DR. MEHMET BODUR

e-kitap

BİLGISAYAR ORGANIZASYONU: RISC DONANIMINA GİRİŞ

BAŞARIM ÖLÇÜTLERİ

KOMUT TAKIMI

BİLGİSAYARLAR İÇİN ARİTMETİK

İŞLEMCİ, VERİYOLU VE DENETİM

EMO YAYIN NO: EK/2011/2

TMMOB
Elektrik Mühendisleri Odası

Solda kalacak boş sayfa

Bilgisayar Organizasyonu: RISC Donanımına Giriş

Dr. Mehmet BODUR

Bilgisayar Organizasyonu: RISC Donanımına Giriş

Ankara-Ocak 2011
EMO YAYIN NO: EK/2011/2
ISBN : 978-605-01-0055-6

TMMOB ELEKTRİK MÜHENDİSLERİ ODASI
Ihlamur Sokak No: 10 • 06420, Kızılay-Ankara
Tel: (0.312) 425 32 72-73 • Faks: (0.312) 417 38 18

004.22 ELE 2011
Elektrik Mühendisleri Odası
Bilgisayar Organizasyonu: RISC Donanımına Giriş
EMO --1.bs.--Ankara: EMO Yayınları, 2011, 244 s.;
(EK/2011/12)
978-605-01-0055-6
Bilgisayar Mimarisi

Dizgi ve Tasarım
Elektrik Mühendisleri Odası

© Bu eserin yayın hakkı ELEKTRİK MÜHENDİSLERİ ODASI'na aittir. Kitaptaki bilgiler kaynak gösterilerek kullanılabilir

İçindekiler

İÇİNDEKİLER iii

ŞEKİL BAŞLIKLARI vii

TABLO BAŞLIKLARI xi

ÖNSÖZ xiii

1 GİRİŞ 1

1.1 BILGİSAYAR TASARIMI SEVİYESİ HAKKINDA 1
1.2 MIPS R2000 IŞLEMCİSİ HAKKINDA 3
1.3 RISC -IN KISA TARİHİ 3
1.4 SONUÇ 4

2 BAŞARIM ÖLÇÜTLERİ 5

2.1 GIRİŞ 5
2.2 BAŞARIM TANIMI 6
2.2.1 Ölçme Koşulları ve Ölçme Birimleri 7
2.3 YAYGIN KULLANILAN YANILTICI BAŞARIM ÖLÇÜTLERİ 9
2.3.1 *MIPS* Başarım Ölçümü 9
2.3.2 *MIPS* ölçümü kullanmanın sakıncaları 9
2.3.3 Tepe *MIPS* 11
2.3.4 Göreceli *MIPS* 11
2.3.5 *MFLOPS* ile Başarım Ölçümü 11
2.3.6 Normalize *MFLOPS* 12
2.3.7 Tepe *MFLOPS* 12
2.4 BAŞARIM DEĞERLENDİRME-PROGRAMLARININ SEÇİMİ 12
2.5 TOPLAM ÇALIŞMA ZAMANININ HESAPLANMASI 14
2.6 SONUÇ 15
2.7 ÇÖZÜMLÜ PROBLEMLER 16

3 KOMUT TAKIMI 19

3.1 GIRİŞ 19
3.2 TEMEL MIPS ÇEKİRDEĞİ 20
3.2.1 MIPS -in Veri Çeşitleri 20
3.2.2 Aritmetik-Mantık komutları 21
3.2.3 Yazmaç Dosyası 22
3.2.4 Bellek Değişkenleri ve Diziler 24
3.2.5 Anlık tamsayı değerler 27
3.3 KOMUTLARIN BILGİSAYARDA GÖSTERİMİ 29
3.3.1 Programların bellekte duruması 32
3.4 KARAR VERME KOMUTLARI 32
3.4.1 Koşullu Dallanma 33

3.4.2 Atla ... 33
3.4.3 Değişkenlerin eşitsizlik testi ... 35
3.4.4 Derleyici , Çevirici, Bağlayıcı ve Yükleyici ... 36
3.5 ALTYORDAM DONANIMI ... 37
3.5.1 jal ve jr komutları ... 37
3.5.2 İçiçe yordam çağrıları ... 38
3.5.3 Yordam Argümanları (Değiştirgeleri) ... 41
3.6 DALLAN VE ATLA KOMUTLARINDA ADRESLEME ... 44
3.6.1 Anlık atla adreslemesi ... 44
3.6.2 PC-göreceli-adresleme ... 45
3.7 MIPS -İN ADRESLEME KİPLERİ ... 47
3.7.1 Yazmaçlı Adresleme ... 47
3.7.2 Baz yada yerdeğişimli adresleme ... 47
3.7.3 Anlık adresleme: ... 48
3.7.4 PC-göreceli adresleme ... 48
3.8 MIPS TEMEL KOMUT ALT TAKIMI ... 49
3.8.1 MIPS-II İşlem Kodu Çizelgesi ... 50
3.9 ÇÖZÜMLÜ PROBLEMLER ... 53

4 BİLGİSAYARLAR İÇİN ARİTMETİK ... 61

4.1 SAYI SİSTEMİ, TOPLAMA VE ÇIKARMA ... 61
4.1.1 İki sayı sistemi arasında dönüştürme ... 62
4.1.2 Eksi sayıların gösterimi ... 64
4.1.3 Toplama ve Çıkarma ... 67
4.1.4 İşaretli-ikilik toplamada sonuç taşması ... 68
4.1.5 Taşmayı algılama ... 69
4.2 MANTIKSAL İŞLEMLER ... 70
4.2.1 Mantıksal ve Aritmetik Kaydırma İşlemleri ... 70
4.2.2 Bit-bite VE ve VEYA ... 71
4.3 ARİTMETİK MANTIK BİRİMİ (ALU) ... 71
4.3.1 Tek-bitlik ALU ... 73
4.3.2 32-bitlik ALU ... 74
4.3.3 Çıkarma işlemi ... 75
4.3.4 Küçükse-Bir-Yap (Set on Less Than) İşlemi ... 77
4.3.5 SLT ve Koşullu Dallanmayı destekleyen 32-bit ALU ... 79
4.3.6 Önden- Eldeli Toplama: ... 81
4.3.7 Çözümlü Problemler ... 85
4.4 ÇARPMA ... 87
4.4.1 İlk Çarpma Algoritması ... 88
4.4.2 İkinci Çarpma Algoritması ... 90
4.4.3 Üçüncü Çarpma Algoritması: ... 92
4.4.4 İşaretli Çarpma ... 93
4.4.5 İşaretli Çarpma Algoritmaları ... 94
4.4.6 Birleşimsel Çarpıcı Ağı ... 98
4.5 BÖLME ... 100
4.5.1 İlk-Bölme Algoritması ... 101
4.5.2 İkinci-bölme algoritması ... 103
4.5.3 Üçüncü bölme algoritması ... 104
4.5.4 İşaretli Bölme ... 107
4.6 GERÇEK SAYILAR VE KAYAN-NOKTALI ARİTMETİK ... 108

4.6.1 IEEE-754 Kayan-Noktalı Sayı Standardı 112
4.6.2 Kayan Noktalı Toplama 115
4.6.3 Kayan-noktalı toplama için veri işlem donanımı 116
4.6.4 Kayan -Noktalı Çarpma ve Bölme 117
4.6.5 Kayan noktalı çarpma için algoritma............................... 120
4.6.6 MIPS FPU Komutları 122
4.6.7 Çözümlü Problemler... 124
4.7 ANUYUMLU VERİ-İŞLEME VE DURUM MAKİNELERİ 127
4.7.1 Durum makinelerinde Veri İşlem ve Denetleç Birimi........ 127
4.7.2 ASM Çizim Elemanları 129
4.7.3 Algoritmadan Durum Makinesine.................................... 131
4.7.4 ASM-çiziminden devre şemasına 138

5 İŞLEMCİ, VERİYOLU VE DENETİM..145

5.1 GİRİŞ 145
5.1.1 Terim, Bileşen ve Uzlaşımlar.. 147
5.1.2 MIPS Komut Altkümesi Gerçeklemesi 151
5.2 TEK SAATLİ DATA YOLLARINI OLUŞTURMA 151
5.2.1 R-formatı komutları 152
5.2.2 Anlık Bellek-Yazmaç Aktarımı Komutları 154
5.2.3 Dallanma (*branch*) Komutu... 157
5.2.4 Ana Denetim Biriminin Gerçeklenmesi 162
5.2.5 Yeni bir komut-tipi için veri yolu değişikliği 167
5.2.6 Komutların Yürütme Ayrıntıları 169
5.3 BİR PROGRAM KODUNUN ÇALIŞMASI 173
5.3.1 Tek-Dönüşlü İşlemenin Sakıncaları................................ 179
5.4 ÇOK-DÖNÜŞLÜ GERÇEKLEME 182
5.4.1 Çok-saat-dönüşlü Veriyolu 184
5.4.2 Çok-saat-dönüşü Denetim Sinyalleri 185
5.4.3 Komut Çalışmasını Saat Döngülerine Bölmek................. 188
5.4.4 PC çoklayıcısı ve PC-yaz-denetimi 192
5.5 ÇOK DÖNÜŞLÜ VERİYOLU VE DENETİM SİNYALLERİ........................ 193
5.5.1 Çok-dönüşlü gerçeklemenin başarımı 195
5.6 ÇOK-DÖNÜŞLÜ VERİYOLU İÇİN DENETİM BİRİMİ............................ 196
5.6.1 Sonlu-Durum Makinesi (FSM) Yaklaşımı......................... 196
5.6.2 Sıralayıcı 206
5.6.3 Mikroprogramlı Denetim Yaklaşımı 208
5.6.4 Komut Kümesinin Genişletilmesi 216
5.7 SONUÇ.. 219

KAYNAKÇA...221

DİZİN..223

Şekil Başlıkları

Şekil 2-1 IBM PowerStation 550 da kullanılan iki farklı derleyici ile ölçülmüş başarım değerleri....14
Şekil 2-2 Son karar için tipik başarım-fiyat grafiği....16
Şekil 3-1 Tipik bilgisayarın öbek şeması....19
Şekil 3-2 Bir aritmetik-mantık komutun yürütülüşü....23
Şekil 3-3 Yazmaç-dosyasının dahili veriyolu....24
Şekil 4-1 VE ve VEYA işlemleri yapan tek-bitlik ALU....73
Şekil 4-2 topla, ve, veya işlemleri yapan bir-bitlik-ALU....73
Şekil 4-3 toplama yapan 3-bitlik-ALU, A=011 ve B=110, kalın çizgiler 1 durumunu gösteriyor....74
Şekil 4-4 32-bit ALU....75
Şekil 4-5 Çıkarma yapabilen bir-bit-ALU....76
Şekil 4-6 Çıkarma yapmak için 32-bit ALU....76
Şekil 4-7 32-bit ALU....77
Şekil 4-8 herhangi bir tek-bitlik-ALU birimi için SLT değişikliği....78
Şekil 4-9 En sol ALU-biti birimi için SLT değişikliği ve taşma algılaması....78
Şekil 4-10 VE VEYA TOPLA ÇIKAR ve SLT işleyebilen 32-bit ALU....80
Şekil 4-11 ALU sembolü....81
Şekil 4-12 Tipik 4-bit peşin-elde-bulmalı toplama bloğu....83
Şekil 4-13 4-bit önden-eldeli toplayıcı öbeklerden oluşmuş 16-bitlik toplayıcı....84
Şekil 4-14 İlk çarpma algoritması ve ASM çizimi....88
Şekil 4-15 İlk çarpıcı için donanım ve veri yolları....89
Şekil 4-16 İkinci çarpıcı devresinin Veri Yolları....90
Şekil 4-17 ikinci-çarpıcı devresinin algoritması ve ASM çizimi....91
Şekil 4-18 Üçüncü Çarpıcı Devresi Veri Yolları....92
Şekil 4-19 Üçüncü çarpıcı algoritması ve ASM çizimi....92
Şekil 4-20 İşaretli Üçüncü-Çarpma Algoritması....95
Şekil 4-21 İşaretli Üçüncü-Çarpıcı devresinin ASM çizimi....95
Şekil 4-22 İşaretli Üçüncü-Çarpıcı VeriYolu....95
Şekil 4-23 Booth'un İşaretli Çarpma Veriyolu....96
Şekil 4-24 Booth'un işaretli çarpma algoritması ve ASM çizimi....97
Şekil 4-25 3-bitlik örgü çarpıcı devresi $011_2 \times 011_2 = 001001_2$ işlemini yapıyor....99
Şekil 4-26 İlk-bölücü için veri işleme donanımı....101
Şekil 4-27 İlk-bölücü algoritması....102
Şekil 4-28 İlk-bölücünün eşdeğer algoritması ve ASM çizimi....102
Şekil 4-29 İkinci-bölücü için veri işleme donanımı....103
Şekil 4-30 İkinci-bölme algoritması ve ASM çizimi....104
Şekil 4-31 Üçüncü bölme algoritması....105
Şekil 4-32 Üçüncü bölmeye eşdeğer algoritma....105
Şekil 4-33 Üçüncü bölme için veri işleme donanımı....106
Şekil 4-34 Üçüncü-bölücünün ASM çizimi....106

Şekil 4-35 Kayan-noktalı toplama algoritması. 116
Şekil 4-36 Kayan-noktalı Toplama/çıkarma veriyolu. 117
Şekil 4-37 Kayan noktalı çarpma algoritması 120
Şekil 4-38 Toplama, çıkarma, çarpma, bölme yapan hareketli-nokta aritmetik birimi için veri yolu diyagramı 121
Şekil 4-39 paralel yüklemeli (load) tipik 4-bit an-uyumlu kaydırıcı yazmaç.. 128
Şekil 4-40 *sıfırla* (*clear*) girişi olan tipik bir 3-bit senkronize sayıcı 129
Şekil 4-41 Bir ASM çiziminin elemanları 130
Şekil 4-42 4 bitlik sayılardaki sıfır-biti miktarını saymak için bir algoritma. 131
Şekil 4-43 Veri işleme donanımı ve denetim birimi sinyalleri................................ 132
Şekil 4-44 Hazırlık kutusuna karşılık gelen ASM bloğu, ve ilgili veriyolu 133
Şekil 4-45 t1 den t2'ye geçiş için zamanlama-diyagramı 134
Şekil 4-46 Diğer işlemlere karşılık gelen ASM bloğu ile veri yolu 135
Şekil 4-47 Sıfır-biti sayıcı için ASM-çizimi ve veriyolunun son biçimi 137
Şekil 4-48 Sıfır-biti sayıcı devresi için ASM-çizimi ve veri işleme donanımı zamanlama diyagramı 137
Şekil 4-49 2'nin tamamlayanı birimi için algoritma ve akış şeması 139
Şekil 4-50 2'nin tamamlayanı biriminin veri işlemcisi ve ASM çizimi 139
Şekil 4-51 1010_2'nin 2'lik tümleyeninin alınışının zamanlama diyagramı. 140
Şekil 4-52 Denetimin durum başı bir yazbozlu devre diyagramı. 141
Şekil 4-53 Controller implemented using a PLA device 142
Şekil 4-54 Denetimin ROM kullanılarak oluşturulması 143
Şekil 5-1 Aritmetik-mantık ve bellek aktarma komutları için MIPS gerçeklemesinin bir özet görünümü. Atlama ve dallanma komutları ek veriyolu ve toplayıcılar gerektirir. 146
Şekil 5-2 D-yazbozu ve tipik zamanlama diyagramı 147
Şekil 5-3 Yazmaç değerini bileşimsel bloğa aktaran 32-bitlik yol 148
Şekil 5-4 Arakat veri yolunu değiştirebilir. Hem E1 hem de E2 aynı anda secilirse çıkışları birbirine bağlı olduğundan çatışma oluşur. 148
Şekil 5-5 2^n den 1 'e ve 2 den 1 'e çoklayıcılar. 149
Şekil 5-6 Sırasal durum devresinde bileşimsel öbek. 150
Şekil 5-7 *write* yada *enable* girişli sırasal elemanlar. 150
Şekil 5-8 Tek saat dönuşünde güncelleme yapan devre ve zamanlama diyagramı 151
Şekil 5-9 4-artır-sayacı 151
Şekil 5-10 PC 'yi 4 arttır 152
Şekil 5-11 Komut okunması ve komutun bileşenleri 153
Şekil 5-12 Yazmaç okuma ve işlem sonucunun yazılması. 153
Şekil 5-13 R-tipi komutta ALU işlemi ve sonucun yazılması 153
Şekil 5-14 R-tipi komutları tek saat dönüşünde çalıştıracak ASM çizimi. Tek ASM öbeği olması denetim devresinin bileşimsel devre olduğunu gösterir. 154
Şekil 5-15 ALU bellek adresini topluyor 155
Şekil 5-16 LW komutu işlenirken kullanımda olan veri yolu 155
Şekil 5-17 SW komutu işlenirken kullanımda olan veri yolu 156
Şekil 5-18 R-tipi (RF) komutlar işlenirken kullanılan veri yolu 156
Şekil 5-19 LW, SW ve R-tipi (RF) komutları gerçeklemek için veri yolu diyagramı 157
Şekil 5-20 LW, SW ve R-tipi komutları gerçekleyen ASM çizimi. 157

Şekil 5-21 BEQ komutunun yeni veri yolu fazladan bir ALU gerektirir. 158
Şekil 5-22 *işaret-genişlet* (sign extend) ve *2-sola-kaydır* (shift-left-2) işlemlerinin devre bağlantısı düzeyinde gerçeklenmesi. 158
Şekil 5-23 R-tipi, lw, sw, beq komutlarını işleyen veri yolu çiziminin tümü 159
Şekil 5-24 *RT, lw, sw, beq* komutlarını işleyen veri yolu çiziminin ASM-çizimi. 159
Şekil 5-25 ALU-denetim devre şeması 162
Şekil 5-26 PLA ile ana-denetim-birimi gerçeklemesi 167
Şekil 5-27 Komut altkümesi (*R-tipi, lw, sw, beq* ve *j*) için veri yolu diyagramı. 168
Şekil 5-28 (*R-Format, lw, sw, beq* ve *j*) komut altkümesi için ASM çizimi 168
Şekil 5-29 Komut okuma (fetch) evresinde kullanılan veriyolları 169
Şekil 5-30 *R-formatı* komut: Komut-okumanın ardından (2. evre) 170
Şekil 5-31 R-formatı komut: 3. evre (son evre): 170
Şekil 5-32 Komut-okuma sonrası *lw* komutu: 171
Şekil 5-33 Komut-okuma sonrası sw komutu: 171
Şekil 5-34 I-tipi *beq* komutunu işleyen veri yolu 172
Şekil 5-35 J-tipi: *jump* (=*atla, j)* komutunu işleyen veri yolu 172
Şekil 5-36 Programın Akış şeması 173
Şekil 5-37 Verı belleğinin öbek gösterimi ve başlangıc değerleri 174
Şekil 5-38 Komut belleğindeki değerler. 174
Şekil 5-39 PC=200, *addi* komutu 175
Şekil 5-40 PC=204, *lw* komutu 175
Şekil 5-41 PC=208, *lw* komutu 176
Şekil 5-42 PC=212, *slt* komutu 176
Şekil 5-43 PC=216, *beq* komutu bir sonraki komuta devam ederken 177
Şekil 5-44 PC=220, *sub* komutu 177
Şekil 5-45 PC=224, *jump* komutu 178
Şekil 5-46 PC=212, *slt* komutu bu kez sıfırla sonuçlanır 178
Şekil 5-47 PC=216, *beq* komutu dallanma-hedef-adresine giderken 179
Şekil 5-48 PC=228, *sw* komutu 179
Şekil 5-49 Basitleştirilmiş bir tek-dönüşlü veri işlem yolu dört ana bölüme ayrılabilir 183
Şekil 5-50 Tipik basitleştirilmiş çok saat dönüşlü veriyolu örneği 183
Şekil 5-51 Çok saat dönüşlü veri yolunun geliştirilme aşaması 184
Şekil 5-52 *R-tipi lw*, *sw*, *beq*, ve *j* için çok-dönüşlü veriyolu. 195
Şekil 5-53 *R-tipi*, *lw*, *sw*, *beq* ve *j* komutları için ASM çizimi 197
Şekil 5-54 Çok-adımlı veriyolu denetlecinin FSM-çizimi 198
Şekil 5-55 Komut okuma, ve çözümleme adımları (t0, t1) 199
Şekil 5-56 R-tipi komutun yürütme ve yazmaca yazma adımlarının 200
Şekil 5-57 Bellek işlem komutları sw ve lw (t2, t3, t4, t5 adımları) 201
Şekil 5-58 *lw* komutu (t2, t3, t4 adımları) 202
Şekil 5-59 *sw* komutu (t2, ve t5 adımları) 203
Şekil 5-60 FSM denetim biriminin ROM ile tipik gerçeklenmesi 204
Şekil 5-61 Denetim biriminin ROM ya da PLC ile en basit gerçeklemesi. 205
Şekil 5-62 Denetim biriminin sıralayıcı devresi 207
Şekil 5-63 Sıralamalı denetim birimi şemasının tamamı 208

Tablo Başlıkları

Tablo 1-1 En Düşükten En Yükseğe Sayısal Bilgisayar Tasarım Seviyeleri. 2
Tablo 2-1 Okyanus ötesi uçakların özellikleri..5
Tablo 2-2 Daha yüksek başarım sonuçları elde edilen makinenin sistem betimlemesi, ...13
Tablo 2-3 İki programın, iki farklı makinedeki yürütme süreleri. 15
Tablo 2-4 Normalize edilmiş aritmetik ortalama yanıltıcı olabilir...............................15
Tablo 3-1 MIPS sisteminde bellek adresleme ve içerikleri 25
Tablo 3-2 Bellek içeriklerinin alışılmış gösterimi 25
Tablo 3-3 MIPS -in temel komut tipi biçimleri...32
Tablo 3-4 İçiçe yordam çağrıları ve yığıt işleyişi örneği...39
Tablo 3-5 Yığıt kullanılan içiçe çağrıda bellek ve yazmaç içerikleri........................39
Tablo 3-6 Seçilmiş MIPS komutlarının makine kodları 49
Tablo 4-1 tam-toplayıcı doğruluk tablosu...73
Tablo 4-2 32-bit-ALU -nun denetim sinyalleri...77
Tablo 4-3 Taşma koşulu için doğruluk tablosu, ve karşılık gelen elde giriş ve çıkış sinyalleri...79
Tablo 4-4 ALU İşlem kodları 81
Tablo 4-5 $0010_2 \times 1011_2$ çarpımının 4-bitlik birinci çarpıcıda izlenişi 89
Tablo 4-6 $0010_2 \times 1011_2$ çarpımının 4-bitlik ikinci-çarpıcı ile izlenmesi...................91
Tablo 4-7 $0010_2 \times 1011_2$ çarpımını ileri çarpıcı devresi ile izlenmesi.......................93
Tablo 4-8 İşaretli Üçüncü-çarpıcı devresi için $0010_2 \times 1011_2$ (= $2_{10} \times -5_{10}$) işaretli çarpımının izlenmesi 96
Tablo 4-9 Booth -un işaretli çarpıcısıyla $0010_2 \times 1011_2$ (=$2_{10} \times -5_{10}$) çarpımının izlenmesi...97
Tablo 4-10 A =1101 0000_2 nın B = 0001 0001_2 a ikilik sistemde bölünmesi............100
Tablo 4-11 Birinci bölücü devresinde 1011_2 / 0011_2 =11_{10} / 3_{10} işleminin izlenmesi.102
Tablo 4-12 İkinci bölme algoritmasıyla 1011_2 / 0011_2 bölmesinin izlenişi. 104
Tablo 4-13 1011_2 / 0011_2 bölümünü 4-bitlik üçüncü-bölücü ile izlenmesi. 107
Tablo 4-14 MIPS FPU Komutları 123
Tablo 4-15 t1 den t2 -ye geçiş için zamanlama-izleme tablosu 134
Tablo 4-16 Sıfır sayıcının U=1010_2 girişi için izleme tablosu.................................137
Tablo 4-17 2 -nin tamamlayanı biriminin izlenme tablosu.....................................140
Tablo 4-18 Denetim çıkışları ve FF-girişleri için Boole bağıntıları 140
Tablo 5-1 R-format MIPS komutlarının biçimi 152
Tablo 5-2 I-tipi MIPS komutlarından BEQ..157
Tablo 5-3 Veriyolu diyagramının işleyebildiği komutlar. 159
Tablo 5-4 32-bit ALU işlem kodları 160
Tablo 5-5 ALU-denetim çıkışı doğruluk Tabloları 161

Tablo 5-6 ALU-Denetim Biriminin Op çıkışı için Karnaugh çizimleri 161
Tablo 5-7 Ana Denetim Birimi Çıkış Sinyalleri 163
Tablo 5-8 ALU-denetimi birimi için gereken denetim birimi çıkışları 164
Tablo 5-9 Dallanma komutunda ALU -nun sıfır (*zero*) çıkışını kullanarak PCSrc seçme sinyalini üretmek için gereken denetim birimi çıkışları 164
Tablo 5-10 Ana Denetim Birimi Giriş Sinyalleri 164
Tablo 5-11 Basitleştirilmiş denetim-birimi çıkışları için doğruluk Tablosu 164
Tablo 5-12 Veri yolundaki dallanma devresinden kurtulmak için kullanılacak ek denetim birimi giriş çıkışları. 165
Tablo 5-13 Dallanma devresiz veriyolu denetlecinin doğruluk Tablosu 166
Tablo 5-14 MIPS komutlarının J-tipi formatı... 167
Tablo 5-15 Komut Belleğinde Program Kodu 173
Tablo 5-16 Ana işlev birimlerinin işlem süreleri.. 180
Tablo 5-17 Tek dönüşlü işlemcide her komutun minimum saat dönüş süresi 180
Tablo 5-18 Başarım bulmakta kullanılacak komut karışımı 181
Tablo 5-19 Çok-dönüşlü veri yolunda her dönüşteki ALU kullanımı 184
Tablo 5-20 Çok ve tek donüşlü veriyolu donanım elemanlarının karşılaştırılması .. 185
Tablo 5-21 Çok-dönüşlü-Veriyolu Denetim sinyalleri... 187
Tablo 5-22 Her komut tipi için işlem adımlarına kaynak dağıtımı 189
Tablo 5-23 Tasarım aşamasında komut okuma adımında verilecek denetim sinyalleri 190
Tablo 5-24 Tasarım aşamasında komut çözümleme adımında verilecek denetim sinyalleri.. 190
Tablo 5-25 Tasarım aşamasında yürütme adımında verilecek denetim sinyalleri ... 191
Tablo 5-26 Tasarım aşamasında bellek erişim adımında gereken denetim sinyalleri 191
Tablo 5-27 Tasarım aşamasında geri-yazma adımında gereken denetim sinyalleri 192
Tablo 5-28 PCSrc çoklayıcısının denetim sinyalleri 192
Tablo 5-29 Dallanma adresleme için ek denetim sinyalleri.................................... 193
Tablo 5-30 Çok dönüşlü veriyolunda gereken denetim çıkışları. 194
Tablo 5-31 Denetim biriminin çıkış ve sonraki durum doğruluk tablosu 205
Tablo 5-32 1.nci dağıtım tablosu (t1 durumu için) 207
Tablo 5-33 2.nci dağıtım tablosu (t2 durumu için) 207
Tablo 5-34 Sıralamalı denetim biriminin çıkış ve sıralama kodu doğruluk tablosu.. 208
Tablo 5-35 Mikrokomut alan adları ve işlevleri 210
Tablo 5-36 Tipik mikrokomut alanları ve bu alanlarda kullanılacak anımsatıcılar ... 210
Tablo 5-37 Tipik mikroprogramlama amaçlı sembolik dağıtım-1 ve dağıtım-2 tabloları 211
Tablo 5-38 *R-tipi*, *lw*, *sw*, *beq*, ve *j* için mikroprogram kodu 211
Tablo 5-39 Mikrokomut alanlarında kullanılan anımsatıcılar. 212

Önsöz

Bu kitap, *Bilgisayar Organizasyonu: RISC Donanımına Giriş"* ileride bilgisayar mühendisliğinin yazılım ve donanım alanlarında çalışması beklenen bilgisayar mühendisi adayları için yazılmıştır.

Kaynağı

Bu kitap temel olarak üniversite seviyesindeki Bilgisayar Mimarisi (Computer Architecture) dersi notlarından derlenmiş ve türkçeye çevirilmiştir. Ders notlarına doğrudan ve dolaylı katkılarından dolayı meslektaşlarım Dr. Hasan Kömürcügil, ve Dr. Muhammed Salamah'a en derin minnet duygularımı sunmak isterim.

Amaç ve Katkı

Metin, bir "Bilgisayar Mimarisi" ders kitabını okumaya yardımcı materyal olarak hazırlanmıştır. Okuyucuya açıksözlu önerim, kitabın bölümlerini okumadan önce diğer ders kitaplarından ilgili bölümleri okumasıdır. Bu kitap herhangi bir ders kitabının basitleştirilmiş örneği değildir. ASM (Algorithmic State Machine) çizimlerine varıncaya dek detaylarıyla, indirgenmiş komut takımlı bir işlemcinin devre bileşenlerini içerir.

Okuyucular

Bu kitabın muhtemel okuyucusunun *Sayısal Tasarıma Giriş (Introduction to Digital Design)* dersini tamamlamış bir üçüncü sınıf öğrencisi olması beklenir. Yine de, okuyucunun sayısal tasarımı her yönüyle bildiği varsayılmaz. Metin, okuyucuya sayısal tasarımın esaslarını ve yöntemlerini hatırlatacak gerekli ipucu ve örneklerle hazırlanmıştır.

Kitap Nasıl Okunmalı

Kitaptaki bölüm numaraları bilinçli olarak çok kullanılan benzer ders kitaplarına paralel düzenlenmiştir. Örnekler, detaylı çözümler içermektedir ve okuyucu tarafından eksiksiz anlaşılmalıdır. Kavramsal akıcılığı sağlamak için dördüncü bölümde, metinde yeri geldikçe çeşitli devrelerin ASM çizimleri verilmiş, ardından bölüm sonunda ASM çizimlerinin kullanımı derinlemesine anlatılmıştır. ASM çizimlerini hiç tanımayan okuyucular dördüncü bölüme başlamadan önce bu kısmı okuyabilirler. ASM çizimleri beşinci bölümde de tipik tek ve çok dönüşlü veriyollarının denetim birimlerini tanımlamak için kullanılmıştır.

Kitaba Genel Bakış

Bu kitapta basitleştirilmiş bir 32-bit RISC işlemciyi gerçekleştirmek için gerekli temel bileşenleri ile işlevlerinin betimlenir.

Birinci bölüm, hedef çalışmanın sınırlarını belirten genel bir tanıtım niteliğindedir.

İkinci bölüm, bilgisayar sistemlerinin başarımını karşılaştırmak için olduğu kadar, yeni işlemcilerin tasarım ve geliştirilmesi için de uygun olan bir başarım ölçütü tanımlar. Bu bölüm ayrıca bilgisayar tarihinde kullanılmış çeşitli başarım ölçütlerini de ele alır.

Üçüncü bölüm, RISC kavramının C-dili programlarını daha verimli işlemek üzere komut takımının basitleştirilmesinin bir sonucu olduğunu hatırlatarak, uygulama seviyesi programlama dilleri ile makine seviyesi komutları arasındaki ilişkiye odaklanır.

Dördüncü bölüm, aritmetik mantık birimiyle kayan noktalı sayı biriminin işlevsel çalışmasını ve fiziksel gerçeklemesini tanıtır. Çarpma ve bölme birimlerinin çalışması yazmaç (register) seviyesine kadar ayrıntılı ASM çizimleriyle betimlenir.

Beşinci ve son bölüm, denetim biriminin komut başına tek-saatlik ve çok-saatlik uygulamalarındaki yapıları ile tasarımını kapsar ve tipik bir RISC işlemcinin, temel yazmaç seviyesindeki bileşenlerden başlayarak tasarım ve mimarisinin anlaşılmasını hedef alır.

Kitabın Önemi

Özellikle son yıllarda flaş bellek ve yerinde programlanır kapı dizileri (FPGA) teknolojısındeki gelişmeler sonucu bu ürünlerin ucuzlaması artık sayısal devre tasarımcılarının kendi uygulamalarına has işlemcilerini ve komut takımlarını geliştirmesine olanak sağlamaktadır. Bu kitap VHDL ortamda kendi işlemcilerini tasarlamak isteyenlere RISC veri yolu ve komut takımı geliştirme tekniklerini tanıtması açısından önem taşır.

Yasal Uyarı: Yazar kitabın hazırlanmasında araştırma, geliştirme, teori ve programların analiz ve testleri dahil üstün çaba göstermiştir. Yazar kitaptaki malzemeyle ilgili hiçbir garanti vermez. Kitaptaki fikir, teori, resim ve yazıların aynen ya da değiştirilerek kullanılmasından doğabilecek hasar ya da sonuçlardan yazar kesinlikle sorumlu tutulamaz.

Ekim 2003,
Dr. Mehmet Bodur.

1 Giriş

1

Dünya 1970'lerden bu yana bilim ve teknolojideki gelişmeyi besleyen ve bunların gelişmesinden destek alan bilgi devrimini yaşıyor. Bilgi devrimi boyunca bilgisayar devrimi de ilerliyor. Moore yasası hala geçerliliğini koruyor:

> İntel'in kurucularından Gordon Moore 1965'de tümleşik devrelerdeki birim yuzey başına düşen transistor sayısının tümleşik devrenin icadından beri her yıl ikiye katlandığını gözlemledi. Moore bu eğilimin, sağlıklı öngörunün mümkün olduğu sürece devam edeceğini öngördü. Takip eden yıllarda, hız biraz azalsa da veri yoğunluğu yaklaşık her 18 ayda iki katına çıktı. Bu olgu Moore yasası olarak tanımlandı. Moore da dahil çoğu uzmanlar, Moore yasasının en azından yirmi yıl daha geçerliliğini koruyacağını beklemekteler.

Bilgisayar devriminin etkilerini hayatımızın heryerinde görüyoruz. Otomobillerimizde, medikal cihazlarımızda, otomatik banka makinelerinde (atm), ve benzer birçok yerde cihaza gömülü bilgisayarlar var. Dizüstü bilgisayarlarımız kafelere ve dersliklere taşınacak derecede küçülmesine karşın birden fazla kullanıcı ve birden fazla iş destekleyebilme yeteneğine erişti. Güçlü süperbilgisayarlar insan geni projelerinde, uydu görüntülerini incelemede, çevre kirliliği ve hava durumu tahminlerinde kullanılıyor. RISC işlemcilerin geliştirilmesi ve yaygın kullanılması son yirmi yılda bilgisayar devriminin önemli bir adımını oluşturdu.

1.1 Bilgisayar Tasarımı Seviyesi Hakkında

Bu kitapta dikkate aldığımız tasarım seviyesinin, bilgisayarların geleneksel tasarım seviyelerinin şeması içindeki yerini belirterek başlayalım. Bilgisayar sistemi tasarımlarının iki temel bileşeni vardır, yazılım ve donanım. Bu nedenle, tablomuzun bilgisayarların donanım ve yazılım tasarımı seviyeleri için iki ana sütunu olacaktır.

Bu kitap Tablo 1.1-1'in üçüncü maddesinde tarif edilen seviye üzerine, özellikle denetim birimlerinin (CU) ve aritmetik mantık birimlerinin (ALU) tasarımı üzerine yoğunlaşmıştır. Donanım tasarımının yanı sıra

komut takımının ve mikroprogramlamanın anlaşılması içeriğin önemli bir bölümünü oluşturur. ALU ve CU -nun karmaşık yapısı bizi olabildiğince basit fakat güçlü ve yaygın kullanılan bir işlemci birimini örnek vermeye zorlar. RISC işlemcileri, komut takımlarının ve donanım yapılarının basitliği ile bize bu olanağı sağlamaktadır.

Tablo 1-1 En Düşükten En Yükseğe Sayısal Bilgisayar Tasarım Seviyeleri.

	Donanım tassarım seviyesi	**Yazılım tasarım seviyesi**
1	**Anahtarlamalı devreler** Anahtar öğe transistör, diyot, röle kontağı, mekanik şalter, vb. olabilir. Devre temel mantık işlemlerini gerçekleştirmek için çalışır	**Fiziksel bağlantılı** Devrenin işlevi yine kendisinin tasarımıyla, yani devre elemanları arasındaki bağlantılarla (hard wired) belirlenir.
2	**Sayısal ve Mantık Devreleri (sayısal tasarım)** "Evirici", "ve" "veya" geçitleri, yazbozlar, yazmaçlar, dekoderler, multiplexerler, PLA -lar vs.	**Fiziksel bağlantılı** İşlev, geçitler ve modüller arasına bağlanmış hatlar (wire) tarafından belirlenir.
3	**Bilgisayarın Bileşenleri** Yalnız-okunur-bellek (ROM), rastgele-erişimli-bellek (RAM), yazmaçlar, veriyolu denetleçleri, adres, veri ve giriş/çıkış tamponları**, denetim birimi, aritmetik mantık birimi, vs.** **İşlemci birimin tasarımı komut takımının tasarımıyla birarada yürütülür.**	**Mikroprogramlama ve Fiziksel yazılım** **Denetim birimini mikroprogramlama, programlamanın en alt düzeyidir ve veriyolu, ALU ve yazmaçlar gibi diğer bileşenlerin donanım tasarımıyla doğrudan etkileşir.**
4	**Bilgisayarlar ve Mikrobilgisayarlar** Çekirdek işlemci modulü bir kere hazırlandıktan sonra, bir bilgisayar oluşturmak üzere RAM, ROM, ve çeşitli giriş/çıkış kapıları (I/O port) çekirdeğe bağlanır.	**Çevirici ve ikili (binary) kodlama** İşlemci sadece ikili program kodlarını işleyebilir. Kod çoğunlukla sembolik çevirici dilinde yazılır. Yazımda Editör programı kullanılır. Ardından, Çevirici (Assembler) çevirici kaynağını ikili koda çevirir.
5	**Bilgisayar ve Mikrobilgisayarların Uygulama Seviyesi** Bilgisayarlar karmaşık bir aygıt ya da cihaza, fabrikaya, kuruluşa vs. gömülü olabilir. Ya da genel amaçlı ve çok görevli, çok kullanıcılı hesaplama işleri için kullanılırlar.	**Ust Düzey Programlama** Bir işletim sistemi, cihaz sürücüleri ve uygulama programı gömülü işlemcinin çevre birimlerine hizmet verir. Uygulama programı genellikle yüksek seviye dillerle yazılır ve ikilik koda dönüştürülür. Genel amaçlı bilgisayarlarda işletim sistemi, hem kullanıcı hem de derleyici ve uygulama programlarının çevrebirim arayüzü için servis sağlar.
6	**Dağıtımlı Bilgisayarlar, Bilgisayar Ağları, vs.** Birbirine bağlı çok sayıda işlemcili bilgisayar sistemleri. ...	**Eşzamanlı programlama** **Ağ (Network) İşletim Sistemleri, vs.**

1.2 MIPS R2000 işlemcisi hakkında

Bu metinde tanıtılan temel mimari, 1983’de Stanford’daki araştırmaların ardından somutlaşmış ve MIPS-II işlemci çekirdeği biçimiyle gerçekleşmiş olan 32-bit RISC mimarisidir. 2002'lerin tipik 0.13 μm tümleşik devre teknolojisiyle, MIPS-II çekirdeği önbellek hariç neredeyse 0.4 mm^2 yer tutar. İşlemcinin harcadığı güç temel olarak çalışma frekansına bağlıdır. Tamamen statik tasarım çekirdeğin dc -den 200 MHz -e kadar herhangi saat hızında çalışmasını sağlar. İndirgenmiş komut takımı bilgisayarı (reduced instruction set, RISC) endüstride geniş kabul görmüş ve desteklenmiş bir tasarımdır, ve işletim sistemleri, derleyiciler, yanlış ayıklayıcılar, devre içi öykünücüleri, mantık çözümleyiciler ve değerlendirme kartları gibi çeşitli yazılım geliştirme araçları ve uygulamaları mevcuttur. Çoğu komut tek saat döngüsünde çalışır ve 250MHz de 0.7mW güç harcar.

Bu işlemcinin çeşitli alanlarda tipik uygulamaları vardır. Mobil telefonlar, kablo modemler, TV ve sayısal kameralar, DVD -ler, endüstriyel kontrolörler, otomobil bilgisayarları, PC çevrebirimleri tipik uygulamalar arasındadır. Aynı mimarinin 64-bit sistem için düzenlenmişi sunucu ve üst düzey bilgisayar istasyonlarında (high-end workstation) geniş ölçüde kullanılır.

1.3 RISC -in Kısa Tarihi

IBM System/360, 1950’lerin sonlarında bir dizi yazılım uyumlu, bunların yazılımlarına yapılan yatırımları çıkarabilecek bilgisayarlar oluşturmaya yönelik araştırma programının sonucunda 7 Nisan 1964 de ortaya çıkmıştır. System/360, karmaşık komut takımının genişlemesine izin veren mikroprogramlanmış bilgisayar mimarisinden piyasada ilk görünendi. CISC (karmaşık komut takımlı bilgisayar) mimarili ilk ticari sistemdi. System/360’ın başarısı CISC mimarisinin yirmi yıl boyunca bilgisayar tasarımına hükmetmesine neden oldu. Ardından, CISC mimarisi mikroişlemcilerde de başarıyla uygulandı.

1970’lerin ortalarında, başarım ölçüm teknikleri geliştikçe, uygulama programlarının çoğunda karmaşık komutların nadiren kullanıldığı ve başarımı birkaç basit komutun belirlediği gösterildi. Ekim 1975’de IBM’in Watson Araştırma Merkezinde donanımı ve komut takımını yeni amaçlar ve başarım ölçütleri için optimize etmek amacıyla bir proje başlatıldı. Dört yıl sonra, ilk 32-bit RISC mikroişlemci prototipi geliştirildiği binanın adından yola çıkılarak "801" adıyla anıldı.

801 hiçbir zaman piyasaya sürülmedi. Onun yerine, "Araştırma/Büro Ürünleri Mikroişlemcisi" (Research/Office Products Microprocessor – ROMP) adıyla benzer tek-yongalık bir işlemci üretildi. IBM Ocak 1986'da ilk RISC sistemi üretim olarak PC RT -yi (PA-RISC sistem) çıkardı. PA-RISC günümüz RISC işlemcilerinin birinci kuşağıydı. 1983'de , yeni kurulan birkaç sunucu şirketi, özellikle Ridge Computer, Pyramid Technology, ve ardından Computer Consoles, RISC temelli sistemlerle piyasaya girerek IBM ve HP -yi zorlamaya başladı., İşletim sistemi ve uygulamaları oluşturmak için yalnızca bir C derleyicisi gerektirdiğinden UNIX bu sistemlerin doğal işletim sistemi oldu. Ancak bu şirketler IBM -in dev gücünü tüketemedi, ve IBM UNIX kullanımına geçince de pazarı terkettiler.

Bu arada RISC özellikle Stanford'da ve Berkeley'de akademik çevrelerin dikkatini çekti. Stanford'da IBM'in ardışıkdüzen (pipeline) verimi ile derleyici optimizasyonuna dayanan, sonradan MIPS mimarisine dönüşen yapı, Berkeley'de ise bol yazmaç kullanarak alışılagelmiş bellek üzerindeki çağrıları hızlandıran ve 1987'de Sun Microsystems Inc. tarafından uyarlanan ürunler geliştirildi. 1988 yılında, RISC işlemciler Motorola 68000'ün iş istasyonu pazarının büyük kısmını ele geçirdi ve birkaç yıl içinde sunucu pazarında hakim oldu. Günümüzde, hemen bütün iş istasyonlarının kalbinde ve sunucu mikroişlemcilerinde RISC çekirdekler bulunmaktadır.

1.4 Sonuç

Son zamanlarda flash bellek ve alanda programlanır kapı dizileri teknolojilerindeki gelişmeler mühendislere bir yonga içinde sisteme uygun adresleme kipleri ve komut kümelerine sahip kendi uygulamaya-özel-işlemcilerini geliştirebilme olanağı sunuyor. Bu kitap bu çeşit uygulamaya özel tek yongalık denetleçlerin VHDL tasarım basamağının önkoşulu sayılabilecek RİSC işlemcilerin en temel tasarım tekniklerini tanıtmaktadır.

2 Başarım Ölçütleri

2

Amaç : Bu bölümün hedefi, çeşitli bilgisayar başarım ölçütleri arasındaki ana farkların, tasarım ve satın alma açısından kavranmasıdır.

2.1 Giriş

Bü bölümde bilgisayarların başarımlarının nasıl ölçüleceği, rapor edileceği ve karşılaştıralacağı tartışılacaktır. Farklı tür bilgisayarların performansını değerlendirebilmek, bu makineler arasında en iyi seçim yapmada anahtar etmendir. Bizim ilgimiz bu seçimin de ötesinde, performansın tasarım projesinde yeni tasarımın başarısına değer biçmede belirleyici olmasıdır.

Performans ölçümündeki karışıklık birçok temel etmenden doğar. Komut takımı ve bu komutları tamamlayan donanım önemli ana etmenlerdir. Aynı donanıma ve komut takımına sahip iki bilgisayar bile bellek ve i/o örgütlenmesi, ve de işletim sistemi nedeniyle ya da sadece testlerde farklı iki derleyici kullanıldığından dolayı farklı başarım verebilir. Bu etmenlerin başarımı nasıl etkilediğini belirlemek, makinenin belirli yönlerinin tasarımının dayanağı olan ana güdüyü anlamak açısından çok önemlidir.

Yolcu uçaklarıyla ilgili bir örnekle başlayalım. Eğer mevcut uçaklarla ilgili aşağıdaki tablo verilse, ve 5000 yolcuyu New York'tan Paris'e (1500 km) taşımamız gereksa, hangi uçak en iyi çözümdür?

Tablo 2-1 Okyanus ötesi uçakların özellikleri

Uçak	*P*: Yolcu kapasitesi	*R*: Uçuş menzili (km)	*S*: Uçuş hızı (km/saat)	$(P \times S)$ Yolcu gönderme-hızı
Boeing 737-100	101	1000	1000	101 000
Boeing 777	375	7400	1000	375 000
Boeing 747	470	6650	1000	460 600
BAC/Sud Concorde	132	6400	2 200	290 400
Douglas DC-8-50	146	14000	880	128 480

Boeing 737-100 uçuş menzili New York'tan Paris'e uçmaya yetmeyeceğinden daha ilk bakışta listeden elenir. Ardından, geriye kalan d arasında Concorde -un kesinlikle en hızlısı olduğunu görürüz.

Ancak, uçağın sadece hızı yeterli midir? Boeing 747 daha yavaş olmasına karşın, bir Concorde -un taşıyabileceğinden üç kat fazla yolcu taşıyor. Uçakların performansları için daha iyi bir ölçüt onların yolcu taşıma hızı olabilir, yani, yolcu sayısının uçağın hızıyla çarpımından çıkan sayı. Bu durumda 747 -nin 5000 yolcuyu taşımada daha başarılı olduğunu görürüz, çünkü onun yolcu taşıma hızı Concorde -dan daha yüksektir. Diğer taraftan, eğer toplam yolcu sayısı 132 -ten az olursa Concorde elbetteki Boeing 747 -den daha iyidir, çünkü onları 747 -den neredeyse iki kat hızlı taşıyacaktır. Yani, performans büyük oranda yapılacak işe bağlıdır.

2.2 Başarım Tanımı

Bir bilgisayarın diğerinden daha iyi başarıma sahip olduğunu söylemekle, tipik uygulama programlarımız açısından o bilgisayarın birim sürede diğerinden daha çok iş bitirebildiğini kastederiz. Bir programı iki farklı iş istasyonunda çalıştırırsanız, işi ilk tamamlayan istasyonun daha hızlı olduğunu söyleriz. Bu basitleştirilmiş başarım tanımında, iki bilgisayar arasındaki fiyat farkını gözardı ettik, ya da ikisinin fiyatını yaklaşık aynı kabul ettik.

Zaman paylaşımlı çok-kullanıcılı çok-görevli bir bilgisayarda, bir programın başlangıcından bitişine kadar geçen toplam zamana *toplam yürütme süresi* denir. Bu süre bilgisayarın başka birçok kullanıcının işi de başarımını paylaşır. Genellikle giriş/çıkış ile programınızın işlemesi için geçen net süre ayrı ayrı sayılır. Bunlar işinizin *CPU süresi* ve *i/o işlem süresi* diye adlandırılır. Zaman paylaşımlı anabilgisayarlarda diğer kullanıcıların işleri arasında çalışan programın çalışma süresi CPU süresinden daha uzundur. Tabii ki bir kullanıcı olarak, sizi toplam çalışma süresi ilgilendirecektir, ama bilgisayar merkezinin yöneticisi bilgisayarın toplam iş-bitirme-hızıyla (*throughput*) ilgilenecektir.

Programların CPU süresini azaltmak için çeşitli yöntemler vardır. Bunlar arasında, bilgisayarı aynı tip daha hızlı sürümüyle değiştirmek ilk akla gelenidir ve başarımı kısmen arttırır. CPU zamanını etkilememesine karşın ekstra işlemci ve bellek birimi eklemek te işletim-hızını iyileştirerek yanıt zamanını azaltabilir.

Belli bir görevde, *X* bilgisayarının başarımı temel olarak programın çalışma zamanıyla ters orantılıdır.

$$\text{X.in Başarımı} = \frac{1}{\text{X.in Çalışma Süresi}}$$

Bu da, *X* ve *Y* bilgisayarlarının başarımı çalışma zamanıyla ters sıralanır demektir. Yani

Y.nin Çalışma Süresi > X.in Çalışma Süresi

ise

X.in Başarımı > Y.nin Başarımı

demektir. Nicel olarak,

$$\frac{X.in\ başarımı}{Y.nin\ başarımı} = \frac{Y.nin\ çalışma\ süresi}{X.in\ çalışma\ süresi} = n$$

ise *X in Y den n kat hızlı* olduğunu söylenir.

2.2.1 Ölçme Koşulları ve Ölçme Birimleri

Çok görevli ve çok kullanıcılı bir bilgisayar ortamında yürütme süresi ve belli bir iş için harcanan işlem süresi farklı kavramlardır.

- Programın başlatılmasından bitişine kadarki zamana *toplam yürütme süresi*, *yanıt zamanı*, *geçen süre* ya da *duvar süresi* denir.
- Programın işlemesinde CPU tarafından harcanan zaman dilimlerinin toplamına *CPU yürütme süresi* ya da basitçe *CPU süresi* denir.
- CPU süresi daha da ayrışarak *program CPU süresi* ve *sistem CPU süresi* -ne bölünür. Sistem CPU süresi içinde i/o, disk erişimi ve benzeri diğer çeşitli sistem görevleri yapılır. Program CPU zamanı ise yalnızca program kodunun yürütülmesi için geçen net süredır.

CPU tasarımında özellikle program CPU yürütme başarımına yoğunlaşacağız. Bu nedenle hesaplamalarımızı test programının net *CPU yürütme süresi* üzerine kuracağız.

Zaman genellikle *saniye* (s) birimiyle ölçülür. Ancak, *saat-dönüş-süresi*, yani bilgisayar saatinin bir periyodu çoğunlukla *nanosaniye* (*nano*= 10^{-9} = 1/ 1 000 000 000) kullanılarak ölçülür. Genelde, bilgisayarların hızlarını verirken saat dönüşü yerine *saat hızı* (=1/saat-dönüşü) tercih edilir. Saat hızının alışılagelen birimi *Hertz* (Hz) -dir. Hertz *saniyedeki-dönüş-sayısı*na eşittir. Daha hızlı saatler için *kilo-Hertz*, *mega-Hertz*, ya da *giga-Hertz* terimleri kullanılır.

Zaman Birimleri	saniye	mili-saniye	mikro-saniye	nano-saniye
kısaltması	s	ms	μ s	ns
saniye eşdeğeri	1	0.001	0.000 001	0.000 000 001

Frekans Birimleri	Hertz	kilo-Hertz	mega-Hertz	giga-Hertz
kısaltması	Hz	kHz	MHz	GHz
saniyedeki dönüş	1	1000	1 000 000	1 000 000 000

Bilgisayarların başarımlarını karşılaştırırken, gerçekte kullanılacak uygulama programlarının iş-bitirme-hızı son derece önemlidir. Bir programın *CPU yürütme süresini* belirleyen temel ifade

$$\text{CPU-yürütme-süresi} = \text{CPU-saat-dönüş-sayısı} \times \text{Saat-dönüş-süresi}\ ;$$

$$\text{CPU-yürütme-süresi} = \frac{\text{CPU-saat-dönüşü-sayısı}}{\text{Saat-hızı}}\ ;$$

biçimindedir.

CPU-saat-dönüşü-sayısı genellikle *komut-başına-ortalama-saat-dönüşü-sayısı*yla *komut sayısı*nın çarpımından elde edilir.

$$\text{CPU-saat-dönüşü-sayısı} = \text{komut-sayısı} \times \text{komut-başına-ortalama-dönüş-sayısı}$$

Komut başına ortalama dönüş sayısı genellikle *CPI* (cycle-per-instruction) olarak kısaltılır.

Örnek 2-1: *A* ve *B* aynı komut takımına sahip iki makine olsun. Herhangi bir program için *A* -nın saatdönüşü 10ns ve *CPI* -si 2.0 ölçülmüş, aynı program için *B* -nin saatdönüşü 20ns ve *CPI* -si 1.2 ölçülmüş. Bu program açısından hangi makine kaç kat hızlıdır?
<<
Çözüm: Programdaki komut sayısının I olduğunu varsayalım. Bu durumda

$$\text{CPU-süresi-A} = \text{CPU-saat-dönüşü-sayısı-A} \times \text{Saat-dönüş-süresi-A}$$
$$= I \times 2.0 \times 10\ ns = 20\ I\ ns$$
$$\text{CPU-süresi-B} = I \times 1.2 \times 20\ ns = 24 \times I\ ns\ ,$$

$\text{CPU-süresi-A} < \text{CPU-süresi-B}$, o halde *A* daha hızlı.

Peki ne kadar?

$$\frac{\text{Başarım-A}}{\text{Başarım-B}} = \frac{\text{Çalışma-süresi-B}}{\text{Çalışma-süresi-A}} = n = \frac{24 \times I\ ns}{20 \times I\ ns} = 1.2$$

A makinesi B -den 1.2 kat hızlıdır. >>

Başarım ölçmenin temel denkliğini yeniden yazarsak:

$$\boxed{\text{CPU-süresi} = \frac{\text{komut-sayısı} \times \text{CPI}}{\text{saat-hızı}}}$$

Bu bölümde, gerçek uygulama programlarının çalışma zamanının ölçümünden çok daha kolay olan çeşitli yaygın başarım ölçütlerini, ve bunların uygulanabilirliği ile hatalarını göreceğiz.

2.3 Yaygın Kullanılan Yanıltıcı Başarım Ölçütleri

MIPS ve MFLOPS, sistem başarımını karşılaştırmak için sık kullanılan başarım ölçütleridir. Bu iki başarım ölçütü birçok durumda yanıltıcı olabilir, ve dikkkatli kullanılmalıdır.

2.3.1 *MIPS* Başarım Ölçümü

MIPS saniyede milyon komut) için kısaltmadır. Bir programda,

$$MIPS = \frac{\text{komut-sayısı}}{\text{yürütme-süresi} \times 10^6} = \frac{\text{komut-sayısı}}{\text{CPU-saat-dönüş-sayısı} \times \text{saat-dönüş-süresi} \times 10^6}$$

$$= \frac{\text{komut-sayısı} \times \text{saat-hızı}}{\text{komut-sayısı} \times CPI \times 10^6}$$

burada $\text{CPU-saat-dönüşü-sayısı} = \text{komut-sayısı} \times CPI$ olduğundan

$$MIPS = \frac{\text{saat-hızı}}{CPI \times 10^6}$$ (doğal MIPS)

Şimdi, biliyoruz ki

$$\text{çalışma-süresi} = \frac{\text{komut-sayısı} \times CPI}{\text{saat-hızı}} = \frac{\text{komut-sayısı}}{(\text{saat-hızı} \times 10^6 \ / \ CPI \times 10^6)}$$

$$\text{çalışma-süresi} = \frac{\text{komut-sayısı}}{MIPS \times 10^6}$$

Baştan sezdiğimiz gibi bu eşitliğe göre de *hızlı makinenin MIPS değeri yüksektir* diyebiliriz. Ancak unutmamalıyız ki, aynı iş makinaların komut takımındaki ya da derleyicilerindeki farklar nedeniyle farklı sayıda komut gerektirebilir.

2.3.2 *MIPS* ölçümü kullanmanın sakıncaları

Birçok durumda *MIPS* başarım ölçütü olarak kullanılmaya uygun değildir.

1. Aynı iş için kullanılan komut sayıları farklı olacağından farklı komut takımlarına sahip bilgisayarları MIPS kullanarak karşılaştıramayız.
2. Aynı bilgisayarda çalıştırılan farklı programlar farklı MIPS değerleri verir. Bir makinenin tek bir MIPS değeri olamaz.

Bazı durumlarda (aşağıdaki örneğe bakınız) *MIPS* gerçek performansa ters yönde değişebilir.

Örnek 2-2: Üç farklı tipte komutu olan bir makine düşünün, *A-tipi* 1, *B-* 2, ve *C-* ise 3 saat dönüşü tutsun. Makinenin saat hızı 100 MHz verilsin. Aynı programın iki farklı derleyiciden çıkmış kodlarının çalışma süresini ölçmeye çalıştığımızı düşünün:

kodu oluşturan	**komut sayısı (milyon)**		
	A-tipi	*B-tipi*	*C-tipi*
Derleyici-1	5	1	1
Derleyici-2	10	1	1

MIPS -e gore hangi derleyicinin kod parçası daha hızlı çalışıyor?
Çalışma süreleri açısından hangi kod parçası daha hızlı çalışıyor?

<<Çözüm:

$$MIPS = \frac{saat\text{-}hızı}{CPI \times 10^6} = \frac{100\ MHz}{CPI \times 10^6};$$

$$Böylece,\ CPI = \frac{CPU\text{-}saat\text{-}dönüşü\text{-}sayısı}{komut\text{-}sayısı};$$

burada, $CPU\text{-}saat\text{-}dönüşü\text{-}sayısı = \sum (CPI_i \times Komut\text{-}sayısı_i)$

Her derleyici için toplam CPI -yı bulmak amacıyla şu eşitliği kullanırız:

$$CPI = \frac{\sum (CPI_i \times Komut\text{-}sayısı_i)}{Komut\text{-}sayısı};$$

Durum 1

$$CPI = \frac{((5\times1)+(1\times2)+(1\times3))\times10^6}{(5+1+1)\times10^6}$$

=10 / 7 = 1.43

$$MIPS_{derleyici\text{-}1} = \frac{100\times10^6}{1.43\times10^6} \cong 70$$

Durum 2

$$CPI = \frac{((10\times1)+(1\times2)+(1\times3))\times10^6}{(10+1+1)\times10^6}$$

=15 / 12 = 1.25

$$MIPS_{derleyici\text{-}2} = \frac{100\times10^6}{1.25\times10^6} \cong 80$$

Demek ki, MIPS-değerine göre, derleyici-2 daha yüksek başarımlı. Şimdi, çalışma sürelerini hesaplayalım:

$$CPU\text{-}süresi = \frac{Komut\text{-}sayısı \times CPI}{Saat\text{-}hızı};$$

$$CPU\text{-}süresi\text{-}1 = \frac{(5+1+1)\times10^6\times1.43}{100\times10^6}$$

=0.10 s

$$CPU\text{-}süresi\text{-}2 = \frac{(10+1+1)\times10^6\times1.25}{100\times10^6}$$

=0.15 s

Çalışma süresine göre ise derleyici-1 daha hızlı. Demek ki MIPS değerine bakarak varılan sonuc yanlış. Bu da ***MIPS değeri,***

bilgisayarların başarımı için doğru ölçüt değildir anlamına gelir.
>>

Örnek 2-2 aynı programın iki sürümünü aynı makinede karşılaştırırken bile *MIPS* değerinin başarımı doğru ölçmede yetersiz kaldığını.gösteriyor. Eğer farklı komut takımına sahip iki makineyi karşılaştırıyorsak, MIPS daha da yanıltıcı olabilir.

2.3.3 Tepe *MIPS*

Tepe-MIPS, işlemcinin *MIPS* ölçümü hesabında kullanılan *CPI* -yi en aza indiren komut karışımı seçilerek elde edilir. Ancak, bu karışım tümüyle gerçekdışı ve uygulanamaz nitelikte olabilir. Bu yüzden *tepe-MIPS* kullanışsız bir ölçüttür.

2.3.4 Göreceli *MIPS*

Göreceli MIPS şu eşitlikle hesaplanır

$$\text{Göreceli MIPS} = \frac{T\text{-bilinen} \times MIPS\text{-bilinen}}{T\text{-ölçülen}} \quad .$$

Burada

T-bilinen = programın bilinen bir makinedeki çalışma zamanı
T- ölçülen = programın ölçülecek makinede çalışma zamanı
MIPS-bilinen = bilinen makinenin, genellikle VAX11/780, kabul görmüş *MIPS* değeri

Göreceli MIPS metriği sadece verilen bir program ve verilen girdi için doğrudur.

1970'lerin ve 1980'lerin süper bilgisayar endüstrisi kayan-nokta yoğun programlar için yeni bir başarım ölçütü, *MFLOPS* ölçütünü getirmiştir.

2.3.5 MFLOPS ile Başarım Ölçümü

MFLOPS saniyede milyon kayan noktalı işlem anlamına gelen *mega floating point operation per second* teriminin kısaltmasıdır. Herzaman "**megaflops**" diye okunur.

$$MFLOPS = \frac{\text{Bir programdaki kayan noktalı işlem sayısı}}{\text{yürütme-süresi} \times 10^6}$$

Ancak *MFLOPS* programa bağımlıdır. Sadece programda kayan noktalı sayı işlemleri kullanılmışsa anlamlıdır.

Komutlar yerine aritmetik işlemlerin üzerine tanımlandığından, *MFLOPS* farklı makineleri karşılaştırmada daha iyi bir ölçüt olma eğilimindedir. Ancak, **farklı makinelerin kayan noktalı işlem takımları birbirine benzemez, ve gerçekte aynı iş için gereken kayan noktalı işlem sayısı her makinede farklı olabilir.**

Örnek: Motorola 68882 kayan noktalı bölme komutunu destekler, ama, Cray-2 -de kayan noktalı bölme işlemi yoktur.

Hızlı ve yavaş kayan noktalı sayı işlemlerinin karışımı da *MFLOPS* değerini değiştirebilir. Sadece kayan noktalı toplama yapan bir program, sadece kayan noktalı bölme yapan programa göre daha yüksek değer verebilir. Bu tür farklılıklar *normalize MFLOPS* kullanılarak düzeltilebilir.

2.3.6 Normalize *MFLOPS*

Normalize MFLOPS, yüksek seviye bir programlama dilindeki kayan noktalı işlemler için denk sayı bulma yöntemi tanımlar. Böylece bölme gibi daha karmaşık işlemlere gerektiğince fazla ağırlık biçeriz.

Bununla birlikte, sayma/ağırlıklandırma farkı nedeniyle, *normalize MFLOPS* aslında kullanacağımız kayan noktalı işlemlerin gerçek sayısından çok farklı olabilir.

2.3.7 Tepe *MFLOPS*

Herhangi bir program parçası için mümkün olan en yüksek MFLOPS değerine *tepe MFLOPS* denir. Başarım ölçmede *tepe-MIPS* gibi, *tepe-MFLOPS* da yanıltıcı bir ölçüttür.

2.4 Başarım Değerlendirme-Programlarının Seçimi

MIPS ve *MFLOPS* yanıltıcı başarım ölçütleridir. Bir bilgisayarın başarımını ölçmek için, "*benchmark*" (karşılaştırma-noktası) adı verilen bir grup karşılaştırma programı kullanarak değerlendiririz.

- Karşılaştırma-programları kullanıcının gerçek işyükünün vereceği başarımı tahmin edecek iş yükünü oluşturur.
- En iyi karşılaştırma-programları gerçek uygulamalardır, ancak bunu elde etmek zordur.

Seçilen karşılaştırma-programları gerçek çalışma ortamını yansıtmalıdır. Örneğin, tipik bazı mühendislik ya da bilimsel uygulama mühendis kullanıcıların işyükünü yansıtabilir. Yazılım geliştirenlerin işyükü ise, çoğunlukla derleyiciler, belge işleme sistemleri, vb. -den oluşur.

Bazı küçük programların çalışma süresinin çoğunu çok küçük bir kod parçasında geçirerek karşılaştırma program takımlarını yanılttığı tecrübeyle sabittir. Örneğin, SPEC karşılaştırma takımının ilk sürümündeki **matrix300** programı, çalışma zamanının %99 -unu tek bir komut satırında geçirir. Bundan yararlanan bazı şirketler matrix300 -ün tek satırındaki başarımı arttırarak karşılaştırmayı yanıltmak üzere özel derleyiler bile geliştirmiştir (Şekil 2-1).

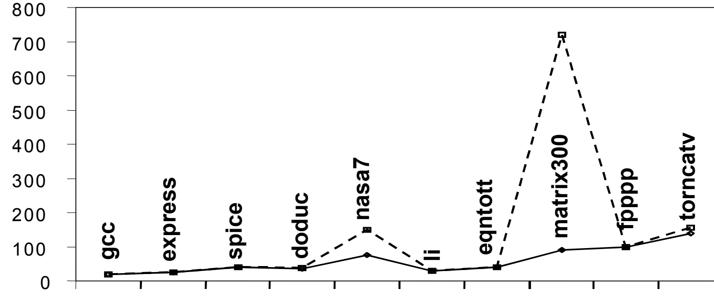
Küçük karşılaştırma programları elle bile hızla derlenebilir ve simüle edilebilir. Bunlar özellikle henüz derleyicisi yazılmamış yeni makinelerin tasarımcıları için kullanışlıdır. Ve de, bunları standartlaştırmak kolay olduğundan, küçük karşılaştırma programlı başarım sonuçları yayınlanmış olarak kolayca bulunabilir.

Benchmark sonuçları rapor edilirken, makinelerin karşılaştırma ölçümleriyle birlikte şu bilgiler de listelenmelidir.

- işletim sisteminin sürümü
- kullanılan derleyici
- programa uygulanan girdiler
- makine yapılanışı (bellek, I/O hızı, vs..)

Tablo 2-2 Daha yüksek başarım sonuçları elde edilen makinenin sistem betimlemesi,

Donanım	
Model no	Powerstation 550
CPU:	41.67 MHz POWER 4164
FPU	Tümleşik
CPU sayısı	1
Önbellek Boyutu	64k veri, 8k komut
Bellek	64Mb
Disk Altsistemi	2-400 SCSI
iletişim ağı arayüzü	Yok
Yazılım	
O/S tipi	AIX v3.1.5
Derleyici sürümü:	AIX XL C/6000 Ver 1.1.5 AIX XL Fortran Ver 2.2
Diğer yazılım	Yok
Dosya sistemi tipi	AIX
Bellenim seviyesi	Yok
Sistem	
Uyum parametreleri	Yok
Art alan yükü	Yok
Sistem durumu	Çok kullanıcı (tek-kullanıcı login)



Matrix300 ve nasa7 -deki en yüksek rakamlar bu iki çekirdek-temelli karşılaştırma programına has özel optimizasyon tekniği uygulan-masından doğuyor. SPEC reports fall-winter-1990 -dan alınmıştır.

Şekil 2-1 IBM PowerStation 550 da kullanılan iki farklı derleyici ile ölçülmüş başarım değerleri.

2.5 Toplam çalışma zamanının hesaplanması

Eğer işyükündeki programlar eşit sayıda çalışırlarsa, karşılaştırma takımındaki *n* programın *toplam yürütme süresi*, her programın çalışma süresinin *aritmetik ortalaması* ile hesaplanır.

$$\textit{Aritmetik Ortalama} = \frac{1}{n}\sum_{i=1}^{n} \textit{süre}_i$$

Burada işyükünde *n* program vardır ve $\textit{süre}_i$, *i* -nci programın *yürütme süresi*dir.

Eğer işyükündeki programlar farklı ağırlığa sahipse, her $\textit{süre}_i$ terimini w_i ağırlığı ile çarpıp *ağırlıklı aritmetik ortalama* hesaplayabiliz:

$$\textit{ağırlıklı aritmetik ortalama} = \frac{1}{\sum_{i=1}^{n} w_i} \sum_{i=1}^{n} w_i \, \textit{süre}_i$$

Normalize zamanın aritmetik ortalamasıyla hesaplanmış toplam çalışma zamanı, özellikle programlardan birinin çalışma zamanı diğerlerinden çok yüksekse, gerçek başarımdan sapar. Normalize edilmiş zamanlar kullanılması durumunda, başarım geometrik ortalama kullanılarak daha iyi karşılaştırılabilir.

$$\frac{\textit{Geometrik-ortalama-X}}{\textit{Geometrik-ortalama-Y}} = \textit{Geometric-ortalama}\left(\frac{X}{Y}\right).$$

ve

$$\textit{Geometrik ortalama} = \left(süre_1 \times süre_2 \times ... \ ...\times süre_n \right)^{(1/n)}$$

Geometrik ortalamanın aritmetik ortalamadan farkı boyutsuz (birimsiz) olmasıdır (artık zaman ölçüsü değildir), ve toplam yürütme süresiyle orantılı gitmez. Bu yüzden de programın yürütme süresini tahminde işe yaramaz.

Tablo 2-3 İki programın, iki farklı makinedeki yürütme süreleri.

karşılaştırma-takımı programları	yürütme süresi (saniye) Bilgisayar A	Bilgisayar B
Program 1	1	10
Program 2	1000	100
Program 3	1001	110

Tablo 2-4 Normalize edilmiş aritmetik ortalama yanıltıcı olabilir.

Karşılaştırma	Yürütme-süresi		A -ya Normalize		B -ye Normalize	
Programları	Ta	Tb	Ta/Ta	Tb/Ta	Ta/Tb	Tb/Tb
Program-1	1	10	1	10	0.1	1
Program-2	1000	100	1	0.1	10	1
Aritm. orta	500.5	55	1	5.05	5.05	1
Geom. orta	31.6	31.6	1	1	1	1

Veri A -ya normalize edildiğinde, B -nin başarımı A -nınkinin 5.05 katıdır, ama aynı veri B -ye normalize edildiğinde, A -nın başarımı B -ninkinin 5.05 katıdır. Geometrik ortalama iki durumda da tutarlıdır.

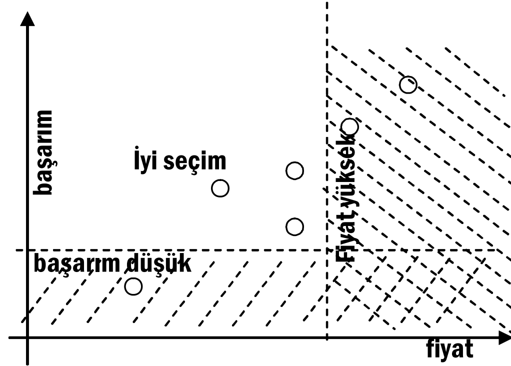
2.6 Sonuç

- Doğru başarım ölçüsü üç parametreyi: *komut sayısı*, *CPI*, ve *saat hızı* -nı şu şekilde içermelidir

$$\textit{Yürütme-süresi} = \frac{\textit{komut sayısı} \times CPI}{\textit{saat-hızı}}$$

- Bir tasarımın farklı yönlerinin bu anahtar parametrelerin her birini nasıl etkilediğini anlamamız gerekir: örneğin,
  - komut takımı tasarımı komut sayısını nasıl etkiler,
  - ardışıkdüzen ve bellek sistemleri *CPI* değerini nasıl etkiler,
  - saat hızı teknoloji ve organizasyona nasıl bağlıdır.
- Sadece başarıma bakmamız yetmez, maliyeti de düşünmemiz gerekir. Maliyet şunları kapsar:
  - parça maliyeti
  - makineyi yapacak işgücü
  - araştırma ve geliştirme giderleri
  - satış, pazarlama, kar, vs.

Örneğin:
Superbilgisayarlarda, başarım maliyetten daha önemlidir, halbuki çoğu PC için tersi doğrudur. Bunun yanında, iş istasyonlarında, herzaman fiyat ve başarım arasında denge vardır. Son karar herzaman test edilen makinenin fiyat-başarım grafiği (Şekil 2.6-1e bakınız) değerlendirilerek verilir.



Şekil 2-2 Son karar için tipik başarım-fiyat grafiği

2.7 Çözümlü Problemler

İki farklı sistemden (S1 ve S2) elde edilmiş aşağıda verilen ölçümleri ele alalım:

Program	S1 deki zaman	S2 deki zaman
1	10 saniye	5 saniye

Ayrıca şu ölçümler de yapılmıştır:

Program	S1 de çalıştırılan komutlar	S2 de çalıştırılan komutlar
1	$20x10^6$	$16x10^6$

Eğer iki sistemin de saat hızı 100 MHz ise, her sistemin MIPS ve CPI değerini belirleyin.

<<Çözüm:

$$MIPS=\frac{komut\ sayısı}{yürütme\ süresi \times 10^6}\ ;\quad CPI=\frac{saat\ hızı}{MIPS \times 10^6}$$

S1 sistemi 10 s.de 20×10^6 komut çalıştırıyor.

$$MIPS_{(S1)}=\frac{20\times10^6}{10\times10^6}=2 \qquad ve \qquad CPI_{(S1)}=\frac{100\times10^6}{20\times10^6}=5$$

Sistem S2 5 sn.de 16×10^6 komut çalıştırır.

$$MIPS_{(S2)}=\frac{16\times10^6}{5\times10^6}=3.2 \qquad ve \qquad CPI_{(S2)}=\frac{100\times10^6}{16\times10^6}=6.25$$

>>

Q2) SPEC03 (S), (I) and (P) bilgisayarlarını A ve B diye iki farklı derleyici için test ederek, dört temel testten çıkan sonuçları aşağıdaki çizelgede yayınlamıştır.

Bütün sayısal değerler testlerin saniye cinsinden yürütme süreleridir.

Özellik \ Bilgisayar	Bilgisayar-S		Bilgisayar-I		Bilgisayar-P	
CPU	SUM-squirk (RISC)		PA-RISC-8 (RISC)		Peltium-3G (CISC)	
Memory	500MB		500MB		500MB	
Cache	4MB		4MB		4MB	
OS	Pegarus		IBN-VM		Wishdown-2003	
Clock-Rate	500MHz		1GHz		3GHz	
programs: \ compilers	A	B	A	B	A	B
(veri-işlem programı) D03	300	300	400	300	600	500
(kayan noktalı program) Fx03	100	100	100	100	200	200
(tamsayı program) Int03	300	200	100	200	300	300
(sunucu uygulamaları) SX03	100	400	300	300	400	400

Satınalma kararı vereceğiniz toplam altı sistem var: derleyici-A ile bilgisayar-S (SA), B ile S (SB), A ile I (IA), B ile I (IB), A ile P (PA) ve derleyici-B ile bilgisayar-P (PB).

a) Hangi bilgisayarın başarımı ilk bakışta elenmesini gerektirecek derecede kötüdür? Neden?
<< *P, bütün testlerde yürütme süresi en yüksek çıktığı için.* >>

b) Gelecek üç yıl için tahmini iş yükünüzün bileşenlerinin 30% veri işleme, 20% kayan-noktalı işlem, 40% tamsayı aritmetik, ve 10% sunucu uygulaması olmasını bekliyorsunuz. Yardımcınız size son kararınız için aşağıdaki tabloyu hazırlamış.

Tahmini iş yükünde yürütme süresi hesabı için yardımcı çizelge

	Aritmetik Ortalama	$T_{D03} \times 0.3$	$T_{Fx03} \times 0.2$	$T_{Int03} \times 0.4$	$T_{SX03} \times 0.1$
SA	200	90	20	120	10
SB	250	90	20	80	40
IA	225	120	20	40	30
IB	225	90	20	80	30
PA	375	180	40	120	40
PB	350	150	40	120	40

bütün sistemlerin fiyatları aynı kabul edilse öngörülen işyükü için hangi sistem en iyi seçenektir

<< *Ağırlıklı yürütme sürelerinin ortalaması:*

$$Tsa = \frac{T_{D03\text{-}SA} \times 0.3 + T_{Fx03\text{-}SA} \times 0.2 + T_{Int03\text{-}SA} \times 0.4 + T_{SX03} \times 0.1}{0.3 + 0.2 + 0.4 + 0.1} = 240 ,$$

Diğerleri Tsb=230, Tia=210, Tib=220, Tpa=380, Tpb = 350 dir. Bu yürütme değerlerinden, en hızlısı IA bulunur, (yani, ***A derleyicili IBN*** *).* >>

c) Derleyici-A ve -B -nin fiyatları 1000 ve 500 dolar, S, I ve P bilgisayarlarının fiyatları ise 1000, 1500 ve 500 dolar, bütçeniz 2000 dolar olsun. Bu fiyatlar gözönüne alınırsa en başarımlı seçim hangisidir? Neden?

<< Sistemlerin fiyat ve ağırlıklı yürütme süreleri :

	SA	**SB**	**IA**	**IB**	**PA**	**PB**
Fıyat (dolar)	2000	1500	2500	2000	1500	1000
yürütme süresi (s)	240	230	210	220	380	350

IA en iyi başarıma sahip ama bütçeyi aşıyor. bir sonraki başarım IB -ninki, ve bütçeyi geçmiyor. ***IBN ile B derleyicisi seçilir****.>>*

Q3) IBN -in şef tasarım mühendisi olarak eldeki (A) CPU -sunu değiştirip iki yeni CPU (B ve C) tasarladınız. CPU -ların R-tip ve I-tip olarak iki farklı tip komutu var. Her CPU -nun saathızı ile her komut tipinin CPI değeri aşağıdaki çizelgede veriliyor.

CPU	Nominal Saat-hızı	I-tipinin CPI -ı	R-tipinin CPI -ı
A	30 MHz	7	10
B	40 MHz	10	10
C	45 MHz	16	10

Test programınında 300×10^6 I-tipi ve 600×10^6 R-tipi komutun yürütülüyor.

a) Herbir CPU -un ortalama CPI -ı kaçtır?

<< "$\times 10^6$ " yı "M" ile gösterelim.

Toplam dönüş sayısı A için = 7×300 M + $10 \times$ 600 M = 8100 M,

B için = 10×300 M + $10 \times$ 600M =9000 M

C için = 16×300 M + $10 \times$ 600M =10800 M

Toplam komut= 300 M + 600 M = 900 M,

CPI_A = 8100/900 ; CPI_B = 9000 / 900 CPI_C = 10800/9

= **9**; = **10**; =**12**.

>>

b) Her bir CPU ile bu test programının yürütme süresi ne kadardır?

<< *T= CPI $\times$ komut sayısı / saat hızı;*

T_A = $9\times$ 900M / 30M T_B = $10\times 900/40$ T_C = $12\times 900/5$

= **270 saniye** **=225s** **=240s**

>>

c) Bu test programı ele alındığında en iyisi hangi CPU -dur?

<< CPU-B en iyidir. Çünkü yürütme süresi en kısadır. >>

3 Komut Takımı

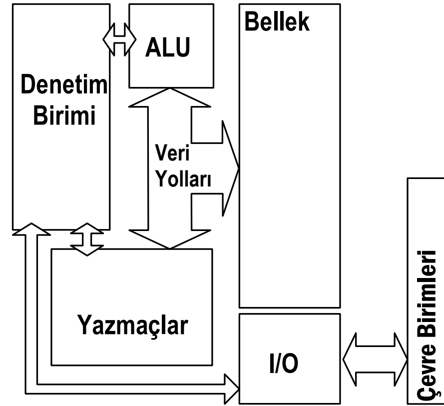
3

Amaç
Bu bölümün hedefi RISC mimarisinin temel bileşenlerine sahip 32-bit RISC işlemcisi olan MIPS-R2000 -in komutlarının temel tasarım ilkeleriyle komut takımındaki gerekli komut tiplerini, ikilik alanlarını tanıtmaktır. Bu bölümde MIPS mimarisinin komut takımını, ve buna bağlı olarak bu komutları gerçekleştirecek donanımı tanıtacağız.

3.1 Giriş

Ana bilgisayarlardan mikrobilgisayarlara çağımızın sayısal bilgisayarları hemen hemen benzer ortak temel öbek yapısına ve ortak temel ilkelere dayanır. Temel yapı bilgisayar tasarımındaki tipik kısıtlamalarla belirlenir, ve beş temel bileşenden oluşur: kontrol birimi, işlemsel mantık birimi, dahili yazmaç takımı, veri yolları, ve bellek ile i/o arayüzleri.

Bilgisayar donanımlarındaki benzerlik komut takımlarına de yansır. Bilgisayarın komut takımı programın atomik birimlerini oluşturur, ve komutları çalıştıran donanımla doğrudan ilişkilidir. Uygulamada tasarım etmenleri ve kısıtlamaları bilgisayar tasarımcısını komutlar için benzer biçimsel yapılar seçmeye zorlar. *"hızlı"* makine tasarımında donatımın *"basitliği"* değerli bir kavramdır.



Şekil 3-1 Tipik bilgisayarın öbek şeması.

- Neredeyse tüm bilgisayarlar benzer donanım teknolojisi kullanılarak yapılır. Bu nedenle, makine dilleri de oldukça benzerdir. Bunlardan birini öğrendiğinizde diğerlerini anlamak kolaylaşır.
- MIPS komut takımının bazı önemli özelliklerini görüp donanım üzerinde nasıl gerçekleştirildiğini ve yüksek seviye programlama

dilleriyle (bizim durumumuzda C kullanarak) makine kodları arasındaki ilişkiyi görmeliyiz.

- Komut takımını küçültmenin temel amacı daha basit, daha küçük ve daha hızlı donanıma sahip olmaktır.

Bilgisayar donanımlarının gelişimi boyunca RISC işlemcilerin donanım tasarımında bazı tasarım ilkeleri yerleşmiştir. Aşağıdaki tasarım ilkeleri gelecek bölümlerde uygulama örnekleriyle tanıtılacaktır.

- Ilk ilke "*basitlik düzenliliği sever*" (*Simplicity favors regularity*). Donanımın basitleşmesini sağlamak üzere komutların tipini ve biçimini basitleştirip sınırlandırırken kullanırız.
- İkinci ilke "*Daha küçük daha hızlıdır* (*Smaller is faster*)" ile özetlenir. Yazmaç sayısı, veriyolu ve komut genişliği, ALU işlemlerinin sayısı vb. gibi donanım birimlerini doğru boyutta seçerken önemlidir.
- Tasarımın üçüncü ilkesi *"iyi tasarım uzlaşma gerektirir" (Good design demands compromise)*. Bu ilkenin uygulamasını anlık veri ve adres boyu seçiminde göreceğiz.
- Dördüncü tasarım ilkesi ***"****yaygın durumu hızlandır" (Make the common case fast)* der. Uygulamasını anlık veri biçimleri ile dallan ve atla komutlarının yapısını saptamada kullanacağız.

3.2 Temel MIPS çekirdeği

MIPS 80'li yılların başında geliştirilmiş bir RISC işlemcidir. O devrin tipik teknolojisi 32-bit veri yolu ve 32-bit adres yoluydu, ve tasarım istemleri ise basit C dili program cümlelerini çok basit gerçekleştirmek, diziler için indeksli adresleme, göreli dallanma, uzak atlama, bir dizi iç yazmaçta hızlı aritmetik-mantıktı. Bu tipik istemler altında, "daha basit ve daha hızlı" işlemci için çözüm istisnasız *32-bit ALU*, *32-bit veri/adres yolları*, ve *32-bit komut takımı* kullanmaya yakınsar.

3.2.1 MIPS -in Veri Çeşitleri

MIPS temel olarak 32-bit *işaretli* ve *işaretsiz* tamsayı biçimini destekler. Her tamsayı 32-bit, yani bir sözcük yer tutar. MIPS bir sözcüğü genel amaçlı yazmaçlarından herhangi birinde tutabilir. Genel amaçlı yazmaçların sayısı 32 ile sınırlıdır, ve bunlardan bazıları, yığıt göstergesi gibi, özel amaçlar için ayrılmıştır. Genelde bütün değişkenler bellekte tutulur. Bir değişken, yazmaça sadece veri

işlemek üzere alınır. Bellekte herhangi bir yere okumak ya da yazmak üzere 32-bitlik bir adres ile erişilir.

Veri çoğu uygulamada *dizi* biçimindedir. Dizideki bir eleman iki bileşenle adreslenir: dizinin bellekteki başlangıcını belirten *başlangıç-adresi*, ve gösterilen elemanın yerini belirten *indeks* veya *yerdeğiştirme (displacement),* yada *kayma (offset)*.

MIPS -te, iki bayt bir "*yarım sözcük" (half-word)*; dört bayt bir "sözcük" *(word)*; ve iki sözcük bir "*çift-sözcük" (double-word)* oluşturur. MIPSin 32-bit tamsayı veri tipleri üzerinde geniş bir aritmetik mantık işlem takımı vardır, ama 8-bit tabanlı karakter ve dizgi veri tipleri için desteği sınırlıdır. Bu özellik MIPS -i, uygulamada özellikle tamsayı yoğun işlerde en iyi gömülü sistem işlemcilerinden biri yapar.

MIPS ayrıca kayan noktalı yan işlemcisi sayesinde IEEE754 biçiminde 32 ve 64-bit kayan noktalı sayıları da destekler. Kayan noktalı sayı aritmetiğini bir sonraki bölümde göreceğiz.

3.2.2 Aritmetik-Mantık komutları

"**a = b + c**", iki değer (yada değişken) ve bir sol-yan değişkeni içeren C dilinde tipik basit aritmetik işlemdir.

add a,b,c

bu işleme karşılık olan MIPS çevirme dili kodlamasıdır. Buradaki **a, b** ve **c** işlemcinin iç yazmaçlarını temsil eden değişken adlarıdır. Bu örnekte *anımsatıcı* **add** -i üç *işlenen* **a, b, c** izliyor. Genelde, daha uzun herhangi bir aritmetik ifade basit aritmetik işlemler zincirine dönüştürülebilir. MIPS aritmetik komutlarının çoğunun 3 işleneni vardır. İlk işlenen, sol-yan değişkenine karşılık olan *varış yazmacı* içindir. Bu komut biçimi, bütün işlemi mümkün olan en basit donanımla tek adımda yapabilmemize izin verir.

Örnek 3-1*:* **b, c, d** ve **e** değişkenlerini toplayıp sonucu **a** değişkenine koymak için çevirici kodunu yazın.

<< Çözüm: Aşağıdaki MIPS aritmetik komut dizisi **a=b+c+d+e** işlemini gerçekleştirir:

```
add a,b,c    # b + c  yi a ya yaz
add a,a,d    # b+c  +  d  yi  a ya yaz
add a,a,e    # şimdi de b+c + d  +  e yi a ya yaz
```

>>

Örnek 3-2: C-program-kodlarına karşılık gelen çevirici kodlarını yazın

soru: C-parçası	çözüm: MIPS çevirici kodu karşılığı	Açıklama
a = b + c	**add a,b,c**	Basit yazmaç aritmetiği
d = a – e	**sub d,a,e**	Basit yazmaç aritmetiği
f = (g+h)–(i+j)	**add t0,g,h # g+h -ı geçici değişken t0 -a koy** **add t1,i,j # i+j -yi geçici değişken t1 -e koy** **sub f,t0,t1 # f, (g+h) – (i+j) olur**	t0 ve t1 geçici değişkenler;

3.2.3 Yazmaç Dosyası

MIPS -te, adreslenen yazmaçların içeriği aritmetik komutların işlenenleri olur. Bir aritmetik komut (örneğin "**add a,b,c**") bellekte tam olarak 32 bit yer tutar. Bu 32 bitte şu üç çeşit bilgi yer alır:

a- komut tipini betimleyen işlem kodu (örneğin, bir aritmetik mantık komutu olduğunu),

b- işlemde uygulanacak işlevi betimleyen işlev kodu (örneğin toplama, çıkarma, mantıksal "ve" vb.);

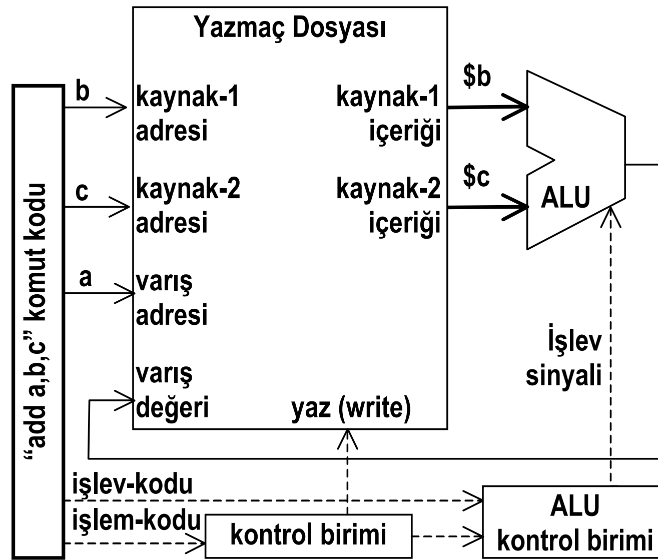
c- komutun işlenen veri yazmaçlarını belirlemek amacıyla üç yazmac adresi.

Basit bir varsayımla, işlem ve işlev için yaklaşık 16 bit gerekse, üç yazmacı adreslemek için geriye 16 bit kalır. Demektir ki, komut kodunda her yazmacın adres alanı için sadece 5-bit kaldı. Bu 5-bitlik adres alanları 32 yazmaçtan birine karşılık gelir. Dikkat ederseniz kodumuz 32-bitten büyük olsa, bellek erişimi ve işleme süresi 32-bite göre daha uzun olacaktır. Yani, ikinci tasarım ilkesi "*daha küçük daha hızlıdır*" yazmaç sayısını 32 olmaya zorlar. Böylece 5-bitlik bir sayı ancak 32 yazmacı adresleyebileceğinden, tam 32 yazmacımız olacaktır. Bu yazmaçlar yazmaç-0, … , yazmaç-31 diye anılır ve r0, r1, r2, ... , r30, r31 ile gösterilir. Dolar işareti "$" adres edilen yazmacın içeriğini belirtir. Örneğin, **add $12,$15,$16** komutu r15 -in ve r16 -nın içeriklerini toplar, ve sonucu r12 -deki değerin üzerine yazar (yeni değer yazılınca eskisi kaybolur).

Özetle:

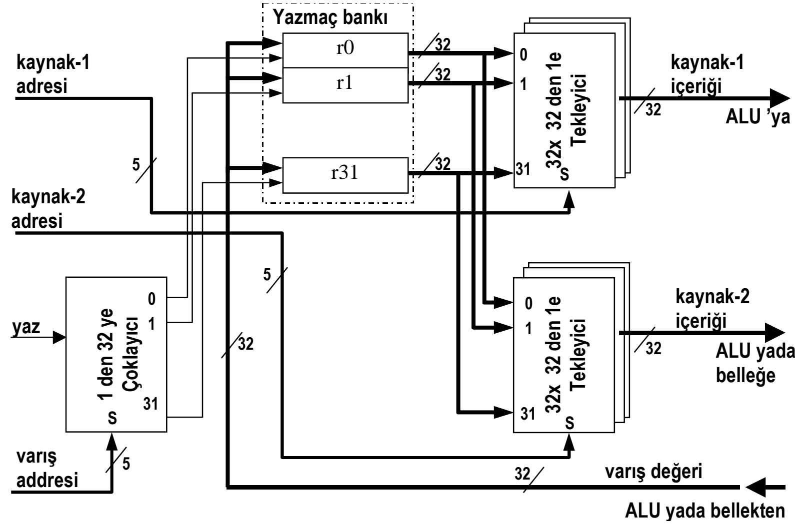
- Aritmetik komutların işlenenleri yazmaçlardır.
- MIPS genel amaçlı yazmaçlarının genişliği 32 bittir.
- MIPS -in 32 genel amaçlı yazmacı vardır: r0, r1, r2, ... , r30, r31.
- Bir yazmacın içeriği önünde dolar-işareti ile gösterilir: $0, .. $31.
- Yazmaç-0 -ın içeriği sabittir ve her zaman sıfırdır ($0 = 0).

Bir *çoklu erişimli yazmaç dosyası* tasarımının veriyolları sıradan bellek öbeğine göre daha karmaşıktir, ve yazmaç sayısı arttırılırsa adres çözümlemesi daha uzun süreceğinden saat dönüş süresini uzatır (donanım tasarımı ilkesi 2: *daha küçük daha hızlıdır*).



Şekil 3-2 Bir aritmetik-mantık komutun yürütülüşü

Yazmaç okuma ve yazma işlemi bellek okuma ve yazma işleminden 10 kat hızlıdır, bu yüzden, yazmaç kulanmak işlemcinin başarım değerinde farkedilir derecede yükseltir.



Şekil 3-3Yazmaç-dosyasının dahili veriyolu.

Örnek 3-3 : Aşağıdaki eşitliği kodlayalım

f = (g+h) – (i+j)

Burada **f**, **g**, **h**, **i**, ve **j** değişkenleri sırasıyla **$16**, **$17**, **$18**, **$19** ve **$20** yazmaçlarından kullanılıyor olsun (bunu **f** ≡ **$16**, **g** ≡ **$17** ... ile de gösteririz ve "**f** yazmaç-16 -da saklanır, yazmaç-17 -de **g** var, vs. diye okuruz.):

<< çözüm:

```
add   $8,$17,$18    # r8 (yazmaç-8 ) g+h -yi içeriyor
add   $9,$19,$20    # r9  i+j -yi içeriyor
sub   $16,$8,$9     # sonuç f -e yani reg-16 -ya yazılıyor
```

>>

3.2.4 Bellek Değişkenleri ve Diziler

Ilk tasarım ilkemiz "*Basitlik düzenliliği getirir*" MIPS komut takımında sadece yazmaçtan yazmaca aritmetik mantık komutlarına izin veriyor. İkinci tasarım ilkesi "*Daha küçük daha hızlıdır*", yazmaç sayısını 32 -yle sınırlıyor. 32 yazmaçlık bir yazmaç dosyası, çoğu programın tüm değişkenlerini yazmaçlarda tutmaya yetmez. Demek ki, değişkenleri ve diğer veri yapılarını bellekte tutmalıyız. Bir değişkenin işlenmesi gerektiğinde, onun değerini bellekteki yerinden alıp yazmaçlardan birine aktarmak, ve veri işlendikten sonra da tekrar belleğe aktarmamız gerekir. Bunu sağlayacak yeni bir komut tipine ihtiyacımız var. Bir değeri bellek ile yazmaç arasında aktaran bu tür komutlar ***"veri aktarma"*** ya da "*bellek erişim*" komutları diye bilinir.

Bir sözcük bellekten yazmaca *load-word* **(***sözcük-yükle***)** komutu (anımsatıcısı **lw**) ile aktarılır. Bir sözcüğü yazmaçtan belleğe taşımak için ise *store-word* **(***sözcük-sakla***)** (anımsatıcısı **sw**) kullanılır.

Tablo 3-1 MIPS sisteminde bellek adresleme ve içerikleri

Bellek (küçük-uçtan)			
Word Adresi	Bayt Adresi	Bayt İçeriği	Word İçeriği
0	0	35	küçük-uçtan (0,0,0,35) =35
	1	0	
	2	0	
	3	0	
1	4	56	(0,0,0,56) =56
	5	0	
	6	0	
	7	0	
2	8	3	(0,0,0,3) =3
	9	0	
	10	0	
	11	0	
3	12	7	(0,0,0,7) =7
	13	0	
	14	0	
	15	0	
…	…	…	….

Sözcükteki baytlar ya en sağ basamaktan (küçük-uçtan) ya da en sol basamaktan (büyük-uçtan) başlayabilir. Bayt sıralamasından (büyük-yada küçük-uçtan) bağımsız olarak, yarım sözcük, sözcük, veya çift-sözcük adresi veriyi oluşturan baytların en küçük adresli olanıdır. Büyük-uçtan sıralama için bu en sol bayt, küçük-uçtan için en sağ bayt olur. MIPS her iki sıralamayı da destekler.

MIPS bellek sisteminde belleğin bir yeri genelde her sözcüğün ilk baytının adresi kullanılarak betimlenir. Tablo 3-1 deki bellek içeriklerinin alışılmış gösterimi Tablo 3-2'de gösterilmektedir.

Tablo 3-2 Bellek içeriklerinin alışılmış gösterimi

Bellek	
Bayt Adresi	**İçeriği**
0	**35**
4	**56**
8	**3**
12	**7**
...	**...**

8-bit veri tipleri (yani, işaretli yada işaretsiz bayt, veya char) birçok programda sıklıkla kullanılır. MIPS sistemi bellekte ayrıca bayt adreslemek te mümkündür. Ama temel tamsayı veri tipi ve MIPS -in

temel komut biçimi 32 bittir. MIPS dört ardışık baytı 32-bit sözcük olarak kullanır. Bellekteki ilk sözcüğün adresi 0 -dır, sıradaki sözcük adres 4 -ten başlar.

sözcük yükle (load-word) ve *sözcük-sakla* (*store-word)* komutlarının biçimleri:

Sözcük yükle	Sözcük-sakla
lw $varış, ofset($baz) bellekteki **ofset+$baz** adresinden başlayarak dört ardışık baytı *$varış* yazmacına yükler.	**sw $kaynak,ofset($baz)** **$kaynak** yazmacındaki değeri bellekte **ofset+$baz** adresinden başlayarak dört ardışık bayta saklar.

burada **ofset** komut kodunda verilen 16-bitlik anlık bayt-yerdeğiştirme-adresidir, **$varış, $kaynak** ve **$baz** İngilizcede destination, source ve base yazmaçları olarak anılır.

Örnek 3-4: **A** adında 100 tamsayı elemanlı bir dizi olsun. **A**[.] "**Astart**"dan başlıyor varsayalım. Aşağıdaki C-dili ifadede

g = h + A[i];

g ve **h** değişkenleri **$17** ve **$18** yazmaçlarda saklansın. Yazmaç-19 -da **4×i** olduğunu varsayalım. Verilen C koduna karşılık gelen MIPS çeviri kodunu yazın

```
<<   lw    $8, Astart($19)    # A[i] geçici yazmaç $8 e gider
     add   $8, $18, $8        # h+A[i] geçici yazmaç $8 e gider
```

Astart($19) adreslenirken, yazmaç-19 “*baz yazmacı*” olarak anılır. Bellekte bu dizinin tutulduğu yerlerinden bazılarının adresleri ve içeriği şöyledir:

Bellek (büyük-uçtan)			
Sözcük Adresi	Bayt Adresi	Bayt İçeriği	Sözcük İçeriği (işaretsiz tamsayı)
Astart: A[0]	**Astart**	...	...
	Astart+1	...	
	Astart+2	...	
	Astart+3	...	
A[1]	**Astart+4**	...	...
	Astart+5	...	
	Astart+6	...	
	Astart+7	...	
...	**Astart+...**	...	...
	Astart+...	...	
	Astart+...	...	
	Astart+...	...	
A[i]	**Astart+4×i**	**B_3**	büyük-uçtan
	Astart+4×i+1	**B_2**	$2^{24} B_3+2^{16} B_2+2^8 B_1+B_0$
	Astart+4×i+2	**B_1**	
	Astart+4×i+3	**B_0**	

>>

Örnek 3-5

Aşağıdaki ifadeyi C-dilinden MIPS makine koduna çeviriniz.

g = h + A[i];

Burada A[] dizisi Astart -tan başlıyor, ve **g, h, i** için sırasıyla **$17**; **$18**, **$19** kullanılıyor olsun. 4 ile çarpma için R-tipi *mantıksal-sola-kaydır* **sll $varış,$kaynak,2** komutunu kullanın.

<< eğer **$19** da i×**4** bulunsaydı, **A[i]** , **Astart($19)** ile adreslenebilirdi. Önce **4**×**i** -yi örneğin **$8** gibi geçici bir yazmaca koymalıyız.

```
sll   $8,$19,2          # geçici yazmaç $8 = $19 ×4
lw    $8, Astart($8)    # A[i] yi $8 e yükledik
add   $17, $18, $8      # g = h + A[i]
```

>>

Örnek 3-6: Aşağıdaki C-dili ifadesinde, **A**[] dizisi ve yazmaç atamaları şöyle verilsin:

A[i] = h + A[i];

h değişkeni **$18** de saklansın, **A[]** tamsayı dizisi **Astart** adresinden başlasın, ve **$19** da **i**×**4** bulunuyor olsun.

Bu ifade için MIPS assembler kodunu yazın

<<

```
lw    $8, Astart($19)   # geçici yazmaç $8, A[i] -yi alır
add   $8, $18, $8       # geçici yazmaç $8, h + A[i] -yi alır
sw    $8, Astart($19)   # h + A[i] -yi A[i] -ye geri yükle
```

>>

Bellekteki adreslere erişim yazmaçlara göre çok daha yavaştır, ve bellekte birden çok yere aynı anda erişilemez. Daha yüksek başarım için MIPS, veri ve komut erişimini hızlandırmak üzere önbellek kullanır. MIPS derleyiciler çevirici kodunu sık kullanılan değişkenleri yazmaç dosyasında, diğer değişkenleri bellekte tutacak biçimde optimize eder. Yazmaçları verimli kullanarak daha yüksek başarımlara ulaşılır.

3.2.5 Anlık tamsayı değerler

Değişik türden işlemlerin tipik sıklık oranı istatistiklerine göre aritmetik-mantık işlemlerindeki işlenenlerin %50 -den fazlası 16-bitten daha küçük işaretli anlık tamsayılardır. Örneğin yaygın kullanılan C derleyicisi “gcc”nin kodundaki aritmetik komutlarının %52 -si işaretli küçük tamsayılardır. Analog devreleri analiz eden uygulama programı

"SPICE"da bu oran neredeyse %70 -tir. Yazmaç aritmetiği ve sözcük-yükle sözcük-sakla yaklaşımımızda, derleyici bu tamsayıların tümünü programla birlikte bellekte tutmak zorundadır.

Örnek 3-7: Sabit 4 değerinin bellekte AddrC4 adresine yüklendiğini varsayalım. Aşağıdaki komut yazmaç-29 -a 4 ekler

```
lw    $24, AddrC4($0)      # $24 ← sabit 4
add   $29, $29, $24        # $29 ← $29 + 4
```

Bellek erişimi yavaş olduğundan, değişmezleri bellekte önceden belirlenmiş yerden yazmaca yüklemek başarımı düşürür. Sabit sayı kullanmanın diğer bir yolu sayıyı komut içinde tutmaktır. Ama, anlık 32-bit değerleri 32-bitlik bir komut içinde barındırmak olanaksızdır. Burada üçüncü tasarım ilkemiz yardıma koşar ve 32-bit yerine daha kısa ama kullanışlı bir genişlikteki anlık değerlere razı oluruz. Anlık aritmetik-mantık komutlarına duyulan yoğun gereksinim MIPS -te 16-bitlik anlık değerler kullanılarak karşılanır. Anlık veri içeren komutlar, anımsatıcılarına eklenen "i" harfiyle belirtilir:

Örnek 3-8: **a**, **b** ve **c** değişkenleri için sırasıyla **$15**, **$16** ve **$17** yazmaçlarını kullanılıyor olsun.

c = (a+43) – (12+b)

için çevirici kodunu yazınız.

```
<<
addi  $8,$15,43     # geçici yazmaç-8, a+43 -ü tutar
addi  $9,$16,12     # geçici yazmaç-9,  b+12 -yi tutar
sub   $17,$8,$9     # sonuç yazmaç-17 -ye (c -ye) gider
>>
```

Bir anlık komutun çalışması, donanımda ekstra veri yolu gerektirir. Bu yüzden işlemsel-mantık komutlarının bir bölümünün anlık biçimi yoktur.

Sözde-Komut

MIPS -in ilginç bir özelliği yazmaç-0 içinde herzaman sabit sıfır değerinin olmasıdır. MIPS çeviricisi yazmaç-0 -ı kullanarak yazmaç-tan-yazmaca "*yolla*" (=**move**) anlık değerden yazmaca "*anlık-yükle*" (=**li** : *load immediate*) veri aktarmayı **add** ve **addi** komutları ile gerçekleştirir. Özgün komut takımında bulunmayan ve çevirici tarafından mevcut komutların bir araya gelmesiyle gerçekleştirilen bu komutlar sözde-komut olarak anılır.

Anlık-değer-yükle (*load-immediate:* **li**) sözde-komutu 16-bitlik bir sabit değerin bir yazmaca aktarılmasını sağlar,

li $dest,imm

Çevirici bu sözde-komutu şu satıra dönüştür.

addi $dest,$0,imm

En soldaki 16-biti de kullanılan sabit değerlerin sol 16-bitlik bölümünü yazmaca yükleyebilmek için özel bir komut tasarlamak gerekir. *üste-anlıkdeğer-yükle* (*load-upper-immediate*) komutunun anımsatıcısı **lui** -dir. Bu komut aynı zamanda yazmacın sağ 16-bitini de temizleyerek anlık 32-bit sabitlerin **lui** ve **addi** kullanılarak aktarılmasını sağlar.

Örnek 3-9:

move $8, $18

MIPS -in temel komut takımında bulunmaz. Sözde-komut olarak çeviricide (assembler) tanımlanmıştır, ve şu temel MIPS komutuna çevrilir:

add $8, $0, $18

Örnek 3-10*:* 0x76543210 değerini 24-üncü yazmaça yükleyecek MIPS kodunu yazınız.

<<

```
lui     $24, 0x7654          # $24 -e 7654 0000 h yükler
addi    $24,$24, 0x3210      # işlem sonrası $24= 76543210 h olur
```

>>

3.3 Komutların Bilgisayarda Gösterimi

Şimdiye dek iki çeşit aritmetik mantık komutu ile sözcük yükle ve sözcük sakla komutlarıyla tanıştık. Bu komutların ikili kodları tam 32-bit tutar.

Ilk olarak yazmaçtan-yazmaca aritmetik mantık komutlarını ele alalım. MIPS -te, bu komutlar *R-tipi* komutlar diye anılır, ve komut kodu 6 ikili alandan oluşur.

MIPS R-tipi komut biçimi

opc	rs	rt	rd	sa	fn
(6 bit)	(5 bit)	(5 bit)	(5 bit)	(5 bit)	(6 bit)

burada

opc : komut işlem kodu
rs : birinci kaynak işleneni yazmacı
rt : ikinci kaynak işleneni yazmacı
rd : varış işleneni yazmacı (sonucu kaydeder)
sa : kaydırma miktarı (bazı komutların üçüncü işlenenidir)
fn : işlev kodu (işlemde kullanılacak işlevi seçer.)

Örnek 3-11: **add $8, $17, $18**

komut bellekte aşağıdaki onlu sayılar dizisi şeklinde saklanır:

0	17	18	8	0	32

ya da ikili biçimde (bitler halinde) gösterimi şöyledir:

000000	10001	10010	01000	00000	100000

Örnek 3-12: **sub $8, $17, $18**

onlu sayılarla şöyle gösterilir:

0	17	18	8	0	34

Şimdi, **lw** komutunu ele alalım. Komutta iki yazmaç (indeks- ve varış yazmaçları) ve bir anlık adres belirtilir. Eğer adres için R-tipi komut biçimindeki 5-bitlik alanlardan sadece birini kullansaydık adres bellekte $2^5 = 32$ yerle sınırlı kalırdı, ki bu uygulamaların çoğu için çok yetersizdir.

Komutların tümünü aynı uzunlukta tutmak istediğimizden 16-bit genişlikte anlık-alan elde etmek için alanların bazılarını birleştirmemiz gerekir. Böylece ortaya ***I-tipi*** biçim denilen yeni bir komut biçimi çıkar. Bu biçim 4 alandan oluşur, ve bellek veri aktarımı, anlık aritmetik ve dallanma komutlarında kullanılır.

MIPS I-tipi komut biçimi

opc	rs	rt	imm
(6 bits)	(5 bits)	(5 bits)	(16 bit)

Burada imm 16-bitlik anlık değer yada adrestir; rt ise, **lw** komutunun varış yazmacı ya da sw -nin kaynak yazmacını belirtir.

Bellekte adreslenecek yerin *bayt-adresi,* rt -deki değer ile 16-bitlik anlık değer toplanarak bulunur.

Örnek 3-13: **lw $8, Astart($19)**

35	19	8	Astart

burada, r8 varış yazmacıdır.

Örnek 3-14: **sw $8, Astart($19)**

43	19	8	Astart

burada, r8 kaynak yazmacıdır.

Örnek 3-15: Anlık topla (*Add immediate*) komutu (addi) yazmaca bir sabit değer ekler. $29 -un mevcut içeriğine 4 ekleyecek makine kodunu yazın:

<< **addi $29, $29,4**

8	29	29	4

ikili olarak

001000	11101	11101	0000 0000 0000 0100

>>

Örnek 3-16:

lui $8, 255 # $8 -in en sol 16 bitine 255 yazar

onluk makine kodu :

15	0	8	255

ikili sayılarla:

001111	00000	01000	0000 0000 1111 1111

Komutun çalışmasından sonra r8 deki değer::

0000 0000 1111 1111	0000 0000 0000 0000

Komut biçimlerinin farklı olması donanımı karmaşıklaştırır. Donanımın karmaşıklığını azaltmak için komut biçimlerini birbirine benzer tutmaya çalışırız (*ilke 3* : *İyi tasarım uzlaşı gerektirir*). Bu yönde, R- ve I-tipi biçimlerde:

- Ilk üç alan için aynı uzunluk ve adlar kullanılır,
- I-tipinin dördüncü alanı, R-tipinin son üç alanının birleşmesiyle oluşur.

Denetim biriminin donanım devresi komut biçimini komutun işlem kodu alanından ayırt ederek işler.

Örnek 3-17 : Aşağıdaki MIPS çevirici koduna dönüştürülmüş şu örneğii

A[i] = h + A[i];

ele alalım

```
lw   $8, Astart($19)
add  $8, $18, $8
sw   $8, Astart($19)
```

Diyelim ki **Astart** = 1200_{10} (= $0000\ 0100\ 1011\ 0000_2$) olsun.
MIPS çevirici kodunu onlu ve ikili sayılara kodlayın.

<< çözüm:

Onlu olarak:

				imm yada		
Komut	**opc**	**rs**	**rt**	**rd**	**sa**	**fn**
lw	35	19	8	1200		
add	0	18	8	8	0	32
sw	43	19	8	1200		

ikili olarak:

				imm yada

Komut	opc	rs	rt	rd	sa	fn
lw	100011	10011	01000	0000 0100 1011 0000		
add	000000	10010	01000	01000	00000	100000
sw	101011	10011	01000	0000 0100 1011 0000		

küçük-uçtan yazılışta bu makine kodu için bellekteki bayt dizilişi 0xB0, 0x04, 0x68, 0x8E, 0x20, 0x40, 0x48, 0x02, 0xB0, 0x04, 0x68, 0xAE olacaktır.
>>

R-tipi ve I-tipi biçimlerine ek olarak, olabildiğince geniş anlık adres gerektiren atla komutu için MIPS -te *J-tipi* diye üçüncü bir komut biçimi bulunur. Tahmin edebildiğiniz gibi, aşağıdaki alanlar tablosunda gösterilen J-tipi biçimde opc alanı dışındaki diğer bütün alanlar tek anlık adres alanına birleşir.

Tablo 3-3 MIPS -in temel komut tipi biçimleri

R-tipi:	**opc** 6-bit	**rs** 5-bit	**rt** 5-bit	**rd** 5-bit	**sa** 5-bit	**fn** 6-bit
I-tipi:	**opc** 6-bit	**rs** 5-bit	**rt** 5-bit	**imm** anlık 16-bit değer yada adr.		
J-tipi:	**opc** 6-bit	**immJ** anlık 26-bit-adres				

3.3.1 Programların bellekte durumasıSayısal bilgisayarlarda programlar bellekte saklanır ve sayı gibi okunup yazılabilir. Programları ve verileri bilgisayarın belleğine bir kere yükleyince bilgisayarın bellekteki komutları belli bir yerden yürütmeye başlalatırız. Bu yerin adresi çoğunlukla program sayacı (*Program Counter* kısaca *PC*) denen özel bir yazmaçta durur.

Von Neumann mimarisinde sadece bir bellek adres uzayı bulunur. Aynı bellek hem verileri hem yürütülecek programı barındırır. Von Neumann mimarisi genel amaçlı uygulamalarda daha yüksek esneklik sağdığından RISC ya da CISC, ana bilgisayar sistemlerin çoğu Von Neumann mimarisinde tasarlanır.

Programı bellekte tutma kavramı ayrık program ve veri belleği birimleri kullanılan işlemci sistemlerinde de geçerlidir. Ayrık program ve veri bellek birimleri olan bilgisayar mimarisi *Harvard* mimarisi diye bilinir. Bu mimari daha hızlıdır, ve özellikle gömülü sistem uygulamaları embedded için uygundur. Ayrıca hızlı bilgisayaralar cache belleklerinde Harvard mimarisini uygular.

3.4 Karar Verme Komutları

Dallanma (branch) ve atla (jump) komutları program akış denetimi için gerekli üyelerdir. Koşulsuz atlamada herhangi bir sınama koşulunun

verilmesine gerek yoktur, ve uzak dallanmalarda çok kullanışlıdır. Diğer taraftan, koşullu dallanma ise "*if*", "*do-while*", "*do-until*", "*conditional goto*" (koşullu gidiş) ve "*case*" gibi program akış yapıları için gerekli elemanlardır.

3.4.1 Koşullu Dallanma

MIPS komut takımında tüm dallanma yapıları yalnızca iki koşullu-dallan komutu kullanılarak gerçekleştirilir. Dallan komutunda, bir dallanma adresi ile iki yazmaç arasında bir deneme koşulu betimlenir.

eşit-ise-dallan	**eşit-değilse-dallan**
beq $reg1, $reg2, Hedef	**bne $reg1, $reg2, Hedef**
reg1 -deki değer **reg2** -deki değere eşit ise, **Hedef** etiketli ifadeye git	**reg1** -deki değer **reg2** -dekinden farklı ise, **Hedef** etiketli ifadeye git

Örnek 3-18: Aşağıdaki C program parçasının

```
        if (i = = j) goto L1;
        f = g + h;
L1:     f = f – i;
```

f, **g**, **h**, **i**, **j** değişkenlerinin **$16**, **$17**, **$18**, **$19**, **$20** de durduğunu varsayarak karşılık gelen MIPS kodunu yazın.

<<çözüm:

```
      beq   $19, $20, L1      # i = j ise L1 -e git
      add   $16, $17, $18     # f = g + h  (i=j ise atlanır)
L1:   sub   $16, $16, $19     # f = f – i (herzaman yürütülür)
```

Burada **L1**, çıkarma (**sub**) komutunun bellek adresine karşılık gelen anlık değerdir. Bildiğiniz gibi bu komutlar bellekte saklıdır ve her komut için bir bellek adresi bulunur.

>>

3.4.2 Atla

MIPS -te, sıradaki çalışacak komutun adresini koşulsuz değiştiren komut atla "**jump**" komutdur, ve anımsatıcısı "**j**" dir. Atla komutunun adres alanı 26-bittir ve sozcuk-adresi kullanır. Bu alan bayt-adrese donüştüğünde 28-bit olur. Geri kalan belirlenmemiş dört bit bellek uzayını 16 yerel sayfaya böler. O an kullanılan yerel bellek sayfasını program sayacının en sol 4-biti belirler. Atla komutu ister yakın ister uzak yalnızca yerel sayfa içinde atlamaya kullanılır.

Örnek 3-19: **f, g, h, i, j** değişkenlerinin sırasıyla **$16, $17, $18, $19, $20** da saklandığını varsayalım. Aşağıdaki C koduna

```
if (i = = j) f = g + h;
else f = g – h;
...
```

karşılık gelen MIPS kodunu yazın.

<< İyi programlamacılık açısından kodun (*koşul*, *ise-bölümü*, *değilse-bölümü*) sıralamasıyla yazılması gerekir. Buna uygun MIPS kodu ise şöyledir.

```
          bne   $19, $20, ElsePart  # eğer i≠j ise Else -e git
          add   $16, $17, $18       # ise bölümü: f=g+h (i≠j ise atla)
          j     ExitIf              # değilse bölümünü atla
ElsePart: sub   $16, $17, $18       # değilse bölümü: f=g–h, (i=j ise geç)
ExitIf:   ...
```

>>

Örnek 3-20: Koşullu döngülü C kodunu ele alın:

```
Loop:     g = g + A[i];
          i = i + j;
          if (i != h) goto Loop;
      ...
```

A bellekte **Astart** -tan başlayan 100 elemanlı bir tamsayı dizisi olsun, ve **g**, **h**, **i**, **j** sırasıyla **$17**, **$18**, **$19**, **$20** de saklansın. Sözcük-adresleme için **i** -yi **4** ile çarpmak gerektiğine dikkat edin. "×**4**" çarpımı için, kaydır (**sll $varış,$kaynak,2**) komutunu kullanın. Karşılık gelen MIPS çevirici kodunu yazın.

<< çözüm:

```
Loop:   sll   $9, $19,2          # geçici yazmaç $9 = 4 × I
        lw    $8, Astart($9)     # geçici yazmaç $8 = A[i]
        add   $17, $17, $8       # g = g + A[i]
        add   $19, $19, $20      # i = i + j
        bne   $19, $18, Loop     # i≠h ise Loop -a git
        ...
```

>>

Örnek 3-21*:* Aşağıdaki:C kodunun

```
while (S[i]= = k)  i = i + j;
...
```

i, **j**, **k** sırasıyla **$19 $20** ve **$21** de saklanıyor, **S** tamsayı dizisi **Sstart** adresinden başlıyor olsun. **$1=$2**×**4** çarpımı için **sll $1,$2,2** kullanarak MIPS çevirici kodunu yazalım.

<< karşılık gelen MIPS çevirici kodu:

```
Loop: sll   $1, $2, 2          # geçici yazmaç $9 =i×4
      lw    $8, Sstart($9)     # geçici yazmaç $8 = S[i]
      bne   $8, $21, Exit      # S[i]≠k ise, Exit -e git
      add   $19, $19, $20      # i = i + j
      j     Loop               # Loop -a git
Exit: ...
```
>>

MIPS -te **mult** komutu **$rs** ve **$rt** yazmaçındaki iki tamsayının çarpımını bulur, ama 64-bitlik sonuç **HI** ve **LO** adlı iki özel yazmaca yerleştirilir. **mfhi** ve **mflo** komutları **HI** ve **LO** yazmaçlarından herhangi genel amaçlı yazmaçlara veri aktarımı sağlar. MIPS çeviricide

```
mul  $rd,$rs,$rt
```

iki komuta dönüştürülen bir sözde-komuttur.

```
mult  $rs,$rt   # (HI,LO) ← $rs × $rt
mflo  $rd       # $rd ← LO
```

.

3.4.3 Değişkenlerin eşitsizlik testi

Tabii ki bütün göreli sınamalar yalnızca eşit-ise-dallan (*branch-if-equal* **beq**) ve eşit-değilse-dallan (*branch-if-not-equal* **bne**) komutlarıyla elde edilemez. MIPS eşitsizlik içeren sınamalar için küçükse-bir-yap (*set-on-less-than* **slt**) diye yazmaçtan-yazmaca aritmetik-mantık komutuna sahiptir. Eğer birinci kaynak yazmacın içeriği ikinci kaynak yazmacın içeriğinden küçükse varış yazmacına bir yazar, aksi durumda hedef yazmaca sıfır yazar.

Örnek 3-22: **slt $8, $19, $20**
komutunun yaptığı iş
eğer **$19** < **$20** ise **$8** -e bir yaz ,
değilse, **$8** -e sıfır yaz.

MIPS derleyicileri ve çeviricisi **slt**, **slti, beq**, **bne** -yi ve **r0**, **r1** ve **r2** yazmaçlarını kullanarak tüm olası göreli koşulları sınayabilir.

Örnek 3-23: Aşağıdaki C koduna karşılık gelen çevirici kodunu yazın

```
if ( a < b ) goto Less ;  // a ve b  $16 ve $17 de dursun
```

```
<<çözüm
        slt     $1, $16, $17     # a < b ise geçici yazmaç $1 1 alır
        bne     $1, $0, Less     # $1≠$0 (yani a<b) ise, Less -e git
>>
```

Bu iki komut “az-ise-dallan” (branch-if-less-than **blt**) koşulunu yerine getirir. Bir *sözde-komut* olan **blt** MIPS çeviricisi tarafından yukarıda görülen slt ve bne komutlarına dönüştürülür. Karmaşık bir **blt** komutu donanımı kurarak saat dönümünü genişletmek, saat hızını yavaşlatmak, ya da daha çok CPI gerektirmektense MIPS işlemci **blt** komutunu hızlı iki komut ile gerçekleştirir.

Örnek 3-24:

Aşağıdaki iş için yazılacak kodu set-on-less than-immediate (**slti**) komutuyla yazalım.

$18 -in içeriği 10 -dan daha küçükse, **LESS** -e dallan

```
<<
 slti   $8, $18, 10        # $18 < 10 ise $8 = 1 değilse $8= 0
 bne    $8, $0, LESS       # $8 ≠ 0   (yani $18 < 10) ise LESS -e dallan
>>
```

3.4.4 Derleyici , Çevirici, Bağlayıcı ve Yükleyici

Bir bilgisayar, belleğine yüklenen makine kodu programı herhangi bir kod çevrimi gerekmeden doğrudan çalıştırabilir. Makine kodu program dosyası "yürütülebilir dosya" "*executable file*" diye de bilinir. Yürütülebilir dosyalar, ikili kodlanmış komutlarla adreslerden oluşur. Böyle bir kodu elle yazmak bezdirici bir iştir. Genellikle, bir *çevirici program* çevirici dili kaynak programını ikili kodlanmış nesne dosyasıne çevirir.

Kullanıcı uygulama programları çoğunlukla C, FORTRAN, Pascal gibi bir sembolik yüksek seviye dilde yazılır. Genel amaçlı yüksek seviye dilleri standart birimsel program akış yapılarını içerir ve geniş bir veri yelpazesini destekler. Sembolik yüksek seviye programların yazıldığı *metin dosyasına* genellikle *kaynak dosyası* denilir.

Derleyici, bir programın kaynak dosyasını çevirici diline, veya nesne dosyasına, ya da doğrudan makine koduna çevirebilen bir ticari yürütülebilir program kodudur. *Nesne dosyası,* yürütülebilir dosya ile çevirici kaynağı arasındaki adımdır. Esasen programın makine

kodudur, ama uzak ve dış adresler halen sembolik biçimdedir. Yani, bu adresler hala bazı etiketlerle belirtilir. Bu etiketlerin dosya yürütülmek üzere belleğe yüklenmeden önce çözümlenerek karşılık gelen ikili adreslerle değiştirilmesi gerekir. Nesne-kodu tüm programı bir seferde ya da bölümler halinde derlemeye ve çevirmeye izin verir. Böylece bir yordamdaki değişiklik sadece ilgili yordamın derlenip ve yeniden çevirilmesini gerektirir. Yordam bir kere tümüyle test edildikten sonra, diğer programcıların kullanabilmesi amacıyla bir nesne kütüphanesine konabilir.

Bir *bağlayıcı programı* ayrı ayrı derlenmiş nesne kodlarıyla kütüphane kodlarını birleştirip yürütülebilir program oluşturur. Bağlayıcı sistem programı, yordamları (birimleri) bellekte birbiri ardına yerleştirir, yordamlar arasındaki çapraz çağrıları onarır, etiket adreslerini gerçek bellek adresine düzeltir, ve bir yürütülebilir dosya oluşturur.

Yükleyici, verilen bir yürütülebilir dosyayı yığın bellekten okuyan, ve işlemci tarafından yürütülebilmesi için doğru edresten başlayarak sistem belleğine yazan, ve ardından bellekteki programın yürütülmesini başlatan küçük bir yürütülebilir program kodudur.

Birçok OS (*işletim sistemi*) programı bağlayıcı ile yükleyici yerine iki işi de yapan *bağlayıcı-yükleyici* kullanırlar. Birçok derleyici, yüksek-düzey kaynağı doğrudan çevirebilen, belleğe yükleyebilen, ve ayıklayıcı denetiminde yürütebilen tümleşik geliştirme ortamına sahiptir. *Ayıklayıcı* işlemcinin denetimli benzeşimini yapabilen bir yazılımdır. Böylece program geliştirmeci programın her adımında veri yapılarının değişimini takip edebilir.

3.5 Altyordam Donanımı

Programcılar bir programı birimselleştirmek için altyordam kullanır. Böylece program anlaşılması kolaylaşır ve kod parçaları tekrar kullanılabilir. Bir komut takımı program birimleri kurmak için *yordam çağır* ve *yordamdan dön* gibi destekleyici komutlar sağlamalıdır. Ayrıca yordama değiştirgeleri nasıl aktaracağımız ve yordam çağrılarını nasıl iç içe koyacağımız gibi konularda kurallar gerekir.

3.5.1 jal ve jr komutları

Daha önce koşulsuz yakın adreslerın yanısıra uzak atlama için de kullanılan *atla* komutunu (j) tanıtmıştık. MIPS, betimlenen 26-bitlik yerel-sözcük-adresine atlarken aynı anda birsonraki yerin adresini yazmaç-31 -e saklayan, atla-ve-bağla *jump-and-link* (anımsatıcısı **jal**) komutuna sahiptir.

jal YordamAdresi

jal önce birsonraki yeri göstermek üzere program sayacını arttırır (*PC*←*PC*+4), ardından bu değeri **r31** -e saklar.

MIPS -in yordamdan dönüş için dönüş-adresine-atla (*jump-return-address* yada *jump to register*) komutu bulunur. Bu komutun anımsatıcısı **jr** -dir, ve dönüş adresini içeren yazmacı belirten bir tek işleneni vardır. **jal** dönüş adresini **r31**-e sakladığından, **jr** genellikle **$31** ile kullanılır:

jr $31

jr $31, yürütürken **r31** -in içeriğini (dönüş adresi) *PC* -ye kopyalar, ve böylece **r31** -te saklanan adrese atlamış olur.

3.5.2 İçiçe yordam çağrıları

Bir yordam çağrısı içinde başka bir yordam çağrısına içiçe yordam çağrısı denir. Kimi durumlarda çağrı içinde çağrı zinciri yüzlere varabilir. Çağrı içindeki çağrı sayısına içiçe çağrı derinliği (depth of the nesting) denir.

MIPS "**jal**" komutu dönüş adresini herzaman **r31** -e kaydeder. Tabii ki, içiçe çağrıda r31 -deki eski dönüş adresi başka bir yazmaca yada bellekte bir yere saklanmazsa, yeni dönüş adresi de **r31** -e saklanarak eski dönüş adresini bozar. Bu sorun, eski dönüş adresleri bir yığıta saklanak çözülür. *Yığıt* son girenin ilk çıktığı bir bellek örgütüdür. Yığıtı gerçekleştirmek için yığıt tepesini gösteren bir yığıt göstergesi kullanılır. MIPS sistemlerinde, **r29** yazmacı yığıt göstergesi olarak kullanılmak üzere ayrılmıştır.

Tablo 3-4 İçiçe yordam çağrıları ve yığıt işleyişi örneği

çağrı yapısı	çevirici kodu	işlemin izlenmesi
A: ...	**A: ...**	**1- $31 = bellideğil ; $29 = its**
...	**...**	**... *(its =baştaki yığıt-tepesi)***
call B	**jal B**	**2- $31 ← rB; B ye git**
rB': **...**	rB': **...**	**10-** *B -den dön (rB' -yi çalıştırmaya)*
	...	**...**
B: push $31	**B: addi $29,$29,–4**	**3- $29 ← its – 4;**
...	**sw $31,0($29)**	**Mem[its – 4] ← $31**
...	**...**	**...**
call C	**jal C**	**4- $31 ← rC; rC' -ye git**
rC': **...**	rC': **...**	**7-** C' -den dön *(rC' -yi çalıştırmaya)*
...	**...**	
pop $31	**lw $31,0($29)**	**8- $31 ← Mem[its – 4] = rB'** ($31 rB' -yi tutar)
	addi $29,$29,4	**$29 ← its ;**
return # to rB'	**jr $31**	**9- jump to $31 = rB'**
C: ...	**C: ...**	**5-** C -yi çalıştır
...	**...**	**...**
return # to rC'	**jr $31**	**6- $31 = rC' -ye git**

Dikkat ederseniz en içteki yordamın dönüş adresini yığıta saklamak gerekmez. Tüm diğer yordamlar **$31** -i yığıta saklamak zorundadır. *rB'* ve *rC'* etiket değildir, çağrıların dönüş adreslerini belirtmek için kullanılmıştır.

Tablo 3-5 Yığıt kullanılan içiçe çağrıda bellek ve yazmaç içerikleri

1- A, B'yi çağırmadan

yazmaçlar			bellek
			..
		ıts–8:	..
		ıts–4:	..
r29: **ts= its**	←	-its:	baştaki yığıt tepesi
r30:			..
r31:			yığıt dibi

2- A, B'yi çağırdıktan sonra

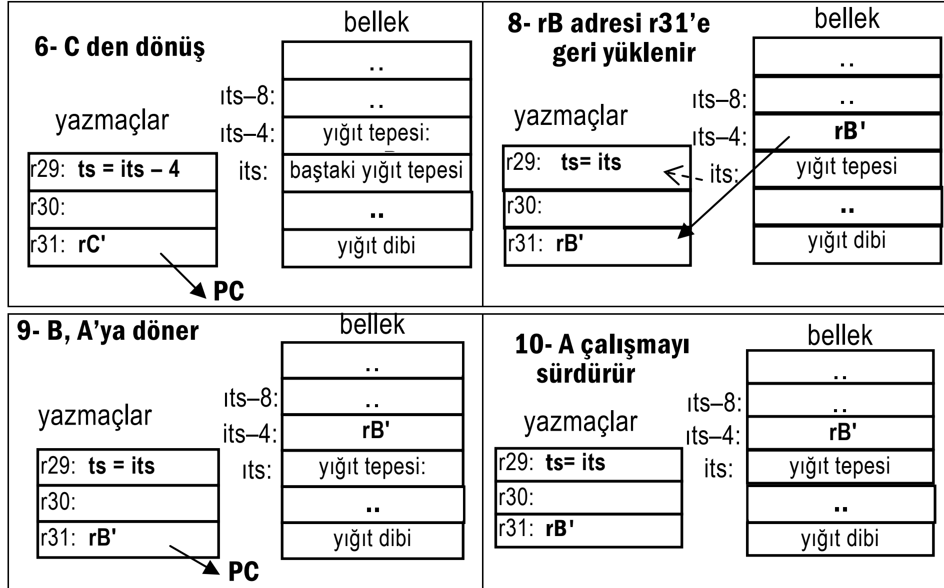
yazmaçlar		bellek
		..
	ıts–8:	..
	ıts–4:	..
r29: **ts= its**	its:	baştaki yığıt tepesi
r30:		..
r31: **rB'** (← **PC**)		yığıt dibi

3- rB' yığıta yazılıyor

yazmaçlar		bellek
		..
	ıts–8:	..
	ıts–4:	yığıt tepesi: **rB'**
r29: **ts =its – 4**	its:	baştaki yığıt tepesi
r30:		..
r31: **rB'**		yığıt dibi

4- B, C'yi çağıdıktan sonra

yazmaçlar		bellek
		..
	ıts–8:	..
	ıts–4:	yığıt tepesi: **rB'**
r29: **ts= its – 4**	its:	baştaki yığıt üstü
r30:		..
r31: **rC'** (← **PC**)		yığıt dibi



Örnekte, **A** etiketli ilk program parçası **B** etiketli altyordamı çağırır, ve bu altyordam da **C** etiketli başka bir yordamı çağırır. Dikkat ederseniz dönüş adresi yığıta kaydedilmemiş olsaydı **C** -den **B** -ye dönüş adresi olan *rC'*, *r***31** -deki **B** -den **A** -ya dönüş adresi olan *rB*' -nin üzerine yazılacaktı.

Bu içiçe çağrı için MIPS çevirici kod parçası:

```
A:      ....
        ....
        jal   B             # B -yi çağır, dönüş adresi rB' -yi $31 -e kaydet
rB'     ....                # rB': çağrıdan dönüş adresi.

B:      addi  $29, $29, – 4 # yeni değere yer açmak için yığıtı ayarla
        sw    $31, 0($29)   # B'nin dönüş adresini yığıta kaydet
        ....
        jal   C             # C -yi çağır, dönüş adresi rC' -yi $31 -e kaydet
rC'     ....                # rC': çağrıdan dönüş adresi.
        ....
        lw    $31, 0($29)   # B -nin dönüş adresini yığıttan $31 -e indir.
        addi  $29, $29, 4   # yığıt tepesini ayarla,
        jr    $31           # B -yi çağıran yordama geri dön

C:      ....                # C -de başka çağrı olmadığından $31 -i saklama
        ....
        jr    $31           # C -yi çağıran yordama geri dön
```

Çeviricinin veya program kodunun çalışması için *rB'* ve *rC'* -yi etiketlemek gerekmez. Bu etiketler sadece okuyucuya çağrıların dönüş edreslerini belirtmek için kullanılmıştır.

3.5.3 Yordam Argümanları (Değiştirgeleri)

Yordam argümanlarının veya değiştirgelerinin yordamın program koduna aktarılması konusunda MIPS'in alışılagelen ilkesi aktarılacak veri yapılarının sayısına ve büyüklüğüne bağlıdır. Değiştirgeler dörtten az ve tamsayıysa **r4** -den **r7** -ye kadarki yazmaçları kullanmak aktarmada kolaylık sağlar. Daha geniş veri yapıları aktarılırken gösterge (adres) kullanılır. Değişkenler dördü aşıyorsa, ilk dördü yazmaçta, geriye kalanları yığıtta aktarılır.

Bir yordam bir başka yordamı çağırırsa, dönüş adreslerinin yanısıra değişken yazmaçlarını da yığıta saklayıp dönüşte geri yüklemelidir. Içiçe çağrı durumunda yazmaç içeriklerini saklamak için pratikte iki yaygın yöntem uygulanır.

***Çağıran saklar*:**
Çağıran yordam yazmaçların saklanması ve geri yüklenmesinden sorumludur. Çağırılan yordam koşulsuz olarak yazmaçların hepsini kullanabilir.

***Çağırılan saklar* :**
Çağırılan yordam yürütülürken içeriğini değiştirdiği yazmaçların saklanıp geri yüklenmesinden sorumludur. Geri dönüşte çağıran yordam koşulsuz olarak bütün yazmaçları eski içeriğiyle bulmalıdır.

En-iyileştirici-derleyiciler genellikle bu iki yöntemden ya en-hızlı-kod yada en-az-komut vereni kullanır.

Örnek 3-25*:* **v[.]** sözcük-dizisinde ardışık iki yerde bulunan değerleri karşılıklı yer-değiştiren *swap* yordamı için MIPS çevirici kodunu yazın.

swap prosedürü için C kodu:

```
swap(int v[ ], int k)
{
  int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

C den assembly diline geçirirken uyulacak adımlar:

1- yordamdaki değişkenlere yazmaç ayır.
2- yordam gövdesinin MIPS kodunu yaz.
3- yordam çağrılışı ve dönüşünde yazmaçları sakayıp geri yükleme işini hallet.

<<çözüm:

1. **swap** -ın **v[ ]** ve **k** diye iki değiştirgesi var. MIPS kurallarına uygun olarak **r4** ve **r5** yazmaçlarını **v**[]'in göstergesi (başlangıç adresi) ve **k** değiştirgesi için kullan. Geçici değerler için **r15** ve **r16** -yı, adres hesaplamaları için de **r2** -yi kullan.

2. Dizimiz *bayt-dizisi* olsaydı, elemanların adresleri birer birer artacaktı.

```
add   $2, $4, $5    # $2 ← v[]+k, böylece $2 v[k] -nin adresini alır
lw    $15, 0($2)    # $15 (temp) ← v[k]
lw    $16, 1($2)    # $16 ← v[k+1]
sw    $16, 0($2)    # v[k] ← $16 = v[k+1]
sw    $15, 1($2)    # v[k+1] ← $15 = v[k]
```

Ancak, MIPS çeviricisinde etiketler bayt-adresi olmasına karşın, diizideki sözcükler gerçekte 4 bayt aralıklıdır. Bu yüzden, **$2** deki **v**[]+k değeri **v[k]** nın adresi değildir. Çünkü **v[k+1]** -nin adresi de **v[k]** -nin adresinin 4 ilerisidir.

```
sll   $2,$5,2       # $2 ←  k ×4
add   $2, $4, $2    # $2 ← v+(k × 4)
lw    $15, 0($2)    # $15 (temp) = v[k]
lw    $16, 4($2)    # $16 = v[k+1]
sw    $16, 0($2)    # v[k] , $16 değerini  ( =v[k+1] ) alır.
sw    $15, 4($2)    # v[k+1] , $15 değerini ( = v[k]  ) alır.
```

3. Yordam dönüşünde yazmaçtaki değerleri koruma:

*çağıran saklar standardı*nı kullandığımızı varsayın. swap sırasında **$2**, **$15**, ve **$16** yazmaçlarındaki değerler değişikliğe uğramaktadır (bu yazmaçlar bazı satırlarda varış yazmacı olarak kullanılmışlar.).

Yığıt göstergesi yazmaç-29 -da durur. Üç sözcüğü (**$2**, **$15** ve **$16**) yığıta saklamak için, yazmadan önce yazmacı $3 \times 4 = 12$ bayt genişletmeliyiz.

```
        addi  $29, $29, –12   # MIPS yığıtı yüksekten
                              #   alçak adrese doğru büyür.
        sw    $2, 0($29)      # $2 -yi yığıta kaydet
        sw    $15, 4($29)     # $15 -i yığıta kaydet
        sw    $16, 8($29)     # $16 -yı yığıta kaydet
```

ve swap yordamının sonunda yazmaçları geri yüklemeliyiz. Swap yordamı başka yordam çağırmadığından **$31** -deki dönüş adresini saklayıp geri yüklemek gerekmez.

```
# swap yordamının MIPS kodu ***
Swap:
# a) swapta değiştirilen yazmaçları sakla ***
        addi  $29, $29, –12
        sw    $2, 0($29)      # $2 -yi yığıta  sakla
        sw    $15, 4($29)     # $15 -i yığıta  sakla
        sw    $16, 8($29)     # $16 -yı yığıta  sakla
# b) yordam gövdesi  ***
        sll   $2,$5,2         # $2  ←  k ×4
        add   $2, $4, $2      # $2  ←  v+(k × 4)
        lw    $15, 0($2)      # $15  ←  v[k]
        lw    $16, 4($2)      # $16  ←  v[k+1]
        sw    $16, 0($2)      # v[k] , $16 -yı ( = v[k+1] ) alır
        sw    $15, 4($2)      # v[k+1] , $15 -i (= v[k] ) alır

# c) swapta kullanılan yazmaçları geri yüklüyor ***
        lw    $2, 0($29)      # $2 -yi yığıttan geri indir
        lw    $15, 4($29)     # $15 -i yığıttan geri indir
        lw    $16, 8($29)     # $16 -yı yığıttan geri indir
        addi  $29, $29, 12    # yığıt göstergesini eski değerine getir
# d) çağıran prosedüre geri dön ***
        jr    $31
```

100 adresinden başlayan programın makine kodu:

bellek adresi		bellekteki değer						açıklama
etiket	**adr**	**op**	**rs**	**rt**	**rd**	**sh**	**fn**	**assembler kodlaması**
swap:	**100**	**8**	**29**	**29**	**–12**			**addi $29,$29,–12**
	104	**43**	**29**	**2**	**0**			**sw $2,0($29)**
	108	**43**	**29**	**15**	**4**			**sw $15,4($29)**
	112	**43**	**29**	**16**	**8**			**sw $16,8($29)**
	116	**0**	**0**	**5**	**2**	**2**	**0**	**sll $2,$5,2**
	120	**0**	**4**	**2**	**2**	**0**	**32**	**add $2,$4,$2**
	124	**35**	**2**	**15**	**0**			**lw $15, 0($2)**
	128	**35**	**2**	**16**	**4**			**lw $16, 4($2)**
	132	**43**	**2**	**16**	**0**			**sw $16, 0($2)**
	136	**43**	**2**	**15**	**4**			**sw $15, 4($2)**
	140	**35**	**29**	**2**	**0**			**lw $2, 0($29)**
	144	**35**	**29**	**15**	**4**			**lw $15, 4($29)**
	148	**35**	**29**	**16**	**8**			**lw $16, 8($29)**
	152	**8**	**29**	**29**	**12**			**addi $29,$29, 12**
	156	**0**	**31**	**0**	**0**	**0**	**8**	**jr $31**

3.6 Dallan ve Atla komutlarında Adresleme

Atla (**j** ve **jal**) ve dallan (**beq** ve **bne**) komutlarında ya anlık-adresler, yada bir yazmaçtaki atlama adresini (**jr**) kullandık. Bundan başka, bir işleneni bellekte olan **sw** ve **lw** komutlarında bellek adresini hesaplarken 16-bitlik yerdeğiştirme ile bir yazmaç içeriğinin kullanıldığını da gördük. Şimdi, bu adresleme kiplerini ayrıntılarıyla görelim.

3.6.1 Anlık atla adreslemesi

Atla komutlarının (**j** ve **jal**) komut kodunda 26-bitlik sözcük-adres alanı vardır. Çevirici dilinde bayt adresi kullanılmasına karşın makine kodunda atlanacak adres sözcük adresi biçiminde kullanılır.

Örnek:

j 40000 # yerel bellek sayfasında bayt-adresi 40000 -e git

j-tipi:	2	10000

MIPS belleği bayt-adresleri 32-bittir. Sözcük-adresinin başlangıç bayt-adresi şöyle bulunur

bayt adresi = 4×*sözcük-adresi* .

Dörtle çarpmak, 2 bit sola-kaydırmaya denktir ve 26-bitlik sözcük-adresinin bayt-adresi 28-bit olur. Bu yüzden Atla komutları (**j** ve **jal**) bellek adresinin üst 4-bitini betimleyemez. En soldaki bu 4-bit, bellek adres uzayını 16 sayfaya böler. 26-bitlik anlık-değer, bellek uzayının ancak onaltıda birini adresleyebilir. Daha uzaktaki adreslere atlamak için MIPS çeviricisi yazmaçtan-adresleme ile birlikte anlık-üste-yükle, ve topla komutlarını (**lui**, **add**, **jr**) kullanır.

3.6.2 PC-göreceli-adresleme

Koşullu dallanmada iki işlenen, ve 16-bitlik bir sözcük-adresi belirtilir.

Örnek: **bne $8, $21, Exit # Eğer $8 ≠ $21 ise Exit -e git**

I-tipi:	**5**	**8**	**21**	**"Exit"**
	6 bit	**5 bit**	**5 bit**	**16 bit**

Bu alana doğrudan sapılacak bellek adresi yazılsaydı, programların adres-uzayı 2^{18} = 256 kBayt -ı aşamayacaktı. Bu da program yazabilmek için oldukça yetersizdir. Bu kısıtlamayı aşmak için göreceli adresleme kullanılır. Göreceli adreslemede, sapılacak adres 16-bitlik anlık-değer ile betimlenen yazmaçtaki değer toplanarak bulunur. Bu çeşit adreslemeye yazmaç-göreceli-adresleme denilir.

32-bitlik *Program Sayacı* yazmacı (*PC*) herzaman yorumlanan komuttan sonraki yerin adresini taşır. göreceli adresin bazı olarak *PC* -nin kullanılması her durumda mümkün olan en yüksek sapma aralığını sağlar. Dallanma adresi, 16-bitlik-işaretli-anlık değer ile *PC* yazmacındaki değer toplanarak bulunur. Eğer sapma koşulu sağlandıysa, bu toplam atlamayı gerçekleştırmek için *PC* -ye aktarılır:

Eğer (dallanma koşulu sağlanıyor)
ise PC + anlık-değer × 4 → PC

Bu yol, 2^{32} baytlık program yazılabilmesine bile olanak sağlar. Ancak, hedef sapma adresi dallan komutundan 2^{17} bayttan daha uzakta olamaz. Tabii ki bu sınırı da 26-bitlik anlık adres kullanabilen atla komutuna saparak aşabiliriz. Baz yazmacı olarak *PC* kullanan baz yazmaçlı adreslemeye *PC-göreceli-adresleme (PC -ye göre adresleme)* denir.

Örnek 3-26*:*

Aşağıdaki çevirici kodlaması için bellekteki başlama adresi 80000 olsun, ve Sstart 1000 -e karşı gelsin.

```
80000  Loop:  sll   $9, $10, 2       # $9 ← i × 4
80004         lw    $8, Sstart($9)   # geçici yazmaç $8 =  Sstart[i]
80008         bne   $8, $21, Exit    # Sstart[i] ≠ k ise Exit -e git
80012         add   $19, $19, $20    # i=i+j
80016         j     Loop             # Loop -a git
80020  Exit:  ....
```

Karşılık gelen MIPS makine kodunu ondalık olarak yazın:

<<

80000	0	0	10	9	2	0
80004	35	9	8	1000		
80008	5	8	21	8 /4 =2		
80012	0	19	20	19	0	32
80016	2	80000 /4 = 20000				
80020	....					

>>

bne komutunda, 8/4=2 anlık değeri 80020 adresini bu adresin tüm bitlerine gerek kalmadan betimler. Bayt-adresi yerine göreceli-sözcük-adresi kullanan bir koşullu-dallanma hemen hemen 2^{17} (=128k) bayt atlayabilir. Daha uzaktaki adreslere atlamak için jump komutu gerekir.

Örnek 3-27: L1 -in 217 bayttan daha ötede olabileceğini varsayarak aşağıdaki C dili ifadesi için makine kodunu yazın

if(a = = b) then goto Hedef;

<<çözüm:

L1 yakındaki bir adresin etiketi değildir, bu yüzden 16-bitlik anlık beq komutunu bu adres ile kullanamayız

```
beq  $18, $19, Hedef   # Hedef uzakta olduğunda kullanılamaz
```

Aynı işlem doğrudan 26-bitlik sözcük-adreslemesine izin veren **j** ve koşulu sınamak üzere **bne** kullanarak yapılabilir:

```
        bne     $18, $19, Gitme
        j       Hedef
Gitme:  ...
```

bne -den sonraki ilk komuttan **Gitme** etiketine olan uzaklık bayt ve sözcük cinsinden tam olarak olarak bilindiğinden, aynı kod doğrudan aşağıdaki biçimde de yazılabilir :

```
bne   $18, $19, 4
j     Hedef
```

burada **4**, atlama yerinin *PC* -ye göre bayt-adresidir, ve makine kodunda bu bayt adresine denk gelen sözcük-adresi 4/4= 1 -dir.

5	18	19	1 (=4/4)
2	**Hedef** -in sözcük-adres karşılığı		

Makine kodu alanında göreceli-sözcük-adresi kullanılması gerekmesine karşın çevirici dilinde daima göreceli-bayt-adresi kullandığımıza dikkat ediniz.
>>

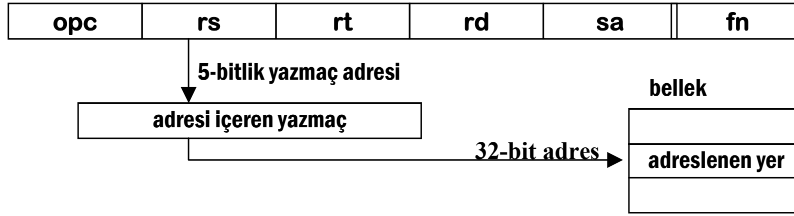
3.7 MIPS -in Adresleme Kipleri

MIPS -in dört temel adresleme kipi vardır:

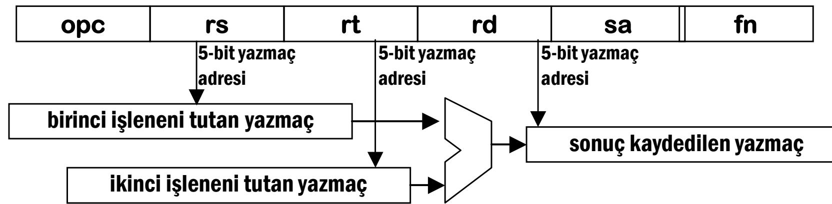
1. Yazmaç adreslemesi (**add**, **and**, ... , **jr**),
2. Baz yada yerdeğiştirme adreslemesi (**lw**, **sw**),
3. Anlık adresleme (**addi**, **andi**, .. , **j**, **jal**),
4. PC-göreceli adresleme (**beq**, **bne**).

3.7.1 Yazmaçlı Adresleme

İşlenen bir yazmaçta ise yazmaçlı adresleme kipi kullanılır. Bir yazmaç adres alanı her zaman 5-bittir.

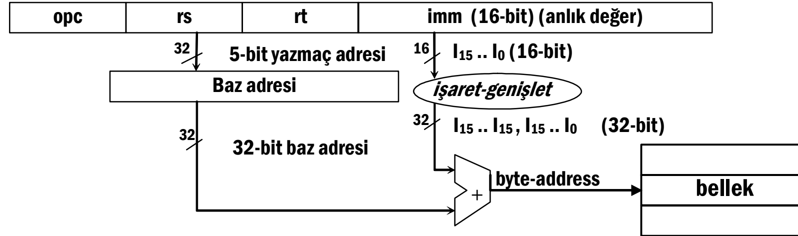


Jump-register (**jr**) komutu yazmaçlı adresleme kipini kullanır. **add** de betimlenen yazmaçlardan hiçbiri bellekte bir yere karşılık gelmiyor olsa da işlenenlerin tümü yazmaçlardaki tamsayılar olan **add** komutu da yalnızca yazmaçlı adreslemeyi kullanır .



3.7.2 Baz yada yerdeğişimli adresleme

İki işlenenden biri bellekteki bir yerse baz yada yerdeğişimli adresleme kullanılır. Bellekteki yerin adresi *komutta belirtilen yazmaçtaki değer* ile *16-bit-işaretli-anlık-bayt-yerdeğişimi* toplanarak hesaplanır.



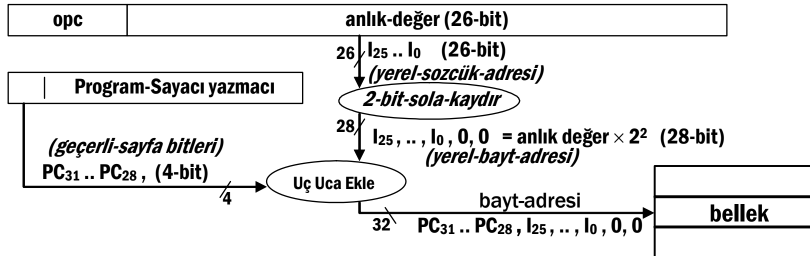
sözcük-sakla ve sözcük-yükle komutları (**sw** ve **lw**) bu adresleme kipini kullanır.

3.7.3 Anlık adresleme:

İşlenen değer komutun içinde bir sabit olarak yer alırsa bu kip kullanılır.

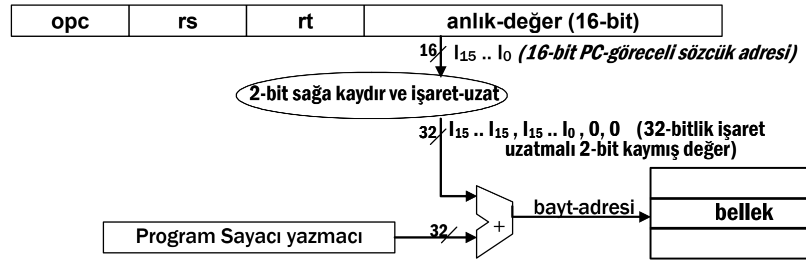
anlık-değer kullanan aritmetik-mantık komutları bu adresleme kipini kullanır (**addi, andi, ori, slti, lui**).

Atla (**j** ve **jal**) komutlarının da anlık adresleme komutları olduğunu hatırlayınız, ancak bunların anlık yerel-sözcük-adres alanı 26-bittir.



3.7.4 PC-göreceli adresleme

PC-göreceli-adreslemede, komuttaki 32-bitlik göreceli bayt-adresine dönüştürmüş 16-bitlik anlık değer ile PC -nin toplamı adresi verir. **beq** ve **bne**, PC-göreceli-adres kullanan komutlardır.



3.8 MIPS Temel Komut Alt Takımı

Aşağıdaki çizelge bu bölümde gördüğümüz komutların tiplerini ve makine kodlarının bir özetidir.

Tablo 3-6 Seçilmiş MIPS komutlarının makine kodları

R tipi biçim *alan-boyut(bit):*	*opc* 6	*rs* 5	*rt* 5	*rd* 5	*sa* 5	*fn* 6	*anımsatıcı $ rd,$rs,$rt* *anımsatıcı $rs,$rt yada anımsatıcı $rd*
Komut	**Bellek Kodu Örneği**						**Çevirici Dili Örneği**
sll	**0**	**0**	**2**	**1**	**3**	**0**	**sll $2,$1, 3 sola mantıksal kaydır**
jr	**0**	**31**	**0**	**0**	**0**	**8**	**jr $31 (5) yazmaçtaki-adrese-atla**
mflo	**0**	**0**	**0**	**1**	**0**	**18**	**mflo $1 LO-yazmacından-yolla**
mult	**0**	**2**	**3**	**0**	**0**	**24**	**mult $2,$3 işaretli-çarpma**
multu	**0**	**2**	**3**	**0**	**0**	**25**	**multu $2,$3 işaretsiz-çarpma**
add	**0**	**2**	**3**	**1**	**0**	**32**	**add $1,$2,$3 topla**
addu	**0**	**2**	**3**	**1**	**0**	**33**	**addu $1,$2,$3 işaretsiz-topla**
sub	**0**	**2**	**3**	**1**	**0**	**34**	**sub $2,$1,$3 çıkar**
subu	**0**	**2**	**3**	**1**	**0**	**35**	**subu $2,$1,$3 işaretsiz-çıkar**
and	**0**	**2**	**3**	**1**	**0**	**36**	**and $2,$1,$3 bitbite-ve**
or	**0**	**2**	**3**	**1**	**0**	**37**	**or $2,$1,$3 bitbite-veya**
slt	**0**	**2**	**3**	**1**	**0**	**42**	**slt $2,$1,$3 küçük-ise-birle**
sltu	**0**	**2**	**3**	**1**	**0**	**43**	**sltu $2,$1,$3 işaretsiz-küçük-ise-birle**

J tipi biçim *alan-boyut(bit):*	*opc* 6	*immJ (sözcük-adr)* 26	*anımsatıcı immJ*
j	**2**	**10000 (4)**	**j 40000 atla**
jal	**3**	**10000 (4)**	**jal 40000 atla-ve-bağla**

I tipi biçim *alan-boyut(bit):*	*opc* *6*	*rs* *5*	*rt* *5*	*imm* *16*	*anımsatıcı $ rd,$rs,anlık-değer (1) (2)*, ya da *anımsatıcı $rt, yerdeğiştirme($rs)*
Komut	**Bellek Kodu Örneği**				**Çevirici Dili Örneği**
beq	**4**	**1**	**2**	**100 (3)**	**beq $1,$2,400 eşit-ise-dallan**
bne	**5**	**1**	**2**	**100 (3)**	**bne $1,$2,400 eşit-değilse-dallan**
addi	**8**	**2**	**1**	**100**	**addi $1,$2,100 (1) anlık-topla**
slti	**10**	**2**	**1**	**100**	**slti $1,$2,100 anlık-azsa-birle**
lui	**15**	**0**	**1**	**100**	**lui $1,100 anlık-uste-yükle**
lw	**35**	**2**	**1**	**100**	**lw $1,100($2) (2) anlık-azsa-birle**
sw	**43**	**2**	**1**	**100**	**sw $1,100($2) (2) anlık-azsa-birle**

(1) *anlık-değer (16-bit);* (2) *yerdeğiştirme (16-bit, bayt-adresi)* (3) *PC-göreceli (16-bit sözcük-adresi)*	(4) *26-bit sözcük-adresi* (5) *yazmaç-adresleme.*

3.8.1 MIPS-II İşlem Kodu Çizelgesi

MIPS-II İşlem kodu çizelgesi bir RISC işlemcinin tüm işlem kodlarının kümesine örnektir. Komut kodu altı ana alandan oluşur:

	b31-b26	b25-b21	b20-b16	b15-b11	b10-b6	b5-b0
Komut Kodu:	**OPC**	**RS**	**RT**	**RD**	**SA**	**FN**

MIPS-II -de üç ana komut tipi vardır:

- Bireysel işlem kodlu komutlar.
- R-tipi (fonksiyon kodlu) komutlar,
- R-anlık (yazmaç-anlık) tipi (rt kodlu) komutlar,

Bireysel işlem kodlu komutların işlem kodu alanı (**OPC**, 31 den 26 ya kadarki bitler) OPCODE tablosunda listelenmiştir.

Tüm R-tipi komutların işlem kodu sıfırdır (OPC=0). Bu işlem-kodunun bir komut takımı için kullanıldığını göstermek üzere işlem-kodu tablosunda 0 hanesine R-Tipi(1) yazılmıştır. Bu komutları birbirinden işlev-alanındaki değer ayrıştırır. R-tipi komutların işlev alanı değerleri "*R-Tipi İŞLEV*" çizelgesinde listelenmiştir.

Gördüğümüz anlık- ve R-tipi komutlar kolay farkedilmesi için daha irice yazılmıştır.

anlık-yazmaç (REG-IMM) tipi komutların işlem kodu birdir (OPC=1) ve yalnızca bir işlenen yazmacı ile bir anlık-değeri bulunur. Komutlar birbirinden RT-alanı ile ayırdedilir. "R-Anlık rt-alanı" tablosu bu komut grubunu içerir. REG-IMM grubu birçok koşullu ve yakalamalı dallanma

komutları içerir. Yakalamalı komutlar yürütüldükten itibaren yakalama koşulu sağlandığı anda kural-dışı-durum oluşturur.

ADDI, **ADDIU**, **SLTI**, **SLTIU**, **ANDI**, **ORI**, **XORI**, **LUI** öğrendiğimiz anlık aritmetik mantık komutlarıdır.

BGTZ, **BLEZ**, **BEQL**, **BNEL**, **BLEZL**, **BGTZL** yakalamalı dallanma komutlarıdır,

COP0, **COP1**, **COP2**, **COP3**, **LWC1**, **LWC2**, **LWC3**, **LDC1**, **LDC2**, **LDC3**, **SC**, **SWC1**, **SWC2**, **SWC3**, **SDC1**, **SDC2**, **SDC3** yan-işlemciye ilişkin komutlardır, işlevi sistem-yan-işlemcisine bağlıdır.

LB, **LH**, **LWL**, **LW**, **LBU**, **LHU**, **LWR**, **LL**, ve **SB**, **SH**, **SWL**, **SW**, **SWR** değişik veri boyutları için (bayt, yarımsözcük, sözcük, ve çift-sözcük) yükle ve sakla komutlarıdır.

MIPS-II Komut İşlem kodu Haritası

	b31-b26	b25-b21	b20-b16	b15-b11	b10-b6	b5-b0
Komut Kodu:	**OPC**	**rs**	**rt**	**rd**	**sa**	**fn**

İşlem Kodları

OPC (31..29 bitler)

29..26bit leri	**bit 31 ve 30** 00....	01....	10....	11....
..0000	*R-tipi* (1)	*COP0*(1,2)	LB	LL
..0001	*R-anlık* (1)	*COP1*(1,2)	LH	LWC1(2)
..0010	**J**	*COP2*(1,2)	LWL	LWC2(2)
..0011	**JAL**	*COP3*(1,2,3)	**LW**	LWC3(2,3)
..0100	**BEQ**	BEQL	LBU	*
..0101	**BNE**	BNEL	LHU	LDC1(2)
..0110	BLEZ	BLEZL	LWR	LDC2(2)
..0111	BGTZ	BGTZL	*	LDC3(2,3)
..1000	**ADDI**	*	SB	SC
..1001	**ADDIU**	*	SH	SWC1(2)
..1010	**SLTI**	*	SWL	SWC2(2)
..1011	**SLTIU**	*	**SW**	SWC3(2,3)
..1100	ANDI	*	*	*
..1101	ORI	*	*	SDC1(2)
..1110	XORI	*	SWR	SDC2(2)
..1111	**LUI**	*	(4)	SDC3(2,3)

R- tipi komutlar

R-tipi (OPC =0) **için fn alanı (5..0 bitleri)**

bitler 3...0	**bit 5 ve 4** 00....	01....	10....	11....
..0000	**SLL**	**MFHI**	**ADD**	TGE
..0001	*	**MTHI**	**ADDU**	TGEU
..0010	**SRL**	**MFLO**	**SUB**	TLT
..0011	**SRA**	**MTLO**	**SUBU**	TLTU
..0100	SLLV	*	**AND**	TEQ
..0101	*	*	**OR**	*
..0110	SRLV	*	**XOR**	TNE
..0111	SRAV	*	**NOR**	*
..1000	**JR**	**MULT**	*	*
..1001	JALR	**MULTU**	*	*
..1010	*	**DIV**	**SLT**	*
..1011	*	**DIVU**	**SLTU**	*
..1100	**SYSCALL**	*	*	*
..1101	**BREAK**	*	*	*
..1110	*	*	*	*
..1111	SYNC	*	*	*

R-anlık komutlar (OPC=1) **için *rt* alanı (20..16 bitleri)**

20,19 bitleri	**18,17,16 Bitleri** ..000	..001	..010	..011	..100	..101	..110	..111
00...	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
01...	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
10...	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
11...	*	*	*	*	*	*	*	*

(*) gelecek kullanımlar için ayrılmış işlem kodu (opc), kuraldışıyla sonuçlanır,
(1) R-tipi, R-anlık ve COPx komut sınıflarını belirtir
(2) Yan-işlemci işlem için ayrılmış
(3): Sonraki sürümde kaldırıldı
(4): Sistem yan-işlemcisi için ayrılmış.

Kaynak: *MIPS-IV Instruction Set, Revision 3.2* By Charles Price, September, 1995
MIPS Technologies, Inc. 2011 North Shoreline Mountain View, California 94039-7311

3.9 Çözümlü Problemler

Q1) Aşağıdaki C kodunu MIPS assemler koduna çevirin ve karşılığı olan ondalık makine kodunu adres 300 den başlayarak yazınız. Yazmaç saklama ve geri yükleme için <u>çağırılan saklar yöntemi</u>ni kullanın.

<u>Ana yordam</u>

```
.....
j=5
f(j,x);
x[3]+=5;
.....
```

<u>f yordamı</u>

```
void f(int j,int x[])
{
  if (j= =7)
       x[2]=6*x[1]–x[0];
  else g(y);
 }
```

<u>g fonksiyonu</u>

```
void g(int y[])
{
  int i;
  for (i=2;i<12;i++) y[i]*=2;
   y[2] – = 4;
 }
```

İstenenler

<u>ana yordamda:</u>
j için $4 -ü kullanın.
x -in başlangıç adresi 10000 olsun.
j ve x[] <u>sadece main -in içinde bilinsin</u>.
x[3] -ün değeri için $10 -u kullanın.

<u>f içinde:</u> (j -yi r4 ile, x -i ise r5 ile aktarın).
7 değeri için $6 -yı kullanın,
x[0] -in değeri için $7 -yi kullanın.
x[1] -in değeri için $8 -i kullanın.
x[2] -in değeri için $9 -u kullanın.
y -nin başlangıç adresi 20000 olsun.
y <u>global değişken</u> olsun.

<u>g içinde:</u>
i için $10 -u kullanın.
i<12 sonucu için $11 -i kullanın.
i×4 sonucu için $12 -yi kullanın.
y[i] değeri için $13 -ü kullanın

<<Çözüm:

(1) F -i çağırırken, j, ve x[] değişkenlerini r4 ve r5 te aktarın. "li $1,2" bir sözde-komuttur ve "addi $1,$0,2" olarak işlenir.

(2) x[3] adreslenırken ya "lw $10,10012($0)", ya da "lw $10,12($5)" kullanın. Burada x[] in başından sayarsak x[3] -ün bayt olarak başlangıç adresi $12=3\times4$ tür..

(3) F te, r4, r6, r7, r8, r9, and r31 varış yazmaçlarını yığıta saklayın. Yıgıtı bu yazmaçlar için genişletmek üzere yığıt göstergesini 6×4 bayt azaltın

(4) "jal G" dönüş adresini r31 e yazacağından r31 i yığıta saklayın.

(5) Yordamın önce bir global, sonra bir lokal değişkenle çağrılabileceğinden dolayı global bile olsa tüm değiştirgeleri aktarın.

(6) F ten tam donerken çağrılanın sakladığı yazmaçları geri yükleyin, Yığıt göstergesini eski değerine getirip "jr $31" komutuyla çağıran yordama dönün.

(7) G de , r10, r11, r12, and r13 varış yazmaçlarını Yığıta saklayın. Yıgıtı bu yazmaçlar için genişletmek üzere yığıt göstergesini 4×4 bayt azaltın.

(8) G den dönüşte çağrılanın sakladığı yazmaçları geri yükleyin, Yığıt göstergesini eski değerine getirip "jr $31" komutuyla çağıran yordama dönün.

```
        Main:  ...
(1)   {        li $4,5
               li $5,10000
               jal F
(2)   →        lw $10,12($5)        # $10 ← x[3]
               addi $10,$10,5       # $10 ← x[3] + 5
               sw $10,12($5)        # x[3] ←  x[3] + 5
               ...
        F:
      {        addi $29,$29,–24
               sw $4,0($29)
               sw $6,4($29)
(3)            sw $7,8($29)
               sw $8,12($29)
               sw $9,16($29)
               sw $31,20($29)
               li $6,7  # if (j! =7)
               bne $4,$6,Else       #   Else e git.
               li   $7,6            # $7 ←6 ,  ($7 geçici)
               lw $8,4($5)          # $8 ←x[1]
(3)            mult  $8,$7          # LO←6 × x[1]
               mflo  $8             # $8 ← 6 × x[1]
               lw  $7,0($5)         # $7 ← x[0]
               sub $7,$8,$7         # $7 ← 6×x[1] – x[0];
               sw  $7,8($5)         # x[2] ←6×x[1] – x[0];
               j ExitIf
        Else:
               li   $4,20000  ←     # y[ ] değiştirgedir
(4)   →        jal G  ←             (5)
        ExitIf:
      {        lw $4,0($29)
               lw $6,4($29)
               lw $7,8($29)
(6)            lw $8,12($29)
               lw $9,16($29)
               lw $31,20($29)
               addi $29,$29,24
               jr $31
        G:
      {        addi $29,$29,–16
               sw $10,0($29)
(7)            sw $11,4($29)
               sw $12,8($29)
               sw $13,12($29)
```

```
                                    # for döngüsü
        li $10,2                    #   $10= i ←2
Loop:
        slti $11,$10,12             #   if (i<12), $11←1
        beq $11,$0, ExitF           #   Eğer (i>=12) ise çık
        sll  $12,$10,2              #   4 × i
        add $12,$12,$4              #   y[]+ 4 × i
(7)     lw $13,0($12)               #   $13 ← y[i]
        add $13,$13,$13             #   2× y[i]
        sw $13,0($12)               #   y[i] ← 2 × y[i]
        addi $10,$10,1              #   $10= i ← i+1
        j Loop                      #   for döngüsü
ExitF:                              #   for döngü sonu
        lw $13, 8($4)               # $13 ← y[2]
        addi $13,$13,–4             # $13 ← y[2] – 4
        sw $13, 8($4)               # y[2] ← y[2] – 4
      { lw $10,0($29)
      { lw $11,4($29)
(8)   { lw $12,8($29)
      { lw $13,12($29)
      { addi $29,$29,16
        jr $31
```

Bu çevirici kodun bellekte bayt adresi 300 den başlayan ondalık makine kodu:

adres	bellek sözcük içeriği (komut alanları)						çevirici ifadesi
	opc	rs	rt	rd	sa	fn	
300:	8	0	4	5			li $4,5
304:	8	0	5	10000			li $5,10000
308:	3	81 *(=324/4)*					jal F
312:	35	5	10	12			lw $10,12($5)
316:	8	10	10	5			addi $10,$10,5
320:	43	12	10	12			sw $10,12($5)
F: 324:	8	29	29	–24			addi $29,$29,–24
328:	43	29	4	0			sw $4,0($29)
332:	43	29	6	4			sw $6,4($29)
336:	43	29	7	8			sw $7,8($29)
340:	43	29	8	12			sw $8,12($29)
344:	43	29	9	16			sw $9,16($29)
348:	43	29	31	20			sw $31,20($29)
352:	8	0	6	7			li $6,7
356:	5	4	6	8 *(= (392–360)/4)*			bne $4,$6,Else
360:	8	0	7	6			li $7,6
364:	35	5	8	4			lw $8,4($5)
368:	0	8	7	0	0	35	mult $8,$7
372:	0	0	0	8	0	18	mflo $8
376:	35	5	7	0			lw $7,0($5)
380:	0	8	7	7	0	34	sub $7,$8,$7
384:	43	5	7	8			sw $7,8($5)
388:	2	100 *(=400/4)*					j ExitIf
Else: 392:	8	0	4	20000			li $4,20000
396:	3	108 *(=432/4)*					jal G
ExitIf:400:	35	29	4	0			lw $4,0($29)
404:	35	29	6	4			lw $6,4($29)
408:	35	29	7	8			lw $7,8($29)

adres	bellek sözcük içeriği (komut alanları)						çevirici ifadesi
	opc	rs	rt	rd	sa	fn	
412:	35	29	8	12			lw $8,12($29)
416:	35	29	9	16			lw $9,16($29)
420:	35	29	31	20			lw $31,20($29)
424:	8	29	29	24			addi $29,$29,24
428:	0	31	0	0	0	8	jr $31
G: 432:	8	29	29	–16			addi $29,$29,–16
436:	43	29	10	0			sw $10,0($29)
440:	43	29	11	4			sw $11,4($29)
444:	43	29	12	8			sw $12,8($29)
448:	43	29	13	12			sw $13,12($29)
452:	8	0	10	2			li $10,2
Loop: 456:	10	10	11	12			slti $11,$10,12
460:	4	0	11	7 =*((492-464)/4)*			beq $11,$0, ExitF
464:	0	0	10	12	2	0	sll $12,$10,2
468:	0	12	4	12	0	32	add $12,$12,$4
472:	35	12	13	0			lw $13,0($12)
476:	0	13	13	13	0	32	add $13,$13,$13
480:	43	12	13	0			sw $13,0($12)
484:	8	10	10	1			addi $10,$10,1
488:	2	114 *(=456/4)*					j Loop
ExitF: 492:	35	4	13	8			lw $13, 8($4)
496:	8	13	13	–4			addi $13,$13,–4
500:	43	4	13	8			sw $13, 8($4)
504:	35	29	10	0			lw $10,0($29)
508:	35	29	11	4			lw $11,4($29)
512:	35	29	12	8			lw $12,8($29)
516:	35	29	13	12			lw $13,12($29)
520:	8	29	29	16			addi $29,$29,16
524:	0	31	0	0	0	8	jr $31
...	..	..	..	..	..	..	...

>>

Q2)Aşağıdaki MIPS kod parçasını ele alın. Kodun bellekte $(70000)_{10}$ -den başladığını ve bellekte peşpeşe ilk iki yerin aşağıdaki değerleri (onlu olarak) tuttuğunu varsayın.

MIPS kodu:

```
           . . . . .
           addi  $10,$0,5
           addi  $11,$0,200
Continue:  lw    $12,0($11)
           addi  $11,$11,4
           lw    $13,0($11)
           sll   $13,$13,1
           sub   $13,$13,$12
           addi  $11,$11,4
           sw    $13,0($11)
           addi  $11,$11,–4
           addi  $10,$10,–1
           bne   $10,$0,Continue
Exit:      . . . . .
```

Bellek

Bellek Adresi	Bellek içeriği
200	3
204	5

a) Yukarıdaki MIPS kodunu izleyin ve yürütüldükten sonraki bellek içeriklerini gösterin.

Bellek adresi	Bellek içeriği
200	3
204	5
208	
212	
216	
220	
224	

<< çözüm:

Bellek adresi	: Bellek içeriği
200	3
204	5
208	7
212	9
216	11
220	13
224	15

>>

b) Komut alanlarının içeriğini ve adreslerini ondalık sayılarla göstererek bu çevirici kodu için MIPS makine kodunu yazın, .

<<çözüm:

adres	opc	rs	rt	Anlık yada rd sa fn
70000	8	0	10	5
70004	8	0	11	200
70008	35	11	12	0
70012	8	11	11	4
70016	35	11	13	0
70020	0	0	13	13 1 0
70024	0	13	12	13 0 34
70028	8	11	11	4
70032	43	11	13	0
70036	8	11	11	– 4
70040	8	10	10	– 1
70044	5	10	0	– 10

>>

Q3) Aşağıdakilerin MIPS -in temel komutları olduğunu varsayın..

Kategori/Tip	**Komutlar**
İşlemsel / yazmaç-tipi:	**add, sub, addu, subu**
İşlemsel / anlık- tip	**addi, addiu**
Mantık / yazmaç- tipi:	**and, or, sll, srl**
Mantık / anlık- tipi:	**andi, ori**
Veri Transferi / yazmaç - tipi:	**lw, sw**
Veri Transferi / anlık- tipi:	**lui**
Şartlı Dallanma	**beq, bne,**
Karşılaştırma / yazmaç-tipi	**slt, sltu**
Karşılaştırma / anlık-tipi:	**slti, sltiu**
Şartsız Jump / yazmaç-tipi	**jr**
Şartsız Jump / J- tipi	**j, jal**

anlık değerler 16-bitliktir,

Aşağıdaki sözde-komutların herbiri için karşılık gelen temel MIPS kodlarını yazınız. Geçici sonuçlar için gerektiğinde $1 -i kullanın.

	sözde-komutlar	<<çözüm: gerçek MIPS kodları
i)	**subi $8, $11, 1** $8 ← $11 – 1	**addi $8, $11, -1**
ii)	**clr $10** $10 ← 0	**add $10, $0, $0**
iii)	**blt $10, $11, ilabel** if ($10 < $11) goto ilabel burada ilabel 16-bitlik anlık yakın adrestir	**slt $1, $10, $11** **bne $1, $0, ilabel**
iv)	**beq $10, $11, flabel** if ($10 = = $11) goto flabel; burada flabel 16-bite sığmayan uzak adrestir	**bne $10, $11, skip1** **j flabel** **skip1:**
v)	**bgez $10, ilabel** if ($10 ≥ 0) goto ilabel burada ilabel anlık yakın adrestir	**slt $1, $10, $0** **beq $1, $0, ilabel**

>>

Q5 Aşağıdaki C program parçası için MIPS kodunu yazın:

```
....
if (A>0) A=C–B;
else A=C+2;
D=0x035AFF21;
A=D–A;
....
```

burada 0x... 16-lık sayıları gösterir.
A için $2 -yi, B için $3 -ü, C için $4 -ü, D için $5 -I kullanın.

```
<<Çözüm:
            slt $1,$0,$2           # 0<A ise $1=1
            beq $1,$0,elsepart     #
 thenpart:
            sub $2,$4,$3           # A=C–B
            j exitif
 elsepart:
            addi $2,$4,2           # A=C+2
 exitif:
            lui $5,0x035A
            add $5,$5,0xFF21
            sub $2,$5,$2           # A=D–A
>>
```

Q6) Aşağıdaki MIPS assembly program parçasını izleyin ve sonucu verilen kutuya yazın. Ilk "addi" komutunun bellekte 200_{10} adresli yerde bulunduğunu varsayın.

```
        ...
200:    addi $5,$0,–8
        addi $6,$5,16
        jal FF
        lw $7,1200($0)
        sll $7,$7,2
        ......
FF:     addi $29,$29,–4
        sw $6,0($29)
        sltiu $6,$5,–1
        add $6,$6,$5
        sw $6,1200($0)
        lw $6,0($29)
        addi $29,$29,4
        jr $31
        ......
```

Mem[1200]=($)_{10}$
$31=($)_{10}$
$6=($)_{10}$

<< *Izleme sonucu:*

$5 = –8
$6 = –8+6 = 8 ,
$31 = 212

$29 = TOS – 4
Mem(TOS–4) = 8
$2^{32}–5 < 2^{32}–1$ ==> $6=1
$6 =$6+$5=1–8= –7
Mem(1200) = $6 = –7
$6 = Mem(TOS–4) = 8
$29 = TOS

$7 = Mem(1200) = –7
$7 = –28

lw $7,1200($0) komutunun adresi 212 dir.
Mem[1200]=(–7 $)_{10}$
$31=(212 $)_{10}$
$6=(8 $)_{10}$
>>

4 Bilgisayarlar için Aritmetik

4

Amaç
Bu bölüm, işaretli tamsayı aritmetiği için negatif sayıların gösterimini, aritmetik-mantık-birimindeki (ALU) ve kayan-noktalı-sayı-birimi gibi tam ve kesirli sayılarla dört işlem (toplama, çıkarma, çarpma, ve bölme) yapan donanımları ayrıntılarıyla tanıtmayı hedefler.

4.1 Sayı sistemi, Toplama ve Çıkarma

En fazla ondalık sisteme alışkın olduğumuzdan gündelik yaşamımızda ondalık sistemi kullanıyoruz. Ondalık sayılarla olan geniş deneyimimiz sonucu ondalık sistemde öğreniyor, kestiriyor, hesaplıyor, ve tahminde bulunuyoruz. Ondalık sistem doğal sayı sistemimizdir. Ondalık sistemin **sayı-tabanı** 10 -dur, yani, bu sistemde on rakam (0, 1, ... , 9) kullanır, ve her basamağın solundaki basamağın konum-ağırlığı kendi konum-ağırlığının 10 katıdır.

Örnek:

basamaklar

$1809.2_{10} = 1\times1000_{10} + 8\times100_{10} + 0\times10_{10} + 9\times1+2\times(1/10_{10});$

konum-ağırlıkları.

En basit basamak yapısına sahip olduğundan ötürü sayısal devrelerde ikili sayı sistemi kullanılır. İkili sistemde her basamağın ancak iki durumu olabilir. Devrenin çıkış voltajının ya düşük yada yüksektir ve basamağın 0 yada 1 olmasını betimler. İkili sistemin sayı tabanı ikidir.

Örnek: *bitler* *konum ağırlıkları*

$101101.001_2 = 1\times32 + 0\times16 + 1\times8 + 1\times4 + 0\times2 + 1\times1$
$+0\times0.5 +0\times0.25+1\times0.125$
$= 32 +0 +8 +4 +0 +1+ 0 +0 +0.125 = 45.125_{10}$

MSB *LSB* *kesir-noktası* *taban*

En soldaki basamağın konum-ağırlığı en yüksektir, ve en-değerli-bit anlamına gelen "most significant bit (MSB)" olarak adlandırılır. Konum ağırlığı en düşük olan en sağ basamağa ise en değersiz bit "least significant bit (LSB)" denir.

Uzun *“sıfır”* ve *“bir”* dizileri oluşturan ikili sayıları hatırlayabilmek zordur. Uzun ikili basamak zincirlerini betimlemek için bazı kodlama yöntemleri kullanırız. En kolay kodlama biçimi ikili basamakları üçer yada dörder gruplamak ve her basamak grubunu birer birer okumaktır. Üçerli gruplama *sekizlik* **"***octal"* sayı sistemini verir. Dörderli gruplar 16 çeşit rakam oluşturur. Doğal olarak, 0 -dan 9 -a rakamlar kendi ondalık anlamlarıyla kullanılır. Daha yüksek değerler için 10=A, 11=B, 12=C, 13=D, 14=E , 15=F gibi alfabetik işaretleri rakam olarak kullanınca onaltılık "*hexadecimal"* sayı sistemini elde ederiz.

Örnek:

$$101101.001_2 = 10_2 \times 16 + 1101_2 + 0010_2 \times (1/16)$$
$$= 2 \times 16 + 13 + 2 \times (1/16); \quad (13 = D_{16})$$
$$= 2D.2_{16}$$

Onaltılık sistem devrelerde yada makinelerde uygulanmaz, mühendisler içindir. İkilik basamakları, kolay okumak için gruplarız. Makineler hep ikilik sistemde çalışır. Onaltılık sayılar "h" sonekiyle yada 0x önekiyle belirtilir.

Örnek: $10\ 1100_2$ = 0b 10 1100 =$2C_{16}$ = 2Ch = 0x2C = 44

Bir *ikilik basamağa* kısaca *bit* denir. Bir bit sadece bir değer alabilir: ya “0” yada “1”.

Dört ardışık bitlik gruba *dörtbit* denir. Her dörtbit bir onaltılık basamak ile kodlanabilir.

Sekiz ardışık bitlik gruba *byte* (byte = 8-bit) denir.

Birçok sayısal devrede, yazmaçlar birkaç bayt tutabilir. Yazmaç genişliği sözcük genişliğini belirler. Bir sözcük tek dörtbit -ten birkaç bayt -a kadar herhangi bir genişlikte olabilir. MIPS te her sözcük bitidört bayttır (32-bit) .

4.1.1 İki sayı sistemi arasında dönüştürme

Onaltılık rakamların ikilik karşılıkları şu tablodadır

onaltılık	0	1	2	3	4	5	6	7
ikilik	0000	0001	0010	0011	0100	0101	0110	0111
ondalık	0	1	2	3	4	5	6	7

onaltılık	8	9	A	B	C	D	E	F

ikilik	1000	1001	1010	1011	1100	1101	1110	1111
ondalık	8	9	10	11	12	13	14	15

Ikilikten ondalığa dönüştürmede konumsal değerlerin toplamı kullanılır.

Örnek: onaltılıktan ondalığa dönüştürme

$76F2_{16} = 7\times16^3 + 6\times16^2 + 15\times16^1 + 2\times16^0$
$= 7\times4096 + 6\times256 + 15\times16 + 2$
$= 28672 + 1536 + 240 + 2 = 30210_{10}$

Örnek: ikilikten ondalığa dönüştürme

$101101_2 = 1\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$
$= 32 + 0 + 8 + 4 + 0 + 1 = 45_{10}$

yada önce onltılığa, sonra ondalığa dönüştürülür.

$= 2\times16 + 13 = 45_{10}$

İkilik ve onaltılık arasında donüşme:

İkilik sayı dörtbitlere gruplayarak onaltılık sayıya dönüştürülür.

Gruplama herzaman kesir noktasından başlar.

Örnek: ikilikten onaltılığa

$101\ 1111\ 0010_2 = 5F2_{16}$
$10.\ 1111\ 1001\ 011_2 = 2.F96_{16}$; son basamağa dikkat!

Örnek: onaltılıktan ikiliğe

$3E25.38_{16} = 11\ 1110\ 0010\ 0101\ .\ 0011\ 1_2$

Ondalıktan Onaltılığa dönüştürme ardarda tamsayı bölmelerle yapılır.

Örnek: $1238_{10} = ?_{16}$

	1238	
$1238=16\times77+6$:	77	6 en sağ bit
$77=16\times4+13$:	4	13 =D
$13=16\times0+6$:	0	4 en sol bit

Tamsayı kısmı 16 -ya böl, bölümü çizginin soluna ve kalanı da sağına yaz, sol taraf sıfıra düşene dek devam et.

Sonuç $4D6_{16}$ çıkar.

Ondalıktan ikiliğe dönüştürme:

Örnek: $103.625_{10} = ?_2$

	103	
$103=2\times51+1$:	51	1 en sağ bit
$51=2\times25+1$:	25	1
$25=2\times12+1$:	12	1
$12=2\times6+0$:	6	0
	3	0
	1	1
	0	1 en sol bit

Tamsayı kısmı 2 -ye böl, bölümü çizginin soluna ve kalanı da sağına yaz, sol kısım sıfıra düşene dek devam et.

tam sayı kısmı = 1100111_2

	.625		
$2\times.625=1.25$	25	1	en sol bit
$2\times.25=0.5$	.5	0	
$2\times.5=1.0$	0	1	en sağ bit

Kesir kısmı 2 ile çarp, tam kısmı sola ve kesir kısmını sağa yaz, sol kısım sıfırlanıncaya yada yeterince kesirli basamak oluncaya dek devam et.
Sayının tümü 1100111.101_2.

4.1.2 Eksi sayıların gösterimi

Eksi (negatif) sayıları göstermekte yaygın kullanılan notasyon biçimlerinden bazıları:

a) **İşaretli-büyüklük (signed-magnitude) biçimi**: most significant bit işareti belirtir, sıfır ise pozitif, bir ise negatif. Diğer tüm bitler sayının büyüklüğünü belirler, yani, 1011_2 4-bit signed-magnitude formatında –3 -e denk düşer.

b) **Eğilimli (biased) gösterim biçimi:** Negatif sayıları negatif olmayan tamsayılara çevirmek için bilinene bir değer eklenir. Örneğin, 4-bitlik 7-eğilimli sayılarda –5 i göstermek için $-5+7=2 = 0010_2$ kullanılır. 7-eğilimli 1110_2 sayısı ondalık sistemde 14–7= 7 eder.

c) **işaretli-ikilik (signed binary) yada 2-lik-tümleyen sayı biçimi:** (bilgisayarlarda çoğunlukla tamsayı işlemlerinde kullanılır)
n-bitlik işaretli-ikilik *B* sayısının *negatifi* şöyle elde edilir

$$-B = 2^n - B .$$

İşaret-büyüklük ve işaretli-ikilik sayı biçimlerinde negatif sayıların en-sol-biti (*MSB*) 1 -dir (yani, bütün pozitif sayılarda en sol bit 0 -dır). Bu bit sayının *işaret-biti* diye adlandırılır.

Örnek : 7-bit yazmaçta çeşitli işaretli sayı biçimlerinde +9 ve –9 :

gösterim biçimi \ değer:	-9	9
işaret-büyüklük gösterimi	1001001	0001001
2-lik tümleyen gösterimi	1110111	0001001
32-eğilimli gösterimi	0010111	0101001

Eksi işaretli ikilik sayıların değerini bulurken işaret konum-ağırlığı negatif alınır:

Örnek 4-1: Işaretli ikilik sayıların ondalık değerleri

$111\ 0111_2 = -1\times2^6 +1\times2^5 +1\times2^4 +0\times2^3 +1\times2^2 +1\times2^1 + 1\times2^0$
$= -64 + 32 + 16 + 0 + 4 + 2 + 1 = -9$.

$1000000_2 = -1\times2^6 = -64$.

$10010.10_2 = -1\times2^4 +1\times2^2 +1\times2^{-1} = -16+2+1/2 = -13.5$

İkilik tümleyenini bulma işlemi işaret değiştirmeye denktir, ve sıklıkla *olumsuzlama* işlemi diye anılır. *n*-bitlik *B* sayısının *2-lik tümleyeni* şöyle tanımlanır:

$-B = 2^n - B$.

(2^n-1) sayısındaki *n* bitin hepsi "bir" değerindedir, ve

$B^- = (2^n-1) - B$

ifadesinde *B'*, *B* -nin *bit bite tümleyeni*, diğer deyişle *1-lik tümleyeni*dir. Böylece B -nin *2-lik tümleyeni*

$-B = (2^n-1) - B + 1 = B' + 1$. olur.

B nin en sağdaki basamakları sıfırsa bir artırma işlemininde en sağ bitlerden elde (carry) çıkar. Eldeyi önlemek için, en sayının sağdaki sıfır basamaklarını aynen kopyalayıp, kalan kısmının 2-lik tümleyeni alınmalıdır.

Örnek: 2-lik tümleyeni eldeleri ilerletmeden bulma

$1110111_2 = -0001001_2$	$1101100_2 = -0010100_2$
2-lik-tüml. | kopyalama yok	2-lik-tüml. | kopyala

Örnek: 16-lık sayıların 16-lık tümleyenini eldesiz bulma

$A2C0_{16} = -5D40_{16}$	$01208_{16} = -FEDF8_{16}$
16-lık-tüml. | kopyala	16-lık-tüml. | kopyalama yok

Örnek: 4-bit yazmaçta işaretli-ikilik-değerler:

0000 = 0	0001 = 1	0010=2	0011=3
0100 = 4	0101 = 5	0110=6	0111=7
1000 =–8 (not)	1001 = –7	1010=–6	1011=–5
1100 = –4	1101 = –3	1110=–2	1111=–1

4-bit işaretli sayıların konum ağırlıkları -2^3, 2^2, 2^1, 2^0 dir. İşaret bitinden ötürü 1000_2 sayısı 8 değil negatif sayıdır. 1000_2 tanım gereği $-2^{n-1} = -2^3 = -8$ -e karşılık gelir.

n-bit gösterimde sayıların üst ve alt limitleri:

	işaret-büyüklük	2-lik tümleyen
Maksimum pozitif değer	$2^{n-1} - 1$	$2^{n-1} - 1$
Minimum negatif değer	$-(2^{n-1} - 1)$	-2^{n-1}

MIPS -te, her sözcük 32 bittir: (yazmaç genişliği n =32)

MSB						LSB
bit-31	bit-30	bit-29	....	bit-2	bit-1	bit-0

Bu yüzden, işaretli ikilik biçimi kullanarak bir MIPS yazmacı -2^{31} den $2^{31}-1$ -e kadarki işaretli tamsayıları gösterebilir.

n-bitik işaretli bir **A** = $a_{n-1}\ a_{n-2}\ \ldots\ a_0$ sayısının büyüklüğü **|A|** şöyle tanımlanır.

$$|\mathbf{A}| = \begin{cases} A, & A \geq 0 \text{ ise;} \\ -A, & A < 0 \text{ ise.} \end{cases}$$

Sayı pozitif ise, doğrudan bit-30... bit-0 sayının büyüklüğünü verir.

A sayısı negatifse ikilik tümleyeni pozitiftir. Negatif **A** sayısının büyüklüğünü **A** sayısının 2-lik tümleyeni olan **–A** verir. Bir sayının 2-lik tümlenmesine o sayının "eksilenmesi" de denir.

Bellekte fiziksel adresler genellikle 0 -dan başlar, ve en yüksek adrese doğru artar. Negatif sayılar fiziksel adreslemede kullanılmaz. Çoğu programlama dillerinde bellek adreslenmesi amacıyla sadece pozitif tamsayılarla işlem yapmak üzere işaretsiz tamsayı gösterimini kullanır.

C-dilinde “**unsigned int**” veri tipi sadece negative olmayan tamsayıları betimler, "**int**" veri tipi ise işaretli tamsayıdır (negatif yada pozitif olabilir). MIPS -teki küçükse-bir-yap komutları **slt** ve **slti** iki biçimdir.

a) **slt** ve **slti,** *işaretli (signed)* tamsayılarla çalışır
b) **sltu** ve **sltiu,** *işaretsiz (unsigned)* tamsayılarla çalışır

Örnek 4-2:

Diyelim ki

bit:	31	30				1	0
$16=	1	1	..	..	..	1	1

ve

bit:	31	30				1	0
$17=	0	0	..	..	..	0	1

olsun.

slt $8, $16, $17 ve **sltu $9, $16, $17**

komutlarının çalışmasından sonra **$8** -deki değer nedir?

<< çözüm:

slt yazmacı işaretli biçimde görür, böylece **$16**= –1 ve **$17**= 1. –1<1 doğru olduğundan **r8** birlenir
$8 ← 0000 0001 h

sltu yazmacı işaretsiz biçimde görür, yani, **$16**= 2^{32}–1 = 4 294 967 295, ve **$17**= 1 olduğundan **$16**<1 sağlanmaz, **r8** sıfırlanır. **$8** ← 0000 0000 h

>>

4.1.3 Toplama ve Çıkarma

Yazmaç boyutunun n-bit olduğunu düşünün. *Artan* ile *eklenen*i *toplama* işlemi en sağdaki bitten başlar, ve her konumun *elde*si bir sonraki konuma eklenir. Elde doğru ilerletilirse bu yöntemle toplama her sayı sisteminde doğru toplamı verir.

Örnek: n=8 bit için, $(45 + 35)_{10}$ şunu verir

		Ondalık	**İkilik**	**16lık**
		1	elde ilerlemesi 1 1 1 1 1	1
	artan	45	0010 1101	2D
+	*eklenen*	+ 35	+ 0010 0011	+ 23
=	*toplam*	= 80	= 0101 0000	= 50

Çıkarma işlemi iki işlenen arasındaki *farkı* verir,

fark = çıkartılan– çıkan.

Çıkanın 2-lik tümleyenini kullanarak farkı toplama yaparak ta elde edebiliriz. Çıkanın 2-lik tümleyeni kullanıldığından toplam fazladan bir elde verir.

4-rakamlı 10 -luk tümleyen –35 = 10000 – 35= 9965		– 0010 0011 in işaretli ikilik biçimi (=1101 1101) 2-lik tümleme işlemiyle bulunur.		
Ondalık		İkilik	2-lik tümleyenle	Onaltılık
0045	0045	0010 1101	0010 1101	**2 D**
– 0035	+ 9965	– 0010 0011	+ 1101 1101	+ **DD**
= 0010	= 1 0010	= 0000 1010	= 1 0000 1010	= **1 0A**

Dikkat edilirse ondalık sayının 10 -luk tümleyeni sayının eksilenmişidir. İkilik sayılarda, sayıyı 2-lik tümleyerek eksileriz.

Örnek: $(– 29 – 90)_{10}$

Ondalık	**İkilik**
$-29_{10} = 10000_{10} - 29_{10} = 9971_{10}$ $-90_{10} = 10000_{10} - 90_{10} = 9910_{10}$	

– 0029	9971	– 0001 1101	+ 1110 0011
– 0090	+ 9910	– 0101 1010	+ 1010 0110
= –119	= 1 9881	= –0111 0111	= 1 1000 1001
9881 negatiftir ve değeri $-10000_{10}-9881_{10} = -119_{10}$ dur.		**1000 1001 negatiftir ve değeri $-1\ 0000\ 0000_2 - 1000\ 1001_2$ $= -256_{10} - 137_{10} = -119_{10}$ dur**	

	Onaltılık	
2-lik tümleyenli işaretli biçimde 89_{16}, büyüklüğü 77_{16} olan negatif sayıdır (kısaca -77_{16}).	– 1 D + – 5 A = – 7 7	E 3 + A 6 = 1 8 9

4.1.4 İşaretli-ikilik toplamada sonuç taşması

İşaretli ikilik toplamada sonucun yanlış işaret vermesi, doğru sonucun yazmaç genişliğine sığmadığını gösterir.

Örnek: 8-bit yazmaçta işaretli ikilik biçimde $(80+60)_{10}$ şunu verir

Ondalık	İkilik	16lık
80	0101 0000	50
+ 60	+ 0011 1100	+ 3C
= 140	= 1000 1100	= 8C

Toplama sonucu işaret biti taştığından hem toplanan hem eklenen pozitif olmasına karşın toplam negatif çıktı.

Örnek: 145 sayısı 8-bitle işaretli ikilik sayı biçimi kullanılarak yazılacak en büyük pozitif sayı sınırını aştığından $(145\text{-}23)_{10}$ işlemi 8-bit yazmaçta bu sayı biçiminde gerçekleştirilemez.

Örnek: 8-bit yazmaçta işaretli ikilik sayılarla $(-40 + -90)_{10}$ işlemi şunu verir:

Ondalık	**İkilik –(40+90)**		**Onaltılık**
elde 1	1 1 1 1	1	1
– 40	– 0010 1000	+ 1101 1000	D 8
– 90	– 0101 1010	+ 1010 0110	+ A 6
= –130	= – 1000 0010	= ~~1~~ **0**111 1110	= ~~1~~ 7 E

Hem artanın hem eklenenin işareti eksiyken toplam artı işaretli olduğuna göre sonuç işaret bitine taşıp işareti bozmuştur. –130 olan toplam işaretli ikili biçimde 8-bite sığmadığından taşma oluşur.

İşaretli toplamalarda, eğer artan ile eklenen farklı işarete sahipse, hiçbir durumda taşma olamaz. Taşma sadece toplanan iki sayı da aynı işarettense oluşabilir.

4.1.5 Taşmayı algılama

Iki n-bit sayı toplanınca sonucu düzgün yazabilmek için n+1 bit gerekebilir. Pozitif sayıların toplamı negatif olduğunda, yada negatif sayıların toplamı pozitif olduğunda taşma vardır.

Z = **A** + **B** toplamının taştığı aşağıdaki Boole işlevi ile **A**, **B**, ve **Z** -nin işaret bitleri kullanılarak anlaşılır.

$$\textit{Taşma} = \overline{\mathbf{a}_{MSB}}\ \overline{\mathbf{b}_{MSB}}\ \mathbf{z}_{MSB} + \mathbf{a}_{MSB}\ \mathbf{b}_{MSB}\ \overline{\mathbf{z}_{MSB}}\ ;$$

burada $\mathbf{a}_{MSB}$, $\mathbf{b}_{MSB}$, ve $\mathbf{z}_{MSB}$ sırasıyla **A**, **B**, ve **Z** -nin en-sol-bitleridir (işaret-biti), ve çizgiler mantıksal tümleyeni gösterir.

Bir aritmetik işlemde taşma durumunda MIPS ilgili "*kuraldışı yordam*"ını çağırır. Kuraldışı, programın normal çalışmasının bir iç koşul nedeniyle yarıda kesilmesidir. Kuraldışına sapılırken onu doğuran komutun adresi kuraldışı program sayacı "*exception program counter EPC* denen özel bir yazmaça saklanır. *EPC* -nin içeriğini genel amaçlı yazmaçlara aktarmak mümkündür.

İşaretsiz tamsayılar çoğunlukla sınırları önceden bilinen bellek adreslerinde kullanıldığından MIPS işaretsiz tamsayılı işlemlerde taşmaları dikkate almaz.

MIPS aritmetik komutları iki çeşittir:

a) Taşma durumunda kuraldışına neden olanlar: **add**, **addi**, **sub**

b) Taşma durumunda kuraldışına sapmayanlar: **addu**, **addiu**, **subu**

Kuraldışı, CPU içinde bir sinyalden kaynaklanan bir olay durumundaki yordam çağrısıdır. C programlama dili taşma durumunda kuraldışı oluşturmaz. MIPS C derleyicisi kodu herzaman işaretsiz aritmetik komutlarından (**addu**, **addiu**, ve **subu**) oluşturur. ADA programlama dili

işlenenlerin tipine bağlı olarak kuraldışılı yada kuraldışısız komutları seçebilir. ADA -da kuraldışı yordamları yazmak mümkündür.

4.2 Mantıksal İşlemler

Mantıksal işlemler mantıksal kaydırma, ve *AND*, *OR* vs. gibi mantık işlemlerinden oluşur. R-tipi komutlarla geçekleştirilirler, ve ayrıca bazılarının I-tipi biçimleri de vardır.

4.2.1 Mantıksal ve Aritmetik Kaydırma İşlemleri

Mantıksal kaydırma işlemlerinde, varış ve kaynak yazmaçlarının yanısıra kaç bit kayacağını belirleyecek anlık değer gerekir.

sola-mantıksal-kaydır (shift-left-logical)

sll $dest,$source,sa

sağa-mantıksal-kaydır (shift-right-logical)

srl $dest,$source,sa

sa bit olarak kaydırma miktarını belirtir.

Mantıksal kaydırmada boşalan bitlere 0 dolar.

Örnek 4-3: Diyelim ki

\$16= 0000 0000 0000 0000 0000 0000 0000 1101_2= $0000000D_{16}$.

sll $10, $16, 8

komutu $16 -yı 8 bit sola kaydırır, ve sonucu $10 -a koyar.

Yürütmenin ardından, r10 da şu değer kalır.

\$10= 0000 0000 0000 0000 0000 1101 0000 0000_2= $00000D00_{16}$.

sll $10, $16, 8 ifadesinin ondalık sayıyla makine kodu:

opc (6-bit)	rd (5-bit)	rs (5-bit)	rt (5-bit)	sa (5-bit)	fn (6-bit)
0	0	16	10	8	0

İşaretsiz tamsayılarda **srl** komutu sayıyı 2^{sa} değerine bölmeye denktir. **srl** komutu işaretli sayılarla kullanıldığında kayan bitlerin yerine sıfır geldiğinden işaret biti değer değiştirebilir. Sayıyı sağa kaydırırken en sol biti işaret bitinden kopyalayan kaydırma işlemi işaret bitini korur. İşaret bitini koruyan sağa kaydırma işlemine aritmetik sağa kaydırma denir ve **sra** ile anımsanır.

aritmetik-sağa-kaydır (shift-right-arithmetic)

sra $dest,$source,sa

sa bit olarak kaydırma miktarını belirtir.

Örnek: Diyelim ki
$16= – 96_{10} = FFFF $FFA0_{16}$ olsun.
sra $10, $16, 4

opc (6-bit)	rd (5-bit)	rs (5-bit)	rt (5-bit)	sa (5-bit)	fn (6-bit)
0	0	16	10	4	3

komutu **r16** daki –96 değerini işaret bitini koruyarak 4 bit sağa kaydırır. Bu işlem **r16** daki işaretli ikilik sayıyı 16 ya bölmeye denktir. Komut işlenince
$10= FFFF $FFFA_{16}$ (= – 96_{10} / 16 = $–6_{10}$) olur.

Sola kaydırmalarda hem aritmetik hem mantıksal komut boşalan bitlere sıfır yerleştirir. MIPS te *Aritmetik-sola-kaydır* (**sla**) komutu yoktur. **sla** sözde-komut olarak gerçeklenir ve çevirici tarafından **sll** komutuna dönüştürülür.

4.2.2 Bit-bite VE ve VEYA

VE (*AND*) ve *VEYA* (*OR*) işlemleri R-tipi (**and, or**) ve I-tipi (**andi, ori**) komutlarla gerçekleştirilir .

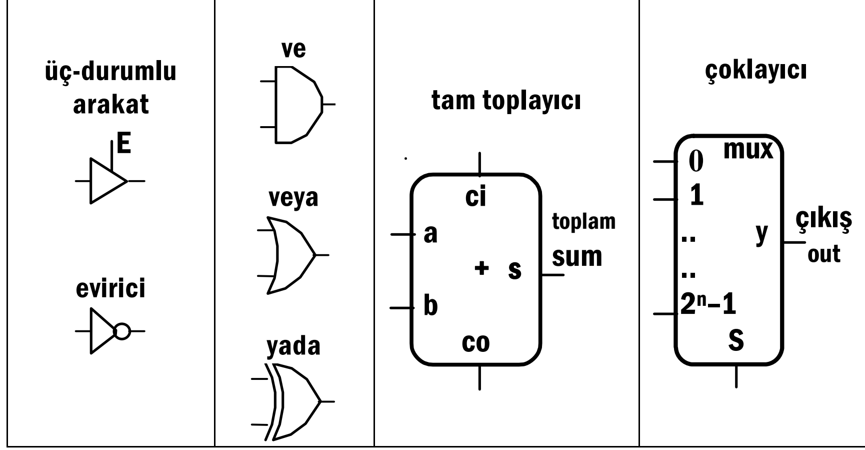
Örnek 4-4:
r9 ve **r10** yazmaçlarında
$9=0000 0000 0000 0000 0011 1100 0000 0000_2 =$00003C00_{16}$.
$10=0000 0000 0000 0000 0000 1101 0000 0000_2 =$00000D00_{16}$ olsun.

and $8, $9, $10 # $8 ← $9 *ve* **$10**
komutu işlendikten sonra **r8**, **$10** ile **$9** -un bit-bite *ve* -sini içerir:
$8 =0000 0000 0000 0000 0000 1100 0000 0000_2 =$00000C00_{16}$

Örnek 4-5
r9 ve **r10** aynı ilk değerleri içeriyor olsun:
$9 =$00003C00_{16}$, $10=$00000D00_{16}$.
or $8, $9, $10 # $8 ← $9 VEYA $10
komutundan sonra **r8** deki değer **$10** ve **$9** -un bit-bite *veya* -sıdır:
$8 =0000 0000 0000 0000 0011 1101 0000 0000_2 =$00003D00_{16}$

4.3 Aritmetik Mantık Birimi (ALU)

Aritmetik Mantık Biriminin (*ALU*) yapı taşları ve, veya geçitleri, tam toplayıcılar, arakat ve eviricilerle çoklayıcılardır.



Üç-durumlu (tri-state) arakat giriş sinyalini yalnızca seçilmiş olduğunda çıkışa iletir.

Bir *evirici* çıkışına giriş değerinin değilini iletir. Değilleme işlemi mantık ifadelerinde bir üst çizgi yada kesme işaretiyle gösterilir, yani,

$\bar{a} = a' = 0$ ise $a=1$; ve $\bar{a} = a' = 1$ ise $a=0$.

Bir *VE* (*AND*) geçitinin çıkışı yalnızca tüm girişleri 1 ise 1 -olur. Mantık ifadelerinde, "*VE*" çarpma gibi gösterilir:

a VE b = *a AND b* = *a* . *b* = *a b*

Bir *VEYA* (*OR*) geçitinin girişlerinden en az biri 1 olduğunda çıkışı 1 -olur. Mantık ifadelerinde, veya toplama işaretiyle gösterilir:

a VEYA b = *a OR b* = *a* + *b*

Bir *YADA* (*XOR*) geçitinin çıkışı sadece girişteki 1 -lerin toplamı tek ise 1 -dir. Mantık ifadelerinde daire içinde bir toplama işareti, "⊕", *XOR* işlemini gösterir:

a YADA b = *a XOR b* = $a \oplus b = a' b + a b'$

*Tam toplayıcı*nın üç girişi (*a*, *b*, *ci*) ve iki çıkışı (*sum*, *co*) vardır. *a*, ile *b* artan ve eklenen bitlerini, *ci* (carry-input) elde girişini, *sum* toplam çıkışını, *co* (carry-output) ise elde çıkışını gösterir. Doğruluk tablosu aşağıdaki gibidir.

Tablo 4-1 tam-toplayıcı doğruluk tablosu

girişler			çıkışlar		açıklama
a	*b*	*ci*	*co*	*sum*	
0	0	0	0	0	$0+0+0 = 00_2$
0	0	1	0	1	$0+0+1 = 01_2$
0	1	0	0	1	$0+1+0 = 01_2$
0	1	1	1	0	$0+1+1 = 10_2$
1	0	0	0	1	$1+0+0 = 01_2$
1	0	1	1	0	$1+0+1 = 10_2$
1	1	0	1	0	$1+1+0 = 10_2$
1	1	1	1	1	$1+1+1 = 11_2$

toplam (*sum*) ve *elde* çıkışı (*carry out*: *co*) için Boole işlevi şöyledir:

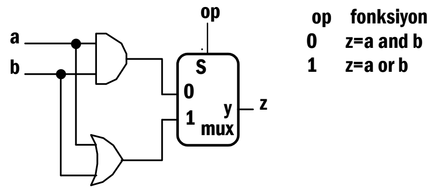
$$co = (a \cdot ci) + (b \cdot ci) + (a \cdot b)$$

$$sum = (a \cdot b' \cdot ci') + (a' \cdot b \cdot ci') + (a' \cdot b' \cdot ci) + (a \cdot b \cdot ci)$$

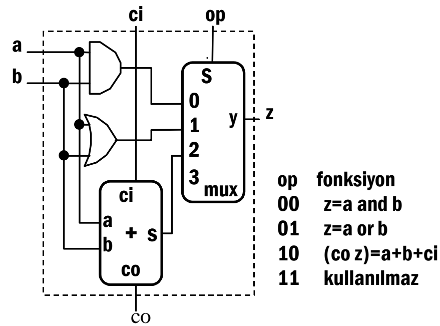
$$= a \oplus b \oplus ci$$

4.3.1 Tek-bitlik ALU

Şekil 4-1 -de *VE* ve *VEYA* mantık işlemleri yapan 1-bitlik-*ALU* görülüyor.



Şekil 4-1 VE ve VEYA işlemleri yapan tek-bitlik ALU

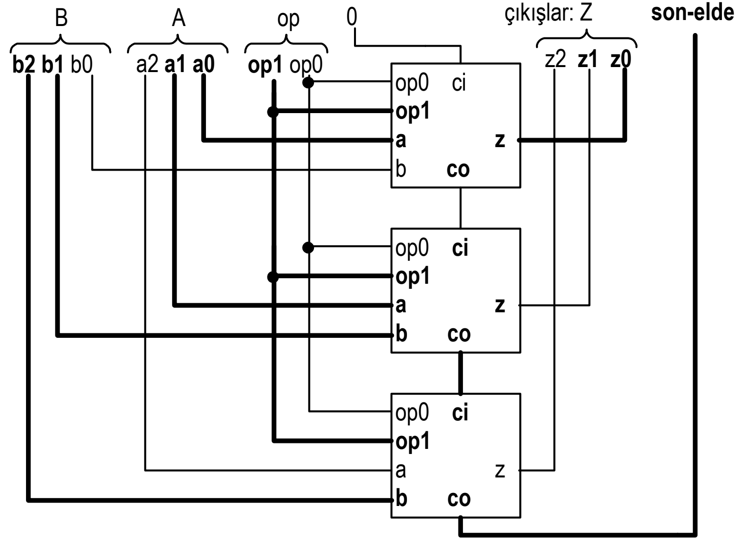


Şekil 4-2 topla, ve, veya işlemleri yapan bir-bitlik-ALU.

Şimdi, toplamayı dahil edelim. Çıkışta daha büyük bir çoklayıcı kullanmalı, ve elde-girişi ile elde-çıkışı yollarını düşünmeliyiz.

Şekil 4-2 -de gösterilen bir-bitlik-ALU biriminin beş girişi ve iki çıkışı vardır. Birimin işlevini betimleyen işlem denetim girişi '*op*' iki hattan, op_0 ve op_1 dan oluşur.

Tek-bitlik-ALU birimleri bit-bite işlemleri (*ve*, *veya*, ve *n-bit-toplama*) yapmak için uç uca zincir gibi birleştirilebilir. Sağ-bitin işlenmesinden oluşan elde-çıkışı, bir soldaki bitin elde-girişine bağlanır. Bu bağlantıları aşağıdaki örnekte üç bitlik veri genişliği için görüyoruz.



Şekil 4-3 toplama yapan 3-bitlik-ALU, **A**=011_2 ve **B**=110_2, kalın çizgiler 1 durumunu gösteriyor.

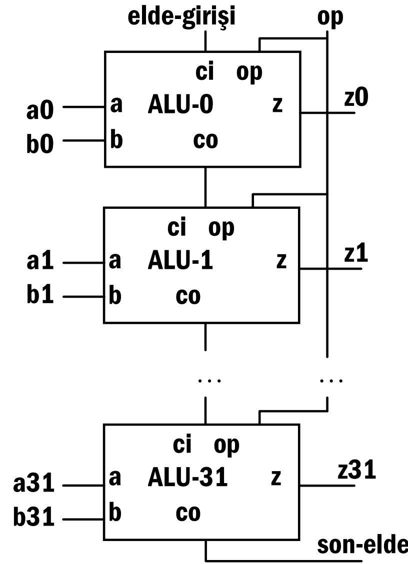
Örnek 4-6*:* Şekil 4-3'te bir-bitlık-ALU -lardan oluşan bir 3-bitlik ALUda, *A* ile *B* sayılarının toplanışı gösterilmektedir. Bu örnekte
A= 011_2 ve
B= 110_2 dir.

Elde:	1 1
	011
+	110
=	1 001

Şekil 4-3, te 1 ileten hatlar kalın çizgiyle gösterilmiştir.

4.3.2 32-bitlik ALU

32-bitlik ALU, 32 tane 1-bitlik-ALU birimi zincirinden oluşur: 32 bit ALU -da, her bitin elde-çıkışı bir sonraki bitin elde-girişine bağlanır. Bu çeşit devrelere yayılan-eldeli devre (ripple-carry) denir. En kötü durum *A* = $(11.....1)_2$ sayısına *B*= $(00......01)_2$ toplandığında oluşur. Devrenin yayılan-eldeli düzeni nedeniyle ilk birimin elde çıkışı 32 birim eldeden eldeye yayılır. Yayılma zaman alır, ve an-uyumlu uygulamalarda, ALU için minimum saat süresini belirler. Yayılan-eldeli gecikmeyi önleyecek çözümlerin bazılarını ileride göreceğiz.



Şekil 4-4 32-bit ALU

4.3.3 Çıkarma işlemi

A–B çıkarmasını toplayıcıda yapmak için, çıkan *B* yi eksilememiz gerekir. *B* nin eksilenmişi bu sayının 2-lik tümlenmişidir. *n*-bitlik *B* sayısı, bit-bite değili olan *B*' değeri bir artırılarak eksilenebilir :

$$-B = B' + 1 = (2^n - 1) - B + 1 \quad .$$

Örnek: 8-bitlik $B = 44 = 0010\ 1100_2$ sayısını düşünün. *B'*, ve *–B* sayılarını bit-bite değilleme ve bir artırma işlemleriyle bulun.

<<

n =8, (8-bitlik sayılar)	1111 1111	1101 0011
2^n = $1\ 0000\ 0000_2$	– 0010 1100	+ 0000 0001
2^n–1 = $1111\ 1111_2$	*B*' = 1101 0011	– *B* = 1101 0100
	B' =255 – 44 =211	–*B* = *B*' +1 = 212

\>>

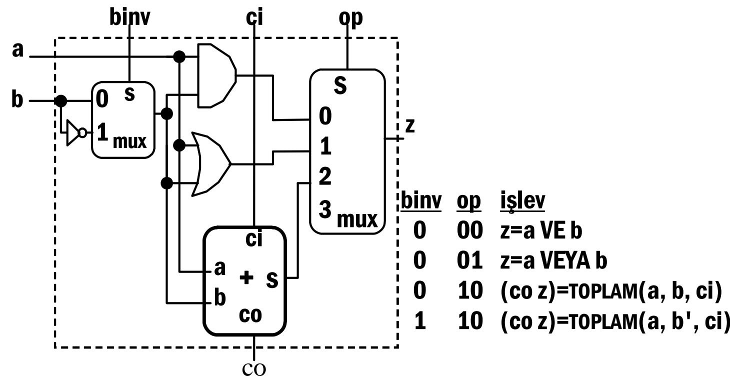
Tek-bitlik-ALU -muzda *B* nin değili için evirici kullanabiliriz. *n*-bitlik 2-lik tümleyen için *B'* sayısına bir eklemek gerekir. Bu toplama n-bit-ALU -nun ilk elde girişi kullanılarak yapılabilir.

Örnek 4-7:
$B = 44 = 0010\ 1100_2$ -yi $A = 50 = 0011\ 0010_2$ -dan 2-lik tümleyenini, yani elde-girişi ile bit-bite değilini kullanarak çıkarın.
<<

A = 0011 0010	elde 1 1 1 1	elde 1 1 1 1 1 1 1
B = 0010 1100	A 0011 0010	0011 0010
B' = 1101 0011	+ ($-B$) 1101 0100	+ (B') 1101 0011
$-B$ = 1101 0100	$A-B$= 1 0000 0110	$A-B$ = 1 1101 0110
A=50; B=44;		elde-girişi
B'=211; $-B$=212	$A-B$ = 50 – 44 = 6	$A-B = A + B' + 1 = 6$

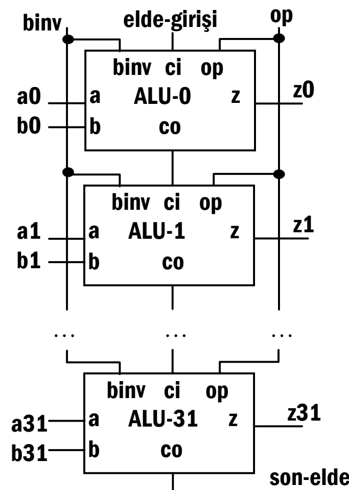
>>

Bir-bitlik-ALU -nun ayrıca 2-lik tümleme ile çıkarma yapmak üzere değiştirilmiş hali Şekil 4-5'de gösterilmiştir.



Şekil 4-5 Çıkarma yapabilen bir-bit-ALU.

Geliştirilen bir-bitlik-ALU -da üç denetim girişi vardır: *binv*, *op1*, ve *op0*. 32-bitlik-ALU, bir-bitlik-ALU -ların 32 -si birleştirilerek oluşturulur. Bu 32-bitlik-ALU Tablo 4-2'de betimlenmiş işlev denetim girişleri kullanılarak *A* ve *B* işlenenleri üzerinde mantıksal-ve (**AND**), mantıksal-veya (**OR**), 32-bit toplama (**ADD**), ve 32-bit çıkarma (**SUB**) işlemlerini yapabilir.

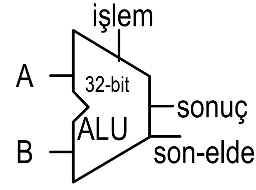


Şekil 4-6 Çıkarma yapmak için 32-bit ALU

Dikkat edilirse bütün işlevler için *binv* ve *c-in* girişleri tümüyle aynıdır, ve tek bir sinyale birleştirilebilirler. *binv* ve *ci* birbirine bağlandığında *bnegate* olarak adlandırılır, ve işlem denetim girişlerinden biri sayılır. Böylece op sinyalimiz 2-bitten 3-bite çıkar. 32-bitlik-ALU tümüyle aşağıdaki ALU sembolüyle gösterilir:

Tablo 4-2 32-bit-ALU -nun denetim sinyalleri

binv	c-in	op	işlev
0	0	00	AND
0	0	01	OR
0	0	10	ADD
1	1	10	SUB



Şekil 4-7 32-bit ALU

4.3.4 Küçükse-Bir-Yap (Set on Less Than) İşlemi

Küçükse-bir-yap (*Set-on-less-than* **SLT**) komutunda, ALU -muzun iki işleneni karşılaştırıp ve sonuç çıkışını ya sıfır yada bir yapması gerekir.

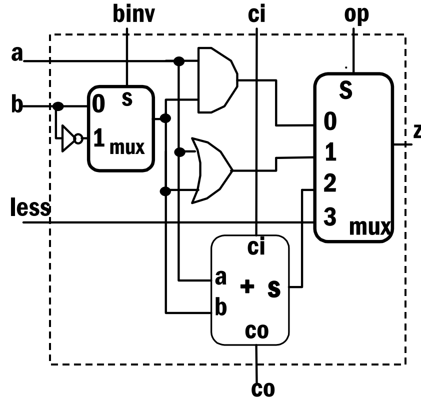
Sonucun alabileceği iki değeri yazarsak bu iki değer arasında sadece bir bit fark olduğunu görürüz.

A<*B* ise 1 = 0b 0000 0000 0000 0000 0000 0000 0000 0001 olur,
değilse 0 = 0b 0000 0000 0000 0000 0000 0000 0000 0000 .

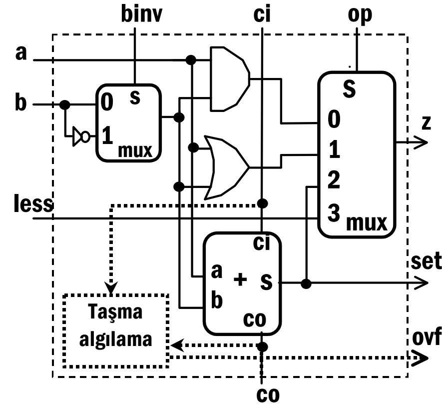
31-1 bitler herzaman 0 -dır. *A* < *B* ise en sağ bit bir yapılır, değilse en sağdaki bit sıfır yapılır.

Son çoklayıcının kullanılmayan girişini küçük (*less*) girişi olarak adlandırıp bu girişi **SLT** işlemi için kullanacağız. Sonuç bitini sıfırlamak üzere en sag biriminki hariç tüm birimlerin *less* girişi sıfıra bağlıdır. Sadece en sag birimin *less* girişi karşılaştırmanın sonucuna gore değiştirilir.

A ile *B* çıkarma işlemi yapılarak karşılaştırılır. Eğer *A*<*B* ise, *A*–*B* pozitif sayıdır ve işaret biti sıfırdır. Ancak *A*–*B* çıkarma işlemi sonucu ALU çıkışına ulaşmaz. Bu yüzden, en sol bitin toplayıcı çıkışının çoklayıcıya bağlandığı noktadaki sinyali kullanabilmek üzere dışarı çıkarmak gerekir. Bu çıkışa *set* çıkışı diyeceğiz.



Şekil 4-8 herhangi bir tek-bitlik-ALU birimi için SLT değişikliği



Şekil 4-9 En sol ALU-biti birimi için SLT değişikliği ve taşma algılaması

ALU31 -e ayrıca taşma algılaması ekleyebiliriz. Işaretli toplama işlemi taşma algılamasında, taşma (*ovf*) Boole işlevini bulmak için işlenenlerin ve sonucun işaret bitleri gerekir:

$$ovf = \overline{a}_{msb}\,\overline{b}_{msb}\,z_{msb} + a_{msb}\;b_{msb}\,\overline{z}_{msb}\ .$$

Ama hatırlarsanız çıkarma işleminde b_{31} bitini değillemiştik, ve 2-lik tümleyeni bir eklenerek bulunduğundan işlem sırasında ilerleyen eldeyle işaret bitinin değişmesi de mümkündür. 32-bit ALU -da taşma sinyali a_{31}, b_{31}, *bnegate*, ve z_{31} kullanmak yerıne yalnızca ci_{31} elde-girişi, ve co_{31} elde-çıkışı kullanılarak ta elde edilebilir.

Şimdi ($a = a_{msb}$, $b = b_{msb}$, $z = z_{msb}$) sinyallerinin bütün kombinasyonları için $ci = ci_{msb}$ ile $co = co_{msb}$ -nin durumlarını yazıp taşma oluşturan durumlarda (ci, co) arasında bağıntı arayalım.

$co = (a\,.\,ci) + (b\,.\,ci) + (a\,.\,b)$

$z = (a\,.\,b'.\,ci') + (a'\,.\,b\,.\,ci') + (a'.\,b'.\,ci) + (a\,.\,b.\,ci)$

Tablo 4-3 Taşma koşulu için doğruluk tablosu, ve karşılık gelen elde giriş ve çıkış sinyalleri.

a_{msb}	b_{msb}	z_{msb}	ovf	ci_{msb}	co_{msb}
0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	0	1	1
0	1	1	0	0	0
1	0	0	0	1	1
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	0	1	1

Görüyoruz ki (ci_{msb},co_{msb},z_{msb}) hiçbir zaman 0,1,1 veya 1,0,0 olamıyor. Taşma (overflow) koşulunun Karnough çiziminde bu iki koşul için farketmez anlamına X yerleştiriyoruz.

overflow taşma	ci_{msb},co_{msb} 00	01	11	10
z_{msb}=0	0	1	0	X
z_{msb}=1	0	X	0	1

$$taşma = \overline{ci}_{msb}\ co_{msb} + ci_{msb}\ \overline{co}_{msb}$$
$$= ci_{msb} \oplus co_{msb}$$

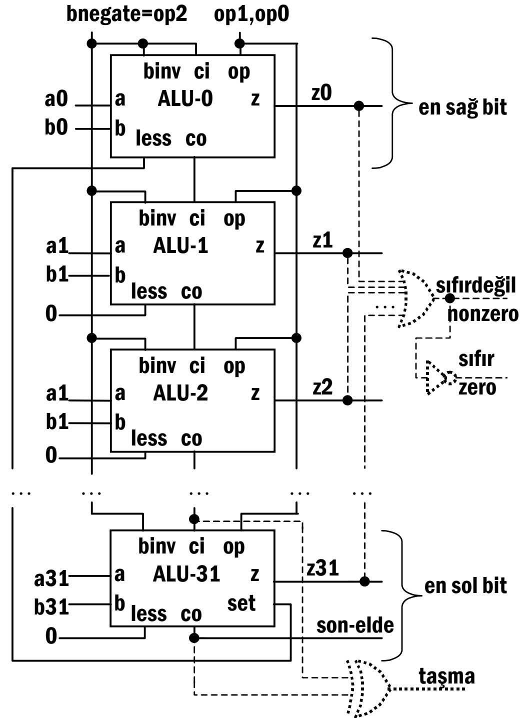
4.3.5 SLT ve Koşullu Dallanmayı destekleyen 32-bit ALU

Şimdi, bütün parçaları biraraya getirip **AND**, **OR**, **ADD**, **SUB** işlemlerinin yanısıra **SLT** işlemini de yapabilen 32-bitlik bir ALU oluşturabiliriz.

32-bit ALU -nun son sürümü iki tip tek-bitlik-ALU birimi kullanır. Bunların giriş çıkış tabloları şöyledir.

MSB birimi	MSB hariç tüm birimler
Girişler: a: işlenen-bit b: işlenen-bit küçük: SLT-ye ilişkin binv: b işlenenini değille ci: elde girişi op: işlev girişi Çıkışlar: z: sonuç biti co: elde çıkış biti set: SLT için işaret ovf: taşma	Girişler: a: işlenen-bit b: işlenen-bit küçük: SLT-ye ilişkin binv: b işlenenini değille ci: elde girişi op: işlev girişi Çıkışlar: z: sonuç biti co: elde çıkış biti

beq (yada bne) koşullu dallanma komutu *rs* ve *rt* yazmaçları birbirine eşitse (yada değilse) verilen adrese sapar. Tüm sonuç bitlerini değerlendiren bir *OR* ve evirici geçit $Z = A - B$ ALU-sonucundan *sıfırdeğil*, ve *sıfır* sinyali oluşturur. Sıfır-bulucu devresi genelde ALU - nun bir parçası olarak düşünülür. *sıfır* (*zero*) sinyali sadece A - B sonucu sıfırsa birdir. Bu yüzden ALU da koşullu dallan (**beq**) yürütülürken, denetim birimine "*zero*" sinyalini sağlamak üzere çıkarma yaptırmak gerekir.



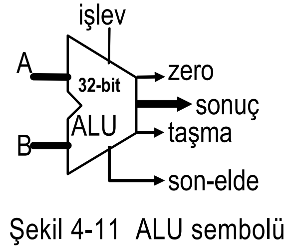
Şekil 4-10 VE VEYA TOPLA ÇIKAR ve SLT işleyebilen 32-bit ALU

32-bit ALU -nun final versiyonu aşağıdaki giriş ve çıkışlara sahiptir.

Girişler:		
a_{31}... a_0	A	işlenen-1 ($rs)
b_{31}...b_0	B	işlenen-2 ($rt)
op_3...op_0	op	işlev denetim kodu

Çıkışlar	
z_{31}...z_0	sonuç
ovf	sonuç taşarsa 1
zero	sonuç sıfırken 1

Özetle, ALU -yu göstermek için aşağıdaki sembolü kullanacagız



Şekil 4-11 ALU sembolü

Tablo 4-4 ALU İşlem kodları

Işlem kodu	işlevi
0 0 0	VE (and)
0 0 1	VEYA (or)
0 1 0	Topla (add)
1 1 0	çıkar (subtract)
1 1 1	küçükse birle (slt)

4.3.6 Önden- Eldeli Toplama:

Her birimin elde-çıkışı *co*, bir sonraki birimin elde-girişi *ci* -ye bağlanmışsa, *gecikerek-yayılan-elde* sonucun durulmasını geciktirir. En kötü durumda, sonuç her *bir-bit-ALU* -nun aktarım gecikmesinden 32 kat daha yavaş durulur. MIPS -te adresler 32-bittir, ve her komut döneminde işlemci adresi yenilemek için ALUyu kullanmak zorundadır. Bu yüzden, bilgisayarın hızını ALU -nun hızı belirler. Bununla beraber, *gecikerek-ilerleyen-eldeyı* hızlandırmak için çeşitli çözümler vardır.

Her tek-bitlik-ALU -nun artan girişini *iki-seviyeli-AND-OR-devresi* ile doğrudan sağındaki bir-bitlik-ALU -ların girişlerinden üreten hızlı bir ALU oluşturulabilir. Bu kısımda, ALU-0 ALU-1, .., ALU-k -nın elde-girişleri ci_0, ci_1, ..., ci_k diye adlandıracağız.

Örnek: *k* -nci elde-girişi için daha önceki birimin elde-çıkışını kullanmadan Boole işlevleriyle şöyle bulabiliriz.

$$ci_1 = a_0 . ci_0 + b_0 . ci_0 + a_0 . b_0 \quad (1)$$

ve

$$ci_2 = a_1 . ci_1 + b_1 . ci_1 + a_1 . b_1 , \quad (2)$$

burada (1) -in sağındaki tüm sinyaller dışarıdan gelen girişlerdir.

(1) -i (2) -de yerine koyup, *elde-girişi-2* (ci_2) için açık ifadeyi doğrudan dış girişler cinsinden bulalım.

$$ci_2 = (a_1 . [a_0 . ci_0 + b_0 . ci_0 + a_0 . b_0]) + (b_1 . [a_0 . ci_0 + b_0 . c_0 + a_0 . b_0]) + (a_1 . b_1)$$
$$= a_1.a_0.ci_0 + a_1.b_0.ci_0 + a_1.a_0.b_0 + b_1.a_0.ci_0 + b_1.b_0.ci_0 + b_1.a_0.b_0 + a_1.b_1 \quad (3)$$

Şimdi, elde-girişi-3 -ü yazabiliriz:

$$ci_3 = a_2 . ci_2 + b_2 . ci_2 + a_2 . b_2 \quad (4)$$

ve (3) -ten ci_2 -yi kullanabiliriz, vb.

ci_{31} -e doğru ilerledikçe ifadeler daha fazla terim içerecektir. İki-seviyeli-mantık kullanarak dış girişlerden ci_k = 1 mi yoksa 0 mı

olduğunu doğrudan bulmak mümkündür, ancak bit sayısı artıkça bu yol pratik olarak imkansız ve son derece pahalı bir yol olur.

Bir diğer yola *önden-eldelı-toplama* denir. Bu yöntem, a_k ve b_k kullanarak elde-girişi ifadesini yazmayı basitleştiren *elde oluşturma* (g_k) ve *elde aktarma* (p_k) terimlerinin tanımlanmasına dayanır.

$g_k = a_k \, . \, b_k$	k -nci birim ci_k dan bağımsız olarak ci_{k+1} yi üretir
$p_k = a_k + b_k$	p_k=1 iken hem de ci_k = 1 ise, k -inci birimin eldesi sonraki birime elde-çıkışı olarak aktarılır, yani, $ci_{k+1} = g_k + p_k . \, ci_k$

k nci birim iki durumda *elde-çıkışı* (ci_{k+1}) verir, ya g_k =1 olursa, yada hem p_k =1 hem ci_k =1 olduğunda.

$$ci_{k+1} = g_k + p_k \, . \, ci_k \qquad (5)$$

Şimdi, her birim için Boole eşitliklerini yazabiliriz:

$$ci_1 = g_0 + p_0 \, . \, ci_0 \, . \qquad (6)$$

$$ci_2 = g_1 + p_1 \, . \, ci_1 \, . \qquad (7)$$

(6) -dan ci_1 -i kullanarak:

$$ci_2 = g_1 + p_1.g_0 + p_1.p_0.ci_0 \qquad (8)$$

$$ci_3 = g_2 + p_2 \, . \, ci_2$$

(8) -den ci_2 -yi kullanarak:

$$ci_3 = g_2 + p_2.g_1 + p_2.p_1.g_0 + p_2.p_1.p_0.ci_0 \qquad (9)$$

Benzer olarak, ci_4 -ü söyle yazabiliriz:

$$ci_4 = g_3 + p_3.g_2 + p_3.p_2.g_1 + p_3.p_2.p_1.g_0 + p_3.p_2.p_1.p_0.ci_0 \qquad (10)$$

Bu eşitlikler 16-bit toplamanın gerçekleştirilebilmesi için bile hala fazla karmaşıktır. Bu nedenle, bütün bir 32-bit adder devresi genellikle *4-bit*-peşin-*elde-bulmalı-toplayıcı* öbeklerin 8 adedi birleştirilerek oluşturulur. Daha üst düzey *elde-aktarma* ve *elde-oluşturma* sinyalleri büyük P ve G ile gösterilir. Bir öbek elde girişine gelen eldeyi elde çıkışına ancak öbekteki tüm bitler elde aktarıyorsa aktarır aktarır.

$$P_0 = p_3 \, . \, p_2 \, . \, p_1 \, . \, p_0 \; ; \qquad (11)$$

$$P_1 = p_7 \, . \, p_6 \, . \, p_5 \, . \, p_4 \; ; \qquad (12)$$

$$P_2 = p_{11} . \, p_{10} \, . \, p_9 \, . \, p_8 \; ; \qquad (13)$$

$$P_3 = p_{15} \, . \, p_{14} \, . \, p_{13} \, . \, p_{12} \, . \qquad (14)$$

Benzer olarak, 4-bitlik bir öbek ancak 4 birimden biri elde oluşturuyorsa ve sağındaki tüm birimler bunu elde çıkışına taşıyorsa artan oluşturur:

$$G_0 = g_3 + p_3 \, . \, g_2 + p_3 \, . \, p_2 \, . \, g_1 + p_3 \, . \, p_2 \, . \, p_1 \, . \, g_0 \; ; \qquad (15)$$

$$G_1 = g_7 + p_7 \, . \, g_6 + p_7 \, . \, p_6 \, . \, g_5 + p_7 \, . \, p_6 \, . \, p_5 \, . \, g_4 \; ; \qquad (16)$$

$G_2 = g_{11} + p_{11} . g_{10} + p_{11} . p_{10} . g_9 + p_{11} . p_{10} . p_9 . g_8$; (17)

$G_3 = g_{15} + p_{15} . g_{14} + p_{15} . p_{14} . g_{13} + p_{15} . p_{14} . p_{13} . g_{12}$. (18)

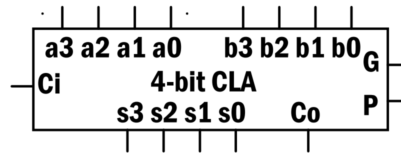
Bu durumda, 16-bit adder için bu üst soyutlama düzeyinde eşitlikler:

$Ci_1 = G_0 + P_0.Ci_0$; (19)

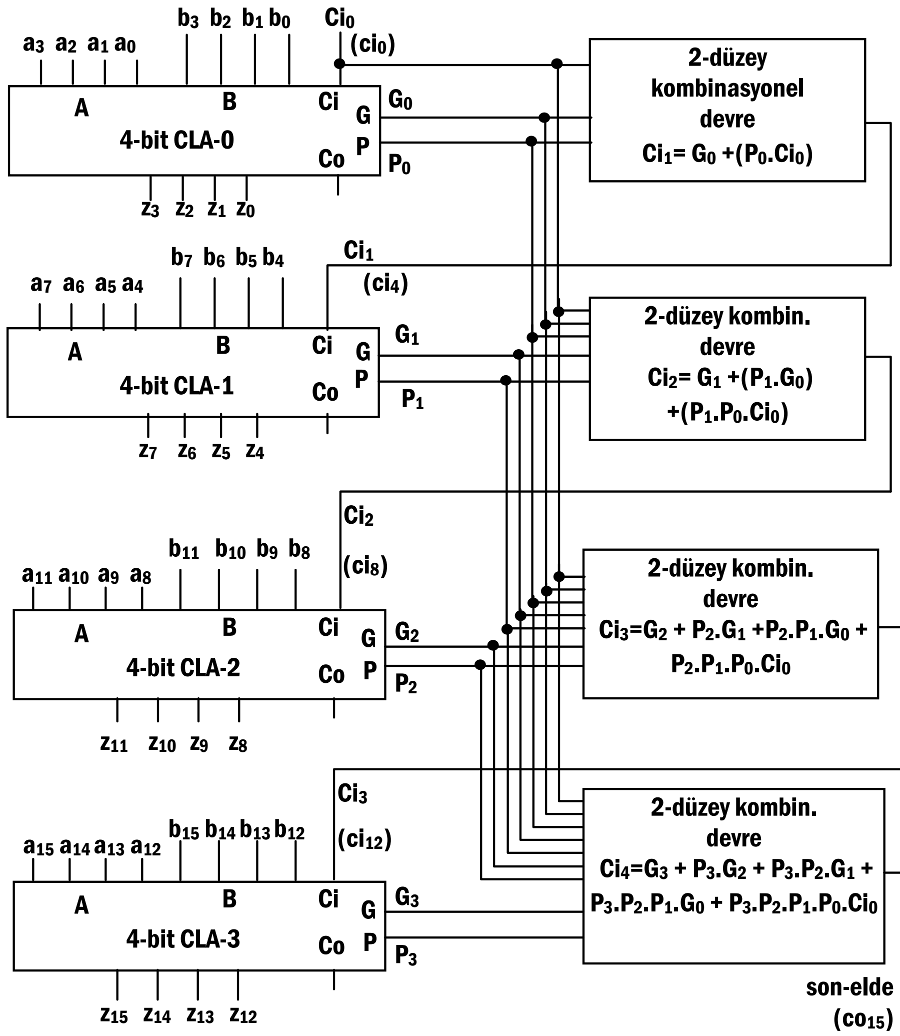
$Ci_2 = G_1 + P_1.G_0 + P_1.P_0.Ci_0$; (20)

$Ci_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.Ci_0$; (21)

$Ci_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0.Ci_0$. (22)



Şekil 4-12 Tipik 4-bit peşin-elde-bulmalı toplama bloğu



Şekil 4-13 4-bit önden-eldeli toplayıcı öbeklerden oluşmuş 16-bitlik toplayıcı

4.3.7 Çözümlü Problemler

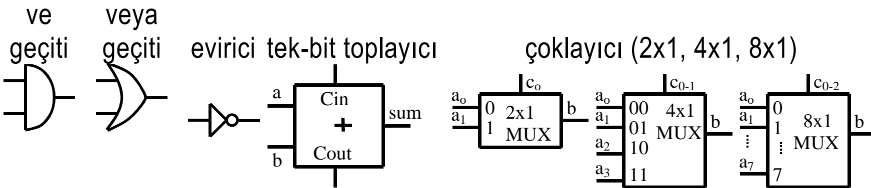
Q1) Aşağıda işlenen girişleri *a* ve *b* , çıkışları C_{out} ve *r* , diğer gerekli giriş/çıkışları *less* ve *set* olan 1-bitlik ALU nun denetim sinyallerine karşı gelen işlevleri veriliyor.

Denetim Sinyalleri			ALU işlemi
cc	***cb***	***ca***	
0	0	0	(C_{out}, r) = sum(*a, b*) (**add** *için)*
0	1	0	r = *and*(*a, b*)
1	0	0	r = *or*(*a, b*)
0	0	1	(C_{out}, r) = sum(*a, –b*) (**sub** *için)*
1	1	1	r = *less* (***slt*** *için)*: Eğer *A – B* < 0 ise, *R* = 1,
Tüm diğer durumlar			önemli-değil

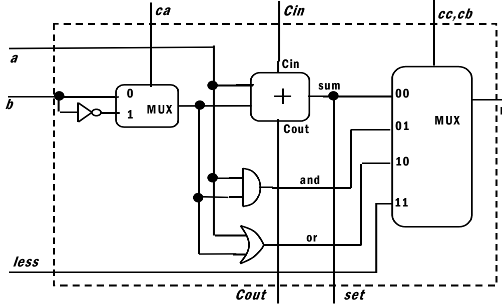
a) i) Bu 1-bitlik ALU -nun b_{invert} sinyali *ca*, *cb*, *cc* denetim sinyallerinden hangisi olabilir?

ii) Aşağıdaki devre elemanlarını kullanarak bu 1-bitlik ALUnun devre şeması çiziniz.

ve geçiti veya geçiti evirici tek-bit toplayıcı çoklayıcı (2x1, 4x1, 8x1)



<< çözüm



i) Yalnız *ca* binvert olabilir, çünkü sadece ca hem **add** için 0 hem de **sub** ve **slt** için 1 -dir.

ii) dikkat ederseniz *AND* ve *OR* geçit girişleri *a* ve *b* girişlerine de bağlanabilir.

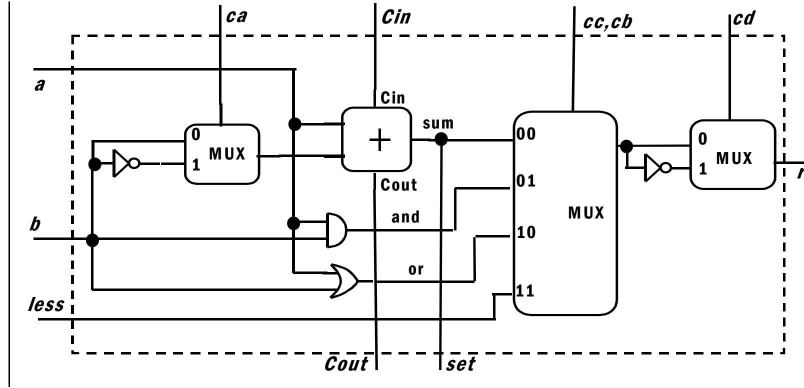
>>

b)

1-bitlik ALU -nuzu ek denetim sinyali *cd* ile aşağıdaki işlemleri yapacak şekilde genişletiniz. Mümkün olduğu kadar küçük çoklayıcı birimleri kullanmaya çalışın (örneğin, bir 4x1 ile bir 2x1 çoklayıcı, 8x1 çoklayıcıdan iki kat hızlı ve ucuzdur).

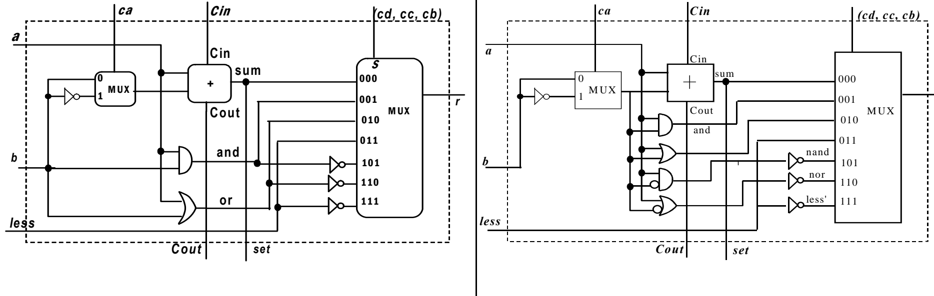
Denetim Sinyalleri				ALU işlemi
cd	***cc***	***cb***	***ca***	***r***
0	0	0	0	(C_{out}, r) =*toplam*(*a***,** *b***,** C_{in}) (**add** *için)*
0	0	1	0	*r* = *ve*(*a, b*)
0	1	0	0	*r* = *veya*(*a, b*)
0	0	0	1	(C_{out}, r) =*toplam*(*a***,** *b***'**, C_{in}) (**sub** *için)*
0	1	1	1	*r* = *less* (***slt*** *için)*: *A* – *B* < 0 ise, ***R*** = 1,
1	0	1	0	*r* = *nand*(*a, b*)
1	1	0	0	*r* = *nor*(*a, b*)
1	1	1	1	$\overline{less}$ (**sge** *için)*: *A* – *B* ≥ 0 ise, ***R*** = 1
Tüm diğer kombinasyonlar				önemli-değil

<<



cc,cb = 01 ve 10 kodlarıyla *NAND* ve *NOR* işlevini sağlamak için MUX çıkışında bir evirici kullanılmalıdır. Ancak bu durumda, *NAND* ve *NOR* un doğru çalışması için *AND* ve *OR* girişlerini a ve b -ye doğrudan bağlamak gerekir.

Diğer muhtemel çözümler:

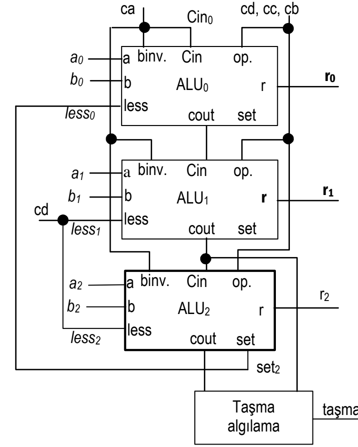


Diğer muhtemel çözümler daha fazla mantık geçidi ve çoklayıcı gerektirir.

>>

c) 1-bit ALU birimleri kullanarak **add**, **sub, and**, **or**, **nand**, **nor**, **slt** ve **sge** komutlarını yürütebilen en sağ bit biriminde taşma sinyali oluşturacak gerekli devre blokları dahil 3-bitlik bir ALU çiziniz.

<< çözüm



>>

4.4 Çarpma

Ç*arpılan* B sayısı ile *çarpan* Q sayısının çarpılması sonucu çıkan A sayısı *çarpım* olarak anılır .

$$\textit{çarpım} = \textit{çarpılan} \times \textit{çarpan}$$

$$A = B \times Q$$

Çarpım, çarpan kadar çarpılanın toplamından hesaplanabilir. Ancak toplamı almak için oldukça uzun zaman gerekir.

$$A = \underset{1.}{B} + \underset{2.}{B} + \ldots + \underset{Q.}{B};$$

Çarpımı daha kısa sürede bulmak üzere çarpılanın basamakları üzerine çarpanın dağılma özelliğinden yararlanılır.

Örnek 4-8:

İşaretsiz 4-bitlik şu iki sayının çarpımına bakınız.

$1101_2 \times 1010_2$.

$1010_2 = 1\times1000_2 + 0\times100_2 + 1\times10_2 + 0\times1$

Çarpımı kolay hesaplayabilecek bir yöntem geliştirmek için çarpma işleminin toplama işlemine dağılma özelliği kullanılır:

$1101_2 \times 1010_2 = 1101_2 \times (1\times1000_2 + 0\times100_2 + 1\times10_2 + 0\times1_2)$

+ **1101** ×1×**0** = +**0000**
+ **1101** ×10×**1** = +**1101**0
+ **1101** ×100×**0** = +**0000**00
+ **1101** ×1000×**1** = +**1101**000

Çarpım = 1000 0010

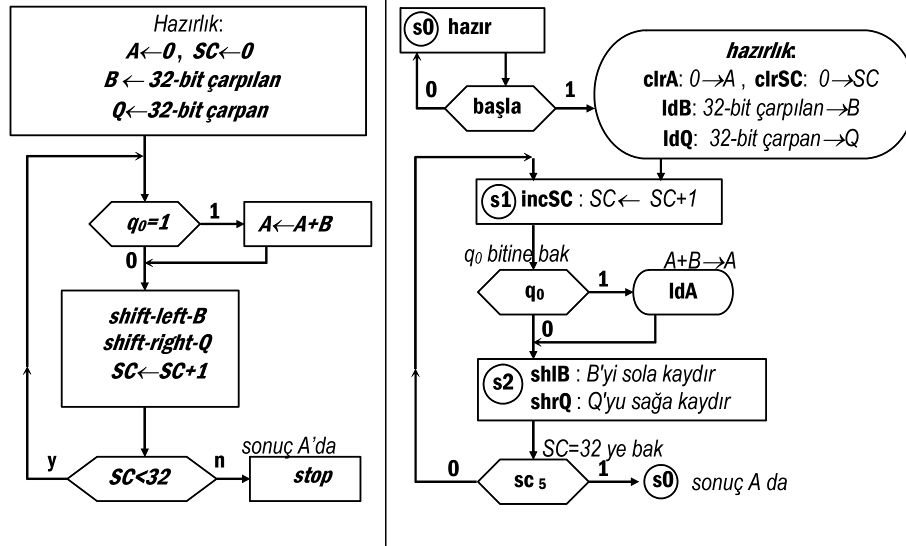
Eğer *Q* çarpanı ile *B* çarpılanı n ve m bit genişlikte iseler, *Q* ve *B* -nin tüm olası değerleri için doğru sonucu tutmak üzere *A* çarpımının n+m

bit olması gerekir. Demek ki hem *Q* hem de *B* sayıları 32-bit ise, $A=Q \times B$ 64-bit olmalıdır.

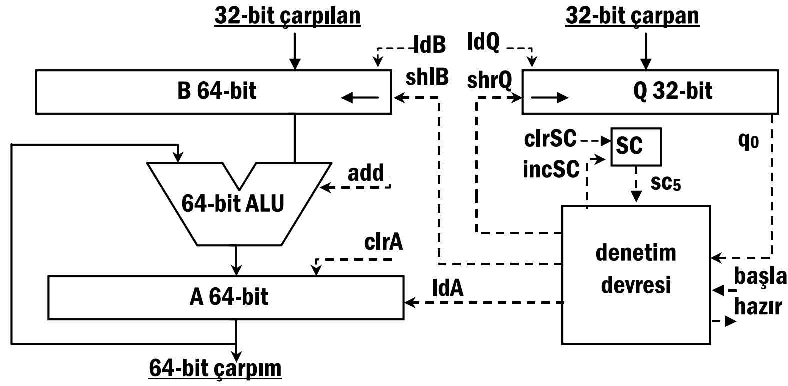
İleriki bölümlerde Bu algoritmaya dayanan üç tip çarpma makinesi tanıtacağız. Bu çarpım makinesi donanımlarından en gelişkini bile, kullanılacağı koşullara bağlı olarak sadeleştirilebilir ve algoritma bakımından da geliştirilebilir.

4.4.1 İlk Çarpma Algoritması

Yukarıda tanımlanan çapma algoritması, kaydırılan çarpılanı tutacak 64-bitlik bir kaydırma-yazmacı, çarpanın bitlerini test etmek için 32-bitlik bir kaydırma-yazmacı, sonucu tutmak için bir 64-bit yazmaç, ve ara terimlerin toplamlarını alacak 64-bitlik bir ALU -dan oluşan bir sayısal devreyle gerçekleştirilebilir. Bu çeşit yazmaçtan yazmaca veri işleyen devreleri tasarlarken algoritmayı betimlemede kullanılan yöntemlerden biri algoritmik durum makinası çizimidir (ASM, Algorithmic-State-Machine Chart). İlk yöntemin algoritması Şekil 4-14 te, veri-işlem donanımı da Şekil 4-15'de görülmektedir.



Şekil 4-14 İlk çarpma algoritması ve ASM çizimi



Şekil 4-15 İlk çarpıcı için donanım ve veri yolları

Algoritmanın 31 -inci biti işleyip sonucu elde edince durabilmesi için sıra sayacı (sequence counter) kullanılır. *SC* diyeceğimiz bu sıra sayacı 32 -ye kadar sayabilmek üzere en az 5-bit olmalıdır.

Örnek: Algoritmanın 4-bit sayılarla çalışması için, *A* ve *B* için 8-bit, *Q* için 4-bit yazmaç kullanmak ve bitiş için sc_2 bitine bakmak gibi çok küçük değişiklikler gerektirir. Bu durumda *SC*=4 olduğunda sc_2 biti 1 olduğundan 3-bitlik *SC* sayacı yeterlidir. Tablo 4-5'te 4-bitlik ilk-çarpıcıyla $0010_2 \times 1011_2$ işleminin yapılışı verilmektedir.

Tablo 4-5 $0010_2 \times 1011_2$ çarpımının 4-bitlik birinci çarpıcıda izlenişi

Durum	koşul	olay	SC	B	A	Q
s0		hazır	...	...	...	...
s0	başla	hazır,ldB, ldQ, clrSC, ClrA	000	0000 0010	0000 0000	1011
s1	q_0	incSC, ldA,	001		0000 0010	
s2		shlB, shrQ,		0000 0100		0101
s1	q_0	incSC, ldA,	010		0000 0110	
s2		shlB, shrQ,		0000 1000		0010
s1		incSC,	011			
s2		shlB, shrQ,		0001 0000		0001
s1	q_0	incSC, ldA,	100		0001 0110	
s2	sc_2	shlB, shrQ,		0010 0000		0000
s0		hazır			0001 0110	

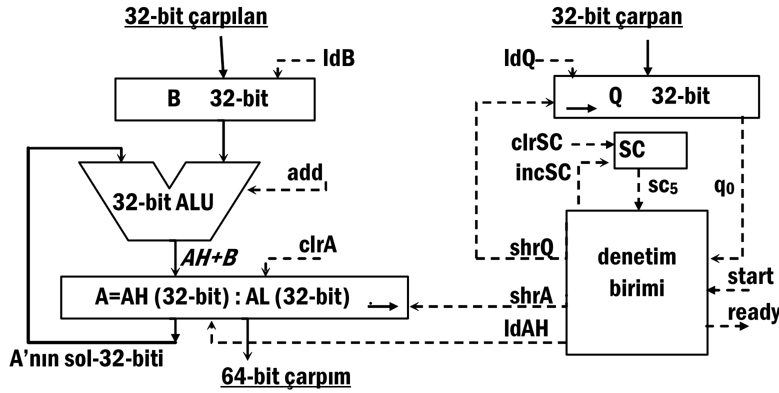
ASM çizimi izleme tablosunda oluşan koşulları ve saat geçişlerinde gerçekleşen durum değişimini görüyorsunuz. Her satır bir saat donüşünü göstermekte, ve satırda sadece değişen yazmaçlar güncellenmektedir.

Bu ASM donanımıyla 32-bitlik bir çarpma 66 saat dönüşünde tamamlanır. Eğer tüm kaydırma (*shlB, shrQ*) ve toplama (*ldA*) işlemleri için tek bir ALU kullanılırsa, algoritma 32×3+2 =98 saat dönüşü tutar. Tüm yazmaçlar s1 ve s2 durumları birleşimi süresince sadece birer işlem yaptığından saat dönüşünü 34 -e düşürmek te mümkündür.

4.4.2 İkinci Çarpma Algoritması

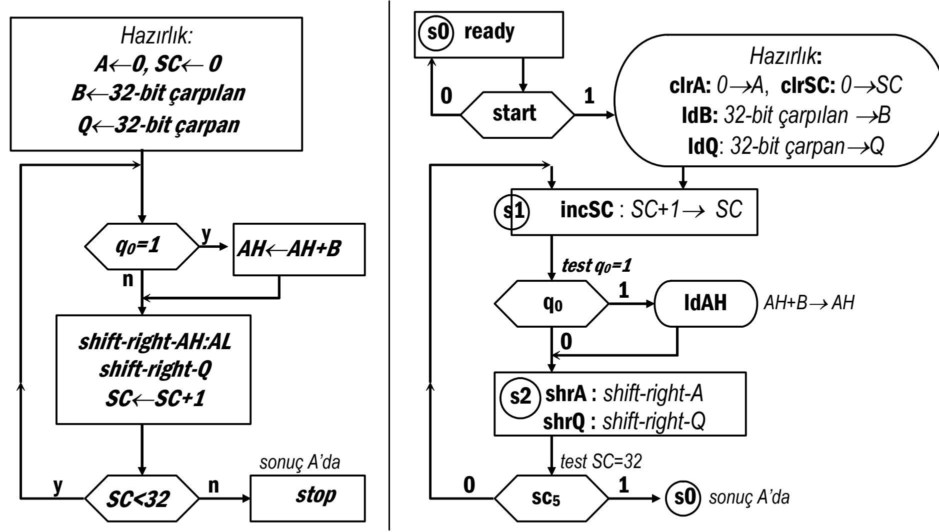
İlk-çarpıcıda, çarpılan yazmacı *B* deki bitlerin yarısı her zaman 0 değerinde kalıyor. Veriler yazmacın sadece yarısını kaplamakta, ancak bu bitler sola doğru kaymaktadır. Eğer kayma referansımızı değiştirerek *B* yazmacını durdurursak *B* 32-bit olduğundan aynı zamanda toplama biriminin genişliği de azalacaktır. ALU girişleri açısından *A* dururken *B* sola kaymaktaydı. Şimdi *B* -yi sabit tutarsak *A* -yı sağa kaydırmalı ve gene *Q* yu sağa kaydırırken q_0 bir olduğunda *A* -nın üst yarısını *B* ile toplamalıyız.

Geliştirilmiş çarpma biriminin algoritması, ASM garfiği, ve veri işlem yolları Şekil 4-16, ve Şekil 4-17'de gösterilmiştir.



Şekil 4-16 İkinci çarpıcı devresinin Veri Yolları.

Bununla birlikte, sıradan bir kaydırıcı yazmaç kullanırsak *A* -yı aynı saat dönüşünde hem *AH+B→AH* ile yükleyip hem de kaydıramayız. Bu nedenle *ldAH* işleminin ardından shrA için yeni bir durum kullanmamız gerekir.



Şekil 4-17 ikinci-çarpıcı devresinin algoritması ve ASM çizimi

Tablo 4-6 $0010_2 \times 1011_2$ çarpımının 4-bitlik ikinci-çarpıcı ile izlenmesi.

Durum	koşul	olay	SC	B	A	Q
s0		ready	...	...	...	...
s0	start	ready,clrA,clrSC,ldB,ldQ	000	0010	0000 0000	1011
s1	q_0	incSC, ldAH	001		0010 0000	
s2		shrA,shrQ			0001 0000	0101
s1	q_0	incSC, ldAH	010		0011 0000	
s2		shrA,shrQ			0001 1000	0010
s1		incSC,	011			
s2		shrA,shrQ			0000 1100	0001
s1	q_0	incSC, ldAH	100		0010 1100	
s2	sc_2	shrA,shrQ			0001 0110	0000
s0		ready			0001 0110	

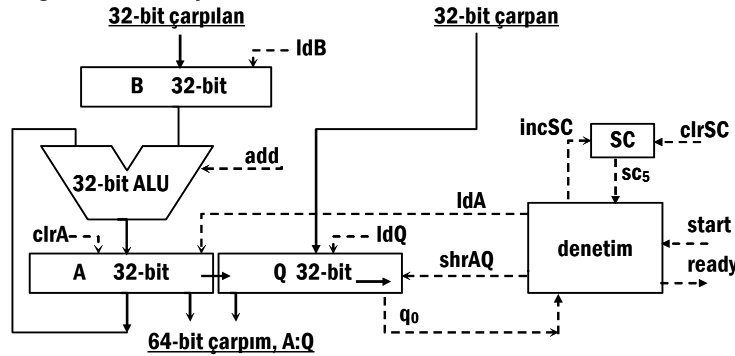
Bu geliştirilmiş uygulama 32-bitlik bir çarpmayı 32×2+2 saat dönüşünde tamamlar. Ancak, 32-bitlik ALU 64-bitlik ALU -dan hemen hemen iki kat daha hızlı çalıştığından bu devrenin saat hızını neredeyse iki kat artırılabilir.

Ayrıca kaydırma işlemleri toplamadan daha kısa saat dönüşü gerektirir, ve bu sayede değişken saat döngüleri kullanarak daha hızlı işlem yapmak mümkündür. Daha hızlandırmak için ASM çizimi değiştirerek saat döngülerinin sayısını 32+(*çarpandaki birlerin sayısı*)+2 -ye düşürmek te mümkündür, fakat bir işlemin çalışma süresinin işlenene bağlı olmasını istemediğimizden bu değişikliklerden kaçınırız.

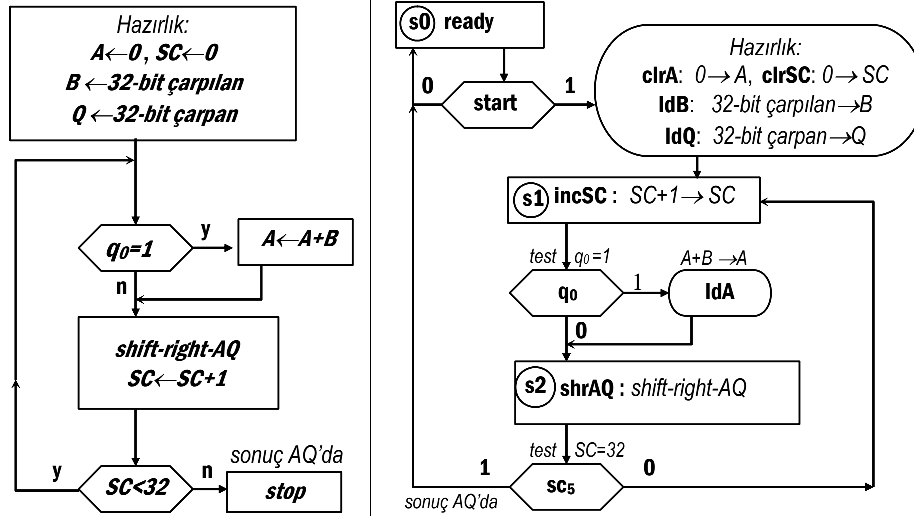
Eğer tüm kaydırma ve toplama işlemleri için tek ALU kullanılırsa, algoritmanın en hızlı yürütülmesi 32×3+2=98 saat döngüsü tutar. Özetle, geliştirilen devrede hem çalışma zamanı azalır hem ALU boyutu küçülür.

4.4.3 Üçüncü Çarpma Algoritması:

Geliştirilmiş çarpan algoritmasında *Q* yazmacının boş kısmı ve *A* yazmacının kullanılan kısmı birbirini tamamlar. Çarpım yazmacı A yı sağ yarısı ile çarpan yazmacı *Q* ile birleştirip 32-bit bir yazmaç elde etmek mümkündür. Değişiklikten sonra, *A* 32-bit olur, ve çarpmanın sonucu (*AQ*) da kalır, yani $B{\times}Q \rightarrow AQ$ olur. Bu algoritmaya üçüncü-çarpma algoritması diyelim.



Şekil 4-18 Üçüncü Çarpıcı Devresi Veri Yolları.



Şekil 4-19 Üçüncü çarpıcı algoritması ve ASM çizimi

Tablo 4-7 $0010_2 \times 1011_2$ çarpımını ileri çarpıcı devresi ile izlenmesi.

Durum	koşul	olay	SC	B	A	Q
s0		ready	...	...	...	...
s0	start	ready, clrA, clrSC, ldB, ldQ	000	0010	0000	1011
s1	q_0	incSC, ldA	001		0010	
s2		shrAQ			0001	**0**101
s1	q_0	incSC, ldA	010		0011	
s2		shrAQ			0001	**10**10
s1		incSC,	011			
s2		shrAQ			0000	**110**1
s1	q_0	incSC, ldA	100		0010	
s2	sc_2	shrAQ			0001	**0110**
s0		ready			**0001**	**0110**

4.4.4 İşaretli Çarpma

$A=(0,a_{n-2},a_{n-3},..,a_0)$ ve $B=(0,b_{n-2},b_{n-3},..,b_0)$ ikilik tümleyen biçimde negatif olmayan n-bitlik işaretli sayılar olsun. n-bitlik negatif işaretli (*signed*) $-B$ sayısı ile n-bitlik pozitif işaretli A sayısının 2n-bitlik çarpımı P şöyledir.

$$P = (HI{:}LO) = -A\times B = A \times (2^n - B) = 2^n\times A - A\times B,$$

Burada ($2^{n-1} > A \geq 0$; $2^{n-1} > B \geq 0$).

A -nın değeri ne olursa olsun, $2^n\times A$ sayısının en sağdaki n-biti her zaman sıfırdır. Bu nedenle, $A\times B$ çarpımı 2^n -den küçükse, LO diye adlandırdığımız P -nin en sağdaki n-biti 2-lik tümleyici biçimde $-A\times B$ çarpımını barındırır.

$$LO = 2^n - A\times B = A\times(-B), \quad (2^{n-1} > A \geq 0,\ 2^{n-1} > B \geq 0,\ 2^{n-1} > A\times B \geq 0)$$

Çarpılan ve çarpanın ikisinin de negatif olduğu, yani $2^{n-1}>A\geq0$ ve $2^{n-1}>B\geq0$ için $-A\times-B$ çarpımını ele alalım. Bu işaretli sayıların işaretsiz tamsayılar gibi çarpıldğını düşünün

$$P = (HI{:}LO) = (2^n - A) \times (2^n - B) = 2^{2n} - 2^n(A+B) + A\times B.$$
$$= 2^n(2^n - (A+B)) + A\times B.$$

$2^n(2^n - (A+B))$ -nin en sağdaki n-biti herzaman sıfır olacağından, eğer çarpım bu n-bit kısımdan taşmıyorsa, P -nin en-sağ n-bitinde $-A \times -B = A\times B$ çarpım değeri vardır.

$$LO = A\times B = (-A)\times(-B), \quad (2^{n-1} > A \geq 0,\ 2^{n-1} > B \geq 0,\ 2^{n-1} > A\times B \geq 0).$$

Özetle, n-bit işaretli çarpma için, alt n-bit sadece taşma durumu yoksa doğru sonucu içerir. Bununla beraber, taşma durumu için basit bir test yoktur. En basit test çarpılan ve çarpanın anlamsız (non-

significant) bitlerinin toplamını kullanır. Işaret bitine bitişik olup işaret bitinin değerini tekrar eden bitlerin anlamı yoktur; yani, 1110_2 -da, b_2 ve b_1 anlamsızdır, sadece b_0 ve işaret biti anlamlıdır. 0011_2 -te, sadece b_2 anlamsızdır. n-bitlik işaretli sayılarda, eğer çarpan ve çarpılanın anlamsız bitlerinin toplamı n-2 den küçükse, işaretli çarpım 2n bitlik yazmacın en sağdaki n-bitlik kısımdan taşmaz.

n-bitlik işaretli iki sayının çarpımını tutmak için en çok 2n bit gerekir. Bu nedenle, 32-bit ile 32-bit işaretli sayıları işaretsiz çarpıcıyla çarpıp 64-bit çarpım elde etmek için iki yol vardır.

Birinci yol 32-bit×32-bit ışaretsiz çarpıcı kullanarak:

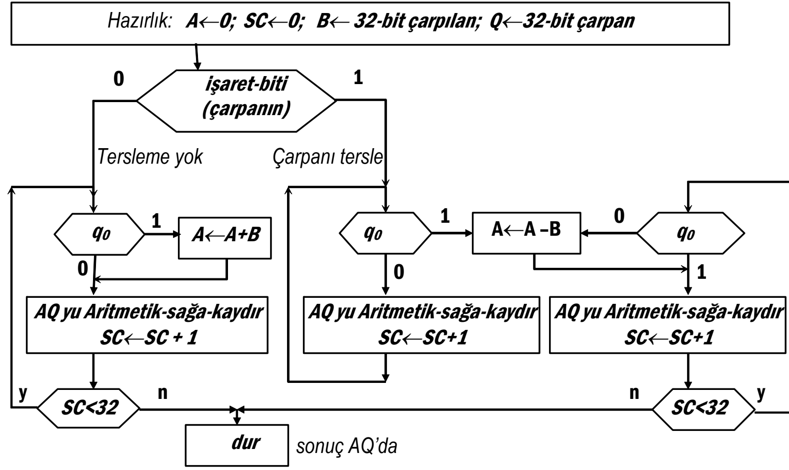
1) A ve B nin işaretinden çarpımın işaretine karar verin.
2) Negatif sayıların tümleyenini (2-lik tümleyeni) kullanın.
3) Negatif olmayan sayılardan 64-bit çarpımı bulun.
4) Çarpım negatif olacaksa çarpımı eksileyin

ikinci yol 64-bit × 64-bit işaretsiz çarpıcı ile:

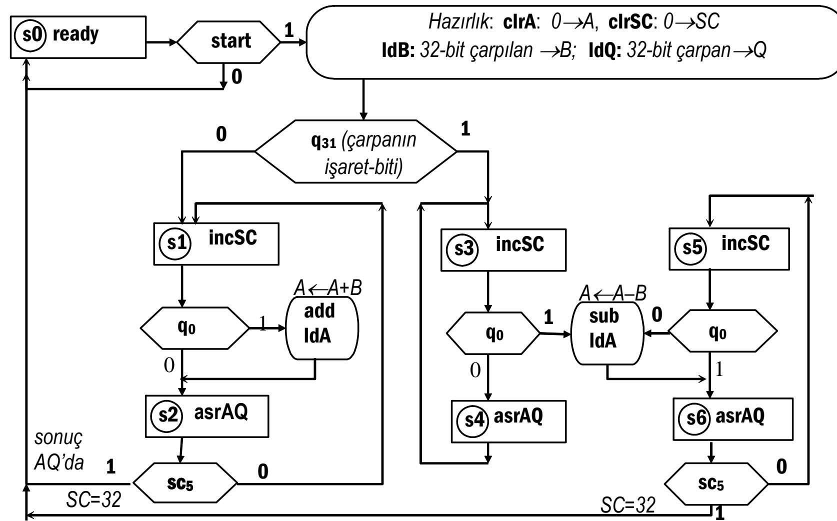
1) çarpılanı ve çarpanı 64 bite işaretli genişletin (sign-extend)
2) işaretli genişletilmiş sayıları işaretsiz çarpıcı ile çarpın
3) işaretsiz 128-bitlik çarpımın sağ-64-bit yarısı işaretli çarpımdır.

4.4.5 İşaretli Çarpma Algoritmaları

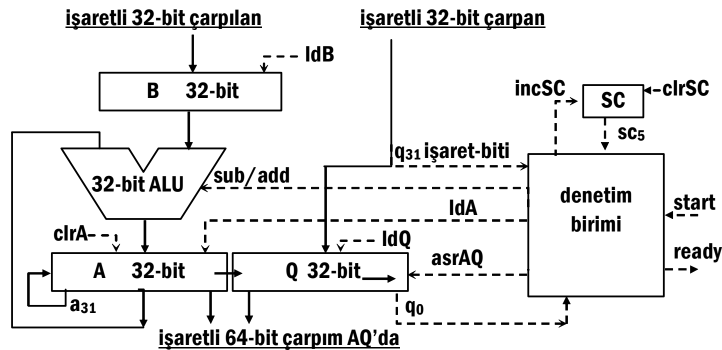
Üçüncü ASM çizimi işlenenleri tersleyecek şekilde değiştirilerek işaretli çarpma yapması sağlanabilir. Toplama yerine çıkarma kullanmak gerekecektir, ve çarpanın bitleri en sağdan sola doğru birer birer kullanılarak çarpanı ardarda tümlenebilir. *AQ* -daki sayı işaretli sayı olduğundan, sağa kaydırma işlemi sırasında işaret biti korunmalıdır. Yazmacın işaret bitini koruyan sağa kaydırma işlemi aritmetik sağa kaydırma (arithmetic-shift-right operation) diye adlandırılır. Bu işlem asrAQ denetim sinyaliyle sağlanır.



Şekil 4-20 İşaretli Üçüncü-Çarpma Algoritması.



Şekil 4-21 İşaretli Üçüncü-Çarpıcı devresinin ASM çizimi.



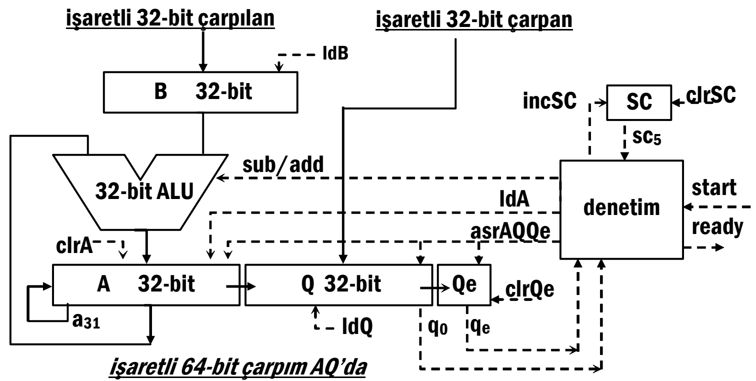
Şekil 4-22 İşaretli Üçüncü-Çarpıcı VeriYolu.

Tablo 4-8 İşaretli Üçüncü-çarpıcı devresi için $0010_2 \times 1011_2$
(= $2_{10} \times -5_{10}$) işaretli çarpımının izlenmesi

Durum	koşul	olay	SC	B	A	Q	açıklama
s0		ready	...	...	...	...	...
s0	start, q_3	ready, clrA, clrSC, ldB, ldQ	000	0010	0000	**1011**	–B = 1110
s3	q_0	incSC, sub, ldA	001		1110		A ← A –B
s6		asrAQ			1111	0**101**	arithm shr
s5	q_0	incSC	010				
s6		asrAQ			1111	10**10**	
s5		incSC, sub, ldA	011		1101		A ← A –B
s6		asrAQ			1110	110**1**	
s5	q_0	incSC,	100				
s6	sc_2	asrAQ			1111	0110	
s0		ready			**1111**	**0110**	AQ = –10

Booth -un İşaretli Çarpma Algoritması

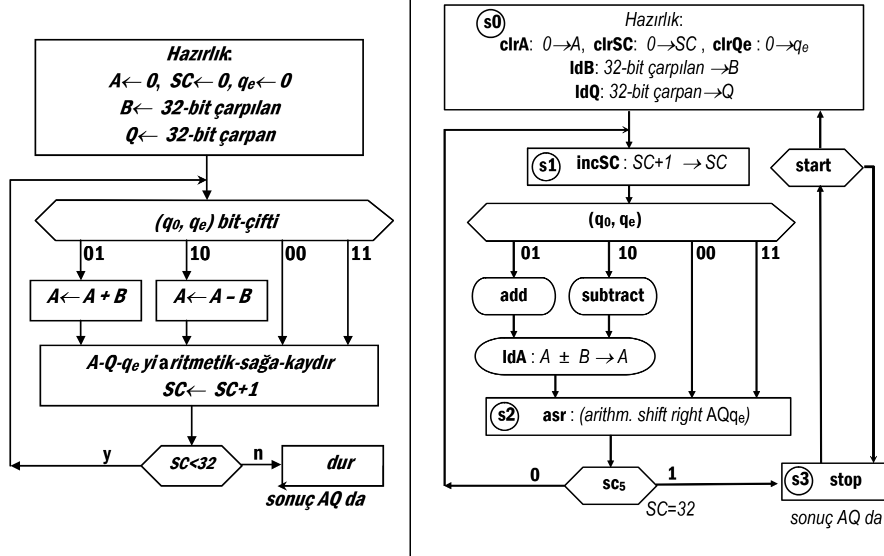
Booth birçok çarpma algoritması geliştirmiştir. Bunlardan biri çarpan kaydırılırken alışılmış sağdaki bitin sınanması yerine en sağdaki iki bitin sınanmasına dayanır. Booth -un algoritması sadece çarpanın iki ardışık biti farklı değerdeyse bir toplama yada çıkarma gerektirir.



Şekil 4-23 Booth -un İşaretli Çarpma Veriyolu.

Booth -un algoritması eğer çarpan aynı durumlu (0...0 veya 1...1) uzun diziler içeriyorsa avantajlıdır. Ancak, çarpan 0 ve 1 değişimleri

içeriyorsa dezavantajlıdır, çünkü değişimler her basamakta çıkarma yada toplamaya neden olur.



Şekil 4-24 Booth -un işaretli çarpma algoritması ve ASM çizimi.

Tablo 4-9 Booth -un işaretli çarpıcısıyla $0010_2 \times 1011_2$ ($=2_{10} \times -5_{10}$) çarpımının izlenmesi.

St	koşul	olay	SC	B	A	Q	q_e	açıklama
s3	**start**	**stop**	**...**	**...**	...	...	...	**algorıthm starts**
s0	**--**	**clrA, clrSC, ldB, ldQ**	**000**	**0010**	0000	**1011**	0	**–B = 1110**
s1	**q_0,q_e=10**	**incSC, sub, ldA**	**001**		1110			**A ← A – B**
s2		**asr (AQQe)**			1111	0**101**	**1**	**arithm. shift rıght.**
s1	**q_0,q_e=11**	**incSC**	**010**					
s2		**asr (AQQe)**			1111	10**10**	**1**	
s1	**q_0,q_e=01**	**incSC, add, ldA**	**011**		0001	10**10**		**A ← A + B**
s2		**asr (AQQe)**			0000	110**1**	**0**	
s1	**q_0,q_e=10**	**incSC, sub, ldA**	**100**		1110			**A ← A – B**
s2	**sc_5**	**asr (AQQe)**			1111	0110	**1**	
s3		**stop**			**1111**	**0110**		

AQ= $\mathbf{11110110_2}$ **=** $-\ 0000\ 1010_2 = -10_{10}$

Örnek 4-9 *:* a) 3C h ve b) AA h çarpan değerleri için, Booth -un algoritmasında ve ileri işaretli çarpım algoritmalarında gerek duyulan ALU işlem sayısını bulunuz.

	B Q	Booth -un algoritması	işaretli algoritma
a)	0011 1100	1 toplama + 1 çıkarma	5 toplama
b)	1010 1010	4 toplama + 4 çıkarma	4 çıkarma

Alıştırma: Booth algoritmasını kullanarak çarpmayı yapın

$$1010_2 \times 0011_2 = -6_{10} \times 3_{10}$$

4.4.6 Birleşimsel Çarpıcı Ağı

Bir birleşimsel çarpıcı çarpma işlemini yapacak örgü yapısında yeterli sayıda ve kapısı ve toplama birimnden oluşturulur. Devre çoğunlukla birleşimsel çarpıcı ağı (combinational array multiplier) olarak anılır.

B ve *Q* çarpılacak n-bitlik sayılar olsun. En soldan en sağa sıralarsak B -nin bitleri b_{n-1}, b_{n-2}, ... , b_2, b_1, b_0 ile ve *Q* -nun bitleri q_{n-1}, q_{n-2}, ... , q_2, q_1, q_0 ile gösterilsin. Aynı şekilde 2n-bitlik $A=B\times Q$ çarpımının bitleri de a_{2n-1}, a_{2n-2}, ... , a_2, a_1, a_0 olsun.

Çarpan ve çarpılan kendi bitleri cinsinden şöyle yazılabilir

$$Q = \sum_{j=0}^{n-1} q_j \, 2^j ,$$

$$B = \sum_{k=0}^{n-1} b_k \, 2^k .$$

Çarpımı *A* ile gösterirsek

$$A = B \times Q$$

$$= B \times \left(\sum_{j=0}^{n-1} q_j \, 2^j \right)$$

$$= \sum_{j=0}^{n-1} q_j \times 2^j \times B \qquad (1)$$

.

Örnek: $B= 1010_2$ ve $Q= 1101_2$

$$1010_2 \times 1101_2 = 1010_2 \times (1000_2 \times q_3 + 100_2 \times q_2 + 10_2 \times q_1 + 1 \times q_0)$$

=	+ **1010** ×1× **1**	=	+ **1010**	$q_0 \times 1 \times B$
	+ **1010** ×10× **0**		+**0 000**0	$q_1 \times 2 \times B$
	+ **1010** ×100× **1**		+**10 10**00	$q_2 \times 2^2 \times B$
	+ **1010** ×1000× **1**		+**101 0**000	$q_3 \times 2^3 \times B$
Çarpım =	10000010	=	1000 0010	= *A*

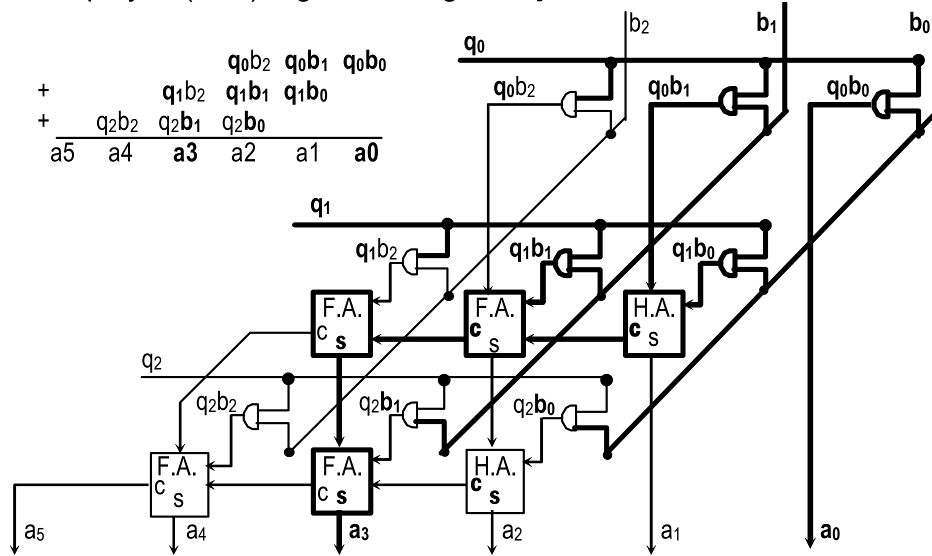
İfade (1) *B* -nin bitleri cınsınden yazılabilecek şekilde gelişir.

$$A = \sum_{j=0}^{n-1} q_j \times 2^j \times B$$

$$= \sum_{j=0}^{n-1} 2^j \times q_j \times \left(\sum_{k=0}^{n-1} b_k \times 2^k \right).$$

$$= \sum_{j=0}^{n-1} 2^j \times \left(\sum_{k=0}^{n-1} q_j \times b_k \times 2^k \right) = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} (q_j \,.\, b_k) \times 2^{j+k}. \quad (2)$$

İfade (2) de n×n adet $q_j \times b_k$ türü 1-bitlik çarpım terimi vardır. Herbir çarpım terimi için iki-girişli bir VE-kapısı kullanılabilir. 2^{j+k} üstel faktörü sadece çarpan terimi $q_j \times b_k$ -nın toplanacağı basamağı belirler. 3-bitlik çarpıcı ağının tüm devresi Şekil 4-25'de $011_2 \times 011_2$ işlemini yaparken '1' olan yolları koyulaştırılarak verilmiştir. Devrede çarpım terimlerinini toplanması (n–1)-tane yarım-toplayıcı (H.A.) ve (n–1)×(n–1)-tane tam-toplayıcı (F.A.) örgüsü ile sağlanmıştır.



Şekil 4-25 3-bitlik örgü çarpıcı devresi $011_2 \times 011_2 = 001001_2$ işlemini yapıyor.

Toplayıcı ve VE-kapılarının gecikme zamanları d_a ve d_g ise, bu devrenin çıkışı en çok $4d_a + d_g$ sürede stabilize olur. Bu süre en fazla sayıda toplayıcı ve kapı içeren sinyal yolu boyunca (elde-aktarımı boyunca q_0 -dan z_5 -e) olan gecikmedir.

4.5 Bölme

Bölünenin sıfırdan farklı bir *bölen* -e bölündüğü işaretsiz bölmeden iki sonuç çıkar: *bölüm* , ve bölenden daima küçük bir *kalan*.

$$\frac{bölünen}{bölen} = bölüm + \frac{kalan}{bölen}.$$

Bölüm, bölen bölünenden çıkarılabildiği sürece ard arda yapılan çıkarmaları sayarak bulunabilir, ancak bu zaman alır. Daha kestirme bır yol bölenin çıkabilen katlarını, ya da aynı nedenle böleni bölünenin en sol basamağından başlayarak çıkarmaktır.

Örnek 4-10*:* 11/3 = 3 + 2/3 bölmesinde
bölünen =11, bölen =3, bölüm =3, kalan =2 dir.

1011 / 11 = 10 + 1 + (10/11) = 11 + 10/11

1100	←	11 × 100 = 1100 > 1011 bu yüzden 100 bölümde değildir
– 110	←	11 × 10 = 110 ≤ 1011 bu yüzden bölüm 10 içerir,
101	←	bölüme 10 ekle ve 1011 den 11 × 10 çıkar.
– 11	←	11 × 1 = 11 ≤ 101, bu yüzden bölüm 1 içerir.
10	←	bölüme 1 ekle ve 101 den 11× 1 çıkar.
		10 < 11 olunca bolme biter. 10 kalır,

İkili bölmede bölenin katlarının bölünenle ardı ardına karşılaştırılması gerekir. Yazmaç-A -da (A= 1101 0000_2 =208) 8-bit bölünenle yazmaç-B -de (***B***=0001 0001_2 =17) 8-bit bölenle başlayan bölme işlemini örnek olarak alalım. Başlarken 8-bit bölünen ve bölen A -da ve BH -da yer alırken bitişte bölüm ve kalan 8-bitlik Q yazmacı ile A yazmacında kalır.

Tablo 4-10 A =1101 0000_2 nın B = 0001 0001_2 a ikilik sistemde bölünmesi

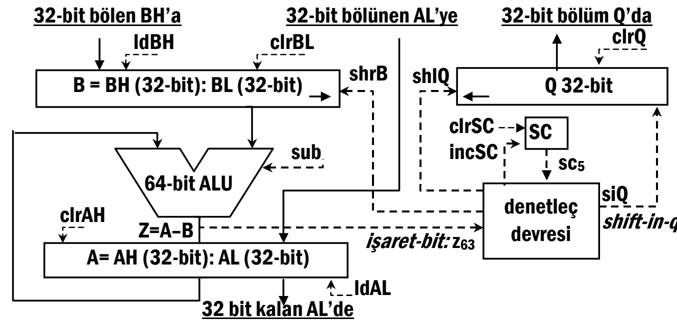
Bölünen A -da	Bölen BH -da	BL	$Z=A–2^k B$ koşul	İşlem	Bölüm Q -da

1101 0000	0001 0001	*0000 0000*	$2^8 B > A$; z_{15}=1	hazırlık	0000 0000
	000 1000	1*000 0000*	$2^7 B > A$; z_{15}=1	$0 \rightarrow q_7$	
	000 100	01*00 0000*	$2^7 B > A$; z_{15}=1	$0 \rightarrow q_6$	
	00 010	001*0 0000*	$2^5 B > A$; z_{15}=1	$0 \rightarrow q_5$	
	0 001	0001 *0000*	$2^4 B > A$; z_{15}=1	$0 \rightarrow q_4$	
0100 1000	000	1000 1*000*	$2^3 B \leq A$; z_{15}=0	$1 \rightarrow q_3$ $A - 2^3 B \rightarrow A$	0000 1000
0000 0100	00	0100 01*00*	$2^2 B \leq A$; z_{15}=0	$1 \rightarrow q_2$; $A - 2^2 B \rightarrow A$	0000 1100
	0	0010 001*0*	$2^1 B > A$; z_{15}=1	$0 \rightarrow q_1$	
		0001 0001	$2^0 B > A$; z_{15}=1	$0 \rightarrow q_0$	
0000 0100					0000 1100
kalan		208/17 = 12 + 4 / 17			bölüm

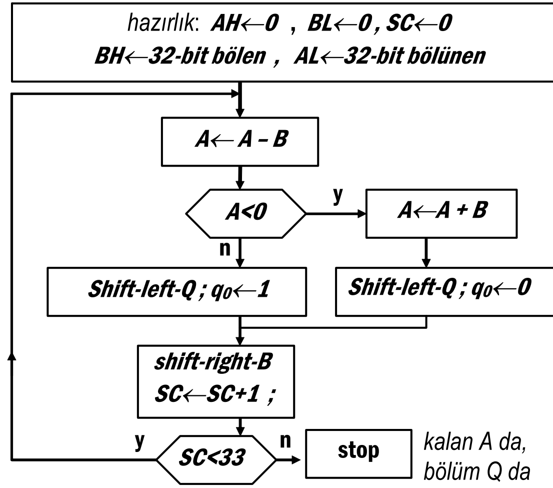
Tablo 4-10 da, k nın 8 den 0 a azalan değerleri için BH:BL nin içinde $2^k B$ vardır; ve $Z = A - 2^k B$ kalanının işaret biti z_{15} in her pozitif olduğunda $Z = A - 2^k B$ kalanı A ya aktarılır.

4.5.1 İlk-Bölme Algoritması

Örnekte, bölüm bitleri soldan sağa kayarak girilecek olsa n kaydırma işlemi sonucu ilk girilen bit en soldaki basamağa gitmesi sağlanmış olur. Koşuldaki ($2^n B>A$) karşılaştırmasındaki çıkarma işlemini yapmak için 2n-bitlik bir ALU gerekir. Bu gözlemlerın ardından, ilk-bölme için ilk-çarpıcı ile benzer donanımı kullanacak bir ASM çizimi geliştirebiliriz. Hatırlarsanız ALU sonucunu AH -ya aktarmaya karar vermek için sonucun işaret bitine (z_{63}) bakıyoruz.

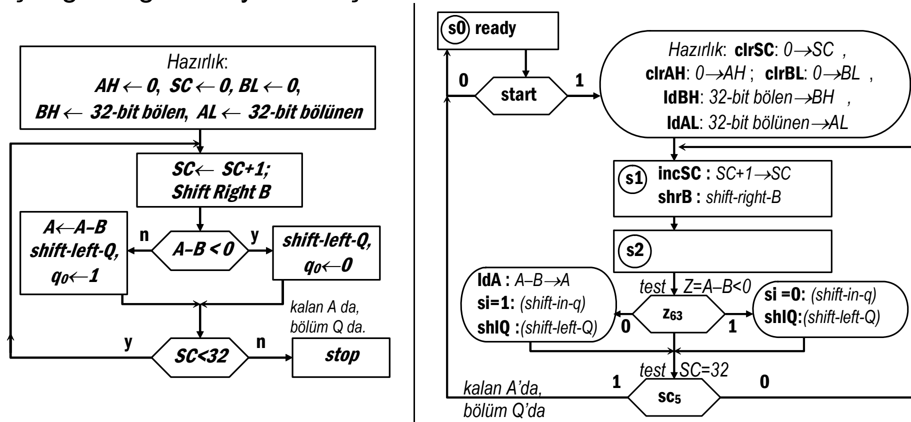


Şekil 4-26 İlk-bölücü için veri işleme donanımı



Şekil 4-27 İlk-bölücü algoritması

Bu algoritma, MIPS makine dili gibi yalnızca A<0 koşulunun sınanabildiği makıneler için tasarlanmıştır. Oysa denetim devresi ALU çıkışının işaret bitini çıkışı A -ya aktarmadan da sınayabilir. Verıyolunun bu özelliğini kullanacak olursak algoritmayı daha basit bir eşdeğer algoritmaya dönuştürebiliriz.



Şekil 4-28 İlk-bölücünün eşdeğer algoritması ve ASM çizimi

Tablo 4-11 Birinci bölücü devresinde 1011_2 / 0011_2 =11_{10} / 3_{10} işleminin izlenmesi.

Durum	koşul	olay	SC	B	A	Q	Z=A–B
s0		**ready**	...	...	...	...	
s0	**start**	**ready, clrAH, clrSC, clrBL, ldBH, ldAL**	**000**	**0011** 0000	0000 **1011**	0000	
s1		**incSC, shrB,**	**001**	0**001 1**000			**1**111 0011
s2	**z_7**	**shlQ**				000**0**	
s1		**incSC, shrB,**	**010**	00**00 11**00			**1**111 1111
s2	**z_7**	**shlQ**				00**00**	
s1		**incSC, shrB,**	**011**	000**0 011**0			**0**000 0101
s2		**ldA, siQ, shlQ**			0000 **0101**	0**001**	
s1		**incSC, shrB,**	**100**	0000 **0011**			**0**000 0010
s2	**sc_2**	**ldA, siQ, shlQ**			0000 **0010**	**0011**	
s0		**ready**			0000 **0010**	**0011**	

Ready sinyali çıktığında kalan A -da ve bölüm Q -dadır

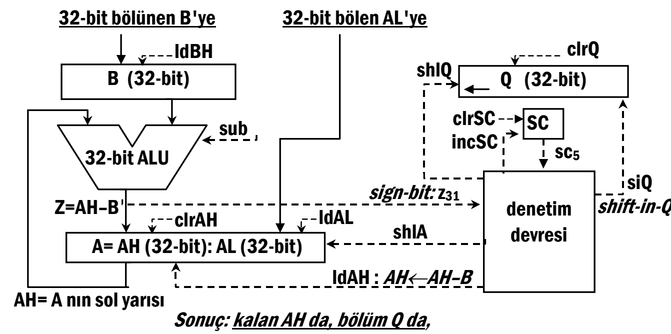
Bu izleme tablosundaki z_7, $Z=A–B$ yi hesaplayan ALU çıkışındaki işaret bitidir. Bölünen ve bölen 4-bitlik olduğundan denetleç döngüyü SC=4 oluncaya dek yani sc_2 =1 oluncaya dek işler.

4.5.2 İkinci-bölme algoritması

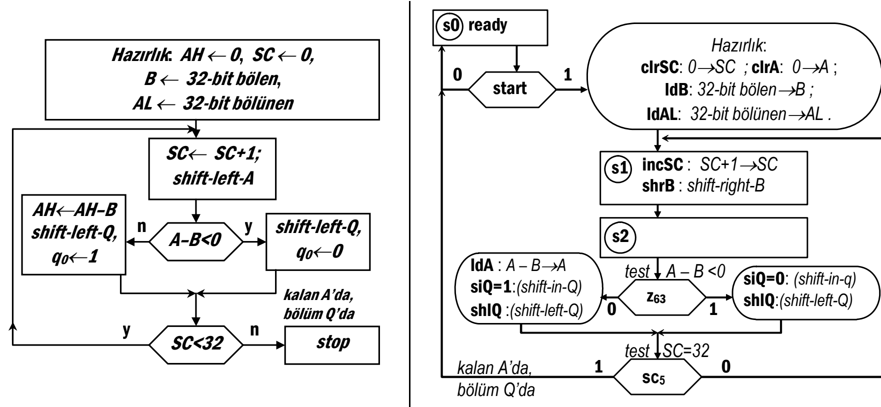
Birinci bölmede *B* yazmacının yarısı boştur ve bu yüzden *A* yazmacının yalnızca yarısı işleme girer. Ancak işlenen bitler sağa doğru kayarlar. ALU boyutunu yarıya indirmek üzere ilk-bölme algoritmasında aşağıdaki değişiklikler sonunda ikinci algoritma geliştirilir:

B yazmacının genişliğini yarı uzunluga düşür, ve böleni durağan tut.

A yazmacındakileri sola kaydır ki böylece ALU -da *A* -nın sadece sol yarısını işlemek yeterli olsun.



Şekil 4-29 İkinci-bölücü için veri işleme donanımı



Şekil 4-30 İkinci-bölme algoritması ve ASM çizimi

Tablo 4-12 İkinci bölme algoritmasıyla 1011_2 / 0011_2 bölmesinin izlenişi.

Durum	koşul	olay	SC	B	A	Q	AH–B
s0	start	ready, clrAH, clrSC, ldB, ldAL	000	0011	0000 1011	0000	
s1		incSC, shlA,	001		0001 0110		1110
s2	z_3	shlQ				0000	
s1		incSC, shlA,	010		0010 1100		1111
s2	z_3	shlQ				0000	
s1		incSC, shlA,	011		0101 1000		0010
s2		ldA, siQ, shlQ			0010 1000	0001	
s1		incSC, shlA,	100		0101 0000		0010
s2	sc_2	ldA, siQ, shlQ			0010 0000	0011	
s0		ready					

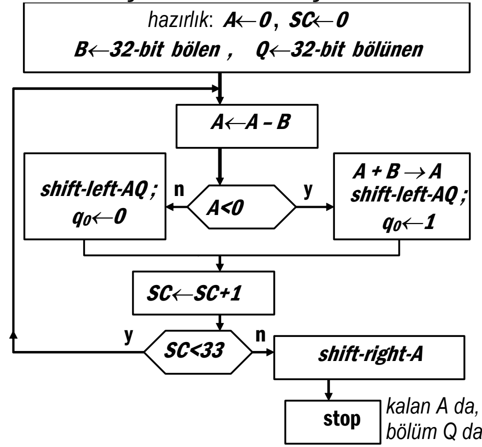
kalan A -da=0010
bölüm Q -da=0011

4.5.3 Üçüncü bölme algoritması

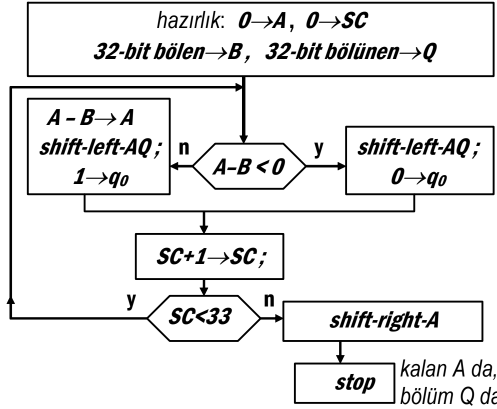
A ve *Q* yu aynı anda kaydırmak mümkündur, ve bölünen *AL* yerine *Q* ya yüklenebilir. Böylece *Q*, *A* -nın boşalan bitlerine kayar. Algoritma sona erdiğinde *Q* böleni, *A* da kalanı barındırır. Ancak aynı yazmacı aynı anda hem yüklemek ($A \leftarrow A - B$) hem bitleri kaydırmak (shift-AQ) mümkün değildir. Bu nedenle bu veriyolunu kullanırken kaydırma ve yüklemeyi tek bir ASM blokunda birleştirerek bir saat dönüşü kazanmak mümkün olamaz.

Ayrıca load-A ve shift-Q için ayrı saat dönüşleri kullanıldığında başka bir sorun çıkar. Q -ya kaydırılacak giriş-biti (shıft-in-bit) A–B -nin işaretine bağlıdır. Başlangıçta *A*=0 ve *A–B* hep negatif olduğundan *Q* -

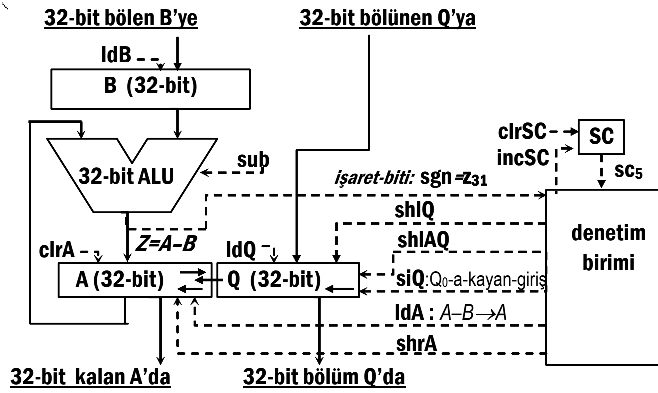
ya istenmeyen bir "0" biti girer. *Q* daki bölünen bitlerinin sonuncusu *A* -ya geçip bölümün son biti *Q* -ya aktarıldığında bu istenmeyen 0-biti *A* -ya kayar. Kalanı bozan bu fazla sıfır biti algoritmanın sonunda *A* -yı tek başına sola kaydırarak dışarı atılır. Üçüncü bölme devresinin algoritması ve donanımı Şekil 4-32 ve Şekil 4-33'de gösterilmiştir.



Şekil 4-31 Üçüncü bölme algoritması

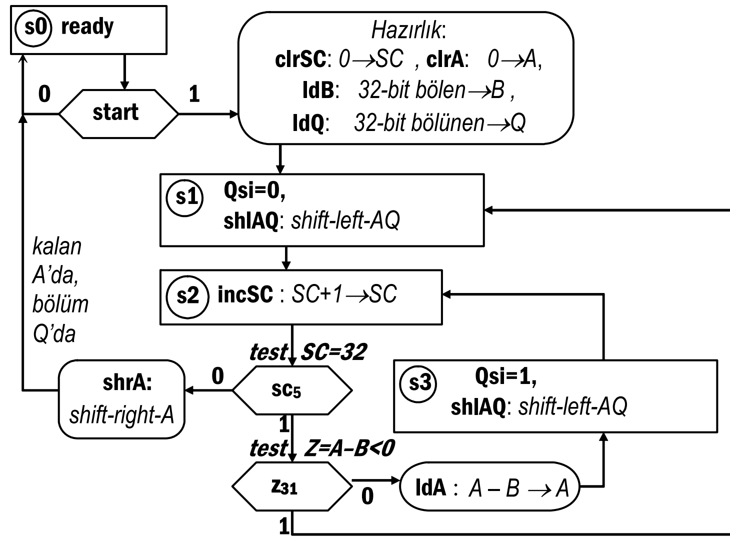


Şekil 4-32 Üçüncü bölmeye eşdeğer algoritma



Şekil 4-33 Üçüncü bölme için veri işleme donanımı

ASM çiziminde döngüye girmeden yazmacı *A*=0 ve *A*–*B* her durumda negatif olduğundan AQ koşul aranmaksızın kaydırılıyor. Döngüdeki ASM öbeği içinde *A*–*B* nin işaret biti döngü-çıkış-koşulundan sonra sınadığından, *SC*=32 olduğunda bölünenin son biti de işlenmiş oluyor.



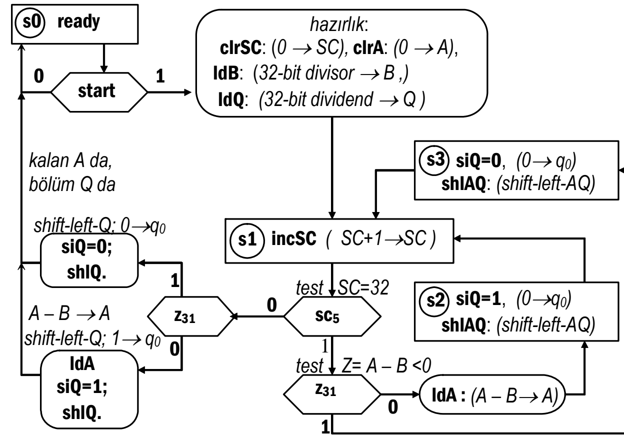
Şekil 4-34 Üçüncü-bölücünün ASM çizimi

Tablo 4-13 1011_2 / 0011_2 bölümünü 4-bitlik üçüncü-bölücü ile izlenmesi.

Durum	koşul	olay	SC	B	A	Q	Qsi	Z=A–B
s0		**ready**	**...**	**...**	...	...		
s0	**start**	**ready, clrSC, clrA, ldB, ldQ**	**000**	**0011**	0000	1011		
s1		**shlAQ**			0001	0110	0	**1**110
s2	**z3**	**incSC,**	**001**					
s1		**shlAQ,**			0010	1100	0	**1**111
s2	**z3**	**incSC,**	**010**					
s1		**shlAQ**			0101	1000	0	**0**010
s2		**incSC, ldA**	**011**		0010			
s3		**Qsi, shlAQ**			0101	0001	1	**0**010
s2		**incSC, ldA**	**100**		0010			
s3		**Qsi, shlAQ**			0100	0011	1	**0**001
s2	**sc2**	**incSC, shrA**	**101**		0010			
s0		**ready**			**0010**	**0011**		

Kalan A=2; bölüm B=3.

Alıştırma: İlk bitin sıfır olma sorunu veriyollarını değiştirmeksizin döngünün en son turunda yalnızca Q yazmacı sola kaydırılarak ta çözülebilir. Aşağıdaki ASM çizimini 1011_2 / 0011_2 bölümü üzerinde izleyiniz.



4.5.4 İşaretli Bölme

Doğru sonuçlu bir tamsayı bölmede, bölünen N, bölen D, bölüm Q ve kalan R arasında şu ilişki geçerlidir

Bölünen / Bölen = Bölüm + (Kalan / Bölen),

$N/D=Q+R/D$

yada

Bölünen = (Bölüm × Bölen) + Kalan.

$N = Q \times D + R$

Pozitif sayı ile pozitif sayı bölmesinde kalanın da pozitif olması beklenir

Örnek: 7 / 3 bölmesinde, $7 = (2 \times 3) + 1$; $Q=2$, ve $R=1$.

Negatif sayının pozitif sayı ile bölünmesinde bölüm kalanın işaretine bağlıdır

Örnek: –7 / 3 işleminde,

$-7 = (-2 \times 3) - 1$; $Q= -2$, ve $R= -1$.

$-7 = (-3 \times 3) + 2$; $Q= -3$, ve $R= 2$.

Bununla birlikte, tamsayı aritmetiğinde ve bilgisayar uygulamalarında kabul gören genel uzlaşıma göre **kalanın ve bölümün işaretleri herzaman aynı türden olmak zorundadır**. Bu uzlaşım altında, *R* ve *Q* ters işaretli olduğundan *Q*= –3 ve R=2 kabul edilebilir sonuçlar değildir.

Bölme algoritması işaretli bölme yapacak biçime dönüştürülebilir, ancak, genel uygulama pozitif sayılarla işaretsiz bölme kullanmak ve gerekiyorsa bölümün işaretini terslemek şeklindedir.

4.6 Gerçek Sayılar ve Kayan-Noktalı Aritmetik

Bir gerçek sayı, tamsayı ve kesirli kısımlardan oluşur. Kesirli kısımda kullanılan basamak sayısı gösterimin kesinlik derecesini, tamsayı kısmın basamak sayısı ise yazılabilecek sayiların büyüklüğünü belirler.
bir seri toplamı olarak tanımlanan

pi = 3.14159265358...... yada

e = 2.71828......

gibi transendental sayıların tam gösterimleri sonsuz sayıda kesir basamağı gerektirir. Ancak, sonlu sayıda bitle çoğu mühendislik ve bilimsel hesaplamalarda yeterli olacak kesinliği sağlamak mümkündür.

Gerçek sayıların önceden belirlenmiş sayıda tamsayı ve kesir basamaklarıyla gösterilmesine *sabit noktalı gösterim* **(***fixed point notation***,** *FixP*) denir**.**

Ölçme ve donatım dalında kesinlık bir dizi müstakil ölçüm değeri yada sonucu arasındaki uyum derecesidir. Kesinliğin aksine, hassaslık ölçülen veya hesaplanan bir değerin gerçek yada olması gereken değerine yakınlığının derecesidir.

Çözünürlük bir ölçüm aletinin farkedebilecegi en yakın iki ayrı değer arasındaki farktır. Kesinlik cinsinden söylersek, çözünürlük bir

niceliğin belirlenebileceği ya da okunabileceği kesinlik derecesidir. Yüksek çözünürlük her zaman yüksek hassaslık anlamına gelmeyebilir.

Bilgisayar bilimlerinde çözünürlük ve kesinlik hemen hemen eşit değerler arasında farkı ayırdetme yeteneğidir. Örneğin üç basamaklı ondalık sayı 1000 olabilirlikten birini ayırdeder. Gösterimin çözünürlük veya kesinliği ya gösterilen değer cinsinden, ya da erim aralığına oranla betimlenir.

Örnek 4-11: pi gerçek sayısı 16-bitlik yazmacın 8-biti tamsayı kalan 8-biti kesirli kısım için (8.8-bit FixP) kullanarak aşağıdaki gibi gösterilebilir

tamsayı kısım		kesirli kısım	
3	**= 1 1**	**.14159265358..**	
1	**1 (LSB)**	**0.28318530716..**	**0 (MSB)**
0	**1 (MSB)**	**0.56637061432..**	**0**
		.13274122864..	**1**
		0.26548245728..	**0**
		0.53096491456..	**0**
		.06192982912..	**1**
		0.12385965824..	**0**
		0.24771931648..	**0 (LSB)**

$pi_{(8.8\text{-bit FixP})} = 0000\ 0011.0010\ 0100_2 = 3+1/8+1/64 = 3.140625_{10}$

Bu gösterimin mutlak kesinliği U = 1/256 = 0.00390625 dir.

Bu gösterimin göreceli kesinliği

Ur = 1/256 /(127.999 – (–128))= 0.00001526 dir.

Gösterim hatası ise (pi – 3.140625) ≈ 0.00096765

FixP gösterimin dezavantajı küçük sayıların gösteriminde Ur göreceli çözünürlüğün (veya $U_{\%}$ çözünürlük yüzdesi) hızla düşmesidir.

Örnek 4-12: A= 152.005_{10} ve B=0.025_{10} sayılarını 8.8-bit FixP gösterimini ele alalım.

$A_{(8+8\text{-bit FixP})} = 10011000.00000001_2$ ve

$B_{(8+8\text{-bit FixP})} = 00000000.00000110_2$.

Her iki durumda da mutlak çözünürlük ya da kesinlik

$U(A) = U(B) = 1/256 \approx 0.004$.

A nın gösterimi için kesinlık veya çözünürlük yuzdesi

$U_{\%}(A)$ = mutlak-kesinlik / anlamlı-büyüklük ×100 %

= (1/256) / 152.005 × 100 % ≈ 0.0025 %.

Ancak, B için

$U_{\%}(B) = (1/256)/0.025 \times 100 \approx 15.6$ % .

B -nin çözünürlük yüzdesindeki kayıp yazmaçtaki anlamlı bit sayısının azalmasından kaynaklanmaktadır.

Diğer bir deyişle 8-bit-tamsayı + 8-bit-kesir gösterimde, A sayısı 0.0025 % lik adımlarla artırılabilirken, B ancak kendisinin 15.4% -ü adımlarla artırılabilir.

Bilimsel gösterim ve *kayan noktalı kodlu sıkıştırma,* değişmez sayıda anlamlı-basamak (figür) kullanır ve çok daha büyük sayı aralığında neredeyse sabit kesinlik yüzdesi sağlar. Bilimsel gösterim üç bileşenden oluşur, *işaret* **S**, *anlamlı-basamaklar* (kısaca *anlamlılar*) **F**, ve *üst* **E**. Anlamlı-basamaklar herzaman pozitiftir. Sayı, kesir noktasının solunda sadece ilk basamakta bir olacak biçimde kaydırılıp sayının üstü buna göre ayarlanarak normalize edilir.

Örnek 4-13: Aşağıdaki gerçek sayıları 2-anlamlı-basamaklı bilimsel gösterime normalize ediniz

$3.1415 = 3.1\times10^{0}$ = +3.1 E0   işaret, anlamlılar, üst

$60000000 = 6.0\times10^{7}$ = +6.0 E+7

$-23.75\times10^{-23} = -2.4\times10^{-22}$ = – 2.4 E–22 (2 basamağa yuvarlandı)

$-0.02\times10^{23} = -2.0\times10^{21}$ = – 2.0 E+21

Örnek 4-14: A = 152.005 ve B = –0.025 sayılarını 6 anlamlı-basamaklı bilimsel gösterimle yazınız ve her duruma karşılık çözünürlük yüzdesini bulunuz.

<<çözüm

$$A = 1.52005 \times 10^{2}, \text{ and } B = -2.50000 \times 10^{-2}.$$

A ve B sayıları için çözünürlük yüzdeleri :

$$U_{\%}(A) = 0.00001 / 1.50005 \times 100 = 0.0005\ \%$$

$$U_{\%}(B) = 0.00001 / 2.50000 \times 100 = 0.0004\ \%.$$

>>

N-basamaklı-bilimsel gösterimde sayı milyarlarca kat değişse bile gösterimin çözünürlük yüzdesi en çok on kat değişebilir.

$$10^{-N+3}\ \% > U_{\%} > 10^{-N+2}\ \%,$$

böylece, N=6 için, $0.001\ \% > U_{\%} > 0.0001\ \%$.

Kayan noktalı (Floating point - FP) gösterim ikili sayıların bilimsel gösterimidir. FP gösterimin biçimi anlamlılar ve üstel alanlarının genişliği ile negatif üstleri gösterme uzlaşımına bağlıdır.

Bir *kayan noktalı sayı biçimi* üç binary-alandan oluşur: *işaret-biti* alanı *S*, *ikilik-üst E*, ve *ikilik-anlamlılar (*ya da *kesir) F*. Sıfırdan farklı bir *R* sayısının değeri *S*, *E* ve *F* değerlerinden şu ifadeyle hesaplanır:

$$R = (-1)^{S} \times F \times 2^{E},$$

Sıfırdan farklı tüm sayılarda *F* öncül bir ve kesirden oluşur (böylece daima, $1 \leq F < 2$ olur).

Örnek 4-15: *A*= 152.005 ve *B*=–0.025 sayısının işaret *S*, kesir *F*, ve üst *E* değerini bulun. *F* için 16-bit, *E* için 8-bitlik ikilik sayılar kullanın.

<<çözüm

$A = 152.005 = 10011000.00000001_2$,

$A = (-1)^0 \times 1.0011\ 0000\ 0000\ 001_2 \times 2^7$,

S = 0; (pozitif)

F -nin mutlaka *öncül-biri*, ve *kesiri* olmalıdır.

$F = 1.0011\ 0000\ 0000\ 001_2 \approx 152.005 / 2^7 \approx 1.187539...$;

Sayının ikilik biçiminde öncül-bir elde etmek için kesir noktası 7 bit sağa kaydırıldı. Her kayma 2 ile bölmeye eşdeğer olduğundan toplam etki $2^{-7} = 1/128$ ile bölmeye denktir. Bu bölünme etkisi kesiri 2^7 değerinde bir üst terimi ile çarparak giderilir. Böylece

$E = 00000111_2 = 7$ olur.

$B = -0.025 = -\ 00000000.00000110\ 0110\ 0110\ 0110\ \underline{0110}_2$

$B = (-1)^1 \times 1.1001\ 1001\ 1001\ 100_2 \times 2^{-6}$,

$S = 1$; (negatif) $E = -6 = 1111\ 1010_2$.

$F = 1.1001\ 1001\ 1001\ 100_2 \approx 0.025/2^{-6} \approx 1.6$

>>

Örnek 4-16: 24-bitlik bir standart dışı kayan noktalı gösterim biçiminde 1-bit işaret *S*, 15-bit anlamlılar *F*, ve işaretli ikilik gösterimde 8-bitlik bir *E* üstü vardır. Bu gösterimde sayı $R = (-1)^S \times F \times 2^E$ olarak temsil edilmektedir. *F* anlamlıları hem öncül-biri hem de kesiri olacak şekilde yazılmaktadır. *A*= 152.005 (*S*=0; $F = 1.0011\ 0000\ 0000\ 00_2$; ve $E = 7 = 0000\ 0111_2$) sayısını ikilik-kayan-nokta gösteriminde yazınız.

<< A şöyle gösterilir

$$A_{(24\text{-bit-FP})} = \underset{0}{\underline{S}}\ \underset{100\ 1100\ 0000\ 0000}{\underline{F\qquad\qquad}}\ \underset{0000\ 0111}{\underline{E\qquad.}}\ _2 .$$

>>

Bu örnekte, *A* -nın en sağdaki basamağı istenilen FP-biçiminin anlamlılar alanına sığamadığından gösterdiği değer 152 oluyor.

Bu 24-bit gösterimde saklanabilecek en büyük pozitif sayı şöyle bulunur:

$$N_{maximum} = 1.1111\ 1111\ 1111\ 111_2 \times 2^{01111111_2}$$
$$= 2 \times 2^{127} - 1 \times 2^{127-15},$$
$$\approx 2^{128}.\ \ (2^{127-15} \text{ sayısı } 2^{127}\text{-den } 2^{15} \text{ kat küçüktür})$$
$$\approx 3.4 \times 10^{38}$$

4.6.1 IEEE-754 Kayan-Noktalı Sayı Standardı

Kayan-noktalı sayı biçimleri arasında günümüzde en geniş kabul gören standart *IEEE-754 gösterimidir* . IBM, SUN, HP, ve Intel dahil olmak üzere yaygın kullanılan çoğu bilgisayarda uygulanır. MIPS *kayan-noktalı-yanişlemci birimi (FPU)* IEEE standardını kullanır. IEEE-754 standardında üst değerleri *eğilimli-sayı* (biased number) olarak yazılır. Üstleri böyle eğilimli-sayı biçiminde yazmak FP sayıların kolay sıralanmasını sağlar. IEEE-754 iki farklı kesinlik için iki tip kayan noktalı gösterim tanımlar.

Tek-kesinlikli kayan sayılar

Tek-kesinlikli-sayı-biçimi 32-bit gerektirir ve kısaca *kayan* olarak anılır. Tek kesinlikli FP biçim aşağıdaki alanlardan oluşur:

b31	1-bit	S	işaret-biti (1:negatif, 0:pozitif)
b23–b30	8-bit	Es	eğilimli-üst (Es = E+127)
b0-b22	23-bit	Fs	ikilik-anlamlı-basamaklar (Fs = F–1)

bit:	31	30	29	28	27	26	25	24	23	22	21	20	..	..	..	2	1	0
alan:	S	Eğilimli-Üst Es=E+127 (8 bit)								İkilik-Anlamlılar Fs=F–1 (23 bit)								

Burada

S, sayı negatifse bir, pozitifse sıfırdır,

Es = *E*+127 –126 -dan 128 -e kadar üst değerlerini gösteren *eğilimli-üsttür*, (*Es*=0 sıfır sayısı için ayrılmıştır).

Fs = *F*–1 sayının normalize edilmiş kesirindeki öncül-bir atıldıktan sonraki ikili-anlamlılardır,

Gösterilen kayan-sayının değeri $R = (-1)^S \times (Fs+1) \times 2^{Es-127}$ dir

IEEE-754 tek-kesinlikli-FP-alanları	Gösterdiği Sayının değeri
S(1-bit), *Es*(8-bit), *Fs*(23-bit)	$R = (-1)^S \times (Fs+1) \times 2^{Es-127}$

IEEE-754 kayan-sayılarda, *Es*=0 (yada *E*= –127) sıfır sayısı için ayrılmıştır, Böylece sıfır sayısı, tüm alanlar sıfırlanarak (32 bitin hepsi sıfır) gösterilir. En küçük pozitif sayı R_{min} için *Es*=1 (*E*= –126) ve *Fs*=0, (*F*=1) -dir. Bu $1\times 2^{-126} = 1.17\times10^{-38}$ -e denk gelir. *Es*=0 *R*=0 -ı göstermek için ayrıldığından 2^{-126} dan küçük sayılar *alttan-taşma*ya

neden olur. En büyük sayı R_{max} için Es=255, (yada E=+128) ve $Fs=1-2^{-23} \approx 1$ ($F\approx 2$) -dir. 8-bit üst alanı Es için yeterince geniş olmadığından 2×2^{128} in üzerindeki sayılar yazılamaz ve *taşma* durumuna neden olur. Bulunan sınır $2\times 2^{128} \approx 6.8\times 10^{38}$ -e denk düşer.

Örnek: A= 152.005 ve B= –0.025 sayılarını IEEE-754 ikilik-kayan biçimde yazın.

$A = 10011000 . 00000001_2$

$A = -1^0 \times 1.001\ 1000\ 0000\ 0001_2 \times 2^7$,

$S = 0$;

$F = 1.0011\ 0000\ 0000\ 001_2 \approx 152.005/2^7 \approx 1.187539...$;

$Fs = .0011\ 0000\ 0000\ 0010_2 \approx F-1 = 0.187539....$

$E = 7$; $Es = 7 + 127 = 128+6 = (1000\ 0110)_2$

	S	Es	Fs
A =	0	1000 0110	0011 0000 0000 0010 0000 000

$B= -00000000 . 00000110\ 01100110\ \underline{0110}_2$

$B = -1^1 \times 1.100\ 1100\ 1100\ 1100_2 \times 2^{-6}$,

$S = 1$;

$F = 1.100\ 1100\ 1100\ 1100_2 \approx 0.025/2^{-6} \approx 1.6$

$Fs = .100\ 1100\ 1100\ 1100_2 \approx 1.6 - 1 \approx 0.6$

$E = -6$. $Es = -6 + 127 = 121 = 0111\ 1001_2$

	S	Es	Fs
B =	1	0111 1001	100 1100 1100 1100 1100 1100

Çift-kesinlikli kayan sayılar

Kayan-noktalı gösterimde, üst alanının genişliği erim aralığını belirlerken anlamlılar alanının genişliği ise kesinliği sağlar. Tek-kesinlikli kayan gösterim (tek-kayan (single-float), veya kayan sayı da denir) 23 bit kesinliğe sahiptir, bu da yaklaşık 7-anlamlı-ondalık-basamak kesinliğe denktir. Tek kesinlikli biçimde yazılabilecek en büyük sayı $2^{129} = 6.8\times 10^{38}$ -dir. IEEE 754 çift-kesinlikli-kayan gösterimde hem kesinlik 53-bite (16-ondalık-anlamlı-basamak) genişler, hem de kayan-noktalı sayıların erim aralığını artırır (10-bit üst, $R_{max} \approx 10^{308}$). en-sağ bitten en-sola doğru, çift kesinlikli biçim aşağıdaki alanlara sahiptir:

W1: Sol Sözcük	b_{31}	1-bit	S	işaret (0:pozitif, 1:negatif)
	b_{20}–b_{30}	11-bit	Ed	eğilimli üst ($=E+1023$)
	b_0-b_{19}	52-bit	Fd	ikilik-anlamlı bitler ($Fd=F-1$)
W0: Sağ Sözcük	b_{31}-b_0			

W1 :Sol -Sözcük											W0: Sağ Sözcük						
31	30	29	....	22	21	20	19	18	....	0	31	30	...	...	2	1	0
S	*Ed*=*E*+1023 Eğilimli-üst (11 bit)						*Fd*=*F*–1 İkilik-anlamlılar (52 bit)										

burada, S sayı negatif ise bir olan 1-bitlik işarettir,

Ed = *E*+1023 *eğilimli-üst* , –1022 ile 1024 arası üstleri gösterebilir,

(*Ed*=0 yada *E*= –1023, sıfır sayısını yazmaya ayrılmıştır).

Fd = *F*–1. (normalize sayının öncül-bir atıldıktan sonraki ikilik anlamlılarıdır).

Gösterilen sayının değeri

$R = (-1)^{S} \times (Fd+1) \times 2^{Ed-1023}$ ile bulunur.

IEEE-754 çift-kesinlikli-FP-alanları	Sayının değeri
S (1-bit), *Ed* (11-bit), *Fd* (52-bit)	$(-1)^{S} \times (Fd+1) \times 2^{Ed-1023}$

Örnek: Daha önce yazdığımız A =150.005 tek-kayan sayıyı çift-kayana çevirin:

	S	*Es*	*Fs*
$A_{\text{(tek-FLP)}}$=	0	1000 0111	0011 0000 0000 0010 0000 000

<<çözüm

E= *Es*–127; *Ed* = *E*+1023 = *Es*–127+1023

=128+7–127+1023 = 8 + 1023 = 1024 +7 .

Fd = *Fs* (noktaya hizalayıp sağdaki boşluğa sıfır doldurun).

$A_{\text{(çift-FLP)}}$=				
	W0:	0	000 0000 0000	0000 0000 0000 0000 0000 000
	W1:	0	100 0000 0111	0011 0000 0000 0010 0000 000
		S	*Es*	*Fs*

Örnek: Aşağıda MIPS tek-kayan yazmacındaki R sayısının değeri nedir:

	S	*Es*	*Fs*
$R_{\text{(tek-FLP)}}$=	1	1000 0001	0100 0000 0000 0000 0000 000

<<

S =1, işaret= $(-1)^1$, negatif. *Es*=128+1, *E* = *Es*–127 = 2 ,

F = 1+*Fs* = $(1.0100 \ldots)_2$ = 1+1/4 = 1.25

$R = (-1)^1 \times 1.25 \times 2^2 = -1.25 \times 4 = -5.$

>>

Aşağıdaki tablo IEEE-754 tek ve çift formatların kesinliğini ve aralığını içerir:

	kesinlik	bit-sayısı	$\lvert R\rvert_{max}$	$\lvert R\rvert_{min}$

tek	7-basamak	*S*:1, *Es*:8, *Fs*:23	2^{129}=6.8E+38	2^{-126} =11.7E–38
çift	16- basamak	*S*:1, *Ed*:11, *Fd*:52	2^{1025}=1.8E+308	2^{-1022}=2.23E–308

4.6.2 Kayan Noktalı Toplama

Bilimsel gösterimde toplama yaparken toplanacak sayıların üstleri en büyük üste ayarlanmalıdır:

Örnek: Dört-basamaklı bilimsel gösterimde toplama

$9.999\times10^1+1.61\times10^{-1}$ ← Üstü 1 oluncaya dek 1.61 i sağa kaydır, her kaydırma için üstü bir artır
$= 9.999\times10^1 + 0.016\times10^1$,
$= (9.999 + 0.016) \times10^1$, ← üstleri ayarlanmış anlamlıları topla.
$= 10.015 \times10^1$, ← Toplamın anlamlı kısmı normalize değil, normalize et.
$= 1.0015 \times 10^2$, ← anlamlı basamaklar fazla.Yukarıya yuvarla.
$= 1.002 \times 10^2$, ← Yuvarladıktan sonraki hali. Çoğu durumda yuvarlama yerine budama uygulanır.

Algoritma:

1- Doğru bir toplama için, sayıların ondalık noktalarını hizala (yani sayıların üstleri eşitleninceye dek küçük sayının anlamlılarını sağa (ondalık noktasını sola) kaydırıp üstünü artır.

2- Hizalanmış sayıların anlamlılarını topla.

3- Toplamı normalize et.

Toplamın öncül sıfırı yoksa, anlamlılarını sola kaydırıp üstünü bir azaltmak ve her seferinde taşma ve alttan taşma durumlarını sınamamız gerekir.

4- Gerekirse sonucu yuvarla yada buda.

IEEE-754 en-yakın-çift (nearest-even) yuvarlamayı uygular, burada "bir" sıfır yanındaysa budanır, birin yanındaysa yukarı yuvarlanır (rounded-up). Ancak, “sıfır” her durumda budanır.

N= 1.01001*0011*$_2$ üzerinde dört yuvarlama yöntemini gösterelim

-yukarı yuvarlama (*round up*)

N1= 1.0101 010 ; N2= 1.0101 01 ; N3= 1.0101 1 ; N4= 1.0110 ;

-en-yakın-çifte yuvarlama (*nearest-even*)

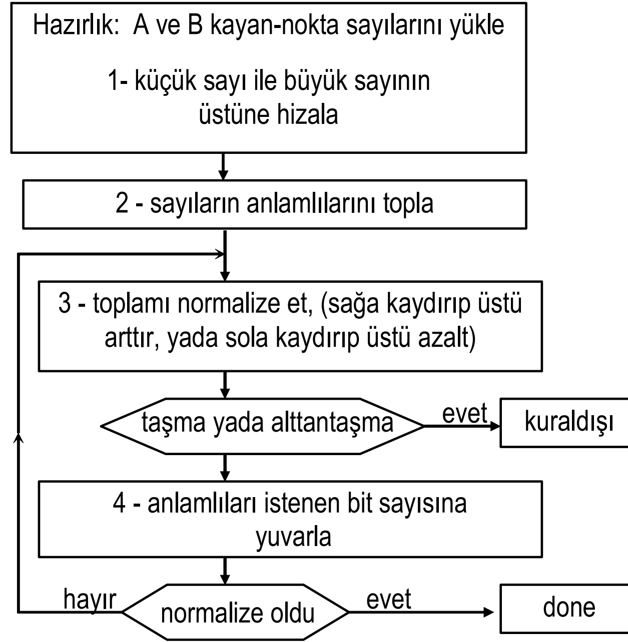
N1= 1.0101 010 ; N2= 1.0101 01 ; N3= 1.0101 0 ; N4= 1.0110 ;

-budama (*truncate*)

N1= 1.0101 001 ; N2= 1.0101 00 ; N3= 1.0101 0 ; N4= 1.0101 ;

-aşağı yuvarlama (*round down*)

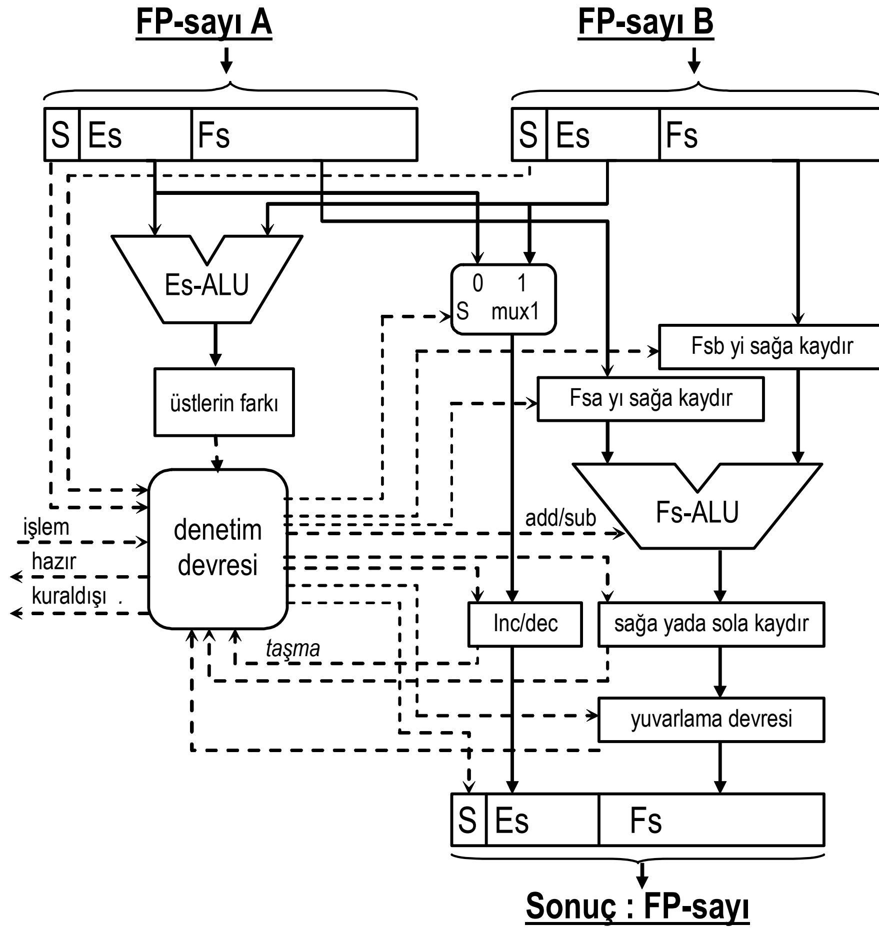
N1= 1.0101 001 ; N2= 1.0101 00 ; N3= 1.0101 0 ; N4= 1.0101 ;



Şekil 4-35 Kayan-noktalı toplama algoritması.

4.6.3 Kayan-noktalı toplama için veri işlem donanımı

Kayan-noktalı toplama devresinin veriyolu çizimi Şekil 4-36 'de gösterilmiştir. Hizalama ya Fs_A yada Fs_B -yi kaydırarak gerçekleştirilir. Yapılacak işlem işlenenlerin işaret-bitlerine bağlı olduğundan *Fs-ALU* hem *A – B* hem de *B – A* yapabilmelidir. *Fs-ALU* çıkış yazmacı sonucu hem sağa hem de sola-kaydırabilir. Normalizasyon için *Fs-ALU* çıkış yazmacındaki sonuç öncül-bir oluşana kadar sağa yada sola sola kaydırılır ve bu sırada Es_A yada Es_B nin büyük olanı artırılır yada azaltılır. Toplama ve çıkarmanın yanısıra çarpma/bölme de yapabilen bir FLP-unitesi neredeyse benzer veri yollarına sahiptir. Ancak bunun denetim algoritması yalnız toplama yapan denetimden daha çok durum ve ayrıntı gerektirir.



Şekil 4-36 Kayan-noktalı Toplama/çıkarma veriyolu.

4.6.4 Kayan -Noktalı Çarpma ve Bölme

Kayan noktalı A ve B sayılarının çarpımının (yada bölümünün) üstü Es_R çarpılanların üstleri Es_A ve Es_B -nin toplamıdır (bölmede farkı):

çarpma R=A×B		bölme R=A/B
$Es_R = Es_A + Es_B$	yada	$Es_R = Es_A - Es_B$
$Fs_R = Fs_A \times Fs_B$		$Fs_R = Fs_A \ / \ Fs_B$
$S_{-R} = S_A$ *XOR* S_B		$S_{-R} = S_A$ *XOR* S_B
her iki işlenen de aynı işaretteyse pozitif		her iki işlenen da aynı işaretteyse pozitif

Kayan-noktalı sayıların bölmesi kalan vermez.

Örnek 4-17: 4-anlamlı-basamaklı

A=1.110×10^{10} ve B= 9.200×10^{-5} bilimsel sayılarını çarpın:

<<çözüm

S_A=(+), Ea=10, F_A=1.110 ; S_B=(+), E_B=–5, F_B=9.200 ;

1. E_R: sonucun üstü,

$E_R = E_A - E_B$ = 10–5 = 5;

2. $F_R = F_A \times F_B$ sonucun anlamlıları

```
   FA             1.110
 × FB      ×      9.200
               0.0 00 000
               0.0 00 00
               0.  222 0
                9.990
      FR     =10.212 000
```

3. Sonucun normalizasyonu

F_R > 10 olduğundan, F_R -yi 10 -a bölüp, E_R -yi artırın;

10.212000 × 10^5 = 1.0212000 ×10^6 .

4. Sonucu 4-anlamlı-basamağa indirilmesi.

F_R= 1.021 ; E_R = 6 .

5. Sonucun işareti:

A pozitif, B pozitif, bu nedenle sonuç pozitif. S_R=0.

Tüm parçalar birleştirince

$R = A \times B$ = 1.021 10^6

Örnek 4-18: A= 0.5_{10} ile B= -0.4375_{10} -yi öncül-bir hariç 4-bit anlamlı ve 3-bit işaretli üst ile ikilik kayar-gösterimde çarpalım.

<< çözüm:

Sayıları 4-bit F ve 3-bit E haline dönüştürün:

A = 0.1000_2 . S_A = 0, E_A = –1 = 111 , F_A = 1.0000_2 .

B = – 0.0111_2 . S_B = 1, E_B = –2 = 110 , F_B = 1.1100_2 .

A ile B -nin çarpımı

```
1. E = EA + EB                      111
     = (–1) + (–2)                  110
     = –3 = 101₂                  1 101  taşma yok.
```

2. $F = F_A \times F_B$ = $1.000_2 \times 1.110_2 = 1.110\ 000_2$

```
         1.000
×        1.110
         0 000
        10 00
       100 0
     1 000
     1.110 000
```

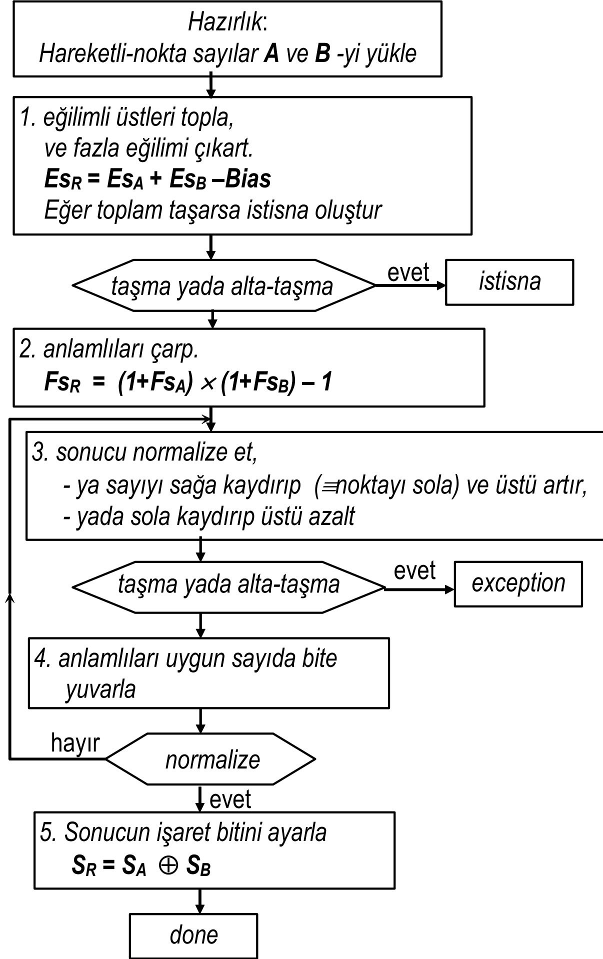
3. Normalizasyon: sonuç zaten normalize olmuş.

4. anlamlıyı 4-bite yuvarlama:
   F= 1.1100 (öncül-bir hariç *F*=.1100 yazılacak)

5. Sonucun işareti:
   $S = S_A \oplus S_B = 0 \oplus 1 = 1$, negatif.

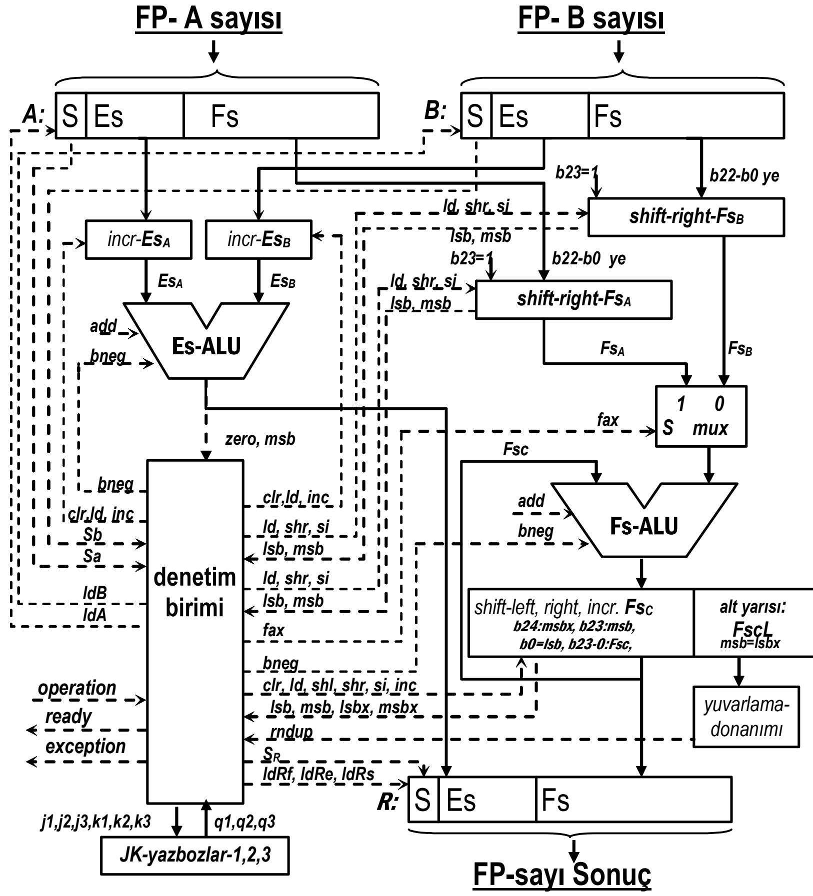
$S_R = 1$, E_R = 0b 101 , $F_R = 1.1100$; R_{FLP} = 1 101 1100

4.6.5 Kayan noktalı çarpma için algoritma



Şekil 4-37 Kayan noktalı çarpma algoritması

Kayan noktalı aritmetik işlemcilerinin veri yolu düzeni çarpma ve bölme algoritmalarını gerçekleştirmeye uygun donanımı da içermelidir. Şekil 4-38 dört işlem yapabilen hareketli-nokta işlemci ünitesi (FPU) için muhtemel bir veri işlem donanımını gösterir.



Şekil 4-38 Toplama, çıkarma, çarpma, bölme yapan hareketli-nokta aritmetik birimi için veri yolu diyagramı.

İşlenenlerin hizalanmasında kaydırma ve artırma işlemlerini de bileşimsel devrelerde yapıp bileşimsel çarpan ve bileşimsel bölen ağı devreleri kullanarak çok hızlı kayan-noktalı işlem birimleri yapılabilmektedir.

Şekil 4-38'deki tek-kesinlik FPU aşağıdaki bileşenlerden oluşur:

FP-yazmaçları *A*, *B*, ve *R* 32-bit genişlikte ve anuyumlu paralel yükleme denetim girişi (*ldA*, *ldB* ve *ldR*) olan yazmaçlardır.

Es_A ve Es_B**,** sıfırlama (*clr-Esa, clr-Esb*), artırma (*inc-Esa, inc-Esb*), ve yükleme (*ld-Esa, ld-Esb*) işlevleri olan 8-bitlik sayıcı yazmaçlarıdır.

Fs_A ve Fs_B paralel yükleme (*ld-Fsa*, *Ld-Fsb*), ve kayma-girişleri (*si-Fsa*, *si-Fsb*) olan tek-yönlü 24-bit kaydırma (*shr-Fsa*, *shr-Fsb*) yazmaçlarıdır. Hatırlarsanız Fs_A ve Fs_B alanları 23-bittir, ve 24-

bitlik Fs_A ve Fs_B yazmaçlarının en sağına hizalanmıştır. Böylece, silinmiş olan öncül-bir her zaman Fs_A ve Fs_B **-**nin b_{23} **-**üne yüklenirler.

8-bitlik *Es-ALU*, Es_A ve Es_B -dekileri toplar (*bneg-Es*=0), yada çıkarır (*bneg-Es*=1). Elde edilen sayı sonucun Es alanına aktarılabilir.

24-bitlik *Fs-ALU* anlamlıları toplar (*bneg-Fs*=0) yada çıkarır (*bneg-Fs*=1).

Fs_C sıfırlama (*clr-Fsc*), yükleme (*ld-Fsc*), çift-yönlü-kaydırma (*shr-Fsc*, *shl-Fsc*, *si-Fsc*), ve yukarı sayma (*inc-Fsc*) işlevlerini yapabilen 25-bitlik bir yazmaçtır. Fs_c -nin kaydırma işlemleri 24-bitlik Fs_{cL} yazmacıyla sağ yönde genişletilir. Fs_{cL} in en-sol biti (*lsbx-Fs*), Fs_c -nin genişletilmiş en-sağ-bitidir (extended-MSB). Fs_c -nin 24 ve 23 -üncü bitleri (*msbx-Fs*) ve (*msb-Fs*), Fsc -nin genişletilmiş en-sol-biti (extended-MSB) ve en-sol-biti (MSB) olarak adlandırılır. Fs_A+Fs_B üç adımda yerine getirilir: 1- $Fs_c \leftarrow 0$ (*clr-Fsc*); 2- $Fs_c \leftarrow 0 + Fs_A$ (*fax, ld-Fsc*) ; 3- $Fs_c \leftarrow Fs_A + Fs_B$ (*ld-Fsc*).

Yuvarlama devresi *rndup* yukarı yuvarlama sinyaline Fs_{cL} nin içindekilere göre karar verir. 24-bitlik bir 2-den-1-e çoklayıcı *Fs-ALU* girişindeki toplama/çıkarma yolunu (*asc*=1) yada çarpma/bölme yolunu seçer.

Sonucun işaret-biti (S_R), üstü (Es_R), ve anlamlı kesirl bitleri Fs_R 32-bitlik FP-yazmacına yüklenir.

F1, F2, ve F3 yazbozları algoritmanın modülerizasyonu için gereken genel amaçlı bayraklardır.

4.6.6 MIPS FPU Komutları

MIPS FP aritmetiği için genel-amaçlı yazmaçları kullanmaz. Tek ve çift kesinlikli kayan noktalı sayıları için 32 özel 32-bit yazmacı (f0, f1, ... f31) vardır. Tek numaralı yazmaçlar FP aritmetiği komutlarında kullanılamaz. Komutlarda, yazmaç-çiftlerindeki sayılar **$f0**, ... , **$f30** (tek ve çift kesinlikli sayılar için sadece 16 yazmaç) ile gösterilir. MIPS bellekle FP-yazmaç arası veri aktarımı için özel load ve store komutları kullanır. r0, ... r31 ve f0, ..., f31 arasında doğrudan veri aktarımı yoktur.

Tablo 4-14 MIPS FPU Komutları

Tek Kesinlikli Aritmetik Komutlar	**Çift Kesinlikli Aritmetik Komutlar**	
add.s dest, src1, srcr2 add.s $f1,$f2,$f3	**add.d** dest, src1, srcr2 add.d $f1,$f2,$f3	tek/çift FP toplama $f1←$f2+$f3
sub.s dest, src1, src2 sub.s $f1,$f2,$f3	**sub.d** dest, src1, srcr2 sub.d $f1,$f2,$f3	tek/çift FP çıkarma $f1←$f2–$f3
mul.s dest, src1, src2 mul.s $f1,$f2,$f3	**mul.d** dest, src1, srcr2 mul.d $f1,$f2,$f3	tek/çift FP çarpma $f1←$f2×$f3
div.s dest, src1, src2 div.s $f1,$f2,$f3	**div.d** dest, src1, srcr2 div.d $f1,$f2,$f3	tek/çift FP bölme $f1←$f2/$f3
Tek kesinlikli aktarma komutları	**Çift kesinlikli aktarma komutları**	
l.s dest, ofset(base) l.s $f1,2000($3)	**l.d** dest, offset(base) l.d $f1,2000($3)	tek/çift FP load(yükle) $f1 ← Hafıza(2000 + $3)
s.s src, ofset(base) s.s $f1,2000($3)	**s.d** src, offset(base) s.s $f1,2000($3)	tek/çift FP store(sakla) Hafıza(2000 + $3) ← $f1
karşılaştır ve dallan		
c.xx.s src1, src2	xx yerinde eq, neq, lt, le, gt, ge olabilir	tek-FP compare sonuç cond-flag -i set eder
c.xx.d src1, src2	xx yerinde eq, neq, lt, le, gt, ge olabilir	çift-FP compare sonuç cond-flag -i set eder
bclt adres	PC -ye göre adresleme	Şart set iken branch
bclf adres	PC -ye göre adresleme	Şart clear iken branch

Örnek 4-19: Şu C-dili ifadeleri için

```
float a, b, c;
...
if(a<c) { a= a + b;}
```

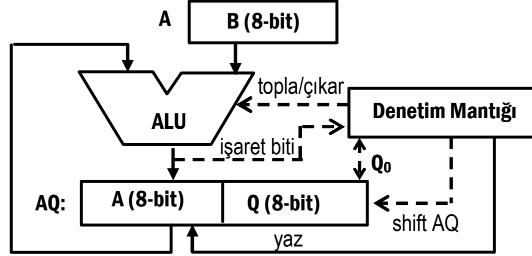
derleyici bellekte **mema**, **memb**, **memc** etiketli yerlerde üç sözcük için yer açar. Toplama ifadesine karşılık gelen çevirici kodu:

```
        l.s $f0,mema($0)  # $f0 = a
        l.s $f2,memc($0)  # $f2 = c
        c.lt.s $f0,$f2
        bclf endif
        l.s $f4,memb($0)  # $f4 = b
        add.s $f0,$f0,$f4
        s.s $f0,mema($0)
endif:  ...
```

Dikkat ederseniz FP-koşullu-dallanma (*branch on condition*) komutu sonuca göre koşul-bayrağını (*condition-flag*) birleyip sıfırlayan FP-karşılaştırma komutunun (*FP-compare*) ardından kullanılmalıdır.

4.6.7 Çözümlü Problemler

Q1) Aşağıdaki diyagram ve algoritmalar 8-bitlik işlemcinin çarpıcı/bölücü donanımlarıdır, ancak taslaktaki bazı önemli ayrıntılar unutulmuştur.



Çarpma Algoritması:
hazırl: *B* = çarpılan ;
Q= çarpan; *A*=0; *SC*=0;
adım1: Eğer (Q_0=1) ise
{*A*+*B* yi *A* ya yaz;}
adım2: *SC* yi artır ve *AQ* yu kaydır
adım3: Eğer ($SC<N_m$) ise
{adım1 -e git}
adım4: çarpım=*AQ* sonucunu döndür.

Bölme Algoritması:
hazırl: *B* = bölen ;
Q= bölünen; *A*=0; *SC*=0;
adım1: *AQ* yu kaydır, *SC* yi artır.
adım2: Eğer (*A* – *B* >= 0) ise
{ Q_0=1 yap; *A* – *B* -yi *A* -ya yaz;}
adım3: Eğer ($SC<N_d$) ise
{Adım1 -e git}
adım4: kalan=*A* bölüm=*Q* sonuçlarını döndür.

a) Çarpma ve bölme için kaymaların yönü ne olmalıdır ?
i) çarpma algoritması? << *sağa* >>
ii) bölme algoritması? << *sola* >>
8-bitlik doğru çarpma için N_m değeri ne olmalıdır? <<N_m= *8*>>

b) i) bölen=0010_2; ve bölünen=0101_2 değerlerini kullanarak algoritmanın işleyişini izleyin.

Adım	SC	B	AQ	A – B
init	*0*	*0010*	*0000 0101*	*1110*

<<çözüm

Step	SC	B	AQ	A – B
init	*0*	*0010*	*0000 0101*	*1110*
1	***1***		***0000 1010***	***1110***
2,3,1	***2***		***0001 0100***	***1111***
2,3,1	***3***		***0010 1000***	***0000***
2			***0000 1001***	***1110***
3,1	***4***		***0001 0010***	***1111***
2,3,4			***0001 0010***	
			R Q	

\>>

ii) 8-bit bölme için N_d değeri ne olmalıdır? <<N_d = *8*>>

c) Gerçekleştirmede, her iki algoritmanın da hazırlık ve adım-3 aşamaları saat dönüşü gerektirmez. Tüm diğer adımların herbiri bir saat dönüşünde çalışıyor kabul edin.
i) 8-bit çarpma için gereken toplam saat döngüsü ne kadardır?
<< *2x8 = 16* >>
ii) 8-bit bölme için gereken toplam saat döngüsü ne kadardır?
<< *2x8 = 16* >>
iii) Eğer koşulu sağlanmayan sınamalar (eğer komutları) hiç saat dönüşü gerektirmezse, çarpma en az ve en çok kaç saat döngüsü tutar?

.<< en çok *16, (çarpan=0),* en az *8* >>

Q2) A ve B sayılarını kullanarak:

Nr.	Ondalık	Binary
A=	25.525	11001.10000110011001100110011001100110011001100110011
B=	–0.0859375	–0.0001011

a) A ve B -yi ondalık bilimsel kayan noktalı formatında 4 basamak kesinlikle yazın:

A: <<*$2.553x10^1$ or 2.553E+1* >>	B: << – *$8.594x10^{-2}$ or –8.594E–2* >>

b) i) A -nın tek-kesinlikli kayan-noktalı formatının her alanı için değerleri bulunuz:

<< *İşaret=0; Üst = 4+bias= 131; Anlamlılar=100 1100 0011 0011 0011 0011* >>

ii) A -nın tek-kesinlikli kayan formatının ikili formunu doldurun:

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MSW	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*1*	*1*	*1*	*0*	*0*	*1*	*1*	*0*	*0*	*0*	*0*	*1*	*1*	*0*	*0*	*1*	*1*	*0*	*0*	*1*	*1*	*0*	*0*	*1*	*1*
LSW																																

c) i) B -nin çift-kesinlikli kayan-noktalı formatının her alanı için değerleri bulunuz:

<< *İşaret=1; Üst = – 4+eğilim= 1023-4 =1019* >>;

<< *anlamlılar =011 0 0000 0000 0000 .... 00 0000 (birlerden sonra 49 sıfır)*>>

ii) B -nin çift-kesinlik float formatının ikilik formunu doldurun:

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MSW	*1*	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*0*	*1*	*1*	*0*	*1*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
LSW	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

d) Aşağıdaki tek-kesinlikli kayan-noktalı ikilik sayıları ele alın:

Sayı	İkilik değer
A=	$1.111_2 \times 2^{-30}$
B=	$1.101_2 \times 2^{-102}$

A ve B -yi algoritmaya uygun olarak çarpın ve tek-kesinlikli $P = A \times B$ kayan-noktalı çarpımı bulun. Algoritmanın her adımını açıkça gösterin.

<<

adım1. üstleri topla	*Ep= Ea+Eb =-30-102=-132*
adım2: anlamlıları çarp	*Sp =Sa Sb = 1.111 x 1.101= 1.1000011*
adım3. normalize et ve taşma ve alttan taşma durumlarını sına	*anlamlılar zaten normalizedir, üst alttan-taşıyor . (-132 < -126) Algoritma kuraldışına yol açar.*

$A \times B = 1.111 \times 1.101 \times 2^{-30-102} = 1.1000011 \times 2^{-132}$

üst–126 -dan küçük, sayı tek-kesinlikli üst alanından alttan taştı

>>

Q3)

Tüm 4-bitlik işaretsiz giriş kombinasyonları için 8-bitlik P=AB çarpımını doğru yapan bir çarpıcı ağını ele alalım. 2-lik tümleyen biçimindeki A= –3 = 1101_2, B =2 =0010_2 sayılarının bu çarpıcıyla nasıl çarpıldığını izleyerek açıklayın.

a) İşaretsiz ikilik ve karşılığı ondalık sistemde bu çarpıcıyla bulunan P çarpımı sonucu kaç çıkar?

<< işaretsiz ikilik
P=1101x0010= 00011010_2
>>

<< ondalık
P= (16 – 3) x 2 = 26
>>

b) Çarpımın sonucu işaretli ikilik sayı olarak kaçtır.

<<
işaretli ikilik formatta 8-bitlik 00011010_2 *sayısı gene 26 eder.*
8-bit için sonuç yanlıştır. Ancak, sağ-dört bit, 4-bitlik işaretli çarpımı doğru verir. 1010 = –6
>>

c) Çarpan işaretli sayılar için doğru çalışır mı?

<< *Hayır, 4-bit sayıların çarpımı 4-bite sığmadığında çarpamaz. Sadece eğer çarpımın işaret ve büyüklüğü çarpım yazmacının sağ 4-bitine sığıyor ise son dört bit doğru sonucu verir.* >>

d) Aşağıdaki algoritma P= X Y işaretli çarpmasını hesaplamak için yeterli midir?

adım1. Eğer X_{msb} *=1 ise A= X -in 2-lik tümleyeni, değilse A=X;*
adım2. Y_{msb} *=1 ise B= Y -nin 2-lik tümleyeni, değilse B=Y;*
adım3. İşaretsiz çarpıcıyla P=A B -yi hesapla;
adım4. X_{msb} *=1 ise P= P -nin 2-lik tümleyeni,*
adım5. Y_{msb} *=1 ise P= P -nin 2-lik tümleyeni,*

Algoritmayı X= – 3, Y= 2 ; ve X= – 3, Y= – 2 sayılarına uygulayarak izleyin;

<< *X= –3 = 1101 , Y= 2= 0010 ;*
1: A=0011 ; 2: B= 0010; 3: P= 00000110 ; 4: P=11111010; 5: işlem yok.
Sonuç P= 11111010 = – 6, doğru.

X= –3 = 1101 , Y= –2= 1110 ;
1: A=0011 ; 2: B= 0010; 3: P= 00000110 ; 4: P=11111010; 5: P= 00000110;
Sonuç P= 00000110 = 6, doğru.
>>

e) Aşağıdaki Boole işlevini hesaplayan bileşimsel devre kullanılarak (b) -deki algoritmanın geliştirilmiş halini yazın.

$$F= (X_{msb})' Y_{msb} + X_{msb} (Y_{msb})'$$

<<
adım1. X_{msb} *=1 ise A= X -in 2-lik tümleyeni, değilse A=X;*
adım2. Y_{msb} *=1 ise B= Y -nin 2-lik tümleyeni, değilse B=Y;*
adım3. İşaretsiz çarpıcıyla P = A B -yi hesapla;
adım4. Eğer ($(X_{msb})' Y_{msb} + X_{msb} (Y_{msb})'$ *=1) ise P= P -nin 2-lik tümleyeni;*
>>

4.7 Anuyumlu Veri-İşleme ve Durum Makineleri

Bir bilgisayar denetim ve mikro-komut birimleri, aritmetik mantık birimi, çarpıcı, kayan noktalı işlemcileri, gibi birçok durum makinesi biriminden oluşan bir sırasal durum makinesidir. Bu birimlerin işlemesi genellikle algoritmalarla betimlenir. Bir *algoritma* takip edildiğinde veri-işleme devresinde bir problemin çözümüne götüren bir sonlu adımlar kümesidir. Bir *durum makinesi, durum* denilen bir dizi adımı izler.

Algoritmik Durum Makinesi çizimi (ASM-chart) özellikle sayısal donanımları ve algoritmalarını tanımlamak için geliştirilmiş bir akış şemasıdır (flowchart). Donanım algoritmasının akış şeması, sözel ifadeyi işlemler dizisine ve bunların çalışması için gerekli durumları sıralandıran çizimsel ifadeye dönüştürür.

ASM, yazmaç düzeyindeki durum makinelerinin sayısal devre tasarımı için algoritmadan başlayarak sistematik bir yöntem sağlar. ASM çizimi her durum için veri yolunda *durumların geçişini*, ve veri yolundakı *donanım birimlerinin her bir durumdakı davranışlarını* betimler.

ASM çizimini gerçeklemek için hem ASM çiziminde betimlenen *veri-işlem devresini* hem de algoritmanın "*fiziksel-bağlantılı programı"* sayılan *denetim biriminin* gerçeklemek gerekir.

4.7.1 Durum makinelerinde Veri İşlem ve Denetleç Birimi

Yazmaçtan-yazmaca veri işleme amaçlı bir durum makinesi iki ana kısımdan oluşur, veri işlem donanımı ve denetim birimi. Veri işlem birimi yada veri yolu (*datapath*) bitleri kaydıran, yukarı ve aşağı sayan yazmaçlar, yada n-bitlik veri işleyebilen ALU, ile çeşitli veri-işleme elemanları arasındaki veri yollarını değiştirebilen çoklayıcı ve arakat gibi bileşimsel devreler içerebilir.

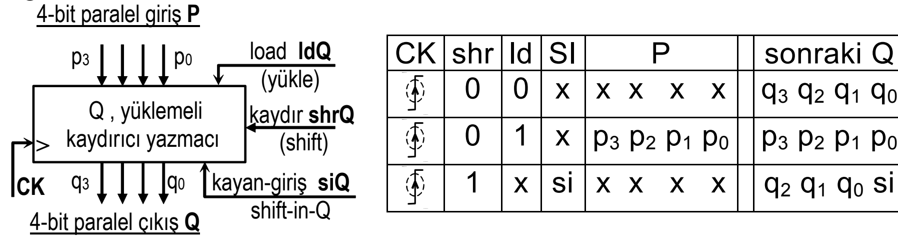
Denetim birimi ASM çiziminde betimlenen durumları oluşturmak için yazbozlar içerir. Bu yazbozların giriş devreleri bir sonraki saat dönümünün doğru durumunu oluşturur. Ve, veri işleme elemanları için denetim sinyallerini yada denetimın dış çıkışlarını oluşturmak için bileşimsel devreler içerir. En basit gerçekleme durum-makinesinin her durumu için bir D-yazbozu kullanılarak sağlanır. Bu metot genellikle “tek-uyaranlı-tasarım” (one-hot-design of the controller) diye anılır.

ASM ünitelerinin veri işleme kısımlarında birçok çeşit sırasal yazmaç kullanılır. Aynı çeşit yazmaçlar bile eğer negatif kenarlı saat çalışması

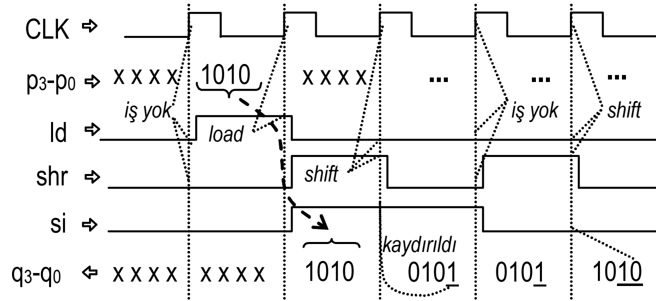
anuyumsuz sıfırlama (clear) yada yükleme (load), gibi değişik denetim girişlerine sahipse gözle görülür zamanlama farkı doğurabilir.

Bu bölümde önce ASM çizimlerimizde kullanmayı düşündüğümüz sırasal kaydırıcı yazmaçları (shift register) ve sayıcıları (counter) tanıtacağız.

En basit 4-bit anuyumlu kaydırıcı yazmacın öbek gösterimi Şekil 4-39 da görülmektedir.

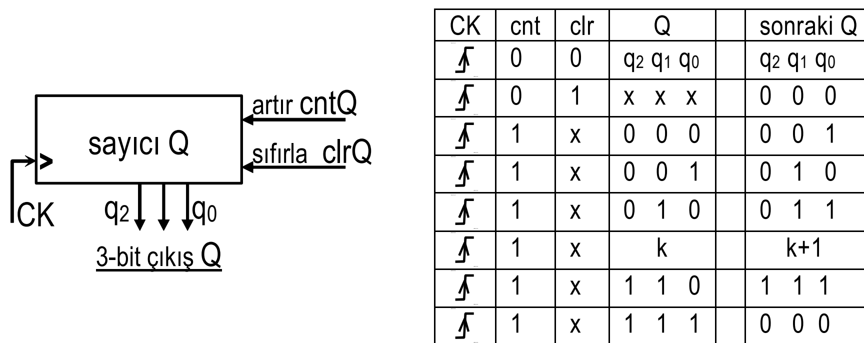


CK	shr	ld	SI	P		sonraki Q
↑	0	0	x	x x x x		$q_3\ q_2\ q_1\ q_0$
↑	0	1	x	$p_3\ p_2\ p_1\ p_0$		$p_3\ p_2\ p_1\ p_0$
↑	1	x	si	x x x x		$q_2\ q_1\ q_0$ si



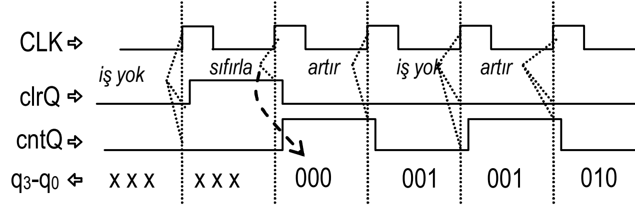
Şekil 4-39 paralel yüklemeli (load) tipik 4-bit an-uyumlu kaydırıcı yazmaç..

Tipik bir 3-bit anuyumlu sayıcı, Şekil 4-40'ta görüldüğü gibi, yukarı sayabilmeye olanak veren *artır* (*count*) girişine sahiptir.



CK	cnt	clr	Q		sonraki Q
↑	0	0	$q_2\ q_1\ q_0$		$q_2\ q_1\ q_0$
↑	0	1	x x x		0 0 0
↑	1	x	0 0 0		0 0 1
↑	1	x	0 0 1		0 1 0
↑	1	x	0 1 0		0 1 1
↑	1	x	k		k+1
↑	1	x	1 1 0		1 1 1
↑	1	x	1 1 1		0 0 0

Şekil 4-40 *sıfırla* (*clear*) girişi olan tipik bir 3-bit senkronize sayıcı



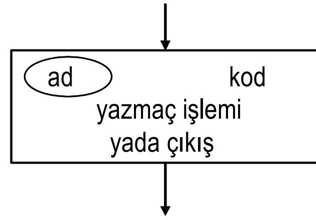
4.7.2 ASM Çizim Elemanları

ASM çizimi, sırasal denetim-birimi durumları ile bir durumdan diğerine geçerken oluşan olaylar arasındaki zamanlama ilişkileri yanında olayların sırasını betimler. Sayısal bir sistemdeki veri-işlem işlemlerini ve denetim sırasını tam ve kesin belirtmek için, sayısal donanımın da şartlarını dikkate alarak özellikle uyarlanmıştır.

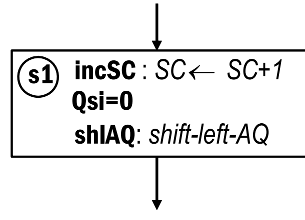
ASM çizimi üç ana elemandan oluşur:

1. Durum Kutusu: sıradaki durumu belirtir. Her durum kutusunun sembolik bir adı vardır ve ikili kodu diğerlerinden farklıdır.

durum kutusu

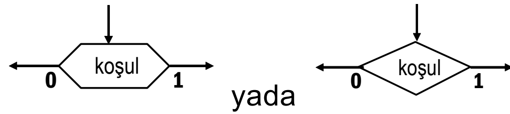


örnek

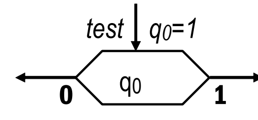


2. Karar Kutusu: bir koşulu sınamak ve akışı ona göre yönlendırmek için kullanılır. Koşul, veri-işlem devresindeki bir elemanın durumu, çıkışı, yada dışarıdan gelen bir sinyal olabilir.

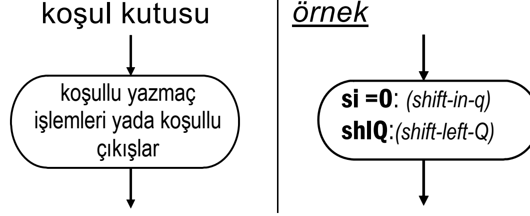
karar kutusu



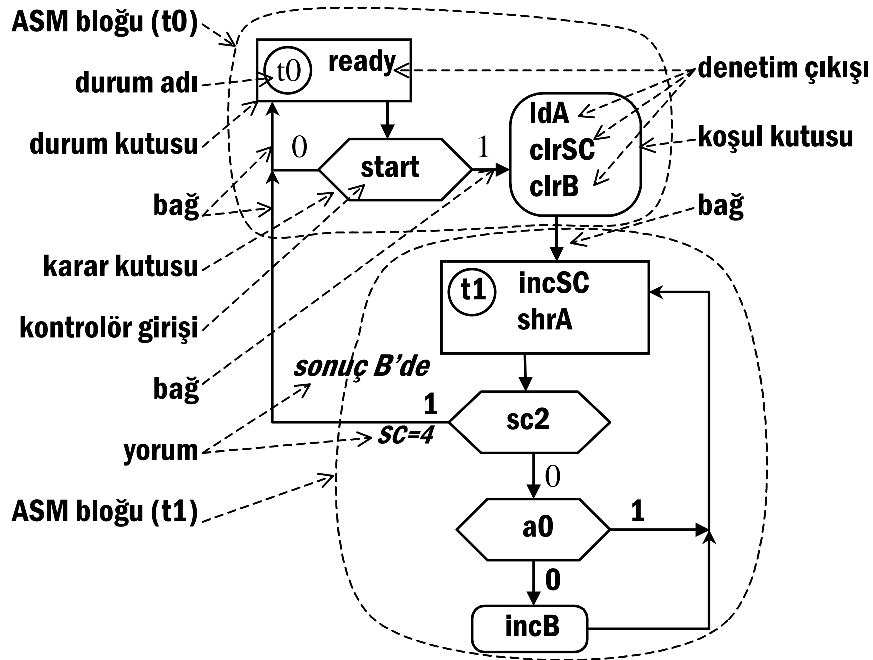
örnek



3. Koşul Kutusu: koşullu yazmaç işlemlerini yada denetim çıkışlarını gösterir. Karar kutularının çıkış yollarında bulunmalıdır.



Bir durum kutusu ve karar kutusu ile onun çıkış yollarına bağlanmış koşul kutuları bir ASM bloğu olarak adlandırılır. Bir ASM bloğundaki tüm işlemler aynı anda bir saat döngüsünde yapılır. İşlemler alışılmış akış şeması işlerinde olduğu gibi tek tek ele alınmazlar, ve aynı zaman döneminde yer alırlar.



Şekil 4-41 Bir ASM çiziminin elemanları

Algoritmalar ve ASM-çizimlerinin karşılaştırması

algoritmik akış şemalarında:

- komutlar genel amaçlıdır. Belirli devre elemanlarına kısıtlı değildir.
- akış şemasındaki komutların sırası komutların işlenme zamanlarıyla aynıdır.
- komut kutusu başına saat döngüsü bilinmemektedir.
- algoritmaların başlama ve durma mekanizmalari belirsizdir.

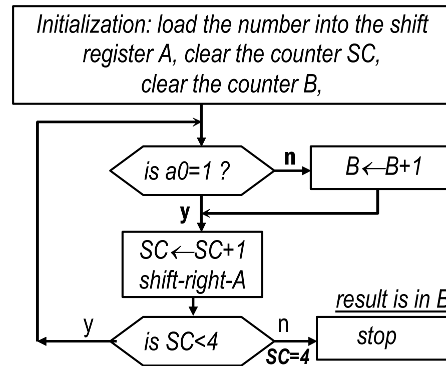
ASM çizimlerinde:

- komutlar, denetim devresinin çıkışlarıdır. Komutlar ya bütün devrenin çıkışı yada devrenin veri işleme kısmındaki elemanlardan birinin denetim girişidir.
- bir ASM bloğu saat dönemi sonunda, o öbekteki çıkışların sırasından bağımsız olarak, tamamen işlenir.
- her ASM bloğu tam olarak bir saat dönemi zaman alır.
- süreç saat aralarında durak-sadığında son durum duran süreci temsil eder.

4.7.3 Algoritmadan Durum Makinesine

Durum makinesinin orijinal algoritmasıyla ASM-çiziminde anlatılan biçimde çalışması için durum makinesindeki veri ve denetim sinyallerinin zamanlamasının uygun olması gerekir. Şimdi algoritmadan ASM-çizimi ve zamanlama-diyagramının tümünü kapsayan bütün bir sırasal durum makinesi uygulaması tasarımı örneği vereceğiz.

Şekil 4-42'deki 4-bitlik bir sayının bitlerinde kaç sıfır olduğunu sayan algoritmik akış şemasını ele alın. Amacımız bu algoritmayı en az sayıda saat dönüşünde, yani en kısa sürede yapabilecek bir sırasal devre kurmaktır.



Şekil 4-42 4 bitlik sayılardaki sıfır-biti miktarını saymak için bir algoritma.

Devremizin son hali iki kısımdan oluşacaktır, bir **veri-işlemci** ve bir **denetim birimi**. Veri işlemcisi bir **A** kaydırma yazmacı (shift-register) , ve **B** ve **SC** olmak üzere iki sayıcı (counter) içerecektir.

Denetim devresinde devrenin durumunu belirleyecek yazbozları bulunmalıdır. Örneğimizde, denetim-biriminde her durum için bir D-

yazbozu olacaktır (yani, **t0** süresince **Q0** çıkışı bir (*yüksek-high*), **t1** süresince **Q1** çıkışı bir, vs.). Denetim biriminde bir sonraki durumu (t0 için D0, t1 için D1, vs. gibi D-yazbozlarının girişlerini) belirlemek üzere bir takım bileşimsel devreler bulunmalıdır. Bu devreler ayrıca o anki durum ile denetim girişlerinin değerine göre denetim çıkışlarını belirler.

Tasarım betimlenen algoritmayı işleyebilecek veriyolu devre elemanlarını belirleyerek başlar. Algoritmada bir kaydırma yazmacı, ve iki sayıcı gerekiyor. Kaydırma yazmacının 4-bitlik paralel girişi, ***U*** diyelim, sayıyı yüklemek üzere dış veri girişlerine bağlı olmalıdır. Sayıcıların senkronize sıfırlama işlevi olmalıdır. Bir sayıcı, **B** diyelim, sıfır-bitlerini saymak için planlanmıştır. Işleme bittiğinde, **B** sonucu veri-işleme devresinin çıkışına yollar. Diğer sayıcı **SC**, sırayla **A** -daki 4-bitin işlenmesini sayacaktır. Böylece sayı dörde ulaştığında makine durdurulabilir.

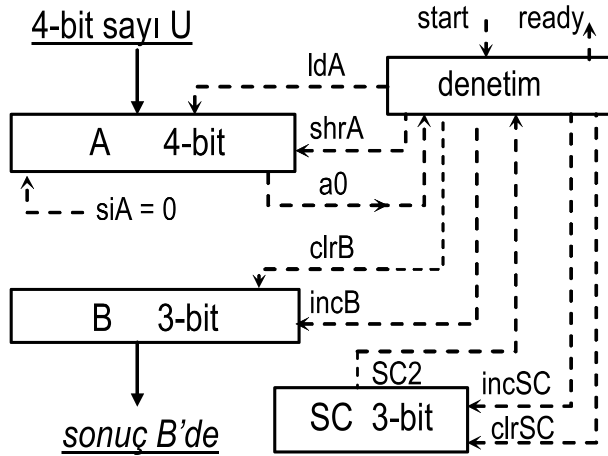
Veriyolundan denetim-birimine iki giriş gerekir;

a0: **A -**nın en az anlamlı biti; ve
sc2 (**SC** sayıcısının 2.nci biti). **SC**=4 olunca **sc2** bir (yüksek) olur.

Denetim-birimi **A**, **B** ve **SC** yazmaçlarına aşağıdaki sinyalleri sağlamalıdır,

ldA: **U** dış girişini **A -**ya yüklemek için,
shrA: **A -**nın içeriğini sağa kaydırmak için,
yani, **a1** → **a0** (al a0 a *gider)*, **a2**→**a1**, .., **siA=0** → **a3**,
clrSC ve **clrB**: **SC** ve **B** sayıcılarını sıfırlamak için.
incSC ve **incB**: **SC** ve **B** sayıcılarındaki sayıları artırmak için.

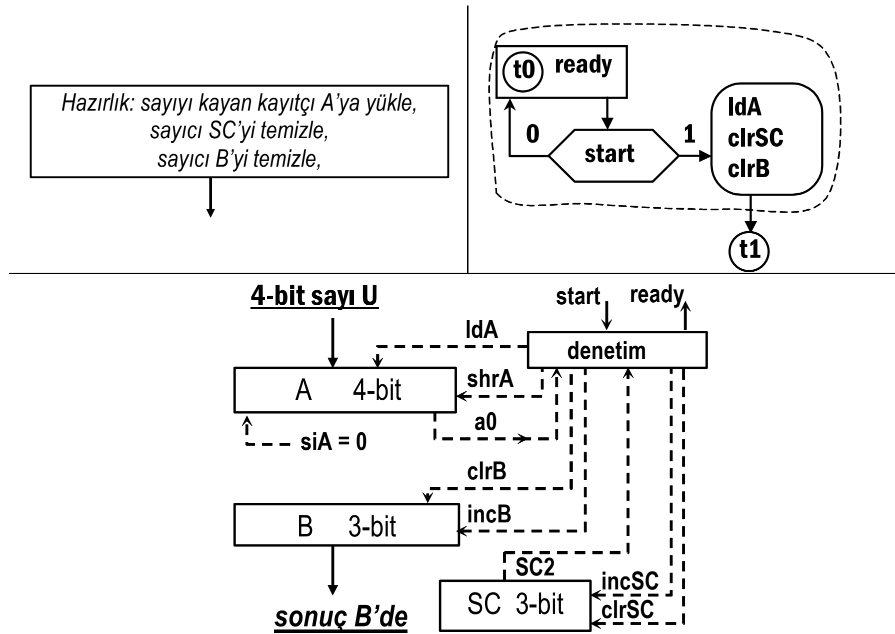
Betimlenen bağlantılar Şekil 4-43'de gösterilmektedir.



Şekil 4-43 Veri işleme donanımı ve denetim birimi sinyalleri

Denetim-biriminin bir dış girişi, **start**; ve bir dış çıkışı, **ready** vardır. Bu sinyaller dış devrelerin zamanlaması için gereklidir. Bir devre işlemi başlatmak için U verisiyle start sinyalini verir. Ready sinyali yüksek olunca sonucu okur.

Algoritmadaki hazırlık kutusu ASM çiziminde sürecin bir dış sinyal ile başlatılmasına dek boşta beklenildiği duruma karşılık gelir. Bu dış denetim girişine "**start**" sinyali dedik. Denetim birimi "*U sayısınıı A kaydırma yazmacına yüklemek", "SC sayıcısını sıfırlamak", ve "B sayıcısını sıfırlamak"* için gerekli sinyalleri çıkarır. Bu noktada, algoritmanın hazırlık kutusundan ilk ASM-bloğumuzu elde ederiz:



Şekil 4-44 Hazırlık kutusuna karşılık gelen ASM bloğu, ve ilgili veriyolu

ready, **ldA**, **clrSC**, **incB** denetim çıkışlarıyla **D1** sonraki durum çıkışını belirleyen Boole ifadeleri **Q0** durum-yazbozu çıkışı ve **start** sinyallerinin fonksiyonudur. '**ready**' ise **Q1** aktifken (yüksek iken) çıkar,

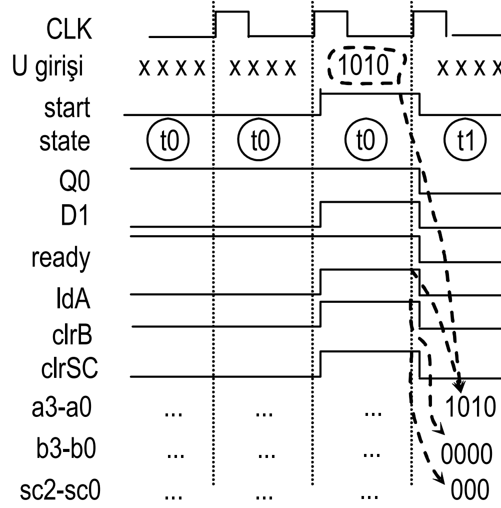
ready = **Q1**.

Diğer sinyaller, **ldA**, **clrSC**, **incB, ve D1** ise hem Q0 aynı anda hem de start yüksekse aktiftir.

ldA = clrSC = clrB
= D1 = Q0 start .

D0, start girişine bağlıdır:
D0 = D0 start'

Saat dönüşünün aktif kenarında, eğer denetim **t0** durumundaysa ve **start** düşük (=0) ise, denetim birimi sonraki saat dönüşünde de **t0** da kalmayı sürdürür. Bu arada dış dev-reye **ready** sinyali yollar, böylece dış devre, devrenin çalışmaya hazır olup olma-dığını sınar. **t0** -ın zamanlaması Şekil 4-45'de gösterilmiştir. Ve Tablo 4-15'te zamanlama diyagramına karşı gelen, sıklıkla kullanılan zamanlama tablosu görülmektedir.



Şekil 4-45 t1 den t2 -ye geçiş için zamanlama-diyagramı

Tablo 4-15 t1 den t2 -ye geçiş için zamanlama-izleme tablosu

Durum	koşul	olay	SC	B	A	giriş
t0		ready	...	...	...	xxxx
t0	start	ready, ldA,clrSC,clrB	000	0000	1010	1010
t1	..	...	...	...	...	xxxx

Koşul ve olay sütunları saatin değiştiği anda aktif olan sinyal adlarını içerir ve veri yolu yazmaçları saat değişiminden hemen sonra satıra yazılı değerleri alır. Örneğin, ikinci satırda, önceki saat kenarından şimdiki saat kenarında kadarki saat dönüşü boyunca denetim-birimi start sinyalini sınar ve ldA, clrSC ve clrB çıkışlarını verir. Ancak, denetim-birimi bu sinyalleri verse de A ya yükleme ve B ile SCyi sıfırlama işi o saat dönüşünden sonundaki saat kenarında gerçekleşir.

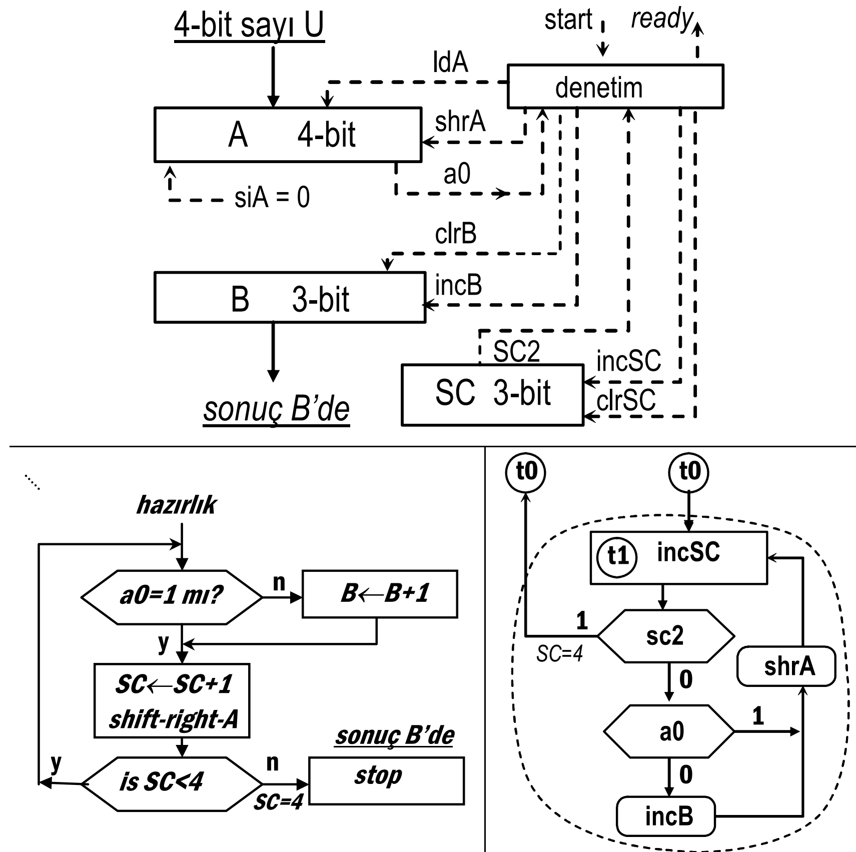
Saat-kenarından ilgili sinyallerin değişimine kadarki zaman farkı genellikle birkaç nanosaniyedir, ancak, sinyallerin ortaya çıkma sıraları net görülebilsin diye bu çizimde gecikme abartılı gösterilmiştir.

Bir ASM-bloğunda işlenebilecek işlemleri sınırlayan iki koşul vardır.

Ilk koşul veri-işlem elemanlarının işlevsel sınırlamalarından doğar, örneğin, denetim-birimi bir yazmacı aynı saat vuruşunda hem yükleyip hem kaydıramaz.

İkinci koşul algoritmadaki işlemler sıralamasının kritik olmasından kaynaklanır. çünkü denetim-birimi bir sonraki duruma geçinceye kadar durum bloğundaki veri-işlemlerinin hiçbiri tamamlanmaz.

Bizim örneğimizde, kalan kısımdaki bütün işlemler farklı yazmaçlar üzerinde olduğundan denetim-birimi algoritmanın kalan kısmındaki tüm işlemleri bir ASM bloğunda yerine getirebilir.



Şekil 4-46 Diğer işlemlere karşılık gelen ASM bloğu ile veri yolu

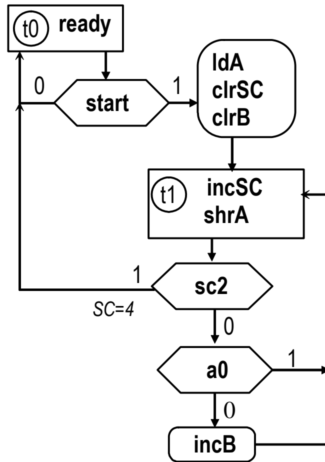
Bu ASM-bloğu ile, denetim birimi çıkışları ve yazboz girişleri için olan Boole ifadeleri aşağıdaki şekilde güncellenirler:

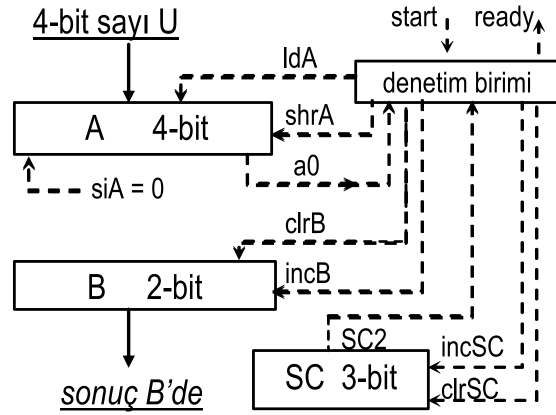
denetim çıkışı		t0-ASM-bloğundan		t1-ASM-bloğundan
D0	=	Q0 start'	+	Q1 sc2
ready	=	Q0		
ldA = clrSC = clrB	=	Q0 start		
D1	=	Q0 start	+	Q1 sc2'
incSC	=			Q1
incB	=			Q1 sc2' a0'
shrA	=			Q1 sc2'

Aşağıda bu durum-makinesinin çalışmasını U=1010 girişi için izleyen zamanlama tablosu gösterilmiştir:

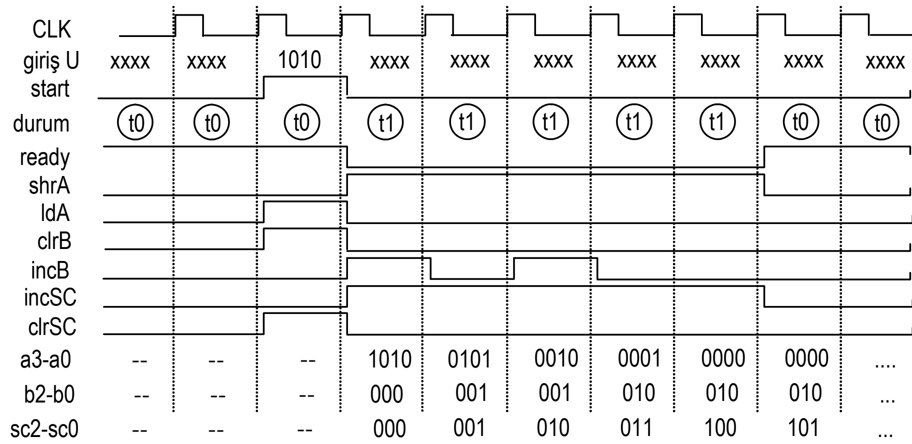
Durum	şart	olay	SC	B	A	giriş
t0		ready	?	?	?	xxxx
t0	start	ready, ldA, clrSC, clrB	000	000	1010	1010
t1		incSC, incB, shrA	001	001	0101	xxxx
t1	a0	incSC, shrA	010		0010	...
t1		incSC, incB,shrA	011	010	0001	...
t1	a0	incSC, shrA	100		0000	...
t1	sc2	incSC	101			...
t0		ready		010		

İzleme tablosundan görüyoruz ki SC dört olduktan sonar A -yı kaydırmak sonucu etkilemez, ama shrA için yazılan Boole ifadesini basitleştirir. Bu iyileştirmeyle birlikte, ASM-çiziminin son hali Şekil 4-47'de görülmektedir.





Şekil 4-47 Sıfır-biti sayıcı için ASM-çizimi ve veriyolunun son biçimi



Şekil 4-48 Sıfır-biti sayıcı devresi için ASM-çizimi ve veri işleme donanımı zamanlama diyagramı

Tablo 4-16 Sıfır sayıcının U=1010_2 girişi için izleme tablosu

Durum	**şart**	**olay**	**SC**	**B**	**A**	**giriş**
t0		ready	...	...	...	xxxx
t0	start	ready, ldA, clrSC, clrB	000	000	1010	1010
t1		incSC, incB, shrA	001	001	0101	xxxx
t1	a0	incSC, shrA	010		0010	...
t1		incSC, incB,shrA	011	010	0001	...
t1	a0	incSC, shrA	100		0000	...
t1	sc2	incSC	101			...
t0		ready		010		

4.7.4 ASM-çiziminden devre şemasına

ASM çizimiyle gösterilen devre seviyesinde tasarım, durum yazbozları, denetim-birimi giriş ve çıkış devreleri ve veri işlem bölümünün yapı öbeklerinin giriş çıkışları arasındaki tüm bağlantıları betimler. Devrenin gerçekleştirilmesi iki aşamalıdır.

Veri İşlemcisi Tasarımı: Verı yolu ya da veri işlemcisini tasarlamak için gereken elemanlar durum ve koşul kutularından doğrudan görülür.

Denetim Devresi Tasarımı: Denetim mantığı karar kutularıyla gerekli durum geçişlerinden belirlenir. ASM çizimindeki 2^n durum için en az n yazboza ihtiyaç duyarız. Önce ASM çizimine dayanarak bir durum tablosu oluştururuz. Denetim durumlarını gösteren yazbozları aşağıdaki gibi seçebiliriz.

JK yazbozlar: Durumların sırasına bakarak, sırayla yazbozların J ve K girişleri için gereken değerleri belirleyerek gerçekleştiririz. Durumlar yazboz çıkışlarından ikilik kodlanmış olarak çıkar.

D yazbozlar: JK yazboz kullanmaya benzer. JK ve D yazboz -ların ikisinde de yazboz çıkışlarına ikilik kodları durum çıkışına (T0, T1, vs.) dönüştürmek için bir dekoder ekleyebiliriz.

Durum başı bir D-yazboz. Bu durumda daha fazla yazboz harcarız ama tasarım doğrudan ASM çizimine bakılarak yapılabilir.

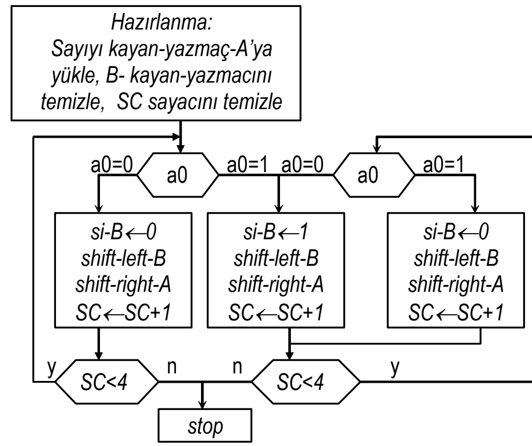
Çoklayıcılar (MUX) ve dekoderler kullanmak probleme daha sistematik bir çözüm getirir. MUX devresi sonraki durumu (next state) belirler, yazmaç seviyesi (yazbozlarla gerçeklenebilir) o anki mevcut durumu tutar ve dekoder ikilik durumu sağlar.

Denetim Devresinin Tasarımı

Betimlenen yazmaçlarla 4-bitlik 2-lik tümleyen alma devresini nasıl gerçekleyeceğimizi bir örnek ile gösterelim. Bir seri eksileme (serial negation) algoritması kullanacağız, ve bu algoritmayı veri işlemcisinin varolan yapı öbeklerini kullanan ASM çizimine çevireceğiz.

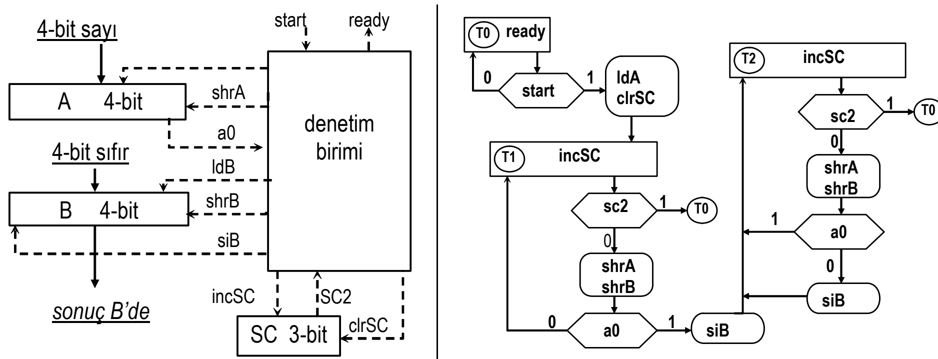
Algoritma: En sağ bitten başlayarak sayının bitlerindeki ilk "bir"i bulun. Bu biri ve sağındaki tüm bitleri olduğu gibi bırakın, solundaki geri kalan bitlerin değilini alın.

SC sayıcısını, ve **A** ve **B** kaydırma yazmacını kullanarak 4-bit sayılar için bu algoritmanın bir akış şemasını oluşturabiliriz.



Şekil 4-49 2 -nin tamamlayanı birimi için algoritma ve akış şeması

Algoritma, aşağıdaki ASM çizimi ve veri işleme donanımı ile betimlenen an-uyumlu durum makinesine dönüştürülebilir.



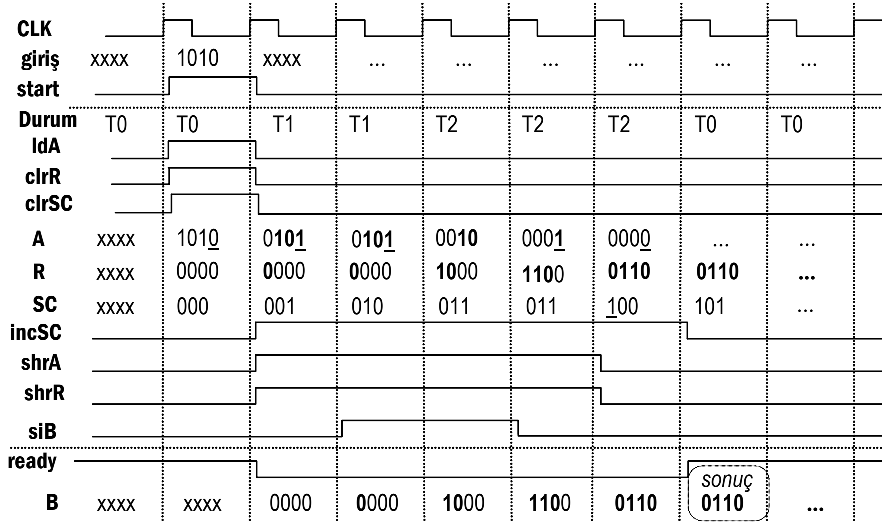
Şekil 4-50 2 -nin tamamlayanı biriminin veri işlemcisi ve ASM çizimi

Denetim devresinin giriş ve çıkışları:

- Durum başı bir yazbozla denetim için **yazboz sayısı =3**.
- Girişler: **reset, start, sc2, a0**.
- Çıkışlar: **ready, IdA, clrSC, incSC, siB, shrB, shrA**

Tablo 4-17 2 -nin tamamlayanı biriminin izlenme tablosu

Durum	koşul	iş	SC	B	A	giriş
T0		ready	xxx	xxxx	xxxx	xxxx
T0	start	IdA,clrSC	000	xxxx	1010	1010
T1		incSC, shrA, shrB	001	0xxx	0101	xxxx
T1	a0	incSC, shrA, shrB, siB	010	10xx	0010	...
T2		incSC, shrA, shrB, siB	011	110x	0001	...
T2	a0	incSC, shrA, shrB	100	0110	0000	...
T2	sc2	incSC	101		...	...
T0		ready	...	0110	...	...

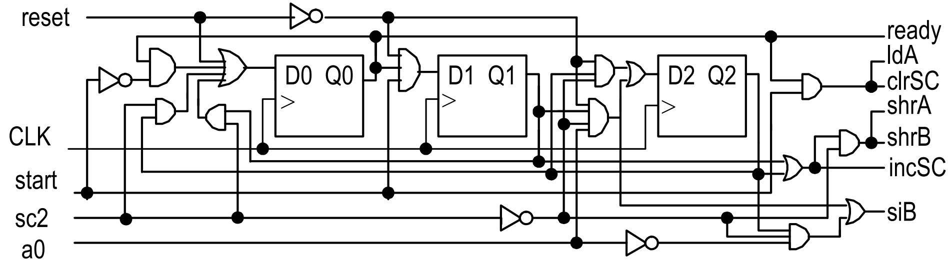


Şekil 4-51 1010_2 -nin 2-lik tümleyeninin alınışının zamanlama diyagramı.

Tablo 4-18 Denetim çıkışları ve FF-girişleri için Boole bağıntıları

sinyal adı	T0 -dan	T1 -dan	T2 -dan	reset eylemi
D0 =	Q0 start '	+ Q1 sc2	+ Q2 sc2	+ reset
D1 = (	Q0 start			) reset '
D2 = (		Q1 sc2 ' a0	+ Q2 sc2'	) reset '
ready =	Q0			
clrSC=ldA =	Q0 start			
incSC =		Q1	+ Q2	
shrA= shrR =		Q1 sc2 '	+ Q2 sc2 '	
siB =		Q1 sc2 ' a0	+ Q2 sc2 ' a0 '	

D-yazbozların durumunu hazır-durumu (t0) dan başlatabilmek için reset sinyali gerekir. Reset sinyali yazbozların hepsinin sıfır vermesini yada aynı saat dönüşünde birden fazla yazbozun bir vermesini önler.



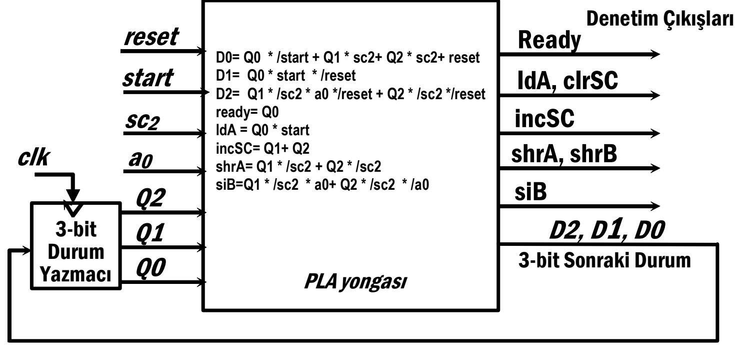
Şekil 4-52 Denetimin durum başı bir yazbozlu devre diyagramı.

Denetim devresinin PLA -le tasarımı

PLA yongalarını programlayıcı aracılığıyla istenen mantıksal giriş-çıkış ilişkisini sağlayacak biçimde programlamak mümkündür. Sonraki durum ve çıkışlar için bulunan ifadeler aşağıdaki devrede kullanılmak üzere PLA programına dönüştürülerek PAL16L8 yongasına programlanır.

```
; Tipik PLA-Program Kaynağı: ilk geçerli satır girişleri belirler.
; ikinci geçerli satır çıkışları belirler. "equation"dan sonraki
; tüm satırlar ve-veya biçimli çıkış fonksiyonlarını betimler.
; inputs  1  2  3  4    5  6  7
        Q0 Q1 Q2  reset sc2 a0 start
; outputs 1  2  3  4  5    6   7
        D0 D1 D2 ldA incSC shrA siB
EQUATIONS
D0= Q0*/start + Q1 * sc2 + Q2 * sc2 + reset
D1= Q0* start */reset
D2= Q1*/sc2 * a0 */reset + Q2 */sc2 */reset
ready= Q0
ldA = Q0 * start
incSC= Q1+ Q2
shrA= Q1 */sc2 + Q2 * /sc2
siB = Q1 */sc2 * a0 + Q2 */sc2 */a0
```

END



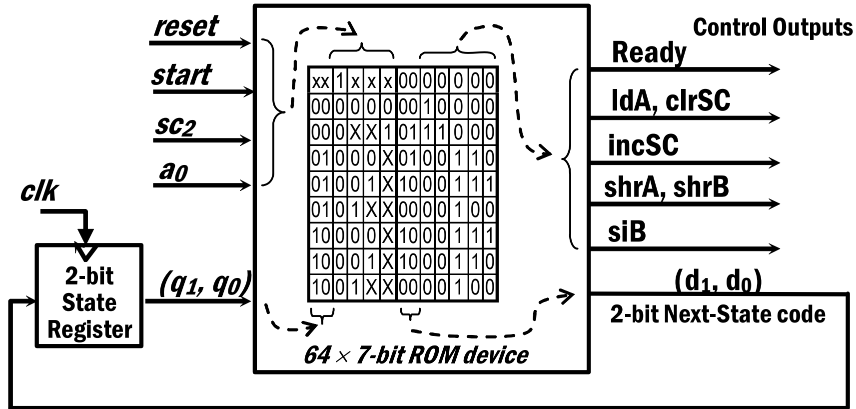
Şekil 4-53 Controller implemented using a PLA device

ROM ve 2-bitlik durum yazmacı ile tasarımı

Tasarım için durumları iki-bitle T0: 00, T1: 01, ve T2: 10 biçiminde kodlarız. Girişler ve durum koduna bağlı olarak çıkışlar ve sonraki durum kodu tablosunu oluştururuz.

ROM adres girişleri					ROM veri çıkışları					
Durum (s1,s0)	reset,	sc2,	a0.	start,	Sonraki Durum	ready	ldA clrSC	incSC	shrA shrB	siB
xx	1	x	x	x	00	0	0	0	0	0
00	0	0	0	0	00	1	0	0	0	0
00	0	X	X	1	01	1	1	0	0	0
01	0	0	0	X	01	0	0	1	1	0
01	0	0	1	X	10	0	0	1	1	1
01	0	1	X	X	00	0	0	1	0	0
10	0	0	0	X	10	0	0	1	1	1
10	0	0	1	X	10	0	0	1	1	0
10	0	1	X	X	00	0	0	1	0	0

Bu tabloyu 6-girişli 7-çıkış verebilecek bir ROM -a yazarsak aşağıdaki devre denetim çıkışlarını verecektir.



Şekil 4-54 Denetimin ROM kullanılarak oluşturulması

5 İşlemci, Veriyolu ve Denetim

5

Amaç

Bu bölümün hedefi tipik bir RISC işlemcisi mimarisinin ve çalışma biçiminin temel yazmaç-seviyesi bileşenlerinden başlayarak, komut başına tek-saat-dönüşlü ve çok-saat dönüşlü yapı ve gerçeklemesi dahil bir tasarım örneği üzerinde anlaşılmasıdır.

5.1 Giriş

Bir bilgisayarın betimlenmiş bir değerlendirme programı için başarımı üç temel etken tarafından belirlenir: komut sayısı, saat hızı, ve CPI, komut-başı-saat-dönüşü .

$$\text{Yürütme-Süresi} = \frac{\text{CPI} \times \text{Komut-sayısı}}{\text{saat-hızı}}$$

Komut sayısı derleyiciye, komut kümesi tasarımına, ve işlemci mimarisine oldukça bağlıdır. Saat dönüş süresi ve CPI ise işlemcinin gerçeklenmesiyle belirlenir.

Ilk bölümde, aşağıdakileri kapsayan MIPS temel komut kümesini işleyebilecek bir işlemcinin tasarlanması ve gerçeklemesini bir örnekle sunacağız

a) Bellek-yazmaç veri aktarımı komutları *lw* ve *sw*

b) Aritmetik-mantık komutları: *add, sub, and, or, slt*

c) Dallanma ve atlama komutları: *beq, j* .

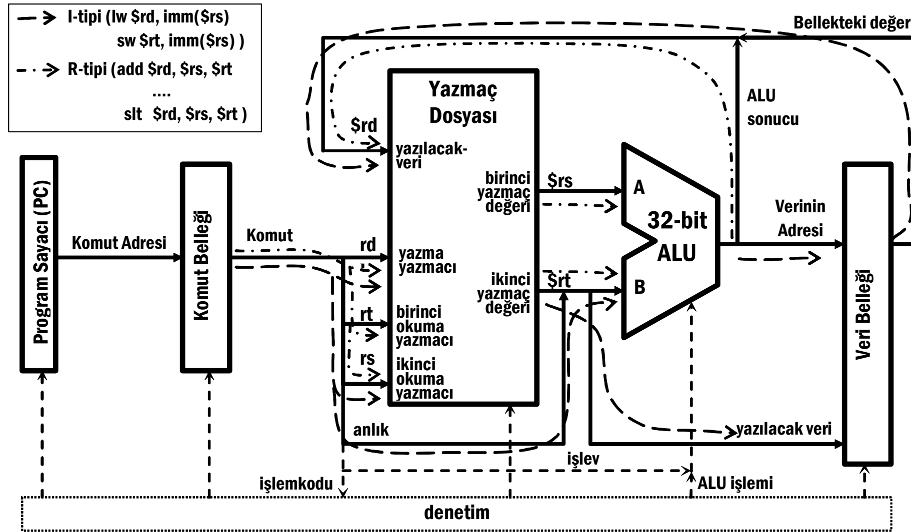
Amacımız, veri yolu geliştirmede ve denetim biriminin anuyumlu sırasal durum makinesi olarak tasarımında kullanılan yöntem ve temel ilkeleri göz önüne sermek olacaktır.

Bellekli-program işlemcilerinin bir dizi komut barındırabilen bir belleği vardır. Işlemcinin amacı bu komutları işlemektir. Sıradaki komut dönüşünde çalıştırılacak bellek adresini belirten atlama yada dallanma komutları hariç tüm komutlar bellekteki-adres sıralamasında çalıştırılır.

Daha basit bir gerçekleme için, program belleği ile veri belleğini birbirinden ayırabiliriz. Bu çeşit ayrık program ve veri bellek birimi olan mimariye *Harvard mimarisi* denir, ve gömülü sistemlerde sıklıkla kullanılırlar.

Çalıştırılacak bir sonraki komutun adresini tutmak için, işlemcide bir sayaç bulunmalıdır. Bu sayaç genellikle program sayacı (*PC*) olarak adlandırılır. Çalıştırılacak komutu *PC* deki adres belirler. Tüm MIPS komutları 32-bitliktir, ve benzer alanlara sahiptirler. Bu alanlar komut formatında (örneğin, *R-tipi: opc*, *rs, rt, rd, sa, fn*; I-formatı: opc, rs, rt, imm; J-formatı: opc, imm-J-adresi) tanımlıdır.

Betimlenmiş iki kaynak-yazmacındaki değerleri çıkışlara aynı anda veren bir yazmaç dosyası yapısını daha önceden görmüştük. Yazmaç-dosyası, *rs* ve *rt* olmak üzere iki kaynak yazmacındaki değerleri çıkışına aynı anda verir, ve ayrıca *write* sinyali geliyorsa 32-bitlik yazılacak-veriyi de *rd* hedef yazmacına yazar. Bu özelliklerin tümü Şekil 5-1 deki aritmetik-mantık komutlarını gerçekleyebilen veri-yolunu oluşturmak için gereklidir.



Şekil 5-1 Aritmetik-mantık ve bellek aktarma komutları için MIPS gerçeklemesinin bir özet görünümü. Atlama ve dallanma komutları ek veriyolu ve toplayıcılar gerektirir.

5.1.1 Terim, Bileşen ve Uzlaşımlar

Aşağıdaki terimler işlemci ve denetim birimi gerçeklemelerinde sıklıkla kullanılır:

çıkış yada sinyal vermek : o çıkışı etkin yapmak yada bir birimi seçili duruma getirmektir. Denetim birimi bir veri yolu birimine istenen işi yaptırmak için gereken sinyali verir (örneğin, R yazmacına girişindeki veriyi yazdırmak için ona *write* sinyalini bir yapar. *write* R -nin denetim-girişi, denetim-devresinin çıkışıdır). Bir sinyalin verilmesinin kesilmesi yada durdurulmasına sinyali *vermeme* denir.

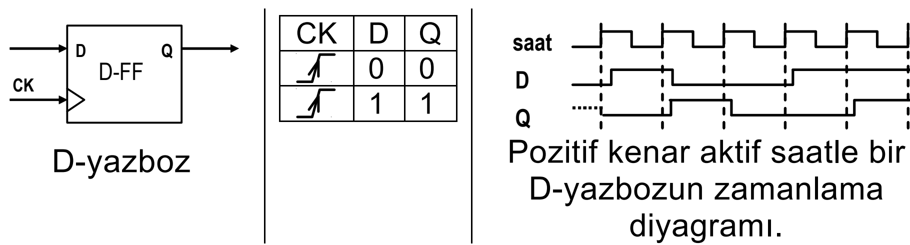
bileşimsel (*combinational*) *öbek*: Belleksiz Boole işlev bloğudur. Çıkış sadece girişin o anki değerine bağlıdır. örneğin ALU bileşimsel bir öbektir.

sırasal öbek: belleği olan öbektir. Çıkışı o anki girişlerle bazı girişlerin daha önceki değerlerine ve bunlarin sırasına bağlıdır.

saat sinyali: Sırasal öbekler arasında veri aktarımını an-uyumlayan sinyaldir. Saat, bir bloğun durum değiştirmesini belirli bir anda başlatır (örneğin, yükselen kenarda ↑ değiştirenin devre sembolü, yada düşen kenarda ↓ ise devre sembolü ile gösterilir).

D-yazboz: D girişi, Q çıkışı ve CK (clock) saat sinyaline sahip, girişteki değeri CK -nın etkin kenarında saklayan (latch) sırasal anuyumlu öbektir. CK -nın bir sonraki etkin kenarına kadar D değeri değişse bile çıkışı (tuttuğu değer) değişmez.

Bir D-yazboz anuyumlu devre elemanlarına en basit örnektir. D-yazbozun çıkışı sadece saatin etkin kenarında değişir.



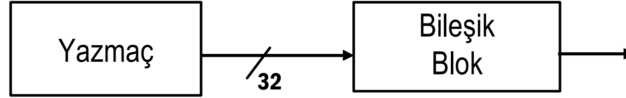
CK	D	Q
↑	0	0
↑	1	1

Şekil 5-2 D-yazbozu ve tipik zamanlama diyagramı

durum elemanı: Giriş değerini saklayan sırasal öbek. Saklanmış değer önceki giriş değeridir, ve elemanın durumu olarak adlandırılır. D- tipik bir durum elemanıdır.

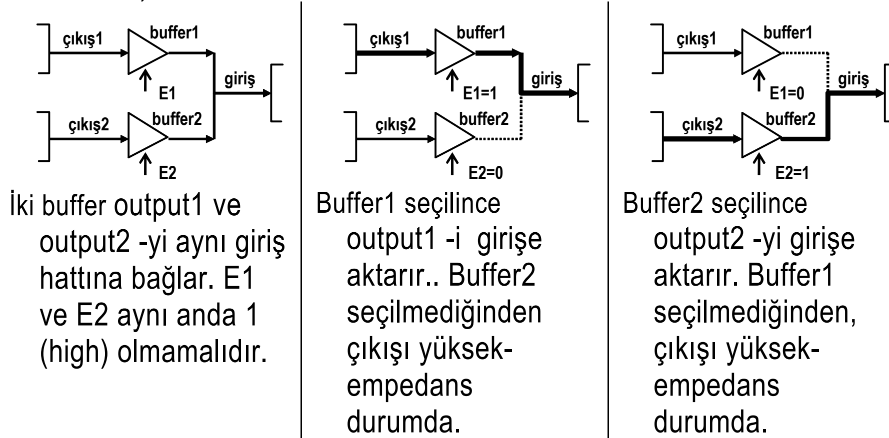
yazmaç: n-bitlik girişi (n>1) aynı anda saklamak üzere n adet D-yazbozundan oluşan anuyumlu sırasal öbektir.

yol: n sinyalin aynı anda aktarılmasını sağlayan veri bağlantısıdır. Basitçe, veri- yada adres-yolu gibi iki öbek arasında n iletkenden oluşabilir, yada denetim-yolu gibi herbiri farklı öbeklere bağlanacak bir sinyal demeti olabilir. Yol genişliği kesme-işareti (/) nin yanına yazılarak belirtilir.



Şekil 5-3 Yazmaç değerini bileşimsel bloğa aktaran 32-bitlik yol

arakat (yada *üç-durumlu-arakat*) : Arakat bir sinyalin gücünü daha fazla giriş hattı sürebilmesi için artırmaya yarayan bir bileşimsel devredir. Bir sinyali değerini değiştirmeden aktarır. Üç durumlu bir arakatın seçme girişi vardır ve bir anahtar gibi çalışır. Seçildiğinde, giriş sinyalini çıkışa gönderir. Seçilmediğinde, çıkışı sanki devre ile bağlantısı kesilmiş gibi çok yüksek empedans olur. Denetim sinyaline bağlı olarak aynı girişe birden fazla çıkış bağlanması gerekirse tri-state-buffer (üç-durumlu-arakat) kullanılır.

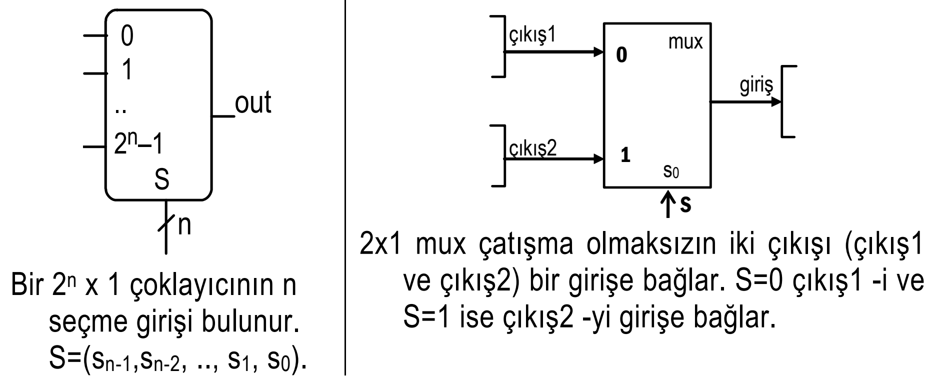


İki buffer output1 ve output2 -yi aynı giriş hattına bağlar. E1 ve E2 aynı anda 1 (high) olmamalıdır.	Buffer1 seçilince output1 -i girişe aktarır.. Buffer2 seçilmediğinden çıkışı yüksek-empedans durumda.	Buffer2 seçilince output2 -yi girişe aktarır. Buffer1 seçilmediğinden, çıkışı yüksek-empedans durumda.

Şekil 5-4 Arakat veri yolunu değiştirebilir. Hem E1 hem de E2 aynı anda secilirse çıkışları birbirine bağlı olduğundan çatışma oluşur.

sayaç: saklanan değeri her etkin saat dönüşünde arttıran yazbozları arasında özel bağlantıları olan yazmaçlardır.

çoklayıcı (multiplexer): birçok girişinden seçilen birini çıkışa veren bileşimsel bir devredir. Üç-durumlu arakatlara benzer olarak, Şekil 5-5 tek gibi birden fazla bloğun çıkış sinyalini seçme sinyaline bağlı olarak tek giriş hattına bağlamada kullanılırlar.



Bir 2^n x 1 çoklayıcının n seçme girişi bulunur. $S=(s_{n-1}, s_{n-2}, .., s_1, s_0)$.

2x1 mux çatışma olmaksızın iki çıkışı (çıkış1 ve çıkış2) bir girişe bağlar. S=0 çıkış1 -i ve S=1 ise çıkış2 -yi girişe bağlar.

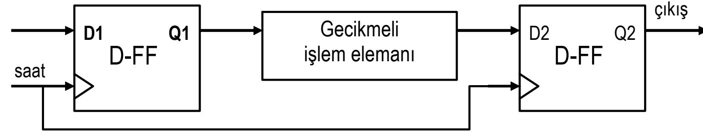
Şekil 5-5 2^n den 1 -e ve 2 den 1 -e çoklayıcılar.

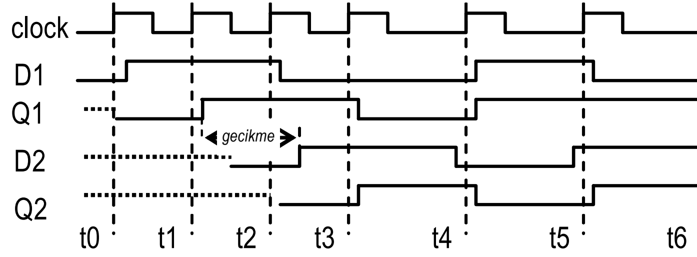
Anuyumlu Sırasal Durum Makinesi

Bir anuyumlu sırasal durum makinesi gerçeklemesi iki kısımdan oluşur: *denetim birimi* durum elemanları ve veriyollarının doğru çalışması için denetim sinyalleri mantığını sağlarken *veriyolu* yada işlemci bölümü anuyumlu ve bileşimsel veri işlem elemanlarını içerir.

Bir anuyumlu veri işlem elemanı saatin sadece etkin kenarında eyleme geçer. Eylem durum değişimiyle sonuçlanabilır. Değişim veri girişlerine ve denetim sinyallerine bağlıdır.

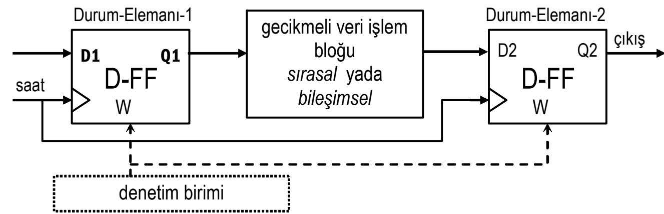
Bileşimsel öbeklerin belleği ya da çıkışlarının değişimini anuyumlayacak saat girişi yoktur. Çıkışı sadece o anki girişlerine bağlıdır. Ancak, girişlerden biri değiştiği zaman, bu girişin etkisi çıkışta genellikle o girişin yayılım zamanı olarak adlandırılan bir gecikmeyle görülür. Bir bileşimsel devrenin (örneğin ALU) yayılım gecikmesi sırasal devrelerde kullanılabilecek en kısa saat dönemi değerine sınırlamalar getirir.

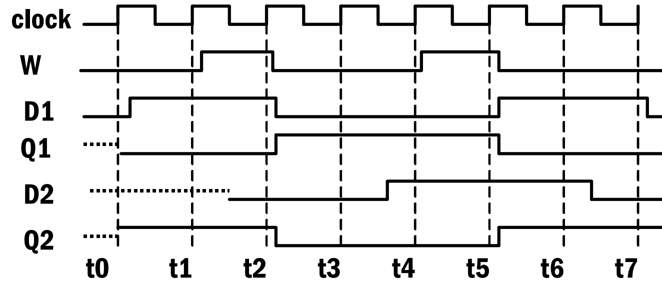




Q1 -den D2 -ye olan gecikme minimum saat periyodunu belirler. Aktif saat kenarları t1, t2 ve t3 işlenen çıkışın durağanlaşma zamanından daha erkendir. t3 -den t4 -e kadar olan periyot doğru çalışma için yeterlidir

Şekil 5-6 Sırasal durum devresinde bileşimsel öbek.



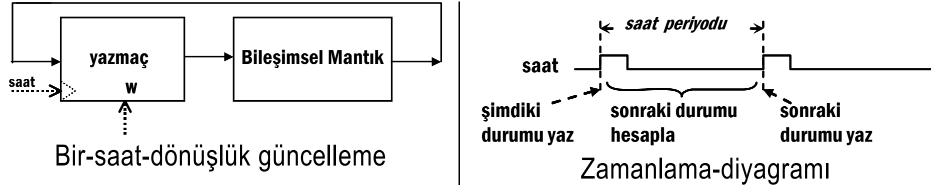


Eğer bir saat dönüşü sinyali işleyen öbek boyunca yaymaya yetmezse *write* sinyali kullanmak gerekir. Durum elemanları saatin etkin kenarında ancak eğer denetim birimi *write* sinyali verirse güncellenir. Zamanlama diyagramında, durum elemanları çıkışlarını t2 ve t5 anında güncellemektedir.

Şekil 5-7 *write* yada *enable* girişli sırasal elemanlar.

Eğer saat süresi bileşimsel öbeklerin gecikmesine göre genişletilemezse, bir write sinyali durum elemanlarını sadece belli sayıda saat periyodunda bir kere seçilmesini sağlar.

Uygulamaya bağlı olarak bir yazmacın *yaz* (*write*) denetim sinyali *yükle* (*load, ld*), yada seç (*enable, E*) olarak ta adlandırılır.

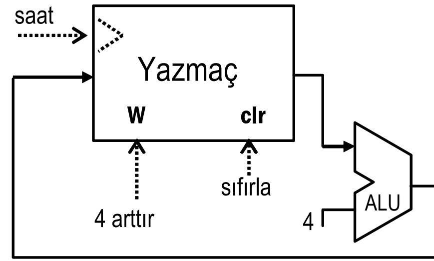


Şekil 5-8 Tek saat dönuşünde güncelleme yapan devre ve zamanlama diyagramı

Ayrıca yazmaçtaki değer bir saat dönüşünde okunup bir bileşimsel mantık ile yazmaçtaki bir sonraki saatte olması gereken değere dönüştürülerek aynı saat dönüşü sonunda yazmaca geri yazılabilir.

Örnek:

Bir sayaç oluşturmak için bir yazmaç ile bir toplayıcı kullanılır. Şekil 5-9 da böyle bir 4-arttır sayacı görülmektedir.



Şekil 5-9 4-artır-sayacı

5.1.2 MIPS Komut Altkümesi Gerçeklemesi

Şimdi komut kümesinden kısıtlı sayıdaki komutların gerçeklenmesini göstermeye geçebiliriz. Önce *tek-saat-dönüşü gerçeklemeyle* başlayacağız. Gerçeklemede iki bileşen gerekir, *veri yolu* ve *denetim birimi*. Şekillerde, *veri yolların*ı (data-path) tam çizgiyle *denetim sinyalleri*ni ise kesikli çizgiyle göstereceğiz.

5.2 Tek saatli data yollarını oluşturma

Hedef işlemcimizin amacı komut belleğinde bir program oluşturabilecek bir dizi komutu (*add, sub, and, or, slt, lw, sw, beq*) çalıştırabilmesidir.

5.2.1 R-formatı komutları

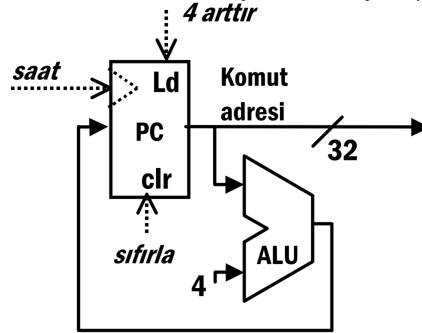
Gerçeklemenin başlangıcında yalnızca *R-tipi* aritmetik-mantık komutlarından *add, sub, and, or, slt* komutlarına odaklanacağız.

Tablo 5-1 R-format MIPS komutlarının biçimi

alanlar / R-tipi komutlar	(6 bits) $b_{31}..b_{26}$	(5 bits) $b_{25}..b_{21}$	(5 bits) $b_{20}..b_{16}$	(5 bits) $b_{15}..b_{11}$	(5 bits) $b_{10}..b_{6}$	(6 bits) $b_{5}..b_{0}$
	opc	rs	rt	rd	sa	fn
add $1,$2,$3	0	2	3	1	0	32
sub $1,$2,$3	0	2	3	1	0	34
and $1,$2,$3	0	2	3	1	0	36
or $1,$2,$3	0	2	3	1	0	37
slt $1,$2,$3	0	2	3	1	0	42

Gerçeklemenin algoritması şöyledir

1- *PC* -yi dört arttır (her komut 32-bit, yani 4 bayttır),



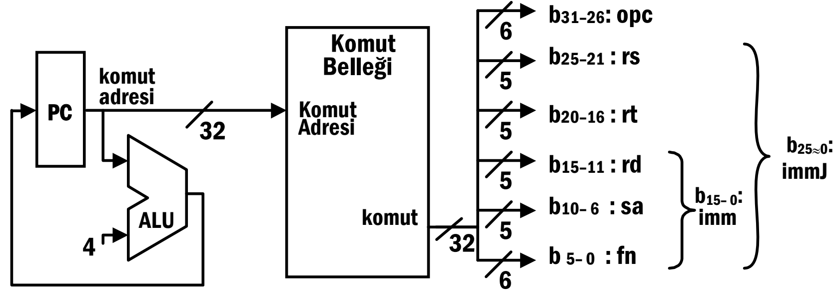
Şekil 5-10 PC -yi 4 arttır

2- Komut belleğinde *PC* yazmacının gösterdigi adresteki komuta ulaş ve aşağıdakileri belirlemek üzere komut alanlarını çözümle.

i- kaynak yazmaçlarının (*rs* ve *rt*) numaraları.

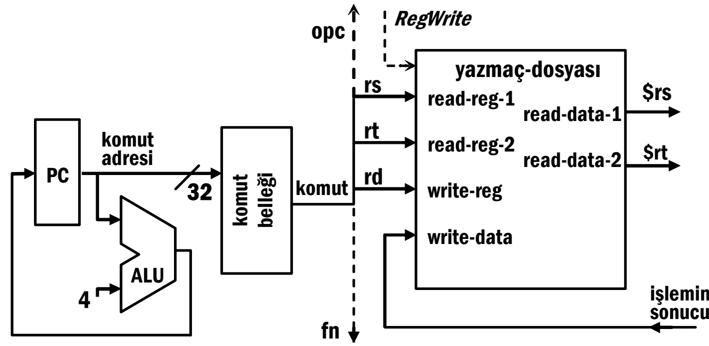
ii- hedef yazmacının (*rd*) numarası.

iii- *opc* ve *fn* alanlarına göre komutta belirtilen işlem ve işlev.



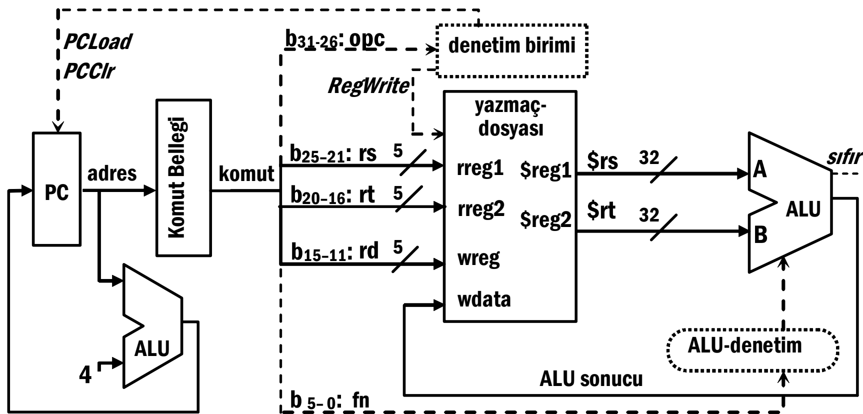
Şekil 5-11 Komut okunması ve komutun bileşenleri

3- yazmaç dosyasından *$rs* ve *$rt* yazmaç içeriklerine erişip ALU -ya aktar.



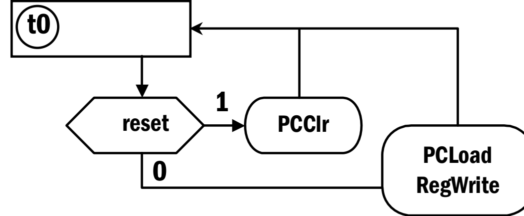
Şekil 5-12 Yazmaç okuma ve işlem sonucunun yazılması.

4- *fn* ile verilen işlevi ALU -da yürüt ve ALU işleminin sonucunu *rd* alanıyla gösterilen hedef yazmaca yaz.



Şekil 5-13 R-tipi komutta ALU işlemi ve sonucun yazılması

Bu veri-yolu düzenlemesi tüm *R-tipi* komutları çalıştırabilir. Bu veri-yoluna ilişkin ASM çiziminin denetlecinin gerçeklenmesinde durum elemanı gerekmez.



Şekil 5-14 R-tipi komutları tek saat dönüşünde çalıştıracak ASM çizimi. Tek ASM öbeği olması denetim devresinin bileşimsel devre olduğunu gösterir.

5.2.2 Anlık Bellek-Yazmaç Aktarımı Komutları

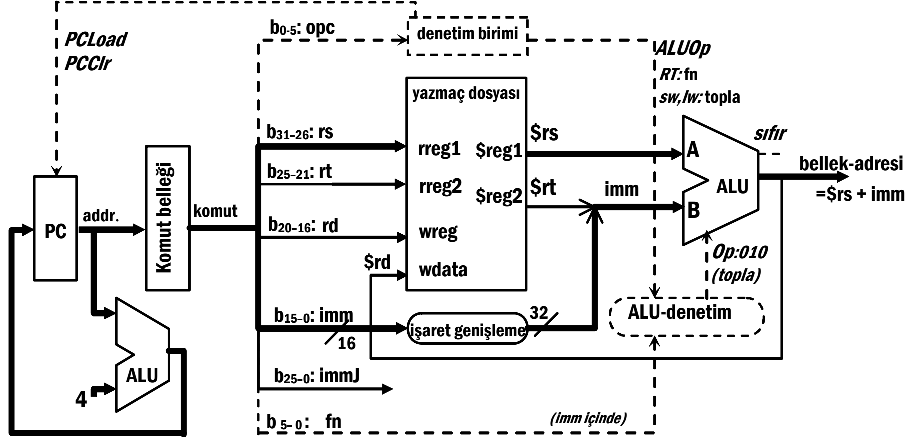
Veri yolunu ayrıca sözcük yükle (*lw*) ve sözcük sakla (*sw*) gibi *I-format* komutlarını gerçeklemek için genişletelim.

alanlar / I-tipi komutlar	(6 bit) $b_{31}..b_{26}$	(5 bit) $b_{25}..b_{21}$	(5 bit) $b_{20}..b_{16}$	(16 bit) $b_{15}..b_0$
	opc	rs	rt	imm : değer veya adres
lw $1, ofset($2)	35	2	1	ofset
sw $1, ofset($2)	43	2	1	ofset

Hatırlarsanız *lw* komutunda bir hedef ve bir kaynak yazmacı belirtirken, *sw* -de belirtilen yazmaçların ikisi de kaynak yazmacıdır. Bu nedenle, komutun işlem koduna bağlı olarak, *rt* -yi *sw* için *rreg2* -ye *lw* içinse *wreg* -e aktarabilmemiz gerekir. Bunu yazmaç dosyasının *wreg* girişine bir çoklayıcı kullanarak sağlayabiliriz.

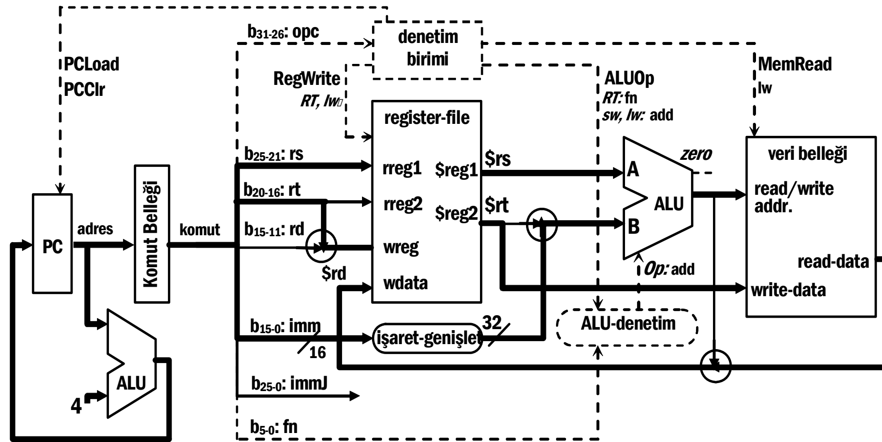
Veri belleğni kullanan komutları tek-saat-dönüşünde gerçeklemek için, veri-belleğini komut belleğinden ayırmalıyız. komut belleğinden ayrı adreslenen Bu tip veri belleğine ayrık veri belleği denir.

1) ALU, hem *lw* hem de *sw* -de, veri-bellek adresini hesaplamalıdır.

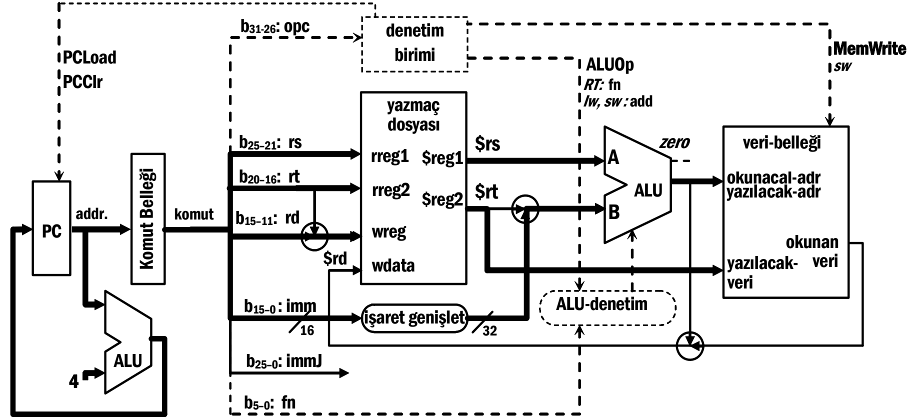


Şekil 5-15 ALU bellek adresini topluyor

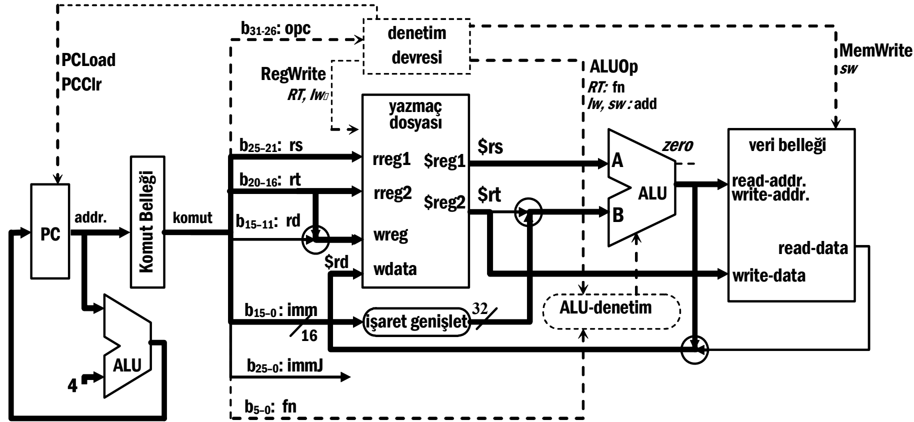
2) *lw* -de bellekteki veriyi yazmaç dosyasının *write-data* girişine göndermek için yeni veri yolu gerekir. *lw* komutunun işlem kodunda denetim birimi bellekten veriyi veri belleğine *MemRead* sinyali vererek okur. *sw* -de belleğin hesaplanan adresine *rt* -deki değeri (*$rt* ile gösterilir) yazmak için de yeni bir veri yolu daha gerekir. Denetim birimi, *sw* komutunun işlem kodunda veri belleğine *MemWrite* sinyalini yollar.



Şekil 5-16 LW komutu işlenirken kullanımda olan veri yolu

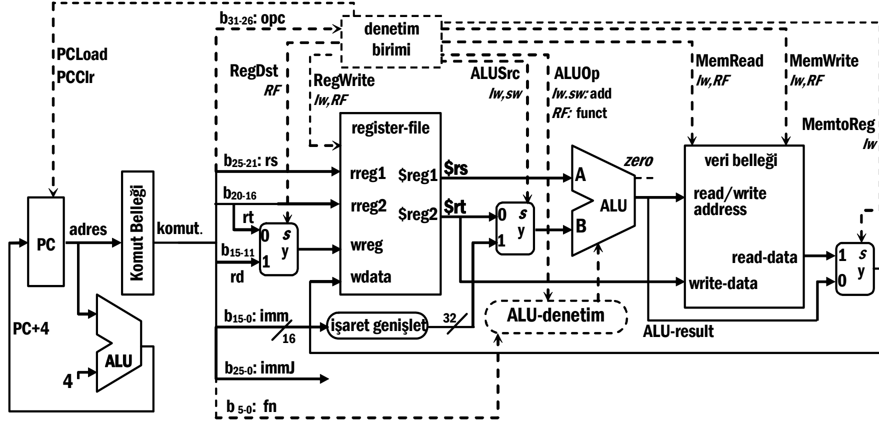


Şekil 5-17 SW komutu işlenirken kullanımda olan veri yolu

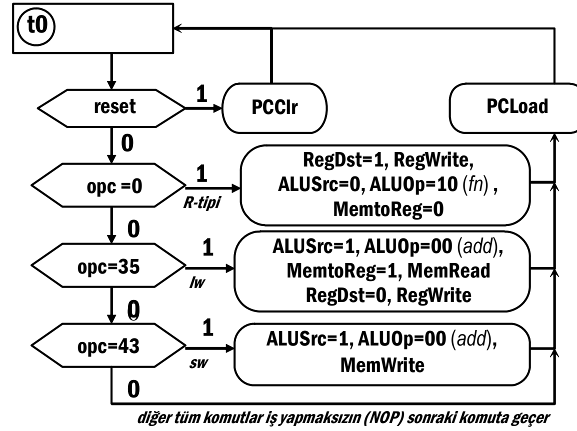


Şekil 5-18 R-tipi (RF) komutlar işlenirken kullanılan veri yolu

Komuta göre *wreg, wdata* ve *ALU-B* girişine bağlanacak hatlardan birini seçebilmek üzere devreye çoklayıcılar eklemeliyiz. Bu çoklayıcıları yazmaç-dosyasının *wreg*, ve *wdata* girişleri ile *ALU-B* girişine koymalıyız.



Şekil 5-19 LW, SW ve R-tipi (RF) komutları gerçeklemek için veri yolu diyagramı



Şekil 5-20 LW, SW ve R-tipi komutları gerçekleyen ASM çizimi.

5.2.3 Dallanma (*branch*) Komutu

Şimdi veri yolu çizimini dallanma ve atlama komutlarını da gerçeklemek üzere genişleteceğiz. eşitse dallan, (*beq, branch if equal*) komutu *I-tipi*dir,

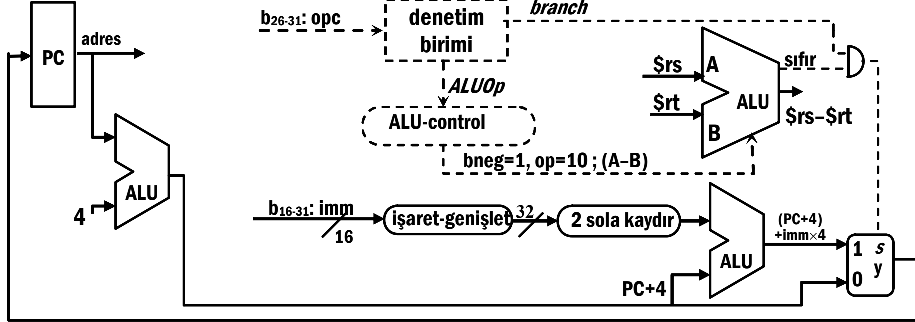
Tablo 5-2 I-tipi MIPS komutlarından BEQ

alanlar / komut	(6 bit) $b_{31}..b_{26}$	(5 bit) $b_{25}..b_{21}$	(5 bit) $b_{20}..b_{16}$	(16 bit) $b_{15}..b_{0}$
	opc	rs	rt	imm : değer veya adres
beq $1,$2,adres	4	2	1	adres/4

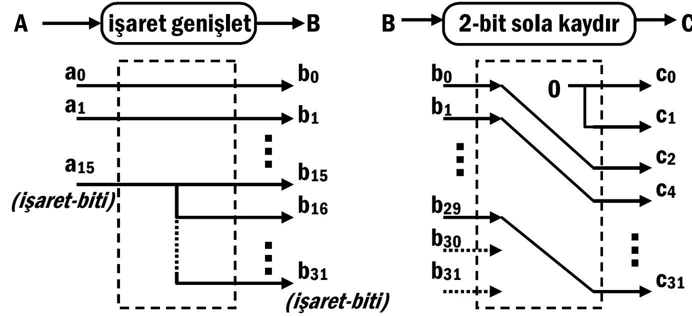
beq eğer *rs* ve *rt* ile belirtilen yazmaçlardaki değerler birbirinin aynıysa *PC* yazmacını değiştirir. *PC* yazmacındaki değer bayt adresiyken, komuttaki anlık değer *PC -ye-göre-sözcük-adresi*dir (=*göreceli-bayt-adresı* $\times$ *4*). Dolayısıyla anlık değer *PC* içeriğine eklenmeden önce *bayt-adresi*ne dönüştürülmelidir.

eğer (\$rs=\$rt) ise PC $\leftarrow$ *(PC + 4) + imm* $\times$ *4 ,*
değilse PC $\leftarrow$ *PC + 4*

Ana ALU *rs* ile *rt* -yi karşılaştırmakta kullanıldığından, *(PC+4) + imm*$\times$*4* -deki toplama işlemi için işlemci yeni bir ALU kullanmalıdır. Hatırlasanız *imm*$\times$*4* aslında *imm* -in 2-bit sola kaymış halidir. Sonucun 32-bitlik olması gerektiğinden 16-bitlik işaretli ikilik bir sayı olan *imm* sola kaydırılmadan işaret genişletme gerektirir.



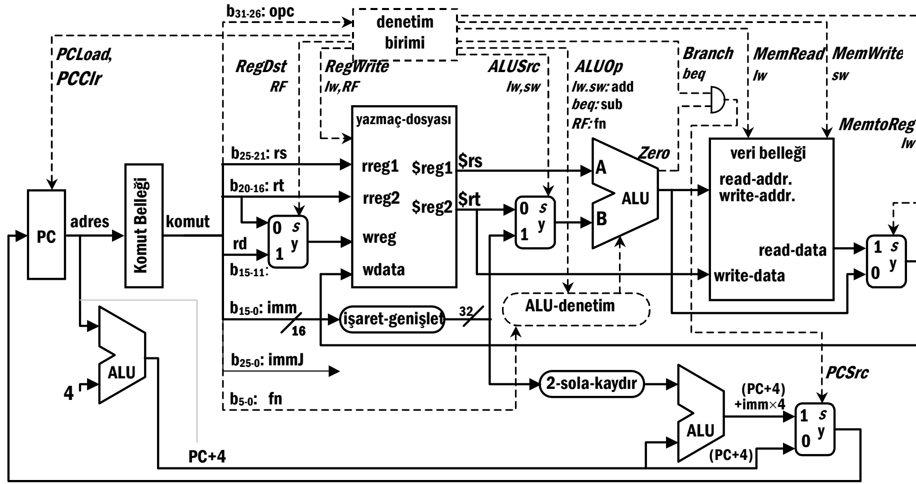
Şekil 5-21 BEQ komutunun yeni veri yolu fazladan bir ALU gerektirir.



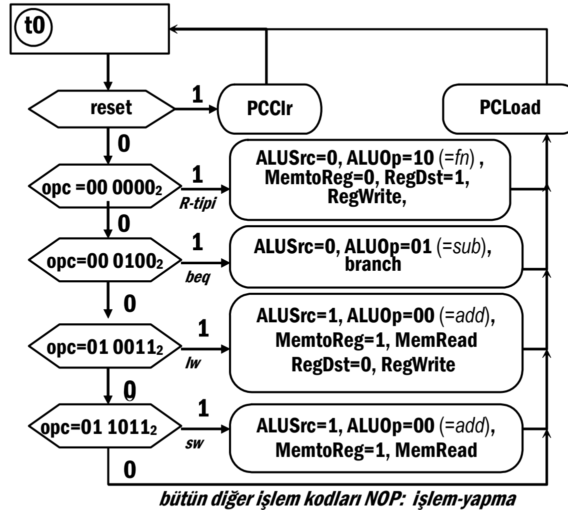
Şekil 5-22 *işaret-genişlet* (sign extend) ve *2-sola-kaydır* (shift-left-2) işlemlerinin devre bağlantısı düzeyinde gerçeklenmesi.

Tablo 5-3 Veriyolu diyagramının işleyebildiği komutlar.

alanlar:	$b_{31}..b_{26}$	$b_{25}..b_{21}$	$b_{20}..b_{16}$	$b_{15}..b_{11}$	$b_{10}..b_{6}$	$b_{5}..b_{0}$
R-tipi	opc	rs	rt	rd	sh	fn
add $1,$2,$3	0	2	3	1	0	32
sub $1,$2,$3	0	2	3	1	0	34
and $1,$2,$3	0	2	3	1	0	36
or $1,$2,$3	0	2	3	1	0	37
slt $1,$2,$3	0	2	3	1	0	42
I-tipi	opc	rs	rt	imm: (değer yada adres)		
beq $1,$2,400	4	2	1	100		
lw $1,100($2)	35	2	1	100		
sw $1,100($2)	43	2	1	100		



Şekil 5-23 R-tipi, lw, sw, beq komutlarını işleyen veri yolu çiziminin tümü



Şekil 5-24 *RT, lw, sw, beq* komutlarını işleyen veri yolu çiziminin ASM-çizimi.

ALU Denetim

ALU-denetim birimi ana denetim biriminin işlevlerini basitleştirmek için tasarlanan bileşimsel öbektir. Ana denetim birimi *opc*-alanını kullanarak ALU-denetim biriminin *ALUOp* girişine karar verir.

- *LW* ve *SW* komutlarında ALU, geçerli bellek adresini toplama yaparak (*$rs* + işaret-genişlemiş-*imm*) hesaplamalıdır.
- Aritmetik-mantık komutları (*RT*) için *ALU*, komuttaki 6-bitlik *fn* alanına göre çalışmalıdır.
- *BEQ* komutu için, ALU çıkarma (*$rs* – *$rt*) yapmalıdır.

Tablo 5-4 32-bit ALU işlem kodları

Op: ALU İşlem kodu (*Op2, Op1, Op2*)	**ALU-işlevi**
0 0 0	and
0 0 1	or
0 1 0	add
1 1 0	subtract
1 1 1	set-on-less-than

ASM-çiziminde, *ALUOp* çıkışı 3 farklı değer (*add*, *sub*, ve *fn*) alır. Bu üç değer iki bitle kodlanır (*add*:00, *sub*:01, ve *fn*:10). *ALUOp* -un bu kodlanışı için ALU-denetim doğruluk tablosu oluşturup Boole ifadesini bulabiliriz. Aşağıdaki bölümde *ALUOp* -u *A* ile ve *fn*-alanını *F* ile kısaltarak göstereceğiz.

komut	**Tip**	***A1,A0***	***F5 .. F0***	***Op2...Op0***	**ALU-işlevi**
lw	I	0 0	xx xxxx	0 1 0	add: (A+B)
sw	I	0 0	xx xxxx	0 1 0	add: (A+B)
beq	I	0 1	xx xxxx	1 1 0	sub: (A–B)
add	R	1 *0*	*10* 0000	0 1 0	add: (A+B)
sub	R	1 *0*	*10* 0010	1 1 0	sub: (A–B)
and	R	1 *0*	*10* 0100	0 0 0	and: (A and B)
or	R	1 *0*	*10* 0101	0 0 1	or: (A or B)
slt	R	1 *0*	*10* 1010	1 1 1	slt: (A–B) işareti

Doğruluk Tablosunda, *fn* -ın en sol 2 biti hep aynı değerdir, ve Boole ifadesinde yeralmaz. Ayrıca *A1*=0 (yani, *ALUOp*=00 yada *ALUOp*=01) iken ALU-denetim sinyali *fn*-alanından bağımsızdır. Doğruluk Tablosunu fn alanının yalnızca 4-bitini kullanıp bağımsız iki bölüme ayırarak yeniden oluşturabiliriz. :

Tablo 5-5 ALU-denetim çıkışı doğruluk Tabloları

A1A0	*F3 .. F0*	*Op*
0 0	x x x x	0 1 0
0 0	x x x x	0 1 0
0 1	x x x x	1 1 0
1 0	0 0 0 0	0 1 0
1 0	0 0 1 0	1 1 0
1 0	0 1 0 0	0 0 0
1 0	0 1 0 1	0 0 1
1 0	1 0 1 0	1 1 1

AOp1=0

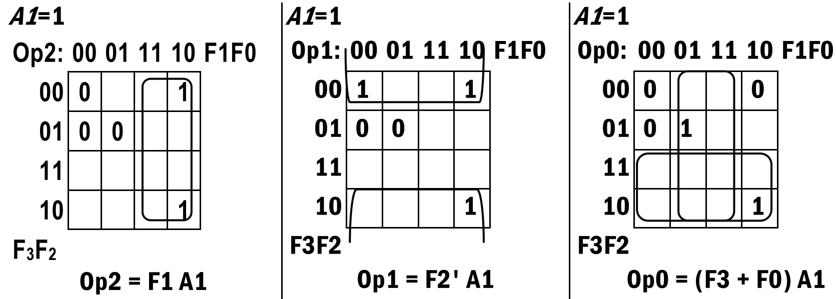
AOp0	*F3 .. F0*	*Op*
0	x x x x	0 1 0
0	x x x x	0 1 0
1	x x x x	1 1 0

AOp1=1

AO0	*F3..F0*	*Op*
x	0 0 0 0	0 1 0
x	0 0 1 0	1 1 0
x	0 1 0 0	0 0 0
x	0 1 0 1	0 0 1
x	1 0 1 0	1 1 1

A1=1 iken, ikinci Tabloda *A0* herzaman 0 -dır ve ifadede yer almayacaktır (Tabloda farketmez (*dont care)* anlamına x ile işaretlenmiştir). Bu durumda Op sadece *F3 .. F0* -a bağlıdır.

Tablo 5-6 ALU-Denetim Biriminin Op çıkışı için Karnaugh çizimleri



A1=0 için ilk Tablodan:

A1=0

A0	funct	Op
0	x x x x	0 1 0
0	x x x x	0 1 0
1	x x x x	1 1 0

Op2 = A1' ; Op1 = A1' ;
yada
Op1' =A1 .

İki Tablodan elde ettiğimiz ifadeleri "VEYA" işlemiyle birleştiririz.

	1. Tablo	VEYA	2. Tablo
Op2 =	*A0*	+	*F1 A1*
Op1 =	*A1'*	+	*F2' A1*
Op0 =			*(F3 + F0) A1*

Op1 ifadesini daha da basitleştirmek mümkündür.

$$Op1 = ((A1' + F2' A1)')' = (A1 \; (F2+A1'))' = (A1\,F2 + \underbrace{A1\,A1'}_{0})' = (A1\,F2)'$$

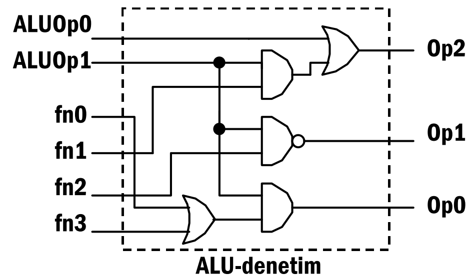
Elde edilen Boole ifadeleri aşağıda yeniden yazalım.

Op0= (fn3 + fn0) ALUOp1;

Op1=(ALUOp1 fn2) ';

Op2= ALUOp0 + fn1 ALUOp1 .

ALU-denetim devresinin şeması Şekil 5-25'de görülmektedir.



Şekil 5-25 ALU-denetim devre şeması

5.2.4 Ana Denetim Biriminin Gerçeklenmesi

Ana denetim birimi her saat-dönüşünde sırasal ve bileşimsel öbeklerin işleyişini denetlemek üzere çıkış sinyalleri verir. Şimdiye kadar devredeki tüm denetim giriş ve çıkışlarını tanıttık. Bu sinyaller aşağıdaki Tabloda özetleniyor.

Tablo 5-7 Ana Denetim Birimi Çıkış Sinyalleri

Sinyal Adı	**Kullanım yeri yada açıklama**	
	Değer	**Yapılan iş**
RegDst	**Yazmaç dosyasının write-register girişi için çoklayıcı seçme girişi**	
	0	**wreg ← rt** *rt*-alanını *wreg* yazılacak-yazmac-girişine bağlar
	1	**wreg ← rd** *rd*-alanını *wreg* yazılacak-yazmaç-girişine bağlar
RegWrite	**Yazmaç dosyasının yaz sinyali.**	
	0	**iş yok**
	1	**$wreg ← wdata** *wdata* -yı *wreg* -in adreslediği yazmaca yazar
ALUSrc	**ALU -nun B girişi çoklayıcısı seçme girişi**	
	0	**ALU B-input ← $rt** *rt* yazmacındaki değeri *ALU-B* girişine bağlar
	1	**ALU B-input ← işaret-genişlet(*imm*)** *IR* -in 16-bit *imm*-alanını genişletip *ALU-B girişin*e aktarır.
MemRead	**Veri belleği oku sinyali**	
	0	**iş yok**
	1	**read-data←Mem[read-addr]** *veri belleğ*inin *okunacak-adres*indeki değeri *okunan-veri* çıkışına yollar
MemWrite	**Veri belleği yaz sinyali**	
	0	**iş yok**
	1	**Mem[write-addr]←write-data** *veri belleğ*inde *yazılacak-veri*yi *yazılacak-adres*e yazar
MemtoReg	**Yazmaç dosyası yazılacak veri (wdata) için çoklayıcı seçme girişi.**	
	0	**wdata ←ALU-result** *ALU-sonucu*nu *yazmaç dosyas*ının *yazılacak-veri* girişine aktarır.
	1	**wdata ← read-data** *veri belleğ*inin *okunan-veri* çıkışı *yazmaç-dosyas*ının *yazılacak-veri* (*wdata*) girişine aktarır.
PCLoad	**Program Sayacı yükleme girişi**	
	0	**iş yok**
	1	**PC ← (PCSrc çoklayıcı çıkışı)** *PC* çoklayıcı çıkışını *PC*ye yükler.
PCClr	**Program Sayacı sıfırlama girişi**	
	0	**iş yok**
	1	**PC ← 0** *PC* yazmacını sıfırlar.

Hatırlarsanız denetim-birimini basitleştirmek üzere veriyoluna iki ekleme yaptık. Bu ek devrelere gereken giriş-çıkışlar Tablo 5-8 ve Tablo 5-9 dadır.

Tablo 5-8 ALU-denetimi birimi için gereken denetim birimi çıkışları

	ALU-denetim birimi işlem seçme girişi	
ALUOp1, ALUOp0.	**00**	ALU -da toplama yaptırır.
	01	ALU -da çıkarma yaptırır.
	10	ALU -da *fn* alanında belirtilen işlemi yaptırır.
	11	kullanılmaz

Tablo 5-9 Dallanma komutunda ALU -nun sıfır (*zero*) çıkışını kullanarak PCSrc seçme sinyalini üretmek için gereken denetim birimi çıkışları

	ALU-sıfır çıkışına bağlı olarak PCSrc seçme girişini değiştirir. *PCSrc= ALU-Zero* VE *Branch*	
Branch	**0**	*PCSrc* sıfır olur, adrese sapılmaz.
	1	Eğer *ALU-Zero* =1 ise *PCSrc*=1 olur ve hedef adrese sapılır.

Ana denetim biriminin giriş sinyalleri ve doğruluk Tablosu Tablo 5-10 ile Tablo 5-11 da verilmektedir.

Tablo 5-10 Ana Denetim Birimi Giriş Sinyalleri

Sinyal Adı	**Gittiği yer ve kullanılış nedeni**	
	Değeri	**Etkisi**
reset	**Ana denetim girişi. Komut Sayacını sıfırlatmak içindir.**	
	0	*İşlem yok.*
	1	*PCClr* verilerek *PC* nin sıfırlanmasını sağlar.
opc5 ... opc0 (opcode)	**Ana denetim girişi. Komuta göre denetim yapmak içindir.**	
	0_{10}	*R-tipi: RegDst, RegWrıte, ALUOp1, PCLoad verilecek.*
	4_{10}	*BEQ: Branch, ALUOp0, and PCLoad verilecek*
	35_{10}	*LW: RegWrıte, ALUSrc, MemRead, MemtoReg, ve PCLoad* verilecek
	43_{10}	*SW: ALUSrc, MemWrite, and PCLoad* verilecek

Tablo 5-11 Basitleştirilmiş denetim-birimi çıkışları için doğruluk Tablosu

	GİRİŞLER							**ÇIKIŞLAR**										
	(Komut işlem kodu) opcode																	
İşlem Komutlar	***opc5***	***opc 4***	***opc 3***	***opc2***	***opc1***	***opc 0***	***Reset***	***RegDst***	***RegWrite***	***ALUSrc***	***MemRead***	***MemWrite***	***MemtoReg***	***branch (PCSrc için)***	***ALUOp1***	***ALUOp0***	***PCLoad***	***PCClr***
Reset	x	x	x	x	x	x	1	0	0	0	0	0	0	0	0	0	0	1
RT	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	1	0
lw	1	0	0	0	1	1	0	0	1	1	1	0	1	0	0	0	1	0
sw	1	0	1	0	1	1	0	x	0	1	0	1	x	0	0	0	1	0
beq	0	0	0	1	0	0	0	x	0	0	0	0	x	1	0	1	1	0

Bu doğruluk tablosunda sıfırlama (reset) girişi işlem-kodu (opcode) girişinden bağımsız olarak yalnızca PCClr çıkışı vermeyi sağlar. Buna bağlı olarak, Tablo 5-11 sıfırlama ve yürütme olmak üzere iki bölüme ayrıştırılabilir. Yürütme için düzenlenen doğruluk tablosunun girişi *opc*, çıkışları ise *RegDst, RegWrite, ALUSrc, MemRead, MemWrite, MemtoReg, Branch* (*PCSrc* için), *ALUOp1, ALUOp0, PCLoad,* ve *PCClr* -dir. Bunlardan *PCLoad,* ve *PCClr* her komut için aynı değeri verdiğinden önemsizdir.

Tablo 5-11 daki doğruluk Tablosunu sağlayan denetim birimi *ALU-Zero* dan bağımsız olarak *BEQ* komutunun her bir yürütülüşünde *Branch* çıkışı verir. Bu durumda, veriyolundaki bir devre *ALU-Zero* yu sınar ve yalnızca hem *Branch* hem de *ALU-Zero* yuksekse *PCSrc=1* üretir. Veriyolunda ALU işlevi ve dallanma için kullandığımız yardımcı devreler tasarım adımlarını izleyebilmemizi kolaylaştırır. CAD araçları ile bu devreleri kullanmaksızın iki-seviye mantık devreleri tasarlayarak daha hızlı çalışan ana-denetim-birimi gerçekleştirmek mümkündür. Örneğin, veriyolundaki dallanma devresinden kurtulmak için denetim biriminde Tablo 5-12 da tanımlanan ALU-Zero girişiyle PCSrc çıkışı bulunmalıdır.

Tablo 5-12 Veri yolundaki dallanma devresinden kurtulmak için kullanılacak ek denetim birimi giriş çıkışları.

<table>
<tr><td>ALU-Zero
(denetim girişi)</td><td colspan="2">ALU -nun zero çıkışı Dallanmada PCSrc üretmek için gerekir. (Daha önce PCSrc = branch . zero çıkışını veriyolunda oluşturmaktaydık).</td></tr>
<tr><td rowspan="3">PCSrc
(denetim çıkışı)</td><td colspan="2">Program Sayacı girişi için multiplexer</td></tr>
<tr><td>0</td><td>PC ← PC + 4
PC+4, PC girişine gider.</td></tr>
<tr><td>1</td><td>PC ← iPC + 4 + (imm ×4)
dallanma hedef-adresi, PC -ye gider.</td></tr>
</table>

Bu durumda, denetim doğrudan *ALU-Zero* sinyaline bağlı olarak *PCSrc* sinyalini verir. Bu duruma karşılık gelen doğruluk Tablosu Tablo 5-13 te görülmektedir.

Tablo 5-13 Dallanma devresiz veriyolu denetlecinin doğruluk Tablosu

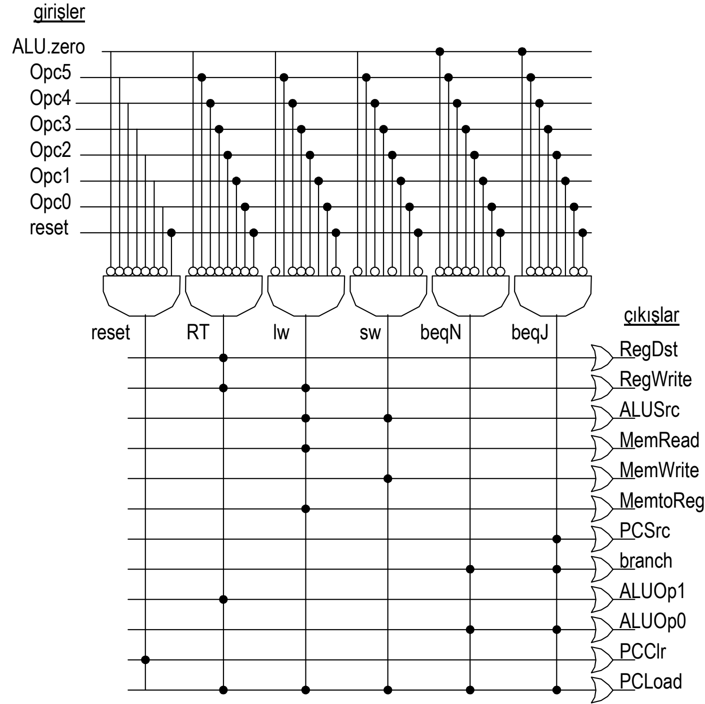
Komut	GİRİŞLER								ÇIKIŞLAR										
	ALU.Zero	opcode bit5	opcode bit4	opcode bit3	opcode bit2	opcode bit1	opcode bit0	Reset	RegDst	RegWrite	ALUSrc	MemRead	MemWrite	MemtoReg	Branch/*PCSrc*	ALUOp bit1	ALUOp bit0	PCClr	PCLoad
Reset	x	x	x	x	x	x	x	1	0	0	0	0	0	0	0	0	0	1	0
RT	x	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0	1
lw	x	1	0	0	0	1	1	0	0	1	1	1	0	1	0	0	0	0	1
sw	x	1	0	1	0	1	1	0	x	0	1	0	1	x	0	0	0	0	1
beq (next)	0	0	0	0	0	1	0	0	x	0	0	0	0	x	*0*	0	1	0	1
beq (jump)	1	0	0	0	0	1	0	0	x	0	0	0	0	x	*1*	0	1	0	1

Bu Boole fonksiyonlarını gerçekleme yöntemlerinden biri de programlanabilir mantık dizisi yongaları (PLA) kullanmaktır. PLA, içindeki tüm kapıların giriş bağlantıları devre bloklarının satır ve sütunlarına gereken voltaj uygulanarak attırılabilen sigortalar sayesinde programlanabilen iki-tabakalı AND-OR yapısıdır. Bir PLA programcısı attırılacak sigortaları ya AND-OR-INVERTER devresini tanımlayan Boole ifadesi programdan ya da bağlantıların bir grafik kullanıcı arayüzü çiziminden bulur ve yongaya Şekil 5-26 daki devreyi yakar.

```
; Tipik PLA-Program Kaynağı:
;  ilk geçerli satır girişleri,
;  ikinci geçerli satır çıkışları belirler.
; equation dan sonraki
;  tüm satırlar ve-veya biçimli
;  çıkış fonksiyonlarını tanımlar.
;inputs   1       2       3        4        5       .....
          ALUZero opc5    opc4    opc3      opc2    .....

;outputs  1       2       3         .....
          RegDst  ALUSrc  RegWrite  .....

equations
RegDst   = /opc5 */opc4 */opc3 */opc2 */opc1 */opc0 */reset
RegWrite = /opc5 */opc4 */opc3 */opc2 */opc1 */opc0 */reset
         +  opc5 */opc4 */opc3 */opc2 * opc1 * opc0 */reset
ALUSrc   =  opc5 */opc4 */opc3 */opc2 * opc1 * opc0 */reset
         +  opc5 */opc4 * opc3 */opc2 * opc1 * opc0 */reset
....
```



Şekil 5-26 PLA ile ana-denetim-birimi gerçeklemesi

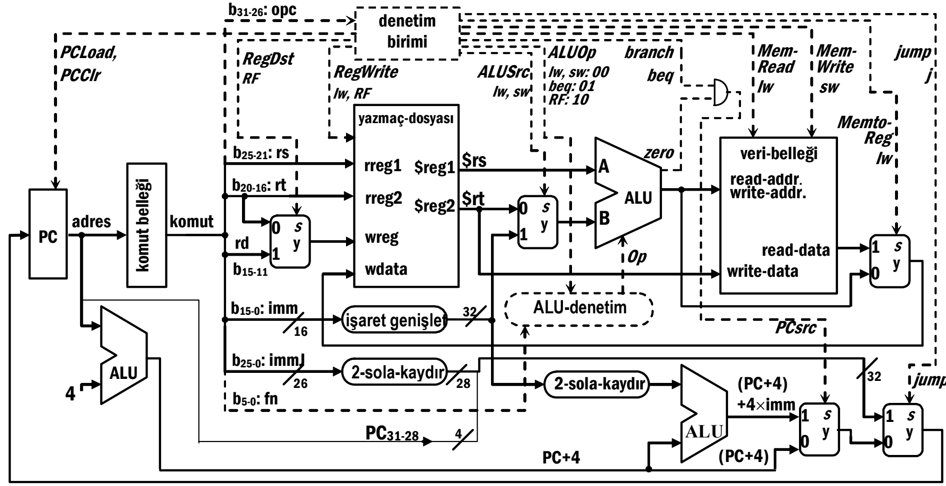
5.2.5 Yeni bir komut-tipi için veri yolu değişikliği

Atla (Jump, *j*) komutu 26-bitlik sözcük-adresi alanına sahiptir.

Tablo 5-14 MIPS komutlarının J-tipi formatı

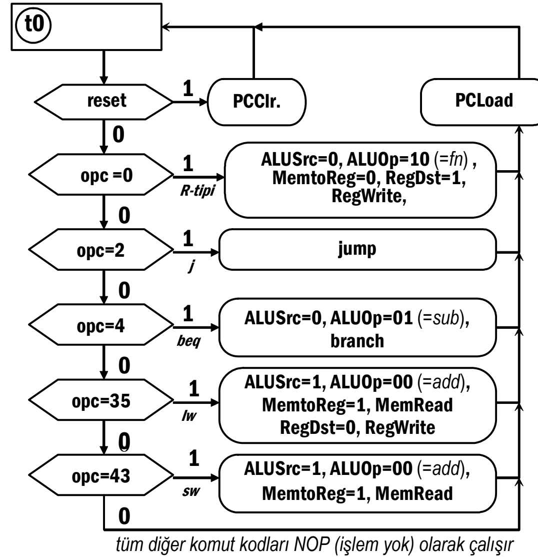
alanlar:	$b_{31}..b_{26}$	$b_{25}..b_{21}$	$b_{20}..b_{16}$	$b_{15}..b_{11}$	$b_{10}..b_{6}$	$b_{5}..b_{0}$
J-tipi	opc	**immJ: yerel-sözcük-adresi**				
j 400000	2	**100000**				

Jump (*j*, opcode=2) komutunun gerçeklenmesi için atlanacak adresi PC -nin sağ 28 bitine aktaracak bir PC-çoklayıcısı daha gerektirir. PCnin en soldaki dört biti olan PC_{31-28} ise değişmeden kalır.



Şekil 5-27 Komut altkümesi (*R-tipi, lw, sw, beq* ve *j*) için veri yolu diyagramı.

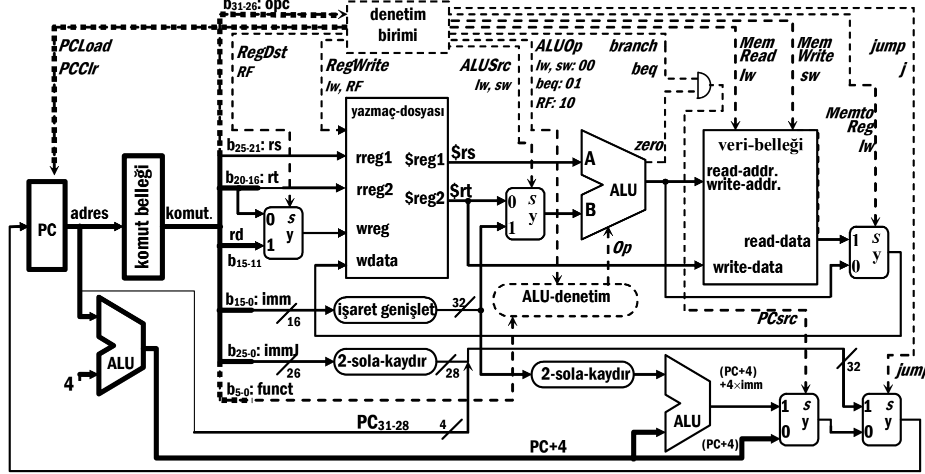
Komut altkümesinin tümü (*and*, *or*, *add*, *sub*, *slt*, *lw*, *sw*, *beq* ve *j*) için çalışan Veriyolu Diyagramının tümü Şekil 5-27'de gösterilmektedir.



Şekil 5-28 (*R-Format, lw, sw, beq* ve *j*) komut altkümesi için ASM çizimi

5.2.6 Komutların Yürütme Ayrıntıları

Bütün komutların yürütülmesinde ilk evre komut okumadır . *PCLoad* sinyali verildiğinde, *PC* nin değeri *branch/jump/PC+4* çoklayıcılarının çıkışından güncellenir. İşlem, *PC* nin değeri değişmesiyle birlikte başlar. *PC* komut belleğine yeni adresi verir. Komut belleği bu adresi çözümler ve işaret edilen komutun içeriğini çıkarır. Komut birkaç alandan (*opc, rt, rs, rd, sa, fn, imm*, yada *immJ*) oluşur.

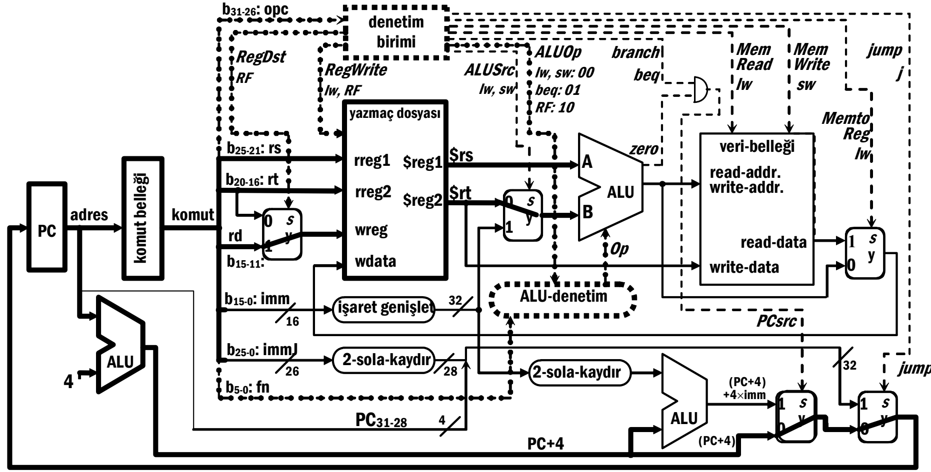


Şekil 5-29 Komut okuma (fetch) evresinde kullanılan veriyolları

Komut-okuma tamamlandığında, komut alanları ileriki işlemler için kararlı ve kullanılabilir duruma gelir. Bellek adresi çözümlerken, ALU devresi aynı adrese 4 ekler. Bu dönem boyunca komut kodu belirsiz olduğundan, çoklayıcı-seçici-hatları da bilinmemektedir.

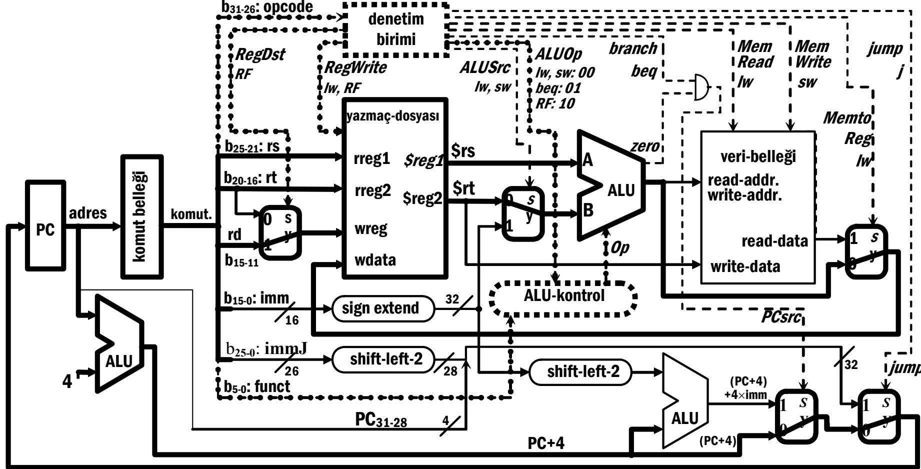
Komut okuma komut işlem kodunun denetim birimine gitmesini sağladığında, denetim birimi komutu çözümler, ve her komut için belirli eylemleri başlatır. Aşağıdaki şekiller denetim unitesinin işini ve ardından çeşitli komutların veri yolunda ilerleyişini gösterir.

Ileriki şekillerde çizimleri basitleştirmek amacıyla *PCLoad* ve *PCClr* sinyalleri kaldırılmıştır. Dikkatinizi çektiyse, neredeyse tüm şekillerde aynı nedenle saat sinyalleri de gösterilmemiştir.



Şekil 5-30 *R-formatı* komut: Komut-okumanın ardından (2. evre)

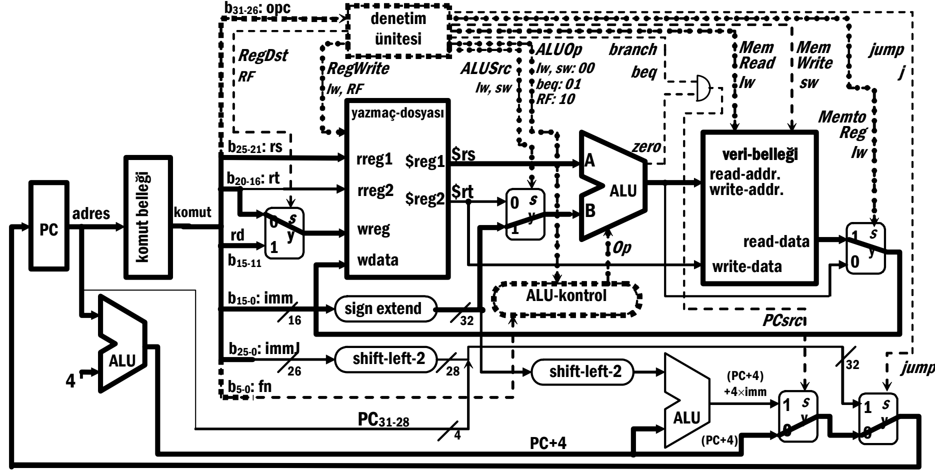
Komut kodu kulllanılmaya hazırdır. Denetim birimi *işlem kodu* alanını çözümler (=0 anlamı R-format); *RegDst*, *RegWrite*, ve *ALUOp*=10 verir. Diğer tüm denetim çıkışları uygulanmazlar (=0). *rt, rs, rd* komut alanları yazmaç dosyasının *rreg1, rreg2* ve *wreg* girişlerine bağlıdır. Yazmaç dosyası *rreg1, rreg2* yazmaçlarındaki değeri *$reg1* **(=*$rs*)** **,** *$reg2* **(=*$rt*)** çıkışlarından verir. *$rs ALU-A*-girişine bağlıdır. *ALUSrc* çoklayıcısı *$rt* -yi *ALU-B*-girişine gönderir. Aynı sırada ALU-denetim-birimi *fn* ve *ALUOp* -dan *Op* -u belirler. Ayrıca *PCSrc* ve *jump* sinyalleri verilmediğinden *PC+4* değeri PC girişine aktarılır. Ancak, *PC -nin* değeri saat etkinliğine dek değişmez.



Şekil 5-31 R-formatı komut: 3. evre (son evre):

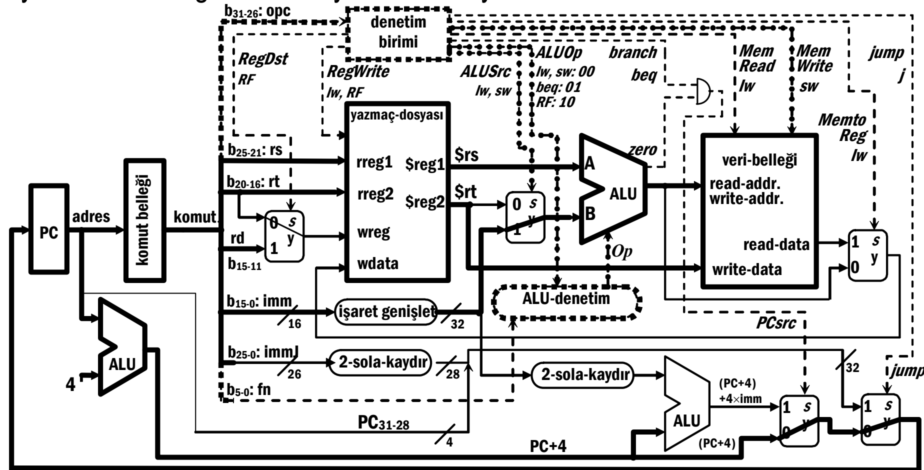
R-format komutta ALU işlemi yürütmenin son aşamasında gerçekleşir. ALU *$rs* ile *$rt* **-**yi işler ve *MemtoReg*=0 sayesinde ALU sonucu yazmaç-dosyasının yazılacak veri (*wdata*) girişine yönlendirilir. Yazmaç-dosyasının

yazılacak-yazmaç (*wreg*) girişi komutun *rd*-alanından gelir. Yürütmenin yayılması tamamlanır ve tüm sinyaller diner. Denetim-birimi *RegWrite* verdiğinden saat sinyalinin etkin kenarında ALU-sonucu *rd* -ye yazılır.



Şekil 5-32 Komut-okuma sonrası *lw* komutu:

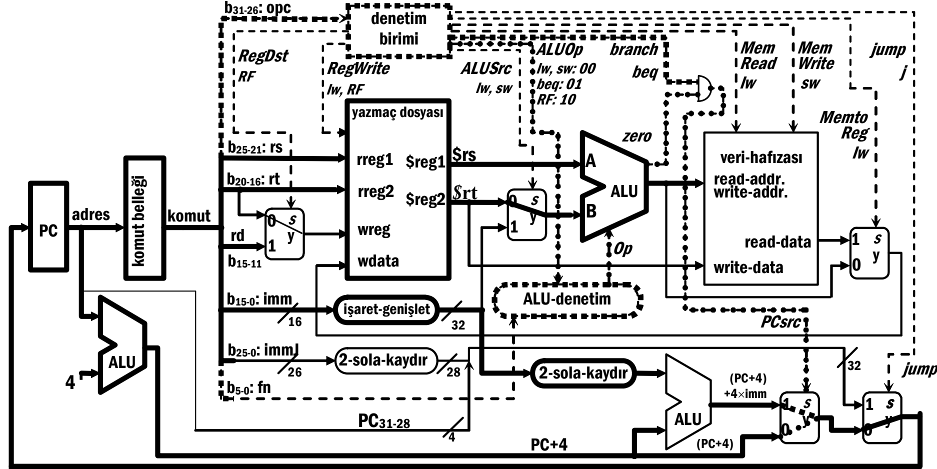
Denetim-birimi komut işlem kodunu çözümleyerek *RegWrite, ALUSrc, ALUOp*=00, *MemRead***,** ve *MemtoReg* sinyallerini verir. ALU-denetim birimi *ALUOp*=00 nedeniyle *Op*=010 (=*add*) çıkışını verir. *ALUSrc* verildiğinden ötürü 32-bite işaret-genişletilen 16-bitlik *imm* alanı *ALU-B*-girişine iletilir. *ALU, $rs+imm* adresini hesaplar ve *MemRead* veri-belleğinin o adresindeki sözcüğün okunmasını sağlar. *MemtoReg* sayesinde veri belleğinin çıkışı yazmaç-dosyasının *wdata* girişine yönlendirilir. *RegWrite* etkin saat kenarında veri belleği çıkışının *rt* yazmacına yazılmasını sağlar. Aynı saat kenarında ayrıca *PC+4* değeri de PC-yazmacına yazılır.



Şekil 5-33 Komut-okuma sonrası sw komutu:

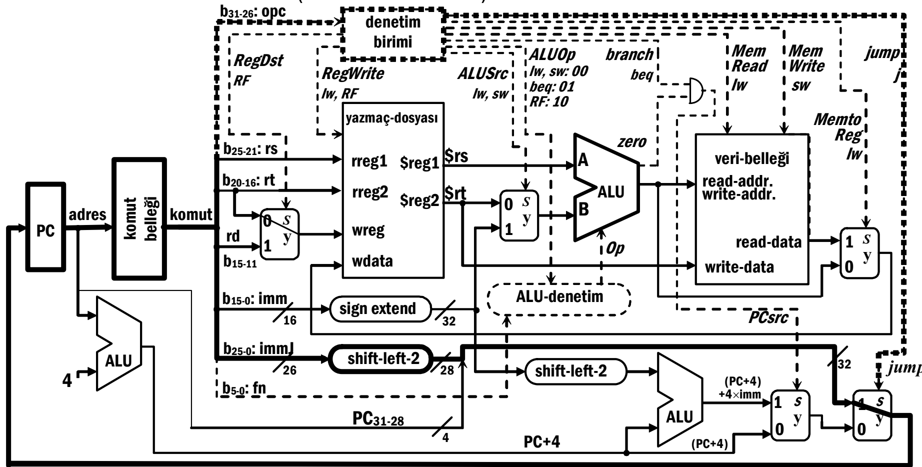
SW komutu için denetim-birimi komutun işlem kodunu çözümler, ve *ALUSrc, ALUOp=00,* ve *MemWrite* sinyallerini verir. ALU-denetim birimi çıkışı *Op*=010

(=*add*) olur. *ALUSrc* 32-bite işaret genişletilmiş 16-bit *imm* alanını *ALU-B*-girişine yönlendirir. ALU, *\$rs+imm* adresini hesaplar. *MemWrite* saatin etkin kenarında *\$rt* nin adreslenen veri belleğine yazılmasını sağlar. Aynı saat kenarında *PC*-yazmacına da *PC+4* yazılır.



Şekil 5-34 I-tipi *beq* komutunu işleyen veri yolu

BEQ komutu yürütülürken denetim birimi *ALUSrc*=01, ve *branch* sinyallerini üretir. ALU-denetim Op=110 (=çıkartma) üretır ve ALU yazmaçtaki değerleri karşılaştırmak üzere \$rs – \$rt işlemini yapar. *ALU-zero* çıkışı*PCSrc* çoklayıcı seçme sinyalini belirler. Aynı sırada bir diğer ALU *dallanma-hedef-adresini* (= *PC+4+imm×4*) hesaplar. Saatin etkin kenarında PC yazmacına ya *sonraki-yerin-adresi* (=PC+4) ya da *dallanma-hedef-adresi* (= PC+4+imm×4) aktarılır.



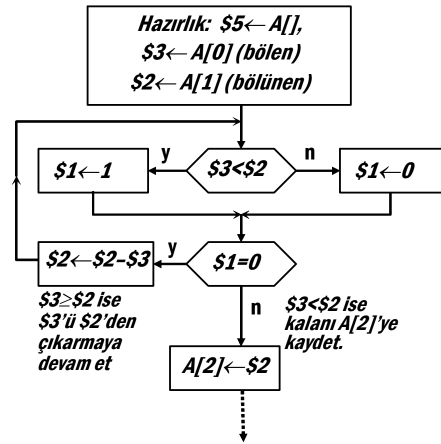
Şekil 5-35 J-tipi: *jump* (=*atla, j*) komutunu işleyen veri yolu

Jump komutunda komut işlem kodu *opc* dışındaki tüm alanlar *immJ* adında 26-bitlik sözcük-adresi taşıyan tek bir alana birleştirilmiştir. Adresi bayt-adresine dönüştürmek için *ımmJ* yi 2-bit sola kaydırmak gerekir. Kaydırma sonucu adres genişliği 28-bite

genişler. Bu nedenle, *j* komutu program sayacının yalnızca 28 bitini değiştirir. Betimlenmemiş olan en soldaki dört-bitini (PC_{31} ... PC_{28}) etkilemez.

5.3 Bir Program Kodunun Çalışması

Bir program kodunun yürütülüşünü tam resmedebilmek için bir örnek verelim. 100 -uncu adresten başlayan pozitif tamsayılar dizimiz olsun. Programımız sonuç negatif oluncaya kadar ilk tamsayıyı ikinciden çıkaracak, ve kalanı (modulus) üçüncü yerde saklayacak, yani, A[2]← (A[1] mod A[0]).



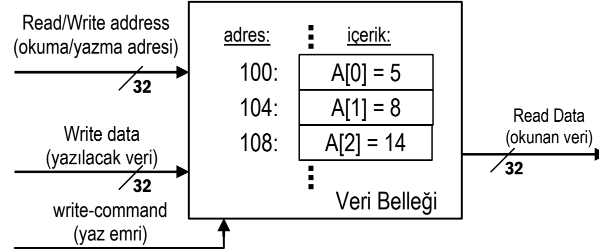
Şekil 5-36 Programın Akış şeması

Programı komut belleğinde 200. cü adresten başlayarak saklarsak Tablo 5-15 daki program kodu oluşur:

Tablo 5-15 Komut Belleğinde Program Kodu

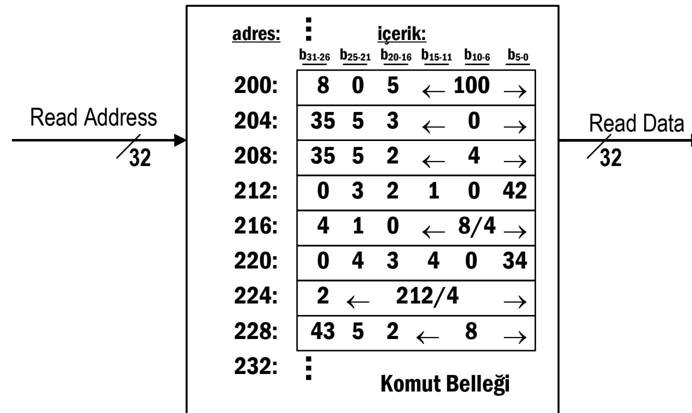
Adres	Etiket	Komut		Yorum
196:		...		
200:	modp:	addi	$5,$0,100	# $5 ← 100 ; A[] starts from 100
204:		lw	$3,0($5)	# $3 ← A[0] ; divisor
208:		lw	$2,4($5)	# $2 ← A[1] ; dividend
212:	loop:	slt	$1,$3,$2	# if($3 < $2) $1←1 else $1←0 ;
216:		beq	$1,$0, exit	# if($1= =0) goto exit ;
220:		sub	$2,$2,$3	# $2 ← $2 – $3 ;
224:		j	loop	
228:	exit:	sw	$2,8($5)	#A[2]←$2 ; (A[1] mod A[0])
232:		...		

A[] dizisi veri belleğinde 100 üncü yerden başlıyor olsun. Veri belleğinin öbek gösterimi Şekil 5-37 de görülmektedir.



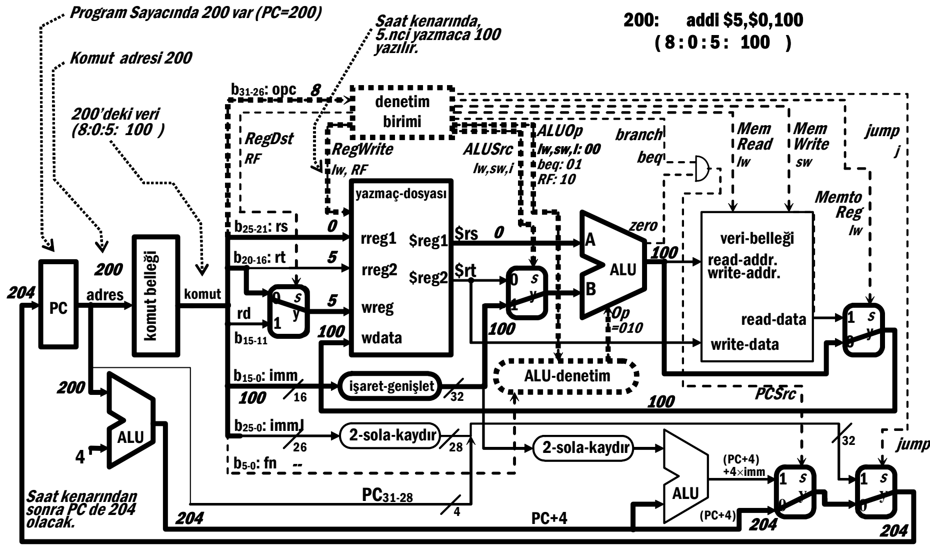
Şekil 5-37 Verı belleğinin öbek gösterimi ve başlangıc değerleri

Ve komut belleğindeki makine kodu ise Şekil 5-38 de görülmektedir.



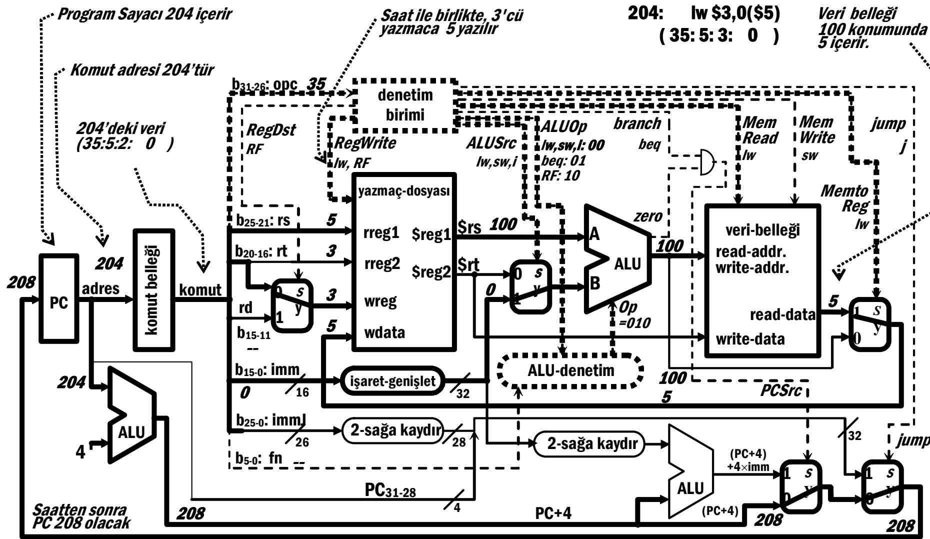
Şekil 5-38 Komut belleğindeki değerler.

İlkel işlemcimizin program sayacını 200 -den başlatarak çalışmasını izleyelim. *PC*=200 olduğundan, ilk olarak komut belleğınde 200.cü adresteki *addi* komutu işlenir. *addi* komutu işlenirken sinyallerin yayılışı Şekil 5-39 da verilmiştir.



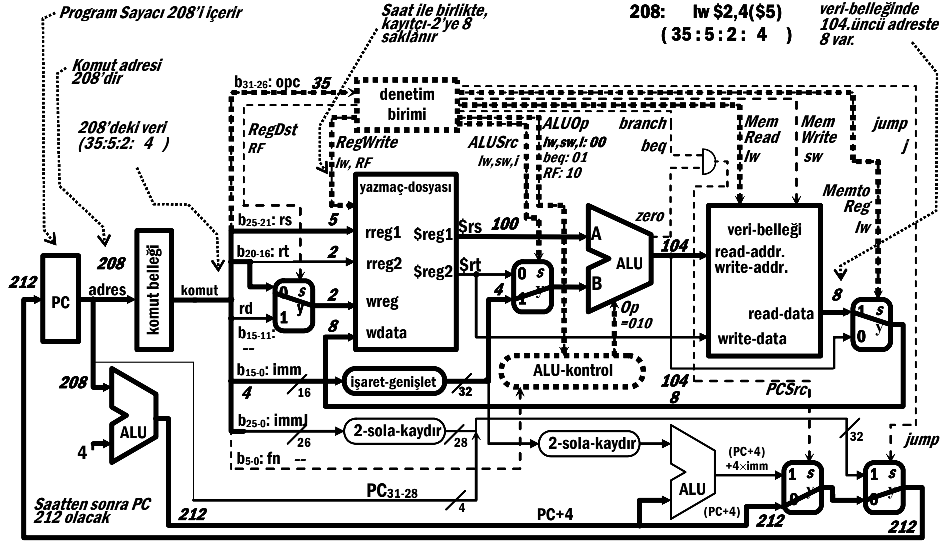
Şekil 5-39 PC=200, *addi* komutu

Saatin etkin kenarıyla birlikte, *PC* -ye 204, ve $5 -e 100 yazılacak ve işlemci komut belleğindeki 204 adresli *lw* komutunu yürütmeye başlayacaktır



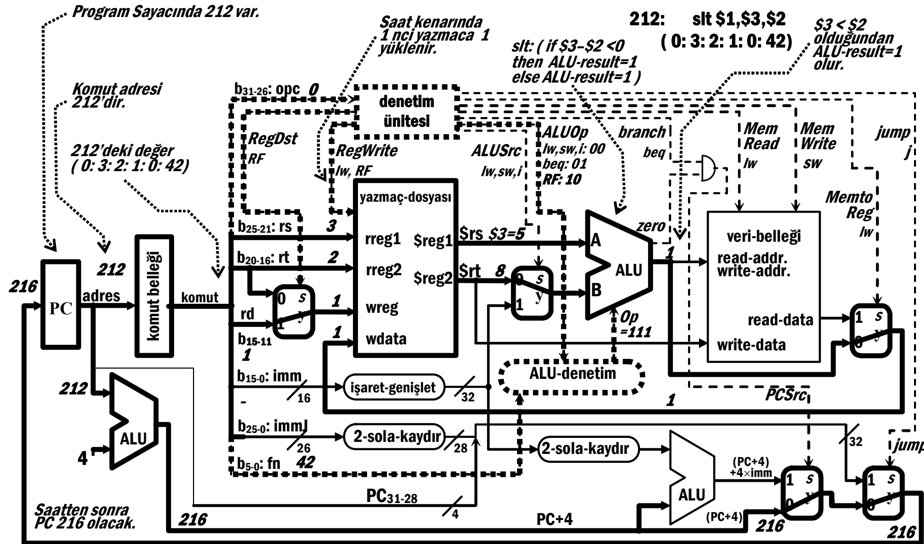
Şekil 5-40 PC=204, *lw* komutu

Saatin etkin kenarının ardından *PC* ye 208, ve $2 ye 8 yazılır. İşlemci komut belleğindeki 208 adresli komutu çalıştırmaya başlar. 208 adresli yerde **lw $2,4($5)** bulunmaktadır.



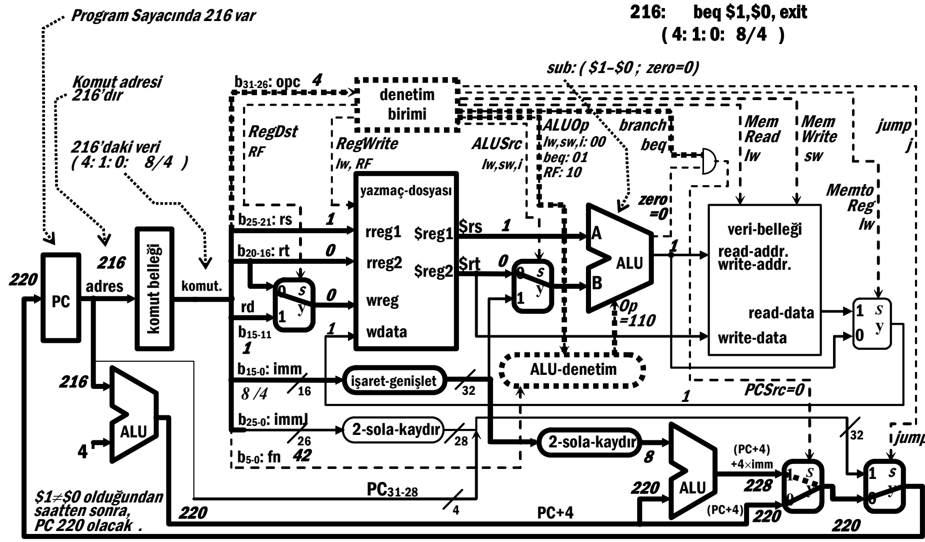
Şekil 5-41 PC=208, *lw* komutu

Veri-belleğinin 104 adresli yerinde 8 olduğundan 2 nci yazmaca 8 yüklenir. Bir sonraki komut, 212 adresindeki **slt $1,$3,$2** yazmaç tipi bir komuttur.



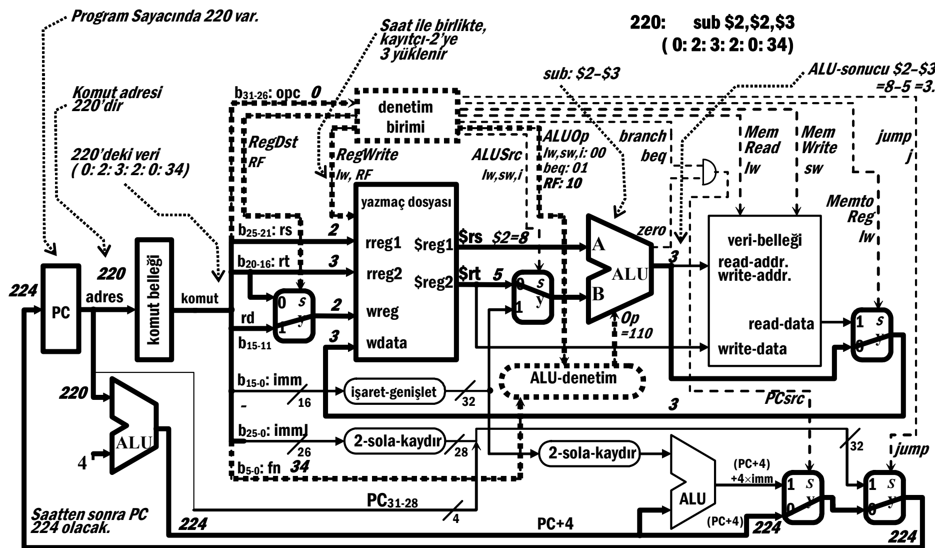
Şekil 5-42 PC=212, *slt* komutu

slt -yi işlerken *opc*=0 nedeniyle *ALUOp* 10_2 olur. *fn* =42 olduğundan ALU-denetim çıkışında *Op*=111_2 çıkartır ve ALU -da *SLT* işlemi yapılır. Bu örnek için $3=5 değeri $2=8 -den küçük olduğundan ALUdan *ALU-result*=1 çıkar. Saatin etkin kenarında, $1 -e "1" yazılır, ve *PC*=216 olur.



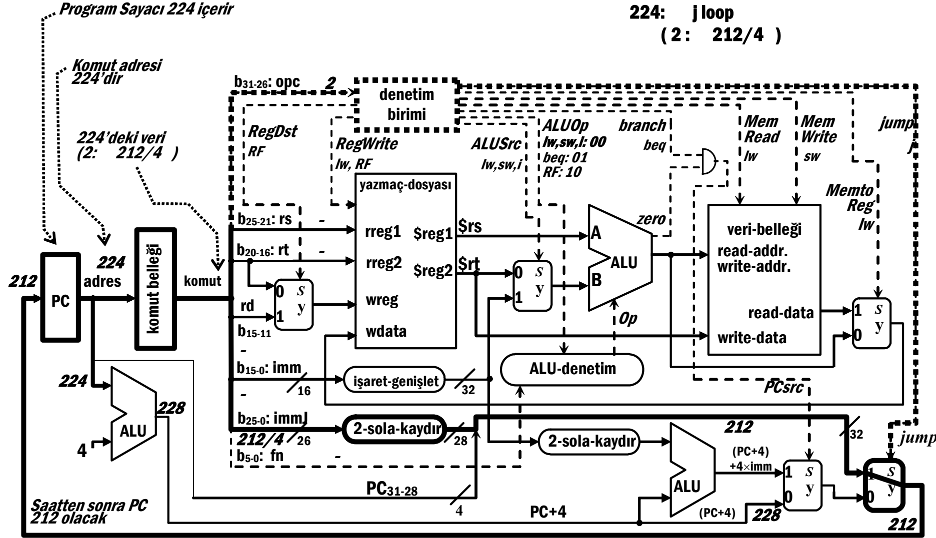
Şekil 5-43 PC=216, *beq* komutu bir sonraki komuta devam ederken

BEQ komutunda ALU -daki çıkarmanın sonucu sıfır çıkmadığından *zero* çıkışı sıfır (düşük) olur. *Branch* ile *zero* çıkışının VE-lenmesinden oluşan *PCSrc*=0 olur. *PCSrc* çoklayıcının PC -ye *dallanma-hedef-adresi* yerine *PC+4* -ü iletmesini sağlar . Böylece, saat kenarından sonra, PC 220 olur.



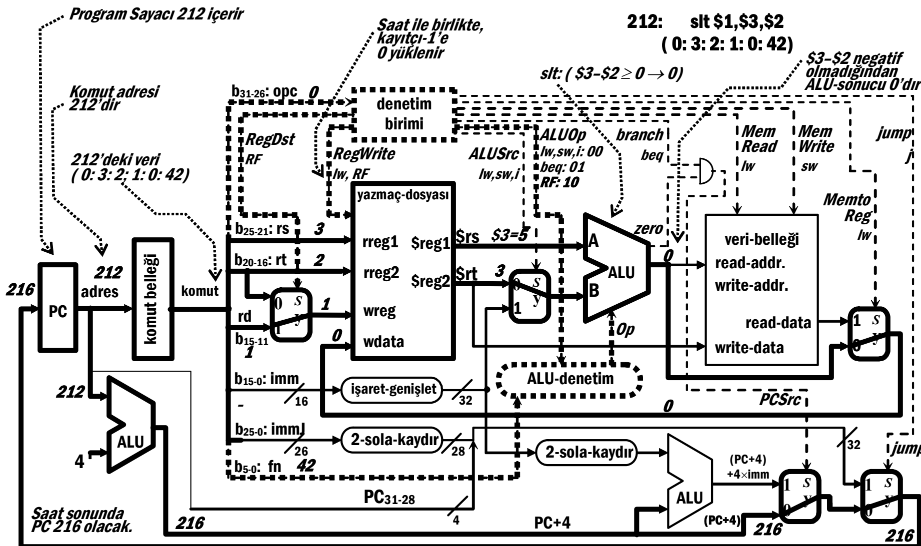
Şekil 5-44 PC=220, *sub* komutu

Komut belleğinin 220 -ci adresinde çıkarma (*sub*) komutu bulunmaktadır. Komutun işlendiği saatin sonunda $2 -ye $2–$3 = 8–5=3 ve PC -ye 224 yazılır. Adres 224 -teki atla (jump) komutu *PC* -ye 212 yazar:



Şekil 5-45 PC=224, *jump* komutu

PC 212 olduğunda, işlemci, program kodunda döngü ile gösterildiği gibi 212 -deki komutu bir kez daha işler. Ancak, dikkat ederseniz her defasında yazmaçta ve veri belleğindeki değerler bir önceki işlemdekinden farklı olabilir. \$2 -ye en son 3 yazıldığından **slt \$1,\$3,\$2** -nin ikinci çalışmasında, \$3=8 < \$2=3 koşulu sağlanamaz. Bu nedenle ALU-çıkışı ve akabinde \$1 0 olur.



Şekil 5-46 PC=212, *slt* komutu bu kez sıfırla sonuçlanır

Bu kez, \$1 sıfır olduğundan **beq \$1,\$0,exit** dallanma-hedef-adresine dallanmayla sonuçlanır .

Şekil 5-47 PC=216, *beq* komutu dallanma-hedef-adresine giderken

Böylece işlemci döngüden çıkar ve "*exit*" etiketiyle belirtilen 228 -ci adresten çalışmaya devam eder.

Şekil 5-48 PC=228, *sw* komutu

PC komut belleğindeki diğer komutların adreslerini gösterdikçe İşlemci o adresteki komutu çalıştırmaya devam edecektir. Dikkat ettiyseniz bu kod parçası çalıştıktan sonra A[] dizisinin üçüncü elemanında hesaplanan değer, yani A[2] = 8 mod 5 = 3 bulunur.

5.3.1 Tek-Dönüşlü İşlemenin Sakıncaları

İşlemci donanımındaki her birim, sinyalleri işleyip çıkış sinyalinin kararlı duruma gelmesi için zaman harcar. Her komutu bir saat-dönüşünde işleyen bir işlemcide, donanım kaynaklarının bir kısmı ya önceki birimlerden gelen sinyallerin kararlılaşması için beklerken, yada sonraki birimlerin işlem süresinin tamamlanmasını beklediğinden boş kalır.

Bizim gerçeklememizde, tüm komutlar bir saat dönüşü sürdüğünden komutların tümü için CPI = 1 -dir. Tüm komutların aynı saat dönüşünde çalışması için saat dönüşü süresi tüm komutların en uzun süren yol seçeneğinden bile uzun olmalıdır. Gerçeklenen komut kümesinde, sözcük yükle (*lw*) komutu komut okumak için *komut belleğini*, baz-adresini okumak için *yazmaç-dosyasını*, adres hesaplaması için *ALU* -yu, adresteki veriyi okumak üzere *veri-belleğini*, ve son olarak veriyi komutta belirtilen yazmaca yazmak için tekrar *yazmaç-dosyasını* kullanır. Bu birimlerin işlem zinciri en uzun veri işleme ardışık düzenini (*pipeline)* oluşturur. Öte yandan diğer bazı komutlar daha az zaman gerektirir. En kısası işlemcinin sadece komut-okumak için komut belleğini kullandığı atla (*j*) komutudur. Ana işlev birimlerinin işleme yada erişim zamanlarını Tablo 5-16 daki gibi alalım.

Tablo 5-16 Ana işlev birimlerinin işlem süreleri

işlem ve işlevsel birim	**yaklaşık işlem süresi (ns)**
Belleği okuma	10
Belleğe yazma	10
ALU ve toplamalar	10
Yazmaç-dosyası okuma	5
Yazmaç-dosyası yazma	5
Mux, Denetim Birimi, PC, işaret-genişletme, sola-kaydırma, iletim hatları	ihmal edilebilir

Komut altkümemizdeki her komut için minimum saat zamanı sürelerini Tablo 5-17 'deki gibi buluruz.

Tablo 5-17 Tek dönüşlü işlemcide her komutun minimum saat dönüş süresi

Komut tipi	**Komut belleği**	**Yaz.dos. okuma**	**ALU**	**Veri belleği**	**Yaz.dos. yazma**	**Toplam (ns)**
R-f	10	5	10	-	5	30
lw	10	5	10	10	5	40
sw	10	5	10	10	-	35
beq	10	5	10	-	-	25
J	10	-	-	-	-	10

Umduğumuz gibi, en fazla zaman alan komut *lw* -dir, ve işlemin tamamlanması için 40ns gerektirir. Yani, eğer tüm komutlar tek saat dönüşünde çalışmak zorunda olsa, işlemci *lw* komutunu tamamlamak için 40ns gerektirir, ve bu yüzden işlemcinin saat-dönüşü en az 40ns olmak zorundadır.

Düşük çalışma zamanlı komutların boşta geçen sürelerini azaltarak işleme hızını iyileştirebilir miyiz? Her komut için değişken saat-dönüşü kullanmamız işlem süresini düşürebilir. Ama bu pratikte sakıncalı bir yoldur. Daha basit bir diğer teknik ise aşağıdaki örnekteki gibi, örneğin 5ns saat-dönüşü gibi daha kısa saat-dönüşleri kullanmak, ve denetim biriminde program sayacını komutun işlem koduna bağlı olarak sadece yeterli sayıda döngü bekledikten sonra *PCLoad* verecek bir dönüş sayıcı kullanmaktır.

Örnek 5-1*:* Tablo 5-18 deki komut karışımı için
a- Sabit uzunlukta tek-saat-dönüşü gerçeklemeli işlemci.
b- Saat-dönüşü=10ns ile her komutun farklı saat-dönüşü sürdüğü değişken süreli gerçeklemeli işlemci
seçeneklerinin başarımlarını bulun

Tablo 5-18 Başarım bulmakta kullanılacak komut karışımı

Komut-tipi	R-tipi.	lw	sw	branch	jump
test programı kompozisyonu:	49%	22%	11%	16%	2%

<< Gerçeklenen veri yolu için Tek ve Çok-döngü durumları.
komut-çalışma-zamanı = CPI × saat-döngü-zamanı.

	İşlem Süresi (nanosaniye)						
Komut tipi	**Komut belleği okuma**	**Yazmaç dosyası okuma**	**ALU**	**Veri belleği**	**Yazmaç dosyası yazma**	**tek dönüş**	**çok dönüş**
R-f (49%)	10	5	10	-	5	30	3×10
lw (22%)	10	5	10	10	5	40	4×10
sw (11%)	10	5	10	10	-	35	4×10
beq (16%)	10	5	10	-	-	25	3×10
J (2%)	10	-	-	-	-	10	1×10

Önce her seçenek için CPU-çalışma-zamanını hesaplayalım
CPU-çalışma-zamanı = Komut sayısı× CPI× saat dönüş süresi
= Komut sayısı × saat dönüş süresi
a) Tek-saat dönüşü gerçeklemesi için:
saat dönüş süresi = 40 ns, CPI= 1 ;
CPU-çalışma-zamanı = I × 1 × 40 ns = **40×I ns**
b) Çok-saat dönüşü gerçeklemesi için ortalama CPI:
CPI = $\sum (CPI_k \times \text{Komut-sayısı}_k)$

Komut-sayısı

$$CPI = (3 \times 0.49)+(4 \times 0.22)+(4 \times 0.11)+(3 \times 0.16)+(2 \times 0.02) = 3.31$$

CPU-çalışma-zamanı = I × 3.31 × 10 ns = **33.1×I ns**

İki gerçeklemenin başarımlarının oranı::

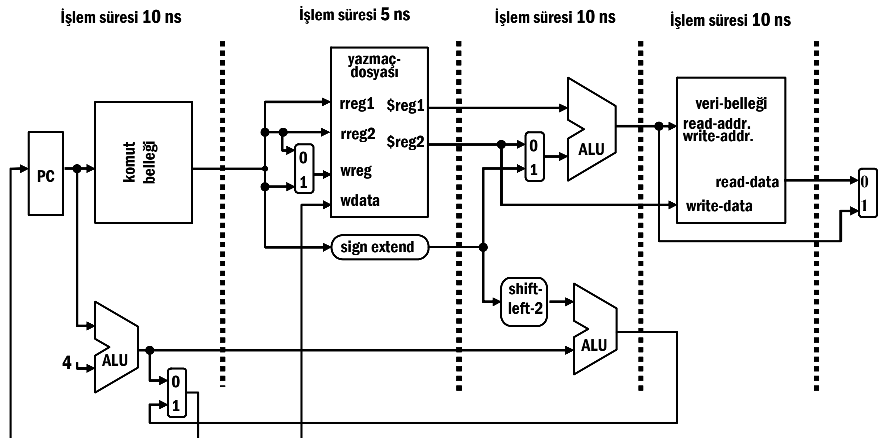
$$n = \frac{\text{Çok-saat-donüşlü Başarım}}{\text{Tek-saat-dönüşlü Başarım}} = \frac{40 \times I \ ns}{33.1 \times I \ ns} = 1.21$$

Değişken saat gerçeklemesi sabit-tek-saat.dönüşlü gerçeklemeye göre 1.12 defa daha hızlıdır.
>>

Eğer tasarımda kayar noktalı işlemler, yada daha karmaşık bir komut kümesi olacaksa, tek saat dönüşlü tasarım çok yavaş saat dönüşü gerektirecektir. Bu problemi günümüz gerçekleme teknikleri her komut için çok saat dönüşü kullanıp saat dönüşünü kısaltarak çözer. Gelecek bölümde, her komutun işlem süresini düşürmek için çoklu veri-işlem birimlerinin ardarda aynı anda çalışmasına izin veren ardışık düzen merkezi işlem biriminin çok saat dönüşlü gerçeklemesi için iyileştirilmiş teknikler göreceğiz.

5.4 Çok-Dönüşlü Gerçekleme

Tek saat-dönüşlü gerçekleme komut okuma ve veri erişimi için ayrı bellek birimleri gerektirir. Basitleştirilmiş bir veri işlem yolu gösterimi komut belleğiyle başlar ve veri belleğiyle biter. Bu veri yolu üzerinde neredeyse aynı işlem süresi gerektiren dört ana bölümümüz vardır.



Şekil 5-49 Basitleştirilmiş bir tek-dönüşlü veri işlem yolu dört ana bölüme ayrılabilir

Bölümlemede, bölümler arasında veriyolu elemanı kalmamasına, ve her bölümün işlem süresi yaklaşık olarak aynı olmasına dikkat edilir.

Çok-dönüşlü işlemden beklenen yararlar şunlardır:

① Komutların çalışması çeşitli adımlara bölünmüştür. Eğer bir komut bir adımı gerektirmiyorsa o komutun çalışma zamanını azaltmak için o adım atlanabilir.

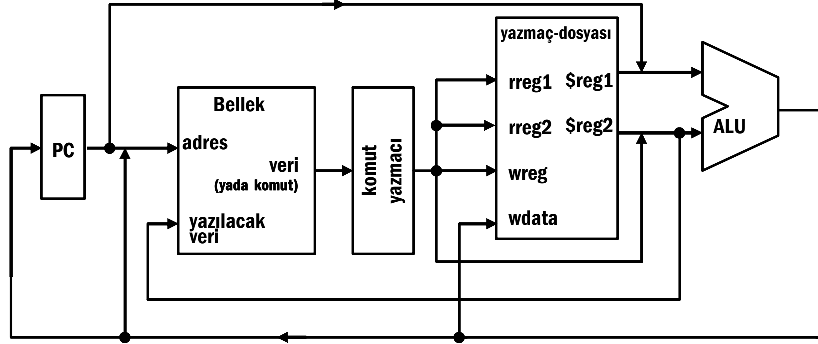
② Her kısım neredeyse aynı işleme zamanı gerektirir ve bir saat dönüşünde tamamlanabilir.

③ Farklı saat dönüşlerinde kullanılmak şartıyla aynı işlem birimini komut başına bir defadan fazla kullanabiliriz. Bu potansiyel, donanımın azalmasını sağlar.

④ Bir tek bellek birimi hem komut okuma hem de veri için kullanılabilir. Bu, işlemciye esneklik sağlar ve bir bellek birimi azaldığından maliyeti düşürür.

Ancak, çok-saat-dönüşlü veriyolunda, program sayacınındaki değer ilk saat dönüşünde değişse bile komutun kaybolmamasını sağlamak üzere komutun bir komut yazmacına saklanması gerekir.

Yukarıda lıstelen yararları sağlayacak basitleştirilmiş çok-saat-dönüşlü bir veriyolu örneği Şekil 5-50 de görülmektedir.



Şekil 5-50 Tipik basitleştirilmiş çok saat dönüşlü veriyolu örneği

Basitleştirilmiş veri yolunda, tüm aritmetik işlemler için aynı ALU kullanılmaktadır. İlk iki adımda henüz komut çözülmemiş olduğundan veriyolu bütün komutlar için aynı çalışır. ALU ilk adımda, PC+4 -ü hesaplar. İkinci adımda yazmaç dosyası işlenenler için henüz hazır değilken dallanma-hedef-adresi hesaplanır. Üçüncü adımda R-tipi komut için ALU komutta fn kodu ile betimlenen aritmetik-mantık işlemini yapar. *lw* ve *sw* baz-adreslemelı komutlarında bellek adresi üçüncü adımda hesaplanir. Dallanma komutunda ise ALU üçüncü adımda yazmaçları karşılaştırır. R-Format komutta dördüncü adımda

işlemci sonucu hedef yazmacına yazar. Aynı adımda *sw* komutunda betimlenen yazmaçtaki deger belleğe yazılır. En uzun komut, *lw*, dördüncü adımda okunan veriyi hedef yazmaca yazmak için beşinci adıma gerek duyar.

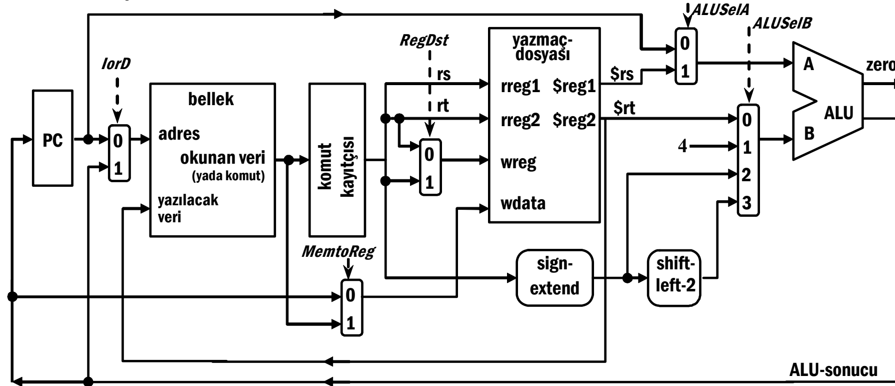
Tablo 5-19 Çok-dönüşlü veri yolunda her dönüşteki ALU kullanımı

komut	adım①	adım ②	adım ③	adım④	adım ⑤
R-tipi	PC+4	dallanma hedefi ×	fn işlev kodu	yazmaca yazış ✓	
lw	PC+4	dallanma hedefi ×	bellek adresi	bellek.okuma ✓	yazmaca yaz ✓
sw	PC+4	dallanma hedefi ×	bellek adresi	bellek.yazma ✓	
beq	PC+4	dallanma hedefi	karşılaştırma		
j	PC+4	dallanma hedefi ×			

×: hesaplanır ve hedef yazmacına yazılır, ancak kullanılmaz.
✓: bir önceki ALU işlemi belirtilen nedenle devam eder.

5.4.1 Çok-saat-dönüşlü Veriyolu

İşlem adımlarının bu adımlarını temel alarak Şekil 5-51'deki veri yolunun ayrıntılarına bakalım.



Şekil 5-51 Çok saat dönüşlü veri yolunun geliştirilme aşaması

Bu veriyolunun tek-dönüşlü veriyolundan temel farkları:

1- Komut ve veri için paylaşılan sadece bir tek bellek birimi vardır.
2- Okunan komutu saklamak üzere komut-yazmacı kullanılmaya başlandı.
3- Adres hesabı dahil tüm aritmetik-mantık işlemleri tek ALU -da yapılır.
4- Çeşitli işlem adımlarında fonksiyonel birimleri paylaşmak üzere önceki veriyolundakinden daha fazla çoklayıcı gerekir.

a- Tek bir bellek hem komut belleği hem veri belleği olarak kullanılmakta, ve bu nedenle ***bellek adresi***, şunlar arasında çoklanır.

(0): PC (komut-adresi),
(1): ALU-sonucu (hesaplanan-adres)

b- Aynı ALU da PC+4, dallanma-adresi, hesaplanan-bellek-adresı, ve R-tipi komutların aritmetik-mantık işlemleri yapılmaktadır.
i- ***ALU nun A-girişi***nde şunlardan birini seçmeye 2x1 çoklayıcı
(0): PC,
(1): yazmaç-dosyası birinci çıkışı

ii- ***ALU nun B-girişi***nde şunları seçmeye 4x1 çoklayıcı gerekir
(0): yazmaç-dosyası ikinci yazmaç çıkışı (R-tipi komutlar için),
(1): PC+4 için bir 4 sabiti (tüm komutlar için),
(2): işaret-genişletilmiş anlık değer
(anlık işlenenlerle hesaplanan-bellek-adresi için),
(3): işaret-genişletilmiş ve 2-bit-sola-kaymış anlık değer
(PC göreceli dallanma adresi için).

c- Yazmaç dosyasının ***yazılacak-veri-girişi***nde şunları seçen 2×1 çoklayıcı gerekir.
(0): ALU-sonucunu (*R-Format* 3.adım) ,
(1): Bellekten okunan veriyı (*lw* 5.adım) .

Tablo 5-20 de görüldüğü gibi donanım maliyetindeki düşüş açıktır.

Tablo 5-20 Çok ve tek donüşlü veriyolu donanım elemanlarının karşılaştırılması

Donanımdaki kazanç	Donanımda ekstra maliyet
-iki yerine bir bellek birimi -üç yerine bir ALU	-komut yazmacı -yeni çoklayıcılar

5.4.2 Çok-saat-dönüşü Denetim Sinyalleri

Bu veriyoluna eklediğimiz yeni çoklayıcı ve yazmaçların çalışmasını denetleyebilmek amacıyla yeni denetim sinyalleri tanımlamalıyız. Birden fazla saat-dönüşlü komutları gerçekleştirirken şu sırasal devre tasarım kurallarını uygulamak gerekir:

1- Bellek, PC, Yazmaç Dosyası, Komut Yazmacı gibi sırasal öbeklerin (durum elemanları) herbirine bir “yazma" (*write*) sinyali gerekir,

2- Bellek için bir de “okuma” (*read*) sinyali gerekir,

3- Önceden tanımlanan bileşimsel birimlerin denetim sinyallerinin bir çoğu burada da gerekir.
   i- Önceki "*ALU-denetim-birimi*" sinyalleri burada da kullanılabilir.
   ii- 2-girişli dört çoklayıcının birer bitlik seçme girişlerine, ve
   iii- 4-girişli tek çoklayıcı için iki-bitlik seçme denetim sinyali gerekir.

Dallanma (branch) komutunu henüz en ince ayrıntısına kadar planlanamadık. Denetim sırasını adım adım düzenlerken bu komut için fazladan bazı donanım birimleri gerekebilir.

Tablo 5-21 Çok-dönüşlü-Veriyolu Denetim sinyalleri

Tek bit Çıkışlar:

Sinyal Adı	Değer	Etkinlik (a→b , "a b ye aktarılır" anlamına kullanılmıştır)
MemRead	**Belleği okuma sinyali,** belleğin okunacak adresindekı veriyi belleğin veri çıkışına aktarır	
	0	-
	1	Bellek[*adres*] → belleğin *veri* çıkışı.
MemWrite	**Belleğe yazma sinyali,** *veri* belleğin adreslenen yerine yazılır	
	0	-
	1	*veri* → Bellek[*adres*]
ALUSelA	**ALU-A-girişi çoklayıcı seçme sinyali**	
	0	*PC* → *ALU-A* girişi
	1	*$rs* kaynak yazmacı değeri → *ALU-A* girişi
RegDst	**Yazmaç-dosyasına yazılacak veri girişi çoklayıcı sinyali**	
	0	komut yazmacının *rt* alanı → yazmaç dosyasının *wreg* girişi
	1	komut yazmacının *rd* alanı → yazmaç dosyasının *wreg* girişi
RegWrite	**Yazmaç dosyasına yazma sinyali,**	
	0	-
	1	Yazmaç dosyası wdata girişi → wreg ile seçilen yazmaç
MemtoReg	**Yazmaç dosyasının yazılacak-veri girişi çoklayıcı sinyali**	
	0	*ALU-sonucu* → yazmaç-dosyası *wdata* girişi
	1	Bellekten *okunan veri* → yazmaç-dosyası *wdata* girişi
IorD	**Bellek adres girişi çoklayıcı sinyali**	
	0	PC → Belleğin adres girişi
	1	*ALU-sonucu* → Belleğin *adres* girişi
IRWrite	**Komut yazmacına yazma (*write*) sinyali**	
	0	-
	1	Bellekten *okunan veri* komut-yazmacına (IR) yazılır

İki-bit çıkışlar:

İsim	Değer	Etkinlik
ALUSelB	**ALU-B-girişi çoklayıcı seçme sinyali**	
	00	*$rt* → *ALU-B* girişi .
	01	sabit değer “4”→ *ALU-B* girişi
	10	*IR* nin işaret genişletilmiş en sol 16 biti (IR_{15} ,... IR_{15} , ... IR_0 $)_2$ → ALU-B girişi
	11	*IR* nin işaret-genişletilmiş ve iki bit sola kaymış en sol 16 biti (IR_{15} ,... IR_{15} , ... IR_0 , *0* , *0* $)_2$ → ALU-B girişi
ALUOp	**ALU-denetim birimine gidecek işlev kodu .**	
	00	ALU toplama yapar (010 → Op)
	01	ALU çıkarma yapar (110 → Op)
	10	ALU “fn” alanının belirlediği işlemi yapar.

5.4.3 Komut Çalışmasını Saat Döngülerine Bölmek

Yeni veriyolunu belirlerken, komutların tipine bağlı olarak işlemcinin her saat dönüşünde ne yapması gerektiğini daha önce planladık. Şimdi, komut altkümemizdeki her komut tipi için ayrıntıları belirlememiz gerekir. İnce ayrıntıları geliştirirken, o komutu eldeki veriyoluyla gerçeklemek için başka bir yol bulamazsak yeni veriyolu elemanları ve denetim sinyalleri eklememiz gerekebilir.
Genelde, şu durumlarda bir geçici yazmaç gereklidir:

a) Bir sinyal bir adımda belirlenip başka bir adımda kullanılırsa, yada
b) Bir işlevsel bloğun girişi, henüz o bloğun çıkışı kullanılmadan (örneğin bir durum değişkenine yazılamadan) değişiyorsa.

Örneğin, komutu çıkaran işlevsel birimin (Bellek) çıkışı *PC* -ye bağlı olduğundan daha ilk adımda *PC* arttırılınca komut kaybolacaktır. Oysa komutun bazı alanları komut tipine bağlı olarak dört adım daha gerektirir. Bu yüzden komutun işlenmesi bitinceye kadar saklamak üzere Komut-Yazmacı (*IR*) kullanmamız gerekir.

ALU nun girişleri kararlı kaldığı sürece çıkışı da kararlıdir ve geçici olarak saklamak gerekmez. ALU girişleri ya yazmaç-dosyasından yada komut alanlarından gelir. Yazmaç-dosyası çıkışları da komut alanları tarafından belirlenir. Bu nedenle, komut kararlı kaldığı sürece bir kez durulduktan sonra, ALU çıkışı da kararlı kalır. Komut, komut-yazmacına sadece yürütmenin ilk adımında yazılır, ve komutun çalışması bitene kadar orada kalması gerekir.

Işi adımlara bölmekle veri yolu öbeklerinin her adımda boşta kalma süresini azaltmak istiyoruz. Diğer bir deyişle, saat döngü zamanını *en aza indirebilmek* için her adımda yapılan *iş miktarını dengelemeyi* amaçlıyoruz. Örneğin her adımı *en fazla bir ALU işlemi*, yada *bir yazmaç-dosyası erişimi*, yada *bir bellek erişimi* içermesi için sınırlandırırız. Dikkat ederseniz saat dönüş süremizi en uzun süren adımın işlem süresi belirleyecektir.

Bir saat dönüşünde sıralanmış tüm işlemler aynı anda paralel gerçekleşir. Bir komutun çalışması için beş adım tanımlıyoruz. İşlemdeki her adımı için elimizdeki donanım kaynaklarını mümkün olan en iyi şekilde dağıtımalıyız.

Tablo 5-22 Her komut tipi için işlem adımlarına kaynak dağıtımı

Komut Tipi	Komut adımına ayrılan işler				
	1	2	3	4	5
	Komut-Oku	**Yazmaç Oku**	**ALU işlemi**	**Bellek-oku/yaz Yazmaç-yaz**	**Yazmaç-yaz**
R-tipi	**bellek-oku**	**komut-çöz** **yazmaç-oku** ALU-adr	**ALU-veri**	**yazmaca-yaz**	
sozcük oku (lw)	**bellek-oku**	**komut-çöz** **yazmaç-oku** ALU-adr	**ALU-adr**	**bellek-oku**	**yazmaca-yaz**
sozcük yaz (sw)	**bellek-oku**	**komut-çöz** **yazmaç-oku** ALU-adr	**ALU-adr**	**bellek-yaz**	
eşitse dallan (beq)	**bellek-oku**	**komut-çöz** **yazmaç-oku** **ALU-adr**	**ALU-veri**		
atla (j)	**bellek-oku**	**komut-çöz** yazmaç-oku ALU-adr	**PC-yaz**		
Koyu olmayan işler o komut adımında yapılan ancak sonucu kullanılmayan işlerdir.					
ALU-veri: ALU da veri hesaplanıyor			ALU-adr: ALU da adres hesaplanıyor		

1- **Komut Okuma** adımı (bellek okuma - 10ns civarı)
2- **Komut Çözümleme ve Yazmaç Okuma** adımı (yazmaç okuma - 5ns. civarı. Komut henüz çözümlenmedi. ALU her durumda dallanma adresini önceden hesaplar. ALU işlemi 10 ns. tutar)
3- **Yürütme, Bellek adresi hesaplaması, yada Dallanma bitiş** adımı (Komut çözümlendi ve veriyolu ona göre ayarlandı. En-uzun-zaman-alan işlem bellek-adresi hesaplamasıdır - 10 ns)
4- **Bellek erişimi, yada R-tipi komut bitiş** adımı (yazmaç-yazma 5ns, bellek oku/yaz 10 ns civarındadır.)
5- **Yazmaca-geri-yazma** adımı (Sadece lw komutu için, veriyi bellekten çıkışından yazmaca yazan yazmaç-yazma operasyonu. Yaklaşık 5ns)

Dallanma (branch) komutunda, hedef adresi ikinci adımda daha komut çözümlenmeden önce hesaplanır. Bu demektir ki, komut dallanma komutu olmasa bile her durumda ALU boş duracağına bu işlem için kullanılır. Komut çözümlendiğinde, eğer komut dallanma komutuysa denetim birimi hedef adresini kullanmaya karar verir.

Işin adımlara dengelenmesi açısından, işlemcinin tamamı için saat-döngü-zamanı 10ns olmalı diyebiliriz. Bu durumda en uzun komut 50ns, en kısa komut 30ns -de çalışacaktır.

Şimdi döngü döngü giderek denetim biriminden hangi denetim sinyallerinin assert edilmesi gerektiğini belirleyebiliriz.

1. Komut Okuma

Komutu bellekten yakalayıp IR -ye getir, ve PC yazmacını arttır. Belirsizliği ortadan kaldırmak üzere PC yazmacındaki başlangıç değeri iPC, toplamadan sonraki değeri nPC, işlenen adımdaki değeri ise PC ile göstereceğiz..

Tablo 5-23 Tasarım aşamasında komut okuma adımında verilecek denetim sinyalleri

Komut-tipi	veriyolu işlemi	denetim sinyalleri
Tüm komutlar için	**(komut okuma)** Mem[PC] $\rightarrow$ IR	***IorD* =0**; ***MemRead*** ; ***IRWrite***
	(PC sıradaki adresi gösterir) PC + 4 $\rightarrow$ PC yada iPC + 4 $\rightarrow$ nPC	***ALUSelA* =0**, ***ALUSelB* = 01**, ***ALUOp* =00**, ***PCWrite***

2. Komut Çözümleme ve Yazmaç Okuma

Tablo 5-24 Tasarım aşamasında komut çözümleme adımında verilecek denetim sinyalleri

Komut.-tipi	veriyolu işlemi	denetim sinyalleri
Tüm komutlar için	**(yazmaç okuma)** Y.Dos[$\overbrace{(IR_{25}, .., IR_{21})_2}^{rs}$] $\rightarrow$ \$rs Y.Dos[$\overbrace{(IR_{20}, .., IR_{16})_2}^{rt}$] $\rightarrow$ \$rt	*sinyal gerekmez*
	(Hedef-Adres-Hesaplama) nPC+ $(\overbrace{IR_{15}, ..IR_{15}}^{\text{işaret-genişlet}}, .., IR_0, \overbrace{0, 0}^{\text{2-sola-kaydır}})_2$ $\rightarrow$ Hedef	***ALUSelA* =0**, ***ALUSelB* = 00**, ***ALUOp* =00**, ***TargetWrite***

Veriyolunda tek ALU olduğundan aynı adımda hem yazmaçların karşılaştırma işlemi hem de dallanma hedef adresi hesaplaması yapılamaz. Bu nedenle karşılaştırmadan onceki adımda hesaplanmış olan hedef adresi tutacak bir hedef yazmacı gerekir. Hedef adresi her durumda, komut henüz çözümlenmeden, yani komut *beq* olmasa bile hesaplanır. Bu şekilde hedef adresi önceden hesaplamak işlem süresini bir adım kısaltır.

3. Yürütme, Bellek Adresi Hesaplama, yada Dallanma Bitiş

Komut bir kez çözümlenince, denetim birimi komut-işlem-koduna -una bağlı olarak farklı sinyaller verir:

Tablo 5-25 Tasarım aşamasında yürütme adımında verilecek denetim sinyalleri

Komut-tipi	**veriyolu operasyonu**	**denetim sinyalleri**
R-tipi:	**(ALU veri işlemi)** fn($rs, $rt) → ALUSonucu	***ALUSelA* = 1, *ALUSelB* = 00, *ALUOp* = 10**
Bellek kullanım: (***lw*** veya ***sw***)	**(ALU adres işlemi)** işaret-genişletme: $rs+ (IR_{15} ...IR_{15} ... IR_0 $)_2$ → ALUSonucu	***ALUSelA* = 1, *ALUSelB* = 10, *ALUOp* = 00**
Dallanma: (***beq***) **komut biter**	**(ALU veri işlemi)** Eğer (A – B == 0) ise Hedef → PC	***ALUSelA* = 1**, ***ALUSelB* = 00**, ***ALUOp* = 01**. Eğer (ALUSonucu==0, yani *Zero* = 1) ise ***PCLoad =1***
Jump: (***j***) **komut biter**	**(PC-yazma)** (iPC+ 4) ün en sağ 4-biti: (nPC_{31}...nPC_{28} , IR_{25} ... IR_0 , 0, 0 $)_2$ → PC; 2-bit-sola-kaydır	*her durumda* ***PCLoad =1***

4. Bellek Erişimi ve R-tipi komutların bitişi adımı

Tablo 5-26 Tasarım aşamasında bellek erişim adımında gereken denetim sinyalleri

Komut-tipi	**veriyolu işlemi**	**denetim sinyalleri**
R-tipi: **komut biter**	**(Yazmaca-Yazma)** ALUSonucu → Y.Dos[(IR_{15}... IR_{11} $)_2$] (rd; $rd = rd nin içeriği)	(***ALUSelA* =1**, ***ALUSelB* =00**, ***ALUOp* =10** önceki gibidir) ***RegDst* = 1**; ***MemtoReg* = 0**; ***RegWrite*** .
Bellek kullanım: {***lw***}	**(Belleği Okuma)** Bellek[ALUSonucu] -nu okur	(***ALUSelA* =1**, ***ALUSelB* =10**, ***ALUOp* =00** önceki gibidir) ***IorD* = 1, *MemRead*** .
Bellek kullanım: {***sw***} **komut biter.**	**(Belleğe Yazma)** $rt; rt: Y.Dos[($IR_{20}$, ..,$IR_{16}$$)_2$] → Mem [ALUSonucu]	(***ALUSelA* =1**, ***ALUSelB* =10**, ***ALUOp* =00** önceki gibidir) ***IorD* = 1**; ***MemWrite*** .

5. Yazmaca Geri Yazma Adımı

Tablo 5-27 Tasarım aşamasında geri-yazma adımında gereken denetim sinyalleri

Komut-tipi	veriyolu işlemi	denetim sinyalleri
Bellek kullanım: {***lw***} **komut biter**	**(Yazmaca yazma)** Mem [ALUSonucu]→ Y.Dos({ $\underbrace{IR_{20}... IR_{16}}_{rt}$)2) (altında: $rt)	(***ALUSelA***, ***ALUSelB***, ***ALUOp,*** ***IorD*** **=1,** ***MemRead*** önceki gibidir) ***MemtoReg*** **= 1** ***RegDst*** **= 0**; ***RegWrite*** ,

5.4.4 PC çoklayıcısı ve PC-yaz-denetimi

Çok-saatli veriyolunun ince ayrıntaları sırasında program-sayacı-yazmacı PC -ye bağlanacak hatları seçmek için bir çoklayıcı kullanmak gerektiğini gördük. *j* komutundakini de sayarsak PC ye bağlanması gereken üç muhtemel kaynak vardır:

(0)- **ALU-sonucu**: sıradaki komutu adreslemek üzere PC+4 için.
(1)- **Hedef Yazmacı**: koşulu sağlanan dallanma komutu için.
(2)- ($nPC_{31}...nPC_{28}$, IR_{25} ... IR_0 , **0, 0** $)_2$: *j* komutu için IR -nin en-sağ-26-biti 2-bit-sola kaydırıp PC -nin en sol 4-bitiyle birleştirilerek oluşturulan atlanacak adres.

Bu üç durum 3-girişli bir çoklayıcıyı denetleyen 2-bitlik *PCSrc* denetim sinyali ile Tablo 5-28 deki gibi kodlanır.

Tablo 5-28 PCSrc çoklayıcısının denetim sinyalleri

PCSrc	Kaynak
0 0	ALU-sonucu
0 1	Hedef (Target) Yazmacı
1 0	IR

PC -ye yazma sinyali *PCLoad* üretilirken aşağıdaki koşullar dikkate alınmalıdır:

- *PC* yazmacı her komut için ilk dönüşte, *j* komutu için üçüncü dönüşte koşulsuz olarak güncellenir.
- *beq* komutunda, eğer ALU -nun *Zero* çıkışı 1 ise Hedef Yazmaçtaki değer PC -ye yazılır.
- Ayrıca ikinci dönüşte ALU-sonucunu Hedef-Yazmaca yüklemek için bir de *TargetWrite* sinyali gereklidir

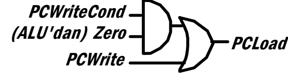
PC-yazmacının *PCLoad* yükle sinyali biri koşullu diğeri koşulsuz iki PC-yazma denetim sinyali gerektirir:

PCWrite PC -ye koşulsuz yazmak üzere; ve

PCWriteCond PC -ye sadece Zero=1 ise yazmak üzere.

Yazma koşulu şöyledir.

PCLoad = PCWriteCond . Zero **+** *PCWrite*



Denetim çıkış sinyalleri PC -yi düzgün biçimde gencellemek için aşağıdaki ilave denetim sinyallerini içermek üzere genişletmek gerekir.

Tablo 5-29 Dallanma adresleme için ek denetim sinyalleri

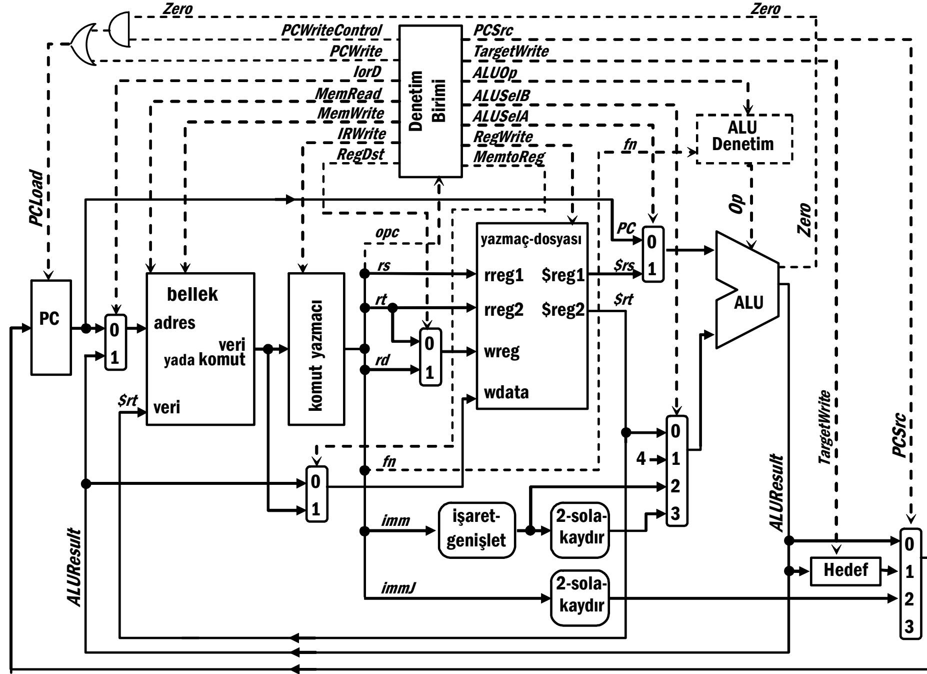
Isim	Açıklama	
	Değer	**Etki**
TargetWrite	**Hedef-yazmacına yazma** sinyali	
	0	-
	1	ALU sonucu hedef-yazmacına yüklenir
PCLoad	**Program-sayacı yazmacına yazma** sinyali	
	0	-
	1	*PCSrc* ile seçilen kaynak *PC* -ye yazılır
PCWriteCond	**Program-sayacına koşullu yazma** sinyali	
	0	-
	1	*PCWriteCond . Zero → PCLoad* ; (Zero ALU çıkışıdır)
PC-Write	**Program-sayacına koşulsuz yazma** sinyali	
	0	-
	1	*1 → PCLoad* (koşulsuz)
PCSrc	**Program Sayacına aktarılacak sinyalı seçer.**	
	00	ALU Sonucu→ PC
	01	Hedef → PC
	10	$((iPC+4)_{31} \ldots (iPC+4)_{28}\ IR_{25} \ldots IR_0\ 0\ 0\ )_2 \rightarrow PC$

5.5 Çok dönüşlü veriyolu ve denetim sinyalleri

Çok dönüşlü veriyolunun dallanma komutunun ayrıntıları ele alınıp düzeltildikten sonraki hali ve eklenen denetim sinyallerinin kullanımı Tablo 5-30 ve Şekil 5-52 de gösterilmiştir:

Tablo 5-30 Çok dönüşlü veriyolunda gereken denetim çıkışları.

<table>
<tr><th>Adım</th><th colspan="5">her komut için gereken denetim çıkışları</th></tr>
<tr><td>1</td><td colspan="5">Bütün komutlar için:
IorD=0; MemRead; IRWrite;
ALUSelA=0; ALUSelB=01; ALUOp=00; PCSrc= 00; PCWrite.</td></tr>
<tr><td>2</td><td colspan="5">Bütün komutlar için:
ALUSelA=0; ALUSelB=11; ALUOp=00; TargetWrite</td></tr>
<tr><td>3</td><td>R-tipi için
ALUSelA=1;
ALUSelB=00;
ALUOp=10;</td><td colspan="2">lw ve sw için
ALUSelA=1;
ALUSelB=10;
ALUOp=00;</td><td>beq için
ALUSelA=1;
ALUSelB=00;
ALUOp=01;
PCSrc=01;
PCWriteCond</td><td>j için
PCSrc=10;
PCWrite</td></tr>
<tr><td>4</td><td>R-tipi için
ALUSelA=1
ALUSelB=0
ALUOp=10
RegDst=1
MemtoReg=0
RegWrite;</td><td>lw için
ALUSelA=1;
ALUSelB=10;
ALUOp=00;
IorD=1;
MemRead;</td><td>sw için
ALUSelA=1;
ALUSelB=10;
ALUOp=00;
IorD=1;
MemWrite;</td><td></td><td></td></tr>
<tr><td>5</td><td></td><td>lw için
ALUSelA=1;
ALUSelB=10;
ALUOp=00;
MemRead
IorD=1;
RegDst=0;
MemtoReg=1
RegWrite;</td><td></td><td></td><td></td></tr>
</table>



Şekil 5-52 *R-tipi lw*, *sw*, *beq*, ve *j* için çok-dönüşlü veriyolu.

5.5.1 Çok-dönüşlü gerçeklemenin başarımı

Bu noktada, her adımın bir saat dönüşü gerektirdiğini ve aşağıdaki komut karışımını varsayarak çok-döngü gerçeklemesi için CPI -yi hesaplayabiliriz:

Komut-tipi	R-tipi.	lw	sw	branch	jump
test programı kompozisyonu:	49%	22%	11%	16%	2%
Her komut tipi için CPI	4	5	4	3	3

Ortalama CPI aşağıdaki ifade ile hesaplanır:

$$CPI = \frac{\sum (CPI_k \times \text{komut-sayısı}_k)}{\text{Komut-sayısı}}$$

$$CPI = (4 \times 0.49)+(5 \times 0.22)+(4 \times 0.11)+(3 \times 0.16)+(3 \times 0.02)$$
$$= 4.04$$
$$\text{CPU-çalışma-zamanı} = I \times 4.04 \times 10 \text{ ns} = 40.4 \times I \text{ ns}$$

Dikkat ederseniz elde edilen CPI en-kötü-durum olan tüm komutların en uzun komutla aynı saat dönüşü (=5) sürdüğünü varsaydığımız CPI -den daha iyidir.

Eğer çok-dönüşlü ve tek-dönüşlü gerçeklemeleri karşılaştırırsak, ikisinin başarım oranları şöyle olur:

$$n = \frac{\text{Çok-dönüşlü Başarım}}{\text{Tek-dönüşlü Başarım}} = \frac{40 \times I \text{ ns}}{40.4 \times I \text{ ns}} = 0.99$$

Çok-dönüşlü gerçekleme farkedilir basitlikte ve çok daha esnek veriyoluna rağmen sabit-tek-saat-dönüşü gerçeklemesiyle neredeyse aynı başarım oranındadır.

Başarımı artırmak üzere çok dönüşlü veriyolunda küçük değişikliklerle her komutun hemen hemen bir saatte çalıştırıldığı ardışık düzen yürütmeye geçilebilir. Tek saat dönüşlü veriyolunda başarımı artıracak bu çeşit esneklikler yoktur.

5.6 Çok-dönüşlü Veriyolu için Denetim Birimi

Komutlar birden fazla adımda tamamlandığından ötürü çok-dönüşlü gerçeklemenin denetim birimi daha karmaşıktır. Bu tür ardışımlı karmaşık denetim birimi tasarımı için iki farklı tasarım tekniği vardır:

- Sonlu-durum makineleri (FSM ve ASM) çizimleri kullanarak
- Mikroprogram yöntemleri ile .

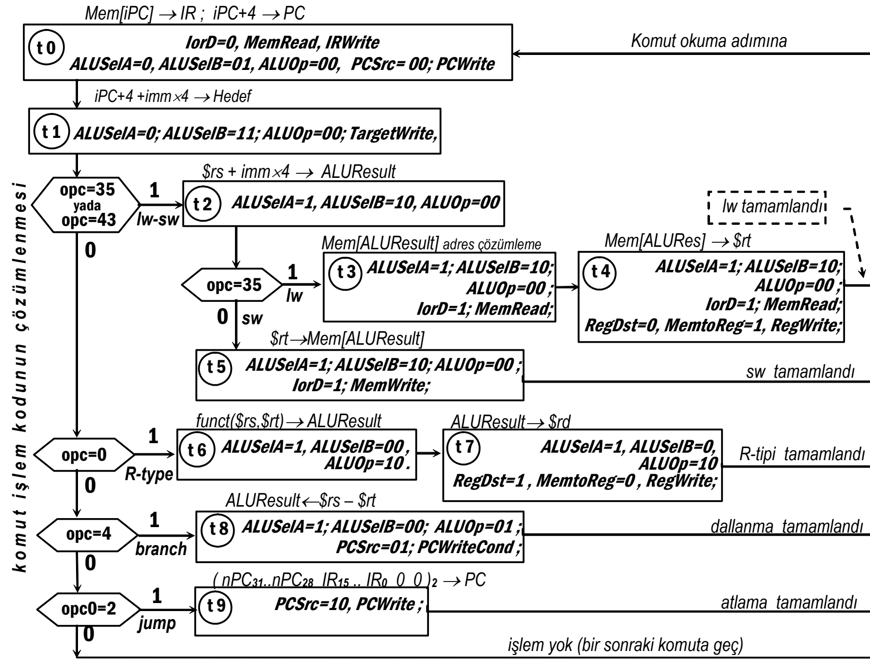
Bu tekniklerin ikisinde de, denetim devresini gerçeklemede bilgisayar-destekli tasarım (CAD) aletleri ve ROM, PLA, FPGA vs. kullanılmasına uygun olarak tanımlarız.

5.6.1 Sonlu-Durum Makinesi (FSM) Yaklaşımı

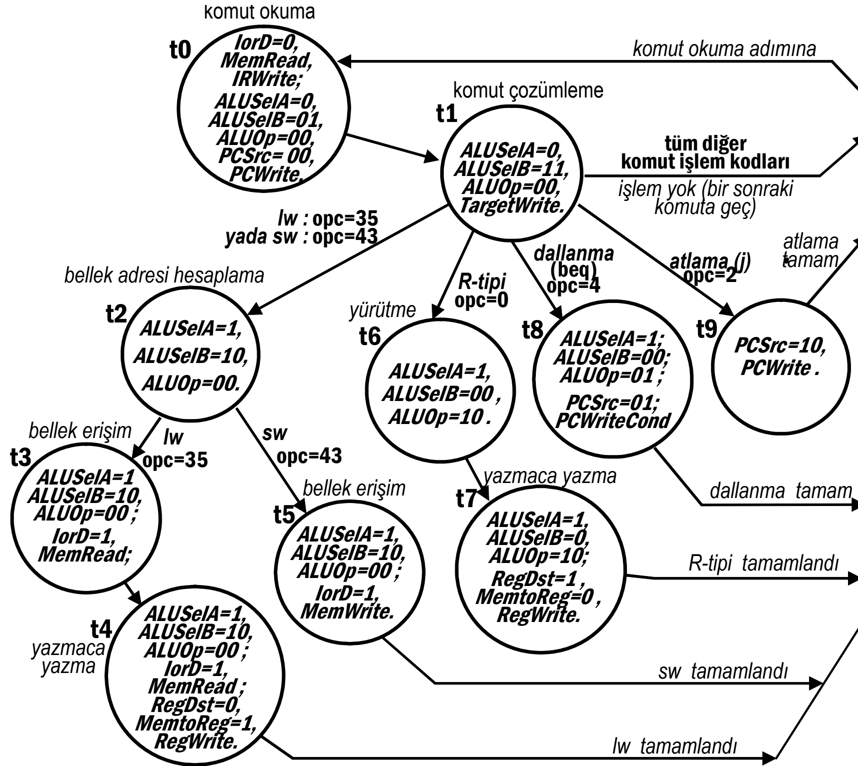
Denetim birimi gibi eşanlı ardışık durum makineleri Sonlu-Durum-Makinesi-çizimi (*Finite-State-Machine FSM* (FSM-çizimi) ile gösterilebilir. FSM-çizimi gerekli sayıda "durum" daireleri, ve *sonraki durum fonksiyonu* -nu betimlemek üzere durumları sonraki durumlara

birleştiren oklu çizgilerden oluşturulur. Sonraki-durum fonksiyonu o anki (*şimdiki*) durumu ve girişleri bir *sonraki* duruma ilişkilendirir . Her durum ayrıca FSM o durumda iken verilecek çıkışları betimler. Bu nedenle, FSM-çizimini betimlerken her durum için tüm denetim sinyallerinin çıkış değerini bilmemiz gerekir.

ASM çizimi Finite State Makinesi denetim birimini betimlemek için en uygun metotlardan biridir. Çizimin tümü pek çok ASM öbeğinden oluşur. Her öbek işlemin bir saat dönüşüne karşılık gelir, ve işlemin o adımında verilecek denetim çıkışlarını betimler. Bir öbeğe bir durum kutusu, birkaç karar kutusu, ve birkaç koşullu-çıkış kutusu olabilir. Denetim biriminin giriş sinyalleri karar kutularında belirir, ve onların değerlerine bağlı olarak denetim karar kutusunun karşılık gelen kolu üzerinde çalışır.



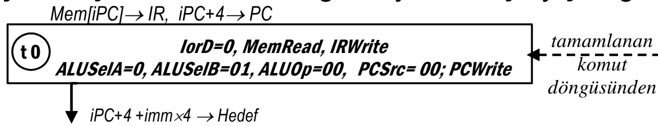
Şekil 5-53 *R-tipi*, *lw*, *sw*, *beq* ve *j* komutları için ASM çizimi

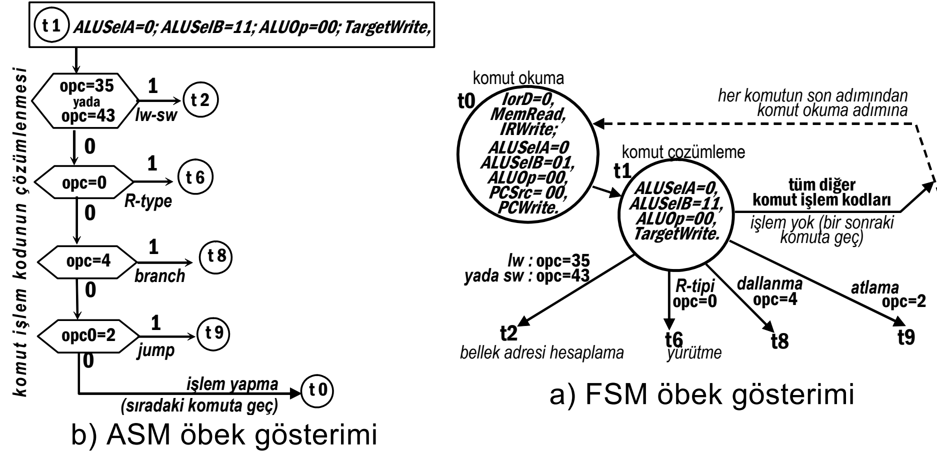


Şekil 5-54 Çok-adımlı veriyolu denetlecinin FSM-çizimi

Denetim biriminin FSM -deki her adımı bir saat dönüşü tutar. Yürütmenin ilk iki durumunda komut-kodu henüz daha çözümlenmediğinden bu iki adımda tüm komutlar için aynı işlemler uygulanır. 3. 4. ve 5. dönüşteki durumlar okunmuş olan komutun işlem koduna göre farklılık gösterir. FSM her komutun son durumu çalıştırıldıktan sonra, sıradaki komutu okumak için *başlama durumu (t0)* -a döner.

Şimdi, çizimdeki her bloğun ayrıntıla işleyişini göreceğiz:





b) ASM öbek gösterimi a) FSM öbek gösterimi

Şekil 5-55 Komut okuma, ve çözümleme adımları (t0, t1)

Komut Okuma Adımı (t0)
Komut, program sayacının adreslediği bellektedir. Şekil 5-55 te görüldüğü gibi denetim birimi

- *IorD* = 0 vererek PC -yi bellek-adres girişlerine bağlar ve
- *MemRead* ve *IRWrite* vererek komutu bellekten alıp komut-yazmacına yazar,
- *ALUSelA*=0 vererek PC -yi ALU-A-girişine gönderir,
- *ALUSelB*=01 vererek ALU-B-girişine *4* gönderir,
- *ALUOp* =00 vererek ALU -da toplama yapıp PC -ye *4* ekler,
- *PCSrc* = 0 vererek *PC +4* olan ALU-sonucunu *P*C -ye gönderir,
- *PCWrite* vererek *PC*+4 -ü *PC* yazmacına yazar.

Bu eylemlerin hepsi aynı sırada aynı saat dönüşünde gerçekleşir. Saat dönüşü tamamlandığında IR -da yürütülen komut, PC de bir sonraki komutun adresi olan nPC vardır. PC yazmacındaki ilk değeri iPC, toplamadan sonraki değeri ise nPC ile gösteririz. Böylece artırmadan sonda nPC = iPC + 4 olur.

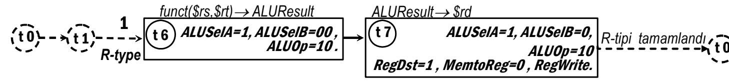
Komut Çözümleme ve Hedef Adres Dönüşü (t1)
Komut kodu komut-yazmacındadır, ve denetim sonraki durumu belirlemek üzere komutun işlem kodu (opc) alanını çözümler. ALU işlenmesi mğmkğn bir dallanma komutunun muhtemel-hedef-adresini önceden hesaplamaya ayrılmıştır. Denetim

- *ALUSelA*=0, *ALUSelB*=11 ve *ALUOp*=00 vererek ALU -da (iPC+4) + (imm×4) toplamasını yapar.
- *TargetWrite* vererek ALU çıkışındaki (iPC+4)+(imm×4) değerini Hedef-Yazmaca yazar.

Saat dönüşü tamamlandığında, hedef-yazmaç muhtemel bir dallanma komutunun hedef-adres-adayını içerir. t0 ve t1 bütün komutlar için işlenmesine karşılık t1 -de çözümlenen komut işlem koduna bağlı olarak bir sonra işlenecek durum her komut kodu farklı olur. Sonraki

durumun işlenen komuta göre farklılaştığı durumlara "*dağıtım*" (dispatch) durumu denir.

R-tipi komutlar: t0, t1, t6, t7.



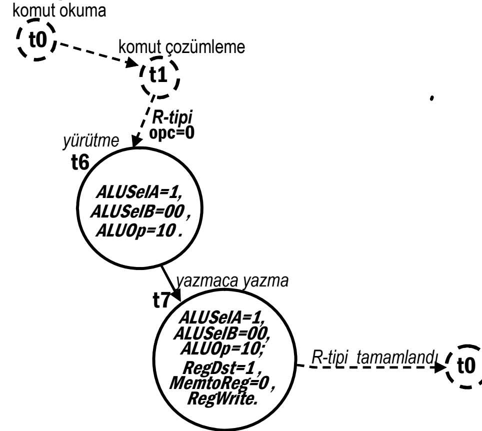
a) ASM-çizimi

R-tipi komutun (üçüncü adımında t6, ve ardından t7 durumu işlenir.

t6 durumunda denetim

- *ALUSelA=1, ALUSelB*=00 ve *ALUOp*=10 verip ALU da $rs ve $rt yi komut yazmacının fn-alanında betimli fonksiyon ile işler. Ancak işlemin bitmesi bir saat-dönüşü zaman alır.

Saat-dönüşü bitince, denetim t7 -ye geçer, ve ALU-sonucu varış yaz-macına yazılmaya hazır bekler.



b) FSM-çizimi

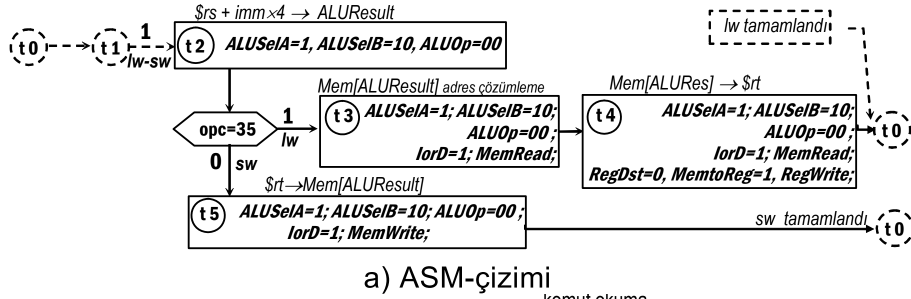
Şekil 5-56 R-tipi komutun yürütme ve yazmaca yazma adımlarının

t7 -de, denetim

- *ALUSelA*=1, *ALUSelB*=00 ve *ALUOp*=10 vermeye devam eder. Böylece ALU sonucu t7 adımı boyunca kullanılabilir kalır,
- MemtoReg=0 ve *RegDst=1* vererek *rd*-alanını yazmaç-dosyasının yazılacak-yazmaç (*WReg*) girişine, ALU-sonucunu ise yazmaç-dosyasının yazılacak veri (*WData*) girişine gönderir.
- *RegWrite* vererek *ALU sonucu*nu *rd* -ye yazdırır.

Saat dönüşü tamamlandığında, *$rd* ye *fn*-alanında betimlenen işlemin *$rs* ve *$rt* ye uygulanmasıyla bulunan sonuç yazılmış, ve komut tamamlanmıştır. Sonraki saat dönüşünde denetim sıradaki komutu okumak üzere t0 adımını işler.

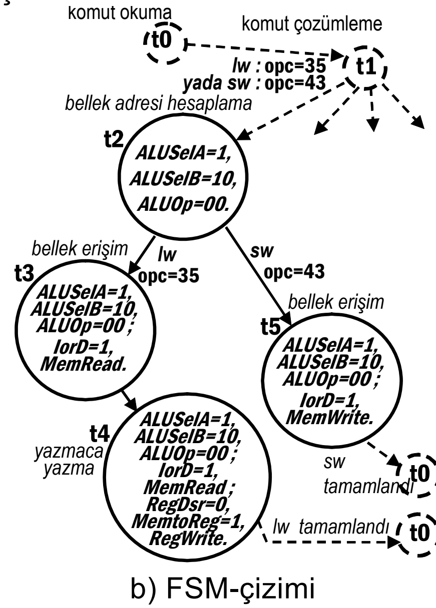
Bellek işlem komutları sw ve lw (t0, t1, t2, t3, t4, t5 adımları)



a) ASM-çizimi

Denetim birimi t1 durumunda komut çözümlerken *lw* ve *sw* işlem kodları bulursa sonraki dönüşte t2 durumuna geçer. Denetim t2 -de veriyoluna bellek-teki veri adresini hesaplaması için gereken çıkışları yollar. Denetim t2 -de

- *ALUSelA* =1, *ALUSelB* =10 *ALUOp*=00 vererek *$rs* ile işaret-genişletilmiş-16-bit-anlık-alanı (imm) ALU -ya gönderip orada toplayarak bellek adresini hesaplatır.
- t2 bir dağıtım bloğudur. eğer opc=35 ise sonraki adımda t3 -ü, değilse t5 -i işlemeye karar verir.

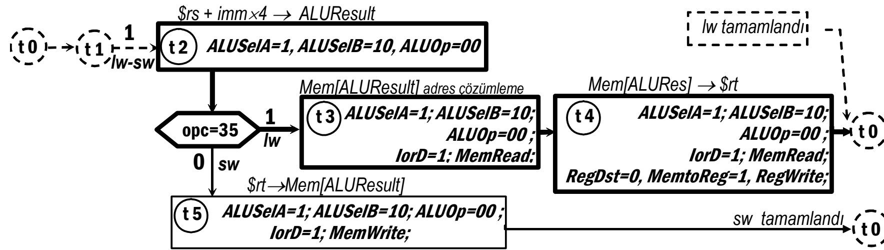


b) FSM-çizimi

Şekil 5-57 Bellek işlem komutları sw ve lw (t2, t3, t4, t5 adımları)

Saat dönüşü sonunda, ALU çıkışında ya okumak yada yazmak için erişilecek bellek adresi bulunur.

lw komutu: t0, t1,t2, t3, t4.



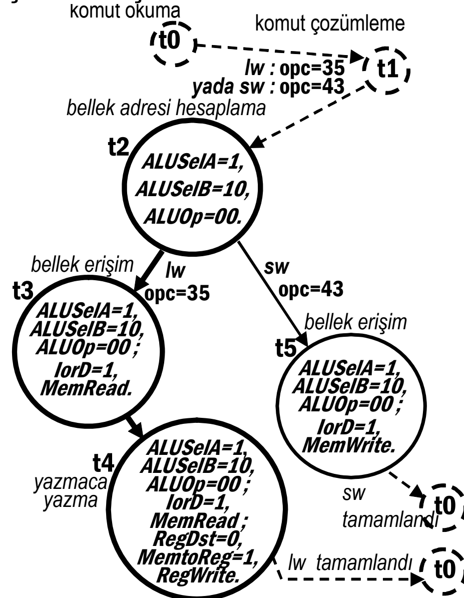
a) lw komutu için ASM-çizimi

lw komutu için t2 durumunun ardından t3 ve t4 -ü başlar. t3 -de, denetim

- *ALUSelA* =1, *ALUSelB* =10, *ALUOp* =00 vermeye devam ederek ALU çıkışında ileriki işlemler için bellek adresini vermesini sağlar,
- *MemRead* vererek bellek okuma işlemini başlatır.

Saat dönüşünün bitimiyle birlikte, adreslenen yerdeki veri bellek biriminin çıkışında kullanıma hazır olur.

t4 yazmaca-yazma adımıdır. Burada denetim



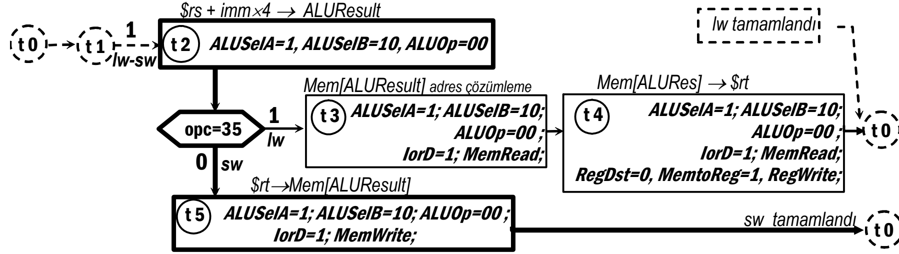
b) lw komutu için FSM-çizimi

Şekil 5-58 *lw* komutu (t2, t3, t4 adımları)

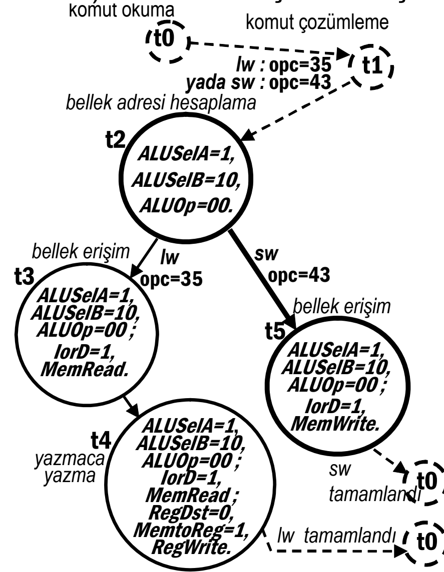
- *ALUSelA* =1, *ALUSelB* =10, *ALUOp* =00 ve *MemRead* vermeyi sürdürerek bellek çıkışını *$rt* yazmacına yazmak üzere hazır tutar,
- *RegDst* =0 vererek komutun *rt*-alanını yazmaç-dosyasının yazılacak-yazmaç (*WReg*) girişine gönderir,
- *MemtoReg* =1 ve *RegWrite* vererek bellek çıkışını yazmaç-dosyasının yazılacak-veri (*WData*) girişine yönlendirip adreslenen bellekteki veriyi *rt* adresli yazmaca yazdırır.

t4 adımı tamamlandığında, belleğin *$rs+imm* adresinde bulunan veri *rt* yazmacına yazılmış olur. Ardından denetim birimi sıradaki komutun çalıştırılması için t0 dönüşüne başlar. 5 saat dönüşü gerektiren *lw* komut-altkümemizdeki en uzun komutudur.

sw-komutu: t0, t1, t2, t5.



a) sw komutu için ASM-çizimi



b) sw komutu için FSM-çizimi

Şekil 5-59 *sw* komutu (t2, ve t5 adımları)

sw komutu t0, t1 ve t2 adımlarından sonra t5 adımına geçer.

t5 sırasında denetim

- *ALUSelA* =1, *ALUSelB* =10, ve *ALUOp* =00 vermeye devam ederek, bellek adresinin ALU çıkışında kararlı kalmasını sağlar,
- *MemWrite* vererek *$rt* yi belleğe yazma işlemini tamamlar.

Saat dönüşünün sonunda, *$rt* belleğin *$rs+imm* adresine yazılmış olur. sw komutunun çalışması t5 -de tamamlanır. Sıradaki durum t0 komut okuma (*instruction-fetch*) durumudur.

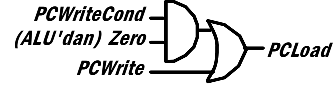
Dallanma (beq) komutu: t0, t1, t8

Komut okuma (t0) ve çözümleme (t1) durumlarından sonra, dallanma komutu t8-durumunu başlatır. t1 bloğunda denetim ALU -da muhtemel dallanma hedef adresini (iPC+4+imm×4) hesaplar, ve sonucu hedef-yazmacına saklar. t8 durumu ALU -da $rs – $rt -nin hesaplandığı ve *PCWriteCond* sinyalinin verildiği adımdır. t8 de denetim

- *ALUSelA* =1 ve *ALUOp* =01, ve *ALUSelB* =00 vererek *$rs* ve *$rt* -yi ALU-girişlerine gönderip *$rs–$rt* çıkarmasını yaptırır. Eğer sonuç sıfır ise ALU -nun *Zero* çıkışı 1 olur.
- *PCSrc* =01 ve *PCWriteCond* vererek hedef-yazmaçtaki adresin koşula bağlı olarak program sayacına yazılabilmesini sağlar. Koşul gerçekleşmezse *PC* deki adres (*nPC* = *ıPC+4*) değişmemiş olur.

PCLoad sinyali veriyolunda *PCWrite*, *PCWriteCond* ve *Zero* sinyalleri işlenerek oluşturulur:

PCLoad* = *PCWrite* + *PCWriteCond Zero



Eğer *$rs–$rt* = 0 ise ve *PCWriteCond* verilmişse, *PCLoad* etkin (yüksek) olur, ve hedef-yazmaçtaki dallanma hedef adresini PC yazmacına yükler. Yoksa *PC* değişmez, ve *PC* -de t0-dönüşünde yazılmış olan *nPC*= *iPC+4* kalır.

Atlama (j) komutu: t0, t1, t9.

Komut okuma (t0) ve çözümleme (t1) adımlarından sonra, *j* atlama komutu t9 durumuna geçer. t9 -da denetim *PCSrc* =11 ve *PCWrite* çıkışlarını vererek 26-bitlik anlık-atlama-adresi (immJ) alanını PC -ye yazar. Dikkat ederseniz bu alan önce 2-bit sola kaydırılarak (×4) bayt adresine dönüştürülür. Ayrıca işlem kodu nedeniyle eksik kalan en sol 4-bit *PC* -nin en sol dört bitinden (= $nPC_{31}...nPC_{28}$) tamamlanır.

Denetim Biriminin Gerçeklenmesi

FSM denetim birimi (genellikle bir durum yazmacı ve AND-OR devreleri, PLA -ler yada bellek-birimi gibi devrelerden oluşan bir bileşimsel-denetim-mantığı devresiyle gerçeklenir.

Şekil 5-60 FSM denetim biriminin ROM ile tipik gerçeklenmesi

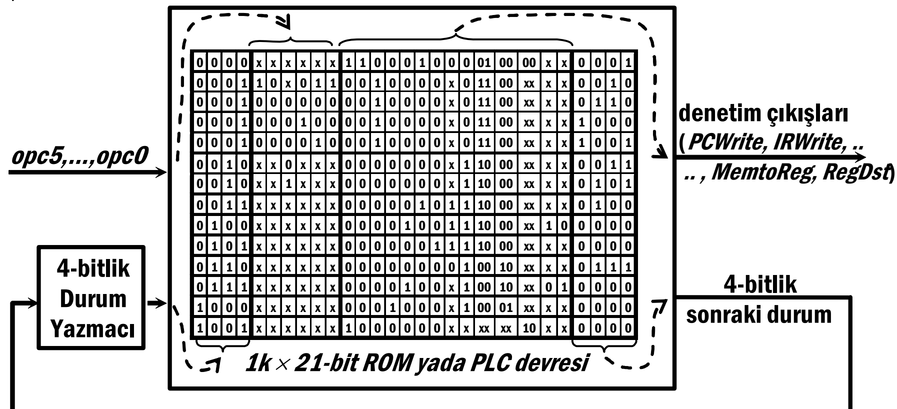
Yüzlerce komutu olan gelişmiş bir bilgisayarın Bileşimsel Denetim Mantığı genellikle devreyi durum geçiş çizimlerinden başlayarak en ince ayrıntısına kadar çözümleyebilen Bilgisayar-Destekli-Tasarım gereçleri kullanılarak tasarlanır. Denetimin gerçeklemesinde ROM birimlerinin kullanılması denetim tasarımı tekniklerinde mikroprogramlanmış denetim denilen yeni bir pencere açar.

Örnek 5-2:

FSM ve ASM çizimlerinde her duruma karşılık denetim çıkışları betimlenmiş olan denetim birimi en basit biçimiyle bir durum yazmacı ve aşağıda verilen doğruluk tablosunu sağlayan bir PLA yada ROM kullanılarak gerçekleştirilir.

Tablo 5-31 Denetim biriminin çıkış ve sonraki durum doğruluk tablosu

Durum adı	Girişler: Durum Kodu bitleri 3	2	1	0	Komut Kodu bitleri 5	4	3	2	1	0	Çıkışlar / Veriyolu Denetim Çıkışları: PCWrite	IRWrite	TargetWrite	PCWriteCond	RegWrite	MemRead	MemWrite	IorD	ALUSelA	ALUSelB	ALUOp	PCSrc	MemtoReg	RegDst	Sonraki Durum Kodu bitleri 3	2	1	0	Sonraki Durum Adı
t0	0	0	0	0	x	x	x	x	x	x	1	1	0	0	0	1	0	0	0	01	00	00	x	x	0	0	0	1	t1
t1	0	0	0	1	1	0	x	0	1	1	0	0	1	0	0	0	0	x	0	11	00	xx	x	x	0	0	1	0	t2
t1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	x	0	11	00	xx	x	x	0	1	1	0	t6
t1	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	x	0	11	00	xx	x	x	1	0	0	0	t8
t1	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	x	0	11	00	xx	x	x	1	0	0	1	t9
t2	0	0	1	0	x	x	0	x	x	x	0	0	0	0	0	0	0	x	1	10	00	xx	x	x	0	0	1	1	t3
t2	0	0	1	0	x	x	1	x	x	x	0	0	0	0	0	0	0	x	1	10	00	xx	x	x	0	1	0	1	t5
t3	0	0	1	1	x	x	x	x	x	x	0	0	0	0	0	1	0	1	1	10	00	xx	x	x	0	1	0	0	t4
t4	0	1	0	0	x	x	x	x	x	x	0	0	0	0	1	0	0	1	1	10	00	xx	1	0	0	0	0	0	t0
t5	0	1	0	1	x	x	x	x	x	x	0	0	0	0	0	0	1	1	1	10	00	xx	x	x	0	0	0	0	t0
t6	0	1	1	0	x	x	x	x	x	x	0	0	0	0	0	0	0	0	1	00	10	xx	x	x	0	1	1	1	t7
t7	0	1	1	1	x	x	x	x	x	x	0	0	0	0	1	0	0	x	1	00	10	xx	0	1	0	0	0	0	t0
t8	1	0	0	0	x	x	x	x	x	x	0	0	0	1	0	0	0	x	1	00	01	xx	x	x	0	0	0	0	t0
t9	1	0	0	1	x	x	x	x	x	x	1	0	0	0	0	0	0	x	x	xx	xx	10	x	x	0	0	0	0	t0



Şekil 5-61 Denetim biriminin ROM ya da PLC ile en basit gerçeklemesi.

ROM -la gerçeklenen bu doğruluk tablosu için 1024 × 21 bitlik bir bellek kullandık. Belleklerde adres bitleri arttıkça erişim süresi de artacağından aynı tasarımı daha küçük bellekle yapmanın yollarını arayacağız.

5.6.2 Sıralayıcı

Örnekte 1024×21 bitlik bir ROM ile gerçeklenen doğruluk tablosunu incelediğimizde denetim çıkışlarının her zaman komut işlem kodu girişlerinden bağımsız olduğunu, yalnızca işlenen duruma bağlı olduğunu gözleriz. Bellek adres bitlerini azaltmanın bir yolu sonraki adres için gereken bilgiyi mümkün olduğunca sıkıştırarak kodlamak ve bu kodu ROM dışında bir devrede çözmektir. Tablodaki sonraki durum çıkışları belirleyen yalnızca dört değişik durum vardır.

- Komut bitimine rastlayan durumlar için sonraki durum adresi t0 olur.
- t1 durumunda sonraki durum opc komut işlem koduna bağlıdır. Bu durumda sonraki durumu bulmak için birinci dağıtım tablosunu kullanırız.
- t2 durumunda sonraki durum opc -nin yalnızca dörduncü bitine (opc3 -e) bağlıdır. Burada da bir dağıtım tablosu kullanmak gerekir.
- Komut işlem kodunun sınanmadığı ve komut sonu olmayan durum-larda sonraki durum adresi o anda işleyen durumun adresine (durum koduna) bir ekleyerek bulunabilir.

Dağıtım tablolarını yalnızca 6-bit-adres-girişli hızlı bir ROM -a koyabiliriz. Dört değişik durum iki bitlik sıralama kodu ile kodlanabilır. 00: t0 -a gitmeyi; 01: t1 dağıtımını; 10: t2 dağıtımını; 11: durumun bir arttılması için kullanılabilir. Böylece sonraki durum her koşul için ayrı bir devrede hesaplanıp şekildeki gibi sıra koduna göre dört girişli bir çoklayıcı ile seçilir.

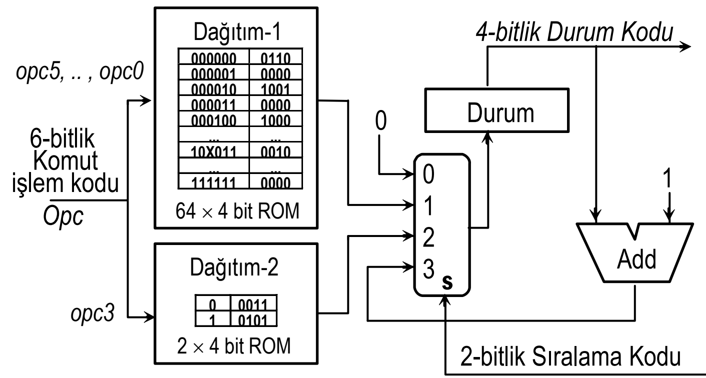
t1 ve t2 durumlarındaki bir-sonra-çalışacak durumun kodunu veren *Dağıtım-1* ve *Dağıtım-2* tabloları Tablo 5-32 ile Tablo 5-33 te verilmiştir.

Tablo 5-32 1.nci dağıtım tablosu (t1 durumu için)

Dağıtım-1 (t1 için)				
Giriş			**Çıkış**	
komut adı	**Opc**	**ikili sayı**	**adım adı**	**kodu**
R-tipi	0	000000	t 6	0110
J	2	000010	t 9	1001
beq	4	000100	t 8	1000
lw	35	100011	t 2	0010
sw	43	101011	t 2	0010
diğer hepsi	...	...	t 0	0000

Tablo 5-33 2.nci dağıtım tablosu (t2 durumu için)

Dağıtım-2 (t2 için)				
Giriş			**Çıkış**	
komut adı	Opc	Opc3	adım adı	kodu
lw	35	0	t 3	0011
sw	43	1	t 5	0101

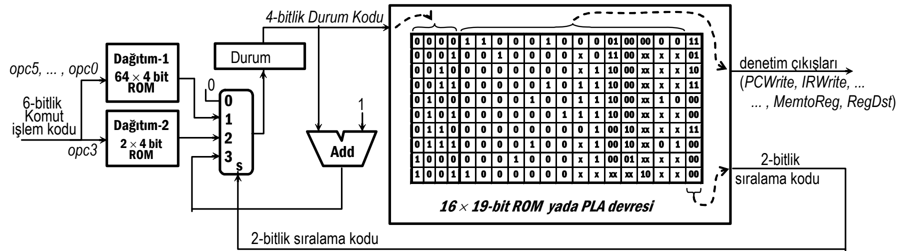


Şekil 5-62 Denetim biriminin sıralayıcı devresi

Denetim biriminin bu sıralayıcı için düzenlenmiş doğruluk tablosu Tablo 5-34 de verilmiştir. Sıralayıcı kullanıldığında birincisi 6-bit ikincisi tek-bit ile adreslenen iki ROM gerekmektedir. Adres bit sayısı az olduğundan bu ROM -larda daha az gecikme olması beklenir. Denetim devresinin tümü Şekil 5-63 de görülmektedir.

Tablo 5-34 Sıralamalı denetim biriminin çıkış ve sıralama kodu doğruluk tablosu

	Girişler				Çıkışlar															
	Durum Kodu bitleri				Veriyolu Denetim Çıkışları														Sıralama	
Durum adı	3	2	1	0	PCWrite	IRWrite	TargetWrite	PCWriteCond	RegWrite	MemRead	MemWrite	IorD	ALUSelA	ALUSelB	ALUOp	PCSrc	MemtoReg	RegDst	Sıralama kodu	sıralama eylemi
t0	0	0	0	0	1	1	0	0	0	1	0	0	0	01	00	00	0	0	11	sıradaki
t1	0	0	0	1	0	0	1	0	0	0	0	x	0	11	00	xx	x	x	01	dağıtım-1
t2	0	0	1	0	0	0	0	0	0	0	0	x	1	10	00	xx	x	x	10	dağıtım-2
t3	0	0	1	1	0	0	0	0	0	1	0	1	1	10	00	xx	x	x	11	sıradaki
t4	0	1	0	0	0	0	0	0	1	0	0	1	1	10	00	xx	1	0	00	okumaya
t5	0	1	0	1	0	0	0	0	0	0	1	1	1	10	00	xx	x	x	00	okumaya
t6	0	1	1	0	0	0	0	0	0	0	0	0	1	00	10	xx	x	x	11	sıradaki
t7	0	1	1	1	0	0	0	0	1	0	0	x	1	00	10	xx	0	1	00	okumaya
t8	1	0	0	0	0	0	0	1	0	0	0	x	1	00	01	xx	x	x	00	okumaya
t9	1	0	0	1	1	0	0	0	0	0	0	x	x	xx	xx	10	x	x	00	okumaya



Şekil 5-63 Sıralamalı denetim birimi şemasının tamamı

Devrede ve tabloda görüldüğü üzere on durumlu denetim için denetim çıkışlarını sağlayan ROM artık sadece 4 bit ile adreslenebilmektedir.

5.6.3 Mikroprogramlı Denetim Yaklaşımı

(Eğitim amaçlı küçük FSM -imizde sadece 10 durum var, ve denetim birimi kolayca elde tasarlanabilir. MIPSin komut setindeki tüm komutlar 100 -den fazladır ve komutlardan bazıları 20 adıma varır. Böyle bir FSM -i elle tasarlamak ve muhtemel değişiklikler için nasıl çalıştığını anlamak son derece zordur.

Mikroprogramlama tekniği denetim işlemlerini anımsatıcılarla kodlayarak "*yapısal*" yönden kolayca anlaşılacak ve gerektiğinde değiştirilebilecek biçimde *sınıflandır*.

- Bir adımda verilen denetim çıkışları işlevine göre kümelenerek *veriyolu* tarafından çalıştırılan *alt düzey komutlar* gibi düşünülür. Her çıkış kümesine uygun bir *komut anımsatıcı* isim atanır.
- Donanım-seviyesindeki bu komutlara *mikrokomut* denilir.
- Her mikrokomut belli adımlarda verilen bir takım veriyolu denetim sinyallerini betimler. Bir mikrokomutu çalıştırmak demek o saat dönüşünde o mikrokomutun denetim çıkışlarını vermek demektir.
- Aksi belirtilmedikçe bir-sonra-çalışacak mikrokomutun sıradaki ilk mikrokomut olduğu varsayılır, ve adresi bir sayacı arttırarak bulunur. Komut-işlem-koduna bağlı dallanmalar gerektiren mikrokomutlarda bu sıralama bozulur ve adres ilgili dağıtım ROM -undan okunur.
- Denetim biriminin mikrokomutlardan oluşan bir program ile tasarlanmasına *mikroprogramlama* denir.
- Mikroprogram ya elde, ya da *mikroprogram-derleyicisi* aracılığıyla dağıtım-ROM -ları ve çıkış-ROM -larına yazılacak sayılara dönüştürülerek *sıralamalı-denetim-birimi-devresi* oluşturulur.

Mikrokomut biçimi oluşturma

Mikrokomut biçimini oluşturuken bir mikrokomutun sahip olması gereken alan sayısını ve her alana karşılık gelen denetim sinyallerini aşağıdaki etkenler belirler:

1) gösterimi basitleştirmek

Örnek: ALU nun işlevini göstermek için *Add*, *Subt*, ve *Func* anımsatıcılarını kullanabiliriz.

2) mikroprogramın yazılışını ve okunuşunu kolaylaştırmak.

Örnek: mikroprogram formatımızda şu tipik alanlar olabilir
i) ALU -yu denetleyen bir alan,
ii) ALU nun iki kaynağını belirleyen iki alan, ve
iii) ALU sonucunun gideceği yeri gösteren bir alan.

3) çelişkili mikrokomutlar yazmayı önlemek.

Örnek: Bir saat-dönüşünde üç yazma sinyalinden (*RegWrite*, *MemWrite* ve *IRWrite*) sadece biri verilmelidir. Bu üç sinyali verdiren anımsatıcı tek ve aynı mikrokomut alanını paylaşırsa bir mikrokomuta bunlardan yalnızca birini yazabiliriz. Böylece bu üç sinyalden bir anda yalnız biri verilebilir.

MIPS komut altkümemizin çok-saat-dönüşlü gerçeklemesi için mikrokomut formatını seçerken, hiçbir zaman aynı anda verilmeyen sinyallerin aynı alana yazılabileceğini varsayabiliriz.

Tablo 5-35 Mikrokomut alan adları ve işlevleri

Alan	Betimlediği işlev veya bağlantı
Etiket	Mikrokomut adreslerini isimlendirmek içindir.
ALU Denetim	ALU -da yapılacak işlem.
SRC1	Birinci ALU işleneninin kaynağı (ALU-A).
SRC2	İkinci ALU işleneninin kaynağı (ALU-B).
ALU Hedef	ALU sonucunun gideceği yer.
Bellek	bellekten okuma ve yazma işlemi ve kaynağı
Bellek Yazmacı	Bellekten okuma ve yazma işleminin hedefi
PCWrite Denetim	*PC* güncelleme biçimi.
Sıralama	bir sonra çalışacak mikrokomutu belirler.

Mikrokomutlar genellikle bir FPGA, ROM yada PLA ile gerçeklenir. Her mikrokomutun bellekteki yerini belirleyen bir adresi vardır. Bir sonra çalışacak mikrokomut bir ilerideki adreste saklanır. Böylece mikroprogram satırlarının biçimini tanımlamak üzere Tablo 5-36 daki gibi bir adres etiketi ile 8 alan kullanılabilir:

Tablo 5-36 Tipik mikrokomut alanları ve bu alanlarda kullanılacak anımsatıcılar

Etiket:	ALU Denetim	SRC1	SRC2	ALU Hedef	Bellek	Bellek Yazmaç	PCWrite Denetim	Sıralama
Kullanılan anımsatıcılar	Add , – Func Subt	rs , – PC	rt , – 4 Extend ExtShft	– rd Target	– ReadPC ReadALU WriteALU	– IR WriteRT ReadRT	– ALU Target Jump	Oku Dağıtım-1 Dağıtım-2 Sıra

Sıralama alanının varsayılan değeri "*Sıra*"dır, ve sıradaki adımda çalıştırılacak mikrokomut mikroprogram belleğinde sıradaki adrestedir anlamına gelir. Bir komutun tüm adımları bitince, bir-sonra-çalışacak durum *yeni-komut-okuma* durumudur. Bu duruma *Sıralama* alanında "*Oku*" ile atlanır. Dağıtım noktalarındaki çalıştırılacak sıradaki mikrokomutun adresini veren dağıtım tabloları aynı Tablo 5-32 ile Tablo 5-33 teki gibidir. Ancak mikroprogramlama dağıtım tabloları hazırlanırken hataları azaltmak ve takipte kolaylık sağlamak üzere aşağıdaki gibi etiketler ve komut kodları için onluk sayı kullanılır.

Tablo 5-37 Tipik mikroprogramlama amaçlı sembolik dağıtım-1 ve dağıtım-2 tabloları

Mikroprogramlama amaçlı Dağıtım-1 Tablosu						
Giriş: Opcode (Komut İşlem Kodu)	0	2	4	35	43	diğer
Çıkış: Mikrokomut Etiketi	RF1	jump1	beq1	lwsw1	lwsw1	oku

Mikroprogramlama amaçlı Dağıtım-2 Tablosu		
Giriş: Opcode (Komut İşlem Kodu)	35	43
Çıkış: Mikrokomut Etiketi	lw2	sw2

Mikroprogram kodlama

Çok-dönüşlü gerçeklememiz için tipik bir mikroprogram kodunun tamamı Tablo 5-38 da verilmiştir. Program alanlarına doldurulan anımsatıcılar o saat dönüşündeki işlemlerin tümünü yaptıracak denetim çıkışlarına karşılık gelir.

Tablo 5-38 *R-tipi*, *lw*, *sw*, *beq*, ve *j* için mikroprogram kodu

Etiket:	ALU Denetim	SRC1	SRC2	ALU Hedef.	Memory	Mem. Reg.	PCWrite Control	Sıralama
t0 oku	Add	PC	4	–	ReadPC	IR	ALU	Sıra
t1 çözüml	Add	PC	ExShift	Target	–	–	–	Dağıtım-1
t2 lwsw1	Add	rs	Extend	–	–	–	–	Dağıtım-2
t3 lw1	Add	rs	Extend	–	ReadALU	–	–	Sıra
t4 lw2	Add	rs	Extend	–	ReadALU	WriteRT	–	Oku
t5 sw2	Add	rs	Extend	–	WriteALU	ReadRT	–	Oku
t6 RF1	Func	rs	rt	–	–	–	–	Sıra
t7 RF2	Func	rs	rt	rd	–	–	–	Oku
t8 beq1	Subt	rs	rt	–	–	–	Target	Oku
t9 jump1	–	–	–	–	–	–	Jump	Oku

Mikroprogramlar ya elde, ya da bir derleyici program aracılığıyla Şekil 5-63 daki sıralamalı denetim biriminin ROM larını hazırlamakta kullanılır. Derleyici aynı zamanda çıkışlardaki muhtemel çelişkileri de uyarır. Tablo 5-39 bu mikrokomutların alanlarında kullanılan anımsatıcılarla bu anımsatıcıların her birinin işlevini ve verdiği çıkışları betimlenmektedir.

Tablo 5-39 Mikrokomut alanlarında kullanılan anımsatıcılar.

Alan Adı	Anımsatıcılar	İşlevi ve *belirlediği denetim çıkışları*	
ALU Denetim			*Belirlediği çıkışlar* ***ALUOp***
	Add veya –	ALU-A + ALU-B → ALU-sonucu	*00*
	Subt	ALU-A – ALU-B → ALU-sonucu	*01*
	Func	fn(ALU-A , ALU-B) → ALU-sonucu	*10*
SRC1			*Belirlediği çıkışlar* ***ALUSelA***
	PC veya –	PC → ALU-A-girişi	*0*
	r s	$rs → ALU-A-girişi	*1*
SRC2			*Belirlediği çıkışlar* ***ALUSelB***
	rt veya –	$rt → ALU-B-girişi	*00*
	4	4 → ALU-B-girişi	*01*
	Extend	imm → ALU-B-girişi	*10*
	ExtShft	imm×4 → ALU-B-girişi	*11*
ALU hedef			*Belirlediği çıkışlar* ***TargetWrıte, RegDst, MemtoReg , RegWrite1****
	–	ALU hedefi yok	*0, 0, 0, 0*
	Target	ALU-sonucu → Hedef-Yazmacı	*1, 0, 0, 0*
	rd	ALU-sonucu → $rd	*0, 1, 0, 1*
	rt	ALU-sonucu → $rt	*0, 0, 0, 1*
Bellek			*Belirlediği çıkışlar* ***lorD, MemRead***
	–	Bellek işlemi yok	*0, 0*
	ReadPC	MEM[PC] → IR	*0, 1*
	ReadALU	MEM[$rs+imm] → $rt	*1, 1*
	WriteALU	$rt → MEM[$rs+imm]	*1, 0*
Bellek Yazmaç			*Belirlediği çıkışlar* ***IRWrite, MemWrite, RegWrıte2****
	–	YazmaçDosyası[rs] → $rs ; YazmaçDosyası[rt] → $rt ;	*0, 0, 0*
	ReadRT	$rt → MEM[$rs+imm]	*0, 1, 0*
	IR	MEM[PC] → IR	*1, 0, 0*
	WriteRT	MEM[$rs+imm] → $rt	*0, 0, 1*
PCWrite Denetim			*Belirlediği çıkışlar* ***PCSrc, PCWrite, PCWriteCond***
	–	PC değişmez	*00,0,0*
	ALU	ALUResult → PC	*00,1,0*
	Target-cont	Eğer($rs–$rt=0) ise target → PC	*01,0,1*
	jump Addr	immJ×4 → PC	*10,1,0*
Sıralama			*Belirlediği çıkışlar* ***sıralama-kodu***
	Oku	t0 durumuna geç	*00*
	Dağıtım-1	ilk dağıtım tablosunu kullan	*01*
	Dağıtım-2	ikinci dağıtım tablosunu kullan	*10*
	Sıra	sıradaki komuta geç	*11*

***** *Açıklama:* ***RegWrite*** *çıkışı* **(** ***RegWrite1*** *VEYA* ***RegWrite2*** **)** *olarak bulunur*

Şimdi okuma ve çözümleme dönüşlerinden başlayarak her bir komut için mikroprogramı anlatalım:

Komut okuma ve işlem kodu çözumleme

Denetim birimi komut işlem koduna henüz erişemediğinden okuma ve çözümleme dönüşü bütün komutlar için aynı şekilde çalışır. Çözümleme dönüşünde komut artık komut yazmacındadır, ve komut-işlem-kodu alanı bu komut için dağıtım bağlantısını belirlenmekte kullanılır.

Etiket:	ALU Denetim	SRC1	SRC2	ALU Hedef.	Memory	Mem. Reg.	PCWrite Control	Sıralama
t0 oku	Add	PC	4	–	ReadPC	IR	ALU	Sıra
t1 çözüml	Add	PC	ExShift	Target	–	–	–	Dağıtım-1

Okuma ve komut çözümleme mikrokomutlarının mikroprogram alanlarında kullanılan anımsatıcıların sonucu yapılan işlemler şöyledir:

oku	Add, PC, 4, ALU,	PC+4 ü hesapla, sonucu PC -ye yaz.
	ReadPC, IR,	PC adresindeki komutu oku ve IR a yaz
	Sıra.	sıradaki mikrokomutu çalıştır
çözümle	ADD, PC, ExtShft, Target,	PC ile işaret-genişlemiş- 2-bit-kaydırılmış-imm alanı topla ve sonucu hedef (target) yazmacına yaz.
	Dağıtım-1.	dağıtım-1 tablosundan komut işlem koduna göre bir sonraki mikrokomut adresini bul

Bir sonraki mikrokomutun adresini dağıtım-1 tablosu belirler:

Mikroprogramlama amaçlı Dağıtım-1 Tablosu						
Giriş: Opcode (Komut İşlem Kodu)	0	2	4	35	43	diğer
Çıkış: Mikrokomut Etiketi	RF1	jump1	beq1	lwsw1	lwsw1	oku

Bellek Erişim Komutları *lw* ve *sw*:

lw komutu

lBellek erişim komutları okuma ve çözümleme adımlarıyla başlar. Dağıtım-1 tablosu, hem *lw* hem de *sw* komutlarını t2 -ye bağlar. Dağıtım-2 tablosu *lw* ve *sw* komutlarının bir sonraki mikrokomut adreslerini belirler.

Mikroprogramlama amaçlı Dağıtım-2 Tablosu		
Giriş: Opcode (Komut İşlem Kodu)	35	43
Çıkış: Mikrokomut Etiketi	lw2	sw2

Tabloda *lw* komutu lw2 etiketli t3 -e yollanır.

Etiket:	ALU Denetim	SRC1	SRC2	ALU Hedef.	Memory	Mem. Reg.	PCWrite Control	Sıralama
t0 oku	Add	PC	4	–	ReadPC	IR	ALU	Sıra
t1 çözüml	Add	PC	ExShift	Target	–	–	–	Dağıtım-1
t2 lwsw1	Add	rs	Extend	–	–	–	–	Dağıtım-2
t3 lw1	Add	rs	Extend	–	ReadALU	–	–	Sıra
t4 lw2	Add	rs	Extend	–	ReadALU	WriteRT	–	Oku

lw için çalışan beş mikrokomuttaki anımsatıcılar şunları yapar:

oku	Add, PC, 4, ALU,	PC+4 ü hesapla, sonucu PC -ye yaz.
	Read-PC, IR,	PC adresindeki komutu oku ve IR a yaz
	Sıra.	sıradaki mikrokomutu çalıştır
çözümle	ADD, PC, ExtShft, Target,	PC ile işaret-genişlemiş- 2-bit-kaydırılmış-imm alanı topla ve sonucu hedef (target) yazmacına yaz.
	Dağıtım-1.	bir sonraki mikrokomut adresini dağıtım-1 den bul
lwsw1:	Add, rs, extend,	ALU da $rs+işaret genişlemiş imm hesaplanır
	Dağıtım-2.	işlem kodunu dağıtım-2 tablosunda kullan.
lw2	Add, rs, Extend, Read-ALU,	bellekte Mem[$rs+imm] adresi için okuma başlar.
	Sıra.	Sıradaki mikrokomut çalıştırılır
lw3	Add, rs, Extend, ReadALU, WriteRT,	Mem[$rs+imm] → $rt , ALU çıkışından adreslenen bellekteki veri rt-yazmacına yazılır
	Oku.	Yeni komut okumaya gider.

sw komutu

sw komutu da çözümle mikrokomutunda dağıtım-1 tablosundan lwsw1 -e, lwsw1 mikrokomutundaki dağıtım-2 tablosundan ise sw2 ye geçer.

Etiket:	ALU Denetim	SRC1	SRC2	ALU Hedef.	Memory	Mem. Reg.	PCWrite Control	Sıralama
t0 oku	Add	PC	4	–	ReadPC	IR	ALU	Sıra
t1 çözümle	Add	PC	ExShift	Target	–	–	–	Dağıtım-1
t2 lwsw1	Add	rs	Extend	–	–	–	–	Dağıtım-2
t5 sw2	Add	rs	Extend	–	WriteALU	ReadRT	–	Oku

sw için çalışan dört mikrokomuttaki anımsatıcılar şunları yapar:

oku	Add, PC, 4, ALU,	PC+4 ü hesapla, sonucu PC -ye yaz.
	Read-PC, IR,	PC adresindeki komutu oku ve IR a yaz
	Sıra.	sıradaki mikrokomutu çalıştır
çözümle	ADD, PC, ExtShft, Target,	PC ile işaret-genişlemiş- 2-bit-kaydırılmış-imm alanı topla ve sonucu hedef (target) yazmacına yaz.
	Dağıtım-1.	dağıtım-1 tablosundan komut işlem koduna göre bir sonraki mikrokomut adresini bul
lwsw1:	Add, rs, extend,	ALU $rs+sign-extended immediate -ı hesaplar
	Dağıtım-2.	Sıradaki dönüşü işlem kodu ve Dağıt-2 belirler
sw1	ADD, rs, Extend, WriteALU, ReadRT	rt deki veri Mem[$rs+imm] yazılır
	Oku .	Yeni komut okumaya gider.

sw , dördüncü dönüşte yeni komut okumaya dönerek tamamlanır.

R-tipi Komutlar

Bütün R-tipi komutların işlem kodu (*opcode*) sıfırdır, ve *Dağıtım-1* tablosu hepsini RF1 etiketli t6 durumuna yollar.

Etiket:	ALU Denetim	SRC1	SRC2	ALU Hedef.	Memory	Mem. Reg.	PCWrite Control	Sıralama
t0 oku	Add	PC	4	–	ReadPC	IR	ALU	Sıra
t1 çözüml	Add	PC	ExShift	Target	–	–	–	Dağıtım-1
t6 RF1	Func	rs	rt	–	–	–	–	Sıra
t7 RF2	Func	rs	rt	rd	–	–	–	Oku

R-tipi mikrokomutların alanlarındaki mnemoniklerin şunlara neden olur:

okuma	Add, PC, 4, ALU,	PC+4 ü hesapla, sonucu PC -ye yaz.
	ReadPC, IR,	PC adresindeki komutu oku ve IR a yaz
	Sıra.	sıradaki mikrokomutu çalıştır
çözümle	ADD, PC, ExtShft, Target,	PC ile işaret-genişlemiş- 2-bit-kaydırılmış-imm alanı topla ve sonucu hedef (target) yazmacına yaz.
	Dağıtım-1.	komut işlem kodunu dağıtım-1 tablosunda kullan.

Dağıtım-1 tablosundan sonra:

RF1	Funct, rs, rt,	ALU da *$rs* ve *$rt* ye *fn* alanındaki işlemi uygula. fn($rs,$rt) → ALU sonucu
	Sıra .	bir sonraki adresteki mikrokomutu çalıştır
RF2	Func, rs, rt, rd,	ALU da fn($rs, rt) -yi hesapla ve sonucu $rd -ye yaz
	oku.	yeni komut okumaya git

Atla ve DallanKomutları

j komutu çözümle mikrokomutunda Dağıtım-1 ile jump1 etiketli t9 durumuna gönderilir.

Etiket:	ALU Denetim	SRC1	SRC2	ALU Hedef.	Memory	Mem. Reg.	PCWrite Control	Sıralama
t0 oku	Add	PC	4	–	ReadPC	IR	ALU	Sıra
t1 çözüml	Add	PC	ExShift	Target	–	–	–	Dağıtım-1
t9 jump1	–	–	–	–	–	–	Jump	Oku

j komutu üçüncü dönüşte tamamlanır

j nin mikrokomut alanlarındaki anımsatıcıların karşılığı şunlardır:

okuma	Add, PC, 4, ALU,	PC+4 ü hesapla, sonucu PC -ye yaz.
	Read-PC, IR,	PC adresindeki komutu oku ve IR a yaz
	Sıra.	sıradaki mikrokomutu çalıştır
çözümle	ADD, PC, ExtShft, Target,	PC ile işaret-genişlemiş- 2-bit-kaydırılmış-imm alanı topla ve sonucu hedef (target) yazmacına yaz.
	Dağıtım-1.	komut işlem kodunu dağıtım-1 tablosunda kullan.
jump1	Jump,	26-bit-immJ×4 anlık değerini PC -ye yaz
	okuma .	yeni komut okumaya git

Ve *beq* komutu *dağıtım-1* tablosunda beq1 etiketli mikrokomuta yönlenir:

Etiket:	ALU Denetim	SRC1	SRC2	ALU Hedef.	Memory	Mem. Reg.	PCWrite Control	Sıralama
t0 oku	Add	PC	4	–	ReadPC	IR	ALU	Sıra
t8 beq1	Subt	rs	rt	–	–	–	Target	Oku

beq -nun mikrokomut alanlarındaki mnemoniklerin sonuçları şöyledir:

okuma	**Add, PC, 4, ALU,**	PC+4 ü hesapla, sonucu PC -ye yaz.
	Read-PC, IR,	PC adresindeki komutu oku ve IR a yaz
	Sequence.	sıradaki mikrokomutu çalıştır
çözümle	ADD, PC, ExtShft, Target,	PC ile işaret-genişlemiş- 2-bit-kaydırılmış-imm alanı topla ve sonucu hedef (target) yazmacına yaz.
	Dağıtım-1.	komut işlem kodunu dağıtım-1 tablosunda kullan.

Dağıtım-1 den sonra:

beq1	**Subt, rs, rt, target-cond,**	\$rs – \$rt hesapla ve ***PCWriteCond*** ver, böylece sonuç sıfırsa Hedef-Yazmacı PC -ye yazılsın.
	okuma .	yeni komut okumaya git

Mikroproglamlama, mühendislere, özel amaçlı çok-dönüşlü komutlar gerçekleştirme olanağı sağlar. Komut-seti son derece geniş olan karmaşık komut-kümeli bilgisayarların denetim birimini başka yollarla tasarlamak hemen hemen olanaksız olduğundan bu önemli yöntemin karmaşık-komut-setli (CISC) bilgisayarların gelişiminde önemli rol oynamıştır.

5.6.4 Komut Kümesinin Genişletilmesi

Varolan veriyolu mimarisi mikroprogramlama ve kimi durumlarda veriyolu seviyesinde bazı ufak değişiklikler ile neredeyse tüm MIPS komutlarını gerçekleştirmek için uygundur.

Örnek 5-3: Tablo 5-38 de verilen mikroprogram kodunundan başlayarak anlık topla (add-immediate) komutunu mikro-programlama ile gerçekleştiriniz. Mikroprogram kodunda ve Dağıtım-Tablolarındaki tüm gerekli modifikasyonları gösteriniz.

I- tipi alan-bit sayısı:	**opc** 6	**rs** 5	**rt** 5	**imm** 16	**anımsatıcı $rd, $rs, imm** (imm işaretli 16-bitlik sayıdır)
addi	8	2	1	100	addi $1,$2,100 anlık topla

<<

t0 dan t9 -a kadarki durumlar, *R-tipi*, *lw*, *sw*, *beq*, ve *j* komutlarını gerçekler. *addi* komutunun gerçeklemesi için yeni durumlar eklemeliyiz. Dağıtım-1 tablosunda opcode=8 i yeni durum t10 a bağlamak için dağıtım-1 tablosu şöyle değiştirilmelidir:

Dağıtım-1 Tablosunda ADDI komutu eklemek üzere değişiklikler							
Giriş: Opcode (Komut İşlem Kodu)	0	2	4	35	43	8	diğer
Çıkış: Mikrokomut Etiketi	RF1	jump1	beq1	lwsw1	lwsw1	addi1	okuma

addi komutunu işleyecek mikrokomut satırları şöyle olacaktır:

Etiket:	ALU Denetim	SRC1	SRC2	ALU Hedef.	Memory	Mem. Reg.	PCWrite Control	Sıralama
t0 oku	Add	PC	4	–	ReadPC	IR	ALU	Sıra
t1 çözümle	Add	PC	ExShift	Target	–	–	–	Dağıtım-1

Ve Dağıtım-1 addi -yi etiketi addi olan t10 durumuna yöneltir.

t10 addi1	Add	rs	Extend	–	–	–	–	Sıra
t11 addi2	Add	rs	Extend	rt	–	–	–	Oku

Son iki mikrokomuttaki anımsatıcılar şu işlemleri gerçekleştirir.

addi1	Add, rs, Extend,	($rs + sign-extended-imm) → ALUResult
	Sıra.	bir sonraki adresteki mikrokomutu çalıştır
addi2	Add, rs, Extend, rt,	ALU ($rs + sign-extended-imm) toplamasını yaparken, sonucu $rt -ye yaz.
	Oku.	yeni komut okumaya git

R-tipi, *lw*, *sw*, *beq*, *j* ve *addi* için mikroprogram kodunun tamamı

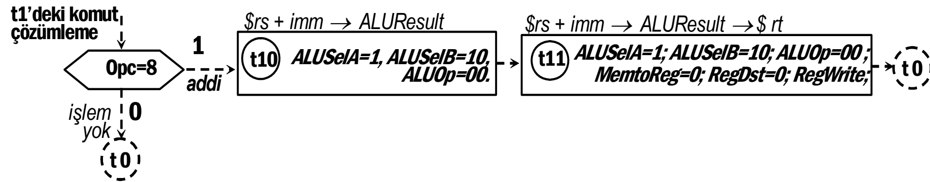
	Etiket:	ALU Denetim	SRC1	SRC2	ALU Dest.	Memory	Mem. Reg.	PCWrite Control	Sıralama
t0	oku	Add	PC	4	–	ReadPC	IR	ALU	Sıra
t1	çözümle	Add	PC	ExShift	Target	–	–	–	Dağıtım-1
t2	lwsw1	Add	rs	Extend	–	–	–	–	Dağıtım-2
t3	lw1	Add	rs	Extend	–	ReadALU	–	–	Sıra
t4	lw2	Add	rs	Extend	–	ReadALU	WriteRT	–	Oku
t5	sw2	Add	rs	Extend	–	WriteALU	ReadRT	–	Oku
t6	RF1	Func	rs	rt	–	–	–	–	Sıra
t7	RF2	Func	rs	rt	rd	–	–	–	Oku
t8	beq1	Subt	rs	rt	–	–	–	Target	Oku
t9	jump1	–	–	–	–	–	–	Jump	Oku
t10	lw1	Add	rs	Extend	–	–	–	–	Sıra
t11	lw1	Add	rs	Extend	rt	–	–	–	Oku

>>

Örnek 5-4: *addi* komutunun gerçeklemesi için Şekil 5-53 ve Şekil 5-54'deki ASM yada FSM-çizimi üzerinde gerekli değişiklikleri gösteriniz.

<<

addi komutunda *fn* alanı *sa* ve *rd* ile birleşerek *imm* anlık değer alanını oluşturur. *fn* alanı bulunmadığından t10 -da ALU daki toplama *ALUOp* = 00 ile sağlanır. t11 -de, *RegDst*=0, *MemtoReg*=0 ve *RegWrite* verilerek *ALUResult $rt* -ye yazılır.



>>

jal ve *jr* komutlarının mikroprogramlanması büyük dikkat gerektirir, ama yeni veriyolları kullanmadan gerçekleştirilebilir.

Bazı komutlar veriyolu devresinde küçük değişiklikler gerektirir. *lui* komutu veriyolunda 16-bit-kaydırma devresiyle kaydırılmış anlık değeri seçecek bir çoklayıcı gerektirir. n-bit kaydırma işlemi çoklayıcı grubu kullanarak bileşimsel olarak yapılabilir. Bileşimsel kaydırıcının bağlantı arayüzü veriyolunda önemli derecede değişiklik gerektirir. *mult* komutu ya bir çok-dönüşlü çarpma birimi veriyolu yada bileşimsel çarpıcı dizisi kullanmayı gerektirir. Her iki durumda da kaydırma veriyolunda olduğu gibi veriyolu değişiklikleri gerekir. 32-bitlik sözcükler çarpıldığında, çarpım 64-bitlik çiftsözcüktür, ve özel yazmaçlara (HI,LO) yazılır. Böylece fazladan çarpma başına bir komut harcayarak gerekli veriyolu değişiklikleri basitleştirir.

5.7 Sonuç

RISC işlemcilerin çok-dönüşlü gerçeklemesi veriyolunun donanım birimlerini basitleştirir. Bununla beraber, denetim birimi tasarımda tek-dönüşlü gerçeklemeye kıyasla daha fazla çaba gerektirir.

Çok-dönüşli denetim tasarımının önemli derecedeki karmaşıklığı yeni gerçekleme tekniklerinin doğmasına neden olmuştur. Mikroprogramlama teknikleri, CAD araçları, ve PLA devreleri kullanmak denetim biriminin tasarım ve geliştirme evrelerini kolaylaştırmak için en yaygın yollardır.

Çok-dönüşlü gerçeklemeyle performans artışı ihmal edilebilir düzeyde olduğu halde, yeni yaklaşım aynı saat dönüşünde birden fazla komut işlemeye izin veren ardışık düzen çalışmaya (datapath-pipelining) uygun olduğundan işlemci tasarımına yeni bir boyut getirir.

Kaynakça

[1] John L. Hennessy, David A. Patterson, "Computer Organization and Design, the Hardware/Software Interface" Morgan Kaufmann Publishers, 1994.

[2] Charles Price "*MIPS-IV Komut Set, Revision 3.2"* MIPS Technologies, Inc. 2011 North Shoreline Mountain View, California 94039-7311 September, 1995

[3] David,A. Patterson "Reduced komut set computers", Communications of the ACM January 1985 Vol.28 Nr.1 p.p.8-21,

[4] Samuel O.Aletan "An overview of RISC architecture" p.p 11-20, 1992 ACM 0-89791-502-X/92/0002/0011

[5] Randal E. Bryant, David O'Hallaron "Computer Systems, A Programmer's Perspective" Pearson Education International, 2003.

[6] Hugh Osborne, "The Postroom Computer: Teaching Introductory Undergraduate Computer Architecture" SIGCSE'02, February 27-March 3, 2002, Covington, Kentucky, USA.

[7] David,A. Patterson "Riscy Patents" *Special Assignment to Computer Architecture News, University of Cafifomia at Berkeley, 1988 NY Times Company*

[8] A. Patterson, E. S. Fehr, and C.H. Sequin" Design Considerations for the VLSI processor of X-tree. *The 6th Annual International Symposium on Computer Architecture* (April 1979).

[9] David A. Patterson and Carlo H. Sequin. "Design Considerations for Single-Chip Computers of the Future." *IEEE Transactions on Computers C-29,* 2 (February 1980), 108-116.

[10] Thomas R. Gross, John L. Hennessy, Steven A. Przybylsky, and Christopher Rowen "Measurement and Evaluation of the MIPS Architecture and Processor ", ACM Transactions on Computer Systems, Vol. 6, No. 3, August 1988, Pages 229-257.

[11] Niklaus Wirth, "Microprocessor Architectures: A comparison based on code generation by compiler" Communications of the ACM October 1986 Volume 29 Number 10 p.p.978-990

[12] M. Morris Mano, Digital Design, 3.ed, Prentice Hall, 2002.

[13] Richard S. Sandige "Digital Design Essentials", Prentice Hall, 2002.

Dizin

:

:1'lik tümleyen (*1's complement*) 63
:2'lik-tümleyen (*2's complement*) 62
:32-bitlik ALU (*32-bit ALU*) 72

A

add (topla) 21, 28
addi (anlık-topla) 28
ağırlıklı aritmetik ortalama (*weighted arithmetic mean*).............................. 14
akış şeması (*flowchart*) 121
algorithm... 121
algoritmik durum makinesi (*algorithmic state machine*, *ASM*) 121
alttan-taşma (*underflow*) 107
altyordam (*procedure, subroutine*) 37
ALU Denetim alanı 200
ALU Hedef alanı 200
ALU-denetim birimi (*ALU-control unit*) 152
ALUOp 178
ALUSelA 178
ALUSelB 178
ALUSrc 155
ana denetim birimi (*main control unit*) 154
anımsatıcı (*mnemonic*)......................... 21
anlamlı-basamaklar, anlamlılar (*significant field*) 105
anlık (*immediate addressing*) 47
anlık (*immediate*) 27
anlık komut (*immediate instruction*)...... 28
anlık veri (*immediate data*)............. 20, 28
anlık-adresleme (*immediate addressing*) 43
anlık-değer-yükle sözde-komutu (*load immediate pseudo-instruction* ***li***)...... 28
anuyumlu (*synchronous*)................... 137
anuyumlu (*synchronous*).................... 139
anuyumlu sırasal durum makinesi (*synchronous sequential state machine*) 141
arakat (*buffer*) 69
ardışık düzen (*pipeline*))..................... 171
argüman [=bağımsız-değişken] (*argument*)....................................... 40
aritmetik ortalama (*arithmetic mean*) 14
aritmetik-mantık-birimi (*arithmetic logic unit,* ***ALU***).................................. 59, 69
artan (*augend*) 64
ASM bloğu yada öbeği (*ASM block*) ... 123
ASM, *algorithmic state machine* algoritmik durum makinesi 121
aşağı yuvarlama (*round down*) 110
atla (*jump* ***j***) 33
atla-ve-bağla (*jump-and-link* ***jal***)........... 37
ayıklayıcı (*debugger*) 37
ayrık veri belleği (*isolated data-memory*) 146
azalan (*minuend*) 65

B

bağlayıcı programı (*linker program*) 36
bağlayıcı-yükleyici (*linking loaders*)...... 37
başarım (*performance*)........................... 6
başarım ölçütü (*performance metric*)...... 9
başarım tanımı (*performance definition*). 6
bayt (*byte*)... 60
belirtmek (*to specify*)............................ 44
bellek adres uzayı (*memory address space*) 32
bellek alanı 200
bellek erişim (*memory access*) 24
bellek kullanım komutları (*memory reference instructions*)................... 191
bellek yazmacı alanı 200
bellekli-program işlemcisi (*stored-program processor*) 137
belleksiz (*memoryless*)....................... 139

beq *branch-if-equal* (eşit-ise-dallan) 35
betimlemek (*describe*) 22
bileşimsel öbek (*combinational block*) 139
bilimsel gösterim (*scientific notation*) . 105
bir-bitlik-ALU (*one-bit-ALU*) 71
birimsel (*modular*) 36
birleşimsel (*combinational*) 94
birleşimsel çarpıcı (combinational array multiplier) 94
bit (*bit*) 60
bne *branch-if-not-equal* (eşit-değilse-dallan) 35
Booth'un algoritması (*Booth's algorithm*) 92
bölüm (*quotient*) 96
budama (*truncate*) 110
büyük-uçtan (*big-endian*) 25

C

clock cycle 123
CPU süresi (*CPU time*) 6

Ç

çağıran (*caller*) 40
çağıran saklar (*caller save*) 40
çağırılan (*callee*) 40
çağırılan saklar (*callee save*) 40
çalışma süresi (*run time*) 12
çapma algoritması (*multiplication algorithm*) 84
çapraz çağrı (*cross-reference*) 36
çarpım (*product of multiplication*) 83
çevirici dili kaynak programı (*assembly language source program*) 36
çevirici program (*assembler program*) .. 36
çıkan (*subtrahend*) 65
çıkarma (*subtraction*) 65
çift-kesinlikli-kayan (*double-precision-float*) 108
çift-sözcük (*double-word*) 21
çok-görevli *(multi-task)* 6
çok-kullanıcılı *(multi-user)* 6
çoklayıcı (*multiplexer*) 69, 140
çoklu erişimli yazmaç dosyası (*multiple-access register file*) 23
çözünürlük (*resoluıon*) 104
çözünürlük (*resolution*) 103

D

dağıtım (*dispatch*) 190
dallanma çıkışı (*branch output*) 156
dallanma komutu (*branch instruction*). 193
dallanma-hedef-adresi (*branch-target-address*) 164, 168
dallanma-hedef-adresi *branch-target-address*) 170
değerlendirme kartı (*evaluatıon board*)... 3
değiştirge (*parameter*) 37
değiştirge (*parameter*) 40
denetim birimi (*controller unit*) 121
denetim birimi (*controller*) 125
denetim devresi tasarımı (*control circuit design*) 131
denetim mantığı (*control logic*) 131
derleyici (*compiler*) 3, 36
devreyi seçmek (*enable a circuit*) 139
dizgi (*string*) 21
dizi (*array*) 21, 165
donatım (*equipment*) 19
döngü (*loop*) 170
dönüş-adresine-atla (*jump-return-address* ya da *jump to register* ***jr***) 37
dörtbit (*nibble*) 60
durum elemanıdır (*state element*) 139
durum kutusu (*state box*) 123
durum makinesi (*state machine*) 121
duvar süresi (*wall time*) 7
D-yazboz (*D-yazboz*) 139

E

eğilimli biçim (*biased format*) 62
eğilimli sayı (*biased number*) 62
eğilimli-üst (biased-exponent) 107, 108
eklenen (*addend*) 64
eksi sayı (*negative number*) 62
eksileme (*negation*) 65
elde *(carry-out)* 64
elde aktarma (*carry propagate*) 78
elde çıkışı (*carry-output*) 70
elde girişi (*carry-input*) 70
elde oluşturma (*carry generate*) 78
en-az-komutlu (*code-optimized*) 41
en-hızlı-kod (*time-optimum*) 41
en-sağ-bit (*least significant bit* ***LSB***) 59
en-sol-bit (*most significant bit MSB*) 59
en-yakın-çifte yuvarlama (*nearest-even*) 110
erim aralığı (*range*) 108
erim aralığı (*representation range*) 104

eşit-değilse-dallan (*branch-if-not-equal* ***bne***) 35
eşit-ise-dallan (*branch-if-equal* ***beq***) 35
etkin sinyal (*active signal*) 139
evirici (*inverter*) 69

F

fark (*difference*) 65
finite-state machines 187
fiziksel-bağlantılı (*hardwired*) 121
FSM denetim birimi (*FSM control unit*) 194

G

gecikerek-ilerleyen-elde *(ripple-carry propagation)* 77
geçen süre (*elapsed time*) 7
geometrik ortalama (*geometic mean*) ... 14
gerçek sayı (real number) 103
gerçekleme (*implementation*) 121
gömülü sistem (*embedded system*) 21, 138
gömülü sistem uygulamaları (*embedded system applications*) 32
göreceli adresleme (*relative addressing*) 44
göreceli MIPS (*relative MIPS*) 11
göreli koşul (*relational condition*) 35
göreli sınama (relational test) 35
gösterim çözünürlüğü (*resolution of representation*) 104
gösterim hatası (*error of representation*) 104
gösterim kesinliği, göreceli (*relative resolution of representation*) 104
gösterim kesinliği, mutlak (*absolute resolution of representation*) 104

H

Harvard mimarisi (*Harvard architecture*) 32
Harvard mimarisi (*Harvard architecture*) 138
hassaslık (*accuracy*) 103
hedef (*destination*) 138
hedef yazmacı (*target register*) 181
Hertz 7
HI yazmacı (*HI register*) 34

I

IEEE-754 gösterimi (IEEE-754 notation) 107
IorD 178
IRwrite 178
I-tipi komut biçimi (*I-type instruction format*) 30

İ

i/o işlem süresi (*i/o processing time*) 6
içiçe (*nested*) 38
içiçe çağrı (*nested call*) 38
içiçe çağrı derinliği (*depth of the nesting*) 38
içiçe yordam çağrısı (*nested procedure call*) 38
iki seviyeli VE-VEYA devresi *(two-level AND-OR circuit)* 78
ikili sayı sistemi (binary number system) 59
ilişkilendirme (mapping) 187
işaret biti (*sign-bit*) 62
işaret çıkışı *(sign output*) 75
işaret genişletme (*sign-extend*) 150
işaret taşması (*sign overflow*) 66
işaret-büyüklüklü sayı (*signed-magnitude number*) 62
işaretli çarpma (*signed multiplication*) .. 89
işaretli tamsayı aritmetiği (*signed integer arithmetic*) 59
işaretsiz tamsayı (*unsigned integer*) 64
iş-bitirme-hızı (*throughput*) 6
işlem (*operation*) 22
işlem kodu (*operation code*) 22
İşlem kodu çizelgesi (*Opcode Map*) 49
işlemci tasarım ilkeleri (*processor design rules*) 20
işlenen (*operand*) 21
işletim sistemi (*operating system*) 3, 37
işlev (*function*) 22
işlev kodu (*function-code*) 22
İyi tasarım uzlaşı gerektirir (*good design demands compromise.*) 31

J

j *jump* (atla) 33
jal jump-and-link (atla-ve-bağla) 37
J-tipi biçim (*J-type format*) 32
J-tipi komut (*J-format instruction*) 164
jump instruction 194

K

kalan (*remainder*) 96
karakter (*character*) 21
karar kutusu (*decision box*) 123

karşılaştırma program takımı (*benchmark suite*) 12
karşılaştırma programı (*benchmark*) 12
kayan (*float*) 107
kayan noktalı kodlu sıkıştırma (*floating point coding compaction)* 105
kayan sayı (*float number*) 108
kayan-nokta (*floating-point*) 11
kayan-noktalı toplama (*floating point addition*) 111
kayan-noktalı-çarpma (floating point multiplication) 112
kayan-noktalı-sayı-birimi (*floating-point-number-unit, **FPU***) 59
kayan-noktalı-yanişlemci birimi (Floating point coprocessor unit FPU) 107
kaynak dosyası (*source file*) 36
kesinlik (*precision*) 103, 108
kesme işareti (*prime mark*) 69
kip (*mode*) 43
komut başına ortalama dönüş (*cycle-per instruction*) 8
komut belleği (*instruction memory*) 144
komut biçimi (*instruction format*) 29
komut çözümleme (*instruction decode*) 190
komut kodu (*opcode*) 156
komut okuma (*instruction fetch*) 189
komut okuma (instruction-fetch) 161
komut-başı-saat-dönüşü (*clock-cycles-per-iınstructıon*, ***CPI***) 137
konum-ağırlığı (*positional weight*) 59
koşul kutusu (*conditional box*) 123
koşulsuz atlama (*unconditional jump*) .. 32
kural (*convention*) 37
kuraldışı yordam (*exception routine*) 67
kural-dışı-durum (*exception*) 49, 50
küçük girişi *(less input*) 75
küçükse bir yap (*set on less than **SLT***) 74
küçükse-bir-yap (*set-on-less-than **slt***).. 35
küçük-uçtan (*little-endian*) 25

L

***li** load immediate pseudo-instruction* (anlık-yükle sözde-komutu) 28
LO yazmacı (*LO register*) 34
LO-dan-aktar (move-from-LO mflo)...... 34
***LSB** least significant bit* (en-sağ-bit) 59
lw komutu (*lw instruction*) 163
lw komutu (*lw-instruction*) 192
***lw** load-word* (sözcük-yükle) 25

M

mantık çözümleyici (*logic analyzer*) 3
mantıksal işlem (*logical operation*) 67
mantıksal kaydırma (*logical shift **sll***) 67
mantıksal kaydırma (*logical shift*) 67
mantıksal tümleyen (logic complement) 66
mantıksal-sola-kaydır (*shift-left-logical **sll***) 27
MemRead 155, 178
MemtoReg 155, 178
MemWrite 155, 178
metin dosyası (*text file*) 36
mfhi move from HI (HI 'den aktar) 34
mflo move from LO (LO 'dan aktar)...... 34
MFLOPS 11
MIPS (saniyede milyon komut *Million Instructions Per Second*) 9
MIPS kayan-noktalı komutları (*MIPS FPU Instructions*) 116
mikrokomut biçimi (*microinstruction format*) 199
mikroprogram kodu (*microprogram code*) 201
mikroprogram yöntemleri *microprogramming methods*) 187
Mikroprogramlı Denetim (*microprogrammed control*) 198
Moore yasası (*Moore's law*) 1
***move** pseudo-instruction* (yolla sözde-komutu) 28
***MSB** most significant bit* (en-sol-bit)..... 59
mul multiply pseudo-instruction (çarp komutumsu) 35
***mult** multiply instruction* (çarp komutu)34
multiple-clock-cycle control signals 177
mutlak kesinlik (*absolute precision*).... 104

N

nesne dosyası (*object fıle*) 36
nesne dosyası (*object file*) 36
nesne kütüphanesi (*object-library*) 36
nesne-kodu (*object-code*) 36
normalize MFLOPS (*normalized MFLOPS*) 12

O

olabilirlik (*possibility*) 104
olumsuzlama (*negate*) 62

onaltılık sayı sistemi (*hexadecimal number system*)............................... 60

Ö

öbek yapısı (*block structure*)................ 19
önbellek (*cache memory*)..................... 27
öncül bir (*leading-one*)....................... 106
önden-eldeli (carry-lookahead) 78
öykünücü (*emulator*) 3

P

PCClr.. 155
PCLoad.................................... 155, 184
PCSrc 184
PCWrite 184
PCWriteCond.................................... 184
PLA programlanabilir mantik dizisi (*programmable logic array*)........... 158
program akış yapıları (*program flow structures*) 36
program kodunun yürütülmesi (*program-code execution*)............................ 165
program sayacı (*Program Counter*)...... 32
programı bellekte tutma kavramı (*stored-program concept*) 32

R

rakam (*numeral*) 59
RegDst...................................... 155, 178
RegWrite................................... 155, 178
R-format instruction 162
R-tipi komut (*R-type instruction*)......... 190
R-tipi komut biçimi (*R-type instruction format*)... 29

S

saat hızı (*clock rate*) 7
saat sinyali (*clock signal*)................... 139
saatdönemi (*clock period*)...................... 7
sağa kaydırma (*shift-right-logical* ***srl***)... 67
saniyede milyon komut (*MIPS*).............. 9
saniyedeki dönüş (*cycles per second*).... 7
sayı-tabanı (*base or radix*)................... 59
seç sinyali (*enable signal*).................. 142
seçilmiş (enable).................................. 69
seçme girişi (*enable input*)................. 140
sekizlik sayı sistemi (*octal number system*) 60
sıfır (*zero*)... 76
sıfırdeğil (*nonzero*)............................... 76
sıfırla (*reset*) 156
sıra sayacı (*sequence counter*)............ 85
sıralama alanı 200
sıralayıcı (*sequencer*) 197
sırasal durum makinesi (*sequential state machine*) 121
sırasal öbek (*sequential block*)........... 139
sign field 105
sinyal vermek (*assert a signal*)........... 139
sinyal vermemek (*deassert a signal*) .. 139
sll *shift-left-logical* (sola kaydırma) 67
slt *set-on-less-than* (küçükse-bir-yap) .. 35
sola kaydırma (*shift-left-logical* ***sll***) 67
son giren ilk çıkar (*Last In First Out*) 38
Sonlu-Durum-Makinesi (*Finite-State-Machine FSM*) 187
sözcük (*word*)...................................... 21
sözcük (*word*)...................................... 60
sözcük-sakla (*store-word* ***sw***) 25
sözcük-yükle (*load-word* ***lw***)................. 25
sözde-komut (*pseudo-instruction*) 28
SRC1 alanı 200
SRC2 alanı 200
srl *shift-right-logical* (sağa kaydırma)... 67
sunucu (*server*)..................................... 3
sw instruction 163
sw komutu (*sw instruction*)................. 193
sw *store-word* (sözcük-sakla) 25

T

tam toplayıcı (*full adder*) 69, 70
TargetWrite 184
taşma (*overflow*) 66, 107
taşma algılaması (*overflow detection*) .. 75
tek saat donüşlü başarım (*single clock cycle performance*) 172
Tek-kesinlikli-sayı (*single precision number*)... 107
tek-saat-dönüşü (single-clock-cycle) .. 143
tek-uyaranlı-tasarım (*one-hot-design*) 121
tepe MFLOPS (*peak MFLOPS*) 12
tepe MIPS (*peak MIPS*)........................ 11
toplam (*sum*)....................................... 64
toplama (*addition*) 64
toplam-yürütme-süresi *(total execution time)* 6
tümleşik geliştirme ortamı (*integrated development environments*)............. 37

Ü

üç-durumlu-arakat (*three-state-buffer*) 140
üst (*exponent field*) 105

üste-anlıkdeğer-yükle (*load-upper-immediate **lui***) 29

V

varış yazmacı (*destination-register*) 21
ve (and) 68
VE geçiti (*AND gate*)............................ 69
veri aktarma (*data transfer*).................. 24
veri işlem birimi (*data processing unit*)121
veri işlemci (*data-processor*).............. 125
veri işlemcisi tasarımı (*data processor design*) 131
veri yolu birimi (*data path*) 121
veriyolu (*data path*).............................. 23
veriyolu tasarımı (*datapath design*).... 131
veya (or) 68
VEYA geçiti (*OR gate*) 69
Von Neumann mimarisi (*Von Neuman architecture*) 32

Y

yada geçiti (*XOR gate*)......................... 70
yakalama koşulu (*trap condition*).......... 49
yakalamalı dallanma (*trap branch*) 49
yanıt zamanı (*response time*)................. 6
yanlış ayıklayıcı (*debugger*) 3
yapılanış (*configuration*)....................... 13
yarım sözcük (*half-word*)...................... 21
yayılan eldeli (*ripple carry*) 72
yayılım zamanı (propagation time) 141
yaz sinyali (*write signal*) 142
yazboz (*flip-flop*) 116, 121
yazmaç (*register*) 139
yazmaç adres alanı (*register address field*) 46
yazmaç dosyası (*register file*)............... 22
yazmaç-göreceli-adresleme (*register-relative addressing*) 44
yazmaçlı adresleme (*register addressing*) 46
yerdeğişimli adresleme (*displacement addressing*)...................................... 46
yığıt (*stack*) 38
yığıt göstergesi (*stack pointer*) 38
yol (*bus*).. 140
yolla (*move*) 28
yordam (*routine*).................................. 36
yordam gövdesi (*procedure body*) 41
yukarı yuvarlama (*round up*) 110
yükle sinyali (*load signal*) 142
yükleyici (*loader*)................................. 36
yürütülebilir dosya (*executable file*)...... 36

Dr. Bodur Doktora (PhD. EEE 1990), Yüksek Lisans (MS. EE 1981) ve Lisans (Kimya Müh. 1979) derecelerini Ankara, Orta Doğu Teknik Üniversitesinden aldı. ODTÜ ve Doğu Akdeniz Üniversitesi (DAU) Elektrik Mühendisliği Bölümlerinde denetim sistemleri, robotik, yapay us ve sayısal tasarım alanlarında çalıştı. Gömülü donanım ve yazılım sistemleri tasarım mühendisi olarak dört yıl Kanada'da NADİ, Inc., firmasında, ve iki yıl karar-destek ve veri-çıkarım yazılımı tasarım, geliştirme ve gerçekleme konusunda araştırmacı olarak çalıştı. Halen Kuzey Kıbrıs Doğu Akdeniz Üniversitesi Bilgisayar Mühendisliği Bölümünde öğretim üyesi olarak çalışmaktadır.

Bilgisayar Organizasyonu: RISC Donanımına Giriş. (226 sayfa)

Bu kitap, *Bilgisayar Organizasyonu: RISC Donanımına Giriş"* ileride bilgisayar mühendisliğinin yazılım ve donanım alanlarında çalışması beklenen bilgisayar mühendisi adayları için yazılmıştır. Kitapta basitleştirilmiş bir 32-bit RISC işlemciyi gerçekleştirmek için gerekli temel bileşenleri ile işlevleri ASM (Algorithmic State Machine) çizimlerine varıncaya dek detaylarıyla betimlenir.

Birinci bölüm, hedef çalışmanın sınırlarını belirten genel bir tanıtım niteliğindedir.

İkinci bölüm, bilgisayar sistemlerinin başarımını karşılaştırmak için olduğu kadar, yeni işlemcilerin tasarım ve geliştirilmesi için de uygun olan bir başarım ölçütü tanımlar. Bu bölüm ayrıca bilgisayar tarihinde kullanılmış çeşitli başarım ölçütlerini de ele alır.

Üçüncü bölüm, RISC kavramının C-dili programlarını daha verimli işlemek üzere komut takımının basitleştirilmesinin bir sonucu olduğunu hatırlatarak, uygulama seviyesi programlama dilleri ile makine seviyesi komutları arasındaki ilişkiye odaklanır.

Dördüncü bölüm, aritmetik mantık birimiyle kayan noktalı sayı biriminin işlevsel çalışmasını ve fiziksel gerçeklemesini tanıtır. Çarpma ve bölme birimlerinin çalışması yazmaç (register) seviyesine kadar ayrıntılı ASM çizimleriyle betimlenir.

Beşinci ve son bölüm, denetim biriminin komut başına tek-saatlik ve çok-saatlik uygulamalarındaki yapıları ile tasarımını kapsar ve tipik bir RISC işlemcinin, temel yazmaç seviyesindeki bileşenlerden başlayarak tasarım ve mimarisinin anlaşılmasını hedef alır.

SONSÖZ:

Bu notlar, Doğu Akdeniz Üniversitesinde Bilgisayar Bölümünde verilmekte olan Bilgisayar Organizasyonu dersinin ders notları olarak Mehmet Bodur tarafından hazırlanmıştır. Bu notların EMO'nun yayın sitesinde e-kitap olarak yayınlanması için katkılarından dolayı Sayın Mehmet Bodur'a teşekkür borçluyuz.

Notlar, başlangıç bölümü dahil beş ayrı bölümden oluşmuştur, toplamı 220 sayfadır. Birinci bölüm, hedef çalışmanın sınırlarını belirten genel bir tanıtım niteliğindedir. İkinci bölüm, bilgisayar sistemlerinin başarımını karşılaştırmak için olduğu kadar, yeni işlemcilerin tasarım ve geliştirilmesi için de uygun olan bir başarım ölçütü tanımlar. Bu bölüm ayrıca bilgisayar tarihinde kullanılmış çeşitli başarım ölçütlerini de ele alır. Üçüncü bölüm, RISC kavramının C-dili programlarını daha verimli işlemek üzere komut takımının basitleştirilmesinin bir sonucu olduğunu hatırlatarak, uygulama seviyesi programlama dilleri ile makine seviyesi komutları arasındaki ilişkiye odaklanır. Dördüncü bölüm, aritmetik mantık birimiyle kayan noktalı sayı biriminin işlevsel çalışmasını ve fiziksel gerçeklemesini tanıtır. Çarpma ve bölme birimlerinin çalışması yazmaç (register) seviyesine kadar ayrıntılı ASM çizimleriyle betimlenir. Beşinci ve son bölüm, denetim biriminin komut başına tek-saatlik ve çok-saatlik uygulamalarındaki yapıları ile tasarımını kapsar ve tipik bir RISC işlemcinin, temel yazmaç seviyesindeki bileşenlerden başlayarak tasarım ve mimarisinin anlaşılmasını hedef alır.

Bu çalışmanın EMO kanalı ile yayınlanması için başından beri desteğini esirgemeyen Orhan (Örücü) Ağabeyimize, notların yayına hazırlanmasında katkılarından dolayı Emre (Metin) ve Hakkı (Ünlü) kardeşlerime teşekkür ederiz. Ayrıca bu tür mesleki yayınların e-kitap olarak çok düşük bedeller ile meslektaşlarına kazandırmak için bu yayın portalını oluşturma kararı alan 42. Dönem EMO Yönetimine bu öncü rölünden dolayı teşekkür ederiz.

E-Kitabı Yayına Hazırlayan
İbrahim Aydın Bodur

Sağda kalacak boş sayfa

DR. MEHMET BODUR

BILGISAYAR ORGANIZASYONU:

RISC DONANIMINA GIRIŞ

e-kitap

EMO Yönetim Kurulu 42. Dönem'de(Kasım 2010) bir yayın portalı oluşturdu. Bu yayın portalı üzerinde,daha önce de sürdürmekte olduğumuz, basılı dergilerimizin İnternet sürümleri, basılı kitaplarımızın tanıtımları ve çevrim içi satın alma olanakları ile doğrudan İnternet üzerinden bilgisayarınıza indirebileceğiniz e-kitapları çok düşük bedellerle edinebilme olanağına sahip olacaksınız.

İnternet sitemiz üzerinden e-kitap dağıtım hizmetini, yakında hizmete girecek olan EMO Yayın Portalı'nın öncülü olan, sitemizin yayın bölümünde yer alan e-kitaplarla uzunca bir süredir veriyorduk. Yayınlarımızı izleyenler hatırlayacaktır, ilk e-kitabımız, EMO üyesi Arif Künar'ın "Neden Nükleer Santrallere Hayır" kitabının PDF baskısıydı. Hükümetin Akkuyu'da nükleer santral kurma inadı maalesef hala kırılamadı. Dört yıl önce bastığımız bu kitap hala güncel!....

EMO'nun İnternet sitesi üzerinden hizmete giren bu yeni sitemizde yeni e-kitaplarla hizmete açıldı. Sizlerde varsa yayınlamak istediğiniz kitaplarınızı, notlarınızı bize iletebilirsiniz. Bu yayınlar yayın komsiyonumuzun değerlendirmesinden sonra uygun bulunursa yayınlanacak ve eser sahibine EMO ücret tarifesine göre ücret ödenecektir. E-Kitaplar tarafımızdan yayınlandıkça üyelerimize ayrıca eposta ile iletilecektir.

Saygılarımızla

Elektrik Mühendisleri Odası
42. Dönem Yönetim Kurulu

EMO YAYIN NO: EK/2011/2

TMMOB Elektrik Mühendisleri Odası
Ihlamur Sokak No:10 Kat:2 Kızılay/Ankara
Tel: (312) 425 32 72 Faks: (312) 417 38 18
http:www.emo.org.tr E-Posta: emo@emo.org.tr