac{17}{4,0 \times 10^9} = 4,25 \text{ ns}$$

(b) 32 yazmaçlı mimari:

5 buyruktan 4'ü aritmetik (1 çevrim), 1'i bellek (3 çevrim):

$$\text{Toplam çevrim} = 4 \times 1 + 1 \times 3 = 7 \Rightarrow \text{BBÇ} = \frac{7}{5} = 1,40$$

$$\text{Yürütme zamanı} = \frac{7}{3,8 \times 10^9} = 1,84 \text{ ns}$$

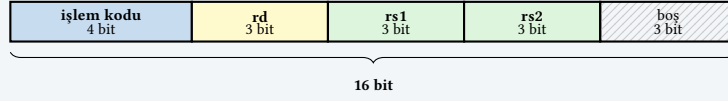
Sonuç: 32 yazmaçlı mimari $4,25/1,84 = 2,31$ kat daha hızlıdır; saat hızının %5 düşük olmasına karşın. Kazanç iki kaynaktan gelir: buyruk sayısı 9'dan 5'e düşmüştür (belleğe taşıma buyrukları ortadan kalkmıştır) ve bellek buyrukları oranı azaldığından ortalama BBÇ 1,89'dan 1,40'a gerilemiştir. Bu örnek, yazmaç sayısı artışının çevrim zamanındaki küçük kaybı fazlasıyla telafi edebildiğini göstermektedir.

Örnek 3.5

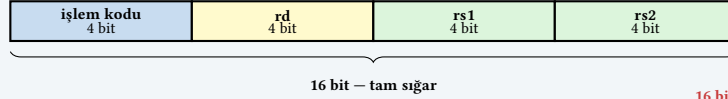
16 bitlik sabit genişlikli buyruk kullanan bir işlemci tasarladığınızı düşünün. Y (Yazmaç) türü buyruklar (ör. add rd, rs1, rs2) üç yazmaç numarası ve bir işlem kodu alanı içermelidir. A (Anlık) türü buyruklar (ör. addi rd, rs1, 5) ise iki yazmaç numarası, bir işlem kodu ve bir anlık değer alanı taşımalıdır. İşlem kodu her iki türde de 4 bit olsun. Yazmaç sayısını sırasıyla 8, 16 ve 32 olarak seçtiğinizde kalan bit sayılarını hesaplayın.

Çözüm:

(a) 8 yazmaç – Y türü



(b) 16 yazmaç – Y türü



(c) 32 yazmaç – Y türü



Şekil 3.7: 16 bitlik buyruklarda yazmaç sayısının etkisi (Y türü buyruklar)

Yazmaç sayısı	Yazmaç num. (bit)	Y türü (3 yazmaç) Toplam	Kalan	A türü (2 yazmaç) Anlık değer (bit)
(a) 8	3	$4 + 3 \times 3 = 13$	3 bit boş	$16 - 4 - 3 \times 2 = 6$ bit (± 32)
(b) 16	4	$4 + 3 \times 4 = 16$	0 bit	$16 - 4 - 4 \times 2 = 4$ bit (± 8)
(c) 32	5	$4 + 3 \times 5 = 19$	sığmaz!	$16 - 4 - 5 \times 2 = 2$ bit (± 2)

Şekil 3.7’de Y türü buyrukların bit dağılımı görselleştirilmiştir.

- 8 yazmaçla (a) her şey rahatça sığar; anlık değer alanı 6 bit olup -32 ’den $+31$ ’e kadar bir aralığı ifade edebilir.
- 16 yazmaçla (b) Y türü buyruk 16 bitin tamamını doldurur, hiç boş bit kalmaz. Anlık değer alanı ise 4 bite düşer; artık yalnızca -8 ’den $+7$ ’ye kadar bir aralık gösterilebilir. Daha büyük sabitler için ek buyruklar gerekir.
- 32 yazmaçla (c) Y türü buyruk 19 bit gerektirir; 16 bite sığmaz. A türünde anlık değer alanı yalnızca 2 bit kalır (-2 ’den $+1$ ’e); bu da neredeyse kullanışsızdır.

Bu örnek, RISC-V’in neden 32 bitlik buyruk genişliği seçtiğini ve RISC-V C (sıkıştırılmış) uzantısının 16 bitlik buyruklarda yazmaç sayısını 8’e düşürmek zorunda kaldığını açıkça göstermektedir.

Günümüz mimarilerinin çoğu 16–32 aralığında bir yazmaç sayısı bu dengeyi kurmaktadır. x86’nın 8 yazmacı yetersiz kalmış, x86-64’te 16’ya çıkarılmıştır. RISC-V ve MIPS’in 32 yazmacı ise derleyicilere yeterli esnekliği sağlarken buyruk kodlama genişliğini kabul edilebilir sınırlarda tutmaktadır.

Yazmaç Yapıları

RISC-V 32 adet genel amaçlı tam sayı yazmacına sahiptir; $x0$ donanım tarafından sıfıra sabitlenmiştir ve üzerine yazılamaz. MIPS de 32 yazmaç sunar ($\$0 = \text{sıfır}$); ancak çarpma ve bölme sonuçları için ayrı HI ve LO yazmaçları bulunur ve bunlara erişmek için özel `mfhi/mflo` buyruklarına gerek duyulur. RISC-V bu durumu M uzantısıyla genel amaçlı yazmaçlara yazarak çözmüştür. ARM (AArch64) 31 adet genel amaçlı yazmaç ($X0$ – $X30$) ile birlikte yazmaç konumuna göre sıfır ya da yığıt göstergesi gibi davranan XZR/SP rolüne sahiptir; bu da etkin olarak 32 konuma karşılık gelir.

x86-64 ise yalnızca 16 genel amaçlı yazmaç sunar (RAX–R15). IA-32’de bu sayı 8’di. Üstelik bazı yazmaçların örtük rolleri vardır: RCX döngü sayacı, RSP yığıt göstergesi, RAX çarpma/bölme sonucu, RDX ise yüksek çıktı yazmacı olarak kullanılır.

3.4.7 Yazmaçlarda Veri Gösterimi

İkilik tabandaki sayı gösterimlerini öğrendiğimize göre, şimdi bu bilgilerin RISC-V yazmaçlarında nasıl kullanıldığına bakalım. RISC-V RV32I temel buyruk kümesi 32 adet 32 bitlik yazmaca sahiptir ve veriler ikiye tümleyen yöntemle saklanır. Yazmaçlardaki basamaklar sağdan sola doğru 0'dan 31'e kadar numaralandırılır: en sağdaki bit (bit 0) en anlamsız, en soldaki bit (bit 31) ise en anlamlı basamaktır.

Bilgi Notu 3.6: RV64I'de 64 Bit Yazmaçlar

RV64I uzantısında yazmaçlar 64 bit genişliğindedir. Bu durumda işaretsiz sayılar için en büyük değer $2^{64} - 1$, işaretli sayılar için en küçük değer -2^{63} , en büyük değer $2^{63} - 1$ olur.

Eğer saklama alanına yazılan verinin işaretsiz olduğu varsayılırsa, RISC-V RV32I’de 32 bitle gösterilebilecek en büyük sayı $2^{32} - 1 = 4.294.967.295$, en küçük sayı ise 0’dır.

Eğer saklanan sayının işaretli olduğu varsayılırsa, 2'ye tümleyen kuralına göre:

- En küçük tam sayı: $-2.147.483.648$ (-2^{31})
- En büyük tam sayı: $2.147.483.647$ ($2^{31} - 1$)

Örnek 3.6

RISC-V'te (RV32I'de) 32 sayısı nasıl gösterilir?

Çözüm:

$(32)_{10} = (100000)_2$. Sayı 32 bite tamamlanır:

[illegible]

Örnek 3.7

RISC-V'te (RV32I'de) -43 sayısı nasıl gösterilir?

Çözüm:

$(43)_{10} = (101011)_2$. Sayı 32 bite tamamlanır ve ikiye tümleyeni alınır:

[illegible]

Kural Dışı Durum

İşlemcinin normal buyruk yürütme akışını bozan beklenmedik olaya **kural dışı durum** (*exception*) denir. Aritmetik taşma, tanımsız buyruk, hizasız bellek erişimi veya sayfa hatası bunlara örnektir. Kural dışı durum oluştuğunda işlemci, denetimi bir hizmet yordamına (*exception handler*) aktarır. Bu kitapta İngilizce *exception* karşılığı olarak, Türk Bilişim Derneği (TBD) sözlüğüyle uyumlu biçimde “kural dışı durum” terimi kullanılır.

maktadır.

Saklanan 32 bitlik bir sayının işaretli ya da işaretsiz olduğunu donanım bilmez. Bu durumu bilecek olan programcı ya da derleyicidir. RISC-V’te aritmetik buyruklar varsayılan olarak taşma kural dışı durumu üretmez; bu nedenle MIPS’teki gibi add/addu ayrımı yoktur. Taşma denetimi gerekiyorsa programcının ya da derleyicinin ek buyruklar ekleyerek bunu açıkça sınaması gerekir.

Çarpma ve bölme işlemlerinde sonucun yazmaç genişliğini aşması sorunu ve farklı mimarilerin bu duruma yaklaşımı, Bölüm 3.8.1’de ele alınmaktadır.

3.5 Bellek Düzeni

Tek tek bitlerle uğraşmak ne kadar zor olurdu, bir düşünün: her seferinde 0 ya da 1’lerle tek tek ilgilenmeniz gerekirdi. Bu yüzden bitler gruplar halinde düzenlenir. Bu grupların en temel birimi, 8 bitten oluşan **bayt**’tır.

Belleği devasa bir kutu dizisi olarak hayal edebilirsiniz: her kutunun bir adresi vardır ve her kutu tam bir baytlık (8 bitlik) veri tutar. Günümüz işlemcilerinin tamamı bu “bayt adreslenebilir” bellek yapısını kullanır (Şekil 3.8).

Ancak bu her zaman böyle değildi. Bayt adresleme IBM System/360 (1964) ile yaygınlaştı; bu mimari 8 bitlik baytı endüstri standardı haline getirdi. Öncesinde işlemciler bellekte en küçük birim olarak *sözcük* kullanırdı ve sözcük boyutları mimariden mimariye farklılık gösterirdi: CDC 6600 (1964) 60 bitlik, Cray-1 (1976) 64 bitlik, PDP-10 (1966) ise 36 bitlik sözcüklerle çalışırdı. Bu işlemcilerde her adres bir sözcüğe karşılık gelirdi; sözcüğün içindeki tekil bir bayta doğrudan erişmek mümkün değildi. Günümüzde bazı sayısal sinyal işlemcileri (DSP) hâlâ 16 veya 32 bitlik sözcük adresleme kullanmaktadır.

3.5.1 Sözcük Kavramı

Baytlar bellekte adreslenebilir en küçük birim olsa da işlemci veri yolunda tek tek baytlarla değil, **sözcük** (*word*) adı verilen daha büyük gruplarla çalışır. Sözcük, bir işlemcinin tek bir işlemde doğal olarak işleyebildiği veri miktarıdır ve boyutu mimariden mimariye değişir:

Mimari	Sözcük Boyutu	Dönem
Intel 8080	8 bit (1 bayt)	1974
Motorola 68000	16 bit (2 bayt)	1979
MIPS, ARM (32 bit), RISC-V RV32	32 bit (4 bayt)	1985–
x86-64, ARM AArch64, RISC-V RV64	64 bit (8 bayt)	2003–
CDC 6600	60 bit	1964
PDP-10	36 bit	1966

Burada iki ayrı kavramı birbirine karıştırmamak gerekir: bellek *bayt* biriminde adreslenir, ancak işlemci belleğe genellikle *sözcük* biriminde erişir. Örneğin RV32I’de bellekte her baytın ayrı bir adresi vardır; ancak *lw* (load word) buyruğu tek seferde 4 ardışık baytı (bir sözcüğü) okur, *lb* (load byte) ise yalnızca tek bir bayt okur. Bu ikili yapı esneklik sağlar: metin işlemede karakterlere tek tek erişmek için *lb*, tam sayı hesaplamalarında 32 bitlik değerlerle çalışmak için *lw* kullanılır.

Adresleme Biriminin Boyutu: Bir Ödünleşme

En küçük adreslenebilir birimin boyutu önemli bir tasarım kararıdır:

Birim küçülürse (bayt adresleme): Bellekteki her bayta ayrı ayrı erişilebilir; metin işleme ve ağ protokolleri gibi bayt düzeyinde çalışan uygulamalar doğal biçimde desteklenir. Ancak aynı bellek kapasitesini adreslemek için daha fazla adres biti gerekir. Örneğin 32 bitlik bir adres uzayında bayt adreslemeyle 4 GB belleğe erişilirken, sözcük adreslemeyle (4 bayt sözcük) aynı adres bitleriyle 16 GB'a erişilebilirdi.

Birim büyürse (sözcük adresleme): Adres bitleri daha verimli kullanılır ve adres hesaplamaları basitleşir (her adres doğrudan bir sözcüğe karşılık gelir, hizalama sorunu ortadan kalkar). Ancak tek bir karaktere erişmek için sözcüğün tamamını okuyup istenen baytı maskeleye ve kaydırma ile çıkarmak gerekir; bu hem fazladan buyruk hem de bant genişliği israfı demektir.

Bayt adreslemesinin bugün evrensel standart olması, yazılım dünyasının karakter ve bayt düzeyinde veri işleme ihtiyacının, adres alanı verimliliğinden daha ağır bastığını gösterir.

(a) Dizi şeklinde Bellek

Adres 0	Bayt 0
Adres 1	Bayt 1
Adres 2	Bayt 2
Adres 3	Bayt 3
Adres 4	Bayt 4
Adres 5	Bayt 5
Adres 6	Bayt 6
Adres 7	Bayt 7

(b) Matris şeklinde Bellek

	Sütun 0	Sütun 1	Sütun 2	Sütun 3
Satır 0	0	1	2	3
Satır 1	4	5	6	7
Satır 2	8	9	10	11
Satır 3	12	13	14	15

Şekil 3.8: Bellek yapısının iki gösterimi: dizi ve matris biçiminde

Şekil 3.8'te belleğin iki farklı görünümü yer almaktadır. Soldaki *dizi gösterimi*, belleği ardışık baytlardan oluşan tek boyutlu bir dizi olarak sunar; her satır tek bir bayta karşılık gelir ve her baytın kendine ait bir adresi vardır. Bu gösterim belleğin kavramsal yapısını en yalın biçimde yansıtır. Sağdaki *matris gösterimi* ise aynı belleği satır ve sütunlar halinde düzenler; her satırda birden fazla bayt yan yana gösterilir. Gerçek bellek yongalarının iç yapısı bu matris düzenine daha yakındır; bellek hücreleri satır ve sütun adresleriyle seçilir. Matris gösterimi aynı zamanda sözcük sınırlarını görmeyi de kolaylaştırır: dört sütunlu bir düzende her satır bir sözcüğe (4 bayt) karşılık gelir.

Bellek Erişim Modelleri

RISC-V, MIPS ve ARM yükle/sakla (*load/store*) mimarisini benimser: aritmetik ve mantık işlemleri yalnızca yazmaçlar üzerinde gerçekleşir, belleğe erişim ise ayrı *load/store* buyruklarıyla yapılır. Bu ayrım boru hattı tasarımını basitleştirir; her buyruk aşamasının ne yapacağı önceden bellidir.

x86 ise bellek işleneni (*memory operand*) kavramına izin verir: `ADD EAX, [rbx+4]` gibi bir buyruk hem belleği okur hem de aritmetik işlemi tek adımda gerçekleştirir. Bu esneklik programcıya kolaylık sağlasa da boru hattında hem bellek hem de yürütme biriminin aynı buyruk için harekete geçmesi gerekir.

Hizalama konusunda mimariler farklılaşır. RISC-V belirtimi hizasız erişime izin verir; ancak donanım bunu iki ayrı bellek okumasına bölerek çözebileceğinden başarımlar düşer. MIPS geleneksel olarak hizalı erişim zorunlu kılar; hizasız erişim kural dışı durum üretir. ARM (AArch64) hizasız erişime izin verir. x86 de hizasız erişimi destekler; ancak önbellek sınırını aşan erişimlerde gecikme oluşur.

RISC-V’te her buyruğun boyutu 4 bayt (32 bit) olduğu için buyrukların bellekten okunması sırasında aynı anda 4 baytlık bir sözcük okunur. Bellek bayt adreslendiğinden ardışık iki buyruğun adresleri arasında 4 bayt fark vardır; dolayısıyla program sayacı her buyruktan sonra 4 artırılır. Sözcük adreslemeli bir işlemcide aynı durum için program sayacı yalnızca 1 artırılırdı; çünkü her adres zaten bir sözcüğe karşılık gelirdi.

Bilgi Notu 3.7: RISC-V Sıkıştırılmış Buyruklar

RISC-V C (Compressed) uzantısı etkinleştirildiğinde bazı buyruklar 16 bit (2 bayt) uzunluğunda olabilir. Bu durumda program sayacı 2 artırılır. Ancak 16 bite sığabilmek için sıkıştırılmış buyruklar yalnızca 8 yazmaca ($x8-x15$, ABI adlarıyla $s0-s1$ ve $a0-a5$) erişilebilir; 32 yazmacının tamamı kullanılamaz. Temel RV32I buyruk kümesinde ise tüm buyruklar 32 bittir ve 32 yazmacının hepsine erişilebilir.

3.5.2 Belleğe İki Tür Erişim

İşlemci belleğe iki farklı amaçla erişir ve bu iki erişim birbirinden bağımsızdır:

1. **Buyruk çekme (*instruction fetch*):** Program sayacının gösterdiği adresten bir sonraki buyruğun okunmasıdır. RV32I’de her buyruk 32 bit olduğundan bu erişim her zaman 4 baytlık bir sözcük okur.
2. **Veri erişimi (*data access*):** lw/sw , lh/sh , lb/sb gibi yükle/sakla buyruklarıyla yapılır. Burada okunan veya yazılan verinin boyutu buyruğa bağlıdır: lb 1 bayt, lh 2 bayt (yarım sözcük), lw 4 bayt (sözcük) okur.

Bu ayrım önemli bir sonuç doğurur: “sözcük” kavramı bağlama göre farklı boyutlara karşılık gelebilir. Buyruk sözcüğü RV32I’de her zaman 32 bittir; ancak C uzantısı etkinleştirildiğinde bazı buyruklar 16 bit olabilir. Veri sözcüğü ise erişim türüne bağlıdır. Farklı mimarilerde bu boyutlar değişebilir: örneğin PDP-11’de sözcük 16 bit, VAX’ta 32 bit, çağdaş 64 bit işlemcilerde ise “çift sözcük” (*doubleword*) olarak 64 bittir.

İşlemcinin adresleyebileceği bellek uzayı adres çıkışındaki bit sayısına bağlıdır. RV32I’de 32 bitlik adres çıkışıyla 2^{32} bayt ya da 2^{30} sözcük adreslenebilir.

3.5.3 Bellek Hizalama

Verilerin bellekte hizalı olarak durması demek, sözcüklerin saklandığı bellek konumunun ilk baytının adresi sözcük boyutunun tam katı olacak biçimde saklanması demektir (Şekil 3.9). RISC-V temel belirtiminde hizalanmamış bellek erişimine izin verilmektedir; ancak hizalı erişim başarımlar açısından tercih edilir ve çoğu RISC-V gerçeklemede sözcükler 4’ün katı olan adreslerden başlanarak konumlandırılır.

(a) Hizalı Erişim					(b) Hizasız Erişim				
Sözcük (4 bayt), Adres 8 (4'ün katı ✓)					Adres 5 (4'ün katı değil ×): 2 satır okunmalı!				
Adres	+0	+1	+2	+3	Adres	+0	+1	+2	+3
0	0	1	2	3	0	0	1	2	3
4	4	5	6	7	4	4	5	6	7
8	8	9	10	11	8	8	9	10	11
12	12	13	14	15	12	12	13	14	15

Şekil 3.9: Bellekte hizalı ve hizasız veri erişimi. Her hücredeki sayı bayt adresini gösterir; satır başlarındaki adresler sözcük sınırlarıdır (4'ün katları).

Peki bir program hizasız adrese erişmeye çalışırsa ne olur? Bu sorunun yanıtı mimariden mimariye değişir:

Hizasız Erişimde Mimari Yaklaşımlar

Mimariler, hizasız bellek erişimine üç farklı stratejiyle yanıt verir:

1. **Yasakla (kural dışı durum üret):** MIPS ve bazı ARM gerçeklemeleri bu yaklaşımı benimser. Hizasız erişim denendiğinde işlemci bir kural dışı durum (*exception*) üretir ve denetimi işletim sistemine bırakır. Donanım en basit haliyle kalır; hizalama sorumluluğu derleyiciye ve programcıya aittir.
2. **Donanımda sessizce çöz:** x86 bu yaklaşımı kullanır. İşlemci hizasız erişimi algılar, gerekli 2 bellek okumasını otomatik olarak yapar ve baytları birleştirerek sonucu yazmaca yazar. Programcı farkına bile varmaz; ancak hizalı erişime göre daha yavaştır; özellikle önbellek satır sınırını aşan erişimlerde gecikme belirgin biçimde artar.
3. **Belirtimde izin ver, gerçeklemeye bırak:** RISC-V bu esnek yolu seçer. Belirtim hizasız erişimi yasaklamaz; ancak gerçekleştirme bunu donanımda çözebilir ya da kural dışı durum üretebilir. Bu esneklik, basit gömülü sistemlerin kural dışı durum yolunu, yüksek başarımlı işlemcilerin ise donanım çözümünü tercih etmesine olanak tanır.

Sonuç olarak “hizasız erişime izin vermek” ile “hizasız erişimi verimli yapmak” birbirinden farklı kavramlardır. Derleyiciler bu nedenle verileri mümkün olduğunca hizalı yerleştirir; böylece hangi strateji kullanılırsa kullanılsın en iyi başarımlar elde edilir.

3.5.4 Verilerin Bellekte Yerleşimi: Bayt Sırası

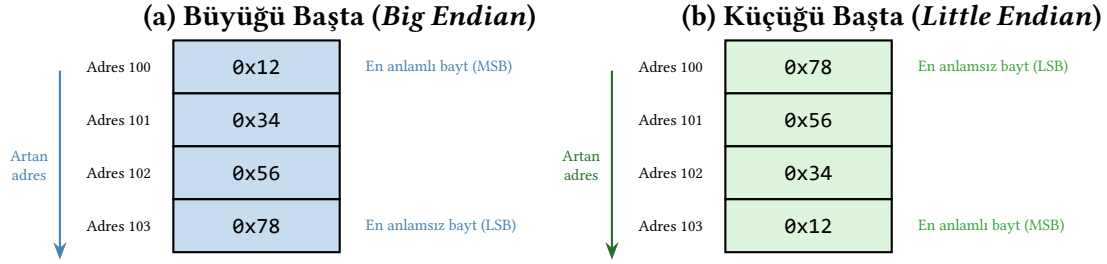
Belleğe birden fazla bayt içeren bir veri yerleştirilirken hangi baytın küçük adres konumuna yazılacağı bir tasarım kararıdır (Şekil 3.10):

- **Küçüğü Başta (Little-Endian):** Sayının en anlamsız baytı (LSB) en küçük adresli bellek konumuna yazılır.
- **Büyüğü Başta (Big-Endian):** Sayının en anlamlı baytı (MSB) en küçük adresli bellek konumuna yazılır.

Bilgi Notu 3.8: Gülliver'in Yumurta Savaşı

Big-endian ve *little-endian* terimleri, Jonathan Swift'in 1726 tarihli *Gülliver'in Gezileri* romanından esinlenilerek bilgisayar bilimine kazandırılmıştır. Romanda Lilliput ve Blefuscu imparatorlukları, haşlanmış yumurtanın hangi ucundan kırılması gerektiği konu-

sunda şiddetli bir savaşa tutuşur: bir taraf kalın ucundan (*big end*), diğeri sivri ucundan (*little end*) kırılmasını savunur. Danny Cohen, 1980 yılında yayımladığı “On Holy Wars and a Plea for Peace” başlıklı makalesinde [8] bu hikâyeye gönderme yaparak bayt sırası tartışmasını *endian* kavramıyla adlandırmıştır. Tıpkı romandaki yumurta savaşı gibi, bayt sırası tercihi de teknik açıdan kesin bir üstünlük sağlamaz; ancak iki farklı düzeni kullanan sistemler arasında veri alışverişi yapılırken ciddi uyumluluk sorunlarına yol açar.



Saklanan 32 bitlik değer: 0x12345678

Şekil 3.10: Büyüğü başta ve küçüğü başta bayt sırası düzenleri

RISC-V belirtiminde temel bellek düzeni **küçüğü başta**'dır. Bu, Intel x86 ile aynı bayt sıralama düzenini kullanır.

Tablo 3.7: İşlemci ailelerinin bayt sırası tercihleri

İşlemci / Mimari	Yıl	Bayt Sırası	Not
IBM System/360	1964	Büyüğü başta	Anabilgisayar geleneği
PDP-11	1970	Küçüğü başta	Unix'in doğduğu platform
Motorola 68000	1979	Büyüğü başta	Macintosh, Amiga
Intel 8086 / x86	1978	Küçüğü başta	PC standardı
SPARC (Sun)	1987	Büyüğü başta	Sunucu/iş istasyonu
MIPS	1985	Çift yönlü	Varsayılan: büyüğü başta
IBM PowerPC	1992	Çift yönlü	Varsayılan: büyüğü başta
ARM	1985	Çift yönlü	Varsayılan: küçüğü başta
Itanium (IA-64)	2001	Çift yönlü	Varsayılan: küçüğü başta
RISC-V	2010	Küçüğü başta	Her ikisine izin verilir; uygulamada küçüğü başta
Ağ protokolleri	–	Büyüğü başta	“Ağ bayt sırası”

Tablo 3.7’te görüldüğü gibi, erken dönem anabilgisayarlar ve Motorola ailesi büyüğü başta düzenini benimserken, Intel ve PDP-11 geleneğinden gelen mimariler küçüğü başta düzenini tercih etmiştir. 1980’lerden itibaren tasarlanan pek çok mimari (MIPS, PowerPC, ARM) her iki düzeni de destekleyen çift yönlü (*bi-endian*) yapıyı benimsemiştir; ancak günümüzde küçüğü başta uygulamada baskın düzen haline gelmiştir.

Bilgi Notu 3.9: Ortası Başta

Az sayıda işlemci, baytları ne tamamen büyükten küçüğe ne de küçükten büyüğe sıralar. Örneğin PDP-11 üzerinde çalışan bazı C derleyicileri, 32 bitlik bir değerin iki 16 bitlik yarısını büyüğü başta sırada tutarken her yarının kendi içindeki baytlarını küçüğü başta sıralar. $0x12345678$ değeri bellekte 34 12 78 56 biçiminde saklanır. Bu düzene *ortası başta* (*middle-endian*) ya da bazen *karma sıralama* (*mixed-endian*) denir. Günümüz mimarilerinde kullanılmaz; ancak eski sistemlerle veri alışverişinde karşılaşılabılır.

Bayt Sırası Tercihleri

x86 her zaman küçüğü başta düzenini kullanır; bu tercih değiştirilemez. RISC-V de varsayılan olarak küçüğü başta çalışır; belirtim büyüğü başta kipini tanımlasa da hemen her RISC-V gerçeği küçüğü başta düzenini benimsemiştir.

MIPS ve ARM çift yönlü mimarilerdir: donanım her iki düzeni de destekleyebilir. ARM işlemciler büyük çoğunlukla küçüğü başta kipinde çalışır; Android ve Linux çekirdekleri de bu kipi kullanır. MIPS ise tarihsel olarak büyüğü başta ile anılır (SGI iş istasyonları, ağ donanımları); ancak günümüz MIPS çekirdekleri küçüğü başta kipini de destekler.

Ağ protokolleri (TCP/IP, UDP) büyüğü başta düzenini, bir başka adıyla “ağ bayt sırası”nı (*network byte order*) kullanır. Küçüğü başta makinelerde ağ paketleri işlenirken `hton1/ntoh1` gibi dönüşüm işlevleri gerekir.

Bir programın en verimli şekilde çalışması için buyruklar mümkün olduğunca yazmaçları kullanılmalıdır. Bellek ile işlemci arasındaki veri aktarım işlemini sağlayan buyruklara **yükleme** (*load*) ve **saklama** (*store*) buyrukları denir.

3.6 Buyruklar

Bellekten gelen bir buyruk, işlemcinin gözünde 32 bitlik bir sayıdan ibarettir. Peki işlemci bu sayıya bakarak “bu bir toplama buyruğu, şu yazmaçları kullan” diye nasıl anlıyor? İşlemci, bu bit dizisini belirli parçalara ayırarak her parçanın anlamını çözümler. Bu aşamaya **çözme aşaması** (*decode*) denir.

3.6.1 Buyrukların Yapısı

Buyruğun içindeki her bitin ayrı bir anlamı vardır. Örneğin “`add s1, s2, s3`” buyruğu (donanım adlarıyla `add x9, x18, x19`) 32 bitle gösterilir ve Tablo 3.8’de anlamlı parçalara ayrılmıştır:

Tablo 3.8: RISC-V’te R türü buyruğun anlamlı parçalara ayrılması

funct7	rs2	rs1	funct3	rd	opcode
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit
0000000	10011	10010	000	01001	0110011

Tablo 3.9: *RISC-V’te bir R türü buyruğun alanları*

Alan	Açıklama
opcode	Buyruğun temel türünü ve yapacağı işlemi belirler (7 bit)
rd	Sonuç yazmacının adresi (5 bit)
funct3	Yapılacak işlemin alt türünü belirler (3 bit)
rs1	Birinci kaynak yazmacının adresi (5 bit)
rs2	İkinci kaynak yazmacının adresi (5 bit)
funct7	İşlemin daha ayrıntılı türünü belirler (7 bit)

Tablo 3.9’de her alanın işlevi özetlenmiştir.

Tasarım İlkesi 2: İyi tasarım, iyi ödünleşme gerektirir (*devam*)

Bölüm 2’de tanıttığımız bu ilkenin burada somut bir yansımasını görüyoruz. Bir değişkenin iyi yönde geliştirilmesiyle elde edilen kazancın diğer değişkenlerdeki olası kayıpları karşılaması ve toplamda sistem başarımını artırması gerekir.

3.6.2 Aynı İşlem, Farklı Biçim: add ve addi Neden Ayrı Buyruklardır

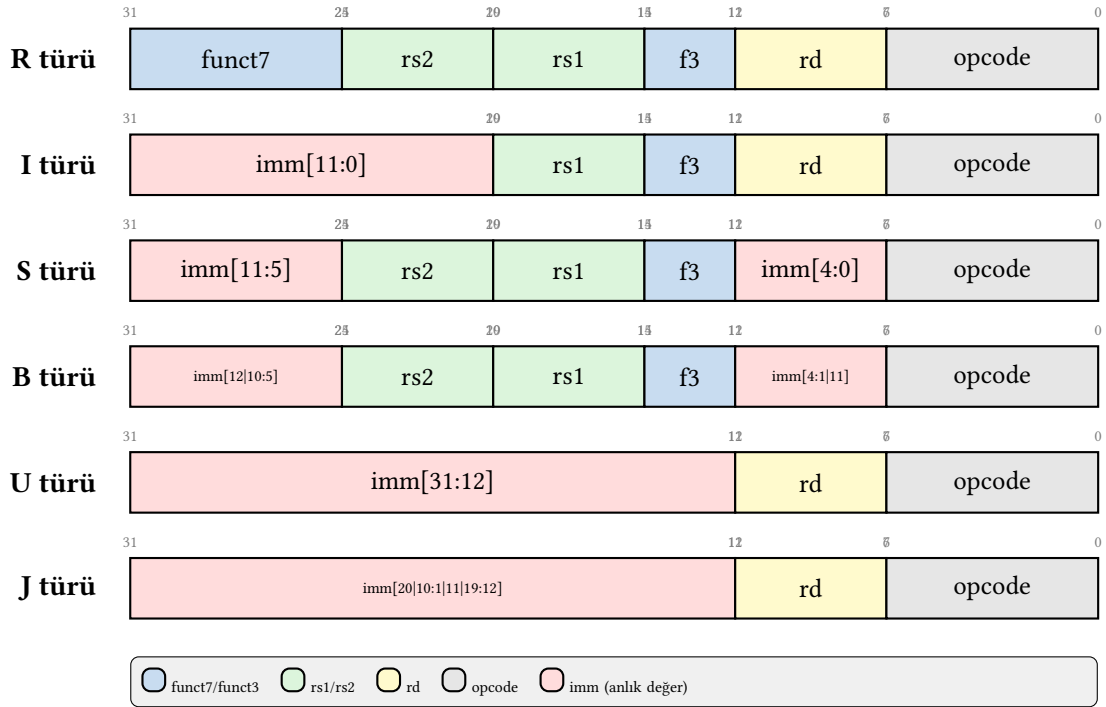
Öğrencilerin sık karşılaştığı bir yanılgı, add s1, s2, s3 ile addi s1, s2, 5 buyruklarını “aynı toplama buyruğunun farklı yazılışları” sanmaktır. Oysa bunlar donanım düzeyinde tamamen farklı buyruklardır:

- add bir R türü buyruktur: iki kaynak yazmaç (rs1, rs2) ve bir hedef yazmaç (rd) kullanır. İşlem kodu alanları: opcode=0110011, funct3=000, funct7=0000000.
- addi bir I türü buyruktur: bir kaynak yazmaç (rs1), bir hedef yazmaç (rd) ve 12 bitlik bir anlık değer (imm) kullanır. İkinci kaynak yazmaç alanı yoktur; onun yerine anlık değer alanı vardır. İşlem kodu alanları: opcode=0010011, funct3=000.

İşlemcinin çözme birimi, buyruğun opcode alanına bakarak hangi biçimle karşı karşıya olduğunu anlar ve bit alanlarını buna göre yorumlar. R türünde rs2 alanı olarak okunan bitler, I türünde anlık değer bir parçasıdır. Bu ayrım, sabit genişlikli buyruk biçimlerinin doğal bir sonucudur: işlenenler farklı türde olduğunda (yazmaç mı, sabit mi?) buyruğun iç yapısının da farklı olması gerekir.

3.6.3 Buyruk Türleri

RISC-V’te altı tür buyruk biçimi vardır: R, I, S, B, U ve J; bu biçimlerin bit düzeyindeki yapısı Şekil 3.11’de gösterilmiştir (ayrıca bkz. Ek C, Bölüm C.5).



Şekil 3.11: RISC-V’te Buyruk Biçimleri (R, I, S, B, U, J türleri)

R (Register) Türü Buyruklar

Yazmaçlardan veri okuyarak bu verilerin üzerinde aritmetik ya da mantık işlemleri yapan buyruklardır. Kaynak verisi olarak *rs1* ve *rs2* ile gösterilen yazmaçların değerlerini kullanıp sonuçlarını *rd* ile gösterilen yazmaca yazarlar.

I (Immediate) Türü Buyruklar

Derleyiciden donanıma sabit değerlerin geçirilmesi gereken buyruklardır. RISC-V’te anlık değerler 12 bit uzunluğundadır ve işaretle genişletilir. Bu da -2048 ile $+2047$ arasındaki değerlerin temsil edilebileceği anlamına gelir.

S (Store) Türü Buyruklar

Bellekte saklama işlemi yapan buyruklardır. Anlık değer, buyruğun iki ayrı alanına bölünerek yerleştirilir.

B (Branch) Türü Buyruklar

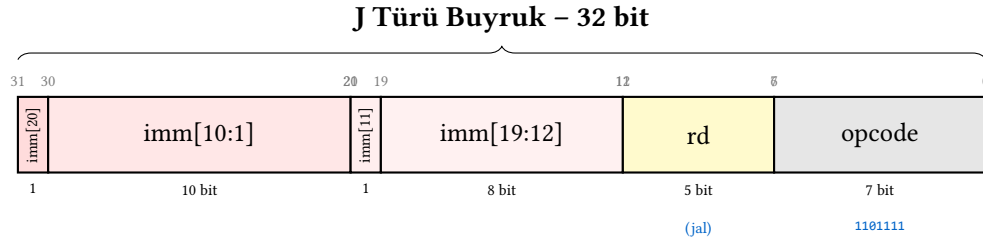
Koşullu dallanma buyruklarıdır. Dallanma uzaklığı PS’ye göreceli olarak hesaplanır ve anlık değer 13 bit genişliğinde olup 2^n ’nin katı olan adresleri gösterir.

U (Upper) Türü Buyruklar

20 bitlik bir anlık değeri hedef yazmacın üst 20 bitine ([31:12]) yazan buyruklardır. Alt 12 bit sıfırlanır.

J (Jump) Türü Buyruklar

Koşulsuz atlama buyruklarıdır. 21 bitlik işaretli bir uzaklık değeri kullanırlar ve bu değer 2'nin katıdır. J türü buyruğun bit alanları Şekil 3.12'da ayrıntılı olarak gösterilmiştir.



Şekil 3.12: RISC-V’te J Türü Buyruk yapısı

Tablo 3.10: *RISC-V* buyruk biçimlerinin karşılaştırması

Tür	Kullanım	Örnek Buyruklar
R	Yazmaç-yazmaç aritmetik/mantık	add, sub, and, or, sll, slt
I	Anlık değerli işlemler, yükleme	addi, lw, lb, jalr
S	Bellekte saklama	sw, sh, sb
B	Koşullu dallanma	beq, bne, blt, bge
U	Üst anlık değer yükleme	lui, auipc
J	Koşulsuz atlama	jal

Tablo 3.10’da her biçimin hangi tür buyrukları kapsadığı özetlenmiştir.

Buyruk Biçimlerinin Karşılaştırması

MIPS mimarisi yalnızca 3 buyruk biçimi (R, I, J) kullanır; bu yapı daha basittir ancak esneklik açısından kısıtlıdır. ARM AArch64 sabit 32 bitlik buyruklar kullanır; biçim gruplandırması farklıdır. x86 mimarisi ise 1–15 bayt arasında değişen uzunlukta buyruklar içerir: önek, REX/VEX, işlem kodu, ModRM, SIB, yer değiştirme (displacement) ve anlık değer alanlarıyla oluşan bu yapının çözümlenmesi sıralı bayt incelemesi gerektirir. RISC-V’in 6 biçimi (R, I, S, B, U, J) ise dengeli bir çözüm sunar: farklı buyruk gereksinimlerini karşılayacak kadar çeşitlilik sağlarken çözümleme mantığını düzenli tutar.

3.7 Veri Aktarma Buyrukları

İşlemci, aritmetik ve mantık işlemlerini yalnızca yazmaçlardaki veriler üzerinde yapabilir. Ancak veriler genellikle bellekte saklanır. O halde bellekten yazmaca veri taşıyan ve yazmaçtan belleğe veri geri yazan buyruklara ihtiyacımız var. RISC-V’te bu amaçla **yükleme** (load) ve **saklama** (store) buyrukları kullanılır; bu buyrukların tamamı Tablo 3.11’de listelenmiştir.

Tablo 3.11: RISC-V'te veri aktarma buyrukları

Buyruk	Söz Dizimi	Anlam	Tür
Sözcük Yükle (lw)	lw rd, imm(rs1)	rd = Bel[rs1+imm]	I
Yarım sözcük Yükle (lh)	lh rd, imm(rs1)	rd = Bel[rs1+imm]	I
Bayt Yükle (lb)	lb rd, imm(rs1)	rd = Bel[rs1+imm]	I
İşaretsiz Yarım s. Yükle (lhu)	lhu rd, imm(rs1)	rd = Bel[rs1+imm]	I
İşaretsiz Bayt Yükle (lbu)	lbu rd, imm(rs1)	rd = Bel[rs1+imm]	I
Sözcük Sakla (sw)	sw rs2, imm(rs1)	Bel[rs1+imm] = rs2	S
Yarım sözcük Sakla (sh)	sh rs2, imm(rs1)	Bel[rs1+imm] = rs2	S
Bayt Sakla (sb)	sb rs2, imm(rs1)	Bel[rs1+imm] = rs2	S
Üst Anlık Yükle (lui)	lui rd, imm	rd = imm << 12	U
PS'ye Üst Anlık Ekle (auipc)	auipc rd, imm	rd = PS + (imm << 12)	U

RISC-V'te MIPS'ten farklı olarak move, mfhi, mflo gibi buyruklar yoktur. Yazmaçlar arası kopyalama `addi rd, rs, 0` buyruğuyla yapılır ve çevirici dilde `mv` sözde buyruğu olarak yazılır.

Örnek 3.8

RISC-V buyruklarını kullanarak 32 bitlik $(00F800A1)_{16}$ sayısını `s3` yazmacına yazın.

Çözüm:

RISC-V'te `lui` buyruğu 20 bitlik anlık değeri yazmacın üst 20 bitine (bit [31:12]) yükler ve alt 12 biti sıfırlar. Ardından `addi` buyruğu ile alt 12 bit eklenir.

$(00F800A1)_{16}$ sayısı için üst 20 bit = $(00F80)_{16}$, alt 12 bit = $(0A1)_{16}$ olur. Alt 12 bitin en anlamlı biti (bit 11) 0 olduğu için `addi` ile doğrudan eklenebilir:

```
1 lui s3, 0x00F80 # üst 20 bit: s3 = 0x00F80000
2 addi s3, s3, 0x0A1 # alt 12 bit: s3 = 0x00F800A1
```

Bilgi Notu 3.10: Anlık Değerlerde İşaret Genişletme Düzeltmesi

RISC-V'te `addi` buyruğu anlık değeri işaretle genişletir. Eğer alt 12 bitin en anlamlı biti (bit 11) 1 ise, işaretle genişletme sonucunda üst bitlere 1'ler ekleneceğinden, `lui`'ye verilen üst 20 bitlik değerin 1 artırılması gerekir.

Örnek 3.9

RISC-V'te `mv t1, s1` sözde buyruğunu donanım buyruklarıyla gösterin.

Çözüm:

```
1 addi t1, s1, 0 # s1 + 0 = s1, sonuç t1'e yazılır
```

Veri Aktarma Yaklaşımları

x86 mimarisinde aritmetik buyruklar doğrudan bellek işleneni kabul eder (örn. `ADD EAX, [adres]`); bu özellik basit durumlar için ayrı yükleme/saklama adımını ortadan kaldırır, ancak boru hattını karmaşılaştırır. MIPS, ARM ve RISC-V mimarileri ise yükleme-saklama (load-store)

ilkesini benimser: verinin işlenebilmesi için önce yazmaçlara yüklenmesi gerekir. ARM, verimlilik açısından çift yükleme/saklama buyrukları (LDP/STP) sunar. MIPS benzer işlevler için lw/sw kullanır. x86'nın bellek işleneni yaklaşımı buyruk karmaşıklığını artırır; buna karşın bellekten yoğun biçimde veri okuyan kodlarda buyruk sayısını azaltır.

3.8 Aritmetik İşlem Buyrukları

Tablo 3.12: RISC-V'te aritmetik işlem buyrukları

Buyruk	Söz Dizimi	Anlam	Tür
Topla (add)	add rd, rs1, rs2	$rd = rs1 + rs2$	R
Çıkar (sub)	sub rd, rs1, rs2	$rd = rs1 - rs2$	R
Anlık Topla (addi)	addi rd, rs1, imm	$rd = rs1 + imm$	I
<i>M Uzantısı (Çarpma/Bölme) Buyrukları:</i>			
Çarp (mul)	mul rd, rs1, rs2	$rd = (rs1 * rs2)[31:0]$	R
Çarp Üst (mulh)	mulh rd, rs1, rs2	$rd = (rs1 * rs2)[63:32]$	R
Böl (div)	div rd, rs1, rs2	$rd = rs1 / rs2$	R
Kalan (rem)	rem rd, rs1, rs2	$rd = rs1 \% rs2$	R

Tablo 3.12'da görülen RISC-V aritmetik buyrukları MIPS'teki addu, addiu, subu gibi ayrı işaret-siz buyruklar içermez. RISC-V aritmetik buyrukları varsayılan olarak taşma kural dışı durumu üretmez; taşma denetimi gerektiğinde programcının ya da derleyicinin ek buyruklar yazması gerekir.

Örnek 3.10

C dilindeki $a = dizi[3] + b$; ifadesini RISC-V buyruklarıyla yazın. Dizinin başlangıç adresi s4, a değeri s2, b değeri s3 yazmacındadır.

Çözüm:

```
1 lw t0, 12(s4)    # dizi[3] değerini yükle (3*4=12)
2 add s2, s3, t0    # a = b + dizi[3]
```

3.8.1 Çarpma ve Bölme: Geniş Sonuç Sorunu

İki N bitlik tam sayının çarpımı $2N$ bitlik bir sonuç üretir. Örneğin RV32I'de iki 32 bitlik yazmaç çarpıldığında sonuç 64 bit genişliğindedir ve tek bir 32 bitlik yazmaca sığmaz. Benzer şekilde bölme işlemi hem bölüm hem kalan olmak üzere iki ayrı değer üretir. Bu durum tüm sabit genişlikli yazmaç mimarileri için bir tasarım kararı gerektirir: geniş sonuç nerede ve nasıl saklanacaktır?

Farklı mimariler bu soruna farklı yaklaşımlar benimsemiştir:

Tablo 3.13: Çarpma sonucunun saklanması: mimari yaklaşımlar

Mimari	Yöntem	Açıklama
MIPS	Özel HI/LO yazmaçları	mult rs, rt sonucu HI:LO çiftine yazar; programcı mfhi/mflo ile genel yazmaca taşır
x86	Örtük RAX:RDX çifti	imul tek işlenenli biçimde çarpan RAX'ta olmalıdır; alt 32 bit RAX'a, üst 32 bit RDX'e yazılır
ARM (AArch64)	Genel amaçlı yazmaçlar	mul xd, xn, xm alt 64 biti, smulh/umulh üst 64 biti ayrı bir yazmaca yazar
RISC-V (M uz.)	Genel amaçlı yazmaçlar	mul rd, rs1, rs2 alt 32 biti, mulh rd, rs1, rs2 üst 32 biti herhangi bir yazmaca yazar

MIPS mimarisinde çarpma buyruğu (mult) sonucu HI ve LO adlı iki özel yazmaca yazar. Programcı bu değerlere erişmek için mfhi (*move from HI*) ve mflo (*move from LO*) buyruklarını kullanmak zorundadır. Özel yazmaçlar boru hattı tasarımını karmaşıktırır ve derleyicinin yazmaç ataması sırasında ek kısıtlamalarla uğraşmasına neden olur.

x86 mimarisinde imul buyruğunun tek işlenenli biçimi, çarpanın RAX yazmacında bulunmasını zorunlu kılar. 64 bitlik sonucun alt yarısı RAX'a, üst yarısı RDX'e yazılır. Bu örtük yazmaç kullanımı, derleyicinin çarpma öncesinde RAX ve RDX'in içeriğini koruma altına almasını gerektirir.

ARM AArch64 mimarisinde 64 bitlik yazmaçlar sayesinde mul xd, xn, xm buyruğu iki 64 bitlik kaynağın çarpımının alt 64 bitini tek bir yazmaca yazar. Tam 128 bitlik sonuç gerektiğinde smulh (işaretli) veya umulh (işaretsiz) buyruğu üst 64 biti ayrı bir yazmaca verir. Her iki hedef de genel amaçlı yazmaç olduğundan özel yazmaç kısıtlaması yoktur.

RISC-V'te M (çarpma/bölme) uzantısı bu soruna en yalın yaklaşımı sunar. mul rd, rs1, rs2 buyruğu çarpım sonucunun alt 32 bitini, mulh rd, rs1, rs2 (işaretli×işaretli), mulhu (işaretsiz×işaretsiz) veya mulhsu (işaretli×işaretsiz) buyruğu ise üst 32 bitini herhangi bir genel amaçlı yazmaca yazar. Bölme ve kalan işlemleri için de ayrı buyruklar tanımlanmıştır:

- div rd, rs1, rs2: Bölüm sonucunu rd'ye yazar.
- rem rd, rs1, rs2: Kalanı rd'ye yazar.
- divu/remu: İşaretsiz bölme ve kalan.

Bu yaklaşımın avantajları açıktır: özel donanım yazmacı gerekmez, sonuç herhangi iki genel amaçlı yazmaca yazılabilir ve boru hattı tasarımı basit kalır.

Bilgi Notu 3.11: RV64I'de 32 Bitlik Çarpma ve Bölme

RV64I'de yazmaçlar 64 bit genişliğindedir. Bu durumda mulw, divw, remw gibi buyruklar 32 bitlik işlem yaparak sonucu 64 bite işaretli genişletir. 64 bitlik tam çarpım ise 128 bit üretir; mul alt 64 biti, mulh/mulhu/mulhsu üst 64 biti verir.

Örnek 3.11

$a = x \times y$ çarpmasının 64 bitlik tam sonucunu dört farklı mimaride elde edin. Her mimaride kaynak değerler 32 bitlik iki yazmaçtadır. Sonucun alt ve üst 32 bitini ayrı yazmaçlara yerleştirin.

Çözüm:

(a) RISC-V (M uzantısı): kaynak: s1, s2; sonuç: s3 (alt), s4 (üst)

```

1 mul s3, s1, s2      # s3 = (s1 * s2)[31:0] (alt 32 bit)
2 mulh s4, s1, s2     # s4 = (s1 * s2)[63:32] (üst 32 bit)

```

(b) MIPS: kaynak: \$s1, \$s2; sonuç: \$s3 (alt), \$s4 (üst)

```

1 mult $s1, $s2      # HI:LO = s1 * s2 (özel yazmaçlara yazar)
2 mflo $s3           # s3 = LO (alt 32 bit)
3 mfhi $s4           # s4 = HI (üst 32 bit)

```

(c) x86 (IA-32): kaynak: EBX (x); sonuç: EAX (alt), EDX (üst)

```

1 mov EAX, EBX      ; çarpan EAX'ta olmak zorunda
2 imul ECX          ; EDX:EAX = EAX * ECX
3                  ; sonuç: EAX = alt 32 bit, EDX = üst 32 bit

```

(d) ARM (AArch64): kaynak: W1, W2; sonuç: X3 (tam 64 bit)

```

1 smull X3, W1, W2   ; X3 = W1 * W2 (tam 64 bit, tek buyruk)

```

Tablo 3.14: Aynı çarpma işleminin dört mimarideki gerçekleştirimi

Mimari	Buy.	Kaynak	Sonuç	Kısıtlamalar
RISC-V	2	herhangi ikisi	herhangi ikisi	Kısıtlama yok
MIPS	3	herhangi ikisi	HI/LO → herhangi ikisi	Özel HI/LO yazmaçları; mfhi/mflo gerekli
x86	2	EAX (zorunlu) + herhangi biri	EAX:EDX (sabit)	Çarpan EAX'ta olmalı; sonuç sabit çifte yazılır
ARM	1	herhangi iki W yazmacı	herhangi bir X yazmacı	64-bit yazmaç; tam sonuç tek yazmacı sığar

ARM, 64 bitlik yazmaçlar sayesinde 32×32 çarpımını tek buyrukla tamamlar. RISC-V iki buyrukla sonucu herhangi iki yazmacı özgürce yazar. MIPS üç buyruk gerektirir ve özel yazmaçlara bağımlıdır. x86 ise iki buyruk kullansa da kaynak ve hedef yazmaçları sabittir; derleyici çarpma öncesinde EAX ve EDX'in içeriğini koruma altına almak zorundadır.

Aritmetik Buyruk Farklılıkları

RISC-V ve MIPS, 3 işlenenli biçim kullanır (add rd, rs1, rs2): hedef yazmaç ayrıdır, kaynak değerler korunur. x86 ise 2 işlenenli biçimi benimser (ADD EAX, EBX): hedef aynı zamanda birinci kaynaktır ve özgün değer üzerine yazılır. ARM AArch64, RISC-V gibi 3 işlenenli biçimi tercih eder. Çarpma ve bölme işlemlerindeki mimari farklılıklar Tablo 3.13'da özetlenmiştir.

3.9 Mantık Buyrukları

Tüm işlemciler, aritmetik işlemlerin yanı sıra bit düzeyinde mantık işlemlerini de destekler. VE, VEYA, Dışlayan VEYA ve DEĞİL işlemleri; bayrak denetiminden veri maskeleye, şifreleme algoritmalarından donanım yazmaçlarının yapılandırılmasına kadar pek çok alanda kullanılır. RISC-V'te kullanılan mantık buyrukları Tablo 3.15'da listelenmiştir.

Tablo 3.15: RISC-V'te mantık buyrukları

Buyruk	Söz Dizimi	Anlam	Tür
VE (and)	and rd, rs1, rs2	$rd = rs1 \& rs2$	R
Anlık VE (andi)	andi rd, rs1, imm	$rd = rs1 \& imm$	I
VEYA (or)	or rd, rs1, rs2	$rd = rs1 \mid rs2$	R
Anlık VEYA (ori)	ori rd, rs1, imm	$rd = rs1 \mid imm$	I
Dışlayan VEYA (xor)	xor rd, rs1, rs2	$rd = rs1 \wedge rs2$	R
Anlık DVEYA (xori)	xori rd, rs1, imm	$rd = rs1 \wedge imm$	I

Örnek 3.12

32 bitlik bir değerin yalnızca alt 8 bitini (en düşük bayt) koruyup geri kalan bitleri sıfırlamak istiyorsunuz. s1 yazmacındaki değere bu maskeleme işlemini uygulayın.

Çözüm:

```
1 andi s1, s1, 0xFF # alt 8 bit korunur, üst 24 bit sıfırlanır
```

$0xFF = (0000 \dots 01111111)_2$ ile VE işlemi yapıldığında yalnızca alt 8 bit aynen kalır; üst bitler sıfırla VE'lenerek temizlenir. Bu yöntem, donanım yazmaç değerlerinden belirli bir alanı çıkarmak için sıkça kullanılır.

3.9.1 DEĞİL İşlemi ve Mimariler Arası Farklılıklar

VE, VEYA ve Dışlayan VEYA iki işlenen gerektirir (*binary operation*); ancak DEĞİL (*NOT*) tek bir işlenen üzerinde çalışır: her 0 bitini 1'e, her 1 bitini 0'a çevirir. Bu basit işlem mimariler arasında şaşırtıcı derecede farklı biçimlerde gerçekleştirilir:

- x86'da NOT gerçek bir buyruktur: NOT EAX yazıldığında donanım doğrudan bit tersini hesaplar.
- ARM AArch64'te MVN (*Move Not*) buyruğu DEĞİL işlemini tek buyrukla gerçekleştirir: MVN X1, X2 buyruğu X2'nin tersini X1'e yazar.
- MIPS'te ayrı bir NOT buyruğu yoktur. Bunun yerine nor (*NOT OR*) buyruğu kullanılır: nor \$t0, \$t1, \$zero işlemi "\$t1 VEYA 0 = \$t1" sonucunun DEĞİL'ini alır. Tasarımcılar ayrı bir NOT buyruğu tanımlamak yerine NOR'un bu özelliğinden yararlanmayı tercih etmiştir.
- RISC-V'te ne not ne de nor gerçek buyruk olarak tanımlanmıştır. Bunun yerine xori buyruğu -1 anlık değeriyle kullanılarak DEĞİL elde edilir. Çevirici not rd, rs sözde buyruğunu otomatik olarak xori rd, rs, -1 biçimine açar.

RISC-V tasarımcılarının ayrı bir DEĞİL buyruğu eklememesinin nedeni, buyruk kümesini olabildiğince küçük tutma ilkesidir: mevcut bir buyrukla (xori) aynı işlem gerçekleştirilebildiğinde yeni bir işlem kodu tanımlamak gereksiz karmaşıklık yaratır.

Örnek 3.13

RISC-V buyruklarını kullanarak s2 yazmacındaki değerin DEĞİL'ini alın.

Çözüm:

```
1 xori s2, s2, -1 # s2'nin tüm bitlerini tersine çevirir
```

Bu buyruk not $s2$, $s2$ sözde buyruğuna karşılık gelir. Anlık değer -1 , 12 bite işaretli genişletildiğinde tüm 32 bitin 1 olduğu $(FFFFFFF)_{16}$ değerini üretir ve Dışlayan VEYA işlemi her biti tersine çevirir: $a \oplus 1 = \bar{a}$ olduğundan bu yöntem DEĞİL ile aynı sonucu verir.

3.10 Bit Düzeyinde Kaydırma Buyrukları

Bir sayının bitlerini sola ya da sağa kaydırmak, aslında 2'nin kuvvetleriyle çarpma veya bölme işlemine karşılık gelir. Örneğin bir değeri 3 bit sola kaydırmak $2^3 = 8$ ile çarpma ile aynı sonucu verir; 2 bit sağa kaydırmak ise $2^2 = 4$ 'e bölmekle eşdeğerdir. Çarpma buyruğu çoğu işlemcide birden fazla saat çevrimi gerektirdiğinden, derleyiciler 2'nin kuvvetiyle yapılan çarpma ve bölmeleri kaydırma buyrukları ile gerçekleştirir. Dizi erişimlerinde eleman boyutuna göre dizin hesaplamak (örn. 4 baytlık `int` dizisinde dizini 2 bit sola kaydırmak) bu yöntemin en sık karşılaşılan kullanımınıdır. RISC-V'teki kaydırma buyrukları Tablo 3.16'da özetlenmiştir.

Tablo 3.16: RISC-V'te bit düzeyinde kaydırma buyrukları

Buyruk	Söz Dizimi	Anlam	Tür
Sola Mantıksal (<code>sll</code>)	<code>sll rd, rs1, rs2</code>	$rd = rs1 \ll rs2$	R
Sağa Mantıksal (<code>srl</code>)	<code>srl rd, rs1, rs2</code>	$rd = rs1 \gg rs2$	R
Sağa Aritmetik (<code>sra</code>)	<code>sra rd, rs1, rs2</code>	$rd = rs1 \ggg rs2$	R
Anlık Sola Mant. (<code>slli</code>)	<code>slli rd, rs1, shamt</code>	$rd = rs1 \ll shamt$	I
Anlık Sağa Mant. (<code>srli</code>)	<code>srli rd, rs1, shamt</code>	$rd = rs1 \gg shamt$	I
Anlık Sağa Arit. (<code>srai</code>)	<code>srai rd, rs1, shamt</code>	$rd = rs1 \ggg shamt$	I

Sola kaydırmalarda sağdan 0 biti girer. Sağa mantıksal kaydırmada soldan 0 girer. Sağa aritmetik kaydırmada ise kaydırılacak yazmacın en soldaki (işaret) biti tekrar edilir.

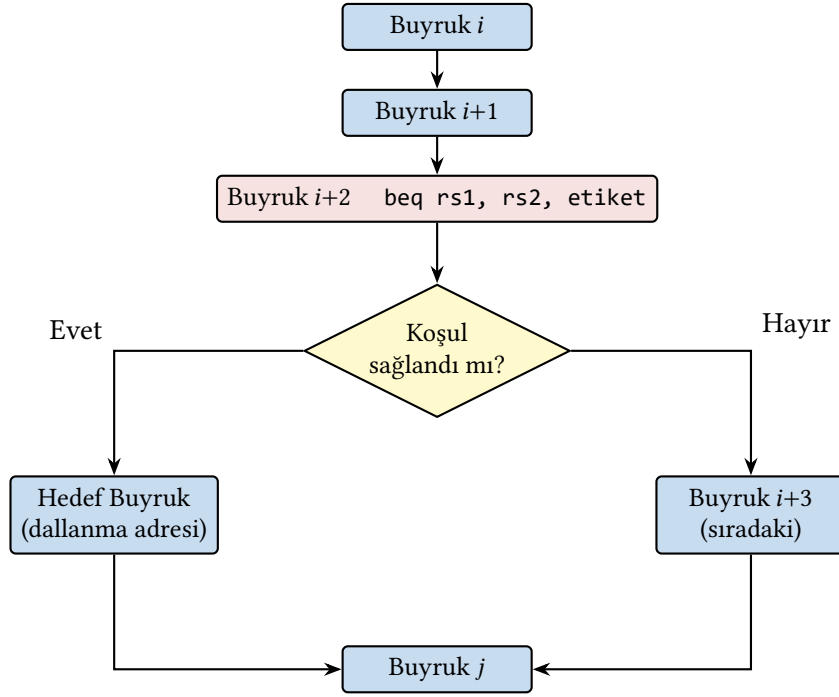
RISC-V'te R türü kaydırma buyrukları (`sll`, `srl`, `sra`) kaydırma miktarını `rs2` yazmacından okurken, I türü buyruklar (`slli`, `srli`, `srai`) kaydırma miktarını anlık değer olarak alır.

Kaydırma İşlemleri

x86 kaydırma buyrukları, değişken miktarlı kaydırmalar için geleneksel olarak `CL` yazmacını kullanır; sabit miktarlı kaydırmalar ise anlık değerle gerçekleştirilir. ARM'ın toplama-çıkarma birimine tümleşik silindiri (barrel shifter), kaydırma işlemini başka bir buyruğun parçası olarak çalıştırabilir (örn. `ADD R0, R1, R2, LSL #3`). MIPS, RISC-V'e benzer biçimde `sll/srl/sra` buyrukları sunar. RISC-V ise kaydırma işlemlerini ayrı buyruklar olarak tutar ve düzenli bir kodlama yapısı sağlar.

3.11 Dallanma ve Atlama Buyrukları

Bir program her zaman buyrukları sırayla yürütmez. Bazen bir koşula bağlı olarak farklı bir yere atlamak, bazen de bir döngüyü tekrarlamak gerekir. İşte dallanma ve atlama buyrukları, programın akışını değiştirerek `if-else`, `for` ve `while` gibi yapıları mümkün kılar (Şekil 3.13).



Şekil 3.13: Programda Dallanma

3.11.1 Koşullu Dallanma Buyrukları

Tablo 3.17: RISC-V’te koşullu dallanma buyrukları

Buyruk	Söz Dizimi	Anlam	Tür
Eşitse Dallan (beq)	beq rs1, rs2, etiket	Eğer $rs1 == rs2$: dallan	B
Eşit Değilse (bne)	bne rs1, rs2, etiket	Eğer $rs1 \neq rs2$: dallan	B
Küçükse Dallan (blt)	blt rs1, rs2, etiket	Eğer $rs1 < rs2$: dallan	B
Büyük/Eşitse (bge)	bge rs1, rs2, etiket	Eğer $rs1 \geq rs2$: dallan	B
İşaretsiz Küçükse (bltu)	bltu rs1, rs2, etiket	İşaretsiz $rs1 < rs2$: dallan	B
İşaretsiz Büyük/Eşitse (bgeu)	bgeu rs1, rs2, etiket	İşaretsiz $rs1 \geq rs2$: dallan	B

Tablo 3.17’de listelenen RISC-V koşullu dallanma buyruklarının MIPS’e göre önemli bir üstünlüğü, doğrudan karşılaştırmalı dallanma buyrukları olan blt ve bge buyruklarına sahip olmasıdır. MIPS’te “küçükse dallan” işlemi için önce slt ile karşılaştırma yapıp ardından bne ile dallanma gerekirken, RISC-V’te bu işlem tek bir buyrukla gerçekleştirilebilir.

Tablo 3.18: RISC-V’te karşılaştırma buyrukları

Buyruk	Söz Dizimi	Anlam	Tür
Küçükse Birle (slt)	slt rd, rs1, rs2	$rd=1$ eğer $rs1 < rs2$	R
Anlık Küçükse (slti)	slti rd, rs1, imm	$rd=1$ eğer $rs1 < imm$	I
İşaretsiz Küçükse (sltu)	sltu rd, rs1, rs2	$rd=1$ eğer $rs1 < rs2$ (işaretsiz)	R
Anlık İşaretsiz K. (sltiu)	sltiu rd, rs1, imm	$rd=1$ eğer $rs1 < imm$ (işaretsiz)	I

Tablo 3.18’deki karşılaştırma buyrukları, koşullu dallanmalarda yardımcı olarak kullanılır.

Örnek 3.14

RISC-V’te $s1$ ve $s2$ yazmaçlarının değerini karşılaştıran, $s1 < s2$ koşulu doğru ise “uzak” etiketine atlayan bir program parçası yazın.

Çözüm:

Yöntem 1: RISC-V’in doğrudan karşılaştırmalı dallanma buyruğu ile (tercih edilen yöntem):

```
1  blt s1, s2, uzak    # s1 < s2 ise uzak'a atla
```

Yöntem 2: slt+bne yaklaşımı ile:

```
1  slt t0, s1, s2      # s1 < s2 ise t0 = 1
2  bne t0, zero, uzak  # koşul doğruysa uzak'a atla
```

Görüldüğü gibi RISC-V’in blt buyruğu sayesinde tek bir buyrukla aynı işlem gerçekleştirilebilmektedir.

Örnek 3.15

Aşağıdaki “if ... then ... else” deyimini RISC-V buyruklarıyla yazın.

```
1  if (a == b) c = a + b;
2  else c = a - b;
```

a, b, c sırasıyla $s1, s2, s3$ yazmaçlarında tutulsun.

Çözüm:

```
1      bne s1, s2, esit_degil  # a != b ise esit_degil'e git
2      add s3, s1, s2          # c = a + b
3      jal zero, son           # son'a atla (j son)
4  esit_degil:
5      sub s3, s1, s2          # c = a - b
6  son:
```

RISC-V’te ayrı bir j (koşulsuz atlama) buyruğu yoktur. Bunun yerine jal buyruğu hedef yazmaç olarak zero (x0) ile kullanıldığında, dönüş adresi atılır ve koşulsuz atlama elde edilir. Çevirici dilde bu işlem j son sözde buyruğu olarak yazılabilir.

Örnek 3.16

Aşağıdaki C döngüsünü RISC-V buyruklarıyla gerçekleştirin:

```
1  int toplam = 0;
2  for (int i = 0; i < N; i++)
3      toplam += dizi[i];
```

dizi dizisinin başlangıç adresi $s1$, N değeri $s2$, toplam değişkeni $s3$ yazmacındadır. Sayaç olarak $s4$ kullanılsın.

Çözüm:

```
1      addi s3, zero, 0        # toplam = 0
2      addi s4, zero, 0        # i = 0
3  don:
```

```

4   bge s4, s2, cik      # i >= N ise döngüden çık
5   slli t0, s4, 2       # t0 = i * 4 (her int 4 bayt)
6   add t0, s1, t0       # t0 = dizi[i] adresi
7   lw  t1, 0(t0)        # t1 = dizi[i]
8   add s3, s3, t1       # toplam += dizi[i]
9   addi s4, s4, 1       # i++
10  jal zero, don        # döngünün başına dön
11  cik:

```

Bu örnek dallanma buyruklarının en yaygın kullanımını göstermektedir: bge döngü koşulunu sınar ve koşul sağlandığında döngüden çıkar. jal zero, don koşulsuz geri atlama için kullanılır; hedef yazmaç olarak zero verildiğinde dönüş adresi saklanmaz, bu da jal don sözde buyruğuna karşılık gelir. Dizi indisinin 4 ile çarpılması (slli ile 2 bit sola kaydırma) her int değerın 4 bayt kaplamasından kaynaklanır.

3.11.2 Koşulsuz Atlama Buyrukları

Tablo 3.19: RISC-V’te koşulsuz atlama buyrukları

Buyruk	Söz Dizimi	Anlam	Tür
Atla ve Bağla (jal)	jal rd, etiket	rd=PS+4; etikete atla	J
Yazmaca Atla ve B. (jalr)	jalr rd, rs1, imm	rd=PS+4; rs1+imm’e atla	I

Tablo 3.19’da gösterilen RISC-V atlama buyrukları ve sözde buyruk karşılıkları:

- jal ra, etiket: Altyordam çağrısı. Dönüş adresini (PS+4) ra yazmacına yazar ve etikete atlar. Sözde buyruk: jal etiket.
- jal zero, etiket: Koşulsuz atlama. Dönüş adresi kaydedilmez. Sözde buyruk: j etiket.
- jalr zero, ra, 0: Altyordamdan dönüş. ra’daki adrese atlar. Sözde buyruk: ret.
- jalr ra, rs1, 0: Yazmaçtaki adrese dolaylı altyordam çağrısı.

Bilgi Notu 3.12: RISC-V’te Geciktirilmiş Dallanma Yoktur

RISC-V’te MIPS’teki geciktirilmiş dallanma (*delayed branch*) kavramı yoktur. MIPS’te dallanma buyruğundan hemen sonraki buyruk (dallanma gecikme yuvası) her durumda yürütülürken, RISC-V’te dallanma gerçekleştiğinde bir sonraki buyruk yürütülmez. Bu, RISC-V programlarının anlaşılmasını ve yazılmasını daha kolay hale getirir. Geciktirilmiş dallanmanın neden ortaya çıktığı ve boru hattı ile ilişkisi ileriki bölümlerde ayrıntılı olarak ele alınacaktır.

Dallanma Mekanizmaları

Dallanma mekanizması, mimariler arasındaki temel farklılıklardan biridir:

- x86, bayrak yazmacı (FLAGS) kullanır: ZF, CF, SF, OF gibi bayraklar karşılaştırma buyruğuyla (CMP) güncellenir, ardından koşullu atlama (JE, JNE, JL, ...) bu bayrakları denetler. İki ayrı buyruk gerekir.
- ARM, geleneksel olarak x86’ya benzer koşul kodları (NZCV) kullanır; AArch64 ise bun-

ların yanı sıra doğrudan karşılaştırmalı atlama buyrukları (CBZ, CBNZ) sunar.

- MIPS'in dallanma desteği sınırlıdır: yalnızca BEQ ve BNE bulunur; küçüktür karşılaştırması için SLT ve BNE birlikte kullanılmak zorundadır. Bunun yanı sıra dallanma gecikme yuvası (branch delay slot) nedeniyle derleyicinin bu yuvayı dolduracak buyruk üretmesi gerekir.
- RISC-V, doğrudan karşılaştırmalı atlama buyrukları sunar (BLT, BGE, BEQ, BNE, BLTU, BGEU): bayrak yazmacı gerekmez, gecikme yuvası yoktur. Bu sayede en sade ve en tutarlı dallanma yapısını elde eder.

Örnek 3.17

Aşağıdaki C kodunu dört mimaride gerçekleştirin ve buyruk sayılarını karşılaştırın:

```
1  if (a < b)
2      c = a + 1;
3  else
4      c = b + 1;
```

RISC-V (a=s1, b=s2, c=s3):

```
1  bge s1, s2, else_   # a >= b ise else'e git (1 buyruk)
2  addi s3, s1, 1      # c = a + 1
3  j    son_           # son'a atla
4  else_:
5  addi s3, s2, 1      # c = b + 1
6  son_:
```

Koşullu yolda 2, koşulsuz yolda 3 buyruk yürütülür.

MIPS (a=\$s1, b=\$s2, c=\$s3):

```
1  slt $t0, $s1, $s2   # t0 = (a < b) ? 1 : 0
2  beq $t0, $zero, else # t0==0 ise dal
3  nop                 # gecikme yuvası
4  addi $s3, $s1, 1     # c = a + 1
5  j    son
6  nop                 # gecikme yuvası
7  else:
8  addi $s3, $s2, 1     # c = b + 1
9  son:
```

MIPS iki fazladan buyruk gerektirir: slt karşılaştırma için (doğrudan blt olmadığından) ve nop'lar gecikme yuvaları için. Koşullu yolda 4, koşulsuz yolda 5 buyruk yürütülür.

x86:

```
1  CMP EAX, EBX        ; bayraklari guncelle (a - b)
2  JGE else_           ; a >= b ise else'e git
3  ADD EAX, 1          ; c = a + 1 (sonuc EAX'ta)
4  JMP son_
5  else_:
6  MOV ECX, EBX
7  ADD ECX, 1          ; c = b + 1
8  son_:
```

x86’da karşılaştırma (CMP) ve dallanma (JGE) iki ayrı buyruk gerektirir; bayrak yazmacı aracılığıyla iletişim kurulur. Ancak x86 buyruklarının bir kısmı daha güçlüdür: ADD EAX, 1 hem toplama hem yazmacı yazma işini tek buyrukta yapar.

ARM AArch64 (a=X1, b=X2, c=X3):

```

1    CMP    X1, X2          ; bayrakları guncelle
2    BGE    else_          ; a >= b ise else'e git
3    ADD    X3, X1, #1      ; c = a + 1
4    B      son_
5  else_:
6    ADD    X3, X2, #1      ; c = b + 1
7  son_:

```

ARM de x86 gibi bayrak yazmacı kullanır; ancak AArch64 ayrıca CSEL (koşullu seçim) buyruğuyla dallanmayı tamamen ortadan kaldırabilir: CMP X1, X2 ve ardından CSINC X3, X2, X1, GE ile tüm iş 2 buyruğa düşer.

3.12 Sözde Buyruklar

Çevirici dilde sık kullanılan bazı işlemler için RISC-V donanımında ayrı bir buyruk tanımlanmamıştır. Bunun yerine çevirici (*assembler*), programcının yazdığı kısa biçimli buyruğu bir veya iki gerçek buyruğa otomatik olarak açar. Bu kısa biçimlere *sözde buyruk* (*pseudo-instruction*) denir.

Sözde buyrukların arkasındaki tasarım fikri, işlem kodu uzayını verimli kullanmak ile programlamayı kolaylaştırmak arasındaki dengeyi kurmaktır. Her sık kullanılan işlem için ayrı bir işlem kodu tanımlamak, sınırlı buyruk alanını hızla tüketir ve kod çözme donanımını karmaşıktırır. Sözde buyruklar bu sorunu çözer: donanım az sayıda temel buyrukla yetinirken, çevirici programcıya zengin bir buyruk kümesi varmış gibi bir deneyim sunar. Örneğin RISC-V donanımında ayrı bir `mv` buyruğu yoktur; ancak programcı `mv s1, s2` yazdığında çevirici bunu `addi s1, s2, 0` buyruğuna dönüştürür. Sonuç olarak donanım basit kalır, işlem kodu uzayı korunur ve programcı bunun farkında bile olmadan rahatça kod yazar. Tablo 3.20’de en yaygın kullanılan sözde buyruklar listelenmiştir (tam liste için bkz. Ek C).

Tablo 3.20: RISC-V'te yaygın sözde buyruklar ve gerçek buyruk karşılıkları

Sözde buyruk	Gerçek buyruk(lar)	Açıklama
nop	addi zero, zero, 0	İşlem yapma (boş buyruk)
mv rd, rs	addi rd, rs, 0	Yazmaçtan yazmaca kopyala
not rd, rs	xori rd, rs, -1	Bit düzeyinde DEĞİL
neg rd, rs	sub rd, zero, rs	İşaret değiştir (ikiye tümle)
li rd, imm	lui+addi	Anlık değer yükle
la rd, sembol	auipc+addi	Adres yükle
j etiket	jal zero, etiket	Koşulsuz atla
jal etiket	jal ra, etiket	Altyordam çağır
ret	jalr zero, ra, 0	Altyordamdan dön
call sembol	auipc+jalr	Uzak altyordam çağır
tail sembol	auipc+jalr	Kuyruk çağırısı (dönüş adresi saklanmaz)
ble rs, rt, etiket	bge rt, rs, etiket	Küçük veya eşitse dallan
bgt rs, rt, etiket	blt rt, rs, etiket	Büyükse dallan

Tablodaki örneklerden görüldüğü gibi, sözde buyrukların büyük bölümü mevcut buyrukların özel durumlarından ibarettir: sıfır yazmacı (x0) veya sıfır anlık değeri kullanılarak farklı işlemler elde edilir. Bu yaklaşım, RISC-V'in yalnızca yaklaşık 47 gerçek buyrukla geniş bir işlem yelpazesini nasıl kapsadığını somut biçimde göstermektedir.

3.13 Adresleme Kipleri

Bir buyruk yürütülürken, işlenecek veriler nereden gelir? Doğrudan buyruğun içinden mi, bir yazmaçtan mı, yoksa bellekten mi okunur? Sonuç nereye yazılır? Bu soruların yanıtı **adresleme kipi** kavramıyla verilir. Her adresleme kipi, verinin kaynağını ya da hedefini belirleyen farklı bir yöntemdir (RISC-V adresleme kiplerinin özeti için bkz. Ek C).

Peki bu kiplere neden gerek vardır? Üst düzey bir dilde yazdığımız her yapı, işlemcinin anlayacağı buyruk dizisine çevrilirken veriye farklı yollarla erişmeyi gerektirir. Tablo 3.21 bu bağlantıyı gösterir: her adresleme kipi, aslında sıkça karşılaşılan bir programlama örüntüsünün donanım düzeyindeki karşılığıdır.

Tablo 3.21: Adresleme kipleri ve üst düzey dil karşılıkları

Adresleme Kipi	Üst Düzey Yapı	C Örneği
Yazmaç	Yerel değişkenler arası işlem	c = a + b;
Anlık	Sabit değerle işlem	x = x + 5;
	Koşulda sabit karşılaştırma	if (x > 10) {...}
Eklemeli (taban+öteleme)	Yapı alanına erişim	s.alan ya da s->alan
	Sabit dizinli dizi erişimi	dizi[3]
	Yığıttaki yerel değişken	int a, b, c;
PS'ye göreceli	Dallanma, döngü, koşul	if/else, while, for
	Alt yordam çağırısı	fonk();
Üst anlık	Büyük sabit yükleme	int x = 0x12345678;
	Genel (global) değişken adresi	extern int g;

Her adresleme kipinin arkasında, üst düzey dildeki bir yapıyı makine düzeyine taşıyan belirli bir süreç vardır:

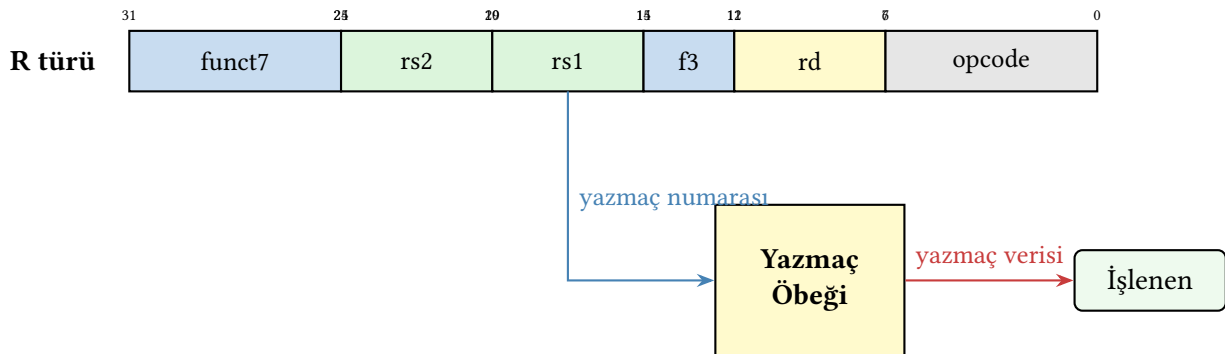
- **Yazmaç:** $c = a + b$; yazıldığında derleyici a , b ve c değişkenlerini yazmaçlara atar. Buyruk yürütülürken veri doğrudan yazmaç öbeğinden okunur ve sonuç yine bir yazmaca yazılır; belleğe hiç gidilmez. En hızlı erişim yoludur.
- **Anlık:** $x = x + 5$; ifadesinde 5 sabiti derleme sırasında bilinen bir değerdir. Derleyici bu sabiti buyruğun içine 12 bitlik bir alan olarak gömer. Yürütme sırasında işlemci sabiti buyruğun kendisinden okur; ne yazmaç öbeğine ne belleğe erişim gerekir.
- **Eklemeli (taban + öteleme):** $s.alan$ yazıldığında derleyici yapının başlangıç adresini bir yazmaca (*taban*) koyar, alanın yapı içindeki konumunu ise sabit bir öteleme olarak buyruğa gömer. Yürütme sırasında taban ve öteleme toplanarak bellek adresi elde edilir. Aynı düzenek yığıttaki yerel değişkenler ($sp + \text{öteleme}$) ve sabit dizinli dizi erişimi ($dizi[3] \rightarrow \text{taban} + 3 \times \text{eleman boyutu}$) için de kullanılır. Ancak dizin değişken olduğunda ($dizi[i]$) öteleme derleme sırasında bilinemez; bu durumda adres hesabı için ek buyruklara gerek duyulur (aşağıdaki karşılaştırma kutusuna bakınız).
- **PS'ye göreceli:** $\text{if } (x > 0) \{A\} \text{ else } \{B\}$ yazıldığında derleyici bir koşullu dallanma buyruğu üretir. Hedef adres, o anki program sayacına (PS) göre bir eklenti olarak kodlanır. Yürütme sırasında koşul sağlanırsa $PS + \text{eklenti}$ hesaplanarak dallanılır, sağlanmazsa bir sonraki buyruğa devam edilir. Bu kip sayesinde kod bellekte nereye yüklenirse yüklensin dallanma doğru çalışır.
- **Üst anlık:** $\text{int } x = 0x12345678$; yazıldığında bu sabit 12 bitlik anlık alana sığmaz. Derleyici lui ile üst 20 bit yazmaca yazar, ardından $addi$ ile alt 12 bit ekler; iki buyrukla tam 32 bitlik sabit oluşturulur. Genel değişken adreslerine erişimde de aynı düzenek kullanılır.

Yalnızca yazmaç adresleme kipiyle bir programlama dili derlenebilseydi, yapı alanlarına erişim, döngü denetimi ve sabit yükleme gibi temel işlemler için çok sayıda fazladan buyruk gerekecekti. Her adresleme kipi, derleyicinin ürettiği buyruk sayısını azaltan ve programı hızlandıran bir kestirme yoldur.

RISC-V'te kullanılan adresleme kipleri aşağıda açıklanmıştır. Farklı mimarilerin hangi kipleri desteklediği, bölüm sonundaki Tablo 3.22'de karşılaştırmalı olarak verilmiştir.

3.13.1 Yazmaç Adresleme Kipi

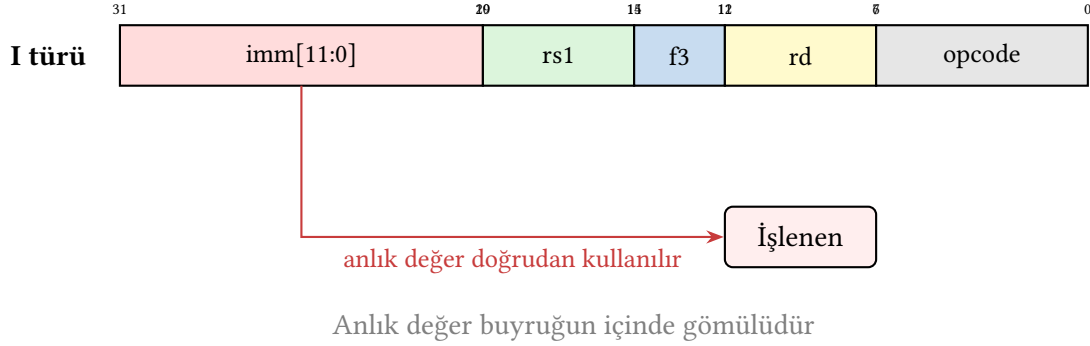
Kaynak verisi bir yazmacın içinde bulunur. RISC-V'teki R türü buyrukların tamamı bu kipe örnektir (Şekil 3.14).



Şekil 3.14: RISC-V'te Yazmaç Adresleme Kipi

3.13.2 Anlık Adresleme Kipi

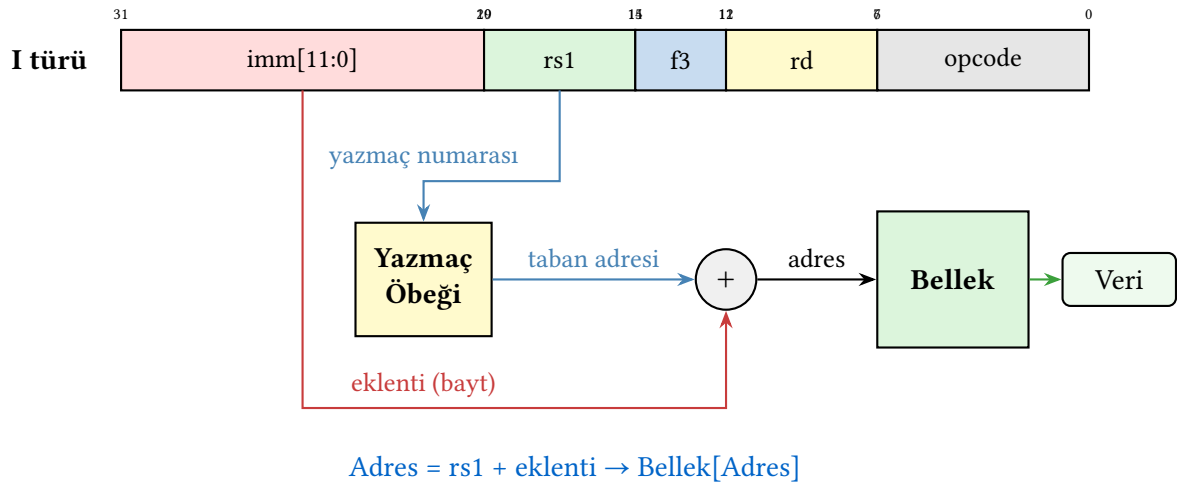
Kaynak verisi buyruğun içinde konumlandırılmış bir sabit değerdir. RISC-V'teki I türü buyruklardaki anlık değerler (12 bit, işaretli genişletilmiş) bu kipe örnektir (Şekil 3.15).



Şekil 3.15: RISC-V'te Anlık Adresleme Kipi

3.13.3 Eklemeli Adresleme Kipi (Taban Adresleme)

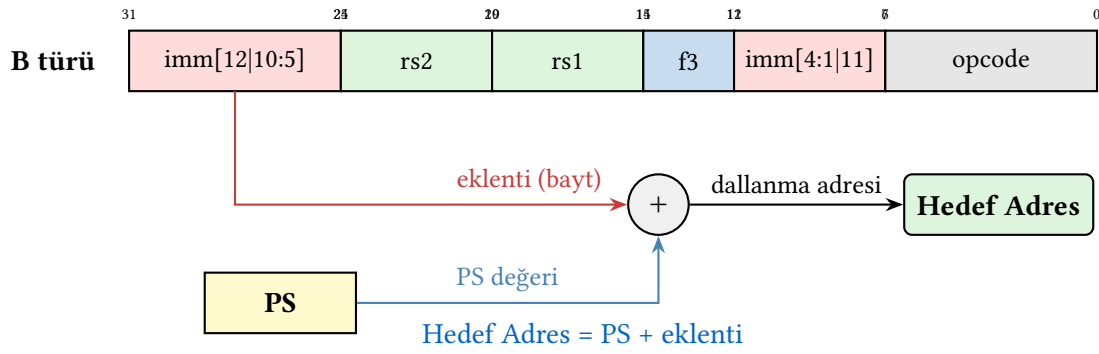
Taban adresine anlık değer eklenerek bellekte erişilecek yer hesaplanır. RISC-V'te yükleme (lw, lb vb.) ve saklama (sw, sb vb.) buyrukları bu kipi kullanır (Şekil 3.16).



Şekil 3.16: RISC-V'te Eklemeli Adresleme Kipi

3.13.4 PS'ye Göreceli Adresleme Kipi

Program sayacının üzerine ekleme yapma yoluyla yeni bellek adresinin bulunması yöntemidir (Şekil 3.17). RISC-V'te B türü dallanma buyrukları ve J türü atlama buyrukları (jal) bu kipi kullanır. Ayrıca auipc buyruğu da PS'ye göreceli adresleme sağlar.

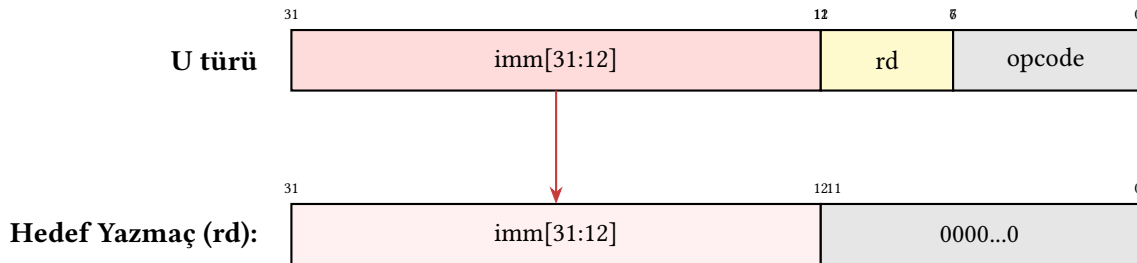


Şekil 3.17: RISC-V'te PS'ye Göreceli Adresleme Kipi

3.13.5 Üst Anlık Adresleme Kipi

RISC-V'te I türü buyrukların anlık alanı yalnızca 12 bittir; bu alan -2048 ile $+2047$ arasındaki değerleri temsil edebilir. Peki 12 bite sığmayan büyük sabitleri bir yazmaca nasıl yükleriz? RISC-V bu sorunu lui (Load Upper Immediate) buyruğuyla çözer. Ancak dikkat edilmesi gereken nokta şudur: lui yazmacın tamamını değil, yalnızca üst 20 bitini (bit 31–12) yazar; alt 12 bit ise sıfırlanır. Dolayısıyla tek başına lui istenen sabiti oluşturamaz; yalnızca üst kısmını hazırlar. Tam 32 bitlik bir sabiti elde etmek için ardından addi ile alt 12 bit eklenir (Şekil 3.18). Bu iki buyrukluk sıralama, RISC-V'in sabit uzunluklu buyruk tasarımının doğal bir sonucudur: 32 bitlik buyrukta hem işlem kodu hem yazmaç numarası hem de 32 bitlik bir sabit aynı anda sığamaz.

Benzer biçimde auipc buyruğu, 20 bitlik anlık değeri program sayacına (PS) ekleyerek 32 bitlik bir adres üretir. Bu, konumdan bağımsız kodda uzak adreslere erişmek için kullanılır.



1. lui rd, üst_20 $\Rightarrow$ rd = üst_20 $\ll$ 12
2. addi rd, rd, alt_12 $\Rightarrow$ rd = tam 32 bit sabit

Şekil 3.18: RISC-V'te üst anlık adresleme kipi: lui buyruğu yalnızca üst 20 biti yazar, alt 12 bit sıfırlanır; tam sabit için addi ile tamamlanır

32 Bitlik Sabitlerin Yüklenmesi: Mimari Karşılaştırması

Bir yazmaca $0x12345678$ gibi 32 bitlik bir sabit yüklemek, buyruk kümesi tasarımının buyruk sayısına etkisini doğrudan gösterir:

Mimari	Kod	Buyruk Sayısı
x86	mov eax, 0x12345678 <i>32 bit sabit doğrudan buyruğa gömülür (değişken uzunluk)</i>	1
ARM	movz x0, #0x5678 movk x0, #0x1234, lsl #16 <i>movz alt 16 biti yazar ve sıfırlar; movk üst 16 biti değiştirir, alt kısmı korur</i>	2
MIPS	lui \$t0, 0x1234 ori \$t0, \$t0, 0x5678 <i>lui üst 16 bite yazar; ori alt 16 biti VEYA ile birleştirir</i>	2
RISC-V	lui t0, 0x12345 addi t0, t0, 0x678 <i>lui üst 20 bite yazar; addi alt 12 biti ekler</i>	2

x86 değişken uzunluklu buyruk biçimi sayesinde 32 bitlik sabiti tek buyruğa gömebilir; ancak bu buyruk 5–6 bayt yer kaplar. RISC mimarileri sabit uzunluklu (32 bit) buyruk biçiminde bu kadar büyük bir sabite yer olmadığından iki buyruğa gerek duyar. MIPS ve RISC-V üst/alt yarıya bölerken, ARM movz/movk çiftiyle 16 bitlik dilimler hâlinde yazar. Dikkat edilirse RISC-V 20+12, MIPS ise 16+16 biçiminde böler; RISC-V'in 20 bitlik üst alanı, lui tek başına daha geniş bir aralığı kapsayabildiğinden bazı sabitlerin addi'ye gerek kalmadan yüklenmesine olanak tanır.

Adresleme Kiplerinin Zenginliği

RISC mimarileri adresleme kiplerini bilinçli olarak sınırlı tutar. RISC-V yalnızca beş kip sunar (yazmaç, anlık, eklemeli, PS'ye göreceli, üst anlık); MIPS de benzer biçimde üç temel kiple yetinir. ARM (AArch64) biraz daha zengin bir seçenek sunar: ön ve son dizinli adresleme, genişletilmiş yazmaç adresleme gibi altı civarında kip tanımlar.

x86 ise tam tersi yönde ilerler: taban, dizin, ölçek ve öteleme bileşenlerini serbestçe birleştirerek 10'dan fazla adresleme kipi üretir (Tablo 3.22). Bu zenginlik, tek bir buyrukla karmaşık bellek erişimi yapabilmeyi sağlar; ancak buyruk çözme donanımını önemli ölçüde karmaşıklarlaştırır. Daha az kip, daha basit bir çözümleyici (decoder) donanımı anlamına gelir; ancak adres hesaplamak için zaman zaman ek buyruklara gerek duyulabilir.

Bu ödünleşmeyi somut olarak görmek için C dilindeki `int dizi[N];` dizisinde `dizi[i]` elemanına erişimi düşünelim. Burada `i` bir değişkendir, derleme sırasında değeri bilinmez:

Mimari	Kod	Buyruk
x86	mov eax, [ebx + ecx*4] <i>Ölçekli dizinli kip: taban + dizin × ölçek tek buyrukta</i>	1
ARM	ldr w0, [x1, x2, lsl #2] <i>Genişletilmiş yazmaç kipi: taban + dizin << 2 tek buyrukta</i>	1
RISC-V	slli t2, t1, 2 # t2 = i * 4 add t2, t0, t2 # t2 = taban + i*4 lw t3, 0(t2) # t3 = dizi[i] <i>Yalnızca taban+öteleme kipi var; dizin hesabı ayrı yapılır</i>	3

x86 ve ARM, zengin adresleme kipleri sayesinde bu işlemi tek buyrukla yapar. RISC-V ise bilinçli olarak basit tuttuğu adresleme kipiyle aynı işi üç buyruğa yayar; ancak her buyruk basit

olduğundan saat çevrimi başına daha yüksek hızda çalıştırılabilir. Bu, RISC ve CISC arasındaki temel ödünleşmenin en somut örneklerinden biridir.

Tablo 3.22: İşlemcilerde kullanılan adresleme kipleri

Adresleme Kipi	Genel Gösterim	x86 Örneği	Anlam	RISC-V	ARM	x86
Yazmaç	Rd, Rs	mov eax, ebx	$Rd \leftarrow Rs$	✓	✓	✓
Anlık	Rd, #sabit	mov eax, 5	$Rd \leftarrow \text{sabit}$	✓	✓	✓
Doğrudan	Rd, (adres)	mov eax, [0x1000]	$Rd \leftarrow \text{Bel}[\text{adres}]$			✓
Yazmaçla dolaylı	Rd, (Rs)	mov eax, [ebx]	$Rd \leftarrow \text{Bel}[Rs]$		✓	✓
Taban + eklenti	Rd, ekl(Rs)	mov eax, [ebx+8]	$Rd \leftarrow \text{Bel}[Rs+ekl]$	✓	✓	✓
Dizinli	Rd, (Rs1+Rs2)	mov eax, [ebx+ecx]	$Rd \leftarrow \text{Bel}[Rs1+Rs2]$		✓	✓
Ölçekli dizinli	Rd, (Rs×ö)	mov eax, [ecx*4]	$Rd \leftarrow \text{Bel}[Rs \times \text{ö}]$			✓
Taban + ölç. dizin	Rd, (Rs1+Rs2×ö)	mov eax, [ebx+ecx*4]	$Rd \leftarrow \text{Bel}[Rs1+Rs2 \times \text{ö}]$			✓
Tbn + ölç. dzn + ekl	Rd, ekl(Rs1+Rs2×ö)	mov eax, [ebx+ecx*4+8]	$Rd \leftarrow \text{Bel}[Rs1+Rs2 \times \text{ö}+ekl]$			✓
PS'ye göreceli	Rd, PS+ekl	mov eax, [rip+ofs]	$Rd \leftarrow \text{Bel}[PS+ekl]$	✓		✓
Üst anlık	Rd, #sabit<<12	—	$Rd \leftarrow \text{sabit} \ll 12$	✓		
Ön/son dizinli	Rd, [Rs, ekl]!	—	$Rd \leftarrow \text{Bel}[Rs+ekl]; Rs \uparrow$		✓	

ö: ölçek çarpanı (1, 2, 4 veya 8); ekl: eklenti (bayt olarak uzaklık); PS: program sayacı; Rs↑: yazmaç otomatik güncellenir. x86 SIB (Scale-Index-Base) baytı sayesinde taban, dizin, ölçek ve eklentiye tek buyrukta birleştirebilir. Bellekle dolaylı adresleme (Bel[Bel[Rs]]) çağdaş mimarilerin hiçbirinde desteklenmez.

Örnek 3.20

Aşağıdaki RISC-V buyrukları yürütülmeden önce yazmaçlar ve bellek şu değerleri taşımaktadır:

Yazmaç	Değer	Bellek adresi	İçerik
t0 (x5)	0x1000_0000	0x1000_0000	42
t1 (x6)	0x0000_0064	0x1000_0008	99
t2 (x7)	0x0000_0000	0x1000_0010	17
PS (program sayacı)	0x0000_1000		

Her buyruk için kullanılan adresleme kipini belirleyin ve hedef yazmacın son değerini hesaplayın.

#	Buyruk	Adresleme Kipi	Sonuç
1	addi t2, t1, 36	?	?
2	lw t2, 8(t0)	?	?
3	add t2, t0, t1	?	?
4	lui t2, 0x12345	?	?
5	auipc t2, 0x00002	?	?

Çözüm:

#	Buyruk	Adresleme Kipi	Sonuç
1	addi t2, t1, 36	Anlık: t1 + 36	100 + 36 = 136
2	lw t2, 8(t0)	Taban + eklenti: Bel[t0 + 8]	Bel[0x1000_0008] = 99
3	add t2, t0, t1	Yazmaç: t0 + t1	0x1000_0064
4	lui t2, 0x12345	Üst anlık: sabit << 12	0x1234_5000
5	auipc t2, 0x00002	PS'ye göreceli: PS + sabit << 12	0x0000_1000 + 0x0000_2000 = 0x0000_3000

Dikkat: 2. buyruk bellekten okuma yaptığından hem adres hesaplaması hem de bellek erişimi gerçekleşir. 4. ve 5. buyruklar 20 bitlik sabiti 12 bit sola kaydırır; lui mutlak değer üretirken auipc PS'ye göreceli adres üretir.

3.14 Altyordam ve Yığıtlar

Bir programda aynı işlemi farklı yerlerde tekrar tekrar yapmak gerektiğinde, o kodu her seferinde yeniden yazmak hem verimsiz hem de hata yapma olasılığını artırır. Bunun yerine, sık kullanılan işlemler **altyordamlar** (*subroutine*) olarak bir kez yazılır ve gerektiğinde çağrılır; tıpkı bir yemek tarifinde “beyaz sos için 12. sayfadaki tarife bakın” demek gibi.

Bir altyordamın çalıştırılması sırasında izlenen adımlar:

1. Ana program, altyordamın kullanacağı değişkenleri erişilebilir bir yere yazar.
2. Denetim ana programdan altyordama aktarılır.
3. Altyordam verileri okur ve işlemi gerçekleştirir.
4. Sonuç verileri ana programın erişebileceği bir alana yazılır.
5. Denetim altyordamın çağrıldığı noktaya geri aktarılır.

3.14.1 RISC-V'te Altyordamların Kullanılması

RISC-V'te altyordamlar için kullanılan özel yazmaçlar:

- ra (x1): Dönüş adresini tutan yazmaç.
- a0–a7 (x10–x17): Altyordama geçirilecek terimleri (argümanları) tutan 8 adet yazmaç.
- a0–a1 (x10–x11): Altyordamdan ana programa iletilecek sonuç verilerini de tutan yazmaçlar.

RISC-V'te jal ra, etiket buyruğu atlamadan önce PS+4 değerini ra yazmacına yazar. Altyordam geri dönmek için jalr zero, ra, 0 buyruğunu (sözde buyruk: ret) kullanır.

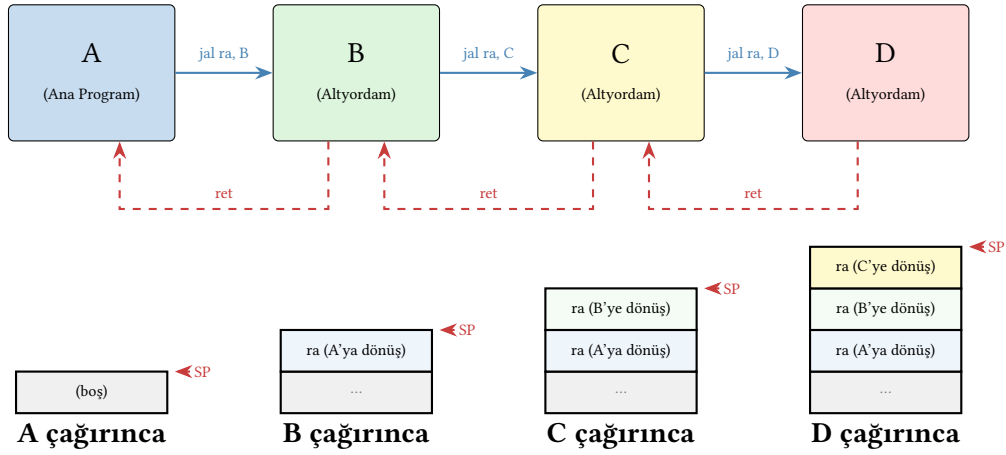
```

1 # Altyordam çağırma
2 jal ra, altyordam    # ra = PS+4, altyordam'a atla
3
4 # Altyordamdan dönüş
5 jalr zero, ra, 0     # ra'daki adrese dön (ret)
```

3.14.2 İç İç Geçmiş Çağrılar

Her altyordamın içinden başka bir altyordam çağrılabilir. Dönüş adreslerinin kaybedilmemesi için **yığıt** kullanılır (Şekil 3.19). Yığıt işaretçisi (sp) yığıtın en tepe noktasını gösterir. Yığıta

veri yazılışında *sp* azaltılır, veri çekildiğinde artırılır.



Şekil 3.19: İç içe altyordam çağrılarında yığıt durumu. $A \rightarrow B \rightarrow C \rightarrow D$ çağrı zincirinde her çağrı yığıta yeni bir çerçeve ekler.

3.14.3 Çağırıcı ve Çağrılan Tarafın Sorumlulukları

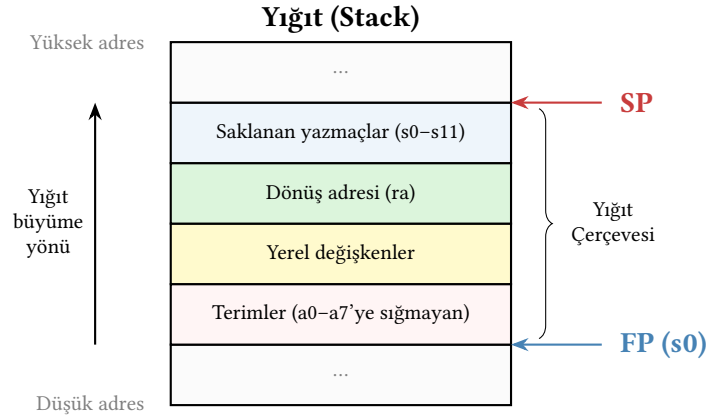
Bir altyordam çağrıldığında hem çağırıcı (*caller*) hem de çağrılan (*callee*) tarafın belirli sorumlulukları vardır. RISC-V çağırma alışkanlıkları (*calling convention*) yazmaçları iki gruba ayırır:

- $t0-t6$ (geçici yazmaçlar): Çağrılan altyordam bunları özgürce değiştirebilir. Çağırıcı taraf bu yazmaçlardaki değerleri korumak istiyorsa çağrı öncesinde yığıta saklamak zorundadır.
- $s0-s11$ (saklanan yazmaçlar): Çağrılan altyordam bunları kullanmak isterse önce yığıta saklamalı, dönmeden önce geri yüklemelidir.
- $a0-a7$ (argüman yazmaçları): Çağırıcı taraf argümanları bu yazmaçlara yazar; dönüşte $a0-a1$ sonuç değerlerini taşır.
- *ra* (dönüş adresi): Çağrılan altyordam başka bir altyordam çağıracaksa *ra*'yı yığıta saklamalıdır.

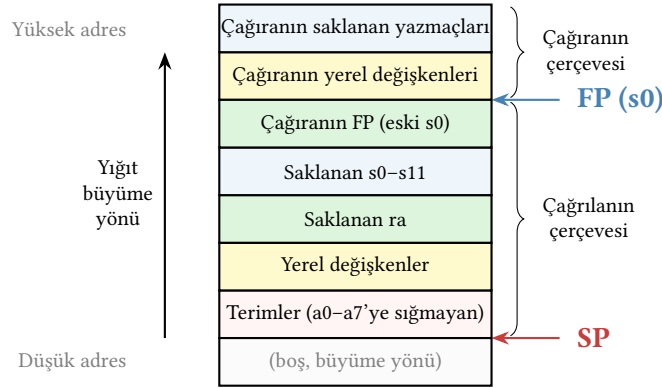
3.14.4 Yığıt Çerçevesi: Giriş ve Çıkış Kodu

Her altyordam, çalışmaya başlamadan önce yığıtta kendine ait bir alan açar (*prologue*, giriş kodu) ve geri dönmeden önce bu alanı kapatır (*epilogue*, çıkış kodu). Bu düzenli yapı, iç içe geçmiş çağrılarının güvenli çalışmasını sağlar.

s0/fp (*frame pointer*) yazmacı yığıt çerçevesinin başlangıç adresini gösterir. Yığıt çerçevesinin tipik yerleşim düzeni Şekil 3.20'de gösterilmiştir.



Şekil 3.20: Altyordam çağrısı sırasında yığıt çerçevesi. Saklanan yazmaçlar, yerel değişkenler ve dönüş adresi tek bir çerçeve içinde yer alır.



Şekil 3.21: İç içe altyordam çağrısında çağırان ve çağrılan tarafın yığıt çerçeveleri. Çağrılanın giriş kodu (*prologue*) çağırانın çerçeve göstergesini (FP) yığıta saklar ve FP'yi kendi çerçevesinin tepesine ayarlar; çıkış kodu (*epilogue*) bu süreci tersine çevirerek çağırانın çerçevesini geri yükler.

Örnek 3.18

Aşağıdaki C işlevini RISC-V çevirici dilinde yazın. İşlev `s0` ve `s1` yazmaçlarını kullandığından bunları yığıta saklamalıdır. İşlev ayrıca başka bir altyordam (yardımcı) çağırıldığından `ra`'yı da korumalıdır.

```
1 int hesapla(int a, int b) {
2     int x = a + b;
3     int y = yardimci(x);
4     return x + y;
5 }
```

Çözüm:

```
1 hesapla:
2     # --- Giriş kodu (prologue) ---
3     addi sp, sp, -12      # 3 söz için yer aç (ra, s0, s1)
4     sw    ra, 8(sp)      # dönüş adresini sakla
5     sw    s0, 4(sp)      # s0'i sakla (callee-saved)
```



```

6      sw    s1, 0(sp)          # s1'i sakla (callee-saved)
7
8      # --- İşlev gövdesi ---
9      add   s0, a0, a1         # x = a + b (s0'da tutuyoruz)
10     mv    a0, s0             # yardımcı'ya argüman olarak x'i geç
11     jal   ra, yardımcı       # y = yardımcı(x) (sonuç a0'da)
12     add   a0, s0, a0         # dönüş değeri = x + y
13
14     # --- Çıkış kodu (epilogue) ---
15     lw    s1, 0(sp)          # s1'i geri yükle
16     lw    s0, 4(sp)          # s0'i geri yükle
17     lw    ra, 8(sp)          # dönüş adresini geri yükle
18     addi  sp, sp, 12         # yığıt çerçevesini kapat
19     ret                                # çağırana dön

```

yardımcı çağrısı sırasında ra üzerine yazılır; ancak giriş kodunda yığta saklandığından, çıkış kodunda güvenle geri yüklenir. s0 ve s1 saklanan yazmaçlar olduğundan çağrılan tarafın bunları koruması zorunludur. Geçici yazmaçlar (t0–t6) ise bu örnekte kullanılmamıştır; kullansaydık yardımcı çağrısı bunları değiştirebilirdi.

Örnek 3.19

Özyinelemeli faktöriyel işlevini (n!) RISC-V çevirici dilinde yazın:

```

1  int faktoriyel(int n) {
2      if (n <= 1) return 1;
3      return n * faktoriyel(n - 1);
4  }

```

Çözüm:

```

1  faktoriyel:
2      # --- Giriş kodu ---
3      addi  sp, sp, -8         # 2 söz için yer aç
4      sw    ra, 4(sp)          # dönüş adresini sakla
5      sw    s0, 0(sp)          # s0'ı sakla
6
7      mv    s0, a0             # s0 = n (koruma altında)
8      li    t0, 1
9      bgt   a0, t0, ozyinele   # n > 1 ise özyineleyle git
10
11     # Taban durumu: n <= 1
12     li    a0, 1               # dönüş değeri = 1
13     jal   zero, geri_don
14
15     ozyinele:
16     addi  a0, s0, -1          # a0 = n - 1
17     jal   ra, faktoriyel      # faktoriyel(n-1), sonuç a0'da
18     mul   a0, s0, a0          # dönüş değeri = n * faktoriyel(n-1)
19
20     geri_don:
21     # --- Çıkış kodu ---
22     lw    s0, 0(sp)          # s0'ı geri yükle
23     lw    ra, 4(sp)          # dönüş adresini geri yükle

```

```

24      addi sp, sp, 8      # yığıt çerçevesini kapat
25      ret

```

Her özyinelemeli çağrıda yığıta yeni bir çerçeve eklenir. Örneğin faktoriyel(3) hesaplanırken yığıt üç kez büyür: her katmanda ra ve s0 saklanır. Taban durumuna ulaşıldığında (n=1) yığıt çerçeveleri ters sırada kapatılarak sonuç üst katmanlara iletilir.

Bilgi Notu 3.13: Yaprak Altyordam Eniyilemesi

Başka bir altyordam çağırmayan işlemlere *yaprak altyordam* (*leaf procedure*) denir. Bu tür işlemlerde ra üzerine yazılmayacağından yığıta saklanmasına gerek yoktur. Saklanan yazmaç kullanılmıyorsa yığıt çerçevesi bile açılmayabilir; bu da işlev çağrısının maliyetini sıfıra indirir. Derleyiciler bu eniyilemeyi otomatik olarak uygular.

Altyordam Çağırma Mekanizmaları

Altyordam çağırma, mimariler arasında farklı yaklaşımlarla çözülür. RISC-V, dönüş adresini ra yazmacına yazar (jal/jalr); çağrı derinleşirse programcı ra değerini yığıta saklamakla yükümlüdür.

MIPS de aynı yaklaşımı benimser (jal + \$ra). ARM (AArch64) ise BL buyruğuyla dönüş adresini x30 (LR) yazmacına yazar; iç içe çağrılarda yığıt kullanımı RISC-V’tekiyle benzerdir.

x86 tamamen farklı bir yol izler: CALL buyruğu dönüş adresini doğrudan yığıta iter, RET ise yığıttan çeker. Bu donanım düzeyindeki yığıt yönetimi programcının işini kolaylaştırır; ancak her çağrıda bellek erişimi gerektirdiğinden başarımlı açısından ek bir bedel taşır.

3.15 Programın Yürütülmesi

C dilinde yazdığınız bir program, “çalıştır” düğmesine bastığınızda doğrudan işlemciye gitmez. Kaynak kodun işlemcinin anlayacağı makine diline dönüşmesi için birden fazla aşamadan geçmesi gerekir; bu süreç Şekil 3.22’de özetlenmiştir:

3.15.1 Derleyici

Derleyici (*compiler*), üst düzey programlama dillerini (C, C++, Java vb.) çevirici dile dönüştüren programdır. Bu dönüşüm sırasında derleyici; söz dizimi denetimi, tür denetimi, en iyileme (*optimization*) ve yazmaç ataması gibi pek çok adımı gerçekleştirir. Yazmaç ataması, Bölüm 3.4’da ele alınan yazmaç sayısı ödünleşmesiyle doğrudan ilişkilidir: az yazmaçlı bir mimaride derleyici daha sık belleğe taşıma kodu üretmek zorunda kalır. Derleyicinin ürettiği çevirici dil dosyası .s (veya .asm) uzantısını taşır.

3.15.2 Çevirici

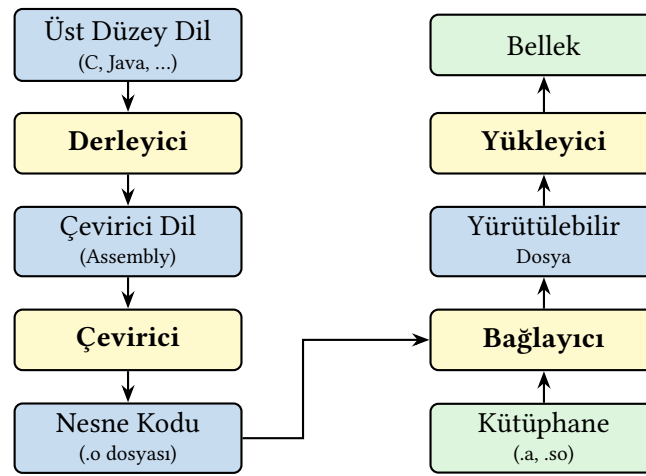
Çevirici (*assembler*), çevirici dilde yazılmış programları nesne dosyasına (.o veya .obj) dönüştüren programdır. Çevirici iki temel iş yapar: (1) buyruk anımsatıcılarını (add, lw gibi) karşılık gelen ikili makine koduna çevirir ve (2) etiketlerin (don:, cik: gibi) adreslerini hesaplayarak bir sembol tablosu oluşturur. Sözde buyruklar da bu aşamada gerçek buyruklara açılır; örneğin mv t0, t1 çevirici tarafından addi t0, t1, 0 olarak kodlanır.

3.15.3 Bağlayıcı

Bağlayıcı (*linker*), birden fazla nesne dosyasını ve kütüphane dosyalarını birleştirerek tek bir yürütülebilir dosya oluşturur. Ayrı derlenen modüllerdeki dış referansları (bir modülde çağrılan ancak başka bir modülde tanımlanan işlevler gibi) çözer ve her modülün adres aralığını belirler. RISC-V’te `jal` buyruğundaki 20 bitlik eklenti, bağlayıcının hesapladığı göreceli adreslere dayanır.

3.15.4 Yükleyici

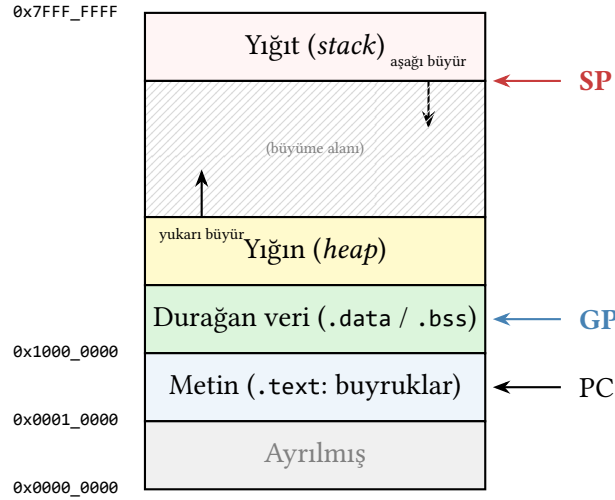
Yükleyici (*loader*), yürütülebilir dosyayı diskten okuyup belleğe yükleyen işletim sistemi bileşenidir. Yükleyici; metin bölümünü (buyrukları), veri bölümünü (ilk değerli küresel değişkenleri) ve yığıt alanını uygun bellek adreslerine yerleştirir, ardından PS’nin değerini programın ilk buyruğunun adresine ayarlayarak yürütmeyi başlatır.



Şekil 3.22: Programın işlenmesi: üst düzey dilden bellekteki yürütülebilir koda uzanan dönüşüm zinciri

3.15.5 Bellek Düzeni ve Bölümler

Yükleyici, programı belleğe yerleştirirken farklı türdeki verileri ayrı bölümlere (*segment*) koyar. Şekil 3.23’de tipik bir RISC-V programının bellek yerleşimi gösterilmiştir.



Şekil 3.23: Tipik bir RISC-V programının bellek yerleşimi

- **.text**: Programın makine kodu buyrukları bu bölümde yer alır. PS başlangıçta bu bölümün ilk adresini gösterir.
- **.data**: İlk değeri atanmış küresel ve durağan değişkenler bu bölümde saklanır. *gp* (*global pointer*) yazmacı bu bölüme hızlı erişim sağlar.
- **.bss**: İlk değeri atanmamış (veya sıfır olan) küresel değişkenler burada yer alır; yükleyici bu alanı sıfırlarla doldurur.
- **Yığın (heap)**: Çalışma zamanında dinamik olarak ayrılan bellek (C’de `malloc`, C++’ta `new`) bu alanda yukarı doğru büyür.
- **Yığıt (stack)**: Altyordam çerçeveleri, yerel değişkenler ve dönüş adresleri burada saklanır. Yığıt yüksek adresten aşağı doğru büyür. *sp* yazmacı yığıtın en tepe noktasını gösterir.

Yığın ve yığıtın birbirine doğru büyümesi, bellek alanının esnek kullanılmasını sağlar. Ancak birbirine çok yaklaşırlarsa yığıt taşması (*stack overflow*) veya bellek tükenmesi oluşabilir.

3.15.6 Sistem Çağrılar ve `ecall`

Bir kullanıcı programı, ekrana yazdırma, dosya okuma veya bellek ayırma gibi işletim sistemi hizmetlerini doğrudan gerçekleştiremez; bu hizmetlere erişmek için işletim sistemine bir istek göndermesi gerekir. RISC-V’te bu istek `ecall` (*environment call*) buyruğuyla yapılır.

`ecall` yürütüldüğünde işlemci, denetimi kullanıcı programından işletim sisteminin çekirdek koduna aktarır. Hangi hizmetin istendiği `a7` yazmacına yazılan hizmet numarasıyla belirlenir; argümanlar `a0–a6` yazmaçlarıyla aktarılır.

```

1  li    a7, 64      # write sistem çağrısı (Linux)
2  li    a0, 1       # dosya tanımlayıcısı: stdout
3  la    a1, mesaj   # yazdırılacak metnin adresi
4  li    a2, 13      # metin uzunluğu (bayt)
5  ecall            # işletim sistemini çağır
```

Bilgi Notu 3.14: `ecall` ve Ayrıcalık Düzeyleri

`ecall` buyruğu, işlemcinin ayrıcalık düzeyini yükseltir (kullanıcı → gözetmen veya makine). Bu geçiş sırasında program sayacı işletim sisteminin kural dışı durum işleyicisine yönlendirilir. Ayrıcalık düzeyleri Ek C’de özetlenmiştir.

3.16 Diğer BKM'lerde Buyruk Kümeleri

Bu bölüm boyunca her konunun ardında yer alan karşılaştırma kutularında RISC-V ile diğer mimarilerin farklılıklarına değindik. Bu kısımda MIPS, ARM ve x86 mimarilerini daha ayrıntılı tanıtarak bu farklılıkların arkasındaki tasarım kararlarını bütüncül bir bakışla ele alacağız.

3.16.1 MIPS Buyruk Kümesi

MIPS (*Microprocessor without Interlocked Pipeline Stages*), 1981'de Stanford Üniversitesi'nde John L. Hennessy öncülüğünde tasarlanmış bir RISC mimarisidir. MIPS, bilgisayar mimarisi eğitiminde uzun yıllar referans mimari olarak kullanılmıştır. RISC-V ile MIPS arasındaki temel farklar Tablo 3.23'da karşılaştırmalı olarak verilmiştir.

Tablo 3.23: RISC-V ile MIPS'in karşılaştırması

Özellik	RISC-V	MIPS
Yazmaç sayısı	32 (x0–x31)	32 (\$0–\$31)
Yazmaç genişliği	32/64/128 bit	32/64 bit
Buyruk biçimi sayısı	6 (R, I, S, B, U, J)	3 (R, I, J)
Anlık değer genişliği	12 bit (I türü)	16 bit (I türü)
Dallanma buyrukları	beq, bne, blt, bge, bltu, bgeu	beq, bne (+ slt)
Çarpma sonucu	Genel yazmaçlara	HI/LO yazmaçlarına
Geciktirilmiş dallanma	Yok	Var
Lisans	Açık kaynak	Ticari
Taşma kural dışı durumu	Yok (varsayılan)	add/sub'da var

Tablo 3.24: Aynı işlemin RISC-V ve MIPS'teki karşılıkları

İşlem	RISC-V	MIPS
Toplama	add s1, s2, s3	add \$s1, \$s2, \$s3
Sözcük yükleme	lw s1, 0(s2)	lw \$s1, 0(\$s2)
Koşulsuz atlama	jal zero, etiket	j etiket
Altyordam çağırısı	jal ra, etiket	jal etiket
Altyordamdan dönüş	jalr zero, ra, 0	jr \$ra
Küçükse dallan	blt s1, s2, etiket	slt \$t0, \$s1, \$s2; bne \$t0, \$zero, etiket
DEĞİL işlemi	xori s1, s2, -1	nor \$s1, \$s2, \$zero

Aynı işlemlerin RISC-V ve MIPS'teki söz dizimi farklılıkları Tablo 3.24'de örneklenmiştir.

3.16.2 ARM Buyruk Kümesi

ARM (*Advanced RISC Machine*), 1985 yılında Acorn Computers tarafından tasarlanmış, ardından ARM Holdings bünyesinde geliştirilen bir RISC mimarisidir. Düşük güç tüketimi ve yüksek işlem başarımı dengesi sayesinde ARM, günümüzün en yaygın kullanılan buyruk kümesi mimarisi konumuna gelmiştir. Dünya genelinde üretilen akıllı telefonların büyük çoğunluğu ARM işlemci kullanmaktadır. Apple'ın M1/M2/M3 serisi işlemcileri ve AWS Graviton sunucu işlemcileri de ARM (AArch64) mimarisine dayanmaktadır. Gömülü sistemlerden süper bilgisayarlara kadar uzanan bu geniş uygulama yelpazesi, ARM'ı pazar payı açısından RISC-V'ten ve x86'dan çok daha önde konumlandırmaktadır.

ARMv8 ile birlikte tanıtılan 64 bitlik AArch64 mimarisi, önceki AArch32 mimarisine kıyasla köklü yenilikler getirmiştir. AArch64'te 31 adet genel amaçlı 64 bitlik yazmaç (x0–x30) ve bir sıfır yazmacı (xZR/WZR) bulunur. Kayan noktalı ve SIMD işlemleri için ayrı 32 adet 128 bitlik yazmaç (v0–v31) kullanılır. Buyruklar sabit 32 bit uzunluğundadır; bu yapı ardışık düzen tasarımını basitleştirir. AArch32'deki koşullu yürütme modeli (her buyruca koşul alanı ekleme) AArch64'te kaldırılmış, bunun yerine CSEL ve CSINC gibi koşullu seçim buyrukları eklenmiştir. ARM ayrıca NEON (Advanced SIMD), SVE ve SVE2 vektör uzantılarını destekler.

Tablo 3.25: RISC-V ile ARM (AArch64) karşılaştırması

Özellik	RISC-V	ARM (AArch64)
Yazmaç sayısı	32 tamsayı + 32 FP	31 tamsayı + xZR + 32 FP
Buyruk uzunluğu	32 bit (sabit)	32 bit sabit (AArch32'de 16/32 bit)
Koşullu yürütme	Yok	CSEL/CSINC koşullu seçim buyrukları
Çarpma/bölme	M uzantısı	Temel kümede var
Adresleme kipleri	4	~6 (ön/son dizinli dahil)
Bayt sırası	Küçüğü başta	İkisi de; uygulamada küçüğü başta
Lisans	Açık kaynak	Ticari lisans
SIMD/Vektör	V uzantısı	NEON / SVE / SVE2

Tablo 3.26: Aynı işlemin RISC-V ve ARM (AArch64)'teki karşılıkları

İşlem	ARM (AArch64)	RISC-V
Toplama	add x0, x1, x2	add a0, a1, a2
Bellekten yükleme	ldr x0, [x1, #8]	lw a0, 8(a1)
Belleğe saklama	str x0, [x1, #8]	sw a0, 8(a1)
Dallanma (eşitse)	b.eq etiket (bayrak tab.)	beq a0, a1, etiket
Altyordam çağırma	bl func	jal ra, func

Tablo 3.25'te görüldüğü üzere, RISC-V ve ARM (AArch64) pek çok açıdan birbirine benzer. Her iki mimari de sabit uzunluklu buyruklar ve yükle-sakla modelini benimser. Temel ayrım lisanslama modelindedir: RISC-V açık kaynaklı ve telif hakkından bağımsız iken ARM ticari lisans gerektirir. Tablo 3.26'deki buyruk söz dizimi karşılaştırmasında dikkat çeken önemli bir nokta, ARM'ın dallanma için bayrak yazmaçlarını (koşul kodları) kullanması, RISC-V'in ise karşılaştırma işlemini doğrudan dallanma buyruğuna dahil etmesidir.

3.16.3 x86 Buyruk Kümesi

Intel'in 1978'deki 8086 işlemcisiyle başlattığı ve 80x86 ailesi olarak da bilinen işlemci serisinin buyruk kümesine x86 BKM'si denir.

x86'nın Tarihçesi

1978 yılında piyasaya sürülen 8086 ile başlayan Intel x86 işlemcileri serisi 80186, 80286, 386 ve 486 ile sürmüştür. x86 BKM'sinin özelliği geçmişe uyumluluk ilkesinin benimsenmiş olmasıdır. Günümüzde kullanılan x86 tabanlı işlemcilerin hepsi ilk çıkan 8086 işlemcisinin buyruklarını işletebilir.

İşlemciye dışarıdan gelen karmaşık makro buyruklar, işlemcinin içinde yalın mikro buyruklar'a dönüştürülür. Bu yapı güncel x86 işlemcilerinin hem RISC hem de CISC özellikleri taşımasını sağlar.

x86 Yazmaçları

x86'da 8 tane genel amaçlı yazmaç bulunur. İlk çıkan 8086'da 16 bitlik olan bu yazmaçlar IA-32'de 32 bite, x86-64'te 64 bite genişletilmiştir. Ayrıca x86-64 ile birlikte R8–R15 yazmaçları eklenerek genel amaçlı yazmaç sayısı 16'ya çıkarılmıştır; bu yazmaçlar ve görevleri Tablo 3.27'da özetlenmiştir.

Tablo 3.27: x86'daki bazı yazmaçlar

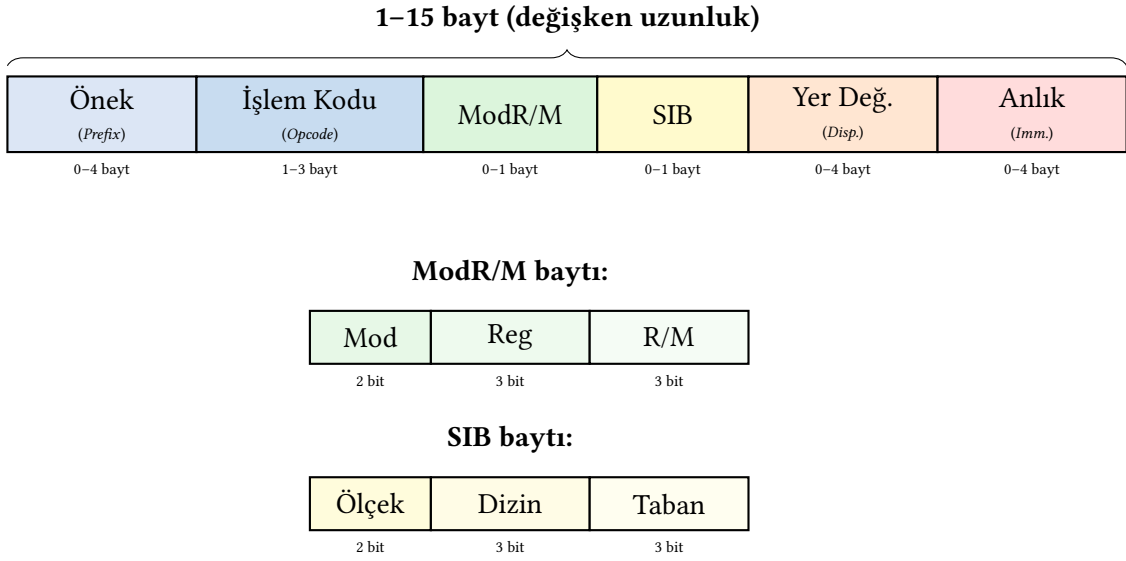
x86-64 (64 bit)	IA-32 (32 bit)	8086 (16 bit)	Üst-Alt	Görev
RAX	EAX	AX	AH, AL	Biriktireç
RBX	EBX	BX	BH, BL	Taban dizini
RCX	ECX	CX	CH, CL	Sayaç
RDX	EDX	DX	DH, DL	Veri
RDI	EDI	DI	–	Hedef dizini
RSI	ESI	SI	–	Kaynak dizini
RBP	EBP	BP	–	Çerçeve göstergesi
RSP	ESP	SP	–	Yığıt göstergesi

x86 Söz Dizimi

Intel söz diziminde işlem kodunun ardından ilk değişken sonuç, sonrakiler kaynak verileridir. AT&T'de ise kaynak önce, sonuç sonda gelir. Intel söz diziminde bellek adresleri köşeli ayraç, AT&T'de yay ayraç içinde gösterilir.

x86 Buyrukları

Intel işlemcilerde buyrukların boyutları aynı değildir. RISC-V'ten farklı olan bu özellik nedeniyle bellekten çekilecek olan buyruğun uzunluğu buyruk okunmadan bilinemez. x86 buyrukları 1 bayttan 15 bayta kadar değişen uzunluklarda olabilir; bu değişken biçim yapısı Şekil 3.24'de gösterilmiştir.

**Şekil 3.24:** x86’da Değişik Buyruk Biçimleri

3.16.4 Dört Mimarinin Genel Karşılaştırması

Tablo 3.28’de RISC-V, MIPS, ARM ve x86 mimarileri temel özellikler açısından karşılaştırılmıştır.

Tablo 3.28: RISC-V, MIPS, ARM ve x86 mimarilerinin karşılaştırması

Özellik	RISC-V	MIPS	ARM (AArch64)	x86-64
Tür	RISC	RISC	RISC	CISC
Lisans	Açık	Ticari	Ticari	Ticari
Yazmaç sayısı	32	32	31+ZR	16
Buyruk boyutu	32 bit	32 bit	32 bit	Değişken
Bayt sırası	Küçüğü başta	İkisi de	İkisi de	Küçüğü başta
Anlık değer	12/20 bit	16/26 bit	12/16 bit	8/16/32 bit
Adresleme kipi	4	3	~6	10+
Koşullu dallanma	beq, blt	slt+beq	NZCV kodları	FLAGS yazmacı
Çarpma/bölme	M uzantısı	HI/LO	Temel kümede	Temel kümede
Altyordam çağırma	jal (yazmaç)	jal (yazmaç)	BL (yazmaç)	CALL (yığıt)
Uzantı modeli	Modüler	Sabit	Profil tabanlı	Monolitik kümülatif

3.17 Diğer Bazı Buyruk Kümesi Mimarileri

Bilgisayar tarihi boyunca pek çok buyruk kümesi mimarisi tasarlanmıştır. Bu mimarilerin bir kısmı artık üretilmese de, her biri günümüz tasarımlarını etkileyen kavramsal katkılar yapmıştır. Tablo 3.29’te bu mimarilerin özellikleri özetlenmiştir.

Tablo 3.29: Bilgisayar tarihinde öne çıkan diğer buyruk kümesi mimarileri

Mimari	Tasarımcı	Yıl	Tür	Öne çıkan özellik
System/360	IBM	1964	CISC	İlk mimari aile kavramı; geriye uyumluluk geleneğini başlattı
VAX	DEC	1977	CISC	En zengin adresleme kipleri; RISC tartışmasının başlamasına neden oldu
M68000	Motorola	1979	CISC	İlk Macintosh ve Amiga işlemcisi; düzenli 32-bit tasarım
SPARC	Sun	1987	RISC	Yazmaç penceresi kavramı; ilk açık kaynak BKM
PA-RISC	HP	1986	RISC	VLIW kavramına geçiş deneyi (Itanium'un öncüsü)
Power	IBM	1990	RISC	Sunucu ve süper bilgisayarlarda yaygın; POWER10 ile hâlâ üretimde
Alpha	DEC	1992	RISC	Döneminin en hızlı işlemcisi; 64-bit RISC öncüsü
SuperH	Hitachi	1992	RISC	16-bit sıkıştırılmış buyruklar; RISC-V C uzantısına ilham verdi

Bu mimarilerin çoğu artık üretilmemektedir. Ancak katkıları yaşamaktadır: System/360'ın başlattığı geriye uyumluluk geleneği x86'da sürer, SPARC'ın açık kaynak yaklaşımı RISC-V'e ilham vermiştir, Alpha'nın 64-bit tasarım ilkeleri ARM AArch64 ve RISC-V RV64'te iz bırakmıştır.

3.18 Gerçek Dünya: Buyruk Kümesi Tasarım Kararları

Buyruk kümesi tasarımı akademik bir konu olmaktan çok ötedir; endüstrideki her yeni mimari nesil, buyruk kümesi düzeyinde verilen kararların sonuçlarını taşır.

x86'nın geriye uyumluluk yükü

Intel'in x86 mimarisi 1978'den bu yana geriye uyumluluğu korur: günümüzün en güçlü Xeon işlemcisi bile 8086 için yazılmış bir programı çalıştırabilir. Bu esneklik muazzam bir ticari avantaj sağlasa da her nesil işlemciye ek karmaşıklık yükler. Çağdaş x86 işlemcileri, CISC buyrukları içeride RISC benzeri mikro-işlemlere çevirirken transistör bütçesinin önemli bir bölümünü bu dönüşüm birimine ayırır.

ARM'ın elde taşınır cihazlardaki hakimiyeti

ARM mimarisi, düşük güç tüketimi odaklı tasarımıyla akıllı telefonların neredeyse tamamında kullanılır. 2020'den itibaren Apple'ın M serisi işlemcilerle dizüstü ve masaüstü pazarına girmesi, ARM'ın "yalnızca düşük güçlü cihazlar için" algısını kökten değiştirdi. ARM'ın lisanslama modeli, farklı şirketlerin aynı buyruk kümesi üzerinde kendi mikromimarilerini tasarlamasına olanak tanır.

Bilgi Notu 3.15: Neden Elde Taşınır Cihazlarda x86 Yok?

Akıllı telefonlar, tabletler ve giyilebilir cihazlar neredeyse tamamıyla ARM tabanlı işlemcilerle çalışır. Intel, 2012–2016 yılları arasında Atom işlemci ailesiyle bu pazara girmeye çalışmış, hatta telefon üreticilerine mali destek sunmuş, ancak başarılı olamamıştır. Bunun temel nedenleri buyruk kümesi mimarisinin doğasında yatmaktadır:

1. *Güç verimliliği*: x86'nın karmaşık, değişken uzunluklu buyrukları her saat dönüşünde mikro-işlemlere çözülür; bu dönüşüm birimi sürekli güç harcar. ARM'ın sabit uzunluklu, basit buyrukları çok daha az enerjiyle çözülebilir. Pille çalışan bir cihazda her miliwatt değerlidir.
2. *Çözme karmaşıklığı*: x86'da buyruk uzunluğu 1 ile 15 bayt arasında değişir; bu durum çözme donanımını büyütür ve ısı üretimini artırır. ARM ve RISC-V'te sabit 4 baytlık buyruklar koştur çözme kolaylaştırır.
3. *Transistör bütçesi*: x86'nın geriye uyumluluk yükü, transistörlerin önemli bir bölümünü "eski buyrukları destekleme" için harcar. ARM tasarımları bu bütçeyi daha fazla çekirdeğe veya daha büyük önbelleğe ayırabilir.
4. *Lisanslama esnekliği*: ARM, BKM lisansını satarak farklı şirketlerin (Apple, Qualcomm, Samsung) aynı buyruk kümesi üzerinde kendi mikromimarilerini tasarlamasına olanak tanır. Intel ise hem BKM'yi hem üretimi kendi elinde tuttuğundan, cihaz üreticilerine bu esnekliği sunamaz.

Apple'ın 2020'de kendi ARM tabanlı M1 işlemcisiyle dizüstü bilgisayar pazarına girmesi, bu esnekliğin gücünü göstermiştir. M1'den M5'e uzanan bu yonga ailesi, her kuşakta aynı güç zarfında x86 rakiplerini geride bırakmaya devam etmektedir.

RISC-V'in yükselişi

RISC-V'nin açık kaynak doğası, onu özellikle gömülü sistemlerde, akademik araştırmalarda ve Çin'in yarı iletken endüstrisinde hızla benimsenen bir seçenek haline getirdi. SiFive, Alibaba (Xuantie serisi) ve Bouffalo Lab gibi şirketler ticari RISC-V işlemcileri üretmektedir.

2025 yılında Meta'nın RISC-V tabanlı sunucu işlemcisi geliştiren RIVOS şirketini satın alması, RISC-V'in gömülü sistemlerin ötesine geçerek veri merkezi pazarına girişinin somut bir göstergesi olmuştur. Bu satın alma, dünyanın en büyük veri merkezi işletmecilerinden birinin ARM ve x86'ya seçenek olarak RISC-V'e yatırım yaptığını göstermekte ve açık kaynak buyruk kümesi mimarisinin sunucu düzeyinde rekabet edebileceği mesajını vermektedir.

Veri koşutluğu ve BKM uzantıları

İşlemcilerin aynı anda birden fazla veri üzerinde aynı işlemi yapabilmesi (*Single Instruction, Multiple Data*, SIMD) kavramı, buyruk kümelerinin evriminde belirleyici bir rol oynamıştır. Bu uzantılar, multimedya işleme, bilimsel hesaplama ve yapay zekâ iş yüklerindeki veri koşutluğunu donanım düzeyinde destekler.

x86 mimarisinde bu evrim kümülatif biçimde ilerlemiştir:

- **MMX** (1997): 64 bitlik yazmaçlarla 8 adet 8 bitlik tam sayı işlemi; multimedya odaklı ilk adım.
- **SSE** (1999): 128 bitlik *xmm* yazmaçları; 4 adet tek duyarlıklı kayan nokta işlemi koştur olarak yürütülür.
- **AVX** (2011): 256 bitlik *ymm* yazmaçları; veri genişliği iki katına çıkar.

- **AVX-512** (2017): 512 bitlik zmm yazmaçları; 16 adet tek duyarlıklı kayan nokta işlemi tek buyrukla yapılır.

Her yeni uzantı öncekini kapsadığından, x86 işlemcileri tüm eski SIMD buyruklarını da desteklemek zorundadır; bu da geriye uyumluluk yükünü artırır.

ARM mimarisi benzer gereksinime farklı yanıt vermiştir. NEON uzantısı (128 bit) sabit genişlikteyken, SVE (*Scalable Vector Extension*) ve onun geliştirilmiş sürümü SVE2 değişken vektör uzunluğu sunar: aynı buyruk, 128 bitten 2048 bite kadar farklı genişliklerde çalışabilir. Bu yaklaşım, yazılımın donanımdan bağımsız olmasını sağlar.

RISC-V “V” (vektör) uzantısı ise en esnek modeli sunar. Vektör uzunluğu buyruk kümesinde sabitlenmez; donanım ne kadar geniş vektör birimi sağlıyorsa yazılım o kadarını kullanır. Bu tasarım, küçük bir gömülü işlemcide 64 bitlik, yüksek başarımlı bir sunucuda 1024 bitlik vektörlerle aynı programın çalışmasını mümkün kılar.

Yapay zekâ çağında buyruk kümesinin dönüşümü

Yapay zekâ iş yükleri, buyruk kümesi tasarımını köklü biçimde etkilemektedir. Geleneksel hesaplama 32 veya 64 bitlik kayan nokta duyarlılığı gerektirirken, sinir ağı çıkarımı çok daha düşük duyarlılıkla çalışabilir. Bu gereksinim, buyruk kümelerine yeni veri türleri ve özel buyruklar eklenmesine yol açmıştır:

- **Düşük duyarlıklı veri türleri:** INT8, FP16 ve BF16 (*bfloat16*) gibi dar veri türleri, aynı bant genişliğiyle iki veya dört kat daha fazla veri işlemeyi mümkün kılar. Intel AMX (*Advanced Matrix Extensions*), ARM SME (*Scalable Matrix Extension*) ve RISC-V’in önerilen matris uzantıları bu veri türlerini doğrudan destekler.
- **Matris buyrukları:** Sinir ağlarının temelindeki matris çarpımı, geleneksel skaler veya SIMD buyrukları yerine özel matris birimlerinde çok daha verimli yürütülür. Google TPU, bu yaklaşımın en uç örneğidir: buyruk kümesinin tamamı matris işlemlerine odaklanmıştır.
- **Alana özgü hızlandırıcılar:** Sinir ağı motoru (NPU) artık birçok yonganın standart bileşenidir (Apple Neural Engine, Qualcomm Hexagon). Bu birimler BKM’den bağımsız yan işlemci olarak çalışır; ancak bazıları ana BKM’den özel buyrukla tetiklenir.

RISC-V’in bu dönüşümde öne çıkmasının nedeni, şirketlerin standart buyruk kümesine kendi özel YZ uzantılarını ekleyebilmesidir. Ticari mimarilerde böyle bir esneklik yoktur; x86 veya ARM kullanıcısı, üretici firmanın sunduğu uzantılarla yetinmek zorundadır. Bu esneklik, RISC-V’i YZ hızlandırıcılarının giderek daha çok tercih ettiği bir temel haline getirmektedir.

3.19 Yanılgılar ve Tuzaklar

Buyruk kümesi mimarisi, sezgisel olarak basit görünse de öğrencilerin sık düştüğü tuzaklar içerir.

Yanılgı: Daha fazla buyruk türü olan mimari daha güçlüdür

CISC mimarilerinin zengin buyruk kümesi, her buyruğun eşit sıklıkta kullanıldığı anlamına gelmez. x86’nın yüzlerce buyruğunun büyük çoğunluğu derleyiciler tarafından nadiren üretilir. RISC felsefesi, az sayıda ama sık kullanılan buyruğu hızlı çalıştırmanın daha verimli olduğunu göstermiştir.

Yanılğı: Yazmaç sayısı arttıkça başarımlar her zaman artar

Daha fazla yazmaç, derleyiciye daha fazla esneklik sunar; ancak her ek yazmaç, buyruk kodlamasında daha fazla bit gerektirir (RISC-V’te 5 bit = 32 yazmaç). Yazmaç sayısının artması buyruk boyutunu büyütür, bu da buyruk önbelleğinde daha fazla yer kaplar ve bellek bant genişliğini tüketir.

Yanılğı: Sabit uzunluklu buyruklar her zaman değişken uzunlukludan iyidir

RISC-V ve MIPS gibi mimarilerde 32 bitlik sabit buyruk uzunluğu, kod çözmeyle basitleştirir. Ancak bu, kod yoğunluğunu azaltır; aynı program daha fazla bellek kaplar. ARM’ın Thumb-2 modu ve RISC-V’in “C” (sıkıştırılmış) uzantısı, 16 bitlik kısa buyruklar ekleyerek bu sorunu hafifletir.

Tuzak: Küçüğü başta ve büyüğü başta sıralamasını karıştırmak

Ağ üzerinden veri aktarımında genellikle büyüğü başta (big-endian) sıralama kullanılırken, x86 ve RISC-V küçüğü başta (little-endian) kullanır. Farklı sıralamaya sahip sistemler arasında veri aktarırken bayt sırası dönüştürülmezse veriler bozulur. Bu, özellikle gömülü sistem ve ağ programlamada sık karşılaşılan bir hatadır.

Tuzak: İşaret uzatmasını göz ardı etmek

1b buyruğu bellekten okuduğu 8 bitlik değeri 32 bite işaret uzatarak yazar; en yüksek değerlikli bit 1 ise yazmaç 0xFFFFFxx biçiminde dolar. İşaretsiz bir değer okunmak isteniyorsa 1bu kullanılmalıdır. Aynı durum 1h/1hu çifti için de geçerlidir. Bu ayrımın gözden kaçması, özellikle karakter işleme ve görüntü verisi gibi işaretsiz bayt dizileriyle çalışırken bulunması güç hatalara yol açar.

Tuzak: Anlık değerin 12 bitlik sınırını aşmak

I-tipi buyruklardaki anlık alan yalnızca 12 bittir ve –2048 ile +2047 arasında değer taşıyabilir. Bu sınırı aşan bir sabiti doğrudan addi ile yüklemek derleme hatasına neden olur. Büyük sabitler için önce lui ile üst 20 bit, ardından addi ile alt 12 bit yüklenir. Çevirici, 1i sözde buyruğunu tam da bu iki gerçek buyruğa dönüştürerek programcıyı bu ayrıntıdan kurtarır; ancak üretilen kodun iki buyruk yer kapladığını bilmek, döngü gövdelerinde buyruk sayısını tahmin ederken önemlidir.

Tuzak: Çağırana ve çağrılan taraf yazmaçlarını karıştırmak

RISC-V çağırma alışkanlığında s0–s11 yazmaçları çağrılan tarafça saklanır; altyordam bu yazmaçları kullanıyorsa giriş kodunda yığıta yazıp çıkış kodunda geri yüklemelidir. Buna karşılık t0–t6 geçici yazmaçları çağırana tarafın sorumluluğundadır ve altyordam bunları serbestçe bozabilir. Bu ayrımı karıştırmak, altyordam dönüşünde beklenmeyen yazmaç değerlerine ve izlenmesi güç hatalara yol açar.

Yanılğı: Sözde buyruklar donanımda ayrı buyruklardır

`mv, li, nop, j` gibi sözde buyruklar programcının işini kolaylaştıran kısaltmalardır; donanımda ayrı bir işlem kodu yoktur. Çevirici bunları gerçek buyruklara dönüştürür: `mv rd, rs1` → `addi rd, rs1, 0`; `nop` → `addi x0, x0, 0`; `j etiket` → `jal x0, etiket`. Sözde buyrukları “gerçek” sanmak, buyruk sayma ve kodlama sorularında hatalı sonuçlara götürür.

3.20 Özet

Bu bölümde bilgisayarın “ana dilini”, yani buyruk kümesi mimarisini, öğrendik. Bilgisayarlar yalnızca 0 ve 1’lerle çalışır; yazılım ile donanım arasındaki köprüyü buyruk kümesi kurar. Karmaşık buyrukları destekleyen mimariler CISC, yalın buyrukları destekleyen mimariler ise RISC olarak adlandırılır.

Bu kitapta kullandığımız RISC-V, UC Berkeley’de 2010 yılında tasarlanmış açık kaynaklı bir RISC buyruk kümesi mimarisidir. Modüler yapısı sayesinde RV32I temel buyruk kümesinin üzerine M, A, F, D, C gibi uzantılar eklenerek işlemci gereksinime göre özelleştirilebilir.

RISC-V’in 32 tamsayı yazmacı, işlemcinin en hızlı saklama birimleridir; sık kullanılan veriler yazmaçlarda tutularak bellek erişimi azaltılır. Yazmaçların yetersiz kaldığı durumlarda derleyici bazı değerleri yığita yazarak yazmaçtan belleğe taşıma yapar.

Bir programcının gereksinim duyduğu temel buyruk türleri (aritmetik, mantık, veri aktarımı, kaydırma ve dallanma) RISC-V’te altı biçimde (R, I, S, B, U, J) kodlanır. Bu buyruklar, verilerine yazmaç, anlık, eklemeli (taban), PS’ye göreceli ve üst anlık gibi farklı adresleme kipleriyle erişir. Çevirici, `mv, li, nop` gibi sözde buyrukları gerçek buyruklara dönüştürerek programcıya kolaylık sağlar; bu sözde buyruklar donanımda ayrı bir işlem koduna karşılık gelmez.

Altyordamlar, kodun yeniden kullanılmasını sağlayan temel yapı taşlarıdır. RISC-V’te `jal` ile altyordam çağrılır, `jalr` ile geri dönülür. İç içe çağrılarda dönüş adresi ve saklanması gereken yazmaçlar yığita yazılır. Çağırma alışkanlıkları, hangi yazmaçların çağırılan, hangilerinin çağırılan tarafça saklanacağını belirleyerek yazılım modüllerinin birbirinden bağımsız biçimde derlenmesini olanaklı kılar.

Bir programın yürütülebilir dosyaya dönüşme süreci dört aşamadan oluşur: derleyici üst düzey dili çeviriye dönüştürür, çevirici bunu nesne dosyasına çevirir, bağlayıcı nesne dosyalarını ve kütüphaneleri birleştirerek yürütülebilir dosyayı oluşturur, yükleyici ise bu dosyayı belleğe yerleştirir. Bellekteki program, metin (.text), veri (.data, .bss), yığın (heap) ve yığıt (stack) bölgelerine ayrılır.

RISC-V, MIPS, ARM ve x86 mimarileri farklı tasarım felsefelerini yansıtır. RISC-V ve MIPS sabit uzunluklu buyruk ve yük-sakla modeliyle yalınlığı öncelerken, x86 değişken uzunluklu buyruklar ve bellek-yazmaç işlemleriyle geriye uyumluluğu korur. ARM ise koşullu yürütme ve esnek kaydırıcı gibi özellikleriyle ikisinin arasında bir noktada durur.

Bu bölümde ele alınan kavramlar, bilgisayar mimarisinde izlenen dört ana ilkeyi somutlaştırır:

1. **Olağan durumu hızlandır:** Sık kullanılan buyruklar kısa ve hızlı tasarlanır; “C” uzantısı en yaygın buyrukları 16 bite sıkıştırarak kod yoğunluğunu artırır.
2. **Yalınlık düzenden gelir:** Sabit uzunluklu buyruk biçimi ve düzenli yazmaç alanları, donanımda kod çözme devresini yalınlaştırır.

3. **Küçük olan hızlıdır:** 32 yazmaçlık dosya, binlerce adresli bellekten çok daha hızlı erişilebilir; bu nedenle sık kullanılan veriler yazmaçlarda tutulur.
4. **İyi tasarım ödünleşme gerektirir:** Yazmaç sayısı, buyruk uzunluğu ve adresleme kipi çeşitliliği arasındaki denge, mimari tasarımın temel ödünleşmesidir.

Alıştırmalar

Alıştırma 3.1. ★ Aşağıdaki ikilik tabanda verilmiş olan sayıları onluk tabana çevirin.

(a) $(10111001100)_2$ (b) $(111010)_2$ (c) $(1000111)_2$

Alıştırma 3.2. ★ Onluk tabanda verilmiş olan sayıları ikilik tabana çevirin.

(a) $(128)_{10}$ (b) $(14)_{10}$ (c) $(557)_{10}$

Alıştırma 3.3. ★★ Aşağıdaki ikilik tabandaki dört işlem alıştırmalarını çözün.

(a) $1001100 + 111011 = ?$ (b) $111001 - 100111 = ?$

(c) $110011 \div 110 = ?$ (d) $101 \times 111 = ?$

Alıştırma 3.4. ★ İşlemcide buyrukların geçtiği aşamalar nelerdir? Bir buyruğun işlemci içindeki ömründe sırasıyla hangi işlemlerin yapılması gerekir?

Alıştırma 3.5. ★ İkiye tümleyen aritmetiğinin bire tümleyen ve işaretli büyüklük gösterim biçimlerine göre üstünlüğü nedir?

Alıştırma 3.6. ★★ İşlemcilerde eksi sayılar nasıl gösterilir? -8 sayısını göstermek için değişik gösterim yöntemlerinde kaç bit gerekir?

Alıştırma 3.7. ★ “Küçüğü Başta” ve “Büyüğü Başta” terimlerini açıklayın.

Alıştırma 3.8. ★★ İşlemcilerde kullanılan adresleme yöntemlerini yazarak bu yöntemlerin nasıl çalıştığını açıklayın. RISC-V’te yükleme ve saklama işlemlerinin yapılması için hangi adresleme yöntemi kullanılır?

Alıştırma 3.9. ★★ $(4A3B2C1D)_{16}$ sözcüğünün bellekte küçüğü başta ve büyüğü başta yöntemlerini kullanan iki ayrı işlemcide nasıl durduğunu gösterin.

Alıştırma 3.10. ★ Yazmaç türlerini açıklayın.

Alıştırma 3.11. ★★ RISC-V mimarisinde buyruklar kaç bit ile ifade edilir? Intel x86 mimarisinde buyruk yapıları nasıldır? Açıklayın.

Alıştırma 3.12. ★★ RISC-V buyruk kümesi mimarisinde kaç tür buyruk biçimi vardır? Bu buyruk biçimlerini açıklayın.

Alıştırma 3.13. ★ Java, C gibi üst düzey bir programlama dilinde yazılmış bir kod parçasının bilgisayar tarafından çalıştırılmasına kadar geçtiği evreleri açıklayın.

Alıştırma 3.14. ★ Derleyicilerin kullanım amaçları ve programcılara sağladığı avantajları açıklayın.

Alıştırma 3.15. ★★ Dört tasarım ilkesini örnek vererek açıklayın: (a) Olağan durumu hızlandır, (b) Yalınlık düzenden gelir, (c) Küçük olan hızlıdır, (d) İyi tasarım için ödünleşme gerekir.

Alıştırma 3.16. ★ Yapılan işlemlerin işleçlerinin ve sonucunun yazmaçlarda tutulmasının amacı nedir?

Alıştırma 3.17. ★★ RISC-V mimarisinde 32 yazmaç bulunur. Bir programda 32'den fazla sayıda değişken bulunduğunda ne olur?

Alıştırma 3.18. ★★ RISC-V'te çarpma işleminin sonucu nereye yazılır? MIPS'teki HI ve LO yazmaçları ile RISC-V'in yaklaşımını karşılaştırın.

Alıştırma 3.19. ★ RISC-V'te bir yazmacın değerini başka bir yazmaca kopyalamak için hangi buyruk kullanılır? Bu işlemi gerçekleştiren kodu yazın.

Alıştırma 3.20. ★ Sözcüklerin bellekte hizalanmış olarak durması ne demektir? RISC-V'te sözcükler bellekte nasıl durur?

Alıştırma 3.21. ★★ Buyruk kümesinde daha fazla genel amaçlı yazmaç olması neden iyidir? Yazmaç sayısı çok düşük olursa ne tür sorunlar ortaya çıkar? Yazmaç sayısının çok yüksek olmasının ne tür yan etkileri olabilir?

Alıştırma 3.22. ★★ RISC-V'te program sayacına her vuruşta 4 eklenmesinin nedeni nedir? Intel'in x86 tabanlı işlemcilerinde program sayacına her vuruşta kaç eklenir?

Alıştırma 3.23. ★★ RISC-V'te b1t buyruğunun MIPS'teki s1t ve bne buyruk çiftiyle karşılaştırıldığında sağladığı avantajı açıklayın.

Alıştırma 3.24. ★★ RISC-V'in modüler buyruk kümesi mimarisinin avantajlarını açıklayın. RV32I temel buyruk kümesine M, F ve D uzantılarının eklenmesi ne tür işlevsellik kazandırır?

Alıştırma 3.25. ★★★ RISC-V, MIPS, ARM ve x86 mimarilerini buyruk boyutu, yazmaç sayısı, adresleme kipleri ve lisans modeli açısından karşılaştırın.

Alıştırma 3.26. ★ Aşağıdaki sözde buyrukların her birinin hangi gerçek RISC-V buyruğuna (veya buyruklarına) dönüştürüldüğünü yazın: (a) `mv t0, t1` (b) `li t0, 42` (c) `j etiket` (d) `not t0, t1` (e) `nop`

Alıştırma 3.27. ★★ $x = x * 32$; ifadesi için derleyici neden `mul` yerine kaydırma buyruğu üretir? Hangi 2'nin kuvvetleri için bu dönüşüm geçerlidir? RISC-V kodunu yazın.

Alıştırma 3.28. ★★ RISC-V'te `dizi[i]` ifadesine erişmek için 3 buyruk gerekirken x86'da tek buyrukla (`mov eax, [ebx+ecx*4]`) erişilebilir. Bu durumda RISC-V neden daha az adresleme kipi sunmayı tercih etmiştir? Donanım karmaşıklığı, saat sıklığı ve boru hattı tasarımı açısından tartışın.

Alıştırma 3.29. ★★ 32 bitlik `0xABCD1234` sabitini bir RISC-V yazmacına yüklemek için gereken buyruk dizisini yazın. `lui` ve `addi` buyruklarının her birinin yazmacı nasıl değiştirdiğini adım adım gösterin.

Alıştırma 3.30. ★★ Aşağıdaki RISC-V buyruk türleri için birer örnek buyruk yazın ve her birinin bit düzenini (opcode, funct3, funct7, rs1, rs2, rd, imm alanları) gösterin: (a) R tipi, (b) I tipi, (c) S tipi.

Alıştırma 3.31. ★★★ Aşağıda C dilinde yazılmış bir fonksiyonun RISC-V çevirici dilindeki karşılığını yazın. Yazmaç kullanımını açıkça belirtin.

```
int fib(int n) {
    if (n == 0 || n == 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}
```

Alıştırma 3.32. ★★★ Aşağıdaki RISC-V çevirici dil kodunun ne iş yaptığını açıklayın. x1 yazmacının başlangıç değeri N olduğuna göre, program sona erdiğinde x6 yazmacındaki değer nedir?

```
addi x2, x0, 0x8000
addi x4, x0, 1
addi x5, x0, 0
dongu:
ld    x8, 0(x2)
ld    x9, 8(x2)
add   x2, x0, x9
mul   x4, x4, x8
addi  x5, x5, 1
blt   x5, x1, dongu
add   x6, x0, x4
```

Alıştırma 3.33. ★★★ Döngünün açılması (*loop unrolling*) yöntemi hakkında aşağıdaki soruları yanıtlayın:

(a) Aşağıdaki döngüyü 4 kere açın ve açılmış halini yazın:

```
for (int i = 0; i < 15000000; i++) {
    b[i] = a[i];
    if (i % 2 == 0) c[i] = d[i];
}
```

(b) Açılmamış ve açılmış döngülerin durağan (*static*) ve devingen (*dynamic*) buyruk sayılarını karşılaştırın.

(c) Döngünün açılmasının buyruk önbelleği başarımına olası etkisini tartışın.

Alıştırma 3.34. ★★★ Aşağıdaki buyruk kümesi tablosu verilmiştir. Bu buyrukları kullanarak, belleğin $0x1000$ adresinden başlayan 64 elemanlı A dizisi ile $0x2000$ adresinden başlayan 64 elemanlı B dizisinin eleman eleman toplamından oluşan, $0x3000$ adresinden başlayan bir C dizisi oluşturan çevirici dil programını yazın. Her eleman 4 bayttır.

Buyruk	Açıklama
topla rt, rs, rd	$rt = rs + rd$
cikar rt, rs, rd	$rt = rs - rd$
mov rt, anlık	$rt \leftarrow \text{SıfırlaGenişlet}(\text{anlık})$
bg rt, rs, anlık	$rt > rs$ ise $PS = PS + \text{İşaretleGenişlet}(\text{anlık})$
lw rt, rs	$rt \leftarrow \text{Bellek}[rs]$
sw rt, rs	$\text{Bellek}[rt] \leftarrow rs$

Alıştırma 3.35. ★★★ Bir işlemci tasarımı projesi için buyruk kümesi tasarlanacaktır. Bu proje için tasarlanan mimarinin aşağıdaki özellikleri sağlaması beklenmektedir:

- 16 adet genel kullanıma ayrılmış yazmaç vardır.
- İlk yazmaç her zaman sıfırdır.
- 4 KB veri ve 4 KB buyruk belleği mevcuttur.
- Bayt adresleme kullanılmaktadır.
- Desteklenmesi istenen buyruklar:
 - **Topla:** İki yazmaç değerini toplar ve başka bir yazmaca yazar.
 - **Çıkar:** İlk kaynak yazmaçtan ikinci kaynak yazmacın değerini çıkarır ve belirtilen yazmaca yazar.
 - **ToplaVeYaz:** İki kaynak yazmacın değerini toplar ve ilk kaynak yazmaca yazar.
 - **ÇıkarVeYaz:** İlk kaynak yazmaçtan ikinci kaynak yazmacın değerini çıkarır ve ilk kaynak yazmaca yazar.
 - **Değiştir:** İki kaynak yazmacın değerlerini birbirleriyle değiştirir.
 - **Sakla:** Kaynak yazmaçtaki değeri anlık değerle verilen bellekteki adrese yazar.
 - **Yükle:** Bellekte anlık değerle adreslenen veriyi hedef yazmacına yazar.
 - **EşitseAtla:** İki kaynak yazmaçtaki değer birbirine eşitse program sayacının o anki değerine anlık değer eklenerek atlama gerçekleştirilir. Bu buyrukla en fazla 2^{20} bayt geri ya da $2^{20} - 1$ bayt ileri gidilebilir. (Anlık değer 2^7 'ye tümleyen biçimindedir.)

(a) Buyrukların sabit uzunlukta olduğu kabul edilirse:

- (i) Buyruk uzunluğu en az kaç bittir?
- (ii) Bu özellikleri sağlayan buyruklar kaç farklı buyruk tipinden oluşmaktadır? Her buyruk tipini gösterin.

(b) Buyruk uzunlukları sabit olmasaydı tasarımınızda ne değiştirdi?

(c) Bu tasarım tamamlandığında topla ve çıkar buyrukları için özel iyileştirmeler yapılmış ve BBC'leri düşürülmüştür. Yapılan benzetimler sonucu tüm buyrukların BBC'lerinin aşağıda verildiği gibi olduğu gözlemlenmiştir. Bu buyrukların oranlarının verildiği bir kodda derleme zamanında yapılacak iyileştirmelerle başarıyı artırmanız istenmektedir. Başarımı artırmak için neler yapabilirsiniz? Bu iyileştirmeler sonucunda başarımla başlangıca göre ne kadar artar?

	Topla	Çıkar	TopVeYaz	ÇıkVeYaz	Değiştir	Sakla	Yükle	EşitseAtla
BBC	1	1	3	3	5	10	10	4
%	5	5	20	20	20	5	5	20

(Yazmaç sayısının bu iyileştirmeler sırasında yeterli olacağını varsayabilirsiniz.)

(d) Bu buyruk kümesi mimarisini gerçekleyen işlemciyi çizip denetim tablosunu oluşturun.

Alıştırma 3.36. ★★★ Aşağıda bir C programının küçük bir parçası verilmiştir. Bu program için derleyici tarafından üretilmesini beklediğiniz RISC-V çevirici dil kodunu yazın. Kodun hedefi RV32-IM mimarisidir. `fonk` fonksiyonu bellekte `0x2000` adresine, `carp` fonksiyonu ise `0x1600` adresine yerleştirilmelidir. RISC-V çağrı alışkanlıkları (*calling convention*) kurallarına uygun bir çözüm yazın.

```
int carp(int a, int b, int c, int d, int e)
{
    return a * b * c * d * e;
}

int fonk(int x, int y)
{
    int z = carp(x, y, x, y, x);
    if (z < 150)
        return 0;
    else
        return 1;
}
```

- (a) Kodun neden belleğe saçılması gerekir? Kısa açıklayın.
- (b) Her iki fonksiyon için RISC-V çevirici dil kodunu yazın. Yığıt (*stack*) kullanımına ve çağırıcı/çağrılan (*caller/callee*) yazmaç saklama kurallarına dikkat edin.

RISC-V ile Programlama



Hattuşa Aslan Kapısı – Çorum

Hititler, 3700 yıl önce kil tabletlere çivi yazısıyla buyruklar kazıdı. Bugün programcılar da benzer bir iş yapar: buyruğu satır satır yazarak bilgisayara ne yapacağını anlatır.

Öğrenme Hedefleri

Bu bölümü tamamladığınızda aşağıdakileri yapabileceksiniz:

- RISC-V çevirici dilinde baştan sona çalışan programlar yazmak.
- Dizi, karakter dizisi ve yapı gibi veri yapılarını bellekte düzenleyip işlemek.
- C dilinde yazılmış bir programın RISC-V çevirici diline nasıl derlendiğini okuyup yorumlamak.
- Derleyicinin farklı eniyileme seviyelerinde ürettiği kodu karşılaştırmak.
- Bellek eşlemeli giriş/çıkış ile basit bir gömülü sistem programı yazmak.
- Döngü ve bellek erişim kalıplarını başarımlar açısından değerlendirmek.
- Çevirici dil programlarında hata ayıklama yapmak.

Giriş

Bir önceki bölümde RISC-V buyruk kümesini öğrendik: yazmaçları, aritmetik ve mantık buyruklarını, bellek erişim buyruklarını, dallanma ve atlama buyruklarını, fonksiyon çağrılarını ve yığıt yönetimini gördük. Ancak bu buyrukları tek tek tanımak, bunlarla gerçek bir program yazabilmek için yeterli değildir. Tıpkı bir dilin sözcüklerini bilmenin o dilde roman yazmaya yetmemesi gibi, buyrukları bilmek de programlamaya yetmez; sözcüklerden cümle kurmayı, cümlelerden paragraf oluşturmaya öğrenmek gerekir.

Bu bölümde, önceki bölümde öğrendiğimiz buyrukları kullanarak baştan sona çalışan programlar yazacağız. Amacımız yalnızca çevirici dil öğretmek değil, aynı zamanda bilgisayar mimarisini “içeriden” anlamaktır: bir C programının makine düzeyinde nasıl çalıştığını görmek, derleyicinin hangi kararları neden aldığını kavramak ve donanımın yazılımla nasıl etkileştiğini gözlemlemek. Bu bakış açısı, ileriki bölümlerde işlemci tasarımı, boru hattı ve bellek hiyerarşisi konularına geçtiğimizde güçlü bir temel oluşturacaktır.

Bölümün akışı kademeli bir zorluk sırası izler: önce en basit yazmaç ve bellek işlemleriyle başlayacağız, sonra program yapısını ve geliştirme ortamını tanıyacağız, ardından algoritmalar ve veri yapılarına geçeceğiz.

4.1 İlk Adımlar

Çevirici dil programlamaya en basit işlemle başlayalım: iki sayıyı toplamak. Aşağıdaki örnekler RARS benzeticisinde çalıştırılmak üzere tasarlanmıştır; her birini yazıp çalıştırmanız ve yazmaç değerlerini gözlemlemeniz önerilir. Her örnekte bir başarımlar çözümlemesi de yapacağız; böylece Bölüm 2’de öğrendiğimiz kavramları uygulamaya koymuş olacağız.

4.1.1 Yazmaçlarla Aritmetik**Örnek 4.1**

İki yazmaca sabit değerler yükleyip bunları toplayın. Sonucu ekrana yazdırın.

Çözüm:

```
1 .text
2 .globl main
```



```

3  main:
4      li    t0, 25          # t0 = 25
5      li    t1, 17          # t1 = 17
6      add   t2, t0, t1      # t2 = t0 + t1 = 42
7
8      # Sonucu ekrana yazdır
9      mv    a0, t2          # a0 = yazdırılacak değer
10     li    a7, 1           # 1 = tam sayı yazdır
11     ecall
12
13     # Programı sonlandır
14     li    a7, 10
15     ecall

```

Programın son dört satırında kullanılan `ecall` (*environment call*) buyruğu, programın işletim sisteminden bir hizmet istemesini sağlar; tıpkı bir restoranda müşterinin garsona sipariş vermesi gibi. Hangi hizmetin isteneceği `a7` yazmacına yazılan numarayla belirlenir: `a7=1` “`a0`’daki tam sayıyı ekrana yaz” demektir; `a7=10` ise “programı sonlandır” anlamına gelir. İşletim sistemi bu numaraya bakarak doğru hizmeti çalıştırır; bu mekanizmaya *sistem çağrısı* (*system call*) denir ve Bölüm 9’da ayrıntılı olarak ele alınacaktır. Bu programdaki `li` ve `mv` buyrukları aslında sözde buyruktur (*pseudo-instruction*); çevirici bunları gerçek RISC-V buyrukları ile değiştirir (Bölüm 3’teki Tablo 3.20’i hatırlayın). Aşağıdaki tablo bu dönüşümü somut bir program üzerinde gösterir:

Sözde buyruk	Gerçek buyruk	Açıklama
<code>li t0, 25</code>	<code>addi t0, x0, 25</code>	<code>x0</code> her zaman 0’dır
<code>li t1, 17</code>	<code>addi t1, x0, 17</code>	$0 + 17 = 17$
<code>add t2, t0, t1</code>	<code>add t2, t0, t1</code>	Zaten gerçek buyruk
<code>mv a0, t2</code>	<code>addi a0, t2, 0</code>	$t2 + 0 = t2$
<code>li a7, 1</code>	<code>addi a7, x0, 1</code>	Sistem çağrısı numarası
<code>ecall</code>	<code>ecall</code>	Zaten gerçek buyruk
<code>li a7, 10</code>	<code>addi a7, x0, 10</code>	Program sonlandırma
<code>ecall</code>	<code>ecall</code>	Zaten gerçek buyruk

Buyruk adlarının açılımları: `li` = *load immediate* (anlık değer yükle), `add` = *add* (topla), `addi` = *add immediate* (anlık değerle topla), `mv` = *move* (taşı), `ecall` = *environment call* (ortam çağrısı).

İşlemcinin gerçekte yürüttüğü buyruklar sağ sütundakilerdir. RARS’ta “Execute” sekmesinde “Basic” sütununa bakarsanız bu dönüşümü görebilirsiniz. Şimdi her buyruğun yazmaçları nasıl değiştirdiğini izleyelim:

#	Gerçek buyruk	t0	t1	t2	a0	a7
	(başlangıç)	0	0	0	0	0
1	addi t0, x0, 25	<u>25</u>	0	0	0	0
2	addi t1, x0, 17	25	<u>17</u>	0	0	0
3	add t2, t0, t1	25	17	<u>42</u>	0	0
4	addi a0, t2, 0	25	17	42	<u>42</u>	0
5	addi a7, x0, 1	25	17	42	42	<u>1</u>
6	ecall	25	17	42	42	1
7	addi a7, x0, 10	25	17	42	42	<u>10</u>
8	ecall	25	17	42	42	10

Altı çizili değerler o adımda değişen yazmaçlardır. Tablodan görüldüğü gibi her buyruk yalnızca *tek bir* hedef yazmacı değiştirir; bu, RISC mimarilerinin temel özelliğidir.

Başarım çözümlemesi: Program 8 gerçek buyruktan oluşur. Tümü yazmaç işlemi veya sistem çağrısıdır; bellek erişimi yoktur. Her buyruk tek çevrimde tamamlansa toplam $8 \times 1 = 8$ çevrimdir ve $BBÇ = 1,0$ olur. Bu, bir programın ulaşabileceği en iyi BBÇ değeridir: bellek erişimi olmayan, dallanma içermeyen düz bir kod akışı.

4.1.2 Bellekten Okuma ve Yazma

Örnek 4.2

Bellekteki bir değeri yazmaca yükleyin, üzerine bir sayı ekleyin ve sonucu belleğe geri yazın.

Çözüm:

```

1  .data
2  sayi:    .word 100          # bellekte saklanan değer
3  sonuc:   .word 0           # sonuç buraya yazılacak
4
5  .text
6  .globl main
7  main:
8      la    t0, sayi          # t0 = sayi'nın adresi
9      lw    t1, 0(t0)         # t1 = bellekteki değer (100)
10     addi  t1, t1, 50         # t1 = 100 + 50 = 150
11
12     la    t2, sonuc          # t2 = sonuc'un adresi
13     sw    t1, 0(t2)         # sonucu belleğe yaz
14
15     # Sonucu ekrana yazdır
16     mv    a0, t1             # yazdırılacak değer
17     li    a7, 1              # 1 = tam sayı yazdır
18     ecall
19
20     li    a7, 10             # 10 = programı sonlandır
21     ecall

```

Bu örnekte dört yeni buyruk kullanılmaktadır: *la* (*load address*, adres yükle) bir verinin bellekteki adresini yazmaca yükler; *lw* (*load word*, sözcük yükle) o adresteki 4 baytlık de-

ğeri yazmaca okur; *sw* (*store word*, sözcük sakla) yazmaçtaki değeri belleğe yazar; *auipc* (*add upper immediate to PC*, üst anlık değeri program sayacına ekle) ise *la*'nın gerçek buyruk karşılığının ilk yarısıdır. Sözde buyruk dönüşümüne bakalım; burada önemli bir fark vardır:

Sözde buyruk	Gerçek buyruk(lar)	Açıklama
<i>la t0, sayi</i>	<i>auipc t0, %hi(sayi)</i> <i>addi t0, t0, %lo(sayi)</i>	Üst 20 bit: PC + üst adres Alt 12 bit: tam adres
<i>lw t1, 0(t0)</i>	<i>lw t1, 0(t0)</i>	Zaten gerçek buyruk
<i>addi t1, t1, 50</i>	<i>addi t1, t1, 50</i>	Zaten gerçek buyruk
<i>la t2, sonuc</i>	<i>auipc t2, %hi(sonuc)</i> <i>addi t2, t2, %lo(sonuc)</i>	Üst 20 bit Alt 12 bit
<i>sw t1, 0(t2)</i>	<i>sw t1, 0(t2)</i>	Zaten gerçek buyruk
<i>mv a0, t1</i>	<i>addi a0, t1, 0</i>	<i>t1 + 0</i>
<i>li a7, 1</i>	<i>addi a7, x0, 1</i>	Sistem çağrısı
<i>ecall</i>	<i>ecall</i>	—
<i>li a7, 10</i>	<i>addi a7, x0, 10</i>	Program sonu
<i>ecall</i>	<i>ecall</i>	—

Dikkat: *la* sözde buyruğu *iki* gerçek buyruğa açılır çünkü 32 bitlik bir adres tek buyruğun 12 bitlik sabit alanına sığmaz. *auipc* üst 20 biti, *addi* alt 12 biti birleştirerek tam adresi oluşturur. Bu nedenle programdaki 10 sözde buyruk aslında 12 gerçek buyruğa dönüşür. Bu durum, Bölüm 2'de tanıttığımız üç farklı buyruk sayısını somut biçimde örnekler:

Buyruk türü	Sayı	Belirleyen
Programcının yazdığı (sözde buyruklar dahil)	10	Programcı
Durağan (bellekteki gerçek buyruklar)	12	Çevirici
Devingen (yürütülen toplam buyruk)	12	İşlemci

Bu programda döngü olmadığı için durağan ve devingen sayı birbirine eşittir. Ancak başarımlı hesaplamalarında her zaman durağan veya devingen buyruk sayısını kullanmalıyız; programcının yazdığı sözde buyruk sayısını değil.

Yazmaç ve bellek değişim tablosu (*sayi* adresi 0x10000000, *sonuc* adresi 0x10000004 varsayılmıştır):

#	Gerçek buyruk	t0	t1	t2	a0	Bellek
	(başlangıç)	0	0	0	0	[sayi]=100, [sonuc]=0
1-2	<i>auipc+addi (la t0, sayi)</i>	<u>0x...0000</u>	0	0	0	—
3	<i>lw t1, 0(t0)</i>	0x...0000	<u>100</u>	0	0	<i>okuma</i>
4	<i>addi t1, t1, 50</i>	0x...0000	<u>150</u>	0	0	—
5-6	<i>auipc+addi (la t2, sonuc)</i>	0x...0000	150	<u>0x...0004</u>	0	—
7	<i>sw t1, 0(t2)</i>	0x...0000	150	0x...0004	0	<u>[sonuc]=150</u>
8	<i>addi a0, t1, 0</i>	0x...0000	150	0x...0004	<u>150</u>	—

Bu tabloda Örnek 4.1'den farklı olarak “Bellek” sütunu eklendi. 3. adımda bellekten okuma, 7. adımda belleğe yazma yapılır. RARS'ta “Data Segment” sekmesini izleyerek bu değişimleri doğrulayın.

Başarım çözümlemesi: İşlemci 12 gerçek buyruk yürütür (sözde buyruk sayısı 10 olsa da). Bunların 2'si bellek erişimi (lw, sw), 10'u yazma işlemidir. Yazma buyrukları 1, bellek buyrukları 2 çevrimle: $10 \times 1 + 2 \times 2 = 14$ çevrim, $BBÇ = 14/12 \approx 1,17$. Sözde buyrukların gerçek buyruk sayısını artırabileceğine dikkat edin: buyruk sayarken her zaman gerçek buyrukları saymalıyız. Bellek erişim gecikmelerinin önbellek hiyerarşisiyle nasıl azaltıldığını Bölüm 8'de ayrıntılı olarak ele alacağız.

Bilgi Notu 4.1: auipc ve Program Sayacı

auipc (*add upper immediate to PC*) buyruğu, 20 bitlik sabiti 12 bit sola kaydırıp *o anki PS değerine* ekler. Burada kritik nokta şudur: PS, auipc buyruğunun kendisinin bellekteki adresini gösterir, bir sonraki buyruğun adresini değil. Bu sayede auipc + addi çifti, buyruğun bellekteki konumundan bağımsız olarak hedef adrese ulaşabilir (*konumdan bağımsız kod, position-independent code*).

Örnek: auipc t0, 0x12345 buyruğu 0x0000_2000 adresindeyse:

$$t0 = 0x0000\_2000 + (0x12345 \ll 12) = 0x0000\_2000 + 0x1234\_5000 = 0x1234\_7000$$

Ardından addi t0, t0, 0x678 ile alt 12 bit eklenerek tam adres oluşturulur. Bu mekanizma sayesinde derleyici ve bağlayıcı, programın belleğe hangi adrese yükleneceğini bilmeden doğru adresleme yapabilir.

4.1.3 Yığıt Kullanımı

Örnek 4.3

Yığıt (*stack*) kullanarak bir yazmacın değerini geçici olarak saklayın ve geri yükleyin.

Çözüm:

```

1  .text
2  .globl main
3  main:
4      li    s0, 7          # s0 = 7
5
6      # s0'ı yığıtta sakla
7      addi  sp, sp, -4      # yığıtta 4 bayt yer aç
8      sw    s0, 0(sp)      # s0'ı yığıtta yaz
9
10     # s0'ı başka bir iş için kullan
11     li    s0, 999         # s0 şimdi 999
12
13     # Eski değeri yığıttan geri yükle
14     lw    s0, 0(sp)       # s0 = 7 (eski değer geri geldi)
15     addi  sp, sp, 4       # yığıt alanını serbest bırak
16
17     # s0'ı yazdır (7 olmalı)
```

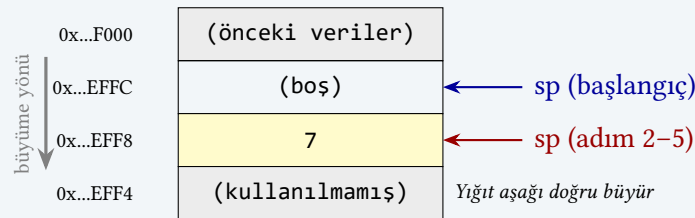
```

18  mv  a0, s0
19  li  a7, 1
20  ecall
21
22  li  a7, 10
23  ecall

```

Yığıt, belleğin yüksek adreslerinden düşük adreslere doğru büyür. Bu programda yeni bir yazmaç adı görüyoruz: *sp* (*stack pointer*, yani yığıt işaretçisi), yığıtın en üst konumunu gösterir. `addi sp, sp, -4` yığıt işaretçisini 4 bayt aşağı kaydırarak yer açar; `sw` bu alana değeri yazar; `lw` ile geri okuruz. Yazmaç ve yığıt değişimlerini adım adım izleyelim (`sp` başlangıç değeri `0x7FFFEFFC`):

#	Buyruk	s0	sp	Yığıt içeriği
	(başlangıç)	0	..EFFC	(boş)
1	<code>addi s0, x0, 7</code>	7	..EFFC	(boş)
2	<code>addi sp, sp, -4</code>	7	..EFF8	(yer açıldı, boş)
3	<code>sw s0, 0(sp)</code>	7	..EFF8	[..EFF8]=7
4	<code>addi s0, x0, 999</code>	999	..EFF8	[..EFF8]=7
5	<code>lw s0, 0(sp)</code>	7	..EFF8	[..EFF8]=7
6	<code>addi sp, sp, 4</code>	7	..EFFC	(serbest bırakıldı)



Şekil 4.1: Yığıt işaretçisinin yazmaç saklama sırasındaki değişimi. Sarı hücre `s0`'ın yığıtta saklanan değerini, oklar ise `sp`'nin hareket yönünü gösterir.

Bu tablodaki kritik an 4. ve 5. adımlar arasındır: `s0` geçici olarak 999 yapılmış olsa da yığıttaki eski değer (7) korunmaktadır. 5. adımda `lw` bu değeri geri yükler. Yığıt işaretçisinin 2. adımda düşüp 6. adımda eski yerine döndüğüne de dikkat edin; her `addi sp, sp, -N` için karşılık gelen bir `addi sp, sp, +N` bulunmalıdır; aksi takdirde yığıt taşması oluşur.

Başarım çözümlemesi: Yazdırma buyrukları dahil program 11 buyruk yürütür; 2'si bellek erişimi (`sw`, `lw`), 9'u yazmaç işlemidir. Yazmaç buyrukları 1, bellek buyrukları 2 çevrimle: $9 \times 1 + 2 \times 2 = 13$ çevrim, $BB\checkmark = 13/11 \approx 1,18$. Bir yazmacı saklamak ve geri yüklemek için 4 fazladan buyruk (2 `addi` + 1 `sw` + 1 `lw`) gerektiğine dikkat edin; fonksiyon çağrılarında birden çok yazmacın saklanması bu ek yükü katlayacaktır.

Yığıt İşlemleri: RISC-V, x86 ve ARM

RISC-V’te ayrı bir push/pop buyruğu yoktur; yığıt işaretçisini ayarlayıp sw/lw ile saklamak programcının sorumluluğundadır. Diğer mimariler bu konuda farklı tercihler yapmıştır:

İşlem	x86	RISC-V
Yığıta it	push eax (1 buyruk)	addi sp, sp, -4 (2 buyruk) sw s0, 0(sp)
Yığıttan çek	pop eax (1 buyruk)	lw s0, 0(sp) (2 buyruk) addi sp, sp, 4

ARM mimarisinde de push ve pop buyrukları bulunur; üstelik tek buyrukla birden çok yazmacı aynı anda saklayabilir: push {r4, r5, lr}.

Peki RISC-V neden özel yığıt buyrukları sunmaz? Çünkü push/pop, yalnızca addi + sw/lw birleşiminin kısaltmasıdır; ayrı bir buyruk olarak eklenmesi işlem kodu alanını tüketir ve çözücü donanımını karmaşıktırır. RISC-V “basit olanı doğru yap” felsefesine sadık kalarak bu görevi yazılıma bırakır. Bununla birlikte, RISC-V’in yeni Zcmp uzantısı cm.push ve cm.pop sözde buyrukları ekleyerek bu boşluğu gömülü sistemler için doldurmaktadır.

Örnek 4.4

Kullanıcıdan iki sayı alın, her birini yığıta itin (*push*), ardından yığıttan çıkararak (*pop*) toplayın ve sonucu yazdırın.

Çözüm:

```

1  .data
2  mesaj1: .string "Birinci sayı: "
3  mesaj2: .string "İkinci sayı: "
4
5  .text
6  .globl main
7  main:
8      # Birinci sayıyı oku
9      la a0, mesaj1
10     li a7, 4          # 4 = karakter dizisi yazdır
11     ecall
12     li a7, 5          # 5 = tam sayı oku
13     ecall
14     mv t0, a0         # t0 = birinci sayı
15
16     # Birinci sayıyı yığıta it
17     addi sp, sp, -4
18     sw t0, 0(sp)
19
20     # İkinci sayıyı oku
21     la a0, mesaj2
22     li a7, 4
23     ecall
24     li a7, 5
25     ecall
26     mv t1, a0         # t1 = ikinci sayı

```

```

27
28     # İkinci sayıyı yığıtta it
29     addi sp, sp, -4
30     sw    t1, 0(sp)
31
32     # Yığıtta iki değeri çıkar ve topla
33     lw    t1, 0(sp)          # t1 = ikinci sayı (son giren)
34     addi sp, sp, 4
35     lw    t0, 0(sp)          # t0 = birinci sayı
36     addi sp, sp, 4
37
38     add a0, t0, t1           # a0 = toplam
39     li    a7, 1
40     ecall
41
42     li    a7, 10
43     ecall

```

Yığıt LIFO (*Last In, First Out*, yani son giren ilk çıkar) düzeninde çalışır. İlk itilen değer altta, son itilen değer üstte kalır. RARS'ta bu programı adım adım yürütürken, yığıt alanında iki değer nasıl biriktiğini ve sonra nasıl geri alındığını gözlemleyin. Bu örüntü (değer sakla, başka işler yap, geri yükle) çevirici dil programlamanın temel yapı taşıdır ve fonksiyon çağrılarında yaygın biçimde kullanılır.

Başarım çözümlemesi: Buyrukları sayalım: $1a \times 2$, $li \times 6$, $mv \times 2$, $addi \times 4$, $sw \times 2$, $lw \times 2$, $add \times 1$, $ecall \times 6$: toplam 25 buyruk. 4'ü bellek erişimi ($sw \times 2$, $lw \times 2$), 21'i yazmaç veya sistem çağrısıdır. Yazmaç buyrukları 1, bellek buyrukları 2 çevrimle: $21 \times 1 + 4 \times 2 = 29$ çevrim, $BB\checkmark = 29/25 = 1,16$. Dört örnekteki BBÇ değerlerini yan yana koyalım: $1,00 \rightarrow 1,20 \rightarrow 1,18 \rightarrow 1,16$. Bellek erişim oranı düştükçe BBÇ 1,0'a yaklaşır; bu programda buyruk sayısı artmasına karşın bellek erişim oranı (%16) düşük kaldığından BBÇ nispeten iyidir. Bölüm 2'deki yürütme süresi formülünü hatırlayın: $T = N \times BB\checkmark \times T_{\text{çevrim}}$; BBÇ'yi düşürmek başarımla doğrudan artırır.

4.1.4 Döngüler

Önceki örneklerde tüm buyruklar yalnızca bir kez yürütülüyordu; bu nedenle durağan ve devingen buyruk sayıları birbirine eşitti. Döngü kullandığımızda bu durum değişir: aynı buyruklar defalarca yürütülür ve devingen buyruk sayısı durağan sayıdan çok daha büyük olabilir.

Örnek 4.5

1'den 10'a kadar tam sayıların toplamını ($1 + 2 + \dots + 10 = 55$) bir döngüyle hesaplayın.

Çözüm:

```

1  .text
2  .globl main
3  main:
4      li    t0, 0             # toplam = 0
5      li    t1, 1             # i = 1
6      li    t2, 10            # üst sınır
7  dongu:
8      add  t0, t0, t1          # toplam += i

```

```

9      addi t1, t1, 1          # i++
10     ble  t1, t2, dongu      # i <= 10 ise tekrarla
11
12     mv   a0, t0              # sonucu yazdır
13     li   a7, 1
14     ecall
15     li   a7, 10
16     ecall

```

Bu programda `ble` (*branch if less than or equal*, küçükse veya eşitse dallan) buyruğu, $t1 \leq t2$ koşulunu sağladığı sürece `dongu` etiketine geri döndürür. Yazmaç izleme tablosunun ilk birkaç yinelemesine bakalım:

#	Buyruk	t0	t1	t2	Dallanma
	(başlangıç)	0	0	0	—
1	<code>addi t0, x0, 0</code>	0	0	0	—
2	<code>addi t1, x0, 1</code>	0	1	0	—
3	<code>addi t2, x0, 10</code>	0	1	10	—
1. yineleme					
4	<code>add t0, t0, t1</code>	1	1	10	—
5	<code>addi t1, t1, 1</code>	1	2	10	—
6	<code>ble t1, t2, dongu</code>	1	2	10	$2 \leq 10$: dallan
2. yineleme					
7	<code>add t0, t0, t1</code>	3	2	10	—
8	<code>addi t1, t1, 1</code>	3	3	10	—
9	<code>ble t1, t2, dongu</code>	3	3	10	$3 \leq 10$: dallan
⋮ (10. yinelemede $t0 = 55$, $t1 = 11$)					
10. yineleme (son)					
31	<code>add t0, t0, t1</code>	55	10	10	—
32	<code>addi t1, t1, 1</code>	55	11	10	—
33	<code>ble t1, t2, dongu</code>	55	11	10	$11 \leq 10$: hayır

Bu örnekte durağan ve devingen buyruk sayıları arasındaki fark ilk kez belirgin biçimde ortaya çıkar:

Buyruk türü	Sayı	Hesaplama
Durağan (bellekteki buyruklar)	10	Döngü gövdesi yalnızca bir kez saklanır
Devingen (yürütülen toplam)	37	$3 + 3 \times 10 + 4 = 37$

Devingen sayının hesabı: 3 başlangıç buyruğu ($1i \times 3$) + döngü gövdesindeki 3 buyruk $\times$ 10 yineleme + 4 son buyruk (`mv`, `li`, `ecall` $\times 2$). Döngü gövdesindeki 3 buyruktan hiçbirisi bellek erişimi değildir (`ble` bir dallanma buyruğudur, bellek erişimi değil).

Başarım çözümlemesi: 37 devingen buyruktan hiçbirisi bellek erişimi değildir ($1w/sw$ yok); tümü yazmaç işlemi veya dallanmadır. Her buyruk tek çevrimde tamamlansa $BB\checkmark = 1,0$ olur. Dallanma buyrukları bazı durumlarda ek çevrimlere neden olabilir; bu konuyu Bölüm 7’de ayrıntılı olarak ele alacağız. Şimdilik her buyruğun tek çevrimde tamamlandığını varsayarsak $37 \times 1 = 37$ çevrim ve $BB\checkmark = 1,0$ elde ederiz. Önceki örneklerle

karşılaştırm: bellek erişimi olan programlarda BBÇ ~1,2'ye çıkıyordu; bu programda bellek erişimi olmadığından BBÇ en iyi değerinde kalmaktadır.

Örnek 4.6

Verilen iki sayının toplamını hesaplayan bir *altyordam* (subroutine) yazın. Ana program bu altyordamı çağırırsın ve sonucu ekrana yazdırırsın.

Çözüm:

```

1  .text
2  .globl main
3  main:
4      li  a0, 30          # birinci argüman
5      li  a1, 12          # ikinci argüman
6      jal  topla          # altyordamı çağır
7
8      # a0 artık dönüş değerini tutar (42)
9      li  a7, 1
10     ecall
11     li  a7, 10
12     ecall
13
14     # topla: a0 + a1 hesapla, sonucu a0'da döndür
15     topla:
16         add a0, a0, a1    # a0 = a0 + a1
17         ret              # çağırana geri dön

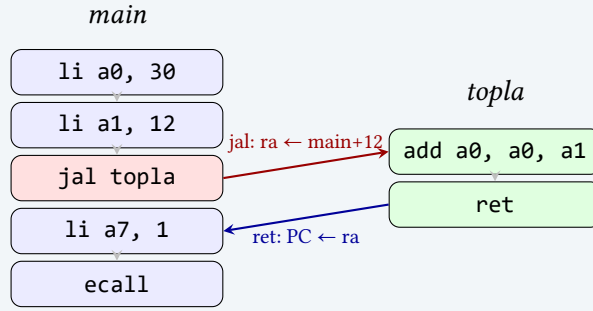
```

jal (*jump and link*, atla ve bağla) buyruğu iki iş yapar: (1) bir sonraki buyruğun adresini ra (*return address*, dönüş adresi) yazmacına kaydeder; (2) topla etiketine atlar. ret ise ra'daki adrese geri döner. Bu mekanizma sayesinde bir altyordam birden çok yerden çağrılabilir ve her seferinde doğru yere geri döner.

topla altyordamı *yaprak fonksiyondur* (leaf function): başka fonksiyon çağırır. Bu nedenle ra'yı yığıtta saklamasına gerek yoktur, çünkü kendi ra değerini bozacak bir jal çağırısı yapmaz. Yaprak olmayan fonksiyonlarda (bir fonksiyonun başka bir fonksiyonu çağırıldığı durumda) ra yığıtta saklanmalıdır; bu durumu Örnek 4.8'de göreceğiz.

Yazmaç izleme tablosu:

#	Buyruk	a0	a1	ra	PC
	(başlangıç)	0	0	0	main
1	addi a0, x0, 30	<u>30</u>	0	0	main+4
2	addi a1, x0, 12	30	<u>12</u>	0	main+8
3	jal topla	30	12	<u>main+12</u>	<u>topla</u>
(altyordamın içinde)					
4	add a0, a0, a1	<u>42</u>	12	main+12	topla+4
5	jalr x0, ra, 0 (ret)	42	12	main+12	<u>main+12</u>
(ana programa dönüş)					
6	addi a7, x0, 1	42	12	main+12	main+16
7	ecall	42	12	main+12	main+20



Şekil 4.2: Altyordam çağrısında kontrol akışı. jal buyruğu PC’yi altyordama yönlendirirken ra’ya dönüş adresini yazar; ret ise ra’daki adrese geri döner.

Tablodaki kritik satırlar 3. ve 5. adımlardır: jal PC’yi topla’ya yönlendirirken ra’ya dönüş adresini yazar; ret ise PC’yi ra’daki adrese geri getirir. PC sütunundaki gidiş-dönüş hareketini izlemek, altyordam mekanizmasını kavramanın en somut yoludur.

Başarım çözümlemesi: Program 9 buyruk yürütür (sözde buyruklar açıldığında). Tümü yazmaç işlemidir; bellek erişimi yoktur. jal ve ret buyrukları da birer çevrimde tamamlanır; altyordam çağırmanın kendisi ek maliyet getirmez. Asıl maliyet, yaprak olmayan fonksiyonlarda ra’yı ve saklanan yazmaçları yığıtta koruma zorunluluğundan doğar: Örnek 4.3’te gördüğümüz gibi her sw/lw çifti BBÇ’yi artırır.

Bilgi Notu 4.2: RARS’ta Adım Adım Yürütme

Yukarıdaki örnekleri anlamamanın en etkili yolu, RARS benzeticisinde adım adım (*single step*) çalıştırmaktır. Her buyruktan sonra üç şeyi kontrol edin: (1) hangi yazmaçların değeri değişti, (2) bellek içeriği nasıl etkilendi ve (3) program sayacı (PC) hangi buyruğa ilerledi. Dallanma buyrukları çalıştığında PC’nin sıradaki buyruğa değil, etiketin gösterdiği adrese atladığını gözlemlemek, kontrol akışını kavramanın en somut yoludur.

Bilgi Notu 4.3: Programlama ve Başarım İlişkisi

Bu bölümdeki her örnekte başarım çözümlemesi yaptık. Buyruk başına çevrim (BBÇ) değerini etkileyen iki temel etken gördük: (1) bellek erişimi: lw/sw buyrukları yazmaç işlemlerinden daha fazla çevrim alır; (2) dallanma buyrukları: bazı durumlarda ek çevrimlere neden olabilir (nedenini Bölüm 7’de göreceğiz). Bölüm 2’deki formülü hatırlayın:

$$T_{\text{yürütme}} = N_{\text{buyruk}} \times \text{BBÇ} \times T_{\text{çevrim}}$$

Bir programı hızlandırmanın üç yolu bu formülde gizlidir: buyruk sayısını azaltmak (daha iyi algoritma), BBÇ’yi düşürmek (bellek erişimini en aza indirmek) veya çevrim süresini kısaltmak (daha yüksek saat sıklığı). Programcı olarak ilk iki etken üzerinde doğrudan kontrolümüz vardır.

Terim Kökenleri

gömülü sistem (*embedded system*): Gömmek fiilinden *-ülü* ekiyle; genel amaçlı olmayan, belirli bir görevi yerine getirmek üzere donanımın içine “gömülmüş” bilgisayar sistemidir.

döngü açma (*loop unrolling*): Döngü gövdesini birden çok kez tekrarlayarak dallanma buyrukları sayısını azaltan bir eniyileme tekniğidir. “Açma” sözcüğü, katlanmış (döngülenmiş) kodun düz bir çizgiye açılmasını ifade eder.

özyineleme (*recursion*): Öz (self) ve yineleme (iteration) sözcüklerinin birleşimi; bir fonksiyonun kendisini çağırması anlamına gelir. Latince *recurrare* (geri koşturmak) kökünden gelen İngilizce terimin Türkçe karşılığıdır.

4.2 Program Yapısı ve Geliştirme Ortamı

4.2.1 Bir RISC-V Programının Anatomisi

Bir RISC-V çevirici dil programı, temelde üç bölümden oluşur: *veri bölümü* (.data), *metin bölümü* (.text) ve isteğe bağlı olarak *salt okunur veri bölümü* (.rodata). Metin bölümü programın buyruklarını, veri bölümü ise program tarafından kullanılan değişkenleri ve sabitleri barındırır.

```

1  .data                                # Veri bölümü
2  ileti: .asciz "Merhaba!\n"          # Karakter dizisi
3  sayi:  .word 42                     # 32 bitlik tam sayı
4  dizi:  .word 5, 3, 8, 1, 9          # 5 elemanlı dizi
5
6  .text                                # Metin (kod) bölümü
7  .globl main                         # Giriş noktasını dışa aç
8  main:
9      # Program burada başlar
10     la  a0, ileti                    # ileti adresini yükle
11     li  a7, 4                        # yazdırma sistem çağrısı
12     ecall                            # işletim sistemini çağır
13
14     li  a7, 10                       # çıkış sistem çağrısı
15     ecall                            # programı sonlandır

```

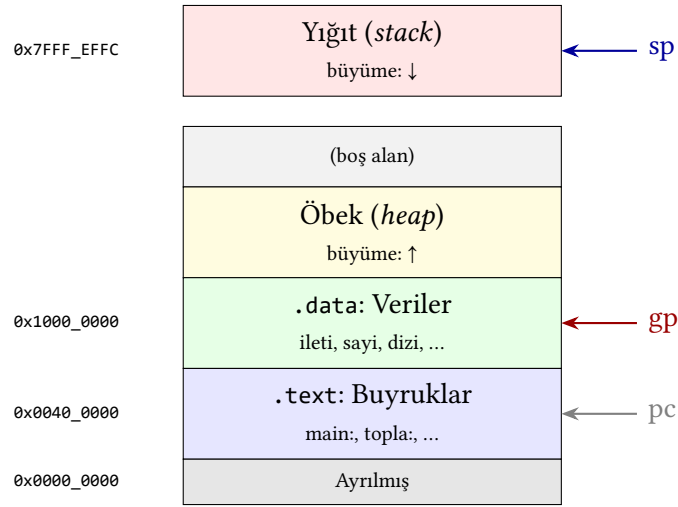
Bu programı adım adım inceleyelim. Program iki ana bölümden oluşur: .data ve .text. Veri bölümünde (.data) üç değişken tanımlanmıştır: *ilet*i adlı bir karakter dizisi, *sayi* adlı 32 bitlik bir tam sayı ve *dizi* adlı beş elemanlı bir dizi. Bu değişkenler belleğin veri kesiminde (*data segment*) yer alır.

Metin bölümünde (.text) programın buyrukları bulunur. *.globl main* yönergesi, *main* etiketinin bağlayıcı (*linker*) tarafından görülebilmesini sağlar; bu, programın giriş noktasıdır. İlk buyruk *la a0, ileti* ile karakter dizisinin bellek adresi *a0* yazmacına yüklenir. Ardından *li a7, 4* ile sistem çağrısı numarası belirlenir ve *ecall* buyruğuyla işletim sistemi çağrılarak karakter dizisi ekrana yazdırılır. Son iki satırda *a7* yazmacına 10 değeri yüklenerek programdan çıkış yapılır.

Veri bölümünde kullanılan yönergeler şunlardır:

Tablo 4.1: Veri tanımlama yönergeleri

Yönerge	Boyut	Açıklama
.byte	1 bayt	8 bitlik tam sayı
.half	2 bayt	16 bitlik tam sayı
.word	4 bayt	32 bitlik tam sayı
.dword	8 bayt	64 bitlik tam sayı (RV64)
.asciz	değişken	Sıfır ile sonlandırılmış karakter dizisi
.string	değişken	.asciz ile aynı
.space n	n bayt	n baytlık sıfırlanmış alan ayırma
.align n	–	Sonraki veriyi 2^n bayt sınırına hizalama



Şekil 4.3: RISC-V programının bellek düzeni. Yığıt yüksek adreslerden aşağıya doğru büyürken öbek düşük adreslerden yukarıya doğru büyür.

4.2.2 Geliştirme Ortamı

RISC-V çevirici dil programlarını denemek ve çalıştırmak için çeşitli araçlar kullanılabilir. Bu kitapta örnekler için RARS (*RISC-V Assembler and Runtime Simulator*) benzetimliği kullanılmıştır. RARS, Java tabanlı bir grafik arayüze sahiptir ve programları adım adım çalıştırma, yazmaç ve bellek içeriklerini izleme gibi olanaklar sunar. Tarayıcı tabanlı bir seçenek olarak Venus de kullanılabilir. Her iki aracın ayrıntılı kurulum ve kullanım bilgileri Ek D’de verilmiştir.

Bilgi Notu 4.4: Sistem Çağrılar

RARS benzetimliğinde `ecall` buyruğu, `a7` yazmacındaki değere göre farklı sistem çağrılarını gerçekleştirir: 1 = tam sayı yazdır, 4 = karakter dizisi yazdır, 5 = tam sayı oku, 10 = programı sonlandır, 11 = karakter yazdır. Gerçek bir RISC-V sisteminde bu çağrılar işletim sistemi çekirdeği tarafından işlenir ve numaralandırma farklıdır.

4.3 Algoritmalar ve Veri Yapıları

Temel yazmaç ve bellek işlemlerini öğrendikten sonra, şimdi bu becerileri gerçek problemlere uygulayalım. Bu bölümde dizilerde arama ve sıralama, karakter dizisi işleme ve yapılarla çalışma konularını ele alacağız.

4.3.1 Dizilerde Doğrusal Arama

Doğrusal arama, bir dizideki elemanları baştan sona tek tek kontrol ederek aranan değeri bulmaya çalışır. Basit ama öğretici bir örnektir: döngü, bellek erişimi, karşılaştırma ve dallanma buyruklarının hepsini bir arada kullanır.

Örnek 4.7

Bellekteki bir tam sayı dizisinde verilen bir değeri arayan ve bulursa dizin numarasını, bulamazsa -1 döndüren bir RISC-V programı yazın.

Çözüm: Önce bu algoritmayı C dilinde ifade edelim, ardından RISC-V çevirici diline dönüştürelim:

```
int dogrusal_ara(int dizi[], int boyut, int aranan) {
    for (int i = 0; i < boyut; i++) {
        if (dizi[i] == aranan)
            return i;
    }
    return -1;
}
```

Şimdi bu fonksiyonu RISC-V çevirici dilinde yazalım. Çağrı düzenine uygun olarak $a0$ = dizi adresi, $a1$ = boyut, $a2$ = aranan değer, dönüş değeri $a0$ 'da olacaktır:

```
1  # a0=dizi adresi, a1=boyut, a2=aranan
2  # Dönüş: a0 = bulunan dizin veya -1
3  dogrusal_ara:
4      li    t0, 0                # t0 = i = 0
5  dongu:
6      bge   t0, a1, bulunamadi   # i >= boyut ise çık
7      slli  t1, t0, 2            # t1 = i * 4 (bayt uzaklığı)
8      add   t1, a0, t1           # t1 = dizi + i*4
9      lw    t2, 0(t1)           # t2 = dizi[i]
10     beq   t2, a2, bulundu      # dizi[i] == aranan ise atla
11     addi  t0, t0, 1            # i++
12     j     dongu               # döngüye devam
13 bulundu:
14     mv    a0, t0              # dönüş değeri = i
15     ret
16 bulunamadi:
17     li    a0, -1              # dönüş değeri = -1
18     ret
```

Bu programda `slli t0, t0, 2` buyruğu dizin numarasını bayt uzaklığına çevirir. Her `.word` 4 bayt olduğu için dizin i , bellekte $i \times 4$ bayt ileridedir.

Başarım çözümlemesi: Döngü gövdesi 7 buyruktan oluşur (`bge`, `slli`, `add`, `lw`, `beq`, `addi`, `j`); bunların 1'i bellek erişimi (`lw`), 2'si dallanma buyruğudur. n elemanlı bir dizide aranan

değer bulunamazsa döngü n kez yinelenir: toplam $7n + 2$ buyruk (giriş ve çıkış dahil). Yazmaç buyrukları 1, bellek buyrukları 2, dallanma buyrukları 2 çevrimle bir yineleme $3 \times 1 + 1 \times 2 + 2 \times 2 + 1 \times 1 = 8$ çevrim alır. $n = 100$ için: toplam ~ 702 buyruk, ~ 802 çevrim, $BB\check{C} \approx 1,14$. Bellek erişiminin döngü başına yalnızca 1 kez yapılması $BB\check{C}$ 'yi düşük tutar, ancak arama başarısız olursa n 'e orantılı buyruk yürütülür. Bölüm 2'deki asimptotik karmaşıklık kavramını hatırlayın: bu algoritma $O(n)$ zamanda çalışır.

Doğrusal arama, bir dizide tek geçişle eleman bulmayı gösterdi. Peki ya diziyi önce sıralamamız gerekirse? Sıralı bir dizide ikili arama $O(\log n)$ zamanda çalışır; ancak sıralama işleminin kendisi en az $O(n \log n)$ zaman alır. Şimdi en basit sıralama algoritmalarından birini çevirici dilde yazarak iç içe döngülerin ve yer değiştirme (*swap*) işleminin nasıl gerçekleştirildiğini görelim.

4.3.2 Kabarcık Sıralama

Kabarcık sıralama, ardışık elemanları karşılaştırarak gerektiğinde yer değiştiren yalın bir sıralama algoritmasıdır. Başarımı $O(n^2)$ olsa da çevirici dilde yazması kolaydır ve bellek erişim kalıplarını anlamak için iyi bir örnektir.

Örnek 4.8

Bellekteki bir tam sayı dizisini küçükten büyüğe sıralayan bir RISC-V kabarcık sıralama programı yazın.

Çözüm: C dilindeki karşılığı:

```
void kabarcik_sirala(int dizi[], int boyut) {
    for (int i = 0; i < boyut - 1; i++)
        for (int j = 0; j < boyut - 1 - i; j++)
            if (dizi[j] > dizi[j+1]) {
                int gecici = dizi[j];
                dizi[j] = dizi[j+1];
                dizi[j+1] = gecici;
            }
}
```

RISC-V çevirici dili karşılığı ($a0$ = dizi adresi, $a1$ = boyut):

```
1  # kabarcik_sirala: a0=dizi adresi, a1=boyut
2  kabarcik_sirala:
3      addi sp, sp, -20          # yığıtta yer aç
4      sw ra, 16(sp)           # dönüş adresini sakla
5      sw s0, 12(sp)           # s0'ı sakla
6      sw s1, 8(sp)            # s1'i sakla
7      sw s2, 4(sp)            # s2'yi sakla
8      sw s3, 0(sp)            # s3'ü sakla
9      mv s0, a0                # s0 = dizi adresi
10     mv s1, a1                # s1 = boyut
11
12     li t0, 0                  # t0 = i = 0
13     addi t3, s1, -1           # t3 = boyut - 1
14     dis_dongu:
```

```

15     bge t0, t3, son           # i >= boyut-1 ise bitir
16     sub t4, t3, t0           # t4 = boyut - 1 - i
17     li t1, 0                 # t1 = j = 0
18 ic_dongu:
19     bge t1, t4, dis_devam     # j >= boyut-1-i ise dış döngüye
20     slli t5, t1, 2            # t5 = j * 4
21     add t5, s0, t5            # t5 = &dizi[j]
22     lw s2, 0(t5)              # s2 = diz[i]
23     lw s3, 4(t5)              # s3 = diz[i+1]
24     ble s2, s3, degistirme    # diz[i] <= diz[i+1] ise atla
25     sw s3, 0(t5)              # diz[i] = diz[i+1]
26     sw s2, 4(t5)              # diz[i+1] = diz[i]
27 degistirme:
28     addi t1, t1, 1            # j++
29     j ic_dongu
30 dis_devam:
31     addi t0, t0, 1            # i++
32     j dis_dongu
33 son:
34     lw s3, 0(sp)              # s3'ü geri yükle
35     lw s2, 4(sp)              # s2'yi geri yükle
36     lw s1, 8(sp)              # s1'i geri yükle
37     lw s0, 12(sp)             # s0'ı geri yükle
38     lw ra, 16(sp)             # dönüş adresini geri yükle
39     addi sp, sp, 20           # yığıtı geri al
40     ret

```

Bu programda dikkat çekilmesi gereken noktalar:

- Fonksiyon giriş kodunda (*prologue*) saklanan yazmaçlar (s0–s3) ve dönüş adresi (ra) yığta kaydedilir; çıkış kodunda (*epilogue*) geri yüklenir. Bu, Bölüm 3'te gördüğümüz çağrı düzeninin doğrudan uygulamasıdır.
- Dizi erişiminde slli ile dizin bayt uzaklığına çevrilir, ardından lw/sw ile bellek okunur/yazılır.
- İç döngüde iki ardışık eleman (0(t5) ve 4(t5)) okunarak karşılaştırılır. Yer değiştirme için üçüncü bir değişken (fazladan yazmaç) kullanılır.

Başarım çözümlemesi: İç döngü gövdesi 9 buyruktan oluşur; bunların 2'si okuma (lw), yer değiştirme durumunda 2'si yazma (sw) olmak üzere 4 bellek erişimi içerir. n elemanlı bir dizi için iç döngü toplam $n(n-1)/2$ kez yinelenir. $n = 10$ için 45 yineme, her biri ~13 çevrim (en kötü durumda): toplam ~585 çevrim + dış döngü ek yükü. BBÇ $\approx 1,4$ civarındadır; bellek erişiminin yoğun olduğu bu algorithmada BBÇ doğrusal aramaya göre belirgin biçimde yükselir. Ayrıca $O(n^2)$ karmaşıklık, n büyüdükçe buyruk sayısının karesel artması demektir: $n = 100$ için ~45.000 buyruk, $n = 1000$ için ~4,5 milyon. Sıralama algoritmalarında verimlilik seçimi başarıyı temelden etkiler.

Dizi üzerinde arama ve sıralama yaparken lw/sw ile 4 baytlık sözcükler üzerinde çalıştık. Ancak metin işleme, bayt bayt erişim gerektirir: her ASCII karakter 1 bayttır ve diziler sıfır (0x00) ile sonlandırılır. Şimdi lb/sb buyrukları ile karakter dizisi işleme fonksiyonları yazacağız.

4.3.3 Karakter Dizisi İşleme

Karakter dizileri (*string*), kısaca *dizgi* de denir; bellekte ardışık baytlar olarak saklanır ve sıfır (0x00) ile sonlandırılır. Bu tür verilere erişim, *lb* (*load byte*, bayt yükle) ve *sb* (*store byte*, bayt sakla) buyruklarıyla yapılır.

Örnek 4.9

Bir karakter dizisinin uzunluğunu hesaplayan *strlen* fonksiyonunu RISC-V çevirici dilinde yazın.

Çözüm:

```

1  # strlen: a0 = karakter dizisi adresi
2  # Dönüş: a0 = uzunluk
3  strlen:
4      mv    t0, a0          # t0 = geçici işaretçi
5      li    t1, 0           # t1 = sayaç = 0
6  strlen_dongu:
7      lb    t2, 0(t0)       # t2 = geçerli karakter
8      beqz  t2, strlen_son   # sıfır ise bitir
9      addi  t0, t0, 1        # işaretçiyi ilerlet
10     addi  t1, t1, 1        # sayacı artır
11     j     strlen_dongu
12  strlen_son:
13     mv    a0, t1           # dönüş değeri = uzunluk
14     ret

```

Burada *lb* buyruğu 1 baytlık veri yükler (işaretle genişleterek); *lb*u ise işaretsiz genişletme yapar. ASCII karakterler 0–127 aralığında olduğu için her ikisi de aynı sonucu verir.

Örnek 4.10

Bir karakter dizisini başka bir bellek alanına kopyalayan *strcpy* fonksiyonunu yazın.

Çözüm:

```

1  # strcpy: a0 = hedef adres, a1 = kaynak adres
2  strcpy:
3      mv    t0, a0          # t0 = hedef işaretçi
4      mv    t1, a1          # t1 = kaynak işaretçi
5  strcpy_dongu:
6      lb    t2, 0(t1)       # kaynaktan bir bayt oku
7      sb    t2, 0(t0)       # hedefe yaz
8      beqz  t2, strcpy_son   # sıfır karakterse bitir
9      addi  t0, t0, 1        # hedefi ilerlet
10     addi  t1, t1, 1        # kaynağı ilerlet
11     j     strcpy_dongu
12  strcpy_son:
13     ret                   # a0 hâlâ hedef adresi gösterir

```

Dikkat edilmesi gereken nokta: sıfır karakteri de hedefe kopyalanır; bu, hedef dizinin düzgün biçimde sonlandırılmasını sağlar. Kopyalama sonrasında *a0* hâlâ hedef dizinin başlangıç adresini gösterir; bu, C standardındaki *strcpy* davranışıyla uyumludur.

Örnek 4.11

Palindrom (çevirmece), soldan sağa ve sağdan sola aynı okunan dizgidir. Bir karakter dizisinin palindrom olup olmadığını belirleyen bir program yazın. Boşluk karakterlerini yok sayın. Giriş dizisi: "ey edip adanada pide ye".

Çözüm: Algoritma iki işaretçi kullanır: biri dizinin başından ilerler, diğeri sonundan geriler. Her adımda boşluklar atlanır ve karşılık gelen karakterler karşılaştırılır. İşaretçiler ortada buluşursa dizi palindromdur.

```

1  .data
2  dizgi:  .asciz "ey edip adanada pide ye"
3  evet:   .asciz "Palindrom!\n"
4  hayir:  .asciz "Palindrom degil.\n"
5
6  .text
7  .globl main
8  main:
9      la    a0, dizgi
10     jal   palindrom_mu      # a0 = 1 veya 0
11
12     bnez  a0, yaz_etevet
13     la    a0, hayir
14     j     yaz
15 yaz_etevet:
16     la    a0, evet
17 yaz:
18     li    a7, 4
19     ecall
20     li    a7, 10
21     ecall
22
23 # palindrom_mu: a0 = dizgi adresi
24 # Dönüş: a0 = 1 (palindrom), a0 = 0 (değil)
25 palindrom_mu:
26     mv     t0, a0           # t0 = sol işaretçi
27
28     # Dizgi sonunu bul
29     mv     t1, a0
30 son_bul:
31     lb     t2, 0(t1)
32     beqz   t2, son_bulundu
33     addi   t1, t1, 1
34     j      son_bul
35 son_bulundu:
36     addi   t1, t1, -1       # t1 = son karakter adresi
37
38 karsilastir:
39     bge    t0, t1, evet_palindrom
40
41     # Soldan boşlukları atla
42 sol_bosluk:
43     lb     t2, 0(t0)
44     li     t3, 32           # boşluk = ASCII 32
45     bne    t2, t3, sag_bosluk

```

```

46     addi t0, t0, 1
47     j     sol_bosluk
48
49     # Sağdan boşlukları atla
50 sag_bosluk:
51     lb  t3, 0(t1)
52     li  t4, 32
53     bne t3, t4, harf_karsilastir
54     addi t1, t1, -1
55     j     sag_bosluk
56
57 harf_karsilastir:
58     lb  t2, 0(t0)           # sol karakter
59     lb  t3, 0(t1)           # sağ karakter
60     bne t2, t3, hayir_palindrom
61     addi t0, t0, 1           # solu ilerlet
62     addi t1, t1, -1           # sağı gerilet
63     j     karsilastir
64
65 evet_palindrom:
66     li  a0, 1
67     ret
68 hayir_palindrom:
69     li  a0, 0
70     ret

```

Bu program, önceki örneklerde öğrendiğimiz kavramları bir araya getirir:

- `lb` (*load byte*, bayt yükle) ile karakter karakter okuma (Örnek 4.9'daki `str1en` gibi).
- İki işaretçi kullanarak dizi tarama (doğrusal aramanın iki yönlü versiyonu).
- Boşluk atlama döngüleri (iç içe döngü yapısı).
- `jal/ret` ile altıyordam çağırma (Örnek 4.6).

Başarım çözümlemesi: “ey edip adanada pide ye” dizgisi 23 karakter içerir (boşluklar dahil). Dizgi sonunu bulmak 23 `lb` buyruğu gerektirir. Palindrom kontrolünde her adımda 2 `lb` (sol ve sağ karakter) ve olası boşluk atlama `lb`'leri yürütülür. Toplam ~70 devingen buyruk yürütülür ve bunların ~40'ı bellek erişimidir (1b). Bu, bellek erişim oranının %57'ye ulaştığı anlamına gelir; önceki örneklerimizde bu oran en fazla %16 idi. Her 1b 2 çevrim alırsa $BBÇ \approx (30 \times 1 + 40 \times 2)/70 \approx 1,57$ olur. Karakter dizisi işleme, doğası gereği yoğun bellek erişimi gerektirir; bu nedenle ön bellek başarımı (Bölüm 8) bu tür programlar için kritik öneme sahiptir.

Şimdiye kadar diziler ve karakter dizileri ile çalıştık; her ikisinde de elemanlar aynı türdendir. Ancak gerçek programlarda farklı türdeki verileri bir arada tutmak gerekir: bir öğrenenin adı (karakter dizisi), numarası (tam sayı) ve notu (kayan nokta) gibi. C dilindeki yapılar (`struct`) bu ihtiyacı karşılar; çevirici dilde ise bu, bellekteki uzaklıkları elle hesaplamak anlamına gelir.

4.3.4 Yapılar ve Bellek Düzeni

C dilindeki yapılar (`struct`), farklı türdeki verileri bir arada tutan bileşik veri türleridir. Çevirici dilde yapılarla çalışmak, bellekteki veri düzenini ve hizalama kurallarını anlamayı gerektirir.

Örnek 4.12

Aşağıdaki C yapısının bellekteki düzenini gösterin ve alanlarına erişen RISC-V kodu yazın:

```
struct Ogrenci {
    int numara;        // 4 bayt, uzaklık 0
    int not_ort;       // 4 bayt, uzaklık 4
    char sinif;        // 1 bayt, uzaklık 8
};
```

Çözüm: Bu yapı bellekte ardışık olarak yerleştirilir. Derleyici hizalama dolgusu ekleyebilir, ancak burada alanlar zaten doğal sınırlarına hizalıdır:

numara (4 bayt)				not_ort (4 bayt)				sinif	dolgu		
+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11

```
1 # a0 = Ogrenci yapısının adresi
2 # Numara alanını oku
3 lw t0, 0(a0)          # t0 = ogrenci.numara
4
5 # Not ortalamasını oku
6 lw t1, 4(a0)          # t1 = ogrenci.not_ort
7
8 # Sınıf alanını oku
9 lb t2, 8(a0)          # t2 = ogrenci.sinif
10
11 # Not ortalamasını güncelle
12 li t3, 85
13 sw t3, 4(a0)          # ogrenci.not_ort = 85
```

Yapı alanlarına erişimde anahtar nokta, her alanın yapı başlangıcından itibaren sabit bir uzaklıkta (*offset*) bulunmasıdır. Derleyici bu uzaklıkları derleme zamanında hesaplar.

4.3.5 Bağlı Liste

Bağlı listeler, her düğümün bir veri ve bir sonraki düğümün adresini tutan işaretçi içerdiği dinamik veri yapılarıdır. Çevirici dilde bağlı liste üzerinde gezinmek, işaretçi aritmetiğini anlamak için değerli bir alıştırmadır.

Örnek 4.13

Her düğümü {int deger, dugum* sonraki} biçiminde olan bir bağlı listedeki elemanların toplamını hesaplayan fonksiyonu RISC-V çevirici dilinde yazın.

Çözüm: Her düğüm 8 bayttır: ilk 4 bayt deger, sonraki 4 bayt sonraki işaretçisi. Son düğümün sonraki alanı 0'dır (NULL).

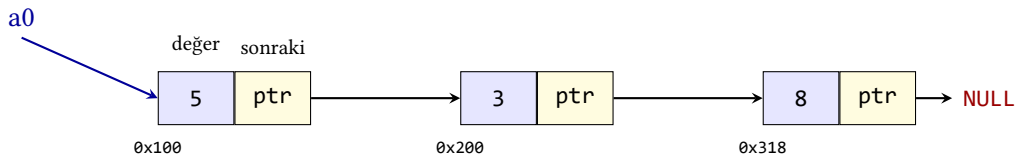
```
1 # liste_toplam: a0 = ilk düğümün adresi
2 # Dönüş: a0 = tüm değerlerin toplamı
3 liste_toplam:
4     li t0, 0          # t0 = toplam = 0
5     liste_dongu:
```

```

6   beqz a0, liste_son      # düğüm NULL ise bitir
7   lw  t1, 0(a0)          # t1 = düğüm.değer
8   add t0, t0, t1         # toplam += değer
9   lw  a0, 4(a0)          # a0 = düğüm.sonraki
10  j   liste_dongu
11  liste_son:
12  mv  a0, t0              # dönüş değeri = toplam
13  ret

```

Bu programda her yinelemede iki `lw` buyruğu kullanılır: biri düğümün değerini okur, diğeri bir sonraki düğümün adresini okur. `lw a0, 4(a0)` satırı, işaretçinin kendisini günceller; bu, C'deki `p = p->sonraki` ifadesinin doğrudan karşılığıdır.



Her düğüm 8 bayt: `lw t1, 0(a0)` değeri, `lw a0, 4(a0)` sonraki adresi okur. Düğümler bellekte ardışık olmak zorunda değildir.

Şekil 4.4: Bağlı liste veri yapısının bellek görünümü. Her düğüm bir değer ve sonraki düğümün adresini içerir; son düğümün işaretçisi sıfırdır.

Bilgi Notu 4.5: Yığın ve Yığıt: İki Farklı Kavram

Türkçede “yığın” sözcüğü hem *heap* (dinamik bellek havuzu) hem de *stack* (yığıt) için kullanılabilir. Bu kitapta karışıklığı önlemek için *stack* karşılığı olarak “yığıt”, *heap* karşılığı olarak “yığın” kullanılmaktadır. Bağlı liste düğümleri uygulamada genellikle yığın (*heap*) üzerinde dinamik olarak oluşturulur.

4.3.6 Özyineleme

Önceki örneklerdeki tüm fonksiyonlar yinelemeli (*iterative*) idi: döngüler kullanarak çalışıyorlardı. Ancak bazı problemler, fonksiyonun kendisini çağırmasıyla daha doğal biçimde ifade edilebilir. Bu tekniğe *özyineleme* (*recursion*) denir. Bölüm 3’te yığıt çerçevesinin nasıl oluşturulduğunu gördük; özyinelemede her çağrı yeni bir yığıt çerçevesi oluşturur. Bu nedenle özyinelemeli fonksiyonlar, yığıt yönetimini anlamak için ideal bir çalışma alanıdır.

Örnek 4.14

n ’inci Fibonacci sayısını özyinelemeli olarak hesaplayan bir RISC-V fonksiyonu yazın. $F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$.

Çözüm: Önce C dilindeki karşılığı:

```

int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

```

```
}

```

RISC-V çevirici dilinde ($a0 = n$, dönüş değeri $a0$ 'da):

```

1  # fibonacci: a0 = n
2  # Dönüş: a0 = F(n)
3  fibonacci:
4      # Taban durumu: n <= 1 ise n döndür
5      li    t0, 1
6      ble   a0, t0, fib_taban
7
8      # Yığıt çerçevesi: ra, s0, s1 sakla
9      addi  sp, sp, -12
10     sw    ra, 8(sp)
11     sw    s0, 4(sp)
12     sw    s1, 0(sp)
13
14     mv     s0, a0          # s0 = n
15     addi   a0, s0, -1      # a0 = n - 1
16     jal    fibonacci      # F(n-1) hesapla
17     mv     s1, a0          # s1 = F(n-1)
18
19     addi   a0, s0, -2      # a0 = n - 2
20     jal    fibonacci      # F(n-2) hesapla
21     add    a0, s1, a0      # a0 = F(n-1) + F(n-2)
22
23     # Yığıt çerçevesini geri yükle
24     lw     s1, 0(sp)
25     lw     s0, 4(sp)
26     lw     ra, 8(sp)
27     addi   sp, sp, 12
28     ret
29
30 fib_taban:
31     ret                    # a0 zaten n (0 veya 1)

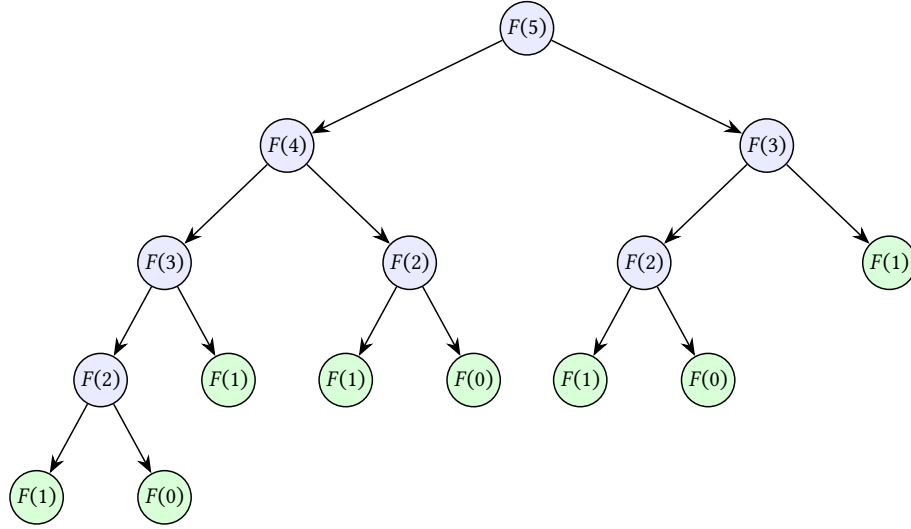
```

Bu programda dikkat edilmesi gereken noktalar:

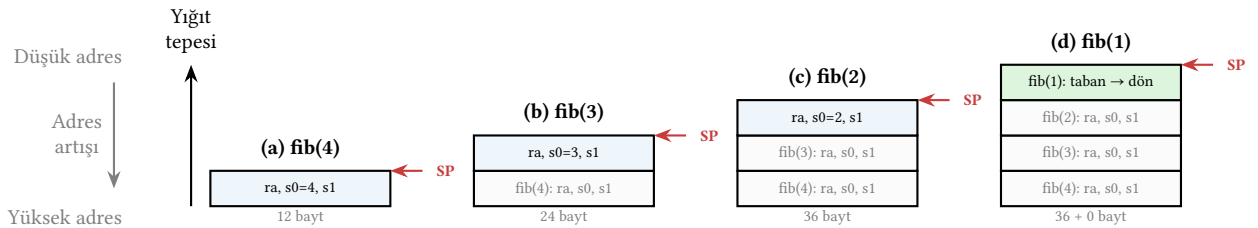
- Her özyinelemeli çağrı öncesinde ra , $s0$ ve $s1$ yığıtta saklanır. Saklanan yazmaçlar ($s0-s11$) çağrılan tarafın sorumluluğundadır; fonksiyon bu yazmaçları bozmamalıdır.
- İlk `jal fibonacci` çağrısı $a0$ 'daki sonucu döndürür. Bu sonuç $s1$ 'e kopyalanır çünkü ikinci çağrı $a0$ 'yı değiştirecektir.
- Taban durumunda yığıt çerçevesi oluşturulmaz; bu, gereksiz bellek erişimini önler.
- $n = 5$ için bu fonksiyon kendini 15 kez çağırır ve yığıtta en fazla 5 çerçeve (60 bayt) birikir. n büyüdükçe çağrı sayısı üstel olarak artar; bu nedenle büyük n değerleri için yinelemeli sürüm tercih edilmelidir.

Başarım çözümlemesi: Her özyinelemeli çağrıda 16 buyruk yürütülür: 4 yığıta yazma ($sw \times 3 + addi$), 4 yığıttan okuma ($lw \times 3 + addi$), 2 fonksiyon çağrısı (`jal`), ve 6 yazmaç işlemi. $n = 5$ için toplam $15 \text{ çağrı} \times 16 \text{ buyruk} = 240 \text{ buyruk}$ yürütülür; bellek buyrukları 2 çevrim alırsa toplam $\sim 336 \text{ çevrim}$, $BB\checkmark \approx 1,40$. Asıl sorun $BB\checkmark$ değil, buyruk sayısının

üstel artışıdır: $n = 10$ için ~ 177 çağrı ≈ 2832 buyruk, $n = 20$ için ~ 21.891 çağrı ≈ 350.000 buyruk. Yinelemeli sürüm ise $n = 20$ için yalnızca ~ 120 buyrukla aynı sonucu verir; algoritma seçimi, BBC'den çok daha büyük bir başarımlı etkisi yaratır.



Şekil 4.5: Özyinelemeli Fibonacci çağrı ağacı ($n = 5$). Yeşil düğümler taban durumlarıdır; toplam 15 çağrı üretilir ve her çağrı yığıtta 12 baytlık bir çerçeve oluşturur.



Şekil 4.6: Özyinelemeli Fibonacci çağrısında yığıt çerçevelerinin birikmesi ($n = 4$). Her özyinelemeli çağrı yığıttın tepesine 12 baytlık bir çerçeve (ra, s0, s1) ekler. En derin noktada (d) üç çerçeve birikmiştir; taban durumu (fib(1)) çerçeve oluşturmadan döner. Yığıt derinliği $n - 1$ 'e ulaşır.

Bilgi Notu 4.6: Kuyruk Özyineleme Eniyilemesi

Bir fonksiyonun son işlemi özyinelemeli çağrı ise buna *kuyruk özyineleme* (*tail recursion*) denir. Derleyici bu durumu algıladığında özyinelemeyi döngüye dönüştürebilir ve yığıt çerçevesi oluşturma gereğini ortadan kaldırır. Yukarıdaki Fibonacci fonksiyonu kuyruk özyinelemeli *değildir* çünkü son işlem add (toplama) buyruğudur, çağrı değil. Ancak bir biriktirici parametresi eklenerek kuyruk özyinelemeli biçime dönüştürülebilir.

4.3.7 Çok Fonksiyonlu Programlar

Fibonacci örneğinde bir fonksiyonun kendisini çağırdığını gördük. Gerçek programlarda ise farklı fonksiyonlar birbirini çağırarak bir iş akışı oluşturur. Bu durumda Örnek 4.6'da öğrendi-

ğimiz yaprak/yaprak olmayan ayrımı kritik hale gelir: hangi fonksiyon ra'yı saklamalı, hangi yazmaçlar korunmalı? Şimdi bu soruları somut bir örnekle yanıtlayalım.

Örnek 4.15

Bellekteki bir dizinin en büyük ve en küçük elemanını bulan, ardından bunların farkını hesaplayan bir program yazın. Program üç fonksiyondan oluşsun: en_buyuk, en_kucuk ve main.

Çözüm:

```

1  .data
2  dizi:    .word 12, 5, 23, 7, 18, 3, 15
3  boyut:   .word 7
4  sonuc:   .word 0
5
6  .text
7  .globl main
8  main:
9      addi sp, sp, -4
10     sw    ra, 0(sp)
11
12     la    a0, dizi
13     lw    a1, boyut          # a1 = 7
14     jal   en_buyuk          # a0 = en büyük
15     mv    s0, a0            # s0 = en büyük
16
17     la    a0, dizi
18     lw    a1, boyut
19     jal   en_kucuk          # a0 = en küçük
20
21     sub    t0, s0, a0        # t0 = en büyük - en küçük
22     la    t1, sonuc
23     sw    t0, 0(t1)         # sonucu belleğe yaz
24
25     lw    ra, 0(sp)
26     addi sp, sp, 4
27     li    a7, 10
28     ecall
29
30     # en_buyuk: a0=dizi adresi, a1=boyut
31     # Dönüş: a0 = en büyük eleman
32     en_buyuk:
33         lw    t0, 0(a0)      # t0 = dizi[0] (geçici en büyük)
34         li    t1, 1          # t1 = i = 1
35     eb_dongu:
36         bge    t1, a1, eb_son
37         slli   t2, t1, 2
38         add    t2, a0, t2
39         lw    t3, 0(t2)      # t3 = dizi[i]
40         ble    t3, t0, eb_atla # dizi[i] <= en büyük ise atla
41         mv    t0, t3         # yeni en büyük
42     eb_atla:
43         addi   t1, t1, 1
44         j      eb_dongu

```

```

45 eb_son:
46     mv  a0, t0
47     ret
48
49 # en_kucuk: a0=dizi adresi, a1=boyut
50 # Dönüş: a0 = en küçük eleman
51 en_kucuk:
52     lw  t0, 0(a0)          # t0 = dizi[0] (geçici en küçük)
53     li  t1, 1              # t1 = i = 1
54 ek_dongu:
55     bge t1, a1, ek_son
56     slli t2, t1, 2
57     add t2, a0, t2
58     lw  t3, 0(t2)          # t3 = dizi[i]
59     bge t3, t0, ek_atla    # dizi[i] >= en küçük ise atla
60     mv  t0, t3             # yeni en küçük
61 ek_atla:
62     addi t1, t1, 1
63     j    ek_dongu
64 ek_son:
65     mv  a0, t0
66     ret

```

Bu programda main fonksiyonu sırasıyla en_buyuk ve en_kucuk fonksiyonlarını çağırır. Dikkat edilmesi gereken noktalar:

- main fonksiyonu başka fonksiyonları çağırdığı için ra'yı yığıtta saklar. Eğer main yaprak fonksiyon olsaydı (başka fonksiyon çağırmayan) bu gerekli olmazdı.
- en_buyuk fonksiyonunun sonucu s0'a kopyalanır. Bu bir saklanan yazmaç olduğu için en_kucuk çağrısı sırasında korunacaktır. Geçici yazmaç (t0–t6) kullanılsaydı, en_kucuk fonksiyonu bu değeri bozabilirdi.
- Her iki alt fonksiyon yalnızca geçici yazmaçlar kullandığı ve başka fonksiyon çağırmadığı için (yaprak fonksiyonlar) yığıt çerçevesi oluşturmazlar.

Başarım çözümlemesi: Bu programda main 12 buyruk, en_buyuk ve en_kucuk ise her biri $6n + 2$ buyruk yürütür (n = dizi boyutu). $n = 7$ için toplam: $12 + 2 \times (6 \times 7 + 2) = 100$ buyruk. Her döngü gövdesinde 1 bellek erişimi (lw) ve 1 dallanma vardır; main'de ise 2 bellek erişimi (sw/lw yığıt) bulunur. Bellek buyrukları 2 çevrim alırsa: yazmaç buyrukları ~72, bellek buyrukları ~16, dallanma buyrukları ~12 ile toplam $72 + 32 + 24 = 128$ çevrim, BBÇ $\approx 1,28$. Dikkat edilmesi gereken bir nokta: yaprak fonksiyonlarda yığıt çerçevesi oluşturulmadığı için bellek erişimi azalır ve BBÇ düşer. Yaprak olmayan main ise ra saklama/yükleme nedeniyle fazladan 2 bellek erişimi gerektirir.

Tasarım İlkesi 3: Olağan durumu hızlandır (devam)

Derleyicilerin uyguladığı eniyilemeler bu ilkenin doğrudan yansımasıdır: en sık yürütülen kod yolları (döngü gövdeleri, sık çağrılan fonksiyonlar) için daha az buyruk üreten ve daha az bellek erişimi gerektiren kod üretilir. Programcı da aynı ilkeyi çevirici dil düzeyinde uygulayabilir: seyrek çalışan hata kontrolü kodunu döngü dışına taşımak, sık erişilen verileri yazmaçlarda tutmak gibi.

4.4 C'den RISC-V'e: Derleme Çıktısı Okuma

Bir bilgisayar mimarı ya da sistem programcısı için en değerli becerilerden biri, derleyicinin ürettiği çevirici dil kodunu okuyabilmektir. Bu bölümde, GCC derleyicisinin C kodundan ürettiği RISC-V çevirici dil çıktısını inceleyeceğiz.

4.4.1 Derleyici Çıktısı Elde Etme

GCC ile RISC-V çevirici dil çıktısı almak için `-S` seçeneği kullanılır:

```
1 riscv64-unknown-elf-gcc -S -O0 -march=rv32i -mabi=ilp32 program.c
```

Bu komut, `program.s` dosyasını üretir. `-O0` eniyileme yapılmadığını, `-march=rv32i` hedef mimarinin RV32I olduğunu belirtir.

4.4.2 Eniyileme Seviyelerinin Etkisi

Derleyicinin eniyileme seviyesi, üretilen kodun yapısını köklü biçimde değiştirir. Aşağıdaki basit fonksiyonu farklı seviyelerde derleyerek farkları görelim:

Örnek 4.16

Aşağıdaki C fonksiyonunun `-O0` ve `-O2` ile derlenen çıktılarını karşılaştırın:

```
int toplam(int dizi[], int n) {
    int sonuc = 0;
    for (int i = 0; i < n; i++)
        sonuc += dizi[i];
    return sonuc;
}
```

Çözüm: `-O0` (eniyileme yok) çıktısı; derleyici her değişkeni yığıtta saklar:

```
1 # -O0: Her değişken yığıtta, her erişimde lw/sw
2 toplam:
3     addi sp, sp, -32
4     sw   ra, 28(sp)
5     sw   s0, 24(sp)
6     addi s0, sp, 32          # çerçeve işaretçisi
7     sw   a0, -20(s0)         # dizi adresini yığıtta sakla
8     sw   a1, -24(s0)         # n'i yığıtta sakla
9     sw   zero, -28(s0)       # sonuc = 0
10    sw   zero, -32(s0)       # i = 0
11    j     dongu_kontrol
12 dongu_govde:
13    lw   t0, -32(s0)          # i'yi yığıttan oku
14    slli t0, t0, 2
15    lw   t1, -20(s0)          # dizi adresini yığıttan oku
16    add  t0, t1, t0
17    lw   t0, 0(t0)            # dizi[i]
18    lw   t1, -28(s0)          # sonuc'u yığıttan oku
19    add  t1, t1, t0
20    sw   t1, -28(s0)          # sonuc'u yığıtta yaz
21    lw   t0, -32(s0)          # i'yi yığıttan oku
22    addi t0, t0, 1
```

```

23     sw    t0, -32(s0)           # i'yi yığıtta yaz
24     dongu_kontrol:
25     lw    t0, -32(s0)           # i'yi yığıttan oku
26     lw    t1, -24(s0)           # n'i yığıttan oku
27     blt   t0, t1, dongu_govde
28     lw    a0, -28(s0)           # dönüş değeri = sonuc
29     lw    s0, 24(sp)
30     lw    ra, 28(sp)
31     addi  sp, sp, 32
32     ret

```

-02 (eniyileme açık) çıktısı; derleyici değişkenleri yazmaçlarda tutar:

```

1  # -02: Değişkenler yazmaçlarda, gereksiz erişim yok
2  toplam:
3      li    a5, 0                # sonuc = 0
4      ble   a1, zero, son        # n <= 0 ise hemen dön
5      slli  a1, a1, 2            # a1 = n * 4
6      add   a1, a0, a1           # a1 = &dizi[n] (bitiş adresi)
7  dongu:
8      lw    a4, 0(a0)            # a4 = dizi[i]
9      addi  a0, a0, 4            # işaretçiye ilerlet
10     add   a5, a5, a4           # sonuc += dizi[i]
11     bne   a0, a1, dongu        # bitiş adresine ulaşmadıysa devam
12  son:
13     mv    a0, a5               # dönüş değeri = sonuc
14     ret

```

-02 çıktısında dikkat çeken farklar:

- Yığıt çerçevesi oluşturulmamıştır; tüm değişkenler yazmaçlarda tutulur.
- Döngü sayacı (i) yerine işaretçi kullanılır; bu, her yinelemede slli ile çarpma yapma gereğini ortadan kaldırır.
- Döngü gövdesinde yalnızca 4 buyruk vardır (-00'da 10'dan fazla).
- Döngü kontrolü sona alınmıştır (*loop inversion*): döngü başında bir defa kontrol yapılır, sonra gövde-kontrol sırası tersine çevrilir.

Bilgi Notu 4.7: Derleyiciye Güvenmek

Çağdaş derleyiciler çoğu durumda insandan daha iyi çevirici dil kodu üretir. Elle çevirici dil yazmanın anlamlı olduğu durumlar sınırlıdır: önyükleme kodu, kesme işleyicileri, çok hassas zamanlama gerektiren gömülü sistemler ve kriptografi gibi alanlarda elle yazılmış kod gerekebilir. Derleyici çıktısını okumak, çevirici dilde programlamaktan çok daha yaygın ve değerli bir beceridir. Derleyicinin ürettiği kodun boru hattıyla nasıl etkileşime girdiğini (dallanma gecikmesi yuvası, yük kullanım gecikmesi gibi kavramları) Bölüm 7'de ele alacağız.

4.4.3 Yapısal Dönüşümler

Derleyicilerin uyguladığı bazı yaygın dönüşümler şunlardır:

Tablo 4.2: Yaygın derleyici eniyilemeleri

Eniyileme	C Düzeyinde	Etki
Sabit katlama	$x = 3 * 4 \rightarrow x = 12$	Derleme zamanında hesaplama
Güç azaltma	$x * 4 \rightarrow x \ll 2$	Çarpma yerine kaydırma
Döngü değişmezi taşıma	Döngüde değişmeyen hesabı dışarı çıkarma	Daha az buyruk yürütme
Ölü kod eleme	Kullanılmayan değişkenleri silme	Daha az buyruk
Döngü açma	for gövdesini k kez tekrarlama	Dallanma sayısını azaltma
Kuyruk çağırısı eleme	Özyinelemeli çağırısı döngüye çevirme	Yığıt taşmasını önleme

Derleyici Çıktısı: RISC-V, ARM ve x86

Aynı C fonksiyonunun farklı mimarilerde derlenmiş çıktıları, BKM tasarım kararlarının etkisini somut biçimde gösterir.

RISC-V: Sabit 32 bitlik buyruklar, yükle/sakla mimarisi. Döngü gövdesi genellikle 4–6 buyruktan oluşur. Yazmaç adları açık ve tutarlıdır ($a0$, $t0$, $s0$).

ARM AArch64: RISC-V'e çok benzer yapıda (sabit 32 bit buyruk, yükle/sakla). Koşullu yürütme ve bayt ters çevirme gibi ek buyruklar daha yoğun kod üretebilir. Derleyici çıktısında ldr/str ve $x0-x30$ yazmaçları kullanılır.

x86-64: Değişken uzunluklu buyruklar (1–15 bayt). Bellek-yazmaç buyrukları sayesinde ayrı yükleme buyruğu gerekmez: `add eax, [rdi]` tek buyrukla hem bellek okur hem toplar. Ancak bu karmaşıklık çözücü donanımını büyütür.

Her üç mimaride de -O2 seviyesinde döngü gövdeleri birbirine oldukça yakın buyruk sayısına sahiptir; asıl fark buyruk kodlama boyutunda ve çözücü karmaşıklığında ortaya çıkar.

Örnek 4.17

Güç azaltma (*strength reduction*) eniyilemesini bir örnekle gösterin.

Çözüm: Aşağıdaki C kodunda dizi erişimi her yinelemede çarpma gerektirir:

```
for (int i = 0; i < n; i++)
    dizi[i] = 0;           // adres = dizi + i * 4
```

Eniyileme öncesi her yinelemede `slli + add` ile adres hesaplanır. Eniyileme sonrası derleyici bunu işaretçi artırımına dönüştürür:

```
1  # Eniyileme öncesi (her yinelemede i*4 hesapla)
2  li    t0, 0                # i = 0
3  dongu1:
4  bge   t0, a1, son1
5  slli  t1, t0, 2             # t1 = i * 4
6  add   t1, a0, t1            # t1 = &dizi[i]
7  sw    zero, 0(t1)          # dizi[i] = 0
8  addi  t0, t0, 1
9  j     dongu1
10 son1:
11
12 # Eniyileme sonrası (işaretçi artır)
```

```

13     slli t0, a1, 2           # t0 = n * 4
14     add  t0, a0, t0         # t0 = bitiş adresi
15     mv   t1, a0             # t1 = geçerli adres
16 dongu2:
17     bge  t1, t0, son2
18     sw   zero, 0(t1)        # *p = 0
19     addi t1, t1, 4          # p += 4
20     j    dongu2
21 son2:

```

İkinci sürümde döngü gövdesinde çarpma yoktur; yalnızca bir sw ve bir addi vardır.

4.5 Gömülü Sistem Programlama

RISC-V mimarisi gömülü sistemlerde giderek yaygınlaşmaktadır. Bu bölümde, işletim sistemi olmadan çalışan (*bare-metal*) programlama ve bellek eşlemeli giriş/çıkış (MMIO) kavramlarını ele alacağız.

4.5.1 Bellek Eşlemeli Giriş/Çıkış

Gömülü sistemlerde çevresel aygıtlar (LED, düğme, UART, zamanlayıcı) bellek adres uzayının belirli bölgelerine eşlenir. Bu adreslere yazma ve okuma işlemleri, doğrudan donanımla iletişim kurar. Bu yönetime bellek eşlemeli giriş/çıkış (*Memory-Mapped I/O*, MMIO) denir.

Örnek 4.18

Bellek adresine eşlenmiş bir LED'i yakıp söndüren bir RISC-V programı yazın. LED kontrol yazmacı 0x10012008 adresinde olsun ve 0. bit LED'in durumunu kontrol etsin.

Çözüm:

```

1  .equ LED_ADDR, 0x10012008    # LED kontrol yazmacı adresi
2
3  .text
4  .globl main
5  main:
6      li   t0, LED_ADDR        # LED yazmacı adresi
7      li   t1, 1               # bit 0 = 1 (LED yak)
8      sw   t1, 0(t0)           # LED'i yak
9
10     # Basit gecikme döngüsü
11     li   t2, 1000000
12 gecikme:
13     addi t2, t2, -1
14     bnez t2, gecikme
15
16     sw   zero, 0(t0)          # LED'i söndür
17     j    main                 # başa dön (sürekli yanıp sön)

```

Bu programda sw t1, 0(t0) buyruğu sıradan bir bellek yazma gibi görünür, ancak hedef adres bir donanım yazmacına eşlendiği için yazılan değer doğrudan LED'in durumunu değiştirir. Bu, RISC-V'in (ve genel olarak RISC mimarilerinin) giriş/çıkış felsefesini yan-

sıtır: ayrı giriş/çıkış buyrukları yerine standart bellek erişim buyrukları kullanılır.

Bilgi Notu 4.8: Uçucu Bellek Erişimi ve Derleyici

Bellek eşlemeli giriş/çıkış adreslerine yapılan erişimlerde derleyicinin eniyileme yapması tehlikeli olabilir. Derleyici, aynı adrese yapılan ardışık okuma işlemlerini gereksiz sayarak birini eleyebilir; ancak donanım yazmacının değeri her okumada farklı olabilir (örneğin durum yazmacı). C dilinde bu sorunu önlemek için volatile niteleyicisi kullanılır. Çevirici dilde ise tüm bellek erişimleri açık biçimde yazıldığı için bu sorun oluşmaz; ancak derleyici çıktısını okurken volatile erişimlerinin elenmediğini doğrulamak önemlidir.

4.5.2 UART ile Seri İletişim

UART (*Universal Asynchronous Receiver/Transmitter*), gömülü sistemlerde en yaygın seri iletişim arabirimidir. RISC-V tabanlı mikrodenetleyicilerde UART da MMIO ile erişilir.

Örnek 4.19

UART üzerinden bir karakter dizisi gönderen bir RISC-V programı yazın. UART veri yazmacı 0x10013000 adresinde, durum yazmacı 0x10013004 adresinde olsun. Durum yazmacının 0. biti “gönderme hazır” bayrağıdır.

Çözüm:

```

1  .equ UART_TX,      0x10013000  # UART veri yazmacı
2  .equ UART_STATUS, 0x10013004  # UART durum yazmacı
3
4  .data
5  ileti: .asciz "Merhaba RISC-V!\n"
6
7  .text
8  .globl main
9  main:
10     la  a0, ileti           # ileti adresini yükle
11     jal uart_yazdir        # yazdırma fonksiyonunu çağır
12     j   main              # sürekli tekrarla
13
14 # uart_yazdir: a0 = karakter dizisi adresi
15 uart_yazdir:
16     li  t0, UART_TX
17     li  t1, UART_STATUS
18 yazdir_dongu:
19     lb  t2, 0(a0)          # bir karakter oku
20     beqz t2, yazdir_son    # sıfır karakterse bitir
21 bekle:
22     lw  t3, 0(t1)          # durum yazmacını oku
23     andi t3, t3, 1         # hazır biti kontrol et
24     beqz t3, bekle        # hazır değilse bekle
25     sw  t2, 0(t0)         # karakteri gönder
26     addi a0, a0, 1        # sonraki karakter
27     j   yazdir_dongu

```



```

28 yazdir_son:
29     ret

```

Bu programdaki “meşgul bekleme” (*busy waiting*) döngüsü (bekle etiketi) gömülü sistemlerde yaygın bir kalıptır. İşlemci, UART’ın önceki karakteri göndermeyi bitirmesini bekler. Daha verimli bir yöntem olarak kesmeler (*interrupts*) kullanılabilir; bu konu Bölüm 9’da ele alınacaktır.

4.6 Başarım ve Eniyileme

Çevirici dil düzeyinde programlamanın önemli bir boyutu da başarım çözümlemesidir. Bu bölümde bir programın kaç buyruk yürüttüğünü saymayı ve döngü eniyilemelerinin etkisini inceleyeceğiz.

4.6.1 Buyruk Sayımı ve Döngü Gövdesi Çözümlemesi

Bir programın yürütme süresini öngörmenin en basit yolu, toplam yürütülen buyruk sayısını hesaplamaktır. Bölüm 2’de gördüğümüz başarım denklemini hatırlayalım:

$$\text{Yürütme süresi} = \text{Buyruk sayısı} \times \text{BBÇ} \times \text{Saat dönem süresi}$$

Örnek 4.20

Örnek 4.3.1’deki doğrusal arama fonksiyonu, n elemanlı bir dizide aranan değer dizide bulunmadığı en kötü durumda kaç buyruk yürütür?

Çözüm: Döngü gövdesindeki buyrukları sayalım:

Buyruk	Açıklama	Sayı
bge t0, a1, bulunamadi	sınır kontrolü	1
slli t1, t0, 2	dizin $\rightarrow$ bayt uzaklığı	1
add t1, a0, t1	adres hesaplama	1
lw t2, 0(t1)	bellek okuma	1
beq t2, a2, bulundu	değer karşılaştırma	1
addi t0, t0, 1	sayaç artırma	1
j dongu	döngü başına dön	1

Döngü gövdesi her yinelemede 7 buyruk yürütür. n elemanlı dizide aranan değer bulunmazsa döngü n kez çalışır. Başlangıçtaki `li` ve sondaki `li + ret` buyruklarıyla birlikte:

$$\text{Toplam buyruk} = 1 + 7n + 1 + 1 + 1 = 7n + 4$$

$n = 1000$ için toplamda 7004 buyruk yürütülür. Eğer BBÇ = 1 ve saat sıklığı 100 MHz ise yürütme süresi yaklaşık 70,04 μs olur.

4.6.2 Döngü Açma

Döngü açma (*loop unrolling*), döngü gövdesini birden çok kez tekrarlayarak dallanma buyruğu sayısını azaltan bir eniyileme tekniğidir.

Örnek 4.21

Aşağıdaki dizi toplama döngüsünü 4 kat açarak (*4-way unroll*) buyruk sayısının nasıl değiştiğini gösterin.

Çözüm: Özgün döngü her yinelemede 1 eleman işler (4 buyruk + 1 dallanma):

```

1  # Özgün: her yinelemede 1 eleman
2  dongu:
3      bge t0, t3, son          # 1 dallanma
4      lw  t1, 0(t0)            # 1 bellek okuma
5      add t2, t2, t1           # 1 toplama
6      addi t0, t0, 4           # 1 işaretçi artırma
7      j   dongu               # 1 dallanma

```

4 kat açılmış döngü her yinelemede 4 eleman işler:

```

1  # 4 kat açılmış: her yinelemede 4 eleman
2  dongu_acik:
3      bge t0, t4, kalan        # t4 = bitiş - 16
4      lw  t1, 0(t0)            # eleman 1
5      lw  t5, 4(t0)            # eleman 2
6      lw  t6, 8(t0)            # eleman 3
7      lw  a4, 12(t0)           # eleman 4
8      add t2, t2, t1
9      add t2, t2, t5
10     add t2, t2, t6
11     add t2, t2, a4
12     addi t0, t0, 16          # işaretçiyi 16 bayt ilerlet
13     j   dongu_acik
14     kalan:                  # kalan elemanları tek tek işle
15         bge t0, t3, son
16         lw  t1, 0(t0)
17         add t2, t2, t1
18         addi t0, t0, 4
19         j   kalan
20     son:

```

Açılmış döngüde her 4 eleman için 11 buyruk yürütülür (2 dallanma + 4 yükleme + 4 toplama + 1 artırma). Özgün döngüde aynı 4 eleman için 20 buyruk gerekir (5×4). Bu, yaklaşık %45 buyruk azalması demektir. Döngü açmanın boru hattı başarımı üzerindeki etkisi Bölüm 7’de ayrıntılı olarak incelenecektir.

4.6.3 Matris İşlemleri ve Bellek Erişim Kalıpları

Matris işlemleri, bellek erişim kalıplarının başarım üzerindeki etkisini görmek için ideal bir çalışma alanıdır. C dilinde iki boyutlu diziler satır öncelikli (*row-major*) düzende saklanır: $A[0][0]$, $A[0][1]$, $A[0][2]$, ... şeklinde ardışık bellek adreslerinde yer alır.

Örnek 4.22

4×4 boyutunda iki matrisin toplamını hesaplayan bir RISC-V programı yazın ve bellek erişim kalıbını çözümleyin.

Çözüm: Matrisler bellekte satır öncelikli düzende, her eleman `.word` (4 bayt) olarak saklanır. 4×4 matris $16 \times 4 = 64$ bayt yer kaplar.

```

1  # matris_toplam: a0=A adresi, a1=B adresi, a2=C adresi
2  # C[i][j] = A[i][j] + B[i][j], 4x4 matris
3  matris_toplam:
4      li    t0, 16          # toplam eleman sayısı
5      li    t1, 0           # sayaç = 0
6  mt_dongu:
7      bge   t1, t0, mt_son
8      slli  t2, t1, 2        # bayt uzaklığı = sayaç * 4
9      add   t3, a0, t2       # &A[i][j]
10     add   t4, a1, t2       # &B[i][j]
11     add   t5, a2, t2       # &C[i][j]
12     lw    t3, 0(t3)        # A[i][j]
13     lw    t4, 0(t4)        # B[i][j]
14     add   t3, t3, t4       # A[i][j] + B[i][j]
15     sw    t3, 0(t5)        # C[i][j] = sonuç
16     addi  t1, t1, 1
17     j     mt_dongu
18 mt_son:
19     ret

```

Bu programda matris erişimi doğrusal (ardışık) olduğu için bellek erişim kalıbı verimlidir: her yinelemede üç ardışık bellek konumuna erişilir. Matris çarpımında ise durum farklıdır: B matrisinin sütunlarına erişim, her adımda $4 \times 4 = 16$ baytlık sıçramalar gerektirir. Bu düzensiz erişim kalıbı önbellek başarımını olumsuz etkiler; bu konu Bölüm 8’de ayrıntılı olarak incelenecektir.

Bilgi Notu 4.9: Alanda Yerellik ve Zamanda Yerellik

Bellek erişim kalıplarında iki temel yerellik türü vardır. *Alanda yerellik* (*spatial locality*): bir bellek konumuna erişildiyse yakındaki konumlara da yakında erişilme olasılığı yüksektir; matris satır erişimi buna örnektir. *Zamanda yerellik* (*temporal locality*): bir bellek konumuna erişildiyse aynı konuma yakında tekrar erişilme olasılığı yüksektir; döngü sayacı buna örnektir. Önbellek tasarımı bu iki yerellik türünden faydalanır. Bir programın erişim kalıplarını anlamak, başarım eniyilemesinin ilk adımıdır.

4.7 Hata Ayıklama

Çevirici dil programlarında hata ayıklama, yüksek düzeyli dillere kıyasla daha zahmetlidir çünkü derleyicinin sunduğu tür denetimi ve bellek güvenliği gibi korumalar yoktur. Bu bölümde yaygın hataları ve hata ayıklama yöntemlerini ele alacağız.

4.7.1 Yaygın Hatalar

Tablo 4.3: Çevirici dilde sık yapılan hatalar

Hata Türü	Belirti	Çözüm
Yığıt hizalama	sp değeri 4'ün (veya 16'nın) katı değil	addi sp, sp, -N ile N'i doğru hesaplayın
ra kaybı	Fonksiyondan dönüşte yanlış adrese atlanma	İç içe çağrılarda ra'yı yığıtta saklayın
Bayt/sözcük karışıklığı	Dizi erişiminde yanlış eleman okunması	Dizin $\times$ eleman boyutu unutulmamalı
İşaret genişletme	lb ile yüklenen > 127 değer negatif olur	İşaretsiz veriler için lbu kullanın
Dal uzaklığı	Uzak etikete dallanma başarısız	j veya ters koşul + j birleşimi kullanın
Saklanan yazmaç ihlali	Çağrılan fonksiyon s0-s11'i bozar	Kullanılan s yazmaçlarını yığıtta saklayın

Örnek 4.23

Aşağıdaki programdaki hataları bulun ve düzeltin:

```

1  # HATALI KOD: Dizi elemanlarını topla
2  toplam:
3      li    t0, 0          # i = 0
4      li    t1, 0          # sonuc = 0
5  dongu:
6      bge   t0, a1, bitti
7      add   t2, a0, t0      # HATA 1: bayt uzaklığı yok!
8      lw    t3, 0(t2)
9      add   t1, t1, t3
10     addi  t0, t0, 1
11     j     dongu
12 bitti:
13     mv    a0, t1
14     ret

```

Çözüm: Hata 1: add t2, a0, t0 satırında dizin doğrudan adrese ekleniyor. Her .word 4 bayt olduğu için dizin önce 4 ile çarpılmalı:

```

1      slli  t2, t0, 2      # t2 = i * 4
2      add   t2, a0, t2     # t2 = &dizi[i]

```

Bu tür bir hata, programın çökmesine veya yanlış bellek konumlarının okunmasına yol açar. Hatanın belirtisi: sonucun beklenenden çok farklı olması ya da hizalama hatası (*misaligned access*) istisnası.

Örnek 4.24

Aşağıdaki programda hata bulun. Program kare fonksiyonunu çağırıp sonucu yazdır-mak istiyor; ancak çalıştırıldığında sonsuz döngüye giriyor.

```

1  # HATALI KOD: a0'daki sayının karesini yazdır
2  .text
3  .globl main
4  main:
5      li    a0, 7
6      jal   kare
7      # a0 = 49 olmalı
8      li    a7, 1
9      ecall
10     li    a7, 10
11     ecall
12
13  kare:
14     mul   a0, a0, a0
15     ret
16
17  # hesapla: kareden sonra bir şey daha yap
18  hesapla:
19     li    a0, 5
20     jal   kare                # ra üzerine yazılır!
21     # ...
22     ret                    # ra artık yanlış adresi gösterir

```

Çözüm: hesapla fonksiyonu yaprak olmayan bir fonksiyondur (başka fonksiyon çağırır), ancak ra'yı yığıtta saklamamıştır. jal kare buyruğu ra'nın üzerine yeni bir dönüş adresi yazar ve eski değer kaybolur. ret çalıştığında ra kare'nin çağırıldığı satırın hemen sonrasını gösterir; bu da jal kare'nin tekrar çalışmasına ve sonsuz döngüye yol açar. Düzeltilmiş sürüm:

```

1  hesapla:
2      addi  sp, sp, -4
3      sw    ra, 0(sp)          # ra'yı sakla
4      li    a0, 5
5      jal   kare
6      # ...
7      lw    ra, 0(sp)          # ra'yı geri yükle
8      addi  sp, sp, 4
9      ret                    # artık doğru adrese döner

```

Bu hata, çevirici dil programlamada en sinsi hatalardan biridir: program çökmez, yalnızca sonsuz döngüye girer. RARS'ta ra yazmacının değerini izleyerek hatanın kaynağını bulmak mümkündür.

4.7.2 RARS ile Adım Adım Yürütme

RARS benzetimliği, hata ayıklama için güçlü araçlar sunar:

- Kesme noktası (*breakpoint*): Programın belirli bir satırda durmasını sağlar.
- Adım adım yürütme (*single step*): Her buyruğu tek tek çalıştırarak yazmaç ve bellek değişikliklerini izler.
- Geri sarma (*backstep*): Son yürütülen buyruğu geri alarak bir önceki duruma döner.
- Bellek görüntüleyici: Belleğin istenilen bölgesini onaltılık, onlu ve ASCII biçimlerinde gösterir.

Hata ayıklama sırasında izlenecek sistematik yaklaşım:

1. Hatanın olduğu noktayı daraltmak için programı ikiye bölerek (*binary search*) kesme noktaları koyun.
2. Her döngü yinelenmesinde beklenen ve gerçek yazmaç değerlerini karşılaştırın.
3. Bellek erişimlerinde adresin doğruluğunu kontrol edin; özellikle `slli` ile çarpma yapıp yapılmadığını.
4. Fonksiyon çağırısı ve dönüşlerinde yığıt işaretçisinin (`sp`) tutarlılığını doğrulayın.

4.8 Gerçek Dünya: RISC-V Geliştirme Ekosistemi

RISC-V çevirici dil programlama yalnızca akademik bir alıştırma değildir; gerçek dünyada gömülü sistemlerden Linux sunucularına kadar geniş bir yelpazede kullanılmaktadır.

4.8.1 Geliştirme Kartları

RISC-V ekosistemi hızla büyümektedir. Öğrencilerin ve hobi geliştiricilerin kullanabileceği bazı yaygın kartlar:

- SiFive HiFive1 Rev B: RV32IMAC çekirdeği, 320 MHz, 16 KB RAM. Gömülü programlama için ideal bir başlangıç kartıdır.
- Espressif ESP32-C3: RV32IMC çekirdeği, Wi-Fi ve Bluetooth destekli. IoT uygulamaları için yaygın biçimde kullanılır.
- StarFive VisionFive 2: RV64GC çekirdeği, 4 çekirdek, 1,5 GHz. Linux çalıştırabilen bir masaüstü düzeyinde karttır.
- Milk-V Duo: RV64GCV çekirdeği, düşük maliyetli. Vektör uzantısı desteğiyle yapay zeka uygulamalarına da uygundur.

4.8.2 Araç Zinciri

RISC-V için tam bir yazılım geliştirme araç zinciri mevcuttur:

- GCC (`riscv64-unknown-elf-gcc`): C/C++ kaynak kodunu RISC-V çevirici diline ve makine koduna derler. `-S` seçeneği ile çevirici dil çıktısı alınır; `-O0`, `-O2`, `-Os` gibi eniyileme seviyeleri seçilebilir.
- LLVM/Clang: GCC'ye seçenek olarak sunulan açık kaynaklı derleyici. Modüler yapısı sayesinde RISC-V için özel eniyilemeler eklemek daha kolaydır.
- GNU Binutils: Çevirici (`as`), bağlayıcı (`ld`), nesne dosyası inceleme araçları (`objdump`, `readelf`) içerir. `objdump -d` ile derlenmiş bir programın çevirici dil dökümü alınabilir.
- GDB (`riscv64-unknown-elf-gdb`): Uzaktan hata ayıklama için kullanılır. OpenOCD ile birlikte gerçek donanım üzerinde adım adım yürütme olanağı sağlar.
- RARS ve Venus: Bu kitapta kullandığımız benzetimli ortamlar. Kurulum gerektirmeyen, eğitim odaklı araçlardır.

Gömülü Geliştirme: RISC-V ve ARM

ARM, gömülü sistemlerde on yılların deneyimine sahiptir; Cortex-M serisi mikrodenetleyiciler milyarlarca adet satılmıştır. ARM ekosisteminde Keil MDK, IAR Embedded Workbench gibi olgun ticari araçlar ve CMSIS gibi standart donanım soyutlama katmanları bulunur.

RISC-V ekosistemi daha genç olmakla birlikte hızla olgunlaşmaktadır. Açık kaynak olmasının en büyük avantajı, araç zincirinin tamamının ücretsiz ve değiştirilebilir olmasıdır. Üniversiteler ve girişim şirketleri için bu, lisans maliyetlerini sıfıra indirmektedir. Dezavantajı ise kütüphane ve sürücü desteğinin ARM kadar geniş olmamasıdır; ancak bu boşluk her geçen yıl daralmaktadır.

4.9 Yanılgılar ve Tuzaklar

Yanılgı: Sözde buyruk sayısı, programın gerçek maliyetini gösterir

Örnek 4.2’de gördüğümüz gibi 1a sözde buyruğu iki gerçek buyruğa (`auipc + addi`) açılır. 10 sözde buyruktan oluşan bir program aslında 12 gerçek buyruk yürütebilir. Başarım çözümlemesi yaparken her zaman çeviricinin ürettiği gerçek buyrukları saymalıyız; RARS’ta “Basic” sütunu bu amaçla kullanılabilir.

Yanılgı: Daha az buyruk her zaman daha hızlı çalışır

Buyruk sayısı tek başına başarımı belirlemez. Bir 1w buyruğu önbellek bulamama durumunda yüzlerce saat vuruşu sürebilirken, bir add buyruğu tek vuruşta tamamlanır. Örneklerimizdeki başarım çözümlemelerinde gördük: bellek erişimi olmayan programda $BB\dot{C} = 1,0$ iken, yalnızca iki 1w/sw eklemek $BB\dot{C}$ ’yi $\sim 1,2$ ’ye çıkardı. Başarım çözümlemesi için buyruk sayısının yanında $BB\dot{C}$ ve bellek erişim örüntüsü de göz önünde bulundurulmalıdır.

Yanılgı: Çevirici dilde yazmak her zaman daha hızlı kod üretir

Çağdaş derleyiciler çoğu durumda insan programcıdan daha iyi kod üretir. Derleyici yüzlerce eniyileme geçişi uygular, yazmaç atamasını küresel düzeyde yapar ve hedef mikromimariyi dikkate alır. Elle yazılmış çevirici dil kodunun derleyici çıktısından hızlı olması ancak çok özel durumlarda (SIMD buyrukları, donanıma özgü eniyilemeler) mümkündür.

Tuzak: Dizi erişiminde eleman boyutunu unutmak

.word dizilerinde dizin i ’yi adrese eklerken $i \times 4$ yapılmalıdır (`slli t0, t0, 2`). Bu, çevirici dilde en sık yapılan hatadır ve genellikle programın çökmesine veya anlamsız sonuçlar üretmesine yol açar. Örnek 4.7’deki doğrusal arama kodunda bu çarpmanın nasıl yapıldığını inceleyebilirsiniz.

Tuzak: Geçici yazmaçların fonksiyon çağrılarında korunacağını varsaymak

Örnek 4.6’da gördük: yaprak fonksiyon olan `top1a` yalnızca geçici yazmaçlar kullandığı için yığıt çerçevesi oluşturması gerekmez. Ancak Örnek 4.3.7’de `main` fonksiyonu `en_buyuk`’un sonucunu bir sonraki çağrıya kadar korumak için `s0`’a (saklanan yazmaç) kopyalamak zorundadır. `t0–t6` geçici yazmaçları çağrılan fonksiyon tarafından serbestçe değiştirilebilir; bu nedenle fonksiyon çağrılarında korunması gereken değerler ya yığıta ya da `s0–s11` yazmaçlarına taşınmalıdır.

Tuzak: Yığıt işaretçisini geri döndürmeyi unutmak

Örnek 4.3'teki yazmaç izleme tablosunda gördüğümüz gibi, her `addi sp, sp, -N` için karşılık gelen bir `addi sp, sp, +N` bulunmalıdır. Bu eşleşme bozulursa yığıt sürekli büyür ve sonunda yığıt taşmasına (*stack overflow*) yol açar. Özellikle özyinelemeli fonksiyonlarda (Örnek 4.14) her çağrı yeni bir yığıt çerçevesi oluşturduğundan bu hata hızla belleğin tükenmesine neden olur.

Tuzak: Yığıt hizalamasını dikkate almamak

RISC-V çağrı düzeni, yığıt işaretçisinin (`sp`) her zaman 16 baytın katı olmasını gerektirir. Yığıtta 12 baytlık yer açmak (`addi sp, sp, -12`) hizalama kuralını bozar ve bazı sistemlerde sessiz hatalara yol açabilir. Her zaman 16'nın katı olacak biçimde yer açılmalıdır.

Tuzak: Yaprak olmayan fonksiyonda `ra`'yı saklamamak

Örnek 4.6'daki `top1a` yaprak fonksiyondur ve `ra`'yı saklamadan `ret` ile döner. Ancak `main` gibi başka fonksiyon çağırarak bir fonksiyonda `jal` buyruğu `ra`'nın üzerine yazar. Eğer eski `ra` değeri yığıtta saklanmazsa, fonksiyon çağırandan geri dönemez; program sonsuz döngüye girer veya çöker.

Yanılğı: Eniyileme seviyesini artırmak her zaman daha iyi sonuç verir

-02 ve -03 seviyeleri çoğu durumda daha hızlı kod üretir; ancak -03'te uygulanan agresif döngü açma ve fonksiyon satır içi yerleştirme (*inlining*) kod boyutunu büyütebilir. Gömülü sistemlerde sınırlı bellek olduğundan -0s (boyut için eniyileme) tercih edilebilir. Eniyileme seviyesi seçimi, hedef platformun kısıtlarına göre yapılmalıdır.

4.10 Özet

Bu bölümde RISC-V çevirici dilini sıfırdan öğrendik. İki yazmacı toplamak gibi en basit programdan başlayarak bellek erişimi, yığıt işlemleri, döngüler ve altyordam çağırma kavramlarını adım adım inşa ettik. Her örnekte yazmaç izleme tabloları ile buyrukların yazmaçları nasıl değiştirdiğini somut biçimde gördük.

Sözde buyrukların (`li`, `mv`, `la`) çevirici tarafından gerçek RISC-V buyrukları ile değiştirildiğini öğrendik. Özellikle `la` buyruğunun iki gerçek buyruğa (`auipc + addi`) açılması, sözde buyruk sayısının gerçek buyruk sayısından farklı olabileceğini gösterdi. Durağan ve devingen buyruk sayıları arasındaki farkı döngü örneğinde ilk kez gözlemledik.

Her örnekte başarımlı çözümlemesi yaparak buyruk sayısı, BBÇ ve bellek erişimi arasındaki ilişkiyi sayısal olarak hesapladık. Bellek erişimi olmayan programlarda BBÇ = 1,0 elde ederken, `lw/sw` buyrukları eklendikçe BBÇ'nin ~ 1,2'ye çıktığını gördük; bu, Bölüm 2'deki başarımlı formülünün programlama düzeyindeki somut yansımasıdır.

Temel kavramları pekiştirdikten sonra dizi arama ve sıralama algoritmalarından karakter dizisi işlemeye, yapı ve bağlı liste gibi veri yapılarından gömülü sistem programlamaya kadar geniş bir yelpazede uygulamalar geliştirdik. Özyinelemeli fonksiyonların yığıt yönetimini nasıl etkilediğini Fibonacci örneğiyle; birden çok fonksiyonun birbirini çağırdığı programlarda çağrı düzeninin önemini çok fonksiyonlu program örneğiyle inceledik. Matris işlemleri üzerinden

bellek erişim kalıplarının başarımlar üzerindeki etkisini ve alanda yerellik ile zamanda yerellik kavramlarını ele aldık.

Derleyicinin C kodundan ürettiği çevirici dil çıktısını okumayı öğrendik ve farklı eniyileme seviyelerinin (-O0, -O2) kod kalitesini nasıl etkilediğini gördük. Gerçek dünya bölümünde RISC-V geliştirme ekosistemini (donanım kartlarını, araç zincirini ve ARM ile karşılaştırmalı konumunu) inceledik.

Bir sonraki bölümde, bu programları çalıştıran donanımın nasıl tasarlandığını göreceğiz: toplama, çıkarma, çarpma ve bölme gibi aritmetik işlemleri gerçekleştiren devreleri inceleyeceğiz.

Alıştırmalar

Alıştırma 4.1. ★ Bellekte `.word` olarak tanımlanmış 10 elemanlı bir dizinin tüm elemanlarının toplamını hesaplayan ve sonucu `a0` yazmacına yazan bir RISC-V programı yazın.

Alıştırma 4.2. ★ Bir karakter dizisindeki tüm küçük harfleri büyük harfe çeviren bir fonksiyon yazın. ASCII’de küçük harfler 97–122, büyük harfler 65–90 aralığındadır; fark 32’dir.

Alıştırma 4.3. ★ İki karakter dizisini karşılaştıran (`strcmp` işlevi) bir RISC-V fonksiyonu yazın. Diziler eşitse 0, birincisi küçükse negatif, büyükse pozitif bir değer döndürün.

Alıştırma 4.4. ★★ Özyinelemeli olarak ikili arama (*binary search*) yapan bir fonksiyon yazın. Fonksiyon sıralı bir dizide aranan değer dizinin numarasını döndürsün. Yığıt çerçevesini doğru biçimde oluşturmaya dikkat edin.

Alıştırma 4.5. ★★ Seçmeli sıralama (*selection sort*) algoritmasını RISC-V çevirici dilinde yazın. Programınız `.data` bölümünde tanımlı bir diziyi yerinde (*in-place*) sıralamalıdır. Kabarcık sıralamayla karşılaştırarak toplam buyruk sayısını hesaplayın.

Alıştırma 4.6. ★★ Aşağıdaki C fonksiyonunu RISC-V çevirici diline çevirin. Çağrı düzenine tam olarak uyun:

```
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

$n = 10$ için kaç jal buyruğu yürütüldüğünü hesaplayın.

Alıştırma 4.7. ★★ Aşağıdaki -O0 ile derlenmiş RISC-V çıktısını okuyun ve hangi C fonksiyonuna karşılık geldiğini belirleyin. Ardından aynı fonksiyonun -O2 ile derlenmiş olası çıktısını elle yazın:

```
1 fonk:
2     addi sp, sp, -16
3     sw   ra, 12(sp)
4     sw   s0, 8(sp)
5     addi s0, sp, 16
6     sw   a0, -12(s0)
```

```

7      sw    a1, -16(s0)
8      lw    t0, -12(s0)
9      lw    t1, -16(s0)
10     bge    t0, t1, L1
11     lw    a0, -16(s0)
12     j      L2
13 L1:
14     lw    a0, -12(s0)
15 L2:
16     lw    s0, 8(sp)
17     lw    ra, 12(sp)
18     addi   sp, sp, 16
19     ret

```

Alıştırma 4.8. ★★ Her düğümü {int anahtar, int deger, dugum* sonraki} biçiminde olan bir bağlı listede verilen anahtara karşılık gelen değeri bulan bir fonksiyon yazın. Anahtar bulunamazsa -1 döndürsün.

Alıştırma 4.9. ★★ UART üzerinden kullanıcıdan bir karakter dizisi okuyan, bu diziyi ters çeviren ve ters çevrilmiş hâlini UART üzerinden geri gönderen bir program yazın. UART adreslerini Örnek 4.5.2’teki gibi kullanın. Giriş yazmacı 0x10013008 adresinde, durum yazmacının 1. biti “okuma hazır” bayrağı olsun.

Alıştırma 4.10. ★★★ 4×4 boyutunda iki matrisin çarpımını hesaplayan bir RISC-V programı yazın. Matris elemanları .word olarak bellekte satır öncelikli (*row-major*) düzende saklanmıştır. Programınızda:

- Üç iç içe döngü kullanarak matris çarpımını gerçekleştirin.
- Toplam yürütülen buyruk sayısını hesaplayın.
- Döngü açma uygulayarak buyruk sayısını azaltın ve kazancı yüzde olarak belirtin.

Alıştırma 4.11. ★★★ Aşağıdaki C programının -00 ve -02 derleyici çıktılarını öngörün. Her iki çıktı için döngü başına yürütülen buyruk sayısını hesaplayın ve eniyileme kazancını belirleyin:

```

void matris_toplam(int A[][4], int B[][4], int C[][4]) {
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4; j++)
            C[i][j] = A[i][j] + B[i][j];
}

```

Alıştırma 4.12. ★★★ Gömülü bir RISC-V sistemi için basit bir zamanlayıcı (*timer*) sürücüsü yazın. Zamanlayıcı MMIO yazmacı 0x0200BFF8 adresindedir ve her mikrosaniyede 1 artar. Program, verilen milisaniye sayısı kadar bekleyen bir gecikme_ms fonksiyonu içermelidir. Meşgul bekleme kullanın.

Alıştırma 4.13. ★★★ Bir RISC-V çevirici dil programı yazarak Öklid’in en büyük ortak bölen (EBOB) algoritmasını hem yinelemeli hem özyinelemeli olarak gerçekleştirin. Her iki sürüm için $n = 48$, $m = 18$ girdisiyle yürütülen toplam buyruk sayısını hesaplayarak karşılaştırın. Hangi sürüm neden daha verimlidir?

Alıştırma 4.14. ★ Aşağıdaki C ifadelerinin her birini tek bir RISC-V buyruğuna çevirin. Değişkenler sırasıyla s1–s5 yazmaçlarındadır.

- (a) $x = y + z;$
- (b) $x = y - 7;$
- (c) $x = y \& 0xFF;$
- (d) $x = y \ll 3;$

Alıştırma 4.15. ★ Aşağıdaki sözde buyrukların her birinin hangi gerçek RISC-V buyruğuna (veya buyruklarına) karşılık geldiğini yazın:

- (a) `mv t0, t1`
- (b) `li t0, 42`
- (c) `nop`
- (d) `j etiket`
- (e) `not t0, t1`

Alıştırma 4.16. ★ Bir yaprak yordam (*leaf procedure*) ile yaprak olmayan bir yordam arasındaki fark nedir? Yaprak yordam yığıta çerçeve oluşturmak zorunda mıdır? Yanıtınızı gerekçelendirin.

Alıştırma 4.17. ★ RISC-V çağrı düzeninde a0–a7 yazmaçları hangi amaçla kullanılır? Bir fonksiyon 9 terim alırsa dokuzuncu terim nereye konulur? Bu durumda çağırıcı ve çağrılan tarafın sorumlulukları nelerdir?

Alıştırma 4.18. ★ Aşağıdaki RISC-V kodunun yürütülmesinden sonra t0 yazmacının değerini bulun:

```
1  li  t0, 0x12345678
2  andi t0, t0, 0xFF
3  slli t0, t0, 4
```

Alıştırma 4.19. ★★ Aşağıdaki C fonksiyonunu RISC-V çevirici diline çevirin. Çağrı düzenine uygun ve yığıt çerçevesini doğru biçimde oluşturun:

```
int toplam(int a, int b, int c) {
    int sonuc = a + b + c;
    return sonuc;
}
```

Bu fonksiyon yaprak yordam mıdır? Yığıta ra saklamak gerekir mi?

Alıştırma 4.20. ★★ Saklanan yazmaçlar (s0–s11) ile geçici yazmaçlar (t0–t6) arasındaki farkı açıklayın. Aşağıdaki senaryoların her birinde hangi tür yazmaç kullanılmalıdır?

- (a) Bir döngü sayacı
- (b) Bir fonksiyon çağrısından önce ve sonra korunması gereken değer
- (c) Bir ara hesaplama sonucu

Alıştırma 4.21. ★★ Aşağıdaki fonksiyon bir yığıt çerçevesi oluşturuyor. Çerçeveedeki her alanın ne için kullanıldığını ve yığıt göstergesinin değişimini adım adım izleyin:

```

1  fonk:
2      addi sp, sp, -16
3      sw   ra, 12(sp)
4      sw   s0, 8(sp)
5      sw   s1, 4(sp)
6      # ... fonksiyon gövdesi ...
7      lw   s1, 4(sp)
8      lw   s0, 8(sp)
9      lw   ra, 12(sp)
10     addi sp, sp, 16
11     ret

```

Yığıt göstergesi başlangıçta 0x7FFFFFF0 ise, ra hangi bellek adresine saklanır?

Alıştırma 4.22. ★★ Aşağıdaki C kodunu RISC-V çevirici diline çevirin:

```

int mutlak(int x) {
    if (x < 0)
        return -x;
    return x;
}

```

Çevirinizde kaç buyruk kullanıldığını sayın. Dallanma buyruğu yerine koşullu seçim kullanarak buyruk sayısını azaltabilir misiniz?

Alıştırma 4.23. ★★ Bellekte .word olarak saklanan 8 elemanlı bir dizide en büyük elemanı bulan bir RISC-V programı yazın. Programınız en büyük değeri a0'a, dizin numarasını a1'e yüklesin. Toplam yürütülen buyruk sayısını en kötü ve en iyi durum için hesaplayın.

Alıştırma 4.24. ★★ Aşağıdaki kodda bir hata var. Hatayı bulun, nedenini açıklayın ve düzeltin:

```

1      .text
2  toplam:
3      li    t0, 0          # sonuc = 0
4      li    t1, 0          # i = 0
5  dongu:
6      bge   t1, a1, son     # i >= n ise cik
7      slli   t2, t1, 2       # t2 = i * 4
8      add    t2, a0, t2      # t2 = dizi[i]
9      lw     t3, 0(t2)       # t3 = dizi[i]
10     add    t0, t0, t3      # sonuc += dizi[i]
11     addi   t1, t1, 1       # i++
12     j      dongu
13  son:
14     mv     a0, t0
15     jr     ra              # don

```

İpucu: Bu fonksiyon başka bir fonksiyondan çağrılırsa ne olur?

Alıştırma 4.25. ★★ lui ve addi buyruklarını kullanarak t0 yazmacına 0xDEADBEEF değerini yükleyin. addi'nin işaret genişletmesi nedeniyle lui'ye verilen üst 20 bitin neden 0xDEADC olması gerektiğini açıklayın.

Alıştırma 4.26. ★★★ Aşağıdaki C fonksiyonunu RISC-V çevirici diline çevirin:

```
int guc(int taban, int us) {
    if (us == 0) return 1;
    return taban * guc(taban, us - 1);
}
```

- (a) Özyinelemeli sürümü yazın ve yığıt çerçevesini gösterin.
- (b) $\text{taban} = 2$, $\text{us} = 5$ için yığıtın en derin noktasındaki toplam çerçeve sayısını ve bayt cinsinden yığıt kullanımını hesaplayın.
- (c) Aynı fonksiyonun yinelemeli sürümünü yazın ve buyruk sayılarını karşılaştırın.

Alıştırma 4.27. ★★★ Bellekteki bir sözcük dizisini (.word) ters çeviren (*reverse*) bir fonksiyon yazın. Fonksiyon dizinin başlangıç adresini a0 'da, eleman sayısını a1 'de alsın ve diziyi yerinde ters çevirsin. İki uçtan ortaya doğru ilerleyen ve elemanları takas eden bir yaklaşım kullanın. Toplam bellek erişimi sayısını eleman sayısı n cinsinden ifade edin.

Alıştırma 4.28. ★★★ Aşağıdaki iki RISC-V döngüsü aynı işi yapmaktadır. Her birinin döngü başına yürütülen buyruk sayısını hesaplayın ve hangisinin neden daha verimli olduğunu açıklayın:

<pre> 1 # Surum A Surum B 2 li t0, 0 3 li t1, 100 4 donguA: 5 bge t0, a1, sonA 6 slli t2, t0, 2 7 add t2, a0, t2 8 lw t3, 0(t2) 9 add t0, t0, t3 10 addi t0, t0, 1 11 j donguA 12 sonA: </pre>	<pre> 1 li t0, 0 2 slli t2, a1, 2 3 add t2, a0, t2 4 donguB: 5 bge a0, t2, sonB 6 lw t3, 0(a0) 7 add t0, t0, t3 8 addi a0, a0, 4 9 j donguB 10 sonB: </pre>

Alıştırma 4.29. ★ RISC-V'te x0 yazmacı neden donanım tarafından sıfıra sabitlenmiştir? Bu tasarım kararının sağladığı en az üç avantajı sözde buyruk örnekleriyle açıklayın.

Alıştırma 4.30. ★★ Bir fonksiyon 12 adet saklanan yazmacı (s0-s11) kullanmak zorunda olsun. Bu fonksiyonun giriş kodu (*prologue*) ve çıkış kodu (*epilogue*) kaç buyruk gerektirir? Yığıt çerçevesinin boyutunu bayt cinsinden hesaplayın (ra dahil).

Aritmetik İşlemler



Divriği Ulu Camisi Taç Kapısı – Sivas (Fotoğraf: Engin Mutlu, CC BY-SA 4.0)

Divriği'nin taş oymalarındaki geometrik desenler, matematiğin sanata dönüşmüş halidir. İşlemcinin aritmetik birimi de benzer bir dönüşüm yapar: ikili sayılar üzerinde toplama, çarpma ve bölme işlemlerini donanıma dönüştürür.

Öğrenme Hedefleri

Bu bölümü tamamladığınızda aşağıdakileri yapabileceksiniz:

- İkilik sayılarda toplama, çıkarma ve taşıma denetimini gerçekleştirmek.
- Dalga eldeli ve önceden elde üreten toplayıcıların çalışma ilkesini ve hız farkını açıklamak.
- Mantık işlemlerini ve AMB'nin iç yapısını kavramak.
- Çarpma ve bölme algoritmalarını (ardışık, Booth, geri yüklemeli/geri yüklemez, SRT) adım adım uygulamak.
- IEEE 754 kayan nokta gösterimini, özel değerlerini ve yuvarlama yöntemlerini açıklamak.
- Kayan nokta toplama ve çarpmasının donanımda hangi adımlarla gerçekleştirildiğini betimlemek.
- BF16, TF32, FP8 gibi yapay zeka kayan nokta biçimlerinin neden geliştirildiğini ve karma duyarlılık eğitiminin nasıl çalıştığını tartışmak.

Giriş

Bilgisayarlar özünde birer hesaplama makinesidir; ilk yapıldıkları günden bu yana ana amaçları, aritmetik işlemleri insanlardan çok daha hızlı ve hatasız biçimde gerçekleştirmektir. Günümüzün bilgisayarları dokunmatik ekranlar, sesli asistanlar ve yapay zeka uygulamalarıyla donatılmış olsa da temellerinde yatan şey hâlâ aynıdır: hesaplama. Bir bilgisayarın kalbi işlemcisiyse, işlemcinin kalbi *aritmetik mantık birimi* (AMB)'dir. Aslına bakılırsa, bir işlemcinin tüm derdi iki şeye indirgenebilir: bu işlem birimlerinin girişlerine veriyi zamanında getirmek ve birim zamanda olabildiğince çok işlem biriminin çalışmasını sağlamak. Kitabın ilerleyen bölümlerinde göreceğimiz boru hattı, önbellek ve sırasız yürütüm gibi tekniklerin hepsi, özünde bu iki hedefe hizmet eder.

Bu ilke, günümüzün yapay zeka çağında her zamankinden daha geçerlidir. Büyük dil modellerini eğiten ve çalıştıran grafik işlemcileri (GPU) ile yapay zeka hızlandırıcıları, özünde devasa toplayıcı ve çarpıcı dizilerinden oluşur. Örneğin NVIDIA'nın güncel GPU'ları on binlerce kayan nokta çarpma-toplama birimi barındırır; Google'ın TPU hızlandırıcısı ise 128×128 'lik çarpıcı dizileriyle tek saat darbesinde on binlerce çarpma-toplama işlemi gerçekleştirir. Bu bölümde öğreneceğimiz toplayıcı, çarpıcı ve kayan nokta birimi tasarımları, tüm bu devlerin temel yapı taşlarıdır.

Peki bir işlemci 0 ve 1'lerden ibaret bitlerle nasıl toplama, çarpma hatta ondalıklı sayılarla işlem yapabiliyor? Üstelik tüm bu farklı sayı türleri (işaretli tam sayılar, kesirler, kayan nokta sayıları) sınırlı bir bit genişliğine sığdırılmak zorunda. Bu bölümde, bilgisayarın bu temel sorununu nasıl çözdüğünü adım adım göreceğiz: mantık kapılarından başlayarak toplama, çıkarma, çarpma ve bölme devrelerini tasarlayacak, sonra da kayan nokta gösterimine geçeceğiz.

5.1 Toplama ve Çıkarma

5.1.1 Tek Bitlik Toplama

Toplama en temel aritmetik işlemidir. Diğer tüm aritmetik işlemler toplama yapılarak gerçekleştirilebilir. Toplama işleminin ikilik tabandaki kuralları yalındır:

$$\begin{array}{l} 0 + 0 = 0 \\ 0 + 1 = 1 \\ 1 + 0 = 1 \\ 1 + 1 = 0 \quad (\text{elde var } 1) \end{array}$$

İkilik tabanda 2 olmadığı için toplanan iki basamakta da 1 rakamı olduğunda sonuç 2 yerine 0 olarak yazılır ve bir sonraki basamağa 1 elde aktarılır. Her bir toplanan basamağa bir önceki basamağın toplamı sonucunda oluşan elde eklenir ve bu toplamdan oluşan elde bir sonraki basamağa geçer. RISC-V’te (RV32I) işlenen veriler 32 bit olduğu için bu toplama işlemi 32 basamağın her biri için ayrı ayrı yapılır.

Örnek 5.1

Soru: $24 + 11 = 35$ işleminin ikilik tabanda nasıl yapıldığını gösterin.

Çözüm: 24 sayısının ikilik tabandaki karşılığı $(11000)_2$, 11 sayısının ikilik tabandaki karşılığı $(1011)_2$ ’dir. Bu iki sayının nasıl toplanacağı aşağıda gösterilmiştir.

$$\begin{array}{rcccccc}
& & & 1 & 1 & & \text{elde} \\
\hline
& 1 & 1 & 0 & 0 & 0 & (24) \\
+ & 0 & 1 & 0 & 1 & 1 & (11) \\
\hline
& 1 & 0 & 0 & 0 & 1 & 1 & (35)
\end{array}$$

Değeri 1 olan iki basamağın toplanmasından ortaya çıkan elde, soldaki basamağın üzerine eklenerek toplama işlemine devam edilir. En son ortaya çıkan $(100011)_2$ değerinin onluk tabandaki karşılığı 35'tir.

Aşağıda RISC-V'te (RV32I) 79 ve 73 değerlerinin nasıl toplandığı görülür:

$$\begin{array}{rcl} \underbrace{0000000000000000000000001001111}_{32\text{ bit}} & \rightarrow & 79 \\ + \quad \underbrace{0000000000000000000000001001001}_{32\text{ bit}} & \rightarrow & +73 \\ \hline \underbrace{00000000000000000000000010011000}_{32\text{ bit}} & \rightarrow & 152 \end{array}$$

5.1.2 Çıkarma

Çıkarma işlemi de aslında toplamanın aynısıdır; çıkarmanın toplamadan tek farkı toplama işleminden önce ikinci işlenenin (çıkarılanın) eksilisinin alınmasıdır. Eğer toplama işlemi $a + b = c$ şeklinde ifade edilirse çıkarma işlemi de toplama kullanılarak $a + (-b) = d$ şeklinde gösterilebilir. Böylece yalnızca toplama işlemi yapabilen bir sayısal devrede kolaylıkla çıkarma işlemi yapılabilir. İkinci işlenenin eksilisinin bulunması, kullanılan sayısal devrenin hangi tür sayı sistemi kullandığına bağlıdır. Güncel işlemciler büyük ölçüde 2'ye tümleyen kullandığı için bu işlemin ikiye tümleyen alınarak yapıldığı varsayılabilir.

2'ye tümleyen alma işlemi iki adımda gerçekleşir: önce sayının her biti ters çevrilir (0'lar 1, 1'ler 0 olur), ardından sonuca 1 eklenir. Bu yöntem donanım açısından son derece güçlüdür: çıkarma işlemi, mevcut toplayıcı donanımı tekrar kullanılarak yapılabilir. Donanımda b 'nin her biti DEĞİL kapısıyla ters çevrilir ve toplayıcının giren elde girişi 1 olarak ayarlanır; böylece tek bir ek adımda hem bit terslemesi hem de +1 işlemi birleştirilmiş olur. Aynı toplayıcı devresi, giren elde sinyalinin değerine göre hem toplama hem çıkarma yapabilir. Bu tasarım ilkesinin donanımdaki somut gerçekleştirilmesi Bölüm 5.3.2'de ele alınmaktadır.

Örnek 5.2

Soru: $24 - 11 = ?$ işlemini yapın.

Çözüm: 24 sayısının ikilik tabandaki karşılığı $(011000)_2$ 'dir. İlk olarak 2'ye tümleyen kullanılarak çıkarma, toplama yardımıyla yapılır:

$$\begin{array}{r} 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ (24) \\ + \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ (-11) \\ \hline 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ (13) \end{array}$$

Bir diğer örnek:

$$\begin{array}{r} 0 \ 1 \ 0 \ 0 \ (9\text{'un alt bitleri}) \\ + \ 1 \ 1 \ 0 \ 1 \ (-5\text{'in alt bitleri}) \\ \hline 0 \ 0 \ 1 \ 0 \ (4\text{'ün alt bitleri}) \end{array}$$

Verilen çıkarma işlemlerinde birinci sayı aynen ve ikincisinin 2'ye tümleyeni alınarak çıkarma işlemleri toplama yardımıyla yapılmıştır.

İkilik düzende doğrudan çıkarma da yapılabilir. Çıkarılmak istenen sayı yeterli değilse bir sonraki basamaktan 1 alınır. 2'lik düzende basamaklar birbirinin 2 katı olarak ilerlediği için bir sonraki basamaktan alınan 1, bir önceki basamakta 2 değerindedir.

Şimdiye kadar yapılan işlemlerde hep büyük sayılardan küçük sayılar çıkarıldığı için sonuçlar artı çıkmıştır. Aşağıdaki işlemde küçük bir sayıdan daha büyük bir sayı çıkararak eksi sonuç elde edilmiştir:

$$\begin{array}{r} 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ (24) \\ + \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ (-32) \\ \hline 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ (-8) \end{array}$$

Yukarıda verilen işlemde sonucun en soldaki bitinin 1 olmasından sayının eksi olduğu anlaşılır.

Bir başka örnekte:

$$\begin{array}{r} 0 \ 1 \ 0 \ 1 \ (5) \\ + \ 1 \ 1 \ 1 \ 1 \ (-1) \\ \hline 1 \ 0 \ 1 \ 0 \ 0 \ (4) \end{array}$$

sonuç toplanan iki sayıdan daha fazla bitten oluşur. İşlenenlerinden biri artı biri eksi olan bir çıkarma işleminde sonucun her iki sayıdan da büyük olması mümkün değildir. Bu nedenle taşan basamak dikkate alınmaz ve sonuç $(0100)_2 = 4$ olarak hesaplanır. İşaretsiz yapılan işlemlerde en anlamlı bitten dışarı çıkan elde taşmayı gösterir. 2'ye tümleyen gösteriminde ise

bu elde biti taşıma bilgisi taşımaz ve önemsenmez; 2'ye tümleyende taşıma, farklı bir yöntemle (en anlamlı bite giren ve çıkan eldelerin XOR'u ile) saptanır (bkz. Bilgi Notu 5.4, s. 230).

5.1.3 32 Bit ile Çıkarma

Toplama gibi çıkarma için de RISC-V (RV32I) üzerinde işlem yapılacak olan veriler 32 bittir:

[illegible]

32 bitle yapılan örnekte yine artan bir elde biti ile karşılaşılır ancak bu bit önemsizdir ve işlemin sonucu 62 olarak bulunur. Bu fazladan ortaya çıkan bit taşmayı ifade etmez.

5.1.4 Taşma

Taşmanın belirli kuralları vardır:

- 2 tane artı sayının toplamı eksi ediyorsa,
- 2 tane eksi sayının toplamı artı ediyorsa,

taşma oluşur. Aşağıda verilen toplama işlemi taşmaya bir örnektir:

$$\begin{array}{rrrrrrrrrr}
& 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & (104) \\
+ & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & (45) \\
\hline
& 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & (-107)
\end{array}$$

Eğer sayı gösterim metodu 2'ye tümleyen olmasaydı toplama işleminin sonucu doğru olacaktı (149). Ancak 2'ye tümleyen yönteminde en anlamlı biti 1 olan sayılar eksi olduğu için bu toplamanın sonucu eksi olmuştur. Belirtilen taşma kuralları gerçekleştiğinde sonuç, mevcut bit sayısı ile doğru biçimde gösterilemez.

Aynı şekilde 2 tane eksi sayının toplanmasından da artı bir sonuç elde edilmesi taşmaya sebep olur:

$$\begin{array}{rrrrrrrrrr} & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & (-103) \\ + & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & (-69) \\ \hline & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & (84) \end{array}$$

Elde biti önemsenmediği için ilk basamaktaki 1 iptal edilir ve sonuç 84 olarak bulunur.

Çıkarma işleminde de taşma kurallarının gerçekleşmesine sebep olabilecek durumlar vardır:

- 1. sayı artı, 2. sayı eksi iken çıkarmanın sonucu eksi ediyorsa,
- 1. sayı eksi, 2. sayı artı iken çıkarmanın sonucu artı ediyorsa,

taşma olur. Dolayısıyla $(A - B) < 0$ iken $A > 0, B < 0$ ve $(A - B) > 0$ iken $A < 0, B > 0$ durumlarında taşma meydana gelir.

5.1.5 RISC-V Aritmetik Buyrukları ve Taşma

RISC-V buyruk kümesinde `add`, `sub` ve `addi` buyrukları taşma durumunda herhangi bir kural dışı durum (*exception*) üretmez. MIPS'ten farklı olarak RISC-V'te işaretli ve işaretli toplama/çıkarma için ayrı buyruklar (örneğin `addu`, `addiu`) bulunmaz. Tüm aritmetik buyruklar taşma denetimi yapmadan sonucu hedef yazmacına yazar. Eğer taşma algılanması gerekiyorsa, yazılım tarafından açıkça denetlenmelidir. Bu buyrukların söz dizimi ve taşma davranışı Tablo 5.1'de özetlenmiştir.

Tablo 5.1: Aritmetik İşlem Buyrukları

Buyruk	Söz Dizimi	Anlam	Taşma
Topla (<i>Add</i>)	<code>add x1, x2, x3</code>	$x1 = x2 + x3$	Yok
Çıkar (<i>Subtract</i>)	<code>sub x1, x2, x3</code>	$x1 = x2 - x3$	Yok
Anlık değerle Topla (<i>Add immediate</i>)	<code>addi x1, x2, imm</code>	$x1 = x2 + \text{Anlık değer}$	Yok
Çarp (<i>Multiply</i>)	<code>mul x1, x2, x3</code>	$x1 = (x2 \times x3)[31:0]$	Yok
Çarp Üst (<i>Multiply High</i>)	<code>mulh x1, x2, x3</code>	$x1 = (x2 \times x3)[63:32]$	Yok
Böl (<i>Divide</i>)	<code>div x1, x2, x3</code>	$x1 = x2 / x3$ (bölüm)	Yok
Kalan (<i>Remainder</i>)	<code>rem x1, x2, x3</code>	$x1 = x2 \% x3$ (kalan)	Yok

2'ye tümleyen yönteminde en anlamlı bitin (en soldaki bit) bir bakıma işaret biti olarak kullanılması buyruk kümesi tasarımını da yalınlaştırır ve daha az buyrukla daha çok iş yapılması sağlanır; bu durum RISC yapısını da destekler. RISC-V'te taşma algılaması donanım tarafından otomatik olarak yapılmaz; gerektiğinde yazılım tarafından işlenenlerin ve sonucun en anlamlı biti denetlenerek taşma olup olmadığı belirlenmelidir. Bölüm 3'te tanıtılan aritmetik buyrukların kodlama biçimi ve işaret genişletme kuralları bu davranışı doğrudan belirler.

5.2 Mantık İşlemleri

Temel mantık kapılarının (VE, VEYA, DEĞİL) yapısı ve çalışma ilkeleri Ek B'de ayrıntılı olarak ele alınmıştır. Burada bu kapıların RISC-V buyruk kümesindeki karşılıklarına odaklanacağız.

Bir işlemci yalnızca toplama ve çıkarma yapmakla kalmaz; bitleri tek tek incelemek, maskelemek ya da karşılaştırmak gibi işlemlere de ihtiyaç duyar. Mantık işlemleri bu tür bit düzeyindeki işlemleri sağlar. Bellek adresleme, IP adresi maskeleme ve bayrak denetimi gibi pek çok temel işlem mantık buyrukları olmadan gerçekleştirilemez.

Temel mantık işlemleri VE (*AND*), VEYA (*OR*) ve DEĞİL (*NOT*)'dir. Bu işlemler, elektronikte *mantık kapıları* adı verilen temel devre elemanlarıyla gerçekleştirilir. Bu kapıların doğruluk tabloları ve devre sembolleri Ek B'de Şekil B.1'de verilmiştir. Kısaca hatırlatmak gerekirse: VE kapısı ancak tüm girişler 1 olduğunda 1 üretir; VEYA kapısı girişlerden en az biri 1 olduğunda 1 üretir; DEĞİL kapısı ise girişin tersini verir.

Mantık işlemlerinin girişleri 1 bit büyüklüğündeki verilerdir. Fakat RISC-V'teki (RV32I) tüm veriler 32 bit olduğu için her basamak bir giriş olarak alınır. Aşağıda 32 bit üzerinde yapılan VE, VEYA ve DEĞİL işlemleri gösterilir:

VE İşlemi:

```

      0000 0000 0000 0000 1010 1101 0100 1101
VE  0000 0000 0000 0011 0010 1010 0111 0001
-----
      0000 0000 0000 0000 0010 1000 0100 0001

```

VEYA İşlemi:

```

      0000 0000 0000 0000 1010 1101 0100 1101
VEYA 0000 0000 0000 0011 0010 1010 0111 0001
-----
      0000 0000 0000 0011 1010 1111 0111 1101

```

DEĞİL İşlemi:

```

      0000 0000 0000 0011 0010 1010 0111 0001
-----
DEĞİL 1111 1111 1111 1100 1101 0101 1000 1110

```

Boole cebirinde VE işlemi “.” (çarpma) ile, VEYA işlemi “+” (toplama) ile gösterilir. Bu gösterim rastlantısal değildir: VE işleminin çarpmayla, VEYA işleminin de toplamayla benzer davrandığını düşünürsek anlam kazanır. VE işleminde girişlerden herhangi biri 0 ise sonuç 0 olur; tıpkı çarpmada bir çarpanın sıfır olması gibi. VEYA işleminde ise girişlerden biri 1 ise sonuç 1 olur; toplama ile benzer bir sezgidir ($1 + 1 = 1$ olması dışında, çünkü mantık cebiri aritmetik değildir). DEĞİL işlemi ise giriş üzerine çizilen bir çizgi ($\bar{A}$) ile gösterilir. Programlama dillerinde farklı simgeler kullanılır: C dilinde VE için `&&`, VEYA için `||` ve DEĞİL için `!` yaygındır; bunlar mantık cebiri gösteriminden farklıdır.

Mantık işlemleri sayının ikilik düzeni (1 ve 0’lardan oluşan gösterimi) üzerinde yapılır.

5.2.1 Kaydırma (Shift) İşlemi

Kaydırma işlemi de mantık işlemleri içerisinde sayılır. RISC-V’te R türü buyruklarda 5 bitlik *shamt* alanı (*funct7*’nin alt bitleri) kaç basamak kaydırma yapılacağını gösterir. Sayıyı kaydırmak demek kaydırılmak istenen yönde sayının basamaklarının ilerletilmesidir. Boş kalan basamaklara mantıksal kaydırmada 0 değeri yazılır. Aritmetik kaydırmada ise sayının işaretini gösteren ilk bit kopyalanır. RISC-V’teki mantıksal kaydırma buyrukları `slli` ve `srl`, aritmetik kaydırma buyruğu ise `srai`’dir; bu buyrukların söz dizimi Tablo 5.2’de yer almaktadır.

`slli t1, s2, 3` # s2’deki değeri sola 3 basamak kaydır (mantıksal)

```

s2  0000 0000 0000 0000 0000 0010 1001 1101
t1  0000 0000 0000 0000 0001 0100 1110 1000
      ← sola 3 kaydırma

```

`srai t0, s0, 5` # s0’deki değeri sağa 5 basamak kaydır (aritmetik)

```

s0  1111 1111 1111 1100 1101 0101 1000 1110
t0  1111 1111 1111 1111 1110 0110 1010 1100
      → sağa 5 aritmetik kaydırma

```

Kaydırma işleminin önemli bir aritmetik anlamı da vardır: bir sayıyı sola k basamak kaydırmak, o sayıyı 2^k ile *çarpmaya* eşdeğerdir; sağa k basamak kaydırmak ise 2^k ’ye *bölmeye* eşdeğerdir. Örneğin bir sayıyı sola 3 basamak kaydırmak (`slli`) o sayıyı $2^3 = 8$ ile çarpar; sağa

Tablo 5.2: Bit Düzeyinde Kaydırma Buyrukları

Buyruk	Söz Dizimi	Anlam	Biçim
Sola Mantıksal Kaydır (<i>slli</i>)	<i>slli</i> x1, x2, DEĞER	$x1 = x2 \ll \text{Değer}$	I
Sağa Mantıksal Kaydır (<i>srli</i>)	<i>srli</i> x1, x2, DEĞER	$x1 = x2 \gg \text{Değer}$	I
Sağa Aritmetik Kaydır (<i>srai</i>)	<i>srai</i> x1, x2, DEĞER	$x1 = x2 \gg (\text{Değer \% } 32)$	I

Tablo 5.3: Mantık Buyrukları

Buyruk	Söz Dizimi	Anlam	Biçim
VE (<i>And</i>)	<i>and</i> x1, x2, x3	$x1 = x2 \& x3$	R
Anlık değerle VE (<i>Andi</i>)	<i>andi</i> x1, x2, DEĞER	$x1 = x2 \& \text{DEĞER}$	I
VEYA (<i>Or</i>)	<i>or</i> x1, x2, x3	$x1 = x2 x3$	R
Anlık değerle VEYA (<i>Ori</i>)	<i>ori</i> x1, x2, DEĞER	$x1 = x2 \text{DEĞER}$	I
Dışlayan VEYA (<i>Xor</i>)	<i>xor</i> x1, x2, x3	$x1 = x2 \oplus x3$	R
Anlık değerle Dışlayan VEYA (<i>Xori</i>)	<i>xori</i> x1, x2, DEĞER	$x1 = x2 \oplus \text{DEĞER}$	I
Küçükse Birle (<i>Slt</i>)	<i>slt</i> x1, x2, x3	$x1 = (x2 < x3)$	R
Anlık Değerle Küçükse Birle (<i>Slti</i>)	<i>slti</i> x1, x2, DEĞER	$x1 = (x2 < \text{DEĞER})$	I

2 basamak kaydırmak (*srli* veya *srai*) ise 4'e bölmekle aynı sonucu verir. Bu ilişki, ikilik sayı sisteminin doğal bir sonucudur; tıpkı onluk tabanda bir sayıya sıfır eklemenin o sayıyı 10 ile çarpması gibi.

İşlem	Buyruk	Aritmetik Karşılığı	Örnek
Sola k kaydır	<i>slli</i> rd, rs, k	$rs \times 2^k$	<i>slli</i> t0, t1, 3 $\rightarrow t1 \times 8$
Sağa k kaydır (mantıksal)	<i>srli</i> rd, rs, k	$\lfloor rs/2^k \rfloor$ (işaretsiz)	<i>srli</i> t0, t1, 2 $\rightarrow t1/4$
Sağa k kaydır (aritmetik)	<i>srai</i> rd, rs, k	$\lfloor rs/2^k \rfloor$ (işaretli)	<i>srai</i> t0, t1, 1 $\rightarrow t1/2$

Bu ilişki yalnızca kuramsal bir gözlem değildir; derleyiciler ve donanım tasarımcıları bunu sürekli kullanır. Çarpma ve bölme buyrukları birden fazla saat çevrimi gerektirirken, kaydırma tek bir çevrimde tamamlanır. Bu nedenle derleyiciler, 2'nin kuvvetiyle yapılan çarpma ve bölme işlemlerini otomatik olarak kaydırma buyruklarına dönüştürür. Örneğin C dilindeki $x * 16$ ifadesi derleyici tarafından *slli* ile 4 basamak sola kaydırmaya, $x / 8$ ifadesi ise *srai* (veya *srli*) ile 3 basamak sağa kaydırmaya çevrilir. Hatta $x \times 10$ gibi 2'nin kuvveti olmayan çarpımlar bile kaydırma ve toplama birleşimiyle gerçekleştirilebilir: $x \times 10 = x \times 8 + x \times 2 = (x \ll 3) + (x \ll 1)$.

Bilgi Notu 5.1: RISC-V'te DEĞİL İşlemi

Birçok buyruk kümesi mimarisinde bitleri terslemek için özel buyruklar bulunur. Örneğin MIPS'te *nor* (VEYA-DEĞİL) buyruğu vardır ve DEĞİL işlemi *nor \$t0, \$s0, \$zero* biçiminde (bir girişi sıfır yaparak) gerçekleştirilir. x86'da ise doğrudan *NOT* buyruğu mevcuttur. RISC-V ise yalınlık ilkesi gereği ayrı bir DEĞİL veya *nor* buyruğu tanımlamaz. Bunun yerine DEĞİL işlemi, mevcut Dışlayan VEYA buyruğu ile gerçekleştirilir: *xori rd, rs, -1* (montaj dilinde *not rd, rs* sözde buyruğu). -1 değeri ikilik tabanda tüm bitleri 1 olan sayıdır; herhangi bir bitin 1 ile Dışlayan VEYA'sı o bitin tersini verir. Bö-

lece tek bir buyrukla tüm bitler terslenir.

5.3 AMB (Aritmetik Mantık Birimi)

Bu bölümde mantık kapılarını (Ek B) kullanarak aritmetik ve mantık işlemlerini gerçekleştirecek bir AMB tasarlayacağız.

Toplama, çıkarma, VE, VEYA: tüm bu işlemleri ayrı ayrı öğrendik. Şimdi sıra geldi hepsini tek bir birimde birleştirmeye. İşte *Aritmetik Mantık Birimi* (AMB), bir denetim sinyaline bakarak hangi işlemin yapılacağına karar veren ve sonucu üreten birimdir. Tasarım stratejimiz yalın ama güçlüdür: önce 1 bitlik bir AMB tasarlayacağız, sonra bu birimden 32 tanesini yan yana koyarak 32 bitlik bir AMB elde edeceğiz.

1 bitlik AMB'nin desteklemesi gereken temel aritmetik işlemler toplama ve çıkarma; mantık işlemleri ise VE ve VEYA'dır.

AMB'nin çalışma ilkesini anlamak için şu noktayı kavramak gerekir: AMB, hangi işlemin yapılacağına kendisi karar vermez; *tüm işlemleri aynı anda koşut olarak gerçekleştirir*. Toplama birimi toplar, VE kapısı VE işlemini yapar, VEYA kapısı VEYA işlemini yapar; bunların hepsi eş zamanlı çalışır. AMB'nin çıkışındaki çoklayıcı, *denetim sinyaline* bakarak doğru sonucu seçer. Bu denetim sinyali nereden gelir? Yürütülmekte olan buyruğun işlem kodu alanından (*opcode* ve *funct* bitleri) bir denetim birimi aracılığıyla üretilir. Böylece add buyruğu yürütülürken toplama sonucu, and buyruğu yürütülürken VE sonucu çoklayıcı tarafından seçilir; ama her iki durumda da diğer işlemler de yapılmış, sonuçları atılmıştır.

5.3.1 AMB ile Toplama

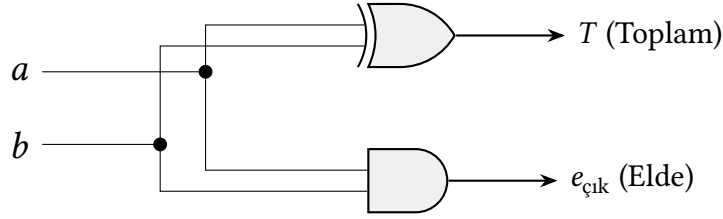
Yarım Toplayıcı

Bir bitlik toplama işleminin doğruluk tablosu ve devre şeması aşağıda verilmiştir. Sonuç sütunu Dışlayan VEYA mantıksal kapısının doğruluk tablosu ve elde sütunu ise VE'nin doğruluk tablosuyla aynı olduğu için girişlerine *a* ve *b*'nin bağlanacağı bir Dışlayan VEYA ve VE kapısıyla “yarım toplayıcı” denilen 1 bitlik toplama işlemi yapan donanım birimi elde edilmiştir. Yarım toplayıcının giriş-çıkış ilişkisi Tablo 5.4'de, devre şeması ise Şekil 5.1'de gösterilmiştir.

Tablo 5.4: Yarım toplayıcı doğruluk tablosu

a	b	Toplam	Elde
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Yarım toplayıcı tek başına kullanışsız görünebilir; elde girişi olmadan gerçek bir toplama yapılamaz. Ancak yarım toplayıcının asıl değeri, tam toplayıcının yapı taşı olmasıdır: Şekil 5.2'de görüleceği gibi, iki yarım toplayıcı ve bir VEYA kapısı birleştirilerek elde girişini de hesaba katan bir tam toplayıcı elde edilir.



Şekil 5.1: Yarım toplayıcı devresi

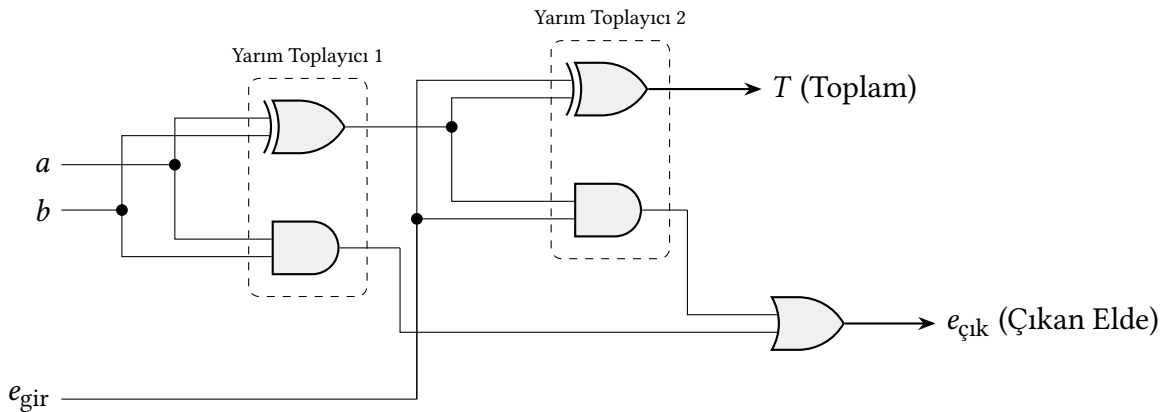
Tam Toplayıcı

Toplama işlemi sadece bir bit ile gerçekleştirilmez. Bir önceki basamaktan gelen eldeyi de toplayabilecek bir “tam toplayıcı” kullanılır. Tam toplayıcının olası giriş kombinasyonlarının tümü Tablo 5.5’de verilmiştir.

Tablo 5.5: Tam toplayıcı doğruluk tablosu

a	b	Giren Elde	Toplam	Çıkan Elde
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tablodan yararlanarak sadece çıkış sütunları olan “sonuç” ve “çıkan elde”ye bakılarak AMB tasarımı tek bir kapıyla gerçekleştirilemez. Bu nedenle fonksiyonların sadeleştirilmesi için Karno haritası yöntemi kullanılır (bkz. Ek B).



Şekil 5.2: Tam toplayıcı devre şeması

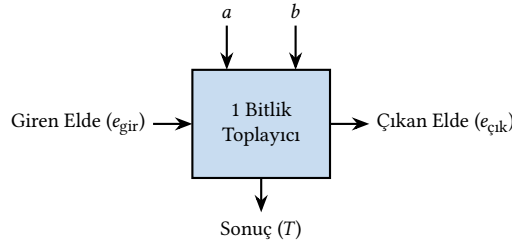
Bir tane tam toplayıcı ile 1 bitlik toplama yapılabilir. Peki çok basamaklı sayıları nasıl toplarsınız? Elle toplama yaparken en sağdaki basamaktan (birler basamağı) başlayıp sola doğru ilerlediğimiz gibi, donanımda da aynı mantık geçerlidir.

n bitlik bir sayının bitleri sağdan sola doğru 0’dan başlayarak numaralanır. En sağdaki bit (bit 0) *en az anlamlı bit* (*Least Significant Bit*, LSB) olarak adlandırılır; çünkü temsil ettiği değer

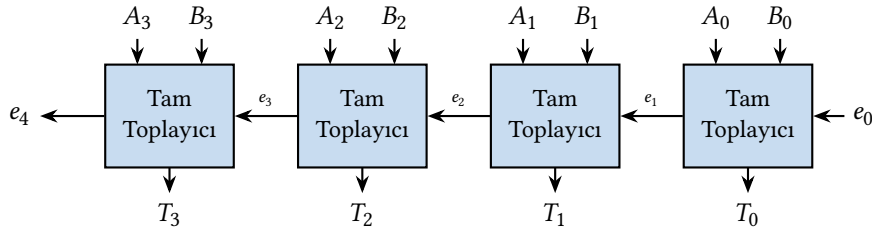
$(2^0 = 1)$ en küçüktür ve sayının toplam değerine en az katkıyı sağlar. En soldaki bit (bit $n-1$) ise *en anlamlı bit* (*Most Significant Bit*, MSB) olarak adlandırılır; temsil ettiği değer (2^{n-1}) en büyüktür ve sayının değerine en çok katkıyı bu bit yapar.

Çok basamaklı toplama yapmak için n tane tam toplayıcı zincirleme bağlanır: her tam toplayıcının çıkan eldesi (C_{out}), bir sonraki (solundaki) tam toplayıcının giren eldesine (C_{in}) bağlanır. Elde sinyali böylece en az anlamlı bitten en anlamlı bite doğru “dalgalanarak” ilerler; bu yapıya *dalga eldeli toplayıcı* (*ripple carry adder*) denir. En sağdaki toplayıcının giren eldesi (C_0) toplama için 0, çıkarma için 1 yapılır. En soldaki toplayıcının çıkan eldesi (C_n) ise taşma bilgisi taşır.

Şekil 5.3’de, Şekil 5.2’deki tam toplayıcı devresinin giriş ve çıkış sinyallerini özetleyen blok gösterimi verilmiştir. Bu blok gösterim, çok bitlik toplayıcı tasarımlarında her bir tam toplayıcıyı tek bir kutu olarak çizmeyi kolaylaştırır. Şekil 5.4’de ise bu blokların zincirlenmesiyle oluşan 4 bitlik dalga eldeli toplayıcının yapısı gösterilmiştir.



Şekil 5.3: 1 bitlik toplayıcının blok gösterimi. Şekil 5.2’deki tam toplayıcı devresinin giriş/çıkışlarını özetler.



Şekil 5.4: 4 bitlik dalga eldeli toplayıcı. Elde sinyali en az anlamlı bitten (sağ) en anlamlı bite (sol) doğru dalgalanarak ilerler.

Bilgi Notu 5.2: Bit Numaralandırması: 0’dan mı 1’den mi?

Şekil 5.4’de en sağdaki bit (en az anlamlı bit) 0 numarasını taşır. Bu, RISC-V dahil çoğu çağdaş mimaride (MIPS, ARM, x86) kullanılan standarttır: bit 0 en az anlamlı bittir ve her bitin değeri 2^i ’dir (i = bit numarası). Ancak IBM System/360 (1964) ve onun mirasçıları olan z/Architecture sistemlerinde numaralandırma *tersine* çalışır: bit 0 en anlamlı bittir. Bu iki gelenek zaman zaman karışıklığa yol açar; bir belge okurken hangi numaralandırma kuralının kullanıldığına dikkat etmek gerekir.

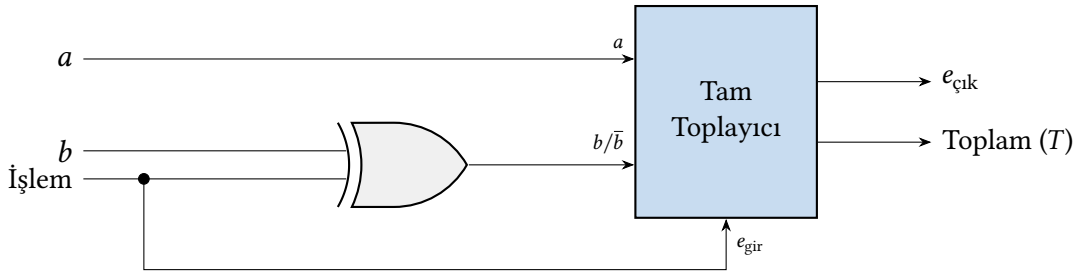
5.3.2 AMB ile Çıkarma

Çıkarma işlemi ikinci sayının eksisinin alınıp birinci sayıyla toplanmasıyla yapılır. Sayının 2’ye tümleyeninin alınması için giren elde biti 0 değil 1 alınır (hatırlatma: 2’ye tümleyen almak için tersi alınan sayıya 1 eklenir).

Şekil 5.3'de gösterilen 1 bitlik toplayıcı bloğunun üzerine çıkarma desteği eklenmiş hali Şekil 5.5'de verilmiştir.

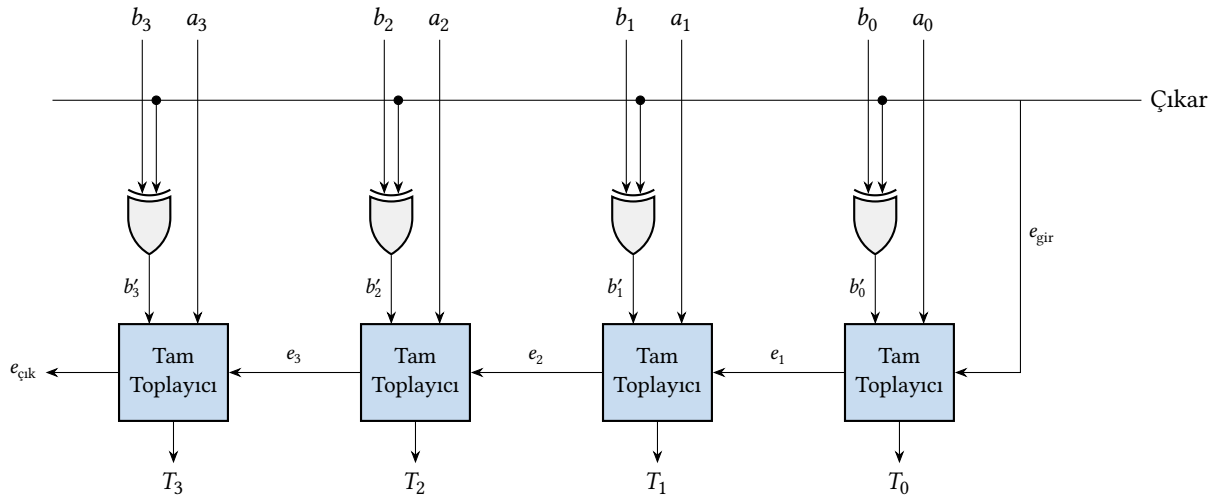
Çok bitlik bir toplayıcı-çıkarıcı devresinde, tek bir *Çıkar* denetim sinyali tüm işlemi yönetir. Bu sinyalin iki etkisi vardır: b 'nin her bitini Dışlayan VEYA kapısından geçirir ve en az anlamlı bitin giren eldesini ayarlar.

- **Çıkar = 0 (toplama):** Dışlayan VEYA kapıları b bitlerini değiştirmeden geçirir; giren elde $e_g = 0$ olur. Devre $a + b$ hesaplar.
- **Çıkar = 1 (çıkarma):** Dışlayan VEYA kapıları b bitlerini tersler; giren elde $e_g = 1$ olur. Devre $a + \bar{b} + 1$ hesaplar; bu ise 2'ye tümleyen tanımı gereği $a + (-b) = a - b$ 'dir.



Şekil 5.5: Toplama ve çıkarma yapabilen 1 bitlik AMB

Şekil 5.5'de b girişini terslemek için Dışlayan VEYA kapısı kullanılmıştır: İşlem sinyali 0 olduğunda bu kapı b 'yi aynen geçirir, 1 olduğunda ise tersler. Gerçek AMB tasarımlarında ise b 'yi terslemek için çoğunlukla DEĞİL kapısı tercih edilir; çünkü CMOS teknolojisinde DEĞİL kapısı yalnızca 2 transistörle gerçekleştirirken Dışlayan VEYA kapısı 8–12 transistör gerektirir.

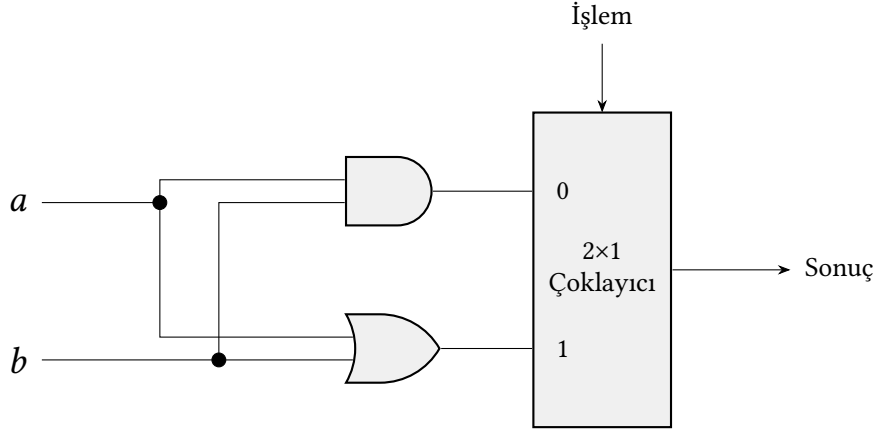


Şekil 5.6: 4 bitlik toplayıcı-çıkarıcı devresi. Çıkar sinyali 0 olduğunda b değerleri aynen geçer ve toplama yapılır; 1 olduğunda b terslenir ve giren elde 1 yapılarak çıkarma (2'ye tümleyen) gerçekleştirilir.

Bu tasarımda ek bir toplama devresi gerekmez: aynı donanım üzerinde yalnızca tek bir denetim sinyaliyle hem toplama hem çıkarma işlemi gerçekleştirilir. 4 bitlik toplayıcı-çıkarıcı devresinin şeması Şekil 5.6'de verilmiştir.

5.3.3 AMB ile Mantık İşlemleri

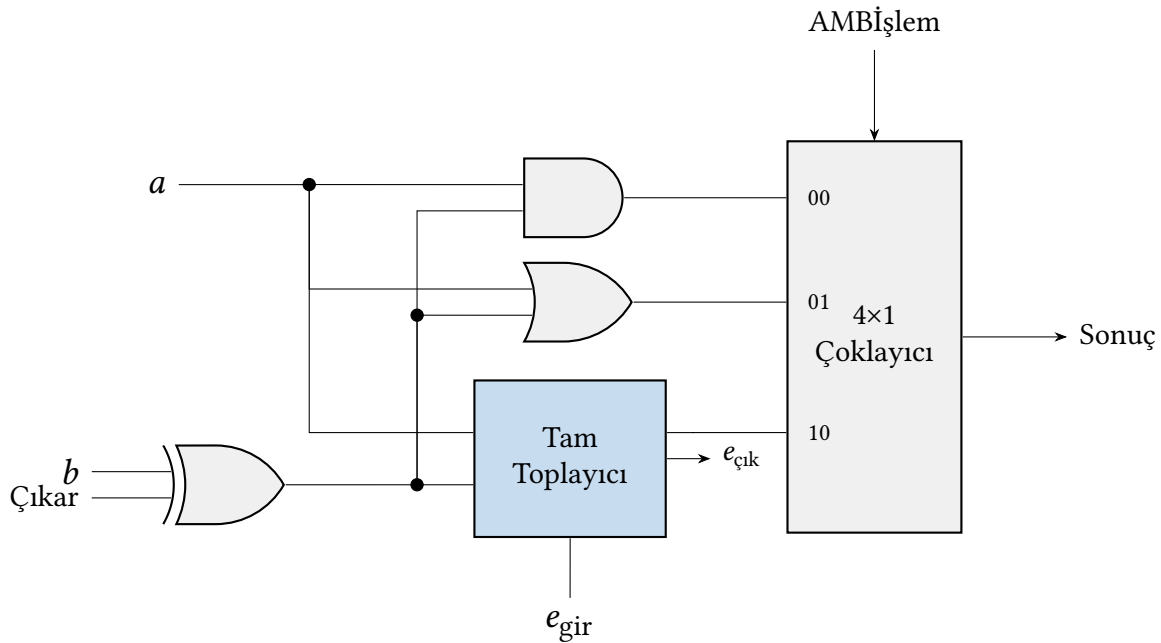
Mantık işlemleri için temel kapıların kullanılması yeterlidir. Bu durumda sadece mantık işlemleri yapacak bir birim VE ve VEYA kapıları ve bir çoklayıcı ile yapılabilir; bu birimin devre şeması Şekil 5.7’de gösterilmiştir. RISC-V buyruk kümesindeki mantık buyrukları Tablo 5.3’de listelenmiştir.



Şekil 5.7: VE ve VEYA işlemleri yapabilen AMB (1 bitlik)

5.3.4 Bir Bitlik AMB

Mantık işlemleri yapan AMB ile aritmetik işlemler yapan AMB birleştirildiğinde ve sonuç seçimi için 4×1 'lik çoklayıcı eklendiğinde 1 bitlik AMB elde edilir; bu birimin tam devre şeması Şekil 5.8’de verilmiştir.



Şekil 5.8: 1 bitlik AMB

Örneğin çıkarma işlemi yapmak için: İşlem denetimi = 10, Çıkar = 1, Giren elde = 1 olacaktır. Toplama yapmak için ise: İşlem denetimi = 10, Çıkar = 0, Giren elde = 0 alınır. Çıkarma

işlemi yapılırken “çıkart” ve “giren elde” girişlerinin ikisi de 1 alındığı için bu iki girişle birlikte “çıkart” girişi adı verilip tek bir değer gönderilebilir.

Bilgi Notu 5.3: Gerçek İşlemcilerde VEYA-DEĞİL ve VE-DEĞİL Kapıları

Bu bölümde AMB’yi VE, VEYA ve Dışlayan VEYA kapılarıyla tasarladık. Ancak gerçek CMOS devrelerinde bu kapılar doğrudan kullanılmaz. CMOS teknolojisinde *VEYA-DEĞİL* (NOR) ve *VE-DEĞİL* (NAND) kapıları, VE ve VEYA kapılarından daha az transistörle gerçekleşir ve daha hızlı çalışır. Bir CMOS VE kapısı aslında bir VE-DEĞİL kapısı ile ardından gelen bir DEĞİL kapısından (evirici) oluşur; iki aşama gerektirir. VE-DEĞİL kapısı ise tek aşamada sonuç üretir. Bu nedenle yüksek başarımlı AMB tasarımlarında tüm mantık VE-DEĞİL ve VEYA-DEĞİL kapılarıyla kurulur; çıkıştaki evirici gereken yerde eklenir.

5.3.5 32 Bitlik AMB

32 bitlik AMB yapmak için 32 tane 1 bitlik AMB birbirine bağlanır; bu yapı Şekil 5.10’de gösterilmiştir. 1 bitlik AMB bazı temel işlemleri gerçekleştirir, fakat RISC-V’te bu AMB’nin yapabildiği temel işlerden daha fazla aritmetik işlem buyruğu vardır. Bunlardan birisi *slt* (*set on less than*) “küçükse birle” buyruğudur.

```
1 slt t1, s0, s2 # s0 < s2 ise t1 = 1
```

$s0 = a$ ve $s2 = b$ olarak alınırsa a, b ’den küçükse sonuç ($t1$) yazmacının değerinin 1 yapılabacağı anlamına gelir. Peki AMB bu karşılaştırmayı nasıl yapar?

Anahtar gözlem şudur: $a < b$ olup olmadığını anlamak için $a - b$ çıkarması yapılır. Eğer sonuç eksiye $a < b$ demektir. 2’ye tümleyen düzeninde eksi bir sayının en anlamlı biti (en soldaki bit) her zaman 1’dir. Demek ki çıkarma sonucunun en anlamlı bitine bakmak, karşılaştırmamızın sonucunu verir:

- $(a - b)$ ’nin en anlamlı biti = 1 $\Rightarrow$ sonuç eksi $\Rightarrow a < b \Rightarrow$ çıkış 1 olmalı
- $(a - b)$ ’nin en anlamlı biti = 0 $\Rightarrow$ sonuç artı veya sıfır $\Rightarrow a \geq b \Rightarrow$ çıkış 0 olmalı

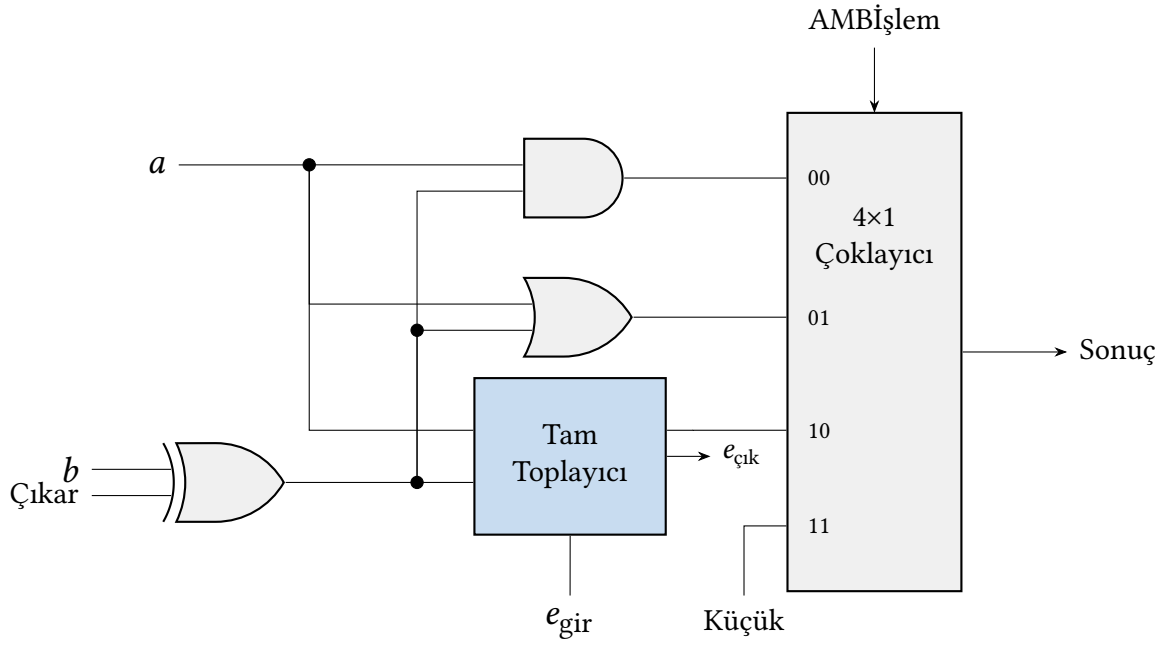
Ancak burada bir incelik vardır: *slt* buyruğunun sonucu 32 bitlik bir yazmaca yazılır ve bu yazmacın değeri ya 0 ya da 1 olmalıdır. “1” değeri ikilik tabanda $00 \dots 001$ ’dir; en az anlamlı bit 1, diğer 31 bit 0. Oysa çıkarma sonucunun işaret bilgisi *en anlamlı* bittedir. Bu bilginin en az anlamlı bite taşınması gerekir.

Çözüm olarak 1 bitlik AMB’ye “küçük” adında dördüncü bir giriş eklenir (Şekil 5.9). 32 bitlik AMB’de bu giriş şöyle kullanılır:

1. Önce $a - b$ çıkarması yapılır (AMBİşlem = 10, Çıkart = 1).
2. En anlamlı biti hesaplayan son AMB birimi (bit 31), çıkarma sonucunun işaret bitini üretir.
3. Bu işaret biti, *en az anlamlı bitin* (bit 0) AMB biriminin “küçük” girişine bağlanır.
4. Bit 0’ın çoklayıcısı AMBİşlem = 11 olduğunda bu “küçük” girişini seçer.
5. Diğer tüm bitlerin (bit 1 – bit 31) “küçük” girişi 0’a bağlıdır; böylece bu bitlerin çıkışı 0 olur.

Sonuçta yazmac ya $00 \dots 001$ ($a < b$) ya da $00 \dots 000$ ($a \geq b$) değerini alır.

Özetle: AMB zaten çıkarma yapabildiği için küçüklük karşılaştırması için yeni bir devre tasarlamaya gerek yoktur; yalnızca çıkarma sonucunun işaret bitini doğru yere yönlendir-



Şekil 5.9: “Küçük” girişi eklenmiş 1 bitlik AMB.

mek yeterlidir. Küçükse birle buyruğunun işleme konulması için AMBİşlem denetim sinyalinin “(11)₂ = 3” olması gerekir.

5.3.6 Dallanma Buyruklarının Desteklenmesi

RISC-V buyruk kümesi mimarisi için AMB’nin yerine getirmesi gereken buyruklardan birisi de dallanma buyruklarıdır. Örneğin, bne (*branch not equal*) eşit değilse dallan buyruğunun çalıştırılabilmesi için bir eşitlik testine ihtiyaç vardır. Bu eşitlik testi,

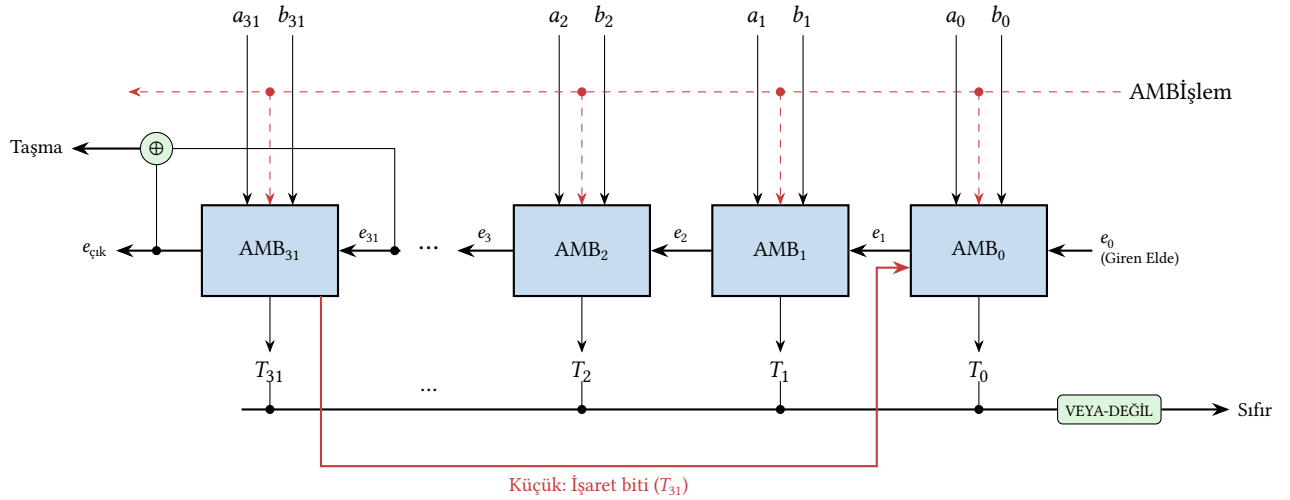
$$a = b \implies a - b = 0$$

eşitliği ile sağlanabilir. Eşitliği test etmek için a ’dan b çıkarılır ve sonucun 0 olup olmadığı araştırılır. Yazmacın değerinin 0 olması demek tüm bitlerinin 0 olması anlamına gelir. Tüm 1 bitlik AMB’lerden çıkan sonuçlar VEYA kapısına sokulduğunda ve tümü 0 olduğunda kapının çıkışı 0 olacaktır. Sıfır çıkışı girdilerin eşit olup olmadığını test etmeyi sağladığı için değerler eşitse (VEYA’nın çıkışı 0 olduğunda) değer 1, çıkış 1 ise değer 0 olmalıdır. Bu nedenle VEYA kapısının çıkışına bir tersleyici bağlanır.

Tablo 5.6: 32 Bitlik AMB İşlem Tablosu

Çıkar	İşlem Denetimi	İşlem
0	00	VE
0	01	VEYA
0	10	Toplama
1	10	Çıkarma
1	11	Küçükse Birle

Bu AMB’ye ayrıca taşma denetimi birimi de eklenmiştir. Taşma denetiminin yapılacağı birime hangi işlemin yapılacağı, işlenecek değerler ve çıkan elde değeri bağlanır. Bölüm 5.1.4’te



Şekil 5.10: 32 Bitlik AMB

anlatılan taşma kurallarının gerçekleşip gerçekleşmediği sınanır. AMB'nin desteklediği işlemler ve bu işlemlere karşılık gelen denetim sinyalleri Tablo 5.6'de özetlenmiştir. AMB'nin tek çevrimli ve çok çevrimli veri yolundaki rolü Bölüm 6'da ele alınacaktır.

Bilgi Notu 5.4: Taşma Denetiminde Elde Kuralı

Toplamada en anlamlı bitlerin bulunduğu kolona giren elde (e_{31}) ve çıkan elde ($e_{\text{çık}}$) birbirinden farklıysa taşma olur. Bu kural, Şekil 5.10'deki dışlayan VEYA kapısının ($\oplus$) temelini oluşturur:

- $e_{31} = 0, e_{\text{çık}} = 1$: İki artı sayı toplanmış, elde en anlamlı bittten dışarı taşmıştır; sonuç yanlışlıkla eksi çıkar.
- $e_{31} = 1, e_{\text{çık}} = 0$: İki eksi sayı toplanmış, en anlamlı bite elde gelmesine karşın dışarı elde çıkmamıştır; sonuç yanlışlıkla artı çıkar.
- $e_{31} = e_{\text{çık}}$ olduğunda ise taşma yoktur.

Çıkarma ($A - B$), donanımda $A + \bar{B} + 1$ biçiminde gerçekleştirildiği için aynı elde kuralı çıkarma taşmasını da doğru biçimde saptar.

5.3.7 Eldenin Önceden Üretilmesi (Carry Lookahead)

1 bitlik AMB, 32 bitlik AMB'nin temel birimidir ve 32 bitlik AMB yapımında 32 tane 1 bitlik AMB kullanılır. AMB'nin bu şekilde tasarlanması pek çok sorunu da beraberinde getirmiştir. Bunlardan en önemlisi elde zincirinin yarattığı gecikmedir: her 1 bitlik toplayıcı, çıkan eldesini hesaplayabilmek için bir önceki basamaktan gelen eldeyi beklemek zorundadır.

Dalga Eldeli Toplayıcının Gecikme Çözümlemesi

Her 1 bitlik toplayıcının elde çıkışını üretmesi 1 kapı gecikmesi (1G) sürsün. 4 bitlik bir dalga eldeli toplayıcıda gecikme şöyle birikir:

- AMB₀: e_0 hazır, 1G sonra e_1 üretilir.
- AMB₁: e_1 'i bekler, 1G daha harcar $\rightarrow e_2$ toplam 2G'de hazır olur.

- AMB_2 : e_2 'yi bekler $\rightarrow e_3$ toplam 3G'de hazır olur.
- AMB_3 : e_3 'ü bekler $\rightarrow e_4$ toplam 4G'de hazır olur.

Genel olarak n bitlik bir dalga eldeli toplayıcının elde gecikmesi nG 'dir. 32 bitlik bir toplayıcıda son elde (e_{32}) ancak 32G sonra hazır olur; bu süre boyunca sonuç bitleri de güvenilir değildir. Bu yöntemle yapılmış olan toplayıcılara *dalga eldeli toplayıcı* denir ve elde sinyalinin basamaktan basamağa dalga gibi ilerlemesinden adını alır.

Geçirme ve Üretme Kavramları

Dalga eldeli toplayıcının temel sorunu, her basamağın bir öncekinin eldesine bağlı olmasıdır. Peki bir basamağın elde üretip üretmeyeceği önceden bilinebilir mi? Bu sorunun yanıtı, her basamaktaki a_i ve b_i değerlerine bakılarak verilebilir.

Elde değeri şöyle ifade edilir:

$$e_{i+1} = a_i b_i + a_i e_i + b_i e_i = a_i b_i + e_i (a_i + b_i) \quad (5.1)$$

Bu denklemde iki ayrı durum gizlidir. a_i ve b_i 'nin ikisi de 1 ise giren eldeden bağımsız olarak elde *üretilir*. a_i ve b_i 'den yalnızca biri 1 ise giren elde varsa çıkan elde de olur; başka bir deyişle elde *geçirilir*. Bu iki davranışı ayrı ayrı tanımlayan p_i ve g_i eşitliklerinden yararlanılır:

- p_i (*propagate* – geçirme): $p_i = a_i \oplus b_i$
- g_i (*generate* – üretme): $g_i = a_i \cdot b_i$

Bitlerin ikisi de 1 ise elde *üretilir* (g_i), ikisi de farklı ise elde *geçirilir* (p_i). Bir sonraki basamakta elde oluşması için bir önceki basamaktan gelmiş olan eldenin geçirilmesi ya da elde *üretilmesi* gerekir:

$$e_{i+1} = g_i + e_i \cdot p_i \quad (5.2)$$

Tüm basamaklardaki elde durumları:

$$e_1 = g_0 + e_0 p_0 \quad (5.3)$$

$$e_2 = g_1 + e_1 p_1 \quad (5.4)$$

$$e_3 = g_2 + e_2 p_2 \quad (5.5)$$

$$e_4 = g_3 + e_3 p_3 \quad (5.6)$$

Eşitlikler birbirinin yerine konularak açılırsa:

$$e_1 = g_0 + p_0 e_0 \quad (5.7)$$

$$e_2 = g_1 + g_0 p_1 + p_0 p_1 e_0 \quad (5.8)$$

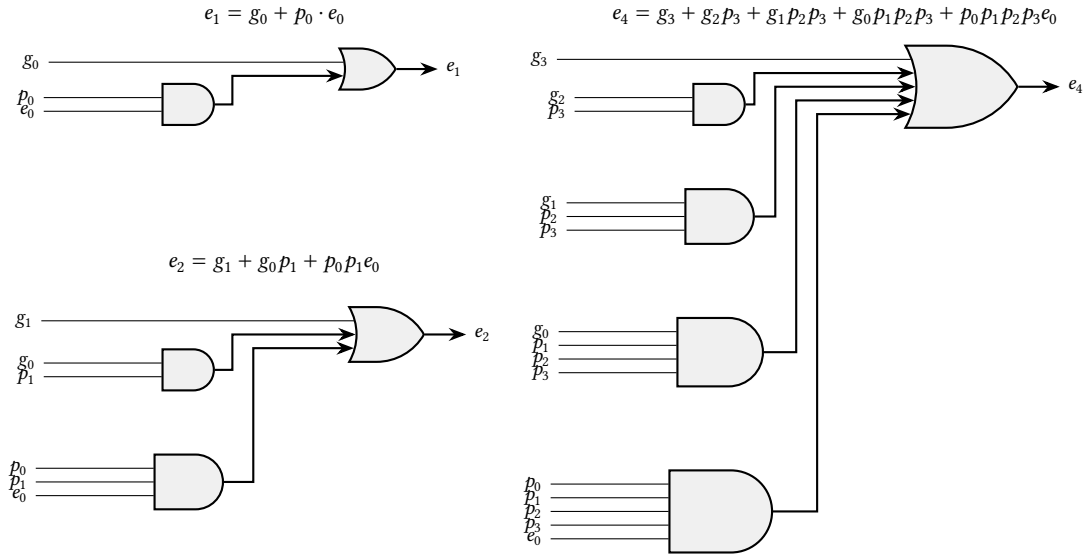
$$e_3 = g_2 + g_1 p_2 + g_0 p_2 p_1 + p_0 p_1 p_2 e_0 \quad (5.9)$$

$$e_4 = g_3 + g_2 p_3 + g_1 p_3 p_2 + g_0 p_3 p_2 p_1 + p_0 p_1 p_2 p_3 e_0 \quad (5.10)$$

Bu denklemlerin mantık kapılarıyla doğrudan gerçekleştirilmesi Şekil 5.11'de gösterilmiştir.

4 bitteki eldeler hesaplanırken hiçbir elde değeri bir öncekine bağlantılı değildir. Yalnızca hepsi ilk giren eldeye (e_0) bağlıdır ve ilk giren elde işlemin başında hazır bulunacağı için gecikmeye sebep olmaz.

Bu yaklaşımın hız kazancı sağlaması için önemli bir ön koşul vardır: çok girişli kapıların (örneğin 5 girişli VE veya VEYA kapısının) gecikmesi, aynı işlevi zincirleme bağlanmış çok



Şekil 5.11: Önceden elde üreten devrenin mantık kapılarıyla gerçekleştirilmesi (e_3 devresi e_2 ile benzer yapıdadır).

sayıda 2 girişli kapıyla gerçekleştirmenin gecikmesinden *az* olmalıdır. CMOS teknolojisinde çok girişli bir kapının gecikmesi giriş sayısı ile birlikte artar; ancak bu artış, aynı sonucu elde etmek için gereken zincirleme 2 girişli kapıların toplam gecikmesinden çok daha düşüktür. Örneğin 5 girişli bir VE kapısı tek bir kapı gecikmesinde sonuç üretirken, aynı işlevi dört adet zincirleme bağlanmış 2 girişli VE kapısıyla yapmak dört kapı gecikmesi gerektirir. Bu fark, önceden elde üretiminin temel üstünlüğünü oluşturur.

Ancak bir kapının giriş sayısı sınırsız değildir. Bir mantık kapısının sahip olabileceği en fazla giriş sayısı *kapı giriş sayısı kısıtı* (*fan-in*) olarak adlandırılır. CMOS teknolojisinde giriş sayısı arttıkça kapının gecikmesi ve alan tüketimi belirgin biçimde artar; belirli bir eşiğin ötesinde kapı güvenilir çalışmaz. Bu nedenle önceden elde üretimi doğrudan 32 veya 64 bit için uygulanamaz; e_{32} denklemi 33 girişli kapılar gerektirirdi ki bu uygulamada gerçekleştirilebilir değildir. Çözüm olarak önceden elde üretimi 4'er bitlik gruplar halinde yapılır ve gruplar arasında hiyerarşik bir yapı kurulur.

Bu yöntem pek çok mantıksal kapının kullanılmasını gerektirir ve donanımı karmaşıklarlaştırır.

Bu nedenle önceden elde üretici devreyi 4'er bitlik gruplar için yaparak gruplardan çıkan değerleri diğer 4'lü gruplara yönlendirecek bloklara ihtiyaç vardır; bu yapı Şekil 5.12'de gösterilmiştir. Blok düzeyinde fonksiyonlar tanımlanır:

$$P_0 = p_0 p_1 p_2 p_3 \quad (5.11)$$

$$G_0 = g_3 + g_2 p_3 + g_1 p_3 p_2 + g_0 p_3 p_2 p_1 \quad (5.12)$$

16 bitte işlem yapan bir toplayıcı için:

$$P_1 = p_4 p_5 p_6 p_7, \quad G_1 = g_7 + g_6 p_7 + g_5 p_7 p_6 + g_4 p_7 p_6 p_5 \quad (5.13)$$

$$P_2 = p_8 p_9 p_{10} p_{11}, \quad G_2 = g_{11} + g_{10} p_{11} + g_9 p_{11} p_{10} + g_8 p_{11} p_{10} p_9 \quad (5.14)$$

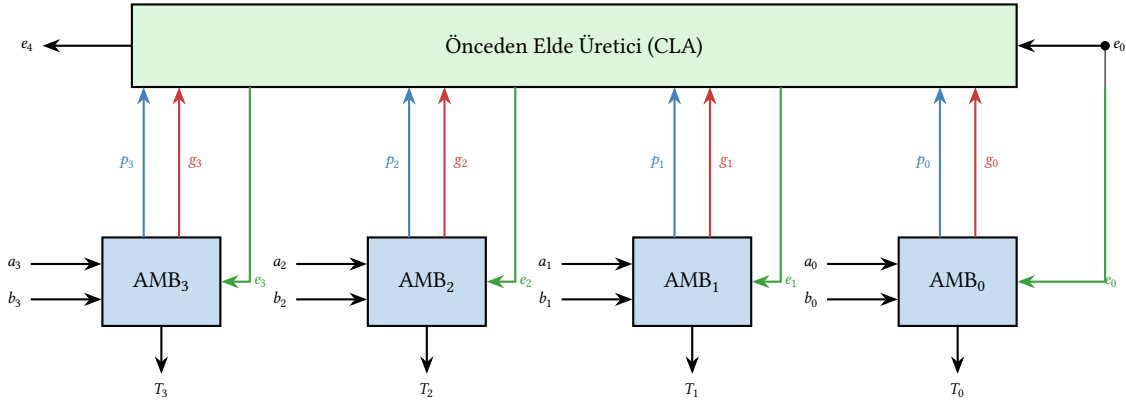
$$P_3 = p_{12} p_{13} p_{14} p_{15}, \quad G_3 = g_{15} + g_{14} p_{15} + g_{13} p_{15} p_{14} + g_{12} p_{15} p_{14} p_{13} \quad (5.15)$$

$$e_4 = G_0 + P_0 e_0 \quad (5.16)$$

$$e_8 = G_1 + P_1 e_4 \quad (5.17)$$

$$e_{12} = G_2 + P_2 e_8 \quad (5.18)$$

$$e_{16} = G_3 + P_3 e_{12} \quad (5.19)$$



Şekil 5.12: 4 Bitlik önceden elde üretici

Yapılacak olan AMB'nin eldeyi önceden üreten devrelerle kurulması, AMB'ye hız kazandırır. Çünkü dalga eldeli toplayıcıdaki gibi her basamakta elde değerinin hesaplanmasının beklemesi süresi yoktur. Bu gecikmelerin boru hattı tasarımına etkisi Bölüm 7'de incelenecektir.

5.3.8 Elde ile Seçmeli Toplayıcı (Carry Select Adder)

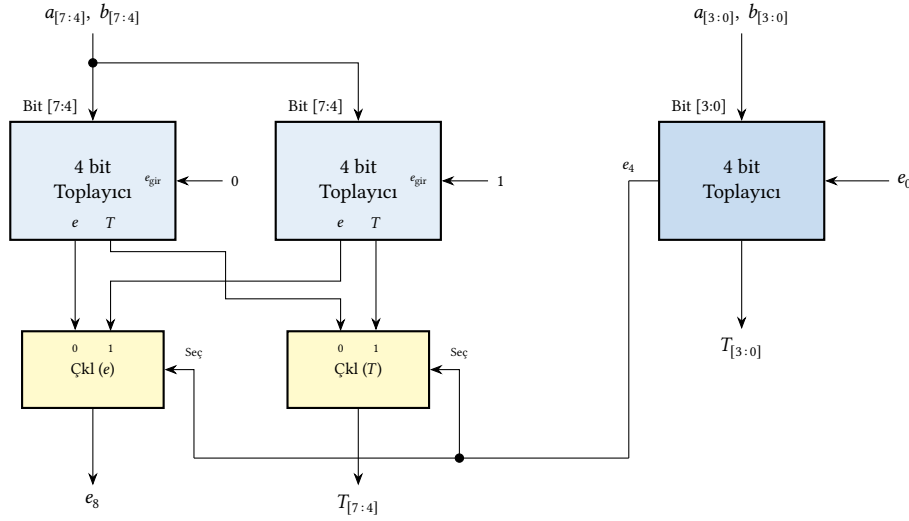
Önceden elde üreten toplayıcı, elde sinyalini mantıksal denklemlerle *önceden üreterek* hızlanma sağlar. Elde ile seçmeli toplayıcı ise farklı bir strateji izler: sonucu *önceden hesaplayarak* hızlanır. Temel fikir şudur: bir sonraki blok, önceki bloğun eldesini beklemek yerine her iki olasılığı da ($e_g = 0$ ve $e_g = 1$) koşut olarak hesaplar. Önceki bloğun eldesi hazır olduğunda bir çoklayıcıyla doğru sonuç seçilir.

Şekil 5.13, 8 bitlik bir elde ile seçmeli toplayıcıyı göstermektedir. İlk 4 bitlik blok normal çalışır. İkinci blok ise iki kopya halinde işlem yapar: biri $e_4 = 0$, diğeri $e_4 = 1$ varsayarak. İlk bloğun e_4 çıkışı hazır olduğunda çoklayıcı doğru toplamı ve eldeyi seçer.

Bilgi Notu 5.5: Güncel İşlemcilerde ve Yapay Zeka Hızlandırıcılarında Hızlı Toplayıcılar

Bu bölümde ele aldığımız önceden elde üreten ve elde ile seçmeli toplayıcıların ötesinde, daha da hızlı toplayıcı yapıları vardır. *Koşut önek toplayıcıları (parallel prefix adder)*, elde hesaplamasını bir ağaç yapısında koşut olarak gerçekleştirir ve $O(\log_2 n)$ gecikme sağlar. Bunların en bilinenleri:

- *Kogge-Stone*: En düşük gecikmeyi sağlar; 32 bit için yalnızca 5 düzey yeterlidir. Ancak $O(n \log n)$ kablo gerektirir; bu da yonga üzerinde yoğun kablo trafiğine yol açar.
- *Brent-Kung*: Kogge-Stone'a göre daha az kablo kullanır ($O(n)$) ama gecikme biraz artar ($2 \log_2 n - 1$ düzey). Alan ve hız arasında daha dengeli bir ödünleşme sunar.



Şekil 5.13: 8 bitlik elde ile seçmeli toplayıcı. İkinci blok, elde girişinin her iki olasılığını koşut hesaplar; e_4 hazır olduğunda çoklayıcı doğru sonucu seçer.

- *Han-Carlson:* Kogge-Stone ve Brent-Kung arasında bir uzlaşma vardır; hem kablo yoğunluğunu düşük tutar hem de gecikmeyi kabul edilebilir düzeyde bırakır. Uygulamada sıkça tercih edilen tasarımıdır.

Bu toplayıcılar günümüzün yüksek başarılı işlemcilerinde, grafik işlemcilerinde (GPU) ve yapay zeka hızlandırıcılarında kullanılan güncel tasarımlardır. Bir GPU'daki on binlerce çarpma-toplama (FMA) biriminin her birinde, son toplama aşaması koşut örnek toplayıcıyla gerçekleştirilir. Çarpıcılarda ise değiştirilmiş Booth kodlaması (radix-4) ile kısmi çarpım sayısı yarıya indirilir ve Wallace ya da Dadda ağaçlarıyla sıkıştırılır. Düşük hassasiyetli yapay zeka iş yüklerinde (BF16, FP8, INT8) toplayıcılar daha kısa bit genişliğinde çalıştığından aynı yonga alanına çok daha fazla birim sığdırılabilir; NVIDIA H100 GPU'nun 16.896 FMA birimi barındırabilmesinin temel nedeni budur.

Gecikme Karşılaştırması

Gecikme çözümlemesi için bir kapı gecikmesini t_g ile gösterelim. n bitlik bir dalga eldeli (ripple carry) toplayıcıda en kötü durum gecikmesi elde zincirinden gelir:

$$T_{dalga} = 2n \cdot t_g$$

Çünkü her tam toplayıcı eldeyi 2 kapı gecikmesinde ($2t_g$) üretir ve bu elde bir sonraki basamağa aktarılır.

Elde ile seçmeli toplayıcıda ise k bitlik bloklar kullanıldığında ilk blok $2k \cdot t_g$ 'de tamamlanır, ardından yalnızca bir çoklayıcı gecikmesi (t_{mux}) eklenir:

$$T_{seçmeli} = 2k \cdot t_g + \left(\frac{n}{k} - 1\right) \cdot t_{mux}$$

$t_{mux} \approx t_g$ kabul edilirse 8 bitlik toplayıcı için:

- Dalga eldeli: $T = 2 \times 8 \times t_g = 16 t_g$
- Elde ile seçmeli (4'erli blok): $T = 2 \times 4 \times t_g + 1 \times t_g = 9 t_g$

Gecikme neredeyse yarıya indi; ancak bunun bir bedeli vardır: ikinci blok iki kez kurulduğu için donanım alanı yaklaşık 1,5 katına çıkar. Bu, tasarım ilkelerinden *ödünleşmenin* güzel bir örneğidir: alandan vererek hız kazanılır. Bu toplayıcı türlerinin gecikme, alan ve karmaşıklık açısından karşılaştırması Tablo 5.7’de verilmiştir.

Tablo 5.7: *Toplayıcı türlerinin gecikme, alan ve karmaşıklık açısından karşılaştırması (n bit genişlik, k bit blok boyutu).*

Türkçe Ad	İngilizce	Strateji	Gecikme	Alan
Dalga eldeli	Ripple Carry	Elde bir sonraki basamağa dalgalanır	$O(n)$	$O(n)$
Önceden elde üreten	Carry Lookahead	Elde mantıksal denklemlerle önceden üretilir	$O(\log n)$	$O(n)$
Elde ile seçmeli	Carry Select	Her iki elde olasılığı koşut hesaplanır, sonuç seçilir	$O(\sqrt{n})$	$O(1,5n)$
Kogge-Stone	Kogge-Stone	Koşut önek ağacı, en düşük gecikme	$O(\log n)$	$O(n \log n)$
Brent-Kung	Brent-Kung	Koşut önek ağacı, az kablo	$O(\log n)$	$O(n)$

Örnek 5.3

Soru: 8 bitlik elde ile seçmeli toplayıcının gecikme ve alan karşılaştırmasını yapın. Dalga eldeli toplayıcıya göre ne kadar hızlanma sağlanır? Alan artışı ne kadardır?

Çözüm:

Gecikme:

- Dalga eldeli (8 bit): $T = 2 \times 8 \times t_g = 16 t_g$
- Elde ile seçmeli (4+4): İlk blok = $8 t_g$, çoklayıcı = $1 t_g \Rightarrow T = 9 t_g$
- Hızlanma: $16/9 \approx 1,78\times$

Alan (tam toplayıcı sayısı):

- Dalga eldeli: 8 tam toplayıcı
- Elde ile seçmeli: $4 + 4 + 4 = 12$ tam toplayıcı + 2 çoklayıcı
- Alan artışı: $\approx 1,5\times$ (çoklayıcılar dahil $\approx 1,6\times$)

5.4 Çarpma

AMB’imiz toplamayı ve çıkarmayı öğrendi; şimdi sıra çarpmaya geldi. Bir çarpma işleminde üç temel bileşen vardır:

- **Çarpılan** (*multiplicand*): çarpma işleminde tekrarlanan sayı,
- **Çarpan** (*multiplier*): çarpılanın kaç kez ve hangi konumda toplanacağını belirleyen sayı,
- **Çarpım** (*product*): işlemin sonucu.

Çarpmayı anlamanın en kolay yolu, ilkokulda öğrendiğimiz elde çarpma yöntemini ikili tabana uygulamaktır. Onluk tabanda çarpanın her bir basamağını çarpılanla çarpar ve sonucu uygun konumda alt alta yazarız; sonra tüm kısmi çarpımları toplarız. İkili tabanda bu işlem çok daha basittir: çarpanın her bir biti yalnızca 0 veya 1 olduğundan, kısmi çarpım ya sıfırdır

(bit = 0) ya da çarpılanın kendisidir (bit = 1). Aşağıdaki örnekte 7×5 çarpımı ikili tabanda gösterilmiştir:

$$\begin{array}{rcl}
 \text{Çarpılan:} & 0111 & \\
 \text{Çarpan:} & \times 0101 & \\
 \hline
 & 0111 & \leftarrow \text{bit 0 = 1, çarpılanı yaz} \\
 & 0000 & \leftarrow \text{bit 1 = 0, sıfır yaz} \\
 & 0111 & \leftarrow \text{bit 2 = 1, çarpılanı yaz} \\
 & 0000 & \leftarrow \text{bit 3 = 0, sıfır yaz} \\
 \hline
 \text{Çarpım:} & 0010011 & = 35
 \end{array}$$

Görüldüğü gibi ikili çarpma, özünde *kaydır ve topla* işleminden ibarettir: çarpanın her biti için çarpılanı uygun miktarda sola kaydırarak kısmi çarpımları oluşturur ve bunları toplarız. Bu yalın ilke, donanımda çarpma biriminin temelini oluşturur.

Çarpma Donanımına Neden İhtiyaç Var?

Bir işlemci toplama yapabiliyorsa, çarpmayı yazılımla da gerçekleştirebilir: çarpma, özünde arka arkaya toplama işlemine indirgenebilir. Nitekim RISC-V'in temel buyruk kümesi RV32I'da çarpma buyruğu yoktur; çarpma desteği isteğe bağlı **M uzantısı** ile eklenir. Peki bu durumda özelleşmiş çarpma donanımına neden ihtiyaç duyulur? Aşağıdaki iki RISC-V programını karşılaştıralım:

Tablo 5.8: Aynı çarpma işlemi (7×5): yalnızca toplama ile (yazılım) ve mul buyruğu ile (donanım).

Yalnızca toplama ile (M uzantısı yok)	mul buyruğu ile (M uzantısı var)
<pre> 1 # 7 x 5 = 7+7+7+7+7 2 li t0, 0 # sonuc = 0 3 li t1, 7 # carpilan 4 li t2, 5 # sayac (carpan) 5 dongu: 6 beqz t2, son # sayac=0? bitir 7 add t0, t0, t1 # sonuc += 7 8 addi t2, t2, -1 # sayac-- 9 j dongu # tekrarla 10 son: </pre>	<pre> 1 li t1, 7 # carpilan 2 li t2, 5 # carpan 3 mul t0, t1, t2 # sonuc = 7x5 </pre>
Durağan buyruk sayısı: 7	Durağan buyruk sayısı: 3
Devingen buyruk sayısı: 24	Devingen buyruk sayısı: 3
Çevrim sayısı (BBÇ ≈ 1): ~24	Çevrim sayısı: 3–5

Soldaki programda döngü 5 kez yinelenir (her yinelemede 4 buyruk) ve başlangıç buyrukları ile birlikte toplam 24 buyruk yürütülür. Çarpan büyüdükçe döngü sayısı doğrusal olarak artar; örneğin 1000×1000 için 4004, 32 bitlik en kötü durumda ise milyarlarca buyruk gerekir. Sağdaki programda ise aynı işlem *tek bir buyrukla* tamamlanır; mul buyruğu birkaç saat çevriminde sonucu üretir. Bölüm 2'deki yürütme zamanı formülünü hatırlayalım:

$$T_{\text{yürütme}} = N_{\text{buyruk}} \times \text{BBÇ} \times T_{\text{çevrim}}$$

Buyruk sayısının 24'ten 3'e düşmesi bile **8 kat** iyileşme demektir; büyük sayılarda bu fark astronomik boyutlara ulaşır. Bu dramatik fark, çarpma donanımının neden hemen her çağdaş işlemcide bulunduğunu açıklar: *buyruk kümesine eklenen tek bir buyruk, sorumluluğu yazılımdan donanıma aktararak hem programcıyı rahatlatır hem de sistem başarımını büyük ölçüde artırır.*

Aynı durum bölme için de geçerlidir ve Bölüm 5.5'de ele alınacaktır.

Çarpmada temel bir kural vardır: *m bitlik bir sayı ile n bitlik bir sayının çarpımı en fazla $m + n$ bit uzunluğundadır.* RISC-V'te (RV32I) veriler 32 bit olduğuna göre çarpım sonucu 64 bite kadar çıkabilir. Bu bölümde önce artı sayıların çarpımını göreceğiz; eksi sayılarla çarpma ise ikiye tümleyen dönüşümü veya Booth algoritması ile yapılır.

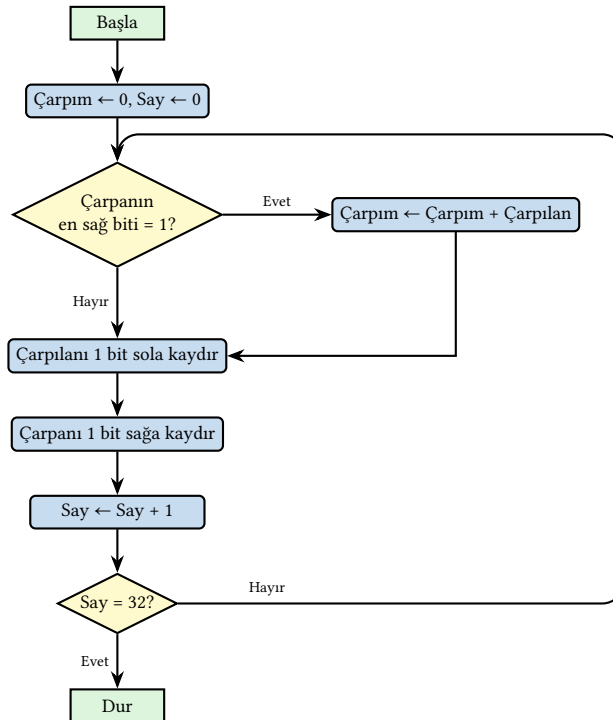
Çarpma yöntemlerini verimsizden verimliye doğru adım adım geliştireceğiz.

5.4.1 Kaydır-Topla Çarpma (İşaretsiz Sayılar)

İlk yöntemimiz “kaydır ve topla” (*shift-and-add*) ilkesine dayanır. Çarpanın her bir bitini sırayla sınavarak çarpılanı uygun konumda çarpıma ekleriz; tıpkı elde kâğıt üzerinde yapılan çarpmada olduğu gibi. Bu temel fikir üzerinde donanımı adım adım iyileştiren üç yol geliştireceğiz.

1. Yol

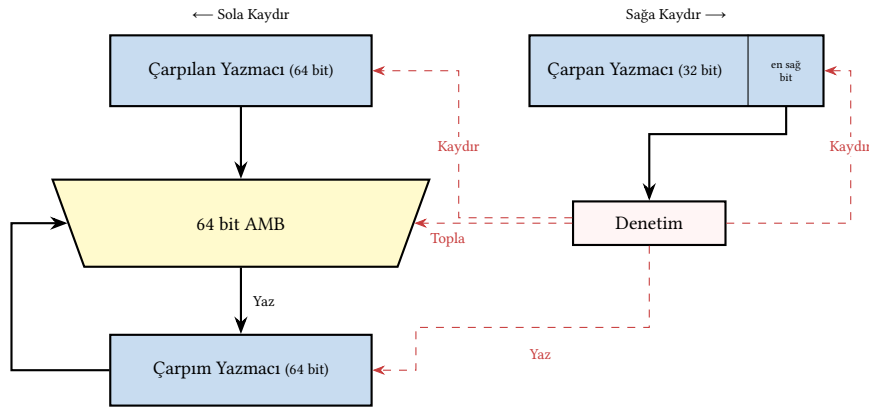
İlk yol, onluk tabanda yapılan normal çarpma işleminin doğrudan gerçekleştirilmesidir. Bu yolda 64 bitlik çarpılan ve çarpım yazmaçları ile 32 bitlik çarpan yazmacı kullanılır. Her adımda çarpanın en sağ bitine bakılır; bit 1 ise çarpılan çarpıma eklenir, 0 ise toplama atlanır. Ardından çarpılan 1 bit sola, çarpan 1 bit sağa kaydırılır. Çarpılanın sola kaydırılması, her adımda çarpılanın bir üst basamağa hizalanmasını sağlar; tıpkı elde çarpma yönteminde kısmi çarpımları bir basamak sola kaydırarak yazdığımız gibi.



Şekil 5.14: Kaydır-Topla Çarpma (1. Yol) akış şeması.

1. yolda kaydırma ve toplama işlemleri kullanılmıştır; algoritmanın akış şeması Şekil 5.14’da, buna karşılık gelen donanım tasarımı ise Şekil 5.15’de gösterilmiştir. Donanım kaydırıcı ve toplayıcıların dışında, 2 tane 64 bitlik ve bir tane de 32 bitlik yazmaçtan oluşur. Toplayıcının kullanılacağı AMB de 64 bit ile işlem yapabilmek için 64 bitlik bir AMB olmalıdır. 64 bitlik AMB, 32 bitlik olandan çok daha karmaşıktır ve yavaştır.

Her yinelemede yapılan işlemler şunlardır: (1) çarpmanın en sağ bitini sına, (2) bit 1 ise çarpılanı çarpıma ekle, (3) çarpılanı sola, çarpmanı sağa kaydır. Burada iki kaydırma birbirinden bağımsızdır (biri çarpılan yazmacını, diğeri çarpman yazmacını etkiler); bu yüzden donanımda *koşut* (aynı anda) yapılabilirler. Dolayısıyla her yineleme aslında üç aşamadan oluşur: test, toplama ve kaydırma. Denetim birimi içindeki bir sayaç yineleme sayısını takip eder: 32 bitlik bir RISC-V çarpmasında bu döngü 32 kez tekrarlanır ve sayaç 32’ye ulaştığında işlem sonlandırılır. Dolayısıyla çarpma işlemi toplam **32 saat vuruşu** sürer (toplamayı, testi ve kaydırmayı tek bir vuruşta yapabildiğimiz varsayımıyla).



Şekil 5.15: Kaydır-Topla Çarpma (1. Yol) donanımı

1. yolun asıl sorunu toplama adımının 64 bitlik AMB ile yapılmak zorunda olmasıdır: 64 bitlik bir toplama, 32 bitlik bir toplamaya göre belirgin şekilde daha yavaştır, daha fazla alan kaplar ve daha çok güç tüketir. Yöntemin iyileştirilmesi gerekir.

Örnek 5.4

Soru: $7 \times 5 = ?$ ($0111 \times 0101 = ?$)

Çözüm: Adımları tek tek izleyelim (Tablo 5.9):

- **Başlangıç:** Çarpılan = 0111, çarpman = 0101, çarpım = 0.
- **Adım 1:** Çarpmanın en sağ biti = 1 $\Rightarrow$ çarpılanı çarpıma ekle: $0 + 0111 = 0111$. Çarpılanı sola, çarpmanı sağa kaydır.
- **Adım 2:** Çarpmanın en sağ biti = 0 $\Rightarrow$ toplama yok. Çarpılanı sola, çarpmanı sağa kaydır.
- **Adım 3:** Çarpmanın en sağ biti = 1 $\Rightarrow$ çarpılanı çarpıma ekle: $00000111 + 000011100 = 00100011$. Kaydır.
- **Adım 4:** Çarpmanın en sağ biti = 0 $\Rightarrow$ toplama yok. Kaydır. Çarpım değişmez.

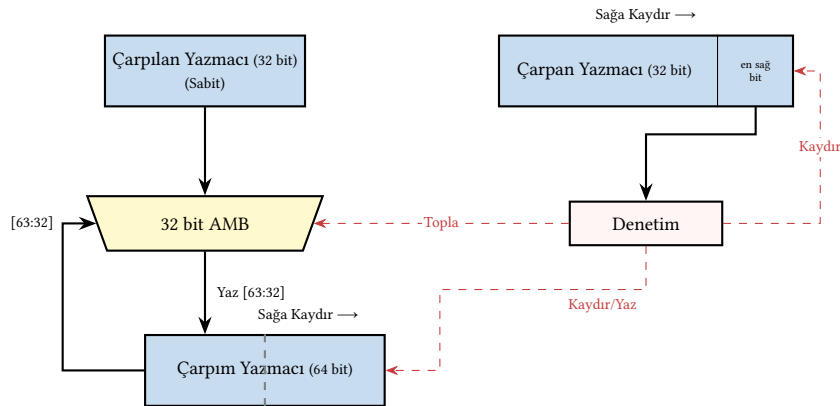
Tablo 5.9: Örnek 5.4 – 1. yol çarpma algoritması

Adım	Çarpılan	Çarpan	Çarpım
Başlangıç	00000111	0101	00000000
1 (Topla + Kaydır)	00001110	0010	00000111
2 (Kaydır)	00011100	0001	00000111
3 (Topla + Kaydır)	00111000	0000	00100011
4 (Kaydır)	01110000	0000	00100011

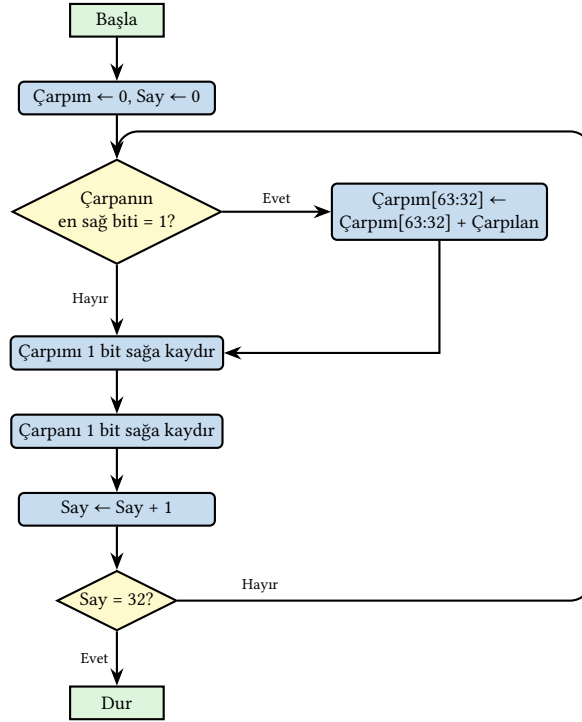
Çarpım = $(00100011)_2 = 35$

2. Yol

1. yoldaki donanıma dikkatli bakıldığında bir verimsizlik göze çarpar: çarpılan yazmacı 64 bit genişliğinde olmasına karşın, herhangi bir anda yalnızca 32 biti anlamlı veri taşır; kalan bitler sıfırdır. Çarpılanın sola kaydırılması, AMB'nin de 64 bit genişliğinde olmasını zorunlu kılmaktadır; oysa her toplama işleminde yalnızca 32 bitlik bir değer toplanır. Buradaki kilit gözlem şudur: çarpılanı sola kaydırmak yerine, çarpımı sağa kaydırmak aynı hizalama etkisini yaratır. Çarpım sağa kaydırıldığında, çarpılanın her zaman çarpım yazmacının üst yarısıyla (sol 32 bit) toplanması yeterli olur.

**Şekil 5.16:** Kaydır-Topla Çarpma (2. Yol) donanımı

2. yol, bu gözlemden yararlanarak donanımı sadeleştirir (Şekil 5.16, akış şeması Şekil 5.17): çarpılan yazmacı artık 32 bitte kalır, AMB 32 bite iner ve toplama sonucu çarpım yazmacının üst yarısına yazılır. Her adımda çarpım yazmacının tamamı sağa kaydırılarak kısmi çarpım doğru konuma yerleşir. Sonuç olarak, aynı işlevsellik yarı boyuttaki bir AMB ile sağlanmış olur; bu da hem alan hem de güç tüketimi açısından önemli bir kazanımdır. Her yinelemede yapılan işlem sayısı (1 test, 1 toplama, kaydırmalar) 1. yol ile benzerdir ve döngü yine 32 kez tekrarlanır; ancak toplama artık 32 bitlik AMB ile yapıldığından her adım daha hızlıdır ve donanım daha az alan kaplar.



Şekil 5.17: Kaydır-Topla Çarpma (2. Yol) akış şeması.

Örnek 5.5

Soru: $7 \times 7 = ?$ ($0111 \times 0111 = ?$)

Çözüm: 2. yolda çarpılan (32 bit) sabittir; her adımda çarpanın en sağ bitine bakılır, bit 1 ise çarpılan çarpımın üst yarısına eklenir. Ardından çarpım yazmacının tamamı 1 bit sağa kaydırılır. Adımları izleyelim (Tablo 5.10):

- **Başlangıç:** Çarpılan = 0111, çarpan = 0111, çarpım = 0.
- **Adım 1:** Çarpanın en sağ biti = 1 $\Rightarrow$ çarpılanı üst yarıya ekle: $0000 + 0111 = 0111$. Sağa kaydır: 00111000.
- **Adım 2:** En sağ bit = 1 $\Rightarrow$ üst yarıya ekle: $0011 + 0111 = 1010$. Sağa kaydır: 01010100.
- **Adım 3:** En sağ bit = 1 $\Rightarrow$ üst yarıya ekle: $0101 + 0111 = 1100$. Sağa kaydır: 01100010.
- **Adım 4:** En sağ bit = 0 $\Rightarrow$ toplama yok. Sağa kaydır: 00110001.

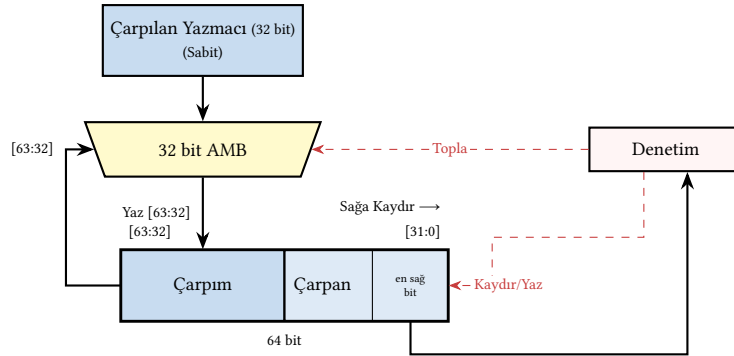
Tablo 5.10: Örnek 5.5 – 2. yol çarpma algoritması

Adım	Çarpılan	Çarpan	Çarpım
Başlangıç	0111	0111	00000000
1 (Topla + Kaydır)	0111	0011	00111000
2 (Topla + Kaydır)	0111	0001	01010100
3 (Topla + Kaydır)	0111	0000	01100010
4 (Kaydır)	0111	0000	00110001

Çarpım = $(00110001)_2 = 49$

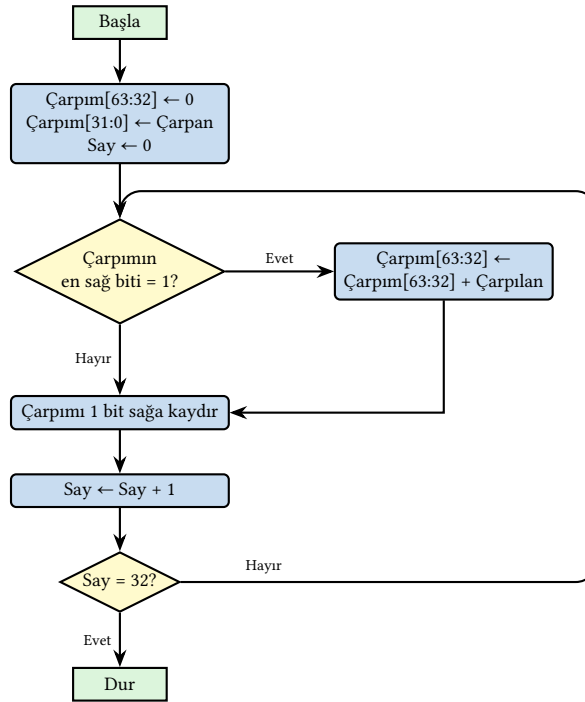
3. Yol

2. yoldaki çarpma sürecini adım adım izlediğimizde ilginç bir örüntü ortaya çıkar: çarpım yazmacının 64 bitlik alanı başlangıçta tamamen sıfırdır; her adımda toplama sonucu üst yarıya (sol 32 bit) yazılıp yazmaç sağa kaydırıldıkça, alt yarı (sağ 32 bit) yavaş yavaş kısmi çarpım bitleriyle dolmaya başlar. Öte yandan çarpan yazmacı da her adımda sağa kaydırılarak tüketilir: en sağdaki bit denetim birimine gönderilir ve kaydırmayla atılır; dolayısıyla çarpandaki anlamlı bit sayısı her adımda bir azalır.



Şekil 5.18: Kaydır-Topla Çarpma (3. Yol) donanımı

Buradaki kritik gözlem şudur: çarpım yazmacının alt yarısı dolarken, çarpan yazmacı aynı hızda boşalmaktadır. Her ikisi de sağa kaydırıldığından, çarpanın kalan bitleri çarpım yazmacının sağ yarısında tutulabilir. Böylece ayrı bir çarpan yazmacına gerek kalmaz; çarpan, çarpım yazmacının başlangıçta boş olan alt yarısına yerleştirilir ve her adımda birlikte sağa kaydırılır. Bu tasarım, 32 bitlik bir yazmacı ortadan kaldırarak donanımı daha da yalınlaştırır; 3. yolun donanım yapısı Şekil 5.18’de, akış şeması ise Şekil 5.19’da gösterilmiştir.



Şekil 5.19: Kaydır-Topla Çarpma (3. Yol) akış şeması.

Örnek 5.6

Soru: $6 \times 3 = ?$ ($0110 \times 0011 = ?$)

Çözüm: 3. yolda çarpan, çarpım yazmacının alt yarısında [31:0] tutulur. Her adımda birleşik yazmacın en sağ bitine (çarpanın en sağ biti) bakılır; bit 1 ise çarpılan üst yarıya [63:32] eklenir. Ardından yazmacın tamamı 1 bit sağa kaydırılır. Adımları izleyelim (Tablo 5.11):

- **Başlangıç:** Çarpılan = 0110. Birleşik yazmaç: üst yarı = 0000, alt yarı = 0011 (çarpan).
- **Adım 1:** En sağ bit = 1 $\Rightarrow$ üst yarıya ekle: $0000 + 0110 = 0110$. Tümünü sağa kaydır: 0011 | 0001.
- **Adım 2:** En sağ bit = 1 $\Rightarrow$ üst yarıya ekle: $0011 + 0110 = 1001$. Sağa kaydır: 0100 | 1000.
- **Adım 3:** En sağ bit = 0 $\Rightarrow$ toplama yok. Sağa kaydır: 0010 | 0100.
- **Adım 4:** En sağ bit = 0 $\Rightarrow$ toplama yok. Sağa kaydır: 0001 | 0010. Çarpan tükendi.

Tablo 5.11: Örnek 5.6 – 3. yol çarpma algoritması

Adım	Çarpılan	Çarpım [63:32] [31:0]=Çarpan
Başlangıç	0110	0000 0011
1 (Topla + Kaydır)	0110	0011 0001
2 (Topla + Kaydır)	0110	0100 1000
3 (Kaydır)	0110	0010 0100
4 (Kaydır)	0110	0001 0010

Çarpım = $(00010010)_2 = 18$

3. yol, kaydır-topla çarpma için en yalın donanım tasarımını sunar: yalnızca bir adet 32 bitlik çarpılan yazmacı, bir adet 32 bitlik AMB ve bir adet 64 bitlik birleşik çarpım/çarpan yazmacı yeterlidir. Yineleme başına yapılan işlemler 2. yol ile aynıdır; kazanım çevrim sayısında değil, bir yazmaç daha az kullanarak donanımın yalınlaşmasındadır.

Karşılaştırma ve RISC-V Çarpma Buyrukları

Üç yolun karşılaştırması Tablo 5.12’de özetlenmiştir. Her üç yolda da döngü 32 kez tekrarlanır ve her yineleme bir saat vuruşunda tamamlanabilir; dolayısıyla 32 bitlik bir çarpma 32 saat vuruşu sürer. Yollar arasındaki temel fark, her vuruşun ne kadar hızlı olabileceği (AMB genişliği) ve donanımın ne kadar yer kapladığıdır (yazmaç sayısı).

Tablo 5.12: Kaydır-topla çarpma yollarının karşılaştırması (32 bitlik RISC-V)

	AMB genişliği	Yazmaç sayısı	Yineleme başına işlem
1. Yol	64 bit	3 (64+32+64 bit)	test + toplama + 2 kaydırma*
2. Yol	32 bit	3 (32+32+64 bit)	test + toplama + 2 kaydırma*
3. Yol	32 bit	2 (32+64 bit)	test + toplama + kaydırma

*İki kaydırma birbirinden bağımsızdır ve koşut yapılabilir.

Bölüm 3.8.1’de gördüğümüz gibi, RV32I’de yazmaçlar 32 bit genişliğindedir; oysa iki 32

bitlik sayının çarpımı 64 bite kadar çıkabilir. 64 bitlik sonucu tek bir 32 bitlik yazmaca yazmak mümkün olmadığından, RISC-V M uzantısı çarpma sonucunu *iki ayrı buyrukla* elde eder: `mul` buyruğu sonucun alt 32 bitini (`çarpım[31:0]`), `mulh` buyruğu ise üst 32 bitini (`çarpım[63:32]`) hedef yazmacına yazar. Her iki buyruk da donanımda bağımsız birer çarpma işlemi gerçekleştirir; dolayısıyla tam 64 bitlik sonuca ihtiyaç duyan bir program bu iki buyruğu art arda çalıştırmalıdır ve toplam maliyet iki çarpma süresi olur. İşaretsiz çarpmanın üst yarısı için `mulhu`, bir işaretli bir işaretsiz çarpmanın üst yarısı için ise `mulhsu` buyruğu kullanılır (bu buyrukların kullanım örneği için bkz. Örnek 3.8.1).

5.4.2 Booth Algoritması (İşaretli Sayılar)

Kaydır-topla çarpma yalnızca işaretsiz sayılarla çalışır. İşaretli sayıları çarpmak için bir yol, işaretleri ayrı tutup işaretsiz çarpma yapmak ve sonuca doğru işareti eklemektir; ancak bu ek adımlar gerektirir. Booth algoritması çok daha zarif bir çözüm sunar: *ikiye tümleyen gösterimindeki sayıları (ister artı ister eksi) hiçbir ayırım yapmadan doğrudan çarpar*. Ayrı bir işaret düzeltilmesine veya farklı bir donanıma gerek kalmaz.

Booth algoritmasının çıkış noktası, ikili sayılardaki *ardışık bit örüntüleridir*. Kaydır-topla yönteminde çarpanın her 1 biti bir toplama gerektirir. Ancak gerçek sayılarda ardışık 1'ler sıkça görülür; örneğin:

- $30 = \underbrace{011110}_{\text{dört ardışık 1}}$; kaydır-topla ile 4 toplama gerekir,
- $126 = 0 \underbrace{1111111}_{\text{yeddi ardışık 1}} 0$; kaydır-topla ile 7 toplama gerekir.

Booth'un gözlemi şudur: ardışık k tane 1 içeren bir sayı, yalnızca *iki* işlemle (bir çıkarma ve bir toplama ile) ifade edilebilir. Örneğin $0111110_2 = 1000000_2 - 0000010_2$ olduğundan, 5 toplama yerine 1 çıkarma + 1 toplama yeterlidir. Algoritma, çarpanın bitlerini ikiyeşerli çiftler halinde tarayarak ardışık 1'lerin *başladığı* yerde çıkarma, *bittiği* yerde toplama yapar; ardışık 0'lar için ise hiçbir işlem gerekmez. Bu şekilde toplama/çıkarma sayısı önemli ölçüde azalır.

Booth algoritmasının işlem sayısı, çarpanın bit örüntüsüne doğrudan bağlıdır:

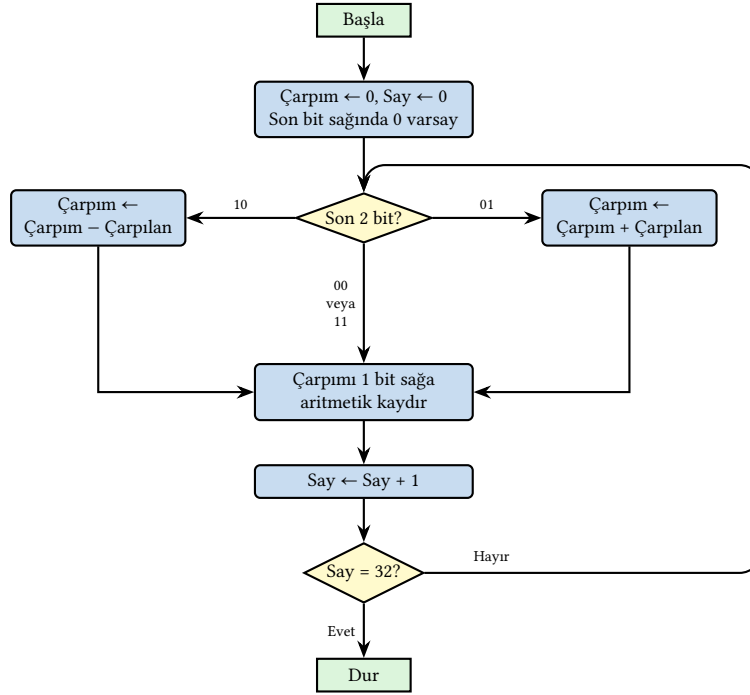
- **En iyi durum:** Çarpanda uzun ardışık 0 veya uzun ardışık 1 blokları varsa, toplama/çıkarma sayısı çok düşer. Örneğin 0111111110_2 gibi bir çarpan için yalnızca 2 işlem (1 çıkarma + 1 toplama) yeterlidir; kaydır-topla ile 9 toplama gerekirdi.
- **En kötü durum:** Çarpanın bitleri sürekli değişiyorsa (...010101...), her bit geçişi bir işlem tetikler ve toplama/çıkarma sayısı kaydır-topla yönteminden bile fazla olabilir. Örneğin 010101010_2 çarpanı için her adımda bir işlem gerekir.

Bu nedenle Booth algoritmasının başarımı *veriye bağlıdır (data-dependent)*: aynı donanım, çarpanın değerine göre farklı sayıda aritmetik işlem yapar. Bu durum, sabit gecikmeli tasarım gerektiren boru hatlı işlemcilerde (bkz. Bölüm 7) ek zorluklar yaratabilir. Algoritmanın kuralları Tablo 5.13'de listelenmiştir, akış şeması ise Şekil 5.20'da gösterilmiştir.

Bilgi Notu 5.6: Booth Algoritmasının Tarihçesi

Andrew D. Booth bu algoritmayı 1950 yılında, Birkbeck Koleji'nde masaüstü hesap makineleri üzerinde çalışırken geliştirmiştir. Orijinal yöntem ardışık 1 dizilerindeki gereksiz toplama işlemlerini ortadan kaldırarak çarpma süresini kısaltır.

Algoritmanın en önemli özelliği işaretli sayılarla işlem yapabilmesidir. Bunun için kaydır-topla yöntemindeki *mantıksal* sağa kaydırma yerine *aritmetik* sağa kaydırma kullanılır. Man-



Şekil 5.20: Booth Çarpım Algoritması akış şeması.

Tablo 5.13: Booth Algoritması Kuralları

Bit Çifti	İşlem
10	1 dizisi başlar → çarpılan çıkarılır, sağa kaydırma
11	1 dizisinin ortası → sağa kaydırma
01	1 dizisi biter → çarpılan toplanır, sağa kaydırma
00	0 dizisinin ortası → sağa kaydırma

tıksal kaydırmada soldan her zaman 0 girer; aritmetik kaydırmada ise en soldaki bit (işaret biti) korunarak tekrarlanır. Böylece eksi bir kısmi çarpım sağa kaydırıldığında işareti bozulmaz: örneğin **1101** aritmetik sağa kaydırılırsa **1110** olur; en soldaki 1 korunmuştur. Bu sayede ikiye tümleyen gösterimindeki eksi ara sonuçlar kaydırma sırasında doğru kalır. Ek olarak, en sağ basamağın da sağında 0 olduğu farz edilir.

Booth algoritması donanım açısından çarpma 3. yolunun (Şekil 5.18) üzerine kuruludur: aynı birleşik çarpım/çarpan yazmacı, aynı 32 bitlik çarpılan yazmacı ve aynı AMB kullanılır. İki temel fark vardır: (1) AMB yalnızca toplama değil, çıkarma da yapabilmelidir ve (2) denetim birimi çarpanın en sağ bitini tek başına değil, bir önceki bitle birlikte iki bitlik çift olarak değerlendirir (Tablo 5.13). Döngü sayısı değişmez: n bitlik çarpan için yine n adım gerekir.

Örnek 5.7

Soru: $3 \times (-4) = ?$ Booth algoritmasıyla 4 bitlik işaretli sayılarla çözün.

Çözüm: İkiye tümleyen gösterimde çarpılan = **0011** (3), çarpan = **1100** (-4). Booth algoritmasında çarpanın sağına başlangıçta 0 eklenir; her adımda en sağdaki iki bit (mevcut bit ve önceki bit) incelenir. Adımları izleyelim:

- **Başlangıç:** Çarpım üst yarı = 0000, çarpan = **11000** (eklenen bit).

- **Adım 1:** En sağ iki bit = 00 $\Rightarrow$ işlem yok (Tablo 5.13). Aritmetik sağa kaydır.
- **Adım 2:** En sağ iki bit = 00 $\Rightarrow$ işlem yok. Aritmetik sağa kaydır.
- **Adım 3:** En sağ iki bit = 10 $\Rightarrow$ ardışık 1'lerin başlangıcı, çarpılanı çıkar: 0000 – 0011 = 1101. Aritmetik sağa kaydır.
- **Adım 4:** En sağ iki bit = 11 $\Rightarrow$ ardışık 1'lerin ortası, işlem yok. Aritmetik sağa kaydır.

Adım	Çarpılan	Çarpım (üst)	Çarpan	Bit çifti	İşlem
Başla	0011	0000	11000	—	—
1	0011	0000	11000	00	İşlem yok, aritmetik sağa kaydır
		0000	01100		
2	0011	0000	01100	00	İşlem yok, aritmetik sağa kaydır
		0000	00110		
3	0011	0000	00110	10	Çıkar: 0000 – 0011 = 1101, kaydır
		1110	10011		
4	0011	1110	10011	11	İşlem yok, aritmetik sağa kaydır
		1111	01001		

Sonuç: 11110100₂ = -12₁₀. Doğrulama: 3 × (-4) = -12 ✓. Booth algoritması *aritmetik* sağa kaydırma (işaret bitini koruyarak) kullandığından, ikiye tümleyen sayılarla doğrudan çarpım yapılabilir; ayrı bir işaret düzeltmesine gerek kalmaz.

5.4.3 Değiştirilmiş Booth Kodlaması (Radix-4)

Bir önceki bölümde gördüğümüz gibi, orijinal Booth algoritmasının başarımı veriye bağlıdır: çarpandaki bitler sürekli 0-1 arasında değişiyorsa (... 010101 ...), her adımda bir toplama veya çıkarma tetiklenir ve kaydır-topla yönteminden bile yavaş kalınabilir. Üstelik Booth algoritması kısmi çarpım sayısını azaltsa da, n bitlik bir çarpan yine n adım gerektirir; her adımda yalnızca 1 bit işlenir.

Tablo 5.14: Radix-4 Booth kodlama tablosu (M = çarpılan)

Bit üçlüsü	Kısmi çarpım	Açıklama
000	0	Toplama yok
001	+ M	Çarpılanı ekle
010	+ M	Çarpılanı ekle
011	+2 M	Çarpılanı 1 bit sola kaydırıp ekle
100	-2 M	Çarpılanı 1 bit sola kaydırıp çıkar
101	- M	Çarpılanı çıkar
110	- M	Çarpılanı çıkar
111	0	Toplama yok

Değiştirilmiş Booth kodlaması (modified Booth encoding, radix-4) bu iki sorunu birden çözer. Her adımda çarpanın 1 biti yerine **2 bitini** birden işleyerek adım sayısını yarıya indirir: 32 bitlik bir çarpan artık 32 değil **16 adımda** tamamlanır. Bunu başarmak için çarpanın bitleri 3'lük örtüşen gruplar halinde incelenir. Her üçlü, Tablo 5.14'de gösterildiği gibi beş olası kısmi

çarpımdan birini üretir.

Bu kodlamanın en büyük kazanımı, n bitlik bir çarpan için yalnızca $n/2$ kısmi çarpım üretmesidir; orijinal Booth'a göre adım sayısı yarıya iner. $+2M$ ve $-2M$ değerleri kolay bir sola kaydırma ile elde edilebildiğinden ek donanım maliyeti düşüktür.

Günümüzde neredeyse tüm yüksek başarımli çarpma birimleri radix-4 Booth kodlamasını kısmi çarpım üretimi için kullanır. Daha yüksek tabanlı kodlamalar (radix-8, radix-16) da mevcuttur ancak $\pm 3M$ gibi değerlerin üretilmesi ek toplayıcı gerektirdiğinden radix-4 en yaygın tercih olmaya devam eder.

Örnek 5.8

Soru: Radix-4 Booth kodlamasını kullanarak 13×11 çarpımını 6 bitlik işaretli sayılarla gerçekleştirin. ($13 = 001101_2$, $11 = 001011_2$)

Çözüm: Çarpan = 001011_2 (11), çarpılan $M = 001101_2$ (13).

Adım 1 – Çarpanı 3'erli örtüşen gruplara ayır. 6 bitlik çarpan için $n/2 = 3$ grup oluşturulur. Her grup, önceki grubun en düşük anlamlı bitiyle örtüşür; başlangıçta sağa sıfır (0) eklenir:

Çarpan bitleri: $b_5 b_4 b_3 b_2 b_1 b_0 = 0 0 1 0 1 1$, $b_{-1} = 0$ (eklenen sıfır).

Grup	Bit üçlüsü ($b_{2i+1} b_{2i} b_{2i-1}$)	Kısmi çarpım	Ağırlık
Grup 0 ($i = 0$)	$b_1 b_0 b_{-1} = 1 1 0$	$-M = -13$	$2^0 = 1$
Grup 1 ($i = 1$)	$b_3 b_2 b_1 = 1 0 1$	$-M = -13$	$2^2 = 4$
Grup 2 ($i = 2$)	$b_5 b_4 b_3 = 0 0 1$	$+M = +13$	$2^4 = 16$

Kodlama tablosuna göre: üçlü 110 $\rightarrow -M$, üçlü 101 $\rightarrow -M$, üçlü 001 $\rightarrow +M$.

Adım 2 – Kısmi çarpımları hizalayarak topla. Her kısmi çarpım, grubun ağırlığıyla çarpılarak hizalanır:

$$\begin{array}{rcl}
 & -13 & \text{(Grup 0: } -M \times 2^0 = -13) \\
 + & -52 & \text{(Grup 1: } -M \times 2^2 = -13 \times 4 = -52) \\
 + & +208 & \text{(Grup 2: } +M \times 2^4 = +13 \times 16 = +208) \\
 \hline
 & 143 &
 \end{array}$$

Sonuç: $-13 + (-52) + 208 = 143 = 13 \times 11$. ✓

Radix-4 Booth, 6 bitlik çarpan için yalnızca 3 kısmi çarpım üretmiştir; orijinal Booth'un 6 adımı yerine 3 adım yeterli olmuştur.

5.4.4 Koşut Çarpıcılar

Şimdiye kadar incelediğimiz tüm çarpma yöntemleri *yinelemeli* yöntemlerdir: yazmaçlar, saat sinyali ve denetim birimi içeren *ardışıl devrelerdir* (*sequential circuits*); her saat çevriminde bir kısmi çarpım işlenir, n bitlik çarpma için n (veya radix-4 ile $n/2$) çevrim gerekir.

Bu bölümde ele alacağımız koşut çarpıcılar ise temelden farklıdır: bunlar saat sinyali gerektirmeyen *birleşik devrelerdir* (*combinational circuits*). Girdiler (çarpılan ve çarpan) devreye verildiği anda, çıkış (çarpım) belirli bir *yayılım gecikmesi* sonrasında hazır olur; herhangi bir saat vuruşu beklenmez. Gecikme yalnızca kapıların ve bağlantıların fiziksel yayılım süresine bağlıdır.

Birleşik çarpıcının toplam gecikmesi tek bir saat çevrimine sığıyorsa çarpma *tek saat vuru-*

şunda tamamlanır. Sığmıyorsa (ki yüksek saat hızlarında bu sık görülür), devrenin arasına yazmaçlar eklenerek birden fazla aşamaya bölünür ve çarpma birkaç saat vuruşunda tamamlanır. Bu teknik, Bölüm 7’de ayrıntılı olarak ele alınacak olan *boru hattı* yapısının bir uygulamasıdır.

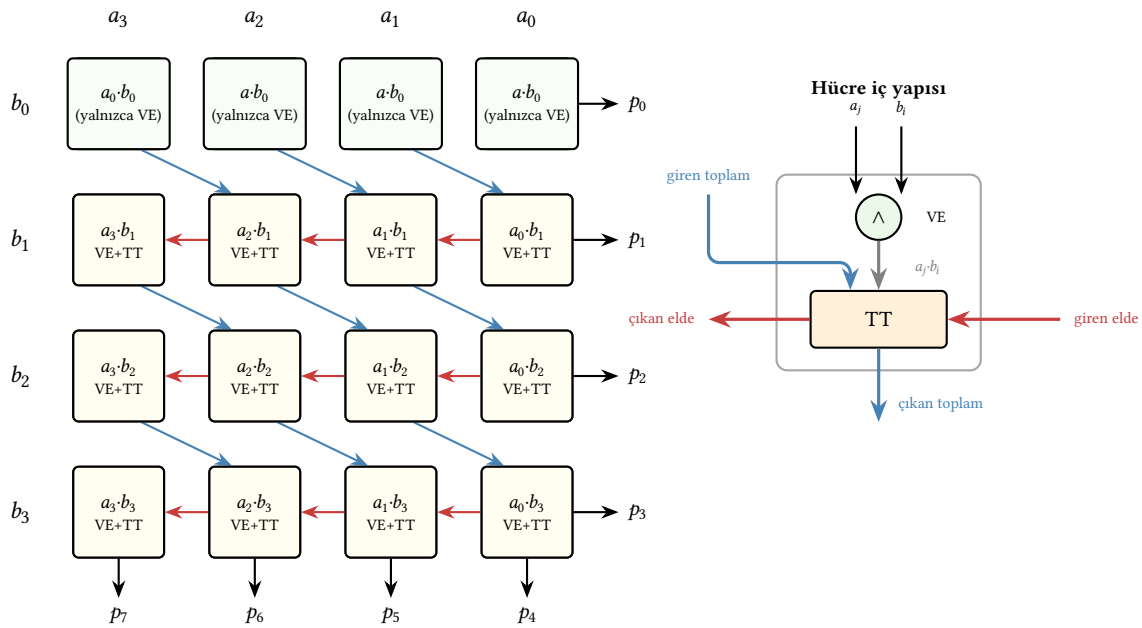
Terim Kökenleri

birleşik devre (*combinational circuit*): Çıkışı yalnızca o anki girdilerin *birleşimine* (kombinasyonuna) bağlı olan devredir. Bellek ögesi (yazmaç, flip-flop) içermez; saat sinyali gerektirmez. Girdiler değiştiğinde çıkış, kapıların yayılım gecikmesi kadar bir süre sonra yeni değerine ulaşır. Toplayıcılar, çoğullayıcılar ve bu bölümde incelenen koşut çarpıcılar birleşik devrelere örnektir.

ardışıl devre (*sequential circuit*): Çıkışı hem o anki girdilere hem de devrenin *önceki durumuna* (belleğine) bağlı olan devredir. Flip-flop veya yazmaç gibi bellek ögeleri içerir ve bir saat sinyaliyle yönetilir. Her saat vuruşunda devre bir sonraki durumuna geçer. Kaydır-topla çarpıcılar, sayaçlar ve işlemcinin denetim birimi ardışıl devrelere örnektir.

Dizi Çarpıcı

Dizi çarpıcı (*array multiplier*), kaydır-topla algoritmasının doğrudan donanıma çevrilmiş halidir. Her kısmi çarpım biti, çarpılanın ilgili biti ile çarpanın ilgili bitinin VE işlemiyle üretilir; ardından bu kısmi çarpımlar bir ızgara düzeninde tam toplayıcılarla birbirine eklenir. $n \times n$ bitlik bir çarpma için n^2 adet VE kapısı ve yaklaşık $n \times (n - 1)$ adet tam toplayıcı kullanılır.



Şekil 5.21: 4 bitlik dizi çarpıcının ızgara yapısı. Her hücrede bir $\wedge$ (VE kapısı) kısmi çarpım bitini üretir; altındaki TT (tam toplayıcı) bu biti, üst satırdan gelen toplamla ve sağ komşudan gelen eldeyle toplar. Her hücrenin iç yapısında **gri ok** VE kapısından tam toplayıcıya giden kısmi çarpım bitini, **mavi oklar** üst satırdan gelen toplamı, **kırmızı oklar** ise elde yayılımını (sağdan sola) gösterir.

Şekil 5.21, 4 bitlik bir dizi çarpıcının yapısını göstermektedir. En üst satırda kısmi çarpım bitleri ($a_i \cdot b_0$) doğrudan çıkışa verilir. Her sonraki satırda bir üst satırdan gelen toplam ve elde bitleri, o satırın kısmi çarpım bitleriyle ($a_i \cdot b_j$) tam toplayıcılarda toplanır. Eldeler bir sonraki sütuna, toplamlar aşağıya aktarılır.

Dizi çarpıcının üstünlüğü yapısının düzenli ve tasarımının yalın olmasıdır; VLSI yerleşiminde düzgün bir ızgara oluşturur. Ancak *gecikme*, eldeler en üst satırdan en alt satıra doğru yayıldığından $O(n)$ mertebesinde ve donanım maliyeti $O(n^2)$ 'dir. 32 bitlik bir dizi çarpıcının gecikme süresi kabaca 32 tam toplayıcının seri gecikmesine eşittir; bu da çağdaş işlemciler için kabul edilemez derecede yavaştır.

Wallace Ağacı Çarpıcı

Wallace ağacı (*Wallace tree*), 1964 yılında Chris Wallace tarafından önerilen ve dizi çarpıcının gecikme sorununu çözen bir yapıdır. Dizi çarpıcıda kısmi çarpımlar bir ızgarada satır satır toplanır ve eldeler en üst satırdan en alt satıra seri olarak yayılır. Wallace ağacının temel fikri, bu sıralı toplamayı ortadan kaldırıp kısmi çarpımları *ağaç yapısında koştur* olarak indirgemektir.

Yöntemin anahtar bileşeni 3:2 sıkıştırıcı (carry-save adder, CSA) adı verilen devredir. Bir 3:2 sıkıştırıcı, aslında her bit sütununa bağımsız olarak uygulanan bir tam toplayıcıdır: üç sayıyı girdi olarak alır ve iki sayı (toplam T ve elde C) üretir. Kritik nokta şudur: elde bitleri bir sonraki sütuna taşınır ancak sütunlar arası seri elde yayılımı *yoktur*. Her sütundaki tam toplayıcı bağımsız çalıştığından bir CSA'nın gecikmesi yalnızca tek bir tam toplayıcı gecikmesine eşittir; girdi sayısından bağımsızdır.

Wallace ağacı bu sıkıştırıcıları şu şekilde kullanır:

1. n adet kısmi çarpım 3'erli gruplara ayrılır.
2. Her grupta bir 3:2 CSA ile 3 sayı $\rightarrow$ 2 sayıya (toplam + elde) indirgenir.
3. Geriye kalan sayılar yeniden 3'erli gruplara ayrılarak işlem yinelenir.
4. Geriye yalnızca 2 sayı kalınca bunlar hızlı bir önceden elde üreten toplayıcıyla (CLA) toplanır ve sonuç elde edilir.

Her aşamada sayı adedi kabaca $\frac{2}{3}$ 'e düştüğünden toplam aşama sayısı $O(\log_{1.5} n) \approx O(\log n)$ 'dir; dizi çarpıcının $O(n)$ gecikmesiyle kıyaslanınca büyük bir kazanımdır. Donanım maliyeti dizi çarpıcıyla benzerdir ($O(n^2)$ tam toplayıcı); ancak bağlantı düzeni ızgara yerine ağaç biçiminde olduğundan daha karmaşıktır. 32 bitlik bir çarpma örneğinde: 32 kısmi çarpım $\rightarrow$ 22 $\rightarrow$ 15 $\rightarrow$ 10 $\rightarrow$ 7 $\rightarrow$ 5 $\rightarrow$ 4 $\rightarrow$ 3 $\rightarrow$ 2 sayı; yalnızca 8 sıkıştırma aşaması yeterlidir.

3:2 Sıkıştırıcı Nasıl Çalışır?

3:2 sıkıştırıcının (CSA) çalışma mantığını küçük bir örnekle görelim. Üç adet 4 bitlik sayıyı toplamak istediğimizi düşünelim: $A = 1101$ (13), $B = 0110$ (6) ve $C = 0011$ (3). Gerçek toplam $13 + 6 + 3 = 22$ olmalıdır.

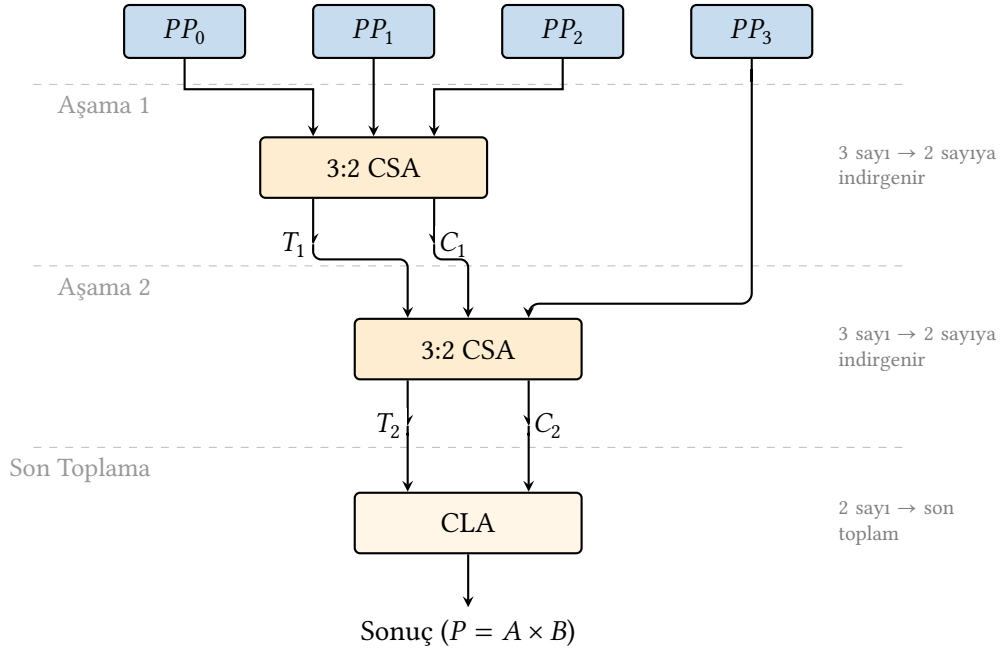
Normal bir toplayıcıda elde bitleri sütun sütun sağdan sola yayılır ve her sütun, soldaki sütunun eldesini bekler. CSA ise her bit sütununa *bağımsız* birer tam toplayıcı uygular; sütunlar birbirini beklemez:

	Bit 3	Bit 2	Bit 1	Bit 0	
A	1	1	0	1	(13)
B	0	1	1	0	(6)
C	0	0	1	1	(3)
Toplam (T)	1	0	0	0	= 8
Elde (C)	0 1	1	1	0	= 14 (1 konum sola kaydırılmış)

Her sütundaki tam toplayıcı, o sütundaki üç biti toplar ve bir toplam biti ile bir elde biti üretir. Elde biti bir sonraki (soldaki) sütuna kaydırılır; ancak *sütunlar arası seri elde yayılımı yoktur*; dört sütundaki dört tam toplayıcı eş zamanlı çalışır.

Sonuçta $T = 1000_2 = 8$ ve $C = 01110_2 = 14$ elde edilir. Dikkat edin: $T + C = 8 + 14 = 22$; doğru sonucu verir! Ancak T ve C henüz *toplanmamıştır*; yalnızca üç sayı iki sayıya *indirgenmiştir*. Gerçek toplama hâlâ yapılmamıştır çünkü gerçek toplama elde yayılımı gerektirir ve bu işlem zaman alır. CSA'nın tüm hız kazancı buradan gelir: elde yayılımından kaçınarak yalnızca sayı adedini azaltır.

Wallace ağacında bu indirgeme işlemi yinelenerek sayı adedi her aşamada $\frac{2}{3}$ 'e düşürülür. En sonda kalan iki sayı için kaçınılmaz olarak *bir kez* gerçek toplama (CLA ile) yapılır. Böylece n kez seri elde yayılımı yerine, $O(\log n)$ adet eldesiz sıkıştırma + tek bir hızlı toplama gerçekleştirilir.



Şekil 5.22: Dört kısmi çarpımın Wallace ağacıyla indirgenmesi: iki 3:2 CSA aşaması dört girdi sayısını ikiye düşürür; ardından önceden elde üreten bir toplayıcı son toplamı üretir.

Bilgi Notu 5.7: Dadda Ağacı

Luigi Dadda 1965'te Wallace ağacına benzer ancak daha az donanım kullanan bir yöntem önermiştir. Dadda ağacı her aşamada yalnızca gerekli minimum sıkıştırmayı yapar ve gereksiz tam toplayıcılardan kaçınır. Uygulamada Dadda ağacı Wallace ağacıyla aynı gecikmeye sahiptir ancak daha az donanım kullanır. Günümüzde her iki yapı da "Wallace-tarzı ağaç çarpıcı" olarak anılır.

Şekil 5.22, dört kısmi çarpımın Wallace ağacıyla iki sayıya nasıl indirildiğini göstermektedir. Her 3:2 sıkıştırıcı aşaması koşut çalışır; gecikme, sıkıştırıcı aşama sayısıyla orantılıdır.

Örnek 5.9

Soru: $1101_2 \times 1111_2$ (13×15) çarpımını Wallace ağacı yöntemiyle gerçekleştirin.

Çözüm: $A = 1101_2 = 13_{10}$ ve $B = 1111_2 = 15_{10}$ sayıları için $b_0=1$, $b_1=1$, $b_2=1$, $b_3=1$ olduğundan dört kısmi çarpımın tamamı sıfırdan farklıdır:

Kısmi Çarpım	İkili (8 bit)	Onlu
$PP_0 = A \cdot b_0$	0000 1101	13
$PP_1 = A \cdot b_1$ (1 sola)	0001 1010	26
$PP_2 = A \cdot b_2$ (2 sola)	0011 0100	52
$PP_3 = A \cdot b_3$ (3 sola)	0110 1000	104

Wallace ağacının iki sıkıştırma aşaması şöyle işler (CSA her bit sütununda bağımsız üç girdi toplar, elde bitleri bir konum sola kaydırılır):

1. *Aşama 1*: PP_0, PP_1, PP_2 bir 3:2 CSA'ya girer:

$$T_1 = 0010\ 0011_2 = 35, \quad C_1 = 0011\ 1000_2 = 56.$$

Doğrulama: $T_1 + C_1 = 35 + 56 = 91 = 13 + 26 + 52$. ✓

2. *Aşama 2*: T_1, C_1, PP_3 ikinci 3:2 CSA'ya girer:

$$T_2 = 0111\ 0011_2 = 115, \quad C_2 = 0101\ 0000_2 = 80.$$

Doğrulama: $T_2 + C_2 = 115 + 80 = 195 = 91 + 104$. ✓

3. *Son toplama (önceden elde üreten toplayıcı)*: $T_2 + C_2 = 115 + 80 = 195 = 1100\ 0011_2$.

Doğrulama: $13 \times 15 = 195$. ✓

Gerçek devrede işlemler bit sütunları üzerinden yürütülür: CSA her sütunu bağımsız toplar, eldeler kaydırılarak bir sonraki sütuna taşınır. Tüm sütunlar *koşut* olarak işlendiğinden herhangi bir seri elde yayılımı oluşmaz.

İki koşut çarpıcı yapısının karşılaştırması Tablo 5.15'de özetlenmiştir.

Tablo 5.15: Dizi çarpıcı ve Wallace ağacı karşılaştırması (n bitlik çarpma)

	Gecikme	Donanım	Düzenlilik
Dizi çarpıcı	$O(n)$	$O(n^2)$	Yüksek (ızgara)
Wallace ağacı	$O(\log n)$	$O(n^2)$	Düşük (karmaşık bağlantı)

Bilgi Notu 5.8: Çağdaş İşlemcilerde Çarpma Birimi

Çağdaş yüksek başarılı işlemcilerde çarpma birimi tipik olarak şu bileşenlerden oluşur:

1. **Radix-4 Booth kodlayıcı:** Çarptandan $n/2$ adet kısmi çarpım üretir.
2. **Wallace/Dadda ağacı:** Kısmi çarpımları $O(\log n)$ aşamada 2 sayıya indirger.
3. **Önceden elde üreten toplayıcı:** Son iki sayıyı toplar.

Bu mimari sayesinde 64 bitlik bir çarpma işlemi tipik olarak 3–5 saat çevriminde tamamlanır. Çarpma birimi genellikle boru hatlı (*pipelined*) olarak tasarlanır: her çevrimde yeni bir çarpma başlatılabilir, her ne kadar tek bir çarpmanın sonucu birkaç çevrim sonra hazır olsa da.

5.4.5 İşlenenlerden Birinin Sıfır Olması

Çarpma işleminde işlenenlerden birinin sıfır olduğu durumda sonucun da sıfır olacağı önceden bellidir. Kaydır-topla algoritmasında çarpan sıfırsa hiçbir kısmi çarpım üretilmez; çarpılan sıfırsa üretilen her kısmi çarpım zaten sıfırdır. Basit bir donanım bu durumu fark etmeden 32 yinelemenin tamamını çalıştırır; ancak bu gereksiz bir zaman kaybıdır.

Çağdaş işlemcilerde çarpma birimleri genellikle bir *erken sonlandırma* mekanizması içerir: çarpan yazmacının içeriği her sağa kaydırma sonrasında denetlenir; kalan bitler tümüyle sıfırsa başka kısmi çarpım oluşmayacağı garanti olduğundan döngü erkenden bitirilir. Bu denetim yalnızca bir VEYA kapısı ağacıyla (tüm bitlerin 0 olup olmadığını sınavan) gerçekleştirilebilir.

Erken sonlandırmanın yararı sıfır işlenenle sınırlı değildir. Çarpanın üst bitlerinin büyük bölümü sıfır olan küçük sayılarda da (örneğin $3 = 00000000 \dots 011_2$) döngü birkaç adımda tamamlanır. Araştırmalar, gerçek programlarda işlenenlerin önemli bir bölümünün az sayıda anlamlı bite sahip *dar değerler* (*narrow values*) olduğunu göstermiştir [11]; bu nedenle erken sonlandırma ortalama çarpma süresini belirgin biçimde kısaltır.

5.5 Bölme

Çarpma toplama üzerine kuruluysa, bölme de çıkarma üzerine kuruludur. Bölme, bir sayının içinde diğerinden kaç tane olduğunu bulmaktır. Tek bir ayrıklık vardır: bölen sıfır olamaz; tam sayı aritmetiğinde sıfıra bölmenin geçerli bir sonucu yoktur. Bölme işleminde dört temel bileşen vardır:

- **Bölünen** (*dividend*): bölünecek sayı,
- **Bölen** (*divisor*): bölen sayı,
- **Bölüm** (*quotient*): bölme sonucu (tam kısım),
- **Kalan** (*remainder*): artık değer.

Aralarındaki ilişki şöyle ifade edilir:

$$\text{Bölünen} = \text{Bölüm} \times \text{Bölen} + \text{Kalan}$$

Çarpmada n bitlik iki sayının çarpımının $2n$ bite kadar çıkabildiğini görmüştük. Bölme, çarpanın tersi olduğuna göre, işlenen boyutları da bu durumla uyumlu olmalıdır. n bitlik iki sayıyı birbirine böldüğümüzde bölüm en fazla n bit, kalan ise en fazla n bit uzunluğundadır. Ancak çarpma sonucunda elde edilen $2n$ bitlik bir değeri orijinal çarpanlardan birine bölmek istediğimizde, bölünenin çift genişlikte olması doğal bir gereksinimdir. Bu durumda bölüm de $2n$ bite kadar çıkabilir: örneğin $2n$ bitlik bir bölünen, 1 değerindeki bir bölene bölündüğünde bölüm bölünenin kendisine eşittir. Kalan ise her zaman bölenden küçük olacağından en fazla n bit kalır. RISC-V’te *div* ve *rem* buyrukları 32 bitlik iki yazmaç üzerinde çalışır ve 32 bitlik sonuç üretir; bu nedenle $2n$ bitlik bölünen doğrudan desteklenmez. Bununla birlikte, ileride göreceğimiz bölme donanımında kalan yazmacının neden $2n$ bit genişliğinde tutulduğu, bu boyut ilişkisiyle doğrudan bağlantılıdır.

Çarpmada olduğu gibi, ikili tabanda bölme işlemi de onluk tabandaki elde bölmeye benzer; ancak çok daha basittir, çünkü her adımda bölümün ilgili basamağı yalnızca 0 veya 1 olabilir: bölen çıkarılabiliyorsa 1, çıkarılamıyorsa 0 yazılır. Aşağıdaki örnekte $35 \div 5$ işlemi ikili tabanda gösterilmiştir:

$$\begin{array}{r|l}
 \text{Bölünen} & \text{Bölen} \\
 (35) \quad 100011 & 101 \quad (5) \\
 \hline
 & 000111 \quad (7) \\
 \hline
 \text{Kalan} \quad 000 & \text{Bölüm}
 \end{array}$$

Doğrulama: $7 \times 5 + 0 = 35 = 35 \checkmark$. İkili bölme de tıpkı çarpma gibi *kaydır ve çıkar* mantığına dayanır: her adımda bölünen çıkarılıp çıkarılamayacağı sınanır, bölüm biti belirlenir ve kalan bir basamak kaydırılır. Bu ilke, bölme donanımının temelini oluşturur.

Bölme Donanımına Neden İhtiyaç Var?

Çarpma bölümünde gördüğümüz gibi, çıkarma yapabilen her işlemci bölmeyi de yazılımla gerçekleştirebilir: bölünen sayıdan bölene arka arkaya çıkararak bölümü bulmak yeterlidir. Ancak bu yaklaşım büyük sayılarda son derece yavaştır:

Tablo 5.16: Aynı bölme işlemi ($35 \div 5$): yalnızca çıkarma ile (yazılım) ve div buyruğu ile (donanım).

Yalnızca çıkarma ile (M uzantısı yok)	div buyruğu ile (M uzantısı var)
<pre> 1 li t0, 0 # bolum = 0 2 li t1, 35 # bolunen 3 li t2, 5 # bolen 4 dongu: 5 blt t1, t2, son # bolunen < bolen? 6 sub t1, t1, t2 # bolunen -= bolen 7 addi t0, t0, 1 # bolum++ 8 j dongu # tekrarla 9 son: # t0=bolum, t1=kalan </pre>	<pre> 1 li t1, 35 # bolunen 2 li t2, 5 # bolen 3 div t0, t1, t2 # bolum = 35/5 4 rem t3, t1, t2 # kalan = 35 </pre>
Durağan buyruk sayısı: 7	Durağan buyruk sayısı: 4
Devingen buyruk sayısı: 32	Devingen buyruk sayısı: 4
Çevrim sayısı (BBÇ ≈ 1): ~32	Çevrim sayısı: 20–40

Soldaki programda döngü, bölüm değeri kadar yinelenir ($35 \div 5 = 7$ yinleme, toplam 32 buyruk). Bölünen büyüdükçe döngü sayısı doğrusal olarak artar; örneğin $10^9 \div 1$ için yaklaşık 4×10^9 buyruk gerekir; bu da saniyeler süren bir işlem demektir. Sağdaki programda ise div ve rem buyrukları aynı sonucu yalnızca birkaç düzine saat çevriminde üretir. Çarpmada olduğu gibi, bölme donanımı da *sorumluluğu yazılımdan donanıma aktararak* buyruk sayısını düşürür ve başarımı büyük ölçüde artırır.

Çarpmada olduğu gibi, bölme algoritmaları da verimsizden verimliye doğru geliştirilmiştir.

5.5.1 Geri Yüklemeli Bölme

Temel bölme algoritması *geri yüklemeli bölme* (restoring division) olarak adlandırılır: bölene, kalanın üst yarısından çıkarılır; sonuç eksiye çıkarma geri alınarak (kalan “geri yüklenerek”) önceki değerine döndürülür. Çarpmada olduğu gibi, donanımı adım adım iyileştiren üç yol geliştireceğiz.

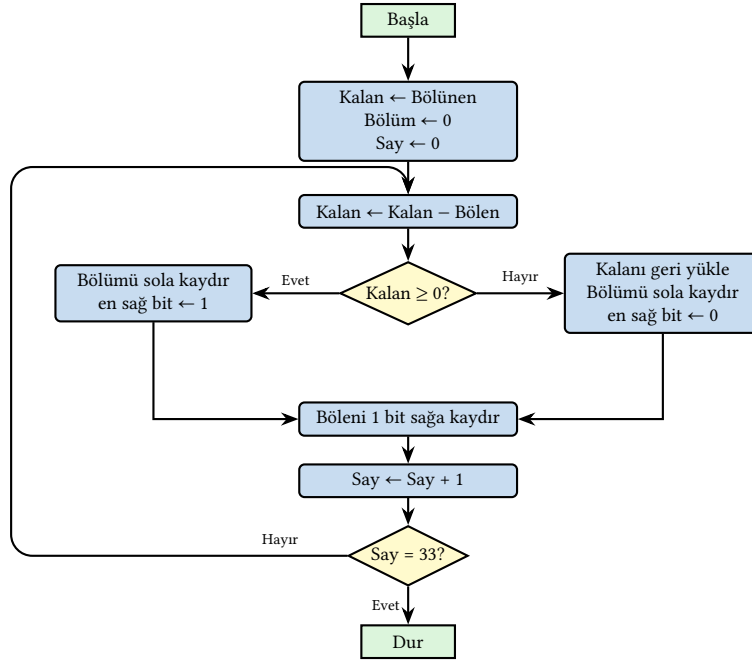
Donanım gerçekleştirmesinde dikkat çekici bir durum gözlemlenir: bölünen, işlem boyunca yazmaçlarda yavaş yavaş *tüketilir* ve yerini iki yeni değere (bölüm ve kalana) bırakır.

Her adımda bölen, kalan ara değerinden çıkarılmaya çalışılır; başarılı her çıkarma bölümün bir bitini belirlerken bölünenin o kısmını da dönüştürür. İşlem sonunda başlangıçtaki bölünen ortadan kalkmış, bilgi bölüm ve kalan arasında paylaşılmış olur. Bu dönüşüm yazmaçlarda fiziksel olarak da gözlemlenecektir: bölünen bitleri adım adım erirken, bölüm bitleri sağdan dolmaya başlar.

1. Yol

Bölmenin birinci yolunda çarpmada olduğu gibi bölme de en temel haliyle, arka arkaya çıkarma ve kaydırmalarla yapılır. İlk olarak bölünecek sayının tümü kalan yazmacına atılır. Verilerin uzunluğunun n bit olduğu kabul edilirse kalan yazmacının büyüklüğü $2n$ 'dir. Bölen yazmacı da $2n$ büyüklüğünde alınır.

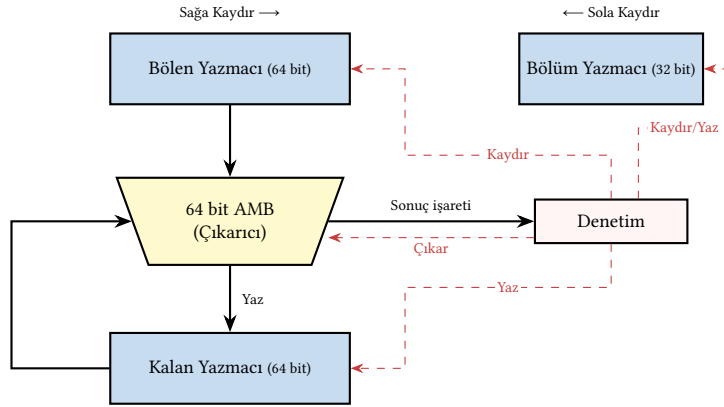
Bölen yazmacındaki değer kalandan çıkarılır. Sonuç eksi ise sayının içinde bölen yoktur ve işlem geriye alınır (kalan yazmacı eski haline getirilir). Bölüm yazmacı bir bit sola kaydırılır ve en sağ bitine 0 yazılır. Eğer kalandan bölen çıkarıldığında sonuç artı çıkarsa, fark kalan yazmacına yazılır ve bölüm 1 bit sola kaydırılarak en sağ bitine 1 yazılır. Döngü $n + 1$ kez devam eder.



Şekil 5.23: Geri Yüklemeli Bölme (1. Yol) akış şeması.

Algoritmanın akış şeması Şekil 5.23'de gösterilmiştir. Her yinelemede yapılan işlemler şunlardır: (1) kalan yazmacından bölen yazmacını çıkar, (2) sonuç eksiye kalanı geri yükle ve bölümün en sağ bitine 0 yaz; artıysa farkı kalana yaz ve bölümün en sağ bitine 1 yaz, (3) bölüni 1 bit sağa, bölümü 1 bit sola kaydır. Bu döngü $n+1$ kez tekrarlanır; 4 bitlik veriler için 5, RISC-V (RV32) gibi 32 bitlik bir mimaride ise 33 adım gerekir.

Buna karşılık gelen donanım tasarımı Şekil 5.24'de gösterilmiştir. Donanım, 64 bitlik bir bölen yazmacı, 64 bitlik bir kalan yazmacı, 32 bitlik bir bölüm yazmacı ve 64 bitlik bir AMB'den (çıkarcı) oluşur. Çarpmadaki 1. yola benzer biçimde, bölen yazmacının sol yarısına yerleştirilen değer her adımda sağa kaydırılarak kalanla hizalanır. Denetim birimi içindeki bir sayaç yineleme sayısını takip eder ve $n+1$ adım tamamlandığında işlemi sonlandırır. 64 bitlik AMB



Şekil 5.24: Geri Yüklemeli Bölme (1. Yol) donanımı

gerekliliği bu yolun temel dezavantajıdır: çarpmada olduğu gibi, 64 bitlik çıkarma 32 bitlik çıkarmaya göre daha yavaş, daha fazla alan kaplayan ve daha çok güç tüketen bir işlemdir.

Örnek 5.10

Soru: $5 \div 3 = ?$ ($0101 \div 0011 = ?$)

Çözüm: 1. yol algoritmasıyla (4 bit işlenenler, 8 bit bölen ve kalan yazmaçları). Başlangıçta bölünen 0101 kalan yazmacının alt yarısına, bölen 0011 ise bölen yazmacının üst yarısına yerleştirilir. Tabloda kırmızı bitler bölünen her adımda sağa kayan anlamlı konumunu, mavi bitler kalanı, yeşil bitler ise sağdan dolarak oluşan bölümü gösterir; gri bitler henüz anlam taşımayan sıfırlardır. Adım adım çözüm Tablo 5.17’de gösterilmiştir.

Tablo 5.17: Örnek 5.10 – 1. yol geri yüklemeli bölme algoritması ($0101 \div 0011$).

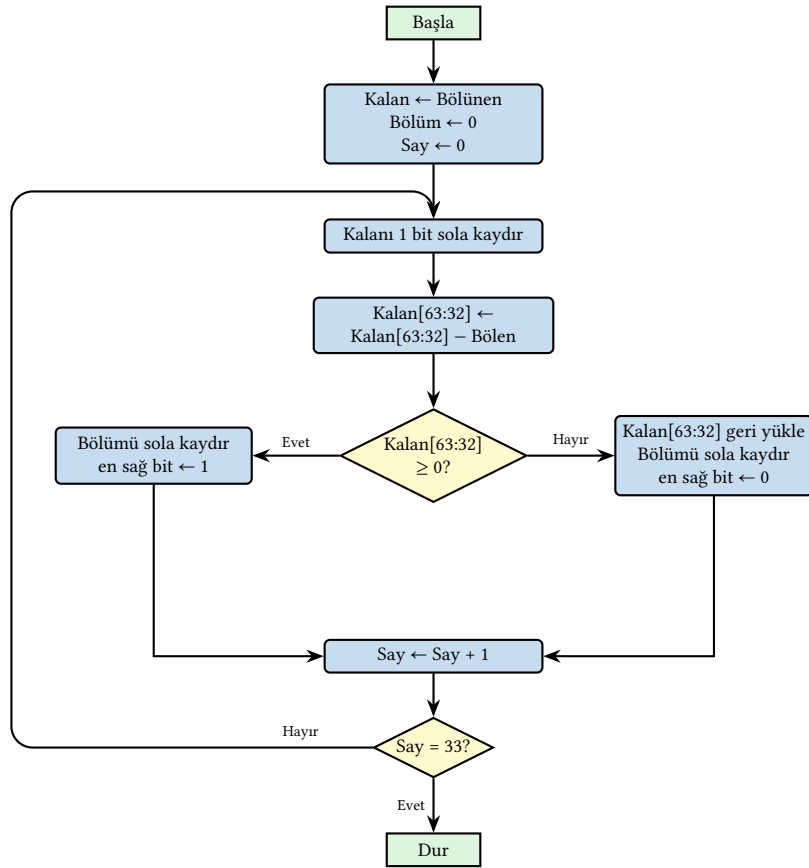
Adım	İşlem	Bölen (8 bit)	Kalan (8 bit)	Bölüm
Başlangıç	—	0011 0000	0000 0101	0000
1	Kalan – Bölen → eksi Geri yükle; Bölüm sola kaydır, bit ← 0	0011 0000 00011000	0000 0101 0000 0101	— 0000
2	Kalan – Bölen → eksi Geri yükle; Bölüm sola kaydır, bit ← 0	00011000 00001100	0000 0101 0000 0101	— 0000
3	Kalan – Bölen → eksi Geri yükle; Bölüm sola kaydır, bit ← 0	00001100 00000110	0000 0101 0000 0101	— 0000
4	Kalan – Bölen → eksi Geri yükle; Bölüm sola kaydır, bit ← 0	00000110 0000 0011	0000 0101 0000 0101	— 0000
5	Kalan – Bölen = 0000 0010 ≥ 0 Bölüm sola kaydır, bit ← 1	0000 0011 0000 0011	0000 0010 0000 0010	— 0001

Bölüm = $(0001)_2 = 1$, Kalan = $(00000010)_2 = 2$

2. Yol

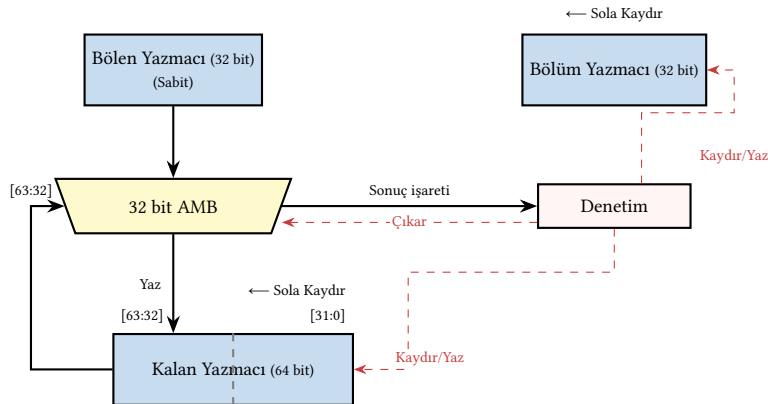
1. yolda $2n$ bitlik yazmaçlar sorun yaratır. 2. yolda bölen yazmacı n bit alınmış ve bölünen sağa kaydırılması yerine kalanın sola kaydırılarak işlemin daha verimli yapılması sağlanmıştır; bu

yolun akış şeması Şekil 5.25’de, donanım yapısı Şekil 5.26’de gösterilmiştir.



Şekil 5.25: Geri Yüklemeli Bölme (2. Yol) akış şeması.

Algoritmanın akış şeması Şekil 5.25’de gösterilmiştir. Her yinelemede yapılan işlemler şunlardır: (1) kalanı 1 bit sola kaydır, (2) kalanın üst yarısından bölüneni çıkar, (3) sonuç eksiye kalanı geri yükle ve bölümün en sağ bitine 0 yaz; artıysa farkı kalanın üst yarısına yaz ve bölümün en sağ bitine 1 yaz. 1. yoldan farklı olarak bölün sabit kalır ve kalan sola kaydırılarak hizalama sağlanır. Denetim birimi içindeki sayaç yineleme sayısını takip eder: n bitlik veriler için döngü n kez tekrarlanır; RISC-V (RV32) gibi 32 bitlik bir mimaride 32 adım gerekir.



Şekil 5.26: Geri Yüklemeli Bölme (2. Yol) donanımı

Buna karşılık gelen donanım tasarımı Şekil 5.26’de gösterilmiştir. Donanım, 32 bitlik sabit

bir bölen yazmacı, 64 bitlik sola kaydırılan bir kalan yazmacı, 32 bitlik bir bölüm yazmacı ve 32 bitlik bir AMB'den (çıkarıcı) oluşur. AMB artık yalnızca kalanın üst yarısı (32 bit) ile bölen (32 bit) arasında çıkarma yaptığından, 1. yolun 64 bitlik AMB'sine göre yarı boyutturulmuş; bu da her adımı daha hızlı ve donanımı daha küçük kılar. Denetim birimi her iki yazmacı da "Kaydır/Yaz" sinyali gönderir: kalan yazmacı her adımın başında 1 bit sola kaydırılır; bölüm yazmacı ise çıkarma sonucuna göre 1 bit sola kaydırılıp en sağ bitine 0 veya 1 yazılır.

Örnek 5.11

Soru: $7 \div 2 = ?$ ($0111 \div 0010 = ?$)

Çözüm: 2. yol algoritmasıyla (4 bit işlenenler, 8 bit kalan yazmacı, bölen 4 bit sabittir). Başlangıçta bölünen **0111** kalan yazmacının alt yarısına yerleştirilir. Her sola kaydırmada bölünenin bir biti üst yarıya taşınarak kalan hesabına katılır (**mavi**); alt yarıdaki **turuncu** bitler adım adım tükenir, sağdan giren **gri** sıfırlar ise boşalan konumları doldurur. Adım adım çözüm Tablo 5.18'de izlenebilir.

Tablo 5.18: Örnek 5.11 – 2. yol geri yüklemeli bölme algoritması ($0111 \div 0010$).

Adım	İşlem	Bölen (4 bit)	Kalan (8 bit)	Bölüm (4 bit)
Başlangıç	—	0010	0000 0111	0000
1	Kalan'ı 1 bit sola kaydır Kalan[üst] – Bölen = $0000 - 0010 \rightarrow$ eksi Geri yükle; Bölüm sola kaydır, bit $\leftarrow 0$	0010 0010 0010	0000 1110 0000 1110 0000 1110	0000 — 0000
2	Kalan'ı 1 bit sola kaydır Kalan[üst] – Bölen = $0001 - 0010 \rightarrow$ eksi Geri yükle; Bölüm sola kaydır, bit $\leftarrow 0$	0010 0010 0010	0001 1100 0001 1100 0001 1100	0000 — 0000
3	Kalan'ı 1 bit sola kaydır Kalan[üst] – Bölen = $0011 - 0010 = 0001 \geq 0$ Bölüm sola kaydır, bit $\leftarrow 1$	0010 0010 0010	0011 1000 0001 1000 0001 1000	0000 — 0001
4	Kalan'ı 1 bit sola kaydır Kalan[üst] – Bölen = $0011 - 0010 = 0001 \geq 0$ Bölüm sola kaydır, bit $\leftarrow 1$	0010 0010 0010	0011 0000 0001 0000 0001 0000	0001 — 0011

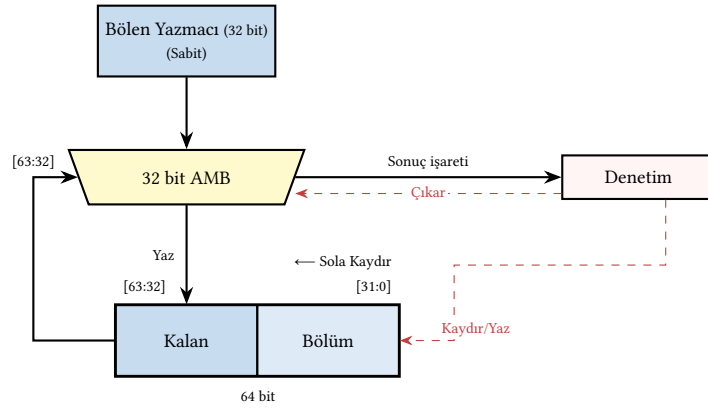
Bölüm = $(0011)_2 = 3$, Kalan = $(0001)_2 = 1$

Doğrulama: $7 = 3 \times 2 + 1$. ✓

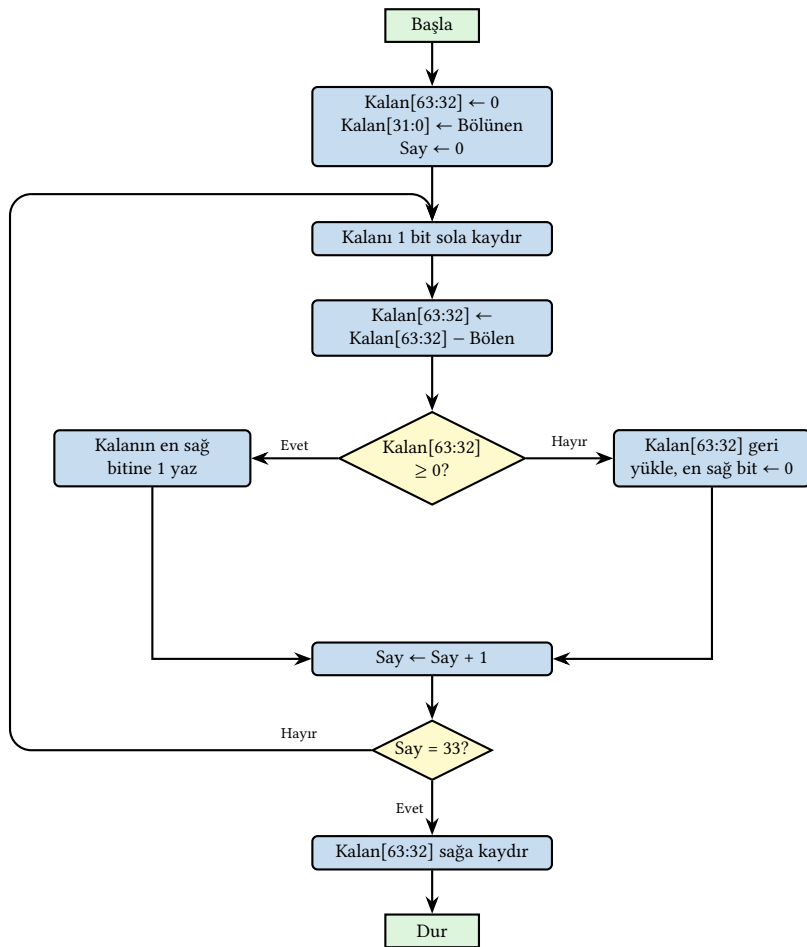
3. Yol

2. yolda kalan ve bölüm yazmaçları aynı yönde (sola) kaydırılır. İkisi ayrı yazmaçlarda tutularak 2 ayrı kaydırma yapılması yerine bölüm, kalan yazmacının alt yarısına gömülür ve tek bir kaydırma yeterli olur; tıpkı çarpmadaki 3. yolda çarpanın çarpım yazmacına gömülmesi gibi. Böylece ayrı bir bölüm yazmacına gerek kalmaz; donanım yalnızca 32 bitlik sabit bir bölen yazmacı, 64 bitlik birleşik bir kalan/bölüm yazmacı ve 32 bitlik bir AMB'den oluşur (Şekil 5.27). Algoritmanın akış şeması Şekil 5.28'de gösterilmiştir.

Her yinelemede birleşik yazmaç 1 bit sola kaydırılır, kalanın üst yarısından bölen çıkarılır ve sonuca göre en sağ bite 0 veya 1 yazılır. Bu en sağ bit artık bölüm yazmacının değil, birleşik yazmacın en sağ bitidir; böylece bölüm bitleri her adımda otomatik olarak alt yarıda birikir. Denetim birimi içindeki sayaç $n+1$ adımı takip eder; RISC-V (RV32) gibi 32 bitlik bir mimaride



Şekil 5.27: Geri Yükleli Bölme (3. Yol) donanımı



Şekil 5.28: Geri Yükleli Bölme (3. Yol) akış şeması.

33 adım gerekir. Döngü bittikten sonra, kalanın üst yarısındaki değer fazladan bir kaydırma ile düzeltilir.

Örnek 5.12

Soru: $6 \div 2 = ?$ ($0110 \div 0010 = ?$)

Çözüm: 3. yol algoritmasıyla (4 bit işlenenler, 8 bitlik birleşik kalan/bölüm yazmacı, bölen 4 bit sabittir). Başlangıçta bölünen birleşik yazmaca yerleştirilir: bölünen n bit ise alt yarıya [31:0] yazılıp üst yarı [63:32] sıfırlanır; $2n$ bit ise tüm yazmaca yazılır. Bu örnekte bölünen **0110** yalnızca 4 bit olduğundan alt yarıya yerleştirilmiştir. Her sola kaydırmada bölünenin bir biti üst yarıya taşınarak kalan hesabına katılır (**mavi**); alt yarıda **turuncu** bölünen bitleri tükenirken, sağdan giren **yeşil** bölüm bitleri dolar. Son adımda üst yarı fazladan bir kaydırmayla düzeltilir. Adım adım çözüm Tablo 5.19’de gösterilmiştir.

Tablo 5.19: Örnek 5.12 – 3. yol geri yüklemeli bölme algoritması ($0110 \div 0010$).

Adım	İşlem	Bölen (4 bit)	Kalan[üst] (4 bit)	Kalan[alt]/Bölüm (4 bit)
Başlangıç	—	0010	0000	0110
1	Sola kaydır Kalan[üst] – Bölen $\rightarrow$ eksi; geri yükle, bit $\leftarrow 0$	0010 0010	0000 0000	1100 1100
2	Sola kaydır Kalan[üst] – Bölen $\rightarrow$ eksi; geri yükle, bit $\leftarrow 0$	0010 0010	0001 0001	1000 1000
3	Sola kaydır $0011 - 0010 = 0001 \geq 0$; bit $\leftarrow 1$	0010 0010	0011 0001	0000 0001
4	Sola kaydır $0010 - 0010 = 0000 \geq 0$; bit $\leftarrow 1$	0010 0010	0010 0000	0010 0011
5	Sola kaydır Kalan[üst] – Bölen $\rightarrow$ eksi; geri yükle, bit $\leftarrow 0$	0010 0010	0000 0000	0110 0110
Son	Kalan[üst] sağa kaydır (düzeltme)	0010	0000	0011

Bölüm = $(0011)_2 = 3$, Kalan = $(0000)_2 = 0$. Doğrulama: $3 \times 2 + 0 = 6$. ✓

3. yol, geri yüklemeli bölme için en yalın donanım tasarımı sunar: yalnızca 1 adet 32 bitlik bölen yazmacı, 1 adet 64 bitlik birleşik kalan/bölüm yazmacı ve 32 bitlik bir AMB yeterlidir. Bu yapı RISC-V’e (RV32I) doğrudan uygulanabilir.

Bilgi Notu 5.9: İşaretili Bölme İşlemi

Yukarıdaki algoritmalar işaretsiz sayılar için tasarlanmıştır. İşaretili sayılarla bölme yapmak için en yaygın yöntem şudur: bölünen ve bölenin işaretleri saklanır, her iki sayı da mutlak değerine dönüştürülür ve işaretsiz bölme yapılır. İşlem sonunda bölümün ve kalanın işaretleri ayrıca belirlenir:

- **Bölümün işareti:** Bölünen ile bölenin işaretleri farklıysa bölüm eksidir (tıpkı çarpmada olduğu gibi).
- **Kalanın işareti:** Kalan her zaman bölünenle aynı işareti taşır. Örneğin $(-7) \div 2 = -3$ kalan -1 ’dir; çünkü $-7 = (-3) \times 2 + (-1)$.

RISC-V bu kuralı doğrudan destekler: div/rem buyrukları işaretili, divu/remu buyrukları

ise işaretli bölme yapar. İşaretli sürümler yukarıdaki kalan işareti kuralına uyar.

5.5.2 Geri Yükleme Bölme

Geri yüklemeli bölme, çıkarma sonucu eksiye kalanı eski değerine döndürmek (geri yüklemek) için ek bir toplama yapılır. Bu geri yükleme adımı zaman kaybına neden olur. *Geri yüklemesiz bölme* (*non-restoring division*) bu sorunu ortadan kaldırır.

Temel gözlem şudur: Geri yüklemeli bölme, eksi bir kalanı geri yükledikten sonra bir sonraki adımda yine bölen çıkarılır. Dolayısıyla sırasıyla yapılan iki işlem “+Bölen” (geri yükleme) ve “-Bölen” (yeni çıkarma) birleştirilirse net etki “+Bölen - Bölen = 0” yerine doğrudan bir sonraki adıma geçilmesidir. Daha doğrusu, eksi kalan elde edildiğinde geri yüklemek yerine bir sonraki adımda *bölen eklenir* (çıkarmak yerine).

Algoritmanın kuralları:

- Kalanı sola kaydır.
- $Kalan \geq 0$ ise: bölenden çıkar, bölüm bitine 1 yaz.
- $Kalan < 0$ ise: bölen ekle, bölüm bitine 0 yaz.

Geri yüklemesiz bölme, her adımda tam olarak bir aritmetik işlem (toplama veya çıkarma) ve bir kaydırma yapar; koşullu geri yükleme yoktur. Bu nedenle geri yüklemeli bölmeden daha hızlıdır ve donanım denetimi daha basittir. İşlem sonunda kalanın eksi olması mümkündür; bu durumda son adımda bölen bir kez eklenerek kalan artı yapılır. Sonuç, geri yüklemeli bölmeyle matematiksel olarak aynıdır.

Donanım açısından geri yüklemesiz bölme, geri yüklemeli bölmenin 3. yol donanımını (Şekil 5.27) kullanır. Tek fark, AMB’nin yalnızca çıkarma değil toplama da yapabilmesidir; tıpkı Booth çarpma algoritmasında AMB’nin hem toplama hem çıkarma yapması gibi. Denetim birimi kalanın işaret bitine bakarak AMB’ye toplama veya çıkarma sinyali gönderir; geri yükleme adımı ortadan kalktığından her yineleme sabit sürede tamamlanır.

Örnek 5.13

Soru: $7 \div 3 = ?$ ($0111 \div 0011 = ?$) işlemini geri yüklemesiz bölme ile çözün (4 bit işlenenler, 8 bit kalan yazmacı).

Çözüm: 3. yol donanımıyla (8 bitlik birleşik kalan/bölüm yazmacı, bölen 4 bit sabittir). Başlangıçta bölünen 0111 birleşik yazmacın alt yarısına yerleştirilir, üst yarı sıfırlanır. Geri yüklemeli bölmeden farklı olarak, burada kalanın işaret bitine bakılır: kalan artıysa bölen çıkarılır, eksiye bölen eklenir. Tabloda kırmızı bölen, mavi kalan (üst yarı), turuncu tükenen bölünen ve yeşil oluşan bölüm bitleridir.

Adım	İşlem	Bölen	Kalan[üst]	Kalan[alt]/Bölüm
Başlangıç	—	0011	0000	0111
1	Sola kaydır ≥ 0 : Çıkar $0000 - 0011 = 1101 (< 0)$	0011 0011	0000 1101	1110 1110
2	Sola kaydır < 0 : Topla $1101 + 0011 = 1110 (< 0)$	0011 0011	1011 1110	1100 1000
3	Sola kaydır < 0 : Topla $1110 + 0011 = 0000 (\geq 0)$	0011 0011	1101 0000	0000 0001
4	Sola kaydır ≥ 0 : Çıkar $0001 - 0011 = 1110 (< 0)$	0011 0011	0001 1110	0010 0010
Son	< 0 : Düzelt $1110 + 0011 = 0001$	0011	0001	0010

Bölüm = $(0010)_2 = 2$, Kalan = $(0001)_2 = 1$. Doğrulama: $7 = 2 \times 3 + 1$. ✓

Son adımdaki düzeltmeye dikkat: geri yüklemesiz bölme işlem sonunda kalan eksi çıkabilir; bu durumda bölen bir kez eklenerek kalan artı yapılır. Geri yüklemeli bölme ise her adımda koşullu geri yükleme yapıldığından kalan hiçbir zaman eksi kalmaz.

5.5.3 SRT Bölme

Yukarıdaki bölme algoritmalarının tümü her adımda yalnızca tek bir bölüm biti üretir. n bitlik bir bölme işlemi için n adım gerekir ve bu durum bölmeyi çarpmadan çok daha yavaş kılar.

SRT bölme (Sweeney–Robertson–Tocher) algoritması bu sorunu çözmek için her adımda birden fazla bölüm biti üretir. Radix-4 SRT bölme her adımda 2 bölüm biti hesaplanır ve adım sayısı $n/2$ 'ye düşer.

SRT algoritmasını anlamak için önce önceki algoritmalarla ortak noktalarına bakalım. Tüm bölme algoritmalarında her adımda bir *kalan ara değeri* hesaplanır; bu ara değere *kısmi kalan* (*partial remainder*) denir. Önceki bölümlerde bu ara değere kısaca “kalan” demiştik; SRT bağlamında “kısmi kalan” terimi tercih edilir çünkü bu değer her adımda değişir ve son adımda gerçek kalana dönüşür. Tüm yöntemlerde temel yineleme aynıdır:

$$P_{i+1} = r \cdot P_i - q_i \cdot d$$

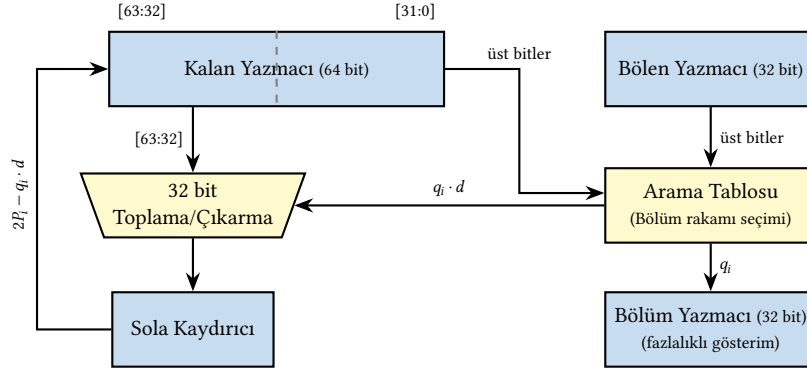
Burada P_i kısmi kalan, d bölen, q_i i adımın bölüm rakamı ve r tabandır (*radix*). Geri yüklemeli ve geri yüklemesiz bölme $r = 2$ ve $q_i \in \{0, 1\}$ 'dir; SRT bölmenin farkı *fazlalıklı rakam kümesi* kullanmasıdır.

SRT algoritmasının temel farkları:

- **Fazlalıklı rakam kümesi:** Bölüm rakamları yalnızca $\{0, 1\}$ değil, daha geniş bir kümeden seçilir. Radix-2 SRT'de $q_i \in \{-1, 0, +1\}$, radix-4 SRT'de $q_i \in \{-2, -1, 0, +1, +2\}$. Bu fazlalıklı (*redundant*) gösterim, bölüm rakamı tahmininde hata payına izin verir: seçim hatalı olsa bile bir sonraki adımda düzeltilebilir.
- **Arama tablosu:** Bölüm rakamı q_i , kısmi kalanın ve bölenin en anlamlı birkaç bitine bakılarak bir *arama tablosu* (*lookup table*) aracılığıyla belirlenir. Bu, koşullu çıkarma sonucunu beklemeye gerek kalmadan hızlı karar verilmesini sağlar.
- **Son dönüşüm:** Fazlalıklı rakamlarla ifade edilen bölüm, işlem sonunda standart ikili gösterime dönüştürülür.

Radix-4 SRT, çağdaş işlemcilerde tamsayı ve kayan nokta bölme birimlerinin standart yöntemidir. Daha yüksek tabanlı sürümler (radix-8, radix-16) daha büyük arama tablolarıyla adım sayısını daha da azaltabilir.

SRT bölme donanımının temel yapısı Şekil 5.29'de gösterilmiştir.



Şekil 5.29: SRT bölme donanımının blok şeması. Kalan yazmacı 64 bitlik birleşik yapıdadır; toplama/çıkarma yalnızca üst yarı [63:32] üzerinde 32 bit olarak yapılır.

Bu algoritmadaki arama tablosundaki küçük bir hata, bilgisayar tarihinin en ünlü donanım hatalarından birine yol açmıştır (bkz. Bölüm 5.16).

Radix-4 SRT'de her adımda 2 bölüm biti üretilir ve $q_i \in \{-2, -1, 0, +1, +2\}$ kümesinden seçilir. 32 bitlik bir bölme 16 adımda tamamlanır. Fazlalıklı küme sayesinde bölüm rakamı tahmininin kesin olması gerekmez; hatalı bir seçim bir sonraki adımda eksi bir rakamla düzeltilir. Bu esneklik, arama tablosunun kısmi kalanın yalnızca birkaç üst bitine bakarak hızlı karar vermesini mümkün kılar.

Bilgi Notu 5.10: Bölmeyi Çarpmaya Dönüştürmek

SRT bölme yinelemeli çıkarma üzerine kurulu olduğundan çarpmadan her zaman daha yavaştır. Bu nedenle bazı çağdaş tasarımlar bölmeyi çarpmaya dönüştüren yöntemler kullanır:

Newton-Raphson yöntemi: $1/d$ değerini yinelemeli olarak yakınsar. $x_{i+1} = x_i \times (2 - d \times x_i)$ formülüyle her yinelemede duyarlık *ikiye katlanır* (kuadratik yakınsama). Başlangıç tahmini bir arama tablosundan alınır ve 32 bit duyarlık için yalnızca 4–5 yineleme yeterlidir. Her yineleme iki çarpma ve bir çıkarma içerir.

Goldschmidt yöntemi: Benzer fikre dayanır ancak pay ve paydayı eş zamanlı olarak ölçekleyerek bölüneni 1'e yakınsatır. Goldschmidt'in üstünlüğü, her yinelemede çarpmaların *bağımsız* olması ve koşul çarpıcılarla hızlandırılabilmesidir.

Bu yöntemler özellikle kayan nokta bölme birimlerinde ve GPU'larda tercih edilir.

5.5.4 Bölme Yöntemlerinin Karşılaştırması

Geri yüklemeli bölmenin üç yolu, donanımı adım adım yalınlaştırarak aynı sonuca ulaşır: 1. yolda 64 bitlik AMB ve sağa kayan bölen kullanılırken, 2. yolda AMB 32 bite iner ve kalan sola kaydırılır; 3. yolda ise bölüm yazmacı ortadan kalkarak kalan yazmacının alt yarısına gömülür. Geri yüklemesiz bölme bu temelin üzerine koşullu geri yüklemeyi kaldırarak her adımı sabit süreli yapar. SRT bölme ise fazlalıklı rakam kümesi ve arama tablosuyla her adımda birden fazla bit üreterek adım sayısını yarıya indirir. Tüm bu yöntemlerin karşılaştırması Tablo 5.20'da

özetlenmiştir.

Tablo 5.20: Bölme yöntemlerinin gecikme, donanım karmaşıklığı ve temel özellik açısından karşılaştırması (n bit genişlik).

Yöntem	Adım	AMB	Yazmaç	Alan	Kayıp
Geri yükl. (1. yol)	$n+1$	64 bit	$2 \times 64 + 32$ bit	Büyük	64 bit AMB yavaş
Geri yükl. (2. yol)	n	32 bit	$64 + 2 \times 32$ bit	Orta	—
Geri yükl. (3. yol)	$n+1$	32 bit	$64 + 32$ bit	Küçük	—
Geri yüklemez	$n+1$	32 bit	$64 + 32$ bit	Küçük	Son düzeltme adımı
SRT (radix-2)	n	32 bit	$64 + 32$ bit + tablo	Orta	Arama tablosu alanı
SRT (radix-4)	$n/2$	32 bit	$64 + 32$ bit + tablo	Orta	Büyük arama tablosu
Newton-Raphson	4-5 yinleme	2 çarpıcı	Arama tablosu (ilk tahmin)	Büyük	Kuadratik yakınsama
Goldschmidt	4-5 yinleme	2 koşut çarpıcı	Arama tablosu (ilk tahmin)	Büyük	Koşut çarpma avantajı

Tüm yinlemeli yöntemlerde bölme çarpmadan yavaştır: n bitlik bir çarpma 32 saat vuruşunda tamamlanırken, bölme en az n vuruş gerektirir ve her vuruşta koşullu karar (çıkarma sonucunun işareti) boru hattını zorlaştırır. Bu nedenle derleyiciler mümkün olduğunca bölmeyi kaydırma veya çarpma ile değiştirmeye çalışır (örneğin $x/4$ yerine $x \gg 2$).

5.5.5 RISC-V Bölme Buyrukları

Çarpma bölümünde olduğu gibi, bölme de RISC-V'in temel buyruk kümesi RV32I'da yer almaz; M uzantısıyla eklenir. M uzantısı dört bölme buyruğu tanımlar:

Tablo 5.21: RISC-V Bölme Buyrukları (M Uzantısı)

Buyruk	Söz Dizimi	Tür	Açıklama
div	div rd, rs1, rs2	İşaretli	Bölümü rd'ye yazar. Sıfıra doğru yuvarlanır.
divu	divu rd, rs1, rs2	İşaretsiz	İşaretsiz bölümü rd'ye yazar.
rem	rem rd, rs1, rs2	İşaretli	Kalanı rd'ye yazar. Kalanın işareti bölünenle aynıdır.
remu	remu rd, rs1, rs2	İşaretsiz	İşaretsiz kalanı rd'ye yazar.

Her dört buyruk da aynı donanım bölme birimini kullanır; aralarındaki tek fark, işlenenlerin ve sonucun işaretli mi yoksa işaretsiz mi yorumlanacağıdır. İşaretli bölmede (div) bölüm *sıfıra doğru yuvarlanır*: örneğin $-7/2 = -3,5$ yuvarlanınca -3 olur, -4 'e değil. Kalan ise bölünen = bölüm $\times$ bölen + kalan özdeşliğinden bulunur ve her zaman bölünenle aynı işareti taşır.

MIPS'ten farklı olarak RISC-V'te bölüm ve kalan ayrı buyruqla elde edilir. MIPS'te div buyruğu her ikisini birden özel yazmaçlara (hi/lo) yazardı; RISC-V'te ise programcı hem bölümü hem kalanı istiyorsa div ve rem buyrukları art arda çalıştırılmalıdır. Ancak çağdaş RISC-V işlemcileri bu örüntüyü tanır ve iki buyruğu tek bir bölme işlemiyle birleştirir (*division fusion*).

Dikkat edilmesi gereken önemli bir nokta: div/rem ile divu/remu buyrukları aynı bit örüntüsünü tamamen farklı yorumlar. Yazmaçtaki bitler değişmez; değişen yalnızca donanımın bu

bitleri okuma biçimidir. Bu nedenle programcının (veya derleyicinin) veri türüne uygun buyruğu seçmesi gerekir: C dilindeki `int` gibi işaretli değişkenler için `div/rem`, `unsigned int` gibi işaretli değişkenler için `divu/remu` kullanılmalıdır. Bu farkın ne denli büyük sonuçlar doğurabileceği aşağıdaki örnekte görülmektedir.

Örnek 5.14

Soru: $-7 \div 2$ işlemini RISC-V’te hem işaretli hem işaretli bölme buyrukları ile gerçekleştirin. Sonuçlar neden farklıdır?

Çözüm: Önce yazmaçları hazırlayalım:

```
1  li  t1, -7    # bölünen: -7 (ikiye tümleyen: 0xFFFFF9)
2  li  t2, 2     # bölen: 2
```

1) İşaretli bölme (`div` ve `rem`): Bu buyruklar yazmaç içeriğini ikiye tümleyen işaretli sayı olarak yorumlar.

```
1  div t3, t1, t2 # t3 = bölüm: -7 / 2 = -3
2  rem t4, t1, t2 # t4 = kalan: (-7) mod 2 = -1
```

Burada `div` buyruğu bölümü `t3` yazmacına, `rem` buyruğu ise kalanı `t4` yazmacına yazar. Bölüm sıfıra doğru yuvarlanır: $-7/2 = -3,5$ yuvarlanınca -3 olur. Kalan ise bölüm $\times$ bölen + kalan = bölünen denkleminde bulunur: $(-3) \times 2 + (-1) = -7$. ✓ Dikkat: işaretli bölmede kalan her zaman bölünenle aynı işareti taşır; burada bölünen -7 olduğundan kalan da eksidir (-1).

2) İşaretsiz bölme (`divu` ve `remu`): Şimdi aynı yazmaçlarla işaretli buyrukları deneyelim:

```
1  divu t5, t1, t2 # t5 = bölüm: 4.294.967.289 / 2 = 2.147.483.644
2  remu t6, t1, t2 # t6 = kalan: 4.294.967.289 mod 2 = 1
```

Bu kez donanım `t1`’deki aynı bit örüntüsünü (`0xFFFFF9`) işaretli bir sayı olarak yorumlar. İkiye tümleyen gösterimde -7 ’yi temsil eden bu bitler, işaretli okunduğunda $2^{32} - 7 = 4\,294\,967\,289$ gibi çok büyük bir değere karşılık gelir. Dolayısıyla `divu` aslında $4\,294\,967\,289 \div 2$ hesaplar: bölüm = $2\,147\,483\,644$ (`t5`’e yazılır), kalan = 1 (`t6`’ya yazılır). Doğrulama: $2\,147\,483\,644 \times 2 + 1 = 4\,294\,967\,289$. ✓

Sonuçların karşılaştırması:

Buyruk	Bölüm	Kalan	Açıklama
<code>div/rem</code>	-3	-1	<code>t1</code> ’deki bitleri -7 olarak yorumlar
<code>divu/remu</code>	$2\,147\,483\,644$	1	Aynı bitleri $4\,294\,967\,289$ olarak yorumlar

Yazmaçtaki bitler iki durumda da aynıdır (`0xFFFFF9`); değişen yalnızca donanımın bu bitleri yorumlama biçimidir. Bu nedenle programcının veri türüne uygun buyruğu seçmesi gerekir: C dilindeki `int` gibi işaretli değişkenler için `div/rem`, `unsigned int` gibi işaretli değişkenler için `divu/remu` kullanılmalıdır.

5.5.6 Sıfıra Bölme ve Taşma

Tam sayı aritmetiğinde sıfıra bölmenin anlamlı bir sonucu yoktur; bölüm veya kalan olarak yazılabilecek geçerli bir tam sayı üretilmez. (Kayan nokta aritmetiğinde ise durum farklıdır:

Bölüm 5.6’da gördüğümüz gibi IEEE 754 standardı sıfırdan farklı bir sayının sıfıra bölünmesini $\pm\infty$ olarak tanımlar.) Peki tam sayı bölme donanımı bölen sıfır olduğunda ne yapar?

Farklı mimariler bu sorunu farklı biçimlerde ele alır. Bazı işlemciler (örneğin MIPS) sıfıra bölme durumunda bir *kural dışı durum (exception)* üreterek programın çalışmasını durdurur ve işletim sistemine denetimi devreder. RISC-V ise farklı bir yaklaşım benimser: sıfıra bölme durumunda **kural dışı durum üretmez**; bunun yerine sonuç yazmaçlarına belirli değerler yazar ve programın çalışmasına devam eder. Bu tasarım kararının arkasında RISC-V’in yalnlık ilkesi yatar; kural dışı durum mekanizması donanıma karmaşıklık katar ve bölme biriminin boru hattıyla uyumunu zorlaştırır.

RISC-V’te sıfıra bölme ve taşma durumlarında döndürülen değerler Tablo 5.22’de özetlenmiştir.

Tablo 5.22: RISC-V’te sıfıra bölme ve taşma durumlarında döndürülen değerler (32 bit).

Durum	Buyruk	Bölüm	Kalan
Sıfıra bölme (işaretli)	div/rem	-1 (0xFFFFFFFF)	bölünen
Sıfıra bölme (işaretsiz)	divu/remu	$2^{32} - 1$ (0xFFFFFFFF)	bölünen
Taşma ($-2^{31} \div -1$)	div/rem	-2^{31}	0

Sıfıra bölmede bölüm olarak tüm bitleri 1 olan değer döndürülmesi, yazılımın bu durumu kolayca saptamasını sağlar: program bölme buyruğundan sonra sonucun 0xFFFFFFFF olup olmadığını denetleyerek gerekli önlemi alabilir. Kalan olarak bölünenin kendisinin döndürülmesi ise bölünen = bölüm $\times$ bölen + kalan özdeşliğiyle tutarlıdır (çünkü bölüm $\times$ 0 + bölünen = bölünen).

İşaretli bölmedeki taşma durumu da dikkat gerektirir: $-2^{31} \div (-1) = +2^{31}$ olması gerekirdi, ancak $+2^{31}$ değeri 32 bitlik ikiye tümleyen ile gösterilemez (en büyük pozitif değer $2^{31} - 1$ ’dir). RISC-V bu durumda bölümü bölünenin kendisi (-2^{31}) olarak döndürür; bu matematiksel olarak yanlış bir sonuçtur, ancak donanım karmaşıklığını artırmamak adına bilinçli bir tercih yapılmıştır. Programcının bu uç durumu yazılımda denetlemesi beklenir.

5.6 Kayan Nokta İşlemleri

Şimdiye kadar yalnızca tam sayılarla çalıştık. Peki 3,14, 0,001 ya da $6,022 \times 10^{23}$ gibi ondalıklı ve çok büyük/küçük sayıları nasıl göstereceğiz? 32 bitle işaretsiz olarak en fazla $2^{32} - 1$, işaretli olarak -2^{31} ile $2^{31} - 1$ arasındaki tam sayıları gösterebiliyoruz; ancak gerçek dünyanın sorunları tam sayılarla sınırlı değil. İşte bu ihtiyaç için *kayan nokta* gösterimi geliştirilmiştir. İkilik taban dönüşümleri ve sayı gösterimi hakkında ayrıntılı bilgi için Ek A’da bakınız.

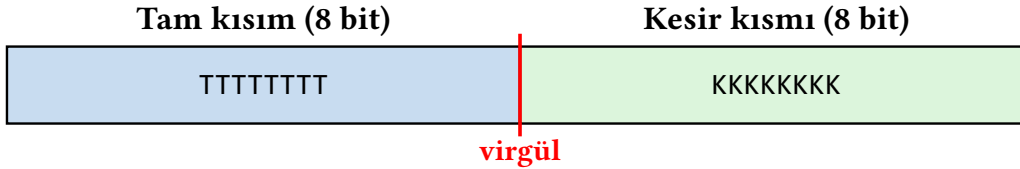
Burada önemli bir noktanın altını çizelim: bellekte ya da yazmaçta saklanan 32 bitlik bir değer tam sayı mı yoksa kayan nokta mı olduğunu **donanım kendiliğinden bilmez**. Aynı bit örüntüsü, kullanılan buyruğa bağlı olarak tamamen farklı yorumlanır. Örneğin add buyruğu yazmaçtaki bitleri ikiye tümleyen tam sayı olarak ele alırken, fadd.s buyruğu aynı bitleri IEEE 754 kayan nokta sayısı olarak yorumlar. Bu ayrımı belirleyen programcı veya derleyicidir; donanım yalnızca kendisine verilen buyruğa uygun işlemi gerçekleştirir. Aynı ilkeyi daha önce işaretli ve işaretsiz tam sayılarda da görmüştük (Bölüm 5.5.5’deki div/divu örneğini hatırlayın); kayan nokta ile tam sayı arasındaki fark da bundan farklı değildir.

5.6.1 Sabit Nokta ve Kayan Nokta

Kesirli sayıları ikilik tabanda göstermenin iki temel yolu vardır: **sabit nokta** ve **kayan nokta**.

Sabit Nokta Gösterimi

Sabit nokta gösteriminde virgölün yeri önceden sabittir: bit dizisinin belirli bir kısmı tam sayı bölümünü, geri kalanı kesir bölümünü temsil eder. Örneğin 16 bitlik bir sabit nokta biçiminde 8 bit tam kısma, 8 bit kesir kısmına ayrılabilir:



Bu düzende gösterilebilecek en büyük sayı $255,996 (\approx 2^8 - 2^{-8})$, en küçük pozitif sayı ise $0,0039 (= 2^{-8})$ olur. Sabit nokta, donanım açısından çok yalındır; toplama ve çıkarma tam sayı devreleriyle aynıdır. Ancak ciddi bir sınırlaması vardır: **değer aralığı çok dardır**.

Bu sınırlamayı somut bir örnekle görelim. Bir mühendislik programında aşağıdaki üç değerle aynı anda çalışılması gerektiğini düşünelim:

Büyüklik	Değer	8.8 sabit noktada	Sonuç
Köprü uzunluğu	1 250,75 m	Tam kısım $1250 > 255$	Taşma
Vida çapı	12,347 mm	$12 + 0,34375$; $12,347 \approx 12,344$	Duyarlık kaybı
Mikrop boyutu	0,00085 mm	En küçük kesir $0,0039 > 0,00085$	Gösterilemez

Sorun açıktır: virgölün yeri sabitlendikten sonra, tam kısmın kapasitesini aşan büyük sayılar *taşar* (*overflow*); kesir kısmının çözünürlüğünden küçük sayılar ise sıfıra yuvarlanarak kaybolur.

Üstelik burada gizli bir *ödünleşme* de vardır. Köprünün uzunluğu (1250,75) tam kısma sığmadığı için gösterilemez; ancak bu sayının kesir kısmı yalnızca 2 ondalık basamak içerir ve 8 bitlik kesir alanına rahatlıkla sığdı. Sabit nokta biçiminde tam kısma ayrılan 8 bit bu sayı için yetersizken, kesir kısmına ayrılan 8 bit boşa harcanmaktadır. Tersine, mikrobun boyutu (0,00085) için tam kısım hiç gerekmezken kesir alanı yetersiz kalmaktadır. Virgölü sağa kaydırsak büyük sayıları gösterebiliriz ama küçük sayıları kaybederiz; sola kaydırsak tam tersi olur. Virgül nereye konursa konsun, 16 bitin tamamını verimli kullanan *tek bir konum yoktur*; sorun virgölün sabit olmasının kendisidir.

Bit genişliğini artırmak (örneğin 16.16 biçimine geçmek) aralığı genişletir ancak sorunun özünü değiştirmez; çünkü virgül hâlâ sabittir ve yeterince büyük ya da yeterince küçük bir sayı yine gösterilemez.

Kayan Nokta: Virgölü Serbest Bırakmak

Kayan nokta gösterimi bu sorunu, virgölün yerini *değişken* yaparak çözer. Sayı, bir *anlamli kısım* ve bir *üs* olarak ikiye ayrılır; üs, virgölün nereye kayacağını belirler. Böylece aynı bit genişliğiyle hem çok büyük (10^{38}) hem de çok küçük (10^{-38}) sayılar ifade edilebilir, tabii ki sınırlı bir duyarlılıkla.

Bu fikir, onluk tabandaki bilimsel gösterimin ikilik tabana uyarlanmasıdır. Nasıl çalıştığını anlamak için önce bilimsel gösterimi ve olağanlaştırmayı ele alalım.

5.6.2 Bilimsel Gösterim ve Olağanlaştırma

Bilimsel gösterim, sayıda virgülün solunda tek basamak olmasıdır. Virgülden önceki basamak 0 değilse buna *olağanlaştırılmış* gösterim denir.

- 21,41239 – bilimsel gösterimde değildir.
- $0,2141239 \times 10^2$ – bilimsel gösterimdedir.
- $2,141239 \times 10^1$ – bilimsel gösterimdedir ve *olağandır*.

Bilimsel gösterim 2'lik düzene uygulandığında:

$$(01,10)_2 = 1,10 \times 2^0$$

$$(10011010,110)_2 = 1,0011010110 \times 2^7$$

İkilik düzende olağanlaştırılmış gösterim $1, \dots \times 2^e$ ile ifade edilir. Çünkü virgülün solundaki basamağın 0'dan farklı olması ve tek bir basamak olması zorunludur. 2'lik düzende 0'dan farklı tek basamak 1 olduğu için virgülden önceki basamağın 1 olması kaçınılmazdır.

Terim Kökenleri

olağanlaştırma (*normalization*): İngilizce *normalize* fiilinden gelen bu terim, Türkçe bilişim yazınında “normalleştirme” ve “normalizasyon” biçimlerinde de geçer. Bu kitapta Türkçe kökenli **olağanlaştırma** tercih edilmiştir: *olağan* sözcüğü “normal” anlamına gelir; olağanlaştırma da sayıyı *olağan biçime* (virgülün solunda tek anlamlı basamak kalacak şekle) getirmek demektir. Aynı kökten türeyen *olağanlaştırılmamış* (*denormalized*) terimi ise IEEE 754'te sıfıra çok yakın özel değerleri tanımlar.

Not: Türkçe bilişim terminolojisinde terim birliği henüz tam sağlanamamıştır; “normalleştirme” veri tabanı ve veri bilimi alanlarında daha yaygınken, “olağanlaştırma” bilgisayar mimarisi ve sayısal çözümleme bağlamında tercih edilen biçimdir.

5.6.3 IEEE 754 Kayan Nokta Standardı

Kayan nokta sayılarının bellekte nasıl temsil edileceğini belirleyen evrensel standart **IEEE 754**'tür (*Institute of Electrical and Electronics Engineers*, 1985). Bu standart öncesinde her işlemci üreticisi kendi kayan nokta biçimini kullanıyordu; bir makinede hesaplanan sonuç başka bir makinede farklı çıkabiliyordu. IEEE 754, işaret, üs ve kesir alanlarının bit genişliklerini, eklentili üs kodlamasını, özel değerleri (sıfır, sonsuz, NaN) ve yuvarlama kurallarını kesin biçimde tanımlayarak *taşınabilir* ve *öngörülebilir* kayan nokta aritmetiğini mümkün kıldı.

Tek Duyarlı Gösterim (32 bit)

Tek duyarlı (*single precision*) biçimde 32 bit üç alana bölünür:

İşaret (1 bit)	Üs (<i>exponent</i>) (8 bit)	Kesir (<i>fraction</i>) (23 bit)
i	$e_7 \dots e_0$	$f_{22} \dots f_0$
31	30	23 22
		0

- **İşaret (bit 31):** 0 ise sayı artı, 1 ise eksidir.
- **Üs (bit 30–23):** Gerçek üs, $e_{\text{gerçek}} = e - 127$ formülüyle hesaplanır. 8 bitlik alan 0–255 arası değer alır; ancak 0 ve 255 özel değerler için ayrıldığından kullanılabilir aralık 1–254’tür. Bu kodlamaya **eklentili gösterim** (*biased / excess-127*) denir: gerçek üsse sabit bir *eklenti* (*bias*) değeri (tek duyarlıda 127) eklenerek üs alanına yazılır. Negatif üslerin ayrı bir işaret bitine gerek kalmadan ifade edilmesini sağlar; ayrıca iki kayan nokta sayısının üs alanlarını işaretlessiz tam sayı gibi karşılaştırarak büyüklük sıralamaya olanak tanır.
- **Kesir (bit 22–0):** Olağanlaştırılmış sayılarda virgülden önceki bit her zaman 1 olduğundan bu bit bellekte *saklanmaz*; donanım tarafından otomatik olarak eklenir. Bu saklanmayan bite **gizli 1** (*hidden bit*) denir. Dolayısıyla 23 bit ile aslında 24 bitlik duyarlık elde edilir.

Burada iki terimi birbirinden ayırt etmek gerekir: bellekte saklanan 23 bitlik alana **kesir** (*fraction*) denir; bu, virgülden sonraki kısımdır. Gizli 1 bu alanın başına eklendiğinde elde edilen 1, kesir değerine ise **anlamli kısım** (*significand*) denir. Eski kaynaklarda anlamli kısım için *mantis* (*mantissa*) terimini görebilirsiniz; ancak bu terim logaritma bağlamından geldiği için IEEE 754 standardı *significand* terimini tercih eder. Özetle:

$$\underbrace{1}_{\text{gizli 1}} \cdot \underbrace{f_{22} f_{21} \dots f_0}_{\text{kesir (23 bit, bellekte saklanır)}} = \underbrace{1, f_{22} f_{21} \dots f_0}_{\text{anlamli kısım (24 bit etkin duyarlık)}}$$

Bilgi Notu 5.11: Gizli 1 Her Yerde Var mı?

Gizli 1, IEEE 754’ün akıllıca bir tasarım kararıdır; ancak tüm kayan nokta biçimleri bu yöntemi kullanmaz. IEEE 754 öncesinde ve sonrasında farklı yaklaşımlar benimsenmiştir:

Biçim	Taban	Gizli 1?	Açıklama
IEEE 754 (FP32/FP64)	2	Evet	Olağanlaştırılmış sayılarda virgülden önceki 1 saklanmaz; 1 bit bedavaya gelir.
Intel x87 genişletilmiş (80 bit)	2	Hayır	64 bitlik anlamli kısım tam olarak saklanır; baştaki 1 açıkça yazılır. Olağanlaştırılmamış sayıların ve ara hesap sonuçlarının özel işlem gerektirmemesi için bu tasarım tercih edilmiştir.
IBM System/360 (HFP)	16	Hayır	Üs 16’nın kuvvetlerini gösterir; anlamli kısım ikilik tabanda en az bir anlamli onaltılık basamakla olağanlaştırılır. 16 tabanlı olağanlaştırmada baştaki bit her zaman 1 olmadığından gizli 1 kullanılamaz; bu yüzden 3 bite kadar duyarlık kaybı olabilir.
DEC VAX (F/D/G/H)	2	Evet	IEEE 754’e benzer biçimde gizli 1 kullanılır; ancak olağanlaştırılmamış sayıları ve NaN’ı desteklemez.

Intel x87'nin 80 bitlik genişletilmiş biçimi özellikle ilginçtir: bu biçim IEEE 754'ün bir parçası olmasına rağmen gizli 1 kullanmaz. Bunun nedeni, x87'nin kayan nokta hesaplamalarında *ara sonuçları* tutmak için tasarlanmış olmasıdır; olağanlaştırılmamış ara değerlerde gizli 1'in varsayılması hatalara yol açabilirdi. Açık bit sayesinde donanımın her durumda doğru davranması garanti edilir; bedeli ise anlamlı kısım için 1 bit fazla alan harcanmasıdır.

IBM System/360'ın onaltılık tabanlı kayan nokta biçimi ise tarihsel bir ders niteliğindedir: 16 tabanlı olağanlaştırma, ikilik tabana göre daha “gevşek” olduğundan anlamlı kısmın ilk 1–3 biti sıfır olabilir. Bu durum etkili duyarlılığı düşürür ve sayısal kararlılık sorunlarına yol açar. IEEE 754'ün ikilik tabana ve gizli 1'e geçişinin temel nedenlerinden biri de bu duyarlık kaybını ortadan kaldırmaktır.

Bilgi Notu 5.12: IEEE 754 Üsleri Neden Eklentili Saklar?

Tam sayı aritmetiğinde ikiye tümleyen çok iyi çalışır; o zaman üs alanı için de aynı yöntemi kullanmak doğal görünür. Ancak kayan nokta sayılarında bir gereksinim daha vardır: *büyüklik sıralaması*. Eğer üs alanı ikiye tümleyenle kodlansaydı, eksi bir üs (örneğin -5 , ikiye tümleyende 11111011) artı bir üsten (örneğin $+3$, ikiye tümleyende 0000011) *büyük* görünürdü; çünkü en anlamlı bit 1'dir. Bu durumda iki kayan nokta sayısını karşılaştırmak için üs alanının ikiye tümleyen mi işaretli mi olduğunu ayırt eden özel bir devre gerekirdi.

Eklentili gösterimde ise tüm üs değerleri 0'dan 255'e kadar artı sayılardır. En küçük gerçek üs (-126) en küçük kodlanmış değere (1), en büyük gerçek üs ($+127$) en büyük kodlanmış değere (254) karşılık gelir; sıralama doğal olarak korunur. Daha da önemlisi, işaret bitleri aynı olan iki kayan nokta sayısı, 32 bitlik gösterimleri *tek bir işaretli tam sayı gibi* karşılaştırılarak sıralanabilir. Bu özellik donanımda karşılaştırma devresini büyük ölçüde yalınlaştırır.

Terim Kökenleri

eklentili gösterim (*biased notation, excess-N notation*) : İngilizce *biased* sözcüğü “önyargılı” anlamına gelir; ancak “önyargılı üs” sanki üste bir hata eklenmiş izlenimi verir. Türkçe kaynaklarda kullanılan “sapmalı” da istenmeyen bir kayma çağrışımı yapar. Oysa yapılan işlem yalındır: gerçek üsse sabit bir değer *eklenir*. Bu nedenle bu kitapta **eklentili gösterim** terimi tercih edilmiştir.

Gerçek değer şu şekilde hesaplanır:

$$(-1)^i \times 1, f_{22} f_{21} \dots f_0 \times 2^{e-127}$$

C dilinde float veri tipiyle karşılık gelir; yaklaşık 7 onluk basamak duyarlık ve $\approx 10^{\pm 38}$ aralık sunar. Örneğin aşağıdaki C kodunda tanımlanan her değişken bellekte 32 bitlik IEEE 754 tek duyarlı biçimde saklanır:

```
1 float sicaklik = 36.6; /* 32 bit IEEE 754 */
2 float pi       = 3.14159; /* 32 bit IEEE 754 */
3 float kutle    = 0.00085; /* 32 bit IEEE 754 */
```

Derleyici bu tanımları gördüğünde her değeri IEEE 754 biçimine dönüştürür ve bellekte 32 bitlik bir alan ayırır. Programcı ondalıklı bir sayı yazmış olsa da bellekteki gösterim işaret, üs ve kesir bitlerinden oluşur; onluk tabandaki basamaklarla doğrudan bir ilişkisi yoktur.

Çift Duyarlı Gösterim (64 bit)

Çift duyarlı (*double precision*) biçimde 64 bit yine üç alana bölünür:



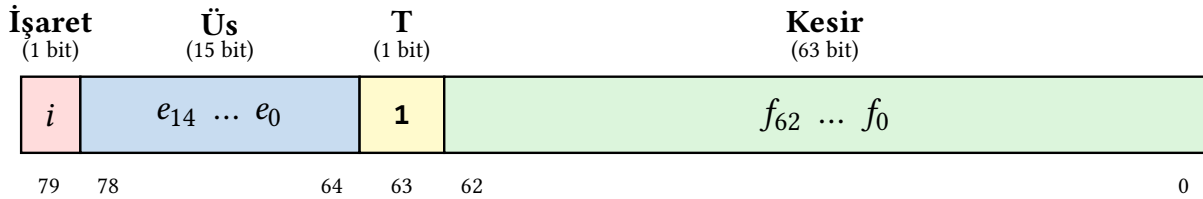
Üs alanı 11 bit olup eklenti değeri **1023**'tür; gerçek üs $e - 1023$ ile bulunur. Kesir alanı 52 bit olup gizli 1 ile birlikte 53 bitlik duyarlık sağlar ($\approx 15-16$ onluk basamak, $\approx 10^{\pm 308}$ aralık). C dilinde *double* veri tipiyle ifade edilir:

```
1 double uzaklik = 149597870.7; /* 64 bit IEEE 754 */
2 double elektron = 9.10938e-31; /* 64 bit IEEE 754 */
```

Daha yüksek duyarlık veya daha geniş değer aralığı gereken hesaplamalarda *float* yerine *double* tercih edilir; ancak bellek kullanımı iki katına çıkar.

Genişletilmiş Duyarlı Gösterim: Intel 80 Bit Biçimi

IEEE 754 standardı tek ve çift duyarlının yanı sıra isteğe bağlı bir *genişletilmiş duyarlı* (*extended precision*) biçim kategorisi de tanımlar. Standart bu biçimin tam düzenini belirlemez; yalnızca asgari gereksinimleri koyar: en az 64 bit anlamlı kısım ve en az 15 bit üs. Intel'in x87 kayan nokta birimi bu gereksinimleri karşılayan 80 bitlik bir biçim tanımlamıştır:



Bu biçimin tek ve çift duyarlıdan en önemli farkı, çizimde sarıyla gösterilen **T** (*integer bit*, tam kısım biti) alanıdır. Tek ve çift duyarlıda virgülden önceki 1 gizli 1'tir; saklanmaz, donanım tarafından varsayılır. 80 bitlik genişletilmiş biçimde ise bu bit **açıkça saklanır**: olağanlaştırılmış sayılarda 1, olağanlaştırılmamış sayılarda 0 değerini alır. Bu tasarım kararının nedeni, genişletilmiş biçimin çok adımlı hesaplamalarda *ara sonuçları* tutmak için tasarlanmış olmasıdır; ara değerler olağanlaştırılmamış olabilir ve gizli 1'in varsayılması hatalara yol açabilirdi. Açık bit sayesinde donanımın her durumda doğru davranması garanti edilir; bedeli yalnızca 1 bit fazla alan kullanılmasıdır.

Üs alanı 15 bit olup eklenti değeri **16 383**'tür. Bu geniş üs alanı $\approx 10^{\pm 4932}$ aralık sağlar; 64 bitlik anlamlı kısım ise $\approx 18-19$ onluk basamak duyarlık verir.

C dilinde karşılığı: 80 bitlik genişletilmiş biçim, C dilinde `long double` veri tipiyle kullanılır; ancak bu davranış platforma bağlıdır:

```
1 long double pi = 3.14159265358979323846L; /* x86'da 80 bit */
```

Platform	long double	Açıklama
x86 Linux (GCC)	80 bit (x87)	Bellekte 12 veya 16 bayt hizalanır
x86 Windows (MSVC)	64 bit (FP64)	long double = double ile aynı
ARM / Apple Silicon	128 bit (FP128)	IEEE 754 dörtlü duyarlı (yazılım emülasyonu)
RISC-V	128 bit (FP128)	Q uzantısı ile donanım desteği tanımlı (henüz yaygın değil)

Görüldüğü gibi `long double` yazıp “yüksek duyarlık” elde ettiğini düşünen bir programcı, kodunu farklı bir platformda derlediğinde beklenmedik sonuçlarla karşılaşabilir. Taşınabilir kod yazarken bu farkın bilincinde olmak gerekir.

Bilgi Notu 5.13: RISC-V’te Dörtlü Duyarlık: Q Uzantısı

RISC-V mimarisi 128 bitlik dörtlü duyarlı (*quad precision*) kayan nokta desteğini isteğe bağlı **Q uzantısı** ile tanımlar. Q uzantısı IEEE 754 dörtlü duyarlı biçimi kullanır: 1 işaret + 15 üs + 112 kesir = 128 bit. Bu biçim ≈ 34 onluk basamak duyarlık ve $\approx 10^{\pm 4932}$ aralık sağlar. Intel’in 80 bitlik genişletilmiş biçiminden farklı olarak gizli 1 kullanır ve tam IEEE 754 uyumludur.

Ancak Q uzantısı henüz yaygın donanım desteğine sahip değildir; günümüzde dörtlü duyarlık gerektiğinde çoğu RISC-V derleyicisi bunu yazılımla gerçekleştirir (*soft float*). Buna karşın F (tek duyarlı) ve D (çift duyarlı) uzantıları neredeyse tüm uygulama sınıfı RISC-V işlemcilerinde donanımla desteklenir.

IEEE 754 yalnızca tek, çift ve genişletilmiş duyarlı biçimleri tanımlamaz; günümüzde yaygın zeka donanımları çok daha dar biçimler de kullanır. Bu güncel biçimler Bölüm 5.8’de ele alınmaktadır.

Kayan Nokta Formülü

Kayan nokta ile gösterilen bir sayı aşağıdaki gibi ifade edilir:

$$(-1)^i \times (1 + \text{kesir}) \times 2^{(e - \text{referans})} \quad (5.20)$$

Burada:

- $(-1)^i$: Sayının işareti ($i = 0$ ise artı, $i = 1$ ise eksi)
- Kesir: Virgülden sonraki kısım (1 ile toplanarak gerçek anlamlı kısım bulunur)
- e : Üst alanındaki değer
- Referans: Tek duyarlıda 127, çift duyarlıda 1023

IEEE 754 standardında üs gösterimi için 2'ye tümleyen yerine eklentili (*biased / excess-N*) gösterim kullanılır. Tek duyarlıda eklenti değeri 127'dir. 8 bitlik üs alanı 0'dan 255'e kadar 256 farklı değer alabilir; ancak 0 ve 255 özel durumlar (sıfır, olağanlaştırılmamış sayılar, sonsuz, NaN) için ayrıldığından kullanılabilir üs değerleri 1'den 254'e kadardır. Gerçek üs = $e - 127$ olduğundan, gerçek üs aralığı -126 'dan $+127$ 'ye kadardır.

Örnek 5.15

Soru: 5,75 sayısını IEEE 754 standardında tek ve çift duyarlılıkta gösterin.

Çözüm: $5 = (101)_2$

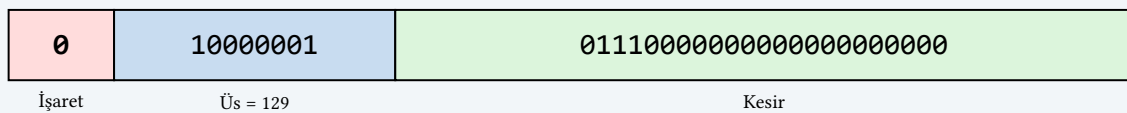
Virgülden sonraki kısım:

$$0,75 \times 2 = 1,5 \rightarrow 1$$

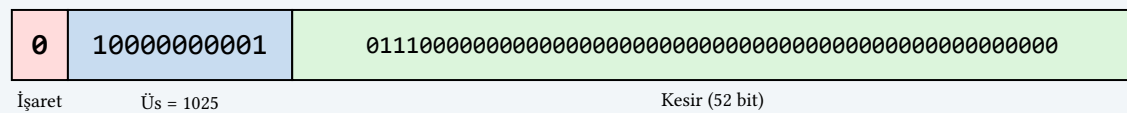
$$0,5 \times 2 = 1,0 \rightarrow 1$$

$$5,75 = (101,11)_2 = 1,0111 \times 2^2$$

Tek duyarlılık: $\ddot{U}s = 2 + 127 = 129 = (10000001)_2$



Çift duyarlılık: $\ddot{U}_s = 2 + 1023 = 1025 = (10000000001)_2$



Örnek 5.16

Soru: 2,1 sayısını IEEE 754 standardında tek duyarlılıkta gösterin.

Çözüm: $2 = (10)_2$

Virgülden sonraki kısım (devreden bir sayı dizisi):

$$0,1 \times 2 = 0,2 \rightarrow 0$$

$$0,2 \times 2 = 0,4 \rightarrow 0$$

$$0,4 \times 2 = 0,8 \rightarrow 0$$

$$0,8 \times 2 = 1,6 \rightarrow 1$$

$$0,6 \times 2 = 1,2 \rightarrow 1$$

$$0,2 \times 2 = 0,4 \rightarrow 0 \quad (\text{tekrar başlar: } 0011 \dots)$$

$$2,1 \approx (10,00011001100110011001100 \dots)_2 = 1,000011001100110011001100 \times 2^1$$

$$\ddot{U}_S: 1 + 127 = 128 = (10000000)_2$$

0	10000000	00001100110011001100110
İşaret	Üs = 128	Kesir

Bu sayıda devreden (1100 tekrar eden) bir sayı dizisi bulunur; 2,1 sayısı ikilik tabanda *tam olarak* gösterilemez.

Örnek 5.17

Soru: Tek duyarlılıkta 11000001101110100000000000000000 şeklinde gösterilen sayı 10'luk tabanda kaçtır?

Çözüm: Sayı parçalara ayrılır:

1	10000011	011101000000000000000000
İşaret	Üs = 131	Kesir

- İşaret: 1 (eksi)
- Üs: $(10000011)_2 = 131$, gerçek üs: $131 - 127 = 4$
- Kesir: 0,011101

$$(-1)^1 \times (1 + 0,011101) \times 2^4 = -1,011101 \times 2^4 = -10111,01$$

$$2^4 + 2^2 + 2^1 + 2^0 + 2^{-2} = 16 + 4 + 2 + 1 + 0,25 = 23,25$$

Sonuç: **-23,25**

5.6.4 Kayan Nokta Özel Değerleri

IEEE 754'te üs alanının tüm bitleri 0 (00000000) veya tüm bitleri 1 (11111111) olan durumlar olağan sayıları değil, *özel değerleri* temsil eder. Bu özel değerler Tablo 5.23'da özetlenmiştir.

Tablo 5.23: IEEE 754 özel değerlerinin kodlanması (tek duyarlı)

Değer	Üs	Kesir	Açıklama
+0 / -0	00000000	= 0	İşaret bitine göre +0 veya -0. Karşılaştırmada eşit kabul edilir.
Olağanlaştırılmamış	00000000	≠ 0	Örtük "1" varsayılmaz; $(-1)^i \times 0, \text{Kesir} \times 2^{-126}$. Sıfıra çok yakın küçük sayıları gösterir.
$+\infty$ / $-\infty$	11111111	= 0	İşaret bitine göre artı veya eksi sonsuz.
NaN	11111111	≠ 0	"Sayı değil" (Not a Number). Tanımsız işlem sonucu.

Sıfır

Kayan nokta formülünü hatırlayalım: $(-1)^i \times (1 + \text{kesir}) \times 2^{(e-127)}$. Formüldeki 1 + kesir ifadesine dikkat edelim: kesir alanının tüm bitleri sıfır olsa bile anlamlı kısım $1,000 \dots 0 = 1,0$ olur; sıfır

değil. Üs alanını ne yaparsak yapalım, $1,0 \times 2^e$ asla sıfır olamaz; çünkü 2^e her zaman artı bir sayıdır. Kısacası, olağanlaştırılmış gösterimde sıfırı temsil etmenin *hiçbir yolu yoktur*; gizli 1, sayının sıfır olmasını engeller.

Bu nedenle IEEE 754, sıfır için özel bir kodlama tanımlar: **üs alanı ve kesir alanının tamamı sıfırdır**. Donanım bu bit örüntüsünü gördüğünde olağan formülü uygulamaz; sonucu doğrudan sıfır olarak yorumlar.

İşaret bitine göre $+0$ ve -0 olmak üzere iki sıfır vardır. Matematiksel olarak $+0 = -0$ 'dır ve IEEE 754'e göre karşılaştırmada eşit kabul edilir. İki ayrı sıfırın bulunmasının nedeni, bazı sayısal çözümleme algoritmalarında sıfıra "hangi yönden" yaklaşıldığının önemli olmasıdır; örneğin $1/(+0) = +\infty$ iken $1/(-0) = -\infty$ olur.

Olağanlaştırılmamış Sayılar

Üs alanı 0 olduğunda ve kesir alanı 0 olmadığında sayı *olağanlaştırılmamış* (*denormalized* veya *subnormal*) kabul edilir. Bu durumda gizli 1 varsayılmaz; sayı $(-1)^i \times 0, \text{Kesir} \times 2^{-126}$ biçiminde yorumlanır. Bu sayede sıfıra çok yakın değerler (duyarlık kaybına rağmen) gösterilebilir. Tek duyarlıda en küçük pozitif olağanlaştırılmamış sayı $2^{-149} \approx 1,4 \times 10^{-45}$ 'tir; olağanlaştırılmamış sayılar olmasaydı sıfır ile en küçük olağan sayı ($2^{-126} \approx 1,2 \times 10^{-38}$) arasında temsil edilemeyen büyük bir boşluk kalırdı.

Sonsuz

Üs alanının tüm bitleri 1 ve kesir alanı 0 olduğunda sayı sonsuz olarak yorumlanır. İşaret bitine göre $+\infty$ ve $-\infty$ vardır. Sonsuz, taşma durumlarında otomatik olarak üretilir; örneğin çok büyük iki sayının çarpımı gösterilebilir aralığı aştığında sonuç $\pm\infty$ olur. Ayrıca sıfırdan farklı bir sayının sıfıra bölünmesi de sonsuz üretir: $5,0 \div 0,0 = +\infty$.

Sonsuz değerlerle aritmetik işlemler tanımlıdır: $n \div \infty = 0$, $\infty + \infty = \infty$, $\infty \times \infty = \infty$ vb. Bu sayede programlar taşma sonrasında bile çalışmaya devam edebilir.

NaN (Sayı Değil)

$0 \div 0$ işleminin sonucu nedir? Sonsuz olamaz; çünkü sonsuz, sıfırdan *farklı* bir sayının sıfıra bölünmesiyle tanımlanır. Sıfır da olamaz; çünkü $0 \times 0 = 0$ olsa da bölümün tek bir değeri yoktur; herhangi bir sayı çarpı sıfır sıfır eder. Sonuç matematiksel olarak *belirsizdir*. IEEE 754 bu tür belirsiz sonuçları temsil etmek için özel bir değer tanımlar: **NaN** (*Not a Number*, sayı değil).

NaN, üs alanının tüm bitleri 1 ve kesir alanı 0'dan farklı olduğunda kodlanır. Matematiksel olarak tanımsız olan her işlem NaN üretir:

İşlem	Neden tanımsız?
$0 \div 0$	Bölüm belirsiz
$\infty - \infty$	Fark belirsiz
$\infty \div \infty$	Oran belirsiz
$\infty \times 0$	Çarpım belirsiz
$\sqrt{-1}$	Gerçek sayılarda tanımsız

NaN'ın en dikkat çekici özelliği *bulaşıcı* olmasıdır: NaN içeren herhangi bir aritmetik işlemin sonucu yine NaN olur. Örneğin $\text{NaN} + 5 = \text{NaN}$, $\text{NaN} \times 0 = \text{NaN}$. Bu bulaşıcılık, programda bir yerde oluşan tanımsız sonucun tüm hesaplamaya sessizce yayılmasına neden olabilir.

Bir diğer sürpriz: NaN kendisine eşit değildir. IEEE 754'e göre $\text{NaN} \neq \text{NaN}$ her zaman *doğrudur* ve $\text{NaN} = \text{NaN}$ her zaman *yanlıştır*. Bu kural birçok programcı için beklenmediktir; ancak NaN'ı saptamanın en kolay yolunu sağlar: bir değişken kendisine eşit değilse NaN'dır.

Bilgi Notu 5.14: NaN ile Programlama Uygulaması

NaN, program çökmesine veya kural dışı duruma yol açmaz; program çalışmaya devam eder. Bu durum onu tehlikeli kılar: hata sessizce oluşur ve sonuçlara yayılır. Aşağıdaki C ve Python örneklerinde NaN'ın nasıl ortaya çıktığı ve ekrana nasıl yazdırıldığı görülmektedir:

```
1 #include <math.h>
2 #include <stdio.h>
3 int main() {
4     double x = 0.0 / 0.0;    // NaN üretilir
5     printf("%f\n", x);      // Çıktı: NaN
6     printf("%d\n", x == x);  // Çıktı: 0 (yanlış!)
7     return 0;
8 }
```

```
1 x = float('nan')
2 print(x)          # Çıktı: NaN
3 print(x == x)     # Çıktı: False
4 print(x + 5)      # Çıktı: NaN (bulaşıcılık)
```

Görüldüğü gibi program hata vermez; nan değeri ekrana yazdırılır ve hesaplamalar devam eder. Bu nedenle sayısal hesaplamalar yapan programlarda NaN denetimi önemlidir; C'de `isnan()` işlevi, Python'da `math.isnan()` işlevi bu amaçla kullanılır.

Bu özel değerlerle yapılan işlemlerin tam listesi Tablo 5.24'de özetlenmiştir.

Tablo 5.24: Kayan nokta özel değerleriyle yapılan işlemlerin sonuçları.

İşlem	Sonuç	Açıklama
$n \div \pm\infty$	0	Sonlu sayı sonsuza bölününce sıfıra yakınsar
$\pm\infty \times \pm\infty$	$\pm\infty$	İki sonsuzun çarpımı sonsuz
$n \div 0$ ($n \neq 0$)	$\pm\infty$	Sıfıra bölme sonsuz üretir
$(+\infty) + (+\infty)$	$+\infty$	Aynı yönlü sonsuzlar toplanabilir
$0 \div 0$	NaN	Belirsiz
$(+\infty) - (+\infty)$	NaN	Belirsiz
$\infty \div \infty$	NaN	Belirsiz
$\infty \times 0$	NaN	Belirsiz

5.6.5 Yuvarlama

Kayan nokta aritmetiğinde her işlemin sonucu, saklama alanının izin verdiği kadar fazla bit içerebilir. Örneğin iki 24 bitlik anlamlı kısmın çarpımı 48 bit üretir; ancak tek duyarlıda yalnızca 23 bit saklanabilir. Fazla bitlerin atılması gerekir; ama nasıl? İşte bu soruya yanıt veren mekanizma *yuvarlamadır*.

IEEE 754 Yuvarlama Yöntemleri

IEEE 754 dört yuvarlama yöntemi tanımlar:

1. **En yakın çifte yuvarlama (*round to nearest, ties to even*):** Varsayılan yöntemdir. Sayı, en yakınındaki gösterilebilir değere yuvarlanır. Eğer sayı tam iki gösterilebilir değerin ortasında ise, son biti çift (0) olan değere yuvarlanır.
2. **Sıfıra yuvarlama (*round toward zero*):** Sayının mutlak değeri küçültülerek sıfıra doğru yuvarlanır. Bu yöntem “kırpma” (*truncation*) da denir; fazla bitler basitçe atılır.
3. **Yukarıya yuvarlama (*round toward +∞*):** Sayı $+\infty$ yönünde, her zaman büyük değere yuvarlanır.
4. **Aşağıya yuvarlama (*round toward -∞*):** Sayı $-\infty$ yönünde, her zaman küçük değere yuvarlanır.

Neden “En Yakın Çifte” Varsayılandır?

Yuvarlama yönteminin seçimi, binlerce işlem art arda yapıldığında hataların birikmesini doğrudan etkiler. Bunu onluk tabanda basit bir örnekle görelim.

Diyelim ki onluk tabanda yalnızca 3 basamak saklayabiliyoruz ve aşağıdaki dört sayıyı toplamamız gerekiyor:

$$1,235 + 1,245 + 1,255 + 1,265$$

Gerçek toplam = 5,000. Şimdi her sayıyı önce 3 basamağa yuvarlayıp sonra toplayalım:

Yöntem	Yuvarlanan değerler	Toplam	Hata
Her zaman yukarı	1,24 + 1,25 + 1,26 + 1,27	5,02	+0,02
Her zaman aşağı	1,23 + 1,24 + 1,25 + 1,26	4,98	-0,02
En yakın çifte	1,24 + 1,24 + 1,26 + 1,26	5,00	0

“Her zaman yukarı” yöntemi sistematik olarak yukarı yanlışlık üretir; “her zaman aşağı” ise aşağı yanlışlık. “En yakın çifte” yöntemi ise tam ortadaki değerleri bazen yukarı bazen aşağı yuvarlayarak *yanlılığı sıfıra yakın tutar*. İşte IEEE 754’ün bu yöntemi varsayılan yapmasının nedeni budur: uzun hesaplamalarda hata birikimini en aza indirir.

Donanımda Yuvarlama: Koruma, Yuvarlama ve Yapışkan Bitler

Yuvarlama kararını verebilmek için donanımda hesaplama sonucunun saklanabilir bitlerinin ötesinde **üç ek bit** tutulur:

- **Koruma biti (*guard, G*):** Saklanan anlamlı kısmın hemen sağındaki ilk ek bit.
- **Yuvarlama biti (*round, R*):** Koruma bitinin sağındaki ikinci ek bit.
- **Yapışkan bit (*sticky, S*):** Yuvarlama bitinin sağında kalan *tüm bitlerin* VEYA’sı. Bu bitlerin herhangi biri 1 ise yapışkan bit 1 olur; kesin değerlerinin bilinmesine gerek yoktur.

Bu üç bitin ne anlama geldiğini somutlaştırmak için tek duyarlıklılı bir çarpma düşünelim. Tek duyarlıda anlamlı kısım 23 bit saklar; dolayısıyla çarpma sonucunda 23 bitten sonra gelen ilk bit koruma (G), ikinci bit yuvarlama (R), geri kalanların VEYA’sı ise yapışkan bittir (S):

$$1. \quad \underbrace{b_1 b_2 \dots b_{23}}_{\text{saklanan 23 bit}} \quad \underbrace{b_{24}}_G \quad \underbrace{b_{25}}_R \quad \underbrace{b_{26} b_{27} \dots}_S = \text{hepsinin VEYA'sı}$$

Saklanacak son bit $L = b_{23}$ 'tür. Yuvarlama kararı, GRS kombinasyonuna bakarak verilir. Bu üç bitin görevi, atılacak kısmın son gösterilebilir değerin *yarısından* küçük mü, büyük mü, yoksa tam yarısına eşit mi olduğunu belirlemektir:

- **G = 0** demek, atılacak kısım yarıdan küçüktür (R ve S ne olursa olsun). $\Rightarrow$ **Kırp** (aşağı yuvarla).
- **G = 1, RS $\neq$ 00** demek, atılacak kısım yarıdan büyüktür. $\Rightarrow$ **Yukarı yuvarla** (L 'ye 1 ekle).
- **G = 1, R = 0, S = 0** demek, atılacak kısım *tam yarıya* eşittir. $\Rightarrow$ **Berabere!** L tek ise (1) yukarı yuvarla, L çift ise (0) kırp. Bu “en yakın *çifte* yuvarlama” kuralıdır ve istatistiksel yanlılığı ortadan kaldırır.

Bunu bir örnekle görelim. Tek duyarlıda bir çarpma sonucu, olağanlaştırmadan sonra şu bit dizisini üretsin (yalnızca son bitler gösterilmiştir):

$$1. \quad \dots \quad \underbrace{1}_{L=b_{23}} \quad \underbrace{1}_G \quad \underbrace{0}_R \quad \underbrace{1}_S$$

$G = 1$ olduğundan atılacak kısım en az yarıdır. R ve S'ye bakıyoruz: $RS = 01$, dolayısıyla $\neq 00$. O halde atılacak kısım yarıdan *büyüktür* $\Rightarrow$ yukarı yuvarlama yapılır: L 'ye 1 eklenir ve saklanan son bit $L = 1 + 1 = 0$ olur (elde bir sonraki bite yayılır).

Eğer bit dizisi $\dots 1100$ ($G = 1, R = 0, S = 0$) olsaydı tam yarı noktası söz konusu olurdu. Bu durumda $L = 1$ tek olduğundan yine yukarı yuvarlanır ve L çift (0) yapılırdı. Ancak $L = 0$ olsaydı (çift), kırpma yapılır ve L olduğu gibi kalırdı.

Yapışkan bitin neden gerekli olduğunu anlamak önemlidir: yuvarlama ve koruma bitleri yalnızca iki ek bit verir; ancak gerçek sonuç bunların da ötesinde bitlere sahip olabilir. Eğer bu uzak bitlerden herhangi biri 1 ise, atılacak kısım “tam yarı” olamaz; mutlaka yarıdan büyüktür. İşte yapışkan bit, tüm bu uzak bitlerin tek bir VEYA işlemiyle özetlenerek bu ayrımın yapılabilmesini sağlar. Donanımda bu, uzun bir VEYA kapısı zinciriyle gerçekleştirilir ve yalnızca bir bitlik yer kaplar.

Yuvarlama Hatası ve Sonuçları

Yuvarlama, her kayan nokta işlemine küçük bir hata ekler. Bu hatanın büyüklüğü, anlamlı kısmın kaç bit sakladığına bağlıdır: tek duyarlıda yuvarlama virgülden sonraki 23. bitte, çift duyarlıda ise 52. bitte gerçekleşir. Bunu somutlaştırmak için $1 \div 3$ işlemini düşünelim; sonuç $0,333 \dots$ olup sonsuz tekrarlıdır ve her duyarlılıkta yuvarlanmak zorundadır:

Duyarlık	Saklanan değer	Hata
Tek (23 bit)	0,333 333 34...	$\sim 10^{-8}$ (virgülden sonra 7 doğru basamak)
Çift (52 bit)	0,333 333 333 333 3...	$\sim 10^{-16}$ (virgülden sonra 15 doğru basamak)

Görüldüğü gibi duyarlılığı artırmak, yuvarlama noktasını daha uzak bir bite taşır ve her işlemdeki hatayı doğrudan azaltır. Tek bir işlemde bu hata en kötü durumda son bitin yarısı

($\frac{1}{2}$ ulp, *unit in the last place*) kadardır ve önemsiz görünebilir; ancak binlerce işlem art arda yapıldığında bu küçük hatalar birikebilir.

Bilgi Notu 5.15: Kayan Nokta Duyarlılığı ve Günlük Etkisi

Kayan nokta aritmetiğinde her işlemde küçük yuvarlama hataları oluşabilir. Örneğin, çoğu programlama dilinde $0,1 + 0,2 \neq 0,3$ sonucu alınır; çünkü 0,1 ve 0,2 ikilik tabanda sonsuz tekrarlı kesir olduğundan tam olarak temsil edilemez:

```
1 >>> 0.1 + 0.2
2 0.30000000000000004
3 >>> 0.1 + 0.2 == 0.3
4 False
```

Bu tür hatalar biriktikçe bilimsel benzetimlerde, finansal hesaplamalarda ve mühendislik uygulamalarında beklenmedik sonuçlara yol açabilir. Bu nedenle sayısal çözümleme alanında hata yayılımı (*error propagation*) ve kararlılık (*numerical stability*) kavramları büyük önem taşır. Finansal uygulamalarda kayan nokta yerine sabit noktalı veya ondalık (*decimal*) aritmetik tercih edilmesinin nedeni de budur.

Özellikle tehlikeli bir durum *yıkıcı yokoluş* (*catastrophic cancellation*) olarak bilinir: birbirine çok yakın iki kayan nokta sayısı çıkarıldığında, anlamlı kısımdaki üst bitler birbirini götürür ve geriye kalan bitler yuvarlama hatalarından ibarettir. Örneğin $a = 1,00000001 \times 2^{10}$ ile $b = 1,00000000 \times 2^{10}$ çıkarıldığında sonuç $0,00000001 \times 2^{10} = 1,0 \times 2^2$ olur; burada 23 bitlik anlamlı kısımdan yalnızca 1 bit gerçek bilgi taşır, geri kalanı güvenilmezdir.

5.6.6 Kayan Nokta İşlemlerinde Toplama, Çarpma ve Bölme

Kayan nokta gösterimindeki sayılarla da tam sayılarla olduğu gibi dört işlem yapılabilir; ancak üs ve anlamlı kısmın ayrı ayrı ele alınması gerektiğinden bu işlemler tamsayı aritmetiğinden daha karmaşıktır.

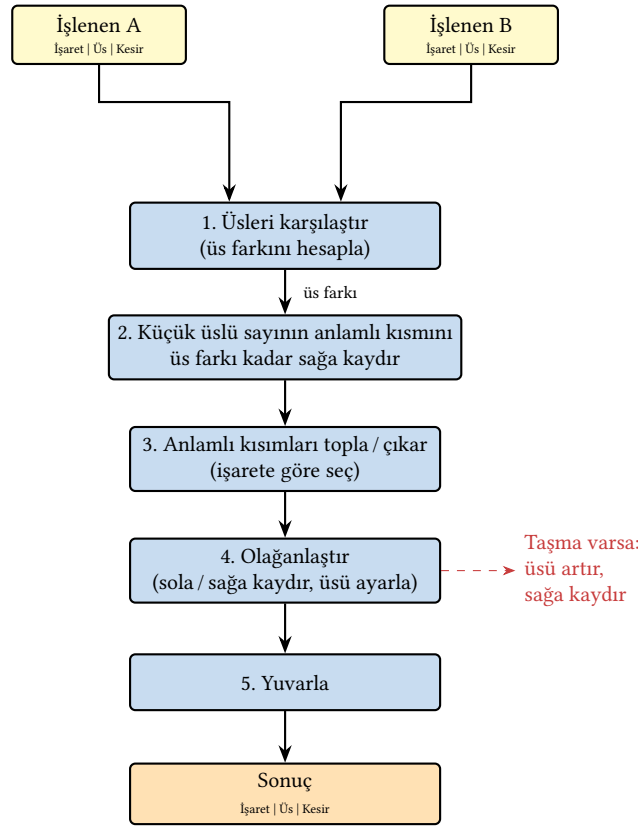
Toplama ve Çıkarma

İki kayan nokta sayısını toplamak için anlamlı kısımlarını doğrudan toplayamayız; çünkü her sayının virgüllü farklı bir konumdadır (üsleri farklıdır). Önce virgülleri aynı hizaya getirmemiz, ardından toplamayı yapıp sonucu yeniden olağan biçime getirmemiz gerekir. Bu süreç beş adımdan oluşur:

1. **Üsleri karşılaştır:** İki sayının üsleri arasındaki fark hesaplanır. Bu fark, virgülün ne kadar kaydırılması gerektiğini belirler.
2. **Anlamlı kısmı kaydır:** Küçük üslü sayının anlamlı kısmı, üs farkı kadar *sağa* kaydırılır. Böylece her iki sayı da aynı üsse sahip olur ve virgüller hizalanır. Sağa kaydırma sırasında dışarı kayan bitler koruma, yuvarlama ve yapışkan bitlerinde tutulur.
3. **Anlamlı kısımları topla veya çıkar:** Üsleri eşitlenen sayıların anlamlı kısımları, işaret bitlerine göre toplanır ya da çıkarılır.
4. **Olağanlaştır:** Sonuç $1,xxx \times 2^e$ biçiminde olmayabilir. Örneğin toplam $0,001 \dots$ biçiminde çıkarsa sola kaydırılıp üs azaltılır; $1x,xxx$ biçiminde çıkarsa sağa kaydırılıp üs artırılır. Bu adımda taşma veya alttan taşma da denetlenir.

5. **Yuvarla:** Sonuç, hedef duyarlılığa yuvarlanır. Yuvarlama sonucu olağan biçim bozulursa (örneğin yuvarlama bir elde üretirse) 4. adıma geri dönülür.

Bu işlemi gerçekleştiren donanımın blok şeması Şekil 5.30'de gösterilmiştir. Dikkat edilirse tamsayı toplamada tek bir toplayıcı yeterliyken, kayan nokta toplamada karşılaştırma, kaydırma, toplama, olağanlaştırma ve yuvarlama birimlerinin hepsine ihtiyaç vardır; bu nedenle kayan nokta toplayıcısı, tamsayı toplayıcısından çok daha fazla donanım alanı kaplar ve daha yavaş çalışır.



Şekil 5.30: Kayan nokta toplama/çıkarma donanımının blok şeması

Örnek 5.18

IEEE 754 tek duyarlıklı gösterimde $0,5 + (-0,375)$ toplamını hesaplayalım.

Çözüm 5.18:

Önce her iki sayıyı ikilik kayan nokta biçiminde yazalım:

$$\begin{aligned} 0,5 &= 1,0 \times 2^{-1} & (\text{üs} = -1) \\ -0,375 &= -1,1 \times 2^{-2} & (\text{üs} = -2) \end{aligned}$$

İki sayının üsleri farklı olduğundan anlamlı kısımları doğrudan toplayamayız; önce virgülleri hizalamamız gerekir.

Adım 1: Üsleri karşılaştır. Üs farkı $= (-1) - (-2) = 1$. Küçük üslü sayı $-1,1 \times 2^{-2}$ 'dir.

Adım 2: Anlamlı kısmı kaydır. Küçük üslü sayının anlamlı kısmı, üs farkı kadar (1 bit) sağa kaydırılır. Böylece her iki sayı da aynı üsse (-1) sahip olur:

$$-1,1 \times 2^{-2} \rightarrow -0,11 \times 2^{-1}$$

Adım 3: Anlamli kısımları topla. Üsler artık eşit olduğundan anlamli kısımları toplayabiliriz:

$$1,00 + (-0,11) = 0,01 \quad (\text{üs} = -1)$$

Adım 4: Olağanlaştır. Sonuç $0,01 \times 2^{-1}$ olağan biçimde değildir; baştaki bit 1 değil. Anlamli kısmı 2 bit sola kaydırıp üsü 2 azaltarak olağanlaştırırız:

$$0,01 \times 2^{-1} \rightarrow 1,0 \times 2^{-3}$$

Adım 5: Yuvarla. Bu örnekte fazla bit olmadığından yuvarlama gerekmez.

Sonuç: $1,0 \times 2^{-3} = 0,125_{10}$. Doğrulama: $0,5 + (-0,375) = 0,125$. ✓

Çarpma

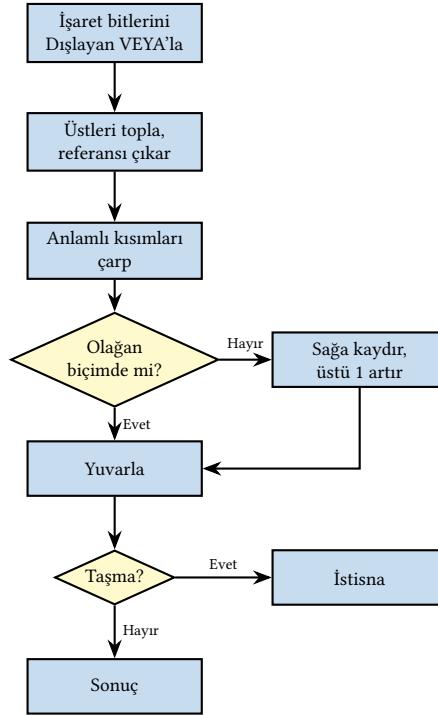
Kayan nokta çarpması, toplamadan farklı olarak üslerin eşitlenmesini gerektirmez; bunun yerine üsler toplanır, anlamli kısımlar çarpılır. Adımlar şu şekildedir:

1. **Özel durumları denetle:** İşlenenlerin sıfır, sonsuz veya NaN olup olmadığı denetlenir. Özel durum varsa sonuç doğrudan üretilir ve sonraki adımlar atlanır: $0 \times \infty = \text{NaN}$, $0 \times n = 0$, $\infty \times n = \pm\infty$ (sıfır hariç). Böylece gereksiz çarpma işlemi engellenir.
2. **İşareti belirle:** İki sayının işaret bitlerine Dışlayan VEYA uygulanır. İşaretler aynıysa sonuç artı, farklıysa eksidir.
3. **Üsleri topla:** Gerçek üsler toplanır. Eklentili gösterimde her iki üste de referans değeri (tek duyarlıda 127) eklenmiş olduğundan, eklentili üsler toplandığında referans iki kez sayılmış olur; bu nedenle referans bir kez çıkarılır:

$$\text{Sonuç üssü} = (E_A + E_B) - 127$$

4. **Anlamli kısımları çarp:** İki sayının anlamli kısımları ($1, f_A$ ve $1, f_B$) doğrudan çarpılır. n bitlik iki anlamli kısmın çarpımı $2n$ bit üretir; fazla bitler yuvarlama adımında kesilecektir.
5. **Olağanlaştır:** Çarpım $1x,xxx$ biçiminde çıkarsa (baştaki 1'in solunda fazladan bir bit varsa) anlamli kısım sağa bir kaydırılıp üs 1 artırılır. Bu adımda taşma veya alttan taşma da denetlenir.
6. **Yuvarla:** Sonuç hedef duyarlığa yuvarlanır. Toplamada olduğu gibi, yuvarlama olağan biçimi bozarsa olağanlaştırma tekrarlanır.

Kayan nokta çarpmasının akış şeması Şekil 5.31'te gösterilmiştir.



Şekil 5.31: Kayan nokta çarpma işleminin akış şeması

Örnek 5.19

IEEE 754 tek duyarlıklı gösterimde $1,5 \times 2^1$ ile $1,25 \times 2^1$ sayılarını çarpalım.

Çözüm 5.19:

Önce her iki sayıyı ikilik kayan nokta biçiminde yazalım:

$$A = 1,5 \times 2^1 = 1,1_2 \times 2^1 \quad (\text{onluk değer: } 3)$$

$$B = 1,25 \times 2^1 = 1,01_2 \times 2^1 \quad (\text{onluk değer: } 2,5)$$

Çarpmada toplamadan farklı olarak üsleri eşitlemeye gerek yoktur; üsler toplanır, anlamlı kısımlar ayrı ayrı çarpılır.

Adım 1: Özel durumları denetle. Her iki işlenen de olağan sayı; özel durum yok, devam.

Adım 2: İşareti belirle. Her iki sayının da işaret biti 0'dır (artı). Dışlayan VEYA: $0 \oplus 0 = 0$, dolayısıyla sonuç artıdır.

Adım 3: Üsleri topla. Gerçek üsleri doğrudan toplayabiliriz: $1 + 1 = 2$. Eklentili gösterimde ise her iki üste de referans (127) eklenmiş olduğundan, eklentili üsler toplandığında referans iki kez sayılır; bu nedenle bir kez çıkarılır:

$$E_{\text{sonuç}} = (1 + 127) + (1 + 127) - 127 = 128 + 128 - 127 = 129 \quad (\text{gerçek üs} = 129 - 127 = 2)$$

Adım 4: Anlamalı kısımları çarp. İkilik çarpmayı adım adım yapalım:

$$\begin{array}{r}
 1,10 \\
 \times 1,01 \\
 \hline
 110 \quad (1,10 \times 1) \\
 0000 \quad (1,10 \times 0, 1 \text{ bit sola}) \\
 1100 \quad (1,10 \times 1, 2 \text{ bit sola}) \\
 \hline
 1,1110
 \end{array}$$

Adım 5: Olağanlaştır. Çarpım $1,1110_2$ zaten $1,xxx$ biçimindedir; baştaki bit 1 olduğundan olağanlaştırma gerekmez. Eğer çarpım $1x,xxx$ biçiminde çıksaydı, sağa bir kaydırıp üsü 1 artıracaktık.

Adım 6: Yuvarla. Çarpım 4 bitlik anlamalı kısma tam olarak sığığından yuvarlama gerekmez. Gerçek donanımda 23 bitin ötesindeki bitler koruma, yuvarlama ve yapışkan bitleri kullanılarak yuvarlanırdı.

Sonuç: $1,1110_2 \times 2^2 = 111,10_2 = 7,5$. Doğrulama: $3 \times 2,5 = 7,5$. ✓

Bölme

Kayan nokta bölmesi çarpmaya benzer bir yapıda ilerler: üsler ayrı ayrı işlenir, anlamalı kısımlar bölünür. Ancak çarpmada üsler toplanırken bölmede *çıkarılır*; anlamalı kısımlar ise çarpılmak yerine bölünür. Bölme adımları şu şekildedir:

1. **Özel durumları denetle:** İşlenenlerin sıfır, sonsuz veya NaN olup olmadığı denetlenir. Özel durum varsa sonuç doğrudan üretilir ve sonraki adımlar atlanır: $n \div 0 = \pm\infty$, $0 \div 0 = \text{NaN}$, $\infty \div \infty = \text{NaN}$. Bölen sıfırsa bölme işlemi hiç yapılmaz.
2. **İşareti belirle:** Çarpmadaki gibi, iki sayının işaret bitlerine Dışlayan VEYA uygulanır. İşaretler aynıysa sonuç artı, farklıysa eksidir.
3. **Üsleri çıkar:** Bölünenin üssünden bölenin üssü çıkarılır. Eklentili gösterimde referans değeri her iki üste de eklenmiş olduğundan, çıkarma sonucunda referans kaybolur ve geri eklenmesi gerekir:

$$E_{\text{sonuç}} = (E_A - E_B) + 127$$

4. **Anlamalı kısımları böl:** Bölünenin anlamalı kısmı $(1, f_A)$ bölenin anlamalı kısmına $(1, f_B)$ bölünür. Bu bölme, daha önce gördüğümüz tam sayı bölme algoritmalarından (geri yükleme, geri yüklemez veya SRT) biriyle gerçekleştirilir.
5. **Olağanlaştır:** Bölüm $1,xxx$ biçiminde olmayabilir; örneğin $1,0 \div 1,1 = 0,110$ gibi bir sonuç çıkarsa sola kaydırılıp üs 1 azaltılır.
6. **Yuvarla:** Sonuç hedef duyarlığa yuvarlanır. Yuvarlama olağan biçimi bozarsa olağanlaştırma tekrarlanır.

Örnek 5.20

IEEE 754 tek duyarlıklı gösterimde $-7,0 \div 2,0$ bölmesini hesaplayalım.

Çözüm 5.20:

Önce her iki sayıyı ikilik kayan nokta biçiminde yazalım:

$$A = -7,0 = -1,11_2 \times 2^2 \quad (\text{üs} = 2)$$

$$B = 2,0 = 1,0_2 \times 2^1 \quad (\text{üs} = 1)$$

Adım 1: Özel durumları denetle. Her iki işlenen de olağan sayı; özel durum yok, devam.

Adım 2: İşareti belirle. A eksi, B artı. Dışlayan VEYA: $1 \oplus 0 = 1$, dolayısıyla sonuç eksidir.

Adım 3: Üsleri çıkar. Gerçek üsler: $2 - 1 = 1$. Eklentili gösterimde:

$$E_{\text{sonuç}} = (2 + 127) - (1 + 127) + 127 = 129 - 128 + 127 = 128 \quad (\text{gerçek üs} = 128 - 127 = 1)$$

Adım 4: Anlamalı kısımları böl.

$$1,11_2 \div 1,0_2 = 1,11_2$$

Bölen 1,0 olduğundan bölüm doğrudan bölünenin anlamalı kısmıdır.

Adım 5: Olağanlaştır. Sonuç $1,11_2$ zaten $1,xxx$ biçiminde olduğundan olağanlaştırma gerekmez.

Adım 6: Yuvarla. Bu örnekte yuvarlama gerekmez.

Sonuç: $-1,11_2 \times 2^1 = -11,1_2 = -3,5$. Doğrulama: $-7,0 \div 2,0 = -3,5$. ✓

5.7 RISC-V Kayan Nokta Buyrukları

Kayan nokta gösterimini öğrendiğimize göre, şimdi RISC-V'in bu sayılarla nasıl işlem yaptığına bakalım. Kayan nokta desteğinin buyruk kümesine nasıl eklendiği, mimariden mimariye farklılık gösterir. x86 mimarisinde kayan nokta işlemleri başlangıçta ayrı bir yardımcı işlemci yongasıyla (Intel 8087) destekleniyordu; daha sonra SSE ve AVX uzantılarıyla ana işlemciye tümleştirildi. ARM mimarisinde ise kayan nokta başlangıçta isteğe bağlı bir VFP (*Vector Floating-Point*) birimi olarak sunulmuş, ARMv8 ile birlikte zorunlu hale getirilmiştir; günümüzde NEON/SVE gibi SIMD uzantıları da kayan nokta işlemlerini destekler.

RISC-V bu yaklaşımlardan farklı olarak kayan nokta desteğini baştan modüler uzantılar biçiminde tasarlamıştır: F uzantısı tek duyarlı, D uzantısı çift duyarlı işlemleri ekler. Ayrı bir yardımcı işlemci yoktur; kayan nokta buyrukları doğrudan ana buyruk kümesinin uzantıları olarak tanımlanmıştır. İşlemci, tam sayı yazmaçlarının ($x0-x31$) yanı sıra $f0-f31$ arasında 32 adet kayan nokta yazmacı içerir. F uzantısında bu yazmaçlar 32 bit, D uzantısında ise 64 bit genişliğindedir. Bu uzantıların sağladığı buyruklar Tablo 5.25'de verilmiştir.

Tablo 5.25: RISC-V Kayan Nokta Buyrukları (F/D Uzantıları)

Buyruk	Söz Dizimi	Açıklama
<i>Yükle/Sakla Buyrukları</i>		
Tek Duyarlı Yükle	<code>flw fd, offset(rs1)</code>	Bellekten 4 bayt yükler (tek duyarlı)
Çift Duyarlı Yükle	<code>fld fd, offset(rs1)</code>	Bellekten 8 bayt yükler (çift duyarlı)
Tek Duyarlı Sakla	<code>fsw fs, offset(rs1)</code>	Yazmaçtan belleğe 4 bayt saklar (tek duyarlı)
Çift Duyarlı Sakla	<code>fsd fs, offset(rs1)</code>	Yazmaçtan belleğe 8 bayt saklar (çift duyarlı)
<i>Hesaplama Buyrukları</i>		
Topla	<code>fadd.s / fadd.d fd, fs1, fs2</code>	Kayan nokta toplama
Çıkar	<code>fsub.s / fsub.d fd, fs1, fs2</code>	Kayan nokta çıkarma
Çarp	<code>fmul.s / fmul.d fd, fs1, fs2</code>	Kayan nokta çarpma
Böl	<code>fdiv.s / fdiv.d fd, fs1, fs2</code>	Kayan nokta bölme
<i>Karşılaştırma Buyrukları</i>		
Eşitlik Karşılaştır	<code>feq.s rd, rs1, rs2</code>	Kayan nokta eşitlik karşılaştırması; eşitse $rd=1$, değilse $rd=0$
Küçüktür Karşılaştır	<code>flt.s rd, rs1, rs2</code>	Kayan nokta küçüktür karşılaştırması; $rs1 < rs2$ ise $rd=1$, değilse $rd=0$
Küçük Eşittir Karşılaştır	<code>fle.s rd, rs1, rs2</code>	Kayan nokta küçük eşittir karşılaştırması; $rs1 \leq rs2$ ise $rd=1$, değilse $rd=0$
<i>Dönüşüm ve Taşıma Buyrukları</i>		
KN'den Tam Sayıya	<code>fcvt.w.s rd, rs1</code>	Kayan noktadan tam sayıya dönüşüm (yuvarlayarak)
Tam Sayıdan KN'ye	<code>fcvt.s.w rd, rs1</code>	Tam sayıdan kayan noktaya dönüşüm
FP Yazmacından Kopyala	<code>fmv.x.w rd, rs1</code>	Kayan nokta yazmacındaki bitleri tam sayı yazmacına kopyalar (değer dönüşümü yapmaz)
Tam Sayı Yazmacından Kopyala	<code>fmv.w.x rd, rs1</code>	Tam sayı yazmacındaki bitleri kayan nokta yazmacına kopyalar (değer dönüşümü yapmaz)

Bilgi Notu 5.16: Intel Pentium FDIV Hatası

Kayan nokta aritmetiği hassas bir konudur; küçük bir tasarım hatası bile büyük sonuçlara yol açabilir. Bilgisayar tarihinin en ünlü donanım hatalarından biri de tam olarak bu alandadır.

İlk dönem işlemcilerde kayan nokta birimi ayrı bir yonga olarak satılırdı. Intel, 486DX işlemcisiyle kayan nokta birimini ana işlemcinin içine yerleştirdi; ardından da Pentium işlemcisini geliştirdi. İki işlemci de kayan nokta bölmesi yaparken SRT bölme algoritmasını (bkz. Bölüm 5.5.3) kullanarak her adımda 2 bölüm biti birden üretiyordu. Her adımda doğru bölüm rakamını seçebilmek için bir SRT arama tablosuna ihtiyaç vardı. Sorun, bu tablodaki 1066 hücreden beşinin yanlış doldurulmasıydı. Bu hatalı işlemlerden en yaygını 4195835/3145727 işleminin sonucunun 1,33382 yerine 1,33374 bulunmasıydı. Bu durumu 1994 Kasım ayında ilk fark eden kişi Lynchburg Kolejinden Thomas Nicely oldu. Intel'e haber vermesine rağmen yeterince ilgilenilmeyince haberi internette duyurdu. IBM, Pentium işlemcili bilgisayarlarının üretimini durdurdu. Intel yaklaşık 300 milyon dolar zarar gördü. Bölme buyruğu FDIV olduğu için bu hataya *FDIV hatası* denir.

Bilgi Notu 5.17: Patriot Füzesi Kazası: Yuvarlama Hatasının Bedeli

FDIV hatası maddi zarara yol açtı; ancak kayan nokta hataları can kaybına da neden olabilir. 1991 Körfez Savaşı'nda ABD ordusunun Patriot hava savunma sistemi, Suudi Arabistan'ın Dhahran kentinde bir Irak Scud füzesini durduramadı; füze bir askeri barınağa isabet etti ve 28 Amerikan askeri hayatını kaybetti.

Sorunun kaynağı, sistemin dahili saatindeki kayan nokta yuvarlama hatasıydı. Patriot'un bilgisayar zamanı 0,1 saniyelik artışlarla sayıyordu; ancak 0,1 ikilik tabanda sonsuz tekrarlı bir kesirdir ($0,0\overline{0011}_2$) ve 24 bitlik sabit noktalı gösterimde her adımda yaklaşık $9,5 \times 10^{-8}$ saniyelik bir hata oluşuyordu. Sistem 100 saatten fazla kesintisiz çalıştığında bu küçük hatalar birikti ve toplam zamanlama kayması yaklaşık 0,34 saniyeye ulaştı. Bu sürede bir Scud füzesi yaklaşık 600 metre yol alır; sistem, füzeyi aradığı yerde bulamadı ve önleme başarısız oldu.

Bu olay, kayan nokta yuvarlama hatalarının “küçük” ve “önemsiz” olmadığına trajik bir kanıttır. Biriken hatalar, yeterli süre geçtiğinde yaşamsal sonuçlara yol açabilir.

5.8 Yapay Zeka Çağında Kayan Nokta Biçimleri

IEEE 754'ün tek duyarlı (FP32) biçimi on yıllarca kayan nokta hesaplamalarının standart birimi oldu. Ancak 2010'ların ortasından itibaren derin öğrenme modelleri büyüdükçe FP32'nin maliyeti sorun olmaya başladı: milyarlarca parametrelili bir modelin her parametresi 32 bit kapladığında, hem bellek gereksinimi hem de veri aktarım bant genişliği devasa boyutlara ulaşır. Üstelik yapay zeka eğitiminde her parametrenin 7 onluk basamak duyarlılıkla saklanması çoğu zaman gereksizdir; ağırlık güncellemeleri zaten gürültülüdür ve birkaç basamak duyarlılık yeterlidir.

Bu gözlem, araştırmacıları daha dar kayan nokta biçimleri geliştirmeye yöneltti. Temel fikir yalındır: *her parametrenin kapladığı bit sayısını yarıya indirirsen, aynı belleğe iki kat daha fazla parametre sığar ve aynı bant genişliğiyle iki kat daha fazla veri taşırsın.* Ayrıca dar çarpıcılar

daha az alan kaplar ve daha az enerji tüketir; dolayısıyla aynı yonga üzerine çok daha fazla çarpma-toplama birimi yerleştirilebilir.

5.8.1 FP16: IEEE 754 Yarım Duyarlı

IEEE 754-2008 ile standartlaştırılan yarım duyarlı (*half precision*) biçim 16 bitten oluşur: 1 işaret + 5 üs + 10 kesir. Eklenti değeri 15'tir; gerçek üs aralığı -14 ile $+15$ 'tir.

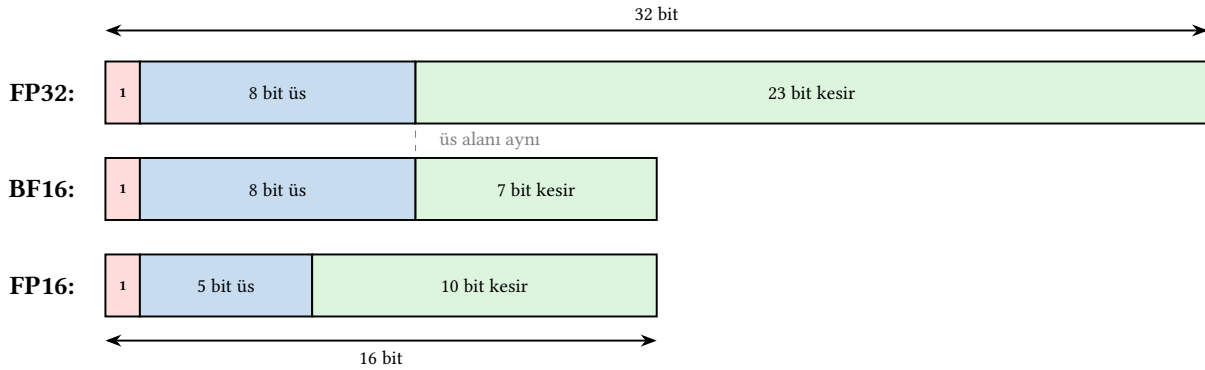
FP16 başlangıçta grafik uygulamaları (doku sıkıştırma, renk hesaplama) için tasarlanmıştı. Yapay zeka alanında da ilk dar biçim olarak yaygınlaştı; NVIDIA'nın 2016'da çıkardığı Pascal mimarisi (P100 GPU) FP16 desteğini yapay zeka eğitimi için öne çıkaran ilk büyük ölçekli donanımdı.

Ancak FP16'nın ciddi bir sınırlaması vardır: 5 bitlik üs alanı yalnızca -14 ile $+15$ arasında gerçek üs değerlerine izin verir; gösterilebilen en büyük sonlu değer yaklaşık $65\,504$ 'tür. Derin öğrenme eğitiminde ağırlık güncellemeleri sırasında gradyanlar ve kayıp değerleri bu aralığı kolayca aşabilir; bu durumda taşma (*overflow*) oluşur ve eğitim kararsızlaşır.

5.8.2 BF16: Google Brain'den Doğan Biçim

2018 yılında Google Brain ekibi, kendi TPU (*Tensor Processing Unit*) hızlandırıcıları için yeni bir 16 bitlik biçim tasarladı: **BF16** (*Brain Floating Point 16*). Tasarımın çıkış noktası şu sorunun yanıtıydı: “FP32'nin bit bütçesini yarıya indirirken hangi alandan kısmalıyız: üstten mi, kesirden mi?”

FP16, üs alanını 5 bite indirerek sayı aralığını daraltmıştı ve bu taşma sorunlarına yol açıyordu. Google mühendisleri tam tersi bir seçim yaptı: **üs alanını FP32 ile aynı (8 bit) tutup kesir alanını 23 bitten 7 bite indirdiler** (Şekil 5.32).



Şekil 5.32: FP32, BF16 ve FP16 bit düzenlerinin karşılaştırması. BF16, FP32 ile aynı 8 bitlik üs alanını korurken kesir alanını 7 bite indirmiştir.

Bu tasarım kararının sonuçları:

- **Sayı aralığı FP32 ile aynı:** 8 bitlik üs alanı sayesinde $\approx 10^{\pm 38}$ aralık korunur. FP16'daki taşma sorunu ortadan kalkar.
- **Duyarlık düşer:** 7 bitlik kesir alanı yalnızca $\approx 2-3$ onluk basamak duyarlık sağlar (FP32'nin 7 basamağına karşı). Ancak yapay zeka ağırlıkları için bu çoğu zaman yeterlidir.
- **FP32'ye dönüşüm önemsiz:** BF16, FP32'nin üst 16 bitinin kesilmesiyle (*truncation*) elde edilebilir; ek bir dönüşüm devresi gerekmez. Bu, karma duyarlık eğitiminde iki biçim arasında geçişi çok ucuz kılar.

BF16 ilk olarak Google’ın TPU v2 ve v3 hızlandırıcılarında kullanıldı. Kısa sürede NVIDIA (Ampere mimarisi, 2020), AMD, Intel ve ARM de BF16 desteği ekledi. Bugün yapay zeka eğitiminin varsayılan standart biçimi haline gelmiştir.

5.8.3 TF32: NVIDIA’nın Uzlaşısı

BF16 ve FP16 bellekte 16 bit kaplar ve FP32’ye göre iki kat hız kazandırır; ancak programcının veri tiplerini açıkça değiştirmesini gerektirir. NVIDIA, 2020’de Ampere mimarisini (A100 GPU) tanıtırken farklı bir yaklaşım izledi: mevcut FP32 kodunu hiç değiştirmeden hızlandırmak. Bunun için **TF32** (*TensorFloat-32*) adında bir ara biçim tasarladı.

TF32, 1 işaret + 8 üs + 10 kesir = 19 bitten oluşur. FP32’nin 8 bitlik üs alanını koruyarak aynı sayı aralığını ($\approx 10^{\pm 38}$) sunar; kesir alanını ise 23 bitten 10 bite indirerek çarpıcı devresini küçültür. 10 bitlik kesir, FP16 ile aynı duyarlılığı ($\approx 3-4$ onluk basamak) sağlar ve yapay zeka eğitimi için yeterlidir.

Peki neden doğrudan FP32 kullanılmaz? Çarpma devresinin alanı, kesir genişliğinin *kare-siyle* orantılı büyür: $23 \times 23 = 529$ bitlik bir çarpıcı yerine $10 \times 10 = 100$ bitlik bir çarpıcı yaklaşık 5 kat daha küçüktür ve daha az enerji tüketir. Aynı yonga alanına çok daha fazla çarpma birimi yerleştirilebilir; A100’ün Tensör Çekirdeği birimleri bu sayede FP32’ye göre 8 kat daha yüksek işlem gücü sunar.

TF32’nin en önemli özelliği, bellekte ayrı bir veri tipi olarak saklanmamasıdır: veriler bellekte FP32 olarak durur, yalnızca Tensör Çekirdeği çarpma birimleri içinde sessizce TF32’ye kırılarak çarpılır ve sonuç FP32 olarak döndürülür. Programcının kod değiştirmesine gerek yoktur; bu nedenle NVIDIA bunu “FP32’nin hızlı modu” olarak konumlandırmıştır.

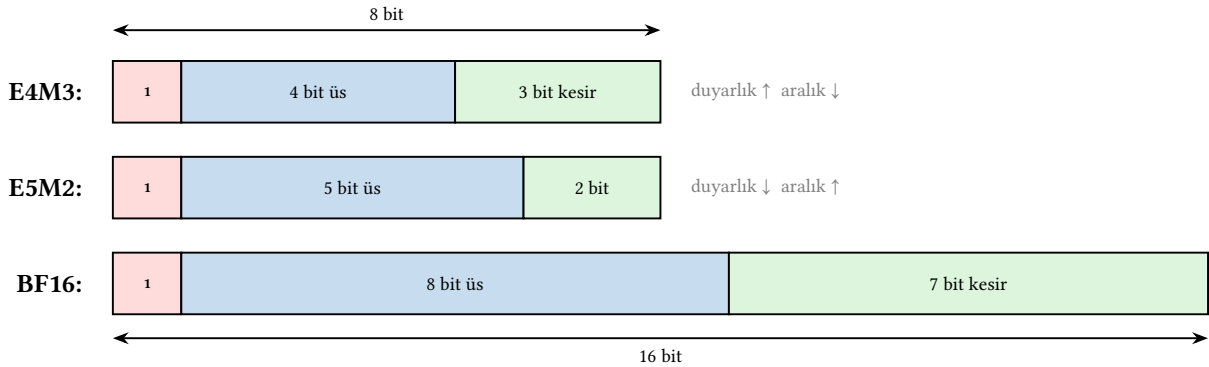
5.8.4 FP8: Çıkarım İçin En Dar Biçim

Yapay zeka çıkarımında (*inference*), yani eğitilmiş bir modelin yeni veriler üzerinde tahmin yapmasında, duyarlık gereksinimi eğitime göre daha da düşüktür. Bu gözlem, 8 bitlik kayan nokta biçimlerinin geliştirilmesine yol açtı.

FP8 standardı (2022’de NVIDIA, ARM ve Intel tarafından önerilen) iki alt biçim tanımlar:

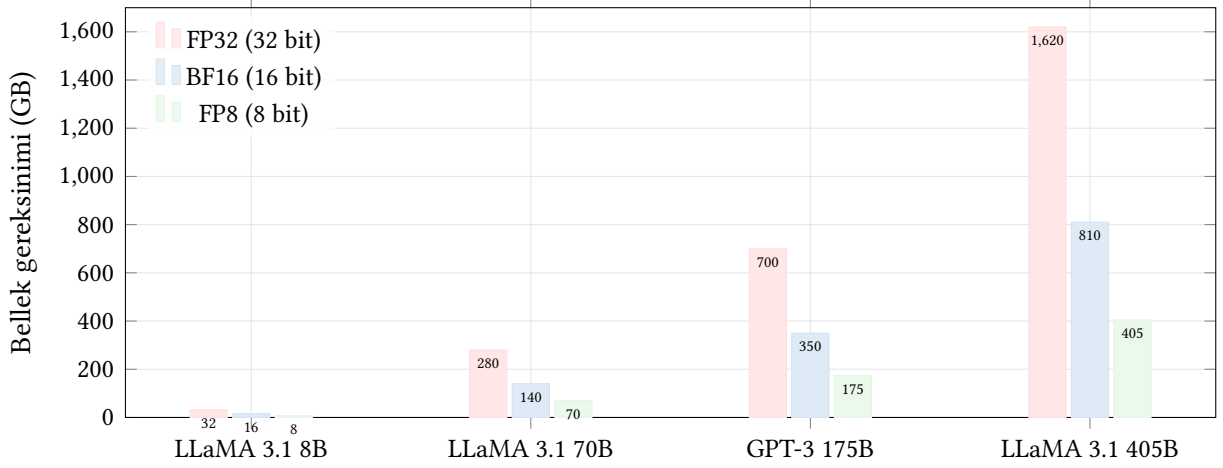
- **E4M3**: 1 işaret + 4 üs + 3 kesir. Daha yüksek duyarlık, daha dar aralık. Ağırlıklar ve etkinleştirmeler için.
- **E5M2**: 1 işaret + 5 üs + 2 kesir. Daha geniş aralık, daha düşük duyarlık. Gradyanlar için.

Bu iki alt biçimin bit düzenleri Şekil 5.33’de gösterilmiştir. Her ikisi de 8 bit kullanır; aradaki fark üs ve kesir alanlarının paylaşımındadır.



Şekil 5.33: FP8’in iki alt biçiminin bit düzenleri, BF16 ile karşılaştırmalı. E4M3, kesir alanına daha fazla bit ayırarak duyarlılığı artırır; E5M2 ise üs alanını genişleterek sayı aralığını büyütür.

Bu dar biçimlerin neden önemli olduğunu somutlaştırmak için günümüz yapay zeka modellerinin boyutlarına bakalım. GPT-3 modeli 175 milyar, LLaMA 3 modeli 405 milyar, bazı büyük modeller ise 1 trilyonun üzerinde parametreye sahiptir. Her parametre bir kayan nokta sayıdır ve bellekte saklanması gerekir. Şekil 5.34, farklı kayan nokta biçimlerinin bellek gereksinimini nasıl etkilediğini göstermektedir.



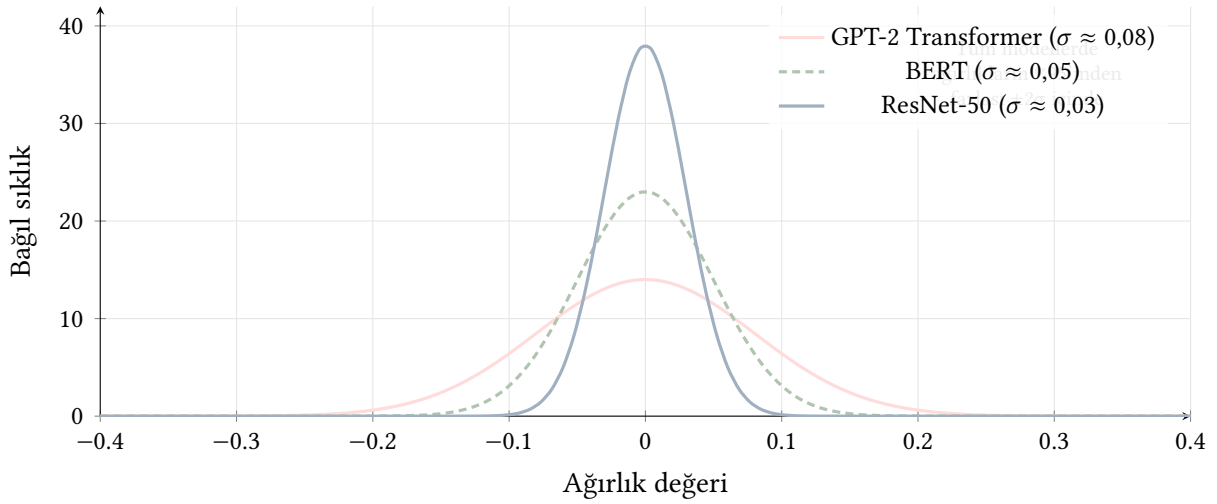
Şekil 5.34: Yapay zeka modellerinin farklı kayan nokta biçimlerindeki bellek gereksinimleri. Bit genişliğini yarıya indirmek, bellek gereksinimini de yarıya indirir.

Örneğin 405 milyar parametrelili LLaMA 3.1 modeli FP8 ile 405 GB belleğe sığar; bu, 80 GB bellekli beş GPU ile çalıştırılabilir bir boyuttur. Aynı model FP32 ile 1 620 GB gerektirir ve 20’den fazla GPU’ya ihtiyaç duyulur. Bit genişliğini daraltmak yalnızca bellek tasarrufu sağlamaz; aynı zamanda bellekten işlemciye veri aktarımını da hızlandırır; çünkü aynı bant genişliğiyle iki veya dört kat daha fazla parametre taşınabilir.

NVIDIA H100 GPU, FP8 modunda FP16’ya göre iki kat daha yüksek işlem gücü sunar.

Peki bu kadar bit kısmak modelin doğruluğunu ne kadar etkiler? Daha önce gördüğümüz gibi bit sayısını azaltmak yuvarlama hatasını artırır; ancak yapay zeka modellerinde ağırlıklar zaten gürültülüdür ve küçük yuvarlama hataları modelin genel davranışını çoğu zaman değiştirmez. Araştırmalar şunu göstermiştir:

- **FP32 → BF16:** Model doğruluğunda kayıp neredeyse sıfırdır (%0,1’in altında). Bunun nedenini anlamak için tipik bir yapay zeka modelinin ağırlık dağılımına bakmak yeterlidir. Şekil 5.35’de görüldüğü gibi ağırlıkların büyük çoğunluğu sıfıra yakın dar bir aralıkta yoğunlaşır; aşırı büyük veya küçük değerler çok nadirdir. Bu dağılımda ağırlıkların %95’inden fazlası yalnızca 2–3 onluk basamak duyarlılıkla yeterince temsil edilebilir; BF16’nın sunduğu duyarlılık tam da bu kadardır. Bu nedenle BF16, eğitimde FP32’nin yerini büyük ölçüde almıştır.
- **BF16 → FP8:** Doğruluk kaybı genellikle %0,5–1 aralığındadır. E4M3 biçiminde yalnızca 3 bitlik kesir alanı ($\approx 1,5$ onluk basamak) kaldığından, aşırı büyük veya küçük ağırlıklarda kırpmaya (*clipping*) gerekebilir. Bunu telafi etmek için eğitim sırasında *niceleme farkında-lıklı eğitim* (*quantization-aware training*) kullanılır: model, düşük duyarlılıkta çalışacağını “bilerek” eğitilir ve ağırlıklarını bu sınırlara göre uyarlar.
- **Çıkarımda daha aşırı niceleme:** Eğitilmiş bir modelin ağırlıkları, çıkarım sırasında 4 bitlik tam sayılara (INT4) kadar indirilebilir. Bu durumda doğruluk kaybı %1–3’e çıkabilir; ancak bellek gereksinimi FP32’ye göre 8 kat azalır. Bu teknik, büyük dil modellerinin



Şekil 5.35: Farklı yapay zeka modellerinin tipik ağırlık dağılımları. Tüm modellerde ağırlıklar sıfır etrafında çan eğrisi biçiminde yoğunlaşır; CNN modelleri (ResNet) daha dar, Transformer modelleri (GPT, BERT) daha geniş kuyruklu dağılım gösterir. Bu yoğunlaşma, düşük bitli kayan nokta biçimlerinin yeterli duyarlık sağlayabilmesinin temel nedenidir.

tek bir tüketici GPU’unda çalıştırılabilmesini sağlamıştır.

5.8.5 Karma Duyarlık Eğitimi

Önceki alt bölümlerde gördüğümüz biçimlerin her birinin güçlü ve zayıf yanları vardır: FP32 yüksek duyarlık sunar ama yavaş ve bellek açısından pahalıdır; BF16 hızlı ve verimlidir ama duyarlığı düşüktür. Peki neden tek bir biçim seçmek yerine ikisini birden kullanmıyoruz?

Sorun şudur: eğitim sırasında ağırlık güncellemeleri (gradyanlar) genellikle çok küçük değerlerdir; örneğin 10^{-5} mertebesinde. Bir ağırlık 1,0 iken 0,00001 eklemek istediğimizde, BF16’nın yalnızca 2–3 onluk basamak duyarlığı bu küçük farkı temsil edemez ve güncelleme *yuvarlanarak kaybolur*. Öte yandan tüm hesaplamayı FP32 ile yapmak, BF16’ya göre iki kat daha fazla bellek ve bant genişliği gerektirir; milyarlarca parametrelili bir modelde bu fark devasa boyutlara ulaşır.

Çözüm, iki biçimin güçlü yanlarını birleştiren *karma duyarlık (mixed precision)* yöntemidir. Tipik bir karma duyarlık eğitim döngüsü şöyle çalışır:

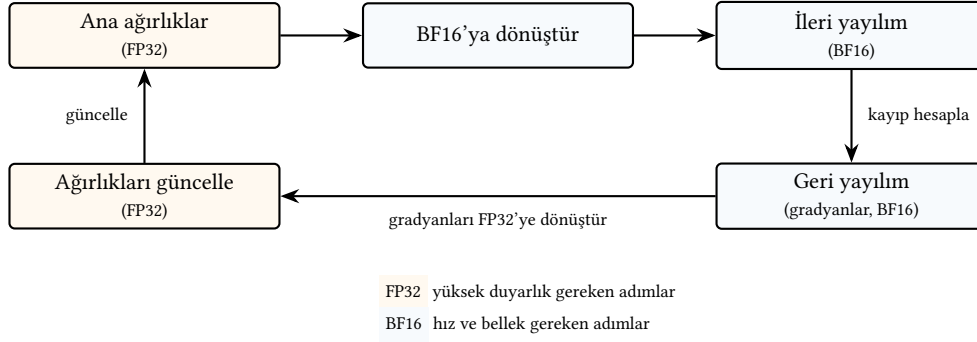
1. Ağırlıkların bir **ana kopyası FP32’de** tutulur (*master weights*); küçük gradyan güncellemelerinin kaybolmaması için.
2. **İleri ve geri yayılım BF16** (veya FP16) ile yapılır; hızlı ve bellek açısından verimlidir.
3. Hesaplanan gradyanlar **FP32’ye dönüştürülüp** ana ağırlıklara uygulanır; yuvarlama hatası birikmesi önlenir.

Bu döngü Şekil 5.36’te gösterilmiştir.

Bu yaklaşım, FP32’nin duyarlığını korurken BF16’nın hız ve bellek avantajından yararlanır. Günümüzün neredeyse tüm büyük dil modelleri (GPT, Gemini, Claude, LLaMA vb.) karma duyarlık ile eğitilmektedir.

5.8.6 Karşılaştırma

Tüm bu biçimlerin bir arada karşılaştırması Tablo 5.26’nde verilmiştir.



Şekil 5.36: Karma duyarlık eğitim döngüsü. Ana ağırlıklar FP32’de tutularak küçük güncellemelerin kaybolması önlenir; hesaplama yoğun ileri ve geri yayılım ise BF16 ile hızlı yapılır.

Tablo 5.26: Kayan nokta biçimlerinin karşılaştırması. Duyarlık sütunu yaklaşık onluk basamak sayısını gösterir.

Biçim	Bit	Üs	Kesir	Eklenti	Duyarlık	Kullanım
FP64 (double)	64	11	52	1023	~16	Bilimsel hesaplama
FP32 (float)	32	8	23	127	~7	Genel amaçlı
TF32	19	8	10	127	~3	GPU Tensör Çekirdeği (iç biçim)
BF16	16	8	7	127	~2	YZ eğitimi
FP16	16	5	10	15	~3	Grafik, hafif YZ eğitimi
FP8 (E5M2)	8	5	2	15	~1	YZ gradyanları
FP8 (E4M3)	8	4	3	7	~1	YZ ağırlıkları, çıkarım

Görüldüğü gibi, yapay zeka donanımları duyarlıktan ödün vererek *aynı yonga alanına daha fazla hesaplama birimi sığdırma* ve *aynı bant genişliğiyle daha fazla veri taşıma* ilkesiyle çalışır. Bu bölümde öğrendiğimiz toplayıcı ve çarpıcı devreleri, bu biçimlerin her birinde aynı temel ilkelerle (yalnızca daha dar bit genişliğiyle) çalışır.

5.9 Yanılgılar ve Tuzaklar

Sayısal aritmetik, donanım düzeyinde çalıştığı için hatalar genellikle sessizce oluşur ve yazılım katmanında fark edilmesi güçtür.

Yanılgı: Kayan nokta aritmetiği gerçek aritmetik gibi çalışır

Kayan nokta sayıları sınırlı duyarlığa sahiptir. Matematikteki toplama işleminin değişme ve birleşme özellikleri kayan noktada her zaman geçerli değildir: $(a + b) + c \neq a + (b + c)$ olabilir. Bu durum, büyük ölçekli bilimsel hesaplamalarda ve yapay zeka eğitiminde ciddi doğruluk sorunlarına yol açabilir.

Yanılgı: Taşma yalnızca büyük sayılarla ortaya çıkar

İşaretsiz aritmetikte iki artı sayının toplamı eksi çıkabilir (örneğin 8 bitte $+120 + 10 = -126$). Bu durum yalnızca büyük sayılarda değil, veri tipi sınırlarına yaklaşıldığında her zaman oluşabilir.

2004'te Comair havayolu şirketinin tüm uçuşlarını iptal etmesi, bir 16 bitlik sayaç taşmasından kaynaklanmıştır.

Yanılığ: Çarpma ve bölme işlemleri aynı sürede tamamlanır

Çarpma, çağdaş işlemcilerde genellikle 3–5 saat çevriminde tamamlanırken, bölme 20–40 çevrim sürebilir. Bu nedenle derleyiciler mümkün olduğunca bölmeyi kaydırma veya çarpma ile değiştirmeye çalışır. Örneğin $x/4$ yerine $x \gg 2$ kullanmak çok daha hızlıdır.

Yanılığ: Daha dar kayan nokta biçimi her zaman daha hızlıdır

BF16 ve FP8 gibi dar biçimler tek başına daha hızlı çarpma ve daha az bellek sunar; ancak duyarlık kaybı birikebilir. Yapay zeka eğitiminde yalnızca BF16 kullanmak, küçük gradyan güncellemelerinin yuvarlanarak kaybolmasına yol açar. Bu nedenle uygulamada tek bir dar biçim değil, karma duyarlık yöntemi kullanılır; hız gereken yerde dar biçim, duyarlık gereken yerde FP32.

Tuzak: İşaretsiz ve işaretli karşılaştırmayı karıştırmak

C dilinde $-1 > 2u$ ifadesi *doğrudur* çünkü derleyici işaretsiz karşılaştırma yapar ve -1 , ikiye tümleyende en büyük işaretsiz sayıya ($2^{32} - 1$) dönüşür. Bu tür tür dönüşüm hataları güvenlik açıklarının yaygın kaynaklarından biridir.

Tuzak: IEEE 754 özel değerlerini göz ardı etmek

Sıfıra bölme sonucu oluşan Inf (sonsuz) ve tanımsız işlemlerden kaynaklanan NaN (Not a Number) değerleri, denetlenmezse programda sessizce yayılarak tüm hesaplamaları geçersiz kılabilir. $\text{NaN} == \text{NaN}$ ifadesinin *yanlış* döndürmesi, yeni programcılar için sık karşılaşılan bir sürprizdir.

Tuzak: Kayan nokta sayılarını doğrudan eşitlik ile karşılaştırmak

Yuvarlama hataları nedeniyle $\text{if } (x == 0.3)$ gibi doğrudan eşitlik denetimleri neredeyse her zaman başarısız olur. Örneğin $0.1 + 0.2$ ifadesinin sonucu 0.30000000000000004 olduğundan 0.3 ile eşit çıkmaz. Doğru yöntem, iki sayının farkının küçük bir eşik değerinden (ϵ) az olup olmadığını denetlemektir: $|a - b| < \epsilon$.

Tuzak: Tam sayı ve kayan nokta sıfıra bölmeyi karıştırmak

Kayan nokta aritmetiğinde sıfıra bölme sessizce $\pm\infty$ üretir ve program çalışmaya devam eder. Tam sayı aritmetiğinde ise sıfıra bölme bir kural dışı durum (*exception*) tetikler; RISC-V'te sonuç olarak özel bir değer yazılır, x86'da ise program çökebilir. Bu iki davranışın karıştırılması, özellikle tam sayı ile kayan nokta arasında tür dönüşümü yapılan kodlarda tehlikeli hatalara yol açar.

5.10 Gerçek Dünyada Aritmetik

Bu bölümde öğrendiğimiz toplayıcı, çarpıcı ve kayan nokta devreleri bugün her işlemcinin ayrılmaz bir parçasıdır. Ancak bu her zaman böyle değildi. Kayan nokta biriminin ayrı bir yonga olarak satıldığı, hatta birçok bilgisayarda hiç bulunmadığı bir dönem yaşanmıştır.

5.10.1 Yardımcı İşlemci Dönemi (1980’ler)

Intel’in 8086 işlemcisi (1978) kayan nokta donanımına sahip değildi. Kayan nokta işlemleri gereken kullanıcılar, ayrı bir yonga olan **Intel 8087** yardımcı işlemcisini (*coprocessor*) satın alıp ana karta takardı. 8087, 80 bitlik genişletilmiş duyarlılıkta kayan nokta aritmetiği sunan ve kendi yazmaç yığına sahip bağımsız bir yongaydı. 8087 takılı değilse, aynı işlemler yazılımla (tamsayı buyrukları kullanılarak) taklit edilirdi; bu ise 10 ila 100 kat daha yavaştı.

Bu ayırık yapı 80286/80287 ve 80386/80387 çiftlerinde de sürdü. Bilgisayar reklamlarında “matematik yardımcı işlemci yuvası mevcut” ifadesi bir satış noktasıydı; çünkü 8087 tek başına birkaç yüz dolar tutuyordu ve birçok kullanıcı bu ek maliyetten kaçınıyordu. Sonuç olarak, yazılım geliştiricileri programlarının kayan nokta birimi olmadan da çalışacağını garanti etmek zorundaydı.

5.10.2 Tümleşik Kayan Nokta Birimi (1989–)

Intel, **80486DX** işlemcisiyle (1989) kayan nokta birimini ilk kez işlemci yongasının içine aldı. Bu geçiş, Moore yasasının sağladığı transistör bütçesi artışıyla mümkün oldu: 80386’daki 275 000 transistörden 80486’daki 1,2 milyona atlama, kayan nokta birimini de içerecek yeterli alanı sağladı. O günden bu yana her genel amaçlı işlemci tümleşik kayan nokta birimleriyle donatılmaktadır.

5.10.3 Günümüz: SIMD, Vektör ve Matris Birimleri

Bugünün işlemcileri tek seferde yalnızca bir kayan nokta işlemi yapan birimlerle yetinmez. ARM mimarisindeki **NEON** ve **SVE/SVE2** uzantıları, tek bir buyrukla birden fazla kayan nokta işlemi gerçekleştiren **SIMD** (*Single Instruction, Multiple Data*) birimleri sunar. Apple’ın M serisi yongaları (M1–M5) bu ARM uzantılarını yüksek başarımlı çekirdeklerinde kullanır; ayrıca **AMX** (*Apple Matrix coprocessor*) adlı özel bir matris çarpma birimi barındırır. AMX, bu bölümde öğrendiğimiz çarpma-toplama birimlerinden oluşan bir dizilimdir ve yapay öğrenme iş yüklerini hızlandırmak için tasarlanmıştır.

x86 dünyasında da benzer bir evrim yaşanmıştır: **MMX** (1997), **SSE** (1999), **AVX** (2011), **AVX-512** (2017) ve en son **AVX10/AMX** (2023) ile aritmetik birimlerin genişliği ve sayısı sürekli artmıştır. NVIDIA ve Google gibi şirketler ise bu bölümde gördüğümüz **Tensör Çekirdekleri** ve **TPU** gibi özel amaçlı aritmetik hızlandırıcılarla bu evrimi daha da ileri taşımıştır.

Terim Kökenleri

yapay öğrenme (*machine learning*): Türkçeye sıklıkla “makine öğrenmesi” olarak aktarılır. Ancak bu çeviri dilbilgisel olarak hatalıdır: Türkçede “makine öğrenmesi” biçiminde bir isim tamlaması kurulmaz; tıpkı “çocuk öğrenmesi” demediğimiz gibi. Doğru tamlama “makinenin öğrenmesi” ya da “makineyle öğrenme” olurdu. TDK’nın önerdiği karşılık *yapay öğrenme* olup “yapay zekâ” ile koştur bir yapıdadır.

Bu tarihsel yolculuk bir noktayı vurgular: bu bölümde öğrendiğimiz temel yapı taşları (toplayıcılar, çarpıcılar ve kayan nokta devreleri) on yıllar boyunca değişmeden kalmıştır. Değişen,

bu yapı taşlarından kaç tanesinin tek bir yonga üzerine sığdırılabildiğidir.

5.11 Özet

Bu bölümde bilgisayarın “hesap makinesi” kısmını, yani aritmetik işlemleri nasıl gerçekleştirdiğini, adım adım gördük. Bölüme başlarken bu konunun günümüz yapay zeka donanımları için ne denli temel olduğunu vurgulamıştık; şimdi bunu somut sayılarla pekiştirelim.

Toplamadan başlayarak, basamaklar arasındaki elde geçişini dalga eldeli ve önceden elde üreten toplayıcılarla çözdük. Çıkarmanın ikiye tümleyen yardımıyla toplamaya dönüştürüldüğünü ve bu sayede aynı donanımın hem toplama hem de çıkarma için kullanılabildiğini öğrendik. Taşma gibi kural dışı durumları da ele aldık.

Mantık işlemlerini de ele aldıktan sonra tüm bu parçaları bir araya getirerek 1 bitlik AMB tasarladık. Bu 1 bitlik AMB’den 32 tanesini yan yana koyarak RISC-V’in temel buyruklarını yürütebilen 32 bitlik bir AMB elde ettik.

Çarpma işlemini tekrarlanan toplama ve kaydırmaya dayanan algoritmalarla gerçekleştirdik; işaretli sayılarla doğrudan çarpım yapabilen Booth algoritmasını inceledik. Bölme için ise geri yüklemeli, geri yüklemesiz ve SRT algoritmalarını gördük.

Tam sayıların ötesine geçerek kayan nokta gösterimini öğrendik. IEEE 754 standardı, eklentili üs gösterimi, özel değerler (sıfır, sonsuz, NaN) ve yuvarlama yöntemlerini gördük. Korumaya, yuvarlama ve yapışkan bitlerinin yuvarlama kararını nasıl belirlediğini, kayan nokta toplama ve çarpmasının adım adım nasıl gerçekleştirildiğini inceledik. RISC-V’te kayan nokta işlemleri F ve D uzantılarıyla desteklenir.

Bölümün son kısmında yapay zeka çağının getirdiği yeni kayan nokta biçimlerini ele aldık: BF16, TF32, FP8 gibi dar biçimlerin neden geliştirildiğini, bu biçimlerin bellek ve hız avantajlarını, ağırlık dağılımının düşük bitli gösterimi neden mümkün kıldığını ve karma duyarlık eğitiminin bu biçimleri nasıl bir arada kullandığını gördük.

Bu bölümde tasarladığımız toplayıcı ve çarpıcı devreleri, günümüzün en güçlü yongalarının temel yapı taşlarıdır. Tablo 5.27’te bazı güncel yongalardaki aritmetik birim sayıları verilmiştir.

Tablo 5.27: Güncel yongalardaki aritmetik birim sayıları

Yonga	Çarpma-Toplama Birimi	Duyarlık	Kullanım
NVIDIA H100 GPU	16 896	FP32	Yapay zeka eğitimi
NVIDIA B200 GPU	18 432	FP32	Yapay zeka eğitimi
Google TPU v5e	16 384	BF16	Yapay zeka eğitimi
Apple M4 GPU	1 280	FP32	Taşınabilir cihazlar
Intel Gaudi 3	1 536	BF16	Yapay zeka çıkarımı

Görüldüğü gibi bir GPU veya yapay zeka hızlandırıcısı, bu bölümde öğrendiğimiz toplayıcı ve çarpıcı birimlerinden on binlercesini bir yonga üzerinde barındırır. Her bir çarpma-toplama birimi, çarpma ve toplama işlemini tek saat darbesinde gerçekleştiren birleşik bir devredir. Bu birimlerin hızı ve enerji verimliliği, doğrudan bu bölümde gördüğümüz tasarım kararlarına (önceden elde üreten toplayıcılar, Wallace ağacı çarpıcılar, kayan nokta yuvarlama devreleri) bağlıdır.

Artık elimizde çalışan bir AMB var. Bir sonraki bölümde bu AMB'yi kullanarak bir veri yolunun ve onu denetleyen birimin nasıl tasarlandığını göreceğiz. Bu işlemlerin RISC-V programlarındaki kullanımı için Bölüm 4'te bakınız.

Alıştırmalar

Alıştırma 5.1. ★★ Aşağıdaki işlemleri yapan 2 adet 4 bitlik veri girişi, 1 elde girişi ve 2 bitlik işlem girişi olan bir Aritmetik Mantık Birimi (AMB) tasarlayın: Toplama, Çıkarma, VE, VEYA. Tasarımınızı yaparken elinizde bir toplayıcı olduğunu varsayabilirsiniz.

Alıştırma 5.2. ★★★ 8 bitlik, A ve B olarak iki girişi olan, 2'ye tümleyen yöntemiyle çıkarma ($A-B$), toplama, Dışlayan VEYA, VEYA, VE, A'nın değili, B'nin değili ve SIFIR (giriş ne olursa olsun sonuç sıfır), A, B, Bir (giriş ne olursa olsun sonuç 1), A'nın ikiye tümleyeni, B'nin ikiye tümleyeni işlemlerini yapan bir AMB tasarlayın, devrelerini çizin ve doğruluk tablolarını gösterin.

Alıştırma 5.3. ★★★ Birisi 4 bitlik diğeri 8 bitlik iki yazmaç ve 4 bitlik bir AMB kullanarak 4 bitlik işaretli iki sayıyı çarpan bir devrenin çizelgesini gösterin ve çarpma algoritmasını açıklayın. Algoritmanın çalıştığını 1001×1100 işleminin sonucunu adım adım yazarak gösterin.

Alıştırma 5.4. ★★ Önceden elde üreten toplayıcının sıradan toplayıcılara göre ne farkı vardır? Çalışma mantığı nedir? Toplayıcının her bir bitindeki girişlere göre ilgili 1 bitlik toplayıcının yaptığı işlemi açıklayın.

Alıştırma 5.5. ★★ Kayan nokta işlemlerinde toplama nasıl yapılır? $1,1001 \times 2^{-2}$ sayısı ile $1,1101 \times 2^{-4}$ sayısını toplayarak kayan nokta toplamasının aşamalarını açıklayın. Bu tür işlemlerde taşma olabilir mi? Olabilirse kaç tür taşma olabilir?

Alıştırma 5.6. ★ Kayan nokta işlemleri yapılırken bir sayı sıfıra bölündüğünde ne sonucu çıkar? Sıfır sıfıra bölündüğünde (0/0) ise ne sonucu çıkar? Bu sonuçlar IEEE 754 biçiminde nasıl gösterilir?

Alıştırma 5.7. ★★ Aşağıdaki onluk tabanda gösterilen sayıları 32 bitlik IEEE-754 biçiminde gösterin:

1. 135,8243
2. 1,125
3. 6,375
4. 11,125
5. 0,00025
6. 17,0625

Alıştırma 5.8. ★★★ Eğer bir işlemci tasarımcısının elinde 16 bitlik bir alan varsa bu alanla gösterebileceği ondalıklı sayıların sınırı nedir? 16 bitlik alanda işaretli ondalıklı sayıların gösterilmesi için bu alanın nasıl bölünmesi gerekir? 135,8343 sayısının, 16 bitlik alanda ikilik tabanda virgülden sonra en fazla kaç basamağı gösterilebilir?

Alıştırma 5.9. ★★★ Ondalıkli sayıların işlemci içinde saklanması için toplam 5 bitlik saklama birimi kullanılacaktır. Saklama biriminin en soldaki biti işaret için ayrılacak geri kalan 4 bit ise sayının gösterilmesi için kullanılacaktır.

1. Eğer 4 bitlik kısım yalnızca anlamlı kısmı göstermek için kullanılırsa gösterilebilecek sayılar hangi aralıkta olur?
2. Eğer 4 bitlik kısım yalnızca üst göstermek için kullanılırsa gösterilebilecek sayılar hangi aralıkta olur? Gösterilebilecek en küçük duyarlılık ne olur?
3. Eğer 2 bitlik kısım üst için 2 bitlik kısım da sayının anlamlı parçası için kullanılırsa gösterilebilecek sayılar hangi aralıkta olur?

Aralıklarda hem gösterilebilecek en küçük ve en büyük sayıları hem de duyarlılık sınırlarını gösterin.

Alıştırma 5.10. ★★ Kayan nokta işlemlerinde toplama nasıl yapılır? $1,0011 \times 2^{-3}$ sayısı ile $1,1101 \times 2^2$ sayısını toplayarak kayan nokta toplamasının aşamalarını açıklayın. Bu tür işlemlerde taşma olabilir mi? Olursa kaç tür taşma olabilir?

Alıştırma 5.11. ★★ Booth algoritmasını kullanarak $(-13) \times (11)$ çarpma işlemini 8 bitlik ikiye tümleyen gösteriminde adım adım gerçekleştirin. Her adımda A yazmacının, Q yazmacının ve sayacın değerlerini gösteren bir tablo oluşturun.

Alıştırma 5.12. ★★ Booth algoritmasının standart ve değiştirilmiş (radix-4) sürümlerini kullanarak 15×7 çarpma işlemini gerçekleştirin. Her iki yöntemde kaç toplama/çıkarma işlemi gerektiğini karşılaştırın.

Alıştırma 5.13. ★★★ 4 bitlik iki işaretli sayının çarpılması sırasında oluşan kısmi çarpımları yazın. Bu kısmi çarpımları Wallace ağacı yöntemiyle indirgeyerek sonucu bulun. Wallace ağacının dizi çarpıcıya göre üstünlüğünü gecikme açısından açıklayın.

Alıştırma 5.14. ★★ 4×4 bitlik bir dizi çarpıcının yapısını çizin. $A = 1011$ ve $B = 1101$ sayılarını bu çarpıcı üzerinde çarpın. Toplam kaç tam toplayıcı ve yarım toplayıcı gerektiğini hesaplayın.

Alıştırma 5.15. ★★ Geri yüklemeli bölme algoritmasını kullanarak $(11010100)_2$ sayısını $(1011)_2$ sayısına bölün. 3. Yol donanımını kullanarak her adımda kalan yazmacının ve bölüm yazmacının değerlerini gösteren bir tablo oluşturun.

Alıştırma 5.16. ★★ Geri yüklemesiz bölme algoritmasını kullanarak $(10110)_2$ sayısını $(101)_2$ sayısına bölün. Her adımda kalan yazmacının değerini ve yapılan işlemi (toplama/çıkarma) belirtin. Geri yüklemeli yöntemle göre üstünlüğünü açıklayın.

Alıştırma 5.17. ★ Aşağıdaki 32 bitlik IEEE 754 gösterimlerinin hangi özel değeri temsil ettiğini belirleyin:

- (a) 0 11111111 000000000000000000000000
- (b) 1 11111111 000000000000000000000000
- (c) 0 11111111 100000000000000000000000
- (d) 0 00000000 000000000000000000000000

- (e) 1 00000000 000000000000000000000000
 (f) 0 00000000 00000000000000000000000001

Alıştırma 5.18. ★ IEEE 754 tek duyarlı gösterimde (32 bit) gösterilebilen en küçük pozitif olağanlaştırılmamış sayının değerini hesaplayın. Bu değeri olağanlaştırılmış gösterimdeki en küçük pozitif sayıyla karşılaştırın.

Alıştırma 5.19. ★★ IEEE 754 tek duyarlı gösterimde $1,101 \times 2^3$ ile $1,011 \times 2^2$ sayılarını çarpın. Üs toplama, anlamlı kısım çarpma, olağanlaştırma ve yuvarlama adımlarını ayrı ayrı gösterin.

Alıştırma 5.20. ★★ IEEE 754 tek duyarlılık gösterimde $1,110 \times 2^5$ sayısını $1,100 \times 2^2$ sayısına bölün. Üs çıkarma, anlamlı kısım bölme, olağanlaştırma ve yuvarlama adımlarını ayrı ayrı gösterin.

Alıştırma 5.21. ★★ IEEE 754 tek duyarlı gösterimde $1,0001 \times 2^5$ ile $1,1100 \times 2^1$ sayılarını toplayın. Üs hizalama, toplama, olağanlaştırma ve yuvarlama adımlarını gösterin. Hizalama sırasında kaybedilen bitlerin sonucu nasıl etkilediğini tartışın.

Alıştırma 5.22. ★★ 4 bitlik iki sayı $A = 1011$ ve $B = 0110$ toplandığında, üretme (G_i) ve yayma (P_i) sinyallerini her bit için hesaplayın. Ardından elden önceden üretme formüllerini kullanarak C_1 , C_2 , C_3 ve C_4 elde sinyallerini doğrudan hesaplayın.

Alıştırma 5.23. ★★★ 16 bitlik bir önceden elde üreten toplayıcı, dört adet 4 bitlik önceden elde üretici bloğu ve bir adet ikinci düzey önceden elde üretme birimi kullanılarak tasarlanır.

- Her 4 bitlik blok için grup üretme (G^*) ve grup yayma (P^*) sinyallerinin nasıl hesaplandığını açıklayın.
- 16 bitlik toplayıcının toplam kapı gecikmesini hesaplayın (bir kapının gecikmesi t_g olsun).
- Aynı boyuttaki zincirleme elde toplayıcının gecikmesiyle karşılaştırın.

Alıştırma 5.24. ★★ Aşağıdaki 8 bitlik ikiye tümleyen sayı çiftleri için toplama ve çıkarma işlemlerini gerçekleştirin. Her işlemde taşma olup olmadığını belirleyin ve taşma tespit kuralını açıklayın.

- $A = 01100100$, $B = 00110010$ (toplama)
- $A = 01110000$, $B = 01010000$ (toplama)
- $A = 10010000$, $B = 10110000$ (toplama)
- $A = 01100000$, $B = 10100000$ (çıkarma: $A - B$)

Alıştırma 5.25. ★★★ Bir işlemci 16 bitlik IEEE 754 yarım duyarlı (half-precision) kayan nokta biçimi kullanır (1 işaret, 5 üs, 10 anlamlı kısım biti).

- Bu biçimde gösterilebilen en büyük sonlu artı sayıyı hesaplayın.
- $(17,125)_{10}$ sayısını bu biçime çevirin.
- Bu biçimde iki sayı toplanırken taşma (overflow) ve alttan taşma (underflow) durumlarını birer örnekle açıklayın.

Alıştırma 5.26. ★★ Dört bitlik $A = 1011$ ve $B = 0110$ işlenenleri için önceden elde üreten toplayıcı tasarımında kullanılan P (yayılm) ve G (üretim) değerlerini hesaplayın.

- Her bit konumu için $P_i = A_i \oplus B_i$ ve $G_i = A_i \cdot B_i$ değerlerini bulun.
- C_4 'ü doğrudan P_i , G_i ve $C_0 = 0$ kullanarak (herhangi bir C_i beklemeden) hesaplayın.
- Bu sonucu dalgalanmalı taşımali toplayıcıyla (RCA) karşılaştırın: kaç kapı gecikmesi farkı vardır?

Alıştırma 5.27. ★★★ Booth kodlaması kullanarak $(-6) \times 5$ çarpma işlemini gerçekleştirin. İşlenenler 8 bitlik ikiye tümleyen biçimindedir ($(-6)_{10} = 11111010_2$, $5_{10} = 0000101_2$).

- Başlangıç değerlerini ayarlayın (çarpan, çarpılan, kısmi çarpım yazmacı).
- Her adımda Booth çizelgesine göre toplama, çıkarma ya da işlem yapmama kararı verin ve adım adım tabloyu doldurun.
- Sonucu 16 bitlik ikiye tümleyen biçiminde gösterin ve 10'lu tabanla doğrulayın.

Alıştırma 5.28. ★★ $(-13,625)_{10}$ sayısını IEEE 754 tek duyarlıklı (32 bit) kayan nokta biçimine dönüştürün.

- Sayının tam kısmını ve ondalık kısmını ayrı ayrı ikili tabana çevirin.
- Olağanlaştırılmış biçimi ($1, \text{anlamli\_kisim} \times 2^{\text{üs}}$) yazın.
- İşaret, eklentili üs ($\text{bias} = 127$) ve anlamli kısım alanlarını belirleyerek 32 bitlik ikili kalıbı oluşturun.

Alıştırma 5.29. ★★ Aşağıdaki 32 bitlik IEEE 754 tek duyarlıklı kayan nokta kalıbını 10'lu tabana çevirin:

1 10000100 1011000000000000000000

- İşaret, üs ve anlamli kısım alanlarını belirleyin.
- Gerçek üs değerini hesaplayın ($\text{önyargı} = 127$).
- Sonucu ondalık sayı olarak yazın.

Alıştırma 5.30. ★★★ IEEE 754 tek duyarlıklı biçimde $A = 1,5 \times 2^3$ ve $B = 1,25 \times 2^1$ sayılarını elle toplayın.

- Üsleri eşitlemek için küçük üslü sayının anlamli kısmını kaydırın.
- Anlamli kısımları toplayın.
- Gerekirse sonucu olağanlaştırın ve en yakın çift (round-to-nearest-even) kuralıyla yuvarlayın.
- Nihai sonucu 32 bitlik IEEE 754 kalıbı olarak yazın.

Alıştırma 5.31. ★ Bir RISC-V işlemcisinde 32 bitlik AMB (Aritmetik Mantık Birimi) aşağıdaki denetim sinyalleri için hangi işlemi gerçekleştirir? Her satır için işlem adını ve RISC-V buyruk karşılığını (mümkünse) yazın.

Alıştırma 5.32. ★★ Aşağıdaki 4-bit işaretli (ikiye tümleyen) toplama ve çıkarma işlemlerinin her birinde taşma oluşup oluşmadığını belirleyin. Taşma tespiti için taşıma girişi ile çıkışı karşılaştırma kuralını ($C_{in} \neq C_{out}$) uygulayın.

AMBİşlem	İnvert A	İnvert B	Taşıma Gir.	İşlem	Buyruk
AND	0	0	0	?	?
OR	0	0	0	?	?
TOPLA	0	0	0	?	?
TOPLA	0	1	1	?	?

- (a) $0111 + 0001$ (artı + artı)
- (b) $1000 + 1111$ (eksi + eksi)
- (c) $0110 + 0101$ (artı + artı)
- (d) $1001 - 0111$ (çıkarma: ikiye tümleyeni ekle)

Alıştırma 5.33. ★★★ Dört adet 4-bit sayıyı (A, B, C, D) koşut olarak çarpan bir Wallace ağacı çarpıcısının kısmi çarpım matrisini oluşturun. $A = 0101, B = 0011, C = 0110, D = 0001$ için:

- (a) Her çift için ($A \times B$ ve $C \times D$) kısmi çarpım matrisini yazın.
- (b) Wallace ağacının ilk indirgeme katmanında hangi tam toplayıcılar (FA) ve yarım toplayıcılar (HA) kullanılır?
- (c) Toplam kaç FA ve HA gerekir? Bir dalgalanmalı çarpıcıyla karşılaştırın.

Alıştırma 5.34. ★★ $13 \div 3$ bölme işlemini 1. yol geri yüklemeli bölme algoritmasıyla adım adım gerçekleştirin. İşlenenler 4 bittir; bölen ve kalan yazmaçları 8 bittir.

- (a) Başlangıç değerlerini tabloya yazın (bölen sola hizalanmış, kalan = bölünen, bölüm = 0).
- (b) Her adımda Kalan – Bölen işlemini yapın; sonuç eksiye geri yükleyin ve bölümü 0-bitleyin, artıysa kalan güncellenip bölümü 1-bitleyin.
- (c) Böleni sağa kaydırın; 4 adım sonra bölüm ve kalanı ondalıkla doğrulayın.

Alıştırma 5.35. ★★ 8 bitlik elde ile seçmeli toplayıcıyı 12 bite genişletmek istiyorsunuz. Üç adet 4 bitlik blok kullanacaksınız: Blok 0 (bit [3:0]) normal, Blok 1 (bit [7:4]) ve Blok 2 (bit [11:8]) elde ile seçmeli yapıda.

- (a) Blok diyagramını çizin. Kaç adet 4 bitlik toplayıcı ve çoklayıcı gerekir?
- (b) Dalga eldeli 12 bitlik toplayıcıya göre gecikme karşılaştırması yapın ($t_{mux} = t_g$ alın).
- (c) Hızlanma oranını ve alan artışını hesaplayın.

Alıştırma 5.36. ★ Yapay zeka uygulamalarında (derin öğrenme, sinir ağı eğitimi ve çıkarımı) IEEE 754 yarım duyarlı (FP16, 16 bit) kayan nokta biçimi neden tercih edilir?

- (a) Tek duyarlı (FP32) ile yarım duyarlı (FP16) biçimlerini bellek kullanımı ve bant genişliği açısından karşılaştırın.
- (b) FP16'nın daha düşük duyarlılığına karşın derin öğrenmede neden yeterli olduğunu açıklayın.
- (c) NVIDIA Tensör Çekirdeği gibi donanımların FP16 işlemlerini neden çok daha hızlı gerçekleştirebildiğini tartışın.

Sayı Sistemleri ve Kodlama



Göbekli Tepe – Şanlıurfa

İnsanlığın bilinen en eski yapısı olan Göbekli Tepe, uygarlığın başlangıç noktasıdır. Sayı sistemleri de bilgisayar biliminin başlangıç noktasıdır: her şey ikilik tabanda bir sayıyla başlar.

Öğrenme Hedefleri

Bu eki tamamladığınızda aşağıdakileri yapabileceksiniz:

- İkilik, sekizlik ve on altılık tabanlar arasında dönüşüm yapmak.
- İkilik tabanda toplama, çıkarma ve çarpma işlemlerini elle gerçekleştirmek.
- İşaret-büyüklik, bire tümleyen ve ikiye tümleyen gösterimlerini karşılaştırmak.
- Bir sayının ikiye tümleyen gösterimini bulmak ve taşıma koşullarını belirlemek.
- Gray kodunun yapısını ve kullanım alanlarını açıklamak.
- İKO (BCD) gösterimini ve onluk tabanlı aritmetikteki rolünü tanımlamak.
- ASCII ve Unicode karakter kodlama standartlarını açıklamak.
- IEEE 754 kayan nokta gösteriminde bir sayıyı bileşenlerine ayırmak ve geri birleştirmek.

Bu ekte, bilgisayar mimarisinin temelini oluşturan sayı sistemleri ve kodlama yöntemleri ele alınmaktadır. İkilik taban, taban dönüşümleri, işaretli sayı gösterimleri ve karakter kodlama gibi konular, kitabın ana bölümlerinde sıkça başvurulan ön bilgileri oluşturur.

A.1 İkilik Tabanda İşlemler

Günlük hayatta 10'luk tabanı kullanırız çünkü on ayrı rakamımız (0–9) vardır. İkilik taban ise yalnızca iki rakamdan (0 ve 1) oluşur. Bu tabanda “2” rakamı yoktur; tıpkı 10'luk tabanda “10” diye tek bir rakam olmadığı gibi. Sayısal devrelerin doğal dili ikilik taban olduğu için bilgisayar mimarisinin temelini oluşturur.

A.1.1 Taban Değiştirme İşlemleri

Değişik sayı tabanlarında gösterilen sayıları 10'luk tabana çevirmek için, her basamağın konum değerini hesaplamaya dayanan çözümleme yöntemi kullanılır. t tabanında n basamaklı bir sayının değeri:

$$(b_{n-1} b_{n-2} \dots b_1 b_0)_t = \sum_{i=0}^{n-1} b_i \times t^i \quad (\text{A.1})$$

Burada b_i , sağdan i . basamaktaki rakamı gösterir (b_0 en sağdaki basamaktır).

İkilik tabanda $t = 2$ olduğu için, her basamaktaki bit ikinin ilgili kuvveti ile çarpılır ve sonuçlar toplanır.

Örnek A.1

İkilik tabandaki 1010011 sayısını 10'luk tabana çevirin.

Çözüm:

$$\begin{aligned} (1010011)_2 &= 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 64 + 0 + 16 + 0 + 0 + 2 + 1 \\ &= (83)_{10} \end{aligned}$$

Örnek A.2

İkilik tabandaki 100101010 sayısının 10'luk tabanda değeri nedir?

Çözüm:

$$\begin{aligned}(100101010)_2 &= 1 \times 2^8 + 0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 256 + 32 + 8 + 2 \\ &= (298)_{10}\end{aligned}$$

Sayılar onluk tabandan ikilik tabana çevrilmek istendiğinde, sayının içerisinde ikinin üssü olabilecek değerler aranır. Sayıdan küçük ve ikinin üzeri olan en büyük değer bulunduğunda, sonuçta ortaya çıkacak ikilik tabandaki sayının ilgili basamağına 1 değeri verilir ve sayıdan ikinin üssü olarak bulunan değer çıkarılır. Bu işlem tüm basamaklar için yinelenir.

Örnek A.3

$(83)_{10}$ sayısını çıkarma yöntemini kullanarak ikilik tabana çevirin.

Çözüm:

İşlem Adımı	İkilik Tabandaki Sayı
$(83)_{10} - 2^6 = 19$	1
$19 < 2^5$	10
$19 - 2^4 = 3$	101
$3 < 2^3$	1010
$3 < 2^2$	10100
$3 - 2^1 = 1$	101001
$1 - 2^0 = 0$	1010011

Onluk tabandaki bir sayıyı ikilik tabana çevirmek için sayıyı sürekli ikiye bölerek kalana bakma yöntemi de kullanılabilir.

Örnek A.4

$(83)_{10}$ sayısını ikiye bölme yöntemini kullanarak ikilik tabana çevirin.

Çözüm:

İşlem Adımı	Kalan
$83/2 = 41$	1
$41/2 = 20$	1
$20/2 = 10$	0
$10/2 = 5$	0
$5/2 = 2$	1
$2/2 = 1$	0, Sonuç = 1

Bölme işlemlerinin ardından, son bölme işleminin sonucu en büyük basamağı ve bölme işlemlerinin kalanları diğer basamakları oluşturacak biçimde kalanlar sırayla yazılır ve

sonuç $(83)_{10} = (1010011)_2$ olarak bulunur.

A.1.2 İkilik Tabanda Dört İşlem

İkilik tabanda yapılan dört işlemde, onluk tabanda kullanılan tüm kurallar geçerlidir.

Toplama işlemi:

0 ve 1 dışında bir rakam kullanılamadığı için, ikilik tabanda onluk tabandan farklı olarak, iki basamağın toplamı 1'den büyük çıktığında toplam basamağına toplamın 2'ye bölümünden kalan yazılır ve soldaki basamağa bir elde aktarılır.

Örnek A.5

$(1101011)_2$ sayısı ile $(11011)_2$ sayısını toplayın.

Çözüm:

$$\begin{array}{r} 1101011 \\ + 11011 \\ \hline 10000110 \end{array}$$

Toplama sağdan sola yapılır. İki bitin toplamı 2'ye ulaştığında $(1 + 1 = 10)$ o basamağa 0 yazılır ve soldaki basamağa bir elde aktarılır. En soldaki basamaktan taşan elde, sonucun sekizinci bitini oluşturur.

Çıkarma işlemi:

Onluk tabandan farklı olarak, elde alınacak olan yan basamaktan 10 yerine 2 alınır.

Örnek A.6

$(1101011)_2$ sayısından $(11011)_2$ sayısını çıkarın.

Çözüm:

$$\begin{array}{r} 1101011 \\ - 11011 \\ \hline 1010000 \end{array}$$

Çıkarma sağdan sola yapılır. Bir basamakta çıkarılan bit daha büyükse, soldaki basamaktan 2 borç alınır (onluk tabandaki 10 borç almaya karşılık gelir).

Bilgisayar donanımında çıkarma işlemi genellikle ayrı bir devre ile yapılmaz. Bunun yerine çıkarılacak sayının ikiye tümleyeni alınır ve toplama devresi kullanılır: $A - B = A + (-B)$. Bu sayede tek bir toplayıcı devresi hem toplama hem de çıkarma işlemini gerçekleştirebilir (bkz. Ek B).

Çarpma işlemi:

İkilik tabandaki çarpma işleminde, ara çarpımların toplanması sırasında ikilik tabanda yapılan toplama işleminin kuralları uygulanır.

Örnek A.7

$(1101)_2$ sayısı ile $(11)_2$ sayısını çarpın.

Çözüm:

$$\begin{array}{r} 1101 \\ \times 11 \\ \hline 1101 \\ 1101 \\ \hline 100111 \end{array}$$

Bölme işlemi:

İkili tabanda, çarpma ve çıkarma işlemlerindeki kurallar gözetilerek bölme işlemi yapılmaktadır.

Örnek A.8

$(110101)_2$ sayısını $(110)_2$ sayısına bölün.

Çözüm:

Onluk bölmedeki gibi, bölünenin solundaki basamaklardan başlanır. Bölen sığmıyorsa bölüm basamağına 1, sığmıyorsa 0 yazılır:

- İlk 3 bit (110) : bölen 110 'a eşit $\Rightarrow$ bölüm biti = 1, kalan = 000.
- Sonraki bit indirilir: $0001 < 110 \Rightarrow$ bölüm biti = 0.
- Sonraki bit indirilir: $00010 < 110 \Rightarrow$ bölüm biti = 0.
- Son bit indirilir: $000101 < 110 \Rightarrow$ bölüm biti = 0.

$$(110101)_2 \div (110)_2 = (1000)_2 \quad \text{kalan: } (101)_2$$

Doğrulama: $(53)_{10} \div (6)_{10} = (8)_{10}$, kalan $(5)_{10}$.

A.1.3 8'lik ve 16'lık Tabanda İşlemler

Onluk tabanda kullanılan sayılar ikili tabanda gösterilirken, onluk tabandaki sayı büyüdükçe, ikili tabandaki sayının basamak sayısı hızla artar. İkili tabanda yalnızca iki rakam olduğu için, ortaya çıkan bu basamak sayısı fazla sayıların programcı tarafından yazılması, okunması ve takip edilmesi zordur. Bu nedenle bilgisayarlarda çalıştırılacak çevirici dil kodlarının yazımında kolaylık sağlaması amacıyla sayılar 8'lik ya da 16'lık sayı tabanlarında gösterilir.

$8 = 2^3$ ve $16 = 2^4$ olduğu için, ikili tabanda yazılmış bir sayıyı 8'lik tabana (İngilizcesi: *octal*) çevirmek için sayının basamaklarını üçerli, 16'lık tabana (İngilizcesi: *hexadecimal*) çevirmek için de sayının basamaklarını dörderli gruplamak gerekir. Tablo A.1'de 16'lık tabandaki her basamağın 10'luk, 8'lik ve 2'lik tabanlardaki karşılıkları verilmiştir.

Tablo A.1: 16'lık tabandaki sayıların diğer tabanlardaki gösterimleri

16'lık	10'luk	8'lik	2'lik
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	8	10	1000
9	9	11	1001
A	10	12	1010
B	11	13	1011
C	12	14	1100
D	13	15	1101
E	14	16	1110
F	15	17	1111

Örnek A.9

Onluk tabandaki 1.223.357 sayısını ikilik, sekizlik ve onaltılık tabanda gösterin.

Çözüm:

İkiye bölme yöntemiyle (bkz. Örnek A.4) sayı ikilik tabana çevrilir:

$$(1.223.357)_{10} = (100\ 101\ 010\ 101\ 010\ 111\ 101)_2$$

Sekizlik tabana çevirmek için basamaklar üçerli gruplanır:

$$\underbrace{100}_4 \underbrace{101}_5 \underbrace{010}_2 \underbrace{101}_5 \underbrace{010}_2 \underbrace{111}_7 \underbrace{101}_5 \Rightarrow (4525275)_8$$

Onaltılık tabana çevirmek için basamaklar dörderli gruplanır:

$$\underbrace{0001}_1 \underbrace{0010}_2 \underbrace{1010}_A \underbrace{1010}_A \underbrace{1011}_B \underbrace{1101}_D \Rightarrow (12AABD)_{16}$$

A.2 İşaretli Sayılar

Şimdiye kadar hep sıfır ve pozitif sayılarla çalıştık. Peki ya eksi sayıları nasıl göstereceğiz? Bellekte saklanan bir bit dizisinin “pozitif mi, negatif mi” olduğunu donanım kendiliğinden bilemez; bu tamamen programcının ya da derleyicinin tercihiine bağlıdır. Aynı bit dizisi, kullanılan buyruğa göre hem işaretli hem de işaretsiz bir sayı olarak yorumlanabilir.

Eksi sayıları ikilik tabanda göstermek için tarih boyunca birçok yöntem önerilmiştir. Bu yöntemlerin ortak noktası, sayının en soldaki bitini (en anlamlı bit) işaret bilgisi için ayırmaktır: bu bit 1 ise sayı eksi, 0 ise artı kabul edilir. Tablo A.2’de dört farklı yöntemle 4 bitlik gösterimler karşılaştırmalı olarak verilmiştir.

A.2.1 Ayrı İşaret Biti

Bir sayının işaretini göstermenin en kolay yolu saklama alanının bir bitini işaret bilgisini tutmak için ayırmaktır. Ayrı işaret biti kullanan sistemlerde sayının en soldaki, en anlamlı bitinin (İngilizcesi: *most significant bit – msb*) işaret biti olduğu kabul edilir. Bu işaret bitinin değeri eğer sayı eksi ise 1, artı ise 0 olarak belirlenir.

Ayrı işaret biti kullanma yönteminde işaret biti sayıya dahil değildir ve ayrıca işlenir. Bu yöntemde +0 ve –0 olmak üzere iki ayrı sıfır değeri ortaya çıkar.

A.2.2 Artık N

Artık-N (İngilizcesi: *Excess-N*) gösteriminde sayılara belirli bir N sayısı eklenir ve sayı bu ekleme işleminin ardından ikilik tabana çevrilir. Örneğin, artık-3'te 0 sayısının nasıl gösterildiğini bulmak için önce sayıya 3 eklenir ve çıkan sonuç ikilik tabana çevrilerek 0011 bulunur. Bu yöntemin en önemli kullanım alanı IEEE 754 kayan nokta standardında üs alanının kodlanmasıdır (bkz. Bölüm A.6). Artık-N gösteriminin avantajı, saklanan değerlerin doğrudan büyüklük karşılaştırmasına olanak tanınmasıdır: saklanan bit dizileri üzerinde işaretsiz karşılaştırma yapmak, gerçek üs değerlerini karşılaştırmakla eşdeğerdir.

Örnek A.10

4 bitlik artık-3 gösteriminde –2 ve +5 sayılarını gösterin.

Çözüm:

Artık-3 gösteriminde her sayıya 3 eklenir ve sonuç ikilik tabana çevrilir:

- –2: $-2 + 3 = 1 = (0001)_2$
- +5: $5 + 3 = 8 = (1000)_2$

4 bitlik artık-3 gösteriminde gösterilebilecek aralık –3 ile +12 arasındadır; 0000 en küçük (–3), 1111 en büyük (+12) değeri temsil eder.

A.2.3 Bire Tümleyen Gösterimi

İkilik tabandaki bir sayının bire tümleyenini almak için her bir basamaktaki rakam 0 ise 1, 1 ise 0 yapılarak sayının her biti ters çevrilir. Örneğin 10101 sayısının bire tümleyeni 01010'dır. Bire tümleyen gösteriminde de +0 ve –0 olmak üzere iki ayrı sıfır değeri vardır. Bu çift sıfır sorunu ve toplama işleminde “daireysel elde” gerektirmesi gibi dezavantajları nedeniyle bire tümleyen gösterimi günümüz bilgisayarlarında neredeyse hiç kullanılmamaktadır; yerini ikiye tümleyen almıştır.

Örnek A.11

8 bitlik bire tümleyen gösteriminde –45 sayısını gösterin.

Çözüm:

Önce 45 sayısı 8 bitlik ikilik tabana çevrilir: $45 = (00101101)_2$. Bire tümleyeni almak için tüm bitler ters çevrilir:

$$-45 = \overline{00101101} = (11010010)_2$$

Doğrulama: Bire tümleyen düzeninde bir sayı ile tümleyeninin toplamı tüm bitleri 1 olan sayıyı verir: $00101101 + 11010010 = 11111111 \checkmark$

A.2.4 İkiye Tümleyen Gösterimi

Bire tümleyen yönteminde sıfır değeri için iki ayrı sayı kullanılması nedeniyle gösterilebilecek sayılardan bir tanesi israf edilmektedir. İkiye tümleyen yöntemi bu sorunu ortadan kaldırmak için tasarlanmıştır. Günümüzde kullanılan sayısal devrelerin büyük çoğunluğu ikiye tümleyen yöntemi kullanmaktadır.

Bir sayının ikiye tümleyenini almak için sayının bire tümleyeni alınır ve sonuca bir eklenir.

Örnek A.12

−1 sayısının 4 bitlik 2'ye tümleyen gösteriminde karşılığı nedir?

Çözüm:

1 sayısının 4 bitle gösterilen karşılığı 0001'dir. Bu sayının bire tümleyeni alınıp 1 eklenerek ikiye tümleyeni bulunur:

$$1 = (0001)_2 \rightarrow -1 = \overline{0001} + 1 = (1110)_2 + 1 = (1111)_2$$

Örnek A.13

−8 sayısının 4 bitlik 2'ye tümleyen gösteriminde karşılığı nedir?

Çözüm:

$$8 = (1000)_2 \rightarrow -8 = \overline{1000} + 1 = (0111)_2 + 1 = (1000)_2$$

İkiye tümleyen yönteminde belirli bir bit sayısı ile gösterilebilen eksi sayıların sayısı artı sayıların sayısından bir fazladır. Bu nedenle 4 bitlik alanda −8 gösterilebilse de +8 gösterilemez. 4 bitlik alanda gösterilebilen en yüksek sayı +7'dir.

Örnek A.14

İkiye tümleyen gösteriminde değeri $(1011010)_2$ olan sayının onluk tabandaki karşılığı nedir?

Çözüm:

Sayının işaret biti 1 olduğu için sayı eksidir. Sayının ikiye tümleyeni alınır:

$$\overline{1011010} + 1 = (0100101)_2 + 1 = (0100110)_2 = (38)_{10}$$

Dolayısıyla ikiye tümleyen düzeninde $(1011010)_2 = -38$ 'dir.

Bir sayının ikiye tümleyenini bulmanın daha hızlı bir yolu vardır: sayının en sağdaki bitinden başla, sola doğru yürü, 1 olan ilk basamağa kadar bütün sıfırları ve ilk 1'i kopyala, sonraki bütün basamakları ters çevir.

Örnek A.15

İkiye tümleyen gösteriminde değeri $(1011010)_2$ olan sayının ikiye tümleyenini hızlı yolu kullanarak bulun.

Çözüm:

Sayının en sağdaki basamağından başlanır, sola doğru yürürken ilk 1'e kadar (ilk 1 dahil)

tüm basamaklar kopyalanır, sonraki tüm bitler ters çevrilir:

$$1011010 \xrightarrow{\text{hızlı yol}} 0100110$$

Örnek A.16

$(-15122)_{10}$ sayısını 2'ye tümleyen yöntemini kullanarak 2'lik, 8'lik ve 16'lık düzenlerde yazın.

Çözüm:

Önce sayının mutlak değeri 2'lik tabana çevrilir:

$$15122 = (011101100010010)_2$$

Sayının ikiye tümleyeni alınır:

$$(-15122)_{10} = (100010011101110)_2$$

8'lik tabana çevirmek için 3'erli gruplara ayrılır:

$$\underbrace{100}_4 \underbrace{010}_2 \underbrace{011}_3 \underbrace{101}_5 \underbrace{110}_6 \implies (-15122)_{10} = (42356)_8$$

16'lık tabana çevirmek için 4'erli gruplara ayrılır:

$$\underbrace{1100}_C \underbrace{0100}_4 \underbrace{1110}_E \underbrace{1110}_E \implies (-15122)_{10} = (C4EE)_{16}$$

Eksi sayılarda 4'erli gruplara ayrılırken en soldaki grubun başına 0 değil 1 eklenir.

Örnek A.17

16'lık tabandaki $FC36A$ sayısını 2'lik, 8'lik ve 10'luk düzende gösterin.

Çözüm:

Her bir hane 4 bitlik ikilik tabana çevrilir:

$$\begin{array}{ccccc} F & C & 3 & 6 & A \\ \hline 1111 & 1100 & 0011 & 0110 & 1010 \end{array}$$

$$(FC36A)_{16} = (11111100001101101010)_2$$

Sekizlik tabana çevirmek için 3'erli gruplara ayrılır:

$$(FC36A)_{16} = (3741552)_8$$

İşaretsiz kabul edilirse, en soldaki bit 1 olduğu için eksidir. İkiye tümleyeni alınarak:

$$(FC36A)_{16} = -(15510)_{10} \quad (\text{işaretsiz})$$

İşaretsiz kabul edilirse:

$$(FC36A)_{16} = (1033066)_{10} \quad (\text{işaretsiz})$$

Tablo A.2: Değişik yöntemlerde 4 bitle gösterilen sayılar

İkilik	Ayrı İşaret	1'e Tümleyen	Artık-3	2'ye Tümleyen
0000	+0	+0	-3	0
0001	1	1	-2	1
0010	2	2	-1	2
0011	3	3	0	3
0100	4	4	1	4
0101	5	5	2	5
0110	6	6	3	6
0111	7	7	4	7
1000	-0	-7	5	-8
1001	-1	-6	6	-7
1010	-2	-5	7	-6
1011	-3	-4	8	-5
1100	-4	-3	9	-4
1101	-5	-2	10	-3
1110	-6	-1	11	-2
1111	-7	-0	12	-1

İkiye tümleyen yönteminde gösterilen işaretli sayıların daha yüksek sayıda bitle ifade edilmesi gerektiğinde **İşaretle Genişletme** (İngilizcesi: *Sign Extension*) kullanılır. Sayının işaret bitinin kendisinin soluna eklenmesiyle sayı daha büyük bir alana yazılabilir hale gelir. Örneğin 4 bitlik 1110 sayısı ikiye tümleyen gösteriminde -2'ye karşılık gelir. 8 bite işaretle genişletilirse ortaya çıkan 11111110 sayısının değeri de -2'dir.

A.2.5 Taşma Tespiti

İşaretili sayılarla aritmetik işlem yapılırken sonuç, kullanılan bit sayısının gösterebileceği aralığın dışına çıkabilir. Bu duruma **taşma** (İngilizcesi: *overflow*) denir. İkiye tümleyen gösteriminde n bitlik bir alan -2^{n-1} ile $+2^{n-1} - 1$ arasındaki sayıları gösterebilir; sonuç bu aralığın dışına çıktığında taşma oluşur.

İkiye tümleyende taşma yalnızca aynı işaretli iki sayı toplandığında ortaya çıkabilir:

- İki pozitif sayı toplanıp sonucun işaret biti 1 olursa (sonuç negatif görünür) $\Rightarrow$ taşma var.
- İki negatif sayı toplanıp sonucun işaret biti 0 olursa (sonuç pozitif görünür) $\Rightarrow$ taşma var.
- Farklı işaretli iki sayının toplamında taşma oluşmaz; sonucun büyüklüğü her zaman işlenenlerin büyüklüğünden küçüktür.

Taşma donanım düzeyinde en anlamlı bit konumundaki elde girişi (c_{n-1}) ile elde çıkışının (c_n) karşılaştırılmasıyla tespit edilir: $c_{n-1} \neq c_n$ ise taşma vardır.

Örnek A.18

4 bitlik ikiye tümleyen düzeninde $5 + 4$ işlemini yapın ve taşma olup olmadığını belirleyin.

Çözüm:

$$\begin{array}{r}
 0101 \quad (+5) \\
 + 0100 \quad (+4) \\
 \hline
 1001 \quad (-7?)
 \end{array}$$

Sonuç 1001'dir ve işaret biti 1'dir; ikiye tümleyen düzeninde bu -7'ye karşılık gelir. İki pozitif sayı toplanmasına karşın sonuç negatif çıktığı için taşma oluşmuştur. Gerçek sonuç olan +9, 4 bitlik alanda (-8 ile +7 arası) gösterilemez.

Örnek A.19

4 bitlik ikiye tümleyen düzeninde $(-6) + (-5)$ işlemini yapın ve taşma olup olmadığını belirleyin.

Çözüm:

$$-6 = (1010)_2, -5 = (1011)_2:$$

$$\begin{array}{r}
 1010 \quad (-6) \\
 + 1011 \quad (-5) \\
 \hline
 1\ 0101 \quad (+5?)
 \end{array}$$

Elde çıkışı atıldığında sonuç 0101, başka bir deyişle +5 olarak görünür. İki negatif sayının toplamının pozitif çıkması taşma olduğunu gösterir. Gerçek sonuç olan -11, 4 bitlik alanda gösterilemez.

Örnek A.20

4 bitlik ikiye tümleyen düzeninde $3 + (-5)$ işlemini yapın ve taşma olup olmadığını belirleyin.

Çözüm:

$$3 = (0011)_2, -5 = (1011)_2:$$

$$\begin{array}{r}
 0011 \quad (+3) \\
 + 1011 \quad (-5) \\
 \hline
 1110 \quad (-2)
 \end{array}$$

Sonuç 1110, ikiye tümleyen düzeninde -2'dir. Beklenen sonuç da $3 + (-5) = -2$ olduğundan taşma oluşmamıştır. Farklı işaretli iki sayının toplamında taşma asla oluşmaz; çünkü sonucun büyüklüğü her zaman işlenenlerden küçüktür.

İşaretsiz sayılarla çalışılırken ise taşma kavramı yerine **elde** (İngilizcesi: *carry*) kavramı kullanılır. En anlamlı bittten bir elde çıkışı oluşursa sonuç n bite sığmamış demektir; bu durum elde bayrağı (carry flag) ile işaretlenir. İşaretli ve işaretsiz taşmanın farklı bayraklarla izlendiğine dikkat ediniz: işlemciler genellikle hem elde (C) hem de taşma (V) bayraklarını ayrı ayrı günceller.

Bilgi Notu A.1: Ariane 5: Taşmanın Bedeli

4 Haziran 1996'da Avrupa Uzay Ajansı'nın Ariane 5 roketi, fırlatılışından 37 saniye sonra yörüngeden saptı ve infilak etti. Kaza raporunda sorunun, uçuş kontrol yazılımında

64 bitlik bir kayan nokta değerinin 16 bitlik işaretli bir tam sayıya dönüştürülmesi sırasında oluşan bir taşmadan kaynaklandığı tespit edildi. Roketin yanal hız değeri 16 bitin gösterebileceği $\pm 32\,767$ aralığını aştığında yazılım kural dışı durumu tetikledi ve yedek sistem de aynı hataya düştü. Yaklaşık 370 milyon dolarlık kayıpla sonuçlanan bu olay, taşma denetiminin ne denli önemli olduğunu gösteren en çarpıcı örneklerden biridir.

A.3 Gray Kodu

Gray Kodu, Bell Laboratuvarları'nda çalışan Frank Gray tarafından bulunmuştur. Sayıların kodlanması sırasında birbiri arasında değer açısından 1 fark olan iki sayının yalnızca 1 bitinin farklı olması esasına dayanır. Bu özellik, birden fazla bitin aynı anda değiştiği geçiş anlarında oluşabilecek hatalı ara değerleri ortadan kaldırır. Gray kodu özellikle döner konum kodlayıcılarda (rotary encoder) ve Karnaugh haritalarında (bkz. Ek B) kullanılır. Tablo A.3'da 3 bitlik ikilik ve Gray kodları yan yana gösterilmiştir; ardışık satırlar arasında yalnızca tek bir bitin değiştiğine dikkat ediniz.

Tablo A.3: 3 Bitlik Gray Kodu

Onluk Taban	İkilik Taban	Gray Kodu
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

İkilik tabandaki bir sayıyı Gray koduna çevirmek için basit bir kural vardır: en soldaki (en anlamlı) bit aynen korunur; sonraki her Gray biti, ikilik sayının o basamağı ile bir soldaki basamağının Dışlayan VEYA (XOR) işlemiyle elde edilir. Ters yönde ise Gray kodundan ikilik tabana çevirirken, en soldaki bit yine aynen korunur ve sonraki her ikilik bit, bir önceki ikilik bit ile o konumdaki Gray bitinin XOR'u olarak hesaplanır.

Örnek A.21

$(13)_{10}$ sayısını Gray koduna çevirin.

Çözüm:

Önce ikilik tabana çevirelim: $13 = (1101)_2$. Şimdi XOR kuralını uygulayalım:

$$\begin{array}{cccc}
 \text{İkilik:} & 1 & 1 & 0 & 1 \\
 & \downarrow & \downarrow \text{XOR} & \downarrow \text{XOR} & \downarrow \text{XOR} \\
 \text{Gray:} & 1 & 0 & 1 & 1
 \end{array}$$

İlk bit aynen 1; ikinci Gray biti $1 \oplus 1 = 0$; üçüncü $1 \oplus 0 = 1$; dördüncü $0 \oplus 1 = 1$. Sonuç: $(13)_{10} = (1011)_{\text{Gray}}$.

A.4 İkiye Kodlanmış Onluk Taban Gösterimi

Onluk tabandaki her bir rakamın ikilik tabanda kodlandığı bu gösterim biçimine **İkiye Kodlanmış Onlu** (İngilizcesi: *Binary Coded Decimal – BCD*) gösterimi denir. Onluk tabanda toplam 10 rakam vardır ve bu 10 rakam ikilik tabanda 4 bitle gösterilebilir. BCD gösteriminin özelliği 9'dan büyük bir sayı yazılacağı zaman her bir sayı basamağı için 4 bitlik rakam kodunun ayrıca kullanılmasıdır. Tablo A.4'da 0–9 arası onluk rakamların BCD karşılıkları verilmiştir.

Tablo A.4: 10'luk tabandaki sayıların BCD yöntemi ile gösterimleri

Onluk Taban	BCD Gösterimi
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	0001 0000
11	0001 0001
154	0001 0101 0100
1128	0001 0001 0010 1000

BCD gösteriminde aritmetik işlemler, olağan ikilik toplama kurallarından farklıdır. Her 4 bitlik grubun sonucu 9'u (1001'i) aşarsa, o gruba 6 (0110) eklenerek düzeltme yapılır; bu düzeltme bir elde oluşturur ve bir üst basamağa aktarılır.

Örnek A.22

BCD düzeninde $27 + 35$ işlemini yapın.

Çözüm:

Her basamağı ayrı ayrı 4 bitlik BCD olarak yazalım ve toplayalım:

$$\begin{array}{r}
 0010\ 0111\ (27)_{\text{BCD}} \\
 +\ 0011\ 0101\ (35)_{\text{BCD}} \\
 \hline
 0101\ 1100
 \end{array}$$

Alt 4 bit: $0111 + 0101 = 1100$. Bu değer 9'dan büyük olduğu için 0110 eklenir: $1100 + 0110 = 1\ 0010$. Alt basamak 0010 olur, bir elde yukarı aktarılır. Üst 4 bit: $0010 + 0011 + 1_{\text{elde}} = 0110$. Sonuç: $0110\ 0010 = (62)_{\text{BCD}}$.

A.5 Karakter Kodlaması

Bilgisayarlar sayıların yanında karakterlerle ve bu karakterlerin oluşturduğu dizgilerle de işlem yaparlar. Her ne kadar işlem yapılan veri türü bir sayı olmasa da, bilgisayarların yalnızca ikilik tabandaki 1 bitlik değerleri saklamaları nedeniyle, karakterler de ikilik tabanda sayı olarak saklanır.

A.5.1 ASCII

Ekrana basılacak harflerin ve her türlü noktalama işaretlerinin ikilik tabanda bir karşılığı olmalıdır. Dünyada en yaygın kullanılan karakter tablosu ASCII'dir (American Standard Code for Information Interchange). ASCII düzeninde her bir karakter için 7 bitlik bir kod tanımlanmıştır; bu da toplam $2^7 = 128$ farklı karakter anlamına gelir. Bu 128 kodun ilk 32'si (0–31) ve son kodu (127) denetim karakterleridir; geri kalan 95'i yazdırılabilir karakterlerdir. ASCII bellekte genellikle 1 baytlık bir alanda saklanır; kullanılmayan 8. bit sıfır yapılıp ya da eşlik denetimi için kullanılır.

İlk ortaya çıkan 128 karakterlik ASCII tablosunda Türkçe'ye özgü harfler (ç, ğ, ı, ö, ş, ü) yoktur. 1980'li yıllarda bu sorunu çözmek için 8. bitin de kullanıldığı "Genişletilmiş ASCII" tabloları (128–255 arası kodlar) tanımlandı; ancak her ülke kendi tablosunu oluşturduğu için uyumsuzluklar ortaya çıktı. Türkçe için ISO 8859-9 (Latin-5) tablosu kullanılmıştır.

Şekil A.1'de yazdırılabilir 96 ASCII karakteri (0x20–0x7E aralığı) gösterilmiştir. Tabloda satır başlığı üst yarım baytı, sütun başlığı alt yarım baytı verir; bir karakterin kodunu bulmak için satır ve sütun değerleri birleştirilir. Örneğin "A" harfi 4x satırı ile 1 sütununda yer aldığı için kodu 0x41'dir; onluk tabanda 65'e karşılık gelir. Tablodaki ilk karakter olan SP (*space*) boşluk karakterini, son karakter olan DEL ise silme denetim karakterini temsil eder.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2x	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Şekil A.1: ASCII Karakter Tablosu (yazdırılabilir 96 karakter)

A.5.2 Unicode ve UTF-8

Bilgi Notu A.2: Unicode'un Doğuşu

1980'lerin sonunda Apple'dan Joe Becker ve Xerox'tan Lee Collins ile Mark Davis, her karakter için 16 bit kullanan evrensel bir kodlama sistemi fikrini geliştirdi. 1991 yılında Unicode Konsorsiyumu resmen kuruldu ve ilk standart (Unicode 1.0) yayımlandı. Başlangıçta 16 bitin (65 536 karakter) yeterli olacağı düşünülse de, dünya dillerinin zenginliği karşısında standart genişletilerek bugün 1 milyondan fazla kod noktasını destekleyen 21 bitlik bir uzaya ulaştı.

Genişletilmiş ASCII tablolarının ülkeler arası uyumsuzluk sorunu, 1991 yılında Unicode

standartı ile kökten çözülmüştür. Unicode, dünyadaki tüm yazı sistemlerini kapsayan evrensel bir karakter kümesidir ve günümüzde 150 000’den fazla karakter tanımlamaktadır. Her karaktere “kod noktası” (İngilizcesi: *code point*) adı verilen benzersiz bir numara atanır; örneğin büyük A harfi U+0041, Türkçe ğ harfi U+011F koduna sahiptir.

Unicode karakterlerini bellekte saklamak için farklı kodlama biçimleri kullanılır. Bunların en yaygını **UTF-8**'dir. UTF-8 değişken uzunluklu bir kodlamadır: her karakter 1 ile 4 bayt arasında yer kaplar. ASCII ile tam uyumludur: ilk 128 karakter (U+0000 ile U+007F arası) aynen tek bir baytla gösterilir. Tablo A.5'da her bayt sayısı için kod noktası aralığı ve bayt deseni gösterilmiştir; baştaki sabit bitler (110, 1110, vb.) bayt sayısını, 10 öneki ise devam baytını belirtir.

Tablo A.5: *UTF-8 kodlama kuralları*

Bayt sayısı	Kod noktası aralığı	Bayt deseni
1	U+0000 – U+007F	0xxxxxxx
2	U+0080 – U+07FF	110xxxxx 10xxxxxx
3	U+0800 – U+FFFF	1110xxxx 10xxxxxx 10xxxxxx
4	U+10000 – U+10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

İngilizce metin UTF-8’de karakter başına 1 bayt harcarken, Türkçe’ye özgü ç, ğ, ı, ö, ş, ü harfleri 2 bayt yer kaplar. Çince ve Japonca gibi diller ise karakter başına 3 bayt gerektirir. Bu değişken uzunluk, depolama ve iletim açısından verimlilik sağlarken, dizgi içinde *i.* karaktere doğrudan erişimi zorlaştırır.

Örnek A.23

“İş” kelimesinin UTF-8 kodlamasını onaltılık tabanda yazın.

Çözüm:

İlk karakter “İ” harfi, Unicode kod noktası U+0130’dur. $0130_{16} = 0001\ 0011\ 0000_2$ olup $007F$ ’den büyük ve $07FF$ ’den küçüktür; bu nedenle 2 baytla kodlanır. 11 bitlik değer 00100110000 ikiye ayrılarak 110 ve 10 örneklerine yerleştirilir:

$$110\ 00100\ 10\ 110000 = C4\ B0_{16}$$

İkinci karakter “ş” harfi, Unicode kod noktası U+015F’dir. $015F_{16} = 0001\ 0101\ 1111_2$:

$$110\,00101 \quad 10\,011111 = C5\,9F_{16}$$

Sonuç: “İş” → C4 B0 C5 9F (4 bayt). Karşılaştırma olarak, ASCII’de gösterilebilen “is” kelimesi yalnızca 69 73 (2 bayt) yer kaplar.

A.6 IEEE 754 Kayan Nokta Gösterimi

Tam sayılar birçok hesaplama için yeterli olsa da bilimsel, mühendislik ve grafik uygulamalarında kesirli sayılarla çalışmak kaçınılmazdır. Kesirli sayıları göstermenin en basit yolu **sabit nokta** (İngilizcesi: *fixed point*) gösterimidir: bit dizisinde virgölün yeri önceden sabitlenerek belirli sayıda bit tam kısma, geri kalanı kesir kısmına ayrılır. Ancak sabit nokta gösteriminin

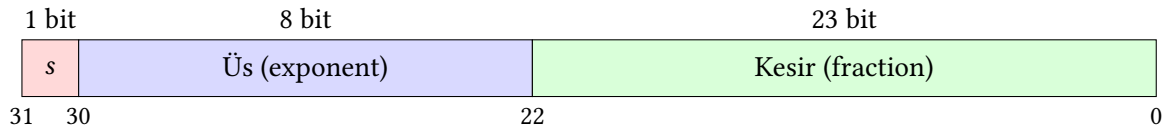
değer aralığı oldukça dardır; bu sınırlamayı aşmak için **kayan nokta** (İngilizcesi: *floating point*) gösterimi geliştirilmiştir. Bu gösterimde bir sayı, işaret, üs ve kesir olmak üzere üç bileşene ayrılır; böylece hem çok büyük hem de çok küçük değerler sınırlı bit sayısı ile ifade edilebilir. Günümüzde neredeyse tüm işlemciler IEEE 754 standardını kullanmaktadır.

Bilgi Notu A.3: IEEE 754 ve William Kahan

1985 yılında yayımlanan IEEE 754 standardı, kayan nokta aritmetiğinde donanımlar arası tutarlılığı sağlamış ve sayısal hesaplamaların güvenilirliğini köklü biçimde artırmıştır. Standardın mimarı, Kaliforniya Üniversitesi Berkeley'den Prof. William Kahan'dır; bu çalışması kendisine 1989 Turing Ödülü'nü kazandırmıştır. Kayan nokta aritmetiği Bölüm 5'te ayrıntılı olarak işlenmektedir; bu bölümde yalnızca gösterim biçimi ele alınmaktadır.

A.6.1 Tek Duyarlı Gösterim (32 bit)

IEEE 754 tek duyarlı (İngilizcesi: *single precision*) gösterimde 32 bit kullanılır. Şekil A.2'te gösterildiği gibi bu 32 bit üç alana bölünür: en soldaki bit işaret bilgisini, sonraki 8 bit üs değerini, kalan 23 bit ise kesir kısmını taşır.



Şekil A.2: IEEE 754 tek duyarlı kayan nokta biçimi

Sayının değeri şu formülle hesaplanır:

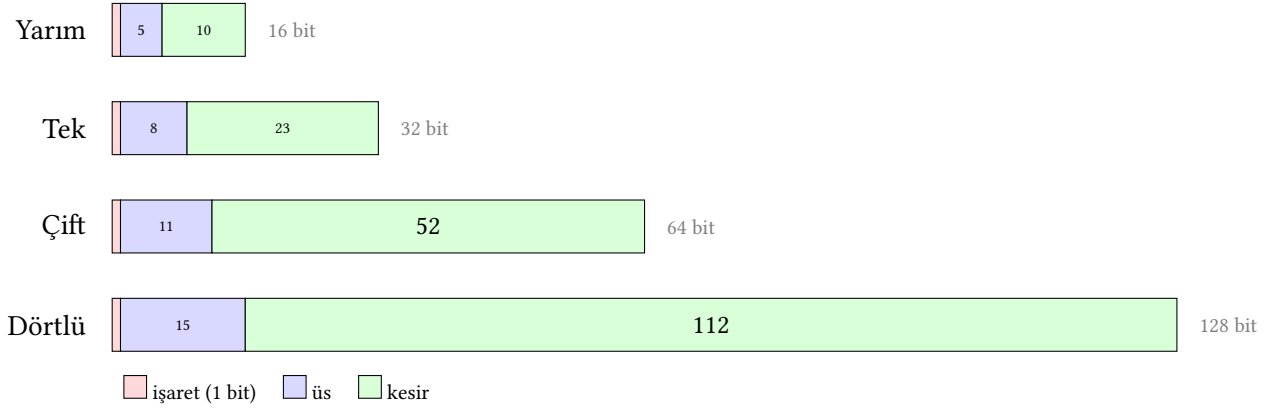
$$\text{değer} = (-1)^s \times 1, f \times 2^{(e-127)} \quad (\text{A.2})$$

Burada s işaret biti, e üs alanının işaretlessiz tam sayı değeri ve f kesir alanıdır. Üs alanında 127 **kaydırması** (İngilizcesi: *bias*) kullanılır: saklanan e değerinden 127 çıkarılarak gerçek üs elde edilir. $e = 127$ ise gerçek üs 0, $e = 130$ ise gerçek üs +3, $e = 124$ ise gerçek üs -3 olur.

Kesir alanının solunda örtük bir "1," bulunur (normalleştirilmiş sayılarda); bu sayede 23 bitlik kesir alanı aslında 24 bitlik bir doğruluk sağlar.

A.6.2 IEEE 754 Duyarlık Düzeyleri

IEEE 754 standardı farklı duyarlık düzeyleri tanımlar. Dört temel biçim Şekil A.3'de karşılaştırmalı olarak gösterilmiştir; tüm biçimlerde yapı aynıdır (işaret + üs + kesir), yalnızca alan genişlikleri farklıdır.



Şekil A.3: IEEE 754 kayan nokta biçimlerinin karşılaştırması

Yarım duyarlı (16 bit): 2008 yılında IEEE 754 standardına eklenen bu biçim (binary16), derin öğrenme ve grafik uygulamalarında bellek ve bant genişliğinden tasarruf etmek için kullanılır. Yalnızca ~3 ondalık basamak doğruluk sağladığı için genel amaçlı hesaplamalar için uygun değildir.

Çift duyarlı (64 bit): Çoğu programlama dilinde varsayılan kayan nokta türüdür. ~16 basamak doğruluk sağlar ve bilimsel hesaplamaların büyük çoğunluğu için yeterlidir.

Dörtlü duyarlı (128 bit): ~34 basamak doğruluk sağlar; astronomik hesaplamalar ve çok hassas sayısal çözücüler gibi özel alanlarda kullanılır. Çoğu işlemci donanımında desteklemez; yazılımla gerçekleştirilir.

Dört duyarlılık düzeyinin alan genişlikleri, üs aralıkları ve gösterim kapasiteleri Tablo A.6'da özetlenmiştir.

Tablo A.6: IEEE 754 duyarlılık düzeylerinin karşılaştırması

Özellik	Yarım	Tek	Çift	Dörtlü
Toplam bit sayısı	16	32	64	128
İşaret biti	1	1	1	1
Üs alanı	5 bit	8 bit	11 bit	15 bit
Kesir alanı	10 bit	23 bit	52 bit	112 bit
Üs kaydırması	15	127	1023	16383
Üs aralığı	-14 - +15	-126 - +127	-1022 - +1023	-16382 - +16383
Ondalık doğruluk	~3	~7	~16	~34
En küçük normalleşt.	$\sim 6,1 \times 10^{-5}$	$\sim 1,2 \times 10^{-38}$	$\sim 2,2 \times 10^{-308}$	$\sim 3,4 \times 10^{-4932}$
En büyük sonlu	$\sim 6,6 \times 10^4$	$\sim 3,4 \times 10^{38}$	$\sim 1,8 \times 10^{308}$	$\sim 1,2 \times 10^{4932}$

A.6.3 Özel Değerler

IEEE 754 standardı, olağan sayıların yanı sıra birkaç özel durumu da tanımlar. Tablo A.7'de üs ve kesir alanlarının özel kombinasyonlarına karşılık gelen değerler listelenmiştir.

Tablo A.7: IEEE 754 özel değerler (tek duyarlı)

Üs (e)	Kesir (f)	Gösterilen değer
0	0	± 0 (işaret bitine bağlı)
0	$\neq 0$	Denormalize sayı: $(-1)^s \times 0, f \times 2^{-126}$
1–254	herhangi	Normalleştirilmiş sayı: $(-1)^s \times 1, f \times 2^{(e-127)}$
255	0	$\pm \infty$ (sonsuz)
255	$\neq 0$	NaN (sayı değil – <i>Not a Number</i>)

Sonsuz (∞) değeri, sıfıra bölme gibi işlemlerin sonucu olarak ortaya çıkar. NaN ise $0/0$ veya $\sqrt{-1}$ gibi tanımsız işlemlerin sonucudur. Bu özel değerler sayesinde program çökmek yerine hesaplamaya devam edebilir ve hata daha sonra tespit edilebilir.

Örnek A.24

–6,625 sayısını IEEE 754 tek duyarlı biçimde gösterin.

Çözüm:

Adım 1 (İşaret biti): Sayı negatif olduğu için $s = 1$.

Adım 2 (İkilik tabana çevirme): $6 = (110)_2$; kesir kısmı: $0,625 = 0,5 + 0,125 = 2^{-1} + 2^{-3} = (0,101)_2$. Dolayısıyla $6,625 = (110,101)_2$.

Adım 3 (Normalleştirme): Noktayı sola kaydırarak $1,10101 \times 2^2$ elde edilir.

Adım 4 (Üs hesabı): Gerçek üs = 2; saklanan üs = $2 + 127 = 129 = (10000001)_2$.

Adım 5 (Kesir alanı): Örtük 1 atılır, kalan 10101 sağdan sıfırlarla 23 bite tamamlanır: 10101000000000000000000.

Sonuç:

$$\underbrace{1}_{s} \underbrace{10000001}_{e=129} \underbrace{101010000000000000000000}_{f} = C0D40000_{16}$$

Örnek A.25

IEEE 754 tek duyarlı biçimde saklanan $41C80000_{16}$ değerinin onluk tabandaki karşılığını bulun.

Çözüm:

Onaltılık değeri ikilik tabana açalım:

$$41C80000_{16} = 0 \ 10000011 \ 100100000000000000000000$$

İşaret: $s = 0$ (pozitif). Üs: $e = (10000011)_2 = 131$; gerçek üs = $131 - 127 = 4$. Kesir: $f = 10010 \dots$; örtük 1 ile: 1,1001. Değer: $1,1001 \times 2^4 = 11001,0 = 16 + 8 + 1 = 25,0$. Sonuç: +25,0.

Örnek A.26

IEEE 754 tek duyarlı biçimde 0,1 sayısını gösterin. Geri dönüşümde tam olarak 0,1 elde edilir mi?

Çözüm:

0,1 sayısının ikilik tabandaki açılımı sonsuz yinelenen bir kesirdir:

$$0,1_{10} = 0,0001\overline{1} \overline{00011} \dots_2$$

Normalleştirme: $1,1001 \dots \times 2^{-4}$. İşaret: $s = 0$. Üs: $-4 + 127 = 123 = (01111011)_2$. Kesir alanına 23 bit sığdırılır ve yuvarlanır. Sonuç:

$$0 \ 01111011 \ 10011001100110011001101 = 3DCCCCD_{16}$$

Geri dönüşümde elde edilen değer tam olarak 0,1 değildir; yaklaşık 0,100000001490 ... olur. Bu fark, kayan nokta aritmetiğinde **yuvarlama hatası** olarak bilinir ve bilgisayar aritmetiğinin temel sınırlamalarından biridir.

A.7 Yanılgılar ve Tuzaklar

Sayı sistemleri ve kodlama yöntemleri ilk bakışta mekanik kurallar bütünü gibi görünse de uygulamada sürpriz hatalar doğuran pek çok yanılgı ve tuzak barındırır. Sezginin yanıltıcı olduğu bu noktalarda küçük bir kavram kargaşası, yazılım hatalarına ya da donanım tasarımında beklenmeyen davranışlara yol açabilir.

Yanılgı: İkiye tümleyende en yüksek değerli bit yalnızca işaret bitidir ve farklı işlenir

n -bitlik ikiye tümleyen gösteriminde en yüksek değerli bit, -2^{n-1} ağırlığına sahip sıradan bir bit konumudur; geri kalan bitler ise $+2^{n-2}, +2^{n-3}, \dots, +2^0$ ağırlıklarıyla katkıda bulunur. Toplama ve çıkarma işlemleri, bu bit için özel bir kural gerektirmeksizin modüler aritmetikle gerçekleştirilir. Örneğin 4-bitlik gösterimde 1101_2 değeri -3 'tür: $-8 + 4 + 0 + 1 = -3$. Donanım toplayıcısı bu sayıyı diğerlerinden farklı biçimde işlemez; tüm bitler aynı devre kurallarına tabidir. "İşaret biti ayrı hesaplanır" yanılgısı, konunun tersine tümleyen ve işaret-büyüklik gösterimleriyle birlikte öğrenilmesinden kaynaklanır; oysa ikiye tümleyene geçilince bu ayırım ortadan kalkar.

Tuzak: İşaretsiz ve işaretli karşılaştırmayı karıştırmak

C ve C++ dillerinde `int` ile `unsigned int` türleri aynı bit kalıbını farklı yorumlar. İki değer karşılaştırıldığında derleyici, kurallar gereği `int` tarafını `unsigned`'a dönüştürür. Bu durumda -1 değeri, 32-bitlik gösterimde $2^{32} - 1 = 4\,294\,967\,295$ olarak yorumlanır ve karşılaştırma " $-1 > 1$ " ifadesi beklenmedik biçimde doğru sonuç verir. Döngülerde `i >= 0` koşuluyla kullanılan işaretsiz sayaç hiç sonlanmaz; çünkü işaretsiz tamsayılar her zaman sıfırdan büyük ya da eşittir. Bu tuzağa düşmemek için karşılaştırma operandlarının türlerini açıkça eşleştirmek gerekir.

Yanılgı: ASCII Türkçe karakterleri kapsar

ASCII, 7 bit ile yalnızca 128 karakter tanımlar; bunlar temel Latin harfleri, rakamlar ve noktalama işaretleridir. Ç, ğ, İ, ı, Ö, ö, Ş, ş, Ü, ü gibi Türkçe karakterler bu kümede yer almaz. Bu karakterleri aktarmak için ISO 8859-9 (Latin-5) ya da UTF-8 kodlaması gereklidir. "Bilgisayar

zaten ASCII kullanıyorsa Türkçe de çalışır” yanılgısı, tarihsel olarak Türkçe metin dosyalarında veri bozulmasına ve platformlar arası uyumsuzluğa yol açmıştır. UTF-8, ASCII ile tam geriye dönük uyumlu olduğundan ve tüm Unicode karakterlerini desteklediğinden çağdaş sistemlerde tercih edilen standarttır.

Tuzak: Kayan nokta sayılarını == ile karşılaştırmak

İkili gösterim, 0,1 gibi pek çok onluk kesri tam olarak ifade edemez; bu nedenle $0,1 + 0,2$ işleminin sonucu IEEE 754 tek duyarlıda tam olarak 0,3 değil 0,30000001192 ... olur. Eşitlik karşılaştırması iki kayan nokta sayısının bit bit örtüşmesini arar; art arda yapılan hesaplar kümülâtif yuvarlama hatası biriktirdiğinden aynı matematiksel değer farklı bit kalıplarına dönüşebilir. Doğru yaklaşım $|a - b| < \epsilon$ biçiminde bir hata payı denetimidir; burada ϵ değeri uygulamaya özgü bir tolerans sınırıdır. Özellikle döngü sayacı ve durdurma koşulu olarak kayan nokta karşılaştırması kullanmak ciddi hataya kapı aralar.

Yanılğı: Kayan nokta aritmetiği sınırsız hassasiyet sunar

IEEE 754 tek duyarlı biçimde kesir alanı yalnızca 23 bit genişliğindedir; bu, yaklaşık 7 onluk basamak duyarlılığı karşılık gelir. Çift duyarlıda bu değer 15–16 basamağa çıkar, ancak yine de sonsuza değil. Bilimsel hesaplamalarda art arda gerçekleştirilen toplama ve çıkarma işlemleri bağlı hata büyüklüğünü artırabilir (felaket iptali); döngüsel toplama yöntemi (Kahan toplamı) bu hatayı hafifletir. Yapay zeka uygulamalarında kullanılan yarım duyarlı (16 bit) biçim ise yalnızca 3 onluk basamak hassasiyet sağlar. “Kayan nokta kesindir, tamsayı gibi güvenilir” anlayışı, finans ve ölçüm yazılımlarında ciddi hatalara neden olmuştur.

Tuzak: BCD ile ikilik gösterimi karıştırmak

BCD’de her onluk basamak bağımsız olarak 4 bit ile kodlanır: 29_{10} değeri BCD’de 0010 1001 olurken ikilik gösterimde 00011101 olur. Bu iki gösterim görünüşte benzer bit dizileri üretebilir, ancak değerleri farklıdır. BCD toplamada her 4-bitlik grup için kural dışı durum denetimi (sonuç 9’u aşıyorsa 6 eklenir) gereklidir; standart ikilik toplayıcı bu düzeltmeyi otomatik yapmaz. Donanım ve yazılım arasında veri aktarımında kodlama biçiminin belgelenmemesi, BCD ile ikilik verinin birbirinin yerine yorumlanmasına ve yanlış sayı gösterimine yol açar.

Tuzak: Negatif ikilik sayı çevrilirken mutlak değer adımını atlamak

Bir onluk negatif sayıyı ikiye tümleyen gösterimine dönüştürmenin klasik yöntemi şudur: (1) sayının mutlak değerini ikilik tabanda yaz, (2) tüm bitleri tersle, (3) sonuca 1 ekle. Yaygın hata, mutlak değer alınmadan doğrudan negatif sayının basamakları üzerinden işlem yapmaktır. Örneğin -5 ’i 4 bitte göstermek için önce $5 = 0101_2$, sonra tersle 1010_2 , son olarak $1 + 1010_2 = 1011_2$ adımları izlenir. “Eksi işaretini kor, kalanı çevir” yaklaşımı hatalı sonuç verir ve doğrulama yapılmadan donanım tasarımına taşındığında zorlu hata ayıklama süreçlerine kapı aralar.

Yanılğı: Onaltılık gösterim yalnızca bir yazım biçimidir, 16 tabanıyla ilgisi yoktur

Onaltılık (hex) gösterim, 16 tabanlı bir sayı sistemidir: her basamak 0’dan F’ye kadar 16 farklı değer alır ve n . konumdaki basamak 16^n ile ağırlıklandırılır. “Hex yalnızca ikilik bitlerin okuna-

bilir biçimde gösterilmesidir” düşüncesi bu sistemi indirger. Onaltılık basamaklar ikilik gruplarla doğrudan eşleştikten (her hex basamağı 4 bit) dönüşüm kolaydır; ancak bu kolaylık, hex’in bağımsız bir sayı sistemi olduğu gerçeğini değiştirmez. Aritmetik işlemler hex basamaklarına doğrudan uygulanabilir: $A_{16} + B_{16} = 15_{16}$ hesabı ikilik tabana başvurmadan yapılabilir.

Özet

Bu ekte bilgisayar mimarisinin temelini oluşturan sayı sistemleri ve kodlama yöntemleri ele alındı. Temel kavramlar şöyle özetlenebilir:

- İkilik taban, bilgisayarların doğal dilidir. Taban dönüşümleri, her basamağın taban kuvvetleriyle çarpılıp toplanması ilkesine dayanır (Denklemler A.1).
- Sekizlik ve onaltılık tabanlar, uzun ikilik dizileri sırasıyla 3’erli ve 4’erli gruplara ayırarak okunabilirliği artırır.
- İşaretsiz sayılar arasında *ikiye tümleyen* yöntemi, tek sıfır değeri ve basit donanım gerçekleştirimi sayesinde günümüz işlemcilerinde standart olmuştur. Taşma, aynı işaretli iki sayının toplamında sonuç aralık dışına çıktığında oluşur; farklı işaretli sayıların toplamında oluşmaz.
- Gray kodu, ardışık sayılar arasında yalnızca tek bitin değişmesini garanti ederek geçiş hatalarını önler. BCD gösterimi ise her onluk basamağı ayrı 4 bitle kodlar.
- ASCII, 7 bitlik kodlamayla 128 karakter tanımlar. Unicode ve UTF-8 ise değişken uzunluklu kodlamayla dünyadaki tüm yazı sistemlerini kapsar.
- IEEE 754 kayan nokta standardı; işaret, üs ve kesir alanlarıyla kesirli sayıları gösterir. Tek duyarlı (32 bit) ve çift duyarlı (64 bit) biçimler en yaygın kullanılanlardır. Kayan nokta aritmetiğinin ayrıntıları Bölüm 5’te ele alınmaktadır.

Alıştırmalar

Alıştırma A.1. ★ $(54)_{10}$ sayısını ikilik, sekizlik ve onaltılık tabanda yazın.

Alıştırma A.2. ★ İkilik tabandaki $(11001110)_2$ sayısını onluk, sekizlik ve onaltılık tabanda yazın.

Alıştırma A.3. ★ Aşağıdaki 8 bitlik *ikiye tümleyen* sayıların onluk tabandaki karşılıklarını bulun: (a) 01100100, (b) 10110011, (c) 11111111, (d) 10000000.

Alıştırma A.4. ★ ASCII tablosunu kullanarak “RISC-V” dizisinin her bir karakterinin onaltılık kodunu yazın.

Alıştırma A.5. ★ $(-23)_{10}$ sayısının 8 bitlik *ikiye tümleyen* gösterimini bulun. Sonucu onaltılık tabanda da ifade edin.

Alıştırma A.6. ★ 3 bitlik Gray kodunu kullanarak $(5)_{10}$ sayısının Gray kodu karşılığını bulun.

Alıştırma A.7. ★★ 8 bitlik *ikiye tümleyen* düzeninde aşağıdaki toplama işlemlerini yapın ve taşma olup olmadığını belirleyin: (a) $64 + 53$, (b) $(-80) + (-60)$, (c) $100 + (-75)$.

Alıştırma A.8. ★★ 4 bitlik bir ikiye tümleyen sisteminde gösterilebilecek en küçük ve en büyük tam sayıları belirleyin. 16 bit için aynı hesabı yineleyin.

Alıştırma A.9. ★★ 12,375 sayısını IEEE 754 tek duyarlı kayan nokta biçiminde gösterin. Sonucu onaltılık tabanda da ifade edin.

Alıştırma A.10. ★★ IEEE 754 tek duyarlı biçimde saklanan $C1480000_{16}$ değerinin onluk tabandaki karşılığını bulun.

Alıştırma A.11. ★★ BCD düzeninde $49 + 38$ işlemini yapın. 4 bitlik her grubun sonucunu ayrı ayrı gösterin ve gerekli düzeltme adımlarını açıklayın.

Alıştırma A.12. ★★ “Öğüz” kelimesinin UTF-8 kodlamasını onaltılık tabanda yazın. Her karakter için kaç bayt kullanıldığını belirtin. (İpucu: Ö = U+00D6, ğ = U+011F, ü = U+00FC, z = U+007A)

Alıştırma A.13. ★★ 8 bitlik 01011010 sayısını aşağıdaki her yöntemle yorumlayın ve onluk tabandaki karşılığını verin: (a) işaretsiz, (b) ayrı işaret biti, (c) bire tümleyen, (d) ikiye tümleyen.

Alıştırma A.14. ★★ $(-19)_{10}$ sayısını 8 bitlik ikiye tümleyen biçiminde bulun; ardından 16 bite işaretle genişletme uygulayın. Her iki gösterimin de onluk tabandaki değerinin aynı olduğunu doğrulayın.

Alıştırma A.15. ★★★ IEEE 754 tek duyarlı biçimde $+\infty$, -0 ve NaN değerlerinin ikilik tabandaki gösterimlerini yazın.

Alıştırma A.16. ★★★ IEEE 754 tek duyarlı biçimde gösterilebilecek, sıfırdan büyük en küçük normalleştirilmiş sayıyı ve en küçük denormalize sayıyı bulun. Bu iki değer arasındaki farkı açıklayın.

Alıştırma A.17. ★★★ Birçok programlama dilinde $0.1 + 0.2 \neq 0.3$ ifadesi doğru (true) sonuç verir. IEEE 754 tek duyarlı aritmetiğini kullanarak bu durumu açıklayın. 0,1, 0,2 ve 0,3 sayılarının kayan nokta gösterimlerini karşılaştırın.

Alıştırma A.18. ★★★ n bitlik ikiye tümleyen düzeninde -2^{n-1} sayısının ikiye tümleyenini almaya çalışın. Sonuç nedir ve bu neden bir sorun oluşturur?

Sayısal Tasarım Temelleri



Galata Kulesi – İstanbul

Galata Kulesi, taş taş üstüne koyularak yükselmiştir. Sayısal devreler de kapı kapı üstüne koyularak inşa edilir: VE, VEYA ve DEĞİL kapılarından toplayıcılara, yazmaçlara ve sonunda bütün bir işlemciye.

Öğrenme Hedefleri

Bu eki tamamladığınızda aşağıdakileri yapabileceksiniz:

- Temel mantık kapılarının (VE, VEYA, DEĞİL) ve türetilmiş kapıların (VE-DEĞİL, VEYA-DEĞİL, Dışlayan VEYA, Dışlayan VEYA DEĞİL) doğruluk tablolarını yazmak.
- Boole cebri kurallarını ve De Morgan kuramlarını kullanarak mantık ifadelerini sadeleştirmek.
- Karno haritası yöntemiyle 2, 3, 4, 5 ve 6 değişkenli fonksiyonları en yalın biçimlerine indirmek.
- Önemsenmeyen durumların Karno haritasında nasıl kullanıldığını açıklamak.
- Quine–McCluskey algoritmasıyla herhangi bir sayıda değişkenli fonksiyonu sadeleştirmek.
- Toplayıcı, kod çözücü ve çoklayıcı devrelerinin yapısını ve işleyişini tanımlamak.
- RS, D, T ve JK flip-floplarının çalışma ilkelerini karşılaştırmak.
- Yazmaç ve sayaç devrelerinin flip-floplardan nasıl oluşturulduğunu göstermek.
- Mealy ve Moore durum makineleri arasındaki farkları açıklamak ve basit bir durum makinesi tasarlamak.

B.1 Giriş

Bir bilgisayarı anlamak, onu oluşturan soyutlama katmanlarını tanımakla başlar. Bölüm 6 ve Bölüm 7’de işlemci, AMB, yazmaç öbeği ve bellek gibi birimler birer kutu olarak çizilmiştir. Bu kutular Yazmaç Aktarım Seviyesi’nde (YAS) tasarlanır; ancak her kutunun içinde mantık kapılarından örülmüş bir devre yatar. Tıpkı bölüm fotoğrafındaki Galata Kulesi’nin taş taş üstüne yükselmesi gibi, bir işlemci de VE, VEYA ve DEĞİL kapılarının üst üste eklenmesiyle inşa edilir.

Sayısal tasarım kavramları kitabın birçok bölümünde karşımıza çıkar: Bölüm 5’te toplayıcı ve çarpıcı devreleri mantık kapılarından kurulur; Bölüm 6’da veriyolu çoklayıcıları ve denetim birimi Boole ifadeleriyle tasarlanır; Bölüm 7’de boru hattı yazmaçları flip-floplardan oluşur; Bölüm 8’de önbellek denetleyicisi bir durum makinesidir. Bu nedenle sayısal tasarım temelleri, kitabın tamamını destekleyen ortak bir altyapıdır.

Bu ek, sayısal tasarım dersini daha önce almış okuyucular için bir anımsatıcı; henüz almamış olanlar için ise bağımsız bir başlangıç kaynağı olarak tasarlanmıştır. Amaç, kitabın ana bölümlerini takip edebilmek için gereken asgari sayısal tasarım bilgisini tek bir yerde toplamaktır.

Sayısal tasarım, **birleşik** (İngilizcesi: *combinational*) ve **ardışıl** (İngilizcesi: *sequential*) olmak üzere iki temel devre sınıfı üzerine kuruludur. Birleşik devrelerde çıkış yalnızca o andaki girişlere bağlıdır; mantık kapıları, toplayıcılar ve çoklayıcılar bu gruptadır. Ardışıl devrelerde ise çıkış hem girişlere hem de devrenin o anki *durumuna* bağlıdır; flip-floplar, yazmaçlar ve sayaçlar bu gruba girer. Bir işlemci, bu iki sınıfın iç içe geçmesiyle oluşur: veriyolundaki AMB birleşik bir devredir, onu besleyen yazmaçlar ise ardışıl devrelerdir. Bu ekin ilk yarısı birleşik yapı taşlarını, ikinci yarısı ardışıl yapı taşlarını ele almaktadır.

Terim Kökenleri

birleşik devre (*combinational circuit*): Birleşmek fiilinden. Çıkış yalnızca girişlerin o anki birleşimine bağlıdır; bellek tutmaz. Türkçe kaynaklarda sıklıkla “kombinasyonel devre” olarak geçer; ancak “kombinasyonel” Türkçe bir sözcük değildir, İngilizce *combinational* sözcüğünün doğrudan aktarımıdır. Bu kitapta *birleşik* terimi kullanılmaktadır.

ardışıl devre (*sequential circuit*): Ardışık (consecutive) kökünden. Çıkış hem girişlere hem de devrenin durumuna bağlıdır; flip-floplarla bellek tutar. “Sıralı devre” de kullanılır; bu kitapta *ardışıl* tercih edilmiştir.

Bilgi Notu B.1: Sayısal Tasarım ve Teknoloji

Bu ekte anlatılan mantık kapıları, Boole cebri ve devre tasarım ilkeleri bugün CMOS (*Complementary Metal-Oxide-Semiconductor*) transistör teknolojisi üzerine kuruludur. CMOS, 1960’lardan bu yana yarı iletken endüstrisinin temel yapı taşıdır ve günümüzdeki tüm işlemciler, bellekler ve sayısal tümleşik devreler bu teknolojiyle üretilir.

Ancak CMOS tek olasılık değildir. Tarihte röleler, vakum tüpleri ve ayrık transistörler de mantık kapıları gerçekleştirmek için kullanılmıştır. Gelecekte ise kuantum hesaplama, spintronic veya fotonik gibi yeni teknolojiler farklı temel yapı taşları sunabilir; örneğin kuantum bilgisayarlarda klasik VE/VEYA kapıları yerine kuantum kapıları (Hadamard, CNOT, Toffoli) kullanılır ve Boole cebri yerine doğrusal cebir devreye girer. Bu ektteki ilkeler CMOS teknolojisine özgü değildir; Boole cebri ve durum makineleri gibi soyutlamalar, altlarındaki fiziksel teknoloji değişse bile geçerli kalır. Değişen, bu soyutlamaların donanımda nasıl gerçekleştirildiğidir.

Ekte sırasıyla şu konular ele alınmaktadır:

- Mantık kapıları ve doğruluk tabloları
- Boole cebri, De Morgan kuramları ve Karno haritalarıyla fonksiyon sadeleştirme
- Quine–McCluskey algoritması
- Birleşik yapı taşları: toplayıcılar, kod çözücüler ve çoklayıcılar
- Birleşik tasarım örneği
- Flip-floplar, yazmaçlar ve sayaçlar
- Mealy ve Moore durum makineleri
- Ardışıl tasarım örnekleri

İkilik tabandaki sayı gösterimleri, taban dönüşümleri ve işaretli sayı aritmetiği Ek A’da ayrıntılı olarak ele alınmıştır.

B.2 Mantık Kapıları ve Doğruluk Tabloları

B.2.1 İkilik Sistem ve Mantık

Bilgisayarlar ikilik sistemde çalışır: her basamak yalnızca 0 veya 1 değerini alır. Fiziksel düzeyde bu iki değer genellikle düşük ve yüksek gerilim seviyeleriyle temsil edilir. Aynı zamanda mantıktaki *yanlış* ve *doğru* kavramlarına karşılık gelirler. Sayısal tasarım, bu iki değer üzerinde işlem yapan devrelerin (mantık kapılarının) birleştirilerek karmaşık sistemler oluşturulmasıdır. İkilik sayı gösterimleri, taban dönüşümleri, işaretli sayılar ve ikiye tümleyen yöntemi

Ek A'da ayrıntılı olarak ele alınmaktadır.

B.2.2 Temel Kapılar ve Devre Sembolleri

Mantık kapıları, temel mantık fonksiyonlarını gerçekleştiren, transistörlerden oluşmuş devrelerdir. Üç temel mantık kapısı şunlardır:

- **VE (AND):** Çıkış, ancak *tüm* girişler 1 olduğunda 1 olur. Günlük hayattan bir benzetme: bir kapı, ancak hem anahtar *ve* hem şifre doğruysa açılır.
- **VEYA (OR):** Çıkış, girişlerden *en az biri* 1 olduğunda 1 olur. Benzetme: bir alarm, kapı *veya* pencere açılırsa çalar.
- **DEĞİL (NOT):** Tek girişli bir kapıdır; girişin tersini üretir. 0 girerse 1, 1 girerse 0 çıkar.

Bu üç kapıdan türetilen VE-DEĞİL (NAND), VEYA-DEĞİL (NOR), Dışlayan VEYA (XOR) ve Dışlayan VEYA DEĞİL (XNOR) kapıları da yaygın olarak kullanılır.

Bir mantık kapısının davranışı **doğruluk tablosu** ile tanımlanır: tablonun her satırı olası bir giriş kombinasyonunu, karşısındaki sütun ise o kombinasyon için üretilen çıkış değerini gösterir. n girişli bir kapının doğruluk tablosu 2^n satırdan oluşur: 2 girişli bir kapı için 4, 3 girişli bir fonksiyon için 8, 4 girişli bir fonksiyon için 16 satır gerekir. Aşağıdaki tablolarda ve şekilde yedi temel kapının doğruluk tabloları ve devre sembolleri verilmiştir.

Terim Kökenleri

Dışlayan VEYA (*Exclusive OR, XOR*): Türkçede sıklıkla “Özel VEYA” olarak anılır; ancak *exclusive* sözcüğünün doğru karşılığı “dışlayan”dır. İşlemin mantığı bu adı açıklar: iki giriş *aynı* ise sonuç 0 olur, aynı değerler *dışlanır*. Yalnızca girişler farklı olduğunda sonuç 1 olur. Boole cebirinde $A \oplus B$ ile gösterilir.

Dışlayan VEYA DEĞİL (*Exclusive NOR, XNOR*): Dışlayan VEYA'nın tersidir. Girişler *aynı* olduğunda 1, farklı olduğunda 0 üretir; bir eşitlik denetleyicisidir. $\overline{A \oplus B}$ ile gösterilir.

VE Kapısı

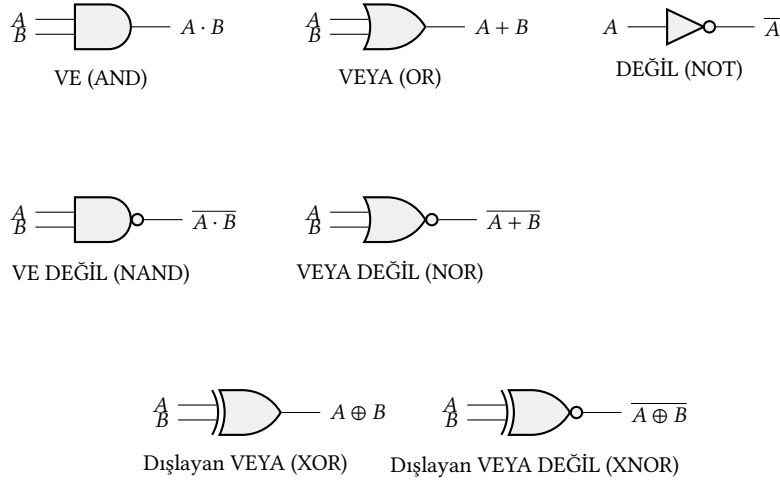
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

VEYA Kapısı

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

DEĞİL Kapısı

A	B
0	1
1	0



Şekil B.1: Temel ve türetilmiş mantık kapılarının devre sembolleri

Giriş sayısı arttıkça doğruluk tablosu hızla büyür; bu nedenle ilerleyen bölümlerde tanıtılacak olan Boole cebri ve Karno haritası gibi yöntemler, fonksiyonların tablo çizmeden sadeleştirilmesini sağlar. Bu yedi kapı arasında özellikle VE-DEĞİL ve VEYA-DEĞİL kapıları ayrı bir öneme sahiptir; bu iki kapı *evrensel* kapılardır; yalnızca bu kapılar kullanılarak diğer tüm kapılar gerçekleştirilebilir.

B.2.3 Evrensel Kapılar: VE-DEĞİL ve VEYA-DEĞİL

VE-DEĞİL (NAND) ve VEYA-DEĞİL (NOR) kapıları **evrensel kapı** olarak adlandırılır; çünkü bu kapılardan herhangi biri tek başına kullanılarak tüm temel mantık kapıları (VE, VEYA, DEĞİL) gerçekleştirilebilir ve dolayısıyla herhangi bir Boole fonksiyonu ifade edilebilir.

- **DEĞİL:** Bir VE-DEĞİL kapısının iki girişi birbirine bağlanırsa DEĞİL kapısı elde edilir: $\overline{x \cdot x} = \bar{x}$.
- **VE:** İki VE-DEĞİL kapısı art arda bağlanarak VE kapısı oluşturulabilir: önce $\overline{x \cdot y}$, ardından bu çıkışın DEĞİL'i alınarak $x \cdot y$ elde edilir.
- **VEYA:** $x + y = \overline{\overline{x} \cdot \overline{y}}$ eşitliğine göre (ilerleyen bölümlerde De Morgan kuramı olarak tanıtılacaktır) bu ifade üç VE-DEĞİL kapısıyla gerçekleştirilebilir.

Aynı mantık, VEYA-DEĞİL kapısı için de geçerlidir. Uygulamada VE-DEĞİL ve VEYA-DEĞİL kapıları, transistor düzeyinde VE ve VEYA kapılarından daha verimli üretilebildiği için entegre devre tasarımında yaygın olarak tercih edilir.

B.3 Boole Cebri

Doğruluk tabloları her durumu teker teker listeleterek bir fonksiyonu tanımlar; ancak giriş sayısı arttıkça bu yöntem kullanışsızlaşır. Boole cebri, mantık fonksiyonlarını cebirsel ifadelerle göstermeye ve bu ifadeleri kurallara dayalı olarak sadeleştirmeye olanak tanır. 1854 yılında George Boole tarafından geliştirilen bu cebir, VE işlemini “ $\cdot$ ” (çarpma benzeri), VEYA işlemini “ $+$ ” (toplama benzeri) ve DEĞİL işlemini üst çizgi ($\bar{x}$) ile gösterir.

Bilgi Notu B.2: George Boole ve Mantığın Cebirleşmesi

George Boole (1815–1864), İngiliz matematikçi ve mantıkçıdır. 1854'te yayımladığı *An Investigation of the Laws of Thought* [5] adlı yapıtında, mantıksal akıl yürütmenin cebirsel denklemlerle ifade edilebileceğini göstermiştir. Boole bu çalışmayı yaparken ne transistör ne de elektrikli devre diye bir şey vardı; ilk transistör ancak 1947'de icat edilecekti. Boole'un amacı, insan düşüncesinin kurallarını matematiksel olarak modellemektir. Boole cebrinin donanım tasarımına uygulanabileceğini 1937'de Claude Shannon keşfetmiştir. Shannon, yüksek lisans tezinde [38] elektrik anahtarlarının (rölelerin) Boole cebrinin fiziksel bir gerçekleştirilmesi olduğunu göstermiş ve böylece sayısal devre tasarımının kuramsal temelini atmıştır. Bugün aynı cebir CMOS transistörleriyle gerçekleştirilmektedir; ancak tarihte röleler, vakum tüpleri ve mekanik düzenekler de mantık kapılarını gerçekleştirmek için kullanılmıştır. Boole cebrinin gücü, altındaki fiziksel teknolojiye bağımsız olmasıdır.

Bilgi Notu B.3: Boole Cebirinde Gösterim ve Öncelik

Boole cebirinde “.” işareti sıklıkla yazılmaz: $x \cdot y$ yerine kısaca xy yazılır. Ayrıca VE işlemi, VEYA işleminden önce gelir (tıpkı aritmetikteki çarpmanın toplamadan önce gelmesi gibi). Örneğin $x + yz$ ifadesi $x + (y \cdot z)$ anlamına gelir.

Boole cebrinin temel özellikleri (Huntington önermeleri) Tablo B.1’nde verilmiştir.

Tablo B.1: Boole cebrinin temel özellikleri

Özellik	VEYA (+) biçimi	VE (·) biçimi
Kapalılık	$1 + 1 = 1$	$0 \cdot 0 = 0$
Birim eleman	$x + 0 = x$	$x \cdot 1 = x$
Değişme	$x + y = y + x$	$x \cdot y = y \cdot x$
Dağılma	$x + (y \cdot z) = (x + y) \cdot (x + z)$	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
Ters	$x + \bar{x} = 1$	$x \cdot \bar{x} = 0$
Etkisizlik	$x + x = x$	$x \cdot x = x$
Yutma	$x + 1 = 1$	$x \cdot 0 = 0$
Çift tümleyen	$\bar{\bar{x}} = x$	$\bar{\bar{x}} = x$

Tablodaki ikiliğe dikkat edilmelidir: her özelliğin bir VEYA biçimi, bir de VE biçimi vardır. Bu ikiliğe Boole cebirinde *eşiklik* (*duality*) denir: herhangi bir doğru ifadede “+” ile “.”’yi ve “0” ile “1”i değiştirirseniz, elde edilen ifade de doğrudur.

Bu kurallar, Boole ifadelerini adım adım sadeleştirmek için kullanılır. Bir kuralın doğruluğu, doğruluk tablosu ile her zaman doğrulanabilir.

Bu üç temel işlemin (·, +, ¬) doğruluk tabloları Bölüm B.2’de verilmiştir. Kısaca anımsatmak gerekirse: $x \cdot y$ yalnızca her iki giriş 1 ise 1 üretir; $x + y$ girişlerden en az biri 1 ise 1 üretir; $\bar{x}$ ise girişin tersini verir.

Bir Boole ifadesinin davranışını anlamak için doğruluk tablosu çıkarılır: her giriş kombinasyonu ifadeye yerleştirilir ve sonuç hesaplanır. Aşağıdaki üç fonksiyon, farklı kapı birleşimlerini (salt VE, salt VEYA ve karışık VE+VEYA) örneklemektedir. Bu tablonun ilerleyen

bölümlerde mini terim gösteriminde (Örnek B.3) kullanılacağına dikkat ediniz.

Tablo B.2: $F_1 = xy\bar{z}$, $F_2 = x + \bar{y} + \bar{z}$, $F_3 = \bar{x}(y + z)$ fonksiyonlarının doğruluk tablosu

x	y	z	$F_1 = xy\bar{z}$	$F_2 = x + \bar{y} + \bar{z}$	$F_3 = \bar{x}(y + z)$
0	0	0	0	1	0
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	0	0	1
1	0	0	0	1	0
1	0	1	0	1	0
1	1	0	1	1	0
1	1	1	0	1	0

Örnek B.1

$F = x\bar{y} + xy + \bar{x}y$ Boole ifadesini cebir kurallarıyla sadeleştirin.

Çözüm:

Adım 1 (Dağılma, VE biçimi): İlk iki terimde x ortaktır; dağılma kuralını ($x \cdot (y + z) = xy + xz$) tersten uygulayarak x 'i paranteze alırız:

$$F = x\bar{y} + xy + \bar{x}y = x(\bar{y} + y) + \bar{x}y$$

Adım 2 (Ters özelliği): Parantez içinde $\bar{y} + y = 1$ olur (ters özelliği, Tablo B.1):

$$F = x \cdot 1 + \bar{x}y$$

Adım 3 (Birim eleman): $x \cdot 1 = x$ olduğundan (birim eleman özelliği):

$$F = x + \bar{x}y$$

Adım 4 (Dağılma, VEYA biçimi): $x + \bar{x}y$ ifadesine VEYA biçimindeki dağılma kuralını ($x + yz = (x + y)(x + z)$) uygulayarak:

$$F = (x + \bar{x})(x + y) = 1 \cdot (x + y) = x + y$$

Burada yine ters özelliği ($x + \bar{x} = 1$) ve birim eleman ($1 \cdot (x + y) = x + y$) kullanılmıştır.

Sonuç: Üç terimli ifade iki değişkenli yalın bir VEYA işlemine indirildi: $F = x + y$.

B.3.1 De Morgan Kuramları

De Morgan kuramları, VE ile VEYA işlemleri arasında tümleyen yardımıyla geçiş yapmayı sağlayan iki temel kuraldır:

$$\overline{x \cdot y} = \bar{x} + \bar{y} \quad (\text{VE-DEĞİL} = \text{tümleyenlerin VEYA'sı})$$

$$\overline{x + y} = \bar{x} \cdot \bar{y} \quad (\text{VEYA-DEĞİL} = \text{tümleyenlerin VE'si})$$

Bu kuramlar herhangi bir sayıda değişkene genelleştirilebilir:

$$\overline{x_1 \cdot x_2 \cdots x_n} = \overline{x_1} + \overline{x_2} + \cdots + \overline{x_n}$$

$$\overline{x_1 + x_2 + \cdots + x_n} = \overline{x_1} \cdot \overline{x_2} \cdots \overline{x_n}$$

De Morgan kuramları, devrelerin VE-DEĞİL (NAND) veya VEYA-DEĞİL (NOR) kapıları ile gerçekleştirilmesinde ve Boole ifadelerinin sadeleştirilmesinde sıklıkla kullanılır.

Örnek B.2

De Morgan kuramını kullanarak $F = \overline{\overline{x} \cdot y} + x \cdot \overline{y}$ ifadesini sadeleştirin.

Çözüm:

Dış tümleyene De Morgan uygulanır:

$$F = (\overline{\overline{x} \cdot y}) \cdot (\overline{x \cdot \overline{y}})$$

Her terime tekrar De Morgan uygulanır:

$$F = (x + \overline{y}) \cdot (\overline{x} + y)$$

Açılır ve sadeleştirilir:

$$F = x\overline{x} + xy + \overline{x}\overline{y} + \overline{y}y = xy + \overline{x}\overline{y}$$

Sonuç, $xy + \overline{x}\overline{y}$; x ve y 'nin eşdeğerlik (Dışlayan VEYA DEĞİL) fonksiyonudur: girişler aynı olduğunda 1, farklı olduğunda 0 üretir.

De Morgan kuramları ile sadeleştirme sırasında karşılaşılan farklı mantık işlemlerini tanımak yararlıdır. İki değişkenli Boole fonksiyonlarının tamamı $2^{2^2} = 16$ tanedir. Tablo B.3'te bu 16 fonksiyonun her biri, sembolü ve açıklamasıyla birlikte listelenmiştir.

Tablo B.3: Standart Mantık İşlemleri

Fonksiyon	Sembol	Ad	Açıklama
0	–	Boş	Her zaman 0
$x \cdot y$	VE	VE	Her ikisi de 1 ise 1
$x \cdot \bar{y}$	–	Engelleme	$x = 1$ ve $y = 0$ ise 1
x	–	Aktarım	x değeri
$\bar{x} \cdot y$	–	Engelleme	$x = 0$ ve $y = 1$ ise 1
y	–	Aktarım	y değeri
$x\bar{y} + \bar{x}y$	$\oplus$	Dışlayan VEYA	Farklı ise 1
$x + y$	VEYA	VEYA	En az biri 1 ise 1
$\overline{(x + y)}$	VEYA DEĞİL	VEYA DEĞİL	Her ikisi de 0 ise 1
$xy + \bar{x}\bar{y}$	$\odot$	Eşdeğerlik	Aynı ise 1
$\bar{y}$	–	Tümleyen	y 'nin tersi
$x + \bar{y}$	–	İçerme	$x = 1$ veya $y = 0$ ise 1
$\bar{x}$	–	Tümleyen	x 'in tersi
$\bar{x} + y$	–	İçerme	$x = 0$ veya $y = 1$ ise 1
$\overline{(x \cdot y)}$	VE DEĞİL	VE DEĞİL	En az biri 0 ise 1
1	–	Birim Eleman	Her zaman 1

Boole cebri kuralları herhangi bir ifadeyi sadeleştirmek için kullanılabilir; ancak sadeleştirmenin sistematik yolunu bulmak özellikle büyük ifadelerde zorlaşır. Bu sorunu çözmek için önce fonksiyonların standart gösterim biçimlerini (mini terimler ve maksı terimler) tanıyacağız; ardından Karno haritası yöntemini öğreneceğiz.

B.3.2 Mini Terimler ve Maksı Terimler

Herhangi bir Boole fonksiyonu, standart iki biçimden biriyle ifade edilebilir. Bu biçimleri anlamak için önce mini terim ve maksı terim kavramlarını tanıyalım.

Mini terim (İngilizcesi: *minterm*), bir fonksiyonun tüm değişkenlerinin VE ($\cdot$) işlemiyle birleştirilmesiyle oluşan bir ifadedir. Her değişken, doğruluk tablosundaki satırda 1 ise kendisi, 0 ise tümleyeni (üst çizgili hali) olarak yazılır. Örneğin 3 değişkenli bir fonksiyonda $x = 1$, $y = 0$, $z = 1$ satırının mini terimi $x\bar{y}z$ olur. Her mini terim, doğruluk tablosunun tam olarak bir satırında 1 üretir ve diğer tüm satırlarda 0 verir.

Maksı terim (İngilizcesi: *maxterm*) ise mini terimin ikilidir: tüm değişkenler VEYA (+) işlemiyle birleştirilir ve bu kez kurallar tersine döner: değişken 1 ise tümleyeni, 0 ise kendisi alınır. Aynı örnek için ($x = 1$, $y = 0$, $z = 1$) maksı terim $\bar{x} + y + \bar{z}$ olur. Her maksı terim, doğruluk tablosunun tam olarak bir satırında 0 üretir.

Tablo B.4: Mini ve maksı terimlerin gösterimi

x	y	z	Mini Terim	Sembol	Maksi Terim	Sembol
0	0	0	$\bar{x}\bar{y}\bar{z}$	m_0	$x + y + z$	M_0
0	0	1	$\bar{x}\bar{y}z$	m_1	$x + y + \bar{z}$	M_1
0	1	0	$\bar{x}y\bar{z}$	m_2	$x + \bar{y} + z$	M_2
0	1	1	$\bar{x}yz$	m_3	$x + \bar{y} + \bar{z}$	M_3
1	0	0	$x\bar{y}\bar{z}$	m_4	$\bar{x} + y + z$	M_4
1	0	1	$x\bar{y}z$	m_5	$\bar{x} + y + \bar{z}$	M_5
1	1	0	$xy\bar{z}$	m_6	$\bar{x} + \bar{y} + z$	M_6
1	1	1	xyz	m_7	$\bar{x} + \bar{y} + \bar{z}$	M_7

Fonksiyonların mini terimler ve maksı terimlerle gösterilebilmeleri için belirli bir formda yazılmaları gerekir. Mini terimler toplamı Σ gösterimi ile, maksı terimler çarpımı ise Π gösterimi ile ifade edilir.

Örnek B.3

Tablo B.2’te tanımlanan $F_1 = xy\bar{z}$, $F_2 = x + \bar{y} + \bar{z}$ ve $F_3 = \bar{x}(y + z)$ fonksiyonlarını mini terimler toplamı (Σ) biçiminde ifade edin.

Çözüm:

Doğruluk tablosundan her fonksiyonun 1 değerini aldığı satırlar belirlenir:

F_1 , yalnızca $x = 1$, $y = 1$, $z = 0$ satırında 1 olur:

$$F_1 = m_6 = x y \bar{z} = \Sigma(6)$$

F_3 , $x = 0$ olup y veya z ’den en az birinin 1 olduğu üç satırda 1 olur:

$$F_3 = m_1 + m_2 + m_3 = \bar{x}\bar{y}z + \bar{x}y\bar{z} + \bar{x}yz = \Sigma(1, 2, 3)$$

F_2 ise yalnızca $x = 0$, $y = 1$, $z = 1$ satırında 0 olur; diğer tüm satırlarda 1 üretir:

$$F_2 = m_0 + m_1 + m_2 + m_4 + m_5 + m_6 + m_7 = \Sigma(0, 1, 2, 4, 5, 6, 7)$$

F_2 ’nin yedi mini terimine karşılık yalnızca bir maksı terimi olduğuna dikkat ediniz: $F_2 = \Pi(3)$. Bu durum, VEYA tabanlı fonksiyonların genellikle çok sayıda mini terim ürettiğini ve maksı terim gösteriminin bu tür fonksiyonlar için daha yalın olduğunu gösterir.

Şimdiye kadar Boole cebrinin kuralları, De Morgan kuramları ve kapı türleri ele alındı. Bir sonraki bölümde, bu bilgileri kullanarak Boole fonksiyonlarını sistematik olarak en yalın biçimlerine indirgeyen Karno haritası yöntemi tanıtılacaktır.

B.4 Boole Fonksiyonlarının Sadeleştirilmesi

Bir Boole fonksiyonunu sadeleştirmek, aynı davranışı daha az kapıyla, dolayısıyla daha az transistörle, daha az gecikmeyle ve daha az güç tüketimiyle gerçekleştirmek demektir. Sadeleştirme için üç temel yaklaşım vardır:

1. **Cebirsel sadeleştirme:** Önceki bölümde gördüğümüz Boole cebri kuralları (dağılma, ters, De Morgan vb.) kullanılarak ifade adım adım yalınlaştırılır. En temel yöntemdir; ancak hangi kuralın hangi sırada uygulanacağı deneyim gerektirir ve en yalın sonuca ulaşma garantisi yoktur.
2. **Karno haritası:** Görsel bir yöntemdir; fonksiyonun doğruluk tablosunu özel bir düzende haritaya yerleştirerek komşu 1 değerlerini gruplamaya ve değişkenleri elemeye dayanır. 2–4 değişkenli fonksiyonlarda sistematik ve güvenilirdir.
3. **Quine–McCluskey algoritması:** Tablo tabanlı bir algoritmadır; değişken sayısından bağımsız olarak çalışır ve bilgisayar programıyla kolayca gerçekleştirilebilir. Büyük fonksiyonlarda Karno haritasının yerini alır.

Cebirsel sadeleştirme örnekleri önceki bölümde (Örnek B.1 ve B.2) verilmişti. Bu bölümde Karno haritası ve Quine–McCluskey yöntemlerini ele alacağız.

B.4.1 Karno Haritası Yöntemi

Karno haritası, 1953'te Maurice Karnaugh tarafından geliştirilen görsel bir yöntemdir ve 2, 3 veya 4 değişkenli fonksiyonları sistematik olarak en yalın biçimlerine indirgemeyi sağlar.

Bilgi Notu B.4: Maurice Karnaugh ve Karno Haritası

Maurice Karnaugh (1924–2024), Bell Laboratuvarları'nda çalışan Amerikalı fizikçi ve matematikçidir. 1953'te yayımladığı makalede [20] Boole fonksiyonlarını görsel olarak sadeleştirmeye yarayan harita yöntemini tanıtmıştır. Yöntem aslında 1952'de Edward Veitch'in önerdiği bir diyagramın geliştirilmiş halidir; Karnaugh'un katkısı, hücrelerin Gray kodu sıralamasıyla dizilmesini sağlayarak komşu hücrelerin yalnızca tek bir değişkende farklılık göstermesini garanti etmesidir. Soyadı İngilizce "KAR-no" biçiminde telaffuz edilir; Türkçe kaynaklarda bu telaffuza uygun olarak "Karno haritası" yazılması yerleşmiştir.

Karno haritasının temel ilkesi şudur: haritada yan yana (komşu) olan hücreler yalnızca *tek bir değişkende* farklılık gösterecek biçimde düzenlenmiştir (Gray kodu sıralamasıyla). Bu sayede komşu hücrelerdeki 1 değerleri gruplandırıldığında, farklılık gösteren değişken sadeleşir ve ifade basitleşir.

Gruplama kuralları:

1. Gruplar yalnızca 2^n (1, 2, 4, 8, ...) boyutunda dikdörtgen olabilir.
2. Gruplardaki tüm hücreler 1 (veya önemsenmeyen durum X) içermelidir.
3. Gruplar ne kadar büyükse sadeleştirme o kadar iyidir.
4. Her 1 değeri en az bir gruba dahil olmalıdır; bir hücre birden fazla grupta yer alabilir.
5. Haritanın kenarları birbirine bağlıdır: sol kenar sağ kenarla, üst kenar alt kenarla komşu kabul edilir (harita bir torus gibi düşünülebilir).

Bilgi Notu B.5: Değişken Sırası ve Mini Terim Numarası

Mini terim numarası (m_i), değişkenlerin soldan sağa sıralandığı ikilik sayının onluk karşılığıdır. Örneğin 3 değişkenli bir fonksiyonda a, b, c sıralamasıyla $a = 1, b = 0, c = 1$ kombinasyonu $101_2 = 5_{10}$ olduğundan m_5 numarasını alır. Karno haritasında sütun ve satır başlıkları Gray kodu (00, 01, 11, 10) sıralamasını izlediğinden, mini terim numaraları ardışık değildir; bu düzenleme komşu hücrelerin yalnızca tek bir bitte farklılaşmasını

sağlar.

İki Değişkenli Karno Haritası

İki değişkenli bir fonksiyonun Karno haritası $2 \times 2 = 4$ hücreden oluşur. Her hücre bir mini terime karşılık gelir. Satırlar a değişkeninin, sütunlar b değişkeninin değerlerini temsil eder.

$a \backslash b$		0	1
a	0	m_0	m_1
	1	m_2	m_3

b

a	b	m_i
0	0	m_0
0	1	m_1
1	0	m_2
1	1	m_3

Şekil B.2: İki değişkenli Karno haritası (a, b) ve mini terim numaraları

Örnek B.4

$F(a, b) = \Sigma(0, 3)$ fonksiyonunu Karno haritası ile sadeleştirin.

Çözüm:

Mini terimler haritaya yerleştirilir: m_0 ($a = 0, b = 0$) ve m_3 ($a = 1, b = 1$) hücrelerine 1 yazılır.

$a \backslash b$		0	1
a	0	1	0
	1	0	1

$\bar{a}\bar{b}$
 $a \cdot b$

m_0 ve m_3 köşegen konumdadır ve komşu değildir (iki bitte farklılaşır: hem a hem b değişir). Bu nedenle gruplandırılmazlar ve ifade sadeleştirilemez:

$$F = \bar{a}\bar{b} + a \cdot b = a \odot b$$

Bu, Dışlayan VEYA DEĞİL (eşitlik) fonksiyonudur: a ve b aynı değerdeyse çıkış 1 olur.

Üç Değişkenli Karno Haritası

Üç değişkenli bir fonksiyonun Karno haritası $2 \times 4 = 8$ hücreden oluşur. Satırlar a değişkeninin değerlerini, sütunlar bc çiftinin Gray kodu sırasıyla (00, 01, 11, 10) dizilmiş değerlerini temsil eder. Bu sıralama, komşu sütunların yalnızca tek bir bitte farklılaşmasını garanti eder. Ayrıca haritanın **sağ kenarı sol kenarına bitişik** sayılır: en sağdaki sütun (10) ile en soldaki sütun (00) yalnızca tek bir bitte farklılaştığından, bu iki sütundaki hücreler de komşudur. Gruplama yaparken bu kenar sarma göz önünde bulundurulmalıdır.

		b			
		00	01	11	10
$a \backslash bc$					
	0	m_0	m_1	m_3	m_2
	1	m_4	m_5	m_7	m_6
		c			

a	b	c	m_i
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

Şekil B.3: Üç değişkenli Karno haritası ve mini terim numaraları (sütunlar Gray kodu sırasıyla dizilir: 00, 01, 11, 10)

Örnek B.5

$F = \bar{a}\bar{b}\bar{c} + abc + ab\bar{c} + \bar{a}b\bar{c}$ fonksiyonunu Karno haritası ile sadeleştirin.

Çözüm:

Fonksiyondaki mini terimler Karno haritasına yerleştirilir ve komşu 1 değerleri gruplandırılır.

		b			
		00	01	11	10
$a \backslash bc$					
	0	1	0	0	1
	1	0	0	1	1
		c			

Örnek B.5: F fonksiyonunun Karno haritası ile sadeleştirilmesi

m_0 yalnızca $\bar{a}\bar{c}$ grubu tarafından, m_7 ise yalnızca $a \cdot b$ grubu tarafından kapsandığından her iki grup da temel asal kapsayandır. Bu iki grup, dört 1 değerinin tamamını (m_0, m_2, m_6, m_7) kapsar; $\{m_2, m_6\}$ çifti ($b\bar{c}$) de geçerli bir grup olsa da her iki hücresi zaten kapsandığından gereksizdir.

Gruplandırma sonucunda:

$$F = a \cdot b + \bar{a} \cdot \bar{c}$$

Dört Değişkenli Karno Haritası

Dört değişkenli harita $4 \times 4 = 16$ hücreden oluşur; hem satırlar (ab) hem sütunlar (cd) Gray kodu sırasıyla dizilir. Üç değişkenli haritada yalnızca sağ-sol kenarlar bitişikken, dört değişkenli haritada **hem sağ-sol hem üst-alt kenarlar birbirine bitişik** sayılır: harita bir torus (simit) gibi düşünülebilir. Dolayısıyla dört köşedeki hücreler de birbirine komşudur.

		c			
$ab \backslash cd$	00	01	11	10	
00	m_0	m_1	m_3	m_2	} b
01	m_4	m_5	m_7	m_6	
11	m_{12}	m_{13}	m_{15}	m_{14}	
10	m_8	m_9	m_{11}	m_{10}	
		d			

Şekil B.4: Dört değişkenli Karno haritası ve mini terim numaraları

Örnek B.6

$F = \bar{a}b\bar{c} + \bar{a}\bar{c}d + ab\bar{c}\bar{d} + bc\bar{d}$ fonksiyonunu Karno haritası ile sadeleştirin.

Çözüm:

		c			
$ab \backslash cd$	00	01	11	10	
00	0	1	0	0	} b
01	1	1	0	1	
11	1	0	0	1	
10	0	0	0	0	
		d			

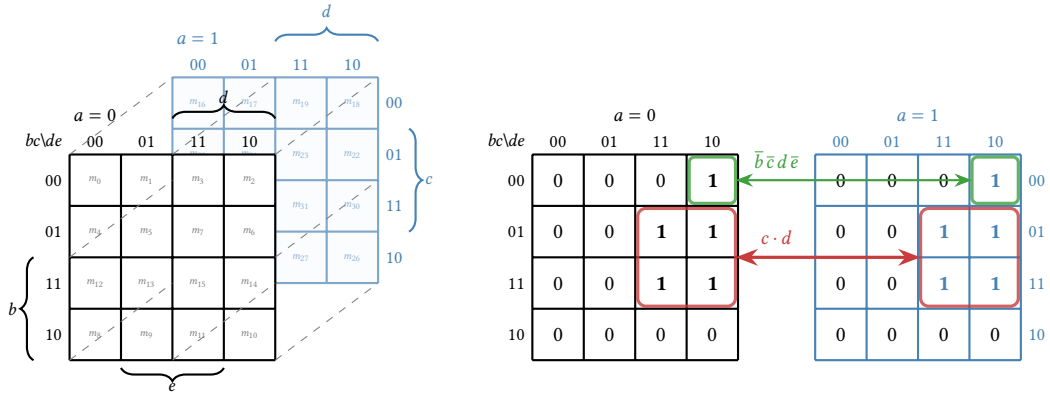
Örnek B.6: F fonksiyonunun 4 değişkenli Karno haritası ile sadeleştirilmesi

Gruplandırma sonucunda:

$$F = b\bar{d} + \bar{a}\bar{c}d$$

Beş ve Altı Değişkenli Karno Haritaları

Dört değişkenden fazlası için tek bir Karno haritası yeterli olmaz. Beş değişkenli fonksiyon iki adet 4×4 haritayla, altı değişkenli fonksiyon ise dört adet 4×4 haritayla gösterilir. Her harita beşinci (ve altıncı) değişkenin belirli bir değerine karşılık gelir. **Aynı konumdaki hücreler haritalar arasında komşu sayılır**; gruplama yapılırken bu komşuluk göz önünde bulundurulmalıdır. Aşağıdaki şekillerde haritalar üç boyutlu perspektifle gösterilmiştir; arka katmandaki hücreler ön katmandaki aynı konumdaki hücrelerle komşudur.

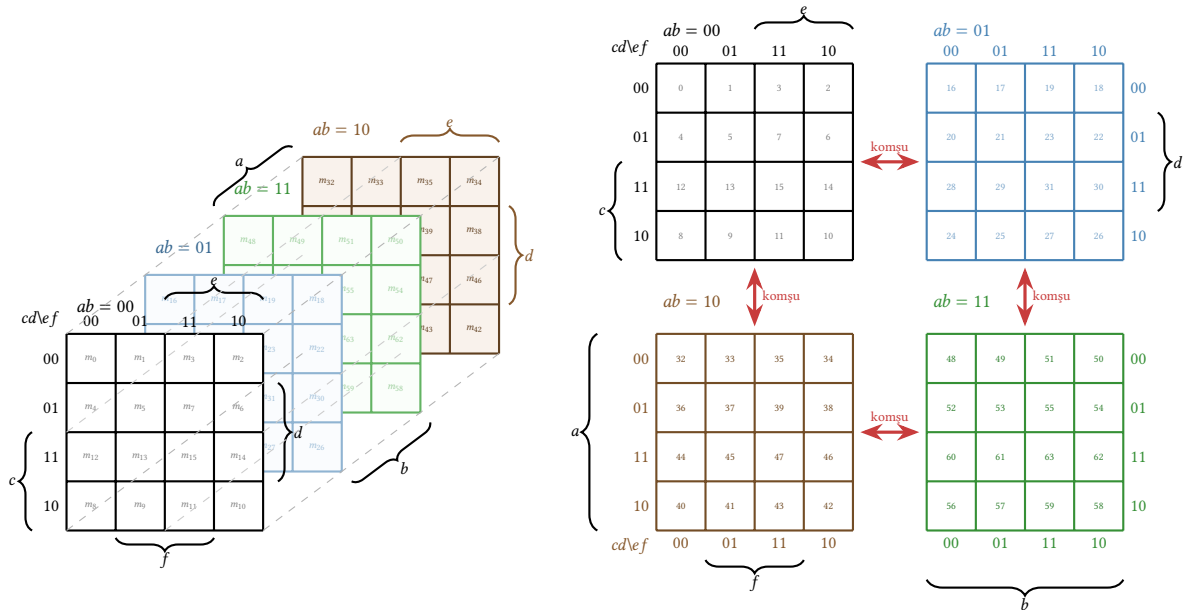


Şekil B.5: Beş değişkenli Karno haritası (sol: 3D görünüm, sağ: 2D açılım ve gruplama örneği, $F = \Sigma(2, 6, 7, 14, 15, 18, 22, 23, 30, 31)$)

Şekil B.5'in sağ tarafındaki 2D açılımda iki tür gruplama görülmektedir:

- **Her iki katmanda ortak grup (kırmızı):** $a = 0$ katmanındaki $\{m_6, m_7, m_{14}, m_{15}\}$ ve $a = 1$ katmanındaki $\{m_{22}, m_{23}, m_{30}, m_{31}\}$ hücreleri aynı konumdadır ve hepsi 1 değerini taşır. Sekiz hücrelik bu grup a değişkenini de eler ve sonuç yalnızca $c \cdot d$ olur.
- **Katmanlar arası ikili grup (yeşil):** m_2 ($a = 0, bc=00, de=10$) ile m_{18} ($a = 1$, aynı konum) komşudur. İkisi arasında yalnızca a değişkeni farklılaştığından bu ikili grubun sonucu $\bar{b} \bar{c} d \bar{e}$ olur.

Sonuç: $F = c \cdot d + \bar{b} \bar{c} d \bar{e}$.



Şekil B.6: Altı değişkenli Karno haritası (sol: 3D görünüm, sağ: 2D açılım, 2x2 düzen). Komşu haritalar arasındaki aynı konumdaki hücreler de komşudur.

Değişken sayısı arttıkça Karno haritası hızla karmaşıklaşır ve elle çözüm zorlaşır. Yedi ve üzeri değişken için üç boyutlu gösterim bile olanaklı değildir. Bu nedenle beşten fazla değişkenli fonksiyonlar için bir sonraki alt bölümde tanıtılacak Quine–McCluskey algoritması tercih edilir.

B.4.2 Önemsenmeyen Durumlar

Bazı giriş kombinasyonları uygulamada hiçbir zaman gerçekleşmez ya da gerçekleşse bile çıkışın ne olduğu önemsizdir. Bu durumlar doğruluk tablosunda ve Karno haritasında “X” ile gösterilir ve **önemsenmeyen durum** (İngilizcesi: *don't care*) olarak adlandırılır.

Önemsenmeyen durumlar, Karno haritasında gruplamayı büyütme için hem 1 hem de 0 olarak ele alınabilir. Bu esneklik, daha büyük gruplar oluşturulmasına ve dolayısıyla daha sadeleştirilmiş ifadelerin elde edilmesine olanak tanır. Tasarımcı, her X hücrelerini gruplama sırasında 1 veya 0 olarak serbestçe seçer; hangisi daha büyük bir grup oluşturuyorsa o tercih edilir.

Önemsenmeyen durumlar özellikle denetim birimi tasarımlarında sıkça karşılaşılır; çünkü buyruğun işlem kodu alanındaki bazı bit kombinasyonları geçerli bir buyrukla eşleşmez (bu ekin ilerleyen bölümlerinde bunun bir örneği verilecektir).

Örnek B.7

Bir BCD (ikilik Kodlanmış Onluk) – sayısal gösterimde yalnızca 0000’dan 1001’e kadar olan kombinasyonlar geçerlidir (onluk 0–9). $F(a, b, c, d) = \Sigma(1, 3, 5, 7, 9)$ fonksiyonunu, geçersiz BCD kombinasyonlarını ($m_{10}-m_{15}$) önemsenmeyen durum olarak kullanarak sadeleştirin.

Çözüm:

Önemsenmeyen durumlar: $d = \{10, 11, 12, 13, 14, 15\}$.

		c			
$ab \backslash cd$		00	01	11	10
a	00	0	1	1	0
	01	0	1	1	0
	11	X	X	X	X
	10	0	1	X	X
		d			

Önemsenmeyen durumlar (X) gruplama sırasında 1 gibi ele alınarak büyük bir grup oluşturulur. Sütun 01 ve 11’deki tüm hücreler (1 ve X değerleri) tek bir 4×2 grup oluşturur. Bu grup yalnızca d değişkenine bağlıdır:

$$F = d$$

Önemsenmeyen durumlar kullanılmasaydı, sonuç çok daha karmaşık olurdu. BCD’de 10–15 kombinasyonları hiçbir zaman oluşmadığından, bu sadeleştirme güvenlidir.

B.4.3 Quine–McCluskey Algoritması

Karno haritası 6 değişkene kadar (üç boyutlu gösterimle) uygulanabilir; ancak 7 ve üzeri değişken için üç boyutlu gösterim bile olanaklı değildir ve görsel gruplama güvenilirliğini yitirir. **Quine–McCluskey algoritması**, 1952’de Willard Van Orman Quine tarafından önerilmiş [36] ve 1956’da Edward McCluskey tarafından geliştirilmiş [25] tablo tabanlı bir yöntem-

dir. Karno haritasıyla aynı sonucu verir, ancak herhangi bir sayıda değişkenle çalışabilir ve bir bilgisayar programıyla kolayca gerçekleştirilebilir.

Algoritma iki aşamadan oluşur:

Aşama 1: Asal Kapsayanları Bulma

Fonksiyonun 1 ürettiği mini terimler, içerdikleri 1 bitlerinin sayısına göre gruplara ayrılır. Ardından komşu gruplar arasında yalnızca tek bir bitte farklılık gösteren terim çiftleri birleştirilir; farklı bit “–” ile değiştirilir. Birleştirilemeyen terimler **asal kapsayan** (İngilizcesi: *prime implicant*) olarak işaretlenir. Bu süreç, hiçbir yeni birleştirme yapılamayınca kadar tekrarlanır.

Aşama 2: Temel Asal Kapsayanları Seçme

Bir kapsama tablosu oluşturulur: satırlar asal kapsayanları, sütunlar mini terimleri temsil eder. Yalnızca tek bir asal kapsayan tarafından kapsanan mini terimler, o asal kapsayanı **temel asal kapsayan** (İngilizcesi: *essential prime implicant*) kılar. Temel asal kapsayanlar seçildikten sonra, kalan mini terimler için en az sayıda asal kapsayan seçilerek en yalın ifade elde edilir.

Örnek B.8

$F(a, b, c, d) = \Sigma(0, 1, 2, 5, 6, 7, 8, 9, 10, 14)$ fonksiyonunu Quine–McCluskey algoritmasıyla sadeleştirin.

Çözüm:

Aşama 1: Mini terimler 1-bit sayısına göre gruplanır ve komşu gruplar birleştirilir:

Grup	Mini terimler	1. birleştirme	2. birleştirme
0 (sıfır 1)	m_0 : 0000	(0,1): 000–	(0,1,8,9): –00– AK
1 (bir 1)	m_1 : 0001	(0,2): 00–0	(0,2,8,10): –0–0 AK
	m_2 : 0010	(0,8): –000	(2,6,10,14): ––10 AK
	m_8 : 1000	(1,5): 0–01	
2 (iki 1)	m_5 : 0101	(1,9): –001	
	m_6 : 0110	(2,6): 0–10	
	m_9 : 1001	(2,10): –010	
	m_{10} : 1010	(8,9): 100–	
3 (üç 1)	m_7 : 0111	(8,10): 10–0	
	m_{14} : 1110	(5,7): 01–1 AK	
		(6,7): 011–	
		(6,14): –110	
		(10,14): 1–10	

Asal kapsayanlar (AI): –00– (bd' yerine $\bar{b}\bar{c}$), –0–0 ($\bar{b}\bar{d}$), ––10 ($c\bar{d}$), 01–1 ($\bar{a}bd$).

Aşama 2: Kaplama tablosu

AK	m_0	m_1	m_2	m_5	m_6	m_7	m_8	m_9	m_{10}	m_{14}
$\bar{b}\bar{c}$	×	×					×	×		
$\bar{b}\bar{d}$	×		×				×		×	
$c\bar{d}$			×		×				×	×
$\bar{a}bd$				×		×				

m_5 ve m_7 yalnızca $\overline{a}bd$ tarafından kapsandığından bu asal kapsayan **temeldir**. Kalan mini terimler için $\overline{b}\overline{c}$ ve $\overline{c}d$ seçildiğinde tüm mini terimler kapsanır:

$$F = \overline{b}\overline{c} + \overline{c}d + \overline{a}bd$$

Python ile Quine–McCluskey

Quine–McCluskey algoritması, bilgisayar programıyla kolayca gerçekleştirilebilir. Aşağıdaki Python programı, algoritmayı sıfırdan kodlamaktadır. Program iki aşamayı (asal kapsayanları bulma ve temel asal kapsayanları seçme) adım adım gerçekleştirir.

```

1 def quine_mccluskey(degisken_sayisi, birler):
2     # --- Aşama 1: Asal kapsayanları bulma ---
3     # Başlangıç: her mini terim tek başına bir kapsayan
4     kapsayanlar = set()
5     for m in birler:
6         kapsayanlar.add((frozenset([m]),
7                                 format(m, f'0{degisken_sayisi}b')))
8
9     asal = set()
10    while True:
11        yeni = set()
12        kullanilan = set()
13        liste = list(kapsayanlar)
14        for i in range(len(liste)):
15            for j in range(i + 1, len(liste)):
16                mi, bi = liste[i]
17                mj, bj = liste[j]
18                # İki kapsayan yalnızca tek bitte mi farklılaşıyor?
19                fark = [k for k in range(degisken_sayisi)
20                        if bi[k] != bj[k]]
21                if len(fark) == 1:
22                    # Farklı biti '-' ile değiştir, birleştir
23                    yeni_bit = list(bi)
24                    yeni_bit[fark[0]] = '-'
25                    yeni.add((mi | mj, ''.join(yeni_bit)))
26                    kullanilan.add(i)
27                    kullanilan.add(j)
28                # Birleşemeyenler asal kapsayandır
29                for i, k in enumerate(liste):
30                    if i not in kullanilan:
31                        asal.add(k)
32                if not yeni: # Yeni birleşme kalmadıysa dur
33                    break
34            kapsayanlar = yeni
35
36    # --- Aşama 2: Temel asal kapsayanları seçme ---
37    kalan = set(birler) # Henüz kapsanmamış mini terimler
38    sonuc = []
39    for mini_terimler, desen in asal:
40        # Bu kapsayanın özel olarak kapsadığı terimler var mı?
41        diger = set()
42        for mt2, d2 in asal:
43            if d2 != desen:
44                diger |= (mt2 & kalan)
45            ozel = (mini_terimler & kalan) - diger
46            if ozel: # Temel asal kapsayan - seçilmeli
47                sonuc.append(desen)
48                kalan -= mini_terimler
49    # Kalan terimleri kapsamak için ek kapsayanlar
50    for mini_terimler, desen in asal:
51        if kalan and (mini_terimler & kalan):
52            sonuc.append(desen)
53            kalan -= mini_terimler
54    return sonuc

```

```

55
56 # --- Örnek:  $F(a,b,c,d) = \Sigma(0,1,2,5,6,7,8,9,10,14)$  ---
57 sonuc = quine_mccluskey(4, [0, 1, 2, 5, 6, 7, 8, 9, 10, 14])
58 print("Asal kapsayanlar:")
59 for desen in sonuc:
60     degiskenler = 'abcd'
61     terim = []
62     for i, bit in enumerate(desen):
63         if bit == '1': terim.append(degiskenler[i])
64         elif bit == '0': terim.append(degiskenler[i] + '~')
65     print(f"{desen}UU->UU{' '.join(terim)}")
66 # Çıktı:
67 # --10 -> cd' ( $c\bar{d}$ )
68 # -00- -> b'c' ( $\bar{b}\bar{c}$ )
69 # 01-1 -> a'bd ( $\bar{a}bd$ )

```

Kod B.1: Quine–McCluskey algoritması: Python ile sıfırdan gerçekleştirme

Çıktıdaki her desende “1” ilgili değişkenin kendisini, “0” tümleyenini, “–” ise o değişkenin sonuca etki etmediğini gösterir. Program, Örnek B.8’deki elle çözdüğümüz sonucun aynısını üretir: $F = \bar{b}\bar{c} + c\bar{d} + \bar{a}bd$.

B.5 Birleşik Yapı Taşları

Şimdiye kadar öğrenilen yöntemler (doğruluk tablosu, Boole cebri, Karno haritası, Quine–McCluskey) birkaç girişli fonksiyonları sadeleştirmek için güçlü araçlardır. Ancak gerçek bir işlemci devreler bu ölçeğin çok ötesindedir. Örneğin 32 bitlik bir AMB’nin (Aritmetik Mantık Birimi) yalnızca iki işleneni için $32 + 32 = 64$ bit veri girişi, bir de işlem seçim sinyalleri gerekir; çıkışı ise en az 33 bittir. Böyle bir devrenin doğruluk tablosu 2^{64} ’ün üzerinde satırdan oluşur ki bu sayı evrendeki tahmini atom sayısına yakındır. Karno haritası çizmek ya da tüm mini terimleri listelemek söz konusu bile olamaz.

Bu gerçek, sayısal tasarımda **modüler yaklaşımı** zorunlu kılar. Karmaşık bir devre sıfırdan ve tek parça olarak tasarlanmaz; bunun yerine, iyi tanımlanmış ve tekrar tekrar kullanılabilen küçük yapı taşları tasarlanır, bu taşlar birbirine eklenerek büyük sistemler oluşturulur. Tıpkı bir yazılımcının her programda sıralama algoritmasını yeniden yazmak yerine hazır bir kütüphane çağırması gibi, donanım tasarımcısı da toplayıcı, kod çözücü ve çoklayıcı gibi temel birimleri birer modül olarak kullanır. Bu yaklaşım hem mühendislik zamanından büyük tasarruf sağlar hem de tasarımı anlaşılır, sınanabilir ve bakımı kolay parçalara böler.

Bu bölümde, birleşik devrelerin en yaygın yapı taşları olan toplayıcı, kod çözücü ve çoklayıcı devreleri ele alınmaktadır. Bu devreler, işlemcinin veriyolunda, denetim biriminde ve aritmetik biriminde temel bileşenler olarak kullanılır.

B.5.1 Toplayıcılar

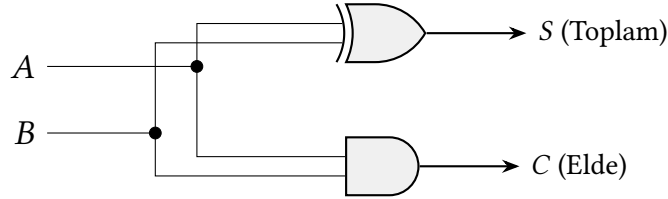
Toplama işlemi, bilgisayarın en temel aritmetik işlemidir; çıkarma, adres hesaplama ve program sayacı güncelleme gibi pek çok işlem de toplamaya dayanır. Ek A’da ikilik tabanda toplama işleminin nasıl yapıldığı gösterilmişti. Bu bölümde aynı işlemi donanım düzeyinde gerçekleştiren toplayıcı devrelerinin genel yapısı tanıtılmaktadır; ayrıntılı devre tasarımları ve hızlandırma teknikleri Bölüm 5’te ele alınmıştır.

Yarım Toplayıcı

Yarım toplayıcı (İngilizcesi: *half adder*), iki tek bitlik girişi (A ve B) toplayarak bir toplam biti (S) ve bir elde biti (C) üretir:

$$S = A \oplus B, \quad C = A \cdot B$$

Toplam çıkışı Dışlayan VEYA kapısıyla, elde çıkışı VE kapısıyla gerçekleştirilir. “Yarım” olarak adlandırılmasının nedeni, bir alt basamaktan gelen elde girişini kabul edememesidir; bu nedenle tek başına yalnızca en düşük anlamlı bitin toplanmasında kullanılabilir.



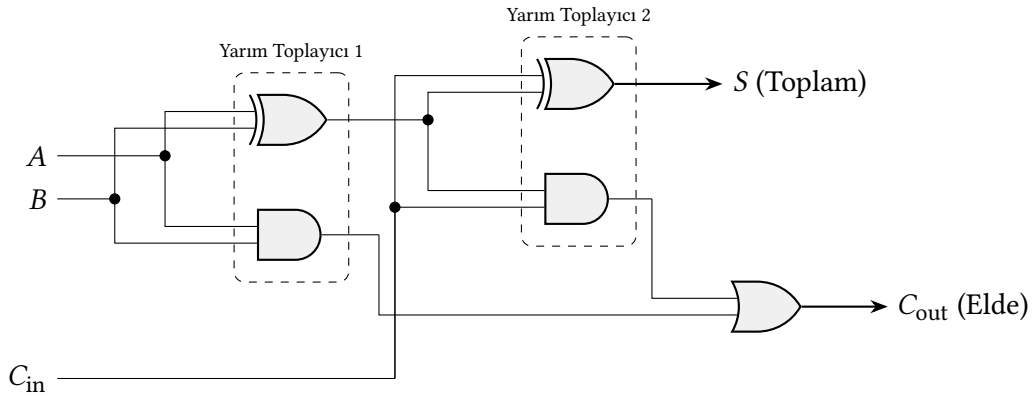
Şekil B.7: Yarım toplayıcı devresi: bir Dışlayan VEYA ve bir VE kapısından oluşur

Tam Toplayıcı

Tam toplayıcı (İngilizcesi: *full adder*), iki giriş bitinin (A ve B) yanı sıra bir önceki basamaktan gelen elde girişini (C_{in}) de kabul eder:

$$S = A \oplus B \oplus C_{in}, \quad C_{out} = A \cdot B + C_{in} \cdot (A \oplus B)$$

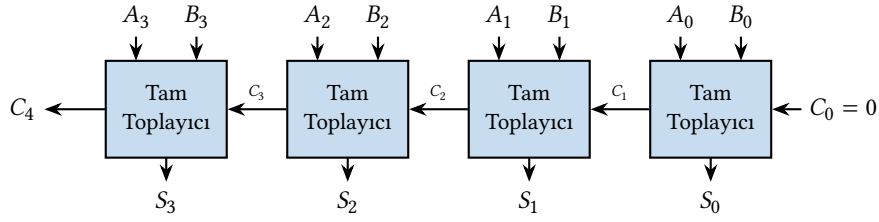
İki yarım toplayıcı ve bir VEYA kapısı birleştirilerek gerçekleştirilebilen bu devre, çok basamaklı toplayıcılarda temel yapı taşı olarak zincirleme kullanılır.



Şekil B.8: Tam toplayıcı devresi: iki yarım toplayıcı ve bir VEYA kapısından oluşur

Dalgalı Elde Toplayıcı

n bitlik iki sayıyı toplamak için n adet tam toplayıcı zincirleme bağlanır: her basamağın elde çıkışı (C_{out}) bir sonraki basamağın elde girişine (C_{in}) bağlanır. Bu yapıya **dalgalı elde toplayıcı** (İngilizcesi: *ripple carry adder*) denir; elde sinyali en düşük bittten en yüksek bite doğru “dalga” hâlinde yayılır. Yapısı basittir ancak gecikme bit sayısı ile doğru orantılı ($O(n)$) olduğundan, geniş sözcük boyutlarında (32 veya 64 bit) elde ileriye bakma (İngilizcesi: *carry lookahead*) gibi daha hızlı tasarımlar tercih edilir.



Şekil B.9: 4 bitlik dalgali elde toplayıcı: elde sinyali en az anlamlı bitten (sağ) en anlamlı bite (sol) doğru yayılır

Örnek B.9

$A = 1011_2$ (11_{10}) ile $B = 0110_2$ (6_{10}) sayılarını 4 bitlik dalgali elde toplayıcı ile toplayın.

Çözüm:

Her basamak en az anlamlı bitten (LSB) başlayarak hesaplanır:

Basamak	A_i	B_i	C_i (Giren Elde)	S_i / C_{i+1} (Çıkan Elde)
0 (LSB)	1	0	0	$S_0 = 1, C_1 = 0$
1	1	1	0	$S_1 = 0, C_2 = 1$
2	0	1	1	$S_2 = 0, C_3 = 1$
3 (MSB)	1	0	1	$S_3 = 0, C_4 = 1$

Sonuç: $C_4S_3S_2S_1S_0 = 1\ 0001_2 = 17_{10}$. Doğrulama: $11 + 6 = 17$. ✓

B.5.2 Kod Çözücü

Kod çözücü (İngilizcesi: *decoder*), n bitlik bir ikilik girişi 2^n çıkıştan tam olarak birine yönlendiren bir devredir. Her çıkış hattı, giriş değişkenlerinin belirli bir mini terimine karşılık gelir; dolayısıyla herhangi bir anda yalnızca bir çıkış etkin (1) olur.

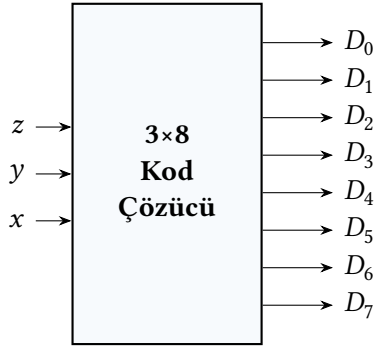
Kullanım alanları

Kod çözücüler çağdaş işlemci ve bellek tasarımının birçok yerinde temel yapı taşıdır:

- **Bellek adres kod çözücüsü:** n bitlik bellek adresini, 2^n hücreden yalnızca birini etkinleştiren seçim sinyaline dönüştürür. Büyük belleklerde tek bir kod çözücü yerine çok katmanlı (hiyerarşik) bir yapı kullanılır; bir katman satırı, diğeri sütunu seçer (Bölüm 8).
- **Buyruk kod çözücüsü:** RISC-V'te 32 bitlik buyruğun 7 bitlik *opcode* alanı bir kod çözücüye girer; çıkış, buyruğun türünü (R, I, S, B, U, J) ve denetim sinyallerini (ALU işlemi, yazmaç yazma izni, bellek yazma/okuma) belirler (Bölüm 6).
- **Yazmaç öbeği yazma kapısı:** 32 yazmaçlı öbekte yazma yapılırken, 5 bitlik hedef yazmaç numarasını (*rd*) 32 yazmaçtan birinin yazma girişine yönlendiren 5×32 kod çözücü kullanılır.
- **Yedi-parçalı gösterge sürücüsü:** Dört bitlik ikilik değeri, yedi LED'in hangilerinin yanacağını belirleyen çıkışlara dönüştürür (0–9 onluk veya 0–F onaltılık gösterim için).
- **Ters-çoklayıcı (demultiplexer):** Etkinleştirme girişi veri olarak kullanıldığında, kod çözücü tek bir giriş verisini seçilen bir çıkışa aktaran ters-çoklayıcıya dönüşür.

3×8 Kod Çözücü

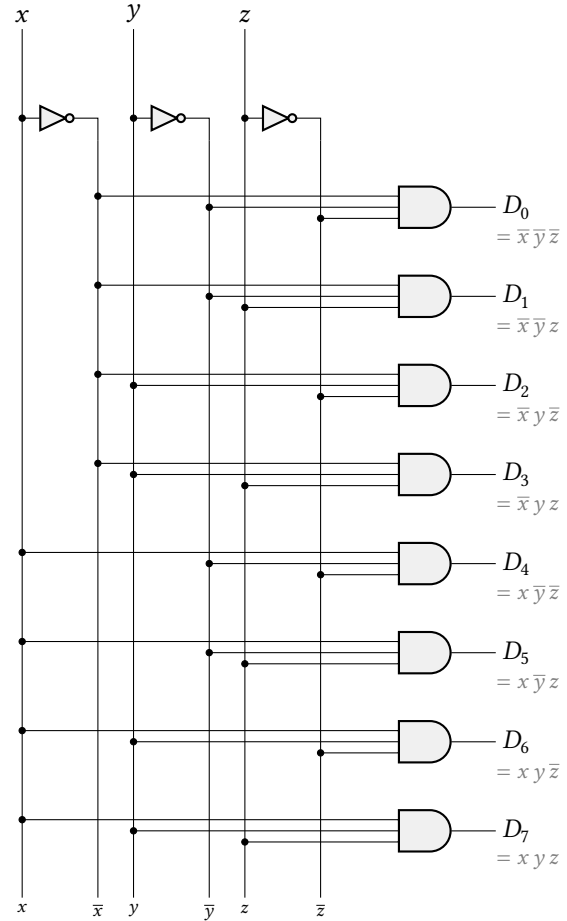
3×8 kod çözücü, üç bitlik girişi $2^3 = 8$ çıkıştan tam olarak birine yönlendirir. Şekil B.10 bu devrenin (a) blok diyagramını, (b) doğruluk tablosunu ve (c) kapı düzeyindeki gerçekleşmesini bir arada gösterir. Üç giriş (x, y, z) DEĞİL kapılarıyla altı dikey hatta yayılır ($x, \bar{x}, y, \bar{y}, z, \bar{z}$); sekiz üç-girişli VE kapısı bu hatlardan uygun üçünü alarak $D_0 = \bar{x}\bar{y}\bar{z}$ 'ten $D_7 = xyz$ 'ye kadar tüm mini terimleri üretir. Doğruluk tablosundan görüldüğü gibi, herhangi bir giriş bileşiminde yalnızca bir çıkış 1 olur.



(a) Blok diyagram

x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

(b) Doğruluk tablosu



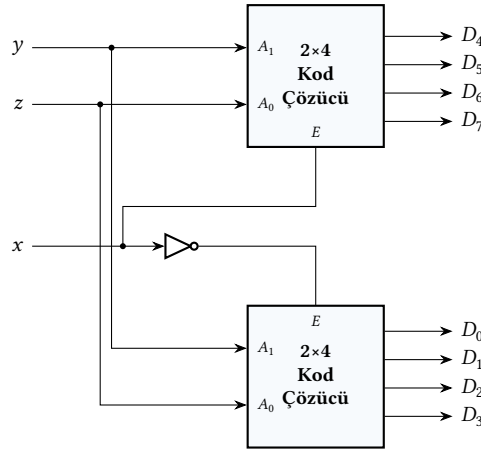
(c) Kapı düzeyinde gerçekleştirme

Şekil B.10: 3×8 kod çözücü: (a) blok diyagram, (b) doğruluk tablosu, (c) kapı düzeyinde gerçekleştirme

Etkinleştirme girişi ve hiyerarşik tasarım

Çoğu kod çözücünün bir *etkinleştirme* (enable) girişi bulunur. Etkinleştirme 0 olduğunda bütün çıkışlar 0'a zorlanır; 1 olduğunda kod çözücü normal çalışır. Bu giriş iki amaca hizmet eder: büyük kod çözücülerini küçüklerden birleştirmek ve kod çözücüye belirli zaman aralıklarında devre dışı bırakmak. Örneğin 3×8 kod çözücü, iki 2×4 kod çözücüyle kurulabilir (Şekil B.11): en anlamlı giriş biti doğrudan bir 2×4'ün etkinleştirme girişine, tümleyeni de diğerine verilir. Bit 0 iken alt kod çözücü D_0-D_3 'ü üretir; bit 1 iken üst kod çözücü D_4-D_7 'yi üretir. Aynı yöntemle

32 yazmaçlık öbek için 5×32 kod çözücü, dört adet 3×8 ve bir adet 2×4 ile elde edilir.



Şekil B.11: 3×8 kod çözücünün iki 2×4 kod çözücü ve bir DEĞİL kapısıyla oluşturulması; x etkinleştirme sinyali hangi alt kod çözücünün etkin olduğunu seçer

Kod çözücü ile Boole fonksiyonu gerçekleştirme

Kod çözücünün her çıkışı bir mini terime denk geldiğinden, n -değişkenli herhangi bir Boole fonksiyonu, bir $n:2^n$ kod çözücü ve tek bir VEYA kapısıyla iki katlı bir devre olarak gerçekleştirilebilir. Yöntem basittir: fonksiyonun SÇT (standart çarpımlar toplamı, İngilizcesi: *sum of products*) biçimindeki mini terimlerine karşılık gelen kod çözücü çıkışları bir VEYA kapısında toplanır. Aynı kod çözücünün çıkışlarını birden çok VEYA kapısıyla paylaştırarak tek geçişte birden çok fonksiyon da üretilebilir; bu yaklaşım küçük ölçekli tümleşim (SSİ) döneminde birleşik mantık tasarımının en hızlı yollarından biriydi ve bugün hâlâ salt okunur bellek (ROM) tabanlı fonksiyon tabloları ile aynı ilkeye dayanır.

Örnek B.10

Üç değişkenli iki fonksiyonu tek bir 3×8 kod çözücü ile gerçekleyin:

$$F_1(x, y, z) = \Sigma(1, 2, 4, 7), \quad F_2(x, y, z) = \Sigma(3, 5, 6, 7).$$

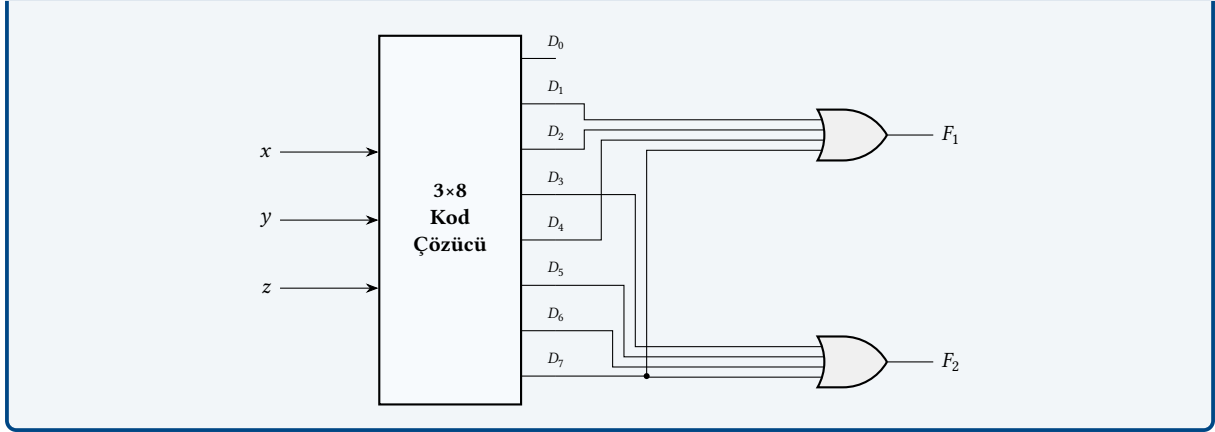
Çözüm. 3×8 kod çözücünün çıkışları D_0 – D_7 sırasıyla $\overline{x}\overline{y}\overline{z}$, $\overline{x}\overline{y}z$, ..., xyz mini terimlerini üretir. F_1 için 1, 2, 4 ve 7 indisli mini terimler bir VEYA kapısında toplanır:

$$F_1 = D_1 + D_2 + D_4 + D_7.$$

F_2 için benzer biçimde:

$$F_2 = D_3 + D_5 + D_6 + D_7.$$

Her iki fonksiyon tek bir 3×8 kod çözücü ve iki adet 4-girişli VEYA kapısıyla üretilir (aşağıdaki şekil). D_7 çıkışının her iki VEYA kapısına da bağlandığına dikkat edin; aynı kod çözücüü paylaşmak, tek fonksiyon için ayrı bir SÇT devresi kurmaya kıyasla kapı sayısını önemli ölçüde azaltır.



RISC-V çekirdeğinde örnekler

RISC-V işlemcisinin denetim ve veriyolu birimlerinde kod çözücüler birden çok yerde tekrarlanır:

- **Buyruk türü çözme:** 32 bitlik buyruğun opcode alanı (`instruction[6×0]`) etkin opcode'lar için seyrek bir kod çözücü ağına girer. Çıkış buyruk türünü (R, I, S, B, U, J) ve birinci kademe denetim sinyallerini (ALUSrc, MemRead, MemWrite, RegWrite, Branch, MemtoReg) belirler. `funct3` ve `funct7` alanları ise alt çözücülere yönlendirilerek aritmetik-mantık işlemini ve dallanma koşulunu seçer (Bölüm 6).
- **Yazmaç öbeği yazma kapısı:** R-tipi ve I-tipi buyrukların hedef alanı `rd` 5 bittir. 5×32 kod çözücü, `rd` değerine karşılık gelen yazmacın yazma izni hattını etkinleştirir; geri kalan 31 yazmacın yazma izni 0'da kalır. Buyruk yazmaca yazmıyorsa (S ve B tipleri gibi) RegWrite sinyali tüm çözücüye devre dışı bırakır.
- **ALU işlem seçici:** ALU denetim birimi 3–4 bitlik bir işlem kodu üretir; bu kod ALU içinde küçük bir kod çözücüye girer. Kod çözücünün her çıkışı toplama, çıkarma, mantıksal VE/VEYA, kaydırma, karşılaştırma alt birimlerinden birinin çoklayıcı seçim sinyalini sürer.
- **Önbellek küme seçimi:** L1 önbellekte etiket karşılaştırması yapılmadan önce, fiziksel adresin orta bitleri (index alanı) bir kod çözücüye verilerek ilgili önbellek kümesinin yolları okunur. Aynı yapı TLB ve sayfa tablosu önbelleklerinde de geçerlidir (Bölüm 8).
- **CSR adres çözme:** Buyruğun 12 bitlik CSR alanı, makine, denetleyici ve kullanıcı kiplerine ait yüzlerce denetim ve durum yazmacından birini seçen seyrek bir kod çözücü ağına girer.

B.5.3 Kodlayıcı

Kodlayıcı (İngilizcesi: *encoder*), 2^n veri girişi içinden 1 değerini taşıyan birinin ikilik indeksini n bit çıkışta üreten devredir. Kod çözücünün tersidir: kod çözücü n bit kodu açarak 2^n çıkıştan birinde 1 üretirken, kodlayıcı 2^n giriş arasında etkin olanın sıra numarasını n bitlik koda sıkıştırır. Bu yönüyle bir tek-sıcak (*one-hot*) gösterimi ikilik koda dönüştürür.

Kullanım alanları

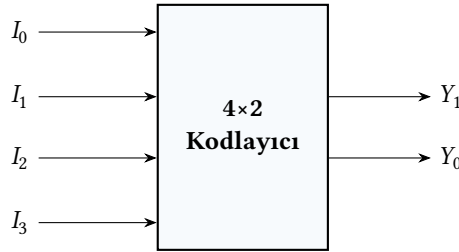
- **Kesme denetleyicisi:** Birden çok aygıt aynı anda işlemciye kesme isteği gönderebilir; öncelikli kodlayıcı bu istekler arasından en yüksek öncelikli olanı seçer ve indeksini

kesme vektör tablosuna geçirir. RISC-V’te bu işlevi PLIC (*Platform-Level Interrupt Controller*) görür.

- **Klavye matrisi:** Tuş takımına basılan tuşun satır-sütun seçimi sonucu beliren tek-sıcak işaret, kodlayıcıyla tuşun ikilik koduna çevrilir.
- **Mikrokod adres oluşturma:** Buyruk türünü gösteren bayrakların tek-sıcak gösterimi öncelikli kodlayıcı ile sıralanır; çıkış mikrokod ROM’una adres olarak verilir.
- **Bant genişliği indirgemesi:** İletişim hatlarında ayrık değerli bir sinyalin tek-sıcak gösterimi $\log_2 n$ bite sıkıştırılarak gönderilir, alıcı tarafta kod çözücü ile geri açılır.

4×2 Kodlayıcı

4×2 kodlayıcı, dört giriş (I_0, I_1, I_2, I_3) içinden 1 olanın indeksini iki bitlik bir kodda (Y_1Y_0) verir. Aynı anda yalnızca tek bir girişin 1 olduğu varsayılır. Şekil B.12’de blok diyagramı, Tablo B.5’de ise doğruluk tablosu verilmiştir.



Şekil B.12: 4×2 kodlayıcının blok diyagramı: dört girişten 1 olan birinin indeksi iki bitlik Y_1Y_0 kodunda üretilir

Tablo B.5: 4×2 kodlayıcının doğruluk tablosu (aynı anda yalnızca bir girişin 1 olduğu kabulü altında)

I_3	I_2	I_1	I_0	Y_1	Y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Tablodan iki çıkış da iki-girişli VEYA kapıları ile üretilebilir:

$$Y_0 = I_1 + I_3, \quad Y_1 = I_2 + I_3.$$

I_0 girişi hiçbir çıkış denkleminde yer almaz; bu, I_0 etkin olduğunda çıkışın doğal olarak 00 değerini alacağını yansıtır. Bu durum bir sınırlama getirir: çıkış 00 iken I_0 ’in mi 1 olduğu, yoksa hiçbir girişin 1 olmadığı mı belirsizdir. Sorunu gidermenin yolu, geçerlilik sinyali eklemek ve girişler arasında öncelik kurmaktır.

Öncelikli kodlayıcı

Yukarıdaki 4×2 kodlayıcı, birden fazla giriş aynı anda 1 olduğunda hatalı çıkış üretir. Örneğin $I_1 = I_2 = 1$ kombinasyonunda VEYA kapıları $Y_1Y_0 = 11$ verir; oysa bu kod $I_3 = 1$ durumuna karşılık gelir. Çözüm, *öncelikli kodlayıcı* (*priority encoder*) yapısıdır: girişler bir öncelik sırasına

dizilir, en yüksek öncelikli etkin giriş çıkışı belirler, ondan düşük öncelikli girişler önemsenmez. Çıkışa ayrıca bir geçerlilik (V) sinyali eklenir: hiçbir giriş 1 değilse $V = 0$ olur ve kod alanı geçersiz sayılır.

8×3 öncelikli kodlayıcı sekiz aygıttan eşzamanlı kesme istekleri arasından en yüksek öncelikli olanı seçer; üç bit kod ile geçerlilik biti, kesme denetleyicisinin işlemciye verdiği temel çıkıştır. Çıkış denklemleri sıradan kodlayıcıya göre ek “ondan düşük öncelikliler 0 olmalı” koşulları içerir; bu nedenle her bit için VE-VEYA katmanı genişler. Tasarım uygulamada ya hazır bir kütüphane bileşeni (74148 ailesi gibi) ya da bir HDL betimi ile gerçekleştirilir.

RISC-V çekirdeğinde örnekler

İşlemci içinde kodlayıcılar (genelde öncelikli kodlayıcılar) tek-sıcak göstergeleri ikilik koda dönüştüren her noktada karşımıza çıkar:

- **Kesme nedeni belirleme:** Eşzamanlı gelen kesme ve istisna istekleri arasından en yüksek öncelikli olanı seçilip kodu `mcause CSR`'sine yazılır. RISC-V'te bu işi dış kaynaklar için PLIC, çekirdek-yerel kaynaklar için CLIC yürütür; her ikisi de öncelikli kodlayıcı çekirdeği barındırır.
- **Çağrışimli önbellek hit kodlaması:** k yollu çağrışimli bir önbellekte etiket karşılaştırması k adet hit/miss biti üretir; en çok biri 1 olabilir. Bu tek-sıcak vektör bir kodlayıcıdan geçirilerek hit eden yolun $\log_2 k$ bitlik kodu üretilir; veri çoklayıcısının seçim sinyali olur.
- **Sırasız yürütümde planlayıcı:** Birden çok buyruk aynı anda yürütülmeye hazır olabilir. Planlayıcı, hazır buyrukların tek-sıcak vektörünü öncelikli kodlayıcıdan geçirerek bir sonraki çevrimde gönderilecek buyruğun kuyruk indeksini belirler. Aynı mantık, serbest yazmaç havuzundan boş bir fiziksel yazmaç seçmekte de kullanılır.
- **Dallanma öngörü tablosu hit kodlaması:** BTB (*Branch Target Buffer*) çok yollu yapıda kurulduğunda, hangi yolun hit yaptığı bilgisini öncelikli kodlayıcı üretir; çıkış, hedef adresi okuyacak çoklayıcının seçim hattını sürer.
- **Kayan-noktalı istisna sıralayıcı:** IEEE 754 standardı birden çok bayrağın aynı anda kalkmasına izin verir (geçersiz işlem, sıfıra bölme, taşma, alttan taşma, yuvarlanma). Öncelikli kodlayıcı, bu bayraklar arasından en öncelikli istisnayı seçip kodlar; sonuç `fcsr CSR`'sine ve istisna mantığına aktarılır.

B.5.4 Çoklayıcı

Çoklayıcı (İngilizcesi: *multiplexer*, kısaca MUX), 2^n veri girişi arasından birini n bitlik denetim (seçim) sinyaline göre seçerek tek bir çıkışa aktaran devredir. Tekleyicinin (*demultiplexer*) tersidir: tekleyici tek bir girişi seçilen çıkışa dağıtırken, çoklayıcı birçok giriş arasından seçilen birini tek bir çıkışta toplar.

Kullanım alanları

Çoklayıcılar işlemci veriyolu tasarımının her yerinde bulunur:

- **ALU ikinci giriş seçimi:** RISC-V'te ALU'nun ikinci girişi ya bir yazmaçtan (`rs2`) ya da buyruğa gömülü anlık değerden (*immediate*) alınır. Buyruk türüne göre seçim yapan bir 2×1 çoklayıcı, denetim biriminin ürettiği `ALUSrc` sinyaliyle sürülür (Bölüm 6).
- **Geri yazım kaynağı seçimi:** Yazmaç öbeğine yazılacak değer, ALU sonucu, bellekten okunan veri veya sonraki buyruğun adresi (`jal/jalr` için) olabilir. Bu kaynaklar arasında

seçim yapan çoklayıcı denetim biriminin MemtoReg türündeki sinyaliyle sürülür.

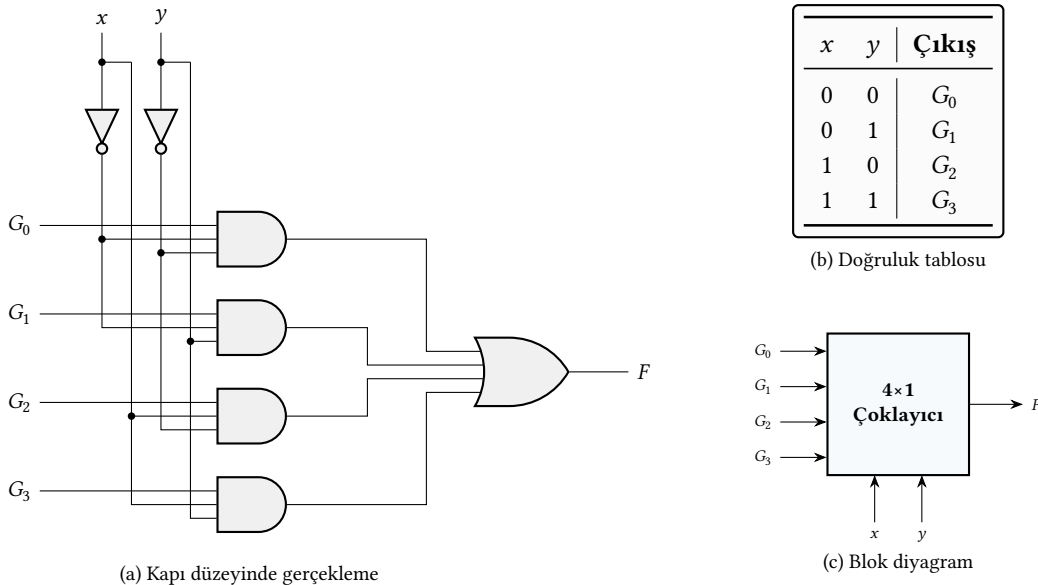
- **Yazmaç öbeği okuma kapısı:** 32 yazmaçlı öbekte kaynak yazmacını seçmek için 5 bitlik kaynak numarası ($rs1, rs2$) 32×1 çoklayıcıları sürer. Çoğu işlemci aynı çevrimde iki eşzamanlı okuma için iki ayrı 32×1 çoklayıcı bulundurur.
- **Önbellek satırı içinden bayt/sözcük seçimi:** 64 baytlık bir önbellek satırından, okuma adresinin alt bitlerine göre belirli bir bayt veya sözcüğü seçmek için çoklayıcılar kullanılır.
- **Paralel-seri dönüşüm:** n bitlik veriyi tek bir çıkış hattına sıralı göndermek için $n:1$ çoklayıcı kullanılır; seçim hatları bir sayaçtan beslenir. JTAG, UART ve diğer çevre birimi iletişim protokollerinde temel bir yapı taşıdır.
- **Ortak veriyoluna kaynak seçimi:** Aynı veriyoluna birden çok kaynak bağlı olduğunda, bir çoklayıcı hangi kaynağın veriyolunu süreceğini seçer; bu yaklaşım, üç durumlu tampon yerine özellikle kırmık (die) içi veriyollarında yaygındır.

4×1 Çoklayıcı

4×1 çoklayıcı dört veri girişi (G_0, G_1, G_2, G_3) arasından, iki bitlik seçim sinyalinin (x, y) gösterdiği birini çıkışa aktarır. Şekil B.13’de devrenin kapı düzeyindeki gerçekleşi verilmiştir. Her seçim kombinasyonu bir mini terim oluşturur: $\bar{x}\bar{y}$, $\bar{x}y$, $x\bar{y}$, xy . Devre dört üç-girişli VE kapısıyla bu mini terimleri ilgili veri girişine zorlar; her VE kapısı yalnızca seçim sinyali kendi mini terimine eşleştiğinde 1 üretir ve veri girişini geçirir. Dört VE kapısının çıkışı dört-girişli bir VEYA kapısında toplanır:

$$F = \bar{x}\bar{y}G_0 + \bar{x}yG_1 + x\bar{y}G_2 + xyG_3.$$

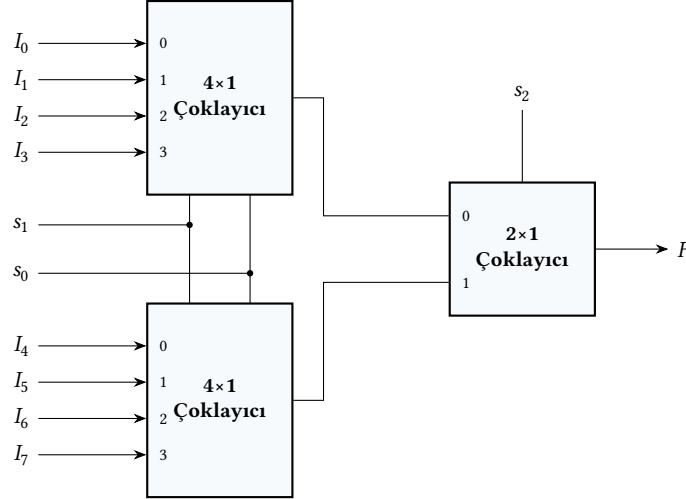
Daha büyük çoklayıcılar (8×1, 16×1, ...) aynı yapıyı genişletir: n bitlik seçim sinyali 2^n adet $(n+1)$ -girişli VE kapısı sürer ve sonuçlar tek bir VEYA kapısında birleşir. Pratikte bu doğrudan kapı düzeyi yerine bir sonraki bölümde göreceğimiz hiyerarşik birleştirme yöntemiyle kurulur.



Şekil B.13: 4×1 çoklayıcı: (a) kapı düzeyinde gerçekleştirilmesi, (b) doğruluk tablosu, (c) blok diyagramı

Hiyerarşik çoklayıcı

Büyük çoklayıcılar küçüklerden kurulur. 8×1 bir çoklayıcı iki adet 4×1 ve bir adet 2×1 çoklayıcıyla gerçekleştirilir (Şekil B.14): en anlamlı seçim biti s_2 , son katmandaki 2×1 çoklayıcıyı sürer; alt iki bit $s_1 s_0$ iki 4×1 çoklayıcının seçim girişlerine aynı anda verilir. Aynı yaklaşım, 32×1 veya 64×1 gibi büyük çoklayıcıları da kaskat biçiminde (örneğin on altı 4×1 + bir 4×1) kurmaya olanak tanır; böylece her katman $\log_2(\text{boyut})$ kapı gecikmesine katkıda bulunur.



Şekil B.14: 8×1 çoklayıcının iki 4×1 ve bir 2×1 çoklayıcı ile oluşturulması; $s_1 s_0$ alt çoklayıcıları, s_2 son katmandaki 2×1 çoklayıcıyı sürer

Çoklayıcı ile Boole fonksiyonu gerçekleştirme

Çoklayıcı, her Boole fonksiyonunu doğrudan üretebilen esnek bir yapıdır. İki yöntem kullanılır:

Yöntem 1: $2^n:1$ çoklayıcı ile n değişkenli fonksiyon. Değişkenleri seçim hatlarına bağlayın; her bir mini terim için fonksiyon değerini (0 veya 1), o mini terime karşılık gelen veri girişine sabit olarak uygulayın. Veri girişlerinin ataması doğruluk tablosundan doğrudan okunur.

Yöntem 2: $2^{n-1}:1$ çoklayıcı ile n değişkenli fonksiyon (küçültme). Shannon ayrışımı gereği, bir n -değişkenli fonksiyon $(n-1)$ -değişkenli iki yarı fonksiyonun ağırlıklı toplamı olarak yazılabilir:

$$F(x_1, \dots, x_n) = \overline{x_n} \cdot F_0(x_1, \dots, x_{n-1}) + x_n \cdot F_1(x_1, \dots, x_{n-1}),$$

burada F_0 ve F_1 , x_n yerine sırasıyla 0 ve 1 konulduğunda elde edilen fonksiyonlardır. $(n-1)$ değişkeni seçim hatlarına bağlarsak, her veri girişine F_0 veya F_1 'in ilgili mini terimindeki değeri gelir; bu değer 0, 1, x_n veya $\overline{x_n}$ 'den biridir. Böylece boyutu yarıya inmiş bir çoklayıcıyla aynı fonksiyon gerçekleştirilir. Yöntem, büyük fonksiyonlar için donanım maliyetini önemli ölçüde azaltır.

Örnek B.11

4×1 çoklayıcı kullanarak iki değişkenli $F(x, y) = \Sigma(1, 2, 3)$ fonksiyonunu gerçekleyin (Yöntem 1).

Çözüm. *Adım 1.* Doğruluk tablosu çıkarılır:

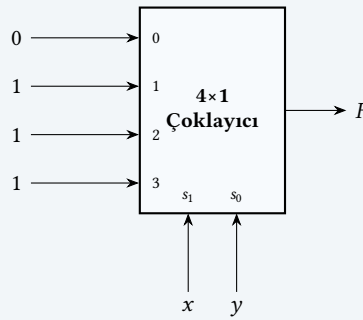
x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

Adım 2. Seçim hatları değişkenlere bağlanır: $s_1 = x$ (en anlamlı), $s_0 = y$ (en az anlamlı).

Adım 3. Veri girişleri fonksiyon değerleriyle atanır:

$$G_0 = 0, \quad G_1 = 1, \quad G_2 = 1, \quad G_3 = 1.$$

Seçim hatları $xy = 00$ iken $G_0 = 0$ çıkışa aktarılır; $xy = 01$ iken $G_1 = 1$; $xy = 10$ iken $G_2 = 1$; $xy = 11$ iken $G_3 = 1$. Devre, F fonksiyonunu mantık kapısı kullanmadan yalnızca sabit 0 ve 1 girişleriyle gerçekleştirir (aşağıdaki şekil).

**Örnek B.12**

4×1 çoklayıcı kullanarak üç değişkenli $F(x, y, z) = \Sigma(1, 3, 5, 6)$ fonksiyonunu gerçekleyin (Yöntem 2, küçültme).

Çözüm. Yöntem 1 uygulansaydı 8×1 çoklayıcı gerekecekti; küçültme yöntemiyle yarı boyuttaki 4×1 çoklayıcı yeterli olacak.

Adım 1. Doğruluk tablosu çıkarılır ve (x, y) sütununa göre yeniden düzenlenir:

x	y	$F(z=0)$	$F(z=1)$	Veri girişi
0	0	0	1	$G_0 = z$
0	1	0	1	$G_1 = z$
1	0	0	1	$G_2 = z$
1	1	1	0	$G_3 = \bar{z}$

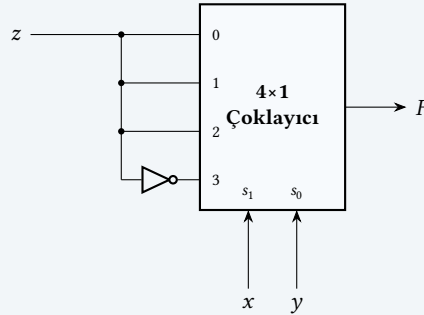
Adım 2. Her satırdaki $(z=0, z=1)$ çifti incelenir:

- $(0, 1)$ örüntüsü z sinyalinin kendisine denktir.
- $(1, 0)$ örüntüsü $\bar{z}$ 'ye denktir.
- $(0, 0)$ sabit 0'a, $(1, 1)$ sabit 1'e denk gelir.

Adım 3. Seçim hatlarına $s_1 = x$ ve $s_0 = y$ atanır. Veri girişleri önceki tablodan okunur:

$$G_0 = z, \quad G_1 = z, \quad G_2 = z, \quad G_3 = \bar{z}.$$

Devre, bir 4×1 çoklayıcı ve z sinyalini tümleyen bir DEĞİL kapısıyla kurulur (aşağıdaki şekil). Yöntem 1'in gerektireceği 8×1 çoklayıcıya göre donanım neredeyse yarıya inmiştir. Fonksiyonun değişken sayısı arttıkça bu kazanç katlanarak büyür.



B.5.5 Tekleyici

Tekleyici (İngilizcesi: *demultiplexer*, kısaca DEMUX), tek bir veri girişini n bitlik seçim sinyaline göre 2^n çıkıştan birine yönlendiren devredir. Çoklayıcının tersidir: çoklayıcı 2^n giriş arasından birini seçerek tek çıkışa aktarırken, tekleyici tek bir girişi seçilen çıkışa dağıtır. Et-kinleştirme girişli kod çözücü ile yapısal olarak özdeştir; bu ilişki Bölüm 8'te bellek satır-sütun seçim mantığında karşımıza çıkar.

Kullanım alanları

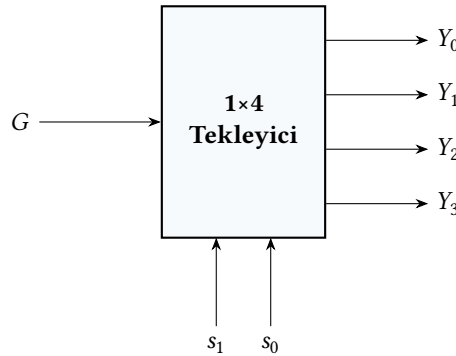
- **Veriyolu dağıtımı:** Tek bir veri kaynağından farklı zamanlarda farklı aygıtlara değer gönderilirken, tekleyicinin seçim girişi hangi aygıtın o anda veri alacağını belirler.
- **Çift yönlü iletişim:** Çoklayıcı ile tekleyici çiftleri kullanılarak tek bir hat üzerinden zaman çoğullamalı çift yönlü veri akışı kurulur; çoklayıcı gönderici tarafta, tekleyici alıcı tarafta yer alır.
- **Bellek yazma yönlendirmesi:** Yazılacak veri tek kaynaktan gelir; hedef hücrenin adres çözücüsü, veriyi seçilen hücrenin yazma kapısına tekleyici davranışıyla aktarır.
- **Paket yönlendirme:** Ağ anahtarlarında gelen paketin başlık alanı seçim sinyali olarak kullanılır; veri yükü doğru çıkış noktasına iletilir.

1×4 Tekleyici

1×4 tekleyici tek bir veri girişini (G) iki bitlik seçim sinyaline (s_1, s_0) göre dört çıkıştan birine (Y_0-Y_3) yönlendirir. Seçili olmayan çıkışlar 0'a zorlanır:

$$Y_i = \begin{cases} G, & s_1 s_0 = i \\ 0, & \text{aksi halde} \end{cases} \quad i = 0, 1, 2, 3.$$

Şekil B.15'te blok diyagramı verilmiştir. Devre dört adet üç-girişli VE kapısıyla gerçekleştirir; her VE kapısının iki girişi seçim bitlerinin uygun değer ya da tümleyenini, üçüncü girişi de veri sinyalini taşır. Seçim bitlerinin tümleyenleri için iki DEĞİL kapısı yeterlidir.



Şekil B.15: 1×4 tekleyicinin blok diyagramı: G verisi s_1s_0 seçim sinyaline göre Y_0 – Y_3 çıkışlarından birine yönlendirilir

Kod çözücünden tekleyiciye dönüşüm

Etkinleştirme girişli bir kod çözücü, etkinleştirme girişi sabit 1 yerine veri girişi olarak kullanıldığında tekleyiciye dönüşür. Kod çözücünün her çıkışı, seçim sinyalinin gösterdiği konumda etkinleştirme değerini, kalan konumlarda 0 üretir. Etkinleştirme yerine G verildiğinde seçili çıkış G değerini, diğerleri 0 değerini taşır; bu tam olarak tekleyici davranışdır. Yapısal eşdeğerlik, mikroişlemci tasarımında veri yönlendirme birimlerinin kod çözücü ile tekleyici işlevini ortak donanımda sürdürmesini sağlar.

Örnek B.13

Birleşik Devre Tasarımı: Trafik Lambası Denetleyicisi. Bir kavşakta Kuzey-Güney (KG) ve Doğu-Batı (DB) yönlerinde iki yol kesismektedir; her yol için kırmızı, sarı ve yeşil olmak üzere üç lamba bulunur. Tasarlanacak devre toplam altı lamba sinyalini (KG_Yeşil, KG_Sarı, KG_Kırmızı, DB_Yeşil, DB_Sarı, DB_Kırmızı) yönetir.

Devrenin tek girişi, dışarıdaki bir zamanlayıcının ürettiği iki bitlik durum sinyalidir (S_1S_0). Zamanlayıcı dört evre arasında sırayla dolaşır:

- $S_1S_0 = 00$: Kuzey-Güney serbest (KG yeşil), Doğu-Batı durur (DB kırmızı).
- $S_1S_0 = 01$: Kuzey-Güney sarıya döner, Doğu-Batı hâlâ kırmızı.
- $S_1S_0 = 10$: Doğu-Batı serbest (DB yeşil), Kuzey-Güney durur (KG kırmızı).
- $S_1S_0 = 11$: Doğu-Batı sarıya döner, Kuzey-Güney hâlâ kırmızı.

Bu davranışı doğruluk tablosu olarak yazmak istenmektedir:

Trafik lambası denetleyicisi doğruluk tablosu

S_1	S_0	KG Yeşil	KG Sarı	KG Kırmızı	DB Kırmızı	DB Sarı	DB Yeşil
0	0	1	0	0	1	0	0
0	1	0	1	0	1	0	0
1	0	0	0	1	0	0	1
1	1	0	0	1	0	1	0

Çözüm. Tablodaki her çıkış sütunu Karno haritasıyla (veya doğrudan mini terimleri okuyarak) sadeleştirilir. Sonuç şu altı yalın ifadedir:

$$KG\_Yeşil = \overline{S_1} \cdot \overline{S_0}$$

$$KG\_Sarı = \overline{S_1} \cdot S_0$$

$$KG\_Kırmızı = S_1$$

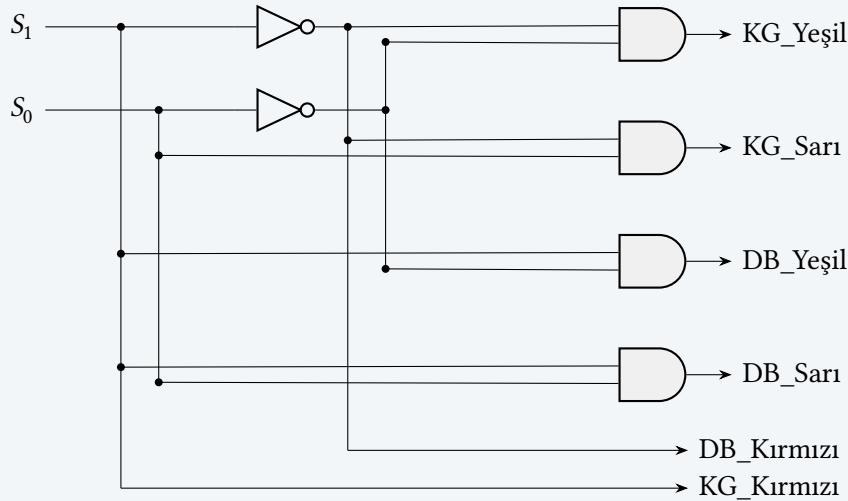
$$DB\_Yeşil = S_1 \cdot \overline{S_0}$$

$$DB\_Sarı = S_1 \cdot S_0$$

$$DB\_Kırmızı = \overline{S_1}$$

İfadelerin yapısı dikkat çekicidir: KG_Kırmızı yalnızca S_1 'e, DB_Kırmızı yalnızca $\overline{S_1}$ 'e bağlıdır; dolayısıyla iki kırmızı sinyali tek bir DEĞİL kapısıyla birbirinden üretilebilir. Yeşil ve sarı sinyallerinin her biri S_1, S_0 çiftinin belirli bir mini terimine karşılık gelir; her birini iki-girişli bir VE kapısıyla gerçekleştirmek yeterlidir. Toplam donanım: 4 VE + 1 DEĞİL kapısı.

Devre. Aşağıdaki kapı düzeyinde gerçekleştirme, S_1 ve S_0 girişlerinden altı lamba çıkışına kadar sinyal akışını gösterir.



Çıkışlar yalnızca o anki S_1S_0 girişine bağlıdır; zamanlayıcının durumları hangi zaman aralıklarında ürettiği bu devrenin ilgi alanı dışındadır. Tasarım sadece birleşik kısmı kapsar; durumlar arası geçişi yöneten zamanlayıcı bir ardışıl devre olduğundan ileride ardışıl tasarım örneklerinde ele alınacaktır.

B.6 Flip-Floplar

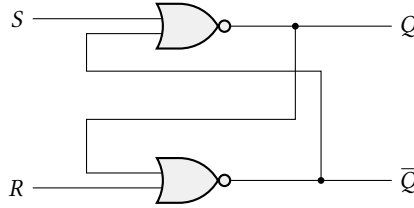
Şimdiye kadar incelenen mantık kapıları, kod çözücüler, çoklayıcılar ve toplayıcılar **birleşik** (İngilizcesi: *combinational*) devrelerdir: çıkışları yalnızca o andaki girişlere bağlıdır ve bellek tutmazlar. Oysa bir bilgisayarın çalışması için verilerin saklanması gerekir: yazmaçlar bir buyruğun sonucunu tutar, program sayacı bir sonraki buyruğun adresini hatırlar. Veri saklayabilen devrelere **ardışıl** (İngilizcesi: *sequential*) devreler denir ve bunların temel yapı taşı **flip-**

floptur.

Flip-flop, mantık kapılarından oluşmuş ve bir bit bilgi saklayabilen bir devredir. Dört temel flip-flop türü bulunmaktadır: RS, D, T ve JK. Bu bölümde her birinin yapısı, doğruluk tablosu ve kullanım alanları açıklanmaktadır.

B.6.1 RS Flip-flop

RS (*Reset-Set*) flip-flopu, çapraz bağlanmış iki VEYA DEĞİL kapısından oluşur. İki girişi (S ve R) ve iki çıkışı (Q ve $\bar{Q}$) vardır. S (Set) girişi flip-flopu 1 durumuna, R (Reset) girişi ise 0 durumuna getirir.



Şekil B.16: RS Flip-flop

Tablo B.6: RS Flip-flop doğruluk tablosu

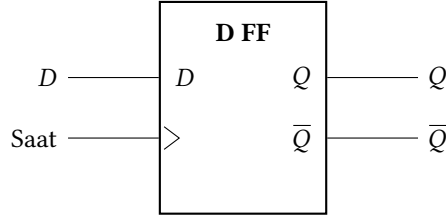
S	R	Q	$\bar{Q}$	Açıklama
0	0	$Q(t)$	$\bar{Q}(t)$	Değişme yok
0	1	0	1	Sıfırla
1	0	1	0	Kur
1	1	–	–	Kararsız durum

RS flip-flopun çalışma prensibi şu şekildedir: $S = 0$ ve $R = 0$ olduğunda flip-flop mevcut durumunu korur. $S = 1$ ve $R = 0$ olduğunda çıkış $Q = 1$ olur (kur). $S = 0$ ve $R = 1$ olduğunda çıkış $Q = 0$ olur (sıfırla). $S = 1$ ve $R = 1$ durumu ise kararsız durumdur ve kaçınılması gereken bir durumdur; çünkü her iki çıkış da aynı anda 0 olur ve bu durum flip-flopun doğru çalışmasını engeller.

RS flip-flop, VEYA DEĞİL kapıları yerine VE DEĞİL kapıları ile de oluşturulabilir. VE DEĞİL kapıları ile yapılan RS flip-flopunda girişler etkin-düşük olarak çalışır.

B.6.2 D Flip-flop

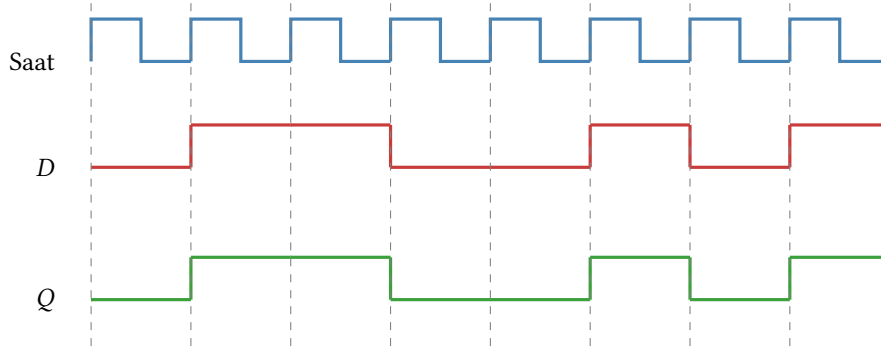
D (*Data*) flip-flopu, tek bir veri girişi (D) ve bir saat girişi olan flip-flop türüdür. Saat sinyalinin etkin kenarında (yükselen veya düşen kenar) D girişindeki değer Q çıkışına aktarılır. RS flip-flopundaki kararsız durum problemi yoktur çünkü yalnızca bir veri girişi vardır. D flip-flopu, çağdaş sayısal tasarımda en yaygın kullanılan flip-flop türüdür ve yazmaçların, boru hattı kademelerinin ve bellek hücrelerinin temel yapı taşıdır.

**Şekil B.17:** D Flip-flop**Tablo B.7:** D Flip-flop doğruluk tablosu — tam (sol) ve özet (sağ)

$Q(t)$	D	$Q(t + 1)$
0	0	0
0	1	1
1	0	0
1	1	1

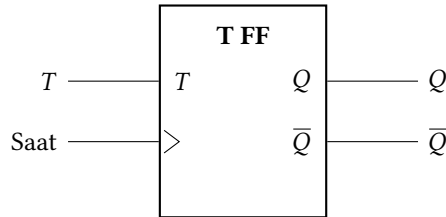
D	$Q(t + 1)$
0	0
1	1

Şekil B.18’de D flip-flopunun zamanlama diyagramı gösterilmiştir. Saat sinyalinin her yükselen kenarında (okla işaretli) D girişindeki değer Q çıkışına aktarılır; kenarlar arasında D ne kadar değişirse değişsin, Q sabit kalır.

**Şekil B.18:** D flip-flop zamanlama diyagramı: saat sinyalinin her yükselen kenarında D değeri Q ’ya aktarılır

B.6.3 T Flip-flop

T (*Toggle*) flip-flopu, tek bir girişi (T) olan flip-flop türüdür. $T = 1$ olduğunda flip-flopun durumu tersine döner (toggle), $T = 0$ olduğunda mevcut durum korunur. T flip-flopu özellikle sayaç devrelerinde yaygın olarak kullanılır.

**Şekil B.19:** T Flip-flop

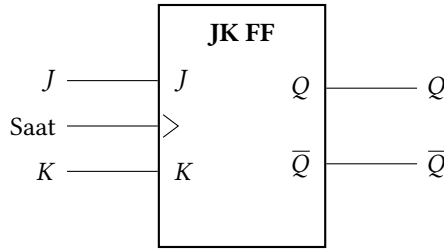
Tablo B.8: *T Flip-flop doğruluk tablosu — tam (sol) ve özet (sağ)*

T	$Q(t)$	$Q(t + 1)$
0	0	0
0	1	1
1	0	1
1	1	0

T	$Q(t + 1)$
0	$Q(t)$
1	$\overline{Q(t)}$

B.6.4 JK Flip-flop

JK flip-flop, RS flip-flopun geliştirilmiş halidir. İki girişi (J ve K) vardır. RS flip-floptaki kararsız durum ($S = R = 1$) problemi JK flip-flopunda çözülmüştür. $J = K = 1$ olduğunda flip-flopun durumu tersine döner.

**Şekil B.20:** JK Flip-flop**Tablo B.9:** *JK Flip-flop doğruluk tablosu*

J	K	$Q(t + 1)$	Açıklama
0	0	$Q(t)$	Değişme yok
0	1	0	Sıfırla
1	0	1	Kur
1	1	$\overline{Q(t)}$	Tersle

JK flip-flopun çalışma prensibi şu şekildedir: $J = 0$ ve $K = 0$ olduğunda flip-flop mevcut durumunu korur. $J = 0$ ve $K = 1$ olduğunda çıkış $Q = 0$ olur (sıfırla). $J = 1$ ve $K = 0$ olduğunda çıkış $Q = 1$ olur (kur). $J = 1$ ve $K = 1$ olduğunda ise çıkış tersine döner ve $Q(t + 1) = \overline{Q(t)}$ olur. Bu özellik JK flip-flop RS flip-flopundan ayıran en önemli farktır.

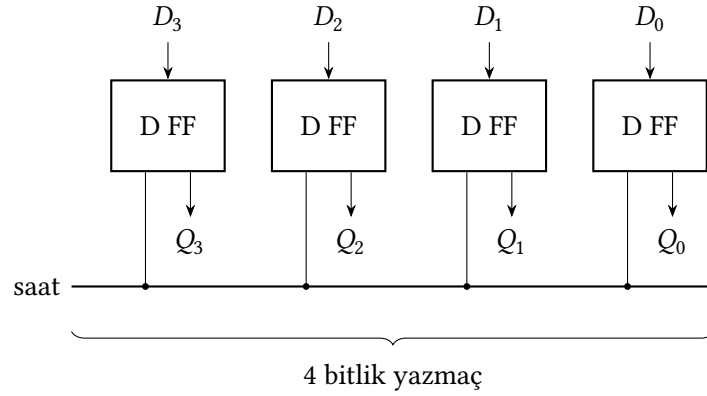
B.7 Yazmaçlar

Tek bir flip-flop yalnızca 1 bit saklar; oysa bir işlemcinin yazmaçları 32 veya 64 bit genişliğindedir. **Yazmaç** (İngilizcesi: *register*), birden fazla D flip-flopunun ortak bir saat sinyaliyle tetiklenerek bir araya getirilmesiyle oluşturulan çok bitlik saklama birimidir. Bölüm 6'daki yazmaç öbeği, program sayacı ve boru hattı kademelerindeki ara yazmaçlar bu yapının doğrudan uygulamalarıdır.

B.7.1 Koşut Yükleli Yazmaç

En basit yazmaç türü olan **koşut yükleli yazmaçta** tüm bitler aynı anda (tek bir saat vuruşunda) yüklenir. Her D flip-flopunun girişine ilgili veri biti bağlanır; saat sinyalinin yükselen

kenarında tüm bitler eşzamanlı olarak güncellenir.



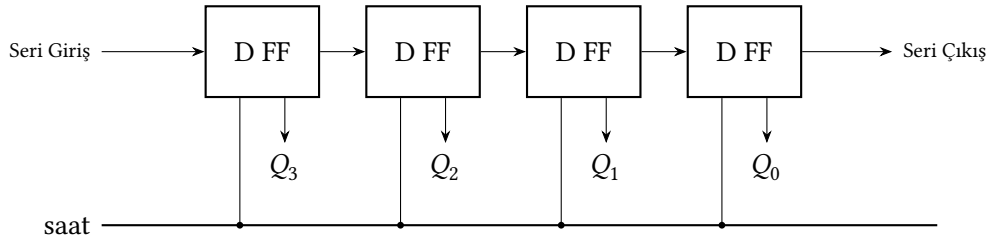
Şekil B.21: 4 bitlik koşut yüklemeli yazmaç

Uygulamada yazmaçlara genellikle bir “yükleme etkinleştirme” (İngilizcesi: *load enable*) denetim sinyali eklenir. Bu sinyal etkin düşük ya da etkin yüksek olarak tasarlanabilir; sinyal etkin değilken yazmaç, saat vuruşlarına karşın mevcut değerini korur, etkin olduğunda ise yeni veri yüklenir. Bu sayede yazmaç yalnızca istenildiğinde güncellenir.

B.7.2 Kaydırmalı Yazmaç

Kaydırmalı yazmaç (İngilizcesi: *shift register*), her saat vuruşunda saklanan bit dizisini bir konum sağa ya da sola kaydıran bir yazmaçtır. Her D flip-flopunun çıkışı (Q), bir sonraki (sağa kaydırmada) ya da bir önceki (sola kaydırmada) flip-flopun girişine (D) bağlanır.

Kaydırmalı yazmaçlar çarpma (sola kaydırma = $\times 2$) ve bölme (sağa kaydırma = $\div 2$) işlemlerinde, seri-koşut veri dönüştürmede ve iletişim arabirimlerinde yaygın biçimde kullanılır.



Şekil B.22: 4 bitlik sağa kaydırmalı yazmaç: seri giriş en soldaki D FF’nin girişine bağlanır; her saat vuruşunda her bit bir konum sağa kayar ve seri çıkıştan çıkar. Q_3-Q_0 koşut çıkışlardır.

Örnek B.14

4 bitlik bir kaydırmalı yazmaçta başlangıç değeri 1011 olsun. Sağa üç kez kaydırma yapıldığında ve boşalan en soldaki bite 0 yazıldığında sırayla hangi değerler elde edilir?

Çözüm: Başlangıç değeri yazmaç girişine yüklenir; ardından her saat vuruşunda tüm bitler bir konum sağa kaydırılır ve soldaki boşalan bite 0 atanır.

Saat vuruşu	Yazmaç içeriği
Başlangıç	1011
1. vuruş	0101
2. vuruş	0010
3. vuruş	0001

Başlangıç değeri $1011_2 = 11_{10}$ idi. Her sağa kaydırma ikiye bölmeye karşılık gelir: $11 \rightarrow 5 \rightarrow 2 \rightarrow 1$ (kesirli kısımlar atılır).

B.8 Sayaçlar

Sayaç (İngilizcesi: *counter*), her saat vuruşunda değerini belirli bir düzende artıran ya da azaltan ardışıl devredir. Bir sayaç, temelde bir yazmaç ile bu yazmacın değerini güncelleyen birleşik mantığın birleşimidir. Sayaçlar, işlemcilerde program sayacı (İngilizcesi: *program counter*) olarak, zamanlama devrelerinde, bellek adresleme birimlerinde ve olay sayma işlemlerinde kullanılır. RISC-V'teki program sayacı, her saat çevriminde 4 artırılan (dallanma durumunda ise hedef adres yüklenen) tipik bir sayaç uygulamasıdır.

B.8.1 Eşzamanlı İkili Sayaç

Eşzamanlı (İngilizcesi: *synchronous*) sayaçta tüm flip-floplar aynı saat sinyaliyle tetiklenir. n bitlik bir yukarı sayaç, 0'dan $2^n - 1$ 'e kadar sayar ve sonra sıfıra döner. Sayacı gerçekleştirmek için T flip-flopları kullanılır: en düşük bit (T_0) her zaman terslenirken, i . bit ancak kendisinden düşük tüm bitler 1 olduğunda terslenmelidir.

$$T_0 = 1, \quad T_1 = Q_0, \quad T_2 = Q_0 \cdot Q_1, \quad T_i = \prod_{j=0}^{i-1} Q_j$$

Örnek B.15

3 bitlik eşzamanlı yukarı sayacın sayma dizisi aşağıda çıkarılır.

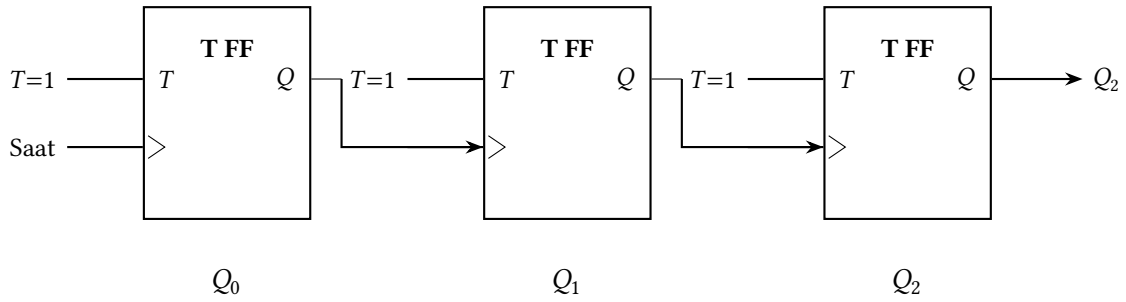
Çözüm:

Saat	Q_2	Q_1	Q_0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
(8)	0	0	0

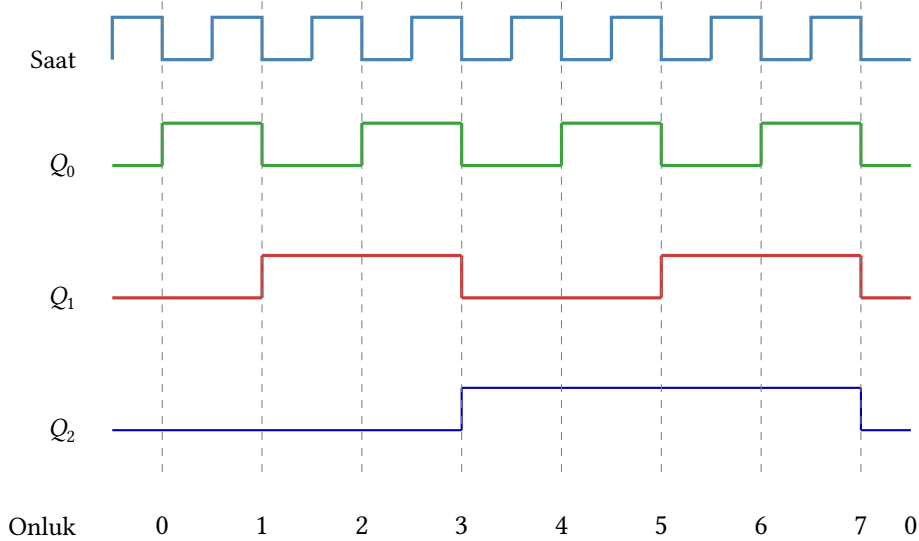
Sayaç 0'dan 7'ye kadar sayar ve saat vuruşu 8'de sıfıra döner. $T_0 = 1$ olduğu için en düşük bit her vuruşta değişir. $T_1 = Q_0$ olduğu için Q_1 , yalnızca $Q_0 = 1$ olduğu vuruşlarda değişir. $T_2 = Q_0 \cdot Q_1$ olduğu için Q_2 , ancak her iki alt bit de 1 olduğunda değişir.

B.8.2 Eşzamansız (Dalgalı) Sayaç

Eşzamansız (İngilizcesi: *asynchronous* veya *ripple*) sayaçta yalnızca ilk flip-flop doğrudan saat sinyaliyle tetiklenir; sonraki her flip-flopun saat girişi, bir önceki flip-flopun çıkışına bağlanır. Bu yapı daha basittir, ancak terslenme sinyali basamaktan basamağa gecikmeyle ilerler, elde sinyalinin “dalga” gibi yayıldığı bu davranış nedeniyle devre yüksek hızda çalışamaz. Yüksek hız gerektiren uygulamalarda eşzamanlı sayaçlar seçilir.



Şekil B.23: 3 bitlik eşzamansız (dalgalı) sayaç: her flip-flopun saat girişi bir önceki flip-flopun çıkışına bağlıdır



Şekil B.24: Şekil B.23'deki sayacın dalga formu: Q_0 her saat düşen kenarında, Q_1 ve Q_2 ise bir önceki bitin düşen kenarında terslenir. Onluk sayım 0'dan 7'ye yükselir, sonra başa döner.

Her flip-flopun tetiklenmesi bir önceki aşamanın çıkışına bağlı olduğundan, yayılma gecikmesi zincir boyunca birikerek artar; yüksek hız gerektiren uygulamalarda bu nedenle eşzamanlı sayaçlar kullanılır.

B.9 Mealy ve Moore Makineleri

Flip-floplar ve yazmaçlar veri saklar; ancak bir işlemcinin denetim birimi gibi karmaşık bir birim, yalnızca veri saklamakla kalmaz, aynı zamanda “şu an hangi aşamadayım?” sorusunu da yanıtlamak zorundadır. Örneğin çok çevrimli bir işlemcide denetim birimi sırayla “buyruk getir”, “çöz”, “yürüt” gibi aşamalardan geçer. Bu tür davranışlar **sonlu durum makineleri** (İngilizcesi: *finite state machine*, FSM) ile modellenir.

Sonlu durum makineleri, çıkışların nasıl belirlendiğine göre iki temel modele ayrılır: **Moore** ve **Mealy** makineleri. Bu iki model, aynı davranışı farklı şekillerde gerçekleştirebilir; hangisinin seçileceği uygulamanın gereksinimlerine bağlıdır.

Mealy Makinesi

Mealy makinesi, çıkışların hem *mevcut duruma* hem de *mevcut girişlere* bağlı olduğu bir sonlu durum makinesidir. Çıkışlar, girişler değiştiğinde (saat vuruşunu beklemeden) hemen güncellenebilir. Mealy makinesi, George H. Mealy tarafından 1955 yılında tanımlanmıştır [27].

Moore Makinesi

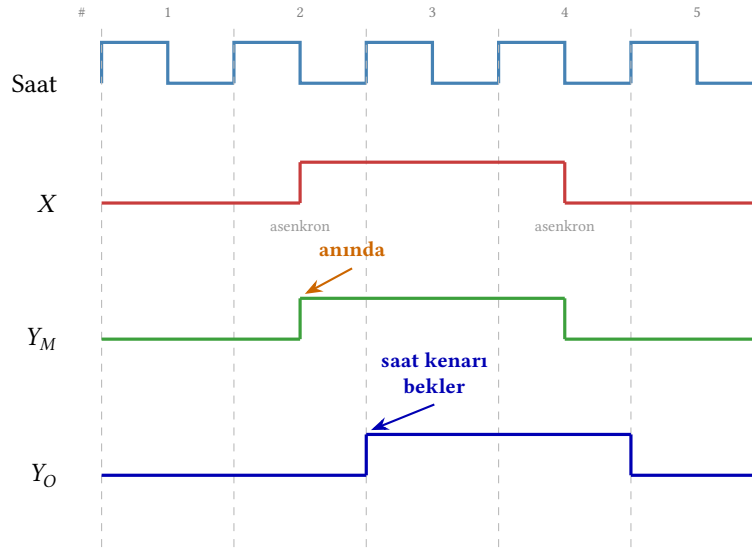
Moore makinesi, çıkışların *yalnızca mevcut duruma* bağlı olduğu bir sonlu durum makinesidir. Çıkışlar, durum geçişi tamamlandıktan sonra güncellenir ve bir sonraki saat vuruşuna kadar sabit kalır. Moore makinesi, Edward F. Moore tarafından 1956 yılında tanımlanmıştır [28].

B.9.1 İki Model Arasındaki Farklar

Mealy ve Moore makinelerinin temel farkları Tablo B.10’de özetlenmiştir.

Tablo B.10: Mealy ve Moore makinelerinin karşılaştırması

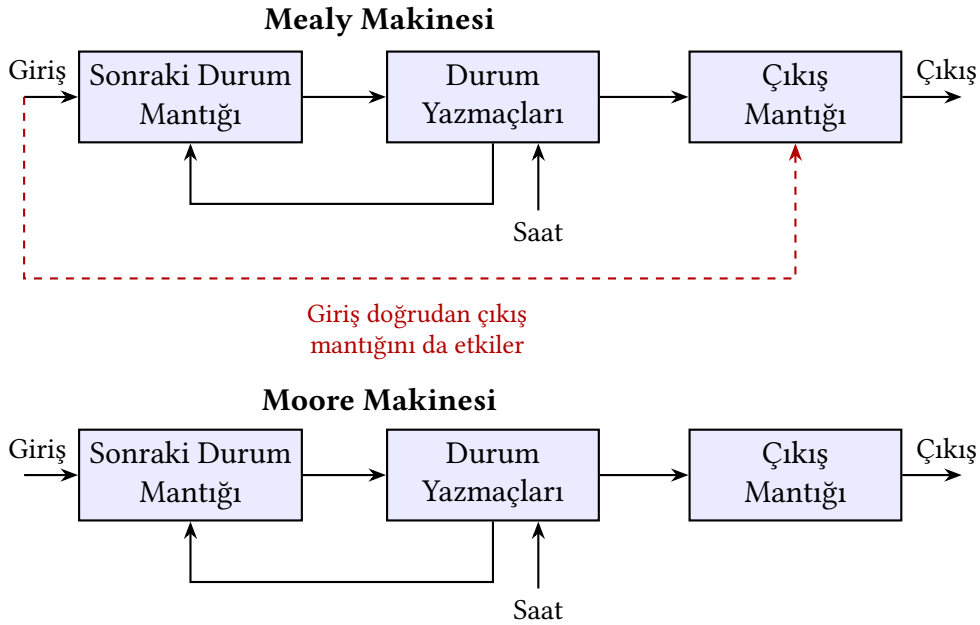
Özellik	Moore	Mealy
Çıkış bağımlılığı	Yalnızca durum	Durum + giriş
Çıkış güncellemesi	Saat vuruşunda	Giriş değiştiğinde
Durum sayısı	Genellikle daha fazla	Genellikle daha az
Çıkış kararlılığı	Daha kararlı	Titreşime açık
Tepki hızı	Bir çevrim gecikmeli	Anında tepki
Durum diyagramında çıkış	Durumun içinde	Geçiş okları üzerinde



Şekil B.25: Mealy ve Moore makinelerinin zamanlama farkı: Mealy çıkışı Y_M giriş X değiştiği anda hemen güncellenir; Moore çıkışı Y_O ise bir sonraki saat yükselen kenarında değişir. Bu nedenle Mealy daha hızlı tepki verirken Moore titreşime karşı daha karardır.

B.9.2 Blok Diyagramları

İki modelin donanım yapısını karşılaştırmak, aralarındaki farkı anlamak açısından faydalıdır.



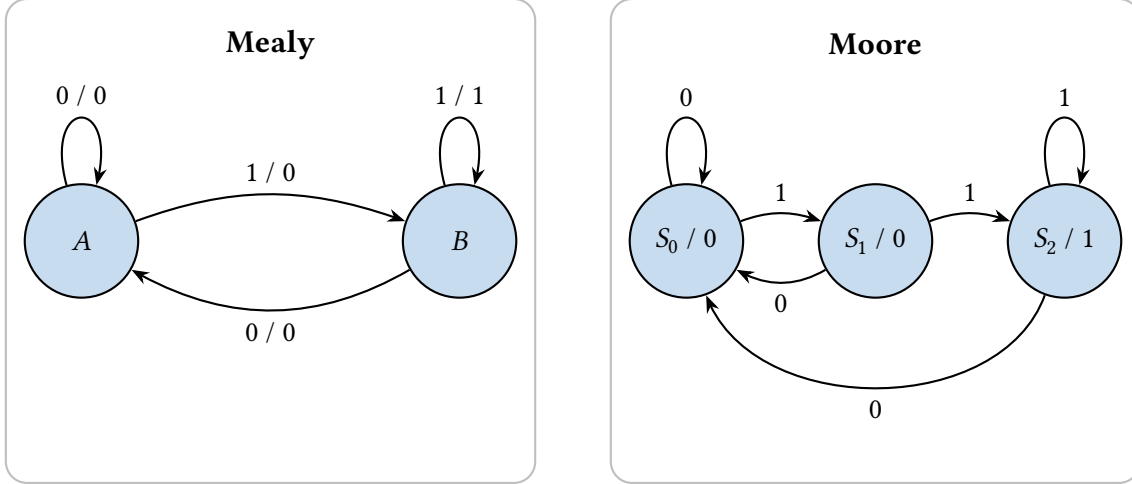
Şekil B.26: Mealy ve Moore makinelerinin blok diyagramları. Saat ortak senkronizasyon sinyalidir; her saat vuruşunda durum yazmaçları yenilenir. Mealy makinesinde giriş sinyali (kesikli kırmızı ok) çıkış mantığını doğrudan etkiler; Moore'da ise çıkışlar yalnızca duruma bağlıdır.

B.9.3 Durum Diyagramı Gösterimi

İki modelin durum diyagramları farklı biçimde çizilir:

- **Moore makinesinde:** Çıkış, durumun *içine* yazılır (durum adı / çıkış). Geçiş okları üzerinde yalnızca giriş koşulu bulunur.
- **Mealy makinesinde:** Çıkış, geçiş okları *üzerine* yazılır (giriş / çıkış biçiminde). Durum düğümlerinde çıkış gösterilmez.

Aşağıdaki şekilde aynı davranışı gerçekleştiren bir Moore ve bir Mealy makinesi karşılaştırılmıştır. Sistem, giriş sinyalinde iki ardışık '1' algılandığında çıkışı '1' yapar.



Şekil B.27: Ardışık iki '1' algılama: Mealy (2 durum, solda) ve Moore (3 durum, sağda) karşılaştırması

Bu örnekte görüldüğü gibi, Mealy makinesi aynı davranışı daha az durumla gerçekleştirebilmektedir (2 durum yerine 3). Ancak Moore makinesinin çıkışı saat sinyaliyle eşzamanlı olduğundan daha kararlıdır.

B.9.4 Kullanım Alanları ve Tercih Kriterleri

- **Moore makinesi tercih edilir:** Çıkışların kararlı ve titreşimsiz (*glitch-free*) olması gereken durumlarda. Örneğin, denetim birimi tasarımlarında Moore modeli yaygındır çünkü çıkışlar yalnızca saat kenarında değişir.
- **Mealy makinesi tercih edilir:** Hızlı tepki süresi gereken ve daha az donanım kaynağı kullanılması istenen durumlarda. Girişe anında yanıt verebilmesi önemli bir avantajdır.
- Uygulamada birçok tasarım, **kayıtlı çıkışlı Mealy** (*registered-output Mealy*) yaklaşımını kullanır: Mealy makinesinin çıkışları bir yazmaç üzerinden geçirilerek hem hız avantajı korunur hem de çıkış kararlılığı sağlanır.

B.9.5 Ardışıl Devre Tasarımı Adımları

Ardışıl devre tasarımı sistematik bir süreçtir; aşağıdaki adımlarla gerçekleştirilir:

1. **Problem tanımı:** Tasarlanacak devrenin işlevi sözlü olarak tanımlanır; girişler, çıkışlar ve istenen davranış belirlenir.
2. **Durumların belirlenmesi:** Davranışı gerçekleştirmek için gereken minimum durum kümesi çıkarılır. Her durum, devrenin "bellek"inde tutulan farklı bir geçmişi temsil eder.
3. **Durum geçiş tablosu:** Her durum–giriş çifti için bir sonraki durum ve çıkış değerleri yazılır. Mealy makinesinde çıkışlar geçişlere, Moore makinesinde durumlara bağlıdır.

4. **Durum diyagramı:** Tablo görsel olarak çizilir; durumlar düğüm, geçişler ok olur. Bu adım hataları görsel doğrulamayı kolaylaştırır.
5. **Durum kodlaması:** Her duruma ikilik bir kod atanır. n durum için $\lceil \log_2 n \rceil$ flip-flop yeterlidir. Kodlama seçimi sonraki adımdaki sadeleştirmeyi etkiler.
6. **Flip-flop seçimi ve uyarma denklemleri:** D, T ya da JK flip-flop seçilir; her flip-flopun girişi için durum geçiş tablosundan denklem türetilir.
7. **Boole sadeleştirme:** Flip-flop ve çıkış denklemleri Karno haritası ya da Boole cebri ile en yalın biçimine indirgenir.
8. **Devre çizimi:** Sadeleştirilmiş denklemler kapı düzeyinde çizilerek devre tamamlanır.

Bu adımlar aşağıdaki örnekte bir içecek otomatı makinesi üzerinden gösterilmektedir.

Örnek B.16

Tasarım Örneği: İçecek Otomatı Makinesi. Bu örnekte basit bir içecek otomatı makinesi tasarlanmaktadır. Makine 1 TL ve 2 TL kabul eder; içecek fiyatı 3 TL'dir. Yeterli para atıldığında içecek verilir ve para üstü varsa iade edilir. Tasarım bir Mealy durum makinesi olarak gerçekleştirilir: içecek verme ve para üstü çıkışları, mevcut durumun yanı sıra atılan paraya (giriş) de bağlıdır; örneğin aynı S2 durumunda 1 TL atılırsa yalnızca içecek verilir, 2 TL atılırsa hem içecek hem para üstü verilir.

Adım 1: Durumların belirlenmesi.

- **S0:** Bekleme (toplam = 0 TL)
- **S1:** 1 TL alındı
- **S2:** 2 TL alındı
- **S3:** 3+ TL (içecek ver)

Girişler: **BirTL** (1 TL atıldı), **İkiTL** (2 TL atıldı). Çıkışlar: **İçecekVer**, **ParaÜstü**.

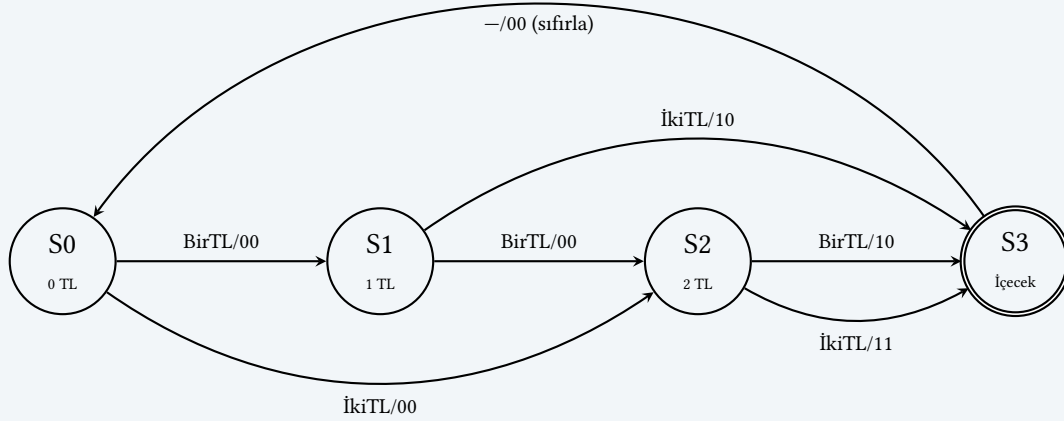
Adım 2: Durum geçiş tablosu.

Otomat makinesi durum geçiş tablosu

Mevcut Durum	Giriş		Sonraki Durum	Çıkış	
	BirTL	İkiTL		İçecekVer	ParaÜstü
S0	0	0	S0	0	0
S0	1	0	S1	0	0
S0	0	1	S2	0	0
S1	0	0	S1	0	0
S1	1	0	S2	0	0
S1	0	1	S3	1	0
S2	0	0	S2	0	0
S2	1	0	S3	1	0
S2	0	1	S3	1	1
S3	—	—	S0	0	0

S3 durumunda içecek verilir ve makine S0'a döner. S2'den İkiTL ile geçişte para üstü (1 TL) iade edilir.

Adım 3: Durum diyagramı.



Otomat makinesi Mealy durum diyagramı. Geçiş etiketleri “giriş/çıkış” biçimindedir; giriş (BirTL, İkiTL), çıkış (İçecekVer, ParaÜstü). Para girişi olmayan saat tıklarında durum değişmez; bu kendi-üstü geçişler görsel sadelik için çizilmemiştir.

Adım 4–5: Durum kodlaması ve flip-flop denklemleri.

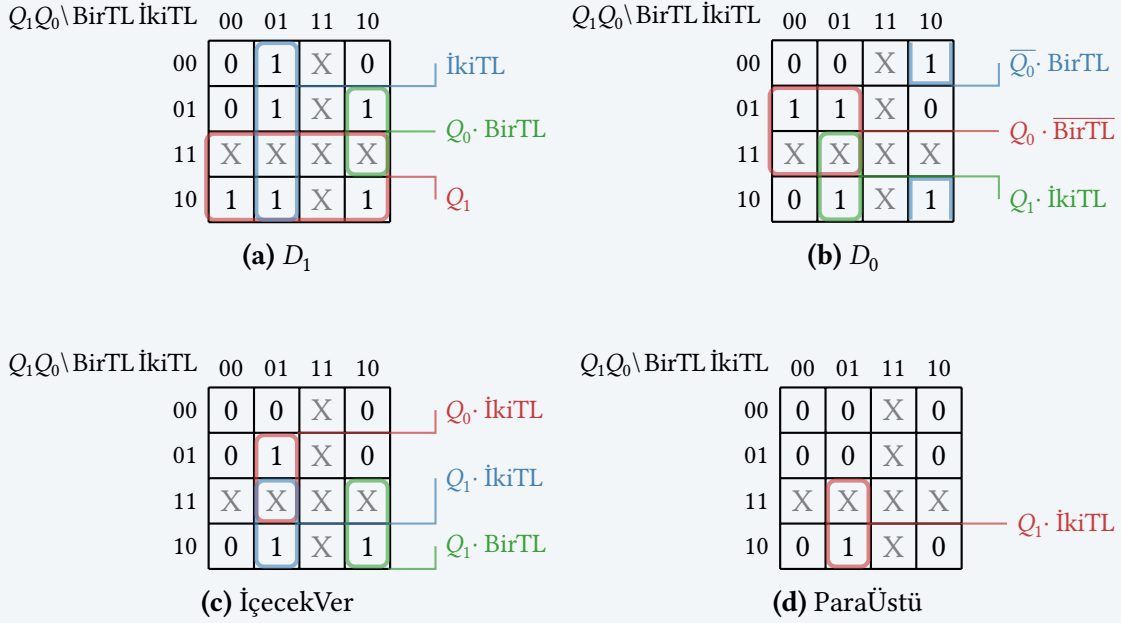
Durumlar 2 bit ile kodlanır: S0= 00, S1= 01, S2= 10, S3= 11. Mevcut durum bitleri Q_1Q_0 , sonraki durum bitleri D_1D_0 olarak adlandırılır.

Otomat makinesi: kodlanmış durum geçiş tablosu

Mevcut		Giriş		Sonraki		Çıkış	
Q_1	Q_0	BirTL	İkiTL	D_1	D_0	İçecekVer	ParaÜstü
0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0
0	0	0	1	1	0	0	0
0	1	0	0	0	1	0	0
0	1	1	0	1	0	0	0
0	1	0	1	1	1	1	0
1	0	0	0	1	0	0	0
1	0	1	0	1	1	1	0
1	0	0	1	1	1	1	1
1	1	X	X	0	0	0	0

Adım 6–7: Boole sadeleştirme — Karno haritaları.

Dört değişken ($Q_1, Q_0, \text{BirTL}, \text{İkiTL}$) üzerinden 2×4 Karno haritaları çizilir. S3 ($Q_1Q_0 = 11$) durumundan çıkışlar için önemsenmeyen (X) satırı kullanılır.



Otomat makinesi: D flip-flop ve çıkış Karno haritaları

Karno haritalarından elde edilen sadeleştirilmiş ifadeler (S3 durumu ile eşzamanlı para atma durumu, BirTL=İkiTL= 1, önemsenmeyen durum olarak ele alınmıştır):

$$\begin{aligned}
 D_1 &= Q_1 + \text{İkiTL} + Q_0 \cdot \text{BirTL} \\
 D_0 &= \overline{Q_0} \cdot \text{BirTL} + Q_0 \cdot \overline{\text{BirTL}} + Q_1 \cdot \text{İkiTL} \\
 \text{İçecekVer} &= \text{İkiTL} \cdot (Q_0 + Q_1) + \text{BirTL} \cdot Q_1 \\
 \text{ParaÜstü} &= Q_1 \cdot \text{İkiTL}
 \end{aligned}$$

Bu örnekte flip-floplar, durum makinesi kavramı ve birleşik mantık birlikte çalışmaktadır: flip-floplar durumu saklar, birleşik devre ise sonraki durumu ve çıkışları hesaplar. İşlemci tasarımında kullanılan denetim birimleri de özünde birer durum makinesidir. Bölüm 6'da göreceğimiz tek vuruşluk ve çok vuruşluk işlemci denetim birimleri, burada öğrendiğimiz ardışıl tasarım ilkelerinin doğrudan uygulamalarıdır.

Örnek B.17

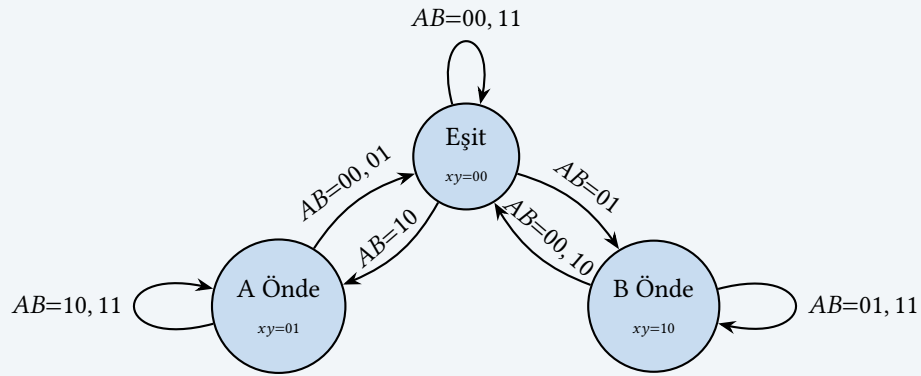
Tasarım Örneği: Yarışma Sistemi. Daha kapsamlı bir örnek olarak iki yarışmacı (A ve B) arasındaki son iki sorunun cevaplarını izleyen bir yarışma sistemi tasarlınsın. Sistemin çıkışları aşağıdaki tabloda gösterilmiştir:

A ve B'nin doğru sayılarına göre çıktı tablosu

Durum	Çıkış
Eşit	000
A, B'den 2 fazla doğru	001
A, B'den 1 fazla doğru	010
B, A'dan 1 fazla doğru	011
B, A'dan 2 fazla doğru	100

Sistem üç durumda bulunabilir:

- Eşit durumu: $xy = 00$
- A önde durumu: $xy = 01$
- B önde durumu: $xy = 10$

*Yarışma örneği durum şeması**Yarışma örneği durumlara göre çıkış değerleri*

x	y	A	B	$x(t+1)$	$y(t+1)$	ζ_2	ζ_1	ζ_0
0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	1	1
0	0	1	0	0	1	0	1	0
0	0	1	1	0	0	0	0	0
0	1	0	0	0	0	0	1	0
0	1	0	1	0	0	0	0	0
0	1	1	0	0	1	0	0	1
0	1	1	1	0	1	0	1	0
1	0	0	0	0	0	0	1	1
1	0	0	1	1	0	1	0	0
1	0	1	0	0	0	0	0	0
1	0	1	1	1	0	0	1	1
1	1	X	X	X	X	X	X	X

Tasarımda JK flip-flopları kullanılacaktır. JK flip-flopunun geçiş tablosu aşağıda veril-

miştir:

JK flip-flopta J ve K değerleri için doğruluk tablosu

x	$x(t + 1)$	J_x	K_x	Açıklama
0	0	0	X	Sıfırla ya da aynı kal
0	1	1	X	Birle ya da tersle
1	0	X	1	Sıfırla ya da tersle
1	1	X	0	Birle ya da aynı kal

Bu geçiş tablosu kullanılarak tüm sistemin doğruluk tablosu oluşturulur:

Tüm sistem için doğruluk tablosu

x	y	A	B	$x(t + 1)$	$y(t + 1)$	C_2	C_1	C_0	J_x	K_x	J_y	K_y
0	0	0	0	0	0	0	0	0	0	X	0	X
0	0	0	1	1	0	0	1	1	1	X	0	X
0	0	1	0	0	1	0	1	0	0	X	1	X
0	0	1	1	0	0	0	0	0	0	X	0	X
0	1	0	0	0	0	0	1	0	0	X	X	1
0	1	0	1	0	0	0	0	0	0	X	X	1
0	1	1	0	0	1	0	0	1	0	X	X	0
0	1	1	1	0	1	0	1	0	0	X	X	0
1	0	0	0	0	0	0	1	1	X	1	0	X
1	0	0	1	1	0	1	0	0	X	0	0	X
1	0	1	0	0	0	0	0	0	X	1	0	X
1	0	1	1	1	0	0	1	1	X	0	0	X
1	1	0	0	X	X	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X	X	X

Her bir çıkış fonksiyonu ve flip-flop girişleri Karno haritası ile sadeleştirilir:

$xy \backslash AB$	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	X	X	X	X
10	0	1	0	0

(a) ζ_2

$x \cdot \bar{A} \cdot B$

$xy \backslash AB$	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	X	X	X	X
10	1	0	1	0

(b) ζ_1

$\bar{x} \bar{y} \bar{A} B$
 $\bar{x} \bar{y} A \bar{B}$
 $y A B$
 $x \bar{A} \bar{B}$

$xy \backslash AB$	00	01	11	10
00	0	1	0	0
01	0	0	0	1
11	X	X	X	X
10	1	0	1	0

(c) ζ_0

$\bar{x} \bar{y} \bar{A} B$
 $\bar{x} \bar{y} A \bar{B}$
 $x \bar{A} \bar{B}$
 $x A B$

$xy \backslash AB$	00	01	11	10
00	0	1	0	0
01	0	0	0	0
11	X	X	X	X
10	X	X	X	X

(d) J_x

$\bar{y} \cdot \bar{A} \cdot B$

$xy \backslash AB$	00	01	11	10
00	X	X	X	X
01	X	X	X	X
11	X	X	X	X
10	1	0	0	1

(e) K_x

$\bar{B}$

$xy \backslash AB$	00	01	11	10
00	0	0	0	1
01	X	X	X	X
11	X	X	X	X
10	0	0	0	0

(f) J_y

$\bar{x} \cdot A \cdot \bar{B}$

$xy \backslash AB$	00	01	11	10
00	X	X	X	X
01	1	1	0	0
11	X	X	X	X
10	X	X	X	X

(g) K_y

$\bar{A}$

Sayaç tasarımı Karno haritaları

Karno haritalarından elde edilen sadeleştirilmiş ifadeler şunlardır:

$$\begin{aligned} J_x &= \bar{y} \cdot \bar{A} \cdot B & K_x &= \bar{B} \\ J_y &= \bar{x} \cdot A \cdot \bar{B} & K_y &= \bar{A} \end{aligned}$$

Flip-flop giriş ifadeleri oldukça yalındır: x flip-flopu yalnızca B doğru yanıtladığında kurulur ($J_x = \bar{y} \bar{A} B$) ve B yanlış yanıtladığında sıfırlanır ($K_x = \bar{B}$). Benzer şekilde y flip-flopu yalnızca A doğru yanıtladığında kurulur ($J_y = \bar{x} A \bar{B}$) ve A yanlış yanıtladığında sıfırlanır ($K_y = \bar{A}$).

$$\zeta_2 = x \cdot \bar{A} \cdot B$$

$$\zeta_1 = \bar{x} \bar{y} (A \oplus B) + (x + y) \cdot (A \odot B)$$

$$\zeta_0 = \bar{x} \bar{y} \bar{A} B + \bar{x} y A \bar{B} + x \bar{A} \bar{B} + x A B$$

ζ_1 ifadesindeki yapıya dikkat ediniz: eşit durumda ($xy=00$) yalnızca A ve B farklı yanıt verdiğinde ($A \oplus B$) skor farkı oluşur; önde olan varsa ($x+y=1$) yalnızca aynı yanıt verdiğinde ($A \odot B$) fark korunur.

Bu ifadeler doğrudan mantık kapılarıyla gerçekleştirilir. JK flip-floplarının J ve K girişleri yukarıdaki ifadelere, saat girişleri ise ortak saat sinyaline bağlanır. Çıkış fonksiyonları (ζ_2 , ζ_1 , ζ_0) da ayrı birleşik devrelerle üretilir. Sonuçta ortaya çıkan devre, her saat vuruşunda A ve B girişlerini okuyarak durumu günceller ve yarışma skorunu gösteren 3 bitlik bir çıkış üretir.

B.10 Yanılgılar ve Tuzaklar

Sayısal tasarım konuları öğrencilere soyut ve kesin görünse de uygulamaya geçildiğinde sezgiyi yanıltacak pek çok durum ortaya çıkar. Karno haritasında gözden kaçan bir gruplama, kurulum-tutma süresi ihlali ya da önemsenmeyen hücrelerin yanlış işlenmesi; bunların hepsi simülasyonda değil gerçek donanımda kendini gösteren hatalardır. Bu bölümde sayısal tasarımda en sık karşılaşılan yanılgıları ve tuzakları somut örneklerle inceliyoruz.

Yanılgı: Karno haritası 4'ten fazla değişken için de yeterlidir

İki, üç ve dört değişkenli fonksiyonlarda Karno haritası görsel gruplamayla en küçük ifadeye kolayca ulaşmayı sağlar. Ancak beş değişkenden itibaren hücreler iki katmanlı ya da üç boyutlu düzene geçer; gözle yapılan gruplama hem hataya açık hem de zaman alıcı olur. Altı ve daha fazla değişkende Quine-McCluskey algoritması ya da ikili karar diyagramları (BDD) tutarlı ve sistematik bir çözüm sunar. “Harita büyütülürse yine de çalışır” anlayışı, karmaşık sadeleştirme görevlerinde gözden kaçan örtük terimlere ve fazladan kapı sayısına yol açar.

Tuzak: Önemsenmeyen hücreleri sadeleştirmede kullanmamak

Karno haritasında önemsenmeyen (X) hücreler, o giriş kombinasyonu gerçekte oluşmayacağından çıkışın ne olduğunun önem taşımadığı durumları gösterir. Bu hücreler hem 0 hem de 1 olarak atanabilir; gruplamalarda X hücrelerini 1 kabul etmek daha büyük gruplar oluşturmayı ve böylece daha az terimli bir Boole ifadesi elde etmeyi sağlar. Yalnızca 1 hücrelerini gruplayıp X'leri yok saymak, ulaşılabilecek en yalın ifadeden daha karmaşık bir devre üretir. Öte yandan X hücrelerini 1 alarak kapatılmış bir grubu oluşturmak ve sonra aynı X'leri 0 sayarak çıkan başka bir grup oluşturmak tutarsızlığa yol açar; her X hücresi için tutarlı bir atama yapılmalıdır.

Yanılgı: Mealy makinesi her zaman Moore makinesinden daha az durum kullanır

Mealy modelinde çıkışlar hem mevcut duruma hem de anlık girişe bağlı olduğundan bazı Moore durumları birleştirilebilir; bu da daha az durum anlamına gelebilir. Ancak bu ilişki evrensel

değildir: bazı belirtilimlerde Moore ve Mealy makinesinin durum sayısı eşit olabilir, hatta Mealy'nin girişe bağımlı çıkış üretmesi kaçınılmaz gecikmeler ya da geçici saçma çıkış (*glitch*) sorunları doğurabilir. Moore makinesi çıkışları yalnızca mevcut duruma bağlı olduğundan saat kenarında kararlı değer üretir; bu özellik çıkış sinyalinin doğrudan başka flip-floplara beslendiği tasarımlarda tercih edilir. “Az durum her zaman daha iyi” yaklaşımı, çıkış kararlılığı gereksinimini göz ardı eder.

Tuzak: Yayılım gecikmesini göz ardı ederek saat sıklığını belirlemek

Bir ardışıl devrenin maksimum saat sıklığı, kaynak flip-flop çıkışından hedef flip-flop girişine uzanan en uzun birleşik mantık yolu ve flip-flop'un kurulum süresiyle sınırlıdır:

$$T_{\text{saat}} \geq t_{c-q} + t_{\text{birleşik,maks}} + t_{\text{kurulum}}$$

Bu eşitsizlik sağlanmazsa hedef flip-flop veriyi doğru saat kenarında yakalayamaz ve durum bozular. Simülasyonda ideal (sıfır gecikmeli) kapılarla çalışan bir devre, FPGA ya da ASIC gerçeklemede zamanlama hatası verebilir. Sentez araçlarının ürettiği zamanlama raporunda kayma payı (*slack*) değeri negatifse bu tuzağa düşülmüş demektir.

Yanılğı: Eşzamansız (dalgalı) sayaç her zaman eşzamanlı sayaçtan daha basit ve hızlıdır

Eşzamansız sayaçta her flip-flop bir öncekinin Q çıkışını saat girişi olarak alır; bu durum devre şemasını görece yalın tutar. Ancak her flip-flopun saat-çıkış gecikmesi (t_{c-q}) kümülatif olarak birikerek yayılır: n -bitlik dalgalı sayaçta en yüksek bitin kararlı değere ulaşması $n \times t_{c-q}$ gecikme ister. Eşzamanlı sayaçta tüm flip-floplar aynı saate bağlı olduğundan bu gecikme sabit ve kısa tutulur; birleşik mantık katmanını biraz karmaşıklaşırsa da çok daha yüksek saat sıklığına erişilebilir. Geniş sayaçlarda eşzamansız yapı saat sıklığını ciddi biçimde sınırlar.

Tuzak: Kurulum ve tutma süresi ihlalini yalnızca kurulum süresine indirgeyerek değerlendirmek

Zamanlama analizinde kurulum süresi (t_{su}) ile tutma süresi (t_{hold}) birlikte dikkate alınmalıdır. Kurulum süresi, saat kenarından önce girişin ne kadar süre kararlı kalması gerektiğini tanımlar; tutma süresi ise saat kenarından *sonra* girişin ne kadar süre sabit tutulması gerektiğini belirtir. Tutma süresi ihlali, kaynak flip-floptan hedef flip-flopa giden kombinasyonel yolun çok kısa olmasından kaynaklanır; bu durum iki flip-flop arasına kasıtlı gecikme eklenerek giderilir. Zamanlama kapatma (*timing closure*) çalışmalarında tutma süresi ihlali kurulum ihlali kadar sık karşılaşılr ve statik zamanlama analizi her iki kısıtı da en kötü durum için ayrı ayrı değerlendirir.

Tuzak: K-haritasında kenar sarma gruplarını kaçırmak

Karno haritası torus (halka) topolojisinde düzenlenmiştir: sol kenar sağ kenarla, üst kenar alt kenarla bitişiktir. Bu sayede köşe ve kenar hücreleri de komşu sayılır; örneğin 4-değişkenli haritanın dört köşesi tek bir grup oluşturabilir. Bu gruplamayı görmezden gelmek, daha küçük ve ayrık gruplar seçmeye yol açar; sonuç olarak gereksiz fazla terimli bir Boole ifadesi ve daha çok kapı kullanan bir devre üretilir. Kenar hücrelerini mutlaka kontrol etmek ve haritayı iki yönde de “sararak” grupları değerlendirmek doğru alışkanlıktır.

Özet

Bu ekte, sayısal devrelerin temelini oluşturan kavramlar ve yapı taşları ele alındı. Buraya kadar öğrendiklerimizi kısaca derleyelim.

Sayısal devrelerin en küçük yapı taşları mantık kapılarıdır. VE, VEYA ve DEĞİL kapıları en temel üçlüyü oluşturur; bunların bileşiminden türeyen NAND, NOR ve XOR kapıları ise pratikte sıkça kullanılır. Her kapının davranışı doğruluk tablosuyla tam olarak tanımlanır. NAND ve NOR kapılarının evrensel niteliği özellikle dikkat çekicidir: bu iki kapıdan yalnızca biriyle, VE, VEYA ve DEĞİL dahil tüm mantık işlemleri gerçekleştirilebilir. Bu özellik, gerçek yonga üretiminde kapı türü sayısını azaltmak ve üretim sürecini yalınlaştırmak açısından büyük bir avantaj sağlar.

Mantık fonksiyonlarını sistematik biçimde ifade etmek ve sadeleştirmek için Boole cebri kullanılır. De Morgan kuramları, VE ile VEYA işlemleri arasında tümleyen yardımıyla geçiş yapılmasını sağlar ve devrelerin farklı kapı türleriyle eşdeğer biçimde yazılmasına olanak tanır. Mini terim ve maksı terim gösterimleri ise herhangi bir mantık fonksiyonunu standart bir biçimde ifade etmeyi mümkün kılar.

Karno haritası, Boole fonksiyonlarının görsel sadeleştirilmesi için kullanılır. İki, üç ve dört değişkenli fonksiyonlarda komşu hücreleri gruplandırarak en az sayıda kapı kullanan ifadeye ulaşmak kolaylaşır. Önemsenmeyen durumlar (X), grublama sırasında esneklik tanır ve daha kısa ifadeler elde etmeye yardımcı olur.

Birleşik devrelerin yapı taşları arasında kod çözücüler, kodlayıcılar, çoklayıcılar ve toplayıcılar yer alır. Bir kod çözücü n bitlik girişi 2^n çıkıştan tam olarak birine yönlendirir; kodlayıcı ise tersini yapar. Çoklayıcı, 2^n girişten birini seçim sinyaliyle göre tek çıkışa aktarır. Toplayıcılar ise ikilik toplama işlemini donanımda gerçekleştirir: yarım toplayıcı elde girişini görmez; tam toplayıcı elde girişini de hesaba katar; dalgalı elde toplayıcı tam toplayıcıları zincirleyerek çok bitli toplamayı üretir. Önceden elde üreten toplayıcılar bu gecikmeyi kısaltır.

Flip-floplar tek bitlik bilgi depolayan ardışıl devre elemanlarıdır. RS, D, T ve JK flip-flopları farklı tetikleme ve tutma davranışları sergiler; hepsinde ortak olan nokta, durum değişiminin saat sinyaliyle eşzamanlanmasıdır. Birden fazla flip-flop bir araya geldiğinde yazmaç oluşur: koştuk yüklemeli yazmaçlar tüm bitleri aynı anda alırken, kaydırma yazmaçları veriyi seri olarak kaydırır. Sayaçlar ise her saat vuruşunda değerini artıran ya da azaltan özel yazmaçlardır; eşzamanlı sayaçlarda tüm flip-floplar aynı anda güncellenir, eşzamansız sayaçlarda ise değişim dalgalanarak ilerler.

Ardışıl devrelerin en genel modeli durum makineleridir. Mealy modelinde çıkışlar hem mevcut duruma hem de o andaki girişe bağlıdır; Moore modelinde çıkışlar yalnızca mevcut duruma bağlıdır. Mealy makinesi daha az durumla daha hızlı tepki verebilir, Moore makinesi ise çıkışların saat sinyaliyle eşzamanlanması sayesinde daha kararlı bir davranış sergiler. Durum makinesi tasarımı sistematik adımlardan oluşur: önce durumlar ve geçişler tanımlanır, ardından durum tablosu oluşturulur, flip-flop türü seçilir, Karno haritasıyla geçiş ve çıkış denklemleri sadeleştirilir ve son olarak devre gerçekleştirilir.

Bu ekte öğrendiğimiz yapı taşları, Bölüm 6'daki işlemci tasarımında doğrudan karşımıza çıkacak. Denetim birimi bir durum makinesidir; yazmaç öbeği flip-floplardan oluşur; veri yolundaki AMB birleşik mantık kapılarıyla kurulur. Günümüz yongalarında bu temel devrelerin milyarlarca kopyası nanometre ölçeğinde bir araya gelir; ancak her birinin davranışı bu ekte gördüğümüz basit kapı ve flip-flop kurallarıyla tam olarak açıklanabilir.

Alıřtırmalar

Alıřtırma B.1. ★ Ařağıdaki Boole ifadelerine De Morgan kuramını uygulayarak eřdeęer ifadelerini yazın:

- (a) $\overline{A \cdot B \cdot C}$
- (b) $\overline{A + B + C}$
- (c) $\overline{A \cdot (B + C)}$

Alıřtırma B.2. ★ VE, VEYA ve DEĞİL kapılarının doęruluk tablolarını yazın. TÜVE kapısının, bir VE kapısı ve ardından gelen bir DEĞİL kapısıyla eřdeęer olduęunu doęruluk tablosuyla gösterin.

Alıřtırma B.3. ★★ Üç giriřli (x, y, z) bir fonksiyon, giriřlerin çoęunluęu 1 olduęunda 1 çıkıřı veren bir çoęunluk devresini temsil etmektedir. Bu fonksiyonun doęruluk tablosunu oluřturun ve Boole ifadesini yazın.

Alıřtırma B.4. ★★ Ařağıdaki Boole fonksiyonlarını Karno haritası kullanarak sadeleřtirin:

- (a) $F(x, y, z) = \Sigma(0, 2, 4, 5, 6)$
- (b) $F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$

Alıřtırma B.5. ★★ Quine–McCluskey algoritmasını kullanarak $F(a, b, c, d) = \Sigma(0, 2, 3, 5, 7, 8, 10, 11, 13, 15)$ fonksiyonunu sadeleřtirin. Ařama 1’de asal kapsayanları, Ařama 2’de temel asal kapsayanları belirleyin ve en yalın ifadeyi elde edin.

Alıřtırma B.6. ★★ 4 giriřli ve 16 çıkıřlı bir kod çözücü tasarlayın. Doęruluk tablosunu oluřturun.

Alıřtırma B.7. ★★ Bir yarım toplayıcının doęruluk tablosunu oluřturun ve S (toplam) ile C (elde) çıkıřları için Boole denklemlerini yazın. Devre řemasını çizin.

Alıřtırma B.8. ★★ Bir tam toplayıcının iki yarım toplayıcı ve bir VEYA kapısı kullanılarak nasıl gerçekteřtirileceęini gösterin. Baęlantı řemasını çizin ve doęruluk tablosuyla doęrulayın.

Alıřtırma B.9. ★★ 4 bitlik dalgalı elde toplayıcısı kullanarak $A = 1001$ ile $B = 0111$ sayılarını toplayın. Her basamaktaki A_i, B_i, C_i, S_i ve C_{i+1} deęerlerini tablo halinde gösterin. Sonucu hem ikilik hem de onluk tabanda doęrulayın.

Alıřtırma B.10. ★★ JK flip-flop kullanarak bir T flip-flopunu nasıl elde edebilirsiniz? Baęlantı řemasını çizin ve doęruluk tablosuyla doęrulayın.

Alıřtırma B.11. ★★ 4 bitlik bir kaydırmalı yazmaçta bařlangıç deęeri 1100 olsun. Sola dört kez kaydırma yapıldıęında (bořalan en saędaki bite 0 yazılarak) sırayla hangi deęerler elde edilir? Her adımdaki onluk tabandaki karřılıęı nedir?

Alıřtırma B.12. ★★★ Üç bitlik bir eřzamanlı yukarı sayıcıyı T flip-flopları kullanarak tasarlayın. Durum tablosunu, her bir flip-flopun T giriř denklemini ve devre řemasını çizin.

Alıştırma B.13. ★★★ 8×1 çoklayıcı kullanarak $F(A, B, C) = \Sigma(1, 2, 6, 7)$ fonksiyonunu gerçekleştirin. Bağlantı şemasını çizin.

Alıştırma B.14. ★★★ Dalgalı elde toplayıcısının en kötü durum gecikmesi neden $O(n)$ ile orantılıdır? 32 bitlik bir toplayıcıda her tam toplayıcının elde gecikmesi 2 ns ise en kötü durum toplam gecikmesini hesaplayın. Bu gecikme 1 GHz'lık bir saat sıklığı için kabul edilebilir midir?

Alıştırma B.15. ★★★ Bir trafik ışığı denetleyicisi tasarlayın. İki yönlü bir kavşakta Kuzey–Güney (KG) ve Doğu–Batı (DB) yönleri için kırmızı, sarı ve yeşil ışıkları denetleyen bir durum makinesi oluşturun. Geçerli durumları, geçişleri ve her durumdaki çıkışları belirleyerek durum diyagramını çizin.

RISC-V Referans Kartı



Efes Antik Kenti, Celsus Kütüphanesi – İzmir

Celsus Kütüphanesi, antik dünyanın en büyük bilgi kaynaklarından biriydi. Bu ek de RISC-V buyruk kümesinin başvuru kaynağıdır: yazarken, okurken ve hata ayıklarken her an dönüp bakılacak bir referans.

Öğrenme Hedefleri

Bu eki tamamladığınızda aşağıdakileri yapabileceksiniz:

- RISC-V'in modüler yapısını ve standart uzantılarını (M, A, F, D, C, V) açıklamak.
- Bir RISC-V buyruğunun ikili kodlamasını çözümleyerek buyruk biçimini (R, I, S, B, U, J) belirlemek.
- RV32I buyruk kümesindeki aritmetik, mantıksal, bellek erişimi ve dallanma buyruklarını sınıflandırmak.
- Yazmaç tablosunu (x0–x31) ve ABI adlandırma kurallarını kullanarak buyrukları yorumlamak.
- Sözde buyrukların gerçek buyruk karşılıklarını belirlemek.
- Verilen bir problemi RISC-V çevirici dili ile kodlamak.

Bu ekte, kitap boyunca kullanılan RISC-V buyruk kümesi mimarisinin özet referansı sunulmaktadır. Buyrukların ayrıntılı açıklaması Bölüm 3'te verilmiştir. İşlemci tasarımıdaki buyruk biçimi detayları için Bölüm 6'da, boru hattı düzeyindeki buyruk yürütme süreci için ise Bölüm 7'de başvurulabilir.

C.1 RISC-V'in Modüler Yapısı

RISC-V, tek bir sabit buyruk kümesi yerine **modüler** bir mimari sunar. Temel bir taban buyruk kümesi üzerine isteğe bağlı uzantılar eklenerek uygulama gereksinimlerine uygun bir işlemci yapılandırılabilir. Bu yaklaşım, gömülü denetleyicilerden yüksek başarımlı sunuculara kadar geniş bir yelpazede aynı mimarinin kullanılmasını mümkün kılar.

C.1.1 Taban Buyruk Kümeleri

RISC-V iki temel taban buyruk kümesi tanımlar:

- **RV32I:** 32 bitlik tamsayı taban kümesi. 32 adet genel amaçlı yazmaç (her biri 32 bit), 47 buyruk içerir. Gömülü sistemlerden masaüstü uygulamalara kadar geniş bir kullanım alanına sahiptir.
- **RV64I:** 64 bitlik tamsayı taban kümesi. RV32I'nin 64 bite genişletilmiş sürümüdür; yazmaçlar ve adres uzayı 64 bit genişliğindedir. Sunucu ve yüksek başarımlı hesaplama uygulamalarında tercih edilir.
- **RV32E:** RV32I'nin 16 yazmaçlı (x0–x15) alt kümesi. Kaynak kısıtlı mikrodenetleyiciler ve küçük gömülü sistemler için tasarlanmıştır; buyruk kümesi RV32I ile aynıdır, yalnızca yazmaç sayısı yarıya indirilmiştir.

C.1.2 Standart Uzantılar

Taban kümenin üzerine eklenen standart uzantılar, tek harfli kısaltmalarla adlandırılır. Tablo C.1, yaygın uzantıları özetlemektedir.

Tablo C.1: RISC-V standart uzantıları

Uzantı	Ad	Açıklama
M	Çarpma/Bölme	Tamsayı çarpma ve bölme buyrukları (mul, div, rem vb.)
A	Atomik İşlemler	Çok çekirdekli eşzamanlama için atomik bellek buyrukları (lr, sc, amo*)
F	Tek Duyarlık	IEEE 754 tek duyarlıklı (32 bit) kayan nokta buyrukları ve 32 adet kayan nokta yazmacı
D	Çift Duyarlık	IEEE 754 çift duyarlıklı (64 bit) kayan nokta buyrukları
C	Sıkıştırılmış	16 bitlik sıkıştırılmış buyruk kodlaması; kod yoğunluğunu artırır
V	Vektör	Veri düzeyinde koşutluk için değişken uzunlukta vektör buyrukları
Zicsr	DSY Erişimi	Denetim ve durum yazmaçlarına (DSY) erişim buyrukları
Zifencei	Buyruk Çiti	Buyruk belleği ile veri belleği arasında tutarlılık çiti

C.1.3 Yaygın Yapılandırmalar

Uzantılar birleştirilerek farklı uygulama alanlarına yönelik yapılandırmalar oluşturulur. **G** kısaltması “genel amaçlı” anlamına gelir ve **IMAFD_Zicsr_Zifencei** birleşimini temsil eder:

- **RV32I**: En yalın yapılandırma; küçük gömülü denetleyiciler için yeterlidir.
- **RV32IMC**: Gömülü sistemlerde yaygın; çarpma ve sıkıştırılmış buyruk desteği ekler.
- **RV32GC**: Genel amaçlı 32 bit; masaüstü ve uygulama işlemcileri için uygundur.
- **RV64GC**: Genel amaçlı 64 bit; Linux çalıştıran işlemcilerin ve sunucuların standart yapılandırmasıdır.

Bilgi Notu C.1: Bu kitapta kullanılan yapılandırma

Kitap boyunca **RV32I** taban buyruk kümesi kullanılmaktadır. RV32I, RISC-V mimarisinin temel kavramlarını (yazmaç işlemleri, bellek erişimi, dallanma, alt yordam çağrıları) öğrenmek için yeterli bir altyapı sunar. Çarpma/bölme (M), kayan nokta (F/D) ve atomik işlemler (A) gibi uzantılar ilgili bölümlerde kavramsal düzeyde ele alınmakta, ancak ayrıntılı buyruk tabloları bu ekin kapsamı dışında tutulmaktadır.

C.1.4 Ayricalık Düzeyleri

RISC-V, yazılım katmanlarını birbirinden yalıtmak için üç ayrıcalık düzeyi tanımlar:

Tablo C.2: RISC-V ayrıcalık düzeyleri

Düzyey	Kodlama	Ad	Açıklama
0	00	Kullanıcı (U)	Uygulama programlarının çalıştığı en düşük ayrıcalık düzeyi.
1	01	Gözetmen (S)	İşletim sistemi çekirdeğinin çalıştığı düzey; sayfa tabloları ve kesme yönetimi burada denetlenir.
3	11	Makine (M)	En yüksek ayrıcalık düzeyi; donanıma doğrudan erişim sağlar. Tüm RISC-V gerçekleştirmeleri en az bu düzeyi desteklemelidir.

En yalın gömülü sistemler yalnızca M modunda çalışabilir. İşletim sistemi gerektiren yapılandırmalarda M + U veya M + S + U birleşimi kullanılır. Ayrıcalık geçişleri `eca11` (üst düzeye çağrı) ve `mret/sret` (geri dönüş) buyrukları ile gerçekleştirilir.

C.2 Bir RISC-V Buyruğunun Anatomisi

RISC-V buyrukları, kitap boyunca çevirici (*assembly*) biçiminde yazılmaktadır. Buyrukları doğru okuyabilmek için temel yazım kurallarını bilmek gerekir.

C.2.1 Genel Yazım Kuralı

Bir RISC-V buyruğunun genel yapısı şöyledir:

buyruk hedef, kaynak1, kaynak2

Burada buyruk yapılacak işlemi, hedef sonucun yazılacağı yazmacı, kaynak1 ve kaynak2 ise işlenecek değerleri belirtir. Örneğin:

```
add x5, x6, x7    # x5 = x6 + x7
```

Bu buyrukta `add` toplama işlemini, `x5` hedef yazmacı, `x6` ve `x7` ise kaynak yazmaçları gösterir. RISC-V’te hedef her zaman en soldaki işlenendir.

C.2.2 İşlenen Türleri

Buyruk işlenenleri (*operand*) üç biçimde olabilir:

- **Yazmaç işleneni:** Doğrudan bir yazmaca başvurur. Yazmaçlar `x0–x31` veya takma adlarıyla (`t0`, `s0`, `a0` vb.) belirtilir. Yazmaç haritası Bölüm C.4’da verilmiştir.
- **Anlık değer (*immediate*):** Buyruğun içine gömülü sabit bir sayıdır; `imm` olarak gösterilir. Örneğin `addi x5, x6, 10` buyruğu `x6` yazmacına 10 ekleyip sonucu `x5`’e yazar.
- **Bellek adresi:** Yükleme ve saklama buyruklarında `offset` (taban) biçiminde yazılır. Aşağıda ayrıntılı olarak açıklanmaktadır.

C.2.3 Bellek Adresleme: Parantez Gösterimi

Yükleme (`lw`, `lb`, `lh` vb.) ve saklama (`sw`, `sb`, `sh` vb.) buyruklarında bellek adresi **taban yazmaç + offset** yöntemiyle hesaplanır. Çevirici dilinde bu, `offset`(`taban_yazmacı`) biçiminde yazılır:


```
lw x2, 100(x0)    # x2 = Bellek[x0 + 100]
sw x7, 150(x3)    # Bellek[x3 + 150] = x7
```

Burada parantez içindeki yazmaç **taban adresi**, parantez dışındaki sayı ise **ofset** değerini verir. Erişilen bellek adresi ikisinin toplamıdır. Örneğin $100(x0)$ ifadesinde $x0$ daima sıfır olduğundan adres doğrudan 100 olur; $150(x3)$ ifadesinde ise adres $x3 + 150$ olarak hesaplanır.

Bilgi Notu C.2: Yükleme ve saklama buyruklarındaki fark

Yükleme buyruklarında (lw) veri bellekten yazmaca *okunur*: hedef bir yazmaçtır. Saklama buyruklarında (sw) ise veri yazmaçtan belleğe *yazılır*: ilk işlenen bellekten değil, belleğe *gönderilecek* olan yazmacı gösterir.

C.2.4 Buyruk Kategorileri

RISC-V buyruklarını beş ana kategoride toplamak mümkündür (ayrıntılı açıklama Bölüm 3'te verilmektedir):

Tablo C.3: RISC-V buyruk kategorileri ve yazım örnekleri

Kategori	Örnek	Anlamı
Aritmetik/Mantık	add x5, x6, x7	$x5 = x6 + x7$
Anlık aritmetik	addi x5, x6, 10	$x5 = x6 + 10$
Yükleme	lw x2, 0(x3)	$x2 = \text{Bellek}[x3 + 0]$
Saklama	sw x7, 40(x3)	$\text{Bellek}[x3 + 40] = x7$
Dallanma	beq x5, x6, etiket	$x5 == x6$ ise etiket'e dallan
Atlama	jal x1, etiket	etiket'e atla, dönüş adresini x1'e yaz

Her buyruk kategorisi farklı bir ikili kodlama biçimi kullanır. Bu biçimler (R, I, S, B, U, J türleri) Bölüm C.5'da ayrıntılı olarak sunulmaktadır.

C.3 RV32I Temel Buyruk Kümesi

Tablo C.4, RV32I temel buyruk kümesindeki tüm buyrukları işlem kodlarıyla birlikte listelemektedir.

Tablo C.4: RV32I temel buyruk kümesi

Buyruk	Tür	İşlem Kodu	f3	f7	Açıklama
Aritmetik Buyruklar					
add rd, rs1, rs2	R	0110011	000	0000000	Toplama: $rd = rs1 + rs2$
sub rd, rs1, rs2	R	0110011	000	0100000	Çıkarma: $rd = rs1 - rs2$
addi rd, rs1, imm	I	0010011	000	—	Anlık toplama: $rd = rs1 + \text{imm}$
Mantık Buyrukları					
and rd, rs1, rs2	R	0110011	111	0000000	Bit düzeyi VE: $rd = rs1 \& rs2$

Sonraki sayfada devam ediyor

Tablo C.4: RV32I temel buyruk kümesi (devamı)

Buyruk	Tür	İşlem Kodu	f3	f7	Açıklama
or rd, rs1, rs2	R	0110011	110	0000000	Bit düzeyi VEYA: rd = rs1 rs2
xor rd, rs1, rs2	R	0110011	100	0000000	Bit düzeyi Dışlayan VEYA
andi rd, rs1, imm	I	0010011	111	—	Anlık VE: rd = rs1 & imm
ori rd, rs1, imm	I	0010011	110	—	Anlık VEYA: rd = rs1 imm
xori rd, rs1, imm	I	0010011	100	—	Anlık Dışlayan VEYA
Kaydırma Buyrukları					
sll rd, rs1, rs2	R	0110011	001	0000000	Sola mantıksal kaydırma
srl rd, rs1, rs2	R	0110011	101	0000000	Sağa mantıksal kaydırma
sra rd, rs1, rs2	R	0110011	101	0100000	Sağa aritmetik kaydırma
slli rd, rs1, shamt	I	0010011	001	0000000	Anlık sola mantıksal kaydırma
srli rd, rs1, shamt	I	0010011	101	0000000	Anlık sağa mantıksal kaydırma
srai rd, rs1, shamt	I	0010011	101	0100000	Anlık sağa aritmetik kaydırma
Karşılaştırma Buyrukları					
slt rd, rs1, rs2	R	0110011	010	0000000	İşaretli küçükse ata
sltu rd, rs1, rs2	R	0110011	011	0000000	İşaretsiz küçükse ata
slti rd, rs1, imm	I	0010011	010	—	Anlık işaretli küçükse ata
sltiu rd, rs1, imm	I	0010011	011	—	Anlık işaretsiz küçükse ata
Veri Aktarma Buyrukları					
lw rd, imm(rs1)	I	0000011	010	—	Sözcük yükle (32 bit)
lh rd, imm(rs1)	I	0000011	001	—	Yarı sözcük yükle (16 bit, işaretli)
lhu rd, imm(rs1)	I	0000011	101	—	Yarı sözcük yükle (16 bit, işaretsiz)
lb rd, imm(rs1)	I	0000011	000	—	Bayt yükle (8 bit, işaretli)
lbu rd, imm(rs1)	I	0000011	100	—	Bayt yükle (8 bit, işaretsiz)
sw rs2, imm(rs1)	S	0100011	010	—	Sözcük sakla (32 bit)
sh rs2, imm(rs1)	S	0100011	001	—	Yarı sözcük sakla (16 bit)
sb rs2, imm(rs1)	S	0100011	000	—	Bayt sakla (8 bit)
Dallanma Buyrukları					
beq rs1, rs2, eklenti	B	1100011	000	—	Eşitse dallan
bne rs1, rs2, eklenti	B	1100011	001	—	Eşit değilse dallan
blt rs1, rs2, eklenti	B	1100011	100	—	Küçükse dallan (işaretli)
bge rs1, rs2, eklenti	B	1100011	101	—	Büyük eşitse dallan (işaretli)
bltu rs1, rs2, eklenti	B	1100011	110	—	Küçükse dallan (işaretsiz)
bgeu rs1, rs2, eklenti	B	1100011	111	—	Büyük eşitse dallan (işaretsiz)
Atlama Buyrukları					
jal rd, eklenti	J	1101111	—	—	Atla ve bağla
jalr rd, rs1, imm	I	1101111	000	—	Yazmaçla atla ve bağla
Üst Anlık Buyruklar					
lui rd, imm	U	0110111	—	—	Üst anlık değer yükle
auipc rd, imm	U	0010111	—	—	Üst anlık değeri PC'ye ekle

Sonraki sayfada devam ediyor

Tablo C.4: RV32I temel buyruk kümesi (devamı)

Buyruk	Tür	İşlem Kodu	f3	f7	Açıklama
Bellek Sıralama Buyrukları					
fence pred, succ	I	0001111	000	—	Bellek sıralama çiti
Sistem Buyrukları					
ecall	I	1110011	000	0000000	Ortam çağırısı
ebreak	I	1110011	000	0000001	Hata ayıklama kesme noktası
DSY Erişim Buyrukları (Zicsr Uzantısı)					
csrrw rd, csr, rs1	I	1110011	001	—	DSY oku ve yaz
csrrs rd, csr, rs1	I	1110011	010	—	DSY oku ve bitleri kur
csrrc rd, csr, rs1	I	1110011	011	—	DSY oku ve bitleri sıfırla
csrrwi rd, csr, uimm	I	1110011	101	—	Anlık DSY yaz
csrrsi rd, csr, uimm	I	1110011	110	—	Anlık DSY bitleri kur
csrrci rd, csr, uimm	I	1110011	111	—	Anlık DSY bitleri sıfırla
Buyruk Çiti (Zifencei Uzantısı)					
fence.i	I	0001111	001	—	Buyruk belleği tutarlılık çiti

Not: “—” işaretleri, ilgili alanın o buyruk türünde kullanılmadığını göstermektedir. I türündeki kaydırma buyruklarında (slli, srli, srai) funct7 alanı, anlık değerin üst 7 bitini oluşturur ve kaydırma yönünü belirler.

C.3.1 Buyruk Kodlama Örneği

Bir buyruğun ikili kodlamasını somut bir örnek üzerinden inceleyelim. add x5, x6, x7 buyruğu R türündedir ve aşağıdaki gibi kodlanır:

funct7	rs2 (x7)	rs1 (x6)	f3	rd (x5)	opcode
0000000	00111	00110	000	00101	0110011

Alanlar soldan sağa: funct7 = 0000000 (toplama), rs2 = 00111 (x7), rs1 = 00110 (x6), funct3 = 000, rd = 00101 (x5), opcode = 0110011 (R türü aritmetik). Tüm alanlar birleştirildiğinde 32 bitlik buyruk sözcüğü elde edilir:

$$0000000001110011000000001010110011_2 = 0x007302B3$$

Bu kodlama düzeni Bölüm 3'te ayrıntılı olarak ele alınmaktadır.

C.4 Yazmaç Tablosu

RISC-V mimarisinde 32 adet genel amaçlı yazmaç bulunur. Her yazmaç 32 bit genişliğindedir (RV32I'de). Tablo C.5, tüm yazmaçları ABI (Uygulama İkili Arayüzü) adlarıyla birlikte listelemektedir.

Tablo C.5: *RISC-V RV32I yazmaç tablosu*

Yazmaç No	ABI Adı	Açıklama	Çağrıda Korunur mu?
x0	zero	Sabit sıfır	—
x1	ra	Dönüş adresi	Hayır
x2	sp	Yığıt göstergesi	Evet
x3	gp	Genel gösterge	—
x4	tp	İş parçacığı göstergesi	—
x5	t0	Geçici yazmaç	Hayır
x6	t1	Geçici yazmaç	Hayır
x7	t2	Geçici yazmaç	Hayır
x8	s0/fp	Kaydedilen yazmaç / Çerçeve göstergesi	Evet
x9	s1	Kaydedilen yazmaç	Evet
x10	a0	İşlev argümanı / dönüş değeri	Hayır
x11	a1	İşlev argümanı / dönüş değeri	Hayır
x12	a2	İşlev argümanı	Hayır
x13	a3	İşlev argümanı	Hayır
x14	a4	İşlev argümanı	Hayır
x15	a5	İşlev argümanı	Hayır
x16	a6	İşlev argümanı	Hayır
x17	a7	İşlev argümanı	Hayır
x18	s2	Kaydedilen yazmaç	Evet
x19	s3	Kaydedilen yazmaç	Evet
x20	s4	Kaydedilen yazmaç	Evet
x21	s5	Kaydedilen yazmaç	Evet
x22	s6	Kaydedilen yazmaç	Evet
x23	s7	Kaydedilen yazmaç	Evet
x24	s8	Kaydedilen yazmaç	Evet
x25	s9	Kaydedilen yazmaç	Evet
x26	s10	Kaydedilen yazmaç	Evet
x27	s11	Kaydedilen yazmaç	Evet
x28	t3	Geçici yazmaç	Hayır
x29	t4	Geçici yazmaç	Hayır
x30	t5	Geçici yazmaç	Hayır
x31	t6	Geçici yazmaç	Hayır

Not: “Çağrıda Korunur mu?” sütunu, bir alt yordam çağrısında yazmacın değerinin korunup korunmadığını belirtir. “Evet” olarak işaretlenen yazmaçlar, çağrılan işlev tarafından korunmalıdır (*callee-saved*). “Hayır” olarak işaretlenenler, çağıranın sorumluluğundadır (*caller-saved*). “—” işareti, yazmacın özel amaçlı olduğunu ve standart çağrı kurallarının dışında kaldığını gösterir. x0 yazmacı her zaman sıfır değerini döndürür ve yazma işlemleri göz ardı edilir.

C.4.1 Çağrı Kuralları ve Yığıt Çerçevesi

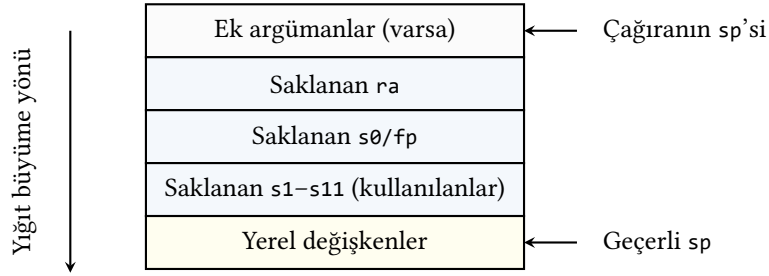
RISC-V çağrı kuralları, alt yordamlar arasında yazmaçların ve yığıtın nasıl kullanılacağını belirler. Bir alt yordam çağrısında aşağıdaki adımlar izlenir:

1. Çağırın (*caller*), a0–a7 yazmaçlarına argümanları yerleştirir (sekizden fazla argüman

yığıt üzerinden aktarılır).

2. jal ra, hedef buyruğu ile çağrı yapılır; dönüş adresi ra'ya kaydedilir.
3. Çağrılan (*callee*), kullanacağı kaydedilen yazmaçları (s0–s11) ve ra'yı yığıta saklar.
4. İşlev gövdesi yürütülür; dönüş değeri a0 (ve gerekiyorsa a1) yazmacına yazılır.
5. Saklanan yazmaçlar yığıttan geri yüklenir, ret ile çağırana dönülür.

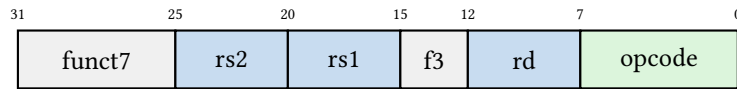
Aşağıdaki şekil, bir alt yordam çağrısı sırasında yığıt çerçevesinin yapısını göstermektedir:



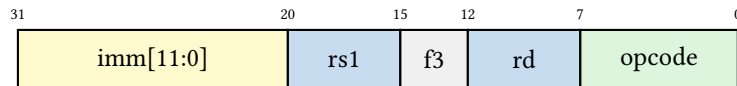
Şekil C.1: RISC-V alt yordam yığıt çerçevesinin yapısı

C.5 Buyruk Biçimleri

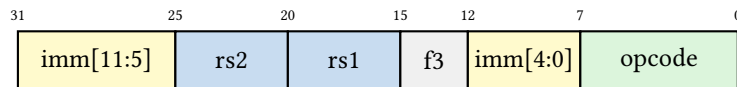
RISC-V, altı temel buyruk biçimi tanımlar. Her biçim 32 bit genişliğindedir ve farklı alan düzenleri içerir. Bu bölümde her biçim görsel olarak sunulmaktadır.



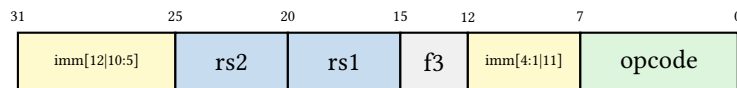
Şekil C.2: R türü buyruk biçimi



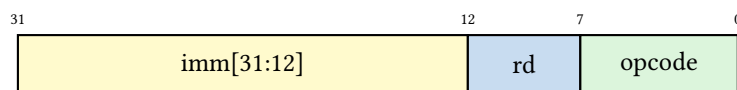
Şekil C.3: I türü buyruk biçimi



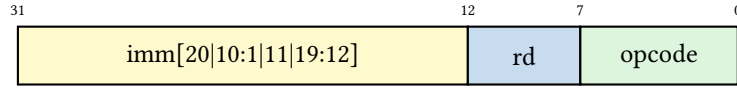
Şekil C.4: S türü buyruk biçimi



Şekil C.5: B türü buyruk biçimi



Şekil C.6: U türü buyruk biçimi

**Şekil C.7:** J türü buyruk biçimi

Şekillerde kullanılan renk kodları:

işlev kodu alanları	yazmaç alanları	işlem kodu	anlık değer alanları

Tablo C.6, her buyruk biçiminin hangi buyruk kategorilerinde kullanıldığını özetlemektedir.

Tablo C.6: Buyruk biçimleri ve kullanım alanları

Biçim	İşlem Kodu	Kullanan Buyruklar
R türü	0110011	Aritmetik ve mantık buyrukları (add, sub, and, or, xor, sll, srl, sra, slt, sltu)
I türü	0010011	Anlık aritmetik ve mantık (addi, andi, ori, xori, slti, sltiu, slli, srli, srai)
I türü	0000011	Yükleme buyrukları (lw, lh, lhu, lb, lbu)
I türü	1100111	Yazmaçla atlama (jalr)
I türü	1110011	Sistem buyrukları (ecall, ebreak)
S türü	0100011	Saklama buyrukları (sw, sh, sb)
B türü	1100011	Dallanma buyrukları (beq, bne, blt, bge, bltu, bgeu)
U türü	0110111	Üst anlık yükleme (lui)
U türü	0010111	Üst anlık PC ekleme (auipc)
J türü	1101111	Koşulsuz atlama (jal)

C.6 Sözde Buyruklar

RISC-V çeviricisi (*assembler*), programcılarının işini kolaylaştırmak için birçok *sözde buyruk* tanımlar. Bu buyruklar doğrudan donanım tarafından desteklenmez; bunun yerine çevirici tarafından bir veya daha fazla gerçek buyruğa dönüştürülür. Tablo C.7, yaygın kullanılan sözde buyrukları listelemektedir.

Tablo C.7: RISC-V sözde buyrukları

Sözde Buyruk	Açılımı	Açıklama
Temel Sözde Buyruklar		
nop	addi zero, zero, 0	İşlem yok
li rd, imm	lui + addi dizisi	Anlık değer yükleme
la rd, sembol	auipc rd, sembol[31:12] + addi rd, rd, sembol[11:0]	Adres yükleme

Sonraki sayfada devam ediyor

Tablo C.7: RISC-V sözde buyrukları (devamı)

Sözde Buyruk	Açılımı	Açıklama
mv rd, rs	addi rd, rs, 0	Yazmaç kopyalama
not rd, rs	xori rd, rs, -1	Bit tersleme
neg rd, rs	sub rd, zero, rs	İşaret değiştirme
Atlama ve Dönüş Sözde Buyrukları		
j eklenti	jal zero, eklenti	Koşulsuz atlama
jr rs	jalr zero, rs, 0	Yazmaçla atlama
ret	jalr zero, ra, 0	Alt yordamdan dönüş
call eklenti	auipc ra, eklenti[31:12] + jalr ra, ra, eklenti[11:0]	Alt yordam çağırısı
tail eklenti	auipc t1, eklenti[31:12] + jalr zero, t1, eklenti[11:0]	Kuyruk çağırısı
Dallanma Sözde Buyrukları		
beqz rs, eklenti	beq rs, zero, eklenti	Sıfıra eşitse dallan
bnez rs, eklenti	bne rs, zero, eklenti	Sıfıra eşit değilse dallan
bgt rs, rt, eklenti	blt rt, rs, eklenti	Büyükse dallan
ble rs, rt, eklenti	bge rt, rs, eklenti	Küçük eşitse dallan
bgtu rs, rt, eklenti	bltu rt, rs, eklenti	İşaretsiz büyükse dallan
bleu rs, rt, eklenti	bgeu rt, rs, eklenti	İşaretsiz küçük eşitse dallan

Not: 1i buyruğunun açılımı, anlık değerin büyüklüğüne bağlı olarak değişir. Değer 12 bitlik işaretli aralığa sığıyorsa yalnızca addi kullanılır. Aksi hâlde önce lui ile üst 20 bit yüklenir, ardından addi ile alt 12 bit eklenir. Benzer şekilde 1a, call ve tail buyrukları da iki buyruktan oluşur: birincisi üst kısımları auipc ile hesaplar, ikincisi alt kısımları ekler.

C.7 Kodlama Örnekleri

Bu bölümde, Bölüm 3'te ele alınan kavramları pekiştirmeye yönelik dört uçtan uca örnek sunulmaktadır. Her örnekte C kaynak kodu, karşılık gelen RISC-V çevirici kodu ve açıklama bir arada verilmektedir.

Örnek C.1

int c = (a + b) * 2; ifadesini RISC-V buyruklarıyla gerçekleştirin. a değeri s1, b değeri s2 yazmacındadır; sonucu s3'e yazın.

Çözüm:

```
1 add s3, s1, s2      # s3 = a + b
2 slli s3, s3, 1      # s3 = (a + b) * 2 (1 bit sola kaydır = x2)
```

Derleyici, 2'nin kuvvetleriyle yapılan çarpmaları çoğunlukla slli (sola kaydırma) buy-

ruğuyla gerçekleştirir; bu, mul buyruğundan daha az donanım kaynağı kullanır ve M uzantısı gerektirmez.

Örnek C.2

Aşağıdaki C döngüsünü RISC-V buyruklarıyla gerçekleştirin:

```
1  int sum = 0;
2  for (int i = 0; i < 10; i++)
3      sum += arr[i];
```

arr dizisinin başlangıç adresi s1, sum değişkeni s2, sayaç s3 yazmacındadır.

Çözüm:

```
1      addi s2, zero, 0      # sum = 0
2      addi s3, zero, 0      # i = 0
3      addi t0, zero, 10     # sinir = 10
4  don:
5      bge s3, t0, cik        # i >= 10 ise dongudan cik
6      slli t1, s3, 2         # t1 = i * 4 (bayt offseti)
7      add t1, s1, t1         # t1 = arr[i] adresi
8      lw t2, 0(t1)          # t2 = arr[i]
9      add s2, s2, t2         # sum += arr[i]
10     addi s3, s3, 1         # i++
11     jal zero, don          # dongiye don
12 cik:
```

Her int değer 4 bayt kapladığından, dizi indisi slli ile 2 bit sola kaydırılarak bayt ofsetine dönüştürülür. bge (büyük veya eşitse dallan) döngü koşulunu sınar; koşul sağlandığında döngü sonrasına atlar. jal zero, don koşulsuz geri atlama için kullanılır; hedef yazmaç olarak zero verildiğinde dönüş adresi saklanmaz.

Örnek C.3

Aşağıdaki C koşulunu RISC-V buyruklarıyla gerçekleştirin:

```
1  if (a > b)
2      c = a - b;
3  else
4      c = b - a;
```

a değeri s1, b değeri s2, c değeri s3 yazmacındadır.

Çözüm:

```
1      bge s2, s1, else_dal  # b >= a ise else dalina atla
2      sub s3, s1, s2        # c = a - b (if dali)
3      jal zero, son         # if blogunun sonuna atla
4  else_dal:
5      sub s3, s2, s1        # c = b - a (else dali)
6  son:
```

Koşul ters çevrilerek yazılır: a > b yerine b >= a sınanır; koşul doğruysa else dalına atlanır. Bu yöntem, çevirici dilde koşullu yapıların standart gerçekleştirme kalıbıdır. Çevirici

rici dilinde b1e (küçük veya eşitse dallan) sözde buyruğu da kullanılabilir; çevirici bunu bge ile yer değiştirmiş kaynaklara çevirir.

Örnek C.4

Aşağıdaki C işlevini RISC-V çevirici dilinde yazın ve çağırın:

```
1 int toplam(int a, int b) {
2     return a + b;
3 }
4 // main'de:
5 int sonuc = toplam(3, 5);
```

Çözüm: Çağırın taraf (main)

```
1     addi a0, zero, 3      # birinci arguman: a = 3
2     addi a1, zero, 5      # ikinci arguman: b = 5
3     jal  ra, toplam       # toplam islevini cagir
4     mv   s0, a0           # sonuc = donus degeri
```

Çözüm: İşlev gövdesi (toplam)

```
1 toplam:
2     addi sp, sp, -8       # yigit cercevesi ac (8 bayt)
3     sw   ra, 4(sp)        # donus adresini sakla
4     sw   s0, 0(sp)        # kaydedilen yazmaci sakla
5     add  a0, a0, a1       # donus degeri = a + b
6     lw   s0, 0(sp)        # kaydedilen yazmaci geri yukle
7     lw   ra, 4(sp)        # donus adresini geri yukle
8     addi sp, sp, 8        # yigit cercevesini kapat
9     ret                  # cagirana don (jalr zero, 0(ra))
```

RISC-V çağrı kurallarına göre argümanlar a0–a7 yazmaçlarıyla aktarılır; dönüş değeri a0'a yazılır. İşlev içinde kullanılan kaydedilen yazmaçlar (s0–s11) ve dönüş adresi (ra) yığıta saklanıp geri yüklenmek zorundadır. ret sözde buyruğu, çevirici tarafından jalr zero, 0(ra) olarak açılır.

Özet

Bu ekte RISC-V buyruk kümesi mimarisinin referans bilgileri sunuldu. Temel kavramlar şöyle özetlenebilir:

- RISC-V modüler bir mimaridir: RV32I/RV64I taban kümeleri üzerine M, A, F, D, C, V gibi uzantılar eklenerek farklı uygulama alanlarına uygun yapılandırmalar oluşturulur. Bu kitap RV32I taban kümesini kullanmaktadır.
- RISC-V üç ayrıcalık düzeyi tanımlar: Makine (M), Gözetmen (S) ve Kullanıcı (U). Tüm gerçeklemeler en az M modunu destekler.
- Buyruklar buyruk hedef, kaynak1, kaynak2 biçiminde yazılır; hedef her zaman en soldaki işlenendir. Bellek erişiminde ofset(taban) gösterimi kullanılır.
- RV32I, 47 buyruktan oluşur: aritmetik/mantık, veri aktarma, dallanma, atlama, üst anlık, bellek sıralama ve sistem buyrukları. Zicsr ve Zifencei uzantıları DSY erişimi ve buyruk

çiti desteği ekler.

- 32 genel amaçlı yazmaç bulunur ($x0-x31$); $x0$ daima sıfırdır. Çağrı kurallarına göre yazmaçlar caller-saved ve callee-saved olarak ikiye ayrılır.
- Altı buyruk biçimi (R, I, S, B, U, J) tüm buyrukları 32 bitlik sabit genişlikte kodlar. Sözde buyruklar ise çevirici tarafından gerçek buyruklara dönüştürülerek programcının işini kolaylaştırır.

Alıştırmalar

Alıştırma C.1. ★ Aşağıdaki RISC-V buyruk dizisinin yürütülmesinden sonra $x5$, $x6$ ve $x7$ yazmaçlarının değerlerini bulun. Başlangıçta tüm yazmaçlar sıfırdır.

```
addi x5, x0, 15
addi x6, x0, 10
add  x7, x5, x6
sub  x5, x7, x6
```

Alıştırma C.2. ★ `slli x5, x6, 3` buyruğu ne işlem yapar? $x6 = 5$ ise $x5$ 'in değeri ne olur? Aynı işlemi aritmetik olarak ifade edin.

Alıştırma C.3. ★ Aşağıdaki sözde buyrukların gerçek RISC-V buyruklarına açılımını yazın:

- `mv x5, x6`
- `li x7, 42`
- `bnez x5, etiket`
- `ret`

Alıştırma C.4. ★★ `add x5, x6, x7` buyruğunun 32 bitlik R türü kodlamasını ikilik ve onaltılık tabanda yazın. Her alanı (`funct7`, `rs2`, `rs1`, `funct3`, `rd`, `opcode`) ayrı ayrı belirtin.

Alıştırma C.5. ★★ `lw x10, 8(x2)` buyruğu I türündedir. Bu buyruğun 32 bitlik kodlamasını oluşturun. Ofset değeri olan 8'in 12 bitlik işaretli anlık değer olarak nasıl kodlandığını gösterin.

Alıştırma C.6. ★★ Aşağıdaki program parçası bir döngü gerçeklemektedir. Döngü kaç kez çalışır ve sonuçta $x6$ 'nın değeri ne olur?

```
addi x5, x0, 5      # x5 = 5 (sayac)
addi x6, x0, 0      # x6 = 0 (toplam)
don:
add  x6, x6, x5      # toplam += sayac
addi x5, x5, -1      # sayac--
bne  x5, x0, don     # sayac != 0 ise tekrarla
```

Alıştırma C.7. ★★ RISC-V çağrı kurallarına göre bir alt yordam çağrısında hangi yazmaçlar çağırılan tarafından, hangileri çağırılan tarafından korunur? $a0-a7$ ve $s0-s11$ yazmaçlarını bu açıdan sınıflandırın.

Alıştırma C.8. ★★★ beq x5, x6, -16 buyruğunun 32 bitlik B türü kodlamasını oluşturun. Eklenti değeri -16'nın B türü anlık değer alanlarına (imm[12|10:5] ve imm[4:1|11]) nasıl dağıtıldığını adım adım gösterin.

Alıştırma C.9. ★★★ RV32I'de 32 bitlik bir sabiti bir yazmaca yüklemek neden iki buyruk gerektirir? lui ve addi buyrukları kullanarak 0xDEADBEEF sabitini x10 yazmacına yükleyen buyruk dizisini yazın. İşaret genişletmesi nedeniyle alt 12 bitte düzeltme gerekip gerekmediğini tartışın.

RISC-V Uygulama Çalışmaları ve Projeler



Sümela Manastırı – Trabzon

Sümela Manastırı, sarp kayalıklara tutunarak yüzyıllar boyunca ayakta kalmıştır. Bu ekteki uygulamalar da öğrenilen kuramsal bilgiyi somut projelere dönüştürerek kalıcı kılar.

Öğrenme Hedefleri

Bu eki tamamladığınızda aşağıdakileri yapabileceksiniz:

- RISC-V araç zincirini (RARS, Venus, Spike) kurmak ve kullanmak.
- Temel Verilog yapılarını (modül, atama, always bloğu) kullanarak basit sayısal devreler tasarlamak.
- Bir yazmaç öbeğini Verilog ile gerçekleştirmek ve benzetimini yapmak.
- Çevirici dilde yazılmış bir programı benzetimlikte adım adım çalıştırarak hata ayıklamak.
- Tek vuruşluk bir RISC-V işlemcisini Verilog modülleriyle birleştirmek.
- Proje çıktılarını belirlenen değerlendirme ölçütlerine göre sınamak.

Bu ekte, kitap boyunca öğrenilen kavramları uygulamaya dönüştürmek için gerekli araçlar, temel Verilog bilgisi ve üç aşamalı bir proje seti sunulmaktadır. Amaç, öğrencilerin hem yazılım hem de donanım düzeyinde bilgisayar mimarisini deneyimlemesidir.

D.1 RISC-V Araç Zinciri

RISC-V programlarını yazmak, derlemek ve çalıştırmak için çeşitli açık kaynaklı araçlar mevcuttur. Bu bölümde, ders kapsamında kullanılacak benzetimlikler ve derleyici araç zinciri tanıtılmaktadır.

D.1.1 Benzetimlikler

Gerçek bir RISC-V donanımı olmadan program geliştirmek için benzetimlikler kullanılır. Tablo D.1, yaygın kullanılan benzetimlikleri karşılaştırmaktadır.

Tablo D.1: RISC-V benzetimliklerinin karşılaştırması

Araç	Tür	Arayüz	Uygun Kullanım
RARS	Fonksiyonel	Grafiksel (GUI)	Başlangıç, ders içi gösterim
Venus	Fonksiyonel	Tarayıcı tabanlı	Kurulum gerektirmeyen hızlı deneme
Spike	Fonksiyonel	Komut satırı	İleri düzey, ayrıcalıklı mod
QEMU	Sistem	Komut satırı	İşletim sistemi çalıştırma

RARS (RISC-V Assembler and Runtime Simulator), Java tabanlı bir grafiksel benzetimliklerdir. Yazmaç ve bellek içeriklerini anlık olarak gösterir, adım adım çalıştırma ve kesme noktası (breakpoint) desteği sunar. Başlangıç düzeyindeki öğrenciler için en uygun araçtır. RARS, GitHub üzerinden açık kaynak olarak dağıtılmaktadır; .jar dosyası indirilerek Java yüklü herhangi bir sistemde çalıştırılabilir.

Venus tarayıcı üzerinde çalışan bir RISC-V benzetimliğidir. Herhangi bir kurulum gerektirmeden doğrudan venus.cs61c.org adresinden kullanılabilir. Öğrencilerin hızlı denemeler yapması için idealdir.

Spike ise RISC-V International tarafından geliştirilen referans benzetimliğidir. Ayrıcalıklı buyrukları ve sanal bellek mekanizmalarını da destekler; bu nedenle işletim sistemi düzeyinde çalışmalar için tercih edilir. Linux ortamında kaynak koddan derlenerek kurulur.

Bilgi Notu D.1: Hızlı Başlangıç

Kurulum kolaylığı açısından **RARS** önerilir: Java 8 veya üzeri yüklü bir sistemde `java -jar rars.jar` komutuyla çalışır. Kurulum gerektirmeyen bir seçenek olarak **Venus** tarrayıcı üzerinden doğrudan kullanılabilir. Her iki araç da RV32I buyruk kümesini tam olarak destekler.

Aşağıdaki kısa program, RARS veya Venus'te doğrudan çalıştırılabilir. Program, ilk n Fibonacci sayısını hesaplayarak belleğe yazar; yazmaç ve bellek değişimlerini adım adım izleyerek buyruk yürütüm döngüsünü gözlemlemek için uygundur.

```

1  # fibonacci.s -- İlk 10 Fibonacci sayisini bellek dizisine yazar
2  # RARS: java -jar rars.jar fibonacci.s
3  # Venus: Kodu yapistirip "Run" tusuna basin
4
5  .data
6  dizi:  .space 40          # 10 x 4 bayt alan ayir
7
8  .text
9  .globl main
10 main:
11      li    t0, 10          # n = 10 (kac sayi)
12      la    t1, dizi        # dizi baslangic adresi
13      li    t2, 0           # F(0) = 0
14      li    t3, 1           # F(1) = 1
15      sw    t2, 0(t1)       # dizi[0] = 0
16      sw    t3, 4(t1)       # dizi[1] = 1
17      addi  t1, t1, 8        # sonraki adres
18      addi  t0, t0, -2       # 2 eleman zaten yazildi
19
20  dongu:
21      beqz  t0, bitti        # kalan 0 ise cik
22      add   t4, t2, t3        # t4 = F(k-2) + F(k-1)
23      sw    t4, 0(t1)        # diziyeye yaz
24      mv    t2, t3           # F(k-2) = F(k-1)
25      mv    t3, t4           # F(k-1) = F(k)
26      addi  t1, t1, 4        # adresi ilerlet
27      addi  t0, t0, -1       # sayac azalt
28      j     dongu
29
30  bitti:
31      li    a7, 10          # ecall 10 = programi bitir
32      ecall

```

Kod D.1: Fibonacci sayıları (RARS/Venus ile çalıştırılabilir)

D.1.2 GNU Araç Zinciri

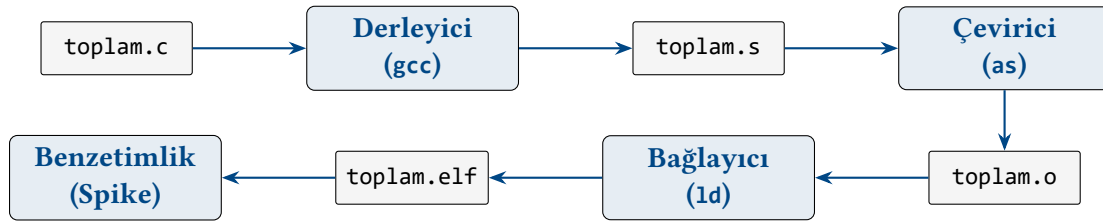
RISC-V GNU araç zinciri, C/C++ kodunu RISC-V makine koduna derlemek için kullanılan açık kaynaklı derleyici paketidir. Temel bileşenleri Tablo D.2'da listelenmiştir.

Tablo D.2: RISC-V GNU araç zinciri bileşenleri

Araç	Komut	Görevi
Derleyici	riscv64-unknown-elf-gcc	C/C++ → çevirici dil → makine kodu
Çevirici	riscv64-unknown-elf-as	Çevirici dil → nesne dosyası
Bağlayıcı	riscv64-unknown-elf-ld	Nesne dosyalarını birleştirme
Tersine çevirici	riscv64-unknown-elf-objdump	Makine kodu → çevirici dil

D.1.3 Derleme–Bağlama–Çalıştırma Akışı

Bir C programının RISC-V üzerinde çalışması için izlenen adımlar Şekil D.1’nde gösterilmiştir.

**Şekil D.1:** C programının RISC-V üzerinde derleme–bağlama–çalıştırma akışı

Bilgi Notu D.2: Tek Adımda Derleme

riscv64-unknown-elf-gcc -o toplam toplam.c komutu, derleme, çevirme ve bağlama adımlarını tek seferde gerçekleştirir. Ara dosyaları görmek için -S (çevirici dil çıktısı) ve -c (nesne dosyası) seçenekleri kullanılır.

Derlenen bir programın çevirici dil karşılığını görmek için tersine çevirici kullanılır:

```

1 # riscv64-unknown-elf-objdump -d toplam.o
2 00000000 <toplam>:
3 0: 00b50533 add a0, a0, a1 # a0 = a0 + a1
4 4: 00008067 ret # ra'ya don

```

Kod D.2: Tersine çevirici çıktısı örneği

D.2 Verilog ile Donanım Tasarımı

Verilog, sayısal devreleri tanımlamak ve benzetmek için kullanılan bir donanım tanımlama dilidir (HDL). Bu bölümde, RISC-V işlemci projelerinde kullanılacak kadar temel Verilog bilgisi sunulmaktadır. Ayrıntılı Verilog öğrenimi için IEEE 1364 standardına ve Ek B’deki mantıksal tasarım temellerine başvurulabilir.

Verilog’un güncel sürümü olan **SystemVerilog** (IEEE 1800), dil yapısını genişleterek logic veri tipi, always_comb/always_ff blokları, interface ve doğrulama amaçlı sınıf tabanlı yapılar gibi özellikler sunar. Endüstride SystemVerilog giderek daha yaygın kullanılmakla birlikte, bu ekteki örnekler taşınabilirlik ve basitlik amacıyla klasik Verilog söz dizimiyle verilmiştir.

D.2.1 Modül Yapısı

Verilog'da temel tasarım birimi *modül*dür. Her modülün giriş/çıkış portları ve iç mantığı vardır.

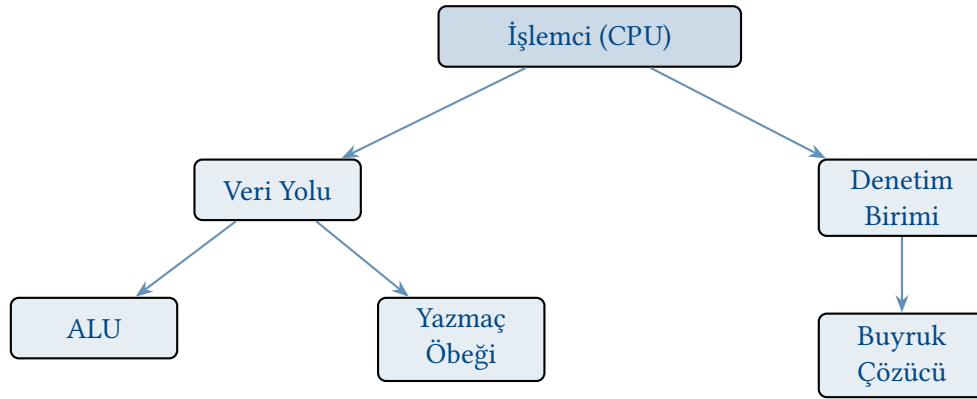
```

1 module ve_kapisi (
2     input wire a,      // giris 1
3     input wire b,      // giris 2
4     output wire y      // cikis
5 );
6     assign y = a & b; // surekli atama
7 endmodule

```

Kod D.3: Verilog modül yapısı: 2 girişli VE kapısı

module ile endmodule arasındaki blok, devrenin tanımını oluşturur. input ve output anahtar sözcükleri port yönünü belirler. Verilog'da karmaşık devreleri küçük modüllere ayırarak hiyerarşik bir yapı kurmak temel tasarım ilkesidir. Şekil D.2'de basit bir işlemci veri yolunun modül hiyerarşisi gösterilmektedir.



Şekil D.2: Basit bir işlemci tasarımında Verilog modül hiyerarşisi. Her kutu ayrı bir module olarak tanımlanır ve üst modül alt modülleri örnekleyerek (instantiate) kullanır.

D.2.2 Veri Tipleri: wire ve reg

Verilog'da iki temel veri tipi vardır:

- **wire:** Bileşimsel (kombinasyonel) mantıkta kullanılır. Fiziksel bir teli temsil eder; assign ile sürekli atama yapılır.
- **reg:** Sıralı (ardışıl) mantıkta ve always bloklarında kullanılır. Adına rağmen her zaman bir yazmaç anlamına gelmez; sentez aracının kullanım bağlamına göre karar verdiği bir depolama elemanıdır.

Çok bitli sinyaller köşeli parantezle tanımlanır: wire [31:0] veri ifadesi 32 bitlik bir sinyal tanımlar.

D.2.3 Sürekli Atama ve always Blokları

Bileşimsel mantık, assign ile veya duyarlılık listeli always bloğuyla tanımlanır. Sıralı mantık ise saat kenarına duyarlı always bloğu gerektirir.

```

1 // Bilesimsel: 2'ye 1 coklayici
2 module mux2 (
3     input wire [31:0] a, b,
4     input wire      sec,

```

```

5     output reg [31:0] y
6 );
7     always @(*) begin
8         if (sec)
9             y = b;
10        else
11            y = a;
12    end
13 endmodule
14
15 // ıSiral: D flip-flop (saat yükselisinde)
16 module yazmaç (
17     input wire      clk, rst,
18     input wire [31:0] d,
19     output reg [31:0] q
20 );
21     always @(posedge clk) begin
22         if (rst)
23             q <= 32'b0;
24         else
25             q <= d;
26     end
27 endmodule

```

Kod D.4: Bileşimsel ve sıralı mantık örnekleri**Bilgi Notu D.3: Bloklı ve Bloksuz Atama**

Bileşimsel always bloklarında = (bloklı atama), sıralı bloklarda <= (bloksuz atama) kullanılır. Bu kural, sentez ve benzetim tutarlılığı için can alıcıdır.

D.2.4 Doğrulama Ortamı Yazma

Doğrulama ortamı (*testbench*), tasarlanan modülü doğrulamak için yazılan benzetim ortamıdır. Doğrulama ortamı modülünün giriş/çıkış portu yoktur; test edilen modülü içsel olarak örnekler (instantiate).

```

1  `timescale 1ns / 1ps
2
3  module mux2_tb;
4      reg [31:0] a, b;
5      reg      sec;
6      wire [31:0] y;
7
8      // Test edilen modul
9      mux2 uut (.a(a), .b(b), .sec(sec), .y(y));
10
11     initial begin
12         a = 32'd10; b = 32'd20; sec = 0;
13         #10; // 10 ns bekle
14         $display("sec=0 -> y=%0d (beklenen: 10)", y);
15
16         sec = 1;
17         #10;
18         $display("sec=1 -> y=%0d (beklenen: 20)", y);

```

```

19
20     $finish;
21 end
22 endmodule

```

Kod D.5: 2'ye 1 çoklayıcı doğrulama ortamı örneği

Benzetim araçları olarak açık kaynaklı Icarus Verilog (iverilog) ve dalga formu görüntüleyici GTKWave kullanılabilir:

```

iverilog -o mux2_sim mux2.v mux2_tb.v
vvp mux2_sim
gtkwave dalga.vcd

```

Kod D.6: Icarus Verilog ile benzetim

Bilgi Notu D.4: FPGA ile Gerçekleme

Bu ektteki Verilog tasarımları, bir FPGA geliştirme kartı üzerinde sentezlenerek gerçek donanımda çalıştırılabilir. Eğitim amaçlı yaygın kullanılan kartlar arasında Xilinx (AMD) Basys 3 ve Digilent Nexys serisi, Intel (Altera) DE10-Lite ve Terasic DE0-Nano sayılabilir. Sentez ve yerleştirme için Xilinx Vivado veya Intel Quartus Prime araçları kullanılır. İşlemci tasarımını FPGA üzerinde gerçeklemek, zamanlama kısıtları, kaynak kullanımı ve fiziksel tasarım kavramlarını deneyimleme fırsatı sunar.

D.3 Yazmaç Öbeği Tasarımı

Bölüm 6'da öğrenilen yazmaç öbeğinin Verilog ile gerçekleştirilişi, işlemci projeleri için temel bir yapı taşıdır. Aşağıdaki modül, RISC-V'in 32 adet 32 bitlik yazmaç öbeğini tanımlar: iki okuma portu ve bir yazma portu içerir; x0 yazmacı her zaman sıfırdır.

```

1  module yazmac_dosyasi (
2      input wire      clk,
3      input wire      yaz_etkin,    // yazma izni
4      input wire [4:0] oku_adr1,    // rs1 adresi
5      input wire [4:0] oku_adr2,    // rs2 adresi
6      input wire [4:0] yaz_adr,     // rd adresi
7      input wire [31:0] yaz_veri,   // yazılacak deger
8      output wire [31:0] oku_veri1, // rs1 degeri
9      output wire [31:0] oku_veri2, // rs2 degeri
10 );
11     reg [31:0] yazmaclar [0:31];
12
13     // Okuma (bilesimsel) -- x0 her zaman 0
14     assign oku_veri1 = (oku_adr1 == 5'b0) ? 32'b0
15                                     : yazmaclar[oku_adr1];
16     assign oku_veri2 = (oku_adr2 == 5'b0) ? 32'b0
17                                     : yazmaclar[oku_adr2];
18
19     // Yazma (saat yukselisinde) -- x0'a yazma engeli
20     always @(posedge clk) begin
21         if (yaz_etkin && yaz_adr != 5'b0)
22             yazmaclar[yaz_adr] <= yaz_veri;

```

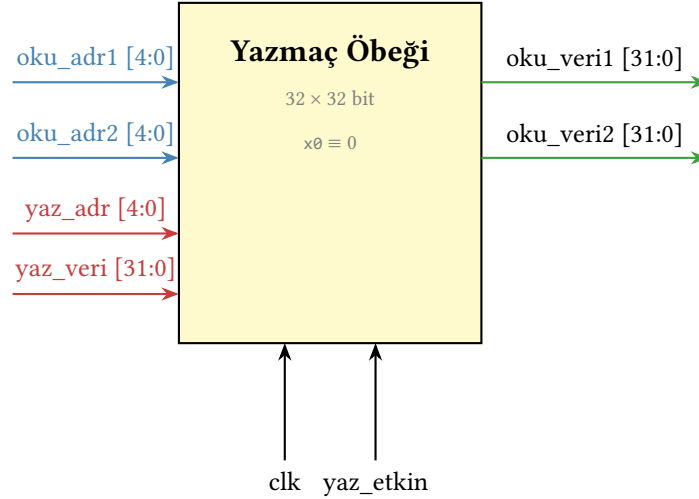
```

23     end
24 endmodule

```

Kod D.7: RISC-V yazmaç öbeği (32×32)

Bu modül, Bölüm 6'daki tek çevrimli veriyolu şemasının doğrudan Verilog karşılığıdır. Okuma portları bileşimsel (assign), yazma portu ise saat kenarına bağlıdır. Şekil D.3'da bu modülün blok diyagramı gösterilmektedir.



Şekil D.3: Yazmaç öbeği blok diyagramı. İki okuma portu bileşimsel (saat bağımsız), yazma portu saat kenarına bağlıdır. $x0$ yazmacı donanım tarafından sıfıra sabitlenmiştir.

D.4 Mini Projeler

Bu bölümde, kitabın farklı konularını pekiştiren üç proje sunulmaktadır. Projeler artan karmaşıklık düzeyinde tasarlanmıştır; her biri bir dönem ödevi veya laboratuvar çalışması olarak kullanılabilir.

Tablo D.3: Mini projelerin özeti

Proje	Düzey	İlgili Bölümler	Süre
A RV32I Yorumlayıcı	Başlangıç	3, 4, Ek C	2–3 hafta
B Boru Hattı Benzetimliği	Orta	5, 6	3–4 hafta
C Önbellek ve Dallanma Öngörücüsü	İleri	6, 7	4–5 hafta

D.4.1 Proje A: RV32I Yorumlayıcı

Amaç: C veya Python ile yazılan bir yazılım yorumlayıcısı aracılığıyla RV32I buyruk kümesinin çalışma mantığını kavramak.

Gereksinimler:

- 32 adet 32 bitlik yazmaç ve en az 4 KiB bellek alanı tanımlayın.
- İkilik dosyadan (binary) buyrukları yükleyin.
- Her buyruk için getir–çöz–yürüt döngüsünü gerçekleyin:

- a) **Getir:** Buyruk sayacının gösterdiği adresten 4 bayt okuyun.
 - b) **Çöz:** İşlem kodu, funct3, funct7 alanlarını çıkarın; buyruk türünü (R/I/S/B/U/J) belirleyin.
 - c) **Yürüt:** ALU işlemini, bellek erişimini veya dallanmayı gerçekleştirin.
4. ecall buyrukunda a7 yazmacına göre basit sistem çağrılarını destekleyin (örn. tam sayı yazdırma, program sonlandırma).
 5. Program sonunda yazmaç ve bellek dökümünü (dump) yazdırın.

Test stratejisi: RARS'ta çalıştırılan küçük programlarla (toplama, döngü, özyinelemeli fonksiyon) yazmaç değerlerini karşılaştırın.

Bilgi Notu D.5: Buyruk Çözme İpucu

RV32I buyruk biçimleri sabit genişliktedir (32 bit). Bölüm 3'teki biçim tablosunu referans alarak bit maskeleme ile alanları çıkarabilirsiniz. Örneğin opcode alanı buyruk & 0x7F ile elde edilir.

D.4.2 Proje B: Boru Hattı Benzetimliği

Amaç: Bölüm 7'deki beş aşamalı boru hattını (IF-ID-EX-MEM-WB) yazılımla benzetim yaparak veri ve denetim tehlikelerini gözlemlemek.

Gereksinimler:

1. Proje A'daki yorumlayıcıyı beş aşamalı boru hattı modeline dönüştürün.
2. Aşamalar arası boru hattı yazmaçlarını (IF/ID, ID/EX, EX/MEM, MEM/WB) modelleyin.
3. Aşağıdaki tehlike çözüm tekniklerini gerçekleştirin:
 - **Veri tehlikesi:** İletme (forwarding) birimi ve gerektiğinde durdurma (stall) mantığı.
 - **Denetim tehlikesi:** Dallanmada basit “her zaman dallanma yok” öngörüsü; yanlış öngöründe boru hattını boşaltma (flush).
4. Her çevrimde boru hattı durumunu yazdıran izleme (trace) çıktısı üretin.
5. Program sonunda aşağıdaki istatistikleri raporlayın:
 - Toplam çevrim sayısı ve BBÇ (Buyruk Başına Çevrim)
 - İletme sayısı, durdurma sayısı, boşaltma sayısı
 - Tehlike türlerine göre dağılım

Örnek çıktı formatı:

Cevrim	IF	ID	EX	MEM	WB
1	add x1..	---	---	---	---
2	sub x2..	add x1..	---	---	---
3	and x3..	sub x2..	add x1..	---	---
4	or x4..	and x3..	sub x2..	add x1..	---
5	beq x1..	or x4..	and x3..	sub x2..	add x1..
...					
Sonuc: 38 buyruk, 47 cevrim, BBÇ = 1,24					
İletme: 8, Durdurma: 3, Bosaltma: 2					

Ek zorluk (isteğe bağlı): Boru hattını Verilog ile de gerçekleştirin ve FPGA üzerinde sentezleyin.

D.4.3 Proje C: Önbellek ve Dallanma Öngörücüsü

Amaç: Bölüm 8’deki önbellek yapılarını ve Bölüm 7’deki dallanma öngörü tekniklerini benzetimle inceleyerek tasarım parametrelerinin başarıma etkisini ölçmek.

Bölüm 1: Önbellek Benzetimlikleri

1. Yapılandırılabilir parametrelili bir önbellek benzetimliği yazın:
 - Önbellek boyutu (örn. 1 KiB – 64 KiB)
 - Blok boyutu (örn. 16 – 128 bayt)
 - İlişkilendirme derecesi (doğrudan eşleme, 2/4/8 yollu, tam ilişkili)
 - Çıkarma politikası: LRU, FIFO, rastgele
2. Bellek erişim izi (trace) dosyasını okuyarak bulma/bulamama oranlarını hesaplayın.
3. Parametre taraması yaparak sonuçları tablolastırın. Örnek:

Tablo D.4: Önbellek parametre taraması sonuçları (örnek)

Boyut	İlişkilendirme	Blok	Bulamama (%)
4 KiB	Doğrudan	32 B	12,4
4 KiB	4 yollu	32 B	6,8
16 KiB	Doğrudan	32 B	5,1
16 KiB	4 yollu	32 B	2,3
16 KiB	4 yollu	64 B	1,9

Bölüm 2: Dallanma Öngörücüsü

1. Aşağıdaki dallanma öngörü tekniklerini gerçekleştirin:
 - Durağan öngörü: her zaman “dallanma yok”
 - 1 bitlik öngörücü
 - 2 bitlik doymalı sayaç öngörücüsü
 - İki düzeyli uyarlanır öngörücü (korelasyonlu)
2. Her öngörücü için dallanma izinden doğruluk oranlarını hesaplayın.
3. Sonuçları karşılaştırmalı olarak sunun.

Bölüm 3: Tümüleşik Analiz (isteğe bağlı)

Proje B’deki boru hattı benzetimliğine önbellek ve dallanma öngörücüsü entegre ederek bir programa ait toplam BBÇ’yi ölçün. Farklı yapılandırmaların başarıma etkisini karşılaştırmalı grafiklerle raporlayın.

Bilgi Notu D.6: İz Dosyası Kaynakları

Bellek erişim izleri için **Valgrind** (--tool=lackey) veya **Pin** gibi enstrümantasyon araçları kullanılabilir. Dallanma izleri için perf veya özel izleme kodları ile çıktı üretilebilir.

D.5 Proje Değerlendirme Ölçütleri

Projelerin değerlendirilmesinde aşağıdaki ölçütler önerilmektedir:

Tablo D.5: Önerilen proje değerlendirme ölçütleri

Ölçüt	Ağırlık (%)
Doğruluk (referans çıktıyla eşleşme)	40
Kod kalitesi ve okunabilirlik	15
Rapor ve çözümleme derinliği	20
Test kapsamı	15
Ek özellikler / yaratıcılık	10

Her proje için öğrenciden, tasarım kararlarını, karşılaşılan zorlukları ve sonuçların yorumunu içeren kısa bir teknik rapor beklenmektedir.

Alıştırmalar

Alıştırma D.1. ★ RARS benzetimliğinde Listeleme D.1'deki Fibonacci programını çalıştırın. Program bittiğinde bellekteki dizi alanının içeriğini yazın. t2 ve t3 yazmaçlarının son değerleri nedir?

Alıştırma D.2. ★ Aşağıdaki C programını riscv64-unknown-elf-gcc -S -O0 ile derleyin ve üretilen çevirici dil çıktısını inceleyin. -O2 ile derlediğinizde çıktı nasıl değişir? Farkları açıklayın.

```
int kare(int x) { return x * x; }
```

Alıştırma D.3. ★ riscv64-unknown-elf-objdump -d komutunun çıktısında her satırda hangi bilgiler yer alır? Adres, makine kodu ve çevirici dil alanlarını bir örnek üzerinde gösterin.

Alıştırma D.4. ★★ Listeleme D.3'deki VE kapısı modülünü genişleterek 4 girişli bir VE kapısı tasarlayın. Doğrulama ortamı yazarak tüm 16 giriş bileşimini denetleyin.

Alıştırma D.5. ★★ Listeleme D.4'daki 2'ye 1 çoklayıcıyı genişleterek 4'e 1 çoklayıcı tasarlayın. 2 bitlik sec girişi kullanın. Doğrulama ortamı ile denetleyin.

Alıştırma D.6. ★★ Listeleme D.7'deki yazmaç öbeği modülüne aşağıdaki değişiklikleri yapın:

- Üçüncü bir okuma portu (oku_adr3, oku_veri3) ekleyin.
- Yazma işleminin saat düşüş kenarında (negedge clk) gerçekleşmesini sağlayın. Bu değişikliğin boru hattı tasarımında ne avantajı olabilir?

Alıştırma D.7. ★★ 32 bitlik iki girişi toplayan ve elde (carry-out) bitini ayrı bir çıkış olarak veren bir toplayıcı modülünü Verilog ile tasarlayın. Doğrulama ortamında taşma durumlarını da denetleyin.

Alıştırma D.8. ★★ Proje A için bir RV32I yorumlayıcısı tasarladığınızı düşünün. addi, add, lw, sw, beq ve jal buyrukları için çözme (decode) mantığını sözde kod (pseudocode) olarak yazın. Her buyruğun bit alanlarını nasıl çıkaracağınızı gösterin.

Alıştırma D.9. ★★★ Listeleme D.7’deki yazmaç öbeği modülüne *yazma yönlendirmesi* (write forwarding) ekleyin: aynı çevrimde yazılan ve okunan yazmaç aynıysa, okuma portunda yeni yazılan değer görünsün. Bu özelliğin boru hattındaki veri tehlikelerini azaltmaya nasıl katkı sağladığını açıklayın.

Alıştırma D.10. ★★★ Proje C’deki önbellek benzetimliğini tasarladığınızı düşünün. Doğrudan eşlemeli, 4 KiB boyutunda, 16 baytlık bloklu bir önbellek için aşağıdaki bellek erişim dizisinin bulma/bulamama sonuçlarını elle hesaplayın (adresler onaltılık): 0x0000, 0x0040, 0x0100, 0x0000, 0x0040, 0x1000, 0x0000. Hangi erişimler çatışma bulamama (conflict miss) üretir?

Terimler Sözlüğü

Türkçe – İngilizce

Türkçe Terim	İngilizce Karşılık
adresleme kipleri	addressing modes
ağırlıklı ortalama	weighted average
Akış Çoklu İşlemci (SM)	Streaming Multiprocessor (SM)
altyordam	subroutine
Amdahl Yasası	Amdahl's Law
AMB	ALU (Arithmetic Logic Unit)
anlamalı kısım	significand / mantissa
en anlamalı bit (MSB)	Most Significant Bit (MSB)
en anlamsız bit (LSB)	Least Significant Bit (LSB)
anlık değer	immediate value
anlık değer üretici	immediate generator
ara bellek	buffer
ardışık tutarlılık	sequential consistency
arama zamanı	seek time
aritmetik buyruklar	arithmetic instructions
aşınma dengeleme	wear leveling
aritmetik mantık birimi (AMB)	Arithmetic Logic Unit (ALU)
aritmetik yoğunluk	arithmetic intensity
artık-N gösterimi	excess-N notation
ASCII	ASCII
ASIC	Application-Specific Integrated Circuit
çevirici dil (Assembly)	assembly language
atlama buyrukları	jump instructions
atomik işlem	atomic operation
aygıt	device
ayrıcalıklı kip	privileged mode
bilimsel gösterim	scientific notation
Booth algoritması	Booth's algorithm
B türü	B-type
bağlam değiştirme	context switching
bağlantı noktası	port
bağlayıcı	linker

(devamı sonraki sayfada)

Türkçe Terim (devam)	İngilizce Karşılık
bakış tablosu (LUT)	Look-Up Table (LUT)
bant genişliği	bandwidth
başarım	performance
bayat veri	stale data
bayt	byte
bayt sıralaması	byte order
büyük sayfa	huge page
BBÇ	CPI (cycles per instruction)
BCD	Binary-Coded Decimal
bellek	memory
bellek birleştirme	memory coalescing
bellek denetleyicisi	memory controller
bellek duvarı	memory wall
bellek koruması	memory protection
bellek hizalama	memory alignment
bellek hiyerarşisi	memory hierarchy
bellek tutarlılığı	cache coherence
bellek tutarlık modeli	memory consistency model
bellekle eşlemeli G/Ç	memory-mapped I/O
benchmark	sınama programı (denektaşı)
benzetim	simulation
benzetimlik	simulator
big.LITTLE	big.LITTLE
bire tümleyen	one's complement
birleşik bellek mimarisi	unified memory architecture
birleşik devre	combinational circuit
bit	bit
blok	block
boru hattı	pipeline
boru hattı aşaması	pipeline stage
boru hattı kaydı	pipeline register
boru hattı tehlikesi	pipeline hazard
boşluk (boru hattı)	bubble / NOP
bölme	division
bulamama (önbellekte)	miss (cache miss)
bulamama cezası	miss penalty
bulamama oranı	miss rate
bulma (önbellekte)	hit (cache hit)
bulma oranı	hit rate
buyruk	instruction
buyruk başına çevrim (BBÇ)	cycles per instruction (CPI)
buyruk belleği	instruction memory
buyruk çözme	instruction decode
buyruk düzeyi koşutluğu	instruction-level parallelism (ILP)
buyruk biçimi	instruction format
buyruk getirme birimi	instruction fetch unit

(devamı sonraki sayfada)

Türkçe Terim (devam)	İngilizce Karşılık
buyruk kümesi mimarisi (BKM)	Instruction Set Architecture (ISA)
buyruk önbelleği	instruction cache
buyruk sayacı	instruction counter
büyüğü başta	big-endian
can alıcı	critical
CISC	Complex Instruction Set Computer
CMOS	Complementary MOS
CUDA	Compute Unified Device Architecture
ÇBB	IPC (instructions per cycle)
çağrı kuralları	calling conventions
çarpma	multiplication
çarpma-toplama (MAC)	multiply-accumulate (MAC)
çatı çizgisi modeli	roofline model
çekirdek	core
çekirdek fonksiyonu (GPU)	kernel function
çerçeve göstergesi	frame pointer
çevirici	assembler
çevrim	cycle
çevrim başına buyruk (ÇBB)	instructions per cycle (IPC)
çevrim zamanı	cycle time
çıkartım (yapay zeka)	inference
çözümleme	analysis
çıkarma	subtraction
yonga içi bağlantı ağı (NoC)	Network on Chip (NoC)
çok çekirdekli işlemci	multi-core processor
çok seviyeli sayfa tablosu	multi-level page table
çok çevrimli	multi-cycle
çok vuruşluk işlemci	multi-cycle processor
çoklayıcı	multiplexer
çözgü	warp
çözgü sapması	warp divergence
dağıtık bellek	distributed memory
durum yazmacı	state register
dalga eldeli toplayıcı	ripple carry adder
dallanma buyrukları	branch instructions
dallanma hedef ara belleği	branch target buffer (BTB)
dallanma öngörüsü	branch prediction
darboğaz	bottleneck
DDR	Double Data Rate
DEĞİL kapısı	NOT gate
dekoder	decoder
denetim birimi	control unit
denetim sinyali	control signal
denetim tablosu	control table
denetim tehlikesi	control hazard
doğrulama ortamı	testbench

(devamı sonraki sayfada)

Türkçe Terim (devam)	İngilizce Karşılık
devingen buyruk sayısı	dynamic instruction count
devingen güç tüketimi	dynamic power consumption
devingen öngörü	dynamic prediction
Dennard ölçeklemesi	Dennard scaling
derin öğrenme	deep learning
derleyici	compiler
dikey mikroprogramlama	vertical microprogramming
dizin	index
dizin tabanlı protokol	directory-based protocol
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
doğrudan eşlemeli önbellek	direct-mapped cache
doğruluk tablosu	truth table
dönme gecikmesi	rotational latency
doğrudan yazma	write-through
doku belleği	texture memory
dolanıklık	entanglement
doluluk	occupancy
donanım	hardware
donanımsal çok kanallılık	hardware multithreading
dönüş adresi	return address
döngünün açılması	loop unrolling
döngüsel kilit	spin-lock
durağan buyruk sayısı	static instruction count
durağan öngörü	static prediction
durağan güç tüketimi	static power consumption (leakage)
durma	stall
düzenleme birimi	hazard detection unit
EDVAC	EDVAC
eğitim (yapay zeka)	training
elde	carry
elde ile seçmeli toplayıcı	carry select adder
elden önceden üretme	carry lookahead
ENIAC	ENIAC
engel	barrier
eşlik (RAID)	parity (RAID)
erişim süresi	access time
eşzamanlama	synchronization
eşzamanlı çok kanallılık (SMT)	Simultaneous Multithreading (SMT)
eşzamanlılık	concurrency
etiket	tag
etkin adres	effective address
Flaş çeviri katmanı (FTL)	Flash Translation Layer (FTL)
flaş bellek	flash memory
flip-flop	flip-flop
FPGA	Field-Programmable Gate Array

(devamı sonraki sayfada)

Türkçe Terim (devam)	İngilizce Karşılık
gecikme	latency
geometrik ortalama	geometric mean
gecikme odaklı tasarım	latency-oriented design
geciktirilmiş dallanma	delayed branching
gerçek veri bağımlılığı	true data dependence
genel amaçlı yazmaç	general-purpose register
genel bellek	global memory
geri yazma	write-back
getir-çöz-yürüt döngüsü	fetch-decode-execute cycle
gevşek tutarlılık	relaxed consistency
giriş/çıkış (G/Ç)	input/output (I/O)
gölgeleştirici	shader
gözetleme protokolü	snooping protocol
GPGPU	General-Purpose computing on GPU
GPU	Graphics Processing Unit
gshare öngörücü	gshare predictor
Gustafson Yasası	Gustafson's Law
güç duvarı	power wall
güç tüketimi	power consumption
Harvard mimarisi	Harvard architecture
HBM	High Bandwidth Memory
heterojen hesaplama	heterogeneous computing
ızgara	grid
I türü	I-type
IEEE 754	IEEE 754
yol	bus
ikincil bellek	secondary storage
IOPS	Input/Output Operations Per Second
ikiye tümleyen	two's complement
ikilik taban	binary
ileti geçirme	message passing
iş çıkarma oranı	throughput
iş parçacığı	thread
iş parçacığı düzeyi koşutluğu	thread-level parallelism
iş parçacığı havuzu	thread pool
işaret biti	sign bit
işaretli genişletme	sign extension
işaretli	signed
işaretsiz	unsigned
işlem kodu	opcode
işlemci	processor
işletim sistemi	operating system
kural dışı durum (boru hattı)	exception (pipeline)
J türü	J-type
kritik yol	critical path
karşı bağımlılık	anti-dependence

(devamı sonraki sayfada)

Türkçe Terim (devam)	İngilizce Karşılık
karşılaştırma buyrukları	comparison instructions
kaydırma buyrukları	shift instructions
kayan nokta	floating point
kayan noktalı sayı birimi	floating point unit (FPU)
katı hal sürücü (SSD)	Solid State Drive (SSD)
kesin kural dışı durum	precise exception
kesme	interrupt
kesme denetleyicisi	interrupt controller
kesme hizmet yordamı	interrupt service routine
kilit	lock/mutex
kilitlenme	deadlock
kod çözücü	decoder
koşullu dallanma	conditional branch
koşulsuz dallanma	unconditional branch
kuantum hesaplama	quantum computing
kübit	qubit
küçüğü başta	little-endian
kırmık	die
küme ilişkili	set associative
L1 önbellek	L1 cache
L2 önbellek	L2 cache
L3 önbellek	L3 cache
M.2	M.2 (form factor)
makine dili	machine language
makro buyruk	macro instruction
MLC	Multi-Level Cell
mantık buyrukları	logic instructions
mantık kapısı	logic gate
matris çarpımı	matrix multiplication
Meltdown	Meltdown
MESI protokolü	MESI protocol
merkezi işlem birimi (MİB)	Central Processing Unit (CPU)
MFLOPS	Millions of Floating Point Operations Per Second
mikro buyruk	micro-operation
mikro program belleği	control store
mikromimari	microarchitecture
mikroprogramlama	microprogramming
MIPS	Millions of Instructions Per Second
MİB	CPU (Central Processing Unit)
Moore Yasası	Moore's Law
NAND Flaş bellek	NAND Flash memory
NUMA	Non-Uniform Memory Access
NVMe	Non-Volatile Memory Express
olağanlaştırma	normalization
on altılık taban	hexadecimal
optimal çıkarma algoritması	optimal (Bélády's) replacement algorithm

(devamı sonraki sayfada)

Türkçe Terim (devam)	İngilizce Karşılık
ortalama bellek erişim süresi (AMAT)	Average Memory Access Time (AMAT)
önbellek	cache
önbellek satırı	cache line
önceden getirme	prefetching
örün	web
örün sunucusu	web server
örün tarayıcısı	web browser
ödünleşme	trade-off
Dışlayan VEYA kapısı	XOR gate
özel fonksiyon birimi (SFU)	Special Function Unit (SFU)
koşut hesaplama	parallel computing
koşut verimlilik	parallel efficiency
paylaşımlı bellek	shared memory
PCI Express	PCI Express
program sayacı	program counter
QLC	Quad-Level Cell
R türü	R-type
RAID	Redundant Array of Independent Disks
RISC	Reduced Instruction Set Computer
RISC-V	RISC-V
S türü	S-type
sıfırla genişletme	zero extension
saat çevrimi	clock cycle
saat sıklığı	clock frequency
sabit bellek	constant memory
sabit disk sürücüsü (HDD)	hard disk drive (HDD)
SATA	Serial ATA
saklama buyruğu	store instruction
sanal adres	virtual address
sanal bellek	virtual memory
sayfa	page
sayfa hatası	page fault
sayfa tablosu	page table
sekizlik taban	octal
semafor	semaphore
sıra dışı yürütme	out-of-order execution
şeritleme	striping
SIMD	Single Instruction, Multiple Data
SLC	Single-Level Cell
SIMT	Single Instruction, Multiple Threads
Sinir Ağı İşlem Birimi (NPU)	Neural Processing Unit (NPU)
sınama programı (denektaşı)	benchmark
sistolik dizi	systolic array
Soğutma Tasarım Gücü (TDP)	Thermal Design Power (TDP)
sonlu durum makinesi	finite state machine
sonuç bağımlılığı	output dependence

(devamı sonraki sayfada)

Türkçe Terim (devam)	İngilizce Karşılık
sonradan yazma	write-back
soyutlama	abstraction
sözde buyruk	pseudo-instruction
sözde-EUZK	pseudo-LRU
Spectre	Spectre
SPEC	SPEC
spekülatif yürütme	speculative execution
SRAM	Static Random Access Memory
Sv39 (RISC-V)	Sv39 (39-bit virtual addressing)
Sv48 (RISC-V)	Sv48 (48-bit virtual addressing)
Sv57 (RISC-V)	Sv57 (57-bit virtual addressing)
çoklu buyruk başlatımlı	superscalar
süperpozisyon	superposition
tam ilişkili	fully associative
Thunderbolt	Thunderbolt
TLC	Triple-Level Cell
TRIM	TRIM
tam sayı	integer
tam toplayıcı	full adder
taşma	overflow
tek çevrimli	single-cycle
tek vuruşluk işlemci	single-cycle processor
Tek Yongada Sistem (SoC)	System on Chip (SoC)
tekdüze bellek erişimi	Uniform Memory Access (UMA)
tekdüze olmayan bellek erişimi	Non-Uniform Memory Access (NUMA)
Tensör çekirdeği	Tensor Core
Tensör İşlem Birimi (TPU)	Tensor Processing Unit (TPU)
TLB	Translation Lookaside Buffer
toplama	addition
transistör	transistor
turnuva öngörücüsü	tournament predictor
tümleşik devre	integrated circuit
uç bilgi işlem	edge computing
U türü	U-type
USB	Universal Serial Bus
UCIe	Universal Chiptlet Interconnect Express
UMA	Uniform Memory Access
Unicode	Unicode
alanda yerellik	spatial locality
üç C modeli (3C modeli)	three Cs model
üs	exponent
üst anlık buyruklar	upper immediate instructions
vakum tüpü	vacuum tube
VE kapısı	AND gate
veri akış mimarisi	dataflow architecture
veri aktarma buyrukları	data transfer instructions

(devamı sonraki sayfada)

Türkçe Terim (devam)	İngilizce Karşılık
veri bağımlılığı	data dependency
veri belleği	data memory
veri tehlikesi	data hazard
veri yolu	datapath
veri yönlendirmesi	forwarding
işlem hacmi	throughput
işlem hacmi odaklı tasarım	throughput-oriented design
Verilog	Verilog
VEYA kapısı	OR gate
VHDL	VHDL
VLSI	Very Large-Scale Integration
Von Neumann darboğazı	Von Neumann bottleneck
Von Neumann mimarisi	Von Neumann architecture
yan kanal saldırısı	side-channel attack
yanlış paylaşım	false sharing
yapay sına programları	synthetic benchmarks
yapay zeka	artificial intelligence
yapısal tehlike	structural hazard
yarım toplayıcı	half adder
yarış durumu	race condition
yatay mikroprogramlama	horizontal microprogramming
yazılım	software
yazma geçersizleştirme	write-invalidate
yazma güncelleme	write-update
yazma serileştirme	write serialization
yazma tahsisli	write-allocate
yazma tahsisiz	no-write-allocate
yazma ara belleği	write buffer
yazma yayılımı	write propagation
yazmaç	register
yazmaç aktarım düzeyi (RTL)	Register Transfer Level (RTL)
yazmaç öbeği	register file
yazmaç yeniden adlandırma	register renaming
yekpare kırmık	monolithic die
yeniden yapılandırılabilir hesaplama	reconfigurable computing
yerel bellek	local memory
yerellik ilkesi	locality principle
yığın	heap
yığıt	stack
yığıt göstergesi	stack pointer
yonga	chip
yongacık	chiplet
yuvarlama	rounding
yükleyici	loader
yükle-kullan sorunu	load-use hazard
yükleme buyruğu	load instruction

(devamı sonraki sayfada)

Türkçe Terim (devam)	İngilizce Karşılık
yürütme birimi	execution unit
yürütme zamanı	execution time
zamanda yerellik	temporal locality

İngilizce – Türkçe

İngilizce Terim	Türkçe Karşılık
abstraction	soyutlama
access time	erişim süresi
addition	toplama
Average Memory Access Time (AMAT)	ortalama bellek erişim süresi (AMAT)
addressing modes	adresleme kipleri
Amdahl's Law	Amdahl Yasası
analysis	çözümleme
AND gate	VE kapısı
anti-dependence	karşı bağımlılık
Application-Specific Integrated Circuit (ASIC)	ASIC
Arithmetic Logic Unit (ALU)	aritmetik mantık birimi (AMB)
arithmetic instructions	aritmetik buyruklar
arithmetic intensity	aritmetik yoğunluk
artificial intelligence	yapay zeka
ASCII	ASCII
assembler	çevirici
assembly language	çevirici dil
atomic operation	atomik işlem
B-type	B türü
Booth's algorithm	Booth algoritması
bandwidth	bant genişliği
barrier	engel
benchmark	sınama programı (denektaşı)
big-endian	büyükü başta
big.LITTLE	big.LITTLE
binary	ikilik taban
Binary-Coded Decimal (BCD)	BCD
bit	bit
block	blok
bottleneck	darboğaz
branch instructions	dallanma buyrukları
branch prediction	dallanma öngörüsü
branch target buffer (BTB)	dallanma hedef ara belleği
buffer	ara bellek
bubble (pipeline)	boşluk (boru hattı)

(continued on next page)

İngilizce Terim (cont.)	Türkçe Karşılık
bus	yol
byte	bayt
byte order	bayt sıralaması
cache	önbellek
cache coherence	bellek tutarlılığı
cache line	önbellek satırı
calling conventions	çağrı kuralları
carry	elde
carry lookahead	elden önceden üretme
carry select adder	elde ile seçmeli toplayıcı
Central Processing Unit (CPU)	merkezi işlem birimi (MİB)
chip	yonga
chiplet	yongacık
clock cycle	saat çevrimi
clock frequency	saat sıklığı
combinational circuit	birleşik devre
Complementary MOS (CMOS)	CMOS
compiler	derleyici
comparison instructions	karşılaştırma buyrukları
Complex Instruction Set Computer (CISC)	CISC
Compute Unified Device Architecture (CUDA)	CUDA
concurrency	eşzamanlılık
conditional branch	koşullu dallanma
constant memory	sabit bellek
context switching	bağlam değiştirme
control hazard	denetim tehlikesi
control signal	denetim sinyali
control store	mikro program belleği
control table	denetim tablosu
control unit	denetim birimi
core	çekirdek
critical	can alıcı
critical path	kritik yol
cycle	çevrim
cycle time	çevrim zamanı
cycles per instruction (CPI)	buyruk başına çevrim (BBÇ)
die	kırmık
data dependency	veri bağımlılığı
data hazard	veri tehlikesi
data memory	veri belleği
data transfer instructions	veri aktarma buyrukları
dataflow architecture	veri akış mimarisi
datapath	veri yolu
deadlock	kilitlenme
decoder	dekoder / kod çözücü
deep learning	derin öğrenme

(continued on next page)

İngilizce Terim (cont.)	Türkçe Karşılık
delayed branching	geciktirilmiş dallanma
Dennard scaling	Dennard ölçeklemesi
device	aygıt
Direct Memory Access (DMA)	DMA
direct-mapped cache	doğrudan eşlemeli önbellek
directory-based protocol	dizin tabanlı protokol
index	dizin
distributed memory	dağıtık bellek
division	bölme
Double Data Rate (DDR)	DDR
dynamic instruction count	devingen buyruk sayısı
dynamic prediction	devingen öngörü
dynamic power consumption	devingen güç tüketimi
Dynamic Random Access Memory (DRAM)	DRAM
edge computing	uç bilgi işlem
EDVAC	EDVAC
effective address	etkin adres
ENIAC	ENIAC
entanglement	dolanıklık
exception (pipeline)	kural dışı durum (boru hattı)
excess-N notation	artık-N gösterimi
execution time	yürütme zamanı
execution unit	yürütme birimi
exponent	üs
false sharing	yanlış paylaşım
fetch-decode-execute cycle	getir-çöz-yürüt döngüsü
Field-Programmable Gate Array (FPGA)	FPGA
finite state machine	sonlu durum makinesi
flash memory	flaş bellek
Flash Translation Layer (FTL)	Flaş çeviri katmanı (FTL)
flip-flop	flip-flop
floating point	kayan nokta
floating point unit (FPU)	kayan noktalı sayı birimi
forwarding	veri yönlendirmesi
frame pointer	çerçeve göstergesi
full adder	tam toplayıcı
fully associative	tam ilişkili
General-Purpose computing on GPU (GPGPU)	GPGPU
general-purpose register	genel amaçlı yazmaç
geometric mean	geometrik ortalama
global memory	genel bellek
Graphics Processing Unit (GPU)	GPU
grid	ızgara
gshare predictor	gshare öngörücü
Gustafson's Law	Gustafson Yasası
half adder	yarım toplayıcı

(continued on next page)

İngilizce Terim (cont.)	Türkçe Karşılık
hard disk drive (HDD)	sabit disk sürücüsü
hardware	donanım
hardware multithreading	donanımsal çok kanallılık
Harvard architecture	Harvard mimarisi
hazard detection unit	düzenleme birimi
heap	yığın
heterogeneous computing	heterojen hesaplama
hexadecimal	on altılık taban
High Bandwidth Memory (HBM)	HBM
hit (cache hit)	bulma (önbellekte)
hit rate	bulma oranı
horizontal microprogramming	yatay mikroprogramlama
huge page	büyük sayfa
I-type	I türü
IEEE 754	IEEE 754
immediate generator	anlık değer üretici
immediate value	anlık değer
inference (AI)	çıkarım (yapay zeka)
Input/Output Operations Per Second (IOPS)	IOPS
input/output (I/O)	giriş/çıkış (G/Ç)
instruction	buyruk
instruction cache	buyruk önbelleği
instruction decode	buyruk çözme
instruction counter	buyruk sayacı
instruction fetch unit	buyruk getirme birimi
instruction format	buyruk biçimi
instruction memory	buyruk belleği
Instruction Set Architecture (ISA)	buyruk kümesi mimarisi
instruction-level parallelism (ILP)	buyruk düzeyi koşutluğu
instructions per cycle (IPC)	çevrim başına buyruk (ÇBB)
integer	tam sayı
integrated circuit	tümleşik devre
interrupt	kesme
interrupt controller	kesme denetleyicisi
interrupt service routine	kesme hizmet yordamı
J-type	J türü
jump instructions	atlama buyrukları
kernel function (GPU)	çekirdek fonksiyonu (GPU)
L1 cache	L1 önbellek
L2 cache	L2 önbellek
L3 cache	L3 önbellek
latency	gecikme
latency-oriented design	gecikme odaklı tasarım
Least Significant Bit (LSB)	en anlamsız bit (LSB)
linker	bağlayıcı
little-endian	küçüğü başta

(continued on next page)

İngilizce Terim (cont.)	Türkçe Karşılık
load instruction	yükleme buyruğu
load-use hazard	yükle-kullan sorunu
loader	yükleyici
local memory	yerel bellek
locality principle	yerellik ilkesi
Look-Up Table (LUT)	bakış tablosu (LUT)
lock/mutex	kilit
logic gate	mantık kapısı
logic instructions	mantık buyrukları
loop unrolling	döngünün açılması
M.2	M.2 (form faktörü)
machine language	makine dili
macro instruction	makro buyruk
matrix multiplication	matris çarpımı
Meltdown	Meltdown
memory	bellek
memory alignment	bellek hizalama
memory coalescing	bellek birleştirme
memory consistency model	bellek tutarlılık modeli
memory controller	bellek denetleyicisi
memory hierarchy	bellek hiyerarşisi
memory protection	bellek koruması
memory wall	bellek duvarı
memory-mapped I/O	bellekle eşlemeli G/Ç
Most Significant Bit (MSB)	en anlamlı bit (MSB)
MESI protocol	MESI protokolü
message passing	ileti geçirme
microarchitecture	mikromimari
micro-operation	mikro buyruk
microprogramming	mikroprogramlama
Millions of Floating Point Operations Per Second (MFLOPS)	MFLOPS
Millions of Instructions Per Second (MIPS)	MIPS
miss (cache miss)	bulamama (önbellekte)
miss penalty	bulamama cezası
miss rate	bulamama oranı
monolithic die	yekpare kırmık
Moore's Law	Moore Yasası
multi-core processor	çok çekirdekli işlemci
multiply-accumulate (MAC)	çarpma-toplama (MAC)
multi-cycle	çok çevrimli
multi-cycle processor	çok vuruşluk işlemci
Multi-Level Cell (MLC)	MLC
multi-level page table	çok seviyeli sayfa tablosu
multiplication	çarpma
multiplexer	çoklayıcı

(continued on next page)

İngilizce Terim (cont.)	Türkçe Karşılık
NAND Flash memory	NAND Flaş bellek
Network on Chip (NoC)	yonga içi bağlantı ağı (NoC)
Neural Processing Unit (NPU)	Sinir Ağı İşlem Birimi (NPU)
Non-Uniform Memory Access (NUMA)	tekdüze olmayan bellek erişimi
Non-Volatile Memory Express (NVMe)	NVMe
no-write-allocate	yazma tahsisiz
normalization	olağanlaştırma
NOT gate	DEĞİL kapısı
occupancy	doluluk
octal	sekizlik taban
one's complement	bire tümleyen
opcode	işlem kodu
operating system	işletim sistemi
optimal (Bélády's) replacement algorithm	optimal çıkarma algoritması
OR gate	VEYA kapısı
out-of-order execution	sıra dışı yürütme
output dependence	sonuç bağımlılığı
overflow	taşma
page	sayfa
parity (RAID)	eşlik (RAID)
page fault	sayfa hatası
page table	sayfa tablosu
parallel computing	koşut hesaplama
parallel efficiency	koşut verimlilik
PCI Express	PCI Express
performance	başarım
pipeline	boru hattı
pipeline hazard	boru hattı tehlikesi
pipeline register	boru hattı kaydı
pipeline stage	boru hattı aşaması
port	bağlantı noktası
power consumption	güç tüketimi
power wall	güç duvarı
precise exception	kesin kural dışı durum
prefetching	önceden getirme
privileged mode	ayrıcalıklı kip
processor	işlemci
program counter	program sayacı
pseudo-instruction	sözde buyruk
pseudo-LRU	sözde-EUZK
Quad-Level Cell (QLC)	QLC
quantum computing	kuantum hesaplama
qubit	kübit
R-type	R türü
race condition	yarış durumu
reconfigurable computing	yeniden yapılandırılabilir hesaplama

(continued on next page)

İngilizce Terim (cont.)	Türkçe Karşılık
Redundant Array of Independent Disks (RAID)	RAID
Reduced Instruction Set Computer (RISC)	RISC
register	yazmaç
register file	yazmaç öbeği
register renaming	yazmaç yeniden adlandırma
Register Transfer Level (RTL)	yazmaç aktarım düzeyi (RTL)
relaxed consistency	gevşek tutarlılık
return address	dönüş adresi
ripple carry adder	dalga eldeli toplayıcı
RISC-V	RISC-V
roofline model	çatı çizgisi modeli
rotational latency	dönme gecikmesi
rounding	yuvarlama
S-type	S türü
SATA (Serial ATA)	SATA
secondary storage	ikincil bellek
seek time	arama zamanı
semaphore	semafor
sequential consistency	ardışık tutarlılık
Serial ATA (SATA)	SATA
set associative	küme ilişkili
shader	gölgelendirici
shared memory	paylaşımlı bellek
shift instructions	kaydırma buyrukları
side-channel attack	yan kanal saldırısı
sign bit	işaret biti
sign extension	işaretli genişletme
signed	işaretli
single-cycle processor	tek vuruşluk işlemci
Simultaneous Multithreading (SMT)	eşzamanlı çok kanallılık (SMT)
Single Instruction, Multiple Data (SIMD)	SIMD
Single Instruction, Multiple Threads (SIMT)	SIMT
Single-Level Cell (SLC)	SLC
single-cycle	tek çevrimli
simulation	benzetim
simulator	benzetimlik
snooping protocol	gözetleme protokolü
software	yazılım
scientific notation	bilimsel gösterim
significand / mantissa	anamlı kısım
Solid State Drive (SSD)	katı hal sürücü (SSD)
spatial locality	alandaki yerellik
SPEC	SPEC
Special Function Unit (SFU)	özel fonksiyon birimi (SFU)
Spectre	Spectre
speculative execution	spekülatif yürütme

(continued on next page)

İngilizce Terim (cont.)	Türkçe Karşılık
spin-lock	döngüsel kilit
stack	yığıt
stack pointer	yığıt göstergesi
stale data	bayat veri
stall	durma
state register	durum yazmacı
static instruction count	durağan buyruk sayısı
static power consumption (leakage)	durağan güç tüketimi
static prediction	durağan öngörü
Static Random Access Memory (SRAM)	SRAM
store instruction	saklama buyruğu
Streaming Multiprocessor (SM)	Akış Çoklu İşlemci (SM)
Sv39 (39-bit virtual addressing)	Sv39 (RISC-V)
Sv48 (48-bit virtual addressing)	Sv48 (RISC-V)
Sv57 (57-bit virtual addressing)	Sv57 (RISC-V)
striping	şeritleme
structural hazard	yapısal tehlike
subroutine	altyordam
subtraction	çıkarma
superposition	süperpozisyon
superscalar	çoklu buyruk başlatımlı
synchronization	eşzamanlama
synthetic benchmarks	yapay sına programları
System on Chip (SoC)	Tek Yongada Sistem (SoC)
systolic array	sistolik dizi
tag	etiket
temporal locality	zamanda yerellik
three Cs model (3C)	üç C modeli (3C modeli)
Tensor Core	Tensör çekirdeği
Tensor Processing Unit (TPU)	Tensör İşlem Birimi (TPU)
texture memory	doku belleği
Thermal Design Power (TDP)	Soğutma Tasarım Gücü (TDP)
testbench	doğrulama ortamı
thread	iş parçacığı
thread pool	iş parçacığı havuzu
thread-level parallelism	iş parçacığı düzeyi koşutluğu
throughput	işlem hacmi
Thunderbolt	Thunderbolt
trade-off	ödünleşme
training (AI)	eğitim (yapay zeka)
transistor	transistör
TRIM	TRIM
tournament predictor	turnuva öngörücüsü
Triple-Level Cell (TLC)	TLC
Translation Lookaside Buffer (TLB)	TLB
true data dependence	gerçek veri bağımlılığı

(continued on next page)

İngilizce Terim (cont.)	Türkçe Karşılık
truth table	doğruluk tablosu
two's complement	ikiye tümleyen
U-type	U türü
unconditional branch	koşulsuz dallanma
Unicode	Unicode
Uniform Memory Access (UMA)	tekdüze bellek erişimi
unified memory architecture	birleşik bellek mimarisi
Universal Chiplet Interconnect Express (UCIe)	UCIe
Universal Serial Bus (USB)	USB
unsigned	işaretsiz
upper immediate instructions	üst anlık buyruklar
vacuum tube	vakum tüpü
Verilog	Verilog
vertical microprogramming	dikey mikroprogramlama
Very Large-Scale Integration (VLSI)	VLSI
VHDL	VHDL
virtual address	sanal adres
virtual memory	sanal bellek
Von Neumann architecture	Von Neumann mimarisi
Von Neumann bottleneck	Von Neumann darboğazı
warp	çözümlü
web	örün
web browser	örün tarayıcısı
web server	örün sunucusu
weighted average	ağırlıklı ortalama
wear leveling	aşınma dengeleme
warp divergence	çözümlü sapması
write-allocate	yazma tahsisli
write-back	geri yazma
write buffer	yazma ara belleği
write-invalidate	yazma geçersizleştirme
write propagation	yazma yayılımı
write serialization	yazma serileştirmesi
write-through	doğrudan yazma
write-update	yazma güncelleme
XOR gate	Dışlayan VEYA kapısı
zero extension	sıfırla genişletme

Kaynakça

- [2] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. İinde: *Proceedings of the AFIPS Spring Joint Computer Conference*. ACM, 1967, ss. 483–485. DOI: [10.1145/1465482.1465560](https://doi.org/10.1145/1465482.1465560).
- [3] Christian Bienia ve diğ. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. İinde: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2008, ss. 72–81. DOI: [10.1145/1454115.1454128](https://doi.org/10.1145/1454115.1454128).
- [5] George Boole. *An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities*. London: Walton ve Maberly, 1854.
- [6] James Bucek, Klaus-Dieter Lange ve Jóakim v. Kistowski. “SPEC CPU2017: Next-Generation Compute Benchmark”. İinde: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2018, ss. 41–42. DOI: [10.1145/3185768.3185771](https://doi.org/10.1145/3185768.3185771).
- [8] Danny Cohen. “On Holy Wars and a Plea for Peace”. İinde: *Computer* 14.10 (1981). İlk dağıtım: IEN 137, 1 Nisan 1980, ss. 48–54.
- [9] Robert H. Dennard ve diğ. “Design of Ion-Implanted MOSFETs with Very Small Physical Dimensions”. İinde: *IEEE Journal of Solid-State Circuits* 9.5 (1974), ss. 256–268. DOI: [10.1109/JSSC.1974.1050511](https://doi.org/10.1109/JSSC.1974.1050511).
- [10] Jack J. Dongarra, Piotr Luszczek ve Antoine Petit. “The LINPACK Benchmark: Past, Present and Future”. İinde: *Concurrency and Computation: Practice and Experience* 15.9 (2003), ss. 803–820. DOI: [10.1002/cpe.728](https://doi.org/10.1002/cpe.728).
- [11] Oğuz Ergin ve diğ. “Register Packing: Exploiting Narrow-Width Operands for Reducing Register File Pressure”. İinde: *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-37)*. Portland, OR, USA: IEEE Computer Society, 2004, ss. 304–315.
- [12] Shay Gal-On ve Markus Levy. *Exploring CoreMark – A Benchmark Maximizing Simplicity and Efficacy*. Tek. rap. EEMBC (Embedded Microprocessor Benchmark Consortium), 2012. URL: <https://www.eembc.org/techlit/articles/coremark-whitepaper.pdf>.
- [16] John L. Henning. “SPEC CPU2006 Benchmark Descriptions”. İinde: *ACM SIGARCH Computer Architecture News* 34.4 (2006), ss. 1–17. DOI: [10.1145/1186736.1186737](https://doi.org/10.1145/1186736.1186737).
- [18] Daniel A. Jiménez ve Calvin Lin. “Dynamic Branch Prediction with Perceptrons”. İinde: *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*. 2001, ss. 197–206. DOI: [10.1109/HPCA.2001.903263](https://doi.org/10.1109/HPCA.2001.903263).

- [20] Maurice Karnaugh. “The Map Method for Synthesis of Combinational Logic Circuits”. İçinde: *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics* 72.5 (1953), ss. 593–599.
- [24] Peter Mattson ve diğ. “MLPerf Training Benchmark”. İçinde: *Proceedings of Machine Learning and Systems (MLSys)*. C. 2. 2020, ss. 336–349.
- [25] Edward J. McCluskey. “Minimization of Boolean Functions”. İçinde: *Bell System Technical Journal* 35.6 (1956), ss. 1417–1444.
- [26] Scott McFarling. *Combining Branch Predictors*. Tek. rap. TN-36. Digital Equipment Corporation, Western Research Laboratory, 1993.
- [27] George H. Mealy. “A Method for Synthesizing Sequential Circuits”. İçinde: *Bell System Technical Journal* 34.5 (1955), ss. 1045–1079. DOI: [10.1002/j.1538-7305.1955.tb03788.x](https://doi.org/10.1002/j.1538-7305.1955.tb03788.x).
- [28] Edward F. Moore. “Gedanken-Experiments on Sequential Machines”. İçinde: *Automata Studies*. Ed. C. E. Shannon ve J. McCarthy. Princeton University Press, 1956, ss. 129–153.
- [32] David A. Patterson ve David R. Ditzel. “The Case for the Reduced Instruction Set Computer”. İçinde: *ACM SIGARCH Computer Architecture News*. C. 8. 6. ACM, 1980, ss. 25–33.
- [35] Primate Labs. *Geekbench 6 – Cross-Platform Benchmark*. <https://www.geekbench.com/>. 2023.
- [36] Willard Van Orman Quine. “The Problem of Simplifying Truth Functions”. İçinde: *The American Mathematical Monthly* 59.8 (1952), ss. 521–531.
- [37] André Seznec ve Pierre Michaud. “A Case for (Partially) TAgged GEometric History Length Branch Prediction”. İçinde: *Journal of Instruction-Level Parallelism*. C. 8. 2006.
- [38] Claude E. Shannon. “A Symbolic Analysis of Relay and Switching Circuits”. Yüksek lisans tezi; 1938’de *Transactions of the AIEE*’de yayımlanmıştır. Yük. lis. tezi. Massachusetts Institute of Technology, 1937.
- [40] Transaction Processing Performance Council. *TPC Benchmarks*. <https://www.tpc.org/>. 2024.
- [41] Alan M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. İçinde: *Proceedings of the London Mathematical Society* s2-42.1 (1936), ss. 230–265. DOI: [10.1112/plms/s2-42.1.230](https://doi.org/10.1112/plms/s2-42.1.230).
- [43] John Von Neumann. *First Draft of a Report on the EDVAC*. Tek. rap. Contract No. W-670-ORD-4926. University of Pennsylvania, Moore School of Electrical Engineering, 1945.
- [45] Steven Cameron Woo ve diğ. “The SPLASH-2 Programs: Characterization and Methodological Considerations”. İçinde: *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*. ACM, 1995, ss. 24–36. DOI: [10.1145/223982.223990](https://doi.org/10.1145/223982.223990).

Türkçe Kaynaklar

- [1] Eşref Adalı. *Mikroişlemciler Mikrobilgisayarlar*. 5. bs. Birsen Yayınevi, 2004.
- [4] Mehmet Bodur. *Bilgisayar Organizasyonu: RISC Donanımına Giriş*. Ankara: TMMOB Elektrik Mühendisleri Odası, 2003.
- [19] Şirzat Kahramanlı. *Bilgisayar Mimarisi*. Nobel Akademik Yayıncılık, 2006.
- [23] M. Morris Mano. *Bilgisayar Sistemleri Mimarisi*. Çevirmenler: Abdüssamet Marşoğlu ve Nurşen Suçsuz; 3. baskıdan çeviri. Literatür Yayıncılık, 2002.
- [34] David A. Patterson ve John L. Hennessy. *Bilgisayar Mimarisi ve Tasarım*. 5. bs. Çeviri editörü: Hasan Dağ. Nobel Akademik Yayıncılık, 2021.
- [42] Cengiz Uğurkaya ve Osman Aliefendioğlu. *Modern Bilgisayar Mimarisi*. Ed. Toros Rifat Çölkesen. Papatya Bilim Yayınevi, 2026.

Önerilen Kaynaklar

- [7] Nick Carter. *Schaum's Outline of Computer Architecture*. McGraw-Hill, 2001.
- [13] John L. Gustafson. "Reevaluating Amdahl's Law". İçinde: *Communications of the ACM* 31.5 (1988), ss. 532–533.
- [14] Sarah L. Harris ve David Harris. *Digital Design and Computer Architecture: RISC-V Edition*. Morgan Kaufmann, 2021.
- [15] John L. Hennessy ve David A. Patterson. *Computer Architecture: A Quantitative Approach*. 6. bs. Morgan Kaufmann, 2019.
- [17] Mark D. Hill ve Michael R. Marty. "Amdahl's Law in the Multicore Era". İçinde: *Computer* 41.7 (2008), ss. 33–38.
- [21] Jack S. Kilby. "Miniaturized Electronic Circuits". İçinde: *US Patent* 3,138,743 (1964). İlk tımlleşik devre patenti.
- [22] M. Morris Mano. *Computer System Architecture*. 3. bs. Prentice Hall, 1992.
- [29] Gordon E. Moore. "Cramming More Components onto Integrated Circuits". İçinde: *Electronics* 38.8 (1965), ss. 114–117.
- [30] Linda Null ve Julia Lobur. *The Essentials of Computer Organization and Architecture*. 5. bs. Jones & Bartlett Learning, 2019.
- [31] Behrooz Parhami. *Computer Architecture: From Microprocessors to Supercomputers*. Oxford University Press, 2005.
- [33] David A. Patterson ve John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 6. bs. RISC-V Edition. Morgan Kaufmann, 2020.
- [39] William Stallings. *Computer Organization and Architecture: Designing for Performance*. 11. bs. Pearson, 2022.
- [44] Andrew Waterman ve Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Document Version 20191213. 2019.

Yazar Hakkında



Prof. Dr. Oğuz Ergin, bilgisayar mimarisi ve yüksek başarımli işlemciler üzerine çalışan bir akademisyen, araştırmacı ve girişimcidir. Lisansını ODTÜ’de, yüksek lisans ve doktorasını New York’ta Binghamton Üniversitesi’nde tamamladı. Kariyerinin başında Intel’in Barselona araştırma merkezinde kıdemli araştırmacı olarak çalıştı; Türkiye’ye döndükten sonra TOBB ETÜ’de öğretim üyesi oldu, uzun yıllar bölüm başkanlığı ve üniversite yönetiminde danışmanlık görevleri yürüttü. 2024’ten bu yana Birleşik Arap Emirlikleri’nde Sharjah Üniversitesi Bilgisayar Mühendisliği Bölümü’nde profesör olarak görev yapmaktadır.

Araştırmaları; mikroişlemci mimarileri, bellek (özellikle DRAM) güvenliği ve güvenilirliği, enerji verimliliği, donanım hızlandırıcılar ve hata toleransı alanlarına odaklanır. Çalışmaları HPCA, MICRO ve ISCA gibi dünyadaki en saygın konferanslarda yayımlandı; 120’nin üzerinde uluslararası yayını ve çeşitli patentleri bulunmaktadır.

Akademi-sanayi köprüsünde etkin rol alan Ergin, Kasırğa ve Yumruk şirketlerini kurarak savunma ve sivil alanlarda işlemci ve gömülü sistem çözümleri geliştirdi; Aselsan’a ve TUSAŞ’a danışmanlık yaptı ve Bilişim Vadisi’nde teknoloji danışma kurulunda görev aldı. ETH Zürih, Edinburgh ve Notre Dame üniversitelerinde misafir öğretim üyesi olarak bulundu.

Bugün, yetiştirdiği öğrenciler dünyanın önde gelen araştırma gruplarında ve teknoloji şirketlerinde görev alıyor; danışmanlığını yaptığı öğrenci ekipleri Teknofest yonga tasarımı yarışmalarında birçok derece elde etti. Oğuz Ergin, bilim ve teknoloji temelli yönetim anlayışını; liyakat, şeffaflık ve yüksek katma değerli üretim odaklı bir perspektifle toplumsal faydaya dönüştürmeyi amaçlıyor.