Computer Architecture

دکتر زرندی

★ جلسه اول – یکشنبه ۱۶ بهمن ★

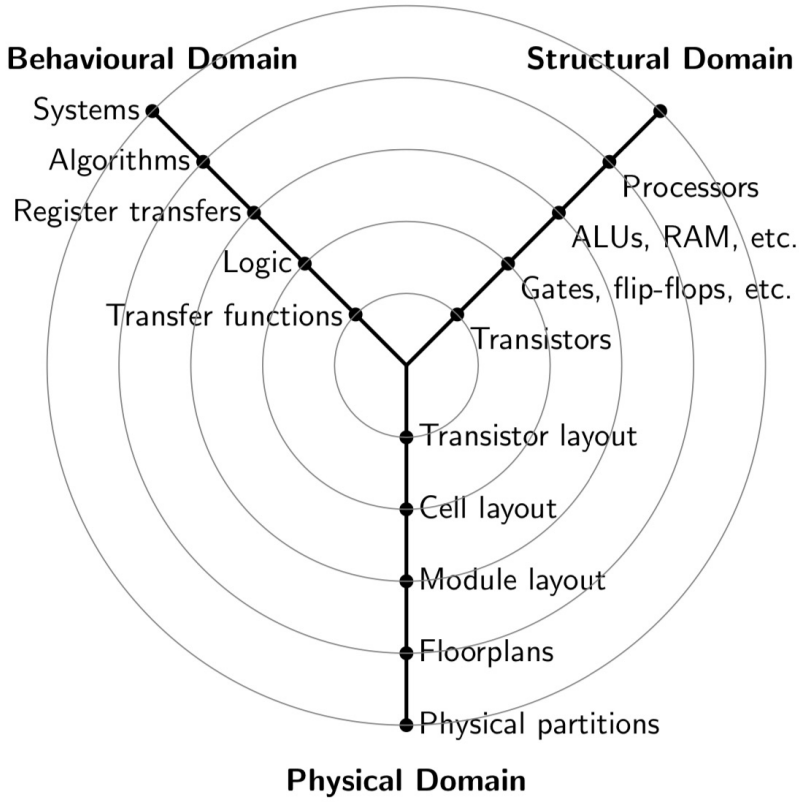
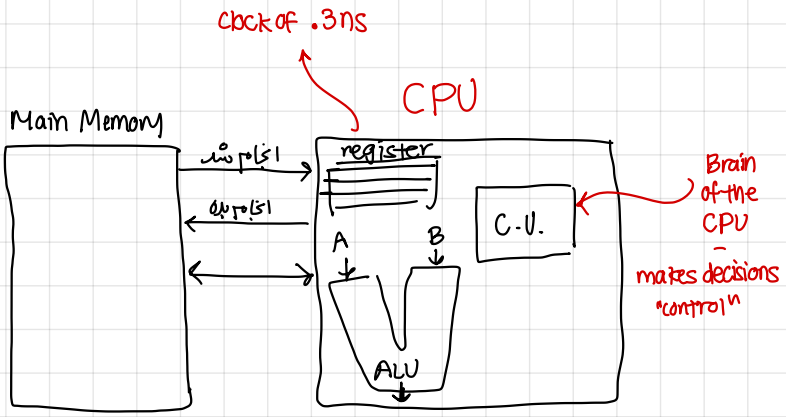
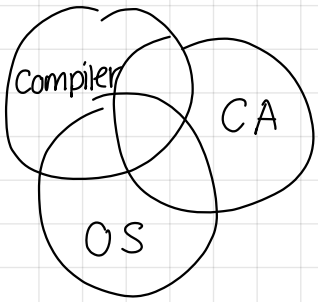


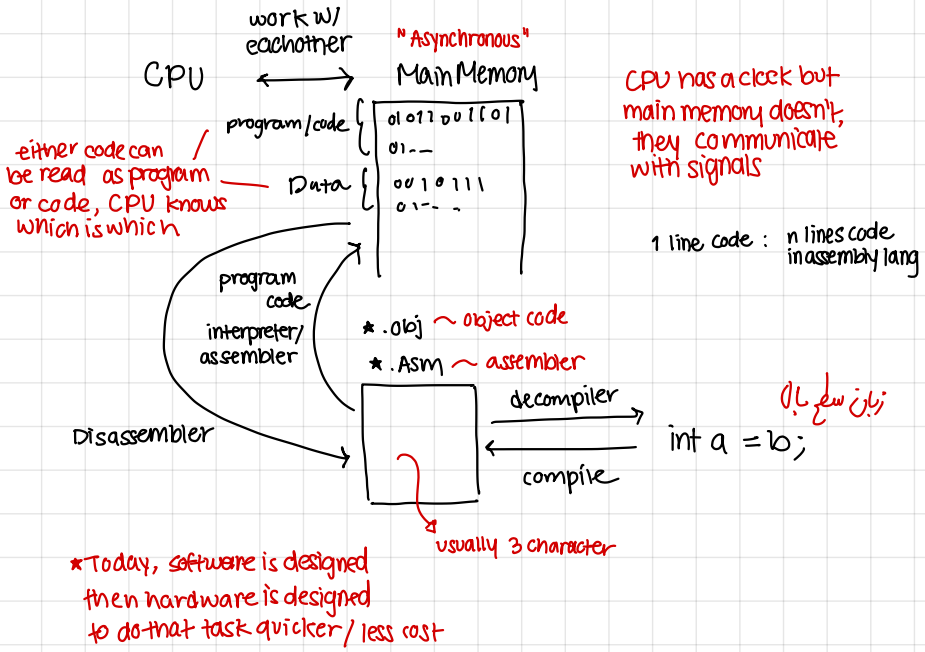
Figure 1: Gajski-Kuhn Y-chart

our task:
get list of instructions → design optimal CPU



RAM is an implementation of Main Memory





1.

memory

- Memory hierarchy
- RAM and ROM
- SRAM and DRAM
- Cache

★ جلسه دوم – سه‌شنبه ۱۸ بهمن ★

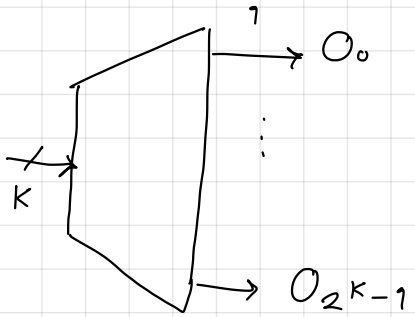
⚠ هیچ ثباتی بدون پایه load کشیده نشود

⚠ هیچ ۲ خروجی مستقیم هم وصل نشوند

⚠ مراقب z باشید (z تولید نکنید)

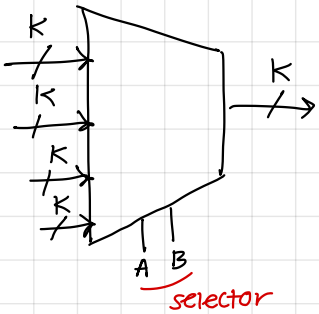
"تبدیل نرم‌افزار به سخت‌افزار"

Decoder



"سوزنبان"

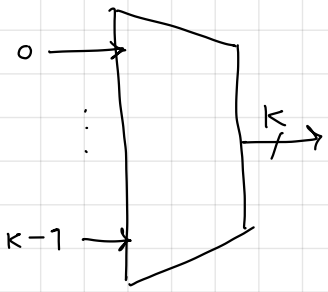
multiplexer



Encoder (priority encoder)

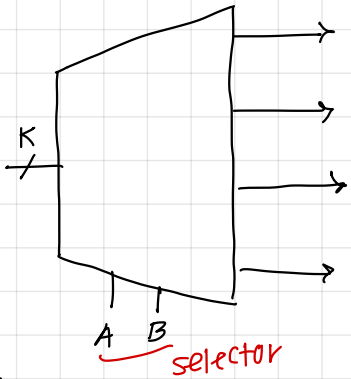
↳ used frequently for decreasing wires (lessens chances of error)

↳ usually used by non-professionals, covers up possible bugs



DeMUX

همه خروجی‌ها به جز یکی z می‌شوند!



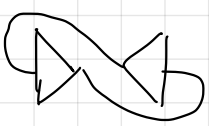
★ هزینه و تاخیر FF بیشتر از latch است پس تنها در مواقع ضروری از FF استفاده می‌شود

↳ FF برای کنترل است.

↳ زمانی که برای ما مهم نیست کی و چگونه داده ذخیره شود از latch استفاده می‌شود

↳ زمانی که feedback داریم ← احتمالاً FF

نمی‌توان با مدارهای منطقی تعریف کرد اما با بررسی در level ترانزیستوری می‌توان دید چطور یک مقدار را نگه می‌دارد



↳ سرعت بالا + گران + توان مصرف بالاتر + مساحت بیشتر

SRAM ← با دو گیت not ساخته می‌شود (Static)

DRAM ← با خازن ساخته می‌شود (Dynamic)

↳ مساحت کمتر + ارزان + مصرف توان بالا + قابلیت refresh

Main Memory

★ جلسه سوم – یکشنبه ۲۳ بهمن ★

★ The CPU works with the main memory

CPU ⟷ main memory

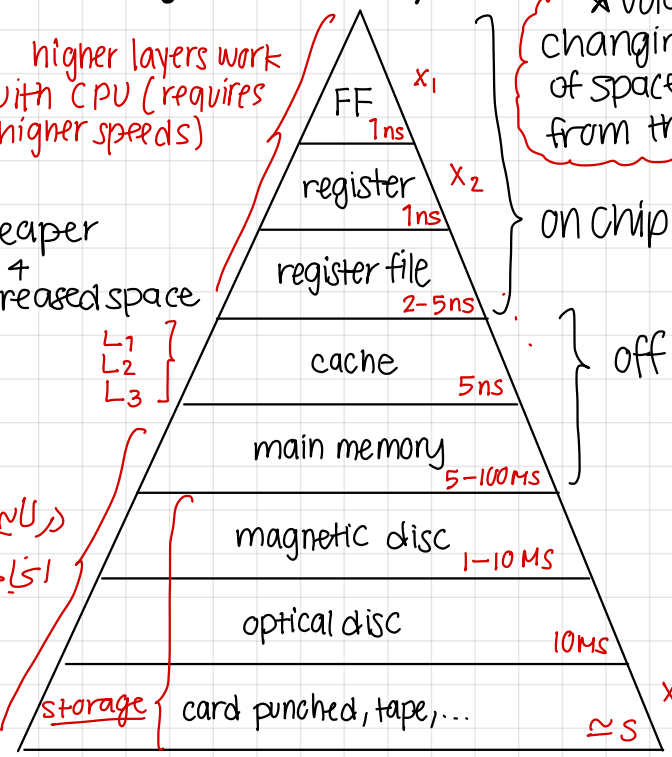
★ Booting a computer:
BIOS → Hard disk (find OS) → CPU

★ choosing type of memory for a specific CPU/task?
buy a proportion of each type of memory, higher layers for better speeds, lower levels for increased space, use each type of memory for most efficient function

memory hierarchy

جابه‌جایی سخت افزاری انجام میشه ← higher layers work with CPU (requires higher speeds)

★ volatile data is constantly changing due to small amounts of space and is sometimes rewritten from the non-volatile data



increased speed + less delay + چگالی بیتی بیشتر

cheaper + increased space

L1 L2 L3

در لایه‌های پایین جابه‌جایی ها نرم‌افزاری (instruction done with clock) انجام می‌شود.
↳ تاخیر کلاک در برابر تاخیر حافظه ناچیز است

$T_1 < T_2 < \dots < T_n$ ← تاخیر
$C_1 > C_2 > \dots > C_n$ ← قیمت هر بیت
$\varepsilon = h_1 < h_2 < \dots < h_n = 1$ ← احتمال پاسخ‌دهی

چگالی بیتی: تعداد بیتی که در یک واحد سطح جا می‌شود bit/area

ppc = performance per cost = $\frac{\text{speed}}{\text{cost}}$

↳ a desktop computer requires high speed + low cost

h = hit rate = احتمال پاسخ‌دهی = $\frac{\text{تعداد موفقیت‌ها}}{\text{تعداد درخواست‌ها}}$ → هرچی حجم حافظه بیشتر باشد احتمال موفقیت بیشتر

پاسخ تقریبی: زمان چک کردن وجود در لایه قبلی محاسبه نشده (تقریبا ناچیز است)

Average Access Time

(I) $h_1t_1 + (1-h_1)(h_2t_2 + (1-h_2)(h_3t_3 \dots))$

(II) $h_1t_1 + (1-h_1)(t_1 + h_2t_2 + (1-h_2)(t_2 + \dots))$ — h_3t_3

(III) $t_1 + (1-h_1)(t_2 + (1-h_2)(t_3 + \dots))$ → با دو تحلیل متفاوت، هردو پاسخ درست و دقیقی به ما می‌دهند

↳ we check each layer for the data, starting at the top of the pyramid until the last layer (where h = 1)

+ زمان تلف شده بیتی چک کردن لایه قبلی حساب شود

مثال: در یک کامپیوتر تاخیر دسترسی به حافظه 7ms است. چنانچه از یک حافظه‌مان با تاخیر 10ns و نرخ موفقیت 95٪ استفاده شود...
الف) متوسط زمان دسترسی؟ ب) سریع؟

(ب) $T = t_1 = 7000$ ns

سریع = $\frac{\text{تاخیر قبلی}}{\text{تاخیر جدید}} = \frac{60}{1000} = .06$

(I) $T = h_1t_1 + (1-h_1)(h_2t_2)$
$= (.95)(10) + (.05)((7)(1000)) = 59.5$ ns

(II) $T = h_1t_1 + (1-h_1)(t_1 + h_2t_2)$
$= (.95)(10) + (.05)(10 + (7)(1000)) = 60$ ns

سریع‌تر > 1 — بی‌اثر $= 1$ — کند‌تر < 1

نهایتا speed up ← $S = \frac{t_{old}}{t_{new}} = \frac{t_{old}}{\alpha\frac{t_{old}}{k} + (1-\alpha)t_{old}} = \frac{1}{\frac{\alpha}{k} + (1-\alpha)}$

★ جلسه چهارم – سه‌شنبه ۲۸ بهمن ★

آدرس data خواسته شده را داریم و با آدرس آن را پیدا می‌کنیم

writting is significantly slower than reading from a drive

"random" here means direct – can be accessed from any point

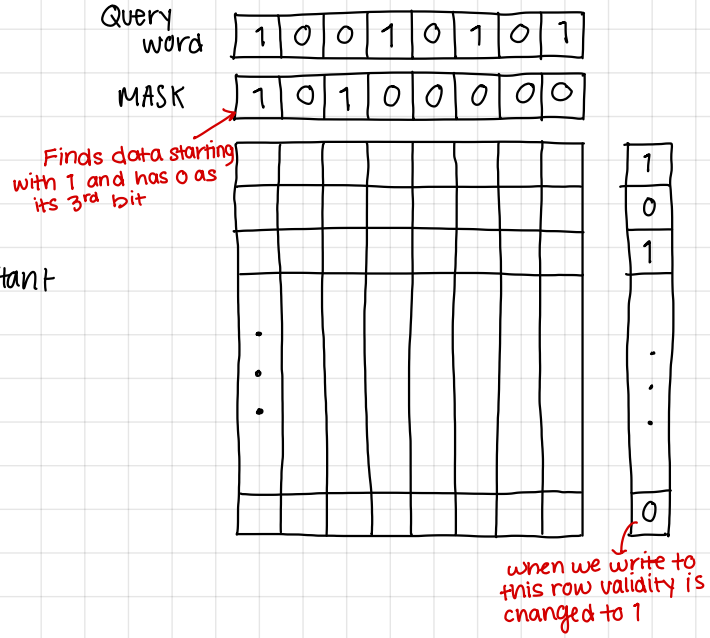
main memory (organized using memory arrays)
شامل حافظه آرایه‌ای درون تراشه + درون board

- Direct Access (دسترسی مستقیم) → addressable
  - RAM – Random Access memory : Read + write
  - ROM – Read Only Memory
    1. classic – ROM
    2. PROM – programmable ROM ↱ UV light
    3. EPROM – Erasable programmable ROM
    4. EEPROM – Electrically Erasable programmable ROM or E^2PROM
- Sequential Access (دسترسی ترتیبی)
  - Tape
  - Queue ↱ buffer
    - FIFO
    - LIFO
- CAM – Content Addressable memory (دسترسی محتوایی) → non addressable

باید data های قبل از data خواسته شده طی شود

CAM – Content Addressable Memory

- CAM is a technology replicating the human brain, storing and finding data in a content related matter
- CAM does not have addresses, it uses the content of the data to find related data
  - ↳ each row/word has validity/invalidity
  - ↳ we give part of the word/data to find it
- Query word : data we want to find other data related to it
  - ↳ when writing to CAM, full word is written (and masked)
    - > written to a random space
- Mask : determines which bits of the query word are important
  - ↳ holds the BITS we want to find from the query word

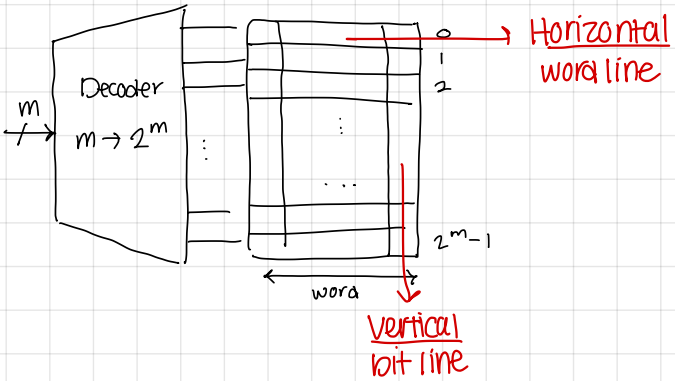


- To retrieve data CAM can be searched in two ways:
  1. Row by Row
     - ↳ involves checking all data row by row
     - ↳ very slow
  2. Parallel
     - ↳ checks parallel columns all at once
     - ↳ requires an xnor gate for each bit
       - > very expensive
       - > توان مصرفی بالا
     - ↳ very quick search

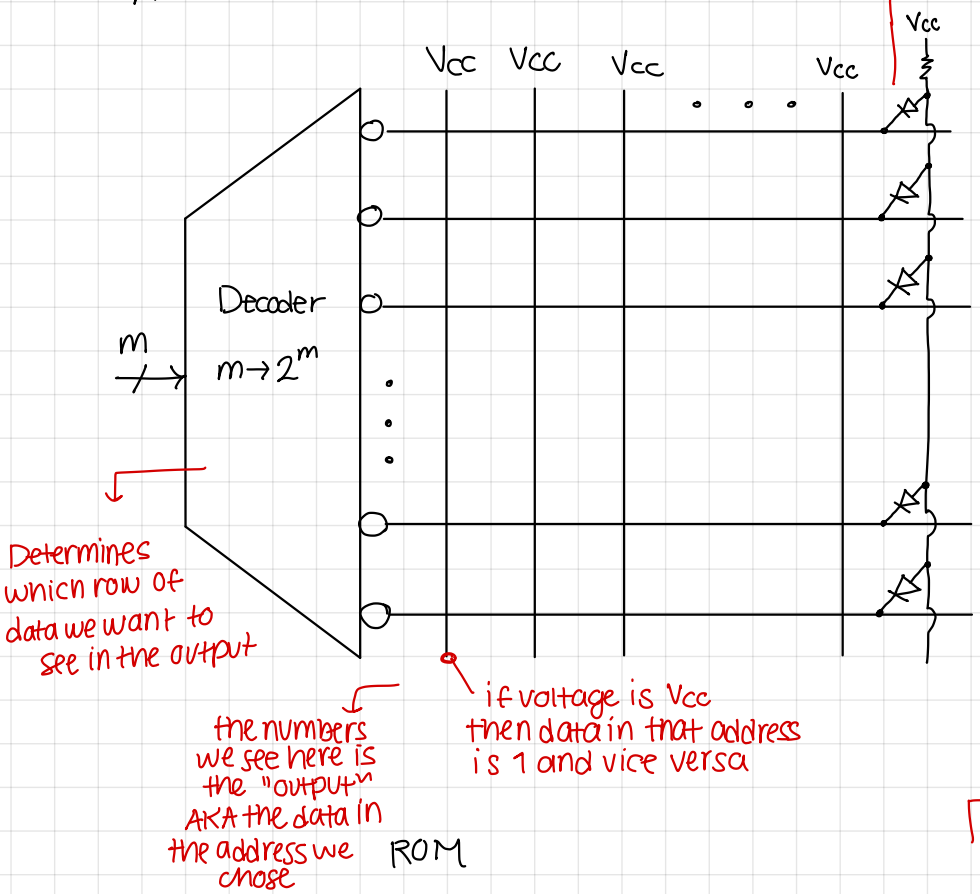
- when searching there are 3 possible scenarios / results
  1. miss – nothing similar is found
  2. single-hit – exactly one similar data
  3. multi-hit – can be one of the found options ↳ شبکه
- conclusion → CAM is too expensive + gets too hot with current technology
  - ↳ CAM can only be used for on board memory where high speeds are needed

Direct Access Memory (RAM & ROM)

- Addressable memory accessed and altered with a decoder
  - ↳ the decoder is the most expensive and volumous part of the RAM/ROM
    - > is the part preventing further advancements
- word: each row in the memory
  - ↳ is the smallest unit of memory
  - ↳ "يعني بيّن حافظة"
- word line: holds data
- bit line: gives data



RAM/ROM:



Diode is either working or burnt:

- if working: when active low decoder chooses that row, horizontal wire has voltage of zero and the vertical wire has voltage of Vcc → diode is on and shows value of zero in output
  as for other non-chosen rows, diode is off (voltage of horizontal wire is larger) so we know the output is only from the chosen row
- if burnt: there is no longer a wire, so the Vcc will show at the output (shows 1)

★ Therefore to store a 1 in our ROM, we burn the diode connecting that bit
↳ diodes are cheap
↳ how we write data (burning chosen diode)
★ burnt diodes are retrieved in EPROM with UV light

Determines which row of data we want to see in the output

the numbers we see here is the "output" AKA the data in the address we chose

if voltage is Vcc then data in that address is 1 and vice versa

ROM

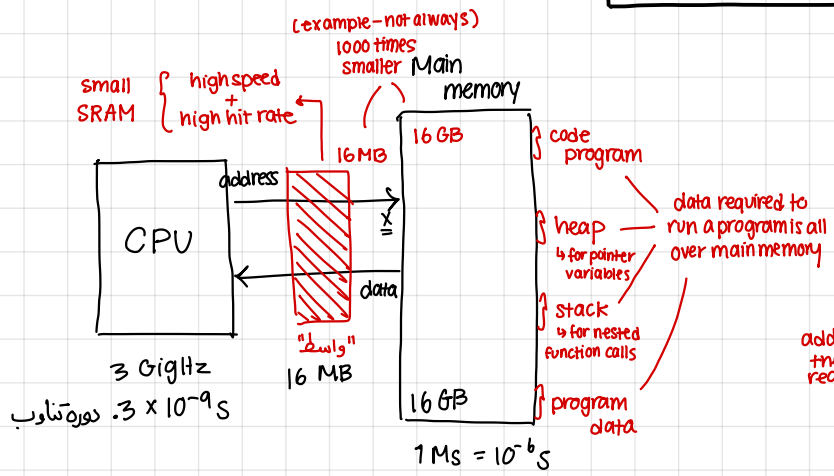
↳ longer Board line increases delay (even small amounts)

For example, lets say an input m = 00...01 is given to the decoder, activating the final row of the decoder. The activated horizontal wire has voltage of 0. in the last column the diode is intact, so the last horizontal line will show a voltage of 0. For the other rows, the wires have V_{cc} voltage (either the diode is off or burnt, both not changing the result)

★ جلسه پنجم – یکشنبه ۳۰ بهمن ★

Cache (حافظه نهان)

نیاز به پیش بینی خوب



- What data do we hold in the cache?
  - ↳ used Address Trace File to determine what data addresses are used when running a program
  - ↳ Address Trace File: Added slot on the mother board that captures all the data/requests interchanged between the CPU and main memory

Sample Address Trace File: (زمان →)

addresses the CPU requested ← 20, 21, 22, 23, 50, 80, 30, 22, 23, 20, 30, 14, 1400, 50, ...

often consecutive — often repetitive

- ↳ spatial locality (همجواری مکانی)
  - programs are run line by line / consecutively which creates spatial locality
    - > not always consecutive for ex. loop breaks function calls, go to commands, etc.
  - bring data near requested data to cache

although data used by the program is not where the program is

در طول زمان هم نزدیک هستند

هرگاه داده‌ای از M.M می‌آوریم، همسایه‌های آن را نیز می‌آوریم.

- ↳ temporal locality (همجواری زمانی)
  - we often use repeated variables / data in a program which creates temporal locality
  - keep data used for a program in the cache

هر داده‌ای که از M.M می‌آوریم را نگه داریم تا در آینده استفاده شود

★ The CPU is 3000 times faster than main memory
- ↳ main memory will slow down CPU processes
- ↳ SOLUTION → smaller + faster intermediary memory that contains recent program data (Cache)

★ Real life examples of cache:
- ↳ librarians check recently returned books before checking shelves for a requested book
- ↳ frequently accessed clothes are left outside the closet
- ↳ frequently accessed documents are left on the table
- ↳ backpack
  - > emptied once too heavy / full
  - > only bring wanted/needed things

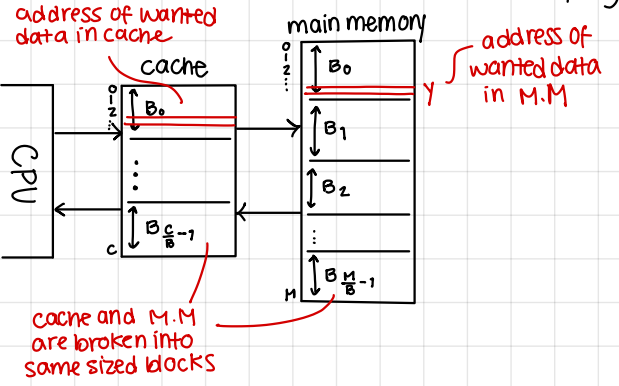
finding the organized item takes more time

★ From the perspective of the CPU, the cache is "transparent"
- ↳ the CPU cannot tell if the M.M has sped up or cache was used
- ↳ برای همان cache یک حافظه نهان است

- Blocks: Main memory is divided into blocks to move data to cache
  - > if block is too small, we lose spatial locality (not enough consecutive data in cache)
  - > if block is too big, we lose temporal locality (cache becomes too full and used data is overwritten)
  - ↳ Block size is a multiple of 2
  - ↳ $B = 2^b$
  - ↳ no consistency (not always for ex. 8 before and after requested data)
  - ↳ each block in the cache has a validity tag (used mostly when booting the computer)

valid { 0 = invalid, 1 = valid } — validity | Block

صریح + ساده + کم هزینه



- Relationship between x and y → mapping function
  - ↳ we bring the block that x is in

- Direct Mapped Cache (DMC)

حافظه نهان نگاشت مستقیم (باقی‌مانده) — خارج قسمت

$X_B = X/B$

$Y_B = X_B \bmod C/B$

ex.

$X = 010110|0110$ — X_B محل نسبی در بسته

number block — 6th item in the block

M = size of Main Memory = 2^m

C = size of Cache = 2^c

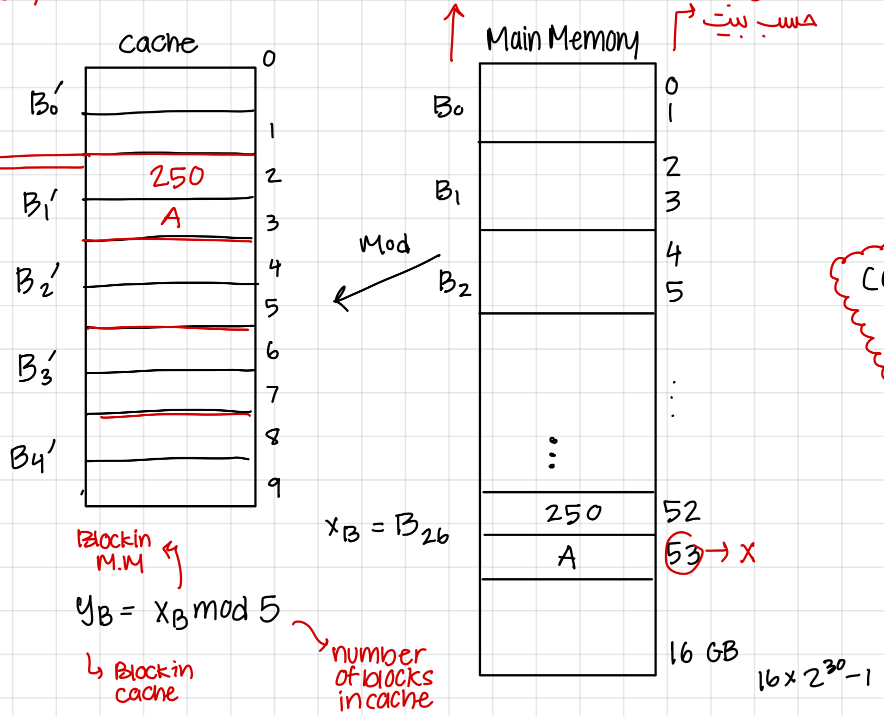
B = size of blocks = 2^b

★ جلسه ششم — سه‌شنبه ۲ اسفند ★

Example : 16 GB main memory with 10 bit cache with block size of $2 = B = 2^1$:

can't have both → overwritten/interchanged → ★ with one cpu, multiple programs are not run at the same time, but run sequentially for small amounts at a time

we need the Tag because many other addresses could be in block 1 of cache ex. 1, 16, 21, ...



$y_B = 26 \bmod 5 = 1$

↳ Data with address 53 will be stored in Block 1 of the cache

Cache : Tag Array + Data Array

Direct Mapped Cache (DMC)

- The CPU itself works with bit, main memory addresses, not cache
  - ↳ The CPU is not aware of the existence of the cache — all processes done between the M.M and cache are independent from the CPU
- We divide the Main Memory into blocks
  - ↳ Block : segments of data with set sizes used to interchange data between cache and M.M
- When moving data from M.M to cache, we use the following to determine where to set each block :

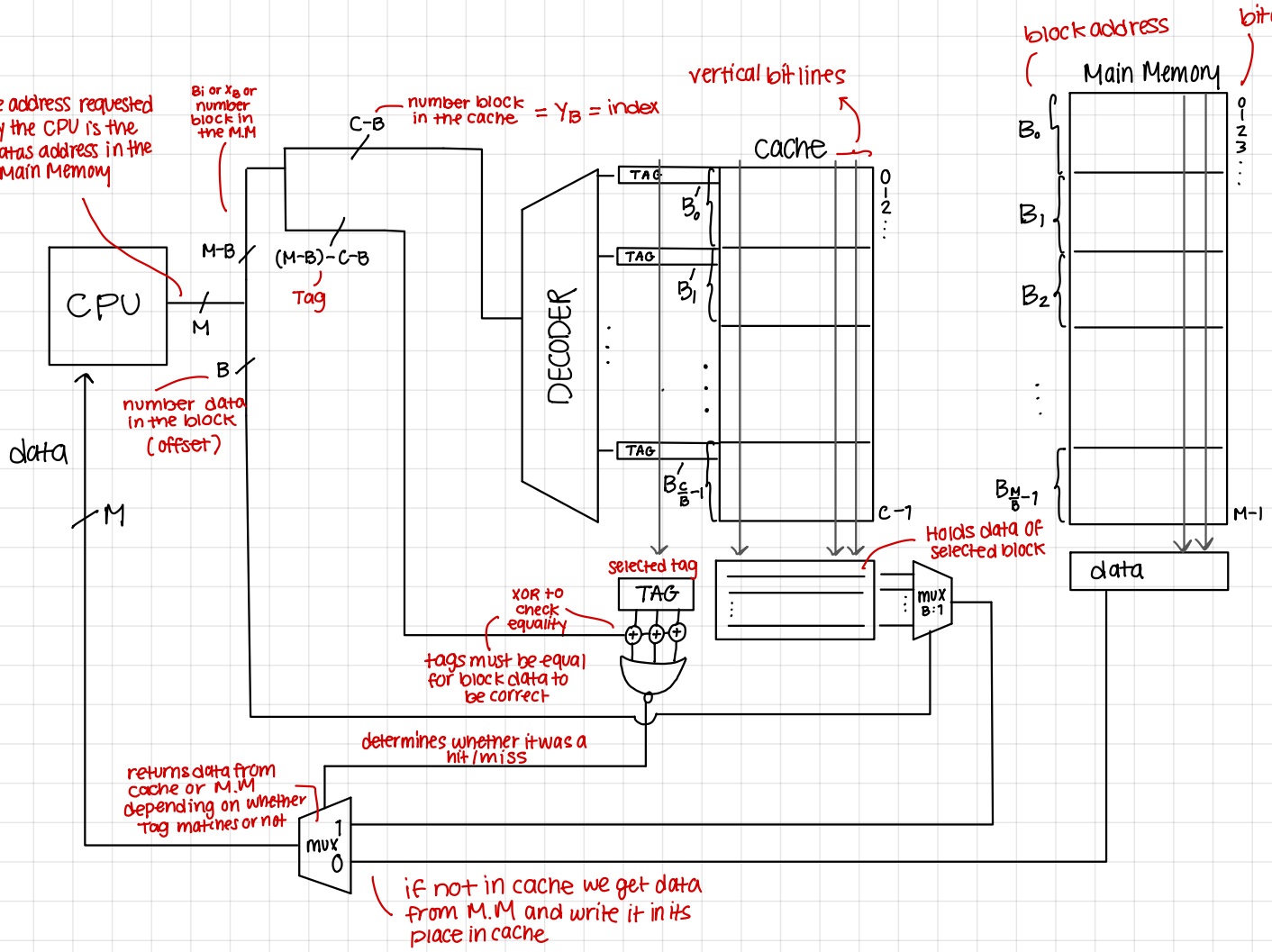
  $y_B = x_B \bmod (C/B)$ or $B_j' = B_i \bmod (C/B)$ — number of blocks in cache

  (x_B: number block in the M.M; y_B: number block in the cache)
  - ↳ with this mapping function there will definetly be collision, so we add a tag for each block
  - ↳ Tag : remainder of the number block the data is in divided by the cache size
    - > without tag we cannot be sure that the contents hasn't changed when reusing the cache
    - > why not just save bit address as tag? more bits → takes up more space
- Since the CPU is not aware of the cache, it cannot do the calculations
  - ↳ calculations done on motherboard by seperating the bits :
  - ↳ offset : relative index of requested data in block
    - > has size of B → size of block
  - ↳ index : number block the data is in (or not) in cache
    - > size is the size of cache minus the size of a block — $\frac{C}{B} = \frac{2^c}{2^b} = 2^{c-b}$
- A set associative cache where k=1

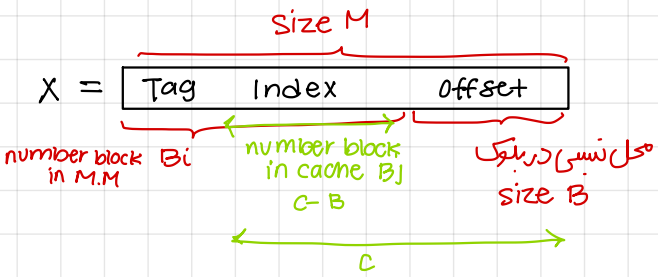
پس اگر همه‌ی مقادیر توانی از 2 باشند بدون نیاز به تقسیم، باقی‌مانده و خارج قسمت دست یابیم ←

$\frac{C}{B} = \frac{2^c}{2^b} = 2^{c-b}$, $B = 2^b$

لم : در مبنای r ، عدد $(x_{n-1} x_n \ldots | x_0)_r$ (خارج قسمت | باقی‌مانده) چنانچه بخواهیم تقسیم r^k انجام دهیم تقسیم لازم نیست بلکه از روی جدا کردن ارقام قابل فهم است.



← براساس آدرس تصمیم می‌گیریم data کجا قرار گیرد (بی ربط)



مثال: در یک حافظه نهان با حافظه 32 KB و یک M.M با حافظه 16 MB و B = 64 :

الف) میدان offset یا جابه‌جایی؟ ← سایز B

ب) اندازه میدان Tag یا برچسب؟ ← $M-(B+\frac{C}{B})$

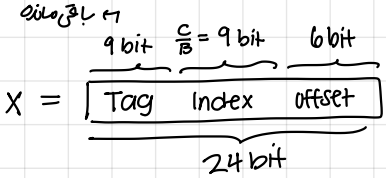
ج) اندازه میدان index یا شاخص؟ ← سایز $\frac{C}{B}$

$32\ KB = 2^5 \times 2^{10} = 2^{15}$

$16\ MB = 2^4 \times 2^{20} = 2^{24}$

$B = 2^6$

باقی‌مانده ↓



★ جلسه هفتم – پنج‌شنبه ۴ اسفند ★

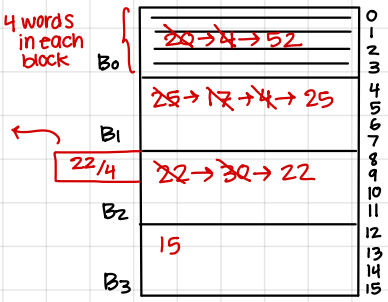
Example 1:

آدرس‌های تولیدی توسط CPU برای حافظه اصلی

	100	101	90	102	70	80	81	82	60	120	70	90	19	20	21	100	210
divided by B=4	25	25	22	25	17	20	20	20	15	30	17	22	4	5	5	25	52
mod number of blocks in cache $\frac{C}{B}=4$	1	1	2	1	1	0	0	0	3	2	1	2	0	1	1	1	0
		hit		hit			hit	hit			hit				hit		

move block 25 in M.M to block 1 in the cache

The content of block 22 is written in block 2 of the cache, and the tag is the result of the division $\frac{22}{4}$

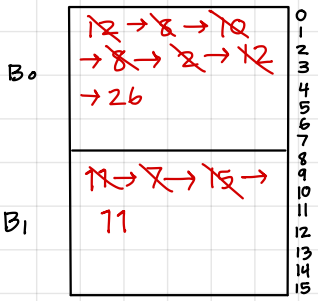


C = 16
B = 4

$$\text{hit rate} = \frac{6}{17}$$

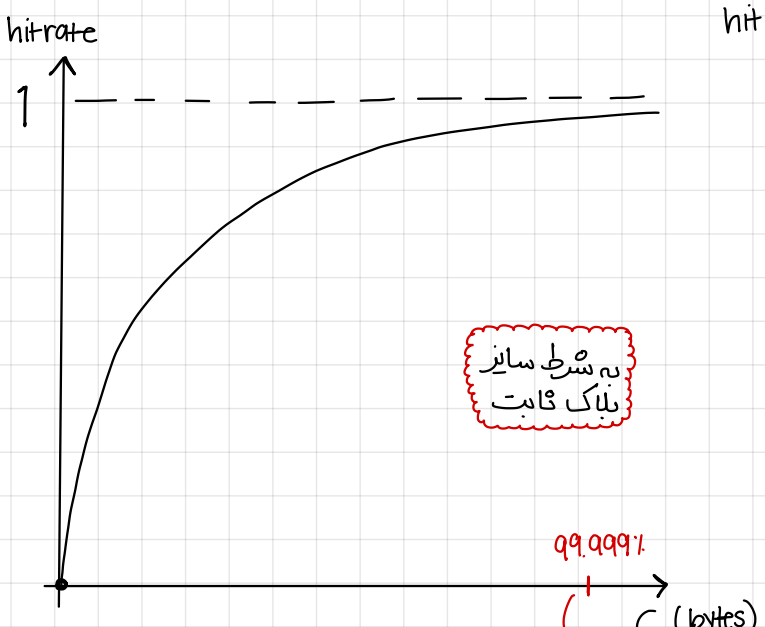
Example 2:

	100	101	90	102	70	80	81	82	60	120	70	90	19	20	21	100	210
÷8	12	12	11	12	8	10	10	10	7	15	8	11	2	2	2	12	26
mod 2	0	0	1	0	0	0	0	0	1	1	0	1	0	0	0	0	0
		hit		hit			hit	hit							hit	hit	

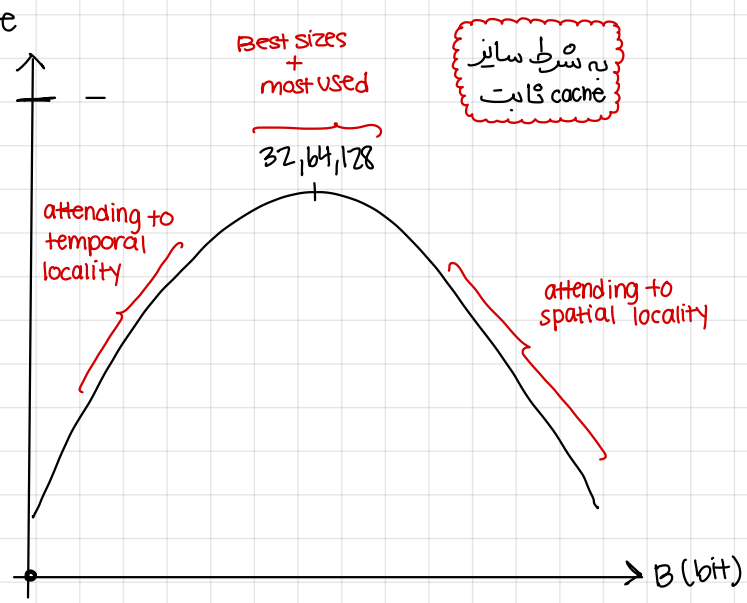


C = 16
B = 8

$$\text{hit rate} = \frac{6}{17}$$



max size of cache is the size of the M.M
↳ this will still not have a hit rate of 100%.
because the very first time data is requested and the cache is empty it will miss



placement policy
1. DMC – Direct Mapped Cache → has a problem, when we need data again it may be overwritten and cause a miss, even though we have temporal locality → some blocks in cache are completely unused while some are constantly being overwritten
2. Set Associative Cache < two-way / k-way
3. Fully Associative Cache

Hot → frequently used
warm → often used
cold → not being used

- In DMC, we do not attend to whether blocks are hot or not:
  - ↳ Hot : frequently used
  - ↳ warm : often used
  - ↳ cold : not being used

- DMC's have a hit rate of 95%.
  - ↳ little increase in hit rate can increase the speed of the computer substantially because of the large amount of requests from the CPU per second for every task
    - > especially on servers

Different programs may have different hit rates (on the same computer):
- increased hit rate: while loops, arrays, ...
- decreased hit rate: goto's, breaks, ...

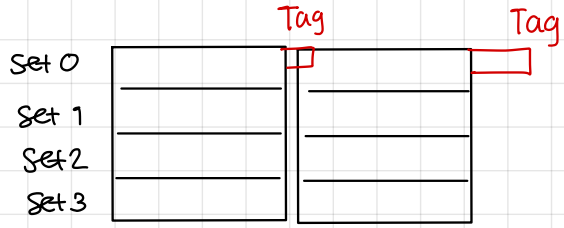
programs can be used to test the speed and evaluate the size/type of cache

Set Associative Cache حافظه نهان مجموعه انجمنی

- In DMC we only had one block of space for the same remainders, so we solve that problem by adding another column for each remainder
  - ↳ there is no priority between the columns
  - ↳ when a new block must be stored in a row, it replaces the oldest block in that row → requires an added piece of information to save the age of each block in cache
- In order to check if the block we want is already in the cache, we must check all k blocks in the row with that blocks remainder
  - ↳ parallel search
  - ↳ similar to CAM, we use xor gates to check all k words at once → same gates used for all rows → only one set uses them at a time
- Each row may have k columns
  - ↳ $K = 2^k$
- is 10-15% more expensive than DMC
- for different values of k:

norm: k=2 96% / k=4 97%
expensive + heat + breaks CPU: k=8 98% / k=16 99% / k=32 99.9% / ⋮ / $k = C/B$

Two-way set Associative cache: Have different locations ↳ need different tags



Fully Associative Cache حافظه نهان تمام انجمنی

- This is a set associative cache where k = max
  - ↳ k=max when $k = C/B$ or size of cache
- This requires an XOR gate for each word to check all tags at once
  - ↳ expensive + gets hot
- If implemented, has hit rate of 99.99%.
  - ↳ possibly used in large corporations

* انتظار حافظه نهان را ضعیف کردیم آوردیم توست

2 options in set: ① empty space ② when full → choose one to overwrite

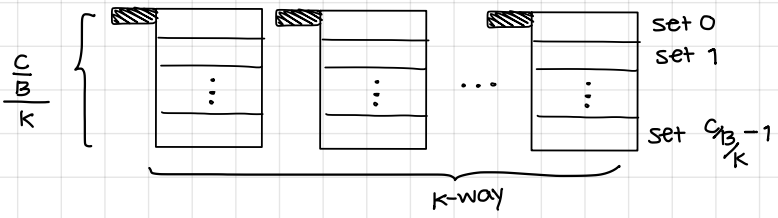
$S_j = B_i \bmod (C/B/2)$ ← number of sets

X/B

خارج قسمت بزرگ‌تری دارد (یک بیت بیشتر)
K=1 ← یک بیت بیشتر
K=2 ← دو بیت بیشتر
K=3 ← سه بیت بیشتر

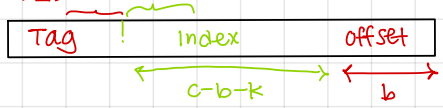
★ جلسه هشتم – یکشنبه ۷ اسفند ★

K-way (KWSA) set Associative cache:



(DMC از) تعداد ست k

KWSA



95.1 – 98

FWA

99 . . ↳ Index = 0

* the more blocks in a row, the more expensive + power needed

$C = 2^c$
$B = 2^b$
$K = 2^k$
$C/B/K = 2^{c-b-k}$

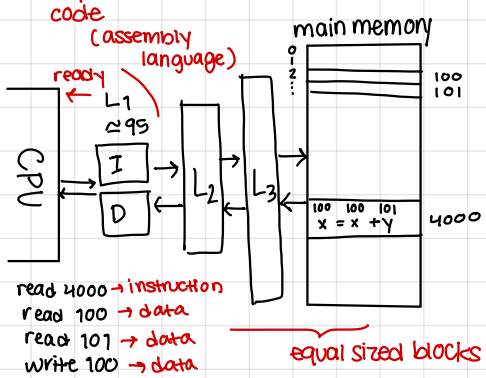
- CPU commands making changes in the memory
  - ↳ changes can be done in cache or M.M
  - ↳ two methods in altering data:
    1. write through → any change in cache must also be done in M.M
       + confirms all data in M.M is updated
       − slows down CPU
    2. write back → changes are done in cache, and whenever block is removed from cache it is written in M.M
       + keeps speed
       − M.M is not updated — could cause problems of data loss with sudden disconnections
  - ↳ requires a tag determining whether a block needs to be rewritten or not

dirty □ { 0 → only read + no change; 1 → write + change in MM }

↳ only one word in the block changes but the whole block is tagged as dirty

ex. compiled code (assembly language)

- Multiple levels of Cache



- Addresses missed from first cache still have some temporal and spatial locality
- We add 3 levels of cache:
  - ↳ As we get further from the CPU, the cache gets slower and larger
- Spliting the cache into 2 segments (Instructions and Data) causes more spatial locality
  - ↳ not always beneficial, could benefit more from more space for data/instructions

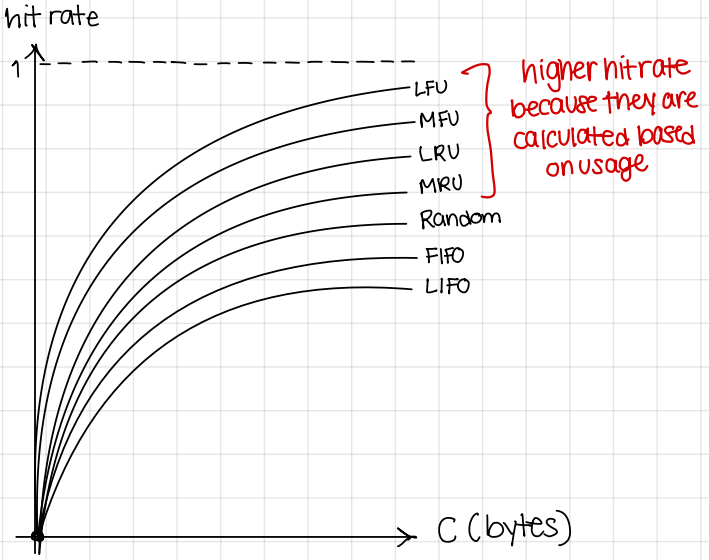
↳ when data was in none of the caches and we get the data from the M.M, the new data is written to all 3 levels of cache as it goes to CPU

- keeping a time stamp is impractical, other methods were introduced:

Replacement Policy

1. FIFO } keeping index
2. LIFO }
3. Random
4. Least Recently Used → calculated from when they enter the cache (restarts when it leaves the cache)
   - "LRU" → pseudo LRU → what's implemented in computers
5. Most Recently used
6. Least Frequently used → calculated from the start of their entrance in the cache (even when they leave the set count is kept)
   - "LFU"
7. Most Frequently used
8. Distance_based → removes the data that is "furthest" address-wise from the new data

- 4: something that was used alot will be used more in the future
- 5: something that was already used alot and won't be used again (temporal locality)
- 6, 7: very expensive + cannot be implemented
  - ↳ have to store the hits for each block in the M.M
  - ↳ number of hits is a big number



- All replacement policies can be implemented with a queue
  - ↳ each block has a counter that determines their place in the queue
    - ↳ when ever hit occurs for a block its counter is reset to zero and all other counters are incremented
      - ↳ the block with the largest counter is replaced

- The larger k is, the more the chosen replacement policy matters + affects hit rate

- Implementing the replacement policies
  - ↳ FIFO
    - > start from right most space, shift to the left for each new data
  - ↳ MRU / LRU (pseudo)
    - > keep track of data entering the cache using a queue
    - > each time data enters the cache + each time data is 'hit', it moves to the start of the queue
    - > for MRU we replace data at the start of the queue
    - > for LRU we replace data at the end of the queue
  - ↳ LFU / MFU
    - > keep count of every time a given data was 'hit' → even when it leaves the cache
    - > when counts are equal, remove oldest data (generally)
    - > for MFU we replace the data with highest hit count
    - > for LFU we replace the data with lowest hit count

by definition, in LRU these should be counted → too expensive → pseudo LRU is used

a, a, a, b, c, d, e

مثال: آدرس‌ها، آدرس بلوک در حافظه اصلی هستند.

$\frac{C}{B} = 4$:

2-way associative (LIFO)

hit rate : $\frac{7}{18}$

0	4	6→2
1	5	3→15→17→15→3→15

5, 4, 3, 5, 6, 4, 3, 3, 6, 15, 17, 6, 2, 17, 17, 15, 3, 15

mod 4/2 ↓

1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1

2-way associative (FIFO)

hit rate : $\frac{9}{18}$

0	4→6	4→6→2
1	5→3→15→17	5→3→15→17→3

5, 4, 3, 5, 6, 4, 3, 3, 6, 15, 17, 6, 2, 17, 17, 15, 3, 15

mod 4/2 ↓

1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1

2-way associative (LRU)

صف‌ها:
(از آخر صف بیرون می‌آریم)

hit rate : $\frac{10}{18}$

0	6	4→2
1	3→17→3	5→15

4-6-4-6-2

5-3-5-3-15-17-15-3-15

5, 4, 3, 5, 6, 4, 3, 3, 6, 15, 17, 6, 2, 17, 17, 15, 3, 15

mod 4/2 ↓

1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1

2-way associative (LFU)

*when equal count remove oldest

hit rate : $\frac{10}{18}$

0	4→17	6
1	5→15	3→2→3

count: (remove smallest)

2
3 ||
4 |
5 |
6 ||
15 |
17 ||

5, 4, 3, 5, 6, 4, 3, 3, 6, 15, 17, 6, 2, 17, 17, 15, 3, 15

mod 4/2 ↓

1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1

Fully associative (MRU)

hit rate : $\frac{9}{18}$

5, 4, 3, 5, 6, 4, 3, 3, 6, 15, 17, 6, 2, 17, 17, 15, 3, 15

0	5	4	3	6→15→17→6→2→17→15

صف:
(از اول صف بیرون می‌آریم)

5-4-3-5-6-4-3-6-15-17-2-17-15-3-15

Fully associative (MFU)

hit rate : $\frac{11}{18}$

5, 4, 3, 5, 6, 4, 3, 3, 6, 15, 17, 6, 2, 17, 17, 15, 3, 15

0	5→17	4→2	3→15	6→3

(remove largest) count:

2
3 ||
4 |
5 |
6 ||
15 ||
17 ||

*when equal count remove oldest

2.

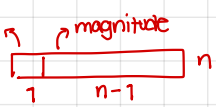
ALU design

- Data Representation
- Adders
- Subtractors
- Multipliers
- Divider
- Fixed Point
- Floating Point

★ جلسه دهم – یکشنبه ۱۴ اسفند ★

Data Representation

- Integer numbers
  - ↳ unsigned ($2^n - 1$ possible (+) numbers)
  - ↳ signed (+ IO with human, − harder calculation)
    - > Sign magnitude ($2^{(n-1)} - 1$ possible (+/−) numbers) → we lose 100000 because of −0
      - ★ $[-2^{(n-1)} - 1, 2^{(n-1)} - 1]$



    - > 1's complement
      - ★ complement all bits in the number
      - ★ $[-2^{(n-1)} - 1, 2^{(n-1)} - 1]$
    - > 2's complement (+ easier calculation, − less compatible w/ humans) → best choice: no negative 0, no symmetry, $+2^{n-1} - 1$ to $-2^{(n-1)}$
      - ★ once you reach first one from the right, complement the rest (and first bit shows +/−)
      - ★ $[-2^{(n-1)}, 2^{(n-1)} - 1]$
    - > excess: try 2's complement bias 1: here, excess is 2^{n-1} → add 2^{n-1} to all numbers (used for showing floating point)
      - ex. 0000 ↔ 128
      - ★ similar to 2's complement, except easier comparisons → any number with 1 first is larger
      - ★ add 128 to all numbers (largest negative number in 2's complement)
      - ★ Two's complement, but first bit from the left is complemented
      - ★ $[-2^{(n-1)}, 2^{(n-1)} - 1]$

★ پردازنده برای اعداد مختلط؟ special purpose CPU

games → rendering: uses lots of mathematical operations (lighting, shading, ...)
↳ requires more operands

(desktop) more assembly code, slower + cheaper ↔ (not all operands are useful day to day) expensive + more capable

less operands or more?

- Decimal numbers
  - ↳ fixed point decimal → more error + cheaper
  - ↳ floating point decimal → less error + expensive
- We choose which data representation we want to use based on client needs ★ change to unsigned when we can
  - ↳ only have data types that will be used (more types will be expensive)
  - ↳ CPU's have backward compatibility → always keeps types from old CPU's

n = 20

- unsigned: $0 \leftrightarrow 2^{20} - 1$
- 2's complement: $-2^{19} \leftrightarrow 2^{19} - 1$
- 1's complement:
- signed: $-(2^{19} - 1) \leftrightarrow 2^{19} - 1$

decimal def

- set place for decimal (ex. 8 bit for whole number and 8 bit for decimal)

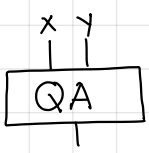
must be able to design ALU for all ↳ add, subtract, multiplication, division

- ★ signed: beneficial for large domains
- ★ sign magnitude: easy to read for humans
- 1's complement: No benefits compared to others
- ★ 2's complement: No negative 0, maximum usage of space
- ★ double precision: cheap + easier to implement
- ★ floating precision: lessens domain, must know where that is → precise + expensive

Adders

augend A + B addend

- Unsigned Adders
  - ↳ Quarter adder → 1 bit adder w/no carry
  - ↳ Half adder → 1 bit adder w/carry
  - ↳ Full adder → 1 bit adder w/carry$_{in}$

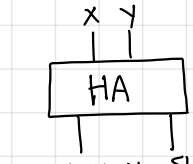


$S = X \oplus Y$

delay of one gate

delay(sum) = d

cost = 1g

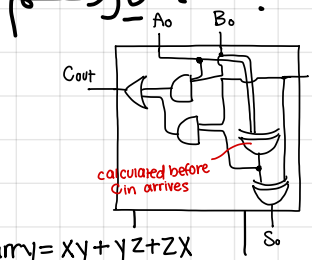


carry = x · y

delay(sum) = d

delay(carry) = d

cost = 2g



carry = xy + yz + zx

sum = x⊕y⊕z

delay(sum) = 1d (carry) = 2d

cost = 5g

- Ripple Adder
  - ↳ The carry-out of the previous FA is the carry-in of the next

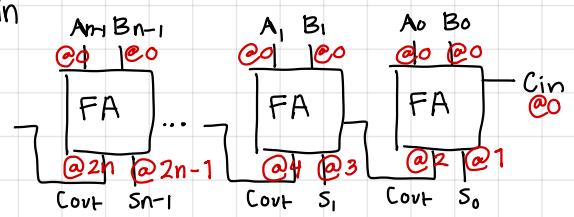
n-bit adder

important because we want the best ppc

delay(sum): 2n−1 d

delay(carry): 2n d

cost: 5n g



* the delay of an FA is Δ = 2d, using the critical path to determine this

- Carry Look-Ahead Adder — fast + expensive

* which has a better ppc?

  - ↳ carry generate: determines whether A = B = 1 (creates carry or not) $G_i = A_i \cdot B_i$
  - ↳ carry propogate: determines whether the carry moves to the next
  - ↳ and's and or's can be done parallel

n for all Pis, n for all Gis — 2n gates

$\frac{n(n+3)}{2}$ gates

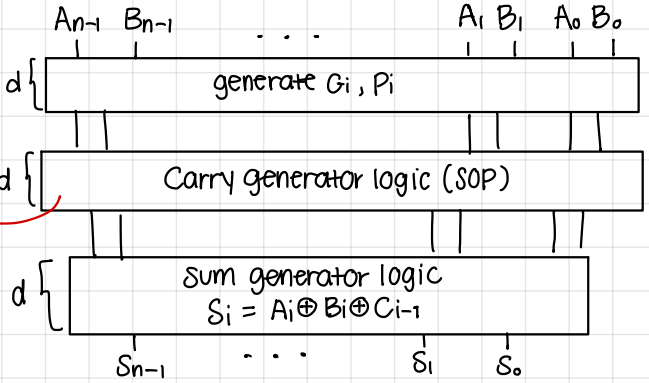
xor gates — n gates

delay(sum): 4d

delay(carry): 3d

cost: $3n + \frac{n(n+3)}{2}$ g

unrealistic, gates with more inputs have a greater delay

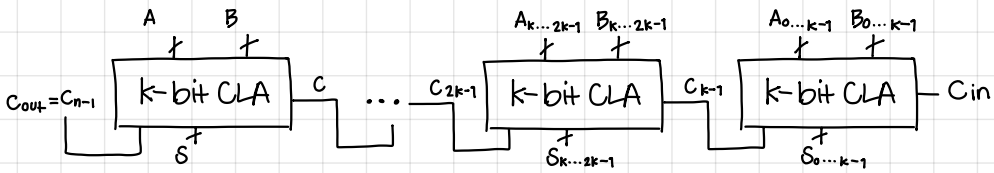


  - ↳ k-bit CLA's: using CLA's for larger n's requires gates with a large number of inputs which is unrealistic, so k-bit CLA's combine CLA's with the ripple adder concept:

delay(sum): $2(\frac{n}{k}) + 1$ d — the first CLA still takes 3d

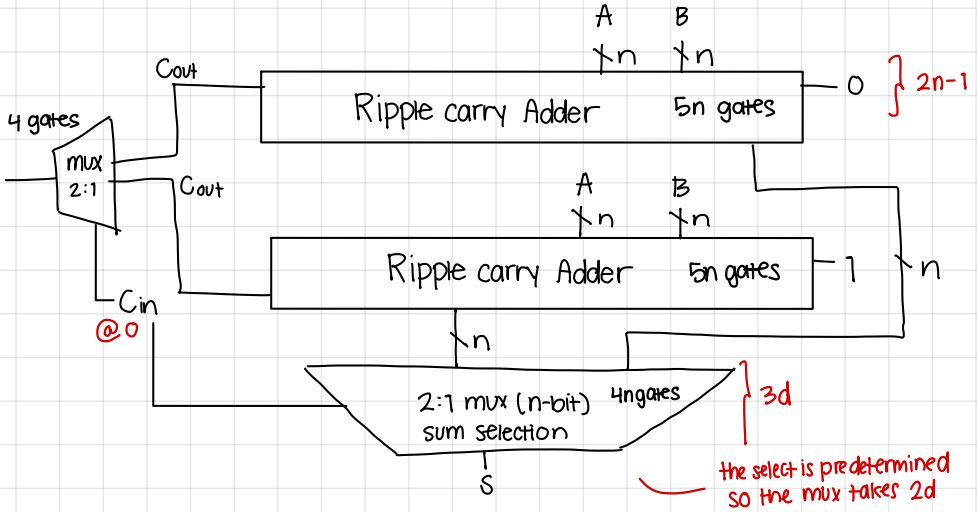
delay(carry): $2(\frac{n}{k}) + 2$d

cost: $\frac{n}{k}\left(3k + \frac{k(k+3)}{2}\right)$ — each CLA



the multiple is actually 2 because the Pi and Gi can be calculated before recieving the Cin of the k-bit CLA

- Carry Select Adder
  - ↳ calculate sum with two scenarios (carry = 1 or 0) then use Cin as input in mux of these two answers



the select is predetermined so the mux takes 2d

mux 2→1

delay: 3d

cost: 4g

A 2k→1k mux will have a delay of 3d and cost of 4kg (the not gate can be shared but will not be feasible for larger k's

delay(sum) = 2n+1 d

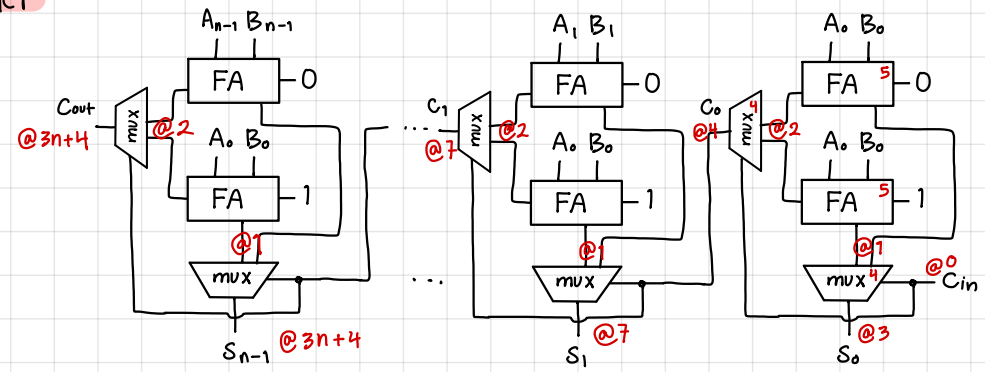
delay(carry) = 2n+2 d

cost = 14n+4 g

★ جلسه دوازدهم - یکشنبه ۲۱ اسفند ★

• Ripple (1-bit Carry Select) Adder

delay (sum): $3n+4$ d
delay (carry): $3n+4$ d
cost: $18n$ g



• k-bit Uniform Carry Select Adder

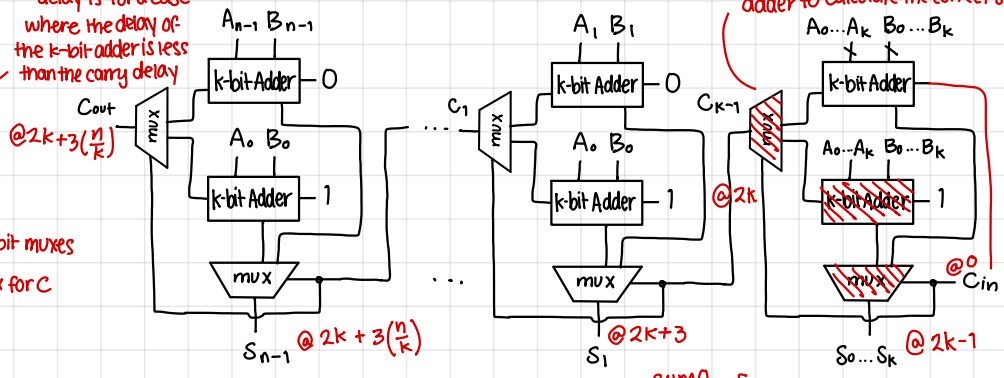
carry delay of used adder (ripple adder)

delay (sum): $2k+3(\frac{n}{k}-1)$ d — delay is for a case where the delay of the k-bit adder is less than the carry delay
delay (carry): $2k+3(\frac{n}{k}-1)$ d
cost: $2(\frac{n}{k})5 + 4(\frac{n}{k}+n)$ g

- $4k(\frac{n}{k})$ k-bit muxes
- $4(\frac{n}{k})$ mux for C_{out}
- cost of used adder (ripple adder)
- more precise

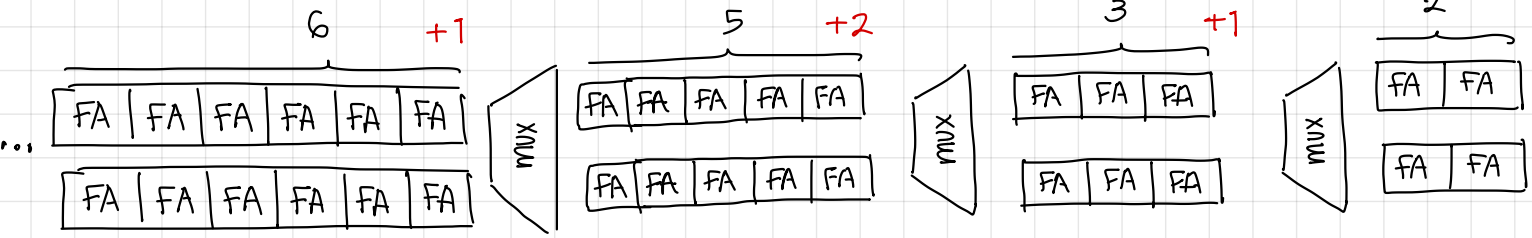
$2(\frac{n}{k}-1)5 + 4(n-1)(\frac{1}{k}+1)$

the crossed out pieces are unnecessary since we have C_{in} and can use one adder to calculate the correct sum



• Non-uniform Carry Select Adder

↳ A full adder takes 2d to calculate the sum
↳ A mux takes 3d to give us the output (from carry) — the extra delay lets us add a FA to the next k-bit adder and two FA's in the next level
↳ Every mux gives us time for 1.5 more FA's → we can add them after each level



↳ 16 bit k-bit CSA → completed in 4 levels — would've taken 8 levels had we kept two FA's in each level, but by adding FA's for each delay in the muxes, the result is calculated with less levels (less cost)

delay (sum/carry): $2k+3(\frac{n}{k})$ d — same as uniform
cost: ? less than uniform

• Adding m n-bit numbers

↳ serial addition
> starting from the first and second numbers, adding their sum with the third number, and so on.
> needs m-1 adders → not efficient cost and time wise

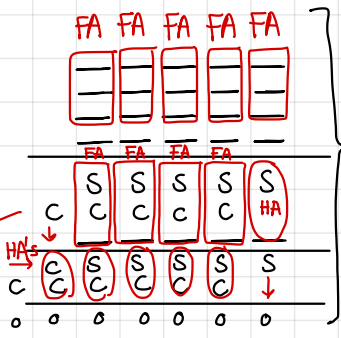
↳ binary two-by-two addition (complete binary tree)
> every two numbers are added together, then their sums...
> height of the tree will be $\lceil \lg_2 (m) \rceil$

↳ carry save adder — each FA can add 3 at a time — parallel
> we add each column individually then add the carry's with the sum and the next number

delay: ?
cost: ?

For m numbers, we require $\frac{m}{3}$ rows of n FA's
↳ each row of FA produces 2 rows (carry / sum) → we will have $\lceil \frac{2}{3} m \rceil$ rows in the next step

Adding 4 5-bit numbers:



$\log_{\frac{3}{2}} m$ levels
with every added level, the bit length increases by 1

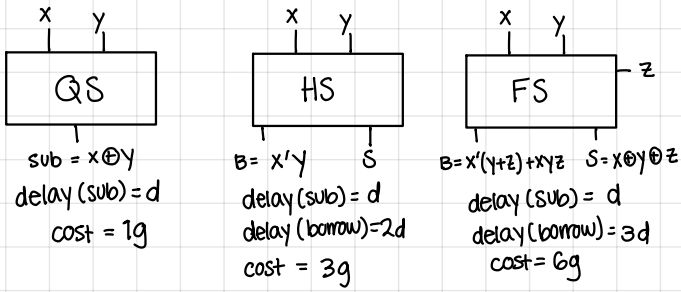
the carry's are produced for the next column, so they need to be pushed one to the left

"ارزش مکانی یکی بیشتر از خود عدد"

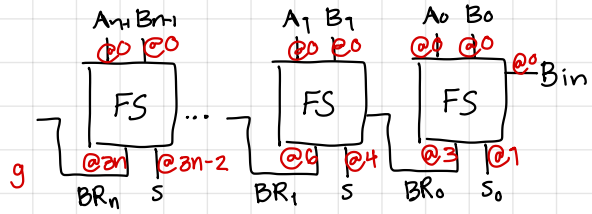
Subtractors

minuend — A − B — subtrahend

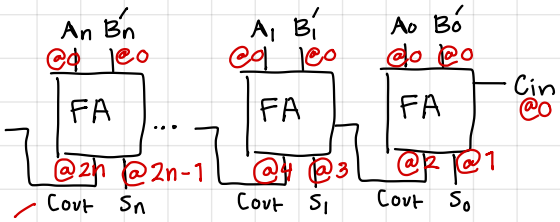
- unsigned subtractor
  - ↳ Quarter subtractor → 1 bit subtractor w/ no borrow
  - ↳ Half subtractor → 1 bit subtractor w/ borrow
  - ↳ Full subtractor → 1 bit subtractor w/ borrow and $borrow_{in}$



- Ripple subtractor
  - ↳ the $borrow_{in}$ of the previous FS is the $Borrow_{out}$ of the next
    - delay (sub) : (3n-2) d
    - delay (borrow) : 3n d
    - cost : 6 ng



- Complement Subtractor
  - ↳ the difference of two numbers can be calculated by: $A-B = 2^n + A - B = A + B' + 1$ (proveable)



this is the complement of $B_{out} = C_{n-1}'$

prove: All unsigned adders and subtractors work for 2's complement

OVERFLOW

- all calculations in the ALU must result in n-bits
  - ↳ we have n-bit registers and the sizes cannot be modified (not dynamic)
  - ↳ we must also be able to use the result in further calculations
- we have set number of n-bits for the solution, if greater, we have OVERFLOW
- we design a signal/flag to let the user know that there has been an overflow and the solution provided is incorrect

only physical solution is to make registers bigger 32 bit ↔ 64 bit (hardware solution)

- Overflow in unsigned Addition/Subtraction
  - ↳ in unsigned addition, overflow occurs when we have a final carry
    - > $C_{out} = 1$
  - ↳ in unsigned subtraction, overflow occurs when we have a final borrow ($B_{out} = C_{out}'$)
    - > $B_{out} = 1$
    - > In complement subtractors, if we DONT have a carry, we have overflow

$$A + B' + 1 \begin{cases} A \geq B \Rightarrow C_{out} = 1 \\ A < B \Rightarrow C_{out} = 0 \end{cases}$$

doesn't need borrow

reverse relationship with borrow

$$\begin{array}{r} 1 \\ -\ 1\ 1 \\ \hline 0 \end{array} \Rightarrow \begin{array}{r} 1 \\ +\ 0\ 1 \\ \hline 0 \end{array}$$

no borrow

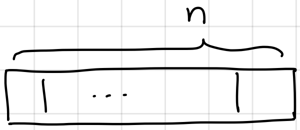
has carry ↓ no borrow

Flags

* جلسه سیزدهم - سه‌شنبه ۱۵ فروردین *

- All unsigned adders can also add 2's complement numbers
- All unsigned subtractors can also subtract 2's complement numbers

} overflow is different



- overflow in signed 2's complement addition — C_{out} doesn't mean anything — overflow → هیچ جایگاهی نداره
  - ↳ there has been an overflow if:
    1. The addition of two positive numbers resulted in a negative or two negative numbers result in a positive { ++ = − ; −− = + }
    2. The last two carrys are NOT equal ($C_{n-1} \neq C_{n-2}$)
       - > carry not created in one before the last but created in last
       - > carry created in one before the last but not created in last
       - (یک‌ها به هم میریزه)
  - ↳ overflow in subtraction can be found the same way with $A + B' + 1$

$$\begin{array}{ll} & A \ (A_s) \\ + & B \ (B_s) \\ \hline & F \ (F_s) \end{array}$$

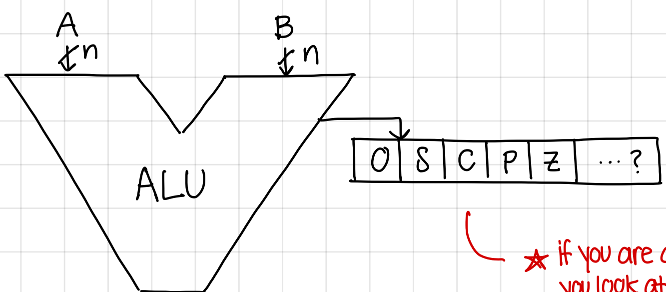
overflow { $A_s'B_s'F_s + A_sB_sF_s' = \text{True}$; $C_{n-1} \oplus C_{n-2} = \text{True}$ }

- overflow solutions: up to the user to expand data size — must let user know overflow occured

- Flags
  - ↳ bits that show the status of ALU calculations
    1. O - overflow: whether calculated data is correct or not — 2's complement overflow logic connected to it
    2. S - sign: holds the F_s (Left most bit) — only for signed numbers
    3. Z - zero: whether calculation result is zero or not (NOR gate)
    4. C - carry: whether calculation has carry out or not — unsigned overflow logic connected to it
    5. P - parity: activated when data is lost (soft error)
       - > reasons could be smaller transistors, sudden hits to the computer, UV sunlight rays, ... — problem is handled for satilites
       - > a single bit is flipped ← leads to problems when more than one bit flips
       - > even parity: adding an extra 1 when needed that keeps the number of 1's even
       - > odd parity: adding an extra 1 when needed that keeps the number of 1's odd (xor)
         - ex. in odd parity, if the data has an odd number of 1's, parity is 1

	P	8-bit data
already has odd 1's ←	0	11101010
parity is 1 to make odd ←	1	11010111

- > when we have parity and the 1's are no longer odd/even, we know we had data loss
  - still don't know which bit has changed (or how many)
- * error correcting code - parity with 1+ bits that know what code changed and corrects that code



* if you are adding unsigned you look at carry flag and if we are adding signed we look at overflow flag

Intel/AMD
8x6

All adders and subtractors work the same so... we have both unsigned + 2's complement the calculations will all be the same

```
DW ?          (define word ...)
SEGMENT CODE
    ADD a,b,c     } a=b+c;
    JO finish     } if(overflow) exit(0);
      or
    JC finish
```

jump if O = 1 (JO)

jump if C = 1 (JC)

If a>b then
else

compiler ⟹

```
DW ?
SUB A,B      → A = A-B
JS myElse    (previously defined function)
  * source code
JMP myEnd
  * myElse source code
myEnd
```

JNC → for unsigned (instead of JS)

★ جلسه چهاردهم - یکشنبه ۲۰ فروردین ★

Multipliers

multiplicant A * B multiplier

	0	1
0	0	0
1	0	1

$a * b = a \cdot b$

- In binary multiplication, we can just AND the two bits
  - ↳ guaranteed the product will be a single binary digit
- with the product of two unsigned numbers the following will always be true $\begin{cases} 0 \le A \le 2^n \\ 0 \le B \le 2^n \\ 0 \le A \times B \le 2^{2n} \end{cases}$ ← the product of two n-bit numbers will always fit in 2n-bits
- There is no chance of overflow in unsigned multiplication
- **unsigned number multiplied**
  - ↳ for multiplying two single bits (AND gate)
    - delay = 1d
    - cost = 1g

A	B	A*B
0	0	0
0	1	0
1	0	0
1	1	1

as partial products are produced, they are added to the product (starting at 0)

partial product moves left ↳ product moves right

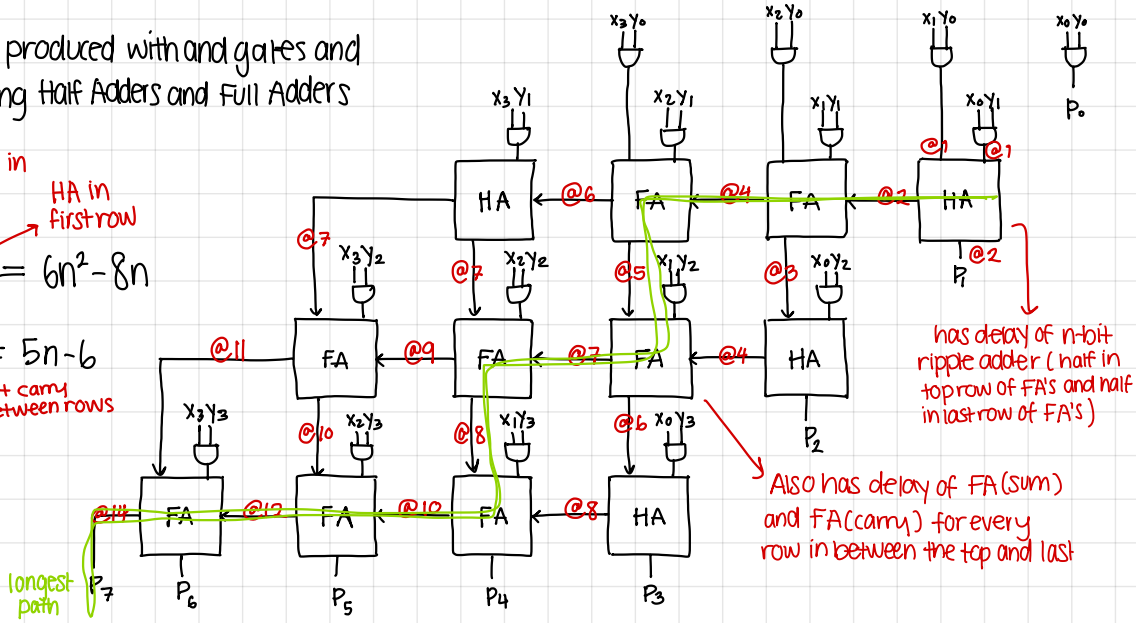
```
  28      multiplicant
x 34      multiplier
 112      partial products
+840
 952  → product
```

- **Array multiplier**
  - ↳ each partial product is produced with and gates and are added together using Half Adders and Full Adders

AND gates — number of additions that take place — FA's — First HA in each row — HA in first row

cost : $n^2 + (n-1)(5n-3) - 3 = 6n^2 - 8n$

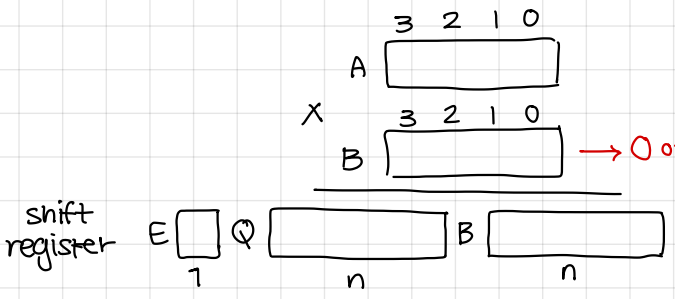
delay : $\underbrace{2}_{\text{first HA}} + \underbrace{2(n-1)}_{\text{n-1 ripple adder}} + \underbrace{3(n-2)}_{\text{FA sum + carry delay between rows}} = 5n-6$



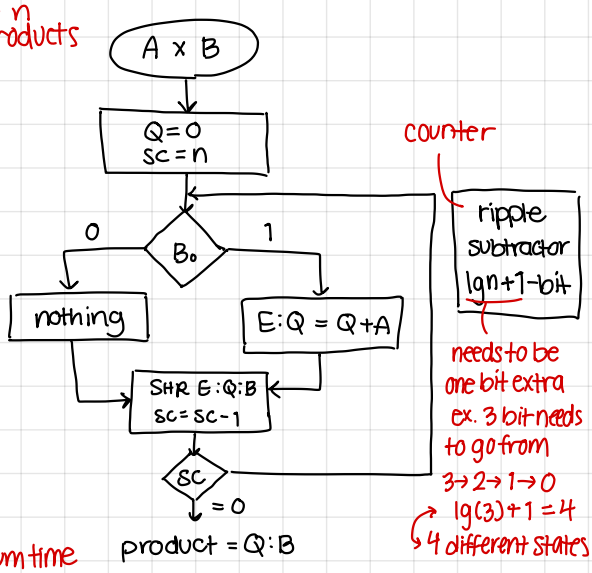
- **Shift-Add multiplier**
  1. Always look at B_0 and shift B to the right in every step so $B_0 = B_i$ to know wether to add A or Shift (0)
  2. Instead of keeping the product in the same place and shifting the partial products to the left, the partials stay in the same place and the product shifts to the right
  3. After multiplying, the first n bits are stored in B's register

partial product is always either 0 or A: $B_i=0$ → 0; $B_i=1$ → A

we have n partial products



only needs to check equality (nor gates)



cost = (counter: lg(n)+1 bit ripple sub) + (product: 2n+1 bit shift Register) + (Q: n bit register) + (comparator: n bit comparator) + (E: 1)

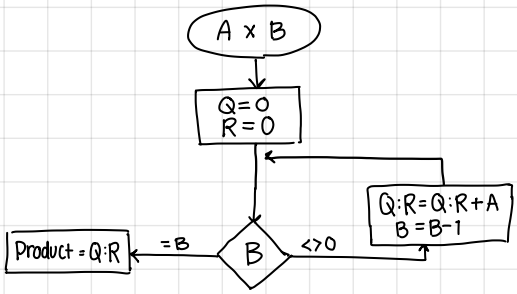
delay = { counter delay : $3(\lg n+1) - 2 + 3n$; adder delay : $\underbrace{2n}_{\text{ripple adder carry}} + \underbrace{(2n+1)}_{\text{shift register}}$ } (occur parallel) $= 4n+1\ d$ ← minimum time that must pass between each clock

delay(product) = $n(4n+1)\ d$

* the non-sequential version of this multiplier will result in the array multiplier

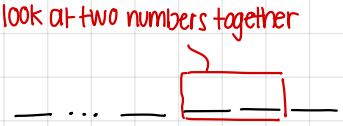
★ جلسه پانزدهم - سه‌شنبه ۲۲ فروردین ★

- **Serial-add multiplier**
  - ↳ add A with itself B times
    - cost = ?
    - delay = ?



- **Booth multiplier** — only for signed numbers
  - ↳ the booth representation of a number consists of 1's, 0's, and -1's
  - ↳ attaining booth representation of a binary number:
    1. beginning from the right, place 0's for every 0 until you reach a train of 1's
    2. place a -1 where the 1's start and place 0's for all following 1's
    3. when you reach the end of the train, place a 1 — 10101 → 0101.. terrible for booth representation
    4. repeat steps 1-3 until you reach the left most bit
       - > if the last bit is 1, the left most non-zero bit in booth will be -1
       - > if the last bit is 0, the left most non-zero bit in booth will be 1

look at two numbers together



$B_0\ G \rightarrow \begin{cases} 0\ 0 \rightarrow B_0' = 0 \\ 0\ 1 \rightarrow B_0' = 1 \\ 1\ 0 \rightarrow B_0' = -1 \\ 1\ 1 \rightarrow B_0' = 0 \end{cases}$

ex. sign extension

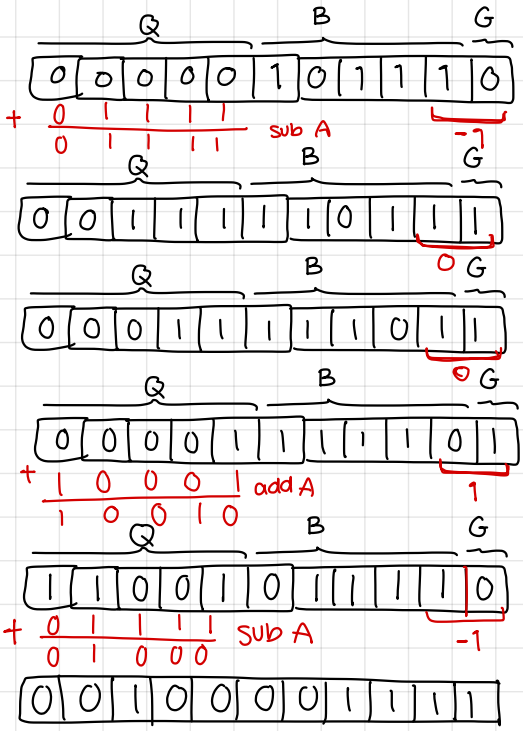
$1111\boxed{01}1\boxed{1}_2 = 0\text{-}110\text{-}1 = -2^3 + 2^2 - 2^0 = -5$

$\boxed{01}110\boxed{}_2 = 100\text{-}10 = 2^4 - 2^1 = 14$

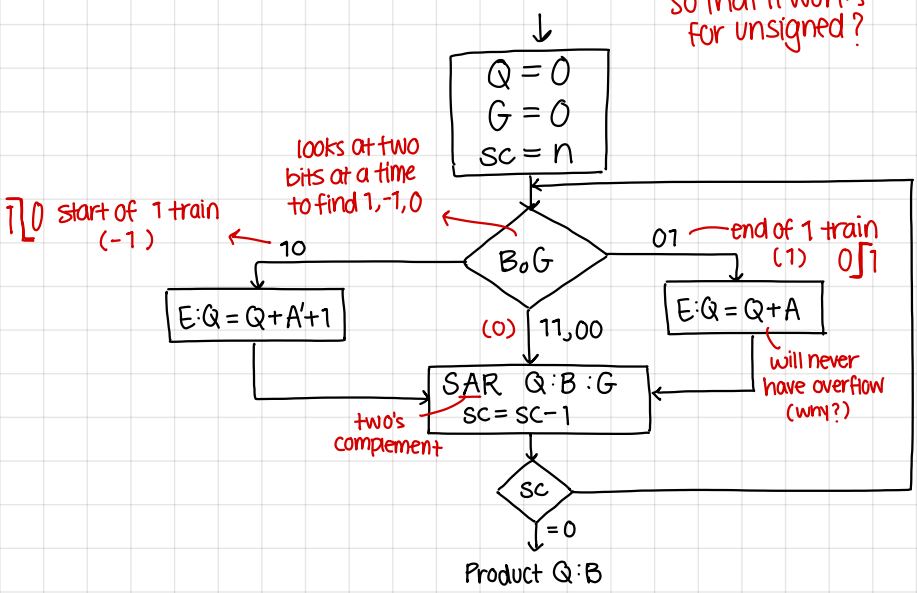
ex.

$15 \times 9 = 135$

$$\begin{array}{r} 10001\ A \\ \times\ 10111\ B \\ \hline 10000111 \end{array}$$

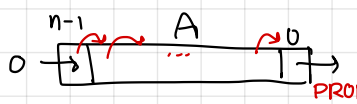


A x B → How do we change so that it works for unsigned?



★ Shifting

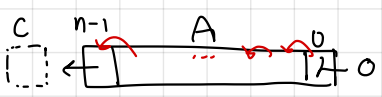
SHR



SHR → dividing by 2 if A is unsigned

PROBLEM: if 2's complement it will make the negative number, positive

SHL

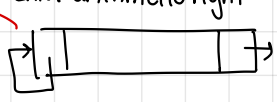


SHL → multiplying by 2 (only if A is unsigned and output digit is not last and not zero)

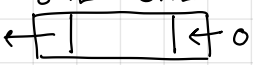
★ sign extension: in 2's complement representation, the extra shifted in from the right must be a 1.

SAR
shift arithmetic right

shifts in same bit that was already in the left most bit position



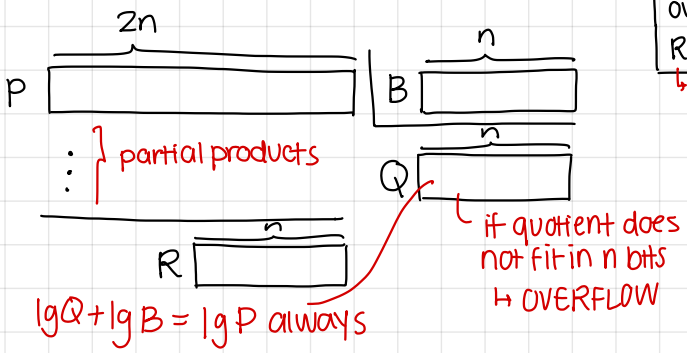
SAL = SHL

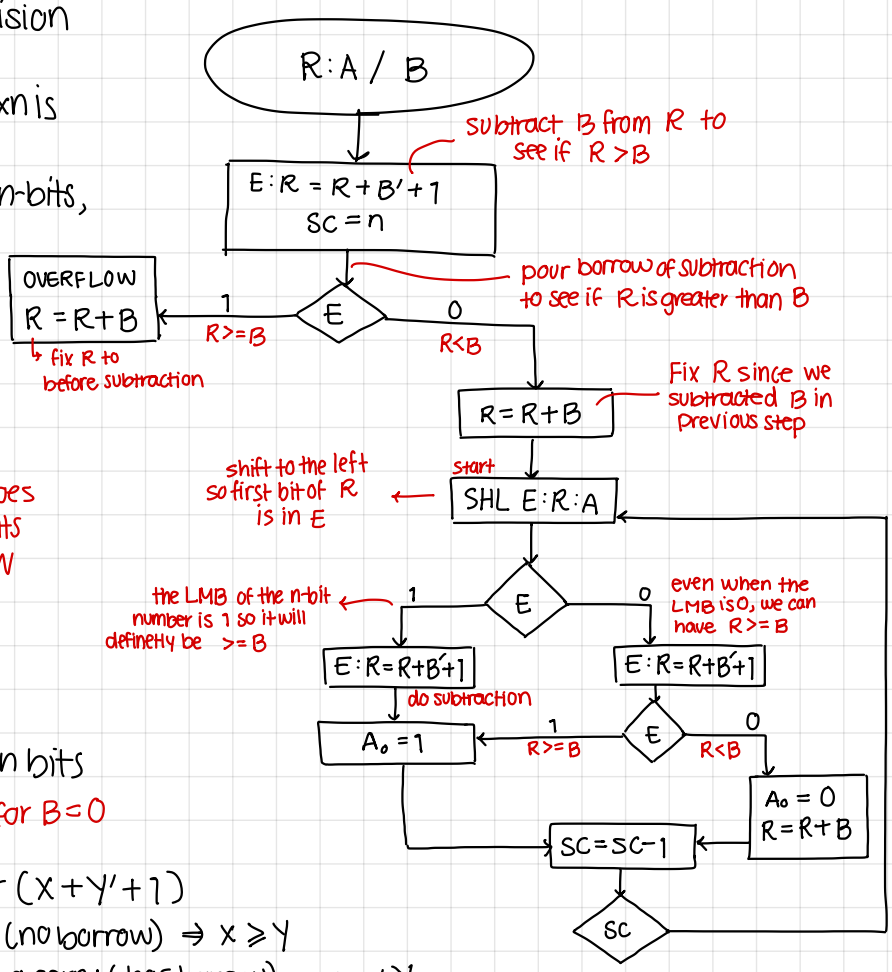


Divider

dividend — A / B — divisor

- Divider logic comes from the idea that division is the opposite operation of multiplication
  - ↳ in multiplication we store the product of nxn is stored in 2n bits
- We assume that the divisor of a 2n dividend is n-bits, and the quotient and result are n-bits.



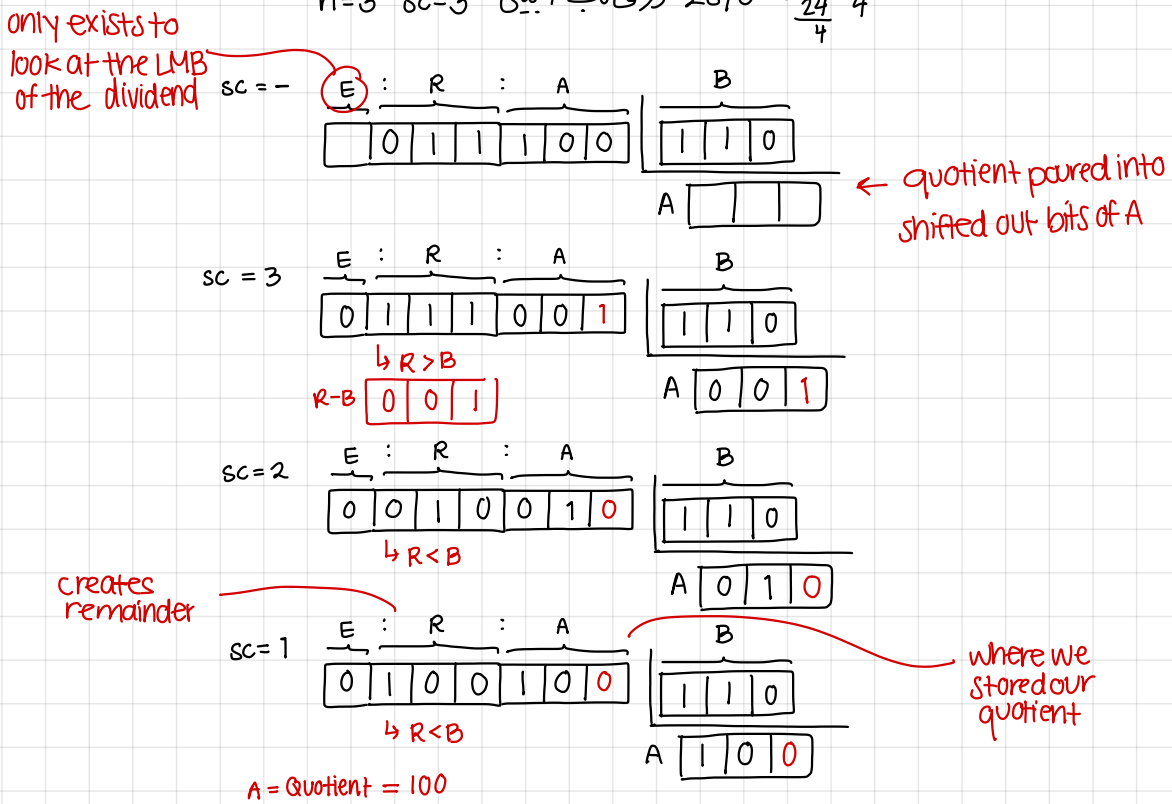


- we check if the answer fits in Q before we start division
  - ↳ Q will be more than n-bits if the first n bits of P are greater than B (also checks for B=0)

- All comparisons are made using a subtractor (X + Y' + 1)
  - ↳ if complement subtraction creates a carry (no borrow) ⇒ X ≥ Y
  - ↳ if complement subtraction doesn't create a carry (has borrow) ⇒ X < Y

ex

n=3 sc=3 ဖြေရှင်းပုံ 28/6 $\frac{28}{24}\ \frac{6}{4}$ $\frac{4}{}$



A = Quotient = 100

R = Remainder = 100

★ جلسه شانزدهم - سه شنبه ۸ اردیبهشت ★

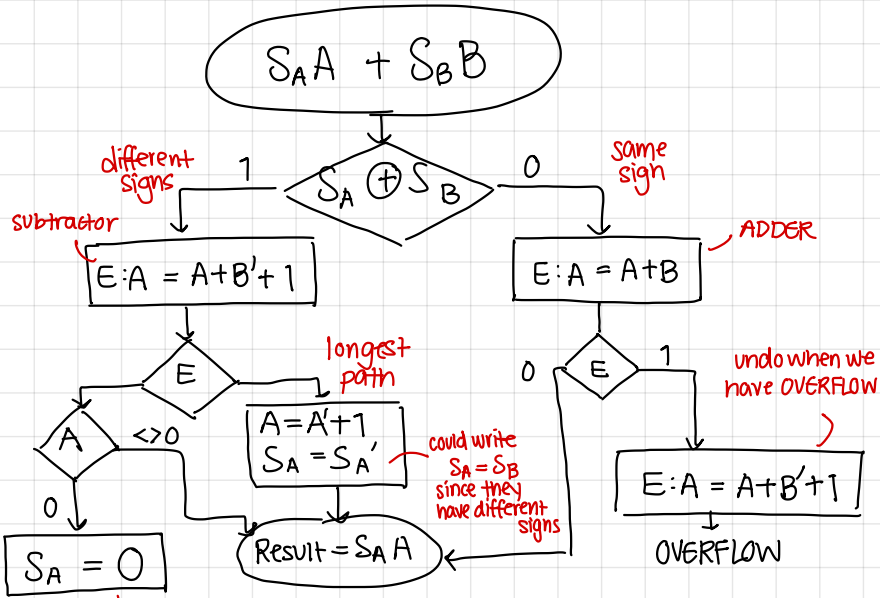
sign magnitude

- calculations are more complex (complex for addition subtraction)
  - ↳ we have to look at the different signs, whether to add or subtract them, we need both an adder and subtractor
  - ↳ simple for multiplication (all multipliers work the same) ★ easy for humans
  - ↳ simple for multiplication

- Compared to two's complement representation and calculations
  - \+ converting from data to IO and displaying is simple (good for score boards)
  - \+ easier to understand for humans
  - \+ addition is complex
  - − we have two zeros (must guarantee we won't have -0)
  - − multiplication is simple

★ we need addition/subtraction more for computers

- Adding sign magnitude numbers
  - ↳ Adding sign magnitude numbers is very similar to two's complement, except we now use the sign bit of the numbers to determine the sign of the sum



must abide by rule to not have -0

- Subtracting sign magnitude numbers

↳ $S_A A - S_B B = S_A A + S_B' B$

↳ everything else is the same as addition

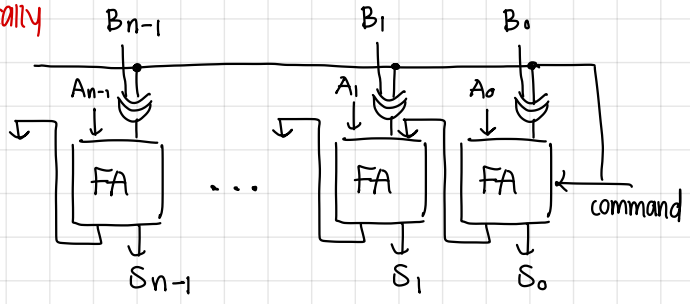
Here we don't have to check. If we have a borrow (wrong order) the correct answer can be found by two's complementing the result

normally we would check whether A<B and subtract the smaller number from the larger

- multiplying sign magnitude numbers
  - ↳ uses the same algorithm as unsigned
  - ↳ if $S_A \neq S_B$ we make the product negative

- dividing sign magnitude numbers
  - ↳ uses the same algorithm as two's complement (mathematically correct)
  - ↳ if $S_A \neq S_B$ we make the R and Q negative
  - ↳ here we don't check if the remainder is positive
    - > at the end we must add Q to the remainder and subtract one
      - ↳ this is not checked. our own CPU's return the remainder as a negative number

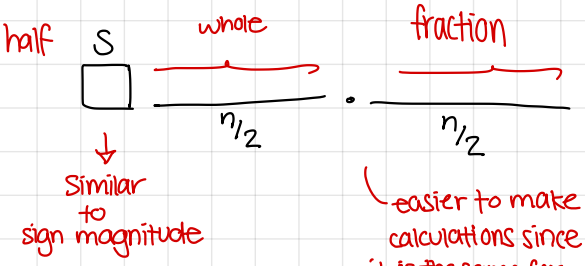
★ preferred design when we switch between addition and subtraction



Fixed point numbers

- fixed point numbers
  - ↳ we have a set point for where the point is in data
  - ↳ doesn't show all the best

usually bits split in half



Similar to sign magnitude

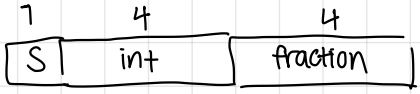
easier to make calculations since it is the same for all the numbers

\+ easy to understand for designers
\+ less expensive and less complex calculations
− little precision
− could have problems displaying the number

* PROBLEM: don't always need so much space for both whole and fraction of numbers
* BENEFIT: very simple calculations

- OVERFLOW
  - ↳ will almost always have "overflow" but instead we will say we have error (low precision)

- Is the data representation good?
  - → Total numbers that can be represented?
  - → largest number that can be represented (+Nmax)?
  - → Smallest positive number that can be represented (Epsilon)?
  - → resolution between two numbers?



→ $\begin{cases} 2 \times 2^{8} - 1 \\ 2^{1} \times 2^{n} - 1 \end{cases}$

→ $N_{max} = 15 + (1 - 2^{-4}) = 16 - 2^{-4}$

$N_{max} = (2^{n/2} - 1) + (1 - 2^{-n/2})$

→ $\epsilon = 2^{-n/2} = 2^{-4} = \frac{1}{16}$

→ resolution $2^{-f} = 2^{-n/2} = 2^{-4} =$

$0.5 = 0.1$

rounds whenever we can no longer display bigger fractions

$1 - 2^{-n/2} = 0.\underbrace{111111\ldots}_{n/2}$

$\frac{1}{2^{n/2}} = 0.\underbrace{000\ldots1}_{n/2}$

↓ not OVERFLOW, just not precise

→ OVERFLOW only when dividing by zero

we have numbers in decimal that are 0.5 but become 0.1 in binary representation

Adder
Subtractor
Multiplier
Divider
} for fixed point representation? *

Float Numbers

* جلسه هفدهم - یکشنبه ۱۵ اردیبهشت *

* Any non zero number can be normalized

- Normal number
  - ↳ A number with only one non-zero digit behind the point
  - ↳ Normalizing the number
    - > have one number behind the point
    - 48.13 → $4.813 \times 10^{+1}$
    - 101.11 → $1.0111 \times 2^{+2}$
    - 0.00110 → 1.1×2^{-3}
  - ↳ Any normalized number can be represented with:

excessthy 2's complement biased (to ma) | unsigned

S	Exponent	Fraction

always one | fraction | exponent

$$\Rightarrow \mp 1.\overbrace{\sim\sim\sim}^{\text{fraction}} \times 2^{\pm \overbrace{E}^{\text{exponent}}}$$

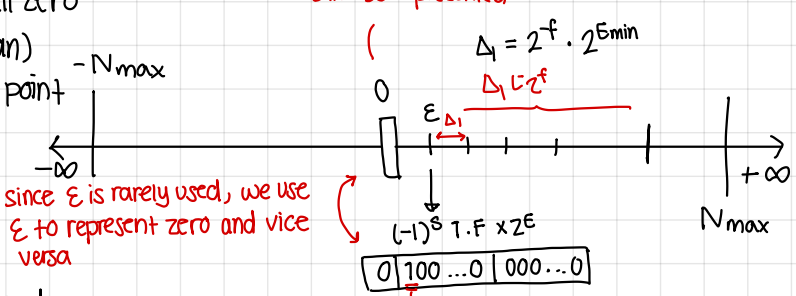
- > standard IEEE 746
  - 32 bit (single precision): S ← 8 → ← 23 → $(-1)^S 1.F (2^E)$
  - 64 bit (double precision): S ← 11 (+3) → ← 52 → $(-1)^S 1.F (2^E)^8$
- ↳ can't represent zero → solution: choosing a random less used representation to show zero

- The exponent is stored in excessthy 2's complement biased representation
  - ① comparing the exponents is easier
  - ② The representation of zero (non-normal) will be all zero
    - > we always check if the number is zero (nan) before making any calculations as a floating point

$E_{min} = -2^{e-1}$

$E_{max} = +2^{e-1} - 1$

can't be represented



since ε is rarely used, we use ε to represent zero and vice versa

the 1 doesn't "show" zero, we use biased so that all the bits are zero and it has better form

NaN : Not a Number = عدد

$\pm\pi$ $\pm e$ $\pm\infty$ $\pm\sqrt{2}$ ...

↳ frequently used numbers that cannot be represented

Octave 1 (equal amount of numbers but the distance is one greater):

$N = (-1)^S \times 1.F \times 2^E$

$N_1 = (-1)^0 \times 1.0 \times 2^{E_{min}}$

$N_2 = (-1)^0 \times 1.\underbrace{000...01}_{2^{-f}} \times 2^{E_{min}} = (1+2^{-f}) \times 2^{E_{min}}$ → $\Delta_1 = 2^{-f} \cdot 2^{E_{min}}$

$N_3 = (-1)^0 \times 1.000...10 \times 2^{E_{min}} = (1+2^{-f}+2^{-f}) \times 2^{E_{min}}$ → $\Delta_2 = 2^{-f} \cdot 2^{E_{min}}$

$\vdots$

$N_? = (-1)^0 \times 1.111...11 \times 2^{E_{min}} =$ (2^f th number)

Octave 2:

$(-1)^0\ 1.000...01 \times 2^{E_{min}+1}$

$(-1)^0\ 1.000...10 \times 2^{E_{min}+1}$

$\Delta_2 = 2^{-f} \cdot 2^{E_{min}+1} = 2\Delta_1$

$N_{max} = (-1)^0 \cdot 1.\overbrace{111...11}^{f} \cdot 2^{E_{max}}$

$(1+1-2^{-f} \cdot 2(2^e - 1)$

The numbers are more dense near zero and have larger distances as we get closer to N_{max}

↳ same amount of numbers being represented

↳ spread out exponentially

* In which octave is the distance equal to ε? in the $f+1^{th}$ octave

- A certain standard → removing the first octave to show NaN's

0	Emin	XX...X

→ when faced with this representation, look up table

overflow ↔ calculation error

+ complex representation and hard to understand

+ عددی + 1111 عددی (- bias 2 → adding 2^n instead of -2^n

look up table عددی ← جمع ← اشتباه ↵

* جلسه هجدهم - سه شنبه ۱۲ اردیبهشت *

$$A: [S_A \mid E_A \mid F_A] \rightsquigarrow (-1)^{S_A}\, 1.F_A \cdot 2^{E_A - b}$$
$$+\ B: [S_B \mid E_B \mid F_B]$$

(unsigned → E_A; the bias used → b)

$+1.001 \times 2^{+5}$
$+\ +1.0100 \times 2^{-3}$

- exponent allignment
  - bring number with larger exponent down to the smaller number
  - bring number with smaller exponent up to the larger number
    (this shift to the right is better because if data is lost the numbers are less significant)

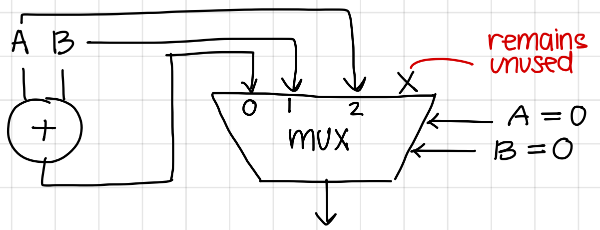
(reach smaller exponent to larger exponent)

$+\ 100100000.0000\ 2^{-3}$
$+\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 1.0100\ 2^{-3}$

$+\ 1.001\ \ \ \ \ \ \ \ \ \ \ \ \ 2^5$
$\ \ 0.00000001 0100 \times 2^5$ / better if the smaller number is ruined

- Addition
  1. Check for zeros $\begin{cases} A=0 \to \text{result}=B \\ B=0 \to \text{result}=A \end{cases}$
  2. Exponent Alignment
  3. Sign magnitude addition
  4. Normalizing the sum

$+1.01 \times 2^0$
$+1.00 \times 2^0$ ← OVERFLOW
اگر همفاز از سمت چپ
تفاوت کنیم یک شیفت به راست
$10.01 \times 2^0 \to 1.001 \times 2^1$



→ Handling the special case zero causes problems when designing hardware

- Subtraction
  1. Check for zero $\begin{cases} A=0 \to \text{result}=-B \\ B=0 \to \text{result}=A \end{cases}$
  2. Exponent allignment
  3. Sign magnitude subtractor
  4. Normalize the result

  (same steps as addition, only here we use B')

1.0001×2^0
-1.0000×2^0
$0.0001 \times 2^0 \to 1.0 \times 2^{-4}$

اگر همفاز از سمت راست
چند تا شیفت
UNDERFLOW
↳ when we reach a number that is 0.00... and we need to normalize

- Multiplication
  1. Check for zero $\begin{cases} A=0 \\ B=0 \end{cases} \to \text{result}=0$
  2. $S_A \oplus S_B$ to determine result sign
  3. Do normal unsigned multiplication
  4. Add the exponents (and an extra b)
  5. Normalize the result
     (when 1x1 no need to normalize)

* اگر مانتیس و اندازه را مکمل ۲ ذخیره کنیم؟ جمع کننده مکمل ۲

$$(-1)^{S_A \oplus S_B}\, (\underbrace{1.F_A \times 1.F_B}_{\text{unsigned}})\, (2)^{(E_A-b)+(E_B-b)+b}$$

(+b: returning back to bias)

- Division
  1. Check if zero $\begin{cases} A=0 \to \text{result}=0 \\ B=0 \to \text{OVERFLOW} \end{cases}$ (if bias 2 → ∞; only time we have overflow in floats)
  2. $S_A \oplus S_B$ to determine result sign
  3. Normal unsigned divider — the overflow that occured in normal division does not occur here → fixed with normalization (multiply by 2^{-k} to shift)
  4. Add exponents (+ extra b)
  5. Normalizing and undoing any shifts
     (when 1÷.5 need to normalize)

(if R ≥ A shift B to the right)

in order to be able to divide (2n/n) we must add 6 zeros to the left or right of the dividend

(1111 0000) 1110000

better to add on this side because adding it on the left will create unnecessary 0's in our result

3.
control unit

- RISC vs. CISC
- control unit
- Von Neumann Algorithm
- Instruction formatting
- RTL
- the Basic Computer

- pipelining
- I/O

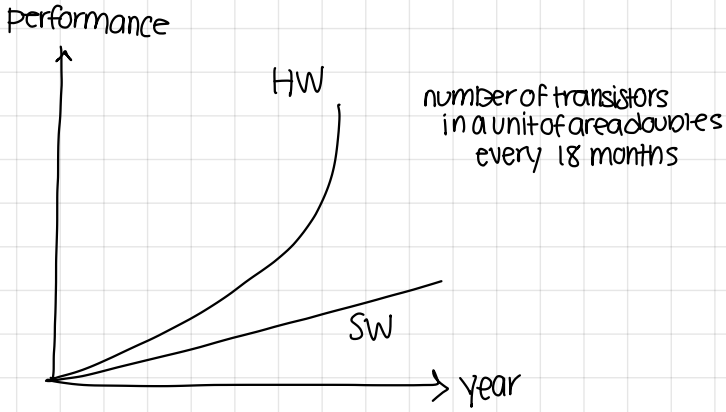
★ جلسه نوزدهم - یک‌شنبه ۱۷ اردیبهشت ★

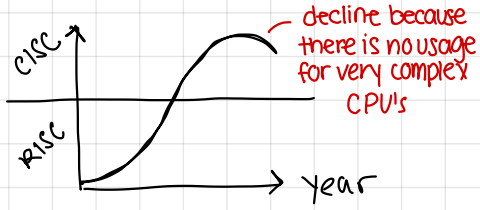
RISC vs. CISC

pentium bug

- Reduced Instruction Set Computer (RISC)
  - ↳ for general usage
  - ↳ includes embedded CPU's
    - >embedded CPU : low complexity CPU's that can only do a set of few tasks
      ex. projectors, digital watches, calculators, etc.
    - >of all CPU's :
      95% — 95 percent are embedded (RISC)
      5% — only 5 percent are intel/AMD desktop CPU's
- Complex Instruction Set Computer (CISC)
  - ↳ consists of very specific and complex capabilities
  - ↳ not often used by the general public ~ special capabilities are not useful to them, just expensive
- RISC vs CISC

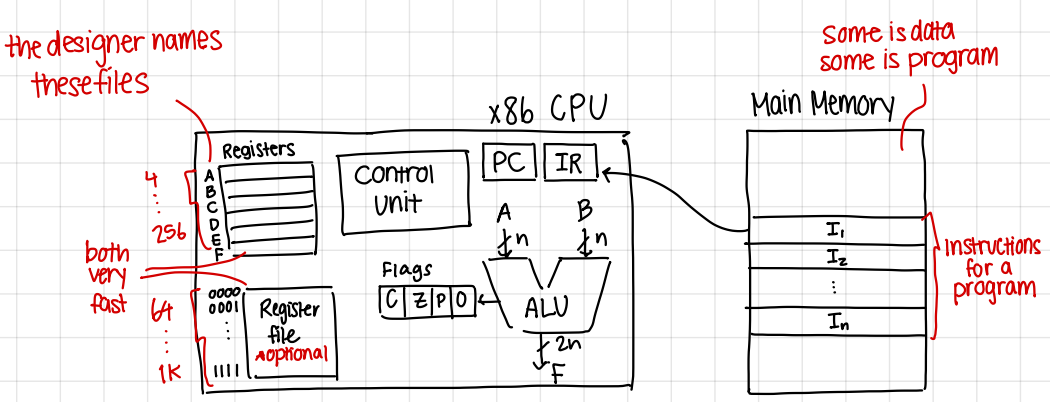
عامل مقایسه	RISC	CISC
تعداد دستورالعمل	کم	زیاد
تعداد کلاک هر دستور	یک	زیاد و متغیر
نوع طراحی دستورات پیچیده	نرم‌افزار محور	سخت‌افزار محور
میزان استفاده از RAM	زیاد	کم
طول و قالب دستورالعمل	ثابت	متغیر
شیوه آدرس‌دهی	کم	زیاد و متنوع
تعداد ثبات	زیاد	کم
تعداد دستورالعمل کار با حافظه	کم	زیاد
تعداد خط برنامه	زیاد	کم
اندازه برنامه	زیاد	کم
سرعت برنامه	؟	؟





* CPU runs the program that is loaded in the MM

* data and address have different register sizes

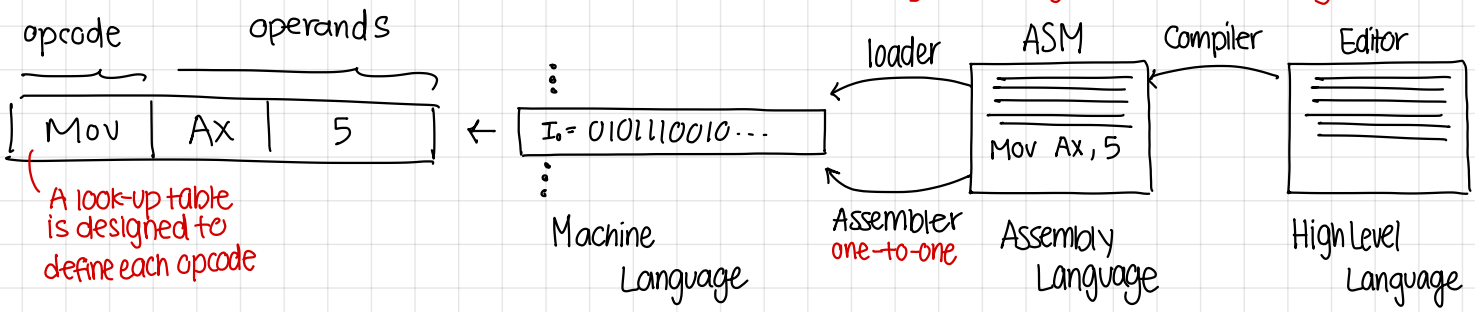


- **Program Counter (PC)**
  - ↳ An n bit register that stores which instruction from the Main Memory is being run
  - ↳ n is the size of an address in the Main Memory ($\lg_2(MM)$)

- **Instruction Register (IR)**
  - ↳ stores the instruction being run
  - ↳ An m bit register where m is a multiple of a word in the MM

- **Registers**
  - ↳ The registers in the CPU can be divided into two groups { General purpose registers, Special purpose registers
    - > General purpose : can be accessed by the user ~ how we get data
    - > Special purpose : only for CPU usage, shouldn't be accessed/changable by the user
  - ↳ number of registers can be between 4 to 256 bytes.
  - ↳ In some CPU's where we want more space, we add register files ~ registers are faster → prefered
    - > registers are used first, if more space is needed register file is used
    - > is between 64 to 1K bytes.

* each line in assembly is a binary instruction in memory



Instruction cycle (the time for one cycle)

Von Neumann CPU

non-Von Neumann

Von Neumann Algorithm for designing CPU

1- Instruction Fetch
2- Instruction Decoder
3- Operand Read — retrieving the data (from Instruction or Registers or MM) 5 vs. X[5]
4- Instruction Execute — only time we use ALU → step could be skipped ex. Mov, JMP, etc. don't need ALU
5- Result Writeback — could be skipped ex. JMP
6- Next line and go to 1 (pc = pc+1)

+ simple
+ makes sense
- forces us to do sequential programming
- step 4 (that takes more time) does not start before other steps
- we are only using the ALU 1/6 of the time

write ↘ $x = x + y$ ↙ read
read

* جلسه بیستم – سه شنبه 19 اردیبهشت *

look at where the data has to go through, any gates, buses, registers, ALU, etc.

- Steps to designing the CPU:
  1. Data path design
  2. Control unit design — like the select line of a mux, ALU operation commands, counters, sequencers, etc.
  - ↳ Data path design consists of determining any part that is involved with moving, storing, or reading data
  - ↳ Control path design consists of determining any part that controls the storing and execution of instructions and control
  - ↳ always design data paths first to determine all the necessary parts ~ shows what needs to be controlled too

- CPU's can be designed as:
  - von Neumann CPU's
  - non-von Neuman CPU's ~ any CPU that does not follow the algorithm ~ ex. keeping the instruction/data in the MM instead of bringing it to the CPU

- CPU's can be designed in two ways:
  - Single cycle – if our instruction cycle happens in one clock
    - ↳ the clock takes a longer time (not neccesarily better/faster)
  - Multi cycle – if our instruction cycle happens in multiple cycles — what we will be designing
    - (number of clocks in each cycle could vary
      - ↳ CISC usually has set number

- In earlier CPU's, first the CPU was designed then software was created to work with it
- Today, the software requirements are determined and then a CPU is designed for it

intel CPU's	4004	→	8085	→	8086 → ...
	4bit		8 bit		16 bit

- Hardware designers are provided with two things when designing a computer:
  - given:
    - Memory Dimensions — tells us desired memory size (for address/data registers)
    - ISA — should be able to answer all of our design questions

- Instruction Set Architecture (ISA)
  - ↳ A set of instructions and restrictions given to a CPU designer that they must provide for
  - ↳ An important part of having a good CPU instruction design
    - > ISA designers find what instructions should be added to the CPU that will be useful
    - > ISA must have a balance of complexity:
      - more instructions provide software engineers with more capabilities vs. a less complex ISA will be easier for

  - ↳ good ISA design means providing many features without making the hardware too complex

$ISA: \{I_1, I_2, \dots I_n\}$

ex. I_1 = MOV <dst>, <src> (operand1, operand2) ~ gives all the information an architect needs when designing
- ↳ * move op2 to op1
- ↳ <dst> = {A, B, C, D}

- ↳ these instructions can be given to both software and hardware designers
  - > Hardware designer can create cpu
  - > Software designer can create corresponding assembly language ~ can work in parallel
    - the source code that involves the commands in ISA

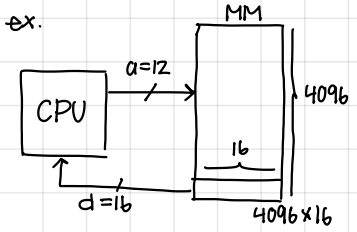
* what is the pin-out of a CPU?
greater pinout ~ more expensive
breaks quicker

Control Unit Design

- Designing the Control unit can be broken into 6 steps: (for our specific CPU)
  1. Determine memory and address path size
  2. Determine data path size
  3. Instruction set register design
  4. drawing the Von Neumann flowchart
  5. converting the flowchart to RTL
  6. Control unit HW design (physical)

- Step 1 of designing the control unit is determining the memory addressing and the address path size
  - ↳ the address path size will be $\lg_2$ of the size of our MM

ex.



- Step 2 of designing the control unit is determining our data path size
  - ↳ this is the smallest size of data we can transfer (usually a word)

- Step 3 of designing the control unit is designing our instruction set register (how we read instructions)

- The size of our instruction register must be a multiple of the size of a word (a line in our MM)
  - having multiple words is slower because we have that many times when to read from the memory we want to run one instruction

- Our instruction formatting must contain these fields:
  - ↳ opcode - determines the operation being done
  - ↳ operands (optional)
  - ↳ addressing methods (optional)
    - > each of the operands may have a different addressing method and that needs to be determined
    - > in some cases the addressing method is implicitly determined in the opcode and doesn't need to be assigned

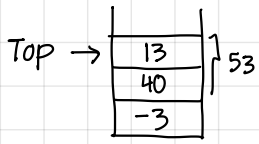
size varies based on number of operations in ISA (not dynamic)

$I =$ | ! ! | opcode | op1 | op2 | ··· |

operands

- An instruction can have varying number of operands (the more operands we have, the happier software designers are)
  - ↳ zero address instructions
    - > Instructions that don't have operands
    - > implemented with a stack
    - > no longer has "operand read" step in the Von Neumann algorithm (could skip the step (save time) or let clock go so it takes the same time as other instructions)

ex. ISA {ADD, SUB, MUL}



  - ↳ single address instructions
    - > the second operand is hardcoded
    - > uses "accumulator" register (AC) (↳ general purpose register used with ALU)

zero-address instructions — CLC

single-address instructions — ADD 5 (AC = AC + 5)

two-address instructions — ADD A, B (scr = scr + dst) (↳ ISA tells us how it is formatted)

three-address instructions — ADD A, B, C (only useful if CPU is aware such instruction exists; this would have to be broken into steps to do with two operands: A = A + B, A = A + C)

⋮

CISC → advanced vector computers — compute matrices, etc.

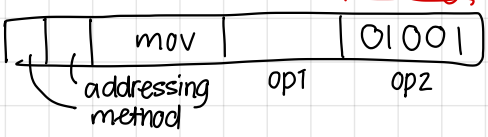
Addressing Methods

① Immediate addressing method
↳ can only use to show data you are currently working with
ex. Calculators, print statements

the actual number 5

Mov Ax, (5)

physically can see the number we are using



② Direct register addressing method
↳ provides addresses of data in the general purpose registers in the CPU
↳ faster than ③, but still only provides a limited number of registers

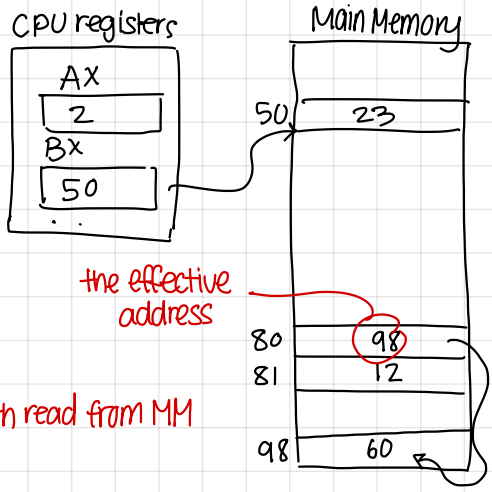
ADD AX, BX → Add data in register A to register B
50 + 2

③ Indirect register addressing method
↳ the value in the register is an address and the data is read from the MM — slower
↳ provides a limited number of pointers and is fast
↳ we can have more variables than ②

direct, indirect

ADD Ax, [Bx]
2 + 23

adds data in MM address 50



④ Direct memory addressing method
↳ the operand is an address in the MM
↳ faster than ⑤, has one less read ~ why ④ is better than ⑤

ADD [81], [80] — Both read from MM
98 + 12

⑤ Indirect memory addressing method
↳ the operand is an address to another address in the MM
↳ this method allows us to have pointers in the MM ~ why ⑤ is better than ④
> pointers to pointers can be implemented in software
> gives us the capability to determine the number of variables during runtime

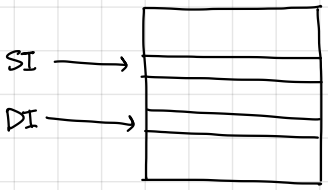
ADD [80], [[81]]
98
12 + 61

there's an address in 81, where we want that value

①-⑤ provides all of our needs
↓
the rest are for CISC or enhancing speed

⑥ Auto Increment / Decrement addressing method
↳ copies SI to DI, CX times — what the user input determines
> SI auto increments each time that memory address is accessed / written
> source index (SI) and destination index (DI)
> ((x = (y−1))++); — allows operations like this

MOVSB CX =4



⑦ Indexed addressing method (เรียงลำดับ)
↳ the SI register is an index register
>used for quickly going through loops/arrays

ADD AX, [SI]
adds all numbers in array which SI starts at to Ax

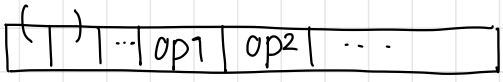
⑧ Base pointer addressing method
↳ finds addresses based on a base register

MOV Ax, [BP]+2
second byte of BP structure

* the index and base registers are special purpose registers in the CPU

• Operands may also be declared implicitly in the operation
ex.
CLC clear carry
STC set carry
two operations where the operand (C) is declared in the operation

addressing methods for op1, op2, ...

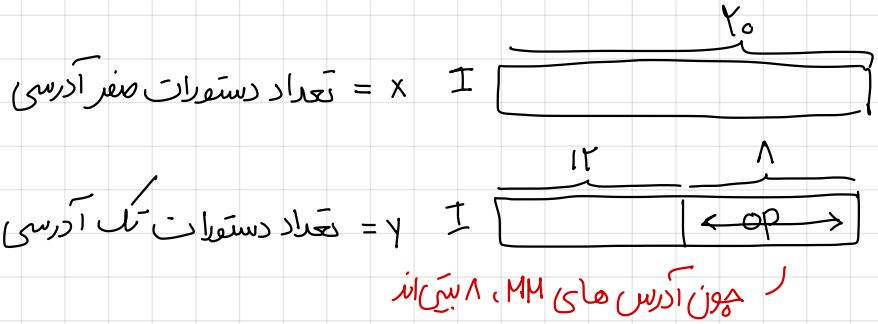


* if each operand has its own addressing method we need to determine each one

★ جلسه بیست و یکم - پنج شنبه ۲۱ اردیبهشت ★

سوالات طراحی قالب دستورالعمل

• مثال ۱ : در یک کامپیوتر با فضای آدرس دهی ۸ بیتی، چنانچه قالب دستورعمل متشکل از دستورات صفرآدرسی و تک آدرسی باشد و ۲۰ بیتی باشد، و تعداد دستورات تک آدرسی ۴۰ تا باشد، حداکثر چند دستور صفرآدرسی می‌توان داشت؟



چون آدرس های MM، ۸ بیتی اند

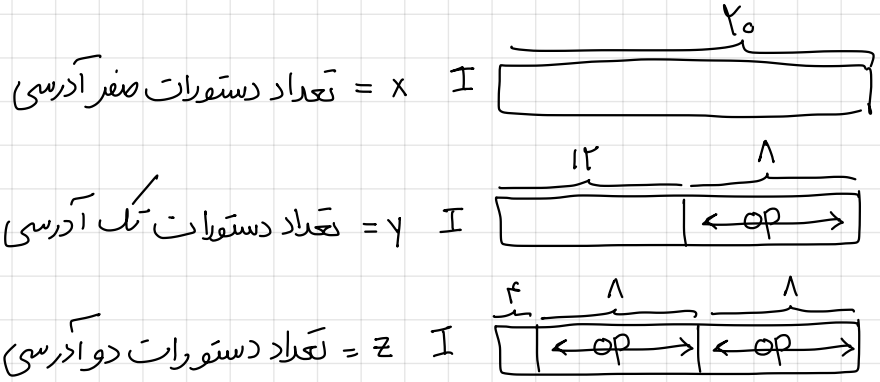
$x + y \leq 2^{20}$

$x + 40 \times 2^8 \leq 2^{20}$

$x \leq 2^{20} - 40 \times 2^8$

حالت‌های مختلف قسمت آدرس

• مثال ۲ : در یک کامپیوتر با فضای آدرس دهی ۸ بیتی، چنانچه قالب دستورعمل متشکل از دستورات صفرآدرسی، تک و دو آدرسی باشد و ۲۰ بیتی باشد، و تعداد دستورات تک آدرسی ۳۰ تا باشد، و تعداد دستورات صفرآدرسی ۴۰ تا باشد، حداکثر چند دستور دو آدرسی می‌توان داشت؟



صرفاً از اینجا می‌توان نتیجه گرفت که ۱۶ حالت داریم چون ممکن است در برخی حالت‌ها با دستورات دیگر conflict داشته باشیم

$x + y + z \leq 2^{20}$

$40 \times 2^0 + 30 \times 2^8 + z \times 2^{16} \leq 2^{20}$

$z \times 2^{16} \leq 2^{20} - 30 \times 2^8 - 40$

$z \leq \frac{2^{20}}{2^{16}} - \frac{30 \times 2^8}{2^{16}} - \frac{40}{2^{16}}$

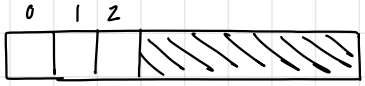
$z \leq 2^4 - \frac{15}{2^7} - \frac{5}{2^{13}}$

$z \leq 2^4 - 1 = 15$

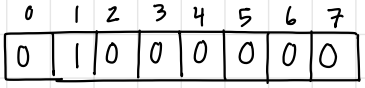
• مثال ۳ : ۸ بیت داریم. ۸ دستور ضمنی (صفر عملگری) را چگونه نمایش دهیم؟

در شرایطی که ۸ بیت فضا را داریم و محدودیت فضایی خاصی نداریم روش ب بهتر از الف است

روش الف) ۳ بیت برای نمایش ۸ حالت

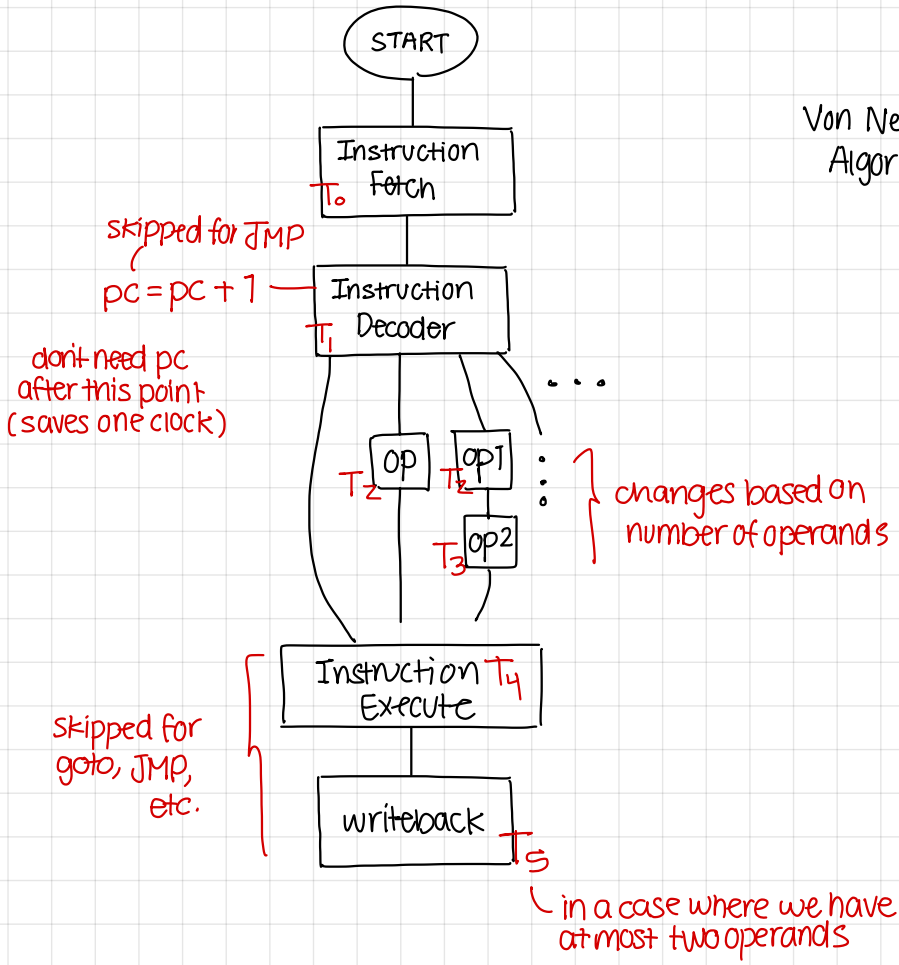


روش ب) هر بیت که یک باشد نشانگر اون دستور العمل است. ← در این روش decoder لازم نداریم

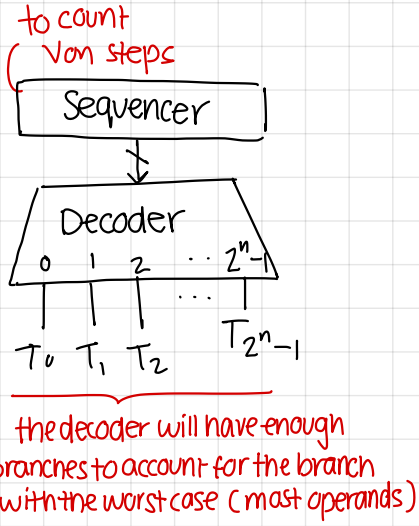


Von Neumann Flowchart

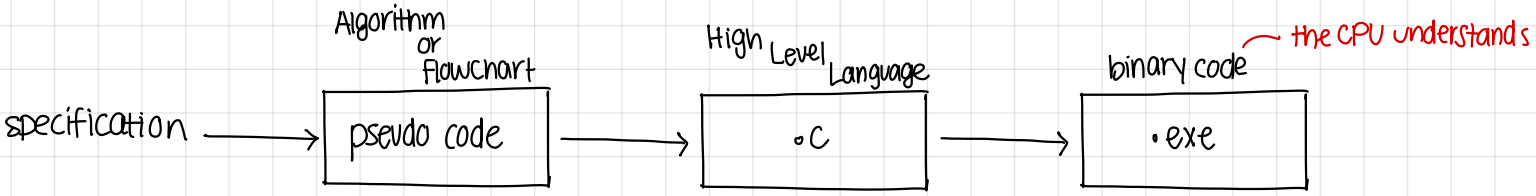
- Step 4 of designing the control unit is creating a Von Neumann flowchart for our ISA
- We follow the Von Neumann algorithm to create a flowchart for our specific CPU



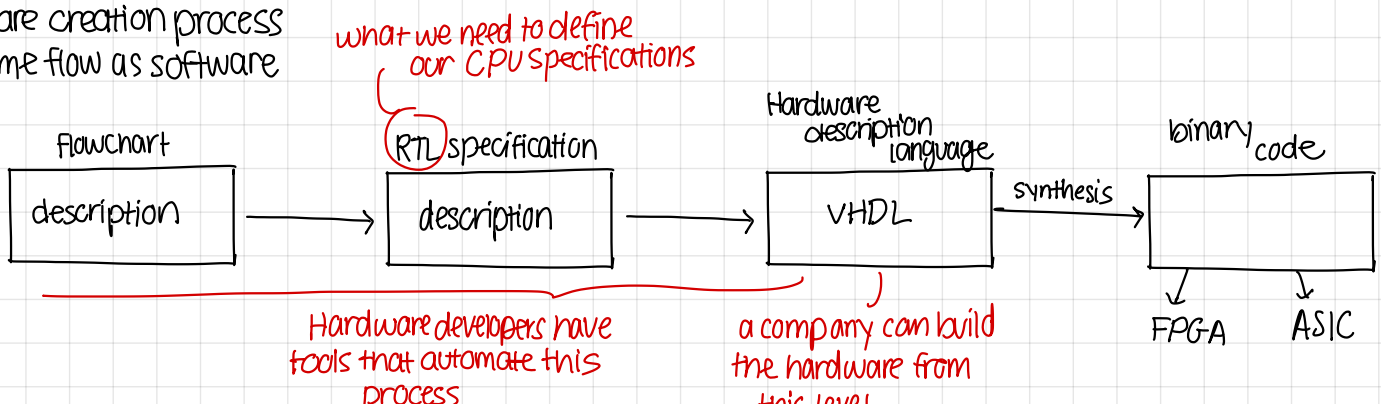
Von Neumann Algorithm
1- Instruction Fetch ~ memory read
2- Instruction Decoder
3- Operand Read
4- Instruction Execute
5- Result Writeback ~ memory write (sometimes)
6- Next line and go to 1



- Step 5 of designing the control unit is converting the Von Neuman Flowchart to RTL:
  - Software creation process



  - Hardware creation process
    ↳ same flow as software



★ جلسه بیست و دوم - پنج شنبه ۲۱ اردیبهشت ★

RTL I

- **Microoperations (μop)**
  - ↳ Any description for hardware in RTL is a series of microoperations
  - ↳ A microoperation can be defined as the tranfering of data from an expression of registers based on a condition

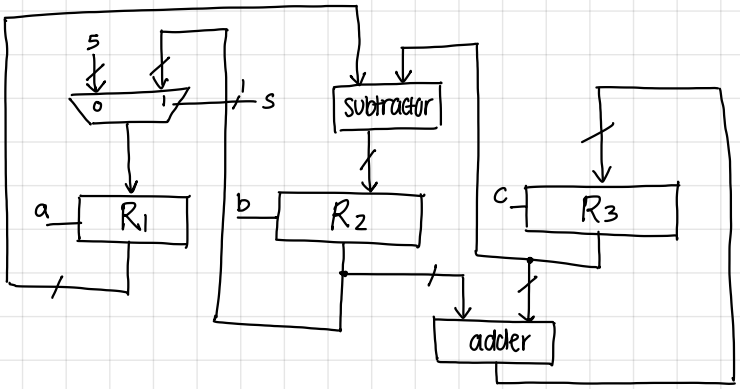
RTL
- Register Transfer Language
  - only describes the transfer between registers
    - must consist of microoperations
- Register Transfer Level / Layer

condition T or F

$T : R \leftarrow$ (expression of registers)

"انتقال داده از expression ها تحت یک شرط"

- **Converting from Hardware to RTL (HW ⟹ RTL)**
  - ↳ decide the condition of each register tranfer by the load input as well as the gates (ex. mux) that come before it



$S.a : R_1 \leftarrow R_2$

$S'.a : R_1 \leftarrow 5$

$b : R_2 \leftarrow R_1 - R_3$

$c : R_3 \leftarrow R_3 + R_2$

* we know that since the circuit has feedback, due to delays there could be an unknown state, so they all have to be sensitive to the rising edge of the clock

- **Converting from RTL to Hardware**
  - ↳ In this conversion, we can be faced with conflict:
    - > two conflicting conditions may be determining the value of a common register
  - ↳ we must determine priority, so that the hardware knows what to do when both conditions are true
  - ↳ the load of said register must still work for both conditions
    - > the load will be the OR of the condition statements

$a\bar{b} : R_1 \leftarrow 10$

$c : R_2 \leftarrow R_1 + R_3$

$\bar{a}b : R_3 \leftarrow R_2$

$abc : R_2 \leftarrow 1$

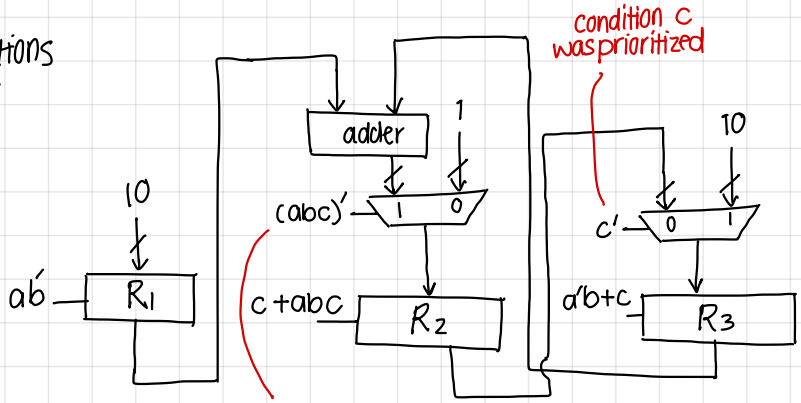
$\bar{c} : R_3 \leftarrow 10$

we OR the two conditions

conflict when both are true

↓

must prioritize one condition

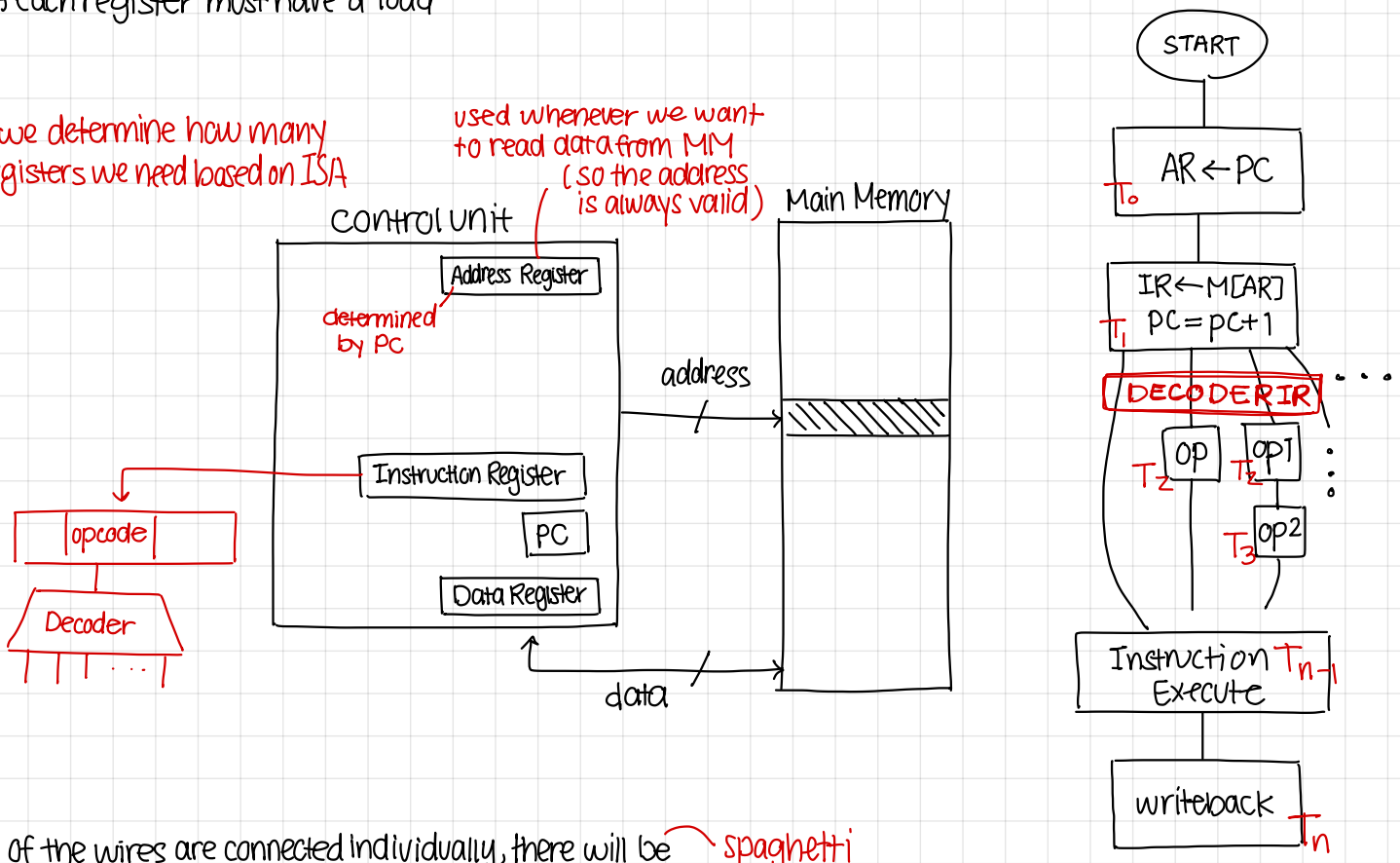


only $R_2 \leftarrow 1$ when condition abc exists, otherwise $R_2 \leftarrow R_1 + R_3$

↓

abc is prioritized to solve conflict

- Step 6 of designing the control unit is converting our RTL to hardware — physical design
  - ↳ in this final step, we draw the HW for our RTL
  - ↳ The registers needed in the RTL must be included here (including their size)
  - ↳ each register must have a load



- If all of the wires are connected individually, there will be a lot of wires which can lead to electrical problems / error — spaghetti wires
  - ↳ to solve this problem we use buses

* reading from memory could take more than one clock → we stay in the same t_i until acknowledge is activated then we move to t_{i+1}

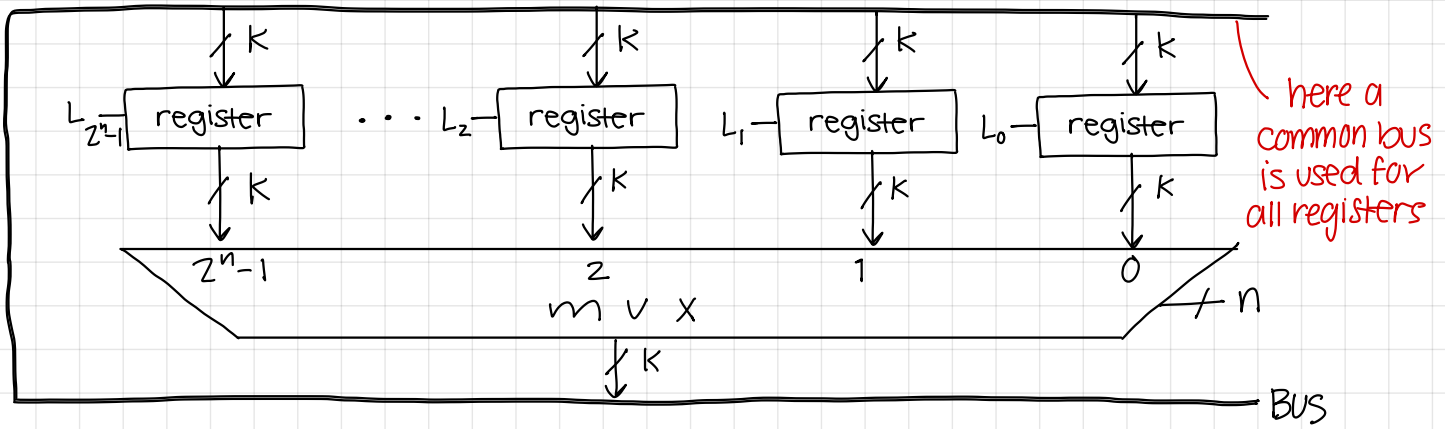
- Bus : A common wire that all the registers use to transfer data
  - \+ prevents spaghetti wiring (less chance of error)
  - − only one register can use it at a time (slower) — cheaper but slower
  - ↳ changes $O(n^2)$ ways of wiring to $O(n)$ where n is the number of registers
  - ↳ we can't directly connect the registers to the bus (can have conflict)
    - > connect a multiplexer to the output of each register

In the Von Neumann algorithm, all steps (except for pc=pc+1) occur sequentially already, so the problems with using a BUS does not affect it very much

only T_1 can't be done in one step

  - ↳ Data bus : bus data moves on (with the size of data)
  - ↳ Address bus : bus address moves on (with the size of MM addresses)

can be combined into one bus

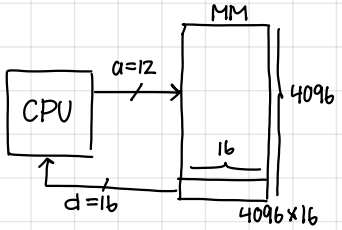


here a common bus is used for all registers

★ جلسه بیست و سوم - یک شنبه ۲۴ اردیبهشت ★

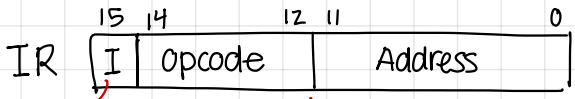
The Basic Computer

- The basic computer is a simple CPU model example.
- Memory { size of word → 16 bits = 2^4 — data bus needs 16 bits
  size of MM → 4096 words = 2^{12} — address bus needs 12 bits



- ISA: the basic computer has 3 groups of instructions:
  1. Memory instructions (7)
  2. Register instructions (12)
  3. Input/Output instructions (6)

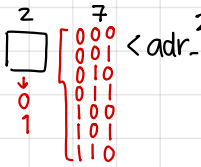
if IR = 2 words each program instruction takes up two lines and fills up the MM quickly



addressing mode — 2^3 different states, one remains unused

- Instruction format design
  - ↳ Instructions are stored in 16 bits
    - > left most bit specifies addressing mode (0 - direct, 1 - indirect)
  - ↳ has a single general purpose register → Accumulator
    - > is an operand and stores result in itself

Memory Instructions: $2 \times 7 \times 2^{12}$ ($2 \times 1 \times 2^{12} = 2^{13}$ states unused)



Register Instructions: 0 | 111 | 12 states

Input/Output Instructions: 0 | 111 | 6 states

any of the 2^{12} states can be chosen to represent the opcodes

- The basic computer uses a common bus for addresses and data
  - ↳ must use the larger size (16 bit)
  - ↳ deciding where the 12 bits are placed is up to the designer
    - ↳ better on the right

P_1: ← AR
P_2: ← AR
so any time $P_1 + P_2$

priority encoder

Anytime AR is on the right of a MOP (sometime its load must be activated)

	S_2	S_1	S_0
AR	0	0	1
PC	0	1	0
⋮		⋮	

Instead of drawing a mux, these show the select input of a mux

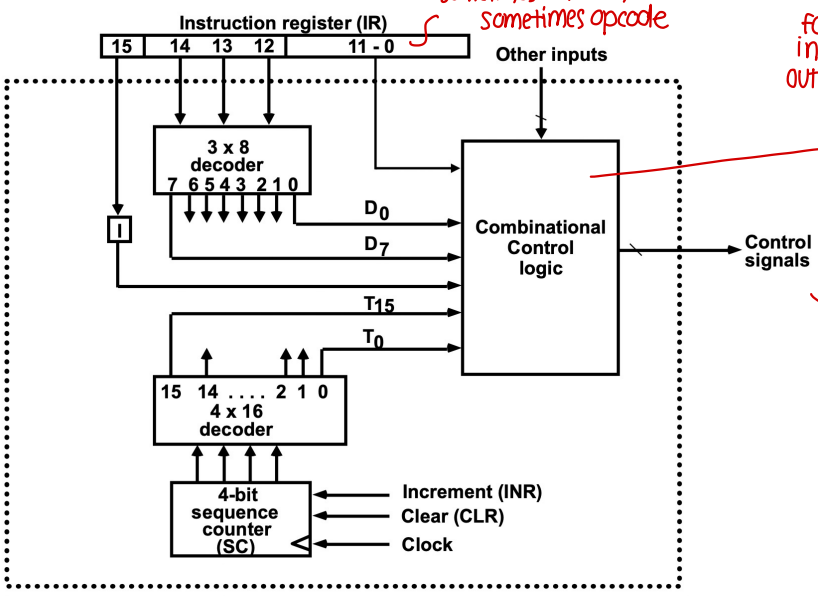
- Control Units are implemented in two ways:
  1. Hardwired Control
     - ↳ physical, permanent circuits (fast)
  2. Microprogrammed Control
     - ↳ programmed + optional (slow)

never have both in Von Neumann Algorithm
11 → ERROR
10 → write to MM
01 → read from MM
00 → nothing

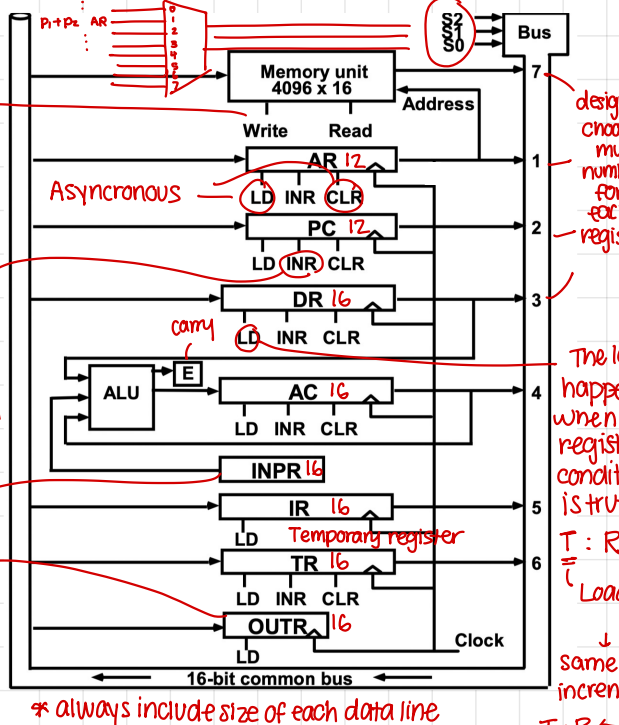
- Decoder is connected to sequencer to keep track of the step in the Von Neumann algorithm
  - ↳ size determined from max clocks in cycle

each register has an increment since its expensive to go to ALU each time we need to add 1

- The other decoder decodes the opcode



sometimes address, sometimes opcode



designer chooses mux number for each register

Asyncronous

for input/output

The load happens when the registers condition is true

T: $R_1 \leftarrow R_2$ (Load)

same with increment

T: $R_1 \leftarrow R_1 + 1$ and clr $R_x \leftarrow 0$

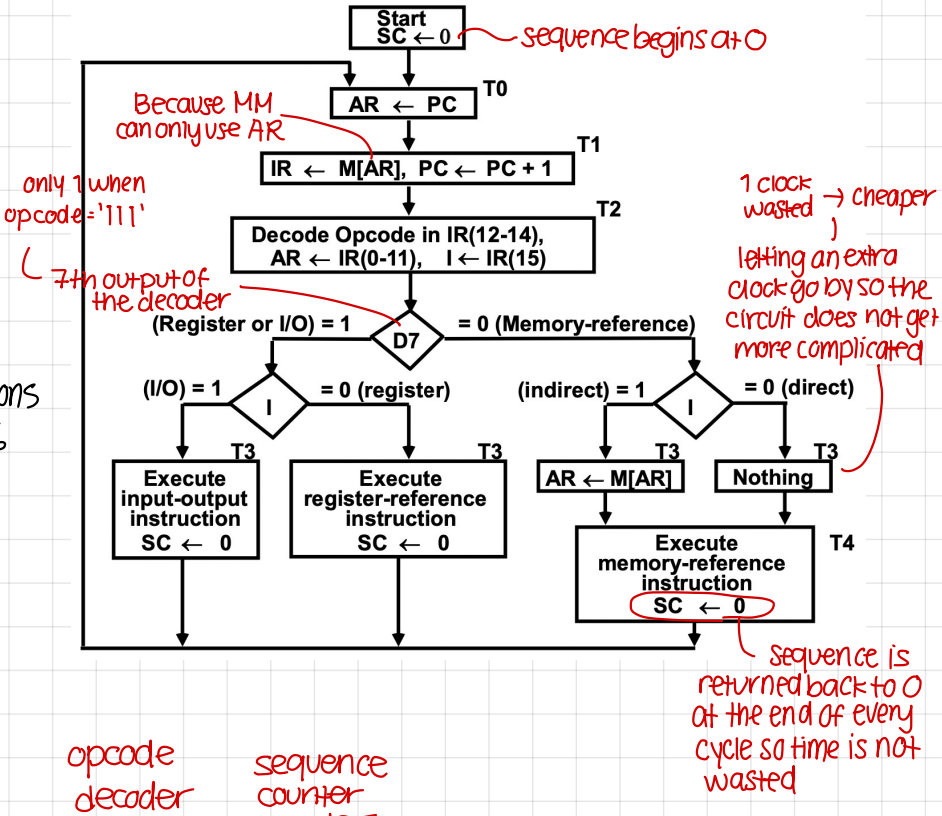
if said MOPs do not exist, we connect to gnd

* always include size of each data line

steps 1-3 done:
1. determining data path sizes
2. determining registers
3. Instruction formatting

- Here step 4 is completed, drawing the von Neumann flowchart for our ISA
  - ↳ the instructions that will take the longest are those referencing the MM
    - > we use that to determine our max T for the sequence counter and decoder

- For step 5 we write all RTL microoperations individually and then convert to HW for step 6



- ALU commands

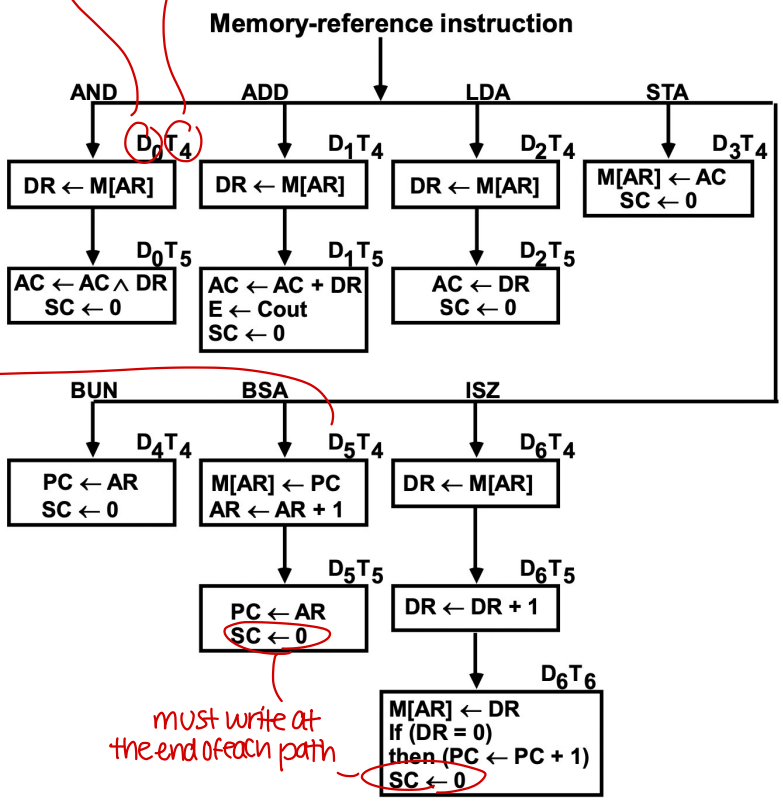
	C1	C2	
AND	0	0	D_1T_5
ADD	0	1	D_0T_5

↳ we OR the conditions in which we want to add

- Read / Write / Load commands
  - ↳ if M[AR] is on the right side, or the conditions of said Mop's
  - ↳ if M[AR] is on the left side, or the conditions of said Mop's

opcode decoder

sequence counter decoder



D_5 must be AND ed with T_4 otherwise all operations wired to T_4 will run each cycle at the same time

must write at the end of each path

Commonly used special purpose registers:

- ↳ PC – program counter
- ↳ AR – adress register
- ↳ DR – data register
- ↳ IR – instruction register
- ↳ OP1 – ALU operand 1
- ↳ OP2 – ALU operand 2
- ↳ SP – stack pointer ~ part of the memory is devoted to a stack
- ↳ RFAR – register file address register
- ↳ AC – accumulator
- ↳ TR – temporary register

جلسه بیست و چهارم - یک شنبه ۳۱ اردیبهشت

Performance Evaluation

- Performance
- Speed
- Throughput
- Bandwidth

Two computers could have the same clock, except one computer completes 4 instructions at a time, while the other completes one instruction

- SPEC : Standard Performance Evaluation Corporation
  - ↳ evaluates all cpu's at a specific standard
  - ↳ runs certain programs that best mimic the real instructions that will often be run on a CPU

- We want to compare Computer$_A$ ⇝ CPU$_A$ and Computer$_B$ ⇝ CPU$_B$
  - ↳ must compare different aspects to make a fair comparison

- **Performance**

$$\text{Performance} \propto \frac{1}{\text{execution time}} \rightarrow \frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution Time}_B}{\text{Execution Time}_A}$$

Not valid for just one program, A could be better than B in a specific area/operation ↓ we test various programs

  - ↳ the better performance of CPU$_A$ in one program does not guarantee that it is better overall
  - ↳ if the OS is also involved, running the one program could be run along many other (very large error)
    - > only valid if its a computer running only the one problem (no jumping threads)

- **Execution time**
  - ↳ in order to calculate the execution time of an instruction in the CPU, we must have the CPI
  - ↳ CPI : clocks per Instruction

$$\text{Execution time} = \sum_{i=0}^{n} (\text{\# of clocks for instruction } i) \times \frac{1}{f} \simeq n \times \overline{CPI} \times \frac{1}{f}$$

CPI$_i$ — frequency of the clock — average number of clocks per instruction

$$CPI = \frac{\text{\# of clocks}}{\text{\# of instructions}} \qquad IPC = \frac{\text{\# of instructions}}{\text{\# of clocks}}$$

کار در خانه :
- ★Calculate the CPI of your own computer
  - ↳ look at ISA add clocks for each one ...
- ★Calculate MIPS

- **Million Instructions per second (MIPS)**
  - ↳ execution time alone did not give a good enough estimate of how well the CPU works overall

$$MIPS = \frac{\text{\# of instructions}}{10^6 \times \text{\# Clocks} \times \frac{1}{f}} = \overline{IPC} \times f \times 10^{-6} = \frac{1}{\overline{CPI}} \times f \times 10^{-6}$$

same program takes less instructions, but MIPS is lower

$MIPS_A = 2$ ← k instructions — one program — 2k instructions → $MIPS_B = 3.5$

  - ↳ MIPS is still not a perfect way to evaluate performance

- **Amdahl's law**
  - ↳ if α percent of a program has sped up by n, the speed up of the program is:

$$\text{Speedup} = \frac{1}{(1-\alpha) + \alpha/n}$$

proof:

$$\frac{t_{old}}{t_{new}} = \frac{t_{old}}{t_{old}(1-\alpha) + \frac{t_{old}\alpha}{n}} = \frac{1}{(1-f) + f/n}$$

unchanged part of the program — speed up of said program — percentage of the program that has sped up

★ جلسه بیست و پنجم - سه‌شنبه ۲ خرداد ★

سوالات محاسبه کارایی

مثال ۱ . در یک کامپیوتری متشکل از دستورات زیر پس از بهینه سازی متوجه شدند که ۱۰٪ دستورات INT به همراه ADD هستند. لذا دستور جدیدی به نام ADDINT ساخته شد که CPI آن ۵ است. مشخص کنید میزان افزایش سرعت چقدر است.

دستور	درصد استفاده	CPI
INT	۲۰	۱۰
ADD	۳۰	۱۸
MUL	۱۰	۲۰
SUB	۴۰	۸

$T_1 \simeq n \cdot CPI \times \frac{1}{f}$ — $n = 100$ دستورات قبل از بهینه سازی

$T_2 \simeq n' \cdot CPI' \times \frac{1}{f}$ — $n' = 98$ دستورات بعد از بهینه سازی

$$CPI_1 = \left(\frac{۲۰}{100} \times 10\right) + \left(\frac{۳۰}{100} \times 18\right) + \left(\frac{10}{100} \times ۲۰\right) + \left(\frac{۴۰}{100} \times ۸\right) = 10/۸$$

$$CPI_2 = \left(\frac{18}{98} \times 10\right) + \left(\frac{۲۸}{98} \times 18\right) + \left(\frac{۲}{98} \times ۸\right) + \left(\frac{10}{98} \times ۲۰\right) + \left(\frac{۴۰}{98} \times ۸\right) = 10/۳۰۶$$

$$\text{speed up} = \frac{T_2}{T_1} = \frac{100 \times 10/۸ \times \cancel{f}}{98 \times 10/۳۰۶ \times \cancel{f}} = 1/۰۳۹۶$$

* در این سوال، α یا درصدی از برنامه که تسریع یافته را دقیق نداریم چون که تعداد خطهای برنامه با این تغییر، عوض می‌شود.

$\frac{۲}{98}$ تنها برای بهینه سازی درست است نه برای $1-\alpha$

← اگر با وزن $\frac{100}{98}$ قانون آمدال را حساب کنیم درست می‌شود.

دستور ADDINT که قبلا در دو دستور اجرا می‌شد حالا در ۱ دستور انجام می‌شود

مثال ۲ : در یک کامپیوتر دستورات گرافیکی را ۲ برابر سریع کرده اند که این دستورات ۳۵٪ کل دستورات برنامه‌ها را تشکیل می‌دهند. تسریع؟

$f = ۳۵\% = 0/۳۵$

$n =$ برابر ۲ $= ۲$

$$\text{Amdahl's law} : \frac{1}{(1-\alpha) + \frac{\alpha}{n}} = \frac{1}{(1 - 0/۳۵) + \frac{0/۳۵}{۲}} = \boxed{\text{تسریع} \quad 1/۲۱}$$

مثال ۳ : مهندسین پس از بررسی ISA، بهینه سازی انجام داده اند بطوری که توانستند CPI دستورات را ۴۰٪ درصد بهبود دهند ولی متاسفانه فرکانس پردازنده ۸۰ درصد بدتر شد. میزان تسریع؟

$\text{Execution time} \propto CPI \times \frac{1}{f}$

CPI ۴۰ درصد بهتر شده — فرکانس ۸۰ درصد بدتر شده

$$\text{Speedup} = \frac{t_{old}}{t_{new}} = = \frac{\text{Exec old}}{\text{Exec new}} = \frac{N \times CPI_{old} \times f_{old}}{N \times CPI_{new} \times f_{new}} = \frac{1}{0/6} \cdot \frac{0/2}{1} = \frac{1}{3}$$

Pipelining

"دست به دست کردن"
can't wait for one task to finish to start another

- we use pipelining when we have a lot of repeated things to do → A series of tasks being repeated

Inputs → (Combinational Circuits, T) → outputs

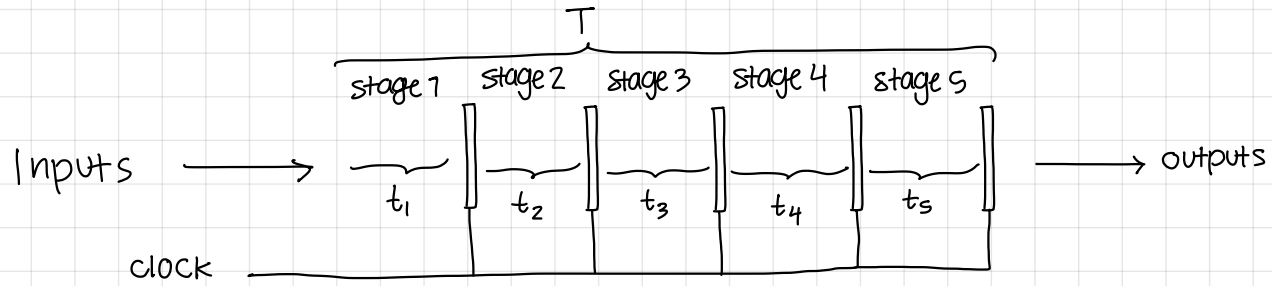
Total time = N x T

N = # of inputs (jobs)

pipelining is good whenever N or T is very large (perfect when both)

$$\text{minimum frequency} = \frac{1}{\text{largest delay in the combination circuit}}$$

- without pipelining, N inputs will take T x N time to be executed.



Total time = $(5 + (N-1)) \times t$ (5 = number of stages)

$t / \text{Period} \geq \max(t)$ ~ the frequency must be equal to the largest t_i

- By dividing the process into smaller steps, it can clearly be seen that when the first part of the task is done, it will remain unused until the process ends and another input is taken
  ↳ with pipelining, we begin another process right after, hence speeding up N processes

- The speed up that occurs with pipelining can be calculated by the following formulas:

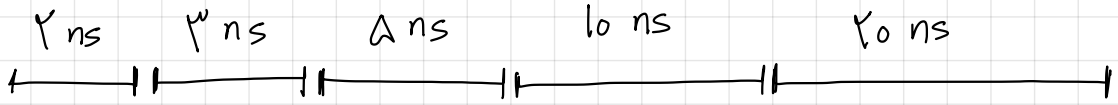
① $\text{Speed up} = \frac{N \times T}{(k + (N-1)) \times t}$

② $\text{speedup when } N \to \infty \approx \frac{T}{t}$

③ $\text{speedup when } N \to \infty \text{ and } t_i \text{ of the stages are equal} \approx \frac{T}{\frac{T}{k}} = k$ ~ equals the number of stages

(more stages → more speedup)

مثال: فرض کنید ۱۰۰۰ تا input داریم و ۵ تا مرحله به صورت زیر است. میزان تسریع؟



$$\text{Total Time} = \frac{1000 \times 20\,ns}{(5 + 999) \times 20\,ns} = \frac{2000}{1004} \approx 1.98$$

(t_{max} = 20 ns)

ب) ۱/۹۸

الف) ۲

ج) ۴

ب) ۵

غلط ← تنها زمانی تسریع اندازه تعداد مراحل است که N به اندازه کافی بزرگ باشد (∞)

- The Von Neumann algorithm is happening ∞ times → we can pipeline
  - ↳ we create 5 stages based on the already existing steps in the algorithm
    1. Instruction Fetch
    2. Instruction Decode
    3. Operands read
    4. Instruction execute
    5. Result writeback

- Space time diagram of 5 Instructions being pipelined:
  - ↳ The height of the diagram changes based on the number of stages
  - ↳ The width of the diagram depends on the number of inputs (∞)
  - ↳ A single row of this diagram shows what each stage is doing over time
  - ↳ A single column of this shows what state each instruction is in during that clock
  - ↳ the n^{th} instruction is done at time $n+4$ (number of instructions)

Clock	1	2	3	4	5	6	7	8	...	T
IF	I_1	I_2	I_3	I_4	I_5					
ID		I_1	I_2	I_3	I_4	I_5				
OR			I_1	○	I_2	I_3	I_4	I_5		
EXE				I_1	○	I_2	I_3	I_4	I_5	
WB					I_1	○	I_2	I_3	I_4	I_5

when we face hazards like data dependencies, "bubbles" are created in our pipeline
↳ become vacant stages

if executing takes more than one clock we can divide into stages $exe_1 \dots exe_m$

- There are cases where we can't start stages of an instruction without the previous instruction completing (hazards)
  - ↳ we can pause the dependent instruction → stall
  - ↳ creates "bubbles" → vacant stages resurface in every stage after

- where else can we use pipelining in the computer?
  - cache X → no spatial or temporal locality
  - ALU ✓ → ex. array multiplier
    - ↳ stage after every AND or every addition

- In the best case, the computers speed is multiplied by 5
  - ↳ not realistic because there are hazards

superscalar computers

solvable issues

- Hazards in pipelining (ex. when we use writeback)
  1. data dependency → when there is data dependency between instructions
     - > possible solution: instruction reorder – moving dependent instructions (done both by the programmer and compiler)
  2. resource conflict → when both instructions require access to a common resource (both instructions needing the ALU)
     - > possible solution: adding new resources (ex. separate adder for a register) (costly ($))
  3. when we have jmp/goto instructions
     - > On average of every 6 instructions, one of them contains jmp ($\frac{1}{6}$)
     - > worse case is when we have conditional jumps since the next instruction is determined by flags (still don't know if we go to I_2 when we operand read in I_1)
     - > possible solutions:
       - Take one randomly – take the jmp's if or else randomly
       - Interleave execution – do both the if and else before hand
       - Branch predictor – guesses which one by holding the history (if a guess worked once it'll be used again)
       - Branch target buffer (BTB) – buffer that stores whether it was if or else before
       - Loop buffer – buffer containing the n most recently fetched instructions (for loops)

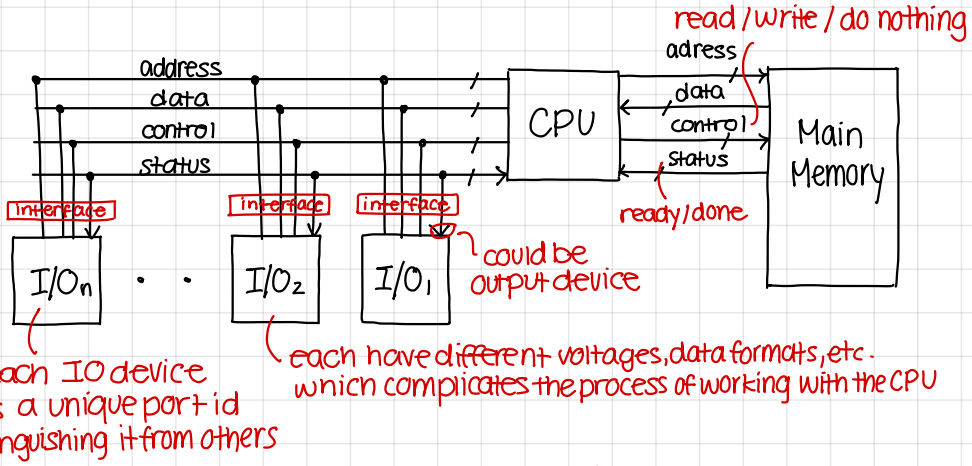
$I_2 \neq I_{1+7}$ "d/c"

★ جلسه بیست و ششم - یک شنبه ۹ خرداد ★

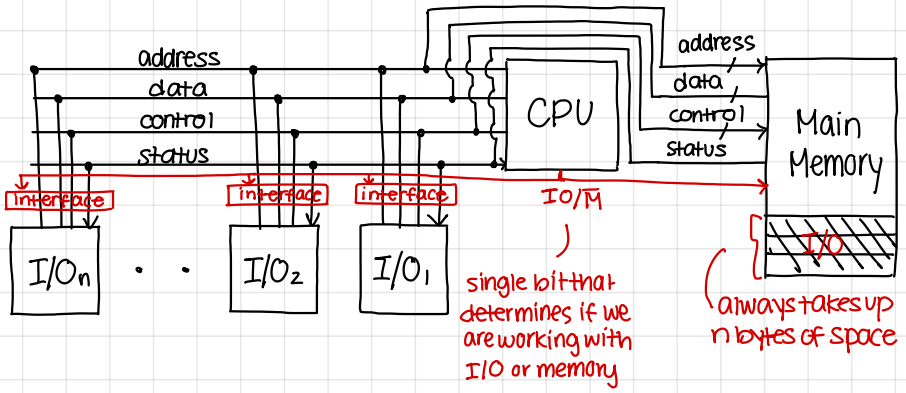
Input/Output Organization

- I/O devices can be connected to the CPU and Main Memory in 3 methods:

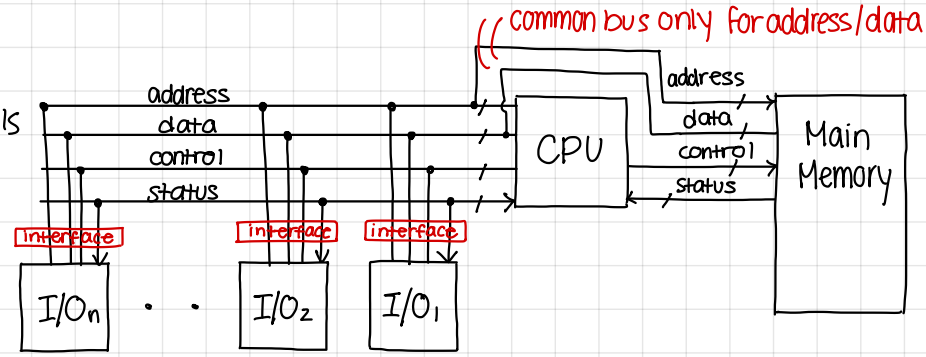
① Seperate buses for data, address, control, and status for each I/O.
having this many pins can cause error
↳ the Main memory and I/O devices are treated as seperate entities
↳ not practical - too many buses going into the CPU
> hard to manage



② common buses for address, data, control and status
memory mapped I/O
↳ the Main memory is being treated as an I/O device
↳ we read and write from I/O devices using memory addresses
↳ wastes n bytes of space in the memory for I/O devices
↳ memory is being used unnecessarily each time we use an I/O
↳ solvable by adding an IO/M̄ bit
↳ also won't take up the extra space anymore



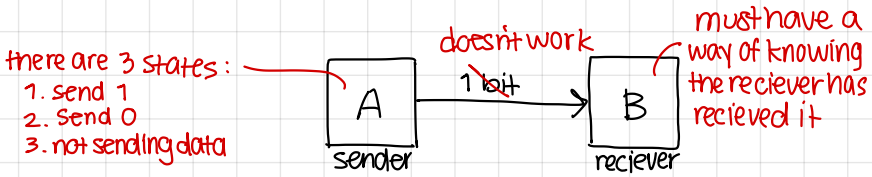
③ Seperate buses for control and status, and common buses for address and data
Isolated I/O
↳ a common bus is used for both I/O and memory, and each one knows if that address is for them based on the seperately activated controls
↳ requires more instructions in our ISA to get/give data from our I/O devices
ISA { IN (port), OUT (port) }
↳ in mapped memory IO devices could be accessed with the same read/write controls



- **Interfaces**: interfaces are used to maintain/organize the connections between the CPU and I/O devices
↳ I/O devices have various types, and all the variety cannot be handled by CPU designers
isolating the CPU from the I/O device
↳ interfaces have 3 main tasks:
① Adjusting voltage levels of the I/O device
② Creates a common data-format design between the I/O and CPU
↳ the interface is responsible for making this change
③ Increases speed between I/O and CPU

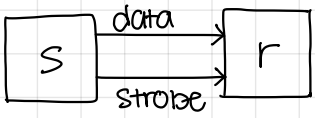
↳ added part on motherboard (not our concern)

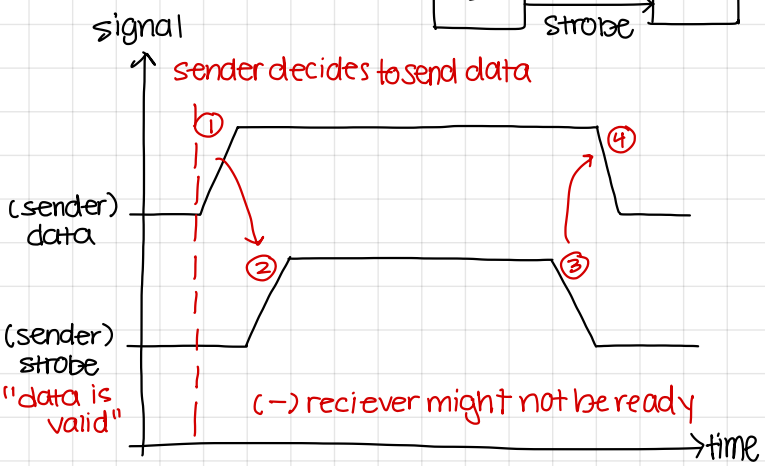
- Data needs to be transferred between the CPU and I/O devices
  - ↳ the devices don't necessarily have a clock, and we want to use minimum wires
- A single bit cannot be used to transfer a bit of data



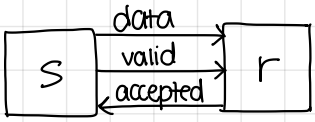
- In general, there are 4 ways to send data between two chips:

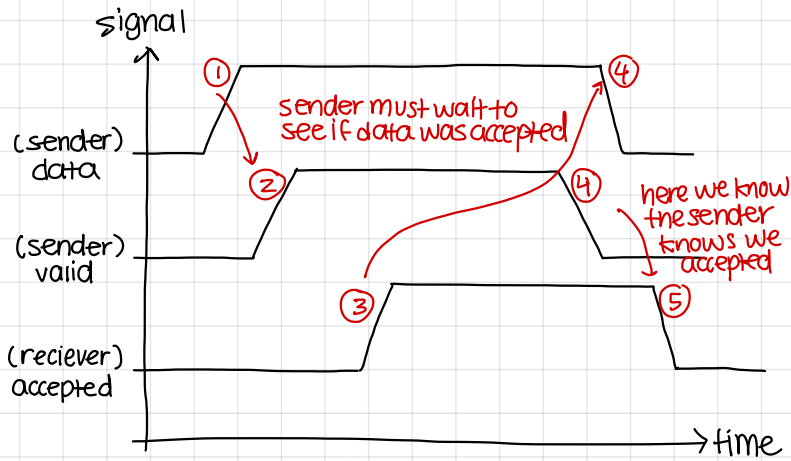
(1) Strobed data transfer
(a) sender controlled



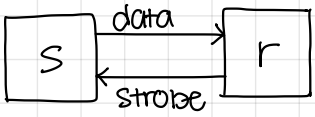


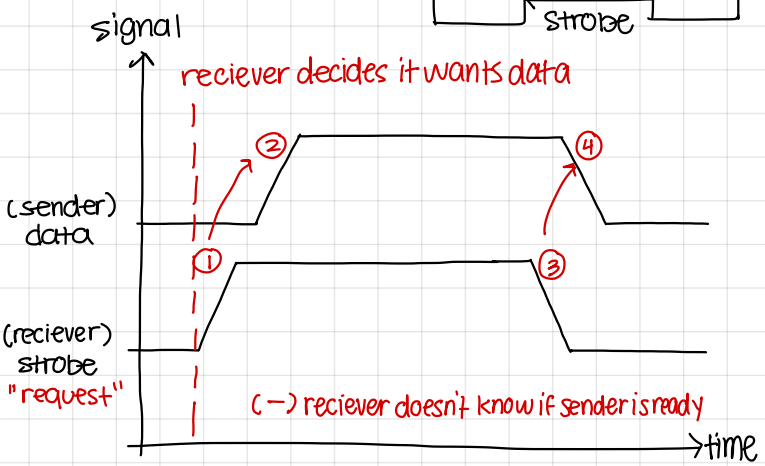
(3) Handshaking data transfer
(a) sender controlled



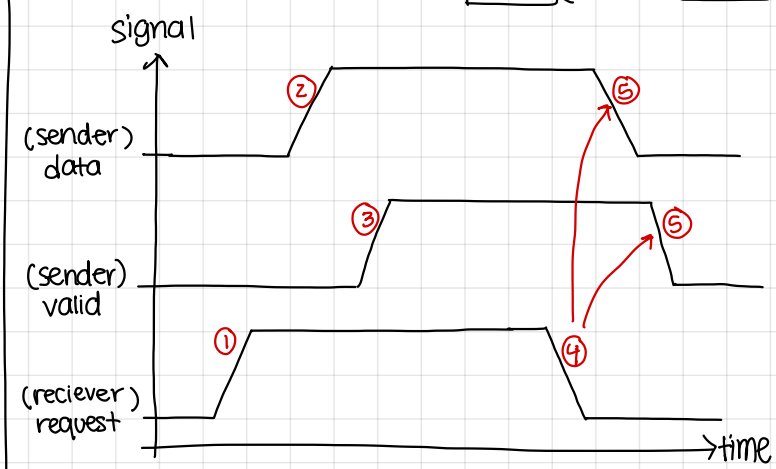


(2) Strobed data transfer
(b) reciever controlled





(4) Handshaking data transfer
(b) reciever controlled



- when the CPU wants to read data:
  - ↳ the CPU sends an address to be read from memory
  - ↳ memory sends back data and shows it is valid

} Handshaking type (b)

- when the CPU wants to write data:
  - ↳ the CPU sends the data and instruction
  - ↳ waits for main memory to complete writeback

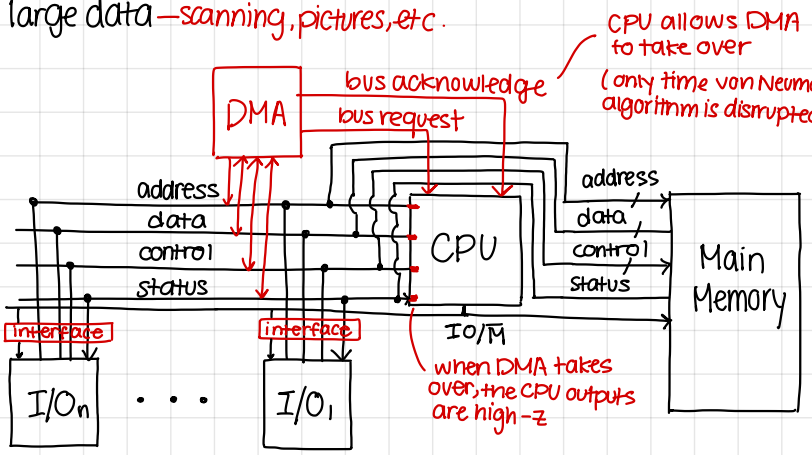
} Handshaking type (a)

- Handshaking methods are how data is transferred between the CPU and I/O devices
  - ↳ works the same as memory

★ جلسه بیست و هفتم - سه شنبه ۸ خرداد ★

Direct Mapped Access

- the I/O can have requests while the CPU is doing the Von Neumann algorithm
  - ↳ the I/O requests must be carried out with the Von Neumann algorithm
- sometimes I/O devices need to send or recieve large data — scanning, pictures, etc.
- each byte transfer takes two instructions
  - one read from I/O and one write to memory
  - 5 mil data → 10 mil instructions → 5 mil transfers
  - ↳ will face slowness in large data transfers



- DMA: A device that temporarily takes over the CPU to send data directly to and from the I/O devices and the main memory
  - ↳ the DMA sends requests to take control to the CPU ~ programmer determines when

- The DMA provides some information to the CPU when requesting to take over
  - ↳ it then waits for a response
  - ① volume of the data
  - ② start address in memory
  - ↳ if the program being run is high critical, the DMA will not be acknowledged

DMA:
- ① Burst transfer → transfering large data is prioritized
- ② Cycle stealing → when the CPU is not in control, one cycle run of the CPU, and one DMA
  - ↳ instruction cycle

- DMA's are not necessary → only speed up large data transfer processes

Interrupt

- When an I/O instruction is being executed, a lot of time is wasted waiting for the device or human response
  - ex. when typing on a keyboard

Mem[x] = keyscanf() ⇒
IN 30 — I/O device at port 30
[40] AC — moves data from AC to memory address 40

- Another program can be run while we wait — only practical in multitasking (multiple program) computers
  - ↳ original pc is stored somewhere to come back to

- Must have a way of returning back to the program once input is entered — keeps CPU busy unnecessarily
  - ① polling - continuously checks if the input was entered ex. while rest<>null
  - ② interrupt - runs a different program located somewhere else

- we have an interrupt flag connected to the I/O devices
  - ↳ also have an interrupt acknowledge which tells us to start the Interrupt Service routine
  - ↳ we check if its activated at the end of each Von Neumann cycle
- Interrupt Service Routine
  - ↳ the pc jumps to this address each time we interrupt
  - ↳ contains a program predefined by OS or user

- the CPU finds our ISR address in 3 ways
  1. predefined
  2. interrupt vector table — look up table
  3. vectored interrupt — the I/O device itself determines the ISR address after Interrupt Acknowledge