(۲۹)

(کتاب پارسه (سعید صفری))

تنها یِ هفته ی بعد ۲۰ دقیقه ی آخر کلاس تا آخر مباحث امروز

Year: Month: Date:

NOTE BOOK

✓ عکس پایین هم قبل هم با رفتن امکان پذیر است.

✓ ⇐ ما ۲ تا عدد نرمال با توان های مختلف داریم ⇐ باید توان ها را یکی کنیم ⇐ بهتر است عدد کوچکتر (توان) را با عدد بزرگتر برسانیم چون ممکن است ما جا برای نمایش بیت ها نداشته باشیم و مجبور شویم یک بیت را دور بریزیم که اگر توان کوچکتر را با توان بزرگتر برسانیم بیت کم ارزش دور ریخته می شود ولی در تبدیل توان بزرگتر با کوچکتر بیت پرارزش دور ریخته می شود.

⇐ پس در جمع و تفریق ⇐ بعد از یکسان سازی توان ها و انجام عمل جمع ⇐ نرمال سازی انجام می دهیم

برای ضرب ⇐ مانتیس ها را در هم ضرب و توان ها را با هم جمع می کنیم ⇐ نرمال سازی در آخر

" تقسیم ⇐ " " " بر " تقسیم و " " از هم کم می کنیم ⇐ " "

$1,648 \times 10^{1}$ ۳ دور ریخته می شود

$4,59 \times 10^{2}$ (در تبدیل توان یک با ۲)

✓ با قرار دادن ۳ بیت درست راست این نمایش های یک (برای تبدیل توان ۲ با یک) دور ریخته می شود می توان انواع محاسبات را انجام داد.

که ما مثلاً وقتی توان های ما مثلاً ۱۳ تا ۱ اختلاف دارند مجبوریم ۱۲ تا بیت از ۲۴ بیت را دور بریزیم (آیا نیاز نیست این ۱۲ تا بیت را در واحد محاسباتی حفظ کنیم یا در واقع ۴۶ بیت در واحد محاسباتی داشته باشیم؟ IEEE می گوید با ۳ بیت همه چیز مشخص می شود و نیازی به این کار نیست.)

۹۲/۷/۱۵

معماری مجموعه دستورات (ISA) Instruction set Arch.

SW
ISA
HW

همیشه مراحل طراحی از بالا به پایین است.

⇐ اطلاعاتی از سخت افزار بدون جزئیات سخت افزاری (پیاده سازی) در اختیار برنامه نویس قرار می دهد تا کد خود را بسازد.

یعنی همیشه یک application را می یابیم و برنامه آن را ایجاد می کنیم و سپس با ISA سخت افزار را می سازیم.

Classic

NOTE BOOK Year: Month: Date:

① کلاس ISA

Ⓘ Register_Memory *

Ⓘ Load_store *

هر دستور ۲ تا بخش دارد ← عملیاتی که باید انجام می‌دهد یعنی (opc) operational code

(opnd) operand ← عملیات روی چه داده‌ای است.

می‌تواند مستقیماً یک خانه‌ی حافظه باشد. operand

Ⓘ ⇒ سری x۸۶ ⇒

add [bx], ax

محتویات ax را با خانه‌ی حافظه با آدرس bx جمع کرده حاصل و نتیجه را در همان خانه‌ی حافظه قرار می‌دهد.

Ⓘ ⇒ تنها دستورات کار با حافظه Load و store است.

خواندن از حافظه

نوشتن در آن

② انواع داده‌ها:

صحیح: بایت، کلمه، دوکلمه‌ای، چهارکلمه‌ای

اعشاری: دقت ساده، دقت مضاعف

۳۲ تا رجیستر در MIPS

R0 - R31

F0 - F31 ⇒ رجیستر float

برای کارهای ممیز شناور ما

③ انواع دستورات عملیاتی:

add, sub, and ...

④ انواع دستورات کنترل جریان برنامه (control flow) ⇒ به طور معمول، پردازنده دستورها را به صورت ترتیبی دستورات را انجام می‌دهد ولی شاید ما بخواهیم پرش داشته باشیم به جای دیگر کد.

پرش شرطی، بدون شرط، فراخوانی تابع، بازگشت از تابع

⑤ سازماندهی حافظه و نحوه‌های آدرس‌دهی حافظه که یعنی چگونه دسترسی به operand های حافظه چگونه باشد. یعنی چگونه آدرس operand را از حافظه بیابیم

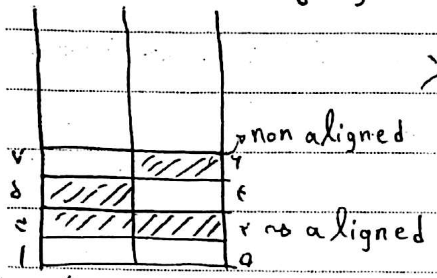
نحوه‌ی دسترسی به حافظه، آدرس‌دهی بایت/کلمه ← [1000] یا [bx] یا [bx+4]

Classic ← همه‌ی موارد بالا به صورت تصمیم‌گیری در سطح بالا هستند و بجز نیاز به سخت‌افزاری

مثلاً add یا بازگشت از تابع یا ... کاری نداریم.

خونه‌های حافظه ←

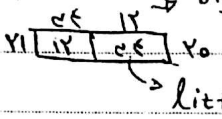
آیا پردازنده ما را مجبور می‌کند که بصورت aligned از حافظه بخوانیم یا نه؟ (اگر aligned باشد باید از زوج‌ها بخوانیم نه فرد.)



little endian

Big endian

big ← یعنی قسمت کوچک در خانه‌ی بزرگتر قرار گیرد.



۱۲۳۴

little ← Mips با این گونه است.

۴) روش Encoding دستورات

* variable length

* Fixed length ← یعنی مثلاً همیشه fetch ها ۳۲ بیتی باشند نه متغیر که مناسب‌تر است.
که کارهای عملیاتی ما

دو نوع اصلی پردازنده

Complex Inst. set computer

CISC

* Reg_Mem
* دستورات پیچیده
* مودهای آدرس دهی متنوع و پیچیده
* تعداد رجیسترها کم

Reduced Inst. set computer

RISC

* load_store
* دستورات ساده و پایه‌ای (هاردور ساده)
* مودهای آدرس دهی ساده و محدود
* تعداد رجیسترها زیاد ← چون دسترسی مستقیم با حافظه نداریم

در طول تاریخ، مقبولیت و موفقیت بیشتری داشته است.

NOTE BOOK Year: Month: Date:

معماری مجموعه دستورات MIPS (که از نوع RISC است):

✓ ۳۲ رجیستر ۳۲ بیتی برای اپرند دستورات محاسباتی داریم.

هرکدام از R_0 تا R_{31} را می‌توانیم در oprand ها قرار دهیم

add R_1, R_2, R_3 ⇒ $R_1 = R_2 + R_3$

sub R_1, R_2, R_3 ⇒ $R_1 = R_2 - R_3$

R_1: مقصد، R_2: اپرند اول، R_3: اپرند دوم

* رجیستر R0 همواره مقدار ثابت 0 را دارد. (عملاً این رجیستر در دسترس نیست)

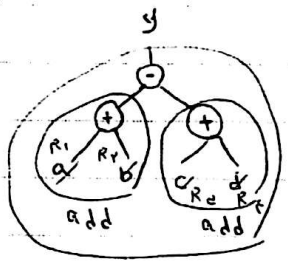
add R0, R1, R2 ⇒ error نمی‌دهد ولی کاری هم انجام نمی‌دهد.

وابسته با نوع ماشین = صفر و یک های مخصوص خود را دارد.

= اسمبلی، عملاً تفاوتی با زبان ماشین (صفر و یک) ندارد فقط برای سادگی صفر و یک ها را تبدیل به یک سری نماد کرده‌اند که کار با آن ساده‌تر است.

compiler این کد سطح بالا ما را به اسمبلی تبدیل می‌کند.

کامپایلر = اول syntax ها را چک می‌کند که با دستورات C مثلاً `y = (a+b) - (c+d);` مشکلی نداشته باشد = error سپس آن را به صورت یک درخت گرافی برای یک inst. درمی‌آورد.



a به R1 map شده روی

$y = (a+b) - (c+d);$

R_3 R_1 R_2 R_4 R_5

$y = (a+b) - (c+d)$

R10 R11

```
add  R10, R1, R2
add  R11, R3, R4
sub  R5 , R10, R11
```

این ۳ تا جایگزین دستوران C شد.

سر خود کامپایلر خواستش است که R10 و R11 در دونش چیزی نیست که پاک شود.

Move = در پردازنده‌ی MIPS نداریم:

mov R_1, R_2 ⇒ (عادلش در MIPS) add R_1, R_2, R_0

Classic

RISC Reduced Instruction set Complex وجه‌ترین با سخت‌افزار ساده

CISC Complex ~ ~ ~ رجیسترهای با سخت‌افزار پیچیده

= صورت = سخت‌افزاری که fetch و store و ... را انجام می‌دهد.

$2^{10} = K$
$2^{20} = M$, $2^{30} = G$

opcode می‌گوید کل ورد را بخوانیم یا یک Byte.

Year: Month: Date: NOTE BOOK

★ حافظه

این حافظه فقط آدرس های مضارب ۴ را می‌گیرد ولی مثلاً ما با بایت ۱۰ کار داریم که اگر ۱۰ را با آن بدهیم ۲ بیت سمت راست را معرفی کند و آدرس ۸ می‌شود

۳۲ خط آدرس

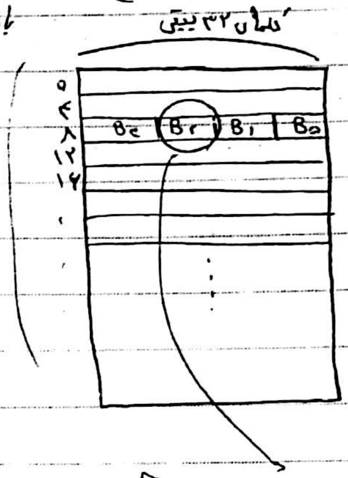
کلمات ۳۲ بیتی

$2^{32} B = 4 GB = 1 G$ word

و کل word ۸ را با ما می‌دهد و ما در دستور بعدی می‌گوییم که با صورت Byte کار می‌کند می‌گوییم با B_2 کار داریم.

0000 0000
0000 0004
0000 0008
0000 000C

0000 0000
0000 0100
0000 1000
0000 1100



داخل پردازنده

B_3 B_2 B_1 B_0

10d = 0000 1010
2^0 00

که ترکیب ۲ بیت سمت راست و opcod مشخص می‌کند که با کدام B کار کنیم.

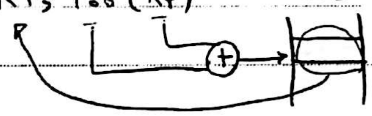
آدرس خانه‌ی حافظه = خودش می‌فهمد که ردیف ۸ برود و با این روش B_2 که ۱۰ است را می‌یابد.

10d

پس Byte و word addressable بودن را داریم.

load word محتویات خانه‌ی $100+R_2$ حافظه را در R_1 ذخیره می‌شود

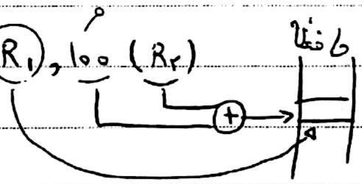
lw R_1, 100(R_2) حافظه



مثل یک base می‌ماند که از آن جا با اندازه‌ی R_2 بالا و پایین می‌شود

store word

sw (R_1), 100 (R_2) حافظه

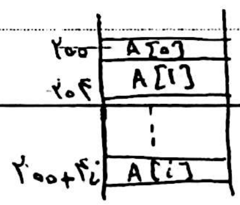


مثال: A یک آرایه‌ی ۱۰ عنصری با آدرس شروع ۲۰۰ است: $4i \to R_1$; $4j \to R_2$

$A[j] = A[i] + k$

R_{10}

R_{11}



Classic

NOTE BOOK Year: Month: Date:

ها چیز رجیستری است.

```
lw   R10, ۲۰۰(R1)
add  R11, R10, R2
sw   R11, ۲۰۰(R2)
```

ما مرA[i] را یک word گرفتیم شاید Byte بود (ولی ما چون داریم ساده برخورد می‌کنیم word گرفتیم)

دستورات پرش:

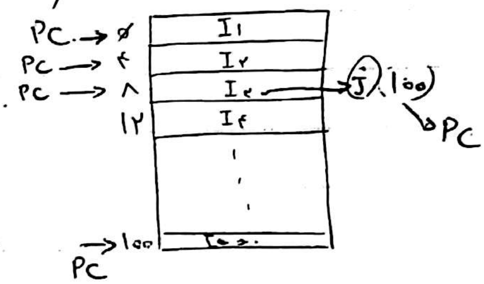
در MIPS (Instruction pointer) در بقیه پردازنده‌های دیگر (IP)

j adr

یعنی دستورات با ترتیب انجام می‌شود. ← Program sequencing

program counter (PC)

حاوی آدرس دستور بعدی است که باید از حافظه خوانده شود.



پرش شرطی:

branch equal → beq R1, R2, L1 → اگر R1 و R2 برابر بودند برو به L1 در غیر اینصورت برو به مرحله بعد

bne R1, R2, L1 => برو به L1 " " " برابر نبودند

branch Not equal

```
if (a==b)            bne R1, R2, Else
    St1;    معادلش ⇒      St1;
else                  j End_If
    St2;            Else: St2;
                    End_If:
```

(a = R1، b = R2؛ اگر برابر باشند ← St1؛ اگر نابرابر باشند ← Else)

می‌خواهیم دستورات بزرگ‌تر و کوچک‌تر را هم بتوانیم بررسی کنیم.

Classic

(۴۵)

داریم با کمترین تعداد دستوران پیاده سازی را انجام می دهیم.

Year: Month: Date: NOTE BOOK

slt R_1, R_2, R_3

set on less than ⇒ { if ($R_2 < R_3$) $R_1 = 1$ else $R_1 = 0$ } نسبت سمت چپ $R_2 - R_3$

مثال) if (a < b)   (R_1, R_2)
st1; ⇒ slt (R10), R_1, R_2 ← با 1 bit
else   if (R10 == R0)
st2;   st2;   این را قبلاً داشتیم.
else
st1;

مثال) if (a > b)   if (b < a)
st1;   st1;
else   else   ⇒ مثال قبل
st2;   st2;
تبدیل بزرگتر با کوچکتر

مثال) if (a ≤ b)   not آن   if (a > b)
st1;   st2;
else   else
st2;   st1;

برای بزرگتر مساوی هم همین طور است.

immediate

{ Jr R_1 — jump register } | addi R_1, R_2, data
| addi R_1, R_1, ۴   این را داریم
| addi R_1, R_1, -۴ همان subi

با کاراکتر R ۴ رجیستر 1 اشاره می کند برش

Classic

برای بازگشت از ⇒ می کند. تابع مفید است.

{ slti (slti) R_1, R_2, data }

عملیات با عدد ثابت انجام شوند.

NOTE BOOK

۲۰، ۷، ۹۲

فراخوانی تابع jump and link

jal adr.

آدرس برگشت تابع (PC+۴) در یک رجیستر (R31) ذخیره می‌شود.

```
        ~                  A: ~                 B: ~
        ~ Push R31            ~ Push R31           ~
1000  jal A           2000  jAl B               return
        ~ PoP R31             ~ PoP R31
        ~                     ~
                            return    <----  Jr R31
```

R31 | ۱۰۰۴ | ۲۰۰۴

آدرس برگشت از دست رفت.

بهتر است از این استفاده نکنیم تا تابعی دیگر هم را صدا نزد و ...

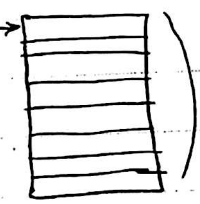
ایراد: نمی‌شود تابعی را داخل تابع دیگر فراخوانی

پس بهترین راه برای حفظ آدرس برگشت، استفاده از stack است.

نوع ساختار داده

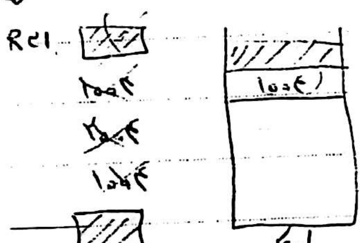
MIPS در R29 ⇐ که SP (stack pointer)

که آدرس آخرین خانه‌ی پر استک را نشان می‌دهد.



⇐ برای Push کردن آدرس stack P را ۴ واحد اضافه می‌کنیم تا با اولین خونه‌ی خالی برسیم. (برای PoP کردن، برعکس است)

stack بخشی از حافظه‌ی اصلی است.



Push { addi R29, R29, ۴ ; sw (R31), Ø(R29) } — store word — Push ریختن روی استک

PoP { lw (R31), Ø(R29) (load) ; addi R29, R29, -۴ } — PoP برداشتن از ""

Classic

دستور Push یا Pop ندارد بلکه Push و Pop معادل ۲ خط کدمن قبل است.

Year: Month: Date:

NOTE BOOK

ارسال پارامتر با تابع:

① استفاده از رجیسترها

```
lw   R4, 100(R20)
lw   R5, 104(R20)
jal  Func
```

⇐ این در صورتی است که پارامترهای ما کم است. (در غیر این صورت از ② استفاده می کنیم)

② استفاده از استک

گذاشتن پارامتر اول (R20) روی استک

```
Push R20
"    R21
"    R22
"    R24
"    R25
Jal  Func
```

(نوشتن R31 روی استک)

برای پارامتر اول

```
Func: lw R1, -20(R26)
```

برای پارامتر دوم

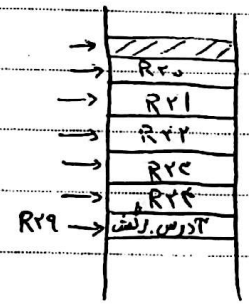
```
lw R2, -16(R26)
```

خواندن آدرس برگشت

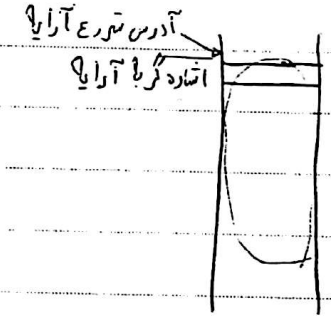
```
addi R29, R29, -20
```

خالی کردن پارامترهای تابع از روی استک (یا Pop کردن)

چون دیگر نیازی به آن ها نیست.



مثلاً برای خواندن یک Structure



Classic

NOTE BOOK Year: Month: Date:

① لود کردن داده ۱۶ بیتی در یک رجیستر :

← این همینا از ۱۶ بیت بیشتر نیست (در دستور addi)

R1 : | ۰۰۰۰ | ۱۲۳۴ |

addi R1, R0, ۱۲۳۴

| ۰۰۰۰ | ۰۰۰۰ |
۱۲۳۴
R1 | ۰۰۰۰ | ۱۲۳۴ |

② لود کردن داده ۳۲ بیتی در یک رجیستر :

R1 : | ۴۵۶۷ | ۱۲۳۴ |

دستور جدید → lui R1, (داده ۱۶ بیتی) ← load upper immediate → R1 | | ۰۰۰۰ |

بنابراین →
{ lui R1, ۴۵۶۷
{ addi R1, R1, ۱۲۳۴

R1 | ۴۵۶۷ | ۰۰۰۰ |
R1 | ۴۵۶۷ | ۰۰۰۰ |
۱۲۳۴
R1 | ۴۵۶۷ | ۱۲۳۴ |

دستورات منطقی :

این ها همه ی دستورات MIPS نیست (ما با آن هایی که ماس (پردازنده) (مجموعه) ...)

این کار را داشتیم با شیفت را آورده ایم

{ and R1, R2, R3 R1 ← R2 & R3
{ or R1, R2, R3 R4 ← R2 | R3
{ andi R1, R2, (داده ۱۶ بیتی)
{ ori R1, R2, (")

Shr R1, R2, تعداد بیت
Shl ~

دستورات ضرب / تقسیم :

علامت دار
{ mult R1, R2
{ multu R1, R2 ← unsigned ← بدون علامت

{ div R1, R2
{ divu R1, R2

Classic

Year: Month: Date: NOTE BOOK

۲ رجیستر ۳۲ بیتی برای ذخیره‌سازی نتایج حاصل از دستورات ضرب / تقسیم. (hi / lo)

در تقسیم: خارج قسمت در hi و باقیمانده در lo قرار می‌گیرد.

در ضرب: حاصلضرب را قسمت ۳۲ بیت بزرگش را در hi و ۳۲ بیت کوچکش را در lo ذخیره می‌کند.

چون ۳۲ بیت را بر ۳۲ بیت تقسیم و خارج قسمت ۳۲ بیتی است overflow نداریم.

دستوری نداریم که یکی چیزی را مستقیم در hi و lo قرار دهیم و فقط حاصل ضرب و تقسیم در آن‌ها قرار می‌گیرد.

```
mult   R10, R11
```

* ~~add R1, R2, R0~~ ⇐ چون فرمت instruction را بهم می‌ریزد

دستورات جدید:

```
mflo   R20
mfhi   R21
add    R1, R2, R20
```

دستورات میز شناور (۳۲ رجیستر غیر از رجیسترهای hi و lo منظوره) F0 - F31

✓ فقط شماره های زوج از این رجیسترها قابل استفاده هستند

میز شناور با دقت ساده

add.s → add.s F2, F4, F6

میز شناور با دقت مضاعف (d.)

F4 (۳۲ بیتی)

F6

F2

۶۴ بیتی

add.d F4, F5

F6, F7

F2, F3

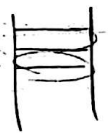
برای میز شناور: sub, mult, div و ... lw.s, lw.d, sw.s, sw.d هم داریم.

Classic

NOTE BOOK Year: Month: Date:

lw.s F۲, ۱۰۰(R۱) ← این در F۲ ذخیره می‌کند

lw.d F۴, ۱۰۰(R۱) ← ۲تای پشت سرهم رادر F۴ F۵ ذخیره می‌کند.



مجموعه دستورات انتخابی: (برای طراحی پردازنده ی MIPS در این کلاس ما این ها فقط نیاز داریم)

① R-type : add , sub , and , or , slt

② Memory type : lw و sw

③ controlFlow : beq , j

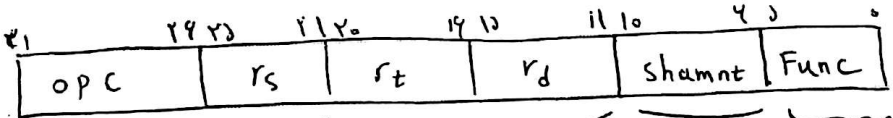
✓ همان دستورات ما ۳۲ بیتی است ، تفاوت دستورها با opcode مشخص می‌شود.

(دل در R-type ما opc ۶ بیت صفر است ← پس چگونه نوع عملیات را تشخیص دهیم؟ از ۶ بیت ۰ تا ۵ استفاده می‌کنیم ، ابتدا opc را خوانده می‌بینیم R-type است سپس از ۰ تا ۵ نوع دستور را می فهمد))

R-type: مثلاً: add rd , rs , rt

rd ← رجیستر مقصد ، rs ← اپرند اول ، rt ← اپرند دوم

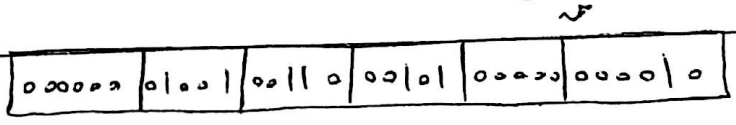
instruction format



opc: نوع دستور (۰۰۰۰۰۰ برای R-type) — rs: اپرند اول (نماره ی رجیستر) — rt: اپرند دوم — rd: رجیستر مقصد — shamnt: مقدار شیفت — Func: نوع عملیات دستور R-type

add R۵ , R۹ , R۴

زبان ماشین ← Classic



مثلاً یعنی add ←

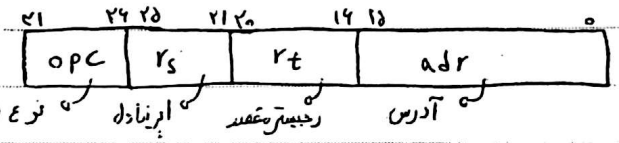
Year: Month: Date: NOTE BOOK

۹۲/۷/۲۲

قالب دستورات:

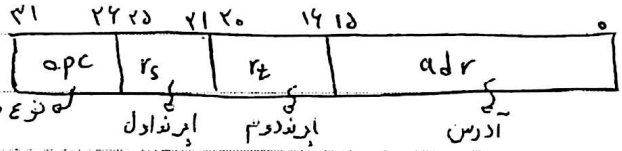
② دستورات Mem. Ref:

lw r_t, adr(r_s)



۶ نوع دستور، تقسیم‌بندی‌های بعدی را مشخص می‌کند.

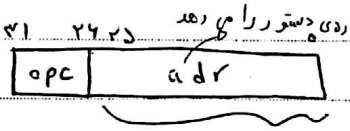
sw r_t, adr(r_s)



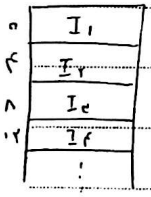
opc در lw و sw با هم متفاوت است و هردو با R-type متفاوت است.

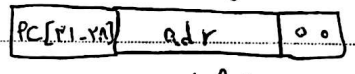
③ دستورات Control Flow:

j adr

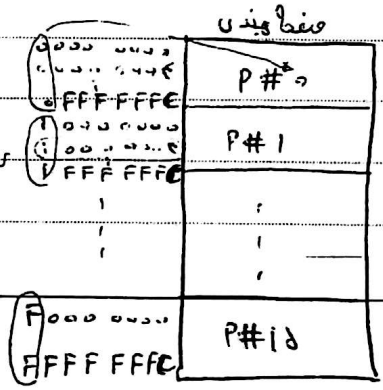


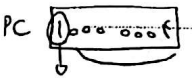
* نکته: آدرس‌ها باید مضربی از ۴ باشند ⇐ پس هر بار ۴ نیاز داریم دیگر ۲ تا شیفت به چپ.





* نکته: در این دستور، پرش در داخل صفحه‌بندی انجام می‌شود.





صفحه‌ی جاری انجام می‌شود.

پس بازهم ۳۲ بیتی شدیم.

Classic

دگر: حافظه یکپارچه است ⇐ تقسیم‌بندی ماست.

NOTE BOOK — Year: Month: Date:

beq $rs, $rt, adr

opc	rs	rt	adr
۶ بیت	۵ بیت	۵ بیت	۱۶ بیت

✓ تعداد دستوراتی که باید از دستور بعدی یا جلو یا عقب پرش کنیم.

آدرس پرش

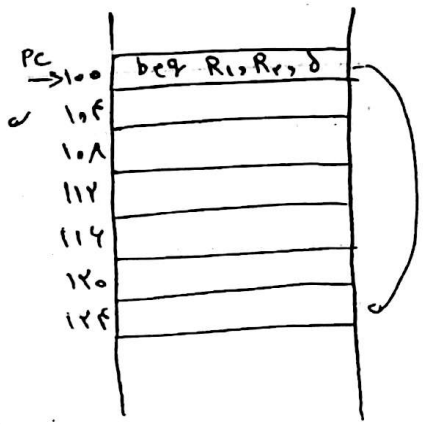
PC + ۴ + adr << ۲

adr ۲ بار شیفت به چپ

۱۰۰ + ۴ + ۵ * ۴ = ۱۲۴

لذا برای پرش به عقب باید adr منفی باشد.

(درواقع PC relative هستیم)

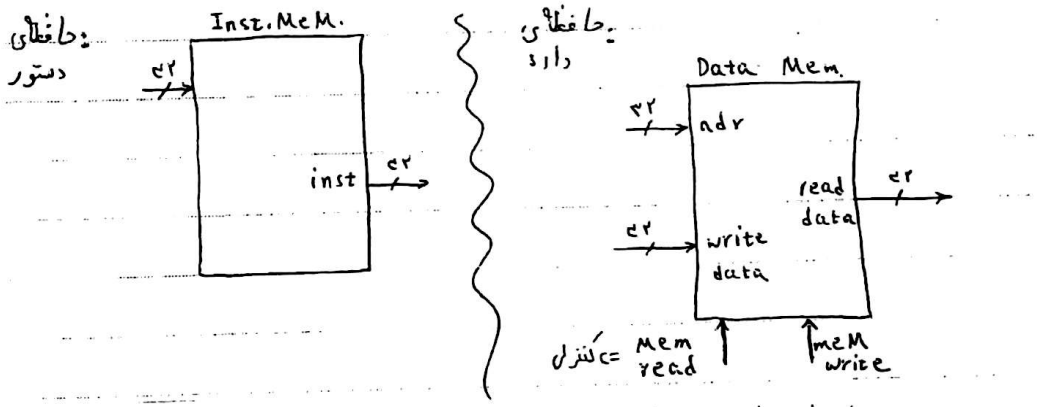


برای طراحی پردازنده، باید موارد بالا را کاملاً آشنا باشیم.

طراحی پردازنده:

① طراحی مسیر داده: (در ابتدا فرض ما این است که حافظه‌ی داده و دستور کاملاً جدایی از هم هستند)

لذا اجزایش را معرفی می‌کنیم.



لذا برای write ← adr اول ← write بعد data

Mem write بعد

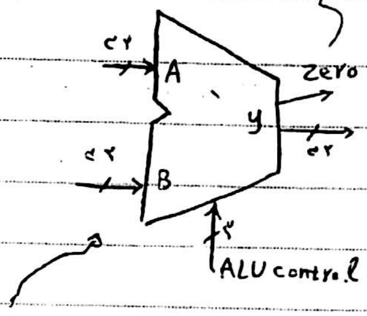
برای read ؛ ① adr ← Mem read بعد ← read data بعد

Classic

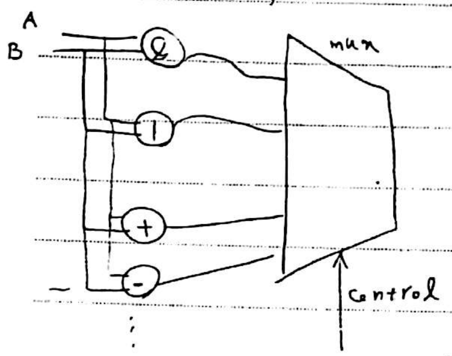
Year: Month: Date: NOTE BOOK

برای beq استفاده می‌شود.

★ALU



ALU Control	عملیات
۰۰۰	A & B
۰۰۱	A | B
۰۱۰	A + B
۰۱۱	A - B
۱۱۱	A slt B



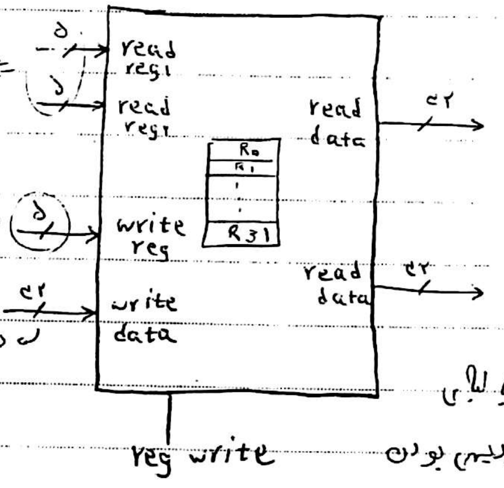
zero = 1 if y = 0

zero = 0 otherwise

★ Register file

دستورات ۲ رجیستر می‌خواهند

⇐ آدرس ۲ تا رجیستر را می‌دهیم و طبق عملیاتی که قبلاً داشتیم آن را انجام می‌دهیم



← داده‌ای که می‌خواهیم بنویسیم.

همان این write ها و سیگنال‌های clk که را می‌گذاشتیم بدیهی بودن آن را نشان نداده‌ایم.

⇐ Component های بالا عاملی هستند و با هیچ‌جا قابل تغییر نیستند.

طراحی مسیر داده: که ما هر قسمت را جداگانه طراحی کرده و باهم در انتها ادغام می‌کنیم.

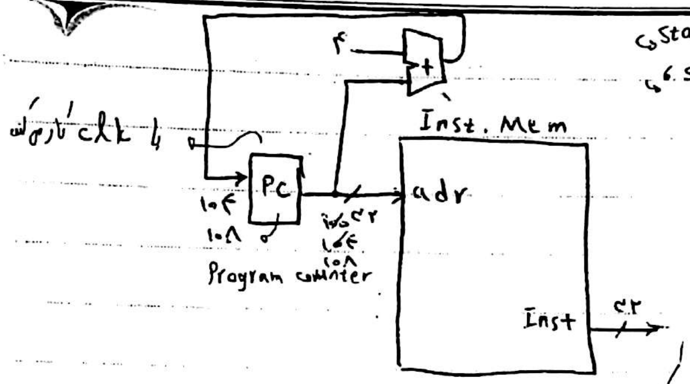
Classic program sequencing مسیر داده مربوط به توالی‌دهی دستورات (۱-۱)

← در این جا فقط می‌خواهیم دستورات را با ترتیب بخوانیم.

۵۴) پردازنده ی MIPS ⇐ ۳ آپرندی است ⇐ ولی مثلاً اینتل ۲ آپرندی است : add (an) bn که مقصد هم همین است.

یک آدرسه : add 100 ⇐ با آکومولاتور مثلاً جمع می کند و در آکومولاتور هم می نویسد.

صفر آدرسه : add ⇐ ۲ تا را از stack ، pop می کند و سپس نتیجه را در stack ، push می کند.



۱-۲) مسیرداده مربوط به اجرای دستورات R-T : (در این جا ما کاری با اینکه دستور چگونه ایجاد شده نداریم بلکه فقط می دانیم ورودی ما یک دستور ۳۲ بیتی است)

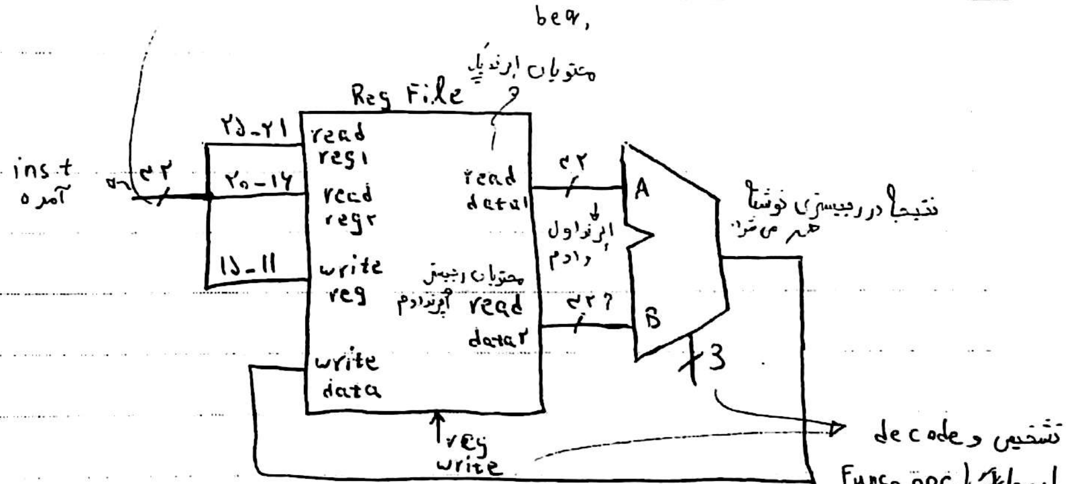
یادآوری ⇐

RT:

۳۱–۲۶	۲۵–۲۱	۲۰–۱۶	۱۵–۱۱	۱۰–۶	۵–۰
opc	rs	rt	rd		func

lw, sw, beq,

۳۱–۲۶	۲۵–۲۱	۲۰–۱۶	۱۵–۰
opc	rs	rt	adr

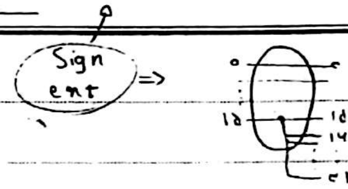


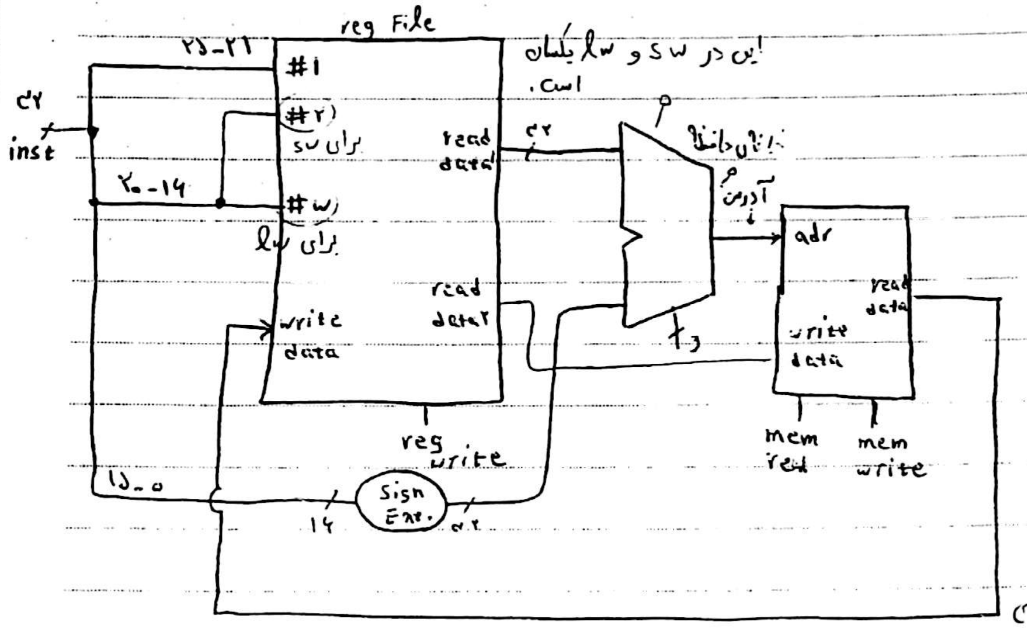
(در رجیستر ۳۲ بیتی بالا) در واحد کنترل انجام می شود که بعداً نحوه ی ساخت این ها را می بینیم.

۱-۳) ادغام مسیر داده دو مرحله ی قبل که با سبز در بالا نشان داده شده است.

Classic ۴-۱) مسیرداده مربوط به اجرای دستورات Mem. ref :

Hw کل wiring





دقیقاً معادل نحوه عملکرد آن در رجیستر ۳۲ بیتی است.

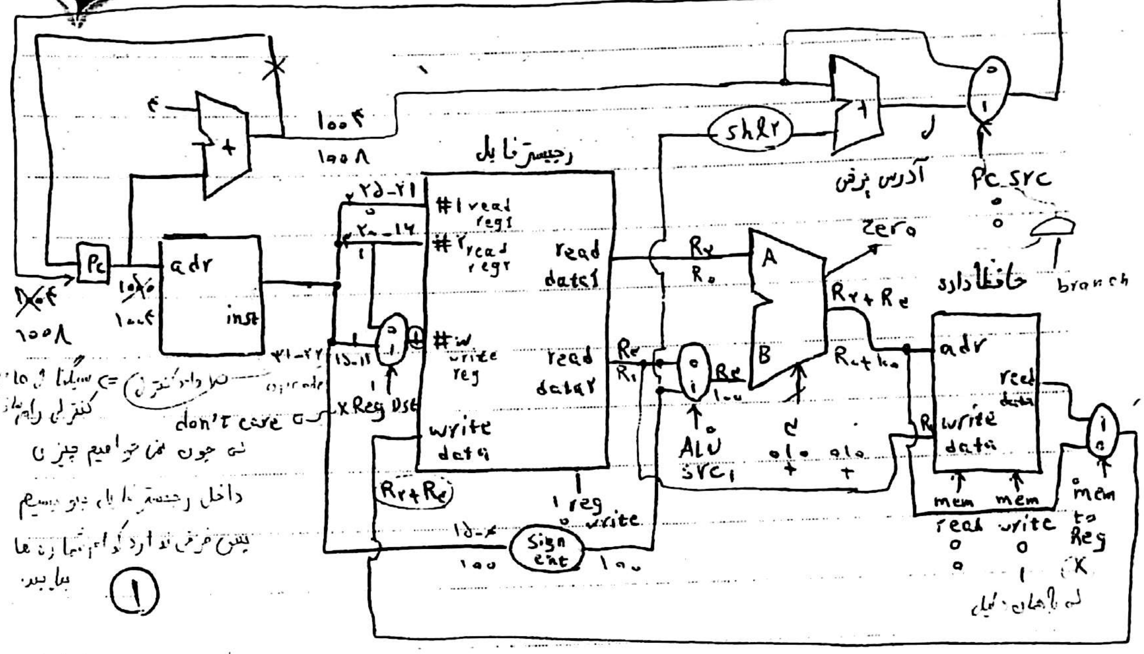
در lw : ۲۱_۲۵ را در read data1 داریم در ALU ، آدرس خانه‌ی حافظه ساخته می‌شود سپس از حافظه می‌خوانیم و در آدرس ۱۶_۲۰ (رجیستر مقصد) نتیجه را در write data می‌نویسیم.

در sw : تا ساخت آدرس خانه‌ی حافظه یکسان است ولی برای این‌که از کجا بخوانیم ۱۶_۲۰ در #2 نیاز است و ... بنابراین از رجیستری که ۱۶_۲۰ ، #2 نشان می‌دهیم می‌خوانیم و با read data2 در حافظه می‌نویسیم. (مسیر سبز)

(۱_۵) ادغام ۲ مرحله‌ی قبل (درس بعد)

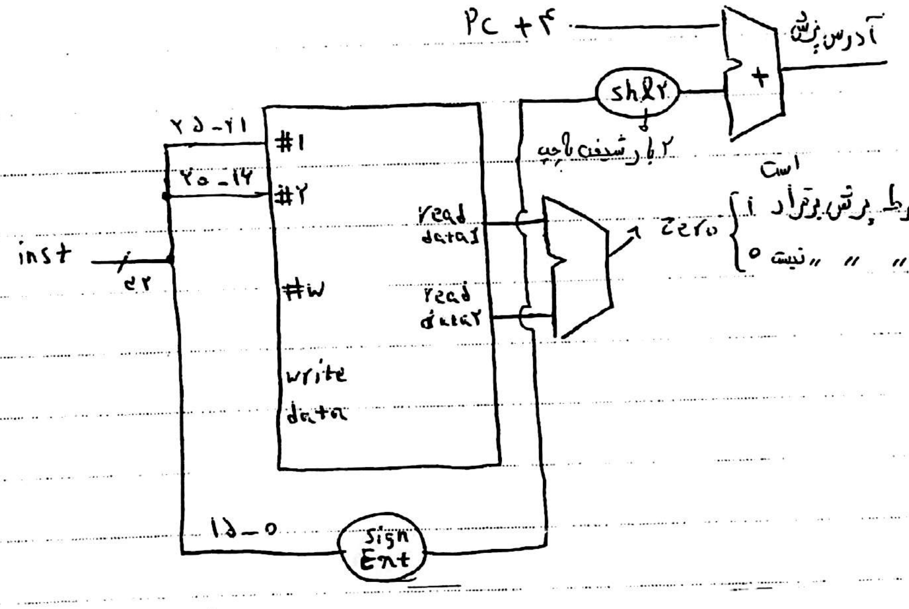
reg Dst (register Destination)
← آدرس مقصد

ALU source



۱

۴-۱ مسیر داده مربوط به اجرای دستور beq : (فعلاً آن یکی دستور را بی‌خیال می‌شویم)



ادغام با مسیر در بالا

Year: Month: Date:

NOTE BOOK

✓ شکل ① من قبل ، پردازنده ی single cycle می باشد با این معنی است که در یک clk cycle مرحله مراحلش انجام می شود ولی در جلسه ی بعد توضیحات کاملش را می بینیم .

۹۲/۷/۲۷

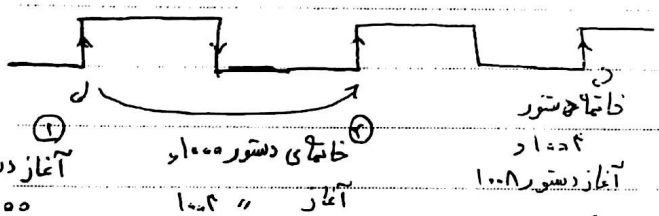
بررسی چند دستور در این پردازنده :

دستور R-type

۱۰۰۰	add R1,R2,R3
۱۰۰۴	sw R1,100(R2)

۱۰۰۰	۰۰۰۰۰۰	۰۰۰۱۰	۰۰۰۱۱	۰۰۰۰۱	—	—
۱۰۰۴	sw	۰۰۰۰۰	۰۰۰۰۱	۱	؟	؟

← مسیر این دستور در شکل ① من قبل مشخص شده است . (با خودکارهای سبز و مشکی)

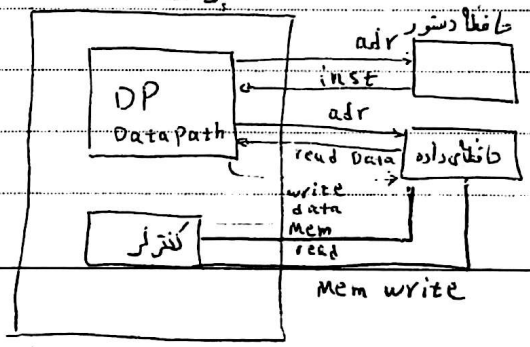


← مطابق نمودار بالا هر دستور در یک cycle انجام می شود .

بنابراین با آن single cycle می گویند (که بعداً می گوییم مشکلاتی دارد)

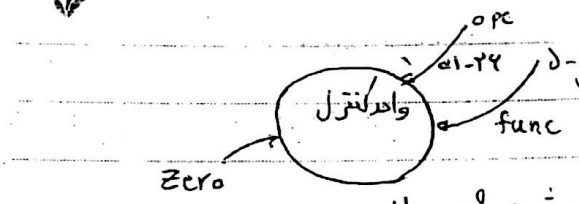
← واحد کنترل combinational می باشد و در واقع حافظه برای خود ندارد .

← بنابراین حافظه ها را Datapath یا پردازنده جدا است .



Classic

NOTE BOOK Year: Month: Date:



برای تشخیص مثلاً add یا ...

پس ⇐ ۱۳ تا ورودی دارد پس کنترل واردم شود و باهمین دلیل ؛ واحد کنترل دارد بزرگ می شود و ولی من میبینم که فقط سیگنال های ورودی های ALU control و func و zero نیازدارد و همچنین Pc src هم از zero استفاده می‌کنند ول بقیه‌ی سیگنال‌های کنترل این‌گونا نیستند پس می توان واحد کنترل را کوچک نمود.

تولید Pc src :

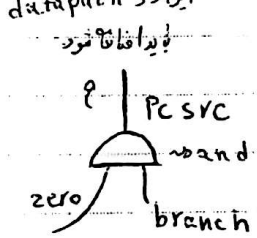
① دستورات نوع beq : Pc src = ∅ ✓ branch = 0

② دستور beq : (۲-۱) شرط پرش برقرار نباشد ؛ Pc src = ∅ ✓ zero = 0
branch = 1

(۲-۲) " " " باشد : Pc src = 1 ✓ zero = 1

⇐ پس Pc src را با یک واسطه می سازیم ؛

واحد کنترل یک سیگنال کنترلی با نام branch تولید می کند.

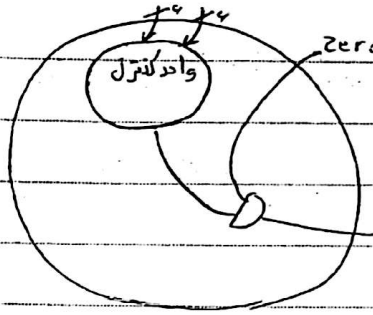


① اگر دستور beq باشد : branch = 1

② " " " نباشد : branch = ∅

Classic

Year: Month: Date: NOTE BOOK



بنابراین واحدکنترل بزرگ را دوبخش کردیم ⇒ تا واحدکنترل کوچک ما ورودی های کمتری داشته باشیم تا حالت های don't care کمتری داشته باشیم و واحدکنترل ساده شود.

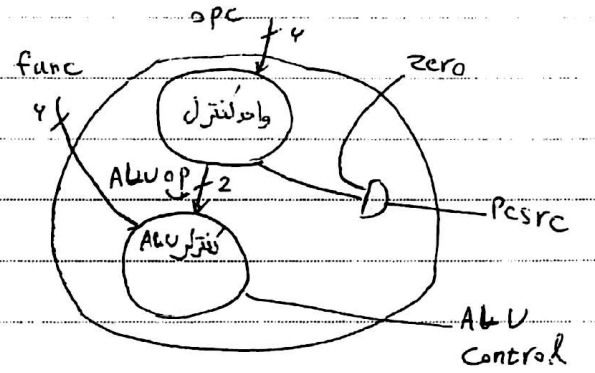
هم اکنون می خواهیم ۶ ورودی func را هم جدا کنیم:

ALU op

① اگر دستور Mem Ref باشد: عملیات ALU ⊕ را انجام می دهد. ۰۰

② " " beq " : " " ⊖ " " " ۰۱

③ " " R-T " : " براساس func عملیات را انجام می دهد ۱۰



کنترل ALU

	ALU op	func	ALU Control
①	۰۰	xxxxxx	۰۱۰ ⇝ ⊕
②	۰۱	xxxxxx	۱۱۰ ⇝ ⊖
③	۱۰	add	۰۱۰
		sub	۱۱۰
		and	۰۰۰
		or	۰۰۱
		slt	۱۱۱

در این تار debuge مدار را ساده تر می کنیم چون نقش بخش کنترل را طراحی کرده ایم.

Classic

در sw ⇐ چیزی داخل Register file نمی‌نویسیم پس Mem to Reg و Reg Dst یا آدرس Register مقصد را مشخص نمی‌کند X (don't care) می‌شوند

حال واحد کنترل کوچک اصلی را باید طراحی کنیم. چون در R-type کاری با حافظه نداریم.

opc	Reg Dst	Reg write	ALU src	Mem read	Mem write	Mem to Reg	branch	ALU op
R-T	1	1	ø	ø	ø	ø	ø	1ø
lw	ø (۲۰-۱۶)	1 (چیزی که از حافظه خوانده ایم را باید در rt بنویسیم)	1	1 (از حافظه می‌خوانیم)	ø (چیزی در حافظه نمی‌نویسیم)	1	ø	øø
sw	X	ø	1	ø	1	X	ø	øø
beq	X	ø	ø	ø	ø	X	1	ø1
addi	ø	1	1	ø	ø	ø	ø	øø

اگر دستوری اضافه شود به پردازنده‌ی ما ⇐ باید ابتدا Datapath آن دستور را تولید کنیم و با پردازنده ادغام کنیم ببینیم چیزی به DP اضافه می‌شود یا نه ولی در کنترلر باید حتما بیت ایجاد آن دستور را اجرا کنیم.

مثال → addi rt , rs , data $r_t = r_s + data$

۳۱ - ۲۶	۲۵ - ۲۱	۲۰ - ۱۶	۱۵ - ۰
opc	rs	rt	data

ø ⇐ تبدیل به sign ext data و rt به ALU می‌رود

همان مسیرهای موجود برای اجرای این دستور در DP ما موجود است.
با فراخوانی کنترلی نحوه‌ی اجرای آن را ایجاد می‌کنیم:

مثال جدید → Push R1 ← محتویات R1 را بالای stack قرار می‌دهد.
PoP R2 ← بالای stack را در R2 قرار می‌دهد.
⇒ هر دستور یا Push یا PoP کار داریم از

۲ دستور ایجادشده ⇐ ما اجازه‌ی تغییر reg file یا ALU یا حافظه نداریم و فقط می‌توانیم mux یا shifter اضافه کنیم.

(۹۱) در دستور PoP هر ۲ ی نوشتن ها در رجیستر انجام می شود پس در یک clk قابل انجام نیست.

در دستور PoP
$\begin{cases} R_2 \leftarrow M[R29] \\ R29 \leftarrow R29 - 4 \end{cases}$ و ⇒ مانند Push نیست.

ولی در دستور Push یکبار در رجیستر می نویسیم یکبار در حافظه

PoP R₂ ⇒ امکان اجرا؟

نه ؟ هیچ وجا وجود ندارد چون یکبار باید محتویات حافظه را برداریم و سپس آدرس را ۴ واحد کم کنیم ⇐ بنابراین در یک cycle قابل اجرا نیست. (دستور swap هم همین گونه است) ⇐ چون باید هر ۲ مقدار در رجیستر قابل نوشتن شود (هم R₂ و هم R29) که چنین یادی چون یک مسیر برای write reg در رجیستر نداریم و همچنین حق نداریم رجیستر فایل را تغییر دهیم در یک کلاک قابل اجرا نیست.

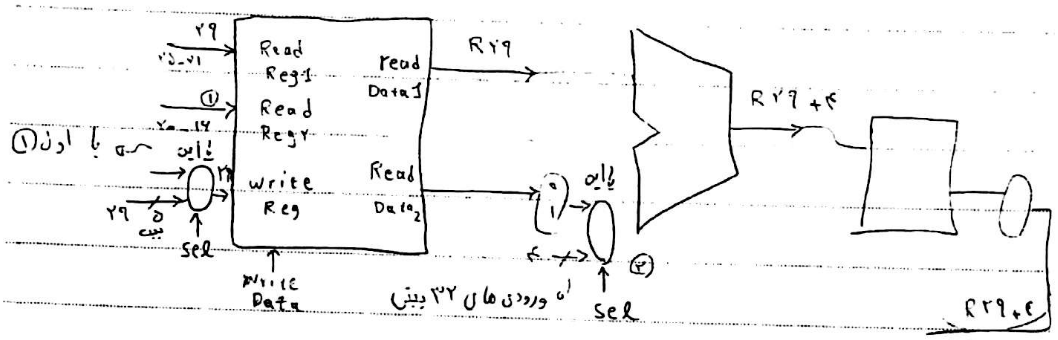
دستور Push با کلاک ر وبا رو قابل اجرا است:

Push R₁

$\begin{cases} R29 \leftarrow R29 + 4 \\ M[R29+4] \leftarrow R_1 \end{cases}$

هم زمان اتفاق می افتد

	۳۱ ... ۲۶	۲۵ ... ۲۱	۲۰ ... ۱۶	۱۵ ... ۰
①	opc	۲۹ (R29)	① (R1)	۴



	۳۱ ... ۲۶	۲۵ ... ۲۱	۲۰ ... ۱۶	۱۵ ... ۱۱	... ۰
② با دوم ⇒	opc	۲۹	①	۲۹	

نه اولی بهتر است چون mux آن ورودی های ۵ بیتی دارد ولی دومی ورودی های ۳۲ بیتی دارد.

⇐ برای هر دستوری همین کار را می توان کرد ⇐ همین دستور Push با عنوان تمرین سعی شود اجرا شود.

⇐ در جدول ص قبل گزینه ی sel هم در ص قبل در آخر جدول اضافه می شود

Classic

✓ خواندن از رجیستر آسان است - نوشتن سخت است.

NOTE BOOK Year: Month: Date:

معیارهای کارآیی پردازنده: ← ابتدا کارآیی پردازنده را بررسی می‌کنیم و می‌بینیم که از نظر single cycle کارایی بالایی ندارد و با سراغ multi cycle و پایپ‌لاین می‌رویم که می‌بینیم سرعت کار ما بالا می‌رود.

۹۲/۷/۲۹: ← معیارهای ما، برای قسمت پردازش است و دیگر با IO ها مثلاً حافظه‌ی کندتر کاری نداریم و فقط CPU را فعلاً بررسی می‌کنیم.

* زمان اجرا: مدت زمان بین شروع و خاتمه‌ی یک کار (Excutation time)

* کارایی: عکس زمان اجرا — Performance

$$P = \frac{1}{Ex.time}$$

* ماشین X، n برابر سریعتر از ماشین Y است.

$$\frac{P_x}{P_y} = n$$

$$\frac{\frac{1}{Ex\ time_x}}{\frac{1}{Ex\ time_y}} = n \Rightarrow \frac{Ex\ time_y}{Ex\ time_x} = n \Rightarrow \begin{cases} Ex\ time_x = 5 \\ Ex\ time_y = 10 \end{cases} \Rightarrow \frac{10}{5} = 2$$

* زمان CPU: زمانی که پردازنده صرف اجرای کار پردازشی می‌کند و شامل موارد زیر می‌شود (user CPU time ← کاربر)

باید زمانی که خود سیستم عامل را هم اجرا می‌کند و CPU فرامین سیستم عامل را انجام می‌دهد را هم باید کم کنیم.

۱- انتظار برای IO

۲- " برای دسترسی به حافظه

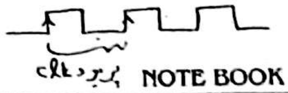
۳- زمان اجرای سربار سیستم عامل

* مقایسه ۲ ماشین ← یک برنامه‌ی واحد روی هر ۲ ماشین اجرا می‌شود
" ۲ الگوریتم ← هر ۲ روی یک پردازنده‌ی واحد باید انجام شود
} ← تا قابل مقایسه شوند.

ما در واقع یک برنامه‌ی واحد ساده را روی هر ۲ اجرا می‌کنیم یا مجموعه‌ای از برنامه‌های bench mark محک) ی جدی و واقعی را روی ۲ ماشین اجرا می‌کند و با طور متوسط ۲ ماشین را مقایسه می‌کند و امروزه برنامه‌های محک با صورت یک استاندارد درآمده‌اند ← Classic

(۴۲) ⇐ البته پردازنده ی تا این جا یعنی ماشین های single cycle دستوراتش یک cycle طول می کشد ولی در پایپ لاین و مولتی سایکل این گونه نیست

که در عمل هم پردازنده ها ۲ مورد دوم است



user cpu time = # clk cycles * clk cycle

این معیار خوبی نیست چون ما نمی توانیم و برایمان سخت است که تعداد cycle ها را محاسبه کنیم

این با آزمایش بدست می آید.

بنابراین ⇐ CPU time = # Instructions * CPI * clk cycle

(# Instructions: تعداد دستورها) ①

CPI: متوسط تعداد سیکل های مصرفی در یک دستور

هدف ما طراحی ما کاهش cpu time است.

⇐ پردازنده CISC ⇐ چون دستوراتش پیچیده است ولی می گوید من تعداد دستورات را کم می کنم

"بنابراین در SISK هدف کاهش تعداد دستورات است".

که ولی پارامترهای دیگری هم وجود دارد که دیتاپس را پیچیده می کند که طول clk cycle را بالا می برد و CPI را هم افزایش می دهد.

⇐ RISK می گوید که من دستورات ساده تر ولی بیشتر را به کار می برم در این صورت تعداد دستورها زیاد شده ولی CPI و clk cycle آن کم شده است.

⇐ که بطور متوسط ولی که دقیق نشان داده شده است که RISK بهتر از SISK است در حالی که در بعضی از application ها ممکن است SISK بهتر باشد.

$$① = \left(\sum_{i=1}^{n} C_i * CPI_i\right) * clk\ cycle$$

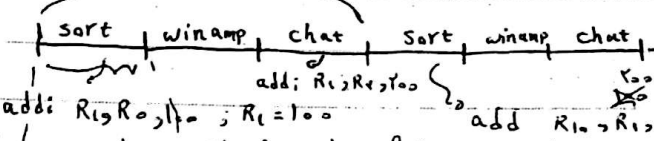
C_i: تعداد دستورات از کلاس i ام — CPI_i: CPI کلاس i ام

⇐ (۹ ، ۱۱) (۱ ، ۱۹) متوسط هر ۲ برابر ۱۰ است ولی ما برای اولی مناسب نیست پس دستورات را کلاس بندی می کنند که به رابطه ی بالا می رسیم.

⇐ پردازنده های ما بصورت multi task کار می کنند نه single task که مثلاً {چت، آهنگ، بازی} و ....

NOTE BOOK Year: Month: Date:

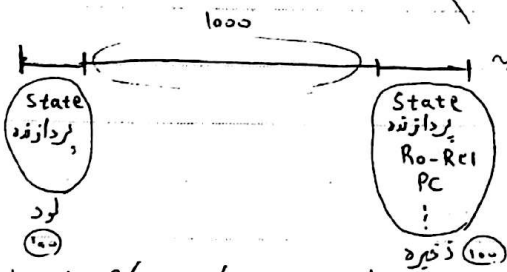
واحد زمان CPU



① چون همای دستوران از یک resorce استفاده می‌کنند این مشکل ⇐ رخ می‌دهد.

بنابراین زمان اجرای دستور مثلاً sort در multi task با Single task فرق دارد چون واسطش CPU کارهای دیگر نیز هم انجام می‌دهد.

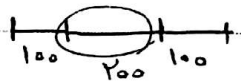
⇐ چون خیلی سریع بین task ها پردازه ها CPU را از یک task ای می‌گیریم و به task دیگری می‌دهیم ما وقفه ها را حس نمی‌کنیم ولی وقتی سر CPU شلوغ می‌شود این مثلاً در قطع و وصل شدن لوزیک متشهود است. ⇐ task switching



برای رفع مشکل ① از این سردستورهای غیرقابل تغییر استفاده می‌شود.

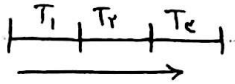
سردستور ۲۰٪ از زمان مفید اجرای دستور را با خود اختصاص داد.

⇐ بازدهی زمان اجرا: توان عملیاتی (Through put): تعداد کارهایی که در واحد زمان انجام می‌شود.

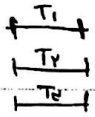


این جا سردستورها ۵۰٪ از زمان مفید را با خود اختصاص داد که مناسب نیست.

concurrent همروند



Simultaneous همزمان



در پردازنده‌های چندهسته‌ای این اتفاق می‌تواند رخ دهد.

پس از معرفی عملیات‌ها داریم:

فرض:

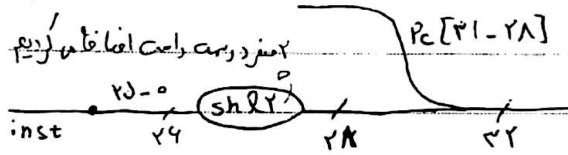
تأخیرها، ALU و حافظه‌ها: 2^{ns} — بیرون کشیدن دستور از حافظه (دستور) ← واکشی Fetch

تأخیر رجیستر فایل: 1^{ns} — با دست آمدن فایل حافظه

سایر واحدها: $\emptyset$ — از رجیستر فایل می‌خواند و در حافظه می‌نویسد

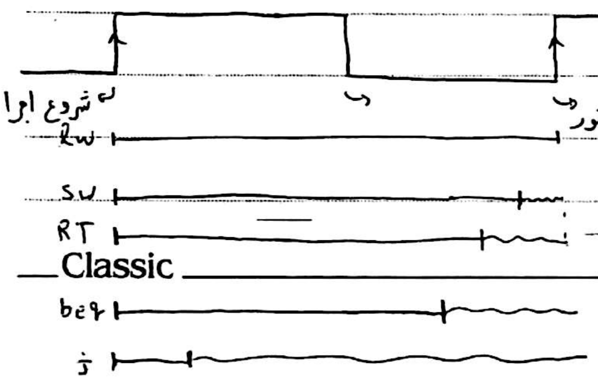
	خواندن دستور (Fetch)	خواندن اپرندها (از رجیستر فایل)	انجام عملیات ALU	دسترسی به حافظه (برای خواندن از آن)	نوشتن نتیجه در رجیستر مقصد	زمان کل
lw	۲ns	۱ns	۲ns	۲ns	۱ns	۸ns
sw	۲ns	۱ns	۲ns	۲ns	—	۷ns
R-T (R-type)	۲ns	۱ns	۲ns	—	۱ns	۶ns
beq	۲ns	۱ns	۲ns	—	—	۵ns
J	۲ns	—	—	—	—	۲ns

⇐ ما قبلاً دستور Jump را کنار گذاشتیم (به علت تفاوتش با بقیه‌ی دستورات) ⇐ حال می‌خواهیم آن را بررسی کنیم.



یک MUX دیگر هم ... آدرس پرش دستور J در مسیر داده برای انتخاب ...

جمع برای یافتن آدرس نیاز دارد ⇐ که باید در دیتاپس بررسی شود.



Hardware Utilization

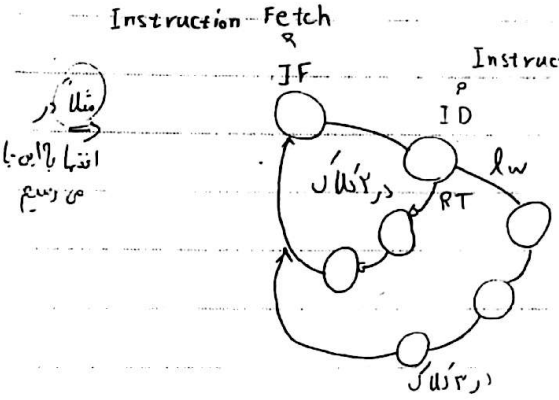
Classic

که می‌بینیم که طول clk ما باید بزرگترین حالت یعنی lw متناسب باشد ولی این مورد باعث می‌شود در دستورهای دیگر زمان تلف‌شده داریم.

NOTE BOOK Year: Month: Date:

⇐ پس با فکر طول کلاک با توجه به نوع دستور می‌توان این کار را کرد ولی این کار در عمل انجام نمی‌شود هیچ کاه تجاری نشده است.

⇐ راه دیگر این است که یک دستور در یک clk انجام نشود و دیگر این قید هم دستور در یک clk را کنار می‌گذاریم و clk ها را کوچک کرده و مثلاً lw در ۴ تا کلاک 2^{ns} ای انجام می‌شود و در این مورد زمان تلف شده خیلی کاهش می‌یابد.



⇐ پس می‌بینیم که باید پردازنده را با single cycle، resource sharing و multi cycle می‌سازیم که هزینه‌ای هم می‌دهیم component که انشاءالله در جلسات بعدی بررسی می‌کنیم.

Classic

Subject
Date

۹۲/۸/۴

مثال از کارایی: ①

	CPI
A	۱
B	۲
C	۳

(متوسط — CPI؛ کلاس‌ها در یک پردازنده — A, B, C)

	A	B	C
P_1 ← برنامهٔ اول	۲	۱	۲
P_2 ← برنامهٔ دوم	۴	۱	۱

① کدام برنامه دستورات بیشتری دارد؟

$P_1: ۲+۱+۲=۵$

$P_2: ۴+۱+۱=۶$

این همان بحث RISK و SISK است.

② کدام برنامه سریع‌تر است؟

(از یک کلاس تعداد دستورات × CPI آن کلاس: $\sum t_i (CPI_i)$)

$P_1: ۲×۱+۱×۲+۲×۳=۱۰ \text{ c.c.}$

$P_2: ۴×۱+۱×۲+۱×۳=۹ \text{ c.c.}$

③ CPI متوسط؟

$P_1: CPI_1 = \frac{۱۰}{۵} = ۲$

$P_2: CPI_2 = \frac{۹}{۶} = ۱/۵$ ⇐ با اینکه P۲ تعداد دستورات بیشتری دارد ولی سریع‌تر است. ⇐ پس فقط از روی تعداد دستورات تنها نمی‌توان در رابطه با سرعت صحبت کرد.

---

② ⇐ برای نشان دادن بدی single cycle

	فراوانی دستورات	
lw	۲۴٪	۸ns
sw	۱۲٪	۷ns
RT	۴۴٪	۶ns
beq	۱۸٪	۵ns
J	۲٪	۲ns

(میزان ٪ دستورات در برنامه‌ها)

⇐ ما داریم کارایی را بررسی می‌کنیم و برایمان مهم نیست که lw در آن اجرا می‌شود یا نه ⇐ بنابراین یک پردازنده با میانگین وزن‌دار این زمان‌ها که عملکردش از لحاظ کارایی مانند single cycle است. ⇐ clock‌های ۸ns ای هستند.

فقط ما داریم فرض می‌کنیم یک پردازنده با طول کلاک متغیر داریم:

$۲۴٪×۸ns+۱۲٪×۷ns+۴۴٪×۶ns+۱۸٪×۵ns+۲٪×۲ns=۶/۳ns$

⇐ $speed\ up = \frac{۸^{ns}}{۶/۳^{ns}} = ۱/۲۷$ ← بنابراین با داشتن کلاک متغیر، سرعت بالا می‌رود و بنابراین single cycle مناسب نیست.

Subject
Date

⇐ حال اگر دستورات ممیز شناور را هم وارد کنیم اوضاع بدتر می‌شود:

زمان عملیات جمع ممیز شناور : ۸ns
" " ضرب " " " : ۱۶ns

دستورات جمع ممیز شناور : خواندن دستور + خواندن operand + محاسبه + write back = ۱ + ۸ + ۱ + ۲ = ۱۲ns

دستورات غیر ممیز شناور : ۱ + ۱۶ + ۱ + ۲ = ۲۰ns

⇒ lw	۳۱٪	۸ns	
sw	۲۱٪	۷ns	
RT	۲۷٪	۶ns	
FP+ (floating point)	۷٪	۱۲ns	
FP *	۷٪	۲۰ns	⇐ در این single cycle طول کلاک باید انتخاب شود.
beq	۵٪	۵ns	
j	۲٪	۲ns	

اگر ⇒ طول کلاک متغیر داشتیم

۳۱٪ * ۸ + ۲۱٪ * ۷ + ۲۷٪ x ۶ + ۷٪ x ۱۲ + ۷٪ x ۲۰ + ۵٪ x ۵ + ۲٪ x ۲ = ۷ns

⇒ speed up = ۲۰/۷ = ۲٫۹ حد

اما باید داریم طول کلاک متغیر نمی‌توانیم بسازیم، پس با سراغ مولتی سایکل می‌رویم تا دستورات در چند کلاک انجام شوند.

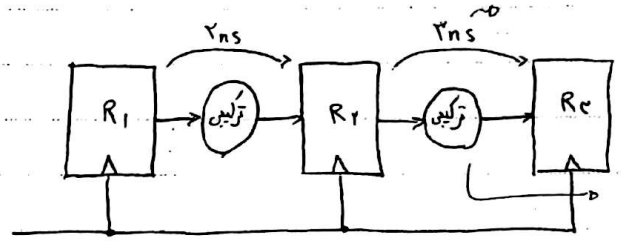
مولتی سایکل: ⇐ طراحی مولتی سایکل از روی single cycle انجام می‌گیرد با این معنی که ابتدا single را طراحی می‌کنیم و سپس از روی آن multy را می‌یابیم.

Subject
Date

۹۲،۸،۶ =

✓ دلیل وجود رجیسترهای A و B در دیتاپس مولتی سایکل درحالی که رجیسترغایل را با اشتراک نگذاشتیم:

طول کلاک ۳ns



همان کار A و B

حال اگر این مدار ترکیبی را با یک ALU ترکیبی و یک ۲ns ترکیبی بشکنیم طول کلاک ۲ns می‌شود.

⇐ چون Memory از این نظر که ورودی را روی خروجی می‌گذارد combinational (ترکیبی) است که تا خیر آن ۱ns است ⇐ وجود A و B ممکن است موجب شود یک کلاک سایکل اضافه شود ولی طول کلاک‌ها ۲ns است درحالی که در صورت این که A و B وجود نداشتند باشند طول کلاک ۳ns است ولی کلاک سایکل یکی کم شده است ⇐

پس عملکرد بهبود یافت. $4 \times 3\,ns = 12ns \Rightarrow 6 \times 2ns = 10ns$

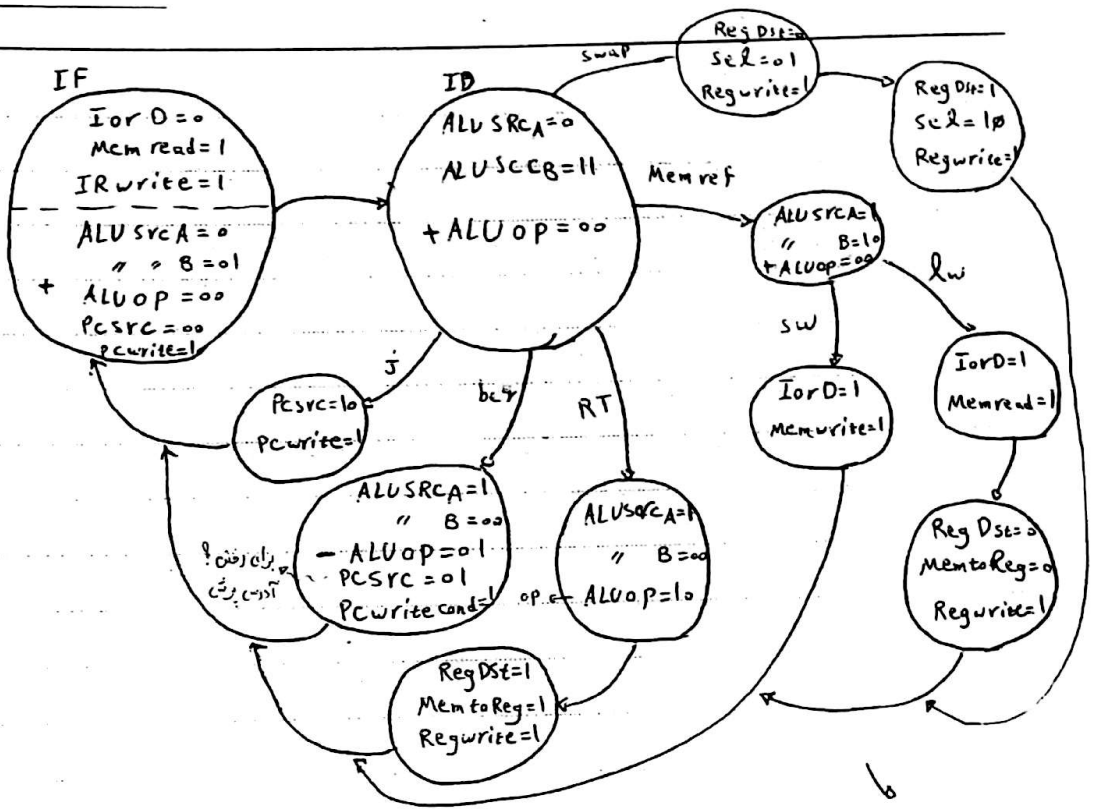
نه بنابراین کم شدن کلاک سایکل لزوماً خوب نیست.

⇐ رجیسترهای A و B و ALU Reg و ... اگر مورد اشتباهی هم در آن‌ها نوشته شود تا آخر مراحل که تای اجرای دستورات با آن کار نداریم و در مراحل بعد که دیتای رجیسترها برای ما معنی دارد از آن‌ها استفاده می‌کنیم و گرنه کاری با آن‌ها نداریم ولی IR در همان مراحل ما نیاز است و بنابراین نباید خودش همین طوری با خروجی رود پس سیگنال کنترلی برایش قرار دادیم.

⇐ طراحی State machine درمن جد.

Subject
Date

با ۴ سیکل ؟ مراحل اجرای دستورات



Hardwired ⇐ واحد کنترل سیم بندی شده

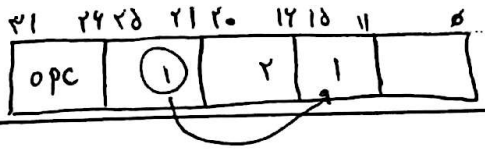
که این روش چون از تعداد FF و gate استفاده می‌کند بسیار سریع است و ولی flexible نیست و در مراحل آزمایشگاهی طراحی، واحد کنترل باید عوض شود و این جا از سختی امکان دارد چون باید از اول طراحی کنیم.

Micro Programming ⇐ واحد کنترل از روش ریز برنامه‌سازی ⇐ کند است چون حافظه دارد ولی flexible است ⇐ پس بهتر است طراحی را با این روش انجام دهیم و بعد از اینکه کار تمام شد آن را با Hardwired تبدیل کنیم.

که البته می‌توان با همان Micro هم طراحی نهایی را انجام داد.

R_1 و R_2 جای R_0 و R_2 را عوض می‌کند.

⇐ فرض کنیم می‌خواهیم دستور Swap را پردازنده اجرا کند ⇐ می‌خواهیم ببینیم مسیر داده این دستور وجود دارد یا نه و چه سیگنال‌های کنترلی باید فعال شوند.



Subject
Date

⇐ برای اجرای دستور swap کار دیگری که می توان کرد این است که در ۱۵ - ... مقدار صفر قرار دهیم و ابتدا R_2 را در R_1 قرار دهیم و سپس با کلاک بعدی خروجی ALU Reg را به write Data در رجیستر فایل وارد کنیم. پس یک mux، ۳۲ بیتی و یک mux ۱ به ۲، mux که ۲ به ۱، ۵ بیتی داریم که در مسیر rt# (یعنی آدرس مقصد)

⇐ دستور POP هم می توان با همین کارها و اضافه و کم کردن موارد مختلف به دیتاپس اضافه کنیم. (POP را در single cycle نداشتیم و قابل اجرا نبود)

۹۲/۸/۱۱:

انواع پردازنده ها:

① Zero - Addressing (Stack Based) — داخل پردازنده یک stack دارد.

مربوط به دستورات محاسباتی است.

add و sub oprand ندارد (یعنی محاسباتی ها oprand ندارند)

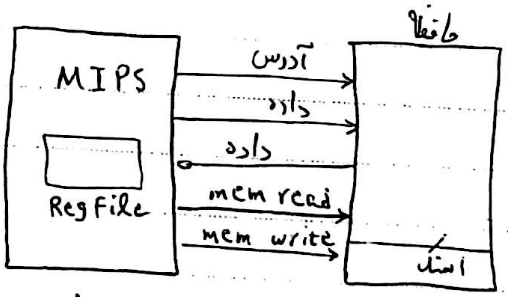
محتویات دو خانه ی بالای استک POP شده و حاصل جمع آن روی استک قرار می گیرد

② one - " (ACC Based) ⇐ یک رجیستر Acc وجود دارد که همیشه یکی از اپرندها رجیستر مقصد دستورات محاسباتی است ← آکومولاتور

③ two - " (سری X86)

add an, bn ⇒ an ← an + bn

④ Three - " ⇒ مانند MIPS

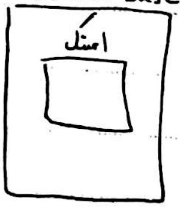


⇐ stack در این نوع پردازنده ها داخل پردازنده نیست.

InFix: $y = (a+b) - c$ ⇐ در این فرم حتماً نیاز به پرانتز دارد.

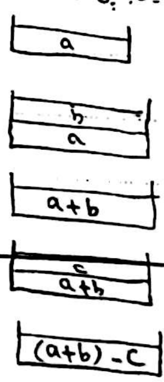
در ① به

Stack Based



⇐ reg File ندارد و stack به عنوان reg-file عمل می کند.

push a
push b
add
push c
sub ⇐ pop y



$y = (a+b) - c$

Revers polish

y = ab+c- ⇒ Postfix ⇒ stack based ها نیاز با پرانتز ندارد و برای
a+b -c خوب است.

y = +ab - c
a+b - c

(۲) همان دستور قبلی را با (۲) اجرا می‌کنیم:

acc ← M[a]; Load a
acc ← acc+M[b] add b
acc ← acc-M[c] sub c
M[y] ← acc Store y

پردازنده‌ای با مشخصات زیر طراحی کنید:
Zero - Addressing با
★ پردازنده دارای کلاس دستورات زیر است:

(۱)

opc (۲ بیت)	adr (۶ بیت)

opc	
00	Push adr
01	PoP adr

(۲)

opc (۲ بیت)	adr (۶ بیت)

10	jz adr

(۳)

opc (۲ بیت)	Func (۶ بیت)

	func	
11	000000	add
	000001	sub
	000010	not
	000100	and
	⋮	

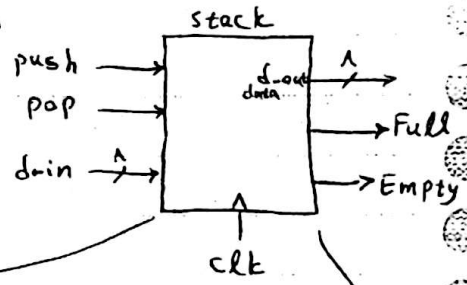
چون دستور پرش شرطی نداریم دستورات if than else
در این پردازنده قابل اجرا نیست
(ولی برای ما مهم نیست چون یک مثال است و پردازنده‌ی کاملی نداریم)

ادامه درس بعد

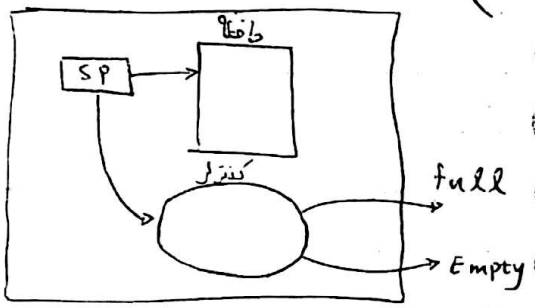
Subject نحوه‌ی کار stack چ‌خوانی و نوشتن، سنکرون است و آسنکرون نیست و با clk نیاز دارند بنابراین الزاماً رجیستری بعد از stack قرار بگیرد محتویات stack را در ۲ کلاک درخود نگاه می‌دارد.
Date

مثلاً 1k

برای این پردازنده‌ی ما، محدود است بنابراین برای Push نباید full باشد یا این‌که برای Pop باید empty باشد.

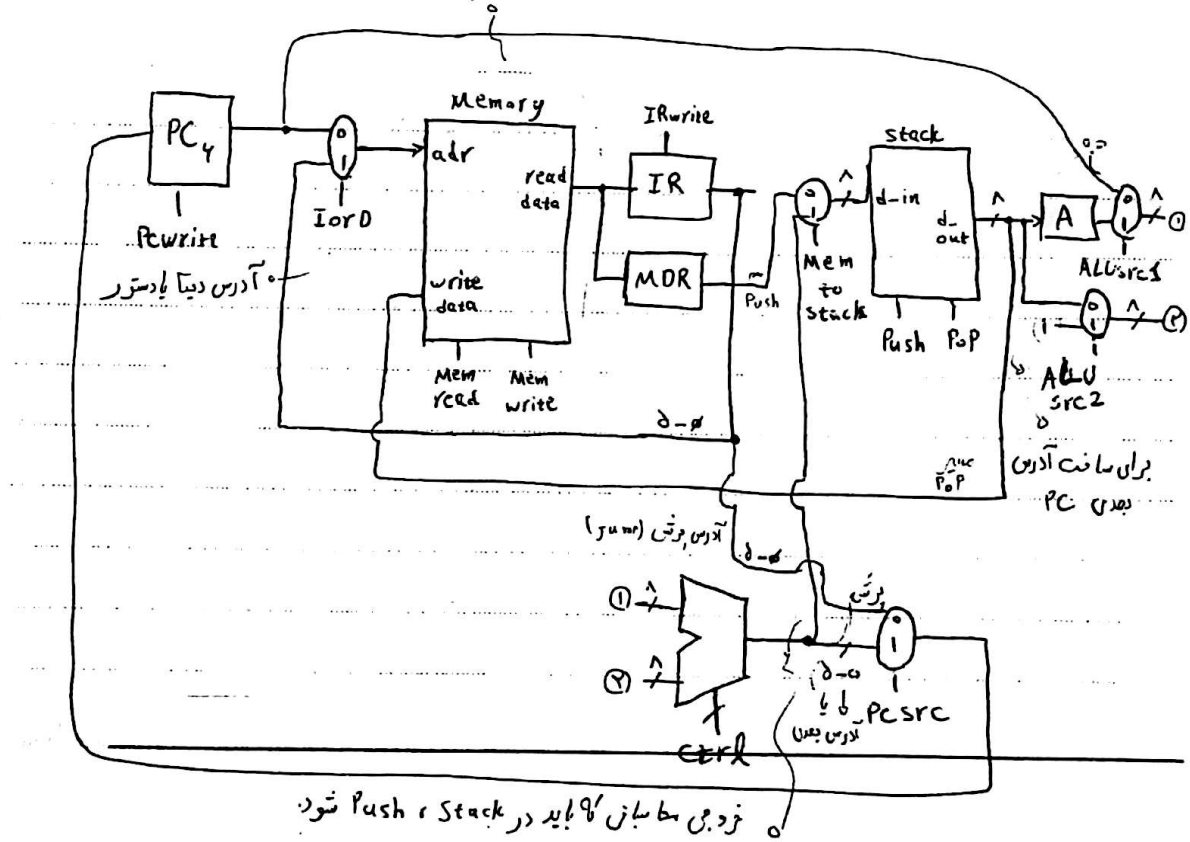


PC ما ۲۰ بیتی است پس حافظه‌ی ما 2^{20} خانه دارد.
چون adr ۲۰ بیتی است.



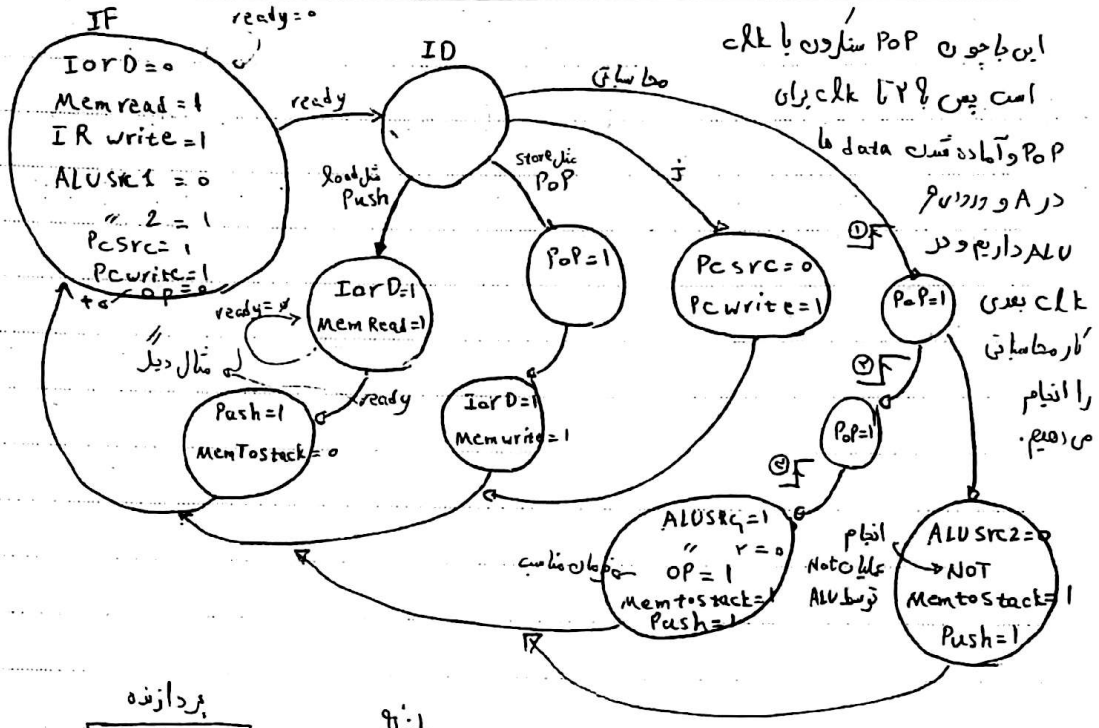
حتماً بررسی شود ⇐ طراحی بقیه‌ی پردازنده‌ها هم با همین صورت است.

جزء پردازنده نیست و بقیه‌ی اجزا همگی داخل پردازنده‌اند.

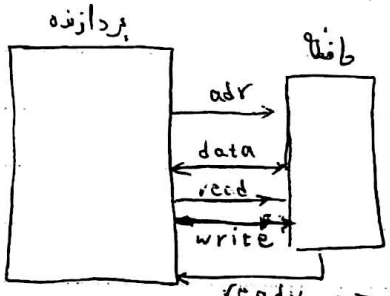


خروجی حساب‌ها باید در Stack، Push شود.

Subject
Date



این جا چون PoP سازدن با clk است پس ۲ تا clk برای PoP و آماده شدن data ها در A و ورودی B ALU داریم و در clk بعدی کار محاسباتی را انجام می‌دهیم.



برای این که وجود دارد تا حافظه‌های ما کند هستند و ممکن است بعضی جاها با ۷ کلاک بعضی جاها با ۵ کلاک دیتا داده را خواهیم کند تا ما می‌خواهیم برای هر کدام پردازنده را عوض کنیم. بنابراین با این سیگنال ready دروازه کنترل حافظه این flexibility را ایجاد می‌کنیم.

(سناریو)

رویدادهایی که باعث تغییر روند اجرای برنامه می‌شود:

Polling ⇐ در زمان‌های مشخص مرتباً سرکشی I/O ها می‌کند تا ببیند سرویس می‌خواهد یا نه

① منشا خارجی ⇐ Interrupt کار خودش را می‌کند تا یک نفی بیاید و در بزند و سپس می‌رود تا سرویس روتین سیستم عامل را انجام می‌دهد (سرعت بالاتر) ⇐ مثلاً کیبورد و ...

② منشا داخلی ⇐ Exception

Subject
Date

... هم داخل پردازنده است.

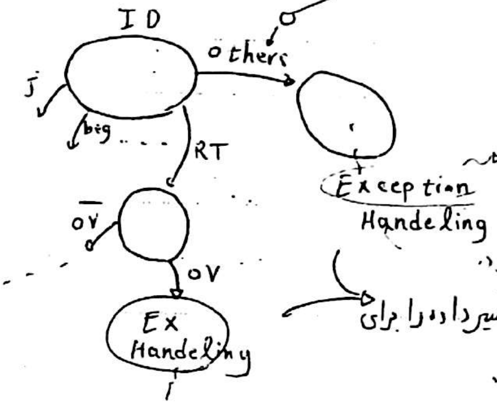
منشأ داخلی ← در پردازنده‌ها مان بررسی نمی‌کنیم چون ضرب و تقسیم نداریم.

← Divide by zero * ⇐ تصمیم‌گیری را با سیستم عامل می‌دهیم که حالت‌های مختلف دارد.

overflow

undefined Inst. ⇐ مثلاً opcode ای داریم که توسط پردازنده شناخته شده نیست.

} این ۲ مورد ⇐ باید در کنترلر اضافه شود



اتفاقی داخل پردازنده افتاده که آن را ← از مسیر اصلی خود خارج کرده است و سرویس روتینی برای اجرای آن باید ایجاد شود.

جلسات بعد می‌خواهیم واحد کنترل و مسیرداده را برای Ex. Handeling طراحی کنیم و انجام دهیم.

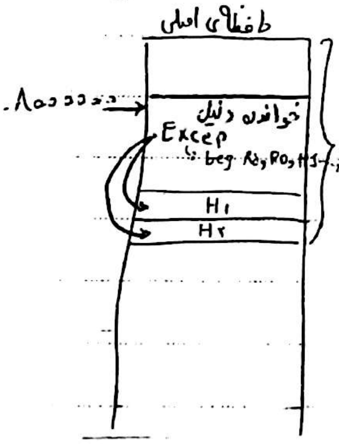
۹۲/۸/۱۳

① نوع Exception

② دستوری که باعث Exception شده است.

C۸۰۰۰۰۰۰

حالت اول
* با بروز Exception همواره به یک بخش خاص از سیستم عامل پرش کنیم و در آن بخش با اجرای برنامه‌ای نوع Excep. تشخیص داده شده و به روتین متناظر آن پرش کنیم.



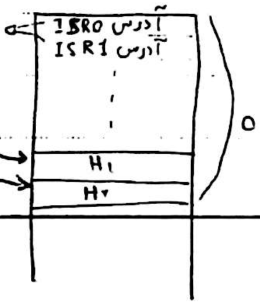
OS ← جدول اول (حالت اول)

فرض می‌کنیم که CR در رجیستر R5 نایل وصل شده.

beq R5, R0, H1

حالت دوم
* با بروز Excep. براساس نوع آن مستقیماً به روتین متناظر پرش کنیم.

درستی XA۶ (که در درس میکرو می‌خوانیم) ←



OS ← جدول دوم (حالت دوم)

مربوط به اینتراپت

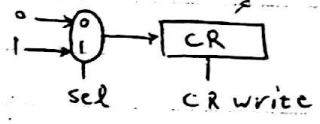
(۷۶) handle کردن Exception برعهده‌ی سیستم‌عامل (OS) است ولی ما باید در پردازنده‌ی خود مسیر لازم برای رسیدن به آن آدرس را برایش فراهم کنیم.

Subject
Date

Exception

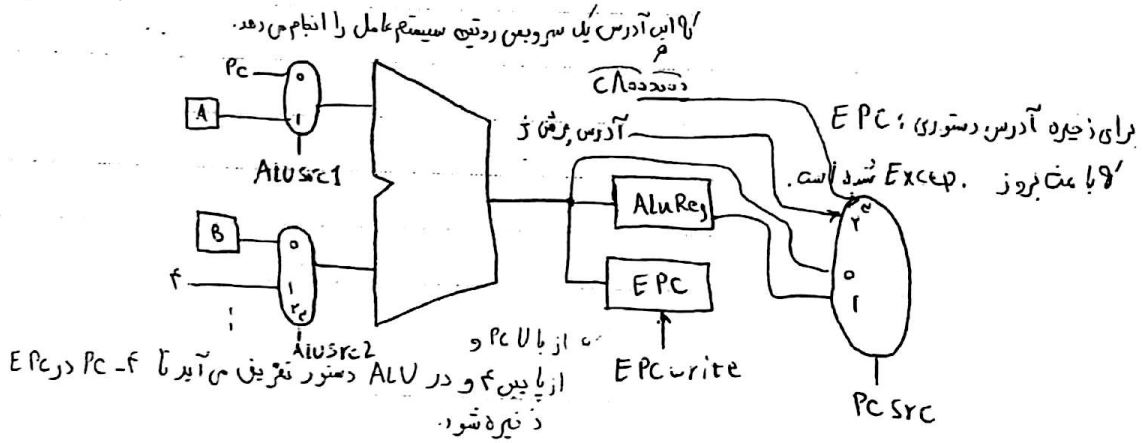
CR : Cause Reg. (رجیستری که دلیل را در خود نگاه می‌دارد)

یک فلیپ فلاپ



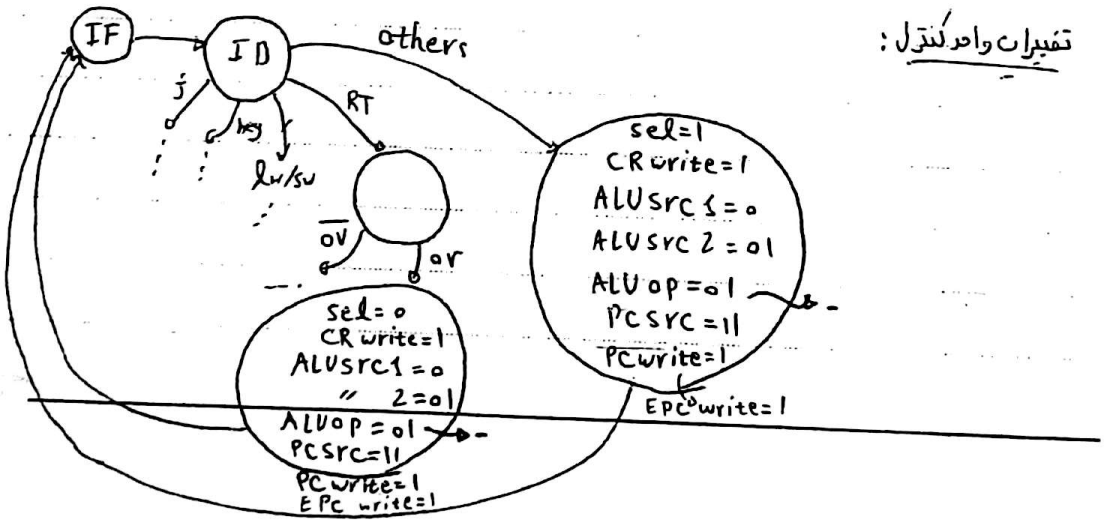
* برای Handle کردن
overflow ① 0
Undefined Inst. ② 1

Program Counter یا PC هم نیاز داریم (ولی ما در Fetch شماره PC+4 را داشتیم)



✓ برای این‌که بتوان محتویات CR را در سیستم عامل چک کرد باید حتماً مسیری برای نوشتن محتویات CR در یکی از رجیسترهای رجیستر فایل فراهم کرد. (یا Data path اضافه شود)

تغییرات واحد کنترل:



(۷۷)

درس قبل برای هر Excep. ما یک جای خاص در حافظه می بردیم ولی در روش وجود ISR ها در ناحیه های اول حافظه

ما برای هر Excep. یک جای حافظه می بردیم و آن را می خوانیم و بعد متناسب با آن یا H1 یا H2 یا ... می بریم پس

Subject
Date

واحد کنترل ما در حالت ۲ عوض می شود چون به ... باید تشخیص دهد ابتدا که کدام ISR بوده و همچنین باید دستور خواندن از حافظه هم فعال شود.

get rect و set rect ⇐ برای کار با سرویس روتین اینتراپت می باشد در x86 .

⇐ این کار قبلی در واحد کنترل و مسیرداده را سیستم کامل یک پردازنده Force کرده و برای ساختن نیازهای خودش است

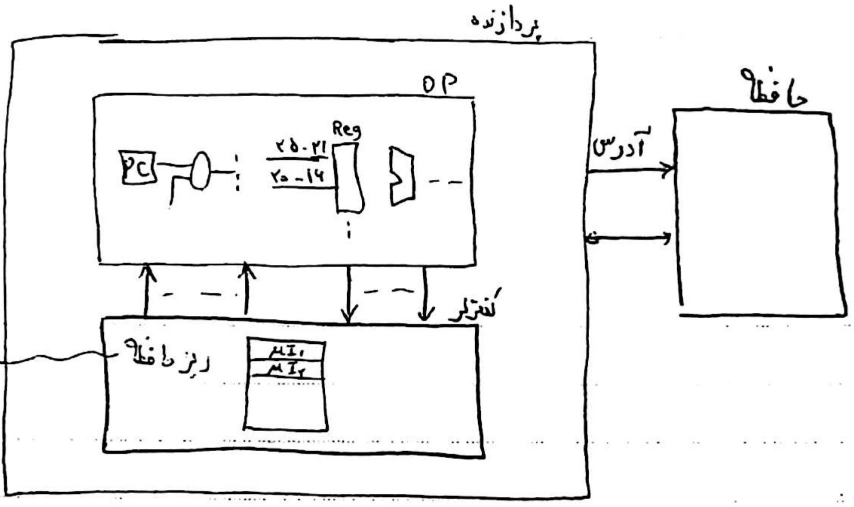
---

* واحد کنترل ریز برنامه سازی "Microprogrammed Controller" ⇐ در مقابل Hard wiring که قبلاً دلایلش را ذکر کردیم.

① ریزدستور Micro Inst. : مجموعه های سیگنال های کنترلی مسیرداده یک ریزدستور را تشکیل می دهد.

② اجرای ریزدستور: یه معنی صدور سیگنال های کنترلی موجود در ریز دستور

③ ریزحافظه (Micro Memory): حافظه ای که ریزدستورات در آن ذخیره می شوند.

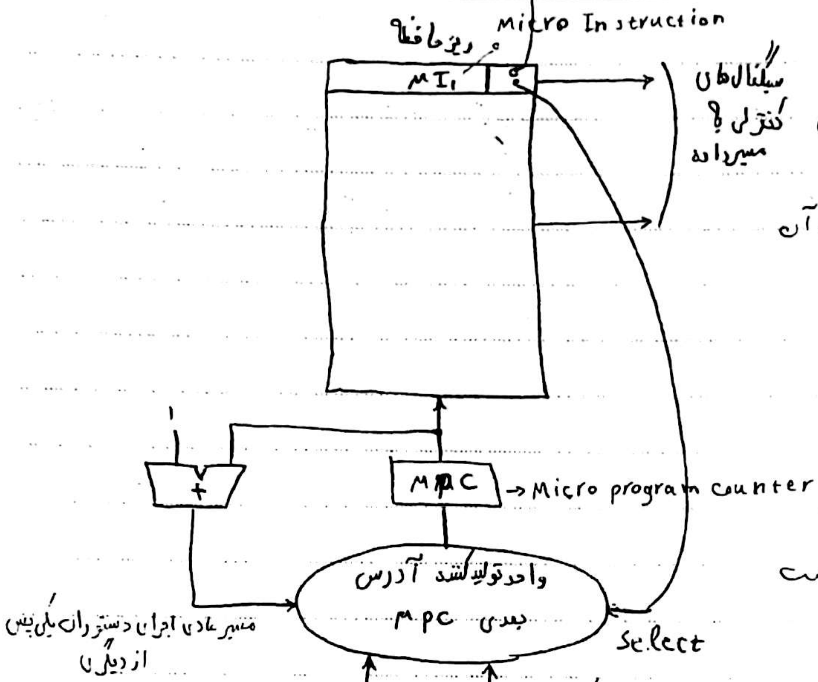


داخل واحد کنترل است و داخل پردازنده است و یه حافظه اصلی که داده و دستورات در آن قرار دارد ربطی ندارد.

راه های ما یک state machine
ریزحافظه شامل MIها (مثلاً IF ...) می باشند که به صورت ترتیب اجرا می شوند.

ادامه درس بعد

انتخاب mpc بعدی



درست کردن واحد کنترل در واقع می‌شود پر کردن ریزحافظه بنابراین flexibility را بالا می‌برد چون بعداً می‌توانیم به راحتی آن را عوض کنیم.

این برای کنترلر معمول است

یعنی آدرس ریزدستور بعدی از 1+ Mpc بدست می‌آید

برای سیگنال‌های کنترل در این قسمت می‌آید

برای این که مثلاً در j با بدکدام‌ها اجرا شوند

این‌ها یک ریزدستوراست

ریزحافظه: ریزدستور بعدی

		IorD	Memread	Mem write	Reg Dst	Mem to Reg	Reg write	ALU Src1	ALU Src2	ALU OP	PC src	Pc write	Pc write cont	فیلد انتخاب V
IF →	Ø	0	1	Ø	Ø	Ø	Ø	Ø	01	00	00	1	Ø	ØØ
ID →	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	10	00	-	—	-	Ø1
ریزدستور j →	2	Ø	Ø	Ø	Ø	Ø	Ø	Ø	00	00	10	1	-	1Ø
ریزدستور beq →	3	—	—	—	—	—	—						-	1Ø
ریزدستور RT →	4	-	—	—	—	—	—				—			ØØ
ریزدستور M.R.	5	—	—	—	—		—		—					1Ø
ریزدستور sw	6	.	—								—	—	—	~~Ø1~~

براساس opc آدرس ریزدستور بعدی تعیین می‌شود.

بینشان با آدرس Ø پرش کن

(11)

در اجرای دستورات M.R. برحسب lw یا sw بودن آن انتخاب دوتا

⇐ مثلاً اگر بخواهیم O.V. را هم به مسیر واحد کنترل اضافه کنیم باید قطعاً از ۲ بیت بیشتر باشیم

~~درآخر مرحله‌ی ۱ اگر j بود به j می‌رود و پس از اجرای j به IF با ØØ می‌پرد ⇐ در RT به ۴ و ۵ اجرا شوند~~

⇐ در ریزدستور M.R. ⇐ برای اینکه بتواند بفهمد به lw برود یا sw، نمی‌توان در انتها ای قرار داد چون با 01 به خود M.R. آمده‌ایم

۷۹

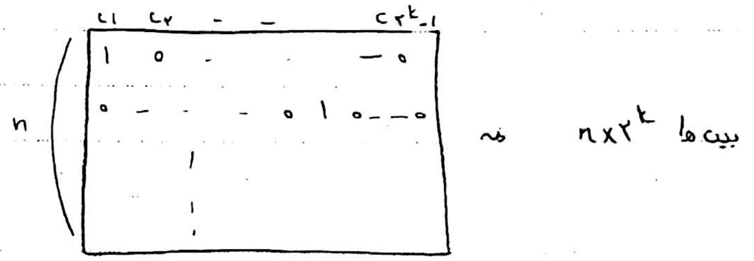
پس با 11 انتخاب در مراهم انجام می دهیم ⇐ بنابراین با توجه به واحد کنترل و این که چه تعداد و چند نوع رفتن به مرحله ی دیگر دارد تعداد بیت های فیلد پیدا می شوند که در این جا ۲ بیت کافی بود.

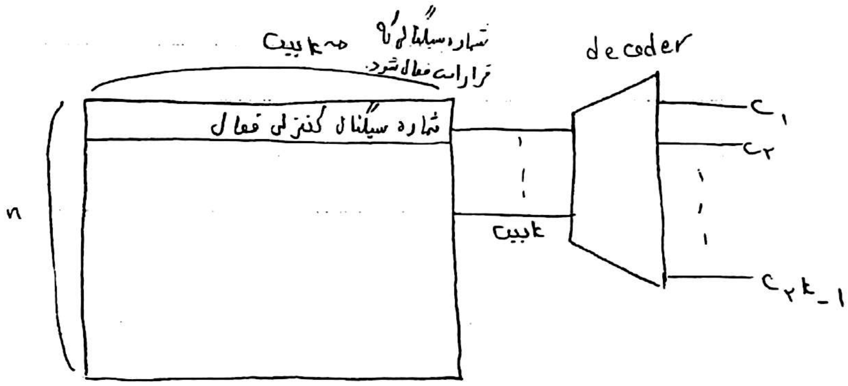
Subject

Date

⇐ ما ترجیح می دهیم ریز حافظه ی ما کوچک باشد چون در داخل خود پردازنده هستیم. (خودش هم کار می کنیم)

✓ فرض می کنیم در هر لحظه فقط یک سیگنال کنترلی فعال باشد و داریم (در ضمن از آدرس دستور بعدی هم فعلاً صرف نظر می کنیم و فقط با سیگنال های کنترلی کار داریم)





$n \times k$ بیت ها ⇐ پس تعداد بیت ها به مراتب کم شد ولی

این فرض اولیه، فرض معقولی نبود و به سراغ روش بعدی می رویم

کوییز ⇐ شنبه‌ی هفته‌ی بعد از مباحث طراحی ریزپردازنده (Single و multi) ⇐ سؤال داده می شود و مسیر داده ای را می دهد و از ما مراحل اجرای دستورات را می خواهد و Fetch و decode و ... (زیاد نیاز به خواندن ندارد)

⇐ حداقل ۲ تا CA دیگر یکی پایپ لاین و دیگری multi که در میان ترم دوم می دهد

۹۲/۸/۱۸ :

پس حافظه را برای فرض معقول داشتن ۲ تلاش می کنیم: