جلسه‌ی اول

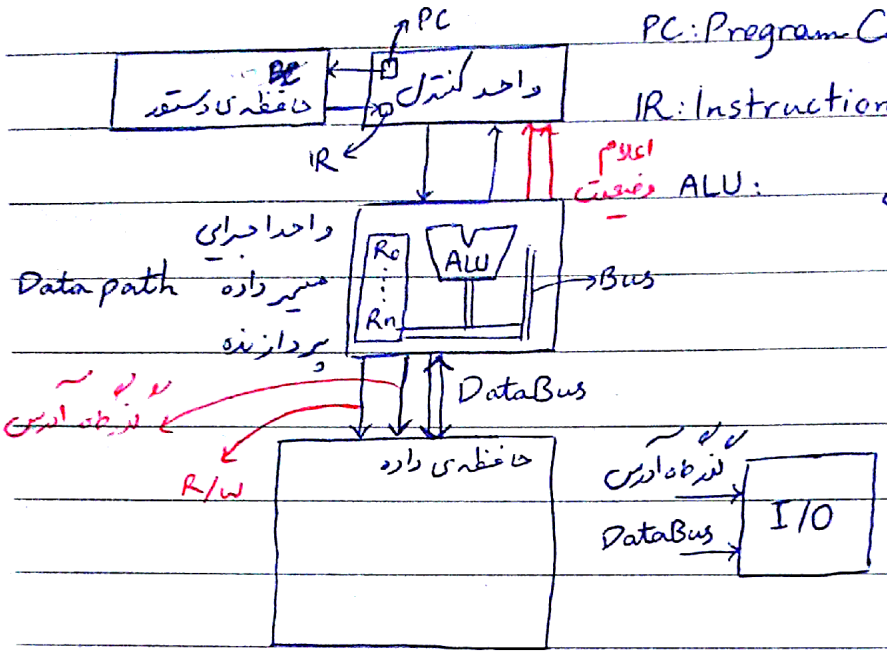
۹۲/۱۱/۲۶

کامپیوتر چیست؟

PC: Program Counter

IR: Instruction Register

ALU: واحد ریاضی / حسابی منطقی



انجمن فناوری اطلاعات
www.it98.ir

اگر حافظه‌ی داده و دستور یکی باشد سه معماری پرینستون

" " جدا " سه " هاروارد

امروزه بیش از یک ALU یا Core داریم که باعث افزایش Performance می‌شود.

افزایش سرعت معمولاً با افزودن Core افزایش می‌یابد.

برای اینکه چند واحد کنترل داشته باشیم باید ALU های خاص خود را داشته باشند.

الگوریتم حاکم بر اجرای دستورات ⟸ فون نیومن

۱- استعلام یا واکشی دستورالعمل از حافظه‌ی دستور از آدرسی که توسط شمارنده‌ی برنامه یا PC یا IP (Instruction Pointer) مشخص شده است (Fetch).

۲- ترجمه‌ی دستورالعمل (Decode A Instruction). دیکد رمزگشای دستورالعمل بایتی است.

۳- خواندن عملوند Read Operand. البته همیشه دستورالعمل‌ها این مرحله را ندارند.

۴- اجرای دستورالعمل (Instruction Execution)

۵- پس نویسی نتایج (Write Back)

۶- به روزرسانی PC (معمولاً با ۱ یا ۲ یا ۴ اجرا می‌شود تا به دستور بعدی دسترسی داشته باشیم و به نوع یا طول دستور بررسی دارد).

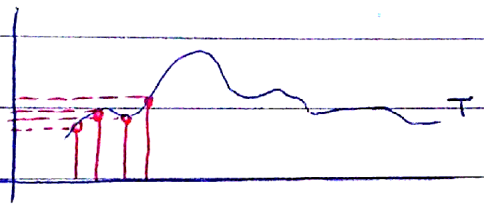
۷- بازگشت به گام ۱

www.8.ir

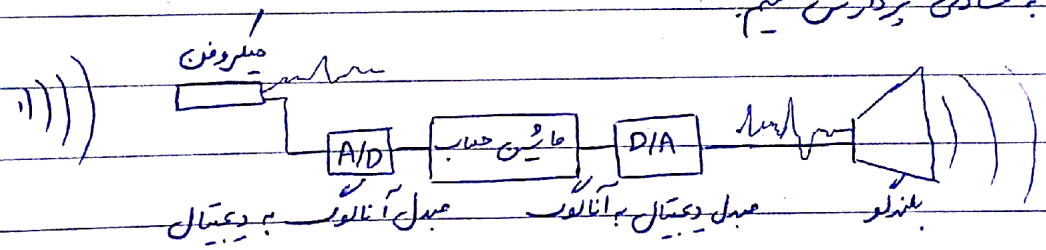
امروزه دستور ۶ با دستور ۱ ادغام شده است. نکته‌ی مهم این است که در این مدل، کارها ترتیب صورت می‌پذیرد یعنی این الگوریتم، الگوریتم ترتیبی است Sequential Alg.
ترتیب ⇐ پردازش موازی نداریم و همیشه بعضی اجزا بیکارند. حتی اگر چند هسته هم داشته باشیم نمی‌توان Piplime اجرا کرد.

جلسه‌ی دوم ۹۲/۱۱/۲۸

چند تعریف: مدارات آنالوگ (کامپیوتر آنالوگ): روی اطلاعات پیوسته کاری انجام می‌دهد.
مدارات دیجیتال (رقومی): تبدیل اطلاعات آنالوگ به یک رشته عدد.



اگر بتوانیم پدیده‌های طبیعی و پیرامون خود را به شکل یک توالی (سری یا رشته) عدد توصیف و بیان کنیم
پس می‌توانیم:
۱- آنها را به سادگی ذخیره و بازیابی و منتقل کنیم.
۲- می‌توانیم به سادگی پردازش کنیم.



* چرا تکنولوژی دیجیتال رونق گرفته است؟
سهولت بازیابی و ذخیره‌ی اطلاعات
از نظر ایمنی و قابلیت اطمینان ⇐ عدم وابستگی به شرایط محیطی نسبت به دما، رطوبت، فشار، ...
انتقال و دانلود فایل
پردازش آسان

* معایب تکنولوژی دیجیتال: بعضی از داده‌ها از بین می‌رود (خطای کوانتیزاسیون).
اگر فرکانس نمونه برداری = تعداد نمونه برداری‌ها در واحد زمان، بیش از دو برابر فرکانس سیگنال ورودی باشد هیچ اطلاعاتی از دست نمی‌رود (قضیه‌ی Shanon).

۱۲-۱۳ دوشنبه
۱۲-۱۳ ۴ شنبه ۲۰۲
۲/۵ ساعت بعد (۳) هر روز

Date: Subject:

پیکسل ← تعداد نمونه برداری ها از تصویر

تعداد پیکسل در واحد مساحت ← Sampling ≠ (فرکانس نمونه برداری)

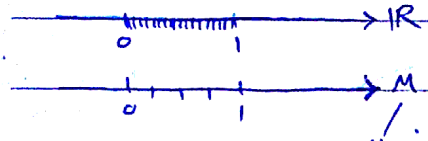
سرعت نمونه برداری ← دقت و قدرت تفکیک را در محاسبات بالا می برد ولی هزینه افزایش می یابد.

اکنون ← ۵ میلیارد برثانیه فرکانس نمونه برداری داریم.

۲- مجموعیم A/D و D/A بگذاریم.

۳- یک سری از اعداد حقیقی در حین تبدیل به ماشین، دچار خطای تقریب می شوند. چون با n بیت می توان 2^n عدد نمایش داد (محدودیت تعداد بیت ثابت ها).

تناظر یک به یک بین اعداد دهدهی و دودویی وجود ندارد مثل ۰/۱.



(خطای کوانتیزاسیون)

به همین دلیل، برای بازه ای از اعداد، قانون شرکت پذیری صدق نمی کند و دقت انجام محاسبات به ترتیب انجام آن وابسته است.

$(A+B)+C \neq A+(B+C)$

$A+B \neq B+A$ $\quad$ $A \times B \neq B \times A$

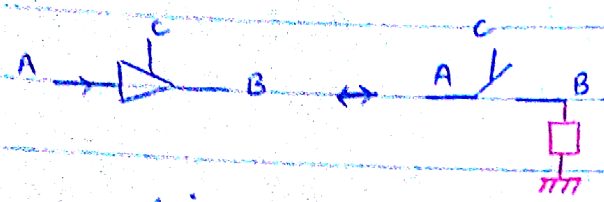
انجمن فناوری اطلاعات
INFORMATION TECHNOLOGY FORUM
www.it98.ir

Raz

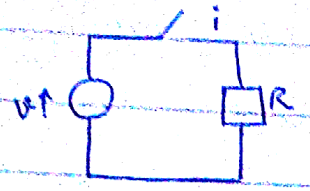
Date: (۴) Subject:

جلسه‌ی چهارم: (روز همدلی) ۱۳۹۲/۱۱/۵

با فرض ۳ حالت: کلید جهت دار



مثلاً لامپ کوچک روی کلیدهای برق، V ثابت و R بسیار بالا داریم و جریان ضعیف است: این لامپ بین فاز و نول



$V=Ri$ ، V ثابت و R بزرگ ⇐ i بسیار ضعیف

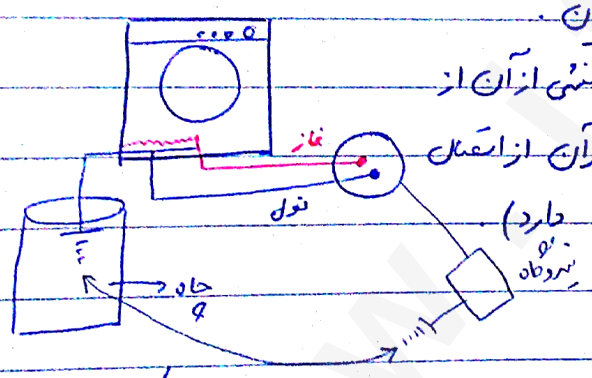
مدارات ۳ حالت: امپدانس بالا: مقاومت بسیار بالا درحدی که اثر ندارد. $c=0$ high z

$c=1$ حالت اتصال

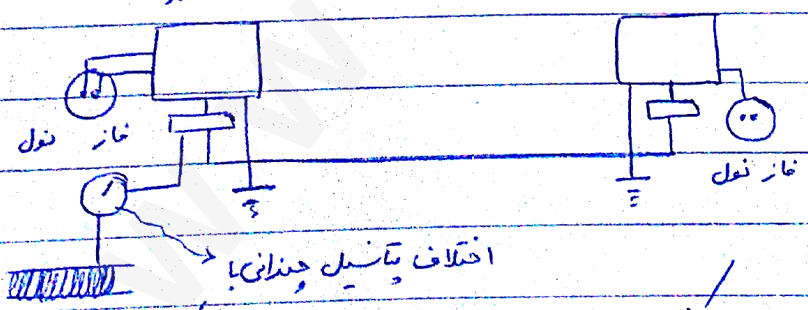
مثلاً کلید برق مثل c است که اگر وصل باشد؟ لامپ می‌تواند روشن شود.

سیم اتصال به زمین: برای خنثی کردن شوک جریان.

جریانی که از نیروگاه به مصرف کننده می‌رسد، یک بخشی از آن از طریق سیم‌های خط انتقال برمی‌گردد و بخش دیگر آن از اتصال به زمین برمی‌گردد (خود نیروگاه هم سیم earth دارد).



اتصال کوتاه: ولتاژی را به مقاومت کم وصل کنیم.



برای رفع این مشکل نول‌ها را به زمین وصل می‌کنیم.

سیم درون نداریم (بدنه‌ی کامپیوتر و نول)

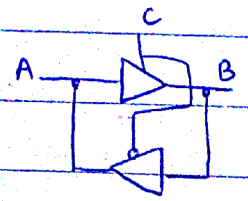
دی با پوشش کابل شبکه ۱۱۰V اختلاف پتانسیل داریم

چون بین نول مرکز محاسبات و دانشکده ۱۱۰V اختلاف پتانسیل داریم.

Raz
wWw.it98.ir

Date: Subject:

حتماً باید بین earth و نول، ولتاژ کمتر از ۵ ولت باشد والا دستگاه می‌سوزد.

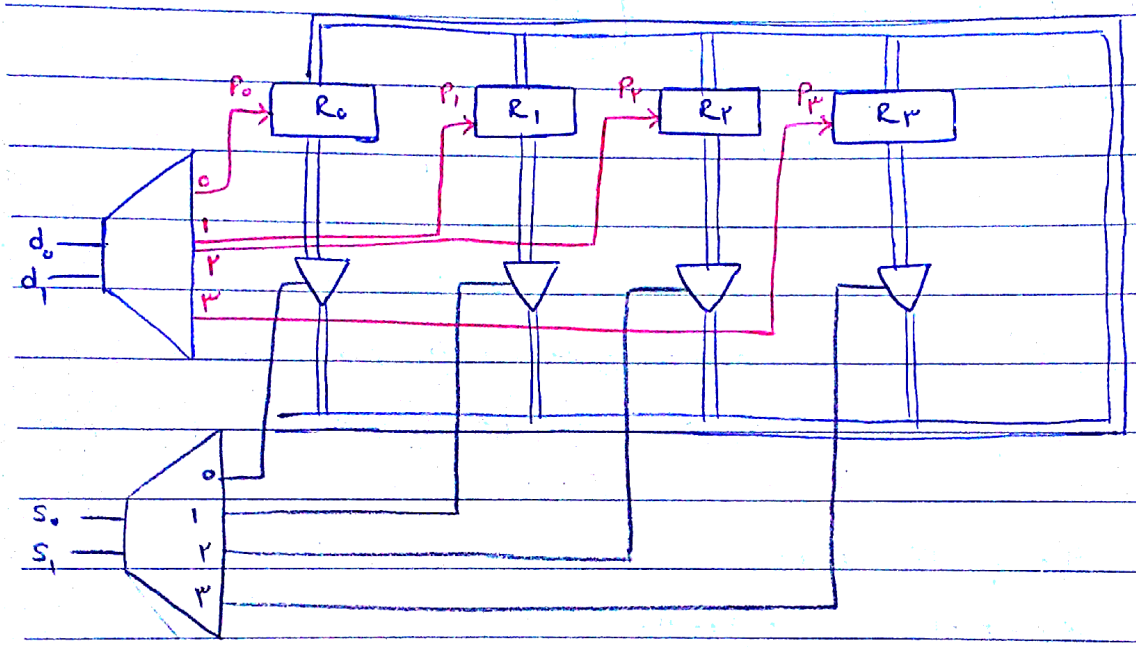


* پورت دو طرفه:

$C=1 \Rightarrow B \leftarrow A$

$C=0 \Rightarrow A \leftarrow B$

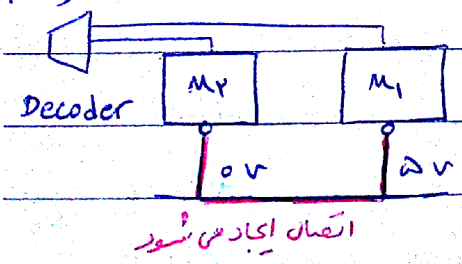
می‌توانیم از هر طرف که خواستیم داده را انتقال بدهیم (در میکروپروسسور).



d_1	d_0	S_1	S_0
۱	۱	۰	۱
مقصد		منبع	

باینر ۳ حالته اجازه می‌دهد که دستگاه‌های مختلفی به باس اعمال شوند و مزاحم هم نشوند به شرطی که دیکدر داشته باشیم.

اگر چند تا جزء روی باس داشته باشیم هر کدام یک دستوری می‌دهند مثلاً یکی ولتاژ ۵ ولت داشته باشیم و دیگری می‌گوید ۰ ولت. در اینجا اتصال کوتاه برقرار می‌شود و دستگاه می‌سوزد. پس باید CS داشته باشیم.



دیکدر باعث می‌شود در هر لحظه فقط یک خروجی داشته باشیم.

تاریخچه‌ی کامپیوتر:

اولین ماشین محاسب ← پاسکال (فقط جمع و تفریق)

لایبنیتز ← ضرب و جمع را با ایده‌ی جمع و تفریق‌های متوالی پیاده‌سازی کرد.

www.it98.ir

Date: ..... Subject: .....

۹۲/۱۲/۱۰ جلسه‌ی پنجم

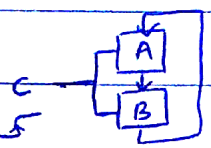
طراحی واحد مسیر داده:

در سطوح مختلف از زبان‌های مختلف استفاده می‌کنیم. مثلاً سطح مدار منطقی، از جبر بولین استفاده می‌کردیم. در سطح انتقال بین ثبات، از زبان RTL یا

وقتی c=1 شد همزمان A را در B و B را در A بریز

همزمان

C: A←B , B←A ;



if(c==1) temp=A

A =B

B =temp

متغیر مصنوعی

معرف به نیت و عقل ۱

راشان می‌دهد ما می‌خواهیم به یعنی ۰ به ارزش

که در سخت افزار وجود ندارد.

c راشان دهد.

مثال بالا ضعف زبان های روالی یا پروسیجرال را نشان می‌دهد. (روالی: طبق روال یا ترتیب خاص اجرا می‌شوند). در زبان‌های غیر روالی با یک، همزمانی را نشان می‌دهیم.

شروط انجام عملیات — ریزعملیات (micro operation)

$a_1 \cdot \bar{b}$: $A \leftarrow B_1$, $PC \leftarrow PC+1$;

$c+d$: $R_1 \leftarrow \bar{R}_1+1$, $Mem[R_4] \leftarrow A$;

t_0 : $R_2 \leftarrow Mem[R_2]$, $A \leftarrow A+1$,

⋮

داخل سخت افزار اجزای زیادی داریم که با هم، همزمان کار انجام می‌دهند بنابراین زبان‌مان باید چنین ویژگی‌ای داشته باشد. زبان‌های توصیف سخت افزار: HDL: Hardware description language

تنازع یا Conflict، در زمان اجرای چندین دستور، منابع مشترک باشند تنازع رخ می‌دهد.

منبع: یا ALU است، یا Bus یا ثبات یا حافظه است.

به جای PC←PC+1 می‌توان ورودی inc مربوط به PC را فعال کرد.

$a_1 \bar{b}$ = inc → [PC]

دو انتقال همزمان به یک ثبات مشترک مجاز نمی‌باشد.

وقتی دو خط و هم بین خطوط متوازی داشته باشیم، کامپایلر warning می‌دهد و می‌پرسد اولویت را به کدامیک می‌دهیم.

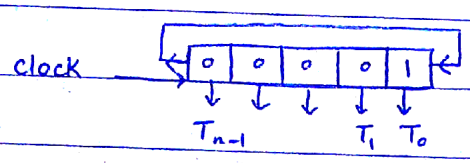
Raz

wWw.it98.ir

Date: Subject:

شروط (سمت چپ) توسط واحد کنترل می‌آید و سمت راست خروجی از واحد اجرایی می‌آید.

مدار توالی ساز:



T_0: $IR \leftarrow Mem[PC]$, $PC \leftarrow PC+1$;

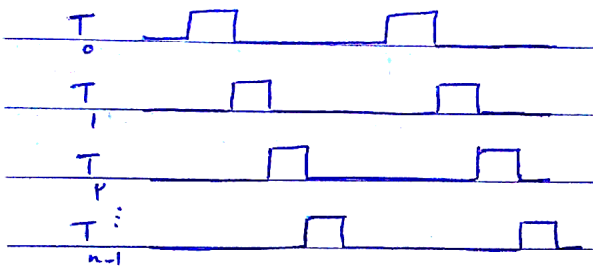
T_1: Decode

} مرحله‌ی ثابت فون نیومن ← Fetch & Decode

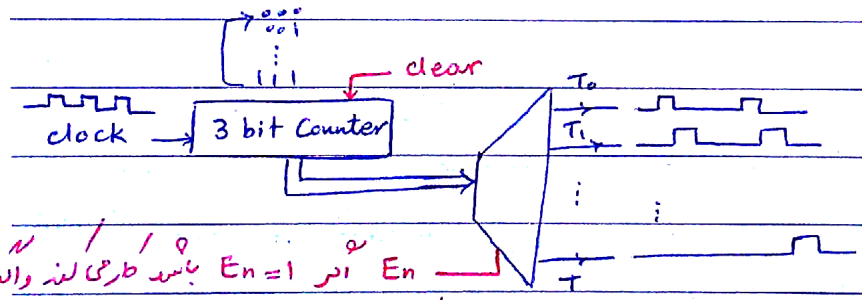
T_2: $R_1 \leftarrow Mem[\ ]$

T_3: $R_3 \leftarrow R_1 + R_2$

اگر عملی $PC \leftarrow PC+1$ ، در شروع T_1 است، یعنی در آخر T_0 وقتی T_1 شروع می‌شود، انجام شده است.



مدل دیگر توالی ساز:



اگر En = 1 باشد کار می‌کند و اگر En = 0 ، کار نمی‌کند و انگار که مثلاً برق رفته است.

شمارنده یکی یکی با clock زیاد می‌شود و بعد در Decoder به ترتیب خروجی‌ها فعال می‌شوند و چون Decoder هر لحظه یکی از ورودی‌هایش فعال است پس در هر لحظه در یکی از حالت‌ها هستیم.

T_0 تا T_{7-1} بیشترین تعداد مراحلی است که

clear: برای دستوراتی که طول اجرایشان کمتر از بقیه است. برای اجرای دستوراتی که CLK آنها با هم متفاوت است و توسط واحد کنترل صادر می‌شود (زمانی که طول دستورات متفاوت هستند).

www.it98.ir

www.it98.ir

Date: Subject:

سطوح طراحی:

رفتاری: الگوریتم (الگوریتم)

سیستم: پردازنده، حافظه، I/O (ریز پردازنده + سیستم عامل)

انتقال بین ثبات RT: ثبات، Bus، حافظه (معماری)

منطقی: جبر بولین، گیت های مختلف (مدار منطقی)

کلیدزنی (سوئیچ): ترانزیستور، معادلات دیفرانسیل حاکم بر جریان و ولتاژ (مدار الکتریکی)

جانمایی: تراشه های نیمه هادی (VLSI)

* آیا می توان یک Hardware Compiler ساخت که الگوریتم را بفهمد و قطعه ی مورد نظر را تولید کند؟

در این صورت تولید بهینه نداریم. در لایه های زیرین جزئیات بیشتری داریم و باید با ابعاد میکرو و متر کار کنیم. در هر سطح باید یک سری اطلاعات افزود.

تجرید بیشتر

جزئیات بیشتر

رویکرد عامه طراحی کامپیوتر (پردازنده ی MIPS):

مثلاً مشتری به ما اطلاعات ISA را می دهد و ما موظفیم با این اطلاعات، کامپیوتر بسازیم.

ISA: مجموعه ی دستورات، انواع داده های هر دستور، مدهای آدرس دهی، ثبات های مختلف

همه ی ساخت و طراحی (معماری مجموعه ی دستورات) ⇐ ISA.

جلسه ی ششم

۹۲/۱۲/۱۲

گام های طراحی

۱- معماری مجموعه ی دستورالعمل ⇐ ISA: تعریف مجموعه دستورات

مدهای آدرس دهی

تعداد ثبات ها و نوع آنها

تعداد گذرگاه ها و ALU

فضای آدرس دهی

Raz

wWw.it98.ir

Date: Subject:

۲- طراحی واحد کنترل

۳- مسیر داده یا Data path

فرض کنیم ISA: Mips را به ما داده اند و حالا از ما خواسته اند کامپیوتر را طراحی کنیم. از جمله محدودیت ها، این است که در این پردازنده طول دستورات ثابت و ۳۲ بیتی است.

یادآوری:

۳ نوع دارد:

R type | op (۶) | Rs (۵) | Rt (۵) | Rd (۵) | shamt (۵) | funct (۶) | → بقیه‌ی op که در ۶ بیت جا نشده است.

دستورات ثابتی

۳۲ ثبات دارد: $s_0 - s_7$, $t_0 - t_9$, ...

I type | op | Rs | Rd | address/constant |

J type | op (۶) | address (۲۶) |

نوع اول دستورات ۳ عملوند هستند. مثلاً $R_1 \leftarrow R_2 + R_3 \Leftrightarrow$ add R_1, R_2, R_3

| 0 | 17 (S1) | 18 (S2) | 8 (t0) | 0 | 32 |

$\leftarrow$ add \$t0, \$S1, \$S2

چرا در add، op=0 است؟ چون add دستور پرکاربردی است و باید دسترسی به آن آسان‌تر باشد.

lw \$t0, 32(\$S3) $t_0 \leftarrow A[8]$

add \$t0, \$S2, \$t0 $t_0 \leftarrow t_0 + h$

فقط lw و sw به حافظه دسترسی دارند یعنی تنها دستورات مراجعه به حافظه در Mips، load و Store است $\Leftarrow$ پردازنده از نوع Risc است.

sw \$t0, 48(\$S3) $A[12] \leftarrow t_0$

$A[12] = A[8] + h$

در دستور اول از A[0] به A[8] می‌رویم، چون هر word، ۴ بیت دارد ۸ خانه جلو می‌رویم.

ماشین‌ها:

- بدون آدرس $\Leftarrow$ پشته‌ای add
- یک آدرس $\Leftarrow$ انباشته‌ای add x $\Leftrightarrow$ $acc \leftarrow x + acc$
- دو آدرس add x, y $\Leftrightarrow$ $x \leftarrow x + y$
- سه آدرس add x, y, z $\Leftrightarrow$ $x \leftarrow y + z$
- گاهی هم بیشتر

Raz

.it98.ir

Date: Subject:

مدهای آدرس دهی : ۵ نوع هستند :

۱- آدرس دهی آنی (imm addr) . مقدار عملوند در داخل دستورالعمل ذکر می گردد، چون محدود به ۱۶ بیت هستیم . مثلاً S3=S4+1 ↔ addi S3,S4,S1

۲- آدرس دهی ثباتی (همان R-type) :

چرا ۳۲ ثبات داریم ؟ چون پردازنده Risc است و فقط با lw و sw به حافظه دسترسی داریم . در اینصورت مراجعه به حافظه کم می شود و سرعت بالا می رود . حافظه سرعتگیر پردازنده است . جسیت مراجعه به حافظه ی اصلی را کم می کند .

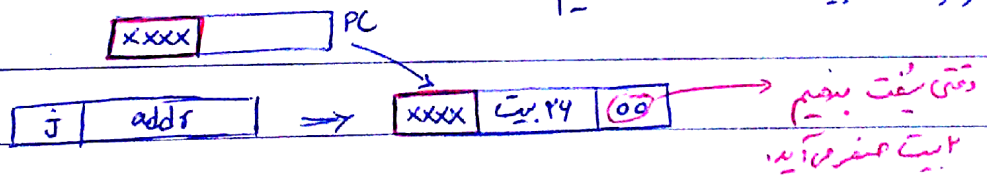
۳- آدرس دهی بر مبنای ثبات پایه (Base/Index addressing) : برای آدرس دهی خانه های جداول و ...

۴- آدرس دهی نسبی (PC relative addr) :

۵- آدرس دهی شبه مستقیم (pseudoDirect) :

۶- آدرس دهی مستقیم یا مطلق (Direct) : آدرس مراجعه را به صراحت ذکر می کنیم .

در آدرس دهی شبه مستقیم برای اینکه ۳۲ بیت پر شود ، ۶ بیت کم داریم که برای پر کردن این ۶ بیت ، اول ۲ بیت به چپ شیفت می دهیم (یعنی ضرب در ۴ می کنیم) که برای تبعیت از محدودیت align است ، سپس در سمت چپ ، ۴ بیت از PC را می گذاریم .



وقتی آدرس دهی شبه مستقیم

در مد شماره ی ۳ ، یک آدرس در یک ثبات قرار می دهد و بعد به کمک یک Constant ، اندازه (شماره ی آرایه) را انتخاب می کنیم (دسترسی نمایه ای یا indexed) .

کاملترین مجموعه نمونه سوالات بهمراه پاسخنامه تستی و تشریحی، جزوه، اسلاید، حل تمرین، گزارش آزمایشگاه و کارآموزی، پروژه و ... در سایت :

www.it98.ir

مرجع دانشجویان رشته کامپیوتر دانشگاه پیام نور

Raz

www.it98.ir

Date: Subject:

جلسه‌ی هفتم ۹۲/۱۲/۱۷

فصل چهارم کتاب هنسی پترسون:

معماری از نوع هاروارد است. امروزه همه‌ی پردازنده‌ها بطور درونی یک حافظه‌ی دستور دارند که جدا از حافظه‌ی داده است، یعنی بصورت درونی هاروارد و بصورت بیرونی پرینستون هستند.

pc ← pc+4 : برای بروز رسانی pc. چرا ۴؟ چون طول دستورات ۳۲ بیتی و ثابت است اگر ثابت نبود (مثل پردازنده‌ی Intel) بعضی وقتا +۴ می‌کردیم بعضی وقتا +۳ و... یعنی سخت افزار پیچیده‌تر می‌شد.

در mips، moveregister وجود ندارد. می‌توانیم ثبات مورد نظر را با zero، add کنیم و در ثبات دلخواه بریزیم.

معماری mips از نوع هاروارد است که باعث همروندی می‌شود.

ص ۲۹۲ کتاب: طرح اولیه‌ی mips:

در این معماری، محاسبه‌ی آدرس با همان ALU داده انجام می‌شود، که باعث کاهش تعداد ALU می‌گردد. ولی ممکن است بخواهیم به معماری محاسبات داده‌ای، یک آدرس هم محاسبه کنیم که بهتر است یک ALU کمکی وجود داشته باشد.

وقتی همه‌ی خروجی های ALU صفر شوند، flag صفر یا (zero Flag)، یک می‌شود.

درگاه Data in که به Register file وارد شده، ۳۲ بیتی است.

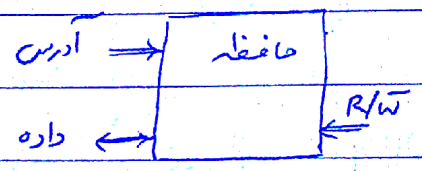
اگر در دستوری مشخص نکرده باشد که Signed است یا unsigned، همه‌ی اعداد را Signed در نظر می‌گیرد.

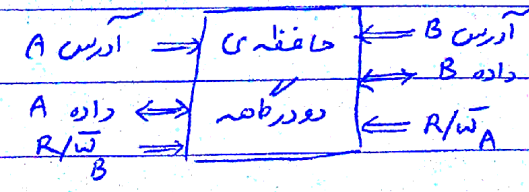
Sign Extend: تکرار کردن بیت علامت.

جلسه‌ی هشتم ۹۲/۱۲/۱۹

داده دو طرفه است چون یا می‌خواهیم بخوانیم یا بنویسیم.

حافظه‌ی دو درگاهی می‌توان دو کار انجام داد مثلاً در آن واحد از یه آدرس بخونیم و توی یه آدرس دیگه بنویسیم.





Raz

www.it98.ir

Date: Subject:

ALU ← نقش کلیدی و Core اصلی محاسباتی. طبق فرمانهای دریافتی، عملیات حسابی و منطقی را انجام می دهد. قسمت اصلی پردازنده.

مقصد مشترک + مبدأ چندگانه ← مالتی پلکسر

** مدار با فرمان سه حالته می تواند یک جایگزین برای مالتی پلکسر باشد.

mux سمت چپ، توی کتاب:

خروجی: load می کند از حافظه و در مقصد می نویسد: Read data آن را از حافظه خوانده و write data آن را در محل مورد نظر می نویسد.

شکل ۴-۱۱:

pc+4 همیشه اجرا می شود طبق الگوریتم فون نیومن. چرا ۴؟ چون دستورات ۴ بایتی هستند.

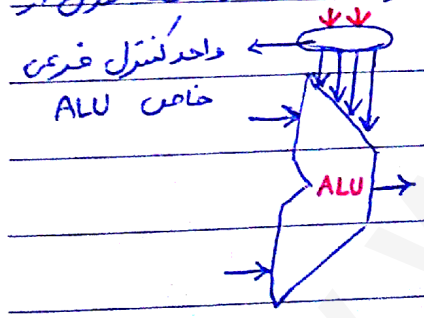
* a Simple implementation: (کتاب خوب نگفته)

خطوط ALU، ۴ تاست، پس ۱۶ کاره است، یعنی ۱۶ فرمان می پذیرد. چرا ۴ تا؟ ممکن است ۱۶ کار اصلی بیشتر نداشته باشیم.

" " بخواهیم با همین ۱۶ تا پایه ی اصلی، بقیه ی کارها را هم انجام دهیم. چون بین تنوع دستورات و سرعت رابطه ی عکس هست و اگر ALU بزرگ شود، سرعت کم می شود.

The ALU Control:

برای صرفه جویی، یک واحد کنترل فرعی برای خودمون می گذاریم و فقط ۲ تا خط کنترل از Control unit می گیریم.



ALU در دستورات lw، آدرس واقعی یا effective Addr را محاسبه می کند.

lw $s4, 6($s5)

+ ⇐ ALU جمع می کند اینارو.

sw هم دقیقاً همینطوره.

beq، عمل مقایسه هم انجام می ده یعنی Subtract رو ALU انجام میده.

با دو بیت مربوط به ALUOp، که کنترل اصلی آن را در اختیار داره.

Raz

wWw.it98.ir

www.it98.ir

Date: Subject:

دو بیت از واحد کنترل می‌گیریم و ۶ بیت از Funct و عمل مورد نظر (Operation) را رمزگشایی می‌کنیم.

مزیت آن این است که کنترل ALU در کنار و نزدیک به خودش انجام می‌شود.

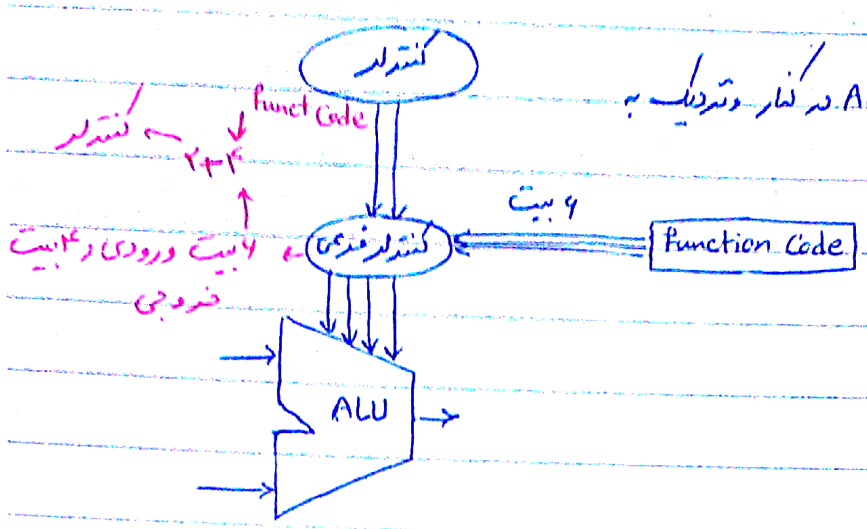


Fig 4.15 : $R_d \leftarrow 16-20$ در I-type و $R_d \leftarrow 11-15$ در R-type پس mux می‌گذارد که معلوم شود کدام را قبول کند؟ (mux سمت چپ).

برای اینکه این mux حذف شود، می‌توان در ساختار دستورات، جای R_d را عوض کرد که مثل هم شوند.

Fig 4.17 : ۶ بیت ابتدای دستور به واحد کنترل رفته است (در بالا Control).

ALUOperation که ۲ بیتی است به واحد کنترل فرعی رفته است (ALU Control).

به ازای هر دستور، خروجی حاد سیگنال‌های واحدی کنترل فعال یا غیرفعال می‌شوند. یعنی واحد کنترل به ازای هر دستور یک سری سیگنال را فعال و یک سری را غیرفعال می‌کند. چگونه این کار را انجام می‌دهد؟

چند چند تایی که از واحد کنترل خارج شده

op ۶ بیتی ← opcode	Sign1	Sign2	Sign3	...
۰۰۰ ... ۰	۰	۰	۱	...
۰۰ ... ۰	۱	۱	۰	...
⋮	⋮			

۶ تا ۲۶

بعد می‌آییم معادله‌ی منطقی هر Signal را بوسیله‌ی جدول کارنو و ... تولید می‌کنیم.

به این روش hardwired Control Logic می‌گویند.

Raz

www.it98.ir

Date: 8-3-2014 1:13 10:19 Subject:

هر مدار ترکیبی با یک حافظه قابل پیاده سازی است :

روش دوم پیاده سازی (بدون استفاده از گیت) : microprogrammed Control Logic

مزیت روش دوم : می توان واحد کنترل را که اکنون با حافظه جایگزین شده ، عوض کرد. یعنی عملکرد سخت افزار بدون اینکه سخت افزار را عوض کنیم ارتقا می یابد.

Fig 4.18 : خودمون بخونیم

جلسه ی هشتم : ۹۲/۱۲/۲۴

عرض حافظه ی دستور ← ۳۲ بیت

آدرس های " " ← ۳۲ بیتی

تعداد دستورهایی که می توان در آن جای داد : 2^{32}

Fig 4.19 : چرا در Read data1 و Read data2 یکی به mux وصل شده ؟

خودمون پیدا کنیم جوابش رو.

Fig 4.19 :

اعضای عمل کننده ، تحت فرمان واحد کنترل هستند.

دستور lw از نوع دستورات I type است . R_s با imm جمع می شود و آدرس جدید تولید می شود که عمل مراجعه به حافظه است (ثبات پایه)

چرا Read Reg2 غیرفعال شده است ؟ چون Instruction 20-16 مربوط به R_d است و می خواهیم در آن بنویسیم و نمی خواهیم بخوانیم

عمل بانک ثبات در اینجا متفاوت نیست یعنی Read بدون فعال شدن فرمان از CU انجام می شود چون تغییری اعمال نمی شود ولی اگر بخواهیم بنویسیم چون باید تغییر اعمال کنیم باید حتماً از CU ، فرمان نوشتن اعمال شود یعنی نوشتن موردی است .

Fig 4.21 : (بررسی دستور beq)

دو ثبات می خوانیم (Read reg1, Read reg2) . برای اینکه بدانیم این دو رجیستر برابرند یا نه (محتوایشان) آنها را از هم کم می کنیم که این کار بوسیله ی ALU صورت می گیرد. هرگاه تفاضل صفر شد ، فلگ Zero ، Set می شود یعنی ۱ . این ۱ ، یکی از ورودی های گیت and در بالا است و ورودی دیگر توسط CU می آید که دستور Branch را فعال می کند.

Raz

wWw.it98.ir

www.it98.ir

Subject
YEAR: MONTH: DATE:

چرا shift Left 2 داریم؟ در اینجا offset بمعنی تعداد دستورات است که باید از روش jump نیم. در واقع این مربوط به طراحی است.

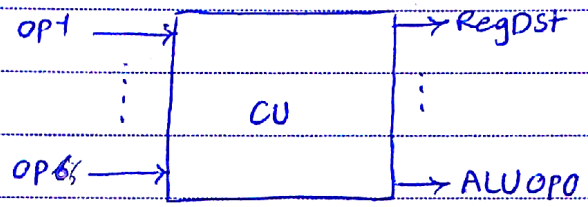
مستقل از دستوری که قرار است اجرا شود، PC+4 باید انجام شود بنابراین حتماً اجرا می‌گردد.

اگر مثلاً offset برابر با 6 باشد، ضرب در 4 می‌شود و به همان اندازه پرش به جلو یا عقب صورت می‌پذیرد تا offset واقعی بدست آید. offset × 4

Fig 4.22:

۶ بیت op بعنوان ورودی واحد کنترل و خروجی هم سیگنال های کنترلی هستند. اگر op=0 باشد، از روی funct تشخیص می‌دهد که چه دستوری است. بقیه شو خودمون بخونیم.

در این جعبه داریم:
(۹ خروجی و ۶ ورودی)



اگر بخواهیم مدار اینو پیاده‌سازی کنیم چکار باید بکنیم؟
باید یک یک سیگنال های خروجی را بصورت boolean بیان کرده یا از hardWiredControlLogic استفاده کنیم یا از microprogrammedControl استفاده کنیم.
در روش دوم عملاً جعبه بصورت داده شده را در حافظه می‌ریزیم:
(با ۶ بیت آدرس دهی می‌کنیم)

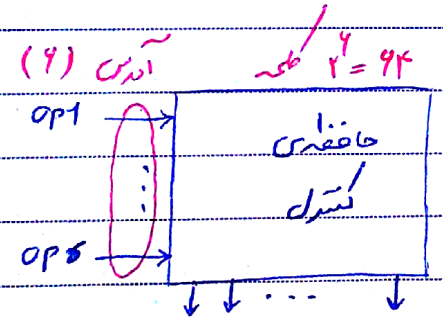


Fig 24:

اینجا به CU یک سیگنال jump افزوده شده، یعنی از کنترلر دستور jump صادر می‌شود.
دستور jump: ۶ بیت op، ۲۶ بیت آدرس و مابقی را از PC می‌گیریم.
۲۶ بیت آدرس را دوتا به چپ شیفت می‌دهد و ضرب در ۴ می‌شود (دستور مضربی از ۴ است پس برای رفتن به آنجا آدرس باید مضرب ۴ باشد) حالا ۴ بیت از PC [۳۱-۲۸] به آدرس ۲۸ بیتی Concat می‌شود و بعد از آن توسط mux که Select آن با کنترلر فعال شده حاصل را به داخل PC بازمی‌گرداند.

www.it98.ir

Subject
YEAR: MONTH: DATE:

نظم کوتاه به pipeline : (Fig 4.26)

CLK باید : 200 PS

همه‌ی دستورات Inst Fetch دارند چون طبق الگوریتم فون نیومن حتماً باید اجرا شوند.
بعضی دستورات Data access ندارند پس Total Time کمتری دارند.

اینجا دو تا انتخاب داریم:

۱- یا Total Time همه برابر باشد (و مساوی با بدترین Total Time باشد). این یک انتخاب محدودی است. این نوع طراحی ساده است. اشکال این است که همه چیز با کندترین دستور تنظیم شده و راندمان پایین است و وقت سیستم تلف می‌شود، مخصوصاً اگر دستورات کند دیر به دیر صورت بگیرند و دستورات سریع، زود به زود صورت بگیرند.

۲- یا از pipeline استفاده کرد

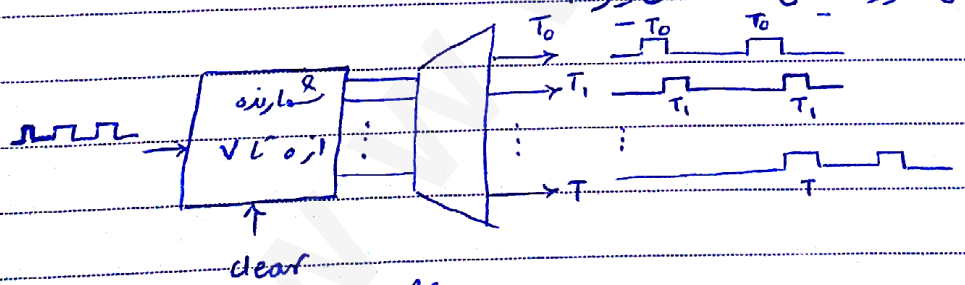
جلسه‌ی نهم — ۹۳/۱/۱۶

mips → ISA :
- مشخصات ثبات‌ها
- مجموعه‌ی دستورات
- نوع‌های قابل پردازش
- مدهای آدرس‌دهی
- توسط دستورات پردازنده
- قالب دستورات

هر چند طول دستورات ثابت و ۳۲ بیتی است، ولی زمان اجرای دستورات (متأسفانه یا خوشبختانه) برابر نیست.

چرا از ۳٫۵GHz به بالا سرعت پردازنده متوقف شد؟ چون در اینصورت پردازنده بسیار داغ می‌شود و عملاً قابل استفاده نیست. پس نمی‌توان فرکانس را زیاد کرد و باید هسته‌ها را بیشتر کرد.

اگر زمان اجرای دستورات متفاوت باشد، در اینصورت واحد کنترل یک Sequence Counter نیاز دارد تا بداند کدام دستور پی تمام می‌شود. یعنی قطعه‌ی زیر:



به ازای هر دستور، واحد کنترل باید Sequence مربوطه را تخمین دهد.

clear : وقتی یک دستور مثلاً در T_3 تمام شد، clear می‌شود تا دستور بعدی اجرا شود.

wWw.it98.ir

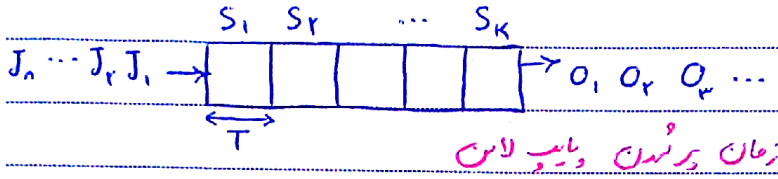
www.it98.ir

Subject
YEAR: MONTH: DATE:

در روش دوم پیچیدگی خاصی برای طراحی Control Unit نداریم ولی بعضی دستورات معطل می‌شوند.

معماری خط لوله یا Pipeline

برای داشتن سرعت بالا، نمی‌توان Pipeline را فراموش کرد.
معماری Pipeline سنکرون (همگام با یک clk پایه) است. پس واحدهای عملیاتی را شبیه به هم طراحی می‌کنند. واحد سریع بیش از یک کار انجام می‌دهد و کار واحد کند تقسیم می‌شود.



$t_{ex\ serial} = n \times K \times T$

زمان پر شدن پایپ لاین

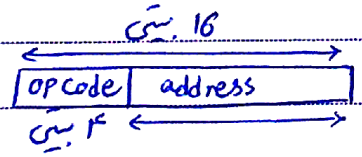
$t_{ex\ pipeline} = KT + (n-1)T$

$$S_{\text{تسریع}} = \frac{\text{زمان سریال}}{\text{زمان Pipeline}} = \frac{mkT}{(K+n-1)T} \Rightarrow S = \frac{mk}{K+n-1} = \frac{K}{1+\frac{K-1}{n}} \xrightarrow{n \to \infty} K$$

شرط موفقیت Pipeline این است که کارها همزمان انجام شوند یعنی زمان اجرایشان یکسان باشد.

در دستورات انشعاب شرطی، نمی‌دانیم پرش داریم یا نه. برای چنین دستوراتی Pipeline مناسب نیست و باعث می‌شود تعدادی دستور بلا استفاده fetch شوند. از عوامل دیگری که باعث انقطاع می‌شوند می‌توان قطع جریان برق و ... را نام برد.

مثال از کتاب مانو: طراحی واحد کنترل (PDP8)



قالب دستورات ۱۶ بیتی که OP آن ۴ بیتی است.
البته همه‌ی دستورات آدرس نیاز ندارند پس OP می‌تواند تغییر کند.
معماری تک آدرسه است (مبتنی بر انباشتگر).

$R_3 = R_1 + R_2$ mips در : Add R_3, R_1, R_2 (۳ آدرسه)

Add X → acc ← Mem[X] + acc

www.it98.ir

Subject
YEAR: MONTH: DATE:

در این پردازنده ۴ مد های آدرس دهی : مستقیم ← Add X

غیرمستقیم

ثباتی ← CLR A ، CLR E

ضمنی ← بدون اینکه عملگر خودش می داند به کدام آدرس باید مراجعه نماید.

سه نوع دستور : * نوع اول : دستورات دسترسی به حافظه و انجام عملی روی آن

۷ دستورالعمل حافظه ای داریم ،

AND	0	BUN	4
Add	1	BSA	5
Ld A	2	ISZ	6
ST A	3		

این پردازنده ؛ RISC نیست چون مستقیماً روی حافظه عملیات حسابی انجام می دهیم و بیش از load و Store با حافظه کار دارد.

* نوع دوم : دستورات ثباتی :

opcode	Reg. Operand
0111	

بخش دوم

opcode	I/O operand
1111	

* نوع سوم : دستورات I/O

VAHDAT

www.it98.ir

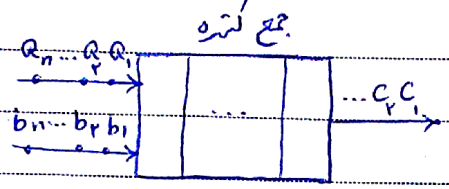
www.it98.ir

Subject
YEAR: MONTH: DATE:

جلسه ی دهم ۹۳/۱/۱۸

دو نوع پایپ لاین داریم: - پایپ لاین دستورالعمل (بررسی می کنیم)
" " حسابی

پایپ لاین حسابی: مثال:



$(c_i = a_i + b_i)$

Fig 4.33 :

مراحل کاری به ۵ بخش تقسیم شده است: IF, ID, EX, MEM, WB

EX: همیشه به معنی عملیات محاسباتی نیست بلکه برای محاسبه ی آدرس برای دستورات branch هم استفاده می شود. البته دو تا را همزمان انجام نمی دهد.

MEM: برای خواندن یا نوشتن از حافظه

WB: نتایج در Reg. File نوشته می شود. هرچه طبقات پایپ لاین زیادتر باشد، سرعت بیشتر است. چرا؟ WB، خروجی MUX به Reg. File و عمل شده در صورتی که عمل آنها باید مستقل باشد. چگونه ممکن است؟

تعداد طبقات پایپ لاین در پردازنده ها بین ۳ تا ۱۵ یا ۲۰ تا می باشد. چرا از این زیادتر نمی شود؟
چون کار پردازنده فی نفسه ریز و جزئی می باشد و نیازی نیست از آن ریزتر شود (دلیل اول).
چون ممکن است ما انشعاب شرطی یا وقفه داشته باشیم. در اینصورت مجبوریم کل پایپ لاین را خالی کنیم و هزینه ی این کار زیاد می شود (دلیل دوم).
بخاطر وابستگی دستورات متوالی، ممکن است بعضی از بخش های پایپ لاین ممکن است بیکار باشند
بنابراین افزایش بخش های پایپ لاین از یک نقطه به بعد به صرفه نیست.

جواب سوال اول:
نوشتن و خواندن مستقل از هم انجام می شوند یعنی Reg. File می تواند همزمان از ۲ جا بخواند و در یک جا بنویسد.

Fig 4.27: بالایی: اجرای سنتی دستورات - دستور دوم صبر می کند که دستور اول تمام شود. چون کارها تقسیم نشده اند (اجرای back-to-back).
دومی: کارها تقسیم شده اند (Pipeline). در هر clk، یک عمل برای هر دستور انجام می شود.
برای دستور Reg، 100ps از clk استفاده می شود. ولی در واقع عمل Decode در حال اجرا شدن است. نیمه ی اول clk، عمل Decode و در نیمه ی دوم، Read Operand را انجام

VAHDAT

www.it98.ir

Subject
YEAR: MONTH: DATE:

می دهد. البته به واقع شاید Decode ، 10ps (کمتر از 100ps) طول بکشد.
کاری که بیشتر از همه طول می کشد ⟸ مراجعه به حافظه. در اینجا فرض کرده ایم که آن هم 200ps طول بکشد که در اینصورت حتماً باید از حافظه ی cache استفاده کنیم.
در مرحله ی Data access اگر Data آماده نباشد، پایپ لاین باید صبر کند و نباید از کار بیفتد.
به این عمل bubble می گویند (Fig 4.30). مثلاً bubble در اینجا دستور no Operation است.

بعضی دستورات به Mem نیازی ندارند مثلاً add. برای این دستورات چکار باید بکنیم؟
دو رویکرد پیش روی ماست:
۱- این مرحله را رد کنیم و به عمل بعدی بپردازیم. البته اگر Conflict در کار نباشد می توانیم این کار را بکنیم. سرپردازنده های امروزی این روش تا حدی پیاده سازی می شود. پیچیدگی این روش زیاد است و نیاز به برنامه ریزی دارد.
۲- این مرحله را کاری نکنیم و تا دستور بعدی صبر کنیم (bubble).

Pipeline Hazards: چیزهایی که مزاحم Pipeline و باعث افت کارایی آن شوند.
سه نوع مخاطره داریم:
۱- مخاطرات ساختاری (Structural Hazard): به ساختار Pipeline مربوط است. یعنی منابع ما محدود هستند (مثلاً Data bus ، ALU و پورت ها). راه حل: افزایش منابع.
۲- مخاطرات داده ای (Data Hazard): از همه مرسوم تر و رایج تر است و به معنی وابستگی داده ای است.

```
add $S0, $t0, $t1
add $t2, $S0, $t3
```

در بند دوم Reg ها خوانده می شوند و در بند سوم، محاسبات صورت می گیرد. پس در اینجا در بند دوم ما S0 ای را می خوانیم که آماده نیست. این چالش اصلی پیش روی Pipeline است.
یک راه حل: bubble است. به معنی بین وابستگی صبر کنیم تا دستور قبلی کامل شود.
به تعداد CLK هایی که باید صبر کرد، noOperation اجرا کنیم (برنامه نویسی).
خود Compiler، دستورات خنثی اجرا کند.
راه میان بر هم می توان استفاده کرد تا قبل از WB، خروجی بصورت زودهنگام به واحد بعدی ارسال گردد (Fig 4.29). پیاده سازی این روش، پیچیدگی بیشتری دارد.
جابجا کردن دستورات (کامپایلر و برنامه نویس).

wWw.it98.ir

www.it9

Subject
YEAR: MONTH: DATE:

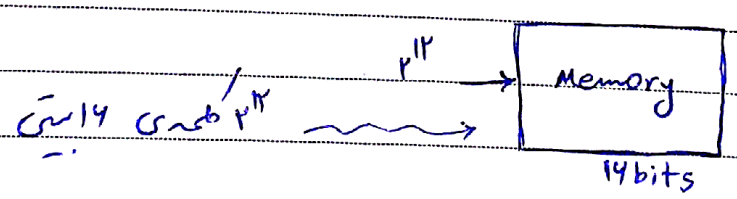
۳- مخاطرات کنترلی (Control Hazard): از نظر کنترلی مخاطره‌ای در برنامه رخ دهد.
مثلاً هنگام انشعاب شرطی روی می‌دهد.
Fig 4.31: در پایان مرحله‌ی ALU، می‌فهمیم که پرش باید بکنیم یا نه.

روش‌های مقابله با این hazard:
تزریق حباب
با توجه به احتمال پرش، دستور بعدی را اجرا بکنیم یا نه (Static Prediction).
با توجه به مراحل قبلی حلقه، حدس بزنیم که پرش می‌کند یا نه (Dynamic Prediction).
البته روش‌های پیشگویی، جزو اسرار تجاری هر شرکت هستند.

جلسه‌ی یازدهم:
طراحی واحد کنترل کامپیوتر PDP8:
در این پردازنده، داده‌ها و دستورات ۱۶ بیتی هستند. ثباتهای این پردازنده:

		تعداد بیت‌ها			تعداد بیت‌ها
DR	Data Register	16	PC	program Counter	12
AR	Address "	12	INPR	Input Reg	8
* AC	Accumulator "	16	OUTR	Output Reg	8
IR	Instruction "	16	TR	Temporary Reg	16

از این ۸ ثبات، جداگانه به چند ثبات می‌توان دسترسی داشت؟
کلیدی‌ترین ثبات داده ← AC همه‌ی محاسبات و اعمال با AC انجام می‌شود.
دستورات ۱ عملوندی هستند (ماشین تک آدرسه است).
حافظه چند در چند است؟



هم PC، ۱۲ بیت دارد (یعنی می‌تواند به 2^{12} خانه اشاره کند) و هم AR ۱۲ بیتی است پس 2^{12} word داریم. چرا word ها ۱۶ بیتی هستند؟ چون داده‌ها و دستورات ۱۶ بیتی هستند.

VAHDAT

it98.ir

Subject
YEAR: MONTH: DATE:

معماری این پردازنده ، پرینستون می باشد.

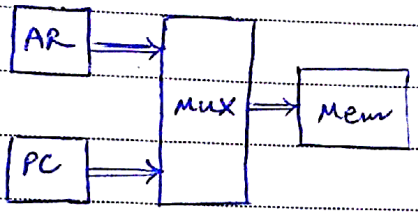
به DR ، MDR و BR ، MBR هم می گویند.

برای این که پردازنده سریع تر باشد، یا باید از Cache استفاده کنیم یا ثبات های بیشتری داشته باشیم تا اطلاعات مورد نظر، در دسترس باشد (رویکرد RISC).

دو راه برای اینکه به حافظه برسیم:

یا برای Instruction به Mem رجوع می کنیم یا آدرس داده را می خواهیم. (AR ← PC)

پس به Mux نیاز داریم.

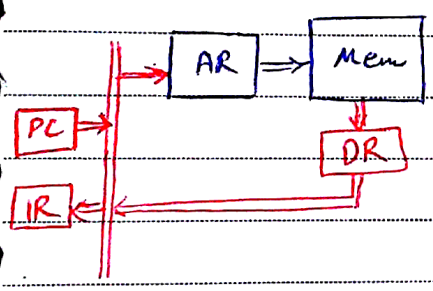


معادل AR در هسته پرسون چیست؟

AR یکی از آن ۳۲ رجیستر بود که از ALU رد می شد و effective address تولید می شد و بعد به PC می رفت و مستقیماً متصل به حافظه نبود.

$t_0 = AR \leftarrow PC$

$t_1 = IR \leftarrow M[AR]$



رویکرد دیگر: در این روش اگر PC بخواهد به Mem دسترسی پیدا کند باید پیشاپیش در یک CLK در AR ریخته شود و سپس در CLK بعدی عمل fetch انجام شود.

پس ۴ رویکرد طراحی داریم:

① $t_0: IR \leftarrow M[PC]$

② $t_0: AR \leftarrow PC$
$t_1: DR \leftarrow M[AR]$
$t_2: IR \leftarrow DR$

③ $t_0: AR \leftarrow PC$
$t_1: IR \leftarrow M[AR]$

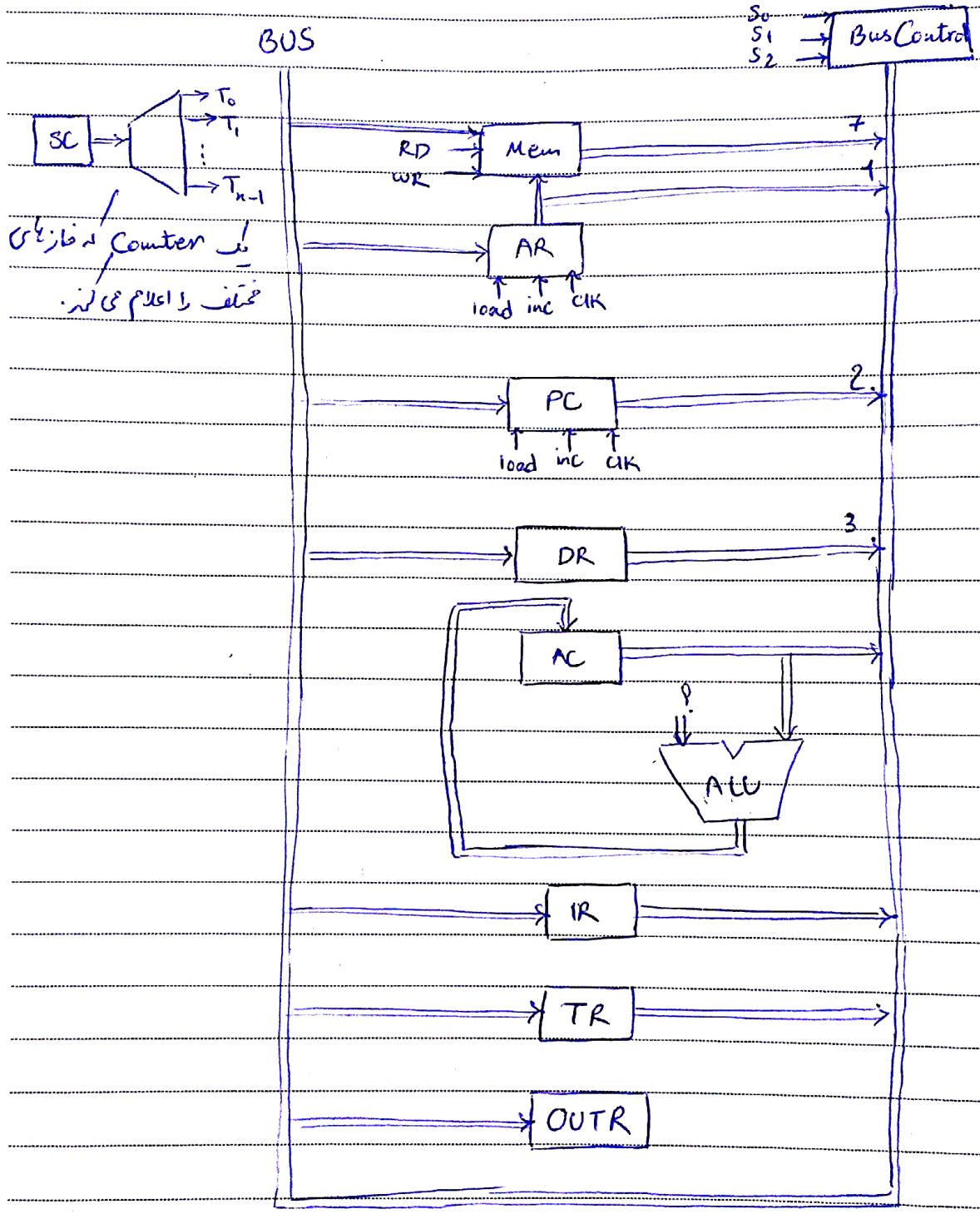
④ $t_0: DR \leftarrow M[PC]$
$t_1: IR \leftarrow DR$

www.it98.ir
wWw.it98.ir

www.it98

Subject
YEAR: MONTH: DATE:

پیاده سازی ۳ در اینجا انجام شده است (سلیقه ی طراحی) :



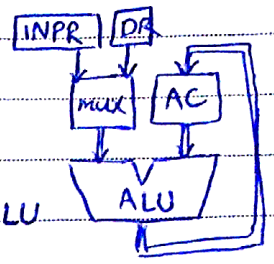
www.it98.ir

VAHDAT

it98.ir

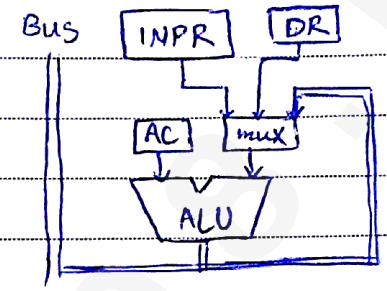
Subject
YEAR: MONTH: DATE:

خروجی دوم ALU را از کجا بیاریم؟ گزینه‌ی اول:

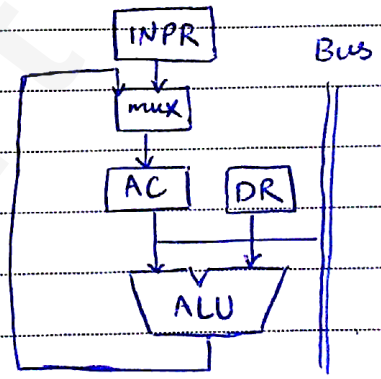


اگر ALU به BUS متصل بود می‌توانستیم خروجی آن را داشته باشیم و روی یکی از ثباتهای دیگر ذخیره کنیم. در اینجا AC برای ما اینکار را انجام می‌دهد. پس گزینه‌ی دوم این است که خروجی ALU را به گونه‌ای به Bus وصل کنیم.

گزینه‌ی سوم:



گزینه‌ی چهارم: Input می‌تواند مستقیماً در ALU ریخته شود.



سوال: چرا INPR ورودی و OUTR خروجی ندارند؟

چون INPR از BUS ورودی نمی‌گیرد، بلکه از وسایل جانبی می‌گیرد و همین طور OUTR خروجی را به bus نمی‌دهد بلکه به وسایل جانبی می‌دهد. یعنی فرض ما این است که وسایل جانبی همانند کی بورد... دارای ثبات می‌باشند.

VAHDAT

www.it98.ir

www.it98.ir

Subject
YEAR: MONTH: DATE:

انواع دستورات: * گونه‌ی اول: دستورات مراجعه به حافظه یا Memory Instruction

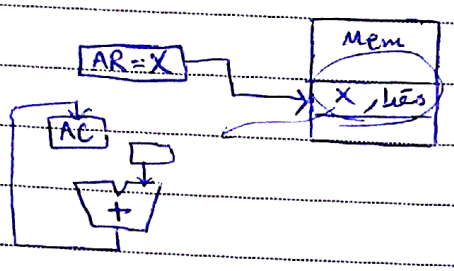
I: indirect Bit

I	OP	address
1	3	12

(16 بیت)

چون 2^{12} کلمه داریم (چون AR، ۱۲ بیتی است).

۲ تا مد آدرس دهی داریم: مستقیم و غیر مستقیم، یعنی مراجعه به حافظه یا به شکل مستقیم صورت می‌پذیرد یا غیر مستقیم.



مستقیم مثل Add X. شکل:
به آدرس X مراجعه کن، مقدار آن را با ac جمع کن و نتیجه را در ac بریز.

آدرس دهی غیر مستقیم: به خانه‌ای که اشاره می‌کند، آدرس محتوای X در آن است.

* گونه‌ی دوم: دستورات ثباتی یا Register Instruction

0111	RegOperation
OP	

* گونه‌ی سوم دستورات I/O

1111	I/O Ope...

چند مثال برای این دستورات:

	I=0		I=1
AND	0000		1000
Add	0001		1001
Load	0010		⋮
STA	0011		
BUN	0100	شبیه پرش	
BSA	0101	Branch & Save Address	
ISZ	0110	→ Increment & Skip if Zero	1110
CIA	0111 0000 0000 000	: AC←0	
CLE	0111 01 00 0000 0000	: E←0	
CMA	0111 0100 0000 0000	: AC←$\overline{AC}$	
⋮			

تنها دستور شرطی که برای پرش بکار می‌رود ⇐

۱۲ دستور ثباتی

دستورات حافظه داریم

www.it98.ir

it98.ir

Subject
YEAR: MONTH: DATE:

مراحل اجرای دستور (کلاً ۲۵ دستور داریم) :

$T_0 : AR \leftarrow PC$

$T_1 : IR \leftarrow M[AR], PC \leftarrow PC+1$ } fetch

$T_2 : Decode, AR \leftarrow IR(11-0), I \leftarrow IR(15)$ → Flag

اقدامی پیش دستانه : در حال Decode کردن (فهمیدن OP برای تشخیص نوع دستور) بخشی از InstructionReg را در AR می‌ریزد.

VAHDAT

www.it98.ir

wWw.it98.ir

www.it98.ir

مراحل اجرای دستور (طا ۲۵ دستور داریم): ↓

$T_0: AR \leftarrow PC$

$T_1: IR \leftarrow M[AR], PC \leftarrow PC+1$ } Fetch

T_2: Decode, $AR \leftarrow IR(11-0)$, $I \leftarrow IR(15)$ → Flag

اقدامی پیش دستانه: در حال Decode کردن (فهمیدن OP برای تشخیص نوع دستور) بخشی از Instruction Reg را در AR می‌ریزد.

جلسه‌ی دوازدهم:

طراحی واحد کنترل PDP8:

پس از واکشی دستور العمل، دستور العمل ۱۶ بیتی روی bus قرار می‌گیرد و IR آن را برمی‌دارد. قسمت OP معمولاً به یک دیکدر عرضه می‌شود و دیکدر خط دلخواه را فعال می‌کند.

OP = ۳ بیت OP خالص و یک بیت I. پس ۸ سیگنال داریم:

$000 \rightarrow D_0$
$001 \rightarrow D_1$
$\vdots$
$111 \rightarrow D_7$

این سیگنال به قسمت های مختلف ارسال می‌شود و بنا به طراحی کارهای مختلفی انجام می‌دهد (مثلاً ورودی En یک mux را فعال می‌کند). این سیگنال ها ($D_0 - D_7$) سیگنال های شناخت کدهای عملیات یا opcode هستند.

* مراحل طراحی: پیش از همه که ISA است.

① همه‌ی گزاره‌های RTL پردازنده را برای انجام کارهای لازم برای تحقق یا پیاده سازی ISA بیان کنید. «گام های بعد در جلسات بعد»

گزاره‌های RTL دو بخش دارند. بخش کنترل یا منبع و بخش اجرایی.

توسط CLK تنظیم ↑ لاشور

	کنترل	بخش اجرایی
فاز اول	T_0:	$AR \leftarrow PC$ → microOperation یا ریز دستور
دوم	T_1:	$IR \leftarrow M[AR], PC \leftarrow PC+1$
⋮	T_2:	$\boxed{\{D_7, \dots, D_0\} \leftarrow Decode}$ (IR(14,12)) OP ۳ بیت خالص, $I \leftarrow IR(15)$, $AR \leftarrow IR(11,0)$

Fetch & Decode

⚠: گزاره‌ی مشخص شده RTL نیست.

در گزاره‌های RTL، حتماً باید انتقال از بیتی به بیت دیگر صورت پذیرد. Reg می‌تواند تک بیتی باشد مثل I در دستور $I \leftarrow IR(15)$ یا چند بیتی. اینو برای خوانایی برنامه نوشتیم.

اگر یک دستور در مثلاً T_5 تمام شد، می‌نویسیم $SC \leftarrow 0$ یعنی توالی ساز را reset می‌کنیم.

www.it98.ir

www.it98.ir

www.it98.ir

wWw.it98.ir

www.it98.ir

شرط کنترل دستور

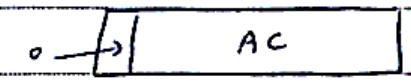
CME rb_8: $E \leftarrow \bar{E}$; Complement E

CIR rb_7: $AC \leftarrow$ → دوران به راست

با شروع CIK بعد آنچه انجام داده‌اید در نظر گرفته می‌شود.
پس از انجام اینها، SC را صفر می‌کند چون کارمان تمام شده است.

Skip if Accumulator is Possitive ← (SPA) rb_4: if (AC(15) == 0 then PC ← PC+1

skip if possitive



هرگاه عدد با صفر شروع شود، عدد مثبت است.

skip: پریدن، جهش

یعنی اگر AC=0 بود از روی دستور بعدی می‌پرد. مثلاً

```
I1
I2
 :
SPA
BUN  L1
BUN  L2
 :
```

دستور Halt (ایست): HLT: SC ← 0

وقتی مشکلی روی دهد و بخواهیم عیب‌یابی کنیم از این دستور استفاده می‌کنیم.

جلسه‌ی سیزدهم: ۹۳/۱/۳۰

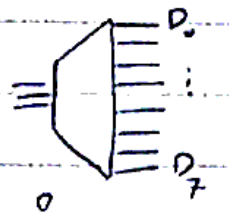
تمام گزاره‌های R.T.L ریزعملوندهای پردازنده را برنامه‌نویسی می‌کنیم:

opcode	نام دستور	خروجی دیگر	کاری که انجام می‌دهد
000	AND	D_0	$AC \leftarrow AC \wedge M[AR]$
001	ADD	D_1	$AC \leftarrow AC + M[AR]$, $E \leftarrow C_{out}$ (Carry Flag)
010	LDA	D_2	$AC \leftarrow M[AR]$
011	STA	D_3	$M[AR] \leftarrow AC$

www.it98.ir

100	BUN	D_4	$PC \leftarrow AR$
101	BSA	D_5	$M[AR] \leftarrow PC,\ PC \leftarrow AR+1$
110	ISZ	D_6	$M[AR] \leftarrow M[AR]+1$
			$if(M[AR]+1=0)\ then\ PC \leftarrow PC+1$

I بدون opcode



در شروع کار باید صفر باشد $SC \leftarrow 0$

$T_0: AR \leftarrow PC$

$T_1: IR \leftarrow M[AR],\ PC \leftarrow PC+1$

$T_2: I \leftarrow IR(15),\ Decode\ IR(12-14),\ AR \leftarrow IR(0-11)$

$D'_7 I T_3: AR \leftarrow M[AR]$

$D'_7 I' T_3: \emptyset$

$D_7 I' T_3:$
- $r: SC \leftarrow 0$
- $rb_{11}: AC \leftarrow 0$ CLA
- ⋮
- $rb_0: S \leftarrow 0$ HLT

دستور 111 یعنی D_7 ، بسته به حالت های b ، کاری انجام می دهد:

T_3 : آدرس نهایی یا موثر در AR قرار گرفته است و آماده ی مصرف برای جستجوی عملوند می باشد.

(گام بین fetch & Decode)

اجرای دستورات ثباتی 4CLK طول می کشد و سریع انجام می شود (۴ مرحله دارد).

گزاره های RTL دستورات:

D_0: AND $\Rightarrow$ $D_0T_4: DR \leftarrow M[AR]$ برمی گردد.

$D_0T_5: AC \leftarrow DR \wedge AC,\ SC \leftarrow 0$ دستور تمام شده و به اول حلقه ی فون نیومن برمی گردد.

D_1: ADD $\Rightarrow$ $D_1T_4: DR \leftarrow M[AR]$

$D_1T_5: AC \leftarrow DR + AC,\ SC \leftarrow 0,\ E \leftarrow C_{out}$

wWw.it98.ir

www.it98.ir

D_2 : LDA $\Rightarrow$ D_2T_4 : $DR \leftarrow M[AR]$

D_2T_5 : $AC \leftarrow DR$, $SC \leftarrow 0$ (AC به خروجی ALU وصل است)

کاستی : دستور در ۵ مرحله اجرا می‌شود

مزیت : کنترل AC را آسان‌تر می‌کند

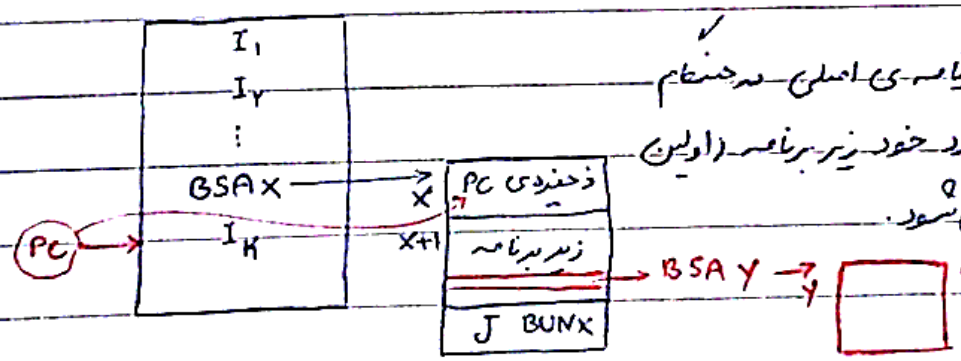
D_3 : STA $\Rightarrow$ D_3T_4 : $M[AR] \leftarrow AC$, $SC \leftarrow 0$

D_4 : BUN $\Rightarrow$ D_4T_4 : $PC \leftarrow AR$, $SC \leftarrow 0$

جلسه‌ی چهاردهم : ۹۳/۲/۱

در طراحی واحد کنترل تمام ریزدستورها را به زبان RTL بازنویسی می‌کنیم

دستور BSA روشی ارزان برای جلوگیری از استفاده از پشته است، چون پشته نداریم.



آدرس مراجعت به برنامه‌ی اصلی در هنگام صدا زدن زیربرنامه نزد خود زیربرنامه (اولین خانه‌ی آن) ذخیره می‌شود.

می‌توان داخل این دستور؛ BSA دیگری صدا زد.

سوال؟ چگونه با چنین پردازنده‌ای اجرای بازگشتی داشته باشیم؟

برنامه‌ی فاکتوریل را به زبان PDP8 بنویسید.

دستور ISZ : D_6 : ISZ X

خواندن معلوم (شماره‌ی حلقه) از حافظه

D_6T_4 : $DR \leftarrow M[AR]$

اضافه کردن آن

D_6T_5 : $DR \leftarrow DR+1$

D_6T_6 : $M[AR] \leftarrow DR$, if (DR = 0) then $PC \leftarrow PC+1$, $SC \leftarrow 0$

نوشتن مقدار جدید (شمارنده‌ی حلقه) در حافظه

www.it98.ir

I	ISZ	Countable

⇓ AR

L1: :

ISZ Count

BUN L1

Next Instruction

Count | -10 |

معادل در زبان C

```
ISZ Count:
    Count++;
    if (Count==0)
        PC++;
```

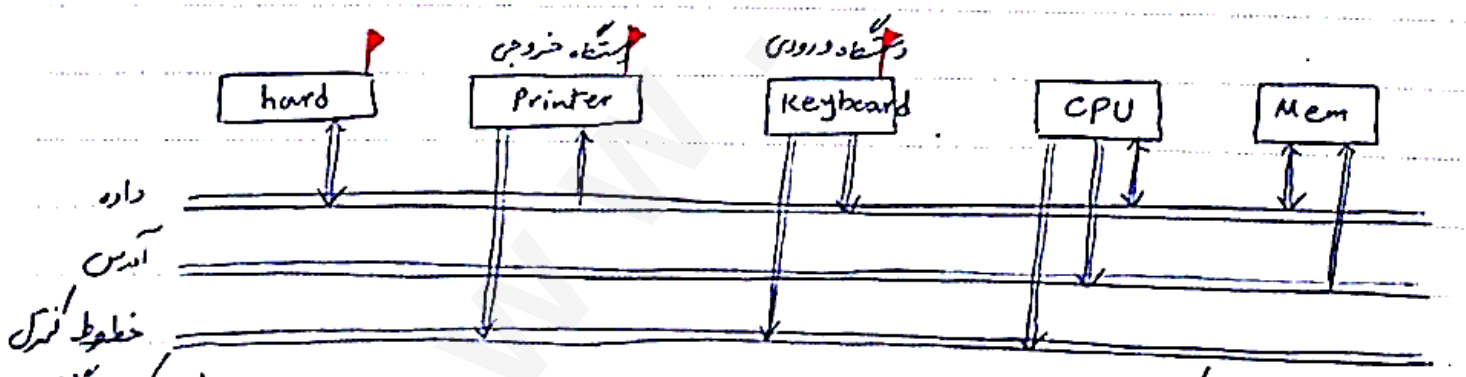
pc Relative addressing تعمیم ISZ است چون در اینجا pc یک خانه جلو می رود ولی در pc relative چند خانه جلو می رود.

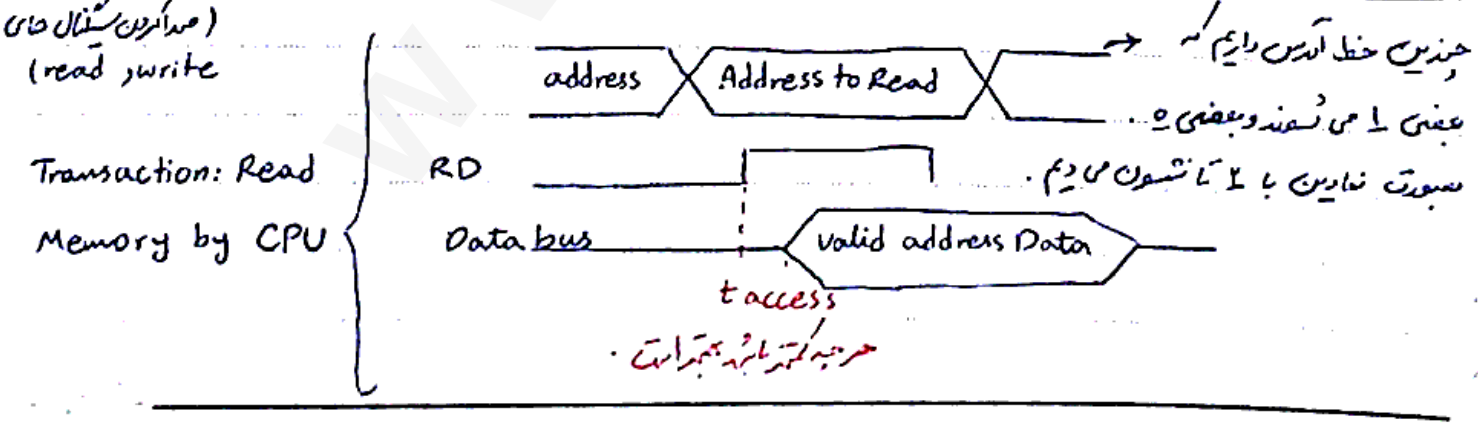
برای ایجاد شرط حلقه (یعنی شرط بزرگتر یا کوچکتر بودن): دستور Sub و 2's Comp نداریم پس اول 1's Comp را بدست می آوریم سپس +1 می کنیم تا 2's Comp بدست بیاید. سپس از روی آن تفریق را بدست می آوریم. اگر حاصل منفی شد عدد اول کوچکتر بوده است و اگر حاصل مثبت شد عدد اول بزرگتر بوده است. یعنی از دستورات Skip if Negative/Positive استفاده می کنیم. البته باید توجه کنیم که عدد علامتدار است یا بی علامت که این را در فصول آینده بررسی می کنیم.

دستور ISZ دستور حافظه ای است چرا که Counter که اینجا ذکر شده توی حافظه است.

بررسی دستورات I/O:

دو روش مراجعه به دستگاه های جانبی داریم: ۱- سرکشی یا polling ۲- وقفه دهی یا Interrupt





چندین خط آدرس داریم که بعضی 1 می شوند و بعضی 0.
بصورت نمادین با ↕ نشون می دیم.

هرچه کمتر باشد بهتر است.

wWw.it98.ir

www.it98.ir

اگر فرمان نوشتن صادر می‌شد؟ باز هم زمانی لازم داشتیم که حافظه بتواند بنویسد پس برای write Data هم یک t access داریم که عموماً در پردازنده‌ها: write taccess = Read taccess.

* روش سرکشی:

مثال: آیا صفحه کلید داده‌ای برای ورود دارد؟ آیا داده‌ای برای ارسال به چاپگر داریم؟ پس هرکدام از این Device ها یک Flag یا پرچم دارند که وقتی به ۱ است، به معنی آماده بودن است یا به این معنی است که داده‌ای دارند.

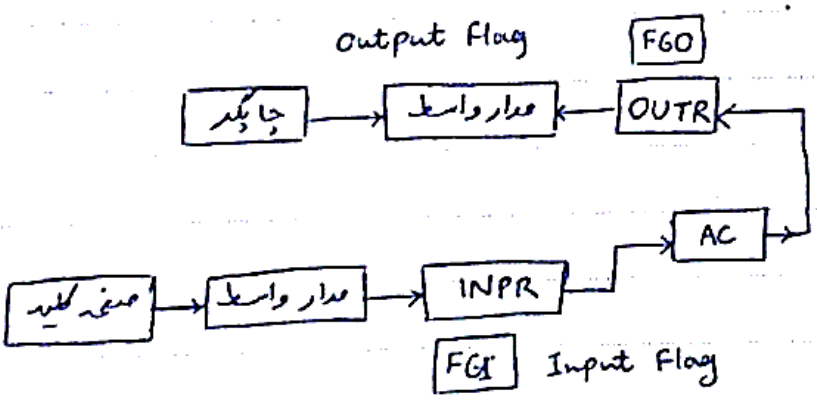
CPU با مراجعات متوالی، سرکشی یا استعلام می‌کند که آیا Device ها آمادگی برای پذیرش داده یا حرفی برای گفتن دارند یا نه؟ در این روش برنامه‌نویس داخل برنامه input، output و check Flag را انجام می‌دهد. یعنی یک حلقه می‌نویسد که این کارها را مکرراً انجام دهد.

* روش وقفه‌دهی:

اجزا به CPU متصل می‌باشند و در مواقع لزوم InterruptRequest می‌دهند.

وقتی درخواست زیاد باشد، روش سرکشی بهتر است. در اینصورت دستورات چک کردن پرچم را خودمان تعبیه می‌کنیم. ولی وقتی بندرت داده وارد یا خارج می‌شود روش دوم به صرفه‌تر است.

قسمت I/O در این پردازنده:



هرکدام از Device های ورودی و خروجی فضایی نیاز دارند که آدرس‌شان در آن قرار بگیرد. البته هرکدام یک حافظه هم دارند تا موقتاً اطلاعات در آن قرار بگیرد. چون سرعت CPU و سرعت این ادوات مثل هم نیست و سرعت CPU خیلی بیشتر است. هم‌چنین ممکن است CPU در آن لحظه آماده نباشد. اطلاعات باید در یک جایی ذخیره شوند تا CPU حاضر شود.

www.it98.ir

همه در فاز T_3 اجرا می‌شوند

$D_7 I (T_3) = P \qquad IR(i) = b_i$

دستورات I/O :

I	III	b_i

I D_7

P: $SC \leftarrow 0$

INP: pb_{11}: $AC(7-0) \leftarrow INPR$, $FGI \leftarrow 0$;

OUT: pb_{10}: $OUTR \leftarrow AC(7-0)$, $\boxed{FGO \leftarrow 0}$;

چاپگر آماده‌ی چاپ است. وقتی چاپگر خالی باشد 1 می‌شود و وقتی چاپگر پر باشد 0 می‌شود.

SKI: pb_9: if (FGI = 1) then $PC \leftarrow PC+1$;

pb_8: if (FGO = 1) then $PC \leftarrow PC+1$;

ION: pb_7: $IEN \leftarrow 1$;

IOF: pb_6: $IEN \leftarrow 0$;

maskable or unmaskable

اگر IEN=1 باشد به interrupt رسیدگی می‌شود در غیر اینصورت رسیدگی نمی‌شود.

۱۳۹۳/

جلسه‌ی پانزدهم

دسترسی به دستگاه جانبی و مراجعه به آن: ۱- سرکشی ۲- وقفه

سرکشی یا polling: پردازنده یکی یکی به سراغ دستگاه‌های I/O می‌رود و از آنها در مورد داشتن اطلاعات و یا آمادگی جهت پذیرش اطلاعات استعلام می‌کند.

برای اینکه یک داده را تکراری تلقی نکنیم و داده بیهوده نباشد و با اطمینان بیشتر آن را بخوانیم؛ با Flag این کار را انجام می‌دهد. بدون چک کردن Flag: مثلاً printer: داده را می‌فرستیم اما کنترلی نیست و داده‌ها از بین می‌روند (مثل وقتی که printer: خاموش یا بدون کاغذ باشد).

UDP ← روشی که داده‌ها را فرستاده‌ایم ولی ممکن است از بین بروند.

سرعت بالاتری دارند مثل بسته‌های صدا و تصویر

برای جلوگیری از اشباع شدن شبکه

در برخی ارتباطات؛ ارسال مطلب بدون در نظر گرفتن آمادگی در مقصد است.

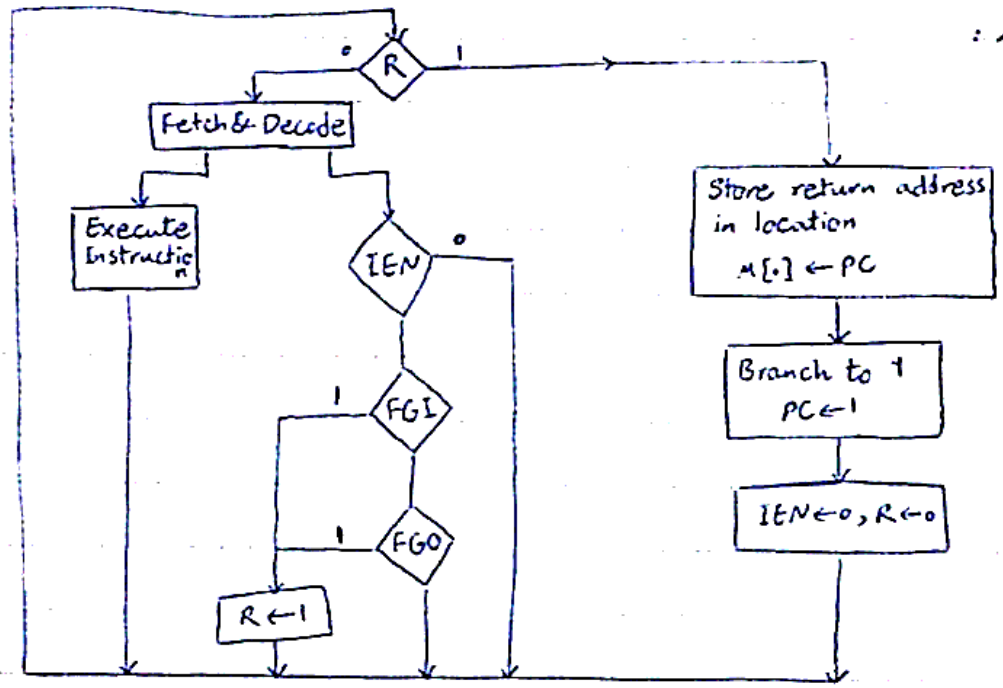
IEN یا Interrupt Enable: هرگاه این Flag؛ 1 باشد؛ آمادگی لازم برای پذیرش وقفه را دارد.

در این پردازنده؛ روش I/O مبتنی بر وقفه است. تغییر فلاپ R، معرف ورود به وقفه و اجرای دستورات روال وقفه است. وقفه در اینجا صرفاً به کار I/O می‌پردازد.

wWw.it98.ir

www.it98.ir

فلوچارت سیکل وقفه :



زمانی که در حال اجرای یک دستور هستیم هم عمل شامل چند ریز عملیات است و چون متوقف کردن ریز عمل و ذخیره کردن داده های آنها و این که مشغول کدام عمل هستند سخت است ؛ به همین دلیل اجازه می دهیم تا دستور تمام شود.

** اجرای وقفه معمولاً تا زمان تمام شدن اجرای دستور جاری به تعویق می افتد.

زمانی که وقفه تشخیص داده شود و وقفه پذیری مجاز باشد ؛ بعد از اتمام دستور الفعل وقفه می پذیرد.

به طور استاندارد ؛ خانه ی اول حافظه (خانه ی صفر را) به PC اختصاص می دهیم.

(این یک انتخاب مهندسی است).

آخرین دستور وقفه : BUN 0 ← پرش به خانه ی 0 و یافتن PC و ادامه ی مسیر

SKI (Skip Input) ، SKO (Skip Output) ← ورود و خروج برنامه را چک می کند.

داخل سیکل وقفه ؛ وقفه ی دیگری را نمی پذیریم.

Fetch: $R'T_0$: $AR \leftarrow PC$

$R'T_1$: $IR \leftarrow M[AR]$, $PC \leftarrow PC+1$

$R'T_2$: decode $IR(14-12)$, $AR \leftarrow IR(11-0)$, $I \leftarrow IR(15)$

$D_7 I T_3 : AR \leftarrow M[AR]$

$T_0 T_1' T_2' IEN (FGI + FGO) : R \leftarrow 1$

$RT_0 : AR \leftarrow 0 , TR \leftarrow PC$

$RT_1 : M[AR] \leftarrow TR , PC \leftarrow 0$

$RT_2 : PC \leftarrow PC+1 , IEN \leftarrow 0 , R \leftarrow 0 , SC \leftarrow 0$

مرحله‌ی سنتز یا پیاده سازی سخت افزاری :

۱- گام اول : همه‌ی گزاره‌های RTL را برحسب مقصد مرتب و دسته بندی کنید.

* گزاره‌های مرتب شده براساس مقصد AR :

می دانیم که AR از چه چیزهایی مقدار خواهد پذیرفت.

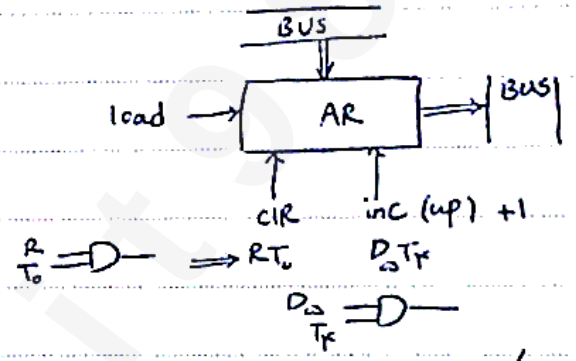
$R'T_0 : AR \leftarrow PC$

$R'T_2 : AR \leftarrow IR(11-0)$

$D_7' I T_3 : AR \leftarrow M[AR]$

$RT_0 : AR \leftarrow 0$

$D_5 T_4 : AR \leftarrow AR+1$



سیگنال‌های کنترل AR :

$CLR(AR) = RT_0$

$INC(AR) = D_5 T_4$

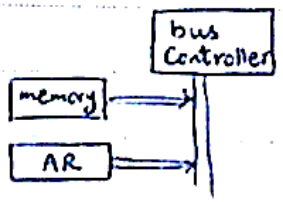
$load(AR) = R'(T_0 + T_2) + D_7' I T_3$

* $Read(memory) = R'T_1 + D_7' I T_3 + (D_0 + D_1 + D_2 + D_4) T_4$

۲- گام دوم : تمام گزاره‌های RTL برحسب مبدأ نوشته شوند تا بدانیم چه مقداری را باید روی گذرگاه قرار دهیم تا ثبات مقصد آن را درست دریافت نماید.

مبدأ AR : $D_4 T_4 : PC \leftarrow AR$

$D_5 T_5 : PC \leftarrow AR$

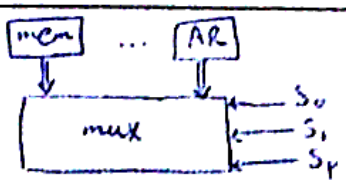


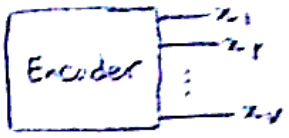
www.it98.ir

www.it98.ir

Subject: (۲۹)

Year: Month: Date:





x_1	x_2	...	x_v	S_0	S_1	S_2	
0	0	...	0	0	0	0	none
1	0	...	0	1	0	0	AR

$x_1 = D_2 T_2 + D_0 T_0$

$\vdots$

$x_v = R' T_1 + D_v' I T_2 + (D_0 + D_1 + D_2 D_4) T_2$

PAPCO

www.it98.ir

جلسه‌ی شانزدهم : ۹۳/۲/۸

طراحی واحد اجرایی مستلزم طراحی واحد حسابی است (Arithmetic Unit).

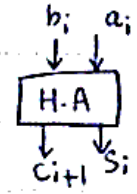
ISA ⇐ کارهایی که یک کامپیوتر مطابق مجموعه دستورات انجام می‌دهد.

- اجرای دستورات حسابی (ریاضی) مثل جمع، تفریق، ضرب، تقسیم و در مراحل بالاتر log، Sinx و ...
- منطقی یا logical
- جابه‌جایی یا Shift و دوران یا Rotate
- دستورات انتقال : ثبات به ثبات، ثبات به حافظه، حافظه به ثبات، حافظه به حافظه (ممکن است نداشته باشیم)، Input به ثبات، ثبات به Input، حافظه به I/O، I/O به I/O، I/O به حافظه. درباره‌ی انتقال چندان صحبت نمی‌کنیم و روی دستورات حسابی تمرکز می‌کنیم.

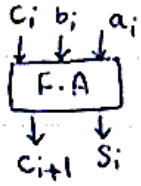
غیر از اینها چه دستوراتی داریم؟

- پرش یا انشعاب
- دستوراتی که اخیراً به پردازنده‌ها افزوده شده‌اند و دستورات مدیریت وقفه یا توان نام دارند.
- دستورات رویداد نگاری یا Event logging

جمع کننده یا adder : دو نوع جمع کننده داریم :



نیم افزا (نیم جمع کننده) = Half adder ⇐ دو بیت جمع می‌زند و دو بیت تولید می‌کند.

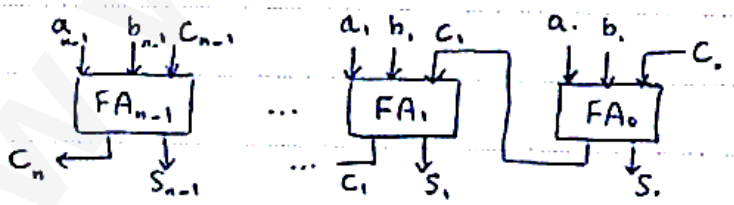


تمام افزا (تمام جمع کننده) = Full adder ⇐ سه بیت جمع می‌زند و دو بیت تولید می‌کند.

بیت Sum، وزن 2^i و بیت Carry وزن 2^{i+1} دارد.

ساده‌ترین جمع کننده: Ripple Carry adder Or Cascade adder

$$\begin{array}{cccccc} a_{n-1} & \cdots & a_1 & a_0 \\ b_{n-1} & \cdots & b_1 & b_0 \\ \hline C_n & S_{n-1} & \cdots & S_1 & S_0 \end{array}$$



دو موضوع در طراحی مد نظر است : ۱- کارآیی (سرعت یا زمان اجرا)
۲- حجم سخت افزار به کار رفته (پیچیدگی آن)

wWw.it98.ir

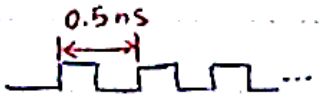
www.it98.ir

اگر زمان تولید c_0 تا c_n را Δ در نظر بگیریم، زمان تولید c_0 تا c_n چقدر است؟ $t_{c_n} = n\Delta = O(n)$

یک جمع ۳۲ بیتی چقدر طول می‌کشد؟

این به فرکانس (سرعت غلط) پردازنده بستگی دارد. اگر مثلاً $f = 2GHz$ باشد:

$$f_{CPU} = 2GHz \Rightarrow T_{CLK} = \frac{1}{2\times10^9} = 0.5\,ns$$



اگر جمع ۴ تا CLK داشته باشد می‌شود: $0.5 \times 4 = 2ns$

پس یک جمع ۴ نانوثانیه طول می‌کشد.

چگونه جمع را بهبود ببخشیم؟

$$\begin{cases} G_i = a_i b_i \\ P_i = a_i + b_i \end{cases}$$

تابع اکثریت $c_{i+1} = a_i b_i + c_i a_i + c_i b_i$

$c_{i+1} = G_i + P_i c_i$

کی Carry داریم؟

۱- خلق Carry در همان مرتبه (a_i و b_i یعنی $a_i b_i$ ۱ باشد) ⇐

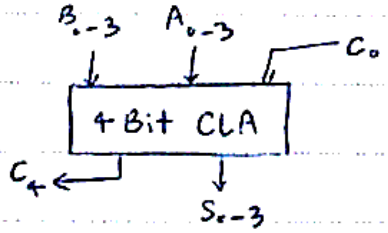
$$\begin{array}{c} c_i \\ a_i \\ b_i \\ \hline c_{i+1}\ s_i \end{array} \qquad \begin{array}{c} 0 \\ 1 \\ 1 \\ \hline 10 \end{array}$$

۲- انتشار Carry به شرط ورود Carry ($c_i(a_i+b_i)$)

$$\begin{array}{c} 1 \\ 0 \\ 1 \\ \hline 10 \end{array} \qquad \begin{array}{c} 1 \\ 1 \\ 0 \\ \hline 10 \end{array}$$

از این فرمول هم می‌شه نوشت: $G_i = a_i b_i$, $P_i = a_i \oplus b_i$

می‌خواهیم جمع‌کننده‌ی ۴ بیتی بسازیم:



CLA = Carry Look ahead Adder

$$c_1 = G_0 + P_0 c_0 = a_0 b_0 + (a_0 + b_0) c_0$$

$$c_2 = G_1 + P_1 c_1 = G_1 + P_1 G_0 + P_1 P_0 c_0$$

$$c_3 = G_2 + P_2 c_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

مثل اینکه یک تقدم Carry رو به ما داده؟ ما منتقل جمع می‌کنیم.

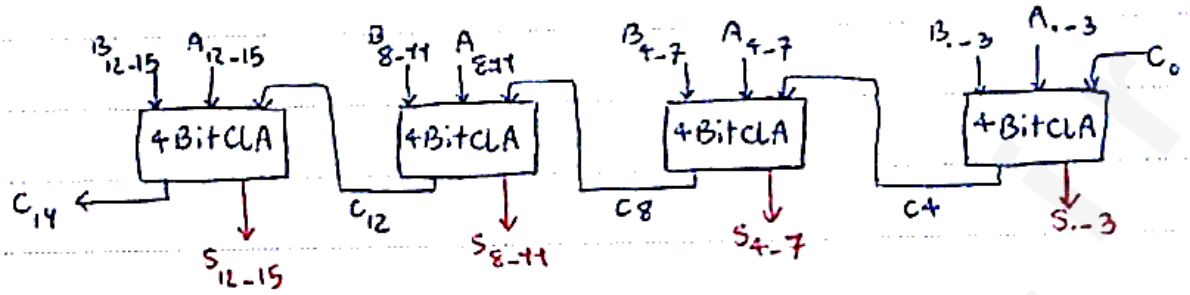
PaPCO

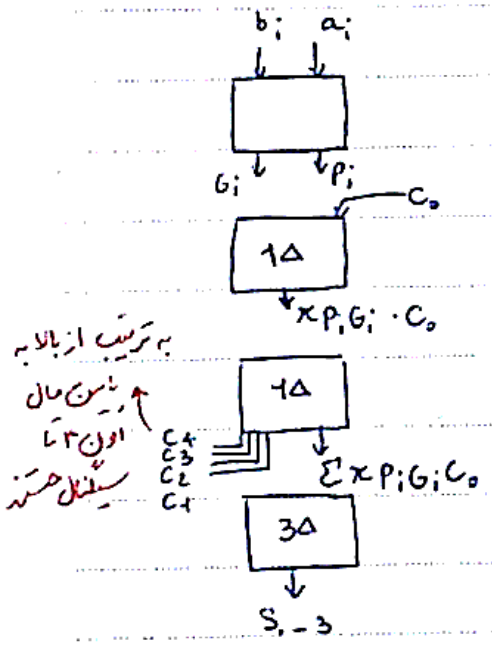
www.it98.ir

اینجا کارایی بالا رفته ولی پیچیدگی زیادی داریم. بزرگترین گیت AND، گیت OR، i+1 ورودی دارند. هرچه تعداد ورودی های گیت بیشتر شود تأخیر گیت هم زیادتر می شود.

جمع کننده ی ۱۶ بیتی به کمک CLA ۴ بیتی:

راه حل اول: Cascade کردن CLA ها به روش سنتی:





$t_{C_4} = 3\Delta$ $t_{C_8} = 5\Delta$ $t_{C_{12}} = 7\Delta$

$t_{C_{16}} = 9\Delta$ $t_{S_{15}} = 12\Delta$

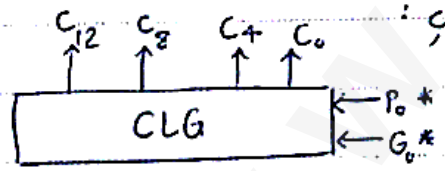
به ترتیب از بالا به پایین مال اون ۳ تا سیگنال هستند

تأخیر Carry-in به Carry-out به شرطی که $P_i G_i$ آماده باشد ۲Δ است.

اگر با Full adder حساب می کردیم:

$t_{C_{16}\ Ripple} = 32\Delta$ $t_{S_{15}} = 33\Delta$

آیا می توانیم مداری بسازیم که Carry ها را تولید کند؟ حل:



wWw.it98.ir

www.it98.ir

۱۳۹۳/۲/۱۳

جلسه‌ی هفدهم

مدار جمع کننده :

$$c_{i+1} = \overset{\text{generation}}{g_i} + p_i c_i$$

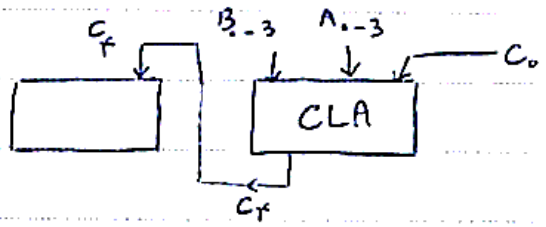
$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

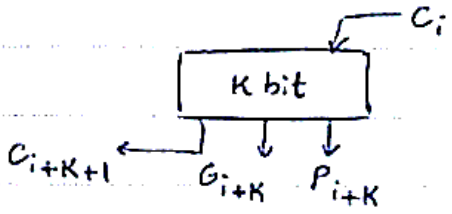
$$c_4 = \underbrace{g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0}_{g_{0-3}} + \overbrace{p_3 p_2 p_1 p_0}^{p_{0-3}} c_0 = g_{0-3} + p_{0-3} c_0$$

c_4 یک است اگر بلوک CLA، یک را generate کند یا نقش propagate داشته باشد، البته به شرطی که ورودی Carry داشته باشیم.

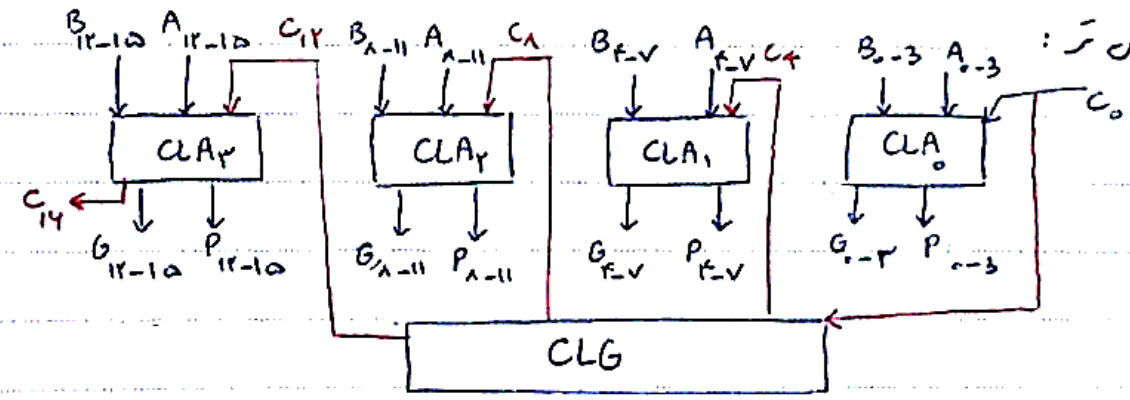


بطور کلی :

$$c_{i+K+1} = g_{i-i+K} + p_{i-i+K} c_i$$



تصحیح شکل قبل تر :



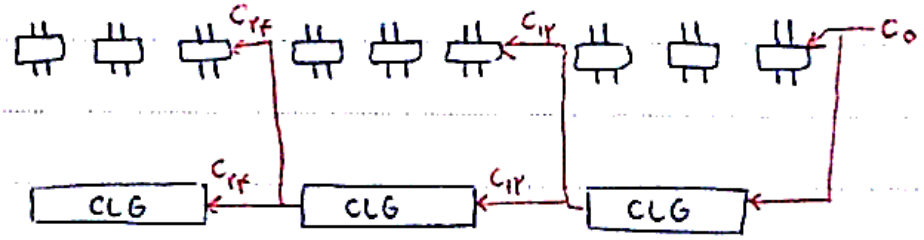
Carry Look ahead Generator

$$c_4 = G_{03} + P_{03} c_0 \qquad c_8 = G_{47} + P_{47} c_4 = G_{47} + P_{47} G_{03} + P_{47} P_{03} c_0$$

$$c_{12} = \cdots = G_{811} + P_{811} G_{47} + P_{811} P_{47} G_{03} + P_{811} P_{47} P_{03} c_0$$

www.it98.ir

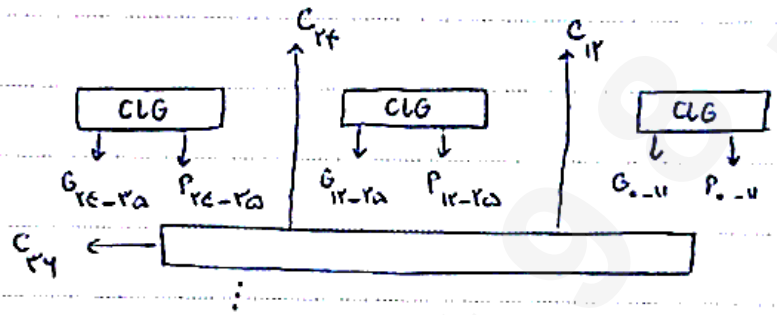
اگر بگوید فقط CLG سه بیتی داریم:



$C_{12} = g_{0-11} + P_{0-11} C_0$

تفسیر این عبارت چیست؟

C_{12} برابر است با ۱ اگر بین ۰ تا ۱۱ تولید Carry صورت گیرد (g_{0-11}) یا اینکه این بلوک قابلیت propagate کردن داشته باشد و در عین حال Carry ورودی به این بلاک، ۱ باشد.



بنابراین این الگوریتم، $O(\lg n)$ است (n = تعداد بیت ها).

یکی از تکنیک های جمع زنی (اختیاری است و در امتحان هم سوال مطرح نمی شود از این):

CSA: Carry Save adder ← برای پیاده سازی جمع حاصلضرب های میانی ضرب دو عدد یا همان ضرب های جزئی (Partial Product):

مثلاً

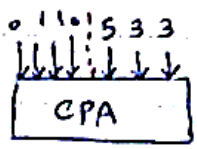
$$\begin{array}{r} 264 \\ +\ 379 \\ \hline 533 \\ 011 \\ \hline 643 \end{array}$$

جمع ها ← 533 → S = Sum Vector

Carry ها ← 011 → C = Carry Vector

Carry vector، باینری است ولی Sum Vector، دهدهی یا Decimal است. در این روش، Carry ها را جدا می نویسیم. فایده ی این روش آن است که هر بار دو تا عدد و یک بردار Carry را با هم جمع می کنیم. اگر در آخرین مرحله باشیم:

مثلاً:



جمع کامل را به مرحله ی آخر موکول می کنند که تا آن موقع صرفه جویی زیادی در زمان شده است.

wWw.it98.ir

www.it98.ir

پیاده سازی تفریق : تفریق ≡ جمع با مکمل

مکمل چیست؟ مکمل عدد مثلاً N، عددی است که اگر N با آن جمع شود، حاصل صفر شود.

چند روش جهت برای مکمل گیری :

مکمل ریشه : اگر ریشه (radix) ، r باشد، مکمل ریشه

مکمل کاستی ریشه (Diminished Radix complement) : $-N = C_{r-1}(N) = r^x - N - 1$

چند مثال : $N = +6$ ، $r = 2$ ، $m = 4\,bit$

$C_2(N_1) = 2^4 - 6 = 10_{10} = 1010_2$ $\qquad$ $C_2(N_2) = 2^4 - 3 = 13_{10} = 1101_2$

$C_2(0011) = (1101)_2$ $\qquad$ $C_2(0110) = (1010)_2$ $\qquad$ $C_1(0110) = (1001)_2 = 9 = 2^4 - 6 - 1$

این دو تعریف، تعاریف رسمی مکمل ۱ و مکمل ۲ هستند.

* تمرین اختیاری : مکمل ۱ یک عدد برابر است با متمم تک به تک حدیث.

سه روش برای گرفتن مکمل ۲ یک عدد باینری :

۱- حدیث را متمم گرفته، نتیجه‌ی کل را با ۱ جمع بزنیم.

۲- از راست به چپ تا اولین بیت ۱، کپی کنیم سپس حدیث را متمم نماییم.

۳- فرمول $r^x - N$ را حساب کرده به باینری تبدیل کنیم.

سوال : نمایش عکس با مکمل یک عدد برای عمل جمع است. می‌توانیم مکمل ریشه را حساب کنیم یا مکمل کاسته را یا علامت عکس + اندازه را. چرا ۳ تا روش داریم؟

وقتی می‌خواهیم تفریق کنیم، تفریق کننده لازم نداریم بلکه جمع کننده و یک مکمل کننده می‌خواهیم.

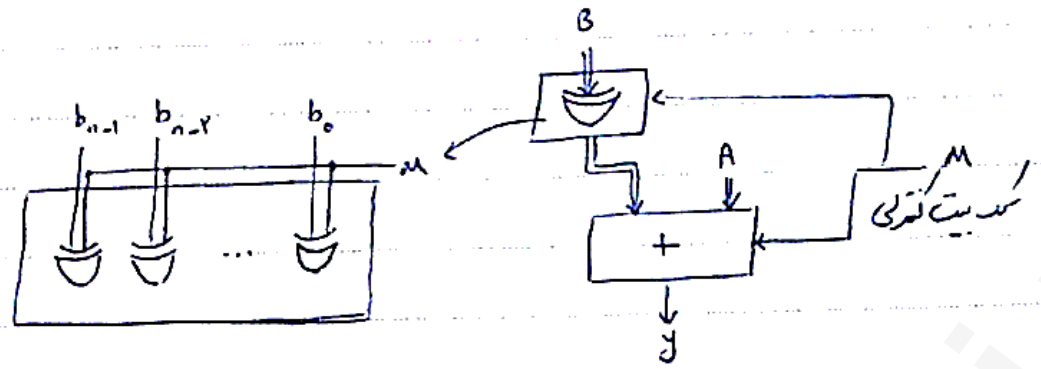
در روش Sign magn tude جمع و تفریق دشوار و در مکمل ۲ آسان است.

در روش

ضرب و تقسیم در روش Sign magnitude آسان و در روش مکمل ۲ دشوار است.

در روش ... Sign برای تشخیص دو تا صفر به سخت افزار اضافه نیاز داریم.

مدار جمع - تفریق کننده :



$m=0 \Rightarrow y=B$ $\qquad$ $m=0 \Rightarrow y=A+B$

$m=1 \Rightarrow y=\bar{B}$ $\qquad$ $m=1 \Rightarrow y=A+\bar{B}+1=A-B$

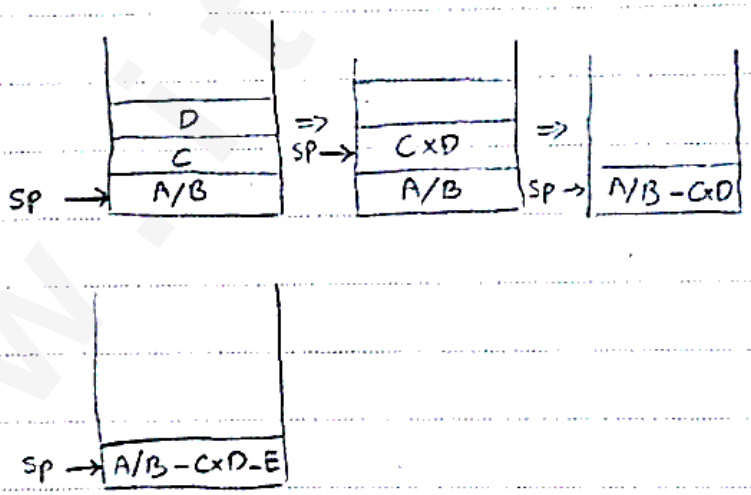
جلسه‌ی هجدهم ، حل تمرین ونمونه سوال

۱۳۹۳/۲/۱۵

۱- این عبارت را با دستورات یک ماشین پشته‌ای و یک ماشین سه آدرسه برنامه نویسی کنید.

$$X=\frac{A}{B}-C\times D+E$$

push A	push E
push B	Add
Div	pop X
push C	
push D	
Mul	
Sub ↗ ادامه	

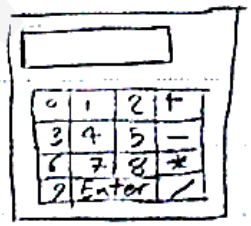


ماشین پشته‌ای ← بدون آدرس

" انباشتگر ← تک آدرسه (مبتنی بر انباشتگر)

دو آدرسه و سه آدرسه

$X=6/7-3\times4+9$



ماشین حساب پشته‌ای

6 7 / 3 4 * - 9 +

wWw.it98.ir

www.it98.ir

Subject:
Year. Month. Date. : :

مقایسه‌ی این دو برنامه با هم:

Div R_1, A, B
Mul R_2, C, D
Sub R_1, R_1, R_2
Add R_1, R_1, E

تعداد خطوط ماشین‌ها فرق دارد. در یک ماشین ۳ آدرسه، دستورات غنی‌تر هستند پس تعداد خطوط برنامه کوتاه‌تر می‌شود چون در یک دستور چند کار می‌کنیم (دید برنامه‌نویسی).

تناسبی بین تعداد خطوط برنامه و تعداد بایت‌ها داریم؟ (در هر متد، چند بایت برنامه نوشته‌ایم؟)

توی ۲۰۰۸ هر دستور ۳۲ بیتی است ← تعداد بایت‌ها = ۴×۳۲

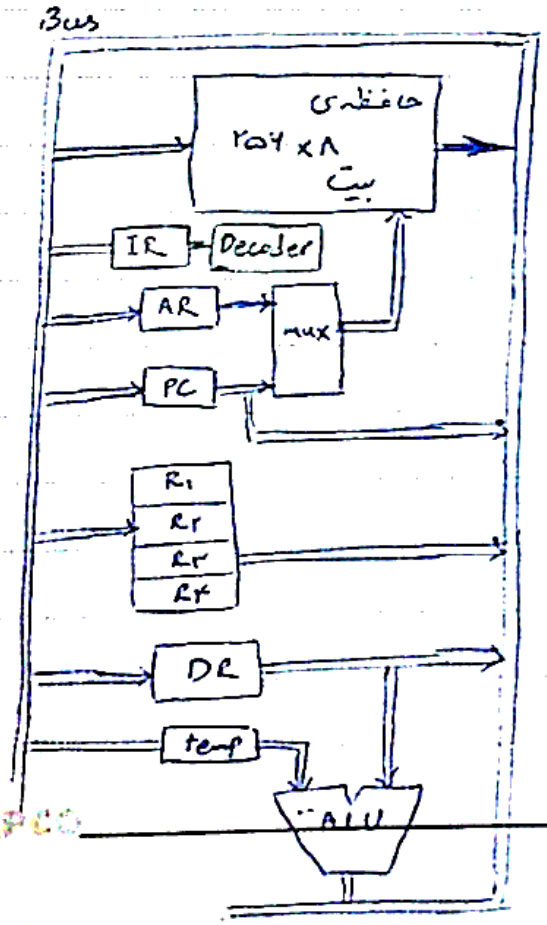
مدرس می‌خواهیم ضمن دستورات بیشتر، کوتاه‌تر باشد. تخمین می‌زنیم که با وجودی که متن بیشتر زیادتر است، تعداد بایت‌ها کمتر است.

زمان اجرا: مطابق الگوریتم نمی‌توان نشان داد.

ماشین بیشتری داریم که stack مراجعه می‌کند و می‌دانیم حافظه سرعت‌گیر پردازنده است ولی حافظه

ماشین ۲ مراجعه به حافظه اش کمتر است.

دستورات در پشته‌ای کوتاه‌تر هستند به جز pop و push طول دستورات پشته‌ای کمتر است و fetch کوتاه‌تر، کار با حافظه کمتر و سرعت بالاتر است.



۲- این شکل را چنان کامل کنید که Decode, Fetch قابل انجام باشند ← افزودن IR

سه دستور داده شده است. گزاره‌های RTL این دستورات را بنویسید:

Load relative R_1, C_1
Add Imm $R_2, R_1, 1$
And R_2, R_3

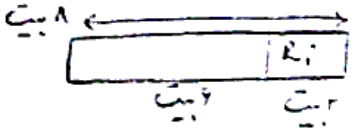
حافظه ۸ بیتی ← دستورات مضربی از ۸

دستور Load relative:

۴ ثابت قابل دسترسی داریم ← با ۲ بیت آدرس دهی می‌کنیم پس تا ۶ بیت برای OP داریم.

www.it98.ir

Subject:
Year. Month. Date. / /



load ← خواندن از حافظه ← Operand های مورد نیاز :

آدرس مقصد (یعنی R_1 ، باید در R_1 ریخته شود)

* از چه جایی باید بخوانیم ؟

load R_1, C_1

$R_1 \leftarrow M[C_1+PC]$ (C ثابتی است که بعنوان offset به PC اضافه می‌کنیم یعنی C_1 مقدار جابجایی است)

Add Imm: | Addi | R_i | R_j | / | imm-Operand |

And: | And | R_2 | R_3 |

fetch: T_0: $IR \leftarrow M[PC]$, $PC \leftarrow PC+1$;

T_1: decode (IR);

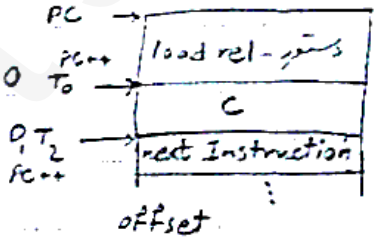
Load relative: D_1

D_1T_2: $DR \leftarrow M[PC]$, $PC \leftarrow PC+1$; offset را در DR می‌ریزد

(*) D_1T_3: $Temp \leftarrow PC$;

D_1T_4: $AR \leftarrow Temp + DR$, $SC \leftarrow 0$

D_1T_5: $R_1 \leftarrow M[AR]$, goto T_0;



(*) چون می‌خواهیم PC و DR را با هم جمع کنیم به ALU نیاز داریم و Temp یکی از ورودی های ALU است . پس اول PC را در Temp می‌ریزیم.

Add Imm: D_2

D_2T_2: $DR \leftarrow M[PC]$, $PC \leftarrow PC+1$;

(**) D_2T_3: $Temp \leftarrow R_1$; البته اینو نشان نداره

D_2T_4: $R_3 \leftarrow Temp + DR$, $SC \leftarrow 0$;

(**) چون مسیر مستقیم از R_1 به ALU نداریم ، مجبوریم اول بریزیم توی Temp.

And R_2, R_3 D_3 :

D_3T_2: $DR \leftarrow R_2$;

D_3T_3: $Temp \leftarrow R_3$;

D_3T_4: $R_2 \leftarrow DR \wedge Temp$, $SC \leftarrow 0$;

wWw.it98.ir

www.it98.ir

Subject:
Year: Month: Date:

جلسه‌ی سوم

ضرب کننده‌ها : ۱- ضرب کننده براساس جمع های متوالی : الگوریتم جمع و جابه‌جایی (add & shift)

multiplicand	376	$376 \times (214) = 376 \times (200 \times 10 \times 4)$
multiplier	* 214	
PP_0	1404	حاصلضرب‌های جزئی را به چپ
PP_1	3760	شیفت می‌دهیم.
PP_2	75200	
	80364	

روش ۱ دیدید مضروب را بنویسید

یک مثال باینری :

nbit	1011	A
mbit ×	101	B
	1011	
	0000	acc ← P_2
	1011	
	110111	: اندازه‌ی acc
		m+n bits

الگوریتم ضرب : روش جمع و جابه‌جایی (متوالی) :

```
ACC = 0
while (B ≠ 0)
{ if B0 = 1
      ACC = A + ACC
  B ← shr B
  A ← shl A
}
```

$$t_{x\,max} = m\,t_{add} + (m-1)\,t_{shift}$$

$$t_{av} = \frac{m}{2}\,t_{add} + (m-1)\,t_{shift}$$

این الگوریتم $O(n)$ است. چگونه می‌توان این الگوریتم را به $O(1)$ تبدیل کرد؟ اگر carry را از پیش حساب کنیم.

ضرب کننده‌ی آرایه‌ای : Array multiplication

چند مرحله این را رسم می‌کنیم (توسط دیدی لیت و ...) :

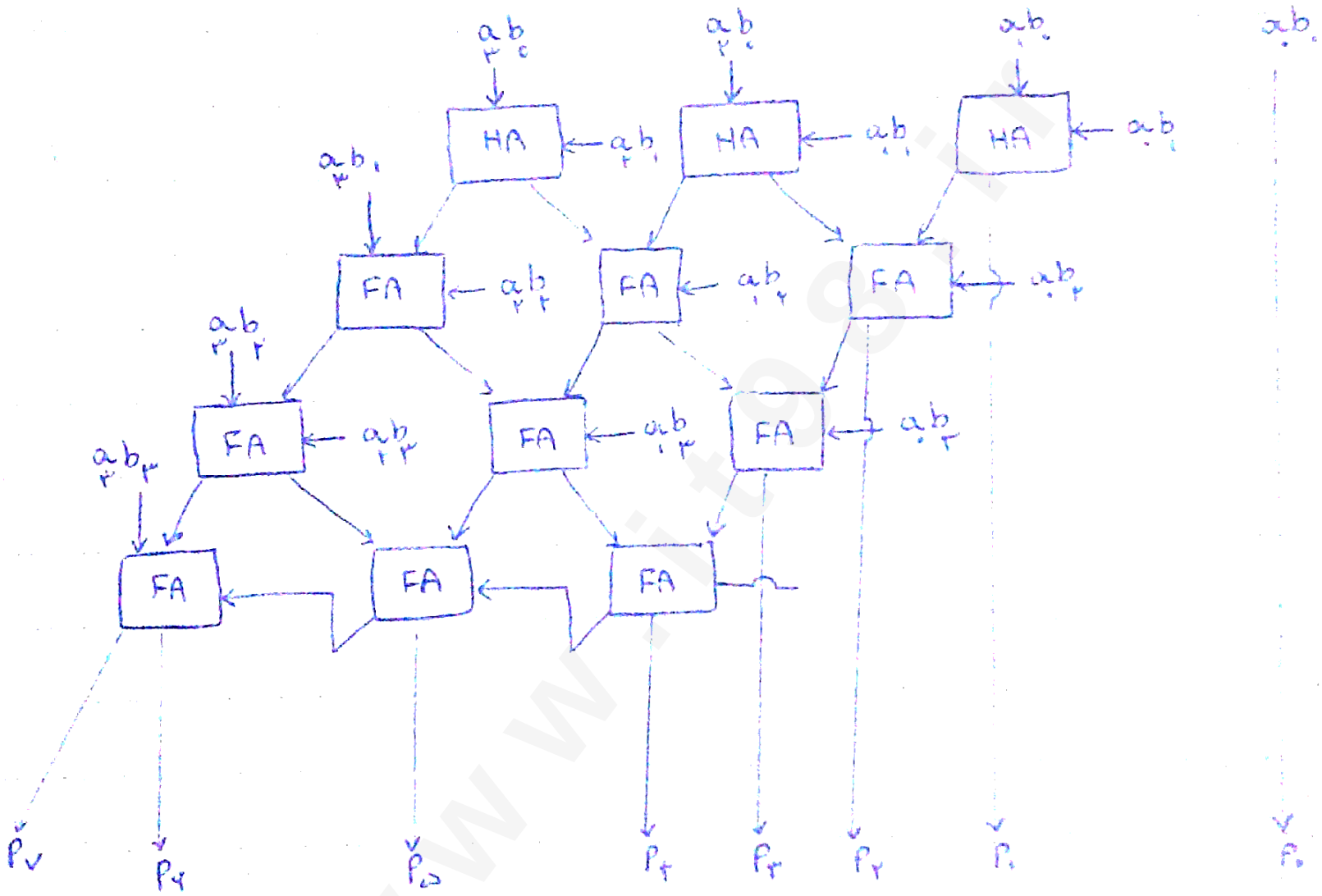
a_3	a_2	a_1	a_0	Partial Product
b_3	b_2	b_1	b_0	
a_3b_0	a_2b_0	a_1b_0	a_0b_0	PP_0
a_3b_1 a_2b_1	a_1b_1	a_0b_1		PP_1
		⋮		PP_2
				PP_3

WWW.it98.ir

www.it98.ir

Subject:

Year Month Date



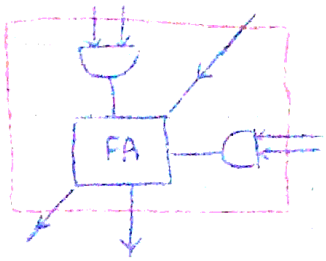
www.it98.ir

wWw.it98.ir

www.it98.ir

Subject:
Year: Month: Date:

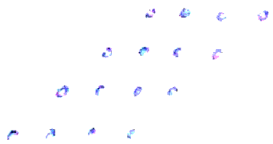
هسته‌ی اصلی (سلول):



این الگوریتم $O(1)$ است.

مدار 1Δ، $a_i b_j$ را حساب می‌کنیم. یعنی and مدار 1Δ محاسبه می‌شوند:

نمایش نقطه‌ای:



اینجا ۱۶ تا حاصلضرب حساب کرده‌ایم.

برای j سمت مرحله‌ی آخر می‌توانیم آن را propagate کنیم (در اینجا این کار را کرده‌ایم) یا اینکه CLA اعمال کنیم.

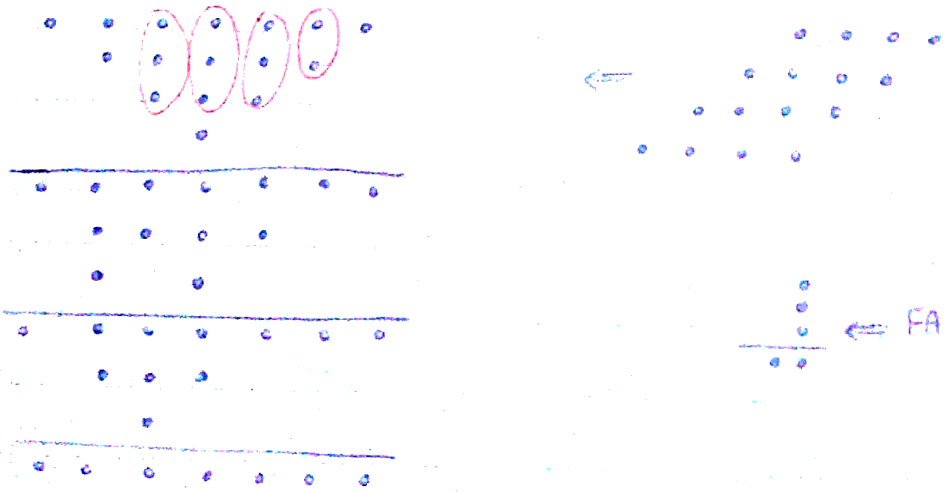
$$t_{arrayMul} = t_{and} + m t_{FA/HA} + (n-1) t_{FA} = t_{and} + (m+n-1) t_{FA}$$

چرا این الگوریتم از add و shift سریع‌تر عمل می‌کند؟

در الگوریتم add & shift: $t = m t_{add} + \underbrace{(m-1) t_{shift}}_{\simeq 0} \simeq m t_{add}$

جمع در دوتش بدون نقطه زمان عبور از FA را مبنا قرار می‌دهیم ولی در قبلی زمان adder در نظر گرفته می‌شود.

هر گوی معرف یک بیت است (می‌تواند ۰ یا یک باشد):



www.it98.ir

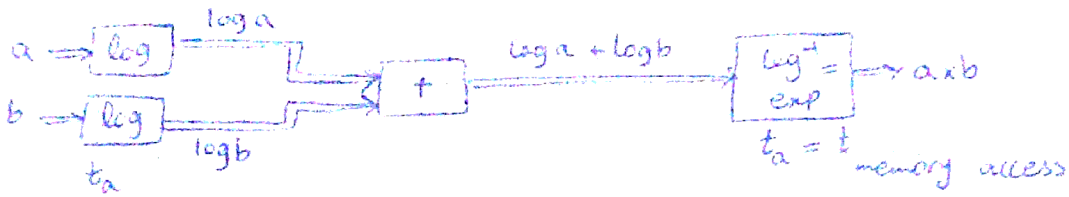
www.it98.ir

Subject:
Year Month Date

جدول یا حافظه log-exp :

(اعداد بدون علامت) $a,b>0$ $\log(a\cdot b) = \log a + \log b$ ⇐ می‌دانیم

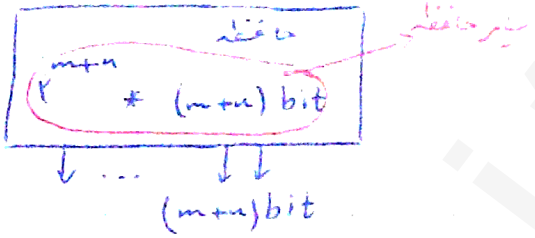
می‌توانیم از لگاریتم اعداد استفاده کنیم.



زمان اجرا = $2t_a + t_{add} = 3t_a$ ⇐ اگر برای جمع هم از حافظه استفاده کنیم : $t = 3t_a$

می‌توانیم جدول ضرب را در حافظه بریزیم و از روی آن ضرب را انجام دهیم :

مثال : a و b دو عدد هستند (به ترتیب n و m بیتی).



جدول ضرب یا جستجو
(look up table)

* قضیه‌ی مهم : هر مدار ترکیبی توسط حافظه قابل پیاده سازی است. یعنی می‌توان جدول صحت function را در حافظه قرار داد. پس هر مدار ترکیبی را می‌توان در $O(1)$ پیاده سازی کرد.

نکته: اگر جدول صحت مثال قبلی را در حافظه بریزیم، m+n بیت خروجی و 2^{m+n} بیت ورودی داریم. این سایز برای سایز بزرگ، بسیار زیاد می‌شود. نصف آن هم خالی است (چون a×b = b×a) (خالی اینجا یعنی بلااستفاده) و مقدار زیادی از آن هم صفر است.

موارد استفاده از این روش :

۱- ضرب‌های کوچک

۲- پیاده سازی توابعی که اجرای آنها برای حافظه زمان بر است مثل log و exp و ...

فرض می‌کنیم که اعداد حتماً binary باشند. می‌توان log آنها را ذخیره کرد. این کار چه مزیت چایی

www.it98.ir

wWw.it98.ir

www.it98.ir

دارد؟ با استفاده از این روش ضرب و تقسیم مدل = جمع و تفریق (= جمع) می‌شود و زمانش کوتاه‌تر می‌گردد.

در کل، الگوریتم add & shift الگوریتم کندی است و الگوریتم array multiplication هم پیچیدگی سخت‌افزاری دارد.

تقسیم: ضرب = جمع متوالی تفریق = sub & shift

$19/5 = ?$

$$\left.\begin{aligned} 19 &= 1\times 5 + 14 \\ 14 &= 1\times 5 + 9 \\ 9 &= 1\times 5 + 4 \\ 4 &= 0\times 5 + 4 \end{aligned}\right\} \quad 19 = 3\times 5 + 4$$

$$\underbrace{D}_{\text{Dividand}} = \overset{\text{quotient}}{Q} \times \underbrace{B}_{\text{Divison}} + \overset{\text{reminder}}{R}$$

* در تقسیم، باقی مانده هم علامت با مقسوم است. پس:

$$-19/5 = -3\times 5 - 4 \neq -4\times 5 + 1$$

طراحی سیستم تقسیم

$D = Q\times B + R$ $D = \boxed{P \mid U}$ $\boxed{B}$ $\boxed{R}$ $\boxed{Q}$

قضیه: اگر $P > B$ باشد سرریز خواهیم داشت یعنی خارج قسمت در n بیت نمی‌گنجد.

حالت خاص: اگر $B = 0$ پس قطعاً سرریز داریم چون هر $P > 0$ طبعاً از B بزرگتر خواهد بود.

$$\begin{array}{r|l} D\ 3782 & 24\ B \\ \hline \vdots & 157\ Q \\ \hline 14\ R & \end{array}$$

$37 > 24 \Rightarrow$ خارج قسمت ۳ رقمی

خارج قسمت ۵ رقمی $\Leftarrow$ اگر $B = 0$ $\Leftarrow$ تقسیم بر صفر

این تقسیم چه دردی می‌خورد؟ کافیست قبل از اینکه تقسیمی را در پردازنده شروع کنیم، چک کنیم که آیا نیمه‌ی بالای B پاک

توی سیستم بردار شماره‌ی صفر بیانگر تقسیم بر صفر است. بنابراین پردازنده چک می‌کند اگر برابر صفر بود که بلافاصله تشخیص می‌دهد والا $P > B$ را تشخیص می‌دهد و وقفه‌ی صفر را صادر می‌کند.

www.it98.ir

www.it98.ir

D از الحاق یا Concat P و U بدست می‌آید.

اثبات قضیه:

$D = 2^n \cdot P + U = Q \times B + R \qquad 0 \le R \le B$

$P \ge B \qquad 2^n B + U \le Q \times B + R \Rightarrow (2^n - Q) B + U \le R \le B \Rightarrow$

$\underbrace{(2^n - 1 - Q)}_{+} \underset{+}{B} + \underset{+}{U} \le 0 \quad \times$

پس Q حتماً از n بیت بزرگتر است.

$R_0 = D$ (در مرحله‌ی صفر، باقی‌مانده برابر خود D است) $\qquad Q_0 = 0$

$R_1 = R_0 - B \qquad Q_1 = Q_0 + 1$

if $(R_1 \ge 0)$

$R_2 = R_1 - B \qquad Q_2 = Q_1 + 1$

$\vdots$

$0 \le R_{n+1} = R_n - B < B$

الگوریتم restoring: الگوریتمی که وقتی باقی‌مانده از صفر کمتر شد آن را برمی‌گرداند و از خارج قسمت یکی کم می‌کند.

مثال:

```
1001 1100 | 1010
 1010     |------
-------   | 1111
 10011
  1010
-------
  10010
   1010
-------
   10000
    1010
-------
R = 0110
```

$1001 < 1010 \Rightarrow$ سرریز داریم

در این الگوریتم: n تغییر، n یا n-1 شیفت و بطور متوسط $\frac{n}{2}$ مرتبه restoring داریم.

این الگوریتم بسیار کند است.

```
10010010 | 1010
 1010    |------
-------  | 1110
 10000
  1010
-------
 001101
   1010
-------
 000110
```

www.it98.ir

wWw.it98.ir

www.it98.ir

Subject:
Year: Month Date

الگوریتم همگرایی :

x عدد هنجار شده است
$(0.5 \le x \le 1)$

$x = (0.1\ldots) \ge 0.5$ ، $A = \frac{1}{x}$

$y = 1 - x \Rightarrow 0 < y < 0.5$ $\quad A = \frac{1}{x} = \frac{1}{1-y} \overset{2}{=} \frac{1+y}{1-y^2} = \frac{(1+y)(1+y^2)}{(1+y^2)(1-y^2)} =$

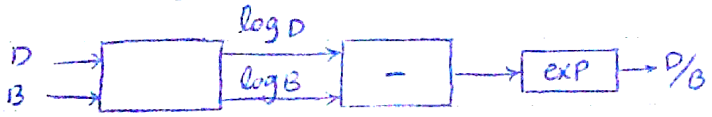
$$\frac{(1+y)(1+y^2)}{1-y^4} = \frac{(1+y)(1+y^2)(1+y^4)}{1-y^8} = (1+y)(1+y^2)(1+y^4)(1+y^8) \qquad (1-y^8 \cong 0)$$

در این الگوریتم وقتی ۱۶ بیت داریم باید ۴ مرحله طی کنیم (کافی است) .
اشکال این الگوریتم این است که باقی مانده ندارد و خودمان باید آن را حساب کنیم.

موارد استفاده از تقسیم : رمزنگاری ، بازار سهام ، پیدا کردن اعداد اول ، محاسبه‌ی توابع لگاریتمی و مثلثاتی
از روش بسط تیلور

روش سوم : $\log$, e^x

$$\log \frac{D}{B} = \log D - \log B$$



اعداد علامت دار :

جمع ← ساده‌ترین روش آن 2's Complement است .
ضرب ← اعداد علامت دار و مکمل ۲ فرض شده اند.
این روش سختی اشتباه است چون این ۱ ها ، علامت هستند و
رقم نیستند (علامت دار هستند عددها)

1011 -5
1111 × -1
1011
1011
1001
0000
روش سختی
جواب

اولین راه حل : مضروب فیه را مثبت کنیم .
چرا فقط مضروب فیه ؟ چون داریم جمع می‌کنیم .

روش دوم : استفاده از کد گذاری جدید (Booth) .

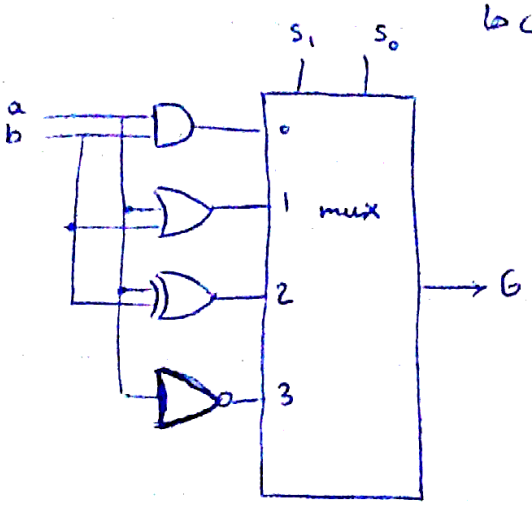
www.it98.ir

w.it98.ir

جلسه‌ی بیست و یکم : واحد ریاضی منطقی (ALU)

فرض : جمع کننده داریم . انجام عملیات منطقی روی تک بیت ها

به کمک mux :



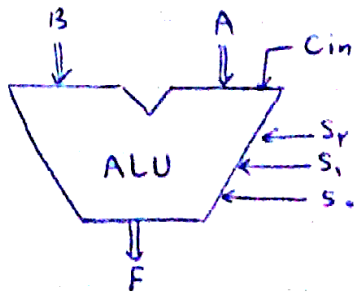
دو بیت فرمان

S_1	S_0	G
0	0	$a \wedge b$
0	1	$a \vee b$
1	0	$a \oplus b$
1	1	$\bar{a}$

مثال از کتاب مانو : طراحی واحد ریاضی منطقی :

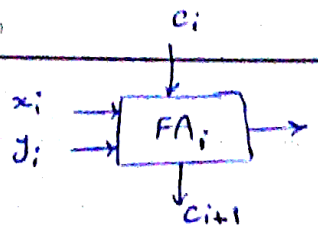
یک ALU طراحی کنید که جدول عملکردش بدین گونه است.

S_2	S_1	S_0	C_{in}	Operation		
0	0	0	0	A	transfer	انتقال صرف
0	0	0	1	A+1	increment	A و B ، n بیتی هستند پس مدار را
0	0	1	0	A+B	add	برای n بیت طراحی می کنیم.
0	0	1	1	A+B+1	add with carry	
0	1	0	0	$A+\bar{B}$		
0	1	0	1	A−B → $A+\bar{B}+1$	Sub	عملیات حسابی
0	1	1	0	A−1	decrement	
0	1	1	1	A	transfer	
1	0	0	∅	$A \wedge B$	And	
1	0	1	∅	$A \vee B$	or	عملیات منطقی
1	1	0	∅	$A \oplus B$	xor	
1	1	1	∅	$\bar{A}$	not	



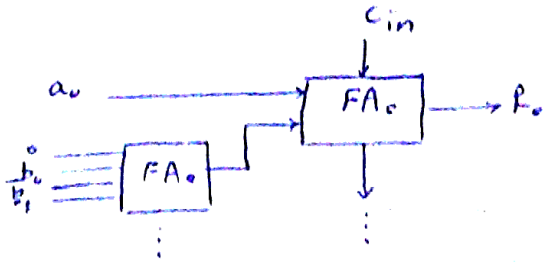
اینها معادل دستورات اسمبلی پردازنده هاست . در پردازنده ها هم چنین دستوراتی داریم ، منتها در پردازنده دستورات opcode دارند که به ALU و سایر قسمت ها عرضه می شود و از روی آن می فهمند چه دستوری است.

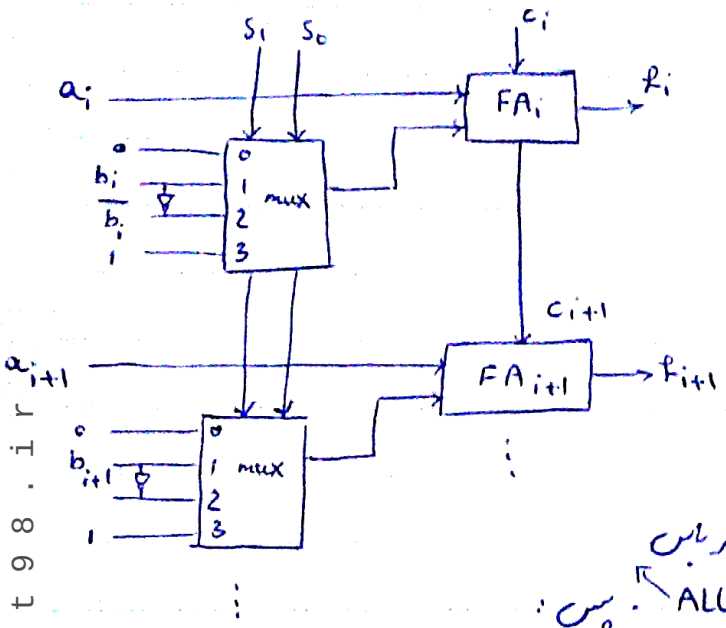
فرض می‌کنیم یک FA داریم.



هر وقت c_{in} داریم، مثل این است که یک ۱ به حاصلجمع افزوده می‌شود.

مرتبه‌ی صفرم:





یکی از راه‌های انتقال داده از یک مبدأ به یک مقصد اگر باس مشترک نداشته باشیم، جمع با صفر است. مثل این ALU. پس:
وقتی باس مشترک نداریم ← جمع با صفر

چرا برای حالت A-1، ورودی ۳ را برای mux فعال کرده‌ایم؟

$1 = (000\ldots001)_2$ $\quad -1 = (C_2(1)) = 111\ldots1$ (2's comp)

اگر c_{in}، صفر باشد ← A-1

A = A-1+1 ← 1

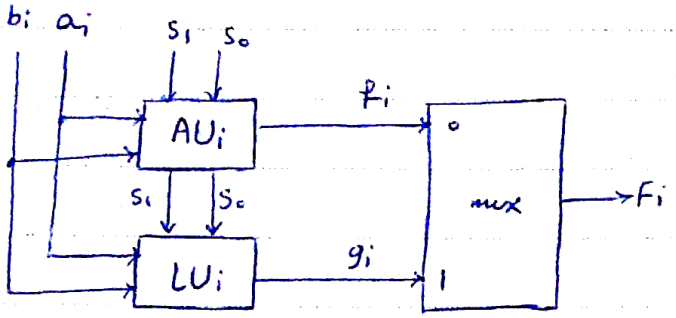
چرا ۲ تا transfer داریم؟ فرق این دو تا transfer در چیست؟

در اولی c_{out} نداریم ولی توی دومی c_{out} برابر ۱ است. لذا دومی از نوع جابجایی‌هایی است که carry را تحت تأثیر قرار می‌دهد. توی بعضی پردازنده‌ها برای دستور move، نوشته که carry flag را تحت تأثیر قرار می‌دهد. هر وقت چنین چیزی دیدیم یعنی move با جمع کننده پیاده‌سازی شده است وگرنه move از طریق bus به carry کاری ندارد.

www.it98.ir

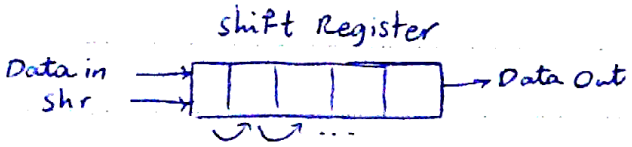
www.it98.ir

قسمت Logic را اول جلسه طراحی کردیم. قسمت قبل Arithmatic Unit بود.



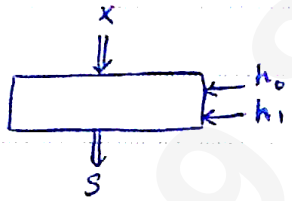
افزودن شیفت: یک بیت جابجا شونده

تفاوت شیفتر و شیفت رجیستر:



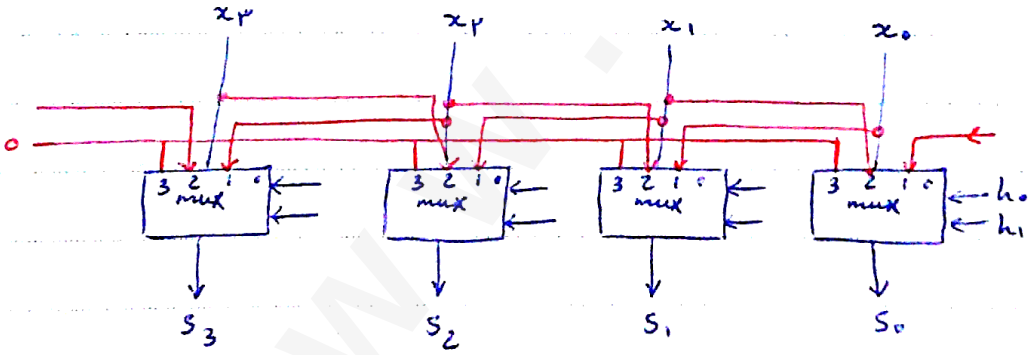
جابجا کننده

h_1	h_0	S
0	0	F
0	1	shl F
1	0	shr F
1	1	0



shifter یا جابجا کننده: یک یا چند بیت ورودی دارد.

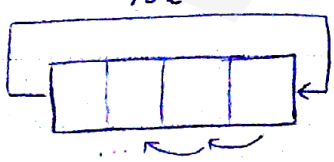
مداری ترکیبی طراحی کنید که به ازای ورودی کنترلی h_0 و h_1 بتواند یا x را بدون تغییر به خروجی منتقل کند؛ یا جابجا به راست، جابجا به چپ تا تبدیل به صفر نماید (به این مدار، شیفتر یا جابجا کننده می گویند).



برای rotate: مثلاً rol(10010) = 00101

بعضی پردازنده ها اجازه ی rotate با Carry می دهند مثل 68000 که

دو نوع rotate دارد: rol و rolwitc.



www.it98.ir

wWw.it98.ir

www.it98.ir

lshr یا lshl ، داده ی ورودی از چپ یا راست ، صفر است : Serial Input = 0

هر وقت شیفت حسابی داشتیم ⇐ عدد علامتدار است.

وقتی شیفت می دهیم باید علامت را حفظ کنیم.

وقتی logical shift داریم داده ها را همانگونه که هست جابجا می کنیم و ورودی صفر تزریق می کنیم.

right: 11101 (−3) ⟶ 11110 (−2)

Arithmetic ← Ashr(x) = $\lfloor \frac{x}{2} \rfloor$

left: 11010 (−6) ⟶ 10100 (−12)

قوی Ashl ، چون ضربدر ۲ می کنیم حداکثر تا اعدادی که نصف بازه ی نمایش هستند را می توانیم Ashl انجام بدهیم و انتظار جواب صحیح (عدد دو برابر شده) را داشته باشیم. مثلاً با ۵ بیت از ۱۶- تا ۱۵+ قابل نمایش است پس ۲×۶- را می تواند نشان دهد.

نمایش اعداد ممیز شناور Floating Point representation

دو نوع نمایش داریم : ۱- نمایش ممیز ثابت ⇐ محل ممیز همیشه ثابت (تعداد ارقام اعشاری ثابت)

۲- " " شناور ⇐ می توان مکان ممیز را عوض کرد (دقت متغیر است).

وقتی ممیز ثابت باشد دقت هم ثابت است، یعنی گاهی اوقات دقت کم و گاهی اوقات زیاد است.

تعریف رسمی نمایش ممیز شناور: $N = (-1)^S \, 1.F \times r^E$

S: علامت S=1 ⇐ عدد منفی و S=0 ⇐ عدد مثبت

F: Fraction ⇐ قسمت اعشاری

r: root یا ریشه

E: نمای جابجا شده (biased)

Single Precision : ۳۲ بیتی 1 | 8 | 23 (S | E | F) bios: $2^7 - 1 = 127$

Double precision : ۶۴ بیتی 1 | 11 | 52 (S | E | F) bios: $2^{10} - 1 = 1023$

www.it98.ir

www.it98.ir

$-1.000\ldots10 \times 2^{127-bias} =$ ← | S | E | F |

$-1.000\ldots10 \times 2^{-102}$

چرا E را biased کرده اند؟

تا بتوان ها مرتب شوند و قابل مقایسه کردن باشند سادگی.

واحدی بیت دوم

* دستور العمل Sub R_1, R_2, R_3 را کدگذاری نمایید (op را پیدا کنید).

کدگذاری نمایید یعنی بی ؟ یعنی یک opcode به من بدهید که اگر opcode را به پردازنده بدهیم، این دستور اجرا شود. یعنی ثبات های مبدأ و مقصد و عمل را می شناسیم.

این پردازنده سه آدرسه است یعنی همزمان می توان دو تا operand را خواند و روی آن عملی انجام داد و در مقصد نوشت

فرمت قالب بندی دستورات

OPA	OPB	Dest	operation
R_2	R_3	R_1	Sub

Control word : 0110 0011 0001 0101₂

hexadecimal : 6 3 1 5

R_2	-	R_3	shl
100		110	11000

* $R_2 \leftarrow shl\ R_3$

همین طور تا این است که می توانیم B و A را وارد فیلد ALU بکنیم و پس از آن shift&rot

بعد بعد ... کنیم

شرکت intel با دستورات پیچیده روبرو شدیم، متوالیا اضافه شدند تا این کلمات کنترل پیچیده تر شدند که پردازنده اضافی را تقسیم می کرد. منطقا اینها مربوط به واحد کنترل دستورات پیچیده می کنیم یعنی این جای CPU است.

CPU ← fetch و decode و execute را مربوط و عملیاتی انجام می دهد. اما مربوط به واحد کنترل و ... یعنی می بینیم که CPU است.

www.it98.ir

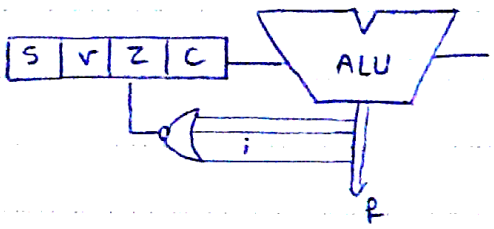
wWw.it98.ir

input	Ø	0	TSFA

output ← input

transfer می‌تواند به کمک جمع یا صفر پیاده‌سازی شود.

وقتی همه‌ی بیت‌های خروجی صفر باشند ⇐ فلگ zero ، set می‌شود.



$Z = \overline{f_0 + f_1 + \dots + f_{n-1}}$

$S = f_{n-1}$

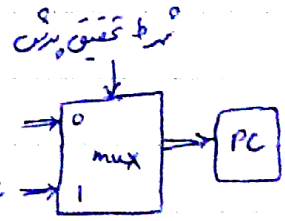
$V = C_{n-1} \oplus C_n$ ترین اختیاری

برای تحقق پرش، اول باید مقدار Carry ها معین شود و براساس آنها تصمیم که پرش صورت می‌گیرد یا خیر. بنابراین ALU باید این flag ها را set کند. بعد از آن CU براساس آنها یک سری selector باز یا بسته می‌کند. مثلاً براساس C، V، Z و S یک سری شروط تعیین می‌شود که مشخص می‌کند pc+1 یا pc+offset در programCounter قرار بگیرد.

JZ , BZ , BE	Z = 1
BNZ , BNE	Z = 0
BC (branch Carry)	C = 1
BNC (branch not Carry)	C = 0
BP (branch plus) +	S = 0
BM , BN (branch minus) −	S = 1
BV , BOV	V = 0
BNV	V = 1

if(شرط = 1) newpc → pc

else if(شرط = 0) pc+1 → pc



مقایسه‌ی اعداد بدون علامت

$A - B = \text{Comp } A, B$

$A - B = A + C_2(B) =$

$A + 2^n - B = \underset{\text{Carry}}{2^n} + \underbrace{(A - B)}_{\geq 0}$

پس فلگ Carry ، set می‌شود.

وقتی اعداد بدون علامت هستند، باز هم برای تفریق از 2's Comp استفاده می‌کنیم.

branch if higher	BHI	$A > B$	$C = 1 \wedge Z = 0$
	BHE	$A \geq B$	$C = 1$
	BLO	$A < B$	$C = 0$
	BLOE	$A \leq B$	$C = 0 \vee Z = 1$
	BE	$A = B$	$Z = 1$

www.it98.ir

www.it98.ir

بنابراین به این ترتیب، شروط پرش را به Control Unit می‌دهیم و Control Unit می‌تواند بوسیله‌ی یک مدار منطقی muxهایی را که PC را update می‌کنند انتخاب کند.

مقایسه‌ی اعداد علامت‌دار: CMP A,B → A−B

A و B، n بیتی و علامت‌دار هستند.

پس $-2^{n-1} \leq A \leq 2^{n-1}-1$

$-2^{n-1} \leq B \leq 2^{n-1}-1$

+7	0111
+6	0110
⋮	⋮
0	0000
-1	1111
⋮	⋮
-7	1001
-8	1000

وقتی دو عدد هم علامتند، A−B در این فاصله می‌گنجد و OverFlow نداریم.
ولی اگر مختلف العلامت باشند ممکن است رخ دهد.

$$\begin{array}{lcccccc} S_A & a_{n-1} & a_{n-2} & \cdots & a_1 & a_0 \\ -\ S_B & b_{n-1} & b_{n-2} & \cdots & b_1 & b_0 \\ \hline S_F & f_{n-1} & f_{n-2} & \cdots & f_1 & f_0 \end{array}$$

مقایسه‌ی اعداد با علامت $A-B = \text{Comp } A,B$

(greater)	BGT	$A>B$	$\bar{Z}\cdot S \odot V = 1$
	BGE	$A \geq B$	$\bar{S}_F \bar{V} + S_F V = 1$
(less than)	BLT	$A<B$	$S_F \oplus V = 1$
	BLE	$A \leq B$	$Z \vee (S_F \oplus V) = 0$
	BE	$A=B$	$Z=1$

ضرب اعداد علامت‌دار:

روش اول: multiplier را مثبت کنید و بعداً علامت حاصل ضرب را اصلاح کنید.

روش دوم: از کد جدیدی بهره جویید (به گونه‌ای جدید به مسئله نگاه کنید).

قضیه:

$$N = 000\underset{2^K}{1}111\cdots\underset{2^{i+1}}{1}\underset{2^i}{1}000\cdots0 \Leftrightarrow 0000\underset{2^{K+1}}{1}00\cdots\underset{2^i}{\bar{1}}00$$

$$N = 2^{K+1} - 2^i$$

www.it98.ir

$$N = 2^i + 2^{i+1} + \dots + 2^K$$
$$2N = 2^{i+1} + 2^{i+2} + \dots + 2^K + 2^{K+1} \Rightarrow N = 2^{K+1} - 2^i$$

$N = 00\ 11\ 00\ 11\ 11\ 10 = 2^{11} - 2^8 + 2^6 - 2^1 = (0\,1\,0\,0\,\bar{1}\,0\,1\,0\,0\,0\,0\,\bar{1}\,0)$ Booth Code

$-2 = \overset{2^3\,2^2\,2^1\,2^0}{1110}$

a_i	a_{i-1}	b_i
0	0	0
0	1	+1
1	0	-1
1	1	0

این کد گذاری چگونه عمل می‌کند؟
بیت سمت راست بیت صفر را صفر و بیت سمت چپ آخرین بیت را تغییر بیت علامت قرار می‌دهند.

مثال هایی از کدگذاری booth :

$v = +23 = \underbrace{0\,1\,0\,1\,1\,1}\,0 \rightarrow (1\,\bar{1}\,1\,0\,0\,\bar{1})$
$+1\ -1\ +1\ 0\ 0\ -1$
$2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

همیشه کد خود را چک کنید $\Rightarrow 23 \overset{?}{=} 2^5 - 2^4 + 2^3 - 2^0 = 23$ ✓

$K = -26 = \underbrace{1\,0\,0\,1\,1\,0}\,0 \rightarrow (\bar{1}\,0\,1\,0\,\bar{1}\,0)$
$-1\ 0\ +1\ 0\ -1\ 0$

$-26 \overset{?}{=} -32 + 8 - 2 = -26$ ✓

هم برای اعداد مثبت مناسب است و هم برای اعداد منفی.

$9 = (1\bar{1})_{10} = 10^1 - 10^0$

$8 = 1\bar{2} =$

$-8 = \bar{1}2$

اگر اعدادمان در این بازه باشند $\{-6, -5, -4, \dots, 5, 6\}$

در این سیستم انتشار رقم نقلی نداریم.

www.it98.ir

جلسه‌ی بیست و سوم

اعداد هنجار شده (normalized): مثلاً $0/00764 \Rightarrow 0/764\ldots \times E-2$ یا 10^{-2}

به این ترتیب می‌توان صفرها را از بین برد چرا که اطلاعات چندانی ندارد و ارقام معنی دار دیگری را نوشت. هر چیزی که تکراری و قابل پیش بینی است، فاقد اطلاعات است.

در تئوری اطلاعات $I(e_i) \propto \log \frac{1}{p(e_i)}$

مقدار اطلاعات — احتمال وقوع

اولین رقم سمت چپ در اعداد هنجار شده مخالف صفر است یعنی معنی دار است (در نمایش دودویی ۱ است و در دهدهی ۹ و ... و ۱ است).

در استاندارد IEEE 754، اعداد هنجار شده هستند.

اما در طبیعت همه چیز هنجار شده نیست. مثلاً ⇐

$$1/1010 \times 2^0$$
$$1/0110 \times 2^0$$
$$0/0100 \times 2^0 = 1/00\ldots \times 2^{-2}$$

برای ذخیره‌ی اینها در کامپیوتر باید عدد را جابجا کرد تا به شکل هنجار شده در بیاید.

آیا همیشه امکان هنجار سازی هست؟

در عدد صفر هم E و هم F صفر هستند ⇐

$$1\ 000\ldots000$$
$$0\ 000\ldots000$$

اگر همه‌ی بیت های E و F صفر بودند می‌فهمیم که عدد صفر است.

غیر از صفر:

اگر مثلاً 10^{-39} کوچکترین مبنای قابل نمایش باشد و یک عدد اعشاری در آن ضرب شود دیگر نمی‌توان آن را هنجار شده کرد. مثلاً $0/01 \times 10^{-39}$

به این اعداد denormal می‌گوییم.

در سیستم اعداد ممیز شناور ۳ تا سوال پیش می‌آید؟

۱- بازه‌ی نمایش

۲- دقت نمایش (مطلق - نسبی)

۳- تعداد اعداد قابل نمایش

www.it98.ir

www.it98.ir

(1/F) ← فاصله (M)max = 1/Fmax = (1/111...111)₂ (f بیت) = 2 − 2^(−f) (ULP)

قسمت معنی دار عدد

unit of lower precission

ulp = r^(−f) 9/999999 = 2 − r^(−f) ⇒ r^(−f) = 10^(−6)

فاصله‌ی دو عدد متوالی، قدرت تمیز سیستم نمایش

کوچکترین مقداری بین دو عدد اعشاری وجود دارد. در اعداد صحیح، ulp = 1 است.

خطای مطلق = ulp/2 ⟸ مقدار بدست آمده = عدد گزارش شده ± ulp/2

$M_{min} = 1/F_{min} = 1/000\ldots01$

	c_2	c_{2+bias}	
+7	0111	1110	→ $2^{e-1}-1$
+6	0110	1101	
+5	⋮	⋮	
⋮			
+1	0001	1000	
0	0000	0111	
−1	1111	⋮	
⋮	⋮		
−6	1010	0001	→ $-2^{e-1}+2$
−7	1001	0000	→ reserved for zero and denormal representation
−8	1000	1111	→ اعداد ناعددی (Not number)

e = 4 ⇒ +7

e = 4 ⇒ −6

که بایستی اعداد با ترتیب اعداد صحیح پیدا می‌کند
بنابراین نمایش و مقایسه‌ی آنها راحت‌تر می‌شود.
مقایسه‌ی تفاضل آنها هم راحت‌تر می‌شود.
(دلایل بکارگیری bios)

$E_{max} = 2^{e-1}-1 \xrightarrow{\text{در این مثال}} E_{min} = -6$

$E_{min} = -2^{e-1}+2 \qquad E_{max} = +7$

bios صرفاً برای encoding بکار برده می‌شود و صرفاً ظاهر که مناسب مقایسه باشد
والا مقدار جبری همان مقدار قبلی است.

پس داریم: $N_{max} = +M_{max} \times r^{E_{max}} = (1/111\ldots1) \times 2^{111\ldots10} =$

$(2-2^{-f}) \times 2^{2^{e-1}-1} = (2-2^{-23})\, 2^{127} = 2^{128} - 2^{124} \approx 10^{39}$

بزرگترین عدد قابل نمایش در Single Precission

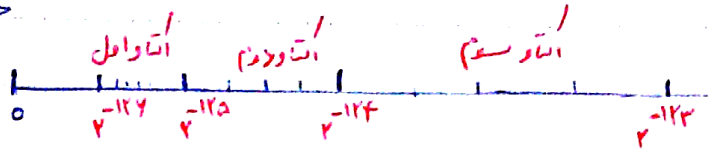
www.it98.ir

$$0 < N_{min} = +M_{min} \times 2^{E_{min}} = 1 \times 2^{-2^{e-1}+2} = 2^{-126} \simeq 10^{-38}$$

$$2^{a} \times 10^{x} \Rightarrow x = a \times \log_{10} 2 \Rightarrow x \simeq 0/38$$

(داخل پرانتز :)

هر اکتاو یک سری عدد با فواصل برابر نسبت به هم دارد، سپس وارد اکتاو بعدی که شدیم فواصل ۲ برابر می شود



چرا تعداد اعداد در داخل هر اکتاو ثابت ولی از هر اکتاو به اکتاو بعدی فواصل ۲ برابر فواصل کنونی می شود؟

در هر اکتاو 2^{f} عدد داریم و کلا $2^{e}-2$ تا اکتاو داریم. یک اکتاو به خاطر ۰ و یک اکتاو به خاطر اعداد نامعلومی حذف می شود.

پس سوال اول را پاسخ داریم (یعنی بازه ی نمایش)

فاصله ی دو عدد متوالی :

$$\Delta = 2^{-f} \times 2^{E_{min}} = 2^{-f} \times 2^{-2^{e-1}+2} = 2^{-2^{e-1}-f+2} = 2^{-126-23} = 2^{-149}$$

← کوچکترین فاصله ای بین دو عدد که در کوچکترین اکتاو اتفاق می افتد

خطای مطلق برای کوچکترین اکتاو ← اعداد کوچک : $2^{-150} \simeq 10^{-50} = \frac{\Delta}{2}$

دقت در اینجا بالا است چون اعداد کوچک را با دقت بالا می سنجد

خطای مطلق برای بزرگترین اکتاو ← اعداد بزرگ :

$$2^{-f} \times 2^{E_{max}} = 2^{-f} \times 2^{2^{e-1}-1} = 2^{2^{e-1}-f-1} = 2^{128-23-1} = 2^{104} \simeq 10^{30}$$

در اینجا خطای مرتکب شده برای تقریب بزرگ است چون خود اعداد هم بزرگند.

$$2^{-f} = \frac{\Delta}{2^{E_{min}}} = 10^{-7} \simeq 2^{-23}$$

f خطای نسبی را تعیین می کند.

www.it98.ir

Subject:
Year. Month. Date. ( )

۶۷

آخرین سوال:

$$N = 2(2^f \times (2^e - 2) + 2) = 2^{f+e+1} - 2^{f+2} + 2$$

تعداد صفر

تعداد اعداد

چون اعداد + و - داریم

تعداد مانتیس

* اگر اعداد به شکل $N = (-1)^s \times 0.F \times 2^E$ باشند، سه سوال را پاسخ دهید:

این سه سوال را برای نمایش IEEE دقت مضاعف حل کنید.

مثال: $N = (-1)^s (0.F)\, 16^E$

S	3bit	12 bit

$$M_{max} = 0.F_{max} = (0.111\ldots111)_2 = 0.FFF_{hex}$$

$$M_{min} = 0.1 \qquad E_{max} = 110 = +3 = 2^{0-1} - 1 \qquad E_{min} = \begin{matrix}110\\000\\111\end{matrix} = -2^2 + 2 = -2 = -2^{2-1} + 2$$

$$N_{max} = (0.FFF) \times 16^{E_{max}} = (1 - 16^{-5}) \times 16^3$$

$$0 < N_{min} = 1 \times 16^{-2}$$

جواب سوال اول ⇐

سوال دوم و سوم را خودتان پاسخ دهید.

نمای ویژه

S	F: 1111	

0 111 ... 1111 ⇒ $+\infty$

1 111 ... 1111 ⇒ $-\infty$

اعداد ناعددی ⇐ F در این اعداد ۱ است مثل ∞ و $\sqrt{-1}$ و ... .

جلسه‌ی بیست و چهارم: سلسله مراتب حافظه:

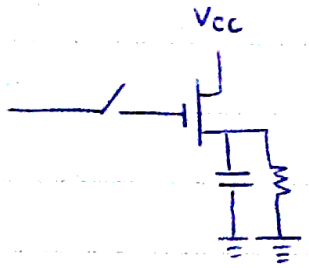
ظرفیت		حافظه‌ی نهان
α GB ~ TB	tape	10 s
α 6GB	DVD	5
α 700 MB	CD	0/1 s
α 100 GB → α 10 TB	Disk	50 ns
α GB → α TB	flash	10 ns
1 MB → 100 GB	DRAM	α ns
4K → 10 MB	SRAM	
1-128 bits	FF	100 ps

www.it98.ir

www.it98.ir

یکی از مسائل طراحی و تصمیمی انجام مصالحه بین حجم حافظه و سرعت آن است.
تکنیک SRAM بر پایه‌ی FF است و هر FF از ۶ تا ترانزیستور تشکیل شده است بنابراین این تکنیک سریع‌تر است. DRAM بر اساس خازن کار می‌کند و خازن نوعی حافظه است.

یک سلول DRAM



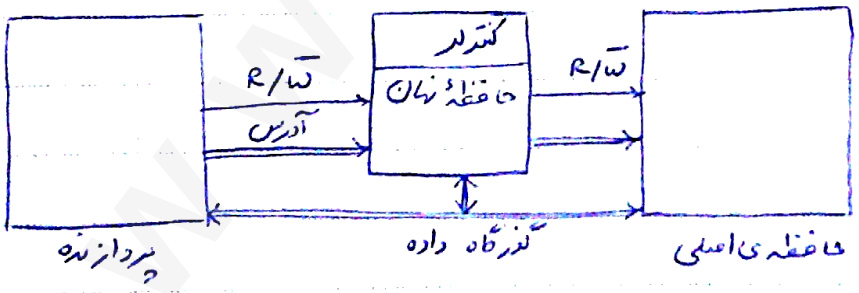
وقتی کلید را بزنیم ترانزیستور هادی جریان می‌شود سپس ولتاژی که به ترانزیستور وصل کرده‌ایم جریان پیدا می‌کند و از مقاومت می‌گذرد. ترانزیستورها دارای یک خازن پارازیت می‌باشند و وقتی ترانزیستور هادی جریان شد شارژ می‌شود (مثل شارژر لپ تاپ).
DRAM ها تشکیل دهنده‌ی حافظه‌ی کامپیوترند (RAM های PC).

مشکل DRAM:
بعد از مدتی DRAM دشارژ می‌شود. برای حل این مشکل هر از چند گاهی فقط مراغ ترانزیستور می‌روند و شارژ به آن می‌دهند و بعد از چند میکروثانیه روشن نگهش می‌دارند. اسم این عمل (سرکشی مجدد به حافظه) refreshing است.

memory refresh = احیا یا تازه سازی حافظه

حافظه‌ی Flash کندتر است و تعداد write در آن محدود است ولی با قطع جریان برق، اطلاعات آن از بین نمی‌رود.

حافظه‌ی پنهان یا cache:
از دید پردازنده پنهان است. بین پردازنده و M.M است و تکنولوژی آن SRAM است.



پردازنده وقتی می‌خواهد مراغ داده‌ها یا دستورالعمل‌های خود برود توی سیکل خواندن آدرس را به باس آدرس عرضه می‌کند و منتظر می‌شود که حافظه آن داده یا دستور را در اختیارش بگذارد. در سیکل write دستور write را فعال می‌کند و مقداری که باید نوشته شود را روی باس Data قرار می‌دهد بعد صبر می‌کند تا حافظه داده را در خود ذخیره کند یعنی پردازنده اصلاً از cache خبر ندارد.

www.it98.ir

www.it98.ir

حافظه‌ی نهان هرگاه آدرسی را مشاهده کرد که قبلاً به این آدرس مراجعه شده باشد داده‌ای که به این آدرس بوده را قبلاً کپی گرفته و در خود ذخیره کرده است و از این کپی استفاده می‌کند.

نرخ برخورد (نرخ بود): h: hit ratio ⇐ با احتمال h ممکن است اطلاعات قبلی را در cache بیابیم.

نرخ فقدان (نرخ نبود): $m = 1 - h = \text{miss ratio}$

زمان متوسط دسترسی به سیستم حافظه ⇐ $t_{av} = h\, t_{cache} + (1-h)\, t_m$

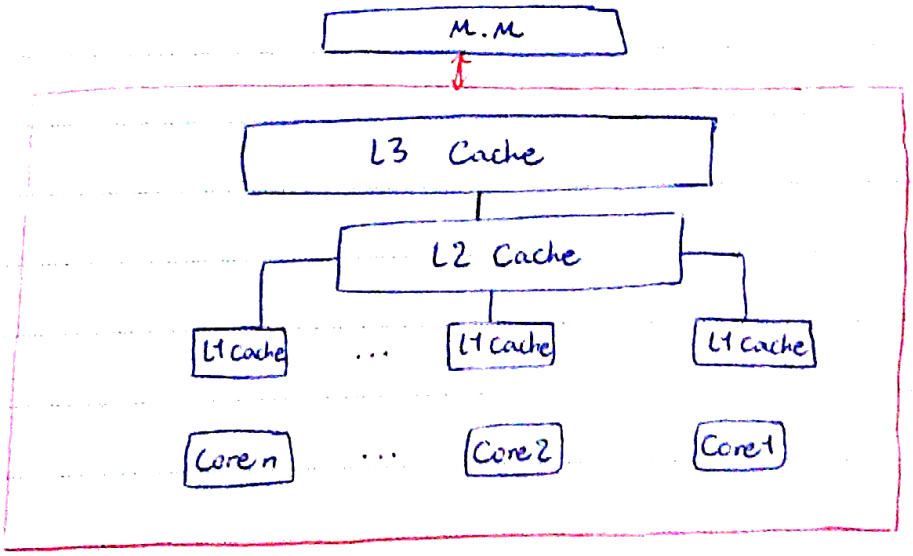
مثال عددی:

$h = 95\%$ ، $t_{cache} = 1ns$ ، $t_m = 10ns$

$$t_{av} = \frac{95}{100} \times 1 + \frac{5}{100} \times 10 = 1.45\ ns$$

$$\text{تسریع} = \frac{\text{زمانی که cache نداریم}}{\text{زمانی که cache داریم}} = \frac{10}{1.45} \simeq 7.8$$

اگر Disk را هم در نظر قرار بدهیم: $t_{av} = h_c\, t_{cache} + (1-h_c)\, h_m\, t_m + (1-h_c)(1-h_m)\, t_D$



ما هیچ وقت نمی‌گوییم فلان داده را در cache بنویس چون cache از دید پردازنده پنهان است. این کنترلر cache است که تشخیص می‌دهد کدام داده کجا نوشته شود.

هرچه cache بیشتر باشد اطلاعات سریع‌تر انتقال می‌یابد بنابراین چند لایه cache داریم چون

www.it98.ir

www.it98.ir

بعضی اطلاعات بیشتر مورد استفاده قرار می‌گیرد و باید دم دست‌تر باشد و بعضی کمتر و ... .

t_{MM} معمولاً زمان یک block است نه byte .

cache براساس اصل زیر ساخته شده است :

اصل مجاورت مراجعات یا LOR (Principle of Locality of Refrence)

«اگر شما (پردازنده) سراغ خانه‌ای رفته باشید، به احتمال زیاد در آینده ی نزدیک به همان خانه یا مجاورین آن خانه رجوع خواهید کرد.»

* مجاورت مکانی (Spatial LOR) : اگر سراغ آدرسی رفتید به احتمال زیاد در دستور بعدی به همان آدرس یا مجاور آن خواهید رفت. مثلاً دسترسی به آرایه‌ها و اجرای متوالی دستورات

* مجاورت زمانی (Temporal LOR) : اگر الان از یک نقطه‌ای گذشتیم پس از گذشت زمانی کوتاه دوباره از آن نقطه خواهیم گذشت. مثلاً حلقه‌ها و تکرارها

* مجاورت دنباله‌ای (Trace LOR) : اگر دنباله‌ای از دستورات را اجرا کردیم در آینده نیز آن دنباله را اجرا خواهیم نمود. مثلاً linked list

در پردازنده‌ی پنتیوم به جای اینکه Instruction cache از جنس یک حافظه‌ی متوالی باشد، Trace cache داریم. Intel هیچ وقت سایز Trace cache خود را آشکار نکرده که این جزو اسرار تجاری است.

روش های جایابی و جایگزینی در cache :

اولاً هر خواندن از حافظه ی اصلی یا نوشتن در آن به صورت بلوکی انجام می‌شود یعنی وقتی سراغ M.M رفتیم یک بلوک را جابجا می‌کنیم (نه فقط یک بایت یا دو بایت را) ⇐ اصل مجاورت مکانی

هر بلوک معمولاً ۸ تا ۳۲ بایت دارد.

بنابراین دانه بندی مراجعه به حافظه بلوک است و هر وقت به حافظه مراجعه کنیم به جای خواندن ۱ بایت یا ۱ word ، ۸ تا بایت می‌خوانیم.

نگاشت ها :

حافظه‌ی نهان ۳ نوع است :

۱- نگاشت مستقیم یا Direct mapping :

بین هر بلوک حافظه ی اصلی و بلوک متناظر در حافظه‌ی نهان یک رابطه‌ی مستقیم و یکتا تعریف می‌کند. یعنی اگر مثلاً به بلوک ۵ احتیاج داشته باشیم می‌دانیم که باید در خانه‌ی اول حافظه‌ی نهان قرارش دهیم.

www.it98.ir

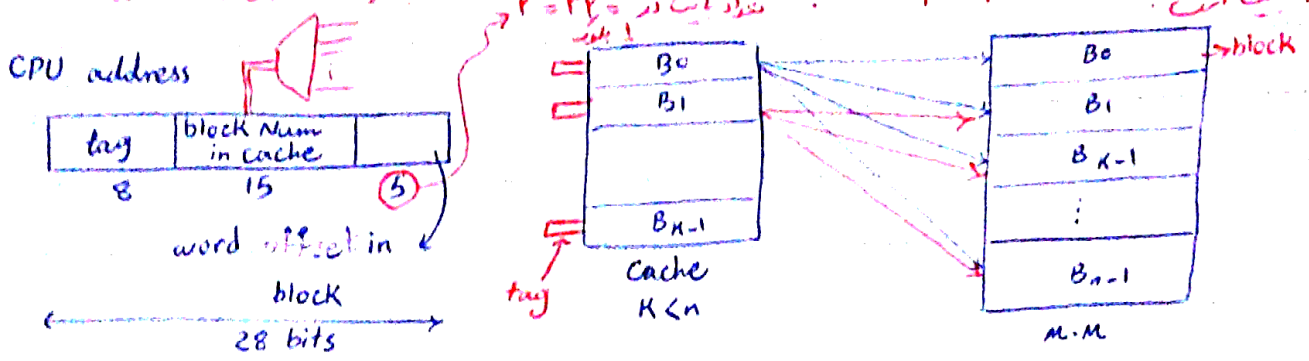
wWw.it98.ir

www.it98.ir

حافظه‌ی اصلی را به n تا بلوک تقسیم می‌کنیم. اینجا هر بلوک ۳۲ بایت است.

هر block در cache متفاوت

تعداد بیت در بلوک = $2^5 = 32$



مثال: Cache = 1 MB , MM = 256 MB

طول آدرس = ? ⟸ 28 Bits

از این ۲۸ بیت، ۵ بیت می‌رود چون offset ۵ بیت دارد و هر بلوک ۳۲ بیت است.

$1\ MB = 2^{20}$ $\qquad \dfrac{2^{20}\ \text{حجم cache}}{2^5\ \text{block Num in M.M}} = 2^{15}$ blocks

در اینجا یک آدرس ۲۸ بیتی از پردازنده issue می‌شود. کنترلر Cache این ۲۸ بیت را به ۳ تا field تقسیم می‌کند. اول به وسطی نگاه می‌کند تا بفهمد Block Number به کجای Cache مربوط می‌شود. براساس این block Number و با استفاده از دیکدر به یک block دسترسی پیدا می‌کند. وقتی می‌خواهد دسترسی پیدا کند فقط به برچسب (tag) نگاه می‌کند. یکسانی برچسب block مبین این است که به این block قبلاً رجوع شده چون هر block جایی واحد در حافظه‌ی نهان دارد.

حسن ⟸ سرعت عمل و سادگی

اشکال ⟸ اگر همزمان به دو تا block نیاز داشته باشیم نمی‌توانیم دسترسی داشته باشیم چون همه چیز fixed است.

block i in M.M = block (i mod K) in cache ← چون cache کوچکتر است

حجم حافظه‌ی نهان = $1\ MB + K \times 8$ bit

K تا tag داریم.

راه حل این مسئله: حافظه‌ی شرکت پذیر یا نگاشت کاملاً شرکت پذیر (Fully associative mapping)

فقط یک tag و یک word offset داریم.

اشکال آن این است که فقط یک برچسب می‌دهیم و باید همه‌ی برچسب‌ها را برای یافتن برچسب مورد نظر

www.it98.ir

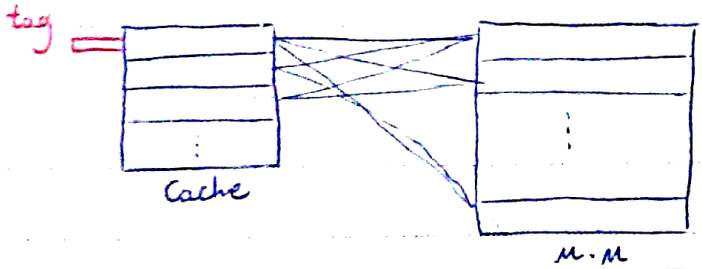
www.it98.ir

جستجو کنیم. برای Sequential Search ⇐ $O(n)$

راه حل حافظه‌ی شرکت پذیر است که همانند مغز انسان طراحی شده و تمام ردیف‌ها جستجوی موازی انجام می‌دهند

یعنی $O(1)$. (فرق حافظه‌ی شرکت پذیر و معمولی)

حافظه‌ی شرکت پذیر از جنس حافظه‌ی Content addressable Mem است.



tag	word offset

هر چیزی هر کجا می‌تواند باشد. حسن: سهولت جایابی و جایگزینی

ایراد: قیمت بالا

نگاشت شرکت پذیر مجموعه‌ای (Set associative memory):

مثلاً

cache را به تعدادی Set تقسیم می‌کنیم. مثلاً ۴ تا ۴ تا جدا می‌کنیم (هر Set = ۴ بلوک)

$\frac{2^{15}\ \text{blocks}}{4} = 2^{13}$ Set block i in M.M = Set $(i \bmod 2^{13})$ in cache

هر Set با MM، direct map است. داخل هر Set، associative است. در این روش، مزیت های دو روش پیش گفته تلفیق شده است.

www.it98.ir

wWw.it98.ir

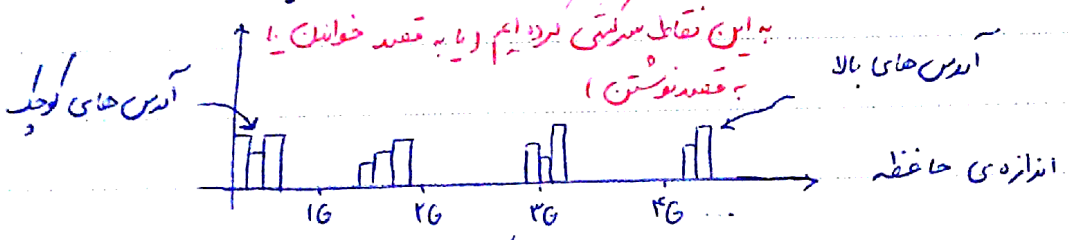
www.it98.ir

Subject:
Year. Month. Date.

جلسه‌ی بیست و پنجم : ادامه‌ی بحث Cache :

۱۳۹۳/۳/۱۵

نمودار میله‌ای نمونه از فراوانی مراجعات به آدرس های مختلف حافظه توسط یک برنامه :



چرا ما در کامپیوتر به نقاط ابتدایی و نقاط انتهایی حافظه دسترسی پیدا می‌کنیم ؟

انتها ⇐ Stack ، ابتدا ⇐ interrupt ها و اطلاعات پایه‌ی سیستم

البته این نمودار مربوط به یک بازه‌ی زمانی است که در آن تغییرات را مشاهده کرده‌ایم. بنابراین وقتی برنامه تمام می‌شود و اجرای برنامه‌ی دیگری آغاز می‌شود، اتفاقات دیگری می‌افتد و این میله ها تغییر پیدا می‌کنند. یعنی رفتارهای آماری برنامه ها تغییر می‌کند. اما تحقیقات نشان داده که این نقاط آرام آرام تغییر می‌کنند و یکباره زیاد یا کم نمی‌شوند.

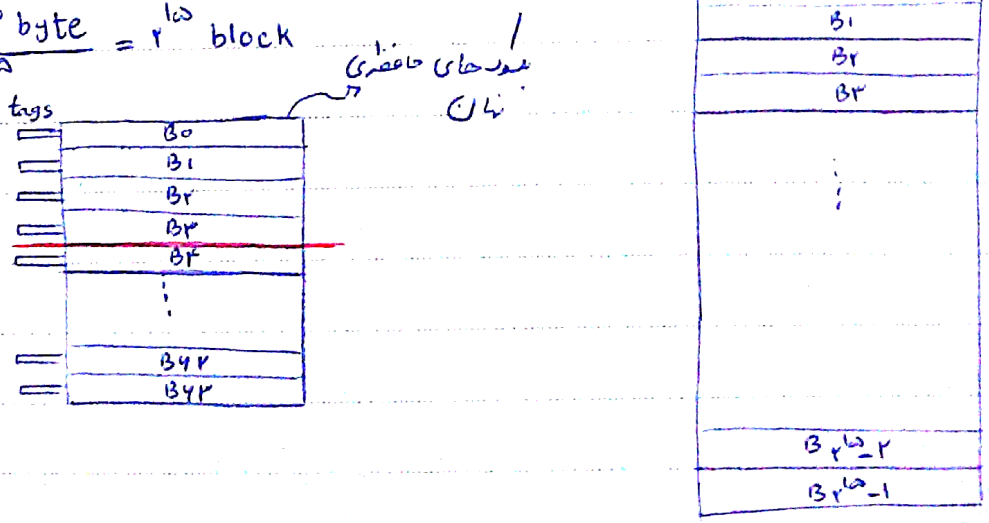
پرکاربردترین روش ذخیره سازی در حافظه‌ی نهان (Set associative mapping) :

یک حافظه‌ی نهان با ۱۶ تا Set داریم. ۶۴ تا بلوک داریم که ۴ تا ۴ تا تقسیم شده اند توی Set. هر بلوک را ۳۲ بایت در نظر می‌گیریم. پس :

$$\text{Cache Size} = \underbrace{2^5}_{\text{بایت بلوک ها}} \times \underbrace{64}_{\text{تعداد بلوک ها}} = 2^5 \times 2^6 = 2^{11}\ \text{byte} = 2^{1}\ \text{KB}$$ (یک Cache کوچک)

(فرض) $\text{M.M Size} = 1\ \text{MB} = 2^{20}\ \text{byte}$

$$\frac{2^{20}\ \text{byte}}{2^5} = 2^{15}\ \text{block}$$



www.it98.ir

Subject:
Year. Month. Date. ( )

Ref1 = 00000 hex آدرس اولین مراجعه به حافظه

Ref2 = A0000

Ref3 = 00008

Ref4 = A000F

Ref5 = AF345

Ref6 = 0F340

Ref7 = 0080D

Ref8 = A000B

با توجه به Refrence های داده شده، تعداد hit و miss را در حافظه ی نهان بیابید.

B(i) in M.M = Set (i mod K) (K تعداد مجموعه ها)

آدرس ها ۲۰ بیتی هستند ← | tag | Set | word |

	tag	Set #	wordOffset	hitOrmiss	Block #	Set #
Ref1	00000000000	0000	00000	miss	B0	0
Ref2	10000000000	0000	00000	miss	B1	0
Ref3	00000000000	0000	01000	hit	B0	0
Ref4	10100000000	0000	01110	hit	B1	0
Ref5	10101111001	1010	00101	miss	B10	10
Ref6	0000 1111 001	1010	00000	miss	B11	10
Ref7	00000000100	0000	01101	miss	B2	0
Ref8	10100000000	0000	01010	hit	B1	0
Ref9	10110000000	0000	00000	miss	B3	0
Ref10	11000000000	0000	00000	miss	?	

توی cache یک warm Up period داریم که وقتی هنوز cache خالی است و پر نشده استفاده می شود. اصطلاحاً می گویند cache هنوز گرم نشده. یعنی cache آرام آرام تمام بلوک هایش پر شود و براساس اصل مجاورت امید داشته باشیم که مراجعات بعدی توی cache پاسخ داده شوند.

| B0 | B1 | B0 | B1 | B2 | B3 |

برای پر کردن ؟ باید اونی که از همه قدیمی تر هست ما بیندازیم بیرون.

previous B0 should be replaced with new one

www.it98.ir

wWw.it98.ir

به این الگوریتم، LRU یا پیرترین (Least Recently Used) می‌گویند.
آن که اخیراً کمتر مراجعه شده یا اصلاً مراجعه نشده یعنی آن که از همه پیرتره می‌ریزیم دور.
حالا که B_0 را دور انداختیم، B_0 را قبل از دور انداختن باید در M.M بنویسیم یا نه؟
وقتی می‌نویسیم که توی آن (یعنی توی B_0) چیزی نوشته باشیم.
بر این اساس یک بیت به نام dirty bit یا بیت کثیف گذاشته‌اند که وقتی set می‌شود که در آن چیزی نوشته باشیم. اگر 0 باشد یعنی فقط عمل خواندن صورت گرفته است.
دو نوع سیاست write داریم:
write through: هر تغییری در حافظه‌ی اصلی منعکس شود (نوشتن برابری یا سینکرونیک).
write back: مرسوم‌تر است. مثلاً توی حلقه وقتی index مدام در حال تغییر است نیازی نیست مرتباً در M.M ثبت شود بلکه در local cache قرار می‌گیرد. پس نویسی وقتی در حافظه‌ی اصلی می‌نویسد که مجبور شود دور ریخته شود.
write through وقتی صورت می‌گیرد که از پردازنده‌ی دیگر یا I/O device بخواهیم استفاده کنیم. مثلاً کارت گرافیک یا کارت شبکه. (اگرچه نفر دارند از حافظه‌ی مشترک استفاده می‌کنند باید اطلاعات را به آنها هم بدهیم). البته این مورد تا حدی سیاست cache را زیر سئوال می‌برد.
LRU بلوکی را دور می‌ریزد که دورتر گذشته‌ی دورتر واقع باشد.
مثال LRU:

1 2 3 4 5 1 2 3 4 5 1 2 3 4 5

M M M M M

بلوک را خودم نمی‌ریزم

مثلاً در حلقه‌ها چنین اتفاقی می‌افتد.
الگوریتم Optimal: بلوکی را جایگزین می‌کنیم که در آینده‌ای دورتر از بقیه مورد استفاده قرار می‌گیرد.
این سیاست قابل پیاده‌سازی نیست چون ما از آینده خبر نداریم. وقتی قابل پیاده‌سازی است که یکبار برنامه اجرا شده باشد و دیده باشیم که برنامه سراغ کدام بلوک‌ها می‌رود. این کار را profiling می‌گویند (سابقه‌نگاری). براساس آن سیاست cache را انتخاب می‌کنیم که LRU باشد یا MRU
Most Recently Used