جلسه‌ی اول - 1403,7,2

زبان ماشین یک زبان سطح پایین (Low level) در سطوح زبان های برنامه‌نویسی - قابل فهم برای ماشین

زبان گفت‌وگو برای اکثر لپ‌تاپ ها با پردازنده Intel ، 80x86 است.

پردازنده ARM (Advanced Risc Machine) خیلی پر استفاده در اکثر موبایل ها

دلایل استفاده از زبان ماشین (Assembly):

- امنیت : هک کردن در جهت + مثلا بازگشایی قفل ها
- firmware modification : رفع محدودیت های نرم افزاری ثابت افزاری / تغییر و ارتقا و firmware debug / تحلیل آسیب پذیری firmware ها و تجهیزات
- (Code/Performance Optimization)
- استفاده بهتر از منابع سیستم به علت ارتباط مستقیم با پردازنده مثلا سریعتر کردن سرعت اجرای برنامه

* ثابت افزار (firmware) (کارخانه افزار) : نرم افزاری که کارخانه روی دستگاه قرار داده و کامپیوتر با آن روشن می شود و به ندرت نیز دچار تغییر می شود مثل تجهیزات ، ماشین ها (ECU) ، ماشین لباسشویی ظرفشویی و کامپیوتر (ROM) ← Boot , Bias

ندرتا عوض می شود ولی امکان update و به روز رسانی دارد.

یک اصطلاح در حوزه‌ی امنیت "Zero day vulnerability" یا "Zero day attack" ← حمله روز نخست

مثلا ۲ بار حمله‌ی سایبری به سامانه‌ی سوخت کشور ، پاتیل فولاد مبارکه ، منفجر شدن لوله های گاز کشور به خاطر حمله سایبری

به معنای وجود یک نقص یا آسیب در قطعه است که هنوز منعکس نشده که یا سازنده نیز نمی داند و یک bug در سیستم وجود دارد و مهاجم آن را کشف می کند، یا سازنده می داند و نگفته - دلایل:

- مسائل تجاری
- مسائل سیاسی ← حمله سایبری به سانتریفیوژهای نطنز به علت تجهیزات زمینی

دیاگ (دیاگ نوستیک) ← تشخیص عیوب ECU ماشین

کار با ۲ پردازنده Intel 80x86 و ARM به خاطر استفاده‌ی حداکثری آنها در کامپیوترها ، موبایل ها و ماشین ها

RISC: Reduce Instruction Set Computers پردازنده‌ی کامپیوترهای کم دستور

CISC: Complex Instruction Set Computers پردازنده‌ی کامپیوترهای پردستور (بیش دستور)

۲ خانواده‌ی پردازنده‌ها که Assembly‌ها براساس آنها طراحی شده

ARM, mips ← RISC

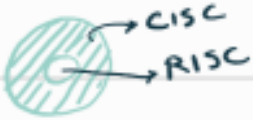
80X86 ← CISC

RISC ← تعداد دستورات کم ، CISC ← تعداد دستورات زیاد

* هرچقدر تعداد دستورات کمتر باشد با تعداد بیت کمتری قابل کدگذاری است مثلاً یک کامپیوتر با ۴ دستور با ۲ بیت قابل کدگذاری ولی در پردازنده های Intel کدها طول دستورات به ۱۱ بیت نیز می‌رسد.

هر چقدر تعداد دستورات بیشتر باشد، کدگذاری سخت‌تری می‌شود و حجم حافظه‌ی بیشتری نیز اشغال می‌شود.

- در حال حاضر پردازنده‌ی RISC بر CISC برتری دارد و اکثر سازنده‌ها پردازنده های RISC طراحی می‌کنند Intel در دام خودش افتاده چون در حال حاضر سرعت پردازش پردازنده‌های RISC خیلی بیشتر از CISC است و در دنیای امروز یکی از دلایل مدنیت سرعت پردازش بالاست. ولی امروزه Intel پردازنده‌ی طراحی کرده که لایه‌ی درونی و طبق پردازنده RISC و لایه‌ی پیرامونی آن CISC است بنابر در تراشه‌ی 80x86 شما به اول دستور میدید و اول در لایه‌ی درونی دستورات خودش رو به RISC ترجمه می‌کنه چون Intel فهمیده که نمی‌تونه با یک ماشین ۱۰۰٪ CISC کار کنه.



ARM و بقیه‌ی پردازنده‌ها از ابتدا RISC طراحی کردند.

سایست نمره دهی: - حداقل ۵ تمرین ۲-۳ نمره

- پروژه روی 80x86 یا ARM ۵-۴ نمره

- حداقل ۲ تا امتحان میانترم و یا پایانترم

ایمیل استاد: jahangir@sharif.edu .

. منبع خیلی شخصی نداریم ولی ممکنه توی یک جمله بگه از کجا درس داده (اگر لطف کنه 😊)

تاریخچه ی کامپیوتر

- ماشین پاسکال (۱۶۴۲)

ماشین مکانیکی ، دارای چرخ دنده و یک سری شماره ، قابلیت انجام جمع و تفریق

یک ماشین شمارنده رو به جلو یا عقب ، اولین ماشین مکانیکی محاسب

برای جمع اعداد ، به مقدار آنها چرخ دنده ها رو به جلو می چرخانند.

وقتی عدد اول (رقم) به ۹ می رسید و می خواست ۱۰ بشه یک رقم نقلی (Carry) تولید می کرد و به

سمت دستی منتقل می کرد و باعث می شد چرخ دنده سمت چپ هم بچرخه.

و اگه تفریق انجام می دادیم یک رقم قرضی (Borrow) قرض می گرفت و کم می کرد و می داد سمت راستی.

- لایب نیتز (۱۶۷۰)

یک ماشین مکانیکی که با الهام از ماشین پاسکال ساخته شد و قادر به انجام جمع و تفریق و ضرب و تقسیم بود.

لایب نیتز گفت ضرب و تقسیم ، جمع و تفریق متوالی هست.

- ضرب رو به صورت جمع های متوالی انجام می داد : $5 \times 4 = \underbrace{5+5+5+5}_{\text{۴ بار}} = 20$

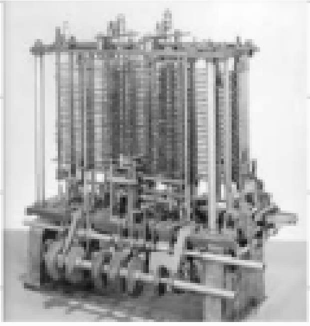
! ضرب عمل اصلی نیست. پس چرا از ضرب استفاده می کنیم؟ برای اینکه با یک دستور به جای مثلا ۴ دستور عمل رو انجام بدیم ولی در لایه های زیرین مثلا Compiler عمل جمع انجام میشه!

اصل : سادگی باعث بهبود کارایی می شود.

- تقسیم رو به صورت تفریق های متوالی انجام می داد : $18 \div 3 = ?$

۱	$18-3=15$
۲	$15-3=12$
۳	$12-3=9$
۴	$9-4=6$
۵	$6-3=3$
۶	$3-3=0$

$18 \div 3 = 6$



- ماشین تحلیلی چارلز بابیج (۱۸۳۰)

پدر کامپیوترهای امروزی نامیده می‌شود هرچند ماشینش مکانیکی بود.

الگوی کاری پردازنده‌ها و ماشین‌های امروزی شبیه این ماشین است.

این ماشین دارای یک حافظه پر از کارت‌های پانچ‌شده بود که فرد با پانچ کردن که برنامه را می‌نوشت

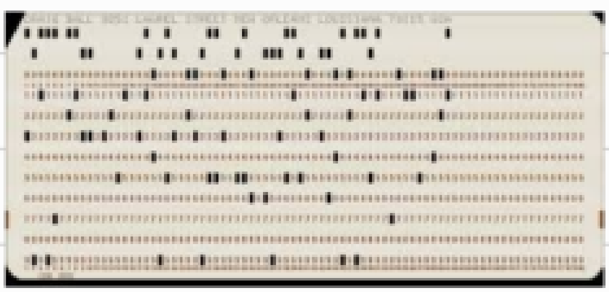
و یک سری کارت دستورالعمل هم تعریف کرد و داخل یک مخزن قرار داد

مبنای این ماشین شبیه ماشین لایب‌نیتس است و همان کارهای جمع و تفریق را انجام می‌دهد با این تفاوت که به جای گفتن ضرب جمع و تفریق شما آن را پانچ کرده اید.

پس یک مخزن برای دستورات و یک مخزن برای داده‌ها داشت که یا داده‌ها را می‌خواند یا به عنوان خروجی کارتی را پانچ می‌کرد

پس CPU این ماشین، ماشین لایب‌نیتس بود که ضرب و جمع و تفریق و تقسیم انجام می‌داد، یک مخزن دستورات و یک سری کارت داده

برای ورودی و خروجی ← مبنای ساختار کامپیوترهای امروزی



کارت پانچ یا Hollerith card برای اواخر قرن ۱۹ و قرن ۲۰

دارای ۸۰ ستون و ۱۲ ردیف، هر ستون معرف یک کاراکتر است

سپس کدها روی تعدادی از کارت‌ها نوشته می‌شدند و با دستگاه کارتخوان

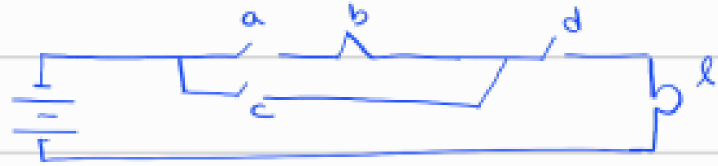
خوانده می‌شدند و برنامه اجرا می‌شد. ← متداول شدن پردازش دسته‌ای یا Batch processing

PHD ← چرایی و چونه معیارها و ساختارها و طراحی های مختلف بی میبه

همه ی کامپیوترها از stored program computer (ماشین با برنامه ی ذخیره شده) پیروی می کنند.

کامپیوترهای امروزی فقط برنامه های ذخیره شده روی آن ها را اجرا میکنند.

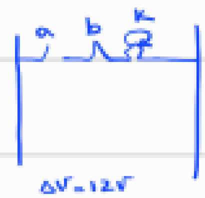
هولریت برای سرشماری اطلاعات آنها را روی کارت پانچ میکرد سپس یک دستگاه مکانیکی آن را می خواند ← تاسیس شرکت ماشین های جدول بندی ← IBM ۱۹۲۴



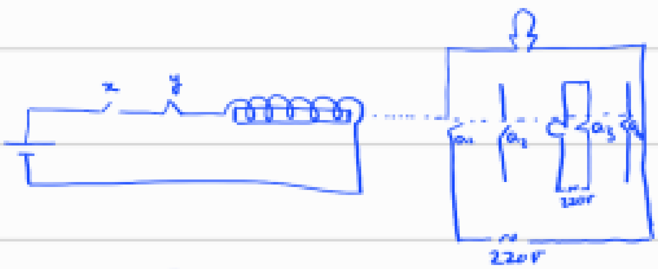
$$\Rightarrow \left((a\cdot\bar{b}) + c\right)d = l$$

21 ← مدار کلیدزنی ترتیبی

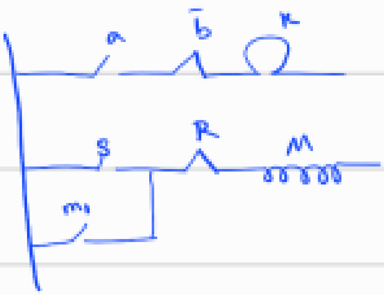
$K = a\cdot\bar{b}$



"نمودار نردبانی (ladder diagram)"



← رله ی الکتریکی یا الکترومکانیکی

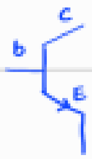


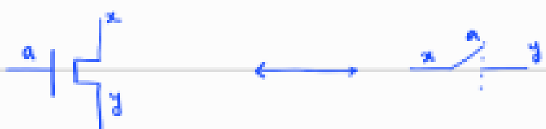
حافظ می سازد ، S را می بندد جریان شار ایجاد شده ، m₁ نیز بسته می شود و حفظ ذخیره می شود ←

ترانزیستور ← پدر کامپیوتر

لامپ های خلا به نقش کلید ، با تحریک ولتاژ قطع و وصل می شوند.

کاربرد مهم ترانزیستور : تقویت ولتاژ و جریان ولی شکل کلید از آن استفاده کنیم.





چرا ترانزیستور؟ چون کوچک است (میکرونی) - کم مصرف - ارزان - مدارات بزرگ را راحت تر می سازد.

چرا FortRan هنوز زنده است؟ چون میراث نرم افزاری است و تغییر آن به زبان دیگر هزینه دارد.

کامپیوترهای امروزی همه از ماشین [illegible] پیروی می‌کنند. این مدل stored program هست یعنی یک برنامه از قبل ذخیره شده و برنامه‌ها بر همین اساس اجرا می‌شوند.

الگوریتم فون نیومن ← شیوه‌ی اجرای دستورات در پردازنده، اساس کار کامپیوتر ! بسیار مهم

پردازنده در هر دستور و در هر سیکل به سراغ حافظه می‌رود و استعلام می‌گیرد که چه کنم؟

۱- واکشی (Fetch) واکشی دستور از محل حافظه‌ای که توسط شمارنده برنامه (program counter) یا اشاره‌گر دستورالعمل (Instruction pointer) ...

آدرس	دستور
PC →	I_1
	I_2
	⋮
	I_n

تمام داده‌ها و دستورات در آدرس‌های مخصوصی در کامپیوتر ذخیره شده‌اند. بنابراین اگر شما به سراغ دستوری (می‌خواهید) یا داده‌ای بروید باید آدرس آن را بدانید.

پس اولین کار برای مراجعه به یک دستور، دانستن آدرس آن است.

در تمام پردازنده‌ها حداقل یک اشاره‌گر (Instruction Pointer) وجود دارد که حاوی دستور بعدی را اشاره قرار می‌دهد. بنابراین برای اجرای دستور پردازنده نگاه می‌کند که اشاره‌گر به چه آدرسی اشاره می‌کند که به آن آدرس مراجعه کند. در زبان Assembly دائما با آدرس‌ها کار می‌کنیم ولی در زبان‌های سطح بالا زیاد از آدرس استفاده نمی‌کنیم ولی در آنجا هم در لایه‌های زیرین بعد از Compile یا Interpret شدن دستورات و متغیرها تبدیل یا ترجمه به آدرس می‌شوند.

پس اولین کار واکشی (Instruction fetch) دستورات از محلی که Program Counter (یا در قدیم Instruction Pointer) به آن اشاره می‌کند.

کد عملیات (Operation Code (OPCode)) ← در حافظه‌ی دستور OPCode ها ذخیره می‌شوند. چون به زبان فارسی نمی‌گوییم! چون ماشین فقط 0 و 1 (زبان Binary) می‌فهمد

کاربردهای Assembly: - افزایش سرعت برنامه‌ها - مهندسی معکوس - شکستن قفل - کشف رخنه‌گاه یا منفذ (Backdoor)

Back Door: به معنای در مخفی یا رخنه‌گاه، وجود یک ضعف و منفذ در تجهیزات صنعتی ← Zero Day vulnerability ← منفذهای کشف نشده

۲- کدگشایی یا رمزگشایی (Instruction Decode): تمام پردازنده‌ها کد عملیات یا دستور را می‌خوانند و آن را ترجمه می‌کنند. (این کار از روی جدول دستورات یا opcodes صورت می‌گیرد.)

۳- خواندن عملوندها (Read Operands): ← یک قالب دستوری متداول در پردازنده‌ها

00000001 | 001 | 100 | 110

- 00000001 ← کد عملیات (Operation code) که مثلا در اینجا 65 است به معنی اینکه جمع بکن (Add)
- 001 ← Register1
- 100 ← R_4
- 110 ← R_6

← یک دستور به زبان Binary و قابل فهم برای ماشین

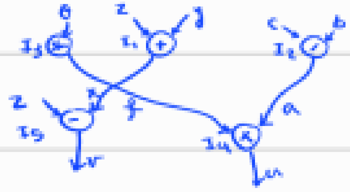
یک دستور Assembly که می‌گوید ← Add R_1, R_4, R_6 یعنی برو محتوای R_4 و R_6 را بخوان و حاصل آنها را جمع کرده و در R_1 قرار بده

۴- اجرای دستور (Execute)

۵- پس‌نویسی نتیجه (Write Back Result) یعنی بعد از جمع R_4 و R_6 نتیجه را در ثبات مقصد قرار بده

۶- به‌روزرسانی PC (PC Update): پس از اجرای دستور، به سراغ دستور بعدی می‌رود

* ما به صورت ترتیبی دستورات را اجرا می‌کنیم یعنی دستورات رو به ترتیب و از بالا به پایین اجرا می‌کنیم (البته مخالفت نکنیم 😅) چرا؟! چون می‌توان دستورات را به صورت موازی اجرا کرد.



DFG → Data Flow Graph

گراف جریان داده‌ها

مثلا:

- $I_1: x = y + z$
- $I_2: a = b/c$
- $I_3: f = \sin(\theta)$
- $I_4: u = a \times f$
- $I_5: r = x - z$

* اولین نقص در کامپیوترهای امروزی که زاده‌ی مدل فون نویمن است همین به ترتیب و از بالا به پایین اجرا کردن برنامه‌هاست.

زندگی پردازنده‌های ما بسیار پوچ و عبث است چون در یک حلقه‌ی تکراری بی‌پایان واقع‌اند (تا لحظه‌ی خاموش کردن کامپیوتر) که فراغت یعنی حتی وقتی شما عملی هم انجام نمی‌دهید پردازنده دارد no operation انجام می‌دهد یعنی مثلا a AND a را در a می‌نویسد یا OR b با b را در b می‌نویسد چون مجبور است تا زمانیکه کامپیوتر روشن است عملیات انجام دهد.

امروزه گفتن چند صد سری فقط یک دستور بخوانیم چون هر سری رفتن به حافظه وقت‌گیره چون باید استعلام بگیریم چکنم، خب هر بار هر سری سری حافظه رفتید ۱۰ تا دستور بخونید

تقریبا تمام پردازنده‌های امروزی هنگام مراجعه به حافظه به جای یک دستور چند دستور (۱۰ تا ۱۰۰ یا ۳۲ تا ۶۴) می‌خوانند چون بهای بالای رسیدن حافظه زیاد شده است. مثلا پردازنده از آدرس 1000 رفته و 20 Byte دستور می‌خواند و چندتایی ترجمه می‌کند و اگر توانایی و دستورات و متغیرها موجود باشد محاسبه‌ها را می‌خوانند.

برای اجرا اگر پردازنده multiple core (یا در تیم multiple ALU (Arithmetic Logic Unit)) باشد یعنی شما چند واحد پردازش داشته باشید همه‌ی داده‌هایی را که خوانده اید می‌توانید محاسبه کنید. یک اتفاق مهم! پردازنده به صورت پنهانی و در پشت صحنه برخلاف تفکر شما دستورات را چندتایی اجرا می‌کند.

بیشتر پردازنده‌ها بدون اطلاع شما دستورات را در پشت صحنه موازی انجام می‌دهند (Out of Order Execution) ← لعنتی‌ها بدون ترتیب دستورات در پشت صحنه بین نوشتن چندگانه ⚠ (خطر!) ← هنوز پردازنده‌ها اجازه ندارند در پشت صحنه نتایج را چندتایی ثبت کنند (Out of Order Commitment)

یعنی پردازنده‌ها بدون ترتیب دستورات را انجام می‌دهند اما به کاربر احترام گذاشته و با ترتیب نتایج را ثبت می‌کنند!

↳ اصطلاحا هنوز Semantic حفظ و رعایت می‌کنند.

امروزه پردازنده‌های super skaller مثل Intel با درجه بالا پیش‌خوانی دستورات را چندتایی اجرا و پیش‌بینی می‌کنند.

امروزه الگوریتم‌هایی برای بهبود پیش‌گویی به پردازنده‌ها اضافه شده است و برای کشف الگوریتم‌ها و راه‌هایی برای پیش‌بینی و پیش‌گویی سریعتر تلاش‌های بسیاری صورت می‌گیرد.

چه پیش‌گویی‌هایی؟ ← یکی از آن‌ها الگوریتم پیش‌بینی پرش (Branch Prediction)

```
for(i=0 ; i<10000 ; i++){
 ⋮
}
```

در این الگوریتم بعد از رسیدن به دستور آخر PC Update صورت نمی‌گیرد و مثلا دستورات loop مدام دوباره اجرا می‌شوند پس PC Prediction انجام می‌دهد.

← برای دستورات شرطی مثل if، (do)while، for یک جدول سوابق ایجاد می‌شود و براساس سوابق محتمل‌ترین حالت بعدی را پیش‌بینی کرد.

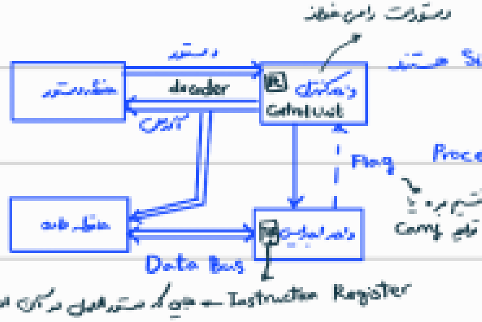
پیش‌گویی‌ها: Branch Prediction، Instruction Prediction، Value Prediction

← انجام محاسبات برحسب داده‌های قبلی زودتر از انجام عملیات‌های دیگر و خروجی دادن سریع‌تر در صورت درست بودن مقادیر داده‌ها

اصطلاح "Approximate Computing" , "Lazy Arithmetic"
محاسبات تقریبی — حساب تنبلانه

$$\begin{cases} x = 3.26^{a} \times 4.97^{b} \\ x' = 3.27 \times 4.86 \end{cases}$$

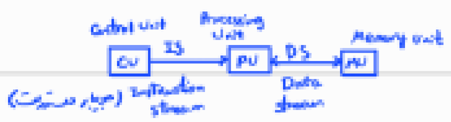
کامپیوتر بعد از محاسبات دو دیگر برای دو محاسبات انجام نمی‌دهد چون تفاوت آنها بسیار کم است.



کامپیوترهایی که با الگوریتم فون نویمان کار می‌کنند معروف به تک پردازنده Single Processor هستند

و شامل یک بخش عملیاتی است که به آن Processing Element (Unit)، ALU، Processor،

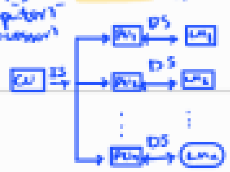
Executer Unit، Execution Unit، Core



این معماری سریع نیست چون قابلیت اجرای چندکاره ندارد

با کامپیوترهای موازی (Vector Computer) (Array Processor) SIMD معماری ← توانایی اجرای یک دستور روی چند داده

مثلاً توانایی جمع ماتریس‌ها با همه درایه‌ها بدون استفاده از حلقه با یک دستور ←



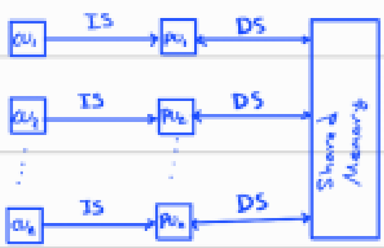
قسمت داغ برنامه (Hot Part): قسمتی که شما بیشترین زمان را در آنجا تلف می‌کنید

که می‌توانیم این قسمت را با Assembly بنویسیم و سرعت را بالا ببریم و هدر رفت زمان را کاهش دهیم.

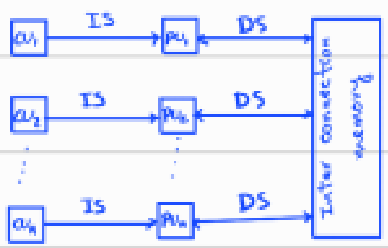
* برای نوشتن برنامه‌های Drive و برنامه‌هایی که با سخت‌افزار در ارتباط هستند باید از Assembly استفاده کنیم.

مدل MIMD (Multiple Instruction Multiple Data Stream)

یعنی با هم کارهایی را انجام می‌دهند که به یک هدف مشترک برسند که یا از طریق حافظه مشترک اطلاعات را با هم مبادله می‌کنند یا از طریق شبکه



MIMD (with shared memory)



جلسه ی چهارم ۱۹/۷/03

برای انجام عملیات های ریاضی عمل جمع مکمل گیری کافی است ← اگر شما قادر به ساخت جمع کننده های مناسب (از نظر سرعت، هزینه، حجم تحت اشغال، توان مصرفی و...) و مکمل گیر باشید ۹۰٪ یک کامپیوتر را تشکیل داده اید

فصل ۲ سیستم های نمایش اعداد (Number Representation Systems)

سیستم : مجموعه ای از اعضا و مولفه هایی که با یکدیگر در ارتباطند و برای یک هدف مشترک فعالیت می کنند.

در سبک نمایش رومی، بعضی از اعداد وزن منفی می گیرند. مثل $IV = 4$ ← زمانیکه عددی با ارزش کمتر در سمت چپ عددی با ارزش بیشتر قرار بگیرد وزن منفی خواهد داشت.
$IC = 99$

Digit = رقم   Number = عدد ← "symbol"نمایش یک مقدار با نمادها

DS (Digit Set) ← مجموعه ی ارقام ← $DS: \{0, 1, \dots, 8, 9\}_{r=10}$ → radix ریشه
Base پایه

$N_1 = 36$ → نمایش یک مقدار - یک نماد است →(به طور صحیح توصیف) $N_1 = 3 \times 10^1 + 6 \times 10^0 = 36$ → Value (مقدار و ارزش)

$N_1 = (36)_8 \rightarrow 3 \times 8^1 + 6 \times 8^0 = (30)_{10}$ → value of N_1

"اعداد اسرار آمیزند، جدیشان نگیرید!"

حروف ابجد

ع = ۷۰ ...

مثلا اسم حضرت "علی" ارزش ۱۱۰ است.

روش امروزی و مرسوم ← Positional System (سیستم موقعیتی) ارزش عدد براساس ارقام و موقعیت ارقام تعیین می شود.

فرمول کلی:

$$(x_{k-1} x_{k-2} \cdots x_0 . x_{-1} x_{-2} \cdots x_{-m})_r \rightarrow \text{value} = \sum_{i=-m}^{k-1} x_i \cdot r^i$$

k رقم صحیح (integral part) — m رقم اعشاری (fraction part)

این فرمول فقط برای اعداد با یک ریشه استفاده می شود

با اصطلاحاً "fixed radix positional system" یا "سیستم نمایش موقعیتی تک ریشه"

سیستم نمایش چند ریشه در روزمره ← زمان و ساعت   R = 0 24 60 60
با ریشه های متفاوت

نمایش اعداد مختلف در کامپیوتر با ۴ بیت

Unsigned Integer

$\{0,1,\dots,15\}$

Signed Integer

01000 = +0 $\{-7,-6,\dots,-0,+0,\dots,6,7\}$

11000 = -0

این سیستم تعداد بیشتری اعداد + و - دارد ولی ۲ تا ۰ دارد

Signed Fraction

$(0.101)_2 = 2^{-1}+2^{-3} = +0.625$

$(0.1011)_2$ ← رقم ۴ ; $(1011)_2 = 11$ ⇒ $\frac{11}{2^4} = \frac{11}{16}$

* راه سریعتر برای محاسبه ی $(0.1011)_2$

Logarithm

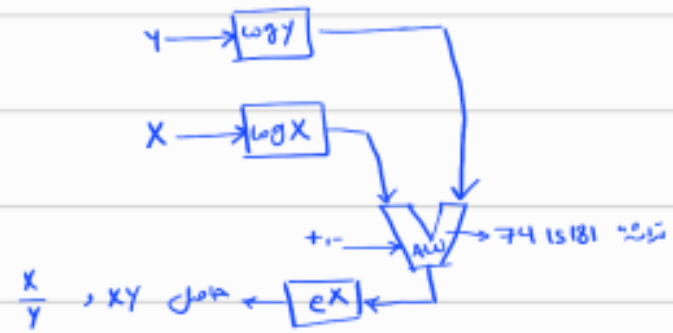
یک کامپیوتر که لگاریتم اعداد را ذخیره می کند

ضرب و تقسیم تبدیل به جمع می شود

ولی جمع و تفریق آسان نیست

$\log XY = \log X + \log Y$, $\log \frac{X}{Y} = \log X - \log Y$

این کامپیوتر فقط برای ضرب ها و تقسیم های زیاد خوب و با صرفه است.



$u = 1-x \Rightarrow 0 < u \leq 0.5$

تبدیل تقسیم به ضرب $\frac{1}{x}$ و $0.5 \leq x < 1$

$$\frac{1}{x} = \frac{1}{1-u} = \frac{1+u}{1-u^2} = \frac{(1+u)(1+u^2)}{1-u^4} = \frac{(1+u)(1+u^2)(1+u^4)}{1-u^8} \to \lim = \frac{1}{x}$$

$$= (1+u)(1+u^2)(1+u^4)\dots(1+u^{2^n})$$

$\lim 1-u^8 = 1$

سیستم های یک ریشه با ریشه های دیگر (مثلا 0.5 یا $\sqrt{2}$)

$r=3$, DS: $\{-1,1\}$ (ترنری)

Signed digit number system سیستم نمایش علامت در اعداد

Ex) $r=10$, DS: $\{-4,-3,-2,-1,0,1,2,3,4,5\}$ → $A=(315)_{10} = 3\times10^2 - 1\times10^1 + 5\times10^0 = 295$

$B=(\bar{3}15)_{10} = -3\times10^2 + 1\times10^1 + 5\times10^0 = -285$

* non redundant ← تعداد ارقام از مقدار ریشه کمتر باشد (مانند نیت)

(1051)

Ex) r=10 , DS : {-6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6} → Digit Set Signed Digit

(۱۳ رقم داریم)

* هنگامی که دامنه‌ی ارقام را زیاد می‌کنیم افزونگی نمایش خواهیم داشت یعنی ممکن است برای یک مقدار چند نمایش داشته باشیم

$$\begin{cases} 1\bar{6}05 = 905 \\ 9\bar{1}5 = 905 \\ 905 = 905 \end{cases}$$

* این سیستم نمایش برای جمع عدد Carry تولید نمی‌کند! هیچ عدد Carry از یک ستون به ستون دیگر منتشر نمی‌شود

چون رقم نقلی هر ستون در همان ستون جذب می‌شود چه بسا با عدد دیگری نمایش داده شود (کاربرد افزونگی)

در حالت عادی مدت زمان جمع n عدد $O(n)$ است ولی در اینجا $O(1)$ است چون می‌توانیم اعداد را موازی جمع می‌کنیم

چه هزینه‌ای میدهیم؟ گام‌های طولانی و سخت افزار بیشتر

سیستم نمایش باقی مانده یا مانند (residual number system)

$$\begin{array}{r} 91 = (2,1,6) \\ + \quad 39 = (0,4,9) \\ \hline 80 = (2,0,3) \end{array}$$

معرفی چند پیمانه مثلا 3 و 5 و 7 (فقط پیمانه‌ها نسبت به هم دوبه‌دو اول‌اند)

در این مدل هم می‌توان موازی جمع کرد و رقم نقلی هم نداریم!

جلسه پنجم 03,7,16

اگر در یک سیستم، اعداد به چند صورت قابل نمایش باشند ← این سیستم ها redundant هستند مثلاً $1\bar{4}=6$ ، $6=6$

$DS=\{-5,-4,\dots,0,1,\dots,5\}_{r=10}$ → در این سیستم عدد Carry در خروجی تولید نمی شود ولی چون در این سیستم یک جمع میانی داریم ممکن است در آنجا Carry تولید شود.

مثال) $DS=\{-6,-5,\dots,0,1,\dots,5,6\}_{r=10}$

$$\begin{array}{r} 4536 \\ +\ 1466 \\ \hline 1\ 1\ 1 \leftarrow c \\ +\ 5\bar{1}\bar{1}2 \leftarrow w \\ \hline 6002 \end{array}$$

جمع میانی (w)

در این مرحله به هیچ عنوان Carry نخواهیم داشت!

$$u_i=(x_i+y_i)-10c_i,\quad c_i=\begin{cases}1 & x_i+y_i\geq 6\\ -1 & x_i+y_i\leq -6\\ 0 & \text{otherwise}\end{cases}$$

در این جمع تمام اعداد موازی جمع می شوند و منتظر رقم نقلی نمی مانند یعنی توانستیم $O(n)$ را به $O(1)$ تبدیل کنیم.

حساب "تراختنبرگ" ← روش های محاسبات سریع ذهنی

نکته قصار 😁 : "اکثرا روش های محاسباتی دستی یا ذهنی، برای پیاده سازی در کامپیوتر مناسب نیستند."

مرور مفاهیم نمایش اعداد:
- ریشه (radix) یا مبنا (base)
- رقم (digit)
- عدد (number) با چند رقم نمایش داده می شود
- ارزش یا مقدار (value)

۲ دسته بندی بزرگ برای نمایش اعداد:
- Fixed point Representation ممیز ثابت
- floating point Repre ممیز شناور

● ممیز ثابت: 27.5 ← که یک رقم اعشاری

تعداد اعداد اعشار ثابت است

- حسن ← خطای مطلق ثابت است (چه برای مقادیر بزرگ و چه کوچک) ← طراحی و تفسیر ساده تر و طراحی آسان تر
- کاستی ← خطای نسبی متغیر است (دقت و ...)

0.75 و 0.5 ← 50% خطای نسبی

0.3%: 300 → 301

100%: 1 → 2

برای اعداد کوچک به بسیار تاثیرگذار است.

$10.9 \pm 0.5mm$ خطا (10 ... 11)

دقت ← مطلق / نسبی — همان دقت نسبی

* در محاسبات مهندسی خطای نسبی اهمیت بیشتری دارد!

● ممیز شناور: 10.14 mV — 320.4 V

دقت نسبی ثابت و دقت مطلق متغیر است.

خطای نسبی کم و خطای مطلق متغیر است.

شبیه خط کش گذاریم ← فاصله‌ی خانه‌ها متغیر است

← دقت نسبی ثابت و خطا کمتر است

تمرین پیش دست نه

شمارش: تعداد اعدادی که با ۱ و ۲ و ۳ و ... شروع می شوند در پدیده‌های موجود در دنیا و طبیعت مثلا موجودی های حساب، جدول ضرب، جمعیت کشورها و شهرها، مساحت کشورها و... هر پدیده ی طبیعی در دنیا

(!) اعداد در جهان به صورت یکنواخت توزیع نشده اند.

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	90
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	98	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

$1 \to 21$
$2 \to 17$
$3 \to 12$
$9 \to 14$
$5 \to 8$
$6 \to 9$
$7 \to 6$
$8 \to 7$
$9 \to 5$

قانون بنفورد (Benford's law)

$$\#d_i \propto \log\left(1+\frac{1}{d_i}\right)$$

تعداد اعدادی که با رقم کوچکتری شروع می شود بیشتر است.

تعداد اعداد شروع شده با رقم d_i

- درخواست اعداد به این صورت خواهد بود که با چه دقت به کار رفته در اعداد پرتقاد را اندازه هم.

برای کشف تقلب در انتخابات، فاکتور های شرکت ها، حساب های مالیاتی و...

نمایش علامت

۱- Signed Magnitude یک علامت به عددی دهد و یک مقدار عدد (علامت اندازه - علامت قدرطلق) $+3, -4$

۲- radix Complement (مکمل ریشه) $C_r(X) = r^n - X$

۳- diminished radix Complement (مکمل ریشه کاسته شده) $C_r(X) = r^n - X - 1$

Ex) ۷ ساعت قبل ساعت ۳ صبح؟ ۸ صبح

$3 - 7 = 3 + C_{12}(7) = 3 + 5 = 8$

$12 - 7 = 5$

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

تبدیل تفریق به جمع

Ex) ۴ رقم داریم و $r = 10$

Value	نمایش
+4999	4999
+4998	4998
⋮	⋮
+1	1
0	0
-1	9999
⋮	⋮
-4998	5002
-4999	5001
-5000	5000

← 10,000 عدد قابل نمایش است. (r^n)

اعداد قابل نمایش با پایه r و n رقم

$r^{n/2} - 1$	اعداد مثبت
1	0
$r^{n/2}$	اعداد منفی

مثال:

$$\begin{array}{r} 5496 \\ -2839 \\ \hline \end{array} \rightarrow C_{10}(2839) = 7161 \rightarrow \begin{array}{r} 5496 \\ +7161 \\ \hline \times 2657 \end{array}$$

جلسه ششم 03,7,21

در ممیز ثابت اگر فرض کنیم اعداد در یک رجیستر ذخیره شوند محل ممیز از قبل پیش‌فرض شده است. رجیستر نگه‌دارنده اعداد

رجیستر ۴ رقمی

| ۳ | ۹ | ۶ | ۷ | $r = 10$

$N_1 = ۳۹۶۷$

بصورت پیش‌فرض محل ممیز سمت راست است که عدد به عنوان یک عدد صحیح (integer) توسط کامپیوتر تلقی می‌شود.

محل ممیز یک محل مشخص نیست بلکه به صورت یک قرارداد با compiler است.

| ۳ | ۹ | ۶ | ۷ | $N_2 = ۳۹/۶۷$

| ۳ | ۹ | ۶ | ۷ | * $N_3 = ۰/۳۹۶۷$

یک ثبات ۴ رقمی با ۴ رقم می‌تواند ۴ value متفاوت را در خودش ذخیره کند.

به اعداد شروع شده با ۰ پیش‌فرض و بزرگتر از ۰ اعداد هنجار (normalize شده) می‌گویند.

| ۳ | ۵ | ۷ | ۲ | $r = 8$ $\rightarrow N = ۳\times۸^۰ + ۵\times۸^{-۱} + ۷\times۸^{-۲} + ۲\times۸^{-۳}$ $\xrightarrow{\text{راه سریعتر}}$ $N_{(value)} = ۳ + \frac{۵\times۸^۲ + ۷\times۸^۱ + ۲\times۸^۰}{۸^۳}$

$= ۳ + \frac{۵}{۸} + \frac{۷}{۶۴} + \frac{۲}{۵۱۲} = ?$ (سیستم مشخص نیست تو)

برای نمایش اعداد منفی، اولین کار، مشخص کردن مجموعه ارقام (Digit Set) است. مجموعه ارقام مختلف در نمایش Binary: (۴ حالت)

$DS_1 = \{0,1\}_{r=2}$, $DS_2 = \{\bar{1},1\}_{r=2}$ (استفاده در RSDB32) , $DS_3 = \{\bar{1},0,1\}_{r=2}$ ← جلوگیری از انتقال رقم نقلی

* مکمل x در ریشه r با n رقم ← $-x = C_n(X) = r^n - x$: نمایش $-x$

* هرگاه جمع دو عدد با یکدیگر برابر یکی از نمایش‌های ۰ شود، آن دو عدد مکمل یکدیگرند.

در متمم ۱ و علامت اندازه، صفر ۲ نمایش دارد

1's Complement: 0000 , 1111

Signed Magnitude: 0000 , 1000

نمایش اعداد علامت دار:

① علامت_اندازه (Signed Magnitude) ← بیت اول نشان دهنده علامت است

0101 = + 101 = +5
1101 = − 101 = −5

← مزیت: مکمل گرفتن تنها با not کردن رقم اول امکان‌پذیر است پس مکمل گیری آسان است.
ضرب و تقسیم نیز بسیار راحت است چون ضرب بیت‌های اول نشان دهنده علامت است.
← کاستی: جمع و تفریق در این روش سخت و پیچیده است.

1's Comp: −5 = 1010
2's Comp: −5 = 1011
SM : −5 = 1101

② مکمل ۲ (tow's Complement) ← $-X = C_2(X) = 2^n - X$ نمایش

  −6 = 1010
+ +6 = 0110
  0 = 10000 ✓

③ مکمل ۱ (One's Complement) ← $-X = C_1(X) = 2^n - X - 1$ نمایش

  −6 = 1001
+ +6 = 0110
  0 = 1111 (0^-) ✓

Decimal	Signed magnitude	One's Comp	Tow's Comp
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	1000	1111	—
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
(*) -8			1000

بازه نمایش اعداد در مکمل ۲ : $[-2^{n-1}, 2^{n-1}-1]$

n = تعداد بیت‌ها

بازه نمایش اعداد در مکمل ۱ و علامت اندازه : $[-2^{n-1}+1, 2^{n-1}-1]$

که یک نمایش برای ۰ بیشتر دارند!

روش های مکمل گیری در ریشه ۲ برای اعداد صحیح :

- مکمل ۱ : not کردن تمام ارقام

$+6 = 0110$, $-6 = 1001$ ⟹ (اثبات تستی) $0110 + 1001 = 1111 = 0$ ✓

- مکمل ۲ : ① $C_2 = C_1 + 1$ ← جمع مکمل ۱ با ۱

$+6 = 0110$, $-6 = 1010$ ⟹ $0110 + 1010 = \cancel{1}0000$ ✓

② از راست به چپ حرکت می‌کنیم تا به اولین ۱ برسیم قبلی ها رو می‌نویسیم بقیه‌ی ارقام رو not می‌کنیم

$C_2(10100) = 01100$

برای اعداد اعشاری : روش اول : مثل روش های قبلی

مکمل ۲ :

$+37.625 = (0100101.101)_2$ (بیت علامت) → $-37.625 = C_2(0100101.101) = 1011010.011$ (مثل روش اعداد صحیح)

مکمل ۱ : باز مثل قبل همه‌ی ارقام، not می‌شوند

روش دوم : $C_r(X) = C_{r-1}(X) + ulp$

unit of lower precision ← کمترین وزن

روش سوم : مراجعه به تعریف رسمی : $-X = C_r(X) = r^n - X$

$+37.625 = (0100101.101)_2$

$-37.625 = C_2(0100101.101) = 2^7 - 37.625$ → $10000000.000 - 0100101.101 = 1011010.011$ ⟹ (اعتبار سنجی) $1011010.011 + 0100101.101 = \cancel{1}0000000.000$ ✓

مثال)

C_{10} ← ulp → !

Value	C_9	C_{10}
+4999	4999	4999
+4998	4998	4998
⋮	⋮	⋮
+1	1	1
+0	0	0
-0	9999	—
-1	9998	9999
-2	9997	9998
⋮	⋮	⋮
-4998	5001	5002
-4999	5000	5001
-5000	—	5000

در اینجا بیت علامت نداریم ولی علامت در دل بیت اول گنجانده شده

$4896 - 2379$ → $4896 + 7621 = \cancel{1}2517$

$C_{10}(2379) = 7621$ ⟹ (اعتبار سنجی) $7621 + 2379 = \cancel{1}0000$ ✓

تفریق ← جمع با مکمل ریشه

جلسه هفتم 03/7/23

مرور : $C_1(X) = r^n - X$, $C_{r-1}(X) = C_r(X) - \text{ulp}$

Ex) $4296 - 3754$ $\quad -3754 = C_{10}(3754) = 6246$ ← روش سریعتر: مکمل اولی نسبت به ۱۰ ، بقیه نسبت به ۹

$4296 + 6246 = 10542$

برای هر ریشه : مکمل گیری «اولی از r ، بقیه از r-1

- تقریبا در هیچ کامپیوتری تفریق کسته نداریم فقط جمع کسته و مکمل گیر

⚠ اگر محاسبات بدون سرریز (overflow) و در نتیجه صحیح باشد [یعنی حاصل جمع یا تفریق در بازه نمایش اعداد بگنجد] یا Carry رقم نقلی را دور می ریزیم.

فرض می کنیم بازه ی اعداد $[+9999, -5000]$ باشد یعنی 10000 مقدار متفاوت داریم که با 4 رقم دهدهی قابل نمایش است.

اگر ریشه 2 بود ما چند رقم نیاز داشتیم ؟ $2^n \geq 10000 \leftarrow n = 14$ پس در ریشه 2 به 14 بیت نیاز داشتیم.

value	C_9	C_{10}
+4999	9999	4999
+4998	9998	4998
⋮	⋮	⋮
+1	1	1
+0	0	0
-0	9999	9999
-1	9998	9998
⋮	⋮	⋮
-4999	5000	5001
-5000	—	5000

$3296 + 4979 = 8275$ ← کامپیوتر نمایش این عدد را نمی بیند overflow رخ داده است

یعنی حاصل جمع یا تفریق درست نیست

و این عدد را یک عدد مثبت حساب می کند

و در ظرف اعداد گنجانده نشده

* می توان ثابت کرد که اگر یک عدد به یک عدد - باشد ، سرریز رخ نخواهد داد.

$-3296.704 = C_{10}(3296.704) = 10^4 - 3296.704$

$= 6703.296$ ← روش میانبر و سریع

راستی آزمایی $3296.704 + 6703.296 = 10000.000$ ✓

چرا می توانیم خط بزنیم ؟ چون علامت آنها متفاوت است پس سرریز رخ نخواهد داد!

$(-1201.122)_3 = C_3(1201.122) = (1021.101)_3$ ← روش میانبر (اولی نسبت به 3 بقیه 2 مکمل گیری)

$= 3^4 - (3^3 + 2\times3^2 + 1\times3^0 + \frac{1}{3} + \frac{2}{9} + \frac{2}{27})$ رسمی و تعریف

حد دوباره باید بریم به مبنای ۳

Good luck

$-3f.4AEB = C_{16}(3f.4AEB)$ میانبر $C0.B515$

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

$= 16^2 - (3\times16 + 15 + \frac{4}{16} + \dots)$ رسمی

← اگر چیزی گفته نشود ریشه 10 یا 16 است و مکمل ، مکمل ریشه و نمایش اعداد در سیستم مکمل ریشه

نمایش ممیز شناور (floating point Representation)

* منطق $N = (-1)^S \times 1.F \times 2^{E-bios}$ ⟸ در مبنای $r=2$

S	Exponent	Fraction
1 bit	e bit	f bit

طول عدد باینری = n

Signed Magnitude سیستم ⟵ $\begin{cases} S=1 \to - \\ S=0 \to + \end{cases}$

استاندارد IEEE 754 ⟵ Scientific Representation مورد استفاده در تمام کامپیوترها برای نمایش اعداد

۲ نمایش و مزیت رایج :

تا 39 رقم قابل نمایش است (تعمیم) ⟵

S	E	F
1 bit	8 bit	23 bit

\+ Single Precision اعداد دقت ساده 32 بیتی

S	E	F
1 bit	11 bit	52 bit

\+ Double Precision 64 بیتی

③ سوال که باید از هر سیستم نمایش عدد پرسیده شود :

۱_ بزرگترین و کوچکترین عدد قابل نمایش چیست؟ [بازه‌ی نمایش اعداد]

۲_ تعداد کل اعداد قابل نمایش

۳_ دقت نمایش (دقت نسبی و مطلق)

* بزرگترین کمیت قابل شمارش یا عدد فیزیکی واقعی؟ ← فکر شود

_ اگر d_1 و d_2 ، ۲ عدد 40 رقمی باشند و $d = \sqrt{d_1^2 + d_2^2}$ باشد d برای کامپیوتر ما قابل محاسبه نخواهد بود.

اگر fixed point باشد کامپیوتر نمی‌تواند این عمل را انجام دهد و دست ما را می‌بندد ولی با floating point

قابل محاسبه خواهد بود چون می‌توانیم اعداد را به صورت $d = a \times 10^b$ نشان دهیم.

F: Fractional

1.F : Matissa $\begin{cases} M_{max} = (1.\overbrace{11\ldots1}^{\text{تا } f})_2 = 2 - 2^{-f} \\ M_{min} = (1.0000\ldots0)_2 = 1 \end{cases}$

مثلاً EX3 ← جمع با 3 (Bias)

معمولاً $Bias = 2^{e-1} - 1$ است که e تعداد بیت‌ها است. مثلاً برای این مثال $Bias = 7$: (Biased Exponent) نمایش نمای جابجا شده

	نمایش نما	نمایش نمای جابجا شده (+7)
+7	0111	1110
+6	0110	1101
+5	0101	1100
+4	0100	1011
+3	0011	1010
+2	0010	1001
+1	0001	1000
0	0000	0111
-1	1111	0110
-2	1110	0101
-3	1101	0100
-4	1100	0011
-5	1011	0010
-6	1010	0001
-7	1001	0000
-8	1000	1111

← هم فاصله و هم ترتیب به ترتیب و درست شده (به غیر از آخری)

در حالت کلی	نمایش نما	نمایش نمای جابجا شده
$2^{n-1}-1$		11...110
⋮		⋮
0		011...11
⋮		⋮
$-2^{n-1}+1$		00...00
-2^{n-1}		11...11

$E_{max} = 2^{e-1} - 1$ (بزرگترین نما)

e ← تعداد بیت‌های exponent

$E_{min} = -2^{e-1} + 1$ (کوچکترین نما)

11...11 ← نا عددی (Not Number) ← این نمای خاص (بخش ۱)

$*N_{max} = M_{max} \times 2^{E_{max}} = (2 - 2^{-f}) \times 2^{2^{e-1}-1}$ (2 ← مقدار ریشه)

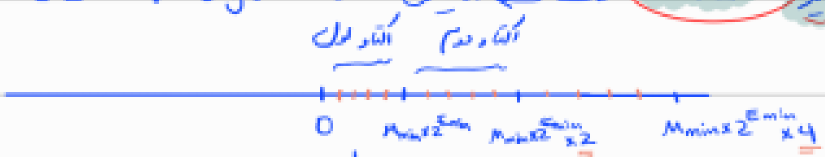
$*N_{min} = M_{min} \times 2^{E_{min}} = 1 \times 2^{-2^{e-1}+1}$

→ برای Single Precision: $N_{max} = (2 - 2^{-23}) \times 2^{2^7-1} \approx 10^{39}$ ، $N_{min} = 1 \times 2^{-2^7+1} \approx 10^{-39}$

← این یعنی اکثر کامپیوترها نمی‌توانند اعداد بزرگتر از 1×10^{39} و کوچکتر از 1×10^{-39} را نمایش دهند.

دقت: فاصله‌ی دو عدد متوالی در یک اکتاو (جایی که e ثابت است) 2^{-f} است. $1.0...0$ و $1.0...1$ → $2^{-f} \times 2^{E}$

پس کوچک‌ترین فاصله‌ی قابل تمیز $2^{-f} \times 2^{E_{min}}$ است ← در نمایش Single Precision: $2^{E_{min}-f} = 2^{-23-127} = 2^{-150}$



← فاصله‌ها در اکتاو بعدی ۲ برابر می‌شود

هرچه به 0 نزدیک‌تر می‌شویم فاصله‌ها کمتر می‌شود

* پس floating point بهتر از fixed point است چون هرچه به 0 نزدیک می‌شویم دقت نیز بیشتر می‌شود

پس خطای نسبی در floating point کمتر و دقت نسبی ثابت است

سیستم رایج در نمایش اعداد حقیقی در کامپیوتر همین نمایش است

← مزیت: دقت مطلق متغیر، دقت نسبی ثابت

⇓

ارجح است

۲ ۱ $f_2 = 110Hz$	۳ ۱ $f_3 = 220Hz$	۴ ۱ $f_4 = 440Hz$
صداها بم هستند		صداها بسیار زیر و نازک هستند

جلسه هشت ← جورچین جلسه نه ← بهرامیان

felixcloutier.com/x86/ (felix 86)

مقصد

add rax rbx ⇒ a = a + b

(application binary interface) abi ← بین متغیرها، ها register ویرازنده

b, a رجیسترهای

print "Hello World" in Assembly:

```
global asm_main
section .data
msg: db "Hello world", 10
section .text
asm_main:
  mov rax,1
  mov rdi,1
  mov rsi, msg
  mov rdx, 12
  syscall
  ret
```

محتویات فایل A.s

abi از x86-64 استفاده کردیم و برای sys.write

بیلد کردن - لینک کردن

gcc A.c asm.o -fno-pie -no-pie غیرفعال کردن دستورات امنیتی

```
void asm_main();
int main() {asm_main();}
```

File Make

```
all.out
|
run: a.out
    ./a.out
asm.o: A.s
  nasm -felf64
```

جلسه دهم ۱۴۰۳،۸،۵

غایش ممیز شناور

حداکثر خطا $Error_{max}$ — حداکثر خطا: ۱۷٫۱ ، ۱۷٫۱۵ ، ۱۷٫۲

به نصف دقت — 17.1 ± 0.05 (خطای مطلق)

خطای نسبی $= \frac{0.05}{17.1} = 0{,}29\%$ ← میزان خطای مرتکب شده نسبت به خود عدد

3	7	6	4

3764 ± 0.05

$\frac{0.05}{3764} = 0{,}0013\%$

3	7	6	4

$0.3764 \pm 5\times10^{5}$

$\frac{5\times10^{5}}{0.3764} = 0{,}013\%$

⊛ خطای نسبی در غایش ممیز شناور ، ثابت است و خطای مطلق متغیر

در سیستم غایش ممیز شناور در اعداد خیلی بزرگ خطای مطلق زیاد است و اندازه دقیق نیست ولی خطا و دقت نسبی حفظ شده است مثل اندازه‌گیری پارچه و فاصله‌ی بین ستارگان

در اعداد حقیقی ($\mathbb{R}$) بینهایت عدد داریم ولی در غایش اعداد در کامپیوتر به علت ثابت بودن تعداد ارقام (بیت) در رجیستر، ما تنها می‌توانیم تعداد ثابت و محدودی را ذخیره کنیم.

$-N_{min}$ ، 0 ، N_{min} ، $2N_{min}$ ، $4N_{min}$

هر عددی در اعداد حقیقی الزاماً یک غایش و معادل یک به یک در کامپیوتر ندارد

Machine Representable Numbers ← اعداد قابل غایش در ماشین ⇐ کامپیوتر نمی‌تواند همهٔ اعداد Real (حقیقی) را غایش دهد

پس کامپیوتر برای غایش اعداد چه می‌کند؟ کامپیوتر دائماً در حال تقریب زدن (Rounding) است و می‌گردد تا نزدیک‌ترین عدد قابل غایش به عدد را پیدا کند.

طراحی شرکت IVM به نفع بانک‌ها برای گرد کردن سود رو به بالا یا پایین

⚠ تبدیل $(0{,}1)_{10}$ به مبنای 2 ؟ ثبات محدود است پس $0{,}0\underline{0011}0011 0011\ldots$ (تناوب) ←

$0{,}1\times2=0{,}2$
$0{,}2\times2=0{,}4$
$0{,}4\times2=0{,}8$
$0{,}8\times2=1{,}6$
$0{,}6\times2=1{,}2$
$0{,}2\times2=0{,}4$

در تغییر مبنا دچار خطا شده اید! پس مثلاً کامپیوتر سود ۱۰٪ را نمی‌تواند دقیق حساب کند

Ex.

$$\begin{cases} S_1(x) = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{x^2} \\ S_2(x) = \frac{1}{x^2} + \frac{1}{(x-1)^2} + \cdots + \frac{1}{2^2} + 1 \end{cases}$$

در کمال تعجب خواهید دید که S_1 با S_2 برابر نخواهد بود و از راست به چپ با چپ به راست متفاوت خواهد بود

چرا؟ به علت خطاهای رخ داده (خطای S_1 بیشتر است چون از عدد بزرگتر شروع شده و هر چه پیش می‌رویم اعداد کوچک نادیده گرفته می‌شوند.)

خطاها : ۱ـ خطای تغییر مبنا ۲ ـ خطای گرد کردن و تقریب زدن ۳ـ خطای محاسبات

۱ استاندارد های IEEE : (حتما باشید!)

fraction / significand / mantissa

① ۳۲ بیت (Single Presicioun)

s	exponent	↑
1 bit	8 bit	23 bit

② ۶۴ بیت

s	exponent	Fraction / Mantissa / Significand
1 bit	11 bit	52 bit

$N = (-1)^S \times 1.F \times 2^{E-bios}$ ← خیلی مهم!

Ex) پیدا کنید معادل نمایش دهدهی عدد زیر که با استاندارد IEEE ۳۲ بیتی نوشته شده است.

1 10110110011 00...0 → 1 1011 0110 011 00..00

طبق استاندارد بیت اول نمایش دهنده ی علامت ، 8 بیت بعدی نمای بایس یا جابجا شده و بقیه قسمت اعشاری

پس $(-1)^1 \times 2^{10110110-(2^{e-1}-1)} \times 1.011 = (-1) \times 2^{55} \times 1.011$ (تفاوت exponent ، Fraction)

(ضمنی) Implicit one ← جایی ذخیره نمی‌شود چون همه می‌دانند مثل شما و تفنن و...

عدد هنجار شده (normalised) : عددی که سمت چپ آن 0 نباشد به همین علت 1 ضمنی را استفاده می‌کنیم

- چرا $2^{e-1}-1$ ؟ چون از 00...0 تا 11...1 داریم یعنی 2^{e-1} ولی 111...11 را برای نمایش (Not a Number) کنار گذاشته‌ایم پس $2^{e-1}-1$ داریم.

Ex) تبدیل عدد +105.625 به floating point با استاندارد IEEE ۳۲ بیتی (کار Compiler)

+105.625 → + → S=0 Step1

→ Step2 تبدیل به باینری : $+105.625 = +1101001,101 = +1,101001101 \times 2^6$ (Fraction) Step3

$E-(2^e-1)=6$

$E=133=10000101$

← 0 1000 0101 101 0011 0100 00...0

Bios ↑ نشان محاسبات

می‌گیرد یکی در میان متفاوت است یعنی یه بار به بالا و یه بار به پایین گرد کن تصمیم

17.15 → 17.2 ولی 17.25 → 17.2

17.11, 17.12, 17.13, 17.14 } → 17.10

17.16, 17.17, 17.18, 17.19 } → 17.20

تقریب زدن {
- گرد کردن (Round to nearest)
- بریدن (Truncation)
- گرد کردن به نزدیک‌ترین عدد زوج (Round to nearest even) ← استاندارد IEEE برای گرد کردن

جلسه یازدهم 1403/8/7

اکثر کامپیوترهای امروزی طبق یک برنامه از قبل ذخیره شده (stored program) مثل ماشین تلفیقی بابیج کار می‌کنند.

یکی از کامپیوترهای نسل اول ← IAS توسط آقای فون نیومن و تیمش در آزمایشگاه IAS، مبتنی بر لامپ خلاء

دستورات ۲۰ بیتی، ماشین باینری، سیستم نمایش 2's Complement، طبق ماشین تلفیقی بابیج

← چون طراحش آقای فون نیومن بوده گفتیم فون نیومن در این کامپیوتر پیاده سازی شده است

فصل سوم: ساختار داخلی پردازنده ها و جایگاه زبان اسمبلی

کامپیوتر IAS ← اولین کامپیوتر باینری لامپ خلاء

بررسی ساختار پردازنده ها:

CU به حافظه دستور یک آدرس می‌دهد و می‌پرسد چیکار کنم حافظه‌ی دستور هم

بر اساس یک کد دودویی دستور را ابلاغ می‌کند بعد CU که را decode می‌کند

و بعد فرمان را به واحد پردازش می‌دهد

بیشتر کامپیوترهای امروزی ۱ واحد کنترل و چند واحد پردازشی دارند

واحدکنترل نقش مغز سیستم را دارد که دستورات را از حافظه می‌خواند، ترجمه می‌کند (decode) و دستور را به واحد پردازشی ارسال می‌کند.

امروزه پردازنده ها بیش از یک هسته‌ی محاسباتی (Core) دارند ← دوران کثرت واحدهای پردازشی

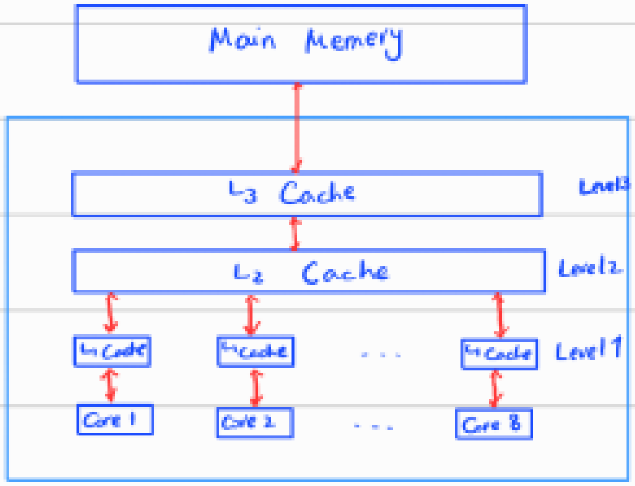
* در کامپیوتر IAS حافظه دستور و داده ادغام بودند ← معماری پرینستون

امروزه حافظه دستور و داده از بیرون کیس (RAM) و در داخل پردازنده جدا است { Instruction Cache / Data Cache }

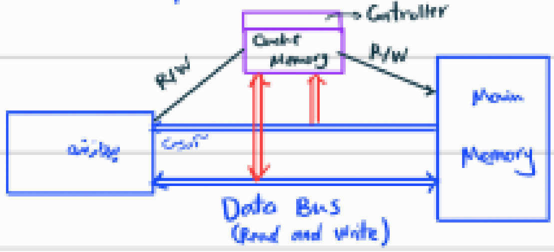
Cache (حافظه نهان): به دور از دسترس و پنهان از کاربر - کوچک - پرسرعت - گران - ذخیره داده‌های پرکاربرد در نزد پردازنده

نگهداری داده های پرمصرف و یا داده هایی که احتمال میرود در آینده‌ای نزدیک از آن استفاده کنیم

پردازنده های پیشرفته امروزی
که تعدد هسته محاسباتی



* تمام کامپیوترها با آدرس کار می‌کنند — نوشتن داده، خواندن داده همه با آدرس دهی انجام می‌شوند



معیارهای مقایسه و ویژگی کامپیوتر: Performance (کارایی و سرعت) – قیمت – توان مصرفی – مسائل زیست محیطی – حجم و وزن – دقت و بازه نمایش

* اکثر حافظه‌های مورد استفاده کنونی از یک آدرس می‌گیرند ← اهمیت مدهای آدرس دهی (چگونگی استفاده پردازنده از حافظه)
- مطلق
- نسبی
- بر اساس انتهای دیگر
⋮
۱۲ تا ۱۶ مد آدرس دهی

حافظه Cache: پردازنده رسماً حافظه نهان را نمی‌بیند ولی حافظه نهان آنچه از گذرگاه‌ها عبور می‌کند را می‌بیند و به محض اینکه ببیند

داده‌ای را که پردازنده می‌خواهد را دارد، آن را سریعاً به پردازنده تحویل می‌دهد ← افزایش سرعت دسترسی به حافظه

ذخیره‌ی داده‌های پرمصرف، پرتکرار و احتمال استفاده در آینده

فرمول حافظه نهان: $t_{av} = h\,t_{cache} + (1-h)\,t_{MM}$ →
- t_{MM} = Main Memory زمان دسترسی به اطلاعات
- t_{cache} = Cache زمان دسترسی به اطلاعات
- h = hit Ratio نرخ بود / برخورد ← احتمال بودن داده در Cache
- $m = 1 - h$ = miss Ratio نرخ نبود / فقدان

Ex) $t_{MM} = 10ns$, $t_{cache} = 1ns$, $h = 95\%$

$$t_{av} = \frac{95}{100} \times 1ns + \frac{5}{100} \times 10ns = 0.95 + 0.5 = 1.95 ns$$

$$* \; S_{تسریع} = \frac{t_{MM}}{t_{av}} = \frac{10ns}{1.95ns} \simeq 6.9$$

با اضافه کردن یک حافظه کوچک، سرعت دسترسی به داده‌ها ۷ برابر شده است

نحوه مراجعه‌ی برنامه‌های امروزی به حافظه ← بهره بردن از اصل "مجاورت مراجعات" Principle of locality of refrence

● مجاورت مکانی: نزدیکی و مجاورت داده‌هایی که در آینده به سراغشان خواهیم رفت (مثل توالی دستورها و بردارها و اعضای آرایه)

خواندن یک بلوک از داده‌ها هنگامی که حافظه نهان به دنبال داده‌ای در حافظه می‌رود که آن را نداشته است. یک

بلوک شامل آن داده و داده‌های اطرافش را از حافظه اصلی برمی‌دارد (به دلیل احتمال استفاده بعدی داده‌های اطرافش h)

● مجاورت زمانی: اگر کاربر الان سراغ داده‌ای بیاید به احتمال زیاد در آینده‌ی نزدیک نیز سراغ آن خواهد رفت پس آن را در حافظه نهان نگه می‌داریم.

مثل tab در Windows که وقتی یک پنجره را حفظ می‌کند چون محتمل است که در آینده‌ی نزدیک سراغ آن بیایم و باید سریع‌تر باز شوند در حافظه سریع‌تر هستند

ایراد Cache ← همسان‌سازی داده‌ها برای داده‌های مشترک در چند حافظه ← زمانبر بودن الگوریتم همسان‌سازی

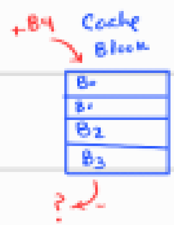
← گاهی آخرین اطلاعات یک داده توسط Cache به Main Memory داده نمی‌شود چه تا زمانیکه Cache بداند کسی دیگری به آن داده احتیاج ندارد
آن را به MM نمی‌فرستد چون زمانبر است
بعداً ۲ حالت می‌شود ← یا دیگر نخواهد و حذف کند
یا کسی دیگری x را بخواهد که از همان Cache باید بگیرد

سیاست جایگزینی Replacement algorithm / policy ← کدام اطلاعات از Cache پاک شوند؟

✦ سیاست‌ها: - Random

کدام محصول را تخفیف بدهیم؟

- FIFO (First in First out) اولی خارج شود ← محصولات کهنه و قدیمی
- LIFO (Last in Last out) آخری خارج شود ← مصرفی بیاید و محصول جدید
- LRU (Least Recently Used) اخیرا استفاده نشده ← افزایش خرید محصول *
- MRU (Most Recently Used) اخیرا زیاد استفاده شده (خرید بوده دیگر؟)
- LFU (Least Frequently Used) کمترین تعداد استفاده شده (به صورتی)
- MFU (Most Frequently Used) بیشترین تعداد استفاده شده
- OPT (Optimal) بهینه: آن بلوکی را جایگزین کنید یا دور بریزید که در آینده‌ای بعید ممکن است دورتر از بقیه به آن نیاز داشته باشیم.



ایده‌ی Cache در پیتزا فروشی ← کاهش زمان آماده شدن غذا ← با تحلیل رفتار کاربران و profiling مثل McDonald

جلسه دوازدهم ۱۴۰۳, ۸, ۱۲

اصل های بنیادی محاسبات کامپیوتری: ① ...

② وجود توالی منطقی در دستورات و اجرای کدها ← مانند ترانزیستورها از بالا به پایین و خط به خط

③ تمام مراجعات به حافظه باید با ذکر آدرس انجام شود

کامپیوتر متشکل از پردازنده، حافظه و دستگاه I/O است

Input/output for Computers

keyboard, mouse, microphone, webcam, scanner : Input devices

Monitor, printer, plater : Output devices

USB port, Network Card, Hard Disc : I/O devices

digitizer (موقعیت ساز) ← وارد کردن نقطه نقطه نقشه با کلیک کردن برای کاربردهای 2D, 3D

مزیت: امکان وارد کردن مختصات یک قسمت نقطه به نقطه ← طراحی و شبیه سازی

کامپیوتر بدون ورودی و خروجی نداریم ← حداقل ۱ ورودی یا ۱ خروجی

← تردد شمار

← زمان سنج

← یعنی stored program است

حافظه دارای ۲ بخش: برنامه، داده — اگر بتوان به صورت فیزیکی جدا کند سرعت افزایش می یابد

پردازنده: انجام محاسبات

مراجعه ی پردازنده و I/O به حافظه بدون آدرس به هیچ عنوان نمی شود. آدرسی که نهایتا به حافظه عرضه می شود آدرس معتبر (effective Address) است.

در مدل فون نیومن ابتدا به سراغ حافظه می رویم تا دستورالعمل را بخوانیم ← چه کسی آدرس اولیه را تعیین می کند؟ PC (Program Counter)

PC (Program Counter) یا IP Register (Instruction Pointer) ← نقش کلیدی

در مدل فون نیومن ابتدا PC آدرس را به حافظه عرضه می کند و به حافظه فرمان Read می دهد و می گوید دستورالعمل این آدرس را به من بده. حافظه در پاسخ دستورالعمل را برای CU می فرستد، بعد ترجمه دستور صورت می گیرد سپس خواندن عملوندها،

سپس اجرای دستور سپس جای نویسی نتایج و نهایتا مراجعه به مرحله اول و به روز رسانی PC

در مرحله بعد به آدرس PC+n می رود (n بایت) معمولا ۴

نحوه I/O در مدل فون نیومن: مثلا فرستادن فایل به چاپگر برای چاپ

- دادن یک آدرس به دستگاه های I/O از قبل مثلا تعریف آدرس 3F0 به عنوان پرینتر

- ارسال سیگنال بین کامپیوتر و دستگاه خارجی برای چک کردن ← Hand Shaking ← یک پروتکل ارتباطی

که به دلیل تفاوت سرعت کامپیوتر و دستگاه

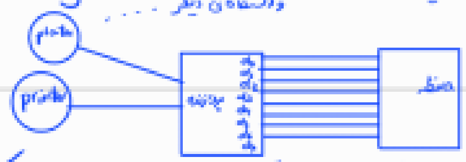
مهم ترین یکی از راه های ارتباط با I/O آدرس دهی می باشد ← وجود خطوط کنترلی علاوه بر Data Bus و Address Bus

مثلا صدا زدن I/O و گشتن کامپیوتر

راه‌های ارتباط سرکشی

Interapt Face , Pulling

پس برخلاف حافظه تنها راه ارتباط با I/O آدرس دهی نیست.



Bus (گذرگاه، مسیر ارتباطی)

بین دستگاه‌ها حتی یک سیم و فیبر نوری باید باشد ولی بی سیم نیست چون تحت تاثیر عوامل محیطی ممکن است خراب شود یا نویز دار کند

معمولا ارتباطات سیمی هستند ← مسی / نقره‌ای
← طلایی: رسانایی بالا و نرخ انتقال زیاد در سرعت های بالا

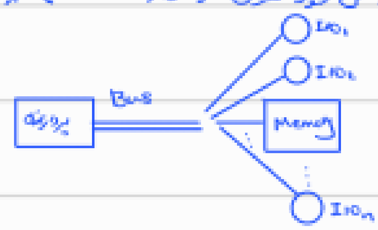
اگر پردازنده کلمات 8 یا 16 یا 32 بیتی از حافظه بخواند باید به همین تعداد سیم بین حافظه و پردازنده باشد

چرا آن را Bus می‌نامند؟ (اتوبوس درستی در نظر گرفته می‌شود)

برای برقراری ارتباط مستقیم بین مثلا n دستگاه و پردازنده به n ارتباط سیم بین آنها نیاز داریم ← خیلی عیب و پیچیده

← یک گراف کامل با $\frac{n(n-1)}{2}$ یال (نودها ← دستگاه‌ها و پردازنده، یال‌ها ← سیم و ارتباط) ایجاد کنیم و اگر تعداد بیت هم w باشد (مثلا ۱۶، ۳۲، ...)

باید این عدد را در تعداد ارتباطات ضرب کنیم که می‌شود عرض Bus (w (width)) که خیلی عدد بزرگی می‌شود ← سیم کشی زیاد



همه را به یک گذرگاه مشترک به اسم Bus وصل می‌کنیم ← مزیت: کاهش سیم کشی
← کاستی: سرعت پایین ← در هر لحظه تنها یک تراکنش (transaction) روی گذرگاه قابل انجام است.

Bus: یک روش ساده و ارزان قیمت که کمترین میزان سیم کشی را مصرف می‌کند خیلی استفاده شده (به علت ارزانی) ـ

در آن واحد تنها 1 عنصر می‌تواند صحبت کند مگر در حالت Broad Cast که پردازنده به همه پیام بدهد

چرا Bus (اتوبوس درستی)؟ هر سری داده‌ها سوار بر آن به مقصد می‌روند و بعد داده‌های دیگر

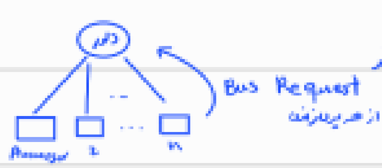
مالک و مدیر و مسئول گذرگاه همیشه پردازنده است پس همواره می‌داند که گذرگاه در اختیار کیست

در سیستم های چند پردازنده گاهی اوقات Bus خالی نمی‌شود ولی دستگاه I/O نیاز دارد در این زمان I/O یک درخواست DMA (Direct Memory Access)

می‌دهد و از پردازنده می‌خواهد که Bus را به آن بدهد مثلا برای سریع‌تر شدن اطلاعات یا برقراری ارتباط سریع با دستگاه‌های دیگر

Single Master ←

اگر گذرگاه چند مالک داشته باشد یک داور (Arbitrator) سر راه می‌گذارند



اگر داور اجازه دهد به هر Processor ، Bus Grant می‌دهد.

گذرگاه ← ساده‌ترین و مرسوم‌ترین روش برای برقراری ارتباط بین اجزای کامپیوتر

* حسن: قیمت کم، سادگی پیاده سازی

* اشکال: در آن واحد تنها یک نفر می‌تواند به یک نفر دیگر اطلاعات بفرستد که سرعت را کم می‌کند

اگر عرض Bus و تعداد سیم‌ها زیاد باشد در فرکانس‌های بالا خوب کار نمی‌کند

مدهای آدرس دهی

برای تعیین و پیدا کردن operand و آدرس آن به کار می‌رود (operand ← انجام operation روی آن)

که یا در ثبات یا در حافظه قرار دارند

حافظه شامل داده‌ها و دستورات ماست و ما فقط با آدرس می‌توانیم به سراغ حافظه برویم پس باید آدرس را تولید کنیم

آدرس دهی:
- آدرس دهی مستقیم direct addressing
- آدرس دهی غیرمستقیم indirect addressing
- آدرس دهی ضمنی Impliing addressing
- آدرس دهی نسبی Relative addressing
- آدرس دهی آنی Immediate Addressing

- آدرس دهی ضمنی: آدرس را به طور مستقیم نمی‌گوییم بلکه به آن اشاره می‌کنیم و پردازنده آن را می‌شناسد

CLEAR AX → محتوای ثبات AX را 0 می‌کند

نیازی نیست حتماً CLEAR , INCREA آدرس A را به آن بدهیم

INCREMENT AX → ثبات AX را با 1 جمع می‌کند

پس یک سری از دستورات هستند که بدون ذکر آدرس یا operand انجام می‌شوند و پردازنده خودش آن‌ها را پیدا می‌کند

ISA (Instruction Set Architecture):
- تعریف و توصیف مجموعه دستورالعمل‌ها
- مدهای آدرس دهی
- تعیین ثبات‌های قابل استفاده
- گونه‌های داده‌ی قابل استفاده

هر کسی که پردازنده‌ای می‌سازد باید این 4 فاکتور را هم معرفی کند. در پردازنده‌های CISC مدهای آدرس‌دهی و تعداد دستورات آن‌ها خیلی زیاد،

تعداد نزدیک به 1000 حالت می‌شود ولی پردازنده‌های RISC هم تعداد مدهای آدرس‌دهی (3 تا 4 تا) و هم تعداد دستورات خیلی کمتری دارند

تعداد ثبات‌های RISC بیشتر است (معمولاً 32 تا، 64 تا) ولی ثبات‌های CISC معمولاً کمتر است (8 تا 16 تا)

* مدهای آدرس دهی ← تعیین کل دسترسی به عملوندها و آدرس آن؟ در حافظه یا ثبات

- آدرس دهی آنی Immediate

Add AX , 2 ⟸ AX ← AX + 2

عملوندی است که به طور آنی در اختیار دستور قرار گرفته است و داخل کد دستور نوشته شده است

هر دستورالعمل یک OP Code دارد

در خود دستور مقدار داده ذکر می‌شود پس آدرس دهی آنی است

عملوند در خود دستورالعمل (OP Code) قرار دارد ← * حسن

اگر عدد خیلی بزرگ باشد از دستور بیرون می‌زند (محدودیت Operand داریم) ← * کاستی

برای اعداد ثابت و کوچک آدرس دهی آنی خوب است

Syntax 68000 Processor → جهت انتقال از چپ به راست (استاتیک)

MOV D0, A3400 → مقدار D0 را در خانه ی 3400 حافظه می‌ریزد

- آدرس دهی مستقیم

تعیین آدرس، سپس نوشتن و خواندن در آن آدرس

آدرس به طور مستقیم در دستور ذکر می‌شود گاهی هم absolute یعنی آدرس دهی مطلق

- آدرس دهی غیر مستقیم

MOV AX, [BX] رفتن به محل ذکر شده در حافظه و پیدا کردن آدرس مقصد در آنجا

کاربرد: گاهی در لحظه آدرس مقصد را نداریم و بعدا یا تعیین می‌شود یا مثلا توسط یک تابع تولید می‌شود

- آدرس دهی نسبی برای پرش های شرطی

رفتن به n خانه‌ی بعدی آدرس کنونی

فاصله
↕
AX+9 مثلا در بیشتر پرش های شرطی به جای قرار دادن آدرس کنونی و بعدی، فاصله این دو را در کد قرار می‌دهند، چرا؟

این که در هر جای حافظه می‌توانند قرار بگیرند و عاملا relocatable است

وقتی آدرس دهی دستورات نسبی (و نه مطلق!) باشد می‌توانند در هر جای حافظه قرار بگیرند و اجرا شوند

جلسه سیزدهم 1403,8,14 — شرکت چینی و جابجا چینی یکی نیست

the intel microprocessor
By Prentice → Reference

شناخت یک پردازنده از ISA ← واسط یک پردازنده توسط سازنده پردازنده. البته در زبان های سطح بالا Compiler این کار را می‌کند

زبان مشترک بین سازنده پردازنده و افرادی که در سطح پایین برنامه می‌نویسند یا افرادی که Compiler تولید می‌کنند

سبک برنامه نویسی و انتظارات یک پردازنده ← ISA

ISA شامل:
- فهرست مجموعه دستورات (Instruction Set) مثلاً INC, DEC, Sub, Add, MOV — در Assembly دستورات 3 یا 4 حرفی هست
  SHL AX,1 ⟷ AX×2 or AX«1 (SHL: shift left)
- شیوه کدگذاری دستورات
- مد های آدرس دهی
- گونه های داده
- مجموعه ثبات های قابل استفاده برنامه‌نویس ← پردازنده پنتیوم 8 ثبات قابل دسترسی و 90 ثبات پنهان دارد ← برای انجام کار ها بابت داده‌ها و نگهداری نتایج موقت
- شیوه آدرس حافظه و I/O

همه ISA آن چیزی است که در پردازنده تهیه می‌شود. پس ممکن است پردازنده شامل چیزهای کمتری نیز باشد مثل ثبات های قابل استفاده و پنهان

چرا Assembly؟ — Code Optimization و افزایش Performance — Debug پردازنده — مهندسی معکوس: رفع قفل نرم‌افزار
- قفل دستگاه های صنعتی، (Zero day Vulnerability, Backdoor)

مدهای آدرس دهی Addressing Modes

- Register — آدرس دهی فقط با ثبات ها با مشخص کردن ثبات مبدا، مقصد و عملوند — mul A,B,X ⟷ A = B.X

* انواع دستورات در تعداد عملوند:
- دستورات بدون عملوند (Zero-Address) ← stack-machine — برداشتن از رأس پشته و قرار دادن در رأس پشته — add
- دستورات تک عملوند (1-Address) ← Accumulator based — add R_1 ⟺ ACC ← R_1 + ACC ← یک رجیستر پیش‌فرض با اسم انباشتگر (Accumulator)
- دستورات دو عملوندی (2-Address) — add R_1, R_3 ⟺ R_1 ← R_1 + R_3
- دستورات سه عملوندی (3-Address) — mul R_1, R_3, R_5 ⟺ R_1 ← R_3 × R_5

تمام 4 عمل اصلی به حداقل 3 عملوند احتیاج دارند. پس چرا پردازنده‌ای باید طراحی شود که مثلاً دستورات 2 عملوندی دارد و ما باید 2 بار استفاده کنیم ← مثل Intel

با پردازنده های بدون عملوند ← تحت اجرا ساده‌تر، تولید آسان‌تر، خطاهای کمتر و سرعت بیشتر — Simpler make faster!

زبان سطح بالا: C ... → Compiler (ترجمه) → ASM → کد باینری

نمایش — Post fix ← کار با پشته
Pre fix

$x = (a + \frac{b}{c}) - d \cdot e$ → prefix

a
b
c
div
add
d
e
mul
sub
→ post fix

- مد آدرس دهی register ← از همه ساده‌تر و سریع‌تر ← درگیر پردازنده ، ثبات‌ها

Add AX,BX ⟺ AX ← AX+BX
(ثبات مقصد)

- استفاده از مد آدرس دهی ثباتی به بهبود سرعت کمک می‌کند.

جهت اطلاع : در کامپیوترهای امروزی قسمت کار با اعداد ممیز شناور (floating Point Unit) ، Stack Processor است برای افزایش سرعت

Refrence ← XB6 Assembly Language and C Fundamentals

INC BH ⟺ Increment BH ⟺ BH ← BH+1 — Based_plus_index

(INC: instruction, BH: operands)

MOV AX,BX ⟺ AX ← BX

(68000 پردازنده) MOVE.W → WORD → 16 bit MOVE.B → Byte MOVE.DB → Double Byte

(جهت ورود پردازنده‌های از راست به چپ)

Sub EX,EX ⟺ EX ← EX-EX → یکی از راه‌های 0 کردن

Xor EX,EX ⟺ EX ← EX xor EX → یکی دیگر از راه‌های 0 کردن

CLEAR EX ⟺ EX ← 0 → " " "

} ← پرسرعت هستند نسبت به MOV EX,0

به علت کیفیت و علاقه به 0 یک رجیستر را به 0 اختصاص داده‌اند و مقدار آن را در رجیستر ذخیره کرده‌اند ← R0

- Immidiate Addressing آدرس دهی آنی ← ذکر مقدار operand به طور آنی در دستور

ADD EAX,3

MOV AX,302 (قرار دادن نمایش هگزادسیمال)

MOV EAX,0AEBCFFFFH → EAX قرار دادن عدد هگزادسیمال ۳۲ بیتی در رجیستر

MOV AX, 1111 1111 1101 1000 ⟷ MOV AX,-40 → Assembler این کد باینری را تولید می‌کند

- اگر چیزی ذکر نشود مقدار decimal در نظر گرفته خواهد شد

- Direct Memory Addressing ← قرار دادن آدرس به صورت مستقیم در دستور

MOV AL,[7A33D4H] (H: shows Hexa decimal)

قرار دادن مقدار داخل این آدرس در ثبات AL

(مقدار آدرس این خانه)

INC DWORD PTR [17H] → اضافه کردن 1 به مقدار آدرس 17H در خود حافظه ← وقت انجام عملیات پنهان ثبات داخلی اختصاص

(DWORD: Double Word, 32 bit)

- Base or Register Indirect Memory Addressing

MOV ECX,[EBX]

(نقش پوینتر)

MOV AX,[BH] → Base Addressing Mode

← انتقال مقدار محل اشاره شده توسط BH به AX

MOV AX,BX → Register Mode

← انتقال مقدار BX به AX

CMP AL,[BX+4]

مقدار ۴ خانه جلوتر از آدرس مقدار رجیستر BH

جلسه چهاردهم 1403,8,19

بنیاد تئوری محاسبات کامپیوتری فرض بر قابل آدرس دهی بودن حافظه است.

آدرس دهی : چگونه داده های مورد نیاز را در حافظه یا نزد پردازنده داخل ثبات ها بیابیم.

(حافظه) ← شامل محتوا و آدرس / شروع از آدرس کم به زیاد

محتوا ، آدرس ← نمایش به صورت هگزادسیمال به علت فشرده بودن آدرس ها

0, 1, 2, ... FFFF ← چقدر حافظه دارد؟

$16^4 = 2^{16} = 65536$ Byte
$= 64$ KB (کیلو بایت)

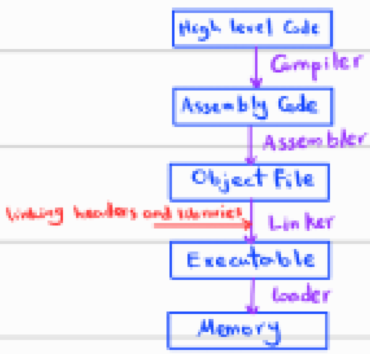
حافظه های امروزی: FFFFFFFF ، 4 GB

$16^8 = 2^{32} = 4 \times 2^{30}$ Byte $= 4$ GB

* $2^{10} = 1$ KB , $2^{20} = 1$ MB , $2^{30} = 1$ GB

چرا آدرس دهی می کنیم؟ چون حافظه و رجیسترها شماره گذاری شده اند و ما جای دیگری نداریم.

Digital Design and Computer Architecture page 329



نحوه اجرای کد نوشته شده با زبان High-level

چرا زبان High level ابتدا به زبان Assembly تبدیل می شود؟
به علت ISA و تبدیل برنامه به زبان قابل فهم پردازنده چون زبان هر پردازنده متفاوت است ،
ISA زبان مشترک بین پردازنده و برنامه نویس یا Compiler ترین است.

چرا زبان های پردازنده ها متفاوت است؟ دلایل تجاری از سوی شرکت ها

Compiler که سطح بالا را به زبان ماشین و قابل فهم برای پردازنده ترجمه می کند.

loader برنامه ها و داده ها را در حافظه می ریزد و به پردازنده می گوید شروع کند.

برای هر زبان چند Compiler داریم چرا؟ سلایق نویسنده ها متفاوت است. یک سری از آنها تمام دستورات ISA را یاد ندارند، بسته به نیازها متفاوت است. بعضی ها کدهای کوتاه تری تولید می کنند. حافظه کمتری اشغال می کنند. تمیزتر هستند و قابلیت های بیشتری دارند.

- بررسی تفاوت های NASM, MASM, EMU8086? کدام راحت تر ، eficient تر یا سریع تر است؟

the intel microprocessor - 8th edition - by prentice - Page 79

مدهای آدرس دهی Intel

- Register مد آدرس دهی ثباتی MOV AX, BX → هیچ آدرسی ذکر نشده! اسامی ثبات ها، اسامی همان آدرس آن ها است
اعداد داخلی به شماره تبدیل می شوند

* ARM : Advanced RISC Machine ماشین RISC پیشرفته ، فروش license پردازنده ARM برخلاف Intel / چرا این شرکت ها قابل رقابت اند؟

- Immediate آدرس دهی آنی ← ذکر مقدار operand در دستور MOV CH, 3AH

- Direct آدرس دهی مستقیم MOV [1234H], AX → قرار دادن مقدار AX در آدرس 1234H حافظه

- Register indirect   MOV [BH],CX   آدرس مقصد توسط مقدار رجیستر BH مورد اشاره قرار گرفته است
پس ابتدا باید مقدار BH را بخوانیم و بعد به آن آدرس برویم

- Base_Plus_Index   (آدرس‌دهی عادی) آدرس‌دهی پایه به علاوه خاص   MOV [BH+SI], BP   شروع از آدرس BH و SI تا جلو رفتن
به طور کلی دسترسی به اعضای آرایه‌ها

هر کدام از درایه‌ها 4 byte است

در حافظه آدرس CS کم در بالا قرار دارند

بستگی به قرار داد دارد

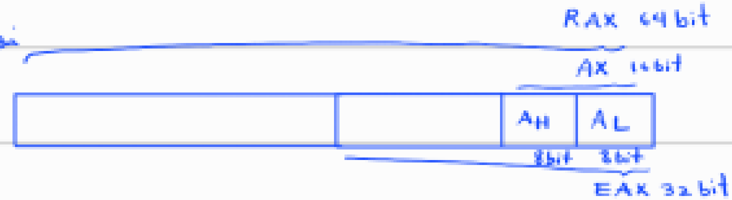
رقم کمتر در آدرس کمتر

فرض می‌کنیم حافظه Byte Addressable است یعنی در هر بایت حافظه یک آدرس قرار می‌گیرد

- Register Relative   MOV CL, [BH+4]

- Base Relative_Plus_Index

- Scaled Index → خودش با توجه به مقیاس تعیین شده جلو می‌رود



Page 82

MOV AH, DL → 8 bit
MOV AX, BX → 16 bit   ← از کجا بفهمیم چند بیت انتقال می‌دهد؟ اسم ثبات‌ها!
MOV EBX, EAX → 32 bit
MOV RAX, RDX → 64 bit

OPCode برای این MOV ها متفاوت است چون تعداد بیت آنها متفاوت است.

در زبان C تایپ Register داریم مثلاً register int a که کامپایلر متغیر را سعی می‌کند در رجیستر ذخیره کند چون تکثیر استفاده از آن باید خیلی سریع باشد.

Page 84

دیتی که تولید شده خروجی توسط اسمبلر:

علامت‌گذاری در اسمبلی ← شروع با ;

stored program مدل

Instruction Memory → جایی که fetch می‌شود

اسمبلر هم آدرس حافظه و هم OPCode را می‌دهد!

Output of Assembler		Assembly Code
0000	8B C3 → OP Code	MOV AX, BX
0002	8A CE	MOV CL, DH
0004	66 | 8B C3	MOV RAX, RBX
0007		

آدرسی که برنامه ما در حافظه پیدا خواهد شد و اگر استفاده کنیم از 0000 شروع کرد تمام دستورات بر کجا قرار خواهند گرفت

۳ بایت رفت جلو ← دستور ۳ بایت بود

برای دستورات 64 bit

آدرس حافظه که دستورات در آن قرار گرفته اند

⚠ طول دستورات 80x86 متغیر است ۲ بایت، ۳ بایت و ...

نحوه کدگذاری دستورات در 80x86 ← شکل که هستند دستورات پیچیده ۱ بایت، کمتر مورد استفاده ۱۲ بایت و ۱۳ بایت و ...

Page 84 → Immediate Addressing

در حالت Immediate که چیزی ننویسیم decimal ملاحظه می‌شود و اگر در آخر آن H قرار دهیم هگزادسیمال ملاحظه می‌شود.

جلسه پانزدهم ۱۴۰۳/۸/۲۱

in ARM:

Add t, b, c جمع t ← b + c
Sub a, t, d جمع a ← t − d

} → C Code: a = (b + c) − d

که یعنی هر خط در دستور سطح بالا ممکن است به چند خط دستور در اسمبلی تبدیل شود

Assembly ARM → 3 Operands → دستورهای کاملا فرضی

Assembly 8086 → 2 Operands

تعداد ثبات های ARM به مراتب بیشتر از دستورات 8086 است.

مثال از سینتکس ARM:

MOV R2, #0 → Immediate Operand = equal to immediate addressing

(Load)
LDR R3, [R2, #8] → جمع می کند سپس مقدار خانه ای حافظه ای که اشاره شده را در R3 قرار می دهد R2 + 8
(بعدا علامت # عدد [illegible])

(Store)
STR R7, [R5, #0x57] → قرار دادن مقدار R7 در آدرس R5 + 57 حافظه
(علامت عدد هگزادسیمال)

* اگر پردازنده ARM → در دستور STR برخلاف تمام دستورات جهت انتقال از چپ به راست است!

پردازنده 68000 → خیلی شبیه به پردازنده های اصلی مثل PowerPC * جهت حرکت از چپ به راست!

Addressing modes: summary

→ مدهای آدرس دهی پردازنده 68000

- Register Direct
  - Data #1
  - Address #2
- Register Indirect
  - Address #3
  - Address with Postincrement #4
  - Address with Predecrement #5
  - Address with Displacement #6
- Address Register Indirect with Index
  - 8-Bit displacement #7
  - Base displacement #8
- Memory indirect
  - Postindexed #9
  - Preindexed #10
- Program Counter Indirect
  - with Displacement #11
- Program Counter Indirect with Index
  - 8-Bit displacement #12
  - Base displacement #13
- Program Counter Memory Indirect
  - Postindexed #14
  - Preindexed #15
- Absolute Data Addressing
  - Short #16
  - Long #17
- Immediate #18

(برای #4 و #5:) ← عجیب!

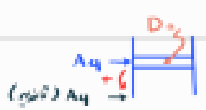
(A4) شبیه به [ ] در 8086

MOV D0, (A4)+ → قرار دادن مقدار D0 در خانه ای که A4 به آن اشاره دارد در حافظه (Postincrement)

سپس یا خانه جلو می رود (A4 بسته به نوع D0 متفاوت است Integer → 2 byte, Word → 4 byte)

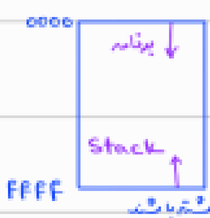
آدرس دهی Postincrement (پسا افزایشی)

که یعنی بعد از انجام کار، آدرس بسته به نوع افزایش یابد



معادل در C:
*A4 = D0
*A4++

حافظه



در پردازنده 68000 دستور PUSH نداریم. دستور Postincrement به چه دردی خواهد خورد؟

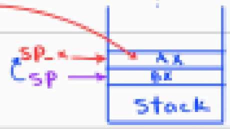
پشته از آخر شروع به پر شدن می کند و برای ذخیره داده های موقت و یا مسیرهای بازگشتی استفاده می شود

ولی برنامه از ابتدا شروع به پر شدن می کند ← برای اینکه حافظه برنامه اصلی و Stack خیلی دیر به هم برسند و با هم تلاقی داشته باشند

یک ثبات SP (Stack Pointer) داریم که همواره به راس پشته اشاره می کند

قرار دادن داده در پشته یا همان Push

PUSH AX



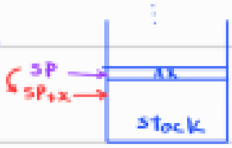
- ابتدا از مقدار SP کم می شود سپس AX در آدرس جدید قرار می گیرد

80x86

```
PUSH AX => { DEC SP
             DEC SP
             MOV [SP], AX
```

مقدار کم شدن از SP به تایپ AX بستگی دارد مثلا اگر AX Integer باشد ۲ تا از SP کم می شود.

POP AX



- ابتدا AX را از جایی مورد اشاره توسط SP برمی دارد سپس بسته به تایپ AX به SP اضافه می شود.

80x86

```
POP AX => { MOV AX, [SP]
            INC SP
            INC SP
```

پیاده سازی Push و Pop در 68000

```
{ PUSH D4 <=> move D4, -(A7)    (Pre Increment) -> -(An)
{ POP  D4 <=> move (A7)+, D4    (Post Increment) -> (An)+
```

- چون دستور PUSH، POP در motorola (شرکت سازنده 68000) نداریم می توانیم با Preincrement, Postincrement پیاده سازی کنیم ولی در Intel Post, Pre increment نداریم ولی Push و Pop داریم!

پردازنده ARM: ثابت ۰ ــ X31 * ثابت ۰ به علت پرکاربرد بودن به صورت built-in داخل پردازنده ها قرار می گیرد که چرا داخل ثابت؟ چون مراجعه به ثابت بسیار سریع تر از مراجعه به حافظه است.

→ ثبات های ARM بسیار بیشتر از 80x86 است

دستورات در پردازنده ARM (3-operand):

```
              ADD  R0, R1, R2   <=>  R0 <- R1 + R2
(Immediate)   ADDI R0, R1, #5   <=>  R0 <- R1 + 5
              AND  R0, R1, R2   <=>  R0 <- R1 & R2
              ANDI R0, R1, #7   <=>  R0 <- R1 & 7
              LSL  R0, R1, #4   <=>  R0 <- R1 << 4   or   R0 <- R1 x 16
(Logical Shift Left)
```

(چرا از ضرب استفاده نکنیم؟ Performance - سرعت شیفت دادن بیشتر از ضرب است)

```
if (a == b)
    a = a + 3
else
    b = b + 7
c = a + b
```

```
Branch if Not Equal  <- اگر مساوی نبودند پرش کن
CMP X0, X1
B.NE ELSEIF          -> Label <- اگر برابر نبودند پرش کن به این برچسب
ADD Immediate <- ADDI X0, X0, #3
Jump یا Branch <- B DONE
ELSEIF:
    ADDI X1, X1, #7
DONE:
    ADD X2, X0, X1
```

به در CMP X0, X1، X1 از X0 کم می شود و اگر حاصل ۰ شود، Flag Zero، ۱ می شود.

```
1000           B.NE  ELSEIF
1004           AddI  X0, X0, #3
1008           B DONE
100C  ELSEIF   AddI  X1, X1, #7
1010  DONE     Add   X2, X1, X0
```

اسمبلر هنگام ترجمه همه چیز را به کد باینری تبدیل و آدرس دهی می کند پس چگونه نام ELSEIF را آدرس دهی کند؟

آیا آدرس مطلق ذکر می شود یا نسبی؟

برای دستورات پرش اغلب آدرس دهی مطلق استفاده نمی‌شود بلکه آدرس دهی نسبی استفاده می‌شود!

الگوریتم فرار نیومن

پس از اجرای خط اول در آدرس 1000، PC (Program Counter) به دستور بعدی در آدرس 1004 اشاره خواهد کرد اما آدرس ELSEIF

خانه‌ی 100C است که تا خانه‌ی 1000 12 فاصله دارد اما PC به آدرس 1004 اشاره می‌کند پس فاصله‌ی دستور بعدی تا PC، 8 است (نه 12)

پس فاصله نسبت به محل PC محاسبه می‌شود! ← آدرس دهی نسبی

! با آدرس دهی نسبی برنامه‌ها به هیچ آدرس خاصی وابسته نخواهند بود و در جای حافظه قابل اجرا خواهند بود ← relocatable

جلسه شانزدهم ۱۴۰۳/۸/۲۷

ISA : - جنبه دستورات پردازنده - ثابت های قابل استفاده - مدهای آدرس دهی - قالب دستوری - گونه داده های قابل استفاده

قالب دستورات (Instruction Format)

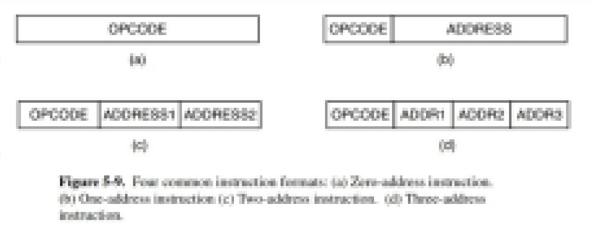


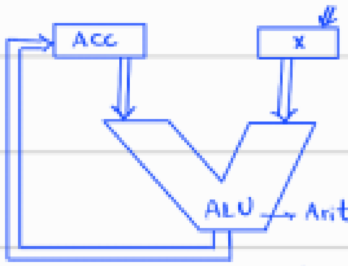
Figure 5-9. Four common instruction formats: (a) Zero-address instruction. (b) One-address instruction. (c) Two-address instruction. (d) Three-address instruction.

↳ Structured Computer Organization By Tanen Baum, Page 387

OPCode ← تنها شامل کد عملیات و بدون عملوند operand مثال - Reset ← راه اندازی مجدد پردازنده یا که
- Halt ← قطع کردن کلیه ی کارهای پردازنده
پردازنده های Zero-Address ← پردازنده های پشته ای (Stack) - Push
- (Increment) INC

| OPCode | Address | ← شامل یک operand و یک Address مثلا - Add X ACC ← X + ACC مثل جمع مبلغ حساب بانکی

پردازنده های One-Address ← جنبه برانبار شده

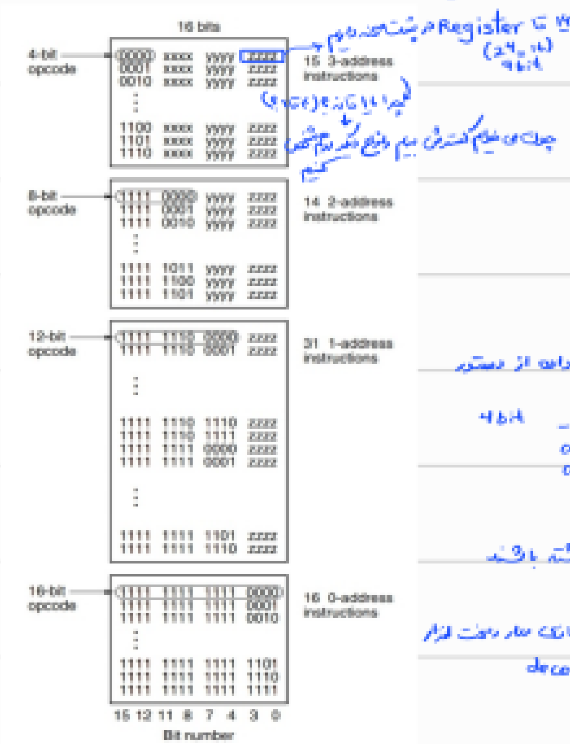


ALU → Arithmetic Logical Unit ← واحد حسابی (ریاضی) منطقی ... می توان با آن محاسبات سنگین مانند tan را انجام داد

↳ واحد انجام کارهای ریاضی پیشرفته ← Arithmetic Coprocessor / FU / FPU (Floating Processor Unit)

اگر کامپیوتری این بخش را نداشته باشد نمی تواند محاسبات پیچیده سنگین مانند log ، exp ، tan و ... را نیز انجام دهد اما حتی کند ولی چگونه؟

با بسط تیلور، سینوس ها و ... اما این بخش کار را بسیار سریعتر انجام می دهد. در کامپیوترهای قدیمی این بخش مجزا بود و باید جداگانه تهیه می شد.



Page 391

** مثال : با قالب دستور 16 بیتی Encoding را بیابید که 15 دستور 3 آدرسه ، 14 دستور 2 آدرسه ، 31 دستور 1 آدرسه و 16 دستور صفر آدرس داشته باشد.

آیا Encoding مهم است ؟ برای اشکال زدایی ، debug - کشف آسیب پذیری - تشخیص داده از دستور

4 bit
0000 Add   0000 Sub ← ۱۶ حالت انتخاب!
0001 Sub   0001 Inc

در طراحی پردازنده آن هایی که کاربرد بیشتری دارند و هم شبیه تر هستند باید در تعداد بیت کمتری شباهت داشته باشند

↳ به دلیل کاهش ، ساده سازی مدار و سرعت بالا و افزایش سرعت decoder

Bytes	0 - 5	1 - 2	0 - 1	0 - 1	0 - 4	0 - 4
	PREFIX	OPCODE	MODE	SIB	DISPLACEMENT	IMMEDIATE

پردازنده Intel از نوع CISC هست
(Complex Instruction Set Computer)

گاها طول دستورات تا ۱۷ byte می‌رسد ← Encoding پردازنده Intel ←
که طول دستورات متغیر است ← به علت مشکلات، Legacy

- در پردازنده های CISC امروزه طول دستورات ثابت و 32 بیتی است.

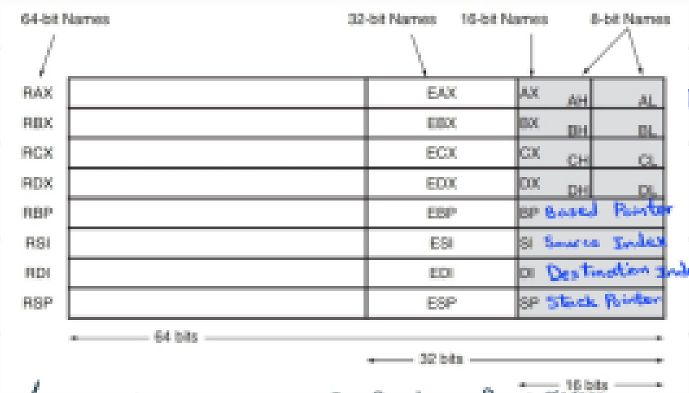
- وقتی طول دستورات متفاوت است چگونه که decode می‌شود؟ ← با توجه به پیشوندها در چند مرحله decode و نوع دستور مشخص می‌شود.
مثلاً پیشوند 11011 به معنای floating point است. که خیلی طول می‌کشد برای همین پردازنده RISC بهتر است.

فصل ۴ : آشنایی با پردازنده Intel (خانواده 80x86)

80x86 یا x86 (خانواده Intel) { 8086, 80186, ..., Pentium, Xeon

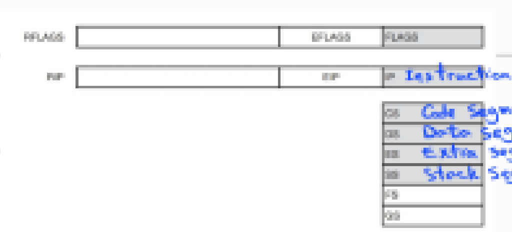
شناخت ISA ← شناخت دستورات - شناخت ثبات ها

8 ثبات قابل استفاده دارد.



کاربرد عمومی ←
(SI) اندیس مبدا، (DI) اندیس مقصد ← کاربرد خاص و دستورات خاص
در عملیات پشته مثل Pop, Push

↳ Intel Microprocessor By Prentice, Page 71 (53)



Instruction Pointer ← نمی‌توان مستقیم از آن استفاده کرد

Register Size	Override	Bits Accessed	Example
8 bits	B Byte	7–0	MOV R9B, R10B
16 bits	W Word	15–0	MOV R10W, AX
32 bits	D Double Word	31–0	MOV R14D, R15D
64 bits	—	63–0	MOV R13, R12

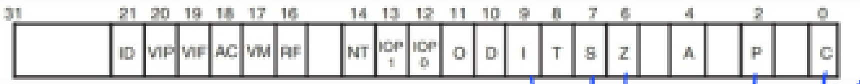
R9, R10 ، ۶۴ بیتی هستند ولی با پسوند B تنها byte اول منتقل می‌شود
۱۶ bit ← پسوندی ندارد

می‌توان با پسوند B ، W ، D تعداد بایت‌های انتقال یافته را تعیین کرد ←

* در 8086 ثبات R نداریم.

Page 54 ← خودمان بخوانیم!

ثبات پرچم‌ها یا وضعیت یا Processor Status Wor چند بیت که وضعیت را نشان می‌دهد و برای تصمیم‌گیری استفاده می‌شود.
شامل تاثیرات اطلاعات نتیجه وضعیت پردازنده



↳ Page 55

- Interrupt ← I
- (علامت) Sign ← S
- Zero ← Z ← اگر جواب 0 شود
- Parity (توازن) ← P → شمارش بیت‌های ۱ (odd, even)
- Carry ← C → اگر در حاصل عملیات رقم نقلی داشت این بیت ۱ می‌شود

Carry Flag

Flag: 000 ···· 001

BC LT

↳ Branch Carry → اگر فلگ Carry، ۱ بود برو به LT

6 5 4 3 2 1 0

000...0 | 0 0 0 0 0 0 1

Zero Flag

BEQ L2

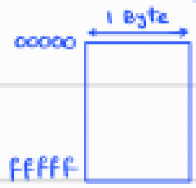
↳ Branch If Equal jump to L2 → در CMP ۲ مقدار را از هم کم می‌کند، اگر حاصل 0 شود فلگ zero ۱ می‌شود و BEQ تنها به فلگ zero اهمیت می‌دهد پس در این مثال شرط برقرار است (فلگ zero ۱ است)

* یک سری از دستورات اسمبلی با فلگ‌ها کار می‌کنند مثلا: توجه به فلگ Sign → BIN → Branch If Negative

در طراحی پردازنده Intel ابتدا باید Segment داده شود سپس Offset (جابجایی نسبت به Segment) تا بتوان به حافظه دسترسی یافت
← ۲ مرحله‌ای

امروزه می‌توان به صورت مستقیم در مد flat نیز آدرس‌دهی کرد ولی در وضعیت سنتی باید کار با Segment را بلد باشیم.

Register Segment
- Code Segment
- Data Segment
- Extra Segment
- Stack Segment



$16^5 = 2^{20} = 1 MB$

برای آدرس‌دهی خانه‌های بین حافظه باید یک آدرس 20 bit بدهیم ولی Intel قبول نمی‌کند ← ثبات‌های Intel، 16 بیتی است

Intel یک کلک زده، ابتدا باید مقدار Segment Register ۱۶ بیتی را برداریم و به آن 4 bit از خودش شیفت دهیم (به چپ) تا 20 bit شود سپس جمع کنیم با فهرس (Index)

DS + SI → Effective Address ← تلفیق ۲ مقدار ۱۶ بیتی برای پیدا کردن آدرس

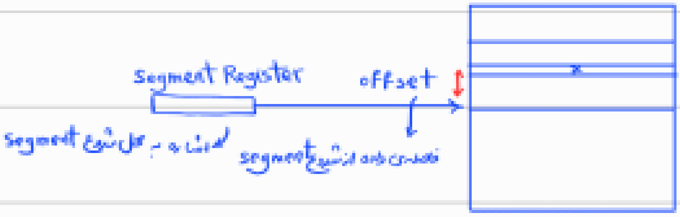
Segment Register	Starting Address	Ending Address
2000H	20000H	2FFFFH
2001H	20010H	3000FH
2100H	21000H	30FFFH
AB00H	AB000H	BAFFFH
1234H	12340H	2233FH

Segment Register <<4

حسن کار کردن با Segment ← تقسیم‌بندی مستقل و منظم ـ کاربرد بهتر حافظه

Effective Address = Segment <<4 + Offset

جلسه هفدهم ۲۸، ۱۴۰۳



وقتی یک آدرس می دهیم offset در نظر گرفته می شود و segment آن را به طور پیش فرض استفاده می کند

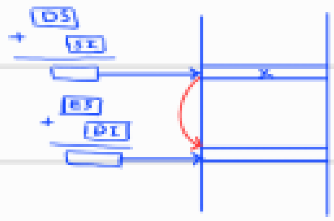
- DS (Data Segment Register): ثبات segment برای اشاره به قطعات حافظه حاوی داده

- CS (Code Segment Register): ثبات segment برای اشاره به قطعات حافظه حاوی دستور (Instruction) یا کد

- SS (Stack Segment Register) → SP و بالا رفتن به مقدار SS در نظر گرفتن مقدار ← Pop و Push موقع
  ↳ Stack Pointer (Offset)

* پردازنده با توجه به نوع دستور می داند از چه segment ای استفاده کند پس نیازی به ذکر آن نیست.
* مقداردهی segment ها با سیستم عامل است.

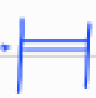
- ES (Extra Segment Register) → movs (move string) برای کارهای خاص یا برای یک سری از دستورات انتقال مثل



- البته در مد Flat می توان آدرس مستقیم خانه ای در حافظه را بدون segment آدرس دهی کرد.

مثال) CS = FACD ، PC = 103A ← پردازنده به سراغ کدام آدرس خواهد رفت؟

Effective Address = CS × 4 + PC = FACD0 + 103A = FBD0A → [instruction یک آدرس ۲۰ bit]
(0 zero in Hexadecimal or 4 bit)



FACD0
+ 103A
FBD0A

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

EA (Effective Address) در Memory Address Register نوشته می شود سپس پردازنده به سراغ این آدرس می رود

Segment Register	Starting Address	Ending Address
2000H	20000H (4 bit shift)	2FFFFH
2001H	20010H	3000FH
2100H	21000H	30FFFH
AB00H	AB000H	BAFFFH
1234H	12340H	2233FH

FFFF → 2^{16} = 64KB: segment اندازه
FFFF → 0

↳ Page 59 prentice

Segment	Offset	Special Purpose
CS	IP (PC) Instruction Pointer	Instruction address
SS	SP or BP	Stack address
DS	BX, DI, SI, an 8- or 16-bit number	Data address
ES	DI for string instructions	String destination address

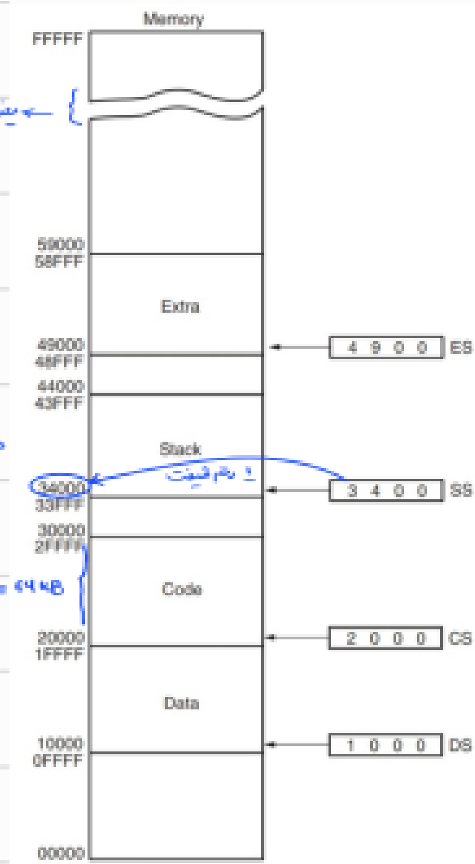
↳ page 60 prentice

Memory

Stack

* یادآوری: چرا Stack Segment از ته چیده می‌شود؟

— تا با حافظه برنامه‌ها برخورد نداشته باشد و زود پر نشود!

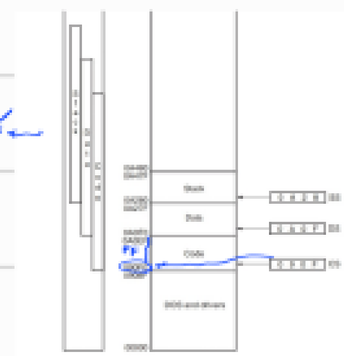


← بقیه حافظه ادامه دارد ...

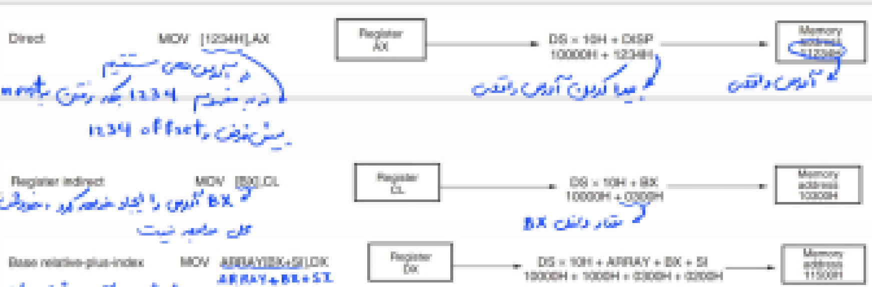
64 KB ← Segment‌ها

لازم نیست!

FFFF = 64 KB ← اندازه



← تکه‌تکه‌شکل ← هم پوشانی سگمنت‌ها (لازم نیست حتماً جدا از هم باشند)



آدرس دهی مستقیم — در حالت مستقیم 1234 یعنی رفتن به Segment — 1234 offset پیش فرض

چطور آدرس واقعی — آدرس واقعی

page 49

BX آدرس را ایجاد خواهد کرد، خودش کل حافظه نیست — BX مقدار داخل

ARRAY+BX+SI — ARRAY یک مقدار constant، آدرس شروع جدول (ثابت آخر در نظر بگیرید)

* برای هر کدام از ترکیب‌های مختلف mov داریم!

Page 84

Assembly Language	Size	Operation
MOV AL,NUMBER	8 bits	Copies the byte contents of data segment memory location NUMBER into AL
MOV AX,COW	16 bits	Copies the word contents of data segment memory location COW into AX
MOV EAX,WATER	32 bits	Copies the doubleword contents of data segment location WATER into EAX
MOV NEWS,AL	8 bits	Copies AL into byte memory location NEWS
MOV THERE,AX	16 bits	Copies AX into word memory location THERE
MOV HOME,EAX	32 bits	Copies EAX into doubleword memory location HOME
MOV ES:[2000H],AL	8 bits	Copies AL into extra segment memory at offset address 2000H

← در واقع آدرس یک شماره است که متغیر Number برای آن تعریف شده، Immediate و direct Addressing!

← خود اسمبلر براساس حجم مقدار را تشخیص می‌دهد

← قدر خانه مقدار آدرسی مبدا شده توسط WATER در EAX در Data Segment

* mov ها در Data Segment کار می‌کند

← یعنی AL در خانه‌ای که Extra Segment، 2000 (offset) ← ES×10H + 2000H ← AL

می‌توانیم Segment دلخواه کار کرد!

MOV CH,DS:[1000H]	8 bits	Copies the byte contents of data segment memory offset address 1000H into CH

چون خودش mov بصورت پیش‌فرض با DS کار می‌کند، پس چرا؟ خوانایی بیشتر برنامه

رفتن به آدرس 1234 در Data Segment و برداشتن 1 byte (AL) و ریختن در AL

EXAMPLE 3–5

```
0000 A0 1234 R        MOV AL,DS:[1234H]
0003 BA 0E 1234 R     MOV CL,DS:[1234H]
```

Relocatable

چرا opcode ها متفاوت است؟ به علت تفاوت AL و CL / دستور AL جهت تداول تر است که آن کوتاه‌تر است

آدرس تولیدی توسط اسمبلر و نحوه چیدمان دستورات در حافظه

که دستور است — opcode خارج قسمت‌ها

برعکس

* دستوراتی که R دارند آدرس دهی مطلق داشته و loader باید بتواند آن را تغییر کند.

جلسه هجدهم ۱۴۰۳,۹,۳

مدهای آدرس دهی ← موثر در طول دستور / نحوه شروع دستورات با opcode

انواع دستورات در 8086 :

- دستورات جابجایی داده (Transfer Data) :

mov (Move) - PUSH (Push) - POP (Pop) - XCHG (Exchange) -

IN (Input from) - OUT (Output to) - LDS - LAHF - PUSHF - POPF

Intel 8086 datasheet , page 26

IN AL, Port1 → دریافت ورودی از دستگاه‌های جانبی (شباهت با mov در Intel 80x86)
در موتورولا 68000 همه با mov انجام میشد

XCHG AX, BX → AX ⇄ BX

Lock XCHG AL, Semaphore

Lock ← قفل میشوند
که دسترسی به متغیر را در میانه کار نمی‌دهد
دستور از جنس [RMW]
↳ Read/Modify/Write
↓
دسترسی و انقطاع / atomic
تغییر کردن متغیر در این لحظه

- دستورات حسابی (ریاضی) (Arithmetic)

Page 27

ADD (Add), ADC (Add with Carry), INC (Increment), DEC (Decrement)

SUB (Subtract), NEG (Change Sign), CMP (Compare), MUL (Multiply (unsigned)), IMUL (Integer Multiply (signed)), DIV, IDIV,

CMP AX, BX ⇒ Set flags based on AX-BX value → Sign - Zero - Carry - Overflow - ...
اگر AX - BX = 0 → Z = 1

MUL AX, BX ⇒ حاصل ضرب 32 بیت را در 2 ثبات ذخیره می کند

* دستورات پردازنده‌های 80X86 ← 2-operand ← کمی محدود
ADD AX, BX ← ساختار دیگر ساده‌تر، سریعتر، کم مصرف‌تر
ملاحظات سخت افزاری

- دستورات منطقی (Logic)

Page 28

مکمل
↑
NOT (Invert), SHL / SAL (Shift Logical / Arithmetic Left) → در شیفت به چپ Arithmetic, Logical تفاوتی ندارند

ROL (Rotate Left), ROR (Rotate Right), AND, OR, XOR

SHR (Shift Logical Right)
با هم فرق دارند!
SAR (Shift Arithmetic Right)

AX = 3F4A | 0011 | 1111 | 0100 | 1010 |

SHL AX | 0111 | 1110 | 1001 | 0100 | ← 0 AX = 7E94

در این شکل فرقی نداشت ←
SHR AX 0 → | 0001 | 1111 | 1010 | 0101 | AX = 1FA5
SAR AX 0 → | 0001 | 1111 | 1010 | 0101 | AX = 1FA5

بیت Carry با اینکه تنها یک بیت است و به همین علت در بسیاری از موارد مثل شیفت دادن یا rotate تاثیر می‌گذارد

شیفت به چپ ← ضرب در ۲ ، شیفت به راست ← کف تقسیم بر ۲ $\lfloor \frac{AX}{2} \rfloor$

BX [1001 | 1111 | 1010 | 0101] BX = 9FA5 ⟶ عدد منفی است
↳ Sign

LSHR BX 0 → [0100 | 1111 | 1101 | 0010] BX = 4FD2 (علامت حفظ نشد)

ASHR BX 1 → [1100 | 1111 | 1101 | 0010] BX = CFD2 (علامت حفظ شد)

LSHR (Logical Shift Right) ⟶ شیفت دادن به سمت راست
ASHR (Arithmetic Shift Right) ⟶ حفظ و مشترک بودن علامت عدد } *

مثال) R_1 = [1110] = -2 ASHR R_1 = [1111] = -1 / R_3 = [1101] = -3 , ASHR R_3 [1110] = -2 = $\lfloor \frac{-3}{2} \rfloor$ ✓

علامت حفظ شد ⟶ منفی!

- دستورات روی رشته REP, MOVS, CMPS, SCAS, LODS

- دستورات کنترلی و شرطی CALL (صدا زدن زیر برنامه), JMP (Unconditional Jump) (پرش بدون شرط), RET (Return)

دستورات شرطی (بعد از دستورات مقایسه) Page 29

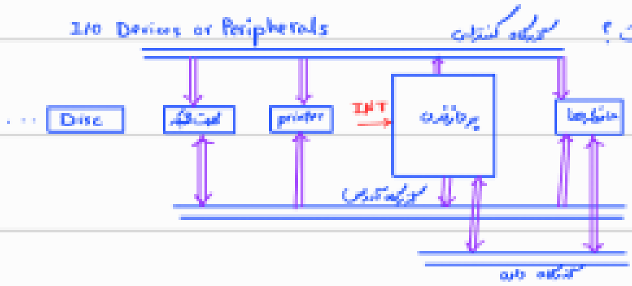
بعد از دستور CMP استفاده می‌شود ⟵ چرا؟ تا دستور برای یک کار؟ ⟵ برای آسانی در برنامه‌نویسی در اسمبلی

- **JE/JZ** = Jump on Equal/Zero
- **JL/JNGE** = Jump on Less/Not Greater or Equal
- **JLE/JNG** = Jump on Less or Equal/Not Greater
- **JB/JNAE** = Jump on Below/Not Above or Equal
- **JBE/JNA** = Jump on Below or Equal/Not Above
- **JP/JPE** = Jump on Parity/Parity Even
- **JO** = Jump on Overflow ⟵ هنگامی که overflow رخ می‌دهد
- **JS** = Jump on Sign ⟵ اگر علامت منفی شود
- **JNE/JNZ** = Jump on Not Equal/Not Zero
- **JNL/JGE** = Jump on Not Less/Greater or Equal
- **JNLE/JG** = Jump on Not Less or Equal/Greater ⟵ برای علامت دارها (چه فرقی دارند؟)
- **JNB/JAE** = Jump on Not Below/Above or Equal ⟵ برای بدون علامت‌ها (Unsigned)
- **JNBE/JA** = Jump on Not Below or Equal/Above

- **LOOP** = Loop CX Times ⟵ ابتدا مقدار CX (تعداد تکرار) را تعیین می‌کنیم (شرط ⟵ CX) و بعد از هر بار تکرار از مقدار CX کم می‌کند
- **LOOPZ/LOOPE** = Loop While Zero/Equal ⟵ علاوه بر CX، شرط های دیگری هم دارد
- **LOOPNZ/LOOPNE** = Loop While Not Zero/Equal
- **JCXZ** = Jump on CX Zero

* دستور INT (Interrupt) به معنای وقفه ⟵ یکی از ورودی‌های پردازنده ـ یادآوری: پردازنده ⟵ ملک (master)، دستگاه‌های جانبی محیط ⟵ رعیت (slave)

شکل کامپیوتر:



پس چرا پردازنده یک ورودی Interrupt برای دستگاه‌های جانبی قرار داده است؟

برای اینکه اگر کسی با پردازنده کار داشت و می‌خواست چیزی بفرستد و

ماهیت آسنکرون (ناهمگام) داشت با فعالیت پردازنده می‌تواند زنگ بزند

در غیر این صورت همواره پردازنده به سراغ بقیه می‌رود

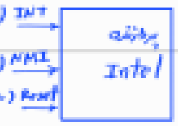
علت وجود Interrupt ⟵ پردازنده دائما به سراغ دستگاه ها نرود برای اینکه از آنها بپرسد آیا درخواستی دارند یا نه (اتلاف وقت!)

ـ روش دسترسی به دستگاه های جانبی :

① Polling (سرکشی) ← دائما می‌پرسد کسی سوالی دارد

② Interrupt (وقفه‌دهی) ← می‌گوید هرکسی سوال داشت دستش را بلند کند / موقتا کار پردازنده متوقف می‌شود

اگر پردازنده خیلی با دستگاه‌ها سروکار داشت از روش سرکشی استفاده می‌کنیم و اگر پردازنده کمتر با کسی کار داشت از روش وقفه‌دهی استفاده می‌کنیم



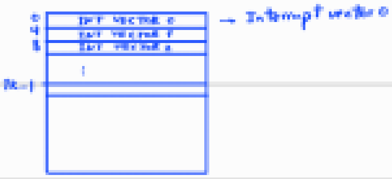
INT (Maskable Interrupt) ← اگر تک INT، ۱ بود پاسخش را می‌دهد

NMI (Non Maskable Interrupt) ← پردازنده باید حتما کارش را رها و به سراغ کار جدید برود (مثل زمانی که یک برنامه به سیستم آسیب بزند)

Reset (بازنشانی، رها کردن کار و برگشت به وضعیت اولیه)

طراحان در پردازنده Intel این امکان را به ما می‌دهند تا در وسط برنامه‌ها (نه حالت اتفاقی) پردازنده را با دستور INT صدا بزنیم (پردازنده اجازه می‌دهد از INT0 تا INT255 را صدا بزنیم مثلا INT26)

INT 56 یعنی چی؟ یعنی زیربرنامه 56 توسط پردازنده اجرا شود، از کجا؟ یک جدول مخفی که همیشه از خانه 0 حافظه شروع می‌شود:



پس می‌رود آدرس زیربرنامه 56 را به صورت غیرمستقیم (indirect) از جدول پیدا کرده، اجرا می‌کند

ـ دستورات کنترلی

Page

CLC = Clear Carry

CMC = Complement Carry

STC = Set Carry

CLD = Clear Direction

STD = Set Direction

CLI = Clear Interrupt

STI = Set Interrupt

HLT = Halt

WAIT = Wait

ESC = Escape (to External Device)

LOCK = Bus Lock Prefix

جلسه نوزدهم ۱۴۰۳/۹/۵

هر اسمبلر یک سری Directive (فرمان نما) دارد
که یک سری دستورالعمل برای اسمبلر

پردازنده اسمبلی نمی فهمد فقط ۰ و ۱ می فهمد. مانند اسمبلی [illegible] تحت افزار نزدیک است می تواند از منابع سخت افزاری بهتر استفاده کند.

اسمبلر کد اسمبلی را به کد باینری که برای پردازنده قابل فهم است تبدیل می کند (با کامپایلر متفاوت است)

با هر زبان برنامه نویسی که بنویسیم نهایتا به اسمبلی و کامپایل شود و تبدیل به کد باینری می شود (کد باینری خوانا نیست) به همین علت اسمبلرها کد [illegible] را در کنار دستور قرار می دهند)

Directive ها اسمبلی را آسانتر می کنند. مثلا در کد بعدی در یک Directive اسمبلر می توان یک متغیر constant تعریف کرد

0000 (آدرس) 10 (opcode) DATA1 (دستور) DB (Define Byte (1 Byte) → Directive) 10H

که اسمبلی است دستور نیست بلکه Directive است برای خوانایی بیشتر برنامه

مورد استفاده برای تعریف متغیر در اسمبلی چند: DB(Byte), DW(WORD) 2 Byte, DD(Double Word) 4 Byte, DQ(Quarter Word) 8 Byte

در کد بعدی برای خوانایی و آسانی مقدار 10H را در DATA1 ریخته و از DATA1 به عنوان متغیر با مقدار 10H استفاده می کنیم

0014 A0000 R — MOV AL, DATA1 — آدرس دهی مستقیم

که در اینجا هیچ 10 نوشته نشده پس حاصل آدرس آن است (اگر بود مقدار ۱۰ که دستور فعلی نوشته می شد)

The Intel Microprocessors, Prentice, Page 104(88)

```
                      .MODEL SMALL     ;choose small model
0000                  .DATA            ;start data segment

0000 10               DATA1 DB 10H     ;place 10H into DATA1
0001 00               DATA2 DB 0       ;place 00H into DATA2
0002 0000             DATA3 DW 0       ;place 0000H into DATA3
0004 AAAA             DATA4 DW 0AAAAH  ;place AAAAH into DATA4

0000                  .CODE            ;start code segment
                      .STARTUP         ;start program

0017 A00000 R         MOV AL,DATA1     ;copy DATA1 into AL
001A 8A 26 0001 R     MOV AH,DATA2     ;copy DATA2 into AH
001E A3 0002 R        MOV DATA3,AX     ;copy AX into DATA3
0021 8B 1E 0004 R     MOV BX,DATA4     ;copy DATA4 into BX

                      .EXIT            exit to DOS
                      END              end program listing
```

آدرس متغیر در حافظه — مقدار اولیه

۱ بایت در حافظه اشغال می کند (DB) — ۲ بایت اشغال می کند (DW)

هر عبارت کد باینری با opcode شروع و به آدرس ختم می شود و دنبالی هم به ترتیب واحد در دستور در دستور اسمبلی تعادل

MOV AL, MOV AH: (Direct) — آدرس دهی مستقیم (Direct)

.EXIT, END → Directive — افزایش خوانایی برنامه

Page 109(90) Prentice

Directive انواع دیگر:

MOV [DI],10H — مبهم است چون نمی دانیم DI به چند بیت اشاره می کند

MOV BYTE PTR [DI],10H → واضح تر و خواناتر

- WORD — 16 bit
- DWORD → 2 WORD — 32 bit
- QWORD → 4 WORD — 64 bit
- OWORD → 8 WORD — 128 bit

MOV AL,[DI] → مقدار محل مورد اشاره توسط DI را در AL می ریزد (مثلا اگر DI=100 باشد مقدار خانه 100 را در AL می ریزد) → Register Indirect Addressing

MOV DI,10H → DI مقدار 10H قرار می دهد → Immediate Addressing

Page 104(83)

Label	Opcode	Operand	Comment
DATA1	DB	23H	;define DATA1 as a byte of 23H
DATA2	DW	1000H	;define DATA2 as a word of 1000H
START:	MOV	AL,BL	;copy BL into AL
	MOV	BH,AL	;copy AL into BH
	MOV	CX,200	;copy 200 into CX

تعریف متغیر با مقادیر مشخص

AL,BL / BH,AL: Register Addressing — CX,200: Immediate Addressing

MOV BX, DATAS → قرار دادن مقدار متغیر DATAS در BX

MOV BX, OFFSET DATAS → قرار دادن فاصله DATAS از شروع سگمنت در BX

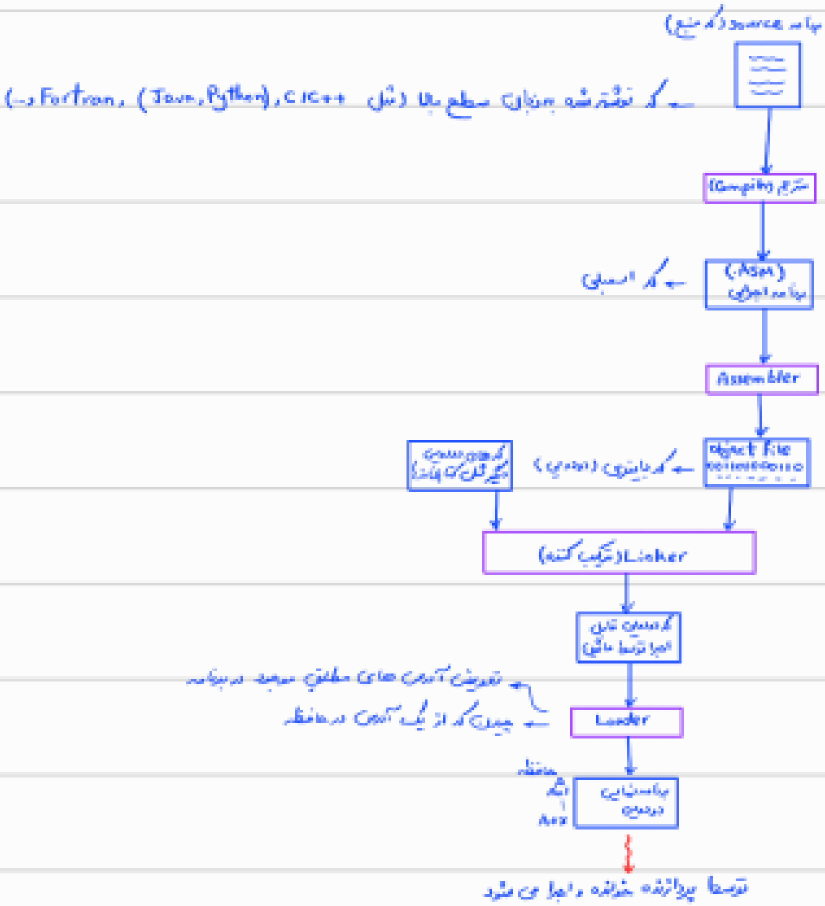
(OFFSET: به معنای فاصله)

```
                        .MODEL SMALL          ;select small model
0000                    .DATA                 ;start data segment

0000 0032 [      DATAS  DW  50 DUP(?) ;setup array of 50 words
              0000
          ]
0000                    .CODE                 ;start code segment
                        .STARTUP              ;start program
0017 B8 0000            MOV  AX,0
001A 8E C0              MOV  ES,AX            ;address segment 0000 with ES
001C B8 0000 R          MOV  BX,OFFSET DATAS  ;address DATAS array with BX
001F B9 0032            MOV  CX,50            ;load counter with 50
0022             AGAIN:
0022 26:A1 046C         MOV  AX,ES:[046CH]    ;get clock value
0026 89 07              MOV  [BX],AX          ;save clock value in DATAS
0028 43                 INC  BX               ;increment BX to next element
0029 43                 INC BX
002A E2 F6              LOOP AGAIN            ;repeat 50 times

                        .EXIT                 ;exit to DOS
                        END                   ;end program listing
```

جلسه بیستم ۱۴۰۳/۸/۱۰



تفاوت Compiler (مترجم) و Interpreter (مفسر)؟ Compiler کل متن را یکجا و یک‌دفعه ترجمه می‌کند و Interpreter خط به خط و در لحظه ترجمه می‌کند
مثل C, C++ — مثل Python, Java

حسن Compiler: بررسی کلی کد و تبدیل آن به کد بهینه‌تر (مثلاً تخمین حلقه‌ها بی‌نهایت) (دیاگرام بالا برای زبان‌های کامپایلی است!)

حسن Interpreter: کد در لحظه خوانده و اجرا می‌شود (ولی لزوماً بهینه‌سازی شده نیست!)

کارهای سریع‌تر ← استفاده از زبان کامپایلی مثل C, Golang • زبان‌های Interpreted کندترند!

ـ دستورات اسمبلی به چند دسته تقسیم می‌شوند؟

۱ـ دستورات انتقالی مثل LOD, STR, mov، OUT، IN
OUT MM, AX ← خروجی دادن مقدار AX به پورت MM

خواندن از حافظه
Lod R0, X ← خواندن مقدار X در حافظه و قرار دادن در ثبات R0

STR R1, Y ← ذخیره کردن مقدار ثبات R1 در Y
قرار دادن در حافظه

به اینگونه دستورات Lod و Str در پردازنده‌های RISC نسبت به دستور mov در پردازنده‌های CISC است، چرا؟!

	تعداد ثبات	مد آدرس‌دهی	تعداد دستور	مراجعه به حافظه	نوع دستورات
RISC	زیاد	کم	کم	فقط با load, store	ساده‌تر
CISC	کم	زیاد	زیاد	با mov و حتی دستورات محاسباتی، منطقی	پیچیده‌تر و قوی‌تر

در پردازنده RISC تنها با دستورات load, store می‌توان از حافظه خواند و در آن نوشت

ولی در پردازنده CISC می‌توان با دستور mov داده را از حافظه به ثبات، از ثبات به حافظه، از حافظه به حافظه و از ثبات به ثبات انتقال داد مثلاً

۲. یک مراجعه به حافظه در ۱ دستور: مثلا اگر بخواهیم مقدار Mem1 در حافظه را جمع کنیم با مقادیر و نتیجه مجددا برود در حافظه برای این عملیات ← Mem1 ← Mem1 + AX مثلا Add Mem1, AX

in CISC		in RISC
Add Mem1, R1	⟷	Lod R0, Mem1
		Add R0, R0, R1
		STR R0, Mem1

در پردازنده RISC این کار ممنوع است ! ← در هر مراجعه تنها یکبار می‌توان از حافظه خواند یا در آن نوشت.

⇓

۱ دستور CISC با ۳ دستور RISC قابل انجام است

پس چرا RISC در سرعت برنده است؟

برای نگهداری از مقادیر حافظه از ثبات استفاده می‌کنیم (علت زیاد بودن ثبات‌ها در RISC)

● فرمول کارایی یا زمان اجرا:

$$t_{ex} = n \times CPI_{avg} \times T_{clock}$$

n: تعداد دستورات (تعداد کل دستورات یک برنامه)

CPI: Clock Per Instruction

$T_{clock} = \frac{1}{freq}$

if $freq = 2GHz \Rightarrow T_{clock} = \frac{1}{2 \times 10^9 Hz} = 0.5 ns$

● فرمول دقیق:

$$t_{ex} = T_{clock} \sum_{i=1}^{k} n_i \, CPI_i$$

k ← تعداد کل انواع دستور

تعداد دستورات تولید شده توسط Compiler

↑

	n	CPI	T_{clock}
RISC	بیشتر	کمتر	کمتر
CISC	کمتر	بیشتر	بیشتر

ثبات‌های RISC بیشتر از CISC ← کمتر به حافظه مراجعه بیشتری دارد ← CPI کمتر در RISC

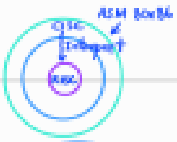
$t_{ex_{RISC}}$ = بیشتر × کمتر × کمتر

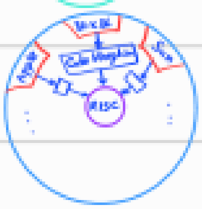
$t_{ex_{CISC}}$ = کمتر × بیشتر × بیشتر

$\Rightarrow t_{ex_{RISC}} < t_{ex_{CISC}} \Rightarrow$ RISC ← بهینه

علت سریع‌تر بودن RISC:
- ثبات‌های زیاد
- طراحی ساده‌تر ← سرعت بیشتر

* پس چرا پردازنده‌های Intel که CISC نیز هستند سریع‌اند؟ هسته‌ی داخلی RISC





* شرکت Transmeta ← پردازنده Crusoe

۲. عملیات حسابی Add, mul, div, ...

۳. عملیات منطقی و جابجایی SHR, SHL, Rotate, and, OR, XOR, ...

۴. پرش - شرطی
- بدون شرط
- زیر برنامه
- وقفه

۵. کنترل و ثبت رویدادها

۶. عملیات ممیز شناور و محاسبات پیچیده مثل log ، exp ، tgh و ... و تغییر شیت و ...

که عملیات برداری و موازی

Intel microprocessor , Page 151

← دستورات mov شرطی (Conditional mov) ← می‌توانیم گفت اگر بیت نشانه با CMP ست نشده بود، دستور mov را انجام بده با conditional mov

* above ← بالاتر ، great ← بزرگتر علامت

TABLE 4-20 The conditional move instructions.

Assembly Language	Flag(s) Tested	Operation
CMOVB	C = 1	Move if below
CMOVAE	C = 0	Move if above or equal
CMOVBE	Z = 1 or C = 1	Move if below or equal
CMOVA	Z = 0 and C = 0	Move of above
CMOVE or CMOVZ	Z = 1	Move if equal or move if zero
CMOVNE or CMOVNZ	Z = 0	Move if not equal or move if not zero
CMOVL	S ! = O	Move if less than
CMOVLE	Z = 1 or S ! = O	Move if less than or equal
CMOVG	Z = 0 and S = O	Move if greater than
CMOVGE	S = O	Move if greater than or equal
CMOVS	S = 1	Move if sign (negative)
CMOVNS	S = 0	Move if no sign (positive)
CMOVC	C = 1	Move if carry
CMOVNC	C = 0	Move if no carry
CMOVO	O = 1	Move if overflow
CMOVNO	O = 0	Move if no overflow
CMOVP or CMOVPE	P = 1	Move if parity or move if parity even
CMOVNP or CMOVPO	P = 0	Move if no parity or move if parity odd

← اگر علامت منفی شد انتقال بده (CMOVS)

Directive ← شبه عملیات یا شبه دستور / تنها قابل فهم برای اسمبلر

Page 199

```
0000 01 02 03      DATA1 DB  1,2,3          ;define bytes
0003 45                  DB  45H            ;hexadecimal
0004 41                  DB  'A'            ;ASCII
0005 F0                  DB  11110000B      ;binary
0006 000C 000D     DATA2 DW  12,13          ;define words
000A 0200                DW  LIST1          ;symbolic
```

برنامه برای خواندن یک کلید از صفحه کلید و نمایش آن

EXAMPLE 4–20

```
                        ;An example DOS full-segment program that reads a key and
                        ;displays it. Note that an @ key ends the program.
                        ;
0000                    CODE_SEG        SEGMENT 'CODE'
                                ASSUME CS:CODE_SEG

0000                    MAIN    PROC    FAR

0000 B4 06                      MOV   AH,06H    ;read a key
0002 B2 FF                      MOV   DL,0FFH
0004 CD 21                      INT   21H
0006 74 F8                      JE    MAIN              ;if no key typed
0008 3C 40                      CMP   AL,'@'
000A 74 08                      JE    MAIN1             ;if an @ key
000C B4 06                      MOV   AH,06H            ;display key (echo)
000E 8A D0                      MOV   DL,AL             on monitor
0010 CD 21                      INT   21H
0012 EB EC                      JMP   MAIN              ;repeat
0014                    MAIN1:
0014 B4 4C                      MOV   AH,4CH            ;exit to DOS
0016 CD 21                      INT   21H

0018                    MAIN    ENDP
0018                            END   MAIN
```

استفاده از صفحه 131 تا 130 کتاب The Intel Microprocessors (by Brey) جهت توضیح کدها

جلسه بیست یکم ۱۴۰۳/۸/۱۲

دستورات پردازنده :

- انتقال　- منطقی + جابجایی (Shift) + دوران (Rotation)　- حسابی (ریاضی)

- ریاضی ممیز شناور (log , exp , tg , tgh , $\sqrt{\ }$, x^2 , sin , cos , ...) (در همه ی پردازنده ها نیست!)

← واحد Floating Point Unit or CoProcessor در پردازنده　چرا؟ چون خروجی این توابع عدد صحیح نیستند پس باید با بخش ممیز شناور کار کنیم

← چرا باید این دستورات را بدانیم؟ ممکن است Compiler نتواند از امکانات به صورت بهینه استفاده کند یا ممکن است یک پردازنده این امکانات را نداشته باشد و نیاز به اسمبلی داشته باشیم.

- دستورات پرش (بدون شرط یا باشرط)　- دستورات کنترلی　- دستورات ثبت رویداد

VTune ابزار پیشرفته Intel برای افزایش Performance (High Performance)، ثبت تعداد Branch miss Prediction، Cach miss و ... که می توان از آن برای تغییر دستورات و افزایش سرعت استفاده کرد.

علت اهمیت سرعت پردازنده (High Performance)؟

حوزه های پردازش سریع:
- Real Time System (سامانه های بانکی یا پاسخ به موقع) در تجهیزات پزشکی در سرعت ثبت رویداد یا مثلا تشخیص جهت اقدام یا سیستم های اطفای حریق
- Simulation شبیه سازی مثلا در هواپیما و خودرو
- AI/Machine Learning ، در GPU سرعت بیشتر، تعداد بیشتر پردازش سریعتر

- Intel از دستورات ۱ و ۲ عملوندی پشتیبانی می کند

دستورات تک عملوندی (1-Address Operand)

MUL CL	AL is multiplied by CL; the unsigned product is in AX
IMUL DH	AL is multiplied by DH; the signed product is in AX
IMUL BYTE PTR[BX]	AL is multiplied by the byte contents of the data segment memory location addressed by BX; the signed product is in AX
MUL TEMP	AL is multiplied by the byte contents of data segment memory location TEMP; the unsigned product is in AX

ضرب ۲ عدد ۸ بیتی ← ۱۶ بیت

افزایش خوانایی و فهم انجام (ضرب علامت دار و بدون علامت)

← ضرب علامت دار و بی علامت دار به صورت ضرب عادی نمی توان انجام داد ← الگوریتم Booth و Baugh Wooley

← Directive (شبه دستور) ← از امکانات اسمبلی برای افزایش خوانایی

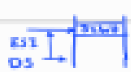
← ضرب ۱ بایت از محل مورد اشاره توسط BX در AL و نوشتن در AX

← این کار در پردازنده RISC غیر مجاز است چون فقط می توان از حافظه خواند یا نوشت نه ضرب و جمع

⇓

چنین عملیات ها فقط در پردازنده و ثبات ها قابل انجام است!

direct Addressing



MUL DWORD PTR [ESI] ← ضرب مقدار مورد اشاره توسط ثبات ESI (۳۲ بیت) در EAX ← ۶۴ بیت جواب　EDX : EAX

* تقسیم با سایر عملیات ها متفاوت است ← ۲ خروجی دارد خارج قسمت و باقی مانده

AX: [باقی مانده | خارج قسمت]

DIV CL ← AX ÷ CL ← AL , AH
(AL خارج قسمت، AH باقی مانده)

اگر CL بسیار کوچک باشد و جواب در ۸ بیت نگنجد چه کنیم؟　مثلا

$$\begin{array}{r|l} 3426 & 16 \\ \hline & 214 \\ 2 & \end{array}$$

خارج قسمت در بیش از ۸ بیت جا نمی شود

← overflow رخ می دهد و مقدار overflow (Flag) ۱ می شود

← پس اگر تقسیم انجام می دهیم و احتمال overflow هست حتما باید با دستورهای شرطی، فلگ overflow را چک کنیم

دستورات پرش (مشروط یا بدون شرط) ← نوعی از دستورات کنترلی

- Control Unit ← دارای یک آدرس دهنده Instruction Register (IP or PC)
  - IP: Instruction Pointer
  - PC: Program Counter
  - (اشاره‌گر)
- Execution Unit (Data Path Unit)

ماشین تورینگ جدا از پاسخ دارای ۲ قسمت (در کامپیوترهای امروزی):
- Control Unit (مغز) — واحد کنترلی
- Data Path Unit (بازو) — واحد اجرای پردازش یا محاسبات ← هسته

هر یک از هسته‌های محاسباتی تحت فرمان Control Unit ← Core 1 | Core 2 | ... | Core n

Control Unit از طریق Program Counter آدرس را به Instruction Memory متصل می‌کند.
← حافظه دستورالعمل یا حافظه برنامه (Code Memory)

پس در کامپیوتر حافظه‌ای مجزا نداریم که برنامه وجود داشته باشد که داخل حافظه جدا هست ← طبق مدل فون نویمن: Stored program Computer

← کامپیوتر دستوراتی از قبل ریخته شده دارد را اجرا می‌کند
← Disc، فلش، Hard، ... ⇐ Firmware ← برنامه‌ای که سازنده از ابتدا در دستگاه قرار داده

- hardware
- software
- firmware
- middleware ← مثل ماشین مجازی VM (Virtual Machine) و لایه‌ای میانی

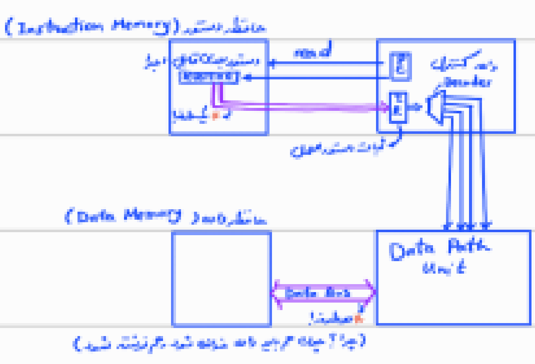
Control Unit آدرس دستورالعمل را به حافظه متصل می‌کند و چون حافظه به آدرس ذخیره شده دارد دستورالعمل را روی Bus data به Control Unit می‌دهد

این دستورات درون Instruction Register ذخیره می‌شوند و از طریق decoder فرمان‌های لازم به Data Path Unit (Execution Unit) داده می‌شود
← ریزدستورات یا فرمان‌های اجرایی

پس برنامه به صورت حافظه رفته و می‌بینم چه باید رخ داد را اجرا کنم (البته حافظه تنها آدرس و محافظت نوشتن هم است!) به برنامه که داخل حافظه و می‌نویسد که باید چکار کند بعد فرمان را به

Data Path Unit می‌دهد، دادن و اجرای تمام داده‌ها را ندارد پس داده‌ها را از حافظه می‌گیرد (جدا سازی بخش دستور از داده داخل برنامه رخ می‌دهد به صورت فیزیکی)
← شامل تعدادی Register، ALU

برای جدا سازی جهت پردازنده علاوه بر اعمال operand روی داده‌ها فرمان دارد آدرس دستورات بعدی را هم می‌خواند و توجه می‌کند



چرا به دستورات پرش (مشروط و غیرمشروط) دستورات کنترلی می‌گویند؟ دستورات انشعاب روال کنترل برنامه را بر هم می‌زنند و تغییر می‌دهند پس Control Unit مجبور است به خاطر ... دستور

ترتیب اجرای دستورات را بر هم می‌زند

مطلب دست ۴ : Control Unit طبق مطالب قبلی پس از اجرای دستورات هر دستور ! واحد Program Counter اضافه می‌کند ولی دستورات انشعاب باعث می‌شوند تا Control Unit

PC را به گونه‌ای دیگر تغییر دهد یعنی باعث می‌شوند تا روال کنترل در ماشین عوض شود و اجرا هم بشود ← دستورات کنترلی (کنترل برنامه)

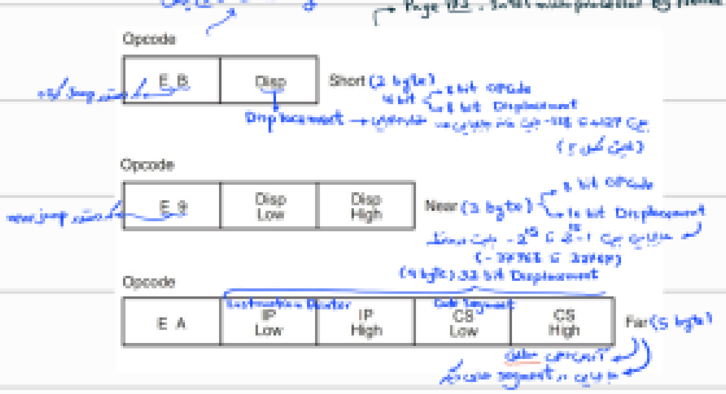
_ نوع اول : دستورات پرش بلا شرط ← JMP (Jump) or BR (Branch) ← به یک مرجعی که کنار می‌شود

در Intel بیشتر پرش‌ها نسبی هستند   ۳ نوع پرش { short - / near - / far -

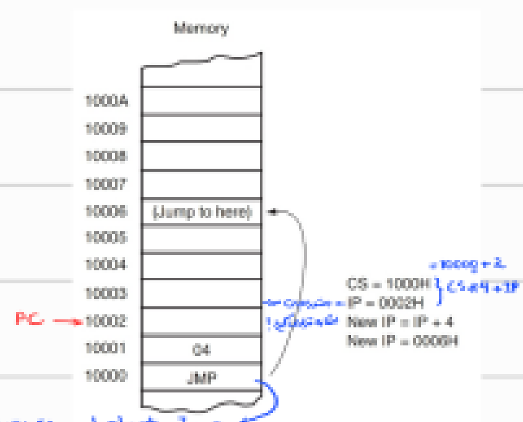
Encoding ۳ نوع پرش   → Page 193 , Intel microprocessor By Brey

(گسترش علامت) Signed extension ← FFF8 + 1002 = 10FFA ← به خاطر منفی بودن علامت

PC ← 1002, 1000 : JMP F8 (04) ← فاصله پرش به کجا



چرا اکثر آدرس دهی‌ها نسبی هستند ؟ راحتی و سرعت بیشتر و relocatable بودن برنامه



Jump ← short است (۲ بایتی است) پس دستور بعدی در خانه 10002 است

BX ← صفر کردن BX

```
0000 33 DB                    XOR  BX,BX
0002 B8 0001          START:  MOV  AX,1
0005 03 C3                    ADD  AX,BX
0007 EB 17                    JMP  SHORT NEXT

                      <skipped memory locations>

0020 8B D8            NEXT:   MOV  BX,AX
0022 EB DE                    JMP  START
```

IP → 0009) ۲ بایت (دستور ۲ بایتی است)

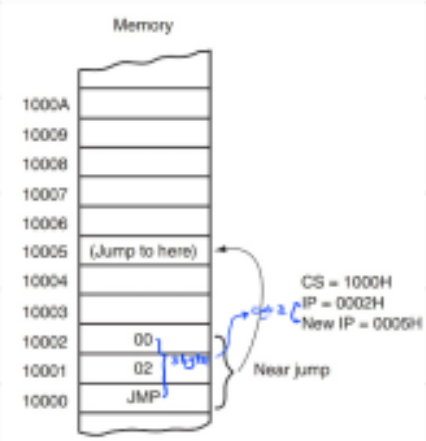
SHORT ← Directive ← همچنین پرش نوع پرش near jump هست

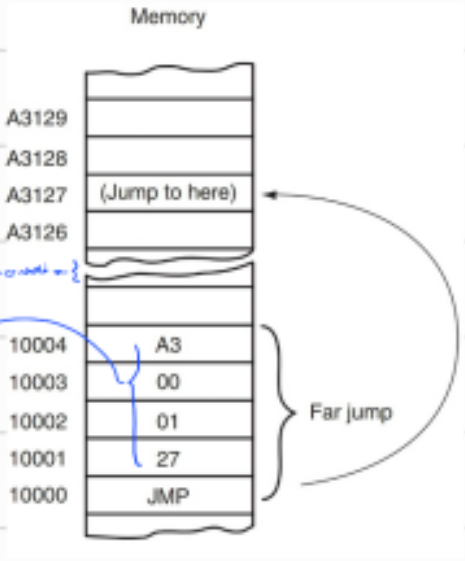
باید به آن بگوییم که پرش کجاست ، اسم آن را با فاصله که کنار می‌کند

$$\begin{array}{r} 20 \\ -\ 09 \\ \hline (17)_{16} \end{array}$$

IP → 0024) short Jump ← ۲ بایتی

$$\begin{array}{r} 0024 \\ +\ FFDE \\ \hline 0002 \end{array}$$ ⟶ START آدرس

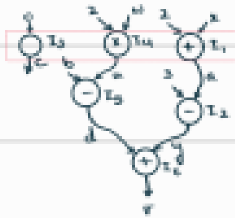

Memory
1000A
10009
10008
10007
10006
10005
10004
10003
10002
10001
10000
(Jump to here)
00
02
JMP
CS = 1000H
IP = 0002H
New IP = 0005H
Near jump


Memory
A3129
A3128
A3127
A3126
(Jump to here)
10004
10003
10002
10001
10000
A3
00
01
27
JMP
Far jump

جلسه بیست و سوم ۱۹ / ۸ / ۱۴۰۳

$I_1: a = x + 2$
$I_2: y = 3 - a$
$I_3: c = 0$
$I_4: w = 2 w$
$I_5: d = a - b$
$I_6: u = d w y$

DFG (Data Flow Graph)



For I_1, I_2, I_3

Occam

multi thread

relative ← short - 1, near - 2

CS + IP ← far - 3

MyConstant EQU 36

```
                                   .MODEL SMALL             ;select SMALL model
0000                               .DATA                    ;start data segment
0000 0030 R                        TABLE:  DW ONE           ;jump table
0002 0034 R                                DW TWO
0004 0038 R                                DW THREE
0000                               .CODE                    ;start code segment
                                   .STARTUP                 ;start program
0017 B4 01                 TOP:    MOV AH,1                 ;read key into AL
0019 CD 21                         INT 21H
001B 2C 31                         SUB AL,31                ;convert to BCD
001D 72 F9                         JB TOP                   ;if key < 1
001F 32 02                         CMP AL,2
0021 77 F4                         JA TOP                   ;if key > 3
0023 B4 00                         MOV AH,0                 ;double key code
0025 03 C0                         ADD AX,AX
0027 BE 0000 R                     MOV SI,OFFSET TABLE      ;address TABLE
002A 03 F0                         ADD SI,AX                ;form lookup address
002C 8B 04                         MOV AX,[SI]              ;get ONE, TWO or THREE
002E FF E0                         JMP AX                   ;jump to ONE, TWO or THREE
0030 B2 31                 ONE:    MOV DL,'1'               ;get ASCII 1
0032 EB 06                         JMP BOT
0034 B2 32                 TWO:    MOV DL,'2'               ;get ASCII 2
0036 EB 02                         JMP BOT
0038 B2 33                 THREE:  MOV DL,'3'               ;get ASCII 3
003A B4 02                 BOT:    MOV AH,2                 ;display number
003C CD 21                         INT 21H
                                   .EXIT
                                   END
```

OPCode — Mnemonic — Comments — Label

Call MyStr
Label
MyStr: I_1 I_2 RET

internal — external

INT (255 ... 0) Call label Call

256 vector interrupt

INT 21

DOS (Disk operating System)

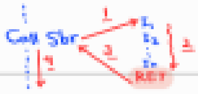
8086 Interrupt

```
0000 0064[                 BLOCK1 DW 100 DUP(?)        ;100 words for BLOCK1
           0000
               ]
00C8 0064[                 BLOCK2 DW 100 DUP(?)        ;100 words for BLOCK2
           0000
               ]
0000                       .CODE                       ;start code segment
                           .STARTUP                    ;start program
0017 8C D8                         MOV AX,DS           ;overlap DS and ES
0019 8E C0                         MOV ES,AX
001B FC                            CLD                 ;select auto-increment
001C B9 0064                       MOV CX,100          ;load counter
001F BE 0000 R                     MOV SI,OFFSET BLOCK1 ;address BLOCK1
0022 BF 00C8 R                     MOV DI,OFFSET BLOCK2 ;address BLOCK2
0025 AD                    L1:     LODSW               ;load AX with BLOCK1
0026 26:03 05                      ADD AX,ES:[DI]      ;add BLOCK2
0029 AB                            STOSW               ;save answer
002A E2 F9                         LOOP L1             ;repeat 100 times
                           .EXIT
                           END
```

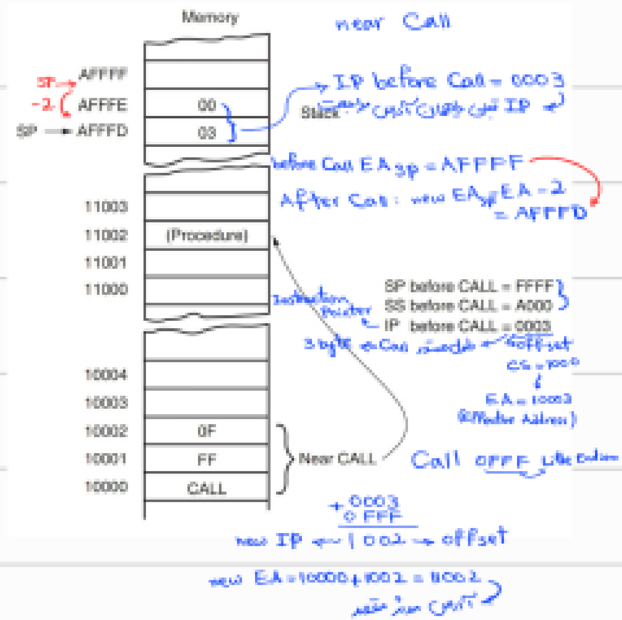
CLD Chang Direction

100 Decimal

procedure ← بعد از هر Call یک return داریم، چرا بعد از هر Call باید Return داشته باشیم و به برنامه اصلی برگردیم؟ (برعکس Jump) به خاطر نوع تعریف Call که باید پس از اینکه زیربرنامه به برنامه برگردد و ادامه دهد



near – } Call
far –



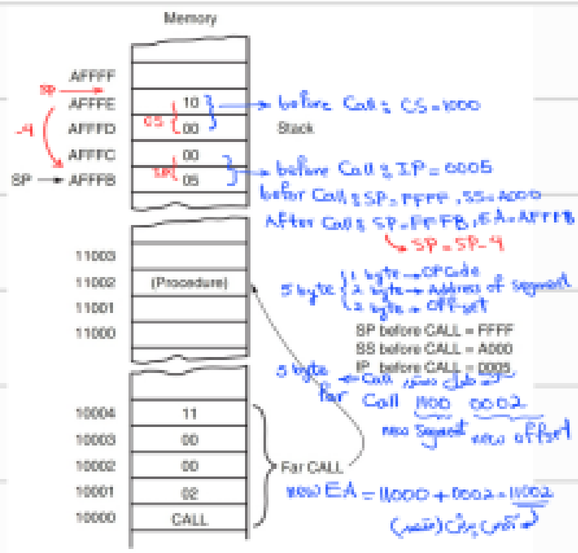
بعد از Call با دستور RET باید برگردیم ← از کجا بفهمیم که باید به کجا برگردیم؟ ذخیرهٔ IP (آدرس مراجعت) در Stack

← باید بعدش یک مقدار ثابت را اضافه کنیم تا برگشت به برنامه اصلی

* اگر near Call بود نیازی به ذخیره‌ی CS نیست ولی در far Call باید ذخیره کنیم

* SP (Stack Pointer) همیشه به بالای پشته اشاره دارد، برای نوشتن ابتدا باید مقدار آن را کم کنیم

* در far Call هم IP و Segment در Stack ذخیره می‌شوند!



جلسه بیست و چهارم ۱۴۰۳/۹/۲۴

- محل ذخیره پرچم ها ← Status Word (Flag Register)

```
 + 10101100
   10010111     } → 8 bit
 [1] 01000011
  ↳ Carry

 + 3496
   9284
 [1] 2780
  ↳ Carry
```

- پرچم رقم نقلی C: Carry اگر بعد از محاسبه رقم نقلی ایجاد شود پرچم Carry، 1 می شود

در جمع، رقم نقلی پرارزش‌ترین بیت است و بنابراین از آن غافل شد ولی به علت محدودیت ثبات ممکن است ذخیره نشود ← برنامه آن‌ها را با 1 کردن Carry اعلام می‌کند

بی توجهی به رقم نقلی باعث خطا در محاسبات می شود

```
رقم قرضی (Borrow)
    5 14
   3264
 - 2139
   1125
```

Processes (tasks)

در متن برنامه یک task انجام می‌دهد بعد به سراغ task بعدی می رود ← Context switching

برنامه موقع تغییر Context وضعیت آن را ذخیره می‌کند بعد به سراغ Process بعدی می رود ← Context saving

- بیت علامت S: Sign $\begin{cases} S=1 \rightarrow - \\ S=0 \rightarrow + \end{cases}$

- بیت ۰ Z: Zero اگر همه‌ی بیت‌های خروجی ۰ شوند (نتیجه ۰ شود) ← Z=1

این بیت برای مقایسه کردن بسیار مهم است

```
CMP AX, BX ↔ (AX - BX)
JZ L1
(Jump on Zero)
```

- بیت سرریز O: Overflow یا V

اگر فرض کنیم ثبات ها ۴ بیتی باشند، دامنه‌ی اعداد علامت دار ← بازه‌ی نمایش $[-8, +7]$

در نمایش مکمل ۲: بازه‌ی نمایش n بیت $[-2^{n-1}, +2^{n-1}-1]$

```
 + 0110     +6
   0101     +5
(-5) 1011   +11  → ⚠ overflow
```

حاصل جمع ۲ عدد مثبت، ۱ عدد منفی شده! ←

چرا سرریز رخ داده؟ چون حاصل جمع در ظرف اعداد نمی‌گنجد پس سرریز رخ می‌دهد

```
 + 1010    -6         + 1010     -6
   0101    +5           1100     -4
   1111    -1      (+6) 0110    -10
                         → overflow
```

در مکمل ۲، در جمع ۲ عدد مختلف العلامه هیچگاه سرریز رخ نمی‌دهد! (عامل اثبات)

پس زمانیکه overflow رخ می‌دهد حاصل محاسبات غلط خواهد بود و برنامه آن را اعلام می‌کند

- بیت توازن P: Parity ← برای تشخیص خطا

Even Parity (توازن زوج) تعداد بیت های ۱ ← زوج

Odd Parity (توازن فرد) تعداد بیت‌های ۱ ← فرد

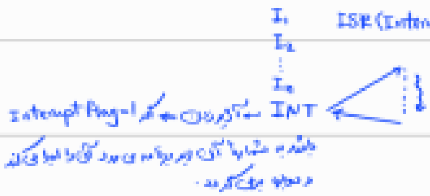
```
P
↓
0100 0001
فرستنده → گیرنده
0100 0011 → مقدار دریافت شده که دچار خطا شده است
```

برای چک کردن درستی داده به یک تعداد بیتی مشخص به گیرنده یک بیت توازن اضافه می‌کنیم به طوری که توازن زوج باشد

برای چک کردن کافی است XOR داده را محاسبه کنیم (برای توازن فرد ← نات XOR) ← تشخیص تعداد فرد خطا

اگر بخواهیم ۲ خطا تشخیص داده شود و ۱ خطا تصحیح شود باید از ۳ بیت توازن استفاده کنیم

- I: Interrupt flag ← اگر ۱ باشد یعنی وقفه مجاز است و اگر ۰ باشد وقفه غیرمجاز است



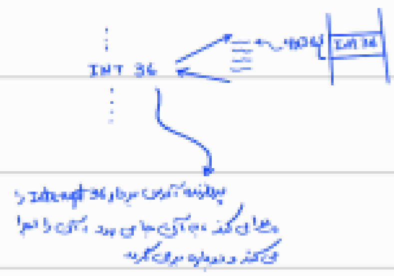
Interrupt (وقفه) ← اعلام نیاز یا رویداد به پردازنده بدون هماهنگی کامل محل اجرا و اجرای دستور کاربر

وقفه بیرونی

وقفه درونی: مثلا تقسیم بر ۰

فرق Jump با Interrupt: Jump ، deterministic است و توسط کاربر تصمیم‌گیری و نوشته می‌شود ولی Interrupt برنامه‌ریزی نمی‌شود ولی اتفاق آسنکرون (ناهنگام و ناخواسته‌ای) است

یا { Jump ← هنگام با برنامه (به اختیار برنامه‌نویس)
Interrupt ← ناهنگام برنامه (خارج از برنامه‌نویس) }



در Intel یک دستور Interrupt دارد که کاربر اجازه دارد از آن استفاده کند، و بردارهای آن را در حافظه قرار داده است

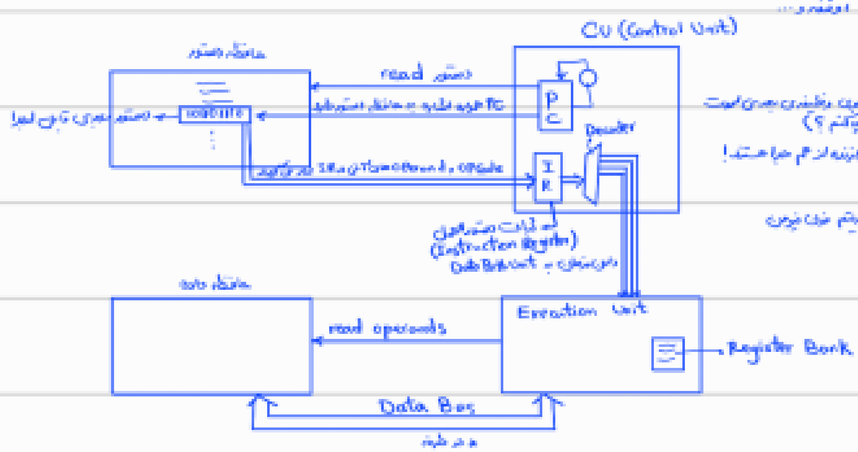
در پردازنده 80x86 : Interrupt = Indirect Call (پرش غیرمستقیم به یک زیر برنامه)

_ البته در Int پرچم‌ها ذخیره می‌شوند ولی در Call ذخیره نمی‌شوند

فرمان وقفه‌ها صبر می‌کنند اجرای دستور جاری تمام شود سپس اجرا می‌شوند (البته اگر اجازه‌ی اجرا داشته باشند) ← استثناء ← reset

در محاسبات بخش Co Processor : در آن هنگام وقفه داده‌های میانی ذخیره و اجرا جایی می‌رود تا دستور اجرا شود و دوباره در مکان‌های اجرای برنامه می‌آید ← به علت طولانی بودن زمان اجرای برنامه

دستورات کنترلی ← پرش‌های شرطی، غیرشرطی، وقفه و...



_ PC کاری به حافظه‌ی داده ندارد و فقط برای جست‌وجوی دستورات است (چرا؟)

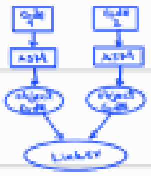
_ حافظه داده و دستور می‌توانند یکی باشند ولی در پردازنده‌ها جدا هستند!

ترتیب اجرا: ① Fetch ② PC update ③ Decode ④ Read Operands ⑤ Execute ⑥ Write Back

جلسه بیست و پنجم   اردیبهشت ۱۴۰۳

مزاحمت‌های زیر برنامه و رد کردن زیر برنامه‌ها

دلایل استفاده از زیر برنامه (SubRoutine): ۱- افزایش خوانایی برنامه ۲- استفادهٔ چندباره از یک زیر برنامهٔ پیاده‌سازی شده ۳- زیر برنامهٔ نوشته شده توسط دیگران مثلاً نوشتن یک برنامه با نوشتن Driver



فایده جداسازی زیر برنامه ← لزوماً Source Code نمی‌دهد بلکه Object Code برنامه را به Linker می‌دهیم
که کد حفاظت شود (به جهت حفظ مالکیت و کد منبع)

چرا Source Code خیلی داده نمی‌شود؟ مثل Windows به دلایل تجاری و مالی

پس Open Source چرا؟ مثلاً کدهایی داخل Github تبلیغ و رزومه‌سازی یا مشارکت جمعی، تسریع توسعه

؟ نوع پرش به زیر برنامه (Call) { Near - Stack / Far - IP+CS

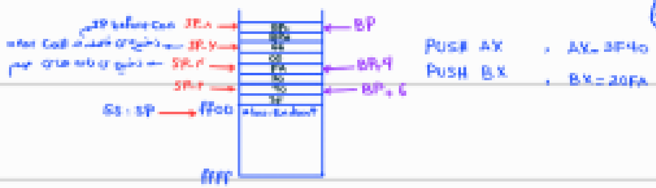
- در Intel به Interrupt رفتن، پردازنده ISR (Interrupt Service Routine) اجرا می‌شود.

- اگر بخواهیم مقادیر رجیسترها در زیر برنامه تغییر نکند، مقادیر آن‌ها را ذخیره کنیم چون زیر برنامه ممکن است آن‌ها را تغییر دهد
← این وظیفه برنامه‌نویس است که داده‌هایی را که می‌خواهد تغییر نکند در Stack ذخیره کند

- SP همیشه به بالای Stack اشاره می‌کند که آخرین متغیر ذخیره شده در آن است

Page 213, Intel microprocessor by Brey

```
0000 B8 001E        MOV   AX,30
0003 BB 0028        MOV   BX,40
0006 50             PUSH  AX          ;stack parameter 1
0007 53             PUSH  BX          ;stack parameter 2
0008 E8 0066        CALL  ADDM        ;add stack parameters
0071           ADDM PROC  NEAR
0071 55             PUSH  BP          ;save BP
0072 8B EC          MOV   BP,SP       ;address stack with BP
0074 8B 46 04       MOV   AX,[BP+4]   ;get parameter 1
0077 03 46 06       ADD   AX,[BP+6]   ;add parameter 2
007A 5D             POP   BP          ;restore BP
007B C2 0004        RET   4           ;return, dump parameters
007E           ADDM ENDP
```



- Near Call ← ذخیره IP در Stack
- Far Call ← هم IP و هم CS در Stack

علاوه بر ذخیره داده‌ها، گاهی global variable (متغیر مشترک) داریم که می‌خواهیم در زیر برنامه بگیریم. متغیرهای لازم آن را در Stack قرار داده‌ایم ...

چون امکان تغییر BP وجود دارد، این برنامه آن را داخل زیر برنامه تعریف و مقداردهی کرده است

چرا MOV BP,SP؟ برای احتیاط و استفاده‌های بعدی SP را ثابت نگه می‌داریم و با BP به عنوان اشاره‌گر به متغیرهای مورد نظر دسترسی داریم
- استفاده از BP به عنوان اشاره‌گر برای دسترسی به متغیرها در Stack

MOV AX,[BP+4] ← BP+4 آدرس ذخیره متغیر در AX

چرا RET 4؟ خالی کردن متغیرهای رفته در Stack (خالی کردن ۴ خانه در Stack، برگشتن SP به مقدار قبلی) چرا از BP استفاده کردیم (بدون SP)؟ چون BP به طور پیش‌فرض با Stack Segment کار می‌کند

Interrupt { - سخت‌افزاری / - نرم‌افزاری , Interrupt { - بیرونی / - درونی

وقفه (Interrupt) در مقابل سرکشی (Polling)

در وقفه هنگامی که دستگاه کار داشته باشد آن را خبر می‌کند (زنگ می‌زند) ولی در سرکشی پردازنده خودش به دستگاه‌ها سر می‌زند

مزیت سرکشی: Programmable است و به همین دلیل Deterministic (قطعی، معین) است یعنی ترتیب سرزدن به دستگاه‌های ورودی/خروجی از قبل توسط برنامه‌نویس تعیین شده است
← اولویت بخشی، ترتیب مراجعه به ورودی/خروجی‌ها

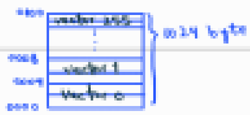
کاستی سرکشی: زمانی که صرف مراجعه به همه دستگاه‌ها می‌شود و از آنها استعلام می‌گیرد

* زمانی مناسب‌تر است به صورت سرکشی عمل کنیم که I/O ها خیلی با پردازنده کار داشته باشند و می‌خواهیم دائما و پردازنده داده دریافت کند

وقفه آسنکرون (ناهمگام) است و یک رخداد حادث می‌شود

وقفه:
- بیرونی ← خط درخواست وقفه در پردازنده هست IF (Interrupt Flag) که در پرچم‌ها نشان داده شد، پردازنده پاسخ آن را نمی‌دهد. خط درخواست را Mask کنیم (IF را صفر کنیم) به غیر از وقفه‌های خاص: NMI (Non-Maskable Interrupt)، Reset
- درونی ← مثلاً تقسیم بر ۰ یا overflow ← پردازنده چه کار می‌کند؟ از جایی که آدرس آن مشخص می‌شود که جدولی است به ISR (Interrupt Service Routine) مراجعه می‌کند و به سراغ روال وقفه اقدام می‌رود

← 256 Interrupt Vector: اسم از وقفه‌های مختلف و Programmable
← قابل تعریف ISR



INT 0 → ISR 0
INT 1 → ISR 1
⋮
INT 255 → ISR 255

تفاوت Interrupt, Call:

- در پردازنده Intel می‌توان ذخیره CS در برنامه را به طور خودکار انجام داد و آن را تنها با عدد صدا زد

Page 212

Number	Address	Microprocessor	Function
0	0H–3H	All	Divide error
1	4H–7H	All	Single-step
2	8H–BH	All	NMI pin ← Non-Maskable Interrupt
3	CH–FH	All	Breakpoint
4	10H–13H	All	Interrupt on overflow
5	14H–17H	80186–Core2	Bound instruction
6	18H–1BH	80186–Core2	Invalid opcode
7	1CH–1FH	80186–Core2	Coprocessor emulation
8	20H–23H	80386–Core2	Double fault
9	24H–27H	80386	Coprocessor segment overrun
A	28H–2BH	80386–Core2	Invalid task state segment
B	2CH–2FH	80386–Core2	Segment not present
C	30H–33H	80386–Core2	Stack fault
D	34H–37H	80386–Core2	General protection fault (GPF)
E	38H–3BH	80386–Core2	Page fault
F	3CH–3FH	—	Reserved
10	40H–43H	80386–Core2	Floating-point error
11	44H–47H	80486SX	Alignment check interrupt
12	48H–4BH	Pentium–Core2	Machine check exception
13–1F	4CH–7FH	—	Reserved
20–FF	80H–3FFH	—	User interrupts ← برای استفاده کاربر

در Interrupt ، Status Register در Stack ذخیره می‌شود، یک بیت وقفه را ۰ می‌کند (آن را Mask می‌کند) تا در حین اجرای آن وقفه، وقفه دیگری رخ ندهد، چون پردازنده وسط یک وقفه، وقفه دیگری را نمی‌پذیرد (nested interrupt). بعد CS و IP را در Stack می‌گذارد و بعد CS ، IP جدید را می‌خواند و به آدرس ISR می‌رود، شبیه به Far Call که رجیسترها ذخیره می‌شوند

Whenever a software interrupt instruction executes, it (1) pushes the flags onto the stack, (2) clears the T and I flag bits, (3) pushes CS onto the stack, (4) fetches the new value for CS from the interrupt vector, (5) pushes IP/EIP onto the stack, (6) fetches the new value for IP/EIP from the vector, and (7) jumps to the new location addressed by CS and IP/EIP.

The INT instruction performs as a far CALL except that it not only pushes CS and IP onto the stack, but it also pushes the flags onto the stack. The INT instruction performs the operation of a PUSHF, followed by a far CALL instruction.

Notice that when the INT instruction executes, it clears the interrupt flag (I), which controls the external hardware interrupt input pin INTR (interrupt request). When I = 0, the microprocessor disables the INTR pin; when I = 1, the microprocessor enables the INTR pin.

* فرق Interrupt با Far Call: در Interrupt پرچم‌ها (Status Register) ذخیره می‌شود.

- طول Interrupt ، 2 byte و Far Call ، 5 byte است.

اگر وسط محاسبات باشیم و Non-Maskable Interrupt رخ دهد پردازنده باید چه کند؟ باید تمام داده‌های مهم را در Stack قرار دهد، بعد آن‌ها را برگردانیم (Pop ، Push)

Push AX
Push BX
⋮
Pop BX
Pop AX

Pop ، Push به ترتیب برعکس

جلسه بیست و ششم   ۳ مارس ۱۴۰۳

TABLE 6–1 Conditional jump instructions.

(شروط انجام دستور → Tested Condition)

Assembly Language	*Tested Condition*	*Operation*
JA	Z = 0 and C = 0	Jump if above
JAE	C = 0	Jump if above or equal
JB	C = 1	Jump if below
JBE	Z = 1 or C = 1	Jump if below or equal
JC	C = 1	Jump if carry
JE or JZ	Z = 1	Jump if equal or jump if zero
JG	Z = 0 and S = 0	Jump if greater than
JGE	S = 0	Jump if greater than or equal
JL (علامت دار)	S != O	Jump if less than
JLE	Z = 1 or S != O	Jump if less than or equal
JNC	C = 0	Jump if no carry
JNE or JNZ	Z = 0	Jump if not equal or jump if not zero
JNO	O = 0	Jump if no overflow
JNS	S = 0	Jump if no sign (positive)
JNP or JPO	P = 0	Jump if no parity or jump if parity odd
JO	O = 1	Jump if overflow
JP or JPE	P = 1	Jump if parity or jump if parity even
JS	S = 1	Jump if sign (negative)
JCXZ	CX = 0	Jump if CX is zero
JECXZ	ECX = 0	Jump if ECX equals zero
JRCXZ	RCX = 0	Jump if RCX equals zero (64-bit mode)

(JL, JLE: کوچک‌تر [illegible] منفی شود)

پرش های شرطی

	اعداد علامت دار	اعداد بدون علامت
بزرگتر	JG	JA
کوچکتر	JL	JB
بزرگتر یا مساوی	JGE	JAE
کوچکتر یا مساوی	JLE	JBE
مساوی	JE , JZ	JE , JZ
صفر بودن نتیجه		
منفی بودن نتیجه		
سرریز (overflow)		
بیت نقلی (Carry)		

(JE , JZ ← برای هر دو [illegible])

* چرا ؟ تا دستور JG ، JA ؟ چون پرش در اعداد بدون علامت با شروطی انجام می شود که در اعداد علامت دار متفاوت است.

* مقایسه معادل تفریق و مراجعه به پرچم هاست. تفریق معادل جمع مکمل ۲ است.

70٪ پردازنده سیستم های نهفته ← ARM (Advanced Risc Machine)

سیستم های نهفته (Embedded System) ← سیستم نهفته یا تعبیه شده

به نام جدیدتر: Cyber Physical Systems

Patterson , Hennessy | computer organization | page 256

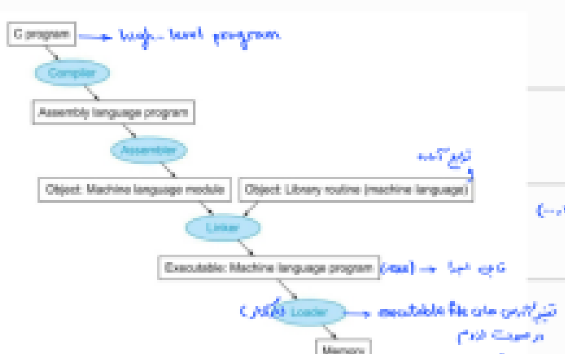


FIGURE 2.21 A translation hierarchy for C.

* همه برنامه relocatable در loader ساده تر ← سرعت اجرای برنامه بیشتر

Interpret ← خط به خط ترجمه و اجرا می شود مثل Python

Compile ← کل برنامه یکجا ترجمه می شود مثل C , C++ ← افزایش سرعت اجرا / هم [illegible] استفاده [illegible]

شبیه ساز پردازنده Open Source و دانلود پردازنده ؟ مثلا دانلود پردازنده Simple Scaler یا Open RISC ← توصیف به زبان C

← اجرا روی پردازنده PC ، تبدیل PC به Super Scaler دارای زبان C ، دستوراتی که Compile می شوند ← توصیف به زبان Verilog

← زبانهایی که Verilog ، VHDL ، Chisel ... ← به سمت FPGA و تولید IC ← [illegible] سخت افزار [illegible]

دانلود پردازنده = دانلود کد Verilog آن که می توانیم آن را تغییر دهیم و چک می کنیم

← ترجمه زبان اسمبلی به زبان ماشین

کار Assembler ← LC (Location Counter) ← شمارنده آدرس دستورالعمل Assembly

↙ Symbol table ← جدول نمادها

تبدیل کدهای OPCode ← تعریف Instruction Set کد باینری در یک جدول

↙ Directive table ← دستورالعمل های خاص اسمبلی

نحوه کارکردن Assemble : ① هنگام خواندن از فایل و پایان هنگام رسیدن به label های تکراری [illegible]

Two Pass Assembler ← اسمبلر دو گذره

مشکل اسمبلرها: اگر reference به سمت بالا باشد تکلیف opcode مشخص نیست

در اسمبلر دو مرحله‌ای تکلیف opcode را معین کند

(Pass 1)
گذر اول: خواندن فایل منبع از ابتدا تا انتهای فایل برنامه، در این مرحله به دنبال پیدا کردن جایگاه label ها و تعریف آنها و محاسبه‌ی ... تعیین تکلیف label ها

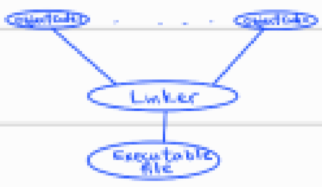
① اگر label تعریف نشده باشد آن را در جدولی قرار می‌دهد و اگر تعریف شده باشد آدرس آن را پیدا می‌کند، opcode و LC و اندازه‌ی دستور را تعیین کند

در مرحله دوم opcode مشخص باشد که ... آدرس ...

One Pass Assembler ← اسمبلر یک مرحله‌ای ... label ... unknown ... label ها را تکمیل ... تعریف شده

```
37 BE  AB ← 68
68 BNE AB → 37
85 BE  AB → 68
 :
 AB: ...
```

linked list تعریف می‌کند که هنگام تعریف label ببینیم کدام از آنها را تکمیل کنیم



Linker ← ترکیب کدهای مختلف (Object codes) و ساخت فایل Object file

به طور سنتی تمام Object code ها توسط Linker، link می‌شوند و در فایل executable file ریخته می‌شود

و توسط loader در حافظه ریخته و اجرا می‌شود ← Statically Linkable File یا Static Linking (پیوند ایستا)

اشکال‌ها:

۱. اگر فایل object دستخوش تغییر شود یا ... باید کل ... compile شده و در تمام فایل‌ها لینک شده اصلاحیه ... یک فایل اجرایی جدید بسازیم

و فایل object ... ← ...

۲. تمام Object code ها لینک می‌شوند و در RAM ریخته می‌شوند ← اشغال حجم زیادی از حافظه

پس شرکت Microsoft گفت لازم نیست تمام Object code ها را لینک کنیم و لینک کردن را به صورت تکه‌ای انجام می‌دهیم ← Dynamically Linked Libraries (DLL)

یعنی همه‌ی دستورات را لینک نمی‌کند بلکه بعضی از آنها را در فایل exe قرار می‌دهد و آنهایی که قرار است reference به توابع کتابخانه‌ای داشته باشند علامت‌گذاری می‌کند

که loader هنگام برخورد دچار Indirection یا Interrupt می‌شود یعنی loader هنگامی که به آن دستورات می‌رسد می‌بیند که آن وجود ندارد و شبیه به یک Indirect Call

آدرس آن را پیدا کرده و نقطه‌ی شروع آن تابع در حافظه load و مورد استفاده قرار می‌گیرد ← اولین بار یک مقدار طول می‌کشد ولی بعدش در حافظه وجود دارد

پس فرایند کلی که exe یک برنامه در همان ابتدا لینک می‌شود مثل word که تمام فایل‌های آن کم حجم می‌گیرد و لینک نمی‌شود ولی هنگام استفاده از هر کدام فایل مورد نظر در حافظه قرار می‌گیرد

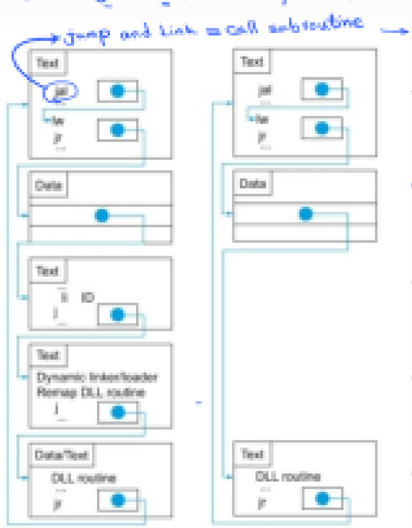


FIGURE 2.22 Dynamically linked library via lazy procedure linkage.

← اگر آدرس یک دستور را نداشت برای بار اول به صورت indirect آن را در حافظه قرار می‌دهد و دفعات بعدی آدرس آن را می‌دهد

Driver ← نرم‌افزار واسط کنترل‌کننده سخت‌افزار توسط سیستم عامل

DLL یا Driver ← ... بارگذاری یک برنامه ← ... از حافظه، Driver ... توسط loader ... به حافظه برسد آن را می‌برد

DLL High Jacking ← ... DLL ... توسط malware ... سیستم ... مشکل ایجاد می‌کند

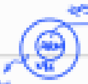
(Virtualization)
مجازی سازی در سطوح مختلف رخ می‌دهد — کامپایلر ← مدل پردازنده ← مدل ISA
- مجموعه دستورات
- مدهای آدرس‌دهی
- ثبات ها
- کدگذاری دستورات
- آدرس‌دهی حافظه

زبان قابل فهم توسط پردازنده ← زبان ISA آن
* ISA ← مثل یک رابط برنامه‌نویسی برای یک پردازنده

دستور اجرای برنامه نوشته شده در حافظه توسط پردازنده ← دستوراتی که قابل فهم برای پردازنده

اگر زبان قابل دسترسی x86 و پردازنده ARM باشد چه باید بکنیم؟ مثلاً یک کد exe یا bin داریم که می‌خواهیم آن را در یک گوشی اجرا کنیم. چاره چیست؟

① کد تنها که باینری داشته باشیم داریم پردازنده ARM را با یک ماشین مجازی x86 معادله کنیم و کدهای آن را می‌بینیم ← دستور ARM به‌عنوان x86



گویی پردازنده واقعی با یک لایه نرم‌افزاری پوشانده می‌شود!

می‌توانیم برنامه C طراحی کنیم که دستورات هر پردازنده را به دستورات پردازنده دیگر تبدیل کند ← مثل ابزار Qemu

کاربرد: اجرای یک برنامه روی پردازنده جدید به علت شبیه‌سازی یک پردازنده در پلتفرم دیگری، تحلیل و مهندسی معکوس بدافزارها روی آن مثلاً کامپیوتر ios

Virtualization پردازنده: اجرای یک پردازنده دیگر توسط پردازنده خودمان — ISA ما ... ماشین مجازی ISA دیگری

مجازی‌سازی تنها در پردازنده محدود نمی‌شود بلکه برای سیستم عامل هم مورد استفاده قرار می‌گیرد. مثلاً اجرا کردن linux روی windows و برنامه‌های بالاتر docker, container

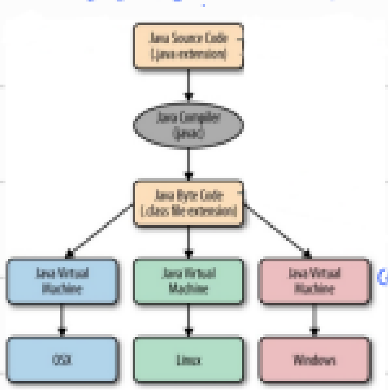
(مجازی سازی کارکردهای شبکه)
در شبکه ← NFV: Network Function Virtualization ← شبیه‌سازی نرم‌افزاری تجهیزات مثل switch و انجام کارهای سخت‌افزاری به صورت نرم‌افزاری
← انجام کارهای سخت‌افزاری روی سرور مثل کارهای routing، firewall و ... جهت روتر و کارت شبکه

زبان Java ← Multi platform، ضمانت شرکت Sun Microsystem برای اجرای برنامه Java در تمام پلتفرم‌ها
← مستلزم یک ماشین مجازی برای اجرای Java در پردازنده مقصد
JVM (Java Virtual Machine) ← با interpreter و تولید ByteCode و اجرای آنها در سمت پردازنده هدف

برای اجرای برنامه‌های تحت وب ... یک لایه روی محیط داشته باشیم که به زبان مشترک همه تکویر ... زبان مشترک ← اسمش Byte codes of Java

ByteCode ← یک ماشین خاص و با زبان اسمبلی و zero-Address

Byte code ← خروجی برنامه کامپایل شده Java / ماشین مجازی تنها Bytecode را از کاربر می‌گیرد و آن را اجرا می‌کند تا کاربر درگیر پردازنده نشود



Android ← ماشین مجازی برای اجرای Bytecode های Java

Byte code ← Stack Machine (Zero-Address) و طول دستورات (opcode) ← 1 Byte

دور کردن application ها از پردازنده آن‌هاست: Bytecode به واسطه JVM به کد مقصد ترجمه می‌شود (JVM)

ARM : Advanced Risc Machine

فصل آخر: پردازنده ARM

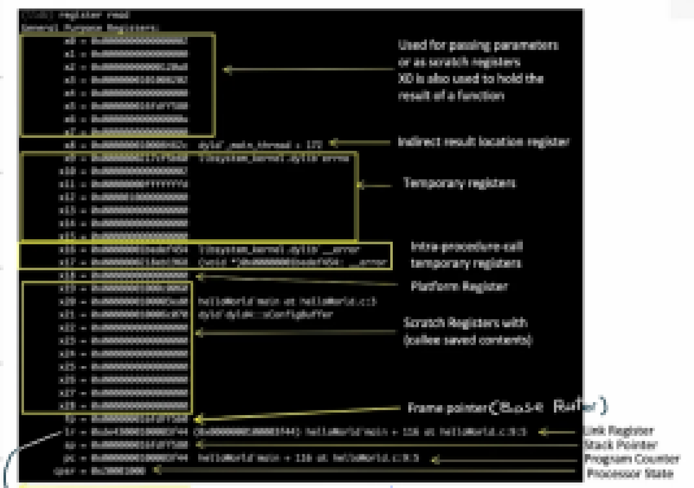
ARM یک نوع پردازنده با ISA یکسان نیست. چون شرکت های متفاوت license خریداری کرده، ویژگی های لازم را اضافه می‌کنند

ویژگی ابتدا: طول دستورات 32 bit ، 3 operand سه ARM یک پردازنده تقریبا RISC
(هرچند no-1, 2 operand هم داریم)

- رجیسترهای زیاد
- طول دستورات ثابت
- تعداد دستورات نسبتا کم
- مدهای آدرس‌دهی نسبتا کم

در ARM ، یک رجیستر کاملا 0 (Zero) هم وجود ندارد ← افزایش سرعت و کارایی

جهت انتقال از راست به چپ به جز STR (store)



return address ← ذخیره آدرس بازگشت هنگام برگشت به برنامه اصلی (آدرس بازگشت)

Shift Instructions

- LSL: logical shift left
  Example: `LSL R0, R7, #5 ; R0=R7 << 5`
- LSR: logical shift right
  Example: `LSR R3, R2, #31 ; R3=R2 >> 31`
- ASR: arithmetic shift right (منتشر کردن بیت علامت)
  Example: `ASR R9, R11, R4 ; R9=R11 >>> R4` (قسمت 8 بیتی)
- ROR: rotate right
  Example: `ROR R8, R1, #3 ; R8=R1 ROR 3`

Shift Instructions: Example 2

- **Register** shift amount (uses low 8 bits of register)
- Shift amount: 0-255

Source registers

R8	0000 1000	0001 1100	0001 0110	1110 0111
R6	0000 0000	0000 0000	0000 0000	0001 0100

Assembly code		Result			
LSL R4, R8, R6	R4	0110 1110	0111 0000	0000 0000	0000 0000
ROR R5, R8, R6	R5	1100 0001	0110 1110	0111 0000	1000 0001

Shift Instructions: Example 1

- **Immediate** shift amount (5-bit immediate)
- Shift amount: 0-31

Source register

R5	1111 1111	0001 1100	0001 0000	1110 0111

Assembly Code		Result			
LSL R0, R5, #7	R0	1000 1110	0000 1000	0111 0011	1000 0000
LSR R1, R5, #17	R1	0000 0000	0000 0000	0111 1111	1000 1110
ASR R2, R5, #3	R2	1111 1111	1110 0011	1000 0010	0001 1100
ROR R3, R5, #21	R3	1110 0000	1000 0111	0011 1111	1111 1000

(R0 ← R5 << 7, R1 ← R5 >> 17, R3 = R5 >> 3; 21 bit)

* ROL R0, R0, n = R0 ← (R0 × 2) % (2^n − 1) → مثال: 0110 ← 6 → 0011 = (6×2)%(2^4−1) = 5

Multiplication

- **MUL:** 32 × 32 multiplication, 32-bit result
  `MUL R1, R2, R3`
  Result: R1 = (R2 x R3)$_{31:0}$ (signed doesn't matter)
- **UMULL:** Unsigned multiply long: 32 × 32 multiplication, 64-bit result
  `UMULL R1, R2, R3, R4` → 4 operands
  Result: {R1,R4} = R2 x R3 (R2,R3 unsigned)
- **SMULL:** Signed multiply long: 32 × 32 multiplication, 64-bit result
  `SMULL R1, R2, R3, R4` → 4 operands
  Result: {R1,R4} = R2 x R3 (R2,R3 signed)

Logical Instructions: Examples

Source registers

R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code		Result			
AND R3, R1, R2	R3	0100 0110	1010 0001	0000 0000	0000 0000
ORR R4, R1, R2	R4	1111 1111	1111 1111	1111 0001	1011 0111
EOR R5, R1, R2	R5	1011 1001	0101 1110	1111 0001	1011 0111
BIC R6, R1, R2	R6	0000 0000	0000 0000	1111 0001	1011 0111
MVN R7, R2	R7	0000 0000	0000 0000	1111 1111	1111 1111

```
LDR R3, [R2, #8]
```
→ Base index : R2 + 8 → R3

```
STR R7, [R5, #0x54]
```
R5 + 54H (84) → R7

ARM Condition Flags

Flag	Name	Description
N	Negative	Instruction result is negative
Z	Zero	Instruction results in zero
C	Carry	Instruction causes an unsigned carry out
V	oVerflow	Instruction causes an overflow

- Set by ALU (recall from Chapter 5)
- Held in *Current Program Status Register (CPSR)*

در پردازنده ARM هم رجیستر وضعیت داریم

```
CMP R5, R6
CMP 0, R5-R6
```

در ARM تمام دستورات ۳ عملوندی نیستند مثلا دستورات ۲ یا ۱ عملوند هم پیاده شده

Unconditional Branching (B)

جهش jmp در x86 یا پرش بدون شرط

ARM assembly

```
    MOV R2, #17         ; R2 = 17
    B   TARGET          ; branch to target
    ORR R1, R1, #0x4    ; not executed

TARGET
    SUB R1, R1, #78     ; R1 = R1 + 78
```

jmp = B

Condition Mnemonics

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\bar{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\bar{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\bar{N}$
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	$\bar{V}$
1000	HI	Unsigned higher	$Z\bar{C}$
1001	LS	Unsigned lower or same	Z OR $\bar{C}$
1010	GE	Signed greater than or equal	$N\oplus V$
1011	LT	Signed less than	$N\oplus V$
1100	GT	Signed greater than	$Z(N\oplus V)$
1101	LE	Signed less than or equal	Z OR $(N\oplus V)$
1110	AL (or none)	Always / unconditional	Ignored

بایت ۴ JMP نیت محتوای از دستورات شرطی استفاده کرد بدون اجازه شرط دستورات ARM در پردازنده های

Condition Mnemonics

- Instruction may be ***conditionally executed*** based on the condition flags
- Condition of execution is encoded as a ***condition mnemonic*** appended to the instruction mnemonic

  Example:
  ```
  CMP   R1, R2
  SUBNE R3, R5, R8
  ```
  - **NE**: not equal condition mnemonic
  - SUB will only execute if R1 ≠ R2 (i.e., Z = 0)

Example:

```
CMP   R5, R9          ; performs R5-R9
                      ; sets condition flags

SUBEQ R1, R2, R3      ; executes if R5==R9 (Z=1
ORRMI R4, R0, R9      ; executes if R5-R9 is
                      ; negative (N=1)
```

Addressing modes of x86

Mode Name	Description or algorithm	Examples
Immediate	X	mov eax, 0x120
Register	R	mov eax, ebp
Register indirect	[R]	mov eax, [ebp]
Stack	[B]	pop ecx
Based	[B + disp]	mov eax, 0x2000[ebx]
Indexed	[I*scale + disp]	mov eax, 0x14[2*edi]
Based without scaled index and displacement	[B + I + disp]	mov eax, 0x14[ebx + 2*edi]
Based with scaled index and displacement	[B + I*scale + disp]	mov eax, 0x14[ebx + 2*edi]
Relative	[PC + disp]	jmp 0x120

Addressing modes of ARM

Name	Alternative Name	ARM Examples
Register to register	Register direct	MOV R0, R1
Absolute	Direct	LDR R0, MEM
Literal	Immediate	MOV R0, #15 ADD R1, R2, #12
Indexed, base	Register indirect	LDR R0, [R1]
Pre-indexed, base with displacement	Register indirect with offset	LDR R0, [R1, #4]
Pre-indexed, autoindexing	Register indirect pre-incrementing	LDR R0, [R1, #4]!
Post-indexing, autoindexed	Register indirect post-increment	LDR R0, [R1], #4
Double Reg indirect	Register indirect Register indexed	LDR R0, [R1, R2]
Double Reg indirect with scaling	Register indirect indexed with scaling	LDR R0, [R1, r2, LSL #2]
Program counter relative		LDR R0, [PC, #offset]

Underlying design principles, as articulated by Hennessy and Patterson:

1. Regularity supports design simplicity
2. Make the common case fast
3. Smaller is faster
4. Good design demands good compromises

if Statement

C Code

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

f = f - i;
```

ARM Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

    CMP R3, R4      ; set flags with R3-R4
    BNE L1          ; if i!=j, skip if block
    ADD R0, R1, R2  ; f = g + h

L1
    SUB R0, R0, R2  ; f = f - i
```