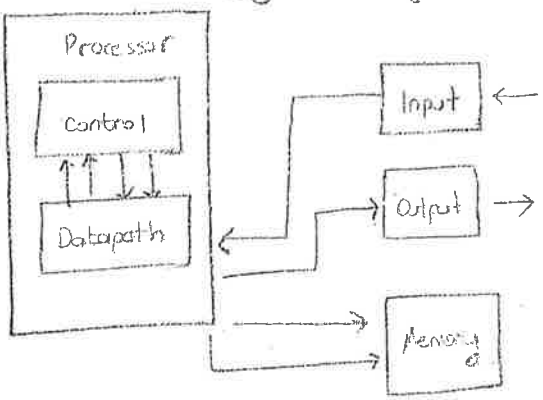
Elif ÇAKMAK :)

BOT

Kıymet

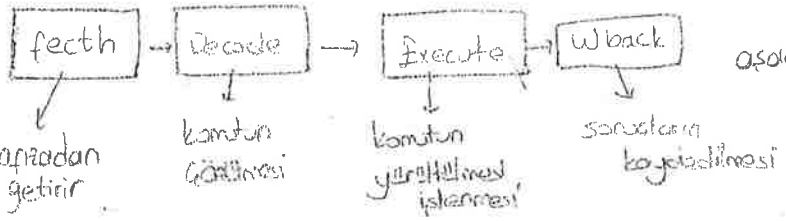
Bilgisayar Dili :

Bir bilgisayarının işlemesi büyük bir lojik devredir



=> Assembler
=> pipeline (3 aşamalı)
=> RAM + ROM
=> R tipi datapath (multicycle kontrol)

İşlemci işlemi :



aşamalarında gerçekleştirir.

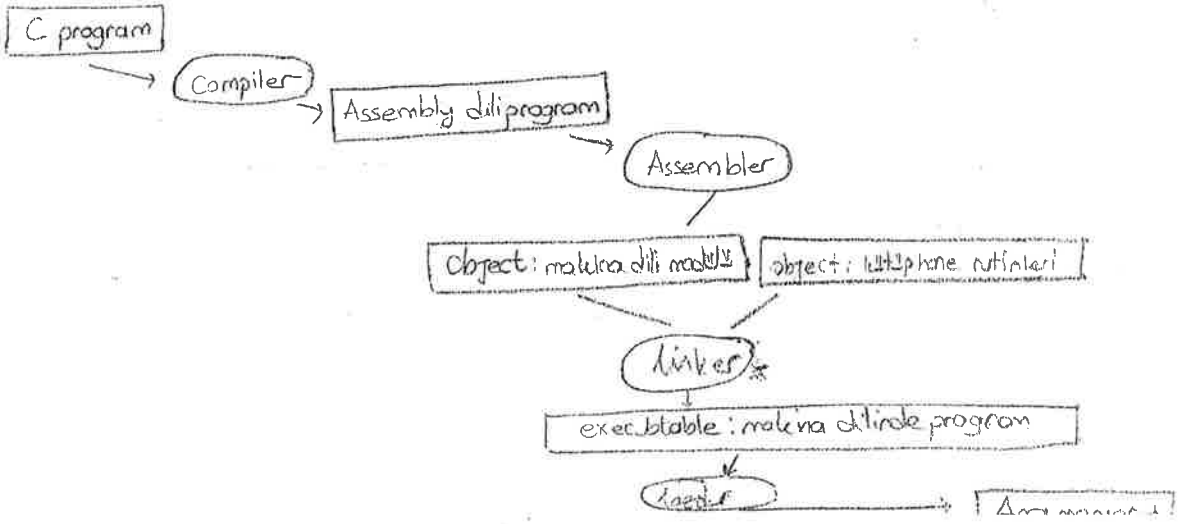
KOMUT SETİ MİMARİSİ (ISA) : Bilgisayarın en önemli soyutlamasıdır.

Yazılım ve donanım arasında bir arayüzdür

Uygulama, işletim sistemi, derleyici → Yazılım
——————— ·····→ (ISA)
Lojik, disk makinaları, giriş-çıkış sis. devreleri vb. } → Donanım ✓

=) Bilgisayar dilinin kelimelerine komut denir.

Hafızalanmış program konsepti :



ISA'nın temel prensipleri:

① Basitlik düzenlilikten yaradır. (komutların belirli düzene göre işlenmesi çözümü basitleştirir)

② En küçük en hızlıdır (işlemlerin küçüklüğü hızı artırır.

③ Çok kullanılanları hızlı yapın.

④ İyi tasarım, iyi tavizler sonucu oluşur.

Donanımın işlenenleri (operandlar):

→ Yüksek seviyeli programlama dillerinden farklı olarak MIPS'te her bir satır sadece bir komutun icra edilmesi içindir.

→ Aritmetik komutlarda işlenenler kaydedilmelidirler.
(Bunun için 32 register yeterlidir)

→ MIPS'te her bir register 32 bitliktir ve 32 register vardır. (MIPS-32'dir)
MIPS-64 ise 32 adet 64-bit register anlamındadır.

Hafıza işlenenleri:

Ana hafıza; yüksek s. programlama dili komutlarının verilerinin saklandığı yerdir.
Registerler; MIPS komutlarının çalıştırıldığı aritmetiksel işlemlerin yapıldığı datayı bulundurur.

✱ Ana hafızadan veriler, registerlere MIPS'in içerdiği transfer komutlarıyla çağrılır.

(lw, sw)
↙ ana hafızadan registere getirir
→ registerden hafızaya getirir.

ana hafıza; veri ile registerleri ilişkilendirmek için derleyici bellekte yer ayırır.

✱ transfer komutları en uzun zaman alan komutlardır.

→ MIPS 32 bitliktir ancak 4'er bitlik bilgi saklayan hafıza birimlerinden oluşur.

✱ Sabitlerle işlem için ise; sabitler program yüklenirken ana hafızaya yüklenirler.
↙ iş yaparak için; addi komutu kullanılır.

Hafıza Adresleri:

→ Compiler veriyi hafızada organize eder.
→ Compiler; sw/lw komutlarının gereği için hafızanın adresini belli bölgelere ayırıp kullanır.

Immediate operands: komutlar sabit giriş değerinde ihtiyaç duyar.

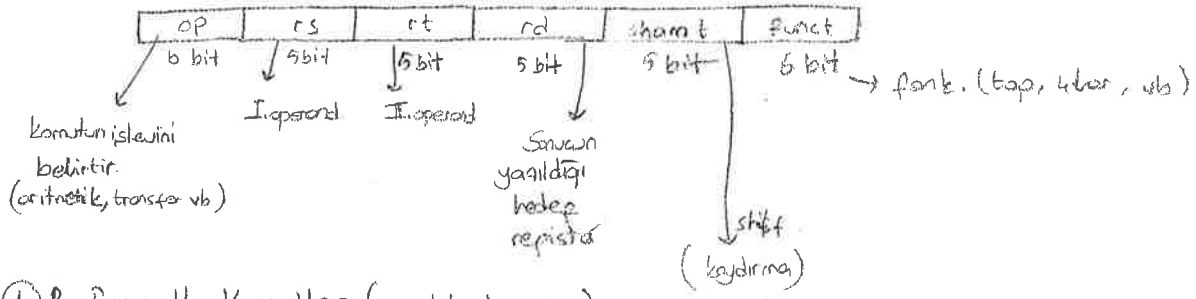
addi $s0, $zero, 1000 → registerlere sabit yazımı bu şekildedir.

✳ Hafızadan registere transfer komutunun formatı:

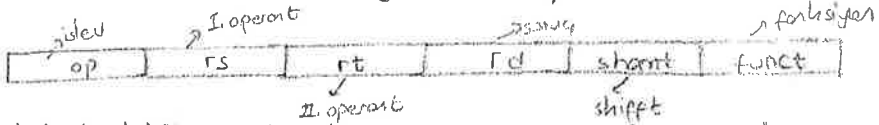
lw $t0, 8 ($t1)

$t0 → hedef
$t1 → kaynak
8 → parantez içindeki registerin içeriğine eklenen sabit değer

MIPS assembler komutunun Makina komutuna dönüşümü:

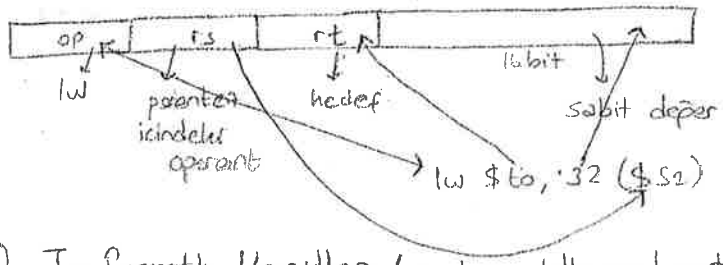


① R-Formatlı Komutlar (registerler için)



registerlerdeki işlem komutlarını gerçekleştirmek için kullanılır

② I-Formatlı Komutlar (sabitler veya veri transfer komutları için kullanılır)



③ J-Formatlı Komutlar (şartsız dallanma komutlarıdır)

↳ Hiçbir registerle işlem yapmaz. 26 bitlik kısmı atlayacağı adresi söyler

k) add ve sub komutları → 3 register ister

k) addi, lw, sw → 2 register ister

Lojik işlemler, operatörler:

(sll komutu → sola kaydırma / srl komutu → sağa kaydırma)

✳ (Bir bit sola kaydırmak sözdeğeri 2 ile çarpılmasıdır)

and $t0, $t1, $t2
andlayıp !

or $t0, $t1, $t2
orlayıp yazılacak!

mips'te;
NOT işlemi yerine NOR komutu kullanılır!

(and ve or komutlarıyla da işlem yapılabilir)

Karar Komutları :

① Branch if equal (beq - eşitse dallan)

beq $t1, $t2, L1 => t1'in içeriği t2'nin içeriğine eşitse L1'e dallan

② Branch if not equal (bne - eşit değilse dallan)

bne $t1, $t2, L2 => t1'in içeriği, t2'nin içeriğine eşit değilse L1'e dallan.

Jal komutu : Ana programdan çağrılan bir programa dallanmayı söyler bize
Jal func.

Jr => prosedürün en son satırıdır. Jal bitmiştir artık nereye gidecekse.

Karşılaştırılanların eşit olması, olmaması testi.

Eğer 1. registerin içeriği 2. registerinin içeriğinden küçük ise 3. registe 1 set eder. (Slt komutuyla gerçekleşir.)

Sabit operantlarla karşılaştırma için ; slti komutu kullanılır.

Bilinmesi gerekenler :

Register 0 ($zero) ⟶ devamlı olarak 0 değeriyle yüklüdür.
Register 2-3 ($v0 - $v1) ⟶ prosedürün döndürdüğü (sonuç) değerlerle yük.
Reg 4-7 ($a0 - $a3) ⟶ prosedür için giriş argümanlarıyla yüklüdür.
Reg 8-15 ($t0 - $t7) ⟶ geçici register yük.
Reg 16-23 ($s0 - $s7) ⟶ değişken değerleriyle yüklü dr.
Reg 24-25 ($t8 - $t9) ⟶ daha fazla geçici rapistor

* Reg 28 ($gp) ⟶ global pointer
* Reg 29 ($sp) ⟶ stack "
* Reg 30 ($fp) ⟶ frame "
* Reg 31 ($ra) ⟶ dönüş adresi yüklüdür.

prosedürler; programı parçalara ayırıp dönüşümde kolaylık sağlıyor.
↳ sadece kendisi için tanımlanmış işi yapabilirler.

MIPS'in prosedürler için kullandığı registerlerin isimleri;

$a0 - $a3 (saklandığı yer)
$v0 - $v1 (prosedürün döndürülen değerleri için) / $ra (gidilecek dönüş adresini bilmek ve bir tane dönüş adres registeri)

→ Yürütülmekte olan komutun adresinin saklandığı özel bir register vardır. Buna PC (program counter) diyoruz

⊛ Jal komutunun prosedür kısmı, çağrılacak prosedürün adresidir.

→ Jal komutu; $ra registerına geri dönüş komut adresi olarak PC+4 adresini set eder.

⊛ Stack pointer ($sp): Stacktaki en son işlenin adresini işaret eder.
- registerler s ile başlıyorsa; korunup stack'a atılabilir.
- " t ile " ; bunlar korunmayabilir.

Özet olarak:

* Jal komutu prosedüre dallanmak için kullanılır, PC içeriğine 4 ekleyerek geri dönüş adres registerına ($ra) yazar
* Argümanlar $a0 - $a3 reglerine, geri dönüş değerleri $v0 - $v1 reglerine atanır.
* Her bir prosedür, yerel değişkenler için bir hafıza yerine ihtiyaç duyar (Stackta yer ayrılması)

Stack: prosedürler için ihtiyaç duyulan organize edilmiş hafıza bölgesidir.
↓
ilk giren son çıkar.

Stack'da yer ayrılması:

Frame pointer: yerel değişkenleri ve kaydedilen registerlerin yerlerini gösterir.

$fp registeri, stacktaki kaydın başını, $sp ise sonunu gösterir.
↓ ↓
yürütülmesi boyunca değişmez ise değişebilir.

Global pointer: Ana program ve prosedürler tarafından kullanılır.

Karakterle Çalışma için temel komutlar:

⇒ Bazı komutlar, 32 bitlik bir kelimeden 8 bitlik bir kelime elde edebilir.
bunlardan lb ve sb transfer komutları 32 bitlik kelime yerine 8 bitlik kelimeleri transfer edilebilir.
MIPS bunu destekleyebilir.

* lb (load byte) komutu; ana hafızadaki 8 bitlik veriyi registerin en sağdaki 8 bitine yükler
* sb (store byte) " ; " " " " " " " en soldaki " "

⇒ MIPS komut seti half-word (16 bitlik) yükleme, depolama ve transfer işlemlerini de destekler.

lh (load half): hafızadan bir halfword'ü registerin en ağırlıksız 16 bitine yükleme işlemini yapar.

sh (store half): " " " en ağırlıklı " " "

Dallanma Adreslemesinin önemi
atlamada

① PC = register + dallanma komutundaki adres (32 bitlik) ⇒ relative adreslemedir.

② PC = PC + dallanma komutundaki adres ⇒ PC-relative adreslemedir. (Genellikle bu kullanılır)

PC = (PC+4) + dallanma komutundaki adres * 4

MIPS'te adresleme modları

① Register Adresleme: rs → operantım registerdir.

② Base veya yerdeğiştirme Adresleme: Komuttaki sabit değerle registerin toplanmasıyla elde edilir.

③ PC-relative adresleme: Komuttaki adresle, PC değerinin toplanmasıyla elde edilir.

④ Pseudo direct adresleme: 26 bitlik atlama adresi — PC ile bulunur.

lui → 16 bitlik sabiti, registerin en ağırlıklı 16 bitine kaydeder.

Bilgisayar Aritmetiği : ⑦

H.K.B ♡

MIPS ←

lb → 32 bitlik işaretli sayının en anlamlı 24 bitini işarete göre doldurur

lbu → işaretsiz sayılarda anlamlı bitlere 0 atar

lh → işaretli sayılarda en anlamlı 16 biti işarete göre doldurur.

lhu → işaretsiz "

slt → işaretli sayılarda karşılaştırma

slti → işaretli tamsayılarla "

sltu → işaretsiz sayılarda karşılaştırma

sltiu → " tamsayılarda "

(*) İkilik tabanda pozitif işaretli 16 bitlik bir sayıyı 32 bit ile ifade etmek için anlamlı bitlerine 0 ekleyecek!

(**) İkilik tabanda negatif işaretli 16 bitlik bir sayıyı 32 bit " " " anlamlı bitlerine 1 ekleyecek!

MIPS ←

- Add, Addu
  - işaretli sayılarda toplama
  - işaretsiz " "
- Addi, Addiu
  - işaretli sayılarda bir tamsayıyla toplama
  - işaretsiz " " " "
- Sub, Subu
  - işaretli sayılarda çıkarma
  - işaretsiz " "

Performans Değerlendirilmesi ve anlaşılması :

Respons time (→ cevap zamanı), Latency (→ gecikme) ve Execution (Elapsed) Time (yürütme zamanı) :

Respons time / Latency: programın son ve başı arasında harcanan zaman

Execution (Elapsed) Time: bir işi tamamlamak için harcanması gereken toplam zaman

Bilgisayarcı bu zaman kısımlarının hızlı çalışması ve erken cevap vermesini ister!

performans ile response zamanı ters orantılıdır

(yani performansı arttırdığım sürece cevap zamanım azalır)

Throughput (Veri-çıktı): Verilen bir zamanda yapılan iştir.
(sabit bir sürede)

* Eğer yeni bir işlemciyle makinayı upgrade yaparsak ne artar?
(Response zamanı düşer ve verimlilik artar)

⊕ Eğer lab.'a yeni bir makina eklersek neyi artırırız?
- Sadece verimlilik (birim zamanda yapılan iş) artar.
- İş yapılma zamanı değişmez sadece toplam iş artar.

CPU time: Giriş-çıkış ve başka programların harcadığı zaman değildir.
CPU nun programı için hesaplamaya harcadığı zamandır.

Kullanıcı CPU zamanı: Bir program için CPU nun harcadığı zamandır.
Sistem " " : Bir programın çalışması sürecinde işletim sisteminin harcadığı zamandır.

⇒ Verim ve Execution (yürütme) time):

$$\text{Performans} = \frac{1}{\text{Execution time}}$$

$$\text{CPU execution time} = \text{CPU clock cycle} \times \text{clock cycle time}$$

→ bir komuttan diğerine geçerken ki zaman (periyot)

yürütme zamanı yerine cycle kullanırız

$$\text{clock rate} = f = \frac{1}{\text{clock cycle time}} = \frac{1}{T} \qquad 1\ \text{Hz} = \frac{1\ \text{cycle}}{\text{second}}$$

⊛ CPU işi bitirmek için kaç clock zamanı sonucuna cevap = response zamanıdır.

$$\text{Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{clock cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{clock cycles}}$$

performansı geliştirmek için;
① Bir program için gerekli cyclelerin sayısının azaltılması
② Clock cycle time (periyot) azaltılması
③ clock rate artırmak

↓
ancak frekans artırmak tehlikeli bir iştir çünkü yüksek ısı ortaya çıkıp devre ısınır

$$\text{CPU time} = \frac{\text{clock cycle}}{\text{clock rate}} \rightarrow \text{yürütme zamanı} \rightarrow \text{işlemci hızı (GHz)} \rightarrow \text{frekansı}$$

⊛ Komutların sayısını clockların sayısına eşit kabul edebilir miyiz?
(Kabul edilmez çünkü; farklı komutlar farklı makinalarda farklı zamanlar alırlar)

Ortalama Clock Cycle (CPI): Bir program için bir komut başına düşen clock cycledir (yürütme zamanıdır)

CPU clock cycle = komut sayısı × CPI

⊛ Bir programda yürütme zamanı komutların sayısına bağlıdır.

Yürütme zamanı için başka bir yolda;

CPU execution time = Komut sayısı × CPI × clock periyotu
(CPU clock cycle)

Yani;

CPU execution time = CPU clock cycle × clock periyodu

⇒ performans yürütme zamanı tarafından belirlenir.

performansa eşit olanlar ise;

- cycle'leri sayısı ⊖ / - komut sayısı ⊖ / - frekans ⊕ (saniye başına düşen cycle sayısı)
- CPI ⊖ (komut başına düşen cycle sayısı) / - MIPS ⊕ (saniye başına komutların ortalama sayısı)

Performans değerlendirilmesi

⊛ Performans değerlendirmek için yalnızca komutları kıyaslamak yanıltıcı olabilir.

⊛ İki bilgisayarı performans = değerlendirmek için; komut sayısı, CPI, periyotu dikkate almak gerekir.

⊛ Bileşenler arasında eşit olan varsa; özdeş olmayanlar kullanılır.
(Örneğin frekans eşit ise; komut ve CPI değerlendirilir)

Benchmarks : Bir ölçme veya değerlendirme standratıdır.

workload : tanımlı bir işlemler kümesi pragramı.

(*) Eğer aynı komut setine ait bilgisayarlar kullanılırsa; en hızlısı en yüksek clock rate (frekans) sahip olacaktır.

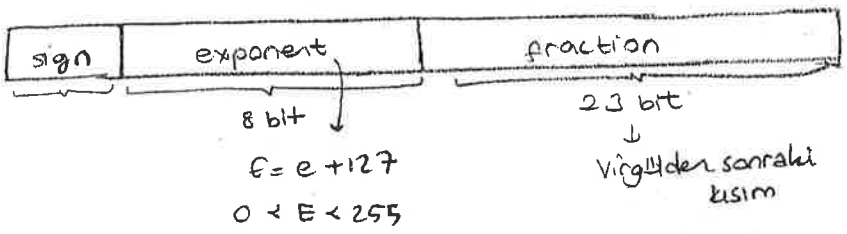
SPECwebog : verime odaklanır. /çoklu işlemciler sık sık kullanılır.

FLOATİNG POİNT SAYILAR #

$(-1)^{sign} * significand * 2^{exponent}$

exponent : 32 bit için bias değeri 127

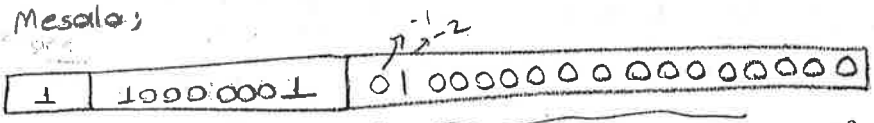
64 " " " " 1023



eğer virgülden sonra sayı 23 bit değilse 0 larla 23 'e tamamlayarak toplamda 32 bit olacak.

İkilik tabandaki floating point sayıyı decimal hale dönüştürmek için;

Mesala;



$(-1)^1$

(129) exponent alanı

fraction alanı $\Rightarrow 1 \times 2^{-2}$

129 - 127 = 2

$(-1)^1 \times (1 + 1 \times 2^{-2}) \times 2^2$

Datapath ve Kontrol :

- Aritmetik - Lojik komutlar = add, sub, and, or, slt
- Hafıza tabanlı " = lw, sw
- Kontrol - akış " = beq, j

op, rs, rt, rd, shamt
6, 5, 5, 5, 5, 6 → funct
R-formatlı

(op, rs, rt, ofset
6, 5, 5, 16)
I - formatlı (transfer)

J - formatlı (atlama)
(6, 26)
op, address

* Bir komutun işleme sürecinde kullanılan donanımların bütününe datapath denir.

İşlemcilerde gerçekleştirme stilleri

① Tek cycle (Single cycle) :

- Her komut 1 clock cycle'de icra edilir.
- Komutları ardışıl olarak işler ve bir komut bittikten sonra diğeri işler.

Mesela ; (7ns), 3ns, 5ns
↓
single cycle bu kabul edir

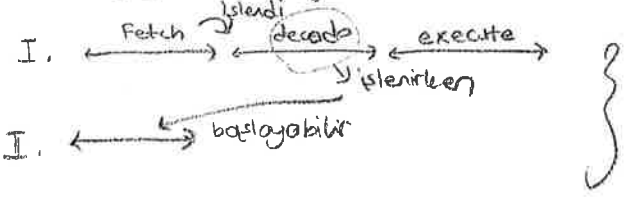
Ve S.C'in kötü yönlerinden biride budur;
- En büyük periyot kabul edilir.
- Çünkü büyük periyot ilk iş olsa bile onun bitmesi beklenmelidir.

② Çoklu (Multi) Cycle :

- Fetch / execute yapılır
- Her bir clock cycle'de bir adım icra edilir.
- Her komut ihtiyacı kadar cycle kullanır.
(zaman israfı yoktur.) Yani ; 7, 5, 3 arasında 3 cycle gerekiyorsa 3 cycle kullanılır.
- Single göre daha hızlıdır.

Yine bir dezavantajı ise ; bir komutun işlenmesi bitmeden diğerine geçilmez.

③ Pipeline : - Her komut çoklu adımlarla yürütülebilir.
- Her Clock cycle bir komut icra edilir.



adımların tek tek bitmesini beklenmez.

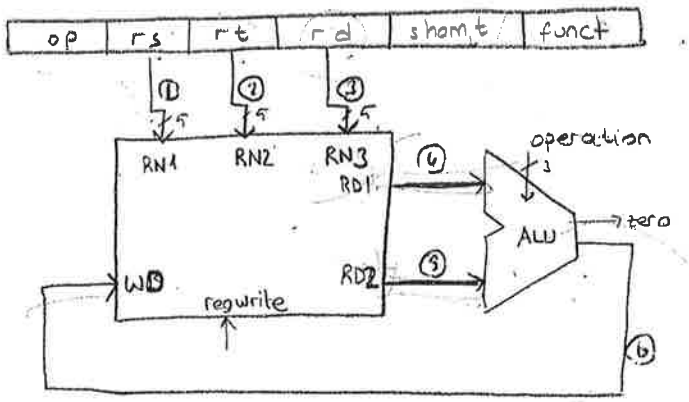
latch → her türlü bilgiyi her şekilde her zaman alır.

flip flop → sadece belirli clock zamanlarında alır bilgiyi.

MIPS'in tek cycle'da gerçekleştirme:

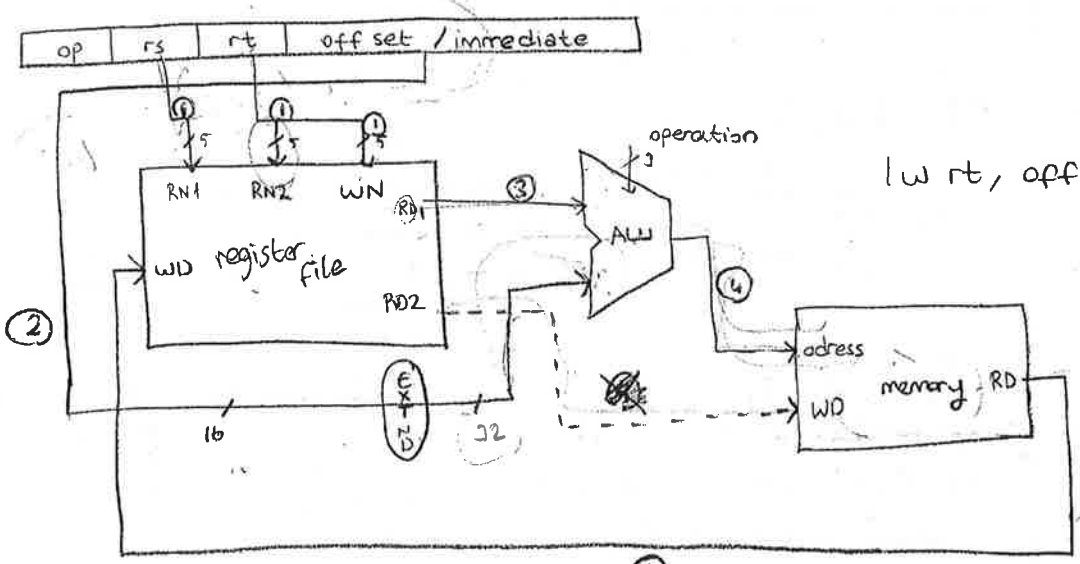
(RD → read data
WD → write data)

DATAPATH : R-tipi Komut



add rd, rs, rt //

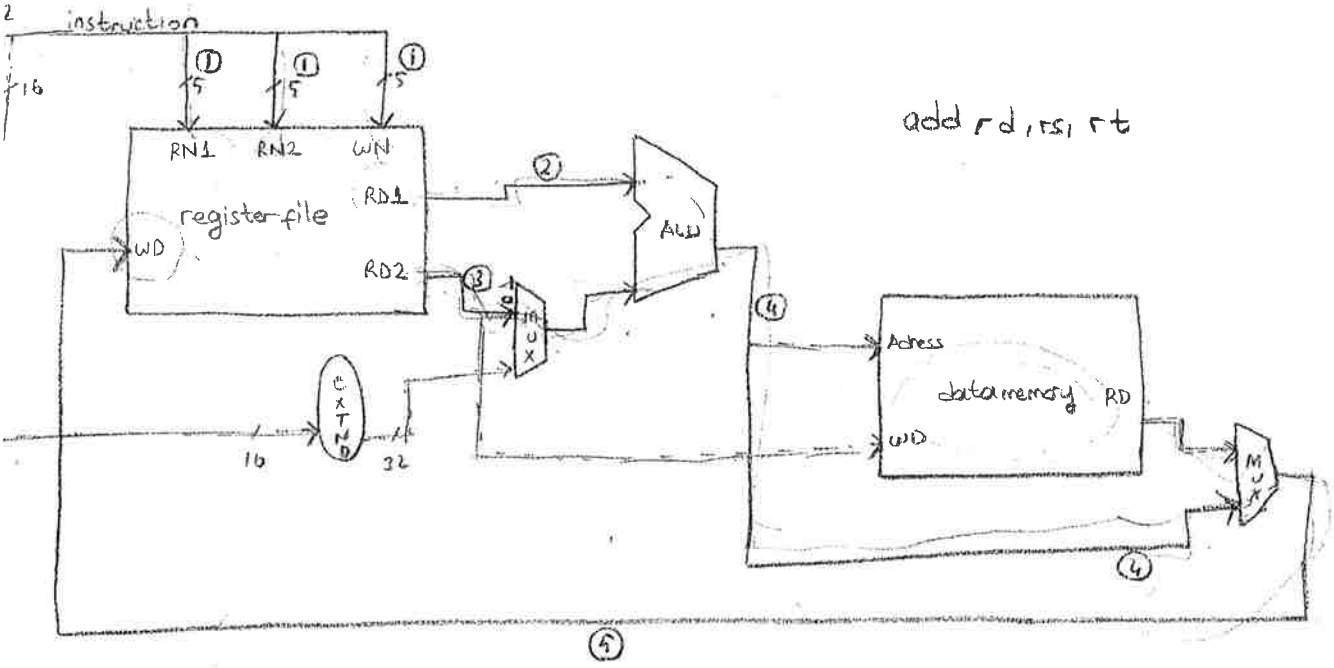
DATAPATH : Lw komutu (register da bitiyor işlem)



lw rt, offset (rs)

(hafızadan geri alarak registere kaydetti)

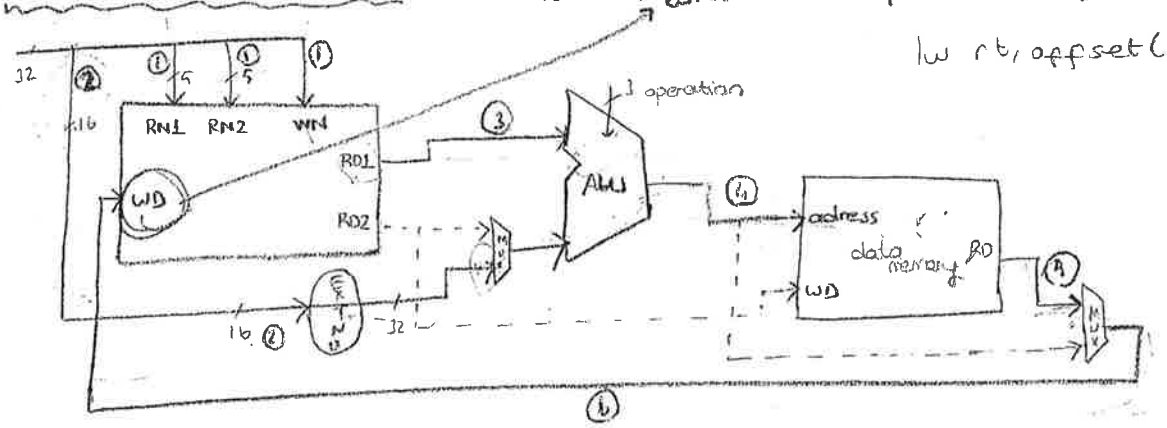
R- tipi komutun Calismasi :



Load komutunun calismasi :

lw'de buradaki wd'ye yaziyor !

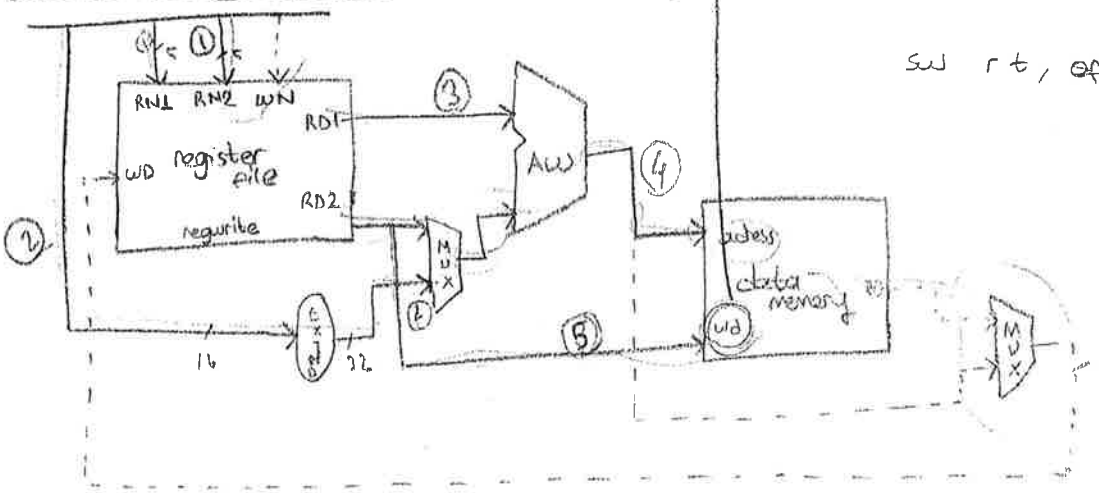
lw rt, offset(rs)



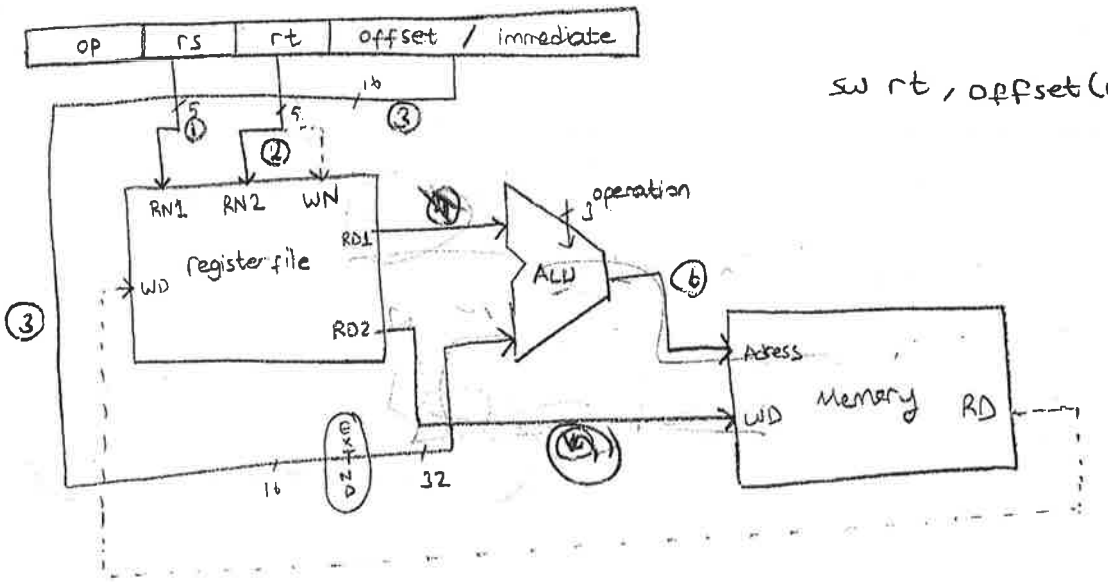
Store komutunun calismasi :

sw'de wd'ye yaziyor !

sw rt, offset(rs)

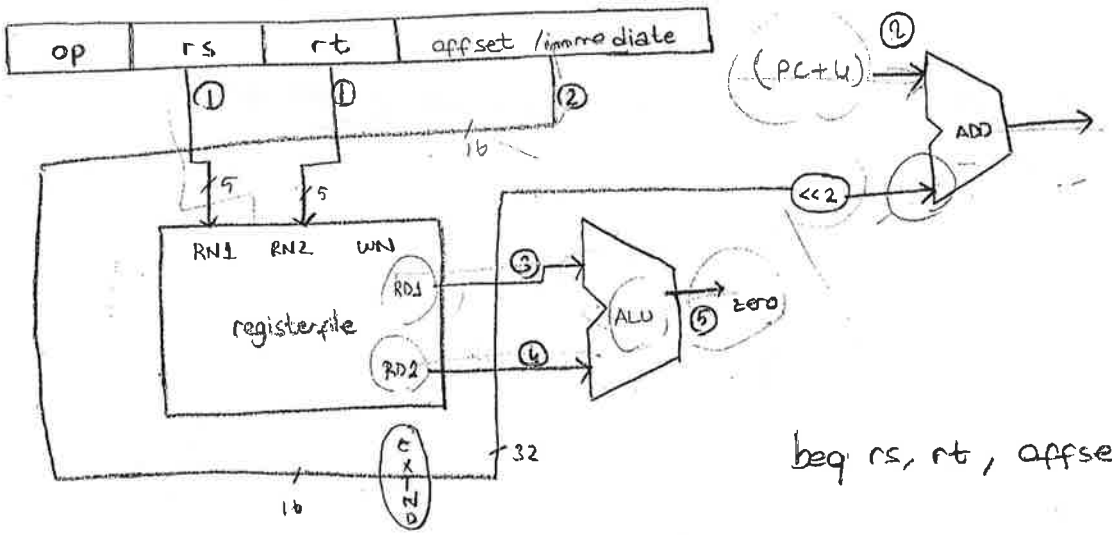


DATAPATH: Sw komutu : (Memory'da biten işlem)



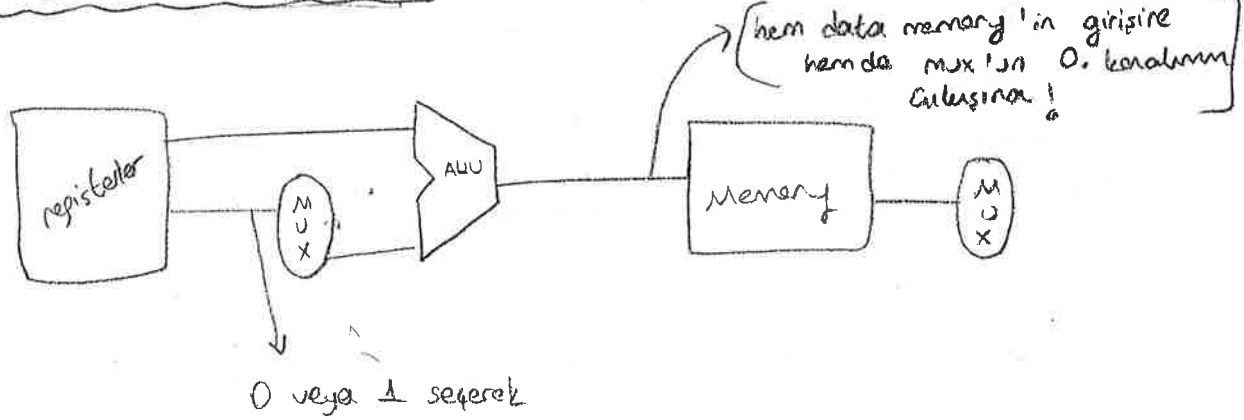
sw rt, offset(rs)

DATAPATH : dallanma komutu



beq rs, rt, offset

MIPS veriyolu I : Single Cycle (r tipi, lw, sw, oluşturmak için 2 mux kullanılır.)



→ ilk kullanılan add'de ; ALU işlemleri ve PC artımı aynı clock cycle'da paralel top. reg. gelir.

MIPS veriyolu II : single cycle : Komutlara ; fetch komutunu ekleme.

MIPS veriyolu III : single cycle (dallanma yeteneğinin ve diğer mux'un eklenmesidir)

Branch işlemi için şart sağlanmıyorsa ; doğrudan PC+4'e dallanmak için yeni mux eklendi.

(*) Single cycle gerçekleştirmede veri, komut boyunca depolanmaz. Yalnızca kombinasyonel lojik devreyle taşınır.

- Dallanma komutu (beq) varsa PC src 1
  yoksa PC src 0 dır.

~~[illegible]~~

DATAPATH, Kontrol :

ALU kontrol planı = ALU kontrol birimi, ALU kontrol birimi girişine 2 bit (ALUop kontrol) gönderir. " " " girişine 6 bitlik funct bitleri uygulanır. Çıkışta, ALU kontrolü için 3 bitli olur.

Alu kontrol alanı		fonk
000	→	and
001	→	or
010	→	add
110	→	sub
111	→	slt

Kontrol birimi devre şeması ;

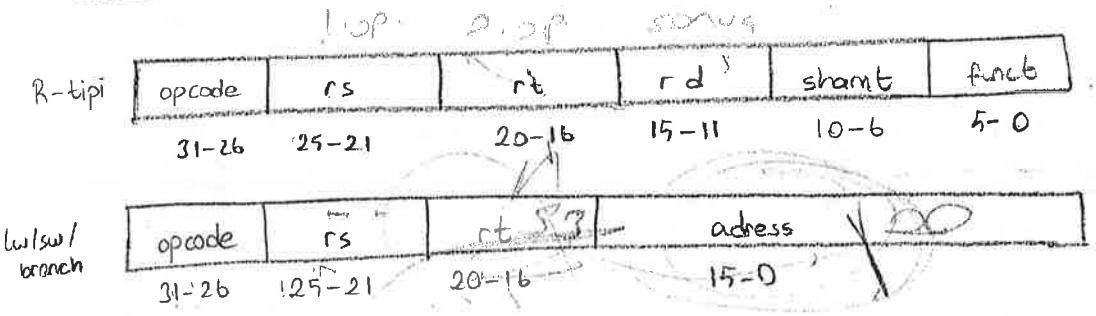
inputlar = 0
Output ⇒ RegDst = 1 , diğerleri 0
RegWrite = 1
ALU op 1 = 1 (dallanma için)

(*) Load/store için add ALU op 00

(*) dallanma için sub ALU op 01

(*) and, or, add, sub, slt, R tipi komutlar için ALUOp 10

Ana Kontrol Birimi Tasarımı



- opcode daima 31-26 bitlerindedir.
- Okunabilir 2 register daima rs(25-21) rt(20-16) dadır.
- Dallanma veya lw/sw için 16 bit ofset daima 15-0 bitleridir.
- Load için hedef register 20-16(rt) bitlerindedir. R tipinde ise 15-11 (rd) bitleridir (bunun için mux ile seçmek gerekliliktir)

KONTROL SİNYALLERİ

DATAPATH ve Kontrol - I

	Lojik 0	Lojik 1
Reg Dst =>	Yazma reg. için hedef reg. sayısı rt alanı (20-16)	Yazma reg. için hedef reg. sayısı rd alanı (15-11)
Reg Write =>	—	Write reg. girişindeki reg. write data input girişindeki değere yazılır
ALU src =>	2. ALU operandı 2. reg. file çıkışından gelir (read data 2)	2. ALU operandı komuttan düşük 16 bitine işareti genişletilerek yazılır.
PC src =>	PC toplayıcının çıkış tarafından değiştirilir bu PC+4 ile hesaplanır	PC toplayıcının ... değiştirilir dallanmayla hesaplanır
Memory-Read =>	—	data memory içeriği giriş adresine göre belirlenir (bu giriş ilk read data çıkışına konur)
Memory-Write =>	—	(bu giriş write data girişinin değeri tarafından değiştirilir)
Memory-Reg =>	ALU'dan gelen değeri Write data giriş reg. yaz	Data memory'den gelen değeri write data giriş reg. yaz!

— Bir devrimin sonunda elde edilen data bir sonraki devrimde kullanabileceğinden saklanmalıdır.

Multicycle da hafıza bir tanedir ancak 2 kısımdır.
- veri hafızası
- komut hafızasıdır.

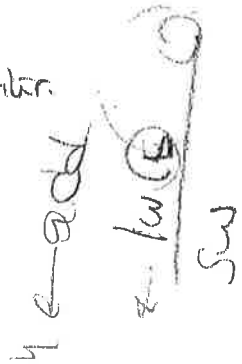
Tek bir ALU kullanılır.

PC → şuanki komut işlenirken bir sonraki komutu bulmak için şu anki adrese 4 ekleyip işlemeye başlar

R tipi add => 4 fazda
lw => 5 fazda
siw => 4 "
beq => 3 "
Jal => 3. "

adres 3. fazda elde edilir

store komutu 4 fazda bitirilir.



J tip komut için ilk 2 faz aynıdır

Komutların adımlara bölünmesi: → PC yi kullanır.

① Komutların getirilmesi ve PC'nin artırılması. (Fetch)
② " çözülmesi ve registere getirilmesi (Decode ve reg. Fetch)
③ yürütme, hafıza adres hesabı, veya dallanmayı tamamlaması. (Execution)
④ Hafızaya erişim veya R tipi komutun tamamlanması (Hafıza erişim / Memory access)
⑤ Hafızaya okumanın tamamlanmasıdır. => (lw komutu kullanır sadece)

en uzun komut 5 önemli fazda gerçekleşir.

* Her komut kendi süresince işler.

lw $s1, 20($s2)

DATAPATH ve kontrol II

Kontrol girişi 6-bit komut opcode alanıdır. Çıkış 7, 1-bit sigral, 2-bit ALUOp sinyalidir.

R Tipi:

RegDst, RegWrite → 1

ALUsrc, Mem-Reg, Mem-read, Mem-write, PCsrc → 0

operation → (çarpa 1, diğer 0)

lw komutları:

ALUsrc, Reg-write, Mem-read, Mem-reg → 0

reg Dst, mem write, PC src → 0

operation → 0 1 0 (alttalwler 1, üsteliler 0)

sw komutları:

ALU src, Mem write → 1

reg write, mem read, PC src → 0

reg Dst, Mem Reg → X

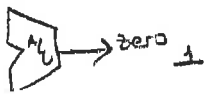
operation → 0 1 0

beq komutları:

reg Dst, Mem Reg → X

rep write, ALU src, Mem read, Mem write → 0

operation 1 1 0



MULTİCYCLE UYGULAMA #

((Her bir saruç diğer işlemin girdisi olcak. Her bir faz için tek fonksiyonel donanım kullanılsın!))

- Komutlar adımlara bölünür.
- Her bir adım bir tek clock cycle'da yapılır.
- " " adımda yapılan iş miktarı yaklaşık olarak birbirine eşittir.
- " " cycle, her önemli fonksiyonel birimi en fazla 1 kez kullanır. böyle birimler çoğaltılacaktır.
- Fonksiyonel birimler, aynı komutun farklı çevrimlerinde kullanılır.

Multicycle yürütme adımı (3) Atlama Tapısı :

→ PC (jump adres), IR'nin, A ve B'nin, ALU out'un oluşturması
→ Register file dosyasına RN1, RN2, WN' nin verilmesi
→ Register A,B ye komutların verilmesi
→ yapıların geri döndürerek PC 'ye gönderilmesidir.

④

Multicycle yürütme adımı (4) Hafıza erişim-Okuma (lw) :

→ PC'nin, IR'nin, A,B'nin, ALU çıkışının oluşturulması
→ ALU çıkışından PC+4' e komutun gelmesi ve ana hafızada bulunması
→ Ana hafızanın RD çıkışından MDR (memory data) ye gelmesi
→ Komutun MDR (memory data) mevcut olmasıdır.

Multicycle yürütme adım (4) Hafıza erişim- yazma (sw) :

→ PC, IR, A ve B, ALU out oluşturulması.
→ ALU out 'un çıkışından PC+4 'e gelmesi, orden hafızaya girişi
→ B registerinin çıkışının hafıza daki wd' ye verilmesi
→ Hafıza komutun mevcut olmasıdır

Multicycle yürütme adım (4) R tipi :

→ PC, IR, A ve B, ALU out 'un oluşturulması
→ IR'den yapının register dosyasından WN' ye verilmesi
→ ALU out 'un çıkışının register file' daki WD' ye verilmesi
→ Register içinde komutun mevcut olmasıdır

⑤ Multicycle yürütme adımı (5) Hafızanın tamamlanıp okunması (lw) :

→ PC, IR, MDR, A ve B, ALU out 'un' oluşturulması
→ IR yapısından WN' ye giriş, MDR'nin çıkışından WD' ye giriş olması
→ Son olarak komutların register dosyasında mevcut olmasıdır.

Multicycle Datapath Kontrol I

Ⓐ PC 'den sonraki ilk mux 'un görevi ; ya komut okuyacak yada veri yazıp okuyacak.

Ⓑ B registerinin çıkışındaki mux 'un göreve ise (branch komutlarını gerçekleştirir.)

→ 0. kanal : B registerinin içeriğini → 2. kanal = şartlı dallanma-yı
→ 1. kanal : PC+4 işlemini yapar → 3. " = şartsız "

① Multicycle yürütme adımı (1) Fetch (Komutun getirilmesi) :
fetch fazındaki işlemler

→ PC'un hafızaya getirilmesi
→ PC+4'ün ana hafıza bulunması
→ Hafızanın RD (okuma veri sinyali) alıcı komutun IR'a verilmesi
→ PC+4'ün ALU bir girişine getirilmesi (ilk değişken)
→ 2. değişkenin ALU'ya verilmesi
→ ALU'nun çıkışından aldığını geri döndürerek PC ye vermesidir.

② Multicycle yürütme adımı (2) Decode ve Register fetch :
↳ (fazında yapılan işlemler)

→ PC'nin ve IR'nin oluşturulması,
→ IR'den çıkan yapının register dosyasındaki RN1, RN2 ye getirilmesi ve işlem yapacağın B registerine getirilmesi
→ PC+4 'den gelen komutun register A 'ya getirilmesi
→ işlemlerin ALU'da yapılıp ALU çıkışa verilmesidir.

③

Multicycle yürütme adımı (3) Hafıza referans yapısı :

→ PC'un düşürülmesi ve register A, B oluşturulması
→ IR'nin yapısı register B nin çıkışına verilmesi
→ A, B nin ALU'da işlemlerinin yapılıp ALU çıkışına verilmesidir.

Multicycle yürütme adımı (3) R tipi :

→ PC'nin ve register A, B nin düşürülmesi
→ IR yapısının oluşturulması
→ A, B registerlerinden gelenlerin ALU'da işlem yapılıp ALU çıkışına verilmesidir.

Multicycle yürütme adımı (3) ~~dallanma~~ branch yapısı :

→ PC ile register A, B nin düşürülması (A ve B operantları birbirine eşitse ALU çıkışındaki zero 1, değilse 0 olur)
→ işlemlerin ALU'ya verilmesi
→ IR'nin oluşması
→ ALU çıkışına verilen komutun çıkışının döndürülüp PC'ye yazılmasıdır.

Multicycle kontrol adımı (4) Hafızaya erişim – okuma (lw) :

Multicycle kontrol adımı (4) Hafıza erişimi – Yazma (sw) :

Multicycle kontrol adımı (4) ALU yapısı R tipi :

Multicycle yürütme adımları (5) Hafızadan dönmenin tamamlanması (lw) :

(Moore tipi → çıkış sadece o ana bağlıysa
Mealy tipi → çıkış o an + girişe ")

Kontrol sinyallerinin değeri ; hangi komutun yürütüldüğüne ve
hangi adımın icra edildiğine bağlıdır.

Özet Olarak ; ① Multicycle datapathler single cycle'a göre 2 büyük avantaj sağlar
→ Fonksiyonel birimler bir tek komutta yeniden kullanılabilir.
→ Daha kısa execution path'ler ile komutlar birkaç cycle tüketilmesiyle daha hızlı tamamlanabilir.

Multicycle kontrol adimi (1) fetch :

—)
—)
—)
—)

Multicycle kontrol adimi (2) Decode ve Register Fetch :

Multicyle kontrol adim (3) Hafıza referans yapisi :

Multicycle kontrol adimi (3) ALU yapisi - R tipi :

Multicycle kontrol adim (3) dallanma yapisi :

Multicycle kontrol adim (3) atlama yapisi :

* Eğer; pipeline ve pipeline olmayan bir görevin tamamlanması için geçen zaman eşitse; pipeline toplam hızlandırması k (kesim sayı) kadar olabilir.

Pipeline işlemenin 2 temel uygulaması vardır

① Aritmetik işlemlerde Pipeline
② Komut yürütmede "

① Aritmetik işlemlerde pipeline :

* Floating Point işlemleri ve karmaşık işlemleri için önemli bir hızlandırıcı etki yapar.

* – Exponentlerin karşılaştırılması
  – Mantisanın hizalanması
  – Mantisanın toplanıp / çıkarılması
  – Sonuç normalizasyonu

(Her bir kesimde harcanan süre farklı olduğundan en kötü durum olarak en uzun clock cycle'ı alınır. Boşta kalmak sistemi yavaşlatır.)

* Kesim sayısı ne kadar fazla olursa sistem o kadar hızlı olur.

* Parçalanan komutlar eşit süreli olursa pipeline daha hızlı olur. (decode, fetch, vb.)

Dört segmentli (kesimli) Komutlar için Pipeline :

① FI : komutu getir ② DA : komutu çöz, etkin adresi hesapla
③ FO : veriyi getir ④ EX : komutu yürüt.

* Komut ve data hafızalarına ayrı ayrı erişildiği düşünüldü.

② Komut yürütmede pipeline :

MIPS'de pipeline neden kolaydır ?

* Bütün komutlar benzer uzunluktadır. (ve tüm komutlar için fetch ve decode eş sürelidir.)

* Yalnızca bir kaç komut formatları vardır. Basit komutlar tek adımda yapılır.

* Hafızada : operandlar belli bir sıraya göre dizili olduğundan erişim kolaydır.

* sw / lw komutlarıyla sadece erişilir

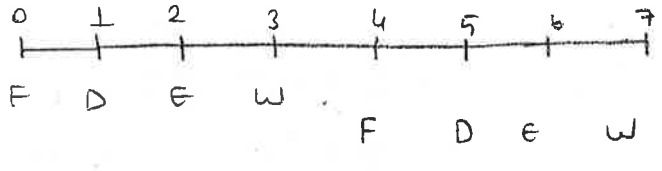
PİPELİNE (BORU HATTI)

Paralel işleme ve pipeline:

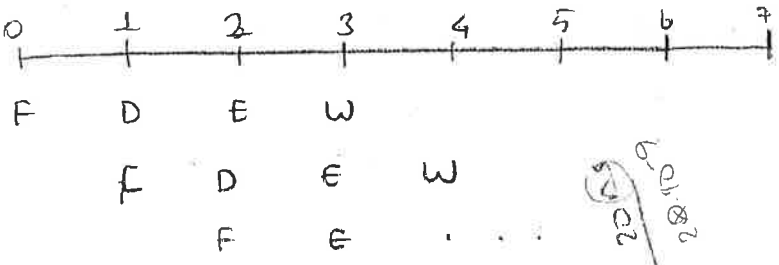
→ aynı zamanda birden fazla veya çok sayıda işlemin yapılabilmesi bilgisayar sisteminin hızını artırır.

→ paralel işlemde aynı anda birden fazla komut icra edilir.

→ parolel çalışma diğer bir örnekte; birden fazla işlemcisi bulunan sistemlerdir.



} pipeline olmaz!



} pipeline olur
(aynı zaman aralığından farklı komutlar devreye girmiştir)

⊛ Pipeline bir makina için;

k = kesim sayısı

tp = clock cycle

tk = n adet görevin bitirilmesi için gerekli süre

$$t_k = (k + n - 1) * t_p$$

⊛ Pipeline olmayan bir makina için;

n = görev sayısı

tn = her görevi bitirmek için gereken süre (clock cycle)

t1 = n " başarılıyla tamamlamak için gereken süre

$$t_1 = n * t_n$$

tüm görev için gerekli süre ise;

$n * t_p * k$ (clock)

(görev sayısı, kesim sayısı)

Hızlanma oranı S (speed up)

$$S_k = \frac{t_1}{t_k} \Rightarrow \frac{n * t_n}{(k+n-1) * t_p}$$

n görev sayısı artıkça n'e yaklaşır.

şu ortaya çıkar →

$$S_k = \frac{t_n}{t_p}$$

MIPS'de pipeline'ı neler zorlaştırır?

✱ Structural hazards (Yapısal tehlikeler): Pipeline işleminde, farklı komutlarda, farklı durumlarda, aynı donanım kaynaklarının kullanılması durumu oluşabilir. (Kaynak çakışması)

✱ Control Hazards (Kontrol tehlikeleri): Pipeline yürütülmesinde bir önceki komutun sonucuna göre karar vermesi gerekebilir. Genelde şartlı ve şartsız dallanma komutlarının işlenmesi durumunda önemlidir (dallanma tahmini.)

✱ Data Hazards (Veri tehlikeleri): Bir sonraki komut, yürütmesi bitmemiş bir önceki komutun datasına ihtiyaç duyabilir. (Mümkünse datayı ilet!)
(Veri bekletisi)

● Pipeline çalışan DATAPATH'i

⇒ Bir komutun tamamlanmasındaki 5 adım;

① Fetch yapısı ve PC'nin artırılması
② Decode yapısı ve Register okuma
③ Yürütme veya hesaplama adresi
④ Hafıza erişimi
⑤ Registerin içine sonucu yazma

} Hepsi birden single clock cycle'da yapılır.

✱ Extra donanımları kaynak şartıyla, bir yürütme işlemini birçok cycle'a bölersek nolur? (Her bir clock çevriminde yeni bir komutu işleyebiliriz.

NOT: Pipeline yürütmede her bir kesimde elde edilen sonuç bir sonraki kesime aktarılabilir. Aynı kesimde farklı komutlar için eş zamanlı olarak farklı datalar ile işlemler yapılabilir.
Dolayısıyla her kesim için bir pipeline register kullanılmalıdır.

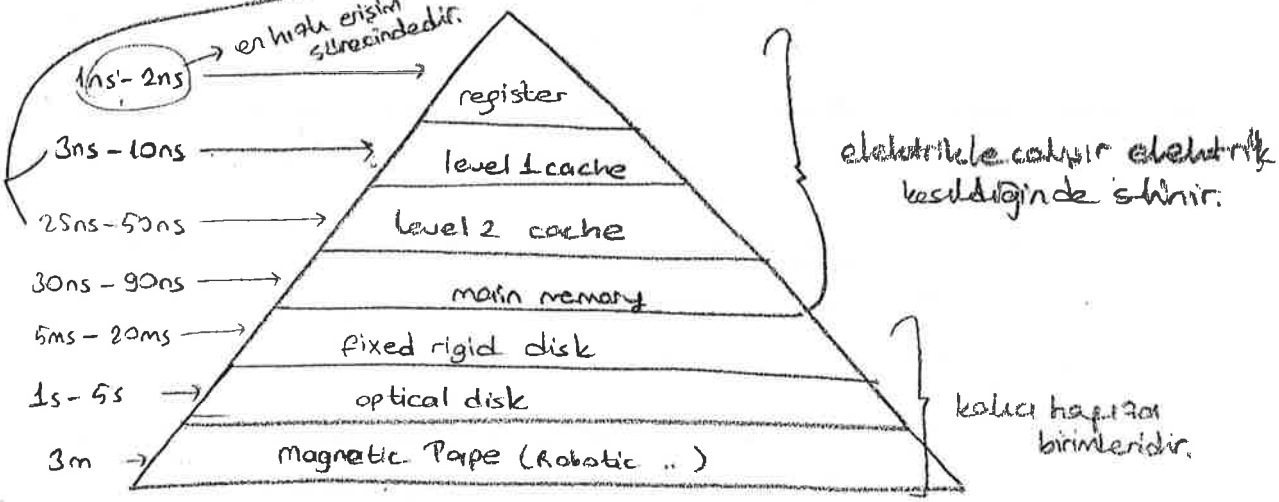
(!!) Önemli olarak; R tipi bir komutta, registera write back işlemi 5. pipeline kesimindedir. oysa multicycle 4. durumda gerçekleşir.
Nedeni; register dosyasına yazarken oluşan yapısal tehlikelerden olabilir.

⇒ İdeal pipeline (gecikme yoksa) için CPI=1'dir neden?

↳

HAFIZA HİYERARŞİSİ #

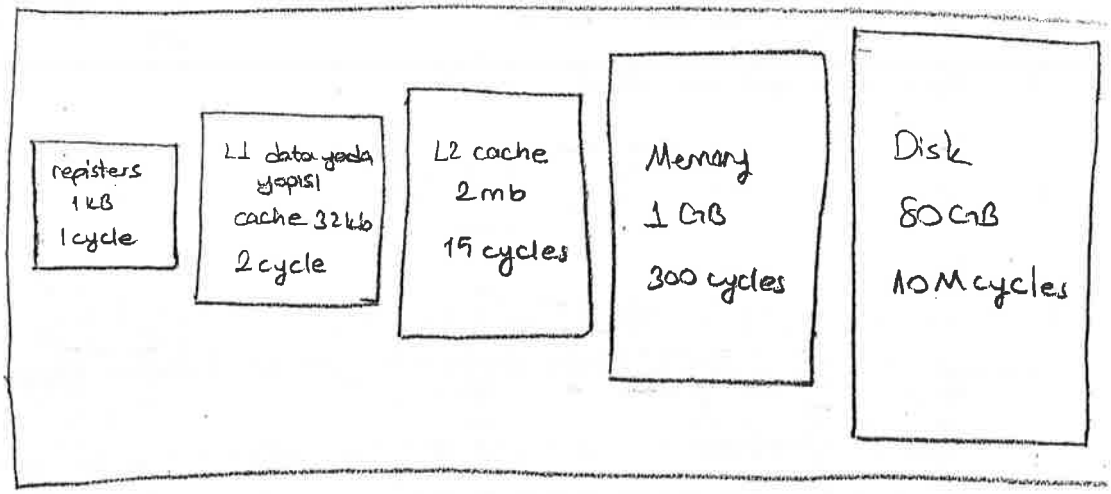
⊛ En hızlı ve en küçük birimdir mikroişlemciler. → daha hızlı işlem yapabilmek için kullanılır.



level 1 cache → kb'lar seviyelerinde (mikroişlemci yapısında kullanılır)
level 2 cache → mikroişlemci entegre içinde ancak yapısı içinde değildir (mb'lar seviyesinde)

⊛ Hiyerarşinin altında mekanik parçaları olan kalıcı HD, manyetik teyp vb. yapılar kullanır.

Hiyearşi listesi :
- registers
- L1 cache
- L2 cache
- main memory
- disk cache
- disk
- optical
- tape



Yarı iletken hafıza çeşitleri

⊛ Ana hafıza ve cache hafızalar yarı iletken hafızalardır.

⇒ Seri erişim paralel erişim registerleri shift registerleridir
(seriden → paralele, paralelden → seriye dönüşümleri sağlar.)

⇒ Statik ram, dinamik ram'e göre 10 kat hızlı çalışır ancak pahalı olduğun dan dinamik ram kullanılır.

Yarı iletken Bellek özellikleri

- Yazmaç - register : FF'lerin uygun şekilde birbirine bağlanmasıyla oluşur.
  Kalıcı değildir. Fiyat ve hızı çok çok yüksektir.

- SDRAM : 16 bitlik, 8 bitlik hafızaların saklandığı ff'lerdir.
  Kalıcı değildir. fiyatı çok yüksek, hızı çook hızlıdır.

- DDRAM : Geçici olarak hafızalarda saklama işini most transistörlerle yapıyor ff'lerle değil
  Kalıcı değildir. Fiyatı orta, hızı hızlıdır.

Okuma yazma var.

- ROM : Sadece okunabilir yapıdadır. Üretildikten sonra bilgileri değişmez.
  Kalıcıdır. Hızı çok hızlıdır. fiyatı düşüktür.

- PROM : Kullanıcıya ilk baş olarak gelir ancak içine sadece bir kez yazılım yazılır
  daha sonra okunur sadece.
  Kalıcıdır. Hızı çok hızlıdır. fiyatı ortadır.

- EPROM : Depolarca silinip yazılabilir bir bellektir (RAM'den farklı silmeden yazılamaz)
  Kalıcıdır. Orta hızlıdır. fiyatı da ortadır.

- EEPROM : Silme işlemi ancak elektriksel işlemle yapılabilir.
  Okuma-yazma vardır. Kalıcıdır. düşük hızlıdır. Yüksek fiyatlı.

program yazılırken çok sık kullanılan bir bellektir

RAM - SRAM - DRAM Bizim ilgileneceğimiz hafızalar;

Cache ve ana hafıza birimleridir. Bunlar yarıiletken yapılıdırlar.

kullanılır genellikle

verilerin, komutların geçici olarak saklandığı bellekleridir.

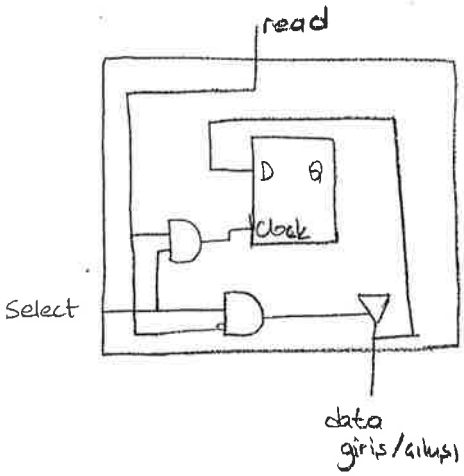
- İşlemci cache bellekle kelime bazında veri transferi yapar
- Ram'daki bilgi ise cache belleğe bloklar halinde transfer edilir.

- Register blok yapısında; adres giriş 0 olduğunda => data yolları açık devre görevinde olur ve değer vermez!

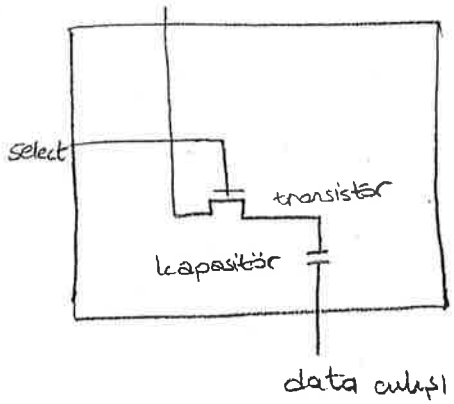
Ana Bellek Bilgisayar sisteminin merkezi depolama birimidir. Büyük ve hızlıdır.

- Komut ve dataların depolandığı birimdir.
- Statik ve Dinamik olmak üzere 2 türlüdür.
  - Statikte bilgi yarı iletkenlerden oluşur. Akım verildiği sürece bilgi depolanır.
  - Dinamikte bilgileri elektrik akımı şeklinde tutar. Kondansatörler tutar.

- Okuma yazma süreleri kısadır.



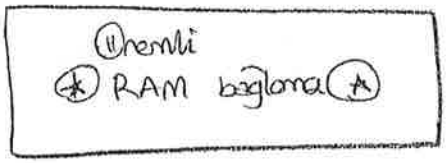
(statik RAM)



(dinamik RAM)

Bellek adres Haritası :

- Ana bellekten ön belleğe verilerin aktarılmasına haritalama denir.
  - ↳ Önbellekte bulunan her kelimenin, ana bellekte kopyası bulunur.



İçerik Adreslemeli Bellek : Cache bellek organizasyonunda kullanılır.

- En hızlı ve çok yönlü kullanılabilen ön bellek tasarımı içerik adreslemeli bellek alanıdır.

Throughtput → Verilen sabit bir zamanda yapılan iş miktarı.

Kıymet KIZIL

Bilgisayarın Hiyerarşik Yapısı.

Level 6: Kullanıcı seviyesi:

Kullanıcı ile bilgisayarın etkileştiği arayüz.

Level 5: Yüksek seviyeli dil seviyesi:

C, Pascal, Java... prog. dilleri ile yazdığımız prog. ile bilgisayarın etkileştiğimiz seviye.

Level 4: Assembler dili seviyesi:

5. seviyedeki yüksek dil ile yazılmış programları assembler'a çevirir

Level 3: Sistem yazılımı seviyesi:

Sistemde işlenen processleri kontrol eder. Assembler dili komutları farklı değişikliğe uğramadan bu seviyeye gelir.

Level 2: Makine dili seviyesi:

ISA - komut seti mimarisi.
Yukarıda mimarisine özel komutların oluşturulduğu seviyedir.
Dahilde yazılmış komutlar derleyici, yorumlayıcı veya assemblera ihtiyaç duymazlar.

Level 1: Kontrol seviyesi:

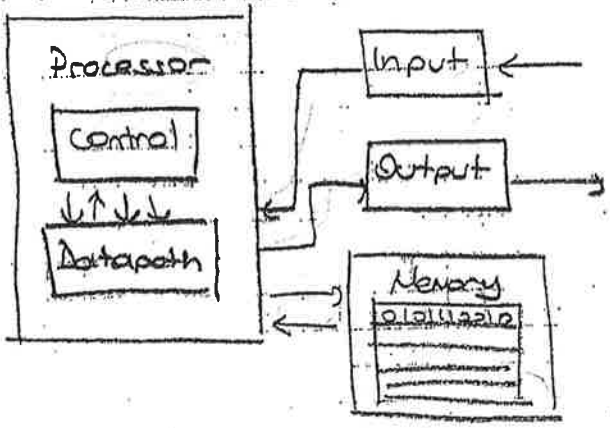
Bir kontrol chitesi komutları decode eder. Verileri sistemde hareket ettirir.

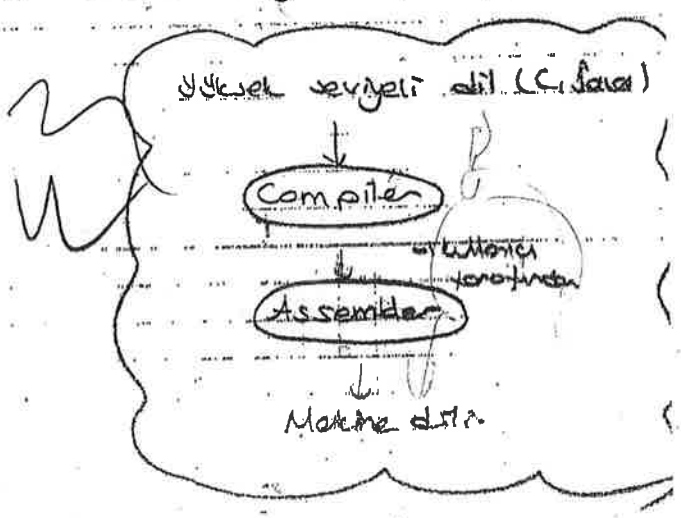
Level 0: Digital Logic seviyesi:

Logik devreler, diğer dil seviyelerdeki matematiksel ve logik işlemleri gerçekleştirir.

⇒ Bir komutun temel safhası: "FETCH, DECODE, EXECUTE, write back."
(alma) (çözme) (yürütme)

Bilgisayar Sistemi:

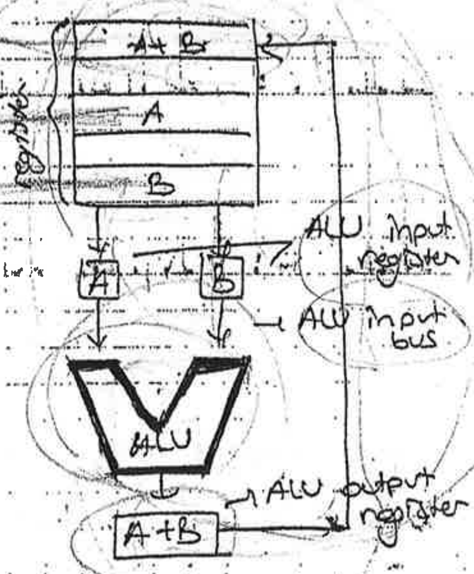




C Program
↓
(Compiler)
↓
Assembly dili programı
↓
(Assembler)
↓
Makine dili modülü + Kütüphane
↓
(Linker)
↓
Yürütülebilir makine dilinde prog.
↓
(Loader)
↓
Ana bellek

register
↓
L1 - (2ns)
L2 - (2-5 ns)
L3 - (5-20 ns) — yavaşlar
kapasite artar
↓
RAM
↓
Flash Memory
↓
HDD
↓
Tape Backup

Datapath



(Fetch - Decode - Execute)

MIPS-32 → 32 bitlik 32 register

MIPS-64 → 64 " 32 "

\$v0, \$v1 ⇒ return, sonuç değeri

\$a0 - \$a3 ⇒ prosedür - fonksiyon

\$t0 - \$t7 ⇒ geçici (8 - 15)

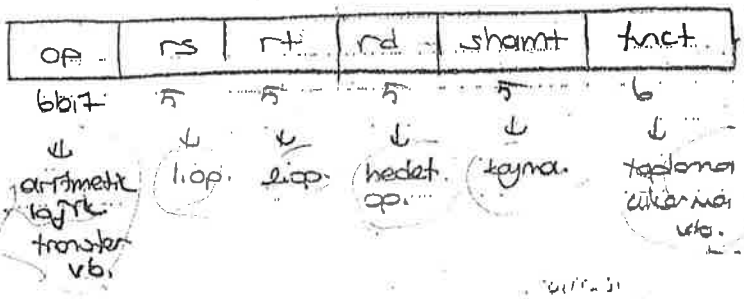
\$s0 - \$7 ⇒ kalıcı (16 - 23 registerlar)

\$gp, \$sp } işaretçinin korunması

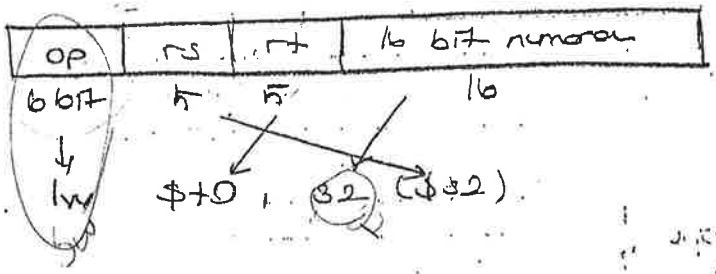
int a, b, c, d[0] , başlangıç adresi 1000 . \$s0

addi \$s0 , \$zero , 1000
addi \$s1 , \$s0 , 0 ⇒ a
" \$s2 , \$s0 , 4 ⇒ b
\$s3 , \$s0 , 8 ⇒ c
\$s4 , \$s0 , 12 ⇒ d[0]
} adresleri

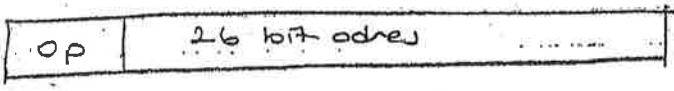
R-tipi : registerlardan işlem komutu için



I - tipi : sabitler veya veri transferi



J - tipi : dallanma



sll → shift left → her kaydırma 2 ile çarpmak (2^1)

srl → // right

beq $r1, r2, L1 → r1 = r2 ise L1'e dallan

bne $r1, r2, L1 → r1 ≠ r2 " " "

etiket → şartsız dallanma

DÖNGÜLER (Loops): her kelime 32 bit olduğu için $2 \times 16 = 32$ → 2 adım kaç bit

while (save[i] == k) i → $s3 dizi[0] → $s6
k → $s5

i = i + 1

1. adım: save[i]'yi geçici operandı yükle. Bunun için önce adres bulur. Fakat önce i değişkenini 4 ile çarp, bunun için 2 kez sola kaydır.

```
Loop: sll  $t1, $s3, 2     // $t1 = 4 * i
      add  $t1, $t1, $s6   // $t1 = save[i] adresi
      lw   $t0, 0($t1)     // $t0 = save[i]
      bne  $t0, $s5, Exit  // save[i] ≠ k → Exit
      addi $s3, $s3, 1     // i = i + 1
      j    Loop            // Loop'a git
```

slt: $t0, $s3, $s4 = s3 < s4 → $t0 = 1

slti: $t0, $s2, 10 = s2 < 10 → $t0 = 1

$ra: dönülecek orijinal noktayı bilmek için bir tane adres register.

jal: prosedür adresi ($ra) — İstenen adrese atlar ve takip eden dönüş adresini de $ra'ya kaydeder.

⇓

Bu adres program counter'daki adres + 1'dir.

jr: $ra: $ra'daki adrese şartsız dallanma

Program Counter (PC) yürütülmekte olan komut adresinin saklandığı özel bir register.

* Registerların kaydedildiği hafıza kısmı "spilling memory"

```
int leaf_ex (int g, int h, int i, int j) {
    int f;
    f = (g+h) - (i+j);
    return f;
}
```

g → $a0, h → $a1, i → $a2, j → $a3, f → $s0

⇒ MIPS kodu: "Prosedürün yapacağı ilk işlem kullanılan regleri stack'e depolamaktır"

```
leaf_ex:                    # Prosedür etiketi
push  addi $sp, $sp, -12    # stack'te 3 birim yer acmak
      sw   $t1, 8($sp)      # t1'i stack'e kaydetme (push)
      sw   $t0, 4($sp)
      sw   $s0, 0($sp)
```

Argümanlar $a0 – $a3
Geri dönüş değerleri $v0 – $v1 (sonuç)

* Prosedür, s ile başlayan reg'leri kullanacaksa bunların içeriklerini mutlaka stack'e atmalıdır.

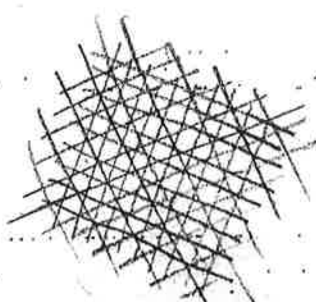
```
add   $t0 , $a0 , $a1     // t0 = g+h
add   $t1 , $a2 , $a3     // t1 = i+j
sub   $s0 , $t0 , $t1     // s0 = t0-t1

add   $v0 , $s0 , $zero   // v0 = f = $s0 + 0

lw    $s0 , 0 ($sp)  }
lw    $t0 , 4 ($sp)  }  pop.
lw    $t0 , 8 ($sp)  }
addi  $sp , $sp, 12       // stack'i ilk konumuna getir
jr    $ra                 // return
```

* 32 bitlik kelimeler yerine 8 bitlik kelimeler transfer edilebilir.
  - lb : ana hafızadaki 8 bitlik veriyi, registerin en sağ 8 bitine yaz.
  - lh : " 16 " " " " 16 " "
  - sb : tersini yapar.
  - sh : " "

lb 8 en sağ
lh 16 en sağ
sb 8 en sol
sh 16 en sol

ör: void ____ (char x[], char y[]){

```
    int i;
    i = 0;
    while ((x[i] = y[i]) != '\0')
        i += 1 ;
        i++ }
```



```
      addi  $sp , $sp , 4
      sw    $s0 , 0($sp)
      add   $s0 , $zero, $zero

L1 :  add   $t1 , $s0 , $a1
      lb    $t2 , 0($t1)
      add   $t3 , $s0, $a0
      sb    $t2 , 0 ($t3)
      beq   $t2 , $zero , L2
      addi  $s0 , $s0 , 1
      j     L1

L2 :  lw    $s0 , 0 ($sp)
      addi  $sp , $sp , 4
      jr    $ra
```

* lui : 16 bitlik sabiti, registerin en ağırlıklı 16 bitine kaydeder.

Relative adresleme: Dallanma adresi, 32 bitlik bir reg'in içeriği ile toplanıp PC'ye yazılır.

Bilgisayar Aritmetiği:

En anlamlı bit (0) : pozitif
(1) : negatif

32 bitlik işaretli sayıların en büyük $2^{32-1} - 1$ $(2^{n-1} - 1)$

en küçük $-2^{32-1} + 1$ $-(2^{n-1})$

işaretli sayılarda add , addi , sub , mult , div
işaretsiz " addu , addiu , subu , multu , divu
(çarpma) (bölme)

lb / lbu: (lb) 32 bitlik işaretli sayının en anlamlı 24 bitini işarete göre doldurur (lbu) işaretsiz sayılarda anlamlı bitlere 0 atar.
lh: işaretli sayıda en anlamlı 16 biti işarete göre doldurur
lhu: işaretsiz " " "
slt / slti: işaretli sayı karşılaştırma / tam sayı ile
sltu / sltiu: işaretsiz " " " "

Floating Point:

exponent : 32 bit için bias değeri 127
64 bit " " " 1023

sign	8 bit exponent (11 bit)	23 bit fraction (20 bit)

0 : +
1 : −

$E = e + 127$ — virgülden sonraki kısım

$0 < E < 255$

$e = 2^{8}$

ör: $-2345{,}125_{10}$

$= -100100101001{,}001$ (Binary)

$= -1{,}00100101001001 \times 2^{11}$ (normalize)

yazılmaz — negatif

$E = 127 + 11 = 138_{10} = 10001010_2$

1	10001010	00100101001001

ör: floating → decimal

1	10000001	010000 ---

129 ; $(1 \times 2^{-2}) = 0{,}25$ = fraction

$(-1)^s \times (1 + \text{fraction}) \times 2^{E - \text{bias}}$

$= (-1)^1 \times (1 + 0{,}25) \times 2^{129-127} = -1 \times 1{,}25 \times 2^2$

Floating Point Toplama:

* Exponentleri aynı olmalı.
* Önce kesir kısımları toplanır.

Ör: $1231823.333_{10} + .0011_{10} = ?$

$1,231823.333_{10} = 1,1231823333 \times 10^6$

$.0011_{10} = 1,1 \times 2^{-3} = 0,0000000011 \times 10^6$
$\times 2^9$

⇒ Üsler eşitlendikten sonra topla

$$\begin{array}{r} 1,1231823333 \times 10^6 \\ 0,0000000011 \times 10^6 \\ + \\ \hline 1,1231823344 \times 10^6 \end{array}$$

Ör: $0,5 + (-0,34375) = ?$ (4 digit ifade et)

⇒ Önce binary çevir.

$0,5_{10} = 0,1_2 = 1,000_2 \times 2^{-1}$

$0,2^{-1} \rightarrow 0$ (1)

$-0,34375_{10} = -0,0111_2 = -1,110_2 \times 2^{-2}$
$\frac{7}{16} \quad \frac{7}{2^4}$ $\times 2^1$

⇒ Exponentleri eşitle.

$$\begin{array}{r} 1,000 \times 2^{-1} \\ -0,111 \times 2^{-1} \end{array}$$

⇒ fraction / kesirli kısım topla

$$\begin{array}{r} 0,111 \times 2^{-1} \\ 1,000 \times 2^{-1} \\ -0,111 \times 2^{-1} \\ \hline 0,001 \times 2^{-1} \end{array}$$

⇒ normalize et.

$1,0 \times 2^{-4}$

⇒ desimal değeri?

$0,0001000_2 = 0,0625_{10}$

$\frac{1}{2^4} = 0,0625$

IEEE 754 standardında toplama!

$12345,125_{10} + (-37,5)_{10}$

X Y

X = | 0 | 10001010 | 10010010001001000 |

Y = | 1 | 10000110 | 00101000000000000 |

⇒ Exponentleri büyük olana eşitle

$E_x > E_y$ ... $E = 10001010 = 138_{10}$

$E_x - E_y = 12$

⇒ F_y 12 bit sağa kayacak

$F_y \times 2^{-12} = 0.000000000000110000$

⇒ Kesirleri topla.

$F_x + (F_y \times 2^{-12})$

$= 1.10010010010001000$
$+ 0.00000000000110000$

1.10010010010111000 ⇒ normalize ✓
overflow-underflow -

⇒ Sonuç:

0	10001010	10010010010111000
1	8	23

*

* add.s - d
sub.s - d
mul.s - d
div.s - d
c.x.s - d → karşılaştırma

eq
neq
lt ... less than

single double

* lwc1 → hafızadan al
swc1 → hafızaya yolla

→ 2 single precision yapıyor

Ör: Fahrenheit (F°)'i, Celsius (C°)'a çevir.

```
float f2c (float fahr) {
    return ((5.0/9.0) * (fahr - 32.0)); }
```

1) sabitlere ihtiyaç $gp-
sıcaklık (fahr) $f12.
elde edilen sonuç $f0.

2) Hafızadan sabitlerin alınması

```
lwc1 $f16, const5($gp) // f16 = 5.0
lwc1 $f18, const9($gp) // f18 = 9.0
```

3) 5.0 / 9.0 için:

```
div.s $f16, $f16, $f18
```

4) fahr - 32.0 + 32.0'ın hafızadan alınması

```
lwc1 $f18, const32($gp) // $f18 = 32.0
sub.s $f18, $f12, $f18 // $f18 = fahr - 32.0
```

5) sonucu çarpılması:

```
mul.s $f0, $f16, $f18 // f0 = (5/9) * (fahr - 32.0)
jr $ra // return
```

Taşma (over-flow) (Sonuç alma)

32 bit 2 sayı toplandığında / çıkarıldığında sonuç 31 bit'te;

2 pozitif ... toplandığında sonuç negatif ise,

2 negatif ... " " pozitif "

pozitif = negatif - negatif ise;

negatif = pozitif - pozitif "

negatif + pozitif
negatif - negatif
pozitif - pozitif
} taşma olmaz.

0010 ⇐ sola kayar

× 0011

0010
0010
0000
0000

00000110

		Mcer	Mcand	Product
		0011	0010	0000 0000
①	1	0011	0010	0000 0010
	2	0011	0100	0000 0010
	3	0001	0100	0000 0010
②	1	0001	0100	0000 0110 (toplam)
	2	0001	1000	0000 0110
	3	0000	1000	0000 0110

* P.V = n.R.T Başlangıç P değeri aşağıdaki formül ile hesaplanıyor. MIPS kodunu yaz. Floating - point dönüşümlerini yap. Sonucu hafızanın 1100 adresine yaz.

```
float gas (float n, float T){
    return ((n * 0,082 * (273,0 + T)) * 1,3) }
```

Datalar hafızanın 1000 adresinden itibaren n, T değerleri saklanmıştır. Başlangıç adresi $f0, n $f13, T $f14 registerlarda. Sabit değerlere erişim $gp registeri ile sağlanıyor ($gp 1500'de). Sabit değerler 0,082, 273.0, 1,3; $gp'nin gösterdiği adres alanından itibaren sırasıyla yazılı.

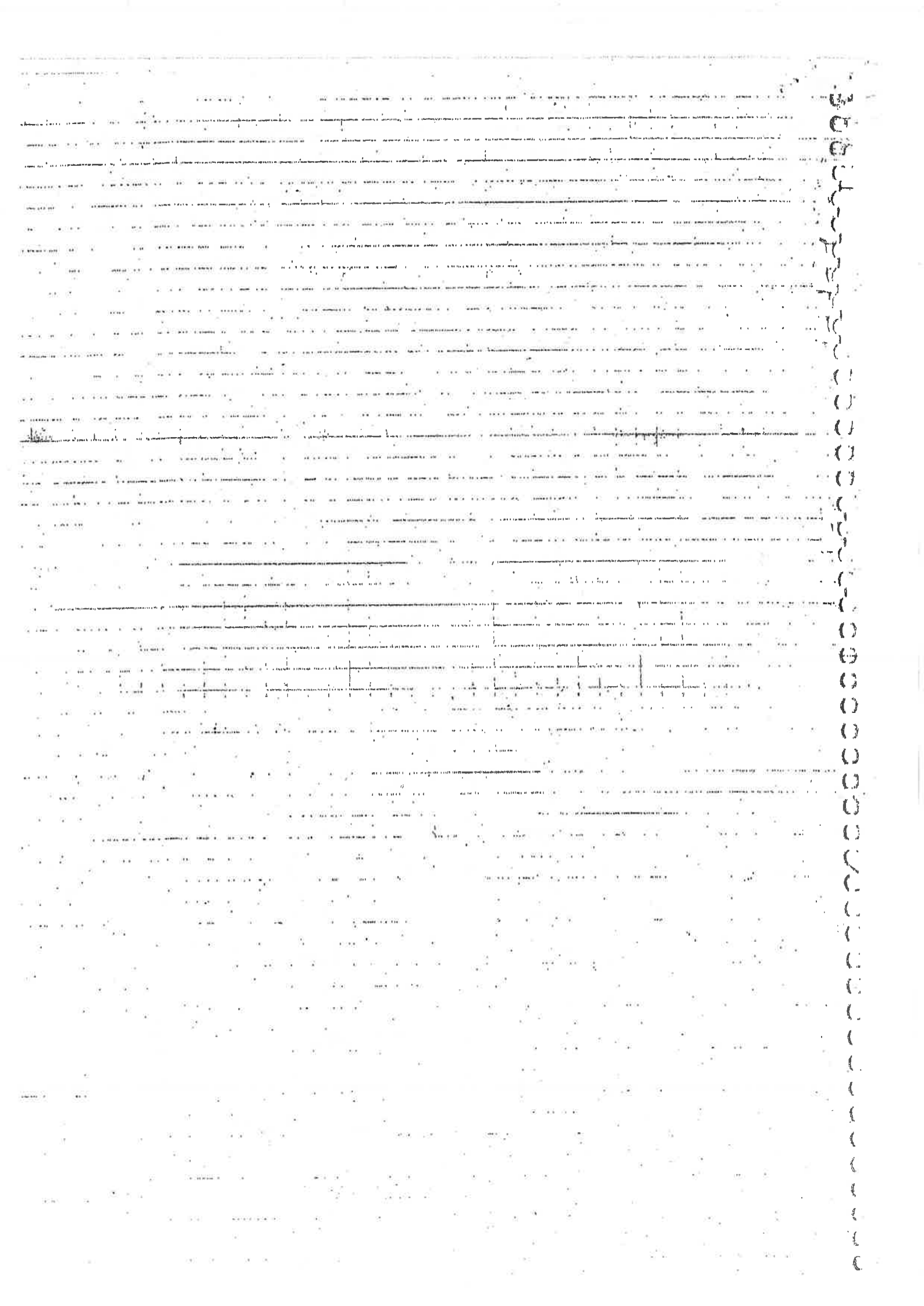
	1020	82300000
	1016	18123444444
	1012	12240000
	1008	8d230000
T ←	1004	41d80000
n ←	1000	3e000000

1520	12300000	
1516	ca200000	
1512	8aad0000	
1508	3fa66666	→ 1,3
1504	43888000	→ 273.0
1500	3da7ef9e	→ 0,082

sonuç

407fd70a

```
1)  lwc1   $f10, 0, $gp        // 0,082'nin $f10'a getirilmesi
2)  lwc1   $f11, 4, $gp        // 273,0'ın $f11'e        "
3)  lwc1   $f12, 8, $gp        // 1,3'ün $f12'ye         "
4)  lwc1   $f13, 0($f0)        // n'nin $f13'e           "
5)  lwc1   $f14, 4($f0)        // T'nin $f14'e           "
6)  add.s  $f14, $f14, $f11    // T+273,0
7)  mul.s  $f13, $f13, $f10    // n * 0,082
8)  mul.s  $f13, $f13, $f14    // ((n * 0,082) * (T+273))
9)  mul.s  $f15, $f13, $f12    // ((n * 0,082) * (T+273)) * 1,3
10) swc1   $f15, 100($f0)      // sonucu 1100'e yaz.
11) jr     $ra
```



PERFORMANS: Bireysel bir bilg. kullanıcısı olarak response (cevap süresi) nin kısalmasını isteriz.
Datacenter yöneticileri ise genelde birim zamanda üretilen veri miktarı veya bant genişliğinin artmasını isteriler.

Response time: ↓min → performans ↑max.

Programın sonu ve başı arasındaki harcanan zaman

Execution time: (Yürütme zamanı)

Bir işi tamamlamak için harcanması gereken top. zaman

Throughput: (Veri - çıktı)

Verilen bir zamanda yapılan top. iş miktarı

Soru: Eğer yeni bir işlemci ile makineyi upgrade (yükseltmek) yaparak neyi arttırmış olurız?

C: Makine hızlanır, performans artar.

CPU time: → giriş/çıkış, başka programlar için harcanan değil hesaplama için

Giriş - çıkış olayları veya başka programların harcadığı zaman değildir

CPU time: CPU'nun bir program için (hesaplamaya) harcadığı zaman

* Kullanıcı CPU time: Bir program için CPU'nun harcadığı zaman

Sistem " " : Programın adlızma süresince işletim sisteminin harcadığı

⇒ Verim ve execution (yürütme) time:

$$\text{Performans} = \frac{1}{\text{Execution time.}}$$

A ve B makinelerinin performans oranı:

$$\frac{P_A}{P_B} = \frac{Ex_B}{Ex_A} = n \qquad P_A = n \cdot P_B$$

clock cycle: Bilgisayarın, olayların donanım tarafından gerçekleştirme zamanı olarak belirlediği zaman çevrimleri

Yürütme zamanı yine cycle terimi kullanılır.

$$\text{CPU execution time} = \text{CPU clock cycle} \times \text{clock cycle time}$$

bir komutun diğer komuta geçerken harcadığı zaman (Pipelining)

* $\text{clock rate} = f = \frac{1}{\text{clock cycle time}} = \frac{1}{T}$

1 Hz = 1 cycle / sec

$1\text{ Hz} = \frac{1 \text{ cycle}}{\text{second}}$

4 GHz'lik clock'un cycle periyodu:

$\frac{1}{4 \times 10^9}$ seconds $= \frac{1}{4 \times 10^9} \times 10^{12} = 0.50$ picoseconds

** $\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$ → yürütme süresi

⇒ Performansı geliştirmek için bu bağıntıya göre neler yapılabilir?

1) Cycle sayısı ↓

2) Clock cycle time (T) ↓

3) Clock rate (F) ↑

* CPU time = $\frac{\text{clock cycle}}{\text{clock rate}}$ — yürütme zamanı — işlemci hızı (GHz)

A → 10s. → CPU time 10 = clock cycle / 4 x 10^9 CC = 40 x 10^9
4 GHz → F

B → 6s → CPU time F = 40 x 10^9 . 1,2 = clock cycle / 6

F = 8 GHz

*** Ortalama Clock Cycle: (CPI)

Bir program için bir komut başına düşen clock cycle'dır. (yürütme zamanı)

CPU clock cycle = komut sayısı x CPI

Not: * Bir programdaki yürütme zamanı komutların sayısına bağlıdır.

* Bir programın yürütme zamanı, yürütülen komut sayısı ile her komut için harcanan ortalama zamanın hesaplanması.

CPU execution time = (komut sayısı x CPI x T) / CPU clock cycle

* Verilen bir prog için cycle zamanı, clock rate, CPI sayısı gerekli (cycle başına sn)

* Diğer değişkenlerin hangisi performansı etkiler?

Yürütülen prog için cycle sayısı (+)

- Komut sayısı. (-)

Frekans / saniye başına cycle. (+)

CPI. (+)

MIPS / saniye başına komutların ort. say. (+)

A → clock cycle : 250 ps.
CPI : 2.0
B → CC : 500 ps.
CPI : 1.2

Hangi makine daha hızlı?
(Aynı komut seti)
(komut sayısı = I = sabit)

CPU clock cycle$_A$ = I x 2.0 ⇒ komut sayısı x CPI
" $_B$ = I x 1.2

CPU time = CPU clock cycle x I
" $_A$ = I x 2.0 x 250 = I x 500
" $_B$ = I x 1.2 x 500 = I x 600

A < B, A daha hızlı.

$\frac{P_A}{P_B} = \frac{E_B}{E_A} = \frac{I \times 600}{I \times 500} = 1.2$ ∴ $P_A = 1.2 \times P_B$

* Kod Segmentlerinin karşılaştırılması

3 farklı komut sayısı var. A: 1 cycle, B: 2 cycle, C: 3 cycle ihtiyacı duyuyor.

1. kod dizisi: 5 komutu içerir. A = 2, B = 1, C = 2. = 5
2. " 6 " : A = 4, B = 1, C = 1 = 6

⇒ Hangisi, ne kadar hızlı?

⇒ CPI$_1$, CPI$_2$?

CPU clock cycle = $\sum_{i=1}^{n} C_i$ komut say. x CPI$_i$

CPU clock cycle$_1$ = 1 x 2 + 1 x 2 + 2 x 3 = 10
" $_2$ = 1 x 4 + 2 x 1 + 3 x 1 = 9

CPI$_1$ = $\frac{10}{5}$ = 2
CPI$_2$ = $\frac{9}{6}$ = 1,5

1. daha hızlı.

** Eğer bileşenlerden bir kısmı ölçüleş ise; performans karşılaştırması ölçüleş olmayan bileşenlere göre yapılır. Aynı şekilde karşılaştırma komut sayısı ve CPI'ya göre yapılabilir.

Performans Değerlendirilmesi:

Benchmarks: Bir ölçme veya değerlendirme standardı'dır. Tipik bir şekilde tanımlı workload'u gerçekleştiren bilgisayar programıdır.

* Bilgisayar benchmark'ları ölçüler metrikler:

Hız: Bir workload'ın ne kadar hızlı tamamlandığı

Throughput: Birim zamanda tanımlanan workload'ların

(workload: tanımlı bir işlemler kümesi programı)

Performans Raporu:

Program	Computer A execution time	Computer B execution time
1	1 sec.	10 secs.
2	1000 secs	100 secs.
Total	1001 secs.	110 secs.

Program 1 için A, B'den 10 kat daha hızlı

" 2 " B, A " 10 " " "

Toplam sonuç nedir?

P1: $\frac{P_A}{P_B} = 10 = \frac{Ex_B}{Ex_A} = \frac{10}{1} = 10$

değişmez.

P2: $\frac{Ex_B}{Ex_A} = \frac{100}{1000} = \frac{1}{10} = \frac{P_A}{P_B}$ $P_B = 10\ P_A$ değişmez

Total: $Ex_B = 110$ $\frac{110}{1001} = \frac{P_A}{P_B}$ B daha hızlı.

$Ex_A = 1001$

* Farz edelim ki iki farklı desktop vectik

- Bu hali bilgisayar en yüksek clock rate (ff) 'e sahip olan

- Bütün bilgisayarlar aynı komut setini kullandığı için benchmarkların aynısı olacak.

Ex: Program: Ex_A Ex_B

1 2 secs 4 secs

2 5 secs 2 secs

P1 , A - B'den daha hızlı

P2 , A - B " " "

P1 ve P2 'nin yürütme sayısı eşit iş yükü için A - B'den daha hızlı

P2 , P1 'den 2 kat daha iş yükü için A - B " " "

Doğru?

DataPath ve Kontrol

* Bir komutu işleme sürecinde kullanılan donanımların bütününe (datapath) denir.

○ Tek cycle:

* Her komut (tek cycle'da) gerçekleşir.
* Daha kısa işlenir fakat komut süreci en uzun olan (→ r tipi)
** Her komut tek cycle'da gerçekleştirmede (en yavaş komut) kadar zaman alır.
→ en geç hangisi bitiyorsa

Çoklu - Cycle:

* Fetch - Execute yapılır.
* Her bir clock cycle'da (1 adım) işlenir.
* Her komut ihtiyacı kadar cycle kullanılır.
* Zaman israfı yok.
* Single'a göre daha hızlı
* Dezavantajı: Bir komutun işlenmesi bitmeden diğerine geçilmez.

○ Pipe-line: → (en avantajlısı bu)

* Her komut ardıl adımlarla yürütülebilir.

Fetch — Decode — Execute — write back } adımların teker teker bitmesini beklemez

iken başlayabilir.

** En önemli öz. aynı anda farklı komutlar işlendiği için daha hızlı

* lw → register file'de bitiyor
 adresi ver. datayı ver. Memory write ile veriler adrese yazıyor.
 sw → memory'de bitiyor. " read " " " okuyor

WD: Write data
RD: Read data

○

MIPS VeriYolu I: Single Cycle

R tip komutlar ve load-store için veriyolu oluşturmak için 2 mux kullanılır. add, lw, sw.

MIPS Veriyolu II: Single Cycle:

Komut fetch'i ekleme

MIPS veriyolu III: Single cycle:

Branch işlemi için şart sağlanmıyorsa doğrudan PC+4'e dallanmak için yeni mux eklendi.

Önemli Not: Single cycle gerçekleştirmede veri, komut boyunca depolanmaz. Yalnızca kombinasyonal lojik devre ile taşınır.

* Dallanma komutu (beq) varsa PcSrc 1
 " 0

Kontrol :

* Ana kontrol birimi ürettiği sinyaller
  - ALU kontrol girişi (2 bit)
  - Yazma - okuma enable her bir depolama elemanları için sinyal
  - her bir mux için seçme kontrolü.

Kontrol Birimi devre şeması:

Inputlar = 0

Output : RegDst = 1 , diğerleri 0.
RegWrite = 1
ALUOp1 = 1 → (dallanma için)

* load/store için add ALUOp 00.
  dallanma için sub ALUOp 01
  and, or, add, sub, slt R tipi komutlar için ALUOp 10

Kontrol Sinyalleri

	Lojik 0	Lojik 1
RegDst	Yazmak reg. için hedef reg. sayısı (20-16) rt alanı	Yazmak reg. için hedef reg sayısı (15-11) rd alanı
RegWrite	—	Girişindeki datayı hedef reg'e yazar
ALUSrc	İkinci ALU operandı ikinci reg. çıkışından gelir (Read Data 2)	İkinci ALU operandı komutun düşük 16 bitı.
PCSrc	PC toplayıcının çıkışı toplanır yeni değiştirilir PC+4 ile hesaplanır.	Dallanmayla hesaplanır.
MemRead	—	Data memory verilen giriş adresine göre belirlenir. Bu giriş ilk Read Data çıkışına konur
MemWrite	—	Data memory verilen giriş adresine belirlenir. Bu giriş WriteData girişinin değeri ile değiştirilir
MemtoReg	ALU'dan gelen değeri WriteData girişi reg. yazar	Data Mem'den gelen değeri WriteData girişi reg yazar

BİLİNECEK!

Kontrol Birimi ile MIPS veri yolu
" girişi 6 bit komut opcode, çıkış 7, 1 bit sinyal ve

Kontrol sinyalleri:

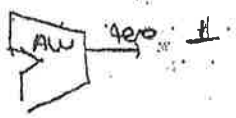
R tipi: RegDst, RegWrite → 1 ; ALUSrc, PCSrc, MemRead, MemWrite, MemtoReg → 0 ; Operation ??? (çıkışa 1, diğer 0)

lw komutu: RegWrite, ALUSrc, MemRead, MemtoReg → 1 ; RegDst, PCSrc, MemWrite → 0 ; Operation 010 (Hafızalar 1, diğerleri 0)

sw komutu: ALUSrc, MemWrite → 1 ; PCSrc, MemRead, RegWrite → 0 ; Operation 010

RegDst, MemtoReg → X

beq komutu: RegDst, MemtoReg → X ; RegWrite, ALUSrc, MemRead, MemWrite → 0 ; Operation 110



multicycle:

* Herbir adım bir clock cycle'da oluşur.
* Her adımda yapılan iş miktarı yaklaşık birbirine eşit
* Her cycle'da her fonk. birimi en fazla 1 kez kullanır.
* Fonksiyonel üniteler aynı komutun farklı çevrimlarında kullanılabilir.

Komut yürütme Adımları:

1) Komutların getirilmesi ve PC'nin arttırılması (Fetch)
2) " çözülmesi ve reg. getirilmesi. (Decode ve reg fetch)
3) yürütme, hafıza adres hesabı veya tamamlanması (Execution)
4) Hafızaya erişim veya R tipi komutun " . (Mem. access)
5) Hafızaya okumanın tamamlanması

R-tipi (add → 4 adım; lw → 5; sw → 4; beq → 3; jal → 3. adım)

ALU Kontrol Bit Ayarları:

	AluOp	İstenen ALU işlemi	ALU girişi
lw/sw	00	add	010
branch eq	01	sub	110
R-tip	10	add	010
		sub	110
		and	000
		or	001
		set on less	111

2345 | 2 | 1172 | 2 | 561 | 2 | 280 | 2 | 14 | 2 | 7 | 2 | 3 | 2 | 1

(1) (0) (1) (0) (0) (1) (1) (1)

1110010.001

−2345.125

0.125 × 2 = (0).250

0.250 × 2 = (0).500

0.500 × 2 = (1).000

0.000

001

2345 | 2
1177
1

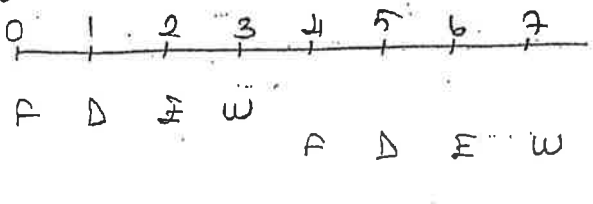
2345 | 2 | 1172 | 2 | 586 | 2 | 293 | 2 | 146 | 2 | 73 | 2 | 36 | 2 | 18 | 2 | 9 | 2 | 4 | 2 | 1

(1) (0) (0) (1) (0) (1) (0) (0) (1) (0)

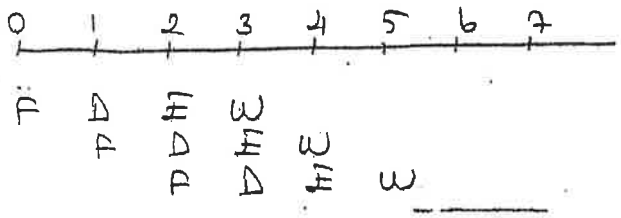
101001010001

PIPELINE

Aynı anda birden fazla komut işlenir.



pipeline olmamış.



pipelined. Zaman kaybetmi

** Pipeline olmayan makine için:

n: görev sayısı

t_n: her görevi bitirmek için gereken süre. (clock cycle)

t_1: n görevi bazen ile tamamlamak için gereken süre

$$t_1 = n * t_n$$

** Pipelined makine için:

k: kesim sayısı

t_p: clock cycle.

t_k: n adet görevin bitirilmesi için gerekli süre.

$$t_k = (k + n - 1) * t_p$$

Hızlanma oranı (speed up):

$$S_k = \frac{t_1}{t_k}$$

$n \uparrow$ $(k+n-1)$ n'e yaklaşır. $t_k = n * t_p$ olur.

$$S_k = \frac{n \cdot t_n}{n \cdot t_p} = \frac{t_n}{t_p}$$

Ör: 4 kesimli pipeline'da bir alt işlemin her bir kesimde işlenebilmesi için gerekli zaman; t_p = 20 ns. Veri edinilecek görev sayısı 100 olsun. Sk hızlanma değeri?

Çözüm:

Pipeline olmayan için; $4 * 20 * 100 = 8000$ ns.

" için; $(4 + 100 - 1) * 20 = 2060$ ns.

$$S_k = \frac{8000}{2060} = 3,88$$

Aritmetik İşlemlerde Pipeline:

* Floating point ve adresleme işl. için hızlandırıcı etki yapar.
* Her kesimde harcanan süre farklı olduğu en kötü durum olarak en uzun clock cycle alınır. Boşta kalmak istemi yavaşlatır.
* Kesim sayısı ne kadar fazla olursa sistem o kadar hızlı olur.
* Paralelanan komutlar eşit süreli olursa pipeline daha hızlı olur.

MIPS'te Pipeline: Neden Kolaydır?

* Bazı komutlar eş süreli gerçekleştirilir (Fetch ve decode vb.)
* " " tek adımda yapılır. Yalnızca birkaç komut için paralelanma vardır.
* Memoryde operandlar belli bir sıraya göre dizilir. Bu için, operandların erişimi kolaydır. lw ve sw ile erişilir.

MIPS'te Pipeline Neden Sorundur?

* Yapı - Kontrol - Veri Tehlikeleri.
* Yapı ⇒ Aynı clock cycle'da birden fazla komutun işlenmesi için donanım yeterli değil.
* Kontrol Tehlikeleri: Yürütmede bir önceki komuttan sonuca göre karar vermek gerekebilir. Genellikle dallanmalardır.

Dallanma komutunun önceki ve sonraki komutları analiz edilerek komutların önüne yeni komutlar yerleştirip akış düzenlenir.

Veri Tehlikeleri: Bir sonraki komut yürütmesi bitmemiş önceki komutun sonuna ihtiyacı duyar. Aşağıda dolayı ilerletmektir. Bunun yetmeyeceği durumlar da vardır. Bu durumlarda akış kesiğini geciktirmesi yapılır.

Pipeline Conflicts Datapath

*** Bir komutun tamamlanmasındaki 5 aşama;

* Instruction fetch & PC Increment
* " Decode and Register Read
* Execution or calculate address.
* Memory access. (MEM)
* Write result into register.

Hepsi birden single clock cycle'ında yapılır.

NOT: Pipeline yürütmede her bir kesimde elde edilen sonuç bir sonraki kesime aktarılabilir. Aynı kesimde farklı komutlar için eş zamanlı olarak farklı donanımlar ile farklı işlemler yapılabilir. Dolayısıyla her kesim için bir pipeline register kullanılmalıdır.

NOT: Bir R tipi komutun yürütülmesi sürecinde; multicycle ve pipeline gerçekleştirmeler arasındaki önemli fark!

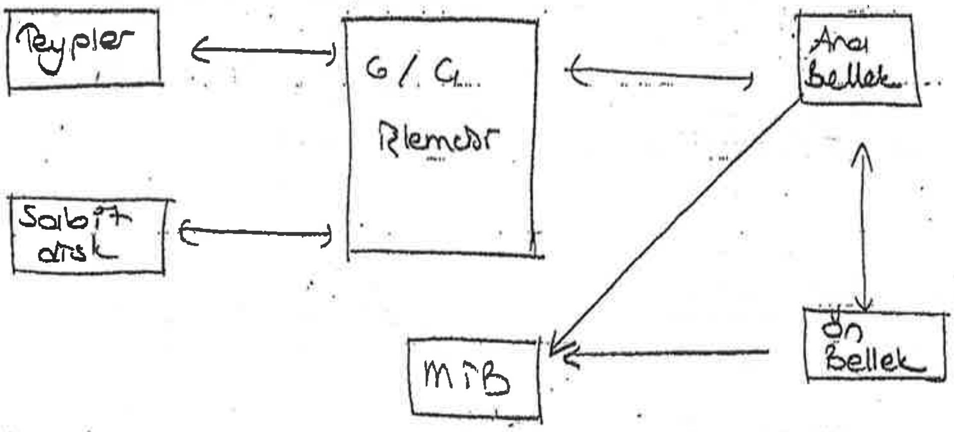
* R tipi bir komuttaki, registere write-back işlemi 5. pipeline kesimindedir. Oysa multicycle gerçekleştirmede de bu işlem 4. durumda gerçekleşir. Niye?

* Reg. file'ı yazarken yapısal tehlikelerden dolayı olabilir.

*** Boru hattı ve multicycle uygulamalar arasındaki temel fark, 5 durumu birbirinden ayırıştırmak için pipeline register yerleştirilmesidir.

BELLEK HİYERARŞİSİ

Elif Çakmak



Sabit disk ve manyetik teypler bellek hiyerarşisinin en altında bulunan yardımcı bellek elemanlarıdır. Ucuz ve geniştirler.

G/Ç birimleri yardımcı bellek ile ana bellek arası veri aktarımı yapar.

Ön bellek küçük ve hızlıdır. MİB ve ana bellek arası aktarım yapar. Ana belleğe erişim zamanı ile MİB arasındaki hız farkını ortadan kaldırmak için kullanılır.

- İletim yolu ile de MİB birbirinden bağımsız programı aynı zamanda icra eder.

ANA BELLEK

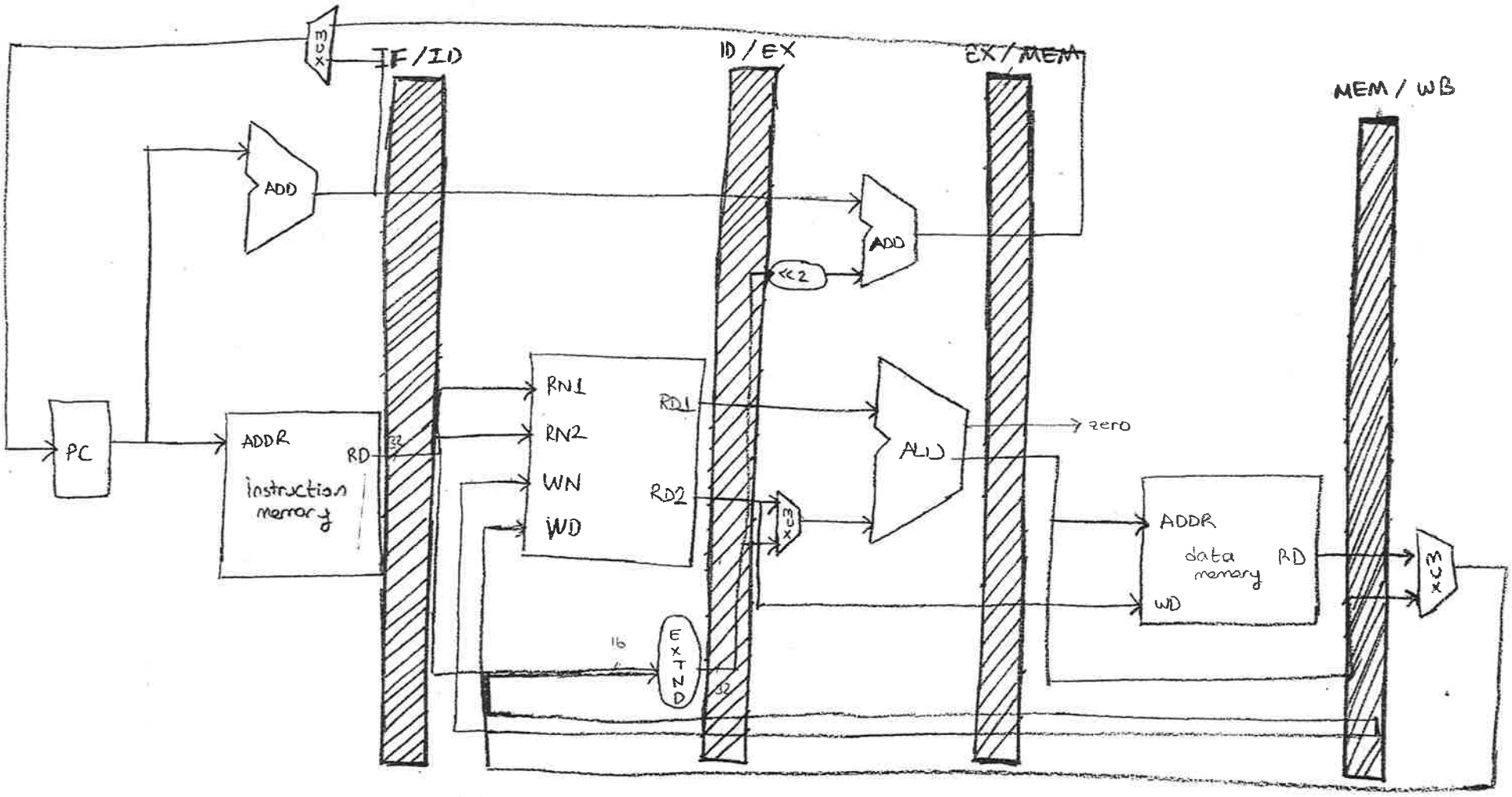
- Merkezi depolama birimidir.
- RAM: de belleğin herhangi bir yerindeki verilere aynı sürede erişilir (RAM mantığı). RAM deki bilgiler değiştirilebilir.
- Programların kalıcı olarak durduğu yer okunabilir bellektir. ROM (BIOS)

RAM # ROM Bağlantıları

* RAM bağlantı MİB ile etkileşim için tasarlanmıştır. Veri yolu çift yönlü (okuma-yazma). Okunurken MİB'e gider, yazılırken MİB'den belleğe.

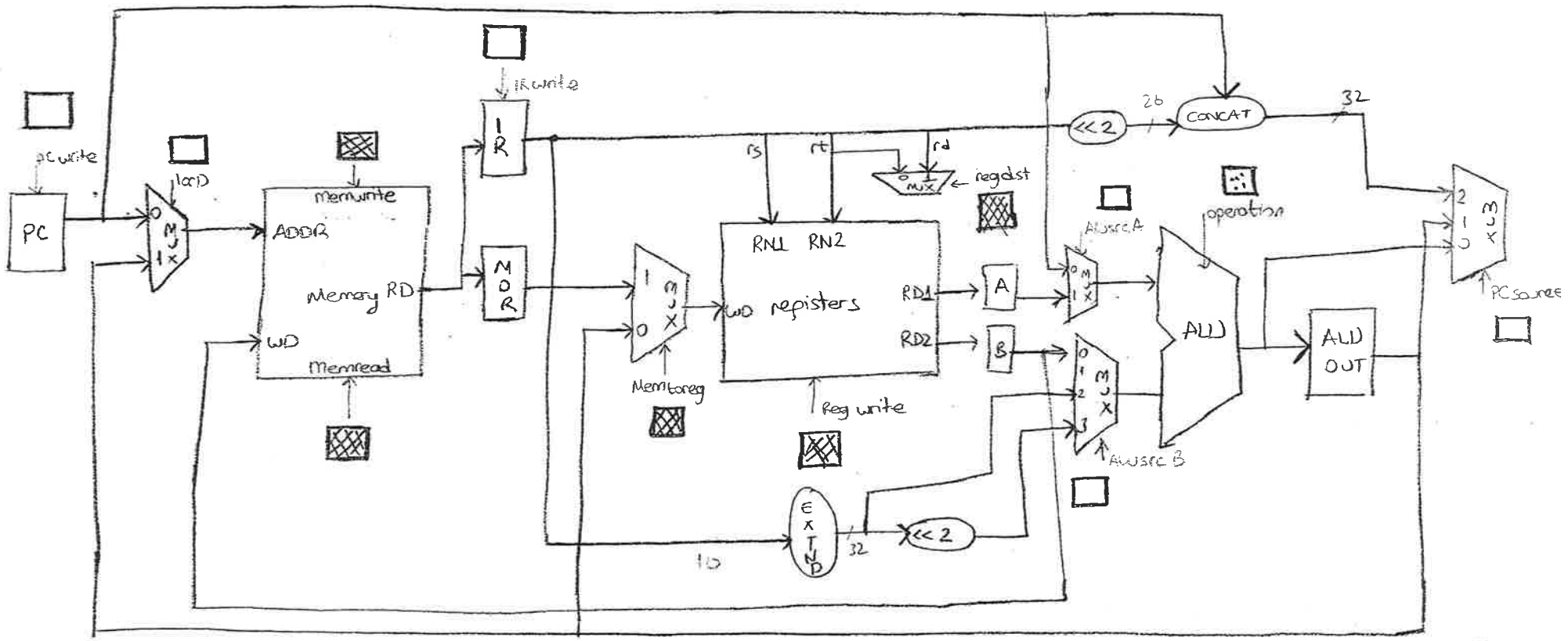
PIPELINE DATAPATH

Genel yapısı



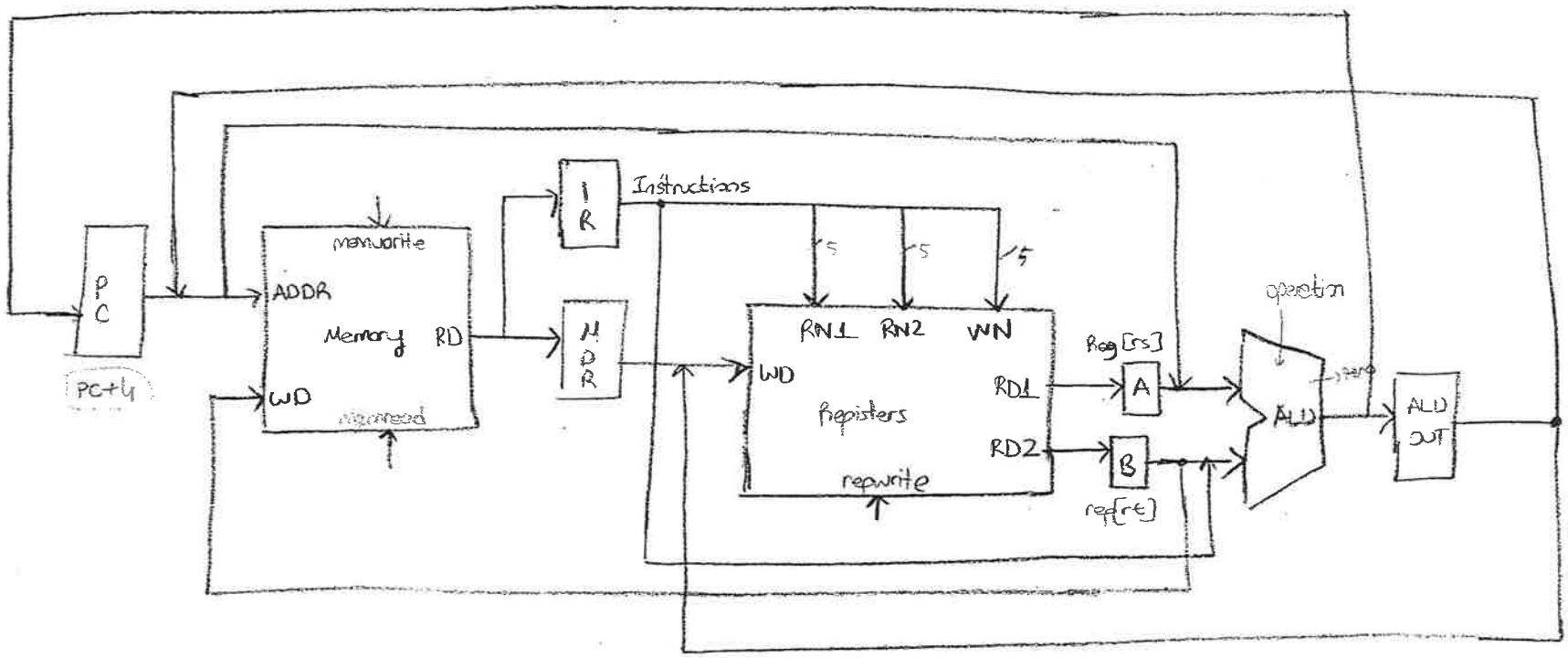
⇒ MULTİCYCLE KONTROL STEP ⇐

Genel Yapısı



000 → and
001 → or
010 → add
110 → sub
111 → slt

MULTICYCLE DATAPATH ADIMI #



$-13,75 \quad (1000\,1101)_2$

0,5 0,125 0,75

1 1 0 $\quad (1101,11)_2$

$1,(1011)\times 2^3 \quad E=130$

1000 0010

1	10000010	1011 1000000000000000000

~~01010101~~

01000111110000011

9 →
10 → A
11 → B
12 → C
13 → D
14 → E
15 → F

+7,125
−3,25
+3,875

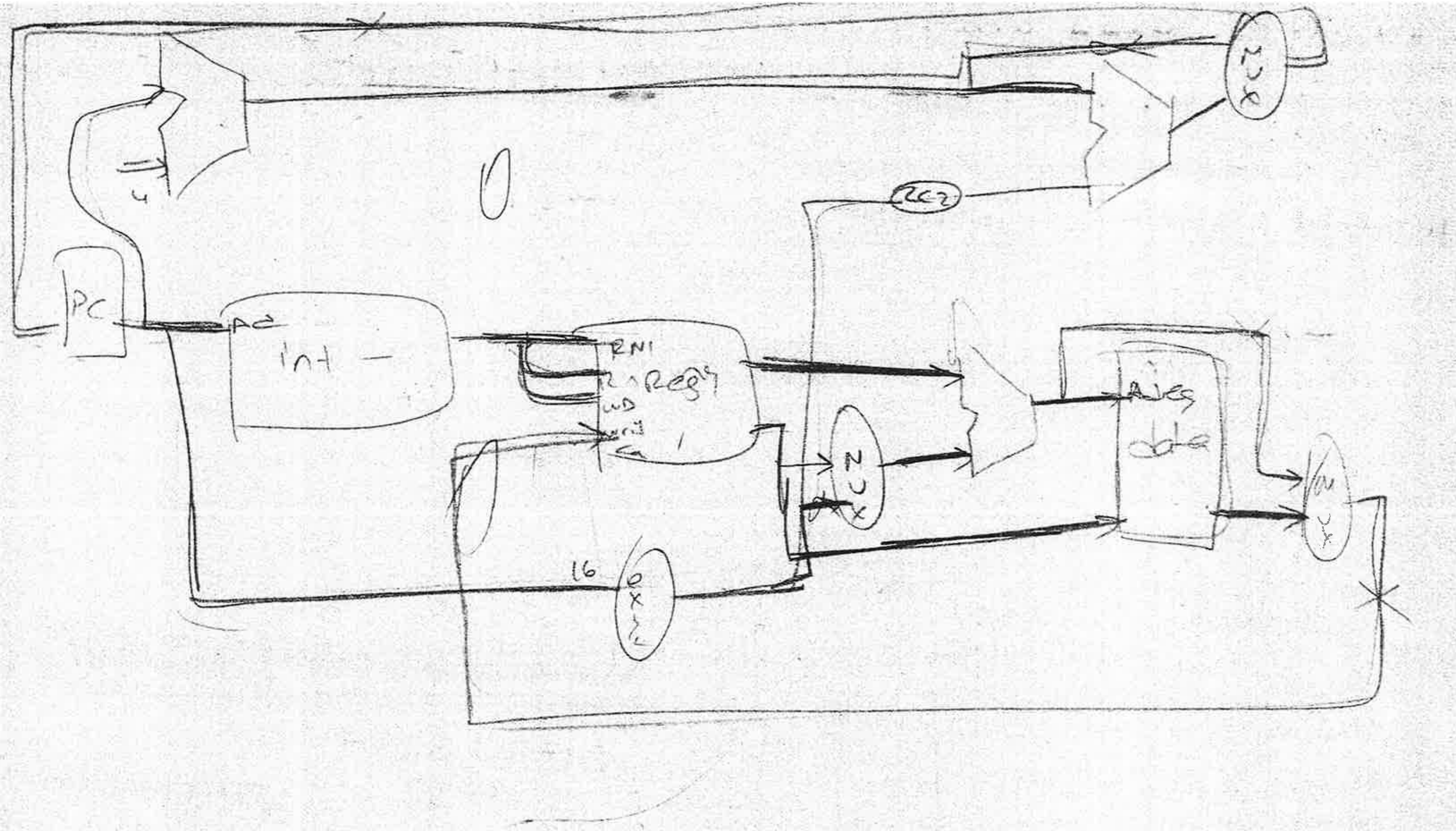
0111,001 (11,01)$_2$ 1,101 × 10^1

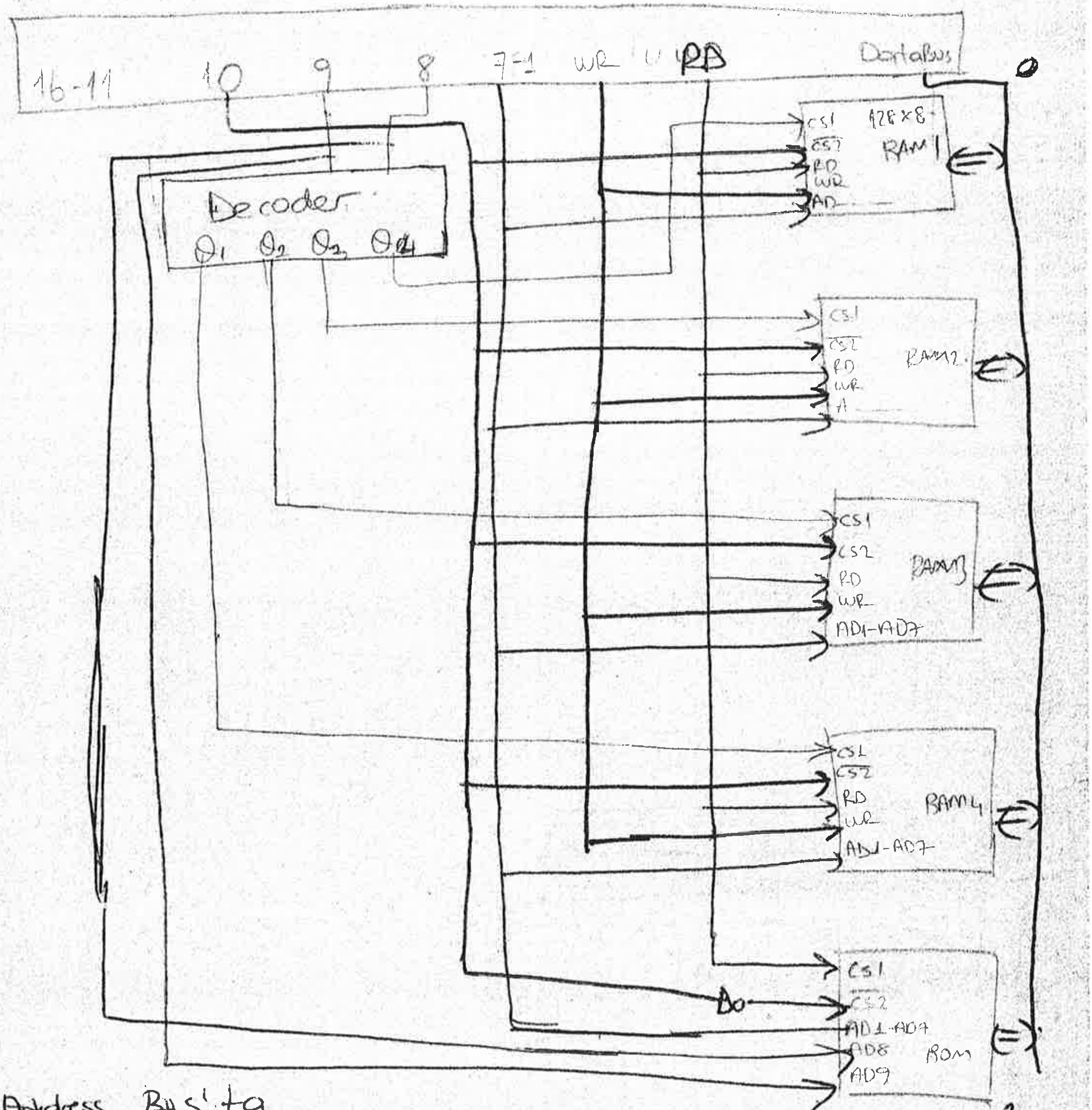
1,11001 × 10^2

0,11001
1,11001

(0111,001)

0,5 − 0,125 − 0,75

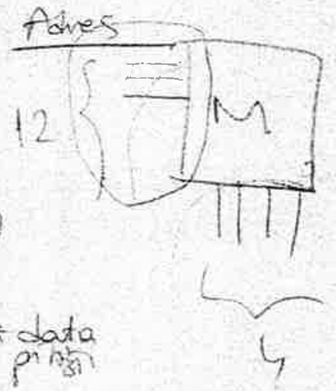

PC
4
16
data



Address Bus'ta

1) RD → RAM1+RAM2+RAM3+RAM4 → RD girişine } uygulanır
ROM'da → CS1 "
2) WR → RAM1+RAM2+RAM3+RAM4 → WR girişine "
3) 7-1 → RAM1+RAM2+RAM3+RAM4+ROM → AD1-AD7.
4) θ_1 → RAM4, θ_2 → RAM3, θ_3 → RAM2, θ_4 → RAM1
5) 8 → AD8 (ROM daki), 9 → AD9
6) 10 → $\overline{CS2}$ → RAM(1,2,3,4)'de
ROM'da ise —Do— ile

12



$2^{12} = 4096/2 \approx 4$

Kapasite = Adres * data pini

2

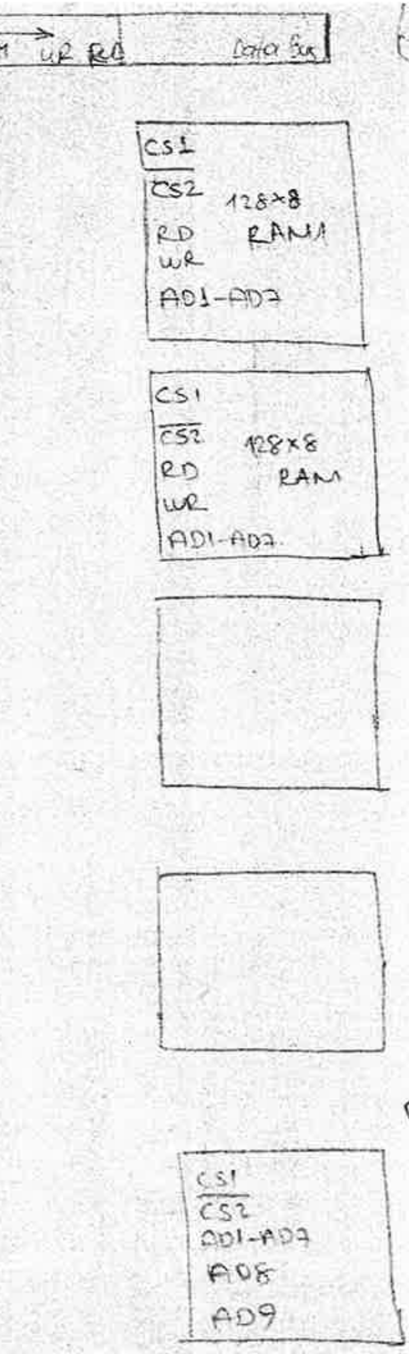
4

512 * 1024 * 8

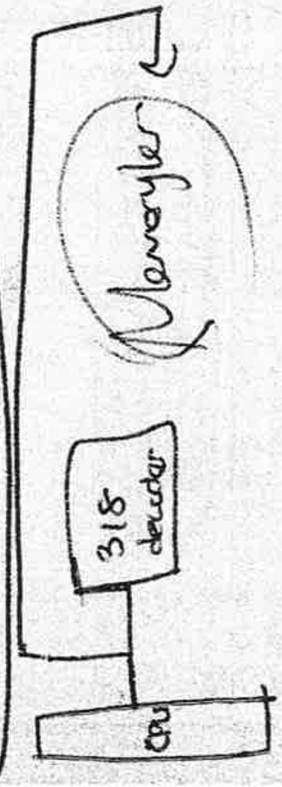
4 194 304

524 288 x 8 tane farklı adres girişi var.

Kapasite ∝ 1 / Hız

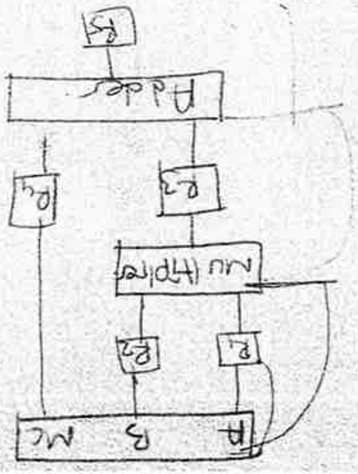


Bellek ile MIB bağlantısı



A*B+C

for i: 1,2,3,...?



tp = Herbir alt işlemin tamamlanması için geçen zaman.

$S_k = \frac{t_1}{t_k} = \frac{t_n}{t_p}$

$\frac{n \cdot t_n}{(k+n-1) \cdot t_p}$