

# Computer Architecture

دکٹر سعید

مسیحیہ



## Grading System

midterm + pre midterm quizzes	7
final + post midterm quizzes	9
Assignments	3
Attendance	1

- \* quizzes will be held at the end of each month
- \* Assignment each week

h-zarandi@gmail.com  
h-zarandi@aut.ac.ir

## Class Content

### ① Memory

- ↳ memory hierarchy
- ↳ DRAM + SRAM
- ↳ CAM
- ↳ RAM / ROM
- ↳ Cache
  - > Direct Mapped Cache
  - > Set associative cache
  - > Fully associative cache

### ② Arithmetic Logical Unit (ALU) Design

- ↳ number representation
- ↳ Adders (unsigned)
  - > Ripple Carry Adder
  - > Carry Look-Ahead Adder
  - > Carry Select Adder (+ uniform k-bit + non-uniform)
- ↳ Subtractors (unsigned)
  - > Ripple subtractor
  - > Complement subtractor
- ↳ Multipliers
  - > shift add multiplier
  - > array multiplier
  - > Booth multiplier

### ↳ Divider

↳

### ③ Control Unit

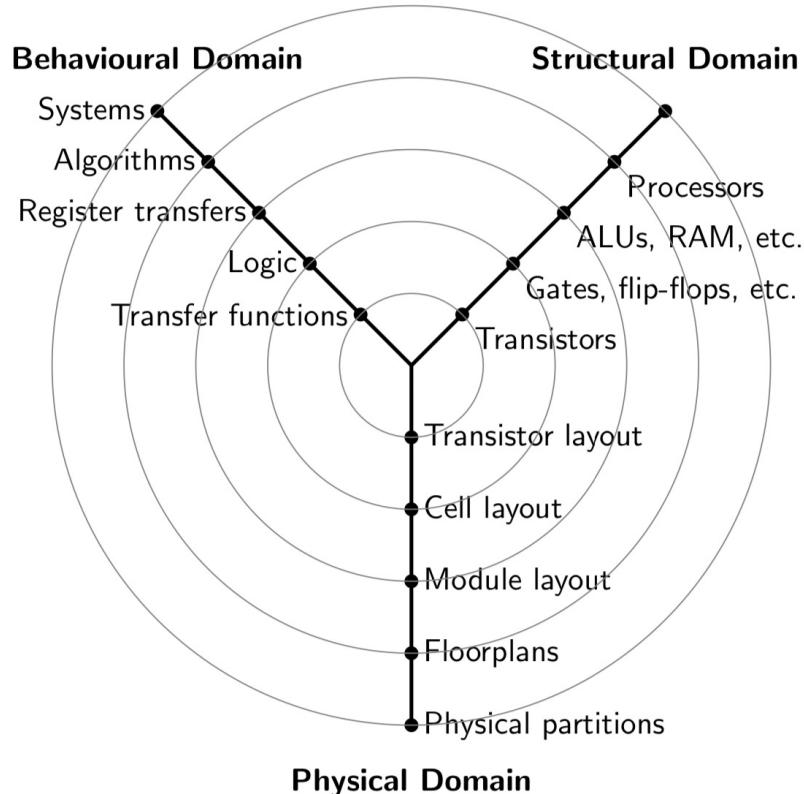
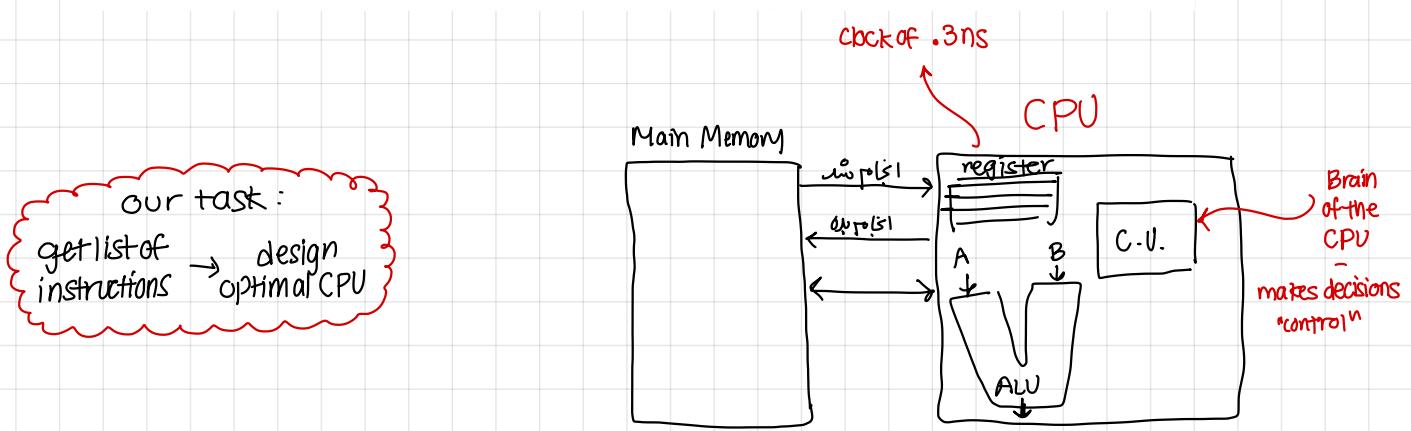
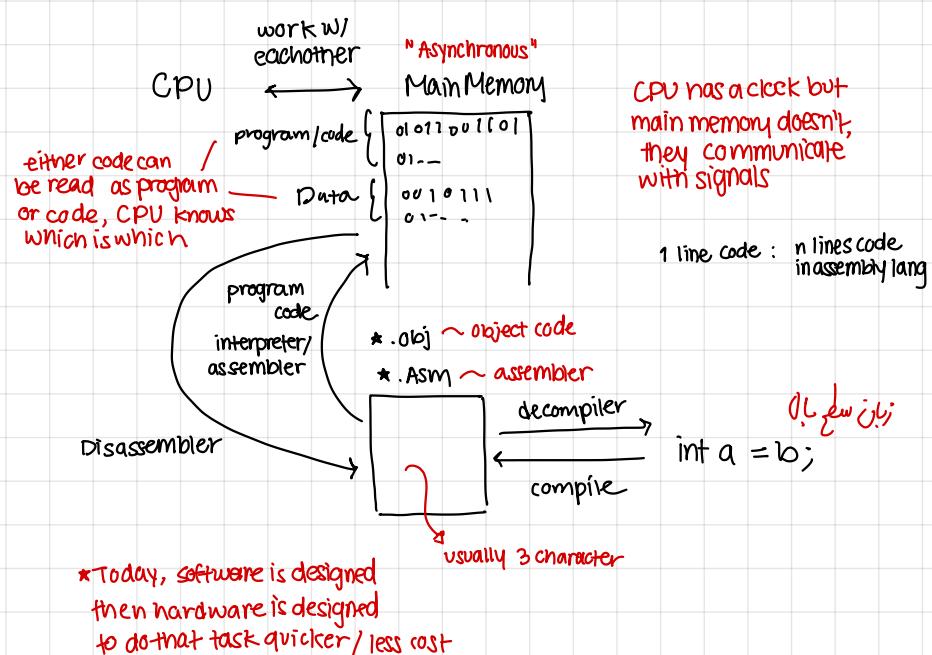
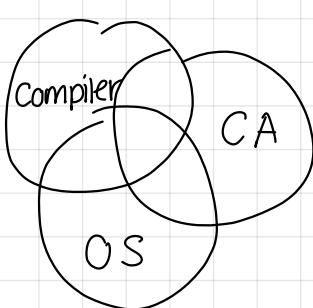


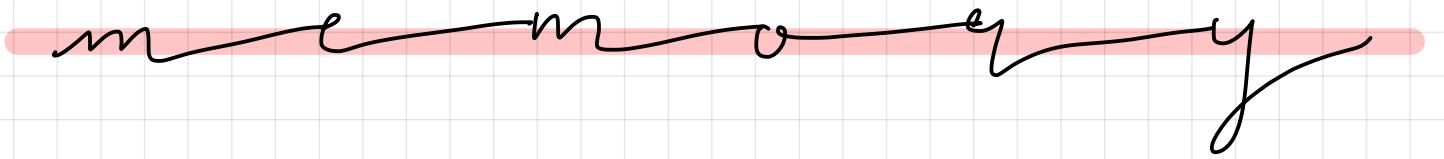
Figure 1: Gajski-Kuhn Y-chart



RAM is an implementation of Main Memory



T.



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

# \* جلسہ دوم - سینیٹ ۱۸ بھمن \*

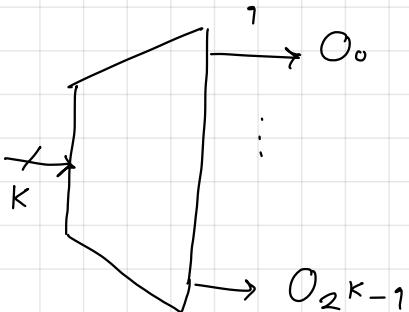
حیچ نہایی بلکہ load کیوں نہیں

٢ خنزير ملوك و ملائكة

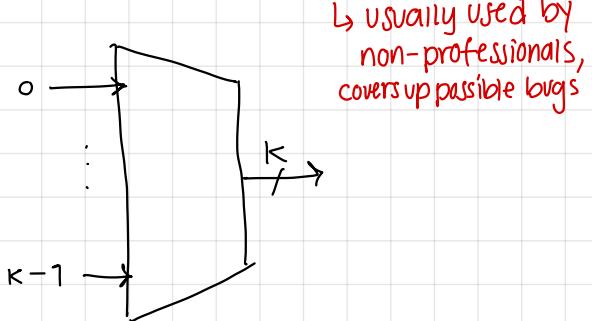
مِرَايَةٌ إِلَيْهِ بَاسِدٌ (جَنَاحِيلُ الْكَلَبِ)

"بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ سَمْعَتْ أَعْذَارًا"

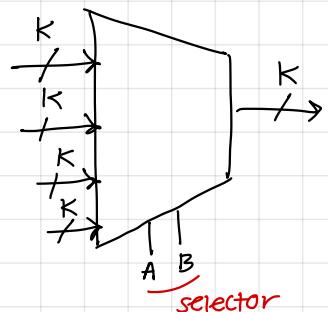
## Decoder



Encoder (priority encoder) → used frequently for decreasing wires (lessens chances of error)

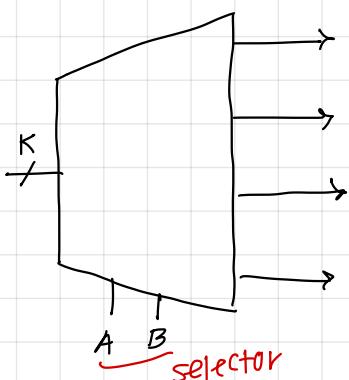


multiplexer "جلىء"



# DeMUX

هم خوش های خیلی  
کیم چ من سینه!



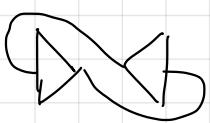
\* میں اسی FF کا لیٹچ (Latch) کہا جاتا ہے اور صورت میں اسے صرفاً مکانیکی طور پر کہا جاتا ہے۔

- Civil Engineering ↗

لے زمانی کے بڑی صورت میں اس کی وحکی نہ داد دخیرہ سعید در لatch اسی میں ہے

FF یا  $\rho_{CD}$  feedback نهایی

نحوی اول: ساده راهی منطقی  
نحوی دوم: کدام امدادی بجزئی از  
level را زیر نویسید و می‌توان  
درین خط روزگاری مختار است  
نه می‌دارد



۲) سمعت بالا + کران + تعالی صرف کریم + مسماۃ بیت

Implementation not  $\tilde{w}$   $\leftarrow$  SRAM

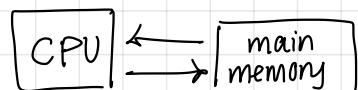
سیو، رام، رولوچر (زیارت)  $\leftarrow$  DRAM

للمعاشرات كمت兀 + ارزاں + صمعونىڭن بىلە + قاباچىت refresh

## Main Memory

# \* حلقة سوم - يلقيبي ٢٣ بهمن \*

\* The CPU works with the main memory



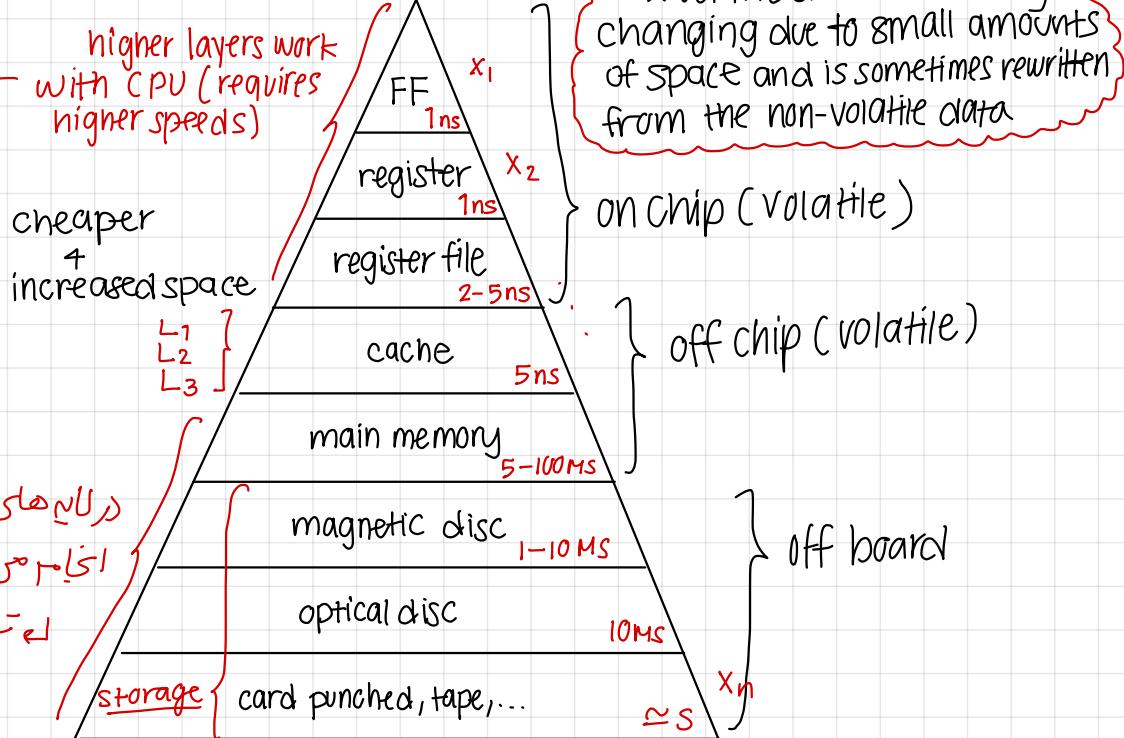
\* Booting a computer:

BIOS → Hard disk → CPU (find OS)

جایگزینی سفت افزاری ایجاد می‌شود

increased speed  
+ less delay  
+ طاقتی بسیار بیشتر

## memory hierarchy



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

on chip (volatile)

off chip (volatile)

off board

$$PPC = \frac{\text{speed}}{\text{cost}}$$

↳ a desktop computer requires high speed + low cost

$$h = \text{hit rate} = \frac{\text{تعداد موقوعات}}{\text{تعداد درخواستها}} = \frac{\text{امتحان پاسخ دهنده}}{\text{امتحان کردن و بعد در نظر گیری}} : \text{زمان} \rightarrow \text{کم کردن} \leftarrow \text{پاسخ تردیدی} : \text{زمان} \rightarrow \text{کم کردن} \leftarrow \text{قبو محاسبه سود} \leftarrow \text{تعیین ناچیز ایجاد}$$

Average Access Time

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

$$\begin{aligned} \text{I} &= h_1 t_1 + (1-h_1)(h_2 t_2 + (1-h_2)(h_3 t_3 + \dots)) \\ \text{II} &= h_1 t_1 + (1-h_1)(t_1 + h_2 t_2 + (1-h_2)(t_2 + \dots)) \\ \text{III} &= t_1 + (1-h_1)(t_2 + (1-h_2)(t_3 + \dots)) \end{aligned}$$

بادوچین متقارن → هردو ماضی درست و دقیقی به می‌دهند + زمان تلف شده بجزی کم کردن لایه قبل حساب شود

$$\begin{aligned} \text{تاریخ} &\leftarrow T_1 < T_2 < \dots < T_n \\ \text{قیمت هرست} &\leftarrow C_1 > C_2 > \dots > C_n \\ \text{امتحان} &\leftarrow h_1 < h_2 < \dots < h_n = 1 \end{aligned}$$

مثال: در یک کامپیوتر تا خرداد سال ۹۵ میلادی ۱۰ ns است. همان‌جا از تاریخ هفته‌مان با تاخیر ۱۰ ns و نیم موفقیت ۹۵٪ است. وقتی زمان رسانی می‌شود...

$$\bar{T} = t_1 = 1000 \text{ ns}$$

$$.06 = \frac{60}{1000} = \frac{\text{تاریخ}}{\text{تاریخ}} = \frac{\text{سرع}}{\text{سرع}}$$

$$\text{speed up} \leftarrow S = \frac{t_{\text{old}}}{t_{\text{new}}} = \frac{t_{\text{old}}}{\alpha \frac{t_{\text{old}}}{K} + (1-\alpha) \bar{T}} = \frac{1}{\alpha + (1-\alpha)}$$

(الف) متوسط زمان رسانی؟

$$\text{I} = h_1 t_1 + (1-h_1)(h_2 t_2)$$

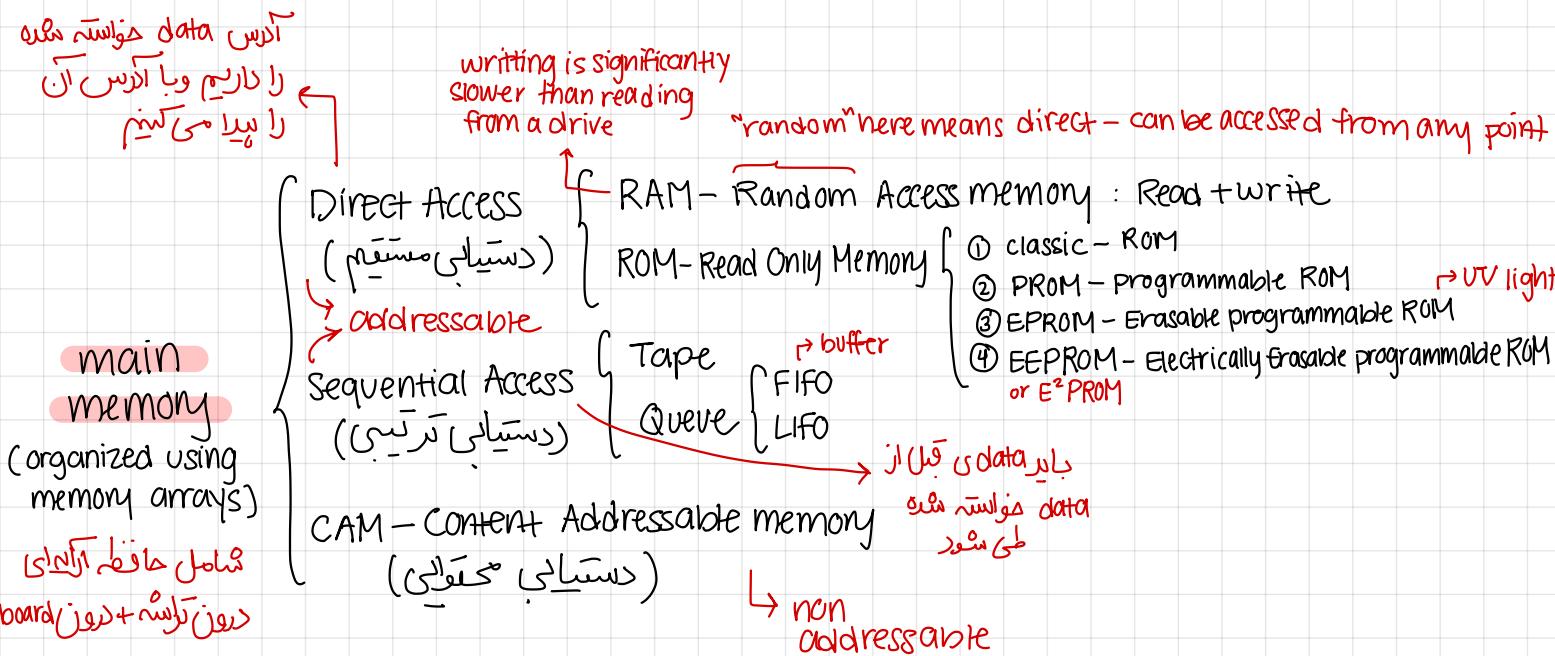
$$= (.95)(10) + (.05)((1)(1000)) = 59.5 \text{ ns}$$

$$\text{II} = h_1 t_1 + (1-h_1)(t_1 + h_2 t_2)$$

$$= (.95)(10) + (.05)(10 + (1)(1000)) = 60 \text{ ns}$$

سرعی؟  
تاریخ قبلی  
 $\frac{1}{\alpha + (1-\alpha)} > 1$   
تاریخ بعدی  
 $\frac{1}{\alpha + (1-\alpha)} < 1$

# \* جملہ حفاظتی - حفاظتی \*

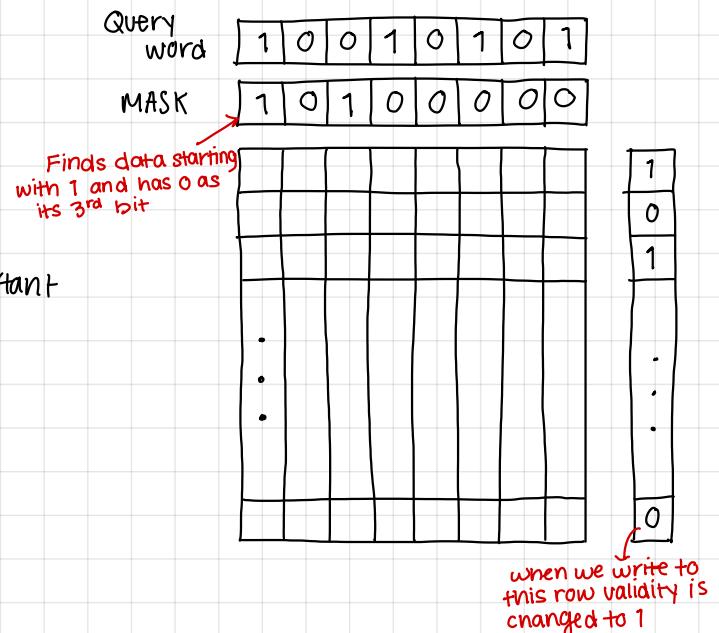


## CAM - Content Addressable Memory

- 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:

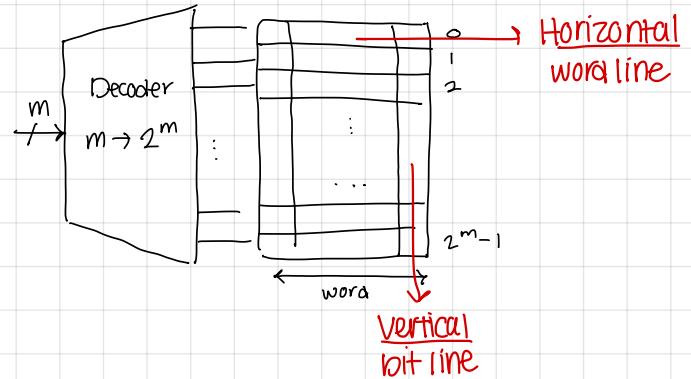
- ① Row by Row
  - ↳ involves checking all data row by row
  - ↳ very slow
- ② Parallel
  - ↳ checks parallel columns all at once
  - ↳ requires an XNOR gate for each bit
  - very expensive
  - بے انتہا میتوں
  - ↳ very quick search



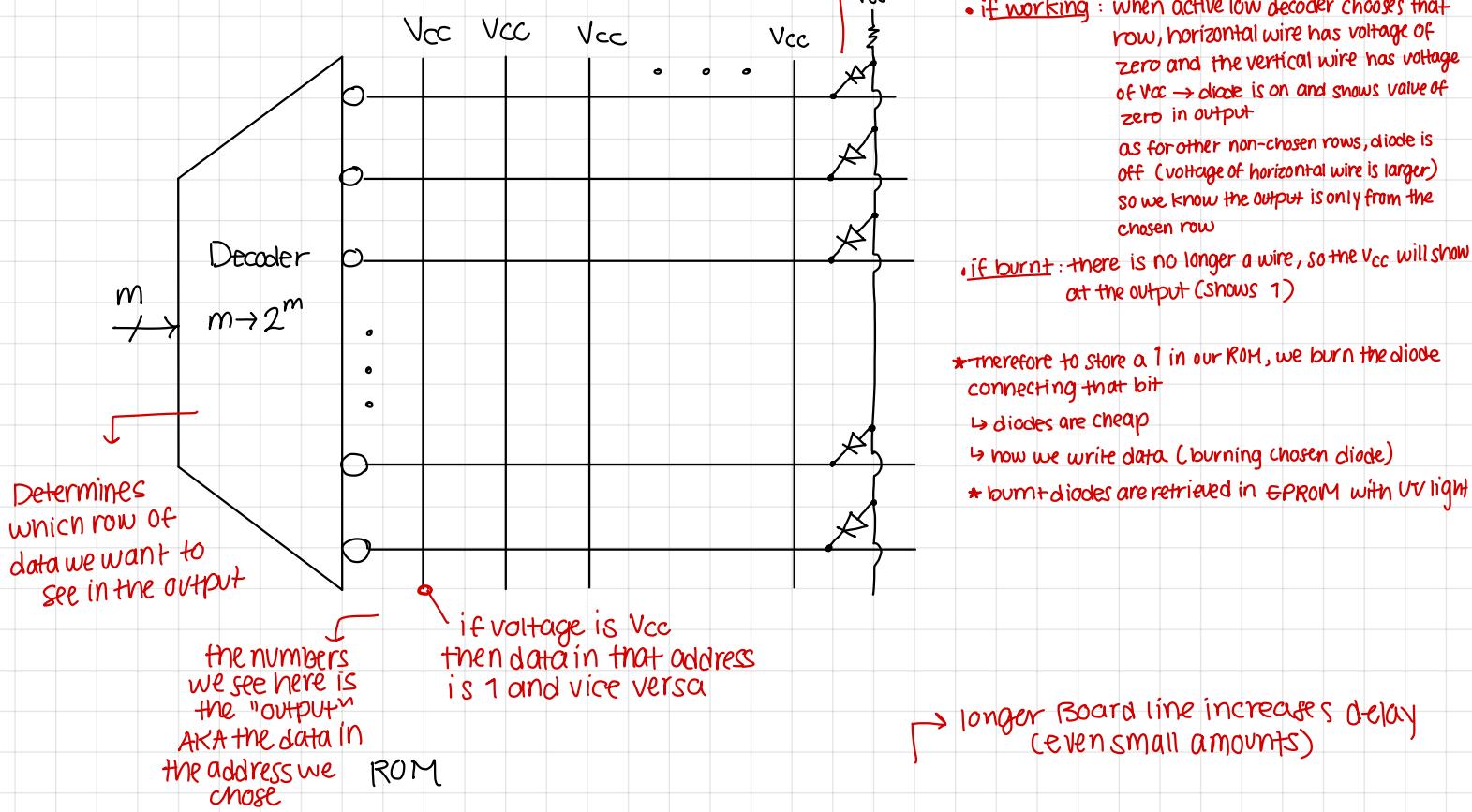
- when searching there are 3 possible scenarios / results
  - ① miss - nothing similar is found
  - ② single-hit - exactly one similar data
  - ③ multi-hit - can be one of the found options
- conclusion → CAM is too expensive + gets too hot with current technology
  - ↳ CAM can only be used for on board 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 voluminous part of the RAM/ROM
    - is the part preventing further advancements
- word: each row in the memory
  - is the smallest unit of memory
    - "مقدار کمی اطلاع"
- word line: holds data
- bit line: gives data



RAM / ROM :

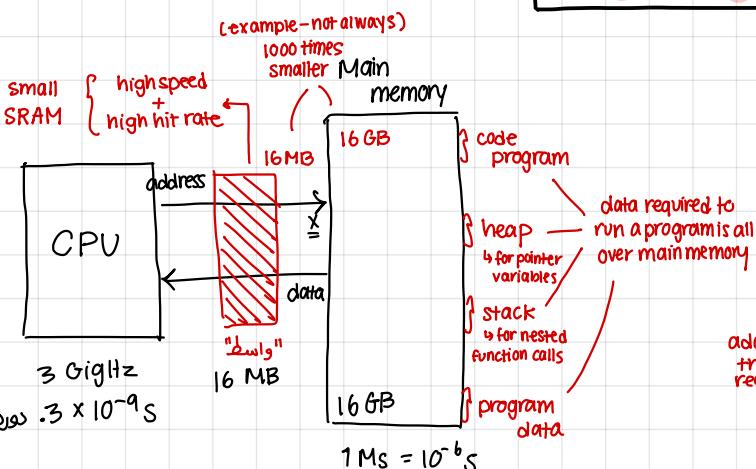


For example, let's say an input  $m = 00\dots01$  is given to the decoder, activating the final row of the decoder. The activated horizontal wire has voltage of 0. In the last column the diode is intact, so the last horizontal line will show a voltage of 0. For the other rows, the wires have  $V_{CC}$  voltage (either the diode is off or burnt, both not changing the result)

# حلقة بيجم - حلقة بيجم ٥٣ - بحث عن حافظة ذاكرة

نیاز ب پیش بینی خوب

## Cache (حافظة ذاكرة)



- 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:  
20, 21, 22, 23, 50, 80, 30, 22, 23, 20, 30, 14, 1400, 50, ...  
often consecutive often repetitive

↳ spatial locality (محظوظ مخاطر)

- 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

چرگاه داده ای از M.M. محسوسیاتی های آن را نیز در آورید

↳ temporal locality (محظوظ زمانی)

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

هر داده ای که نه در M.M. باشد این را نه در Cache ندارد اینها مسأله ای هستند

جستجوی مکرر می شود  
finding the organized item takes more time

\* 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

\* 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 حافظه ذاكرة cache 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)

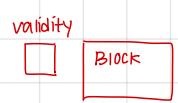
> if block is too big, we lose temporal locality (cache becomes too full and used data is overwritten)

↳ Block size is a multiple of 2

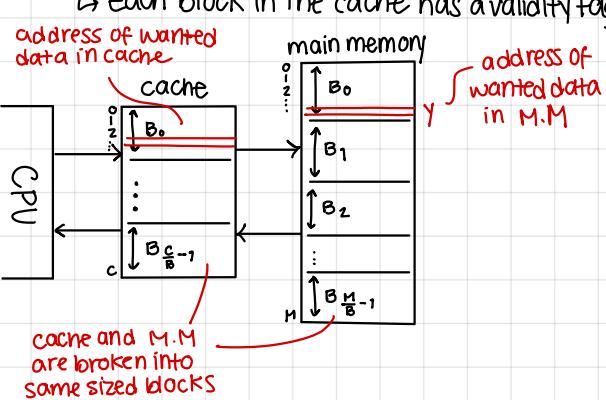
↳  $B = 2^b$

↳ no consistency (not always true for ex. 8 before and after requested data)

↳ each block in the cache has a validity tag (used mostly when booting the computer)



صفر و ۱ و ممکن است ممکن است ممکن است



$$M = \text{size of Main Memory} = 2^m$$

$$C = \text{size of Cache} = 2^c$$

$$B = \text{size of blocks} = 2^b$$

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

• Direct Mapped Cache (DMC)

خواهد بود  
 $x_B = x/B$  (x برابر با مقدار مسأله است) مسأله کوچکتر است

$$Y_B = X_B \bmod C/B$$

ex.

$$x = 010110|0110$$

number block      6th item in the block

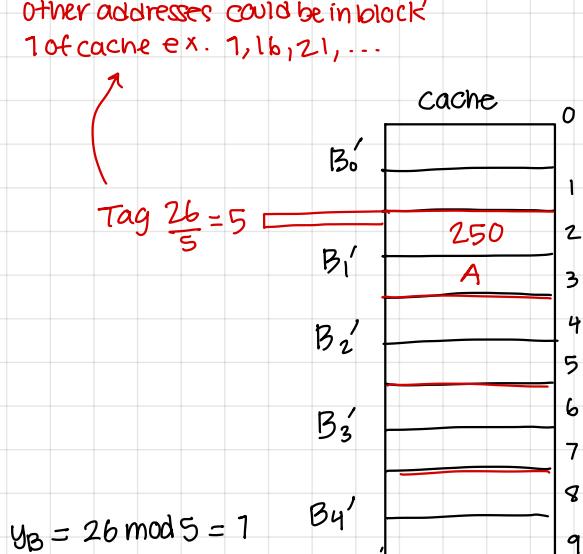
مکانیسم در سیستم مسأله کوچکتر است

# \* view ٢ نسخه - فوئی خواه \*

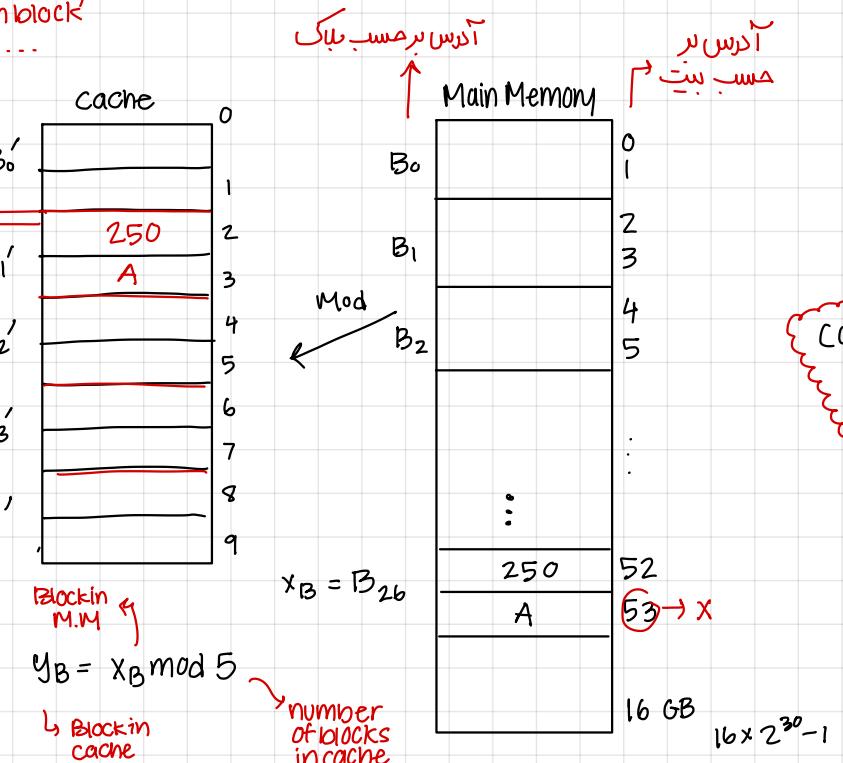
Example : 16 GB main memory with 10 bit cache with block size of  $2 = B = 2^1$ :

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



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

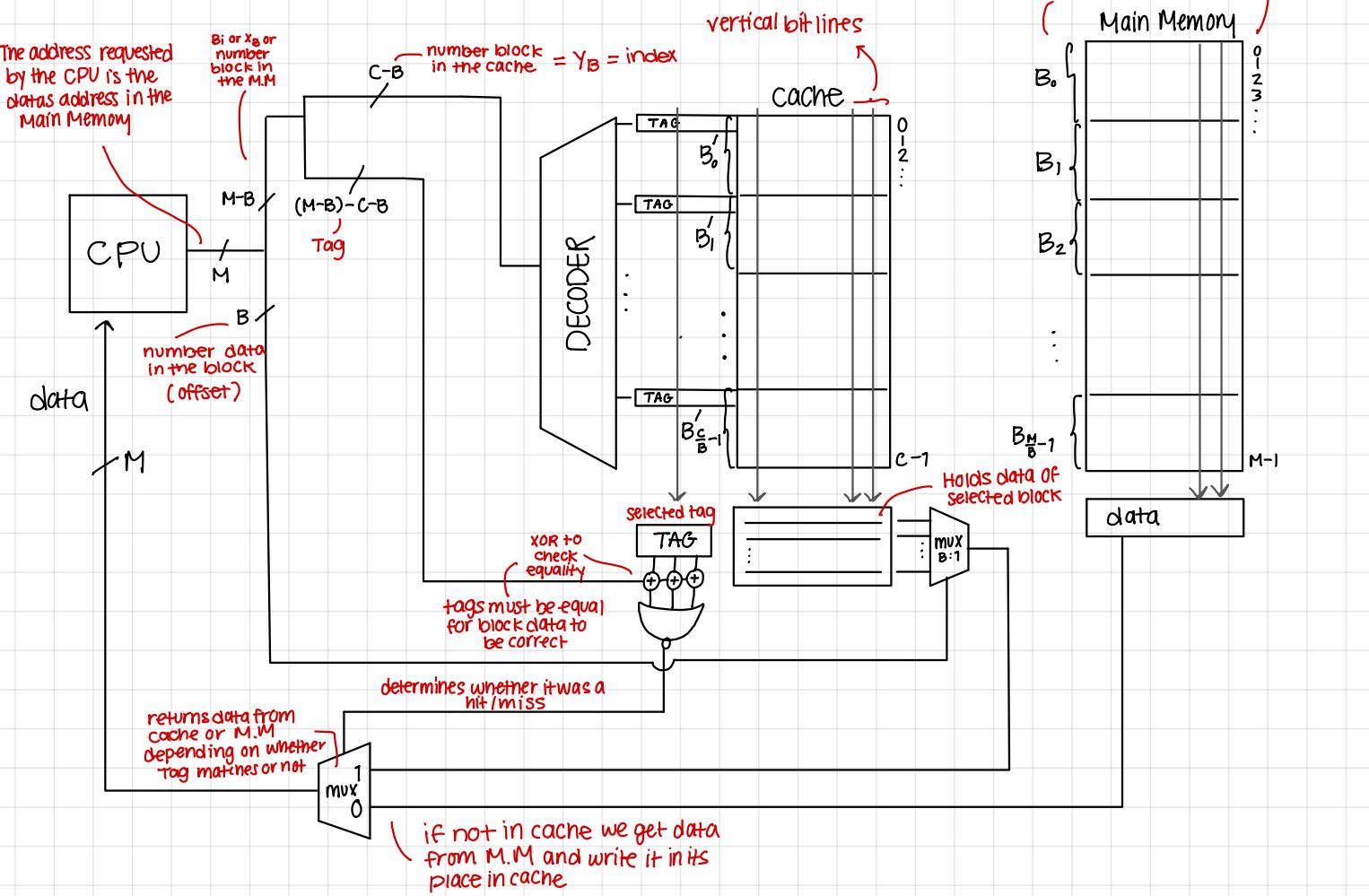


## Direct Mapped Cache (DMC)

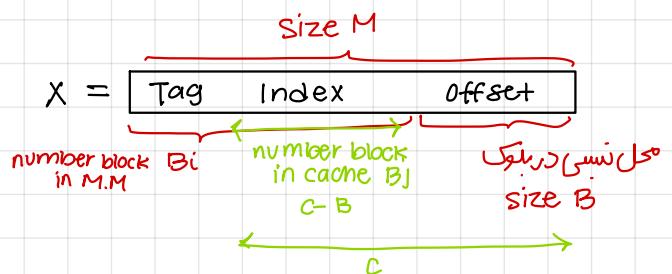
- The CPU itself works with bit, main memory addresses, not cache
  - ↳ The CPU is not aware of the existence of the cache
  - ↳ all processes done between the M.M and cache are independent from the CPU
- We divide the Main Memory into blocks
  - ↳ Block: segments of data with set sizes used to interchange 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:
  $y_B = x_B \bmod (\frac{C}{B})$  or  $B_j' = B_i \bmod (\frac{C}{B})$  number of blocks in cache
    - ↳ with this mapping function there will definitely 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 separating the bits:
    - ↳ offset: relative index of requested data in block
      - > has size of  $B \rightarrow$  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
  - A set associative cache where  $k=1$

پس آگر همی مخادر رفته بازی از 2  
 باشند بعنوان نیاز به تقسیم؟  
 باقی مانده و خارج قسمت (ست میانی)  
 $\frac{C}{B} = \frac{2^C}{2^B} = 2^{C-B}$ ,  $B = 2^B$

لهم (x\_{n-1} x\_n \dots | x\_0) r^{k+1} \rightarrow (x\_{n-1} x\_n \dots | x\_0) r^k  
 بخواهیم داشت  $r^k = 2^B$   
 تا  $n$  تا از روی سایر زدن ارقام قبل خود است.



براساس آدرس تضمین می شود data که در بینر (بی ربط)

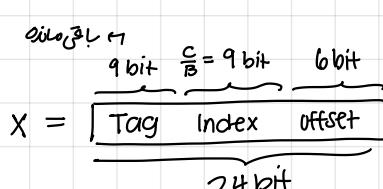


مثال: در یک حافظه دهنگ با حافظه 32 KB و یک حافظه 16 MB می باشد

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

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

$$B = 2^6$$

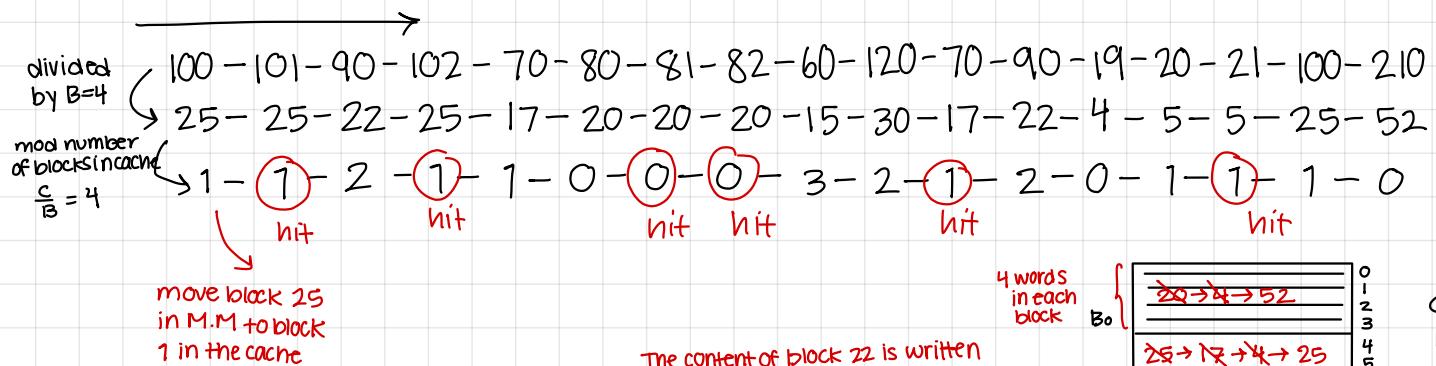


- الف) میان offset چه جای خواهد؟ سایز B  
 ب) اندازه میان Tag چه بحث است؟ سایز  $\frac{B}{B}$   
 گ) اندازه میان index چه بحث است؟ سایز  $\frac{c}{B}$

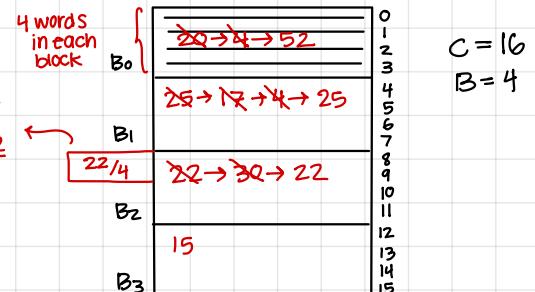
# \* اسکن فریم - حافظه محل \*

## Example 1 :

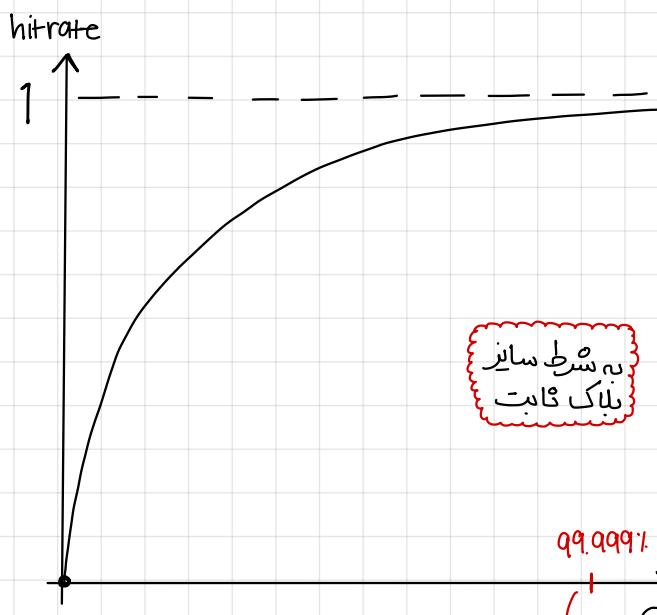
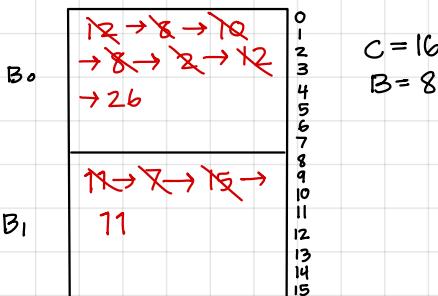
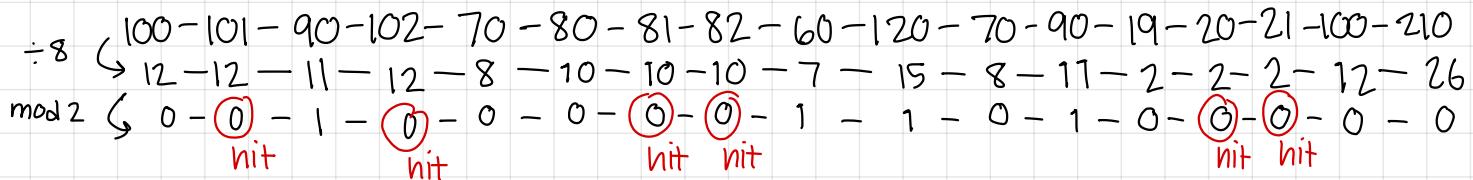
آدرس های تولیدی توسط CPU به حافظه اصلی



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

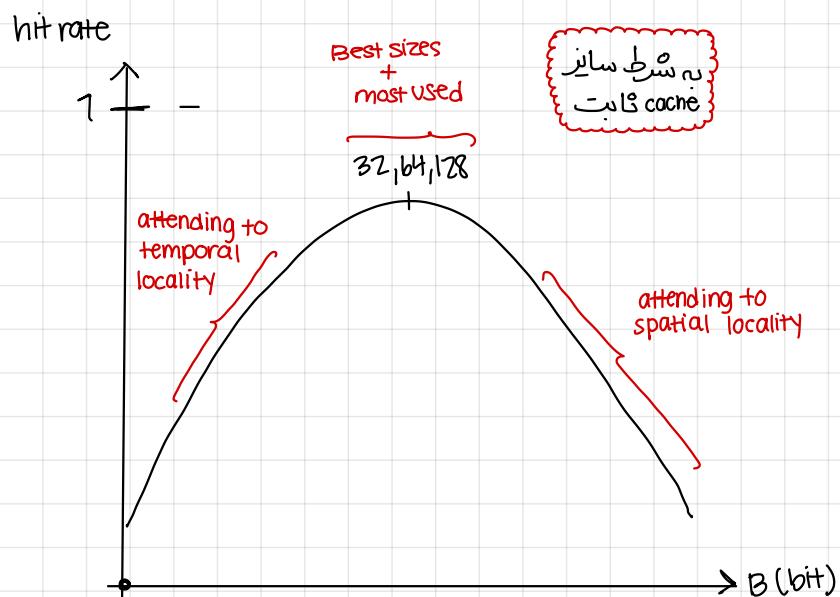


## Example 2 :



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 →
- ② Set Associative Cache < two-way k-way
- ③ Fully Associative 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

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 hitrate 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 مفهوم مجموعات

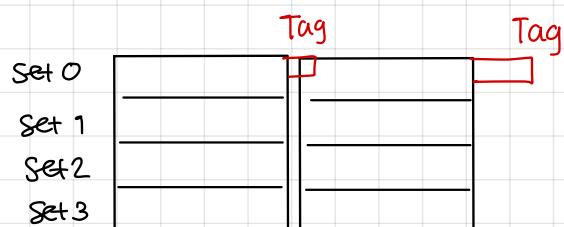
- In DMC we only had one block of space for the same remainders, so we solve that problem by adding another column for each remainder
  - ↳ there is no priority between the columns
  - ↳ when a new block must be stored in a row, it replaces the oldest block in that row
- In order to check if the block we want is already in the cache, we must check all  $k$  blocks in the row with that block's 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 {  
 $k=2 \quad 9.6\%$   
 $k=4 \quad 97\%$
  - expensive + heat + breaks CPU {  
 $k=8 \quad 98.1\%$   
 $k=16 \quad 99\%$   
 $k=32 \quad 99.9\%$   
⋮  
 $k=C/B$

### Fully Associative Cache مفهوم 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

Two-way set Associative cache:

Have different locations  
↳ need different tags



\* انماcache همان را نمی کرد اور دینم و سرت

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

$$S_j = B_i \bmod \left(\frac{C}{B}\right)$$

$\downarrow$   
 $C/B$

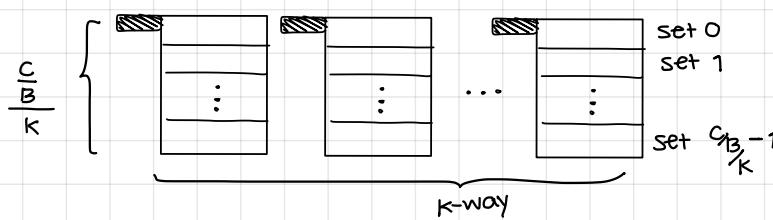
↳ number of sets

خارج فرمات بزرگ تری دارد  
کد بست بسترا  
 $\leftarrow K=1$   
کد بست بست  
 $\leftarrow K=2$   
دو بست بست  
 $\leftarrow K=3$   
سه بست بست

# \* view V nimmt - nimmt mehr \*

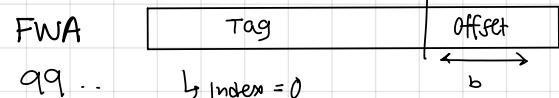
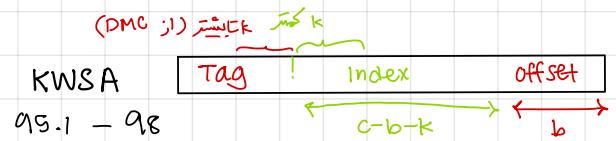
## K-way (KWSA)

set Associative Cache:



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

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



- 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

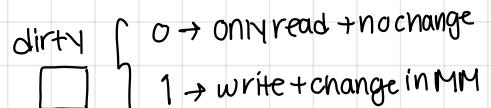
- slows down CPU

② write back → changes are done in cache, and whenever block is removed from cache it is written in M.M

+ keeps speed

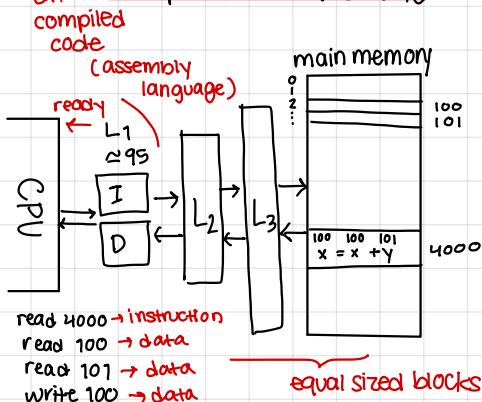
- M.M is not updated — could cause problems of data loss with sudden disconnections

↳ requires a tag determining whether a block needs to be rewritten or not



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

## ex. • Multiple levels of Cache

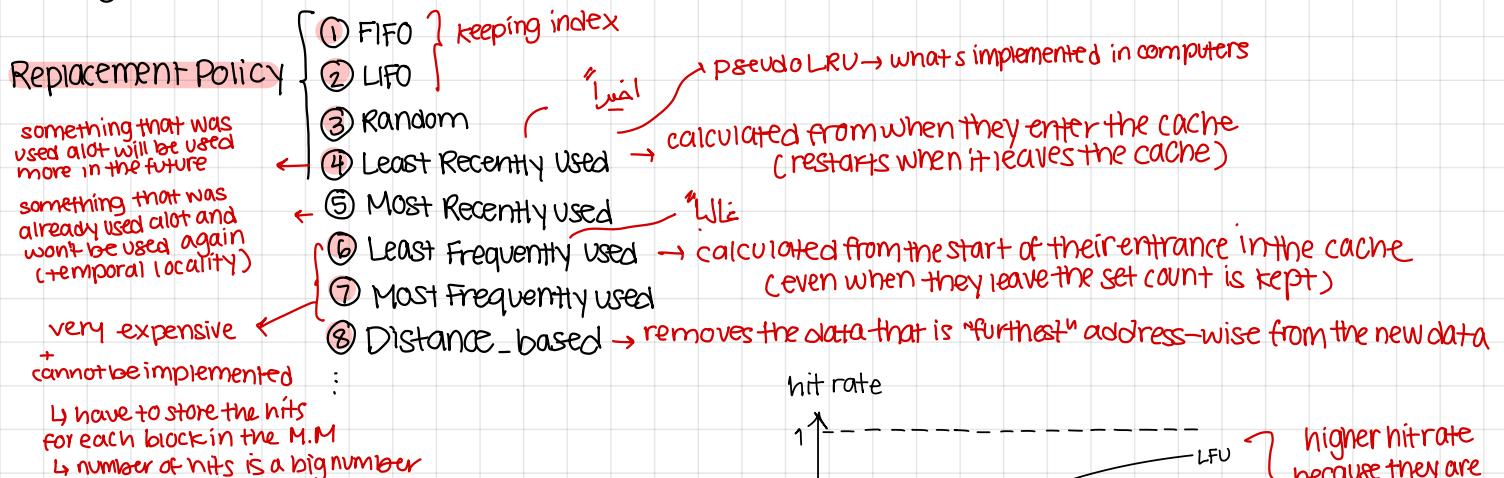


- Addresses missed from first cache still have some temporal and spatial locality
- We add 3 levels of cache:
  - ↳ As we get further from the CPU, the cache gets slower and larger
- Splitting the cache into 2 segments (Instructions and Data) causes more spatial locality
  - ↳ not always beneficial,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

↳ when data was in none of the cache's 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

# \* view 9 min - results \*

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



- 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

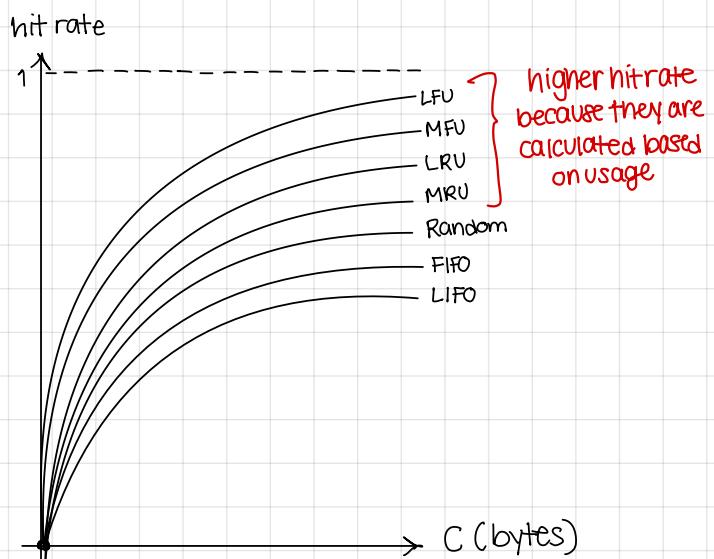
↳ LFU / MFU

> 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

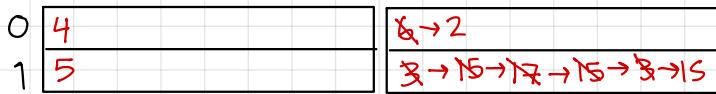
a,a,a,b,c,d,e

مقدمة إلى الدرس الرابع (الذاكرة الصلبة) : الدرس الرابع

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

### 2-way associative (LIFO)

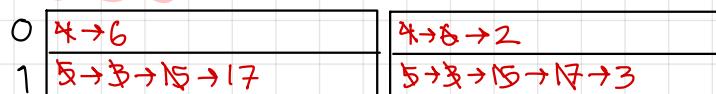
hit rate :  $\frac{7}{18}$



mod 4,2  
 5, 4, 3, 5, 6, 4, 3, 3, 6, 15, 17, 6, 2, 17, 17, 15, 3, 15  
 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1

### 2-way associative (FIFO)

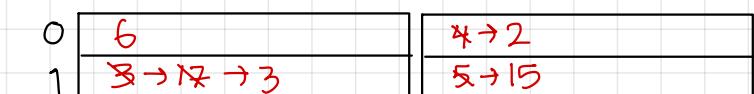
hit rate :  $\frac{9}{18}$



mod 4,2  
 5, 4, 3, 5, 6, 4, 3, 3, 6, 15, 17, 6, 2, 17, 17, 15, 3, 15  
 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1

### 2-way associative (LRU)

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



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

mod 4,2  
 5, 4, 3, 5, 6, 4, 3, 3, 6, 15, 17, 6, 2, 17, 17, 15, 3, 15  
 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1

### 2-way associative (LFU)

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



\* when equal count remove oldest

count: (remove smallest)

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

mod 4,2  
 5, 4, 3, 5, 6, 4, 3, 3, 6, 15, 17, 6, 2, 17, 17, 15, 3, 15  
 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1

### Fully associative (CMRU)

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

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

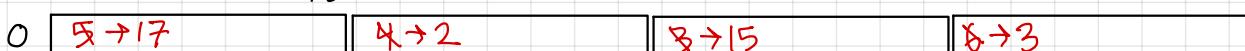


5-4-3-5-6-4-3-6-15-17-2-17-15-3-15

### 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



(remove largest)  
count:

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

\* when equal count remove oldest

2.

## A L U designing

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

★ view if min - max \*

## Data Representation

### • Integer numbers

↳ unsigned ( $2^n - 1$  possible (+) numbers)

+ IO with human  
- harder calculation

↳ signed we lose 100000 because of -0

> Sign magnitude ( $2^{(n-1)} - 1$  possible (+-) numbers)

★  $[-2^{(n-1)} - 1, 2^{(n-1)} - 1]$

> 1's complement

★ complement all bits in the number

★  $[-2^{(n-1)} - 1, 2^{(n-1)} - 1]$

> 2's complement → 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 +/-)

★  $[-2^{(n-1)}, 2^{(n-1)} - 1]$

> excessry 2's complement bias 1: here, excess is  $2^{n-1}$  → add  $2^{n-1}$  to all numbers

used for  
showing  
floating  
point

↳ ex.  $0000 \leftrightarrow 128$

★ 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

★  $[-2^{(n-1)}, 2^{(n-1)} - 1]$

games → rendering: uses lots of mathematical operations

(lighting, shading, ...)

↳ requires more operands

(desktop)  
more assembly code  
slower + cheaper

(not all operands are useful today)  
expensive  
+ more capable

↓  
less operands or more?

### • Decimal numbers

↳ fixed point decimal → more error + cheaper

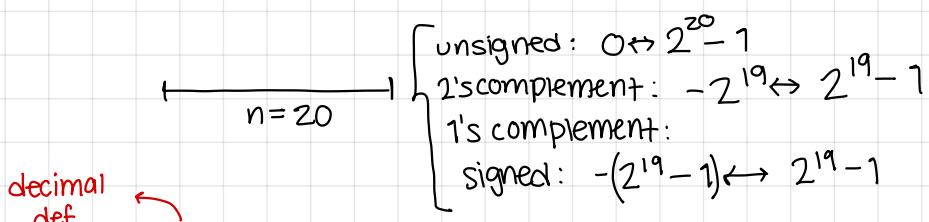
↳ floating point decimal → less error + expensive

• We choose which data representation we want to use based on client needs

↳ 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

\* change to unsigned  
when we can



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

★ 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

must  
be able  
to design  
ALU  
for all

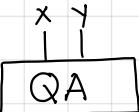
→ add, subtract, multiplication, division

## Adders

augend  
addend  
 $A + B$

### Unsigned Adders

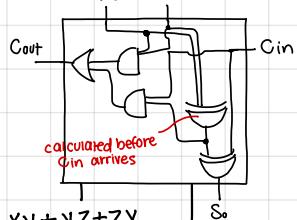
- Quarter adder  $\rightarrow$  1 bit adder w/o carry
- Half adder  $\rightarrow$  1 bit adder w/ carry
- Full adder  $\rightarrow$  1 bit adder w/ carry in



$S = X \oplus Y$   
delay of one gate  
delay(sum) = d  
cost = 1g



carry =  $X \cdot Y$   
delay(sum) = d  
delay(carry) = d  
cost = 2g

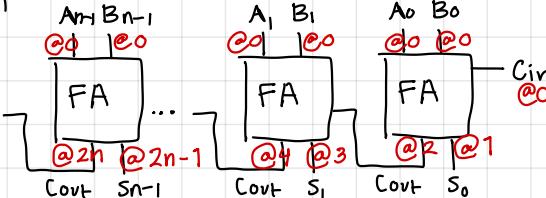


carry =  $XY + YZ + ZX$   
sum =  $X \oplus Y \oplus Z$   
delay(sum) = 1d  
(carry) = 2d  
cost = 5g

### Ripple Adder

- The carry-out of the previous FA is the carry-in of the next

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  $\Delta = 2d$ , using the critical path to determine this

### Carry Look-Ahead Adder

\* which has a better ppc?

- carry generate : determines whether  $A=B=1$  (creates carry or not)  $G_i = A_i \cdot B_i$

- carry propagate : determines whether the carry moves to the next

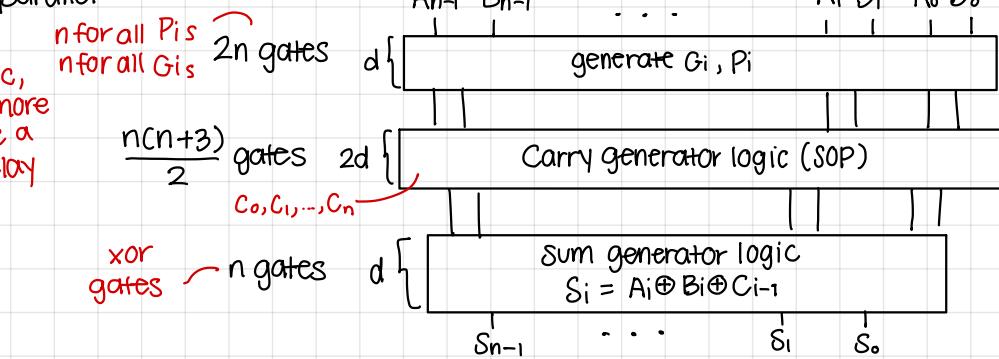
- and's and or's can be done parallel

delay(sum) :  $4d$   
delay(carry) :  $3d$   
cost :  $3n + \frac{n(n+3)}{2}$  g

n for all Pis  
n for all Gis  
2n gates

$\frac{n(n+3)}{2}$  gates  
2d  
 $C_0, C_1, \dots, C_n$

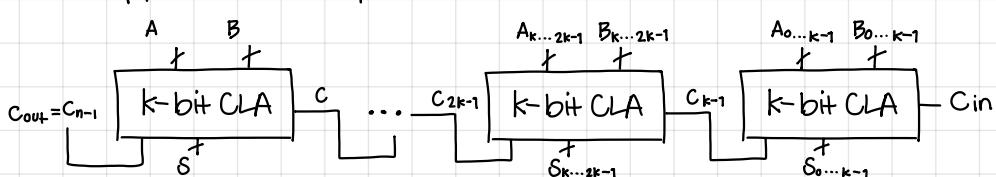
xor gates  
n gates



- 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) :  $2\left(\frac{n}{k}\right)+1$  d  
the first CLA still takes 3d  
delay(carry) :  $2\left(\frac{n}{k}\right)+2d$   
cost :  $\frac{n}{k} \left(3k + \frac{k(k+3)}{2}\right)$  each CLA

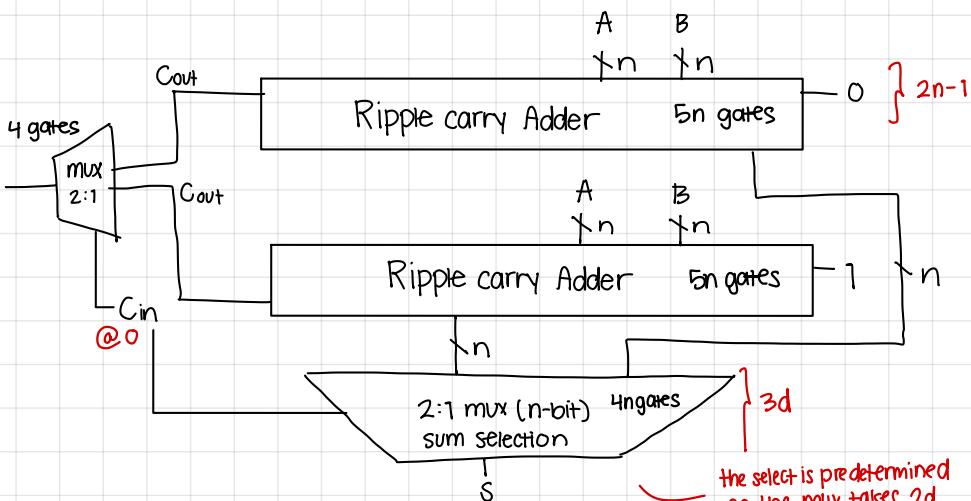
the multiple is actually 2 because the Pi and Ci can be calculated before receiving the Cin of the k-bit CLA



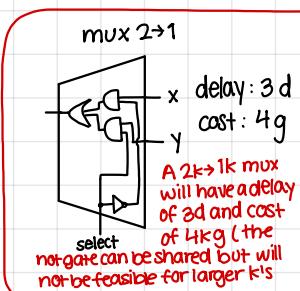
### Carry Select Adder

- calculate sum with two scenarios (carry = 1 or 0)

then use Cin as input in mux of these two answers



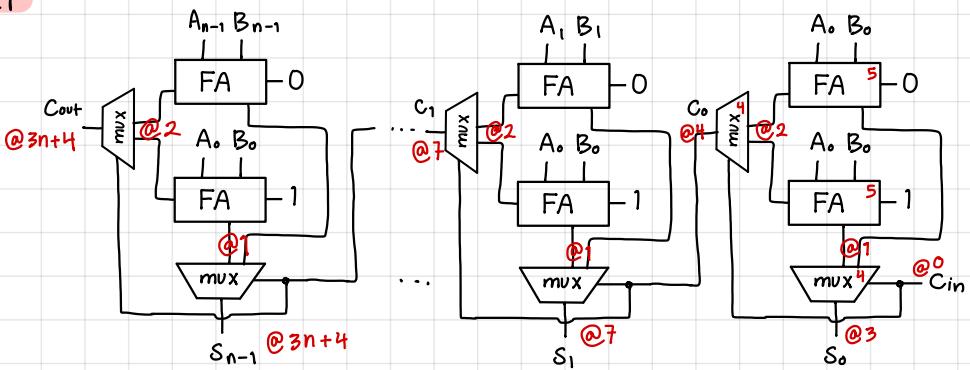
delay(sum) =  $2n+1$  d  
delay(carry) =  $2n+2$  d  
cost =  $14n+4$  g



# ★ view 21 میں کے جوابات کا سوال

## • Ripple (1-bit Carry Select) Adder

$$\begin{aligned} \text{delay (sum)} &: 3n+4 \text{ d} \\ \text{delay (carry)} &: 3n+4 \text{ d} \\ \text{cost} &: 18n \text{ g} \end{aligned}$$



## • k-bit Uniform Carry Select Adder

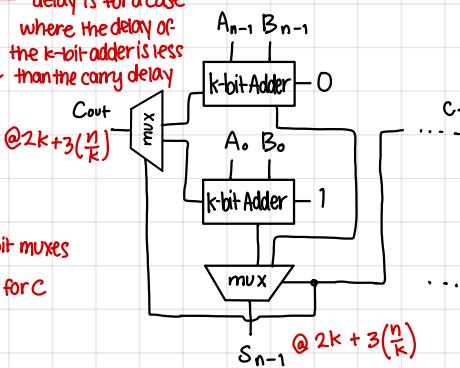
carry delay of used adder (ripple adder)

$$\begin{aligned} \text{delay (sum)} &: 2k + 3\left(\frac{n-1}{k}\right) d \\ \text{delay (carry)} &: 2k + 3\left(\frac{n}{k}\right) d \\ \text{cost} &: 2\left(\frac{n}{k}\right) 5 + 4\left(\frac{n}{k} + n\right) g \end{aligned}$$

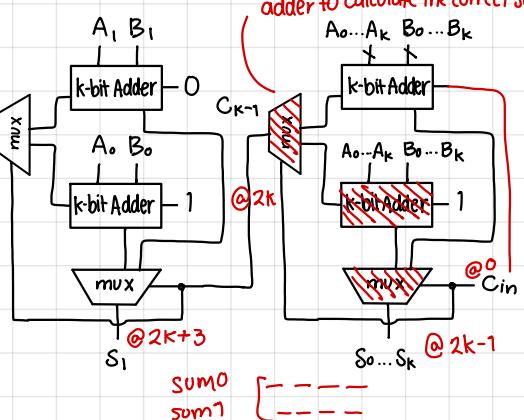
more precise  
cost of used adder (ripple adder)

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

delay is for a case where the delay of the k-bit adder is less than the carry delay



The crossed out pieces are unnecessary since we have Cin and can use one adder to calculate the correct sum



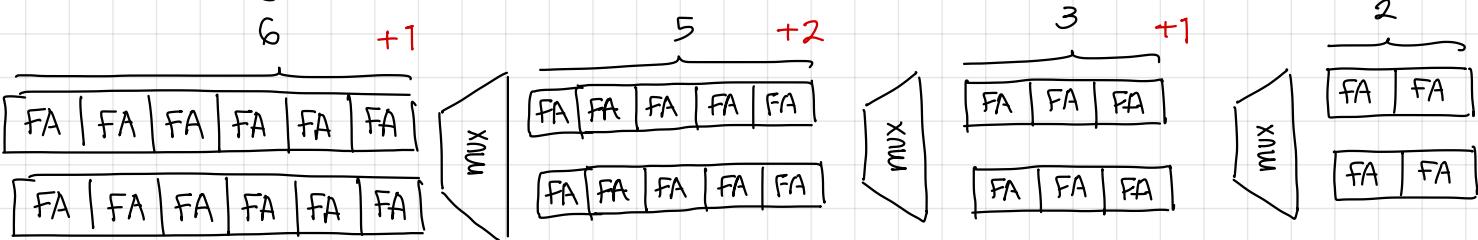
## • Non-uniform Carry Select Adder

↳ A full adder takes 2d to calculate the sum

↳ A mux takes 3d to give us the output (from carry)

↳ Every mux gives us time for 1.5 more FA's → we can add them after each level

the extra delay lets us add a FA to the next k-bit adder and two FA's in the next 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)

$$\text{delay (sum/carry)} : 2k + 3\left(\frac{n}{k}\right) d \sim \text{same as uniform}$$

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: ?

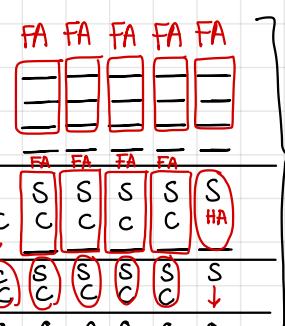
cost: ?

For m numbers, we require  $\frac{m}{3}$  rows of n FA's

↳ each row of FA produces 2 rows of sum and carry

→ we will have  $\frac{2}{3}m$  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  
HAs ↓ C C S C S C S C S C S ↓  
C C C C S C S C S C S C S ↓  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

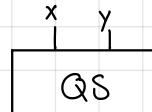
"جس کو کی سین اخراج عدد"

# Subtractors

minuend  $\rightarrow$  A - B  $\rightarrow$  subtrahend

## • Unsigned Subtractor

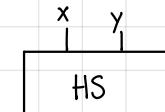
- ↳ Quarter subtractor  $\rightarrow$  1 bit subtractor w/ no borrow
- ↳ Half subtractor  $\rightarrow$  1 bit subtractor w/ borrow
- ↳ Full subtractor  $\rightarrow$  1 bit subtractor w/ borrow and borrow\_in



$$\text{sub} = x \oplus y$$

$$\text{delay (sub)} = d$$

$$\text{cost} = 1g$$

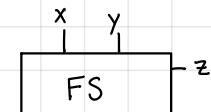


$$B = x'y$$

$$\text{delay (sub)} = d$$

$$\text{delay (borrow)} = 2d$$

$$\text{cost} = 3g$$



$$B = x'(y+z) + xyz$$

$$S = x \oplus y \oplus z$$

$$\text{delay (sub)} = d$$

$$\text{delay (borrow)} = 3d$$

$$\text{cost} = 6g$$

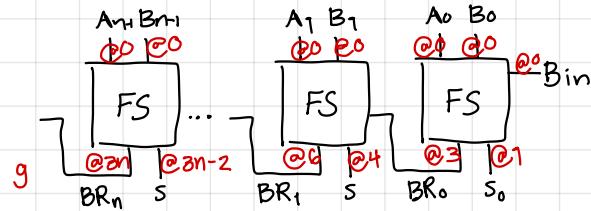
## • Ripple Subtractor

- ↳ the borrow\_in of the previous FS is the Borrow\_out of the next

$$\text{delay (sub)} : (3n-2)d$$

$$\text{delay (borrow)} : 3n d$$

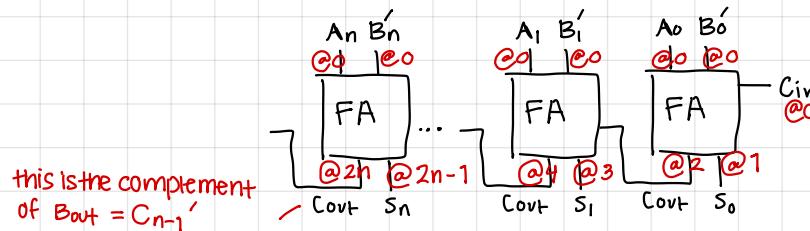
$$\text{cost} : 6ng$$



provable

## • Complement Subtractor

- ↳ the difference of two numbers can be calculated by:  $A - B = 2^n + A - B = A + B'$



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 32bit  $\leftrightarrow$  64bit  
(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

$$> B_{out} = 1$$

- ↳ In complement subtractors, if we DONT have a carry, we have overflow

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

doesn't need borrow  
reverse relationship with borrow

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

has carry  
↓  
no borrow

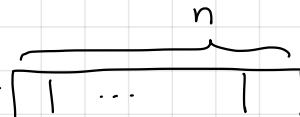
# \* حساب سیگنال - فرودین ۱۸

## Flags

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

} overflow is different

Register



- Overflow in signed 2's complement addition  $C_{out}$  doesn't mean anything

↳ there has been an overflow if:

- The addition of two positive numbers resulted in a negative or two negative numbers result in a positive
- The last two carrys are NOT equal ( $C_{n-1} \neq C_{n-2}$ )

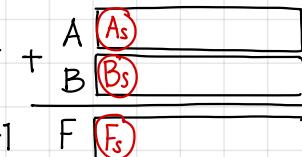
بيانات معمولة  
معينة

- > carry not created in one before the last but created in last
- > carry created in one before the last but not created in last

↳ overflow in subtraction can be found the same way with  $A+B+1$

$$\begin{cases} ++ = - \\ -- = + \end{cases}$$

$$\begin{cases} A'_s B'_s F_s + A_s B_s F_s' = \text{True} \\ C_{n-1} \oplus C_{n-2} = \text{True} \end{cases}$$



- Overflow solutions: up to the user to expand datasize

## Flags

↳ bits that show the status of ALU calculations

2's complement: overflow logic connected to it

① O - overflow: whether calculated data is correct or not

② S - sign: holds the  $F_s$  (Left most bit) — only for signed numbers

③ Z - zero: whether calculation result is zero or not



④ C - carry: whether calculation has carry out or not

unsigned overflow logic connected to it

⑤ P - parity: activated when data is lost

> reasons could be smaller transistors, sudden hits to the computer, UV sunlight rays, ...

problem is handled for satellites

> a single bit is flipped

> even parity: adding an extra 1 when needed that keeps the number of 1's even

P 8-bit data

xor > odd parity: adding an extra 1 when needed that keeps the number of 1's odd

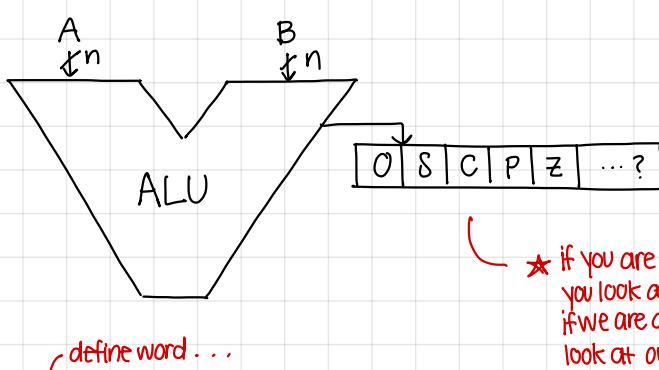
already has odd 1's → 0  
parity is 1 to make odd → 1  
11101010  
11010111

ex. in odd parity, if the data has an odd number of 1's, parity is 1

> 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 ?

SEGMENT CODE

jumpif  $O=1$  ADD a,b,c }  $a=b+c;$   
                  —→ JO finish      if(overflow) exit(0);

jumpif  $C=1$       —→ JC finish      or

If  $a>b$  then  
else

compiler →

for unsigned

JNC ↑ SUB A,B →  $A=A-B$   
J8 myElse → previously defined function  
\* Source code

JMP myEnd

\* myElse source code

myEnd

# \* حملہ چھار دھم - یک اسٹب فرور دین \*

## Multipliers

multiplicand  
A \* B multiplier

0	1
0	0
1	0

$$a * b = a \cdot b$$

- 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

There is no chance of overflow in unsigned multiplication

- unsigned number multiplied

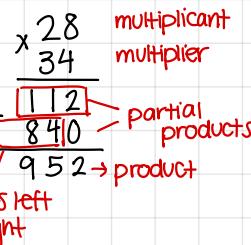
↳ for multiplying two single bits (AND gate)

delay = 1 d

cost = 1 g



the product of two n-bit numbers will always fit in  $2n$ -bits

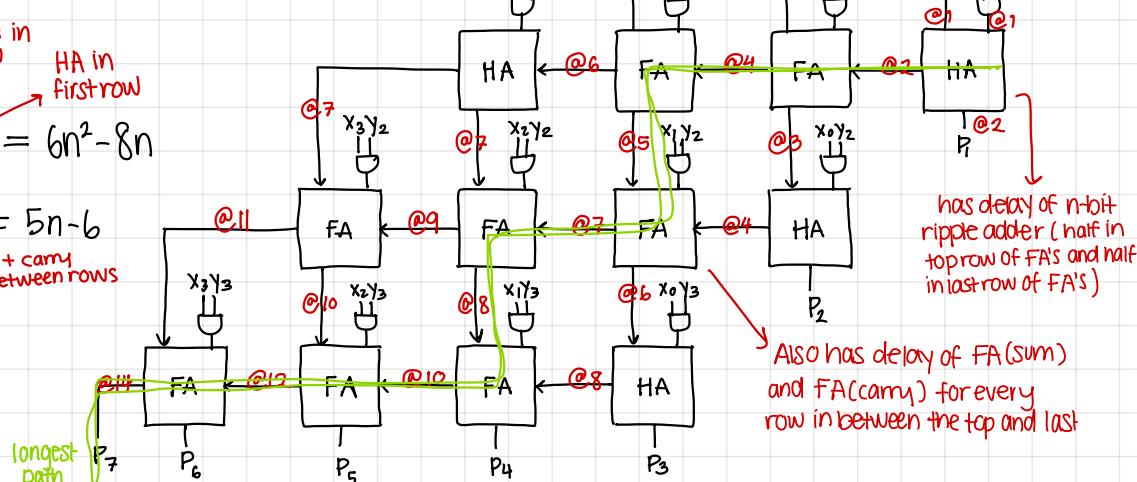
$$\begin{cases} 0 \leq A \leq 2^n \\ 0 \leq B \leq 2^n \\ 0 \leq A \cdot B \leq 2^{2n} \end{cases}$$


- 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 in each row  
HA in first row  
cost :  $n^2 + (n-1)(5n-3) - 3 = 6n^2 - 8n$

delay :  $2 + 2(n-1) + 3(n-2) = 5n-6$



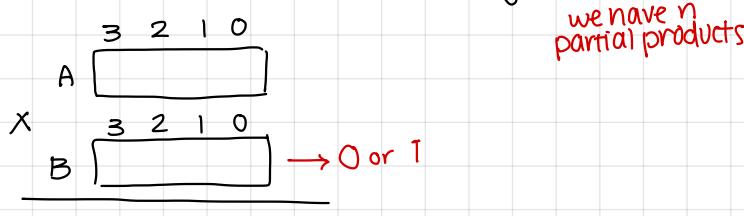
has delay of n-bit ripple adder (half in top row of FA's and half in last row of FA's)

ALSO has delay of FA(sum) and FA(carry) for every row in between the top and last

- Shift-Add multiplier

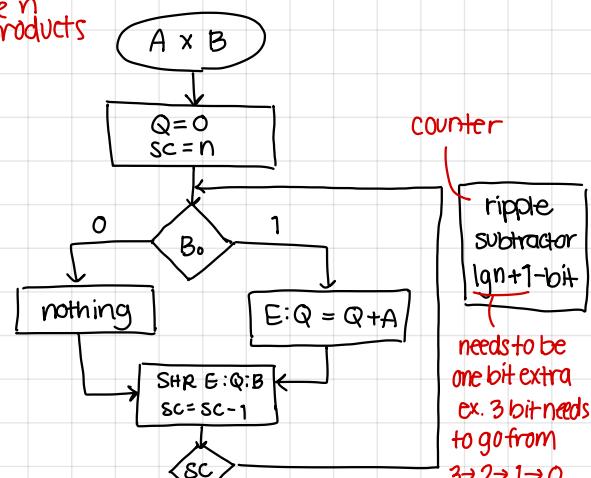
- Always look at  $B_0$  and shift B to the right in every step so  $B_0 = B_i$  to know whether to add A or shift (0)
- Instead of keeping the product in the same place and shifting the partial products to the left, the partials stay in the same place and the product shifts to the right
- After multiplying,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

↳ partial product is always either 0 or A  
 $\begin{cases} B_i = 0 & 0 \\ B_i = 1 & A \end{cases}$



only needs to check equality (nor gates)

cost =  $(\lg(n+1) \text{ bit}) + (2n+1 \text{ bit}) + (n \text{ bit}) + (\text{ n bit comparator}) + 1$



minimum time that must pass between each clock

$$\text{product} = Q:B$$

needs to be one bit extra ex. 3 bit needs to go from 3>2>1>0  $\lg(3)+1=4$   
↳ 4 different states

delay = [counter delay :  $3(\lg n + 1) - 2 + 3n$ ] =  $4n + 1$  d  
occur parallel [adder delay :  $2n + (2n+1)$ ]  
                    ripple adder carry shift register

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

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

# \* حلبات - سیسی - فوریتین ۱۲

## • 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

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

sign extension

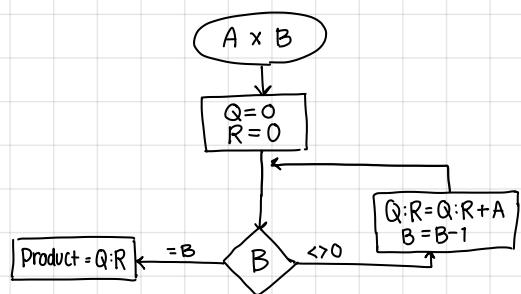
$$\text{ex. } \boxed{111} \boxed{011} = 0 \cdot 110 - 1 = -2^3 + 2^2 - 2^0 = -5$$

$$\boxed{011} \boxed{10} = 100 - 10 = 2^4 - 2^1 = 14$$

10101

0101..

terrible for  
booth representation

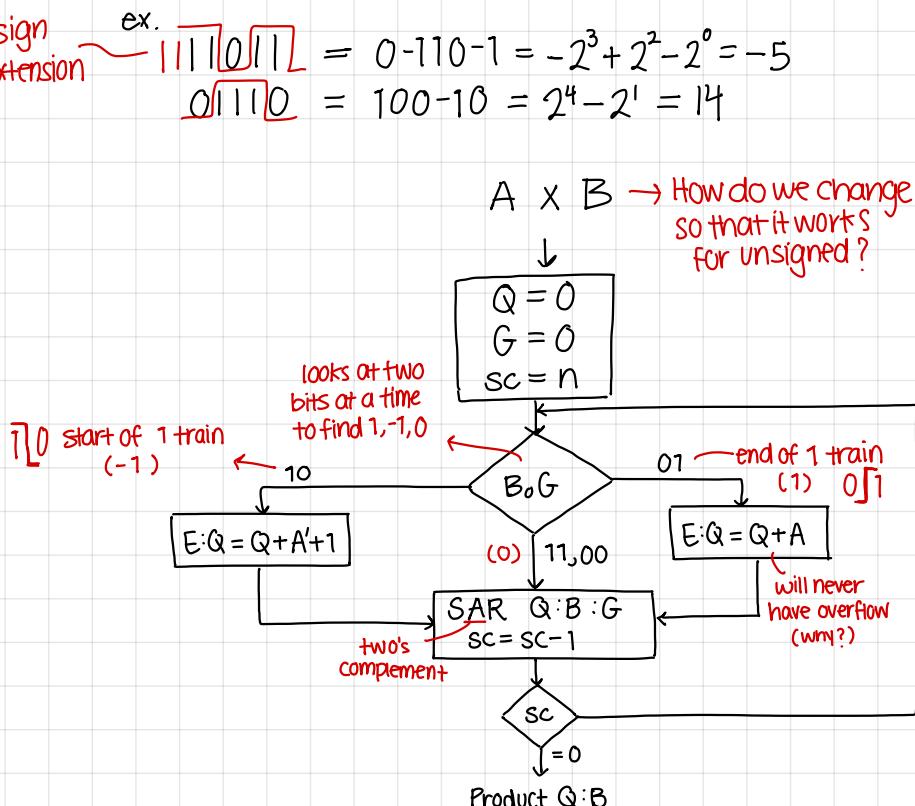
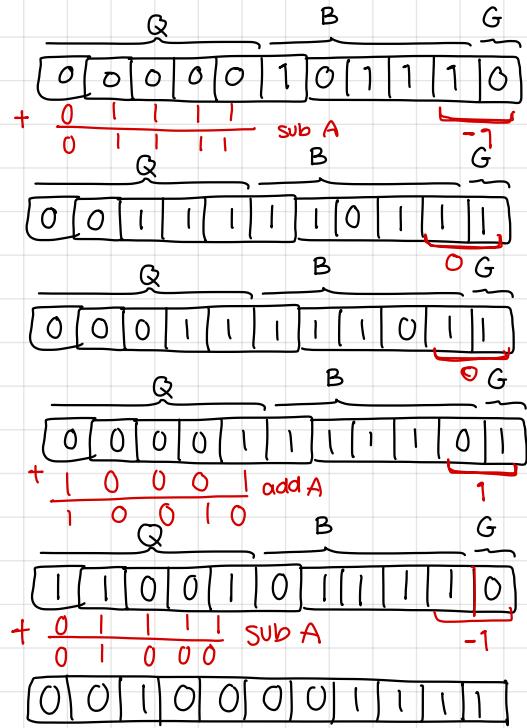


look at two numbers together



$$\begin{array}{l} B_0 \ G \\ \hline 0 \ 0 \rightarrow B'_0 = 0 \\ 0 \ 1 \rightarrow B'_0 = 1 \\ 1 \ 0 \rightarrow B'_0 = -1 \\ 1 \ 1 \rightarrow B'_0 = 0 \end{array}$$

$$\text{ex. } \begin{array}{r} 10001 \ A \\ 15 \times 9 = 135 \\ \times 10111 \ B \\ \hline 10000111 \end{array}$$



## • Shifting

SHR

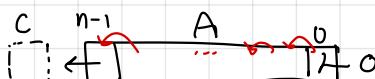


PROBLEM:

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

SHR → dividing by 2  
if A is unsigned

SHL

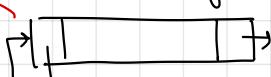


SHL → multiplying by 2 (only if A is unsigned and output digit is not last and not zero)

\* Sign extension: in 2's complement representation, the extra shifted in from the right must be a 1.

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

SAR  
shift arithmetic right



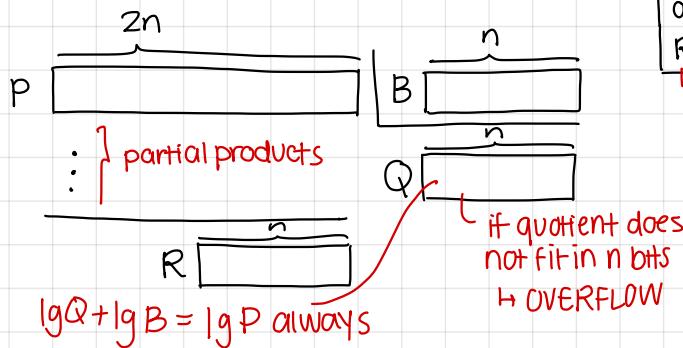
$$\text{SAL} = \text{SHL}$$



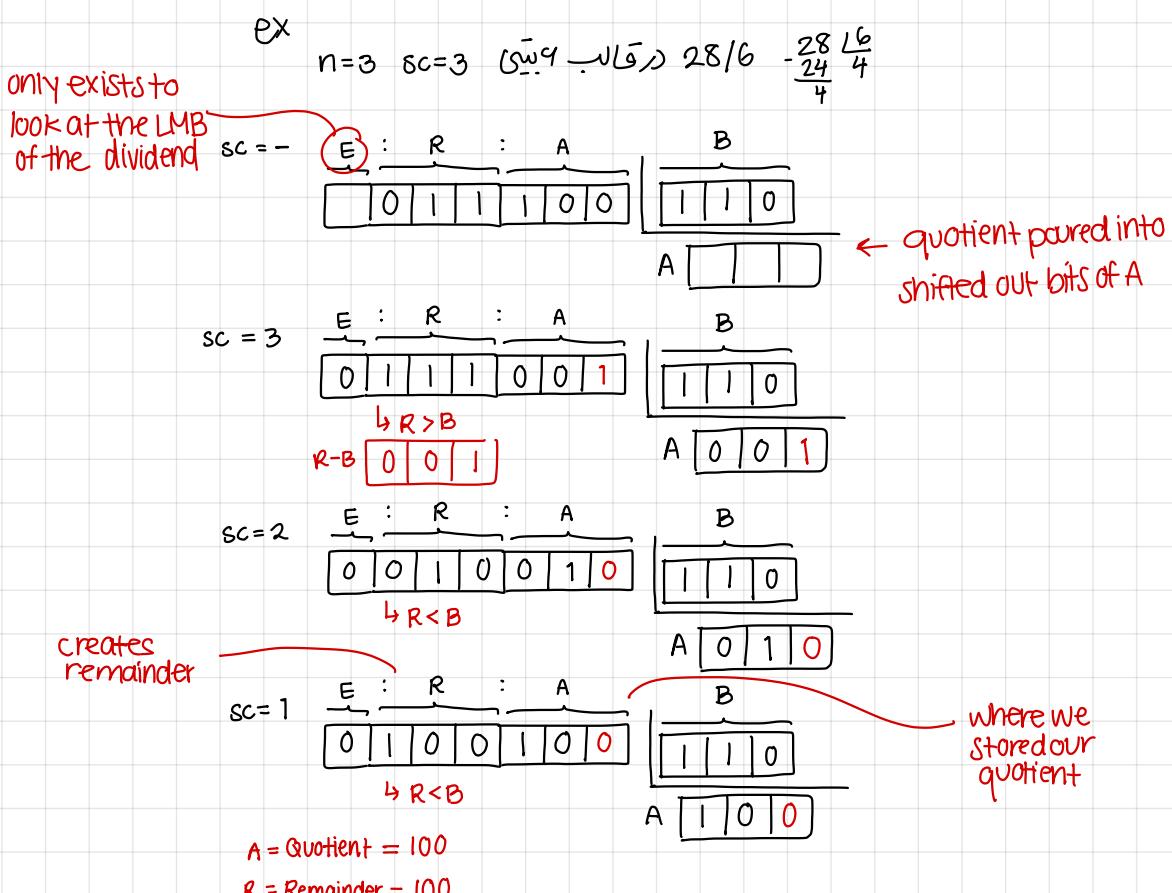
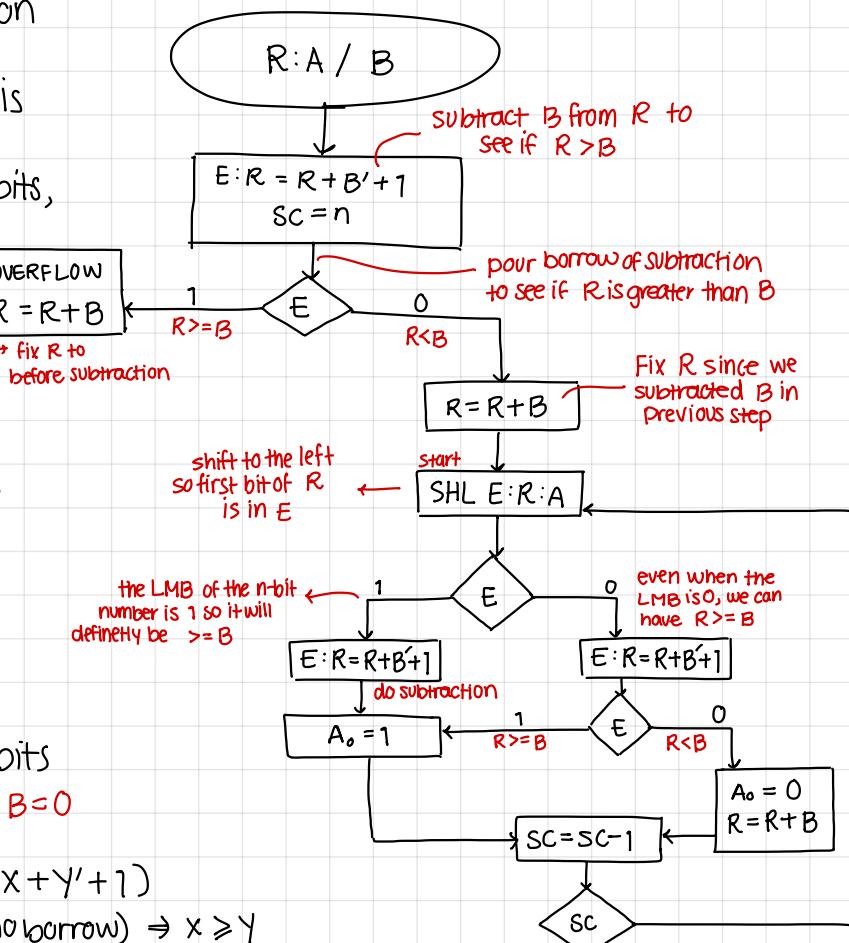
# Divider

dividend  $\rightarrow$  divisor  
 $A / B$

- Divider logic comes from the idea that division is the opposite operation of multiplication
  - ↳ in multiplication we store the product of  $n \times n$  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)  $\Rightarrow X \geq Y$
  - ↳ if complement subtraction doesn't create a carry (has borrow)  $\Rightarrow X < Y$



\* تمايز بين الجمع والطرح في المقدار \*

## Sign magnitude

- calculations are more complex
  - 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)
- Compared to two's complement representation and calculations
  - converting from decimal to IO and displaying is simple
  - easier to understand for humans
  - addition is complex
    - we have two zeros
    - multiplication is simple
  - must guarantee we won't have -0

### 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

### Subtracting sign magnitude numbers

$$S_A A - S_B B = S_A A + S_B' B$$

- everything else is the same as addition

complex for addition subtraction

$$\begin{array}{c} S_A \\ \hline S \end{array} \quad |A| = A$$

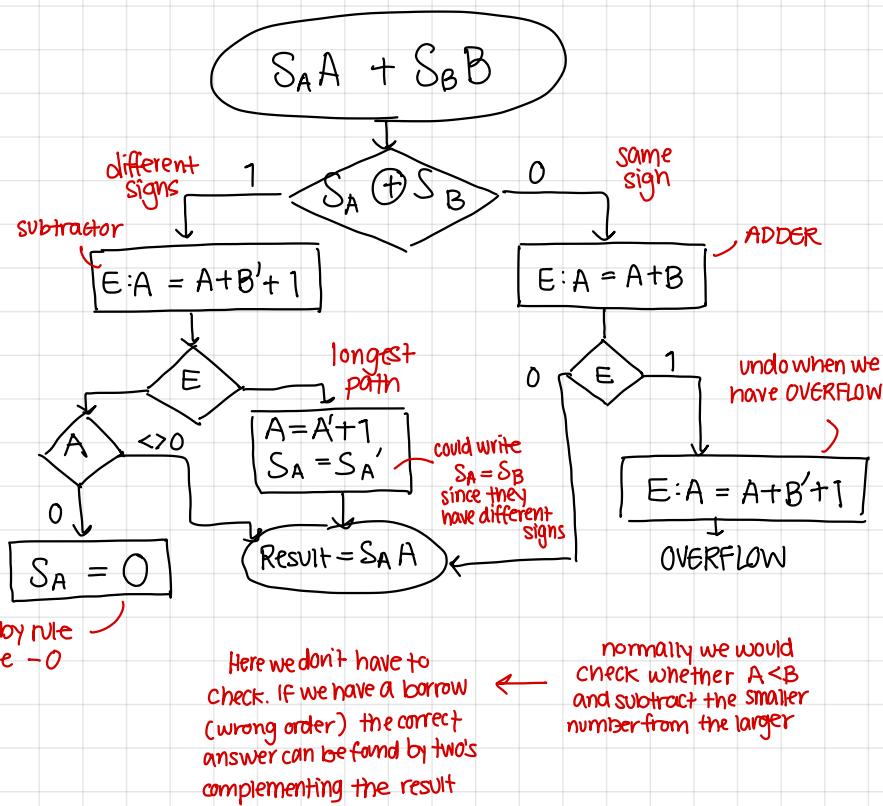
magnitude

\* easy for humans

simple for multiplication

\* we need addition / subtraction more for computers

### Adding sign magnitude numbers



### Multiplying sign magnitude numbers

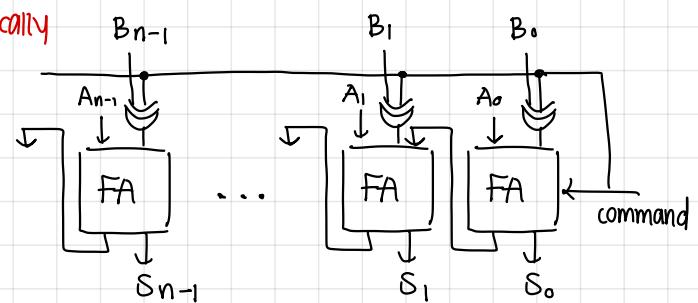
- uses the same algorithm as unsigned
- If  $S_A \neq S_B$  we make the product negative

### Dividing sign magnitude numbers

- uses the same algorithm as two's complement
- 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

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

- 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 ( $N_{max}$ )?
- Smallest positive number that can be represented (Epsilon)?
- resolution between two numbers?



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

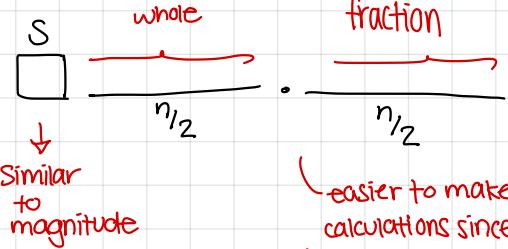
$$\rightarrow N_{max} = 15 + (1 - 2^{-4}) = 16 - 2^{-4}$$

$$N_{max} = (2^{n_2} - 1) + (1 - 2^{-n_2})$$

$$\rightarrow \epsilon = 2^{-n_2} = 2^{-4} = \frac{1}{16}$$

$$\rightarrow \text{resolution } 2^f = 2^{-n_2} = 2^{-4} =$$

usually bits split in half



\* PROBLEM:

don't always need so much space for both whole and fraction of numbers

\* BENEFIT:

very simple calculations

$$0.5 = 0.1$$

$$1 - 2^{-n_2} = 0.\underbrace{11111\dots}_{n_2}$$

rounds whenever we can no longer display bigger fractions

$$\frac{1}{2^{-n_2}} = 0.\underbrace{000\dots}_{n_2}$$

↓

not OVERFLOW, just not precise

→ OVERFLOW only when dividing by zero

we have numbers in decimal that are  $\frac{1}{3}$   
but become  $\frac{1}{3}$  in binary representation

Adder  
Subtractor  
Multiplier  
Divider

\* For fixed point representation?

## Float Numbers

\* تسلیم 10 نیمسکی - مفهوم عدد \*

\* 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  $4.813 \times 10^{+1}$

↳ Any normalized number can be represented with:

excessry 2's  
complement biased



unsigned

101.11

$1.0111 \times 2^{+2}$

0.00110

$1.1 \times 2^{-3}$

always one

fraction

exponent

$$\Rightarrow \mp 1. \text{~~~~~} \times 2^{\pm E}$$

> standard IEEE 754 { 32 bit (single precision) : S  $\leftarrow$  8  $\rightarrow$  23  $\rightarrow$   $(-1)^S \cdot 1.F(2^E)$   
64 bit (double precision) : S  $\leftarrow$  11  $\rightarrow$  52  $\rightarrow$   $(-1)^S \cdot 1.F(2^E)^8$

↳ can't represent zero → solution: choosing a random less used representation to show zero

• The exponent is stored in excessry 2's complement biased representation

① comparing the exponents is easier

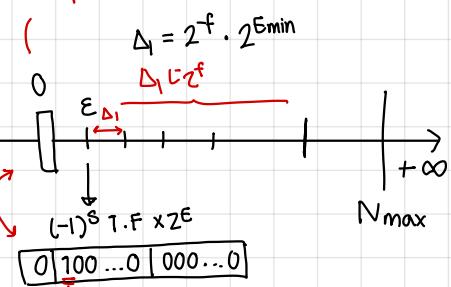
② The representation of zero (non-normal) will be all zero

> we always check if the number is zero (nan)  
before making any calculations as a floating point

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

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

can't be represented



NaN : Not a Number = (S)000

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

↳ frequently used numbers that cannot be represented

Octave 1

$$\begin{cases} N = (-1)^S \times 1.F \times 2^E \\ N_1 = (-1)^0 \times 1.0 \times 2^{E_{\min}} \\ N_2 = (-1)^0 \times 1.\underline{000...01} \times 2^{E_{\min}} = (1+2^{-f}) \times 2^{E_{\min}} \\ N_3 = (-1)^0 \times 1.000...10 \times 2^{E_{\min}} = (1+2^{-f}+2^{-f}) \times 2^{E_{\min}} \\ \vdots \\ N_f = (-1)^0 \times 1.111...11 \times 2^{E_{\min}} = \end{cases} \Delta_1 = 2^{-f} \cdot 2^{E_{\min}}$$

( 2<sup>f</sup> th number )

equal amount  
of numbers  
but the distance  
is one greater

Octave 2

$$\begin{cases} (-1)^0 1.000...01 \times 2^{E_{\min}+1} \\ (-1)^0 1.000...10 \times 2^{E_{\min}+1} \end{cases} \Delta_2 = 2^{-f} \cdot 2^{E_{\min}+1} = 2\Delta_1$$

f

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

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  $E$ ?  
in the  $f+1^{\text{th}}$  octave

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

0 |  $E_{\min}$  | xx...x → when faced with this representation, look up table

overflow  $\leftrightarrow$  calculation error

+ complex representation and hard to understand

+ (S)000 + 111 goes to (- bias 2 → adding  $2^n$  instead of  $-2^n$ )

look up table (S)000 ← E ← Y ←



3.

## control unit

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

- pipelining
- I/O

# \* مجلس نویز هم - یکشنبه IV 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)

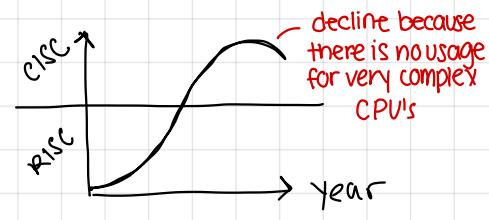
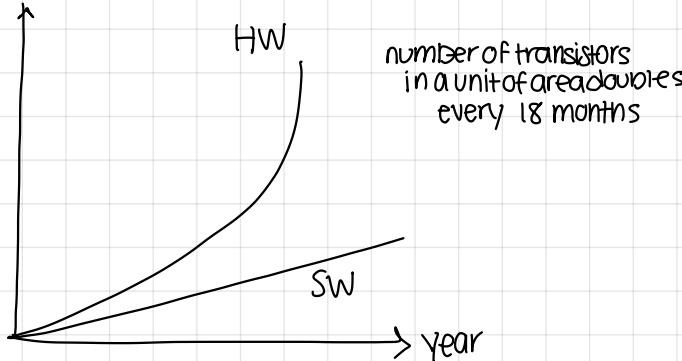
↳ consists of very specific and complex capabilities

↳ not often used by the general public ~ special capabilities are not useful to them , just expensive

### • RISC vs CISC

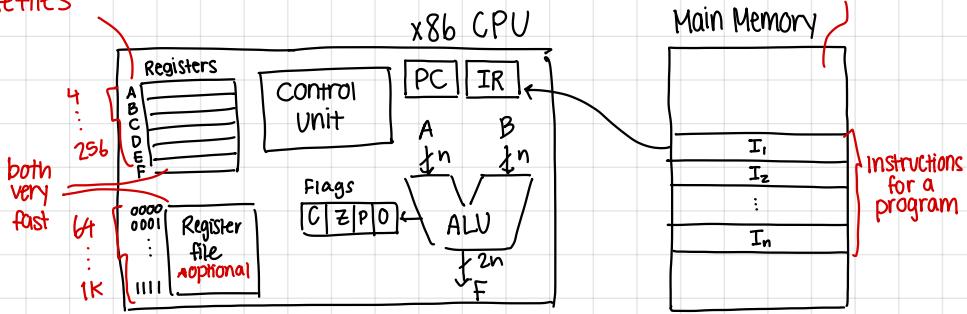
CISC	RISC	عامل مقایسه
زیاد	کم	تعداد دستور العمل
زیاد و متغیر	یک	تعداد کلاک هر دستور
سخت افزار محور	نرم افزار محور	نوع طراحی دستورات پیچیده
کم	زیاد	میزان استفاده از RAM
متغیر	ثابت	طول و قالب دستور العمل
زیاد و متنوع	کم	شیوه آدرس دهی
کم	زیاد	تعداد ثبات
زیاد	کم	تعداد دستور العمل کار با حافظه
کم	زیاد	تعداد خط برنامه
کم	زیاد	اندازه برنامه
؟	؟	سرعت برنامه

Performance



- \* CPU runs the program that is loaded in the MM
- \* data and address have different register sizes

the designer names these files



### • Program Counter (PC)

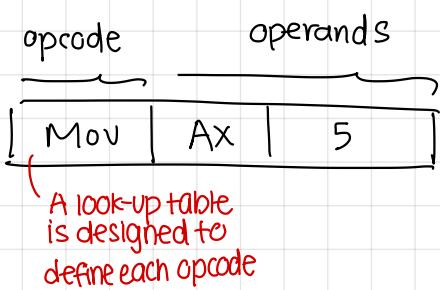
- ↳ An n bit register that stores which instruction from the Main Memory is being run
- ↳ n is the size of an address in the Main Memory ( $\lg_2(\text{MM})$ )

### • Instruction Register (IR)

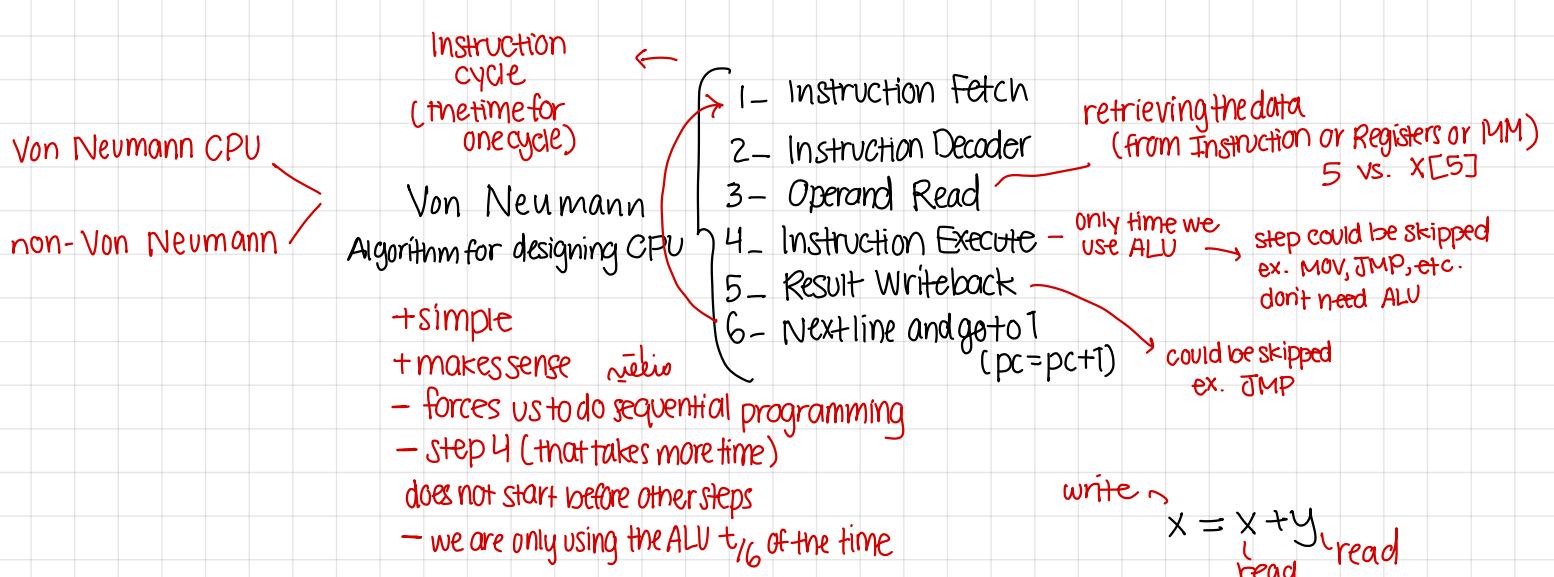
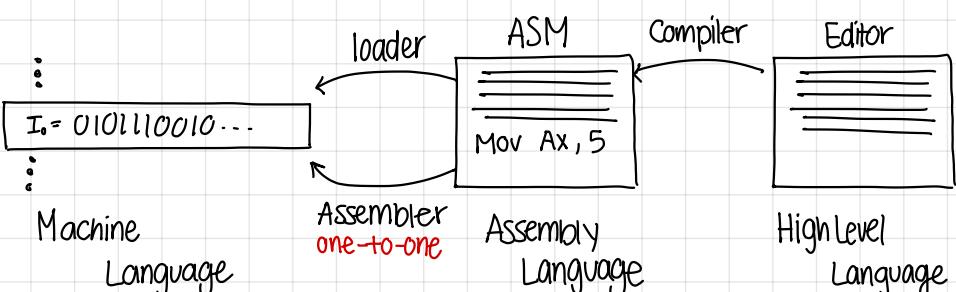
- ↳ Stores the instruction being run
- ↳ An m bit register where m is a multiple of a word in the MM

### • Registers

- ↳ The registers in the CPU can be divided into two groups
  - > General purpose : 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 used first, if more space is needed register file is used ↳ registers are faster → preferred
  - > is between 64 to 1K bytes.



\* each line in assembly is a binary instruction in memory



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
  - ↳ 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

like the select line of a mux,  
ALU operation commands,  
counters, sequencers, etc.

- 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 long time (not necessarily better/faster)
    - Multicycle - if our instruction cycle happens in multiple cycles
  - what we will be designing
  - 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.  $I_1 = \text{MOV } <\text{dst}>, <\text{src}>$  — gives all the information an architect needs when designing  
 ↗ \* move op2 to op1  
 ↗  $<\text{dst}> = \{A, B, C, D\}$

↳ these instructions can be given to both software and hardware designers

> Hardware designer can create CPU

- > Software designer can create corresponding assembly language
  - the source code that involves the commands in ISA
- > can work in parallel

\* what is the pin-out of a CPU?

greater pinout — more expensive

breaks quicker



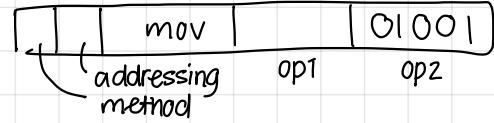
# Addressing Methods

## 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

direct      indirect

ADD Ax, [Bx]

2 + 23 )

adds data in MM address 50

.....

ADD [81], [80]

98 + 12

Both read from MM

ADD [80], [[81]]

12 + 61

theres an address in 81, where we want that value

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

## 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

## 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

## 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
  - > pointers to pointers can be implemented in software better than ④
  - > gives us the capability to determine the number of variables during runtime

## ⑥ Auto Increment / Decrement addressing method

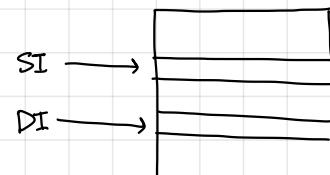
- ↳ copies SI to DI, CX times what the user input determines
  - > SI auto increments each time that memory address is accessed / written
  - > source index (SI) and destination index (DI)
  - >  $((x = (y - 1))++);$  — allows operations like this

## ⑦ Indexed addressing method (العنوان)

- ↳ the SI register is an index register
- ↳ used for quickly going through loops/arrays

ADD AX, [SI]

MOVSB CX = 4



↳ 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

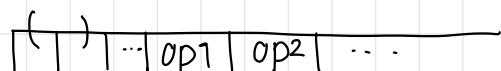
\* Operands may also be declared implicitly in the operation

ex.

CLC      Clear carry  
STC      Set carry

two operations where the operand (C) is declared in the operation

addressing methods for op1, op2, ...

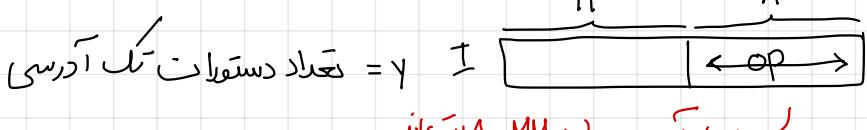
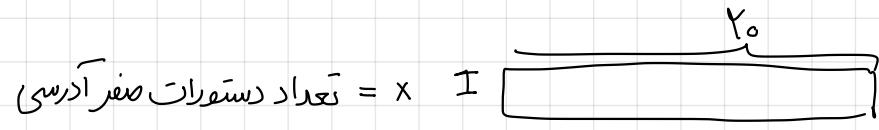


\* if each operand has its own addressing method we need to determine each one

# \* جلسه بیست و یکم - پنج سنبه ۲۱ اردیبهشت \*

## سوالات طراحی قالب دستور العمل

مثال ۱ : در یک کامپیوتر با فضای آدرس ۸ بیتی، چنانچه قالب دستور عمل مسئله از دستورات صفر آدرسی و تک آدرسی باشد و تعداد دستورات تک آدرسی ۳۰ تا باشد، حداقل چند دستور صفر آدرسی می‌توان داشت؟

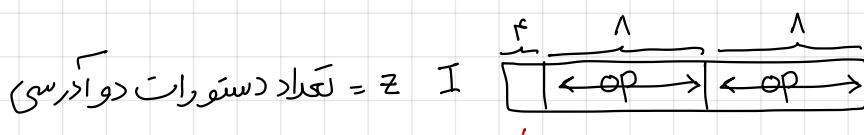
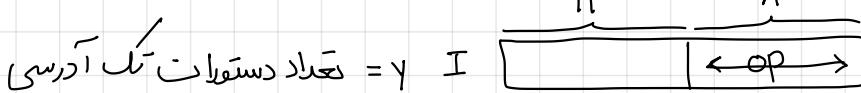
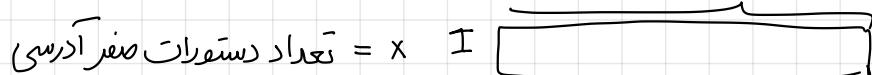


چون آدرس های MM, ۸ بیتی اند

$$\begin{aligned} x + y &\leq 2^{20} \\ x + 30 \times 2^8 &\leq 2^{20} \\ x &\leq 2^{20} - 30 \times 2^8 \end{aligned}$$

حالات های مختلف قسمت آدرس

مثال ۲ : در یک کامپیوتر با فضای آدرس ۸ بیتی، چنانچه قالب دستور عمل مسئله از دستورات صفر آدرسی، تک و دو آدرسی باشد و ۲۵ بیتی باشد، و تعداد دستورات تک آدرسی ۳۰ تا باشد، و تعداد دستورات صفر آدرسی ۳۰ تا باشد، حداقل چند دستور دو آدرسی می‌توان داشت؟



صرف از اینجا من توان نیفهم گرفت که ۱۶ حالت داریم

چون ممکن است درین حالت ها با دستورات

دیگر conflict داشته باشیم

$$30 \times 2^0 + 30 \times 2^1 + 30 \times 2^2 \leq 2^{20}$$

$$z \times 2^{14} \leq 2^{20} - 30 \times 2^8 - 30$$

$$z \leq \frac{2^{20} - 30 \times 2^8}{2^{14}} - \frac{30}{2^{14}}$$

$$z \leq 2^4 - \frac{18}{2^7} - \frac{6}{2^3}$$

$$z \leq 16 - 1 = 15$$

مثال ۳ : ۸ بیت داریم. ۱ دستور صحنی (صفر عملکری) را جلوی نمایش دهیم؟

در سیستمی که ۸ بیت فضای داریم و  
محفوظیت فضایی خاصی نداشیم  
رویس ب بهترین اتفاق است

۳ بیت برای نمایش ۸ حالت



(رسون الف)

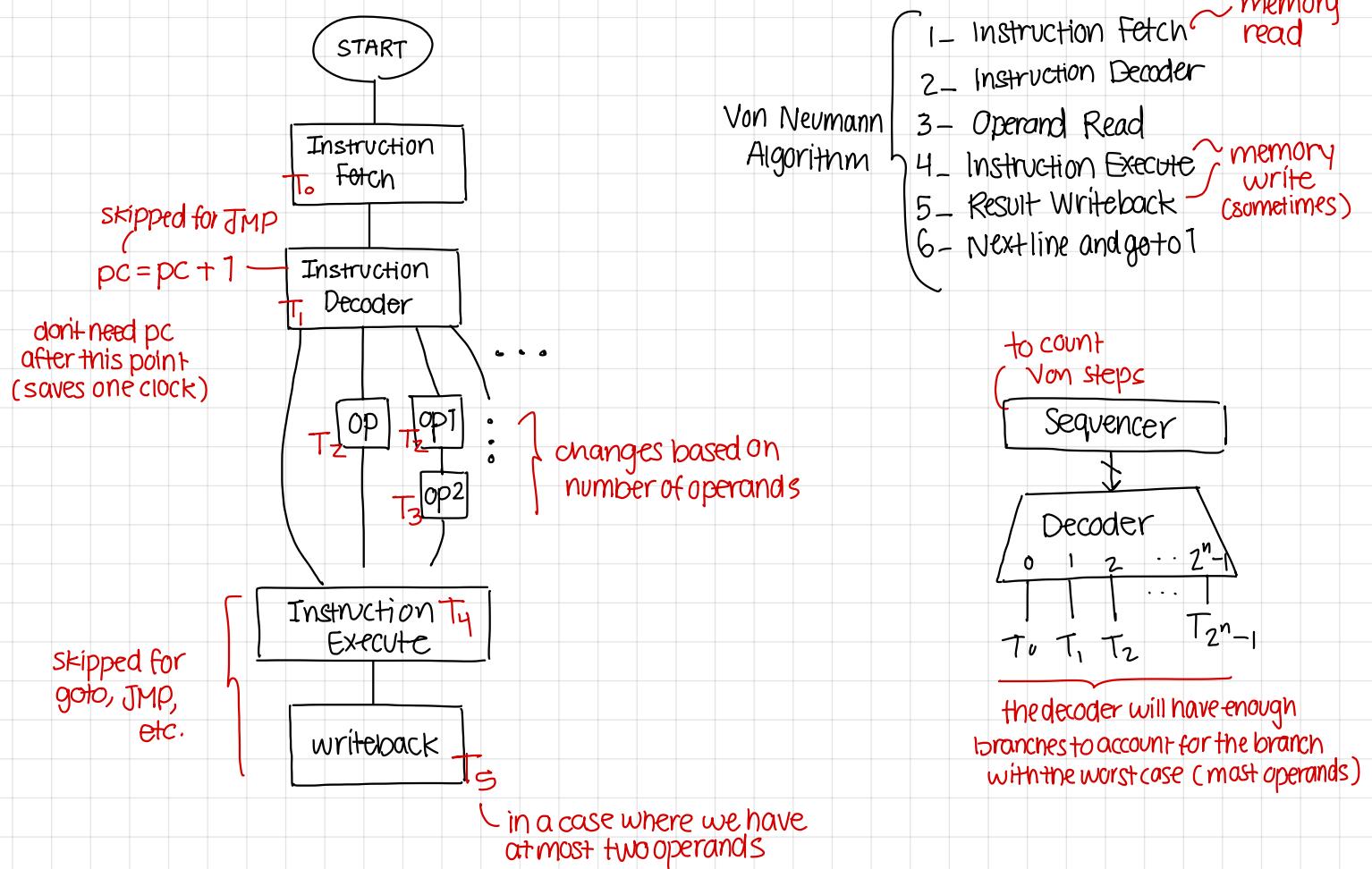
هر بیت که باید نشاند اون دستور العمل است. ← در این رویس decoder لازم نداریم

۰	۱	۲	۳	۴	۵	۶	۷
۰	۱	۰	۰	۰	۰	۰	۰

(رسون ب)

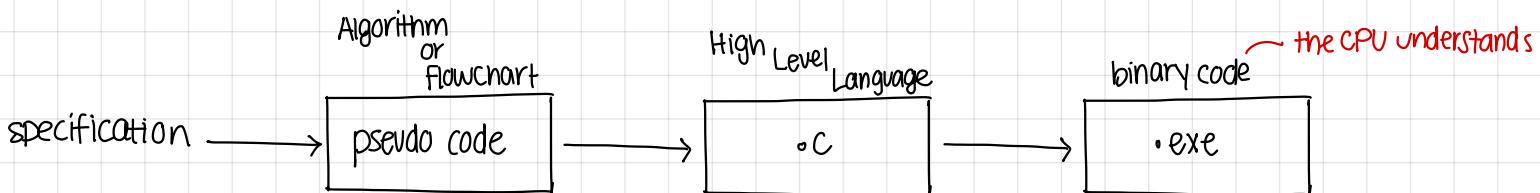
## Von Neumann Flowchart

- 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

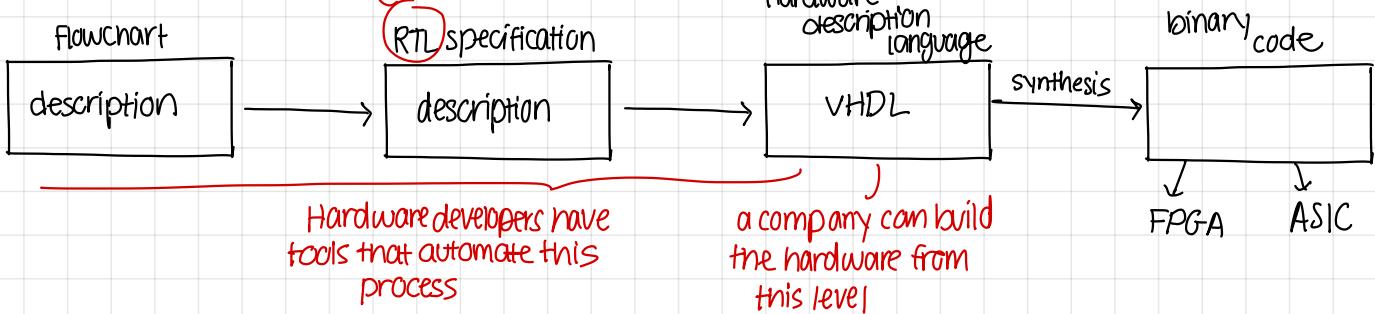


- Step 5 of designing the control unit is converting the Von Neuman Flowchart to RTL:

Software creation process



Hardware creation process  
↳ same flow as software



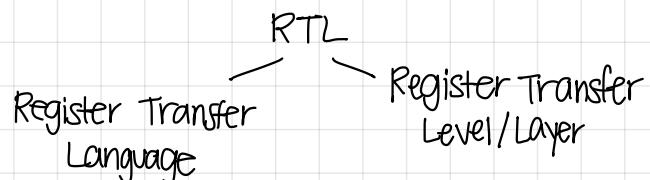
## RTL I

### Microoperations ( $\mu$ op)

- Any description for hardware in RTL is a series of microoperations
- A microoperation can be defined as the transferring of data from an expression of registers based on a condition

*condition T or F*  $\curvearrowleft T : R \leftarrow \langle \text{expression of registers} \rangle$

"*جُنْهُ لِـ expression؛ لِـ الْعَلَى!*"



*(only describes the transfer between registers)*

*must consist of microoperations*

### Converting from Hardware to RTL (HW = RTL)

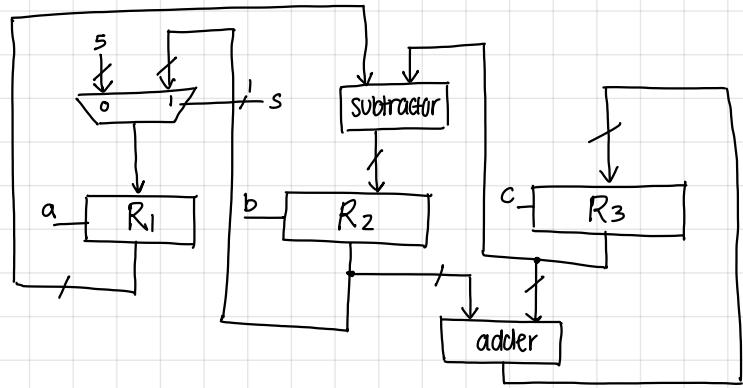
- decide the condition of each register transfer by the load input as well as the gates (ex. mux) that come before it

$$S.a : R_1 \leftarrow R_2$$

$$S'.a : R_1 \leftarrow 5$$

$$b : R_2 \leftarrow R_1 - R_3$$

$$c : R_3 \leftarrow R_3 + R_2$$



\* 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

$$\bar{ab} : R_1 \leftarrow 10$$

$$c : R_2 \leftarrow R_1 + R_3$$

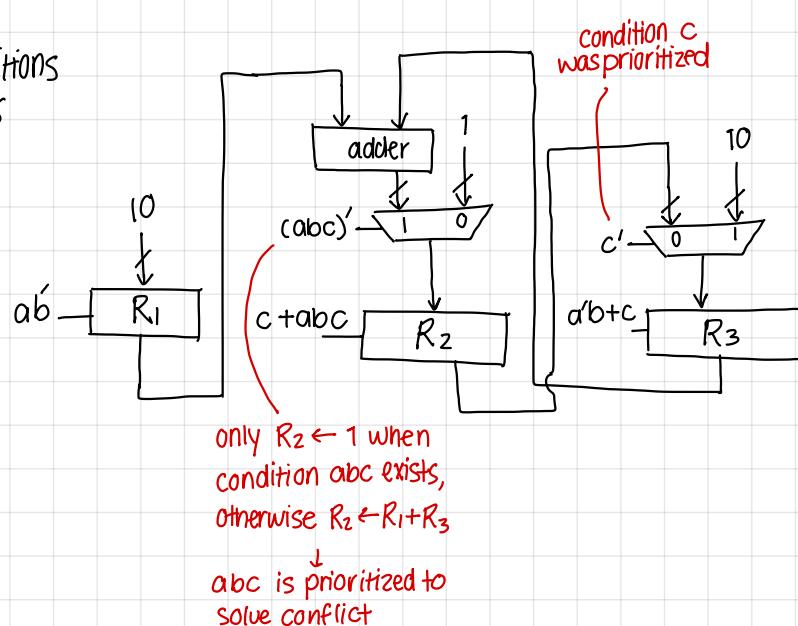
$$\bar{ab} : R_3 \leftarrow R_2$$

$$abc : R_2 \leftarrow 1$$

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

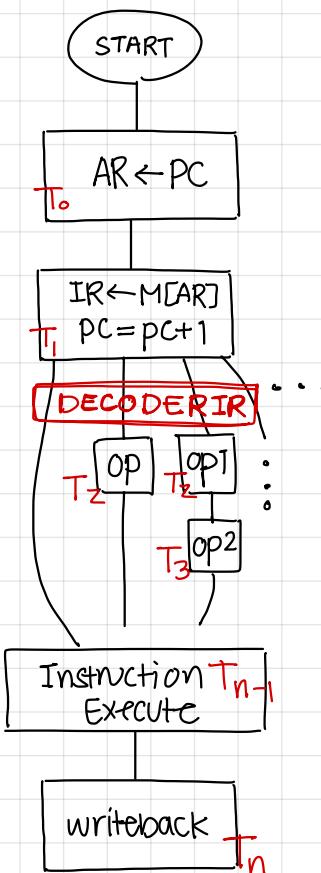
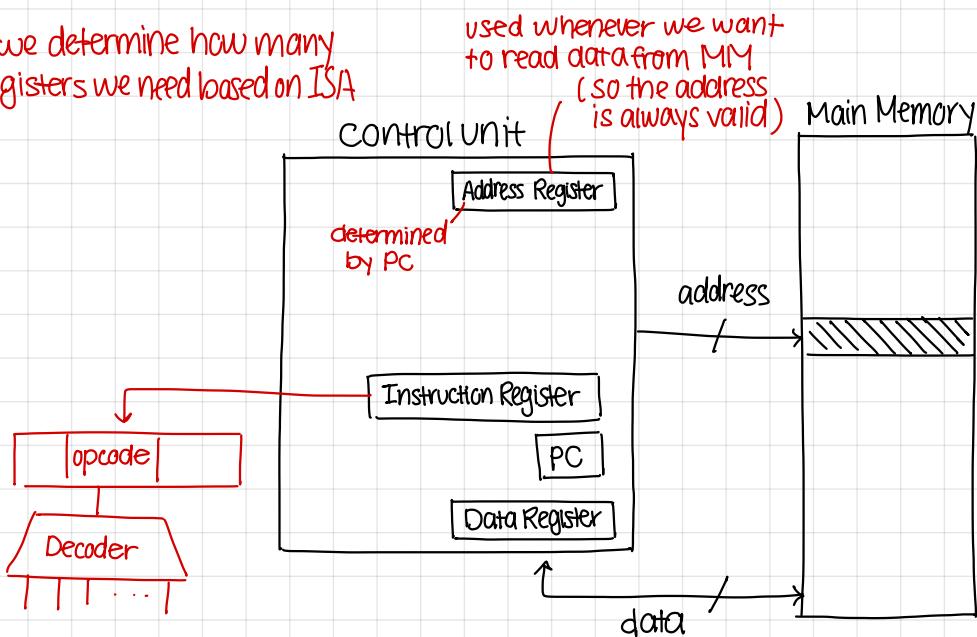
*we OR the two conditions*

*conflict when both are true  
must prioritize one condition*



- Step 6 of designing the control unit is converting our RTL to hardware
  - ↳ 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

\* we determine how many registers we need based on ISA



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

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

- **Bus** : A common wire that all the registers use to transfer data

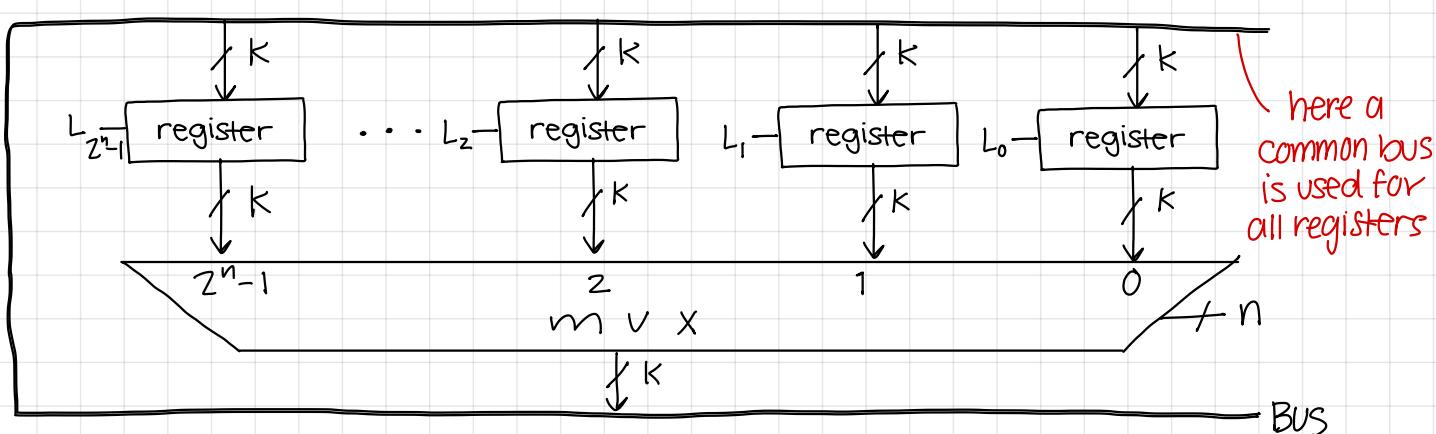
- + prevents spaghetti wiring (less chance of error)
- only one register can use it at a time (slower)
  - ↳ cheaper but slower

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  $T_i$  can't be done in one step

- ↳ 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

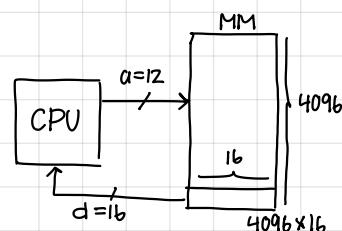
- ↳ Data bus : bus data moves on (with the size of data)
- ↳ Address bus : bus address moves on (with the size of MM addresses) ↳ can be combined into one bus



# جامعة سوهاج - كلية طب اسيوط

## The Basic Computer

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

- Instruction format design

$\hookrightarrow$  Instructions are stored in 16 bits
 

- / 0 - direct
- > left most bit specifies addressing mode
  - \ 1 - indirect

$\hookrightarrow$  has a single general purpose register  $\rightarrow$  Accumulator  
 > is an operand and stores result in itself

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

- Control Units are implemented in two ways:

- ① Hardwired Control

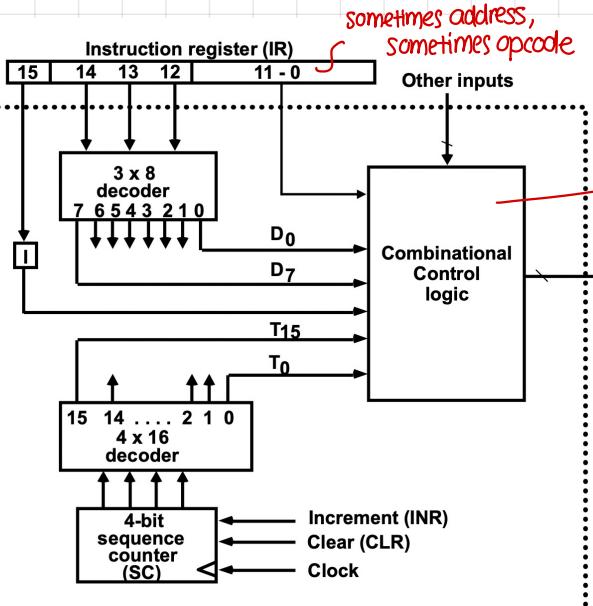
$\hookrightarrow$  physical permanent circuits (fast)

- ② Microprogrammed Control

$\hookrightarrow$  programmed + optional (slow)

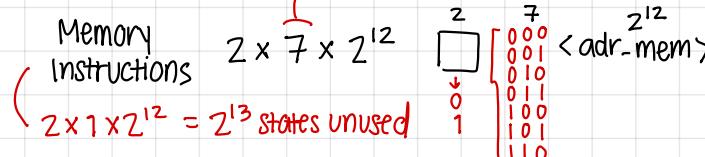
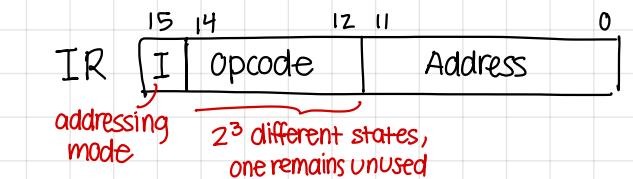
- Decoder is connected to sequencer to keep track of the step in the Von Neumann algorithm
  - $\hookrightarrow$  size determined from max clocks in cycle

- The other decoder decodes the opcode



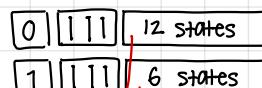
never have both in  
 Von Neumann Algorithm  
 11  $\rightarrow$  ERROR  
 10  $\rightarrow$  write to MM  
 01  $\rightarrow$  read from MM  
 00  $\rightarrow$  nothing

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



Register Instructions

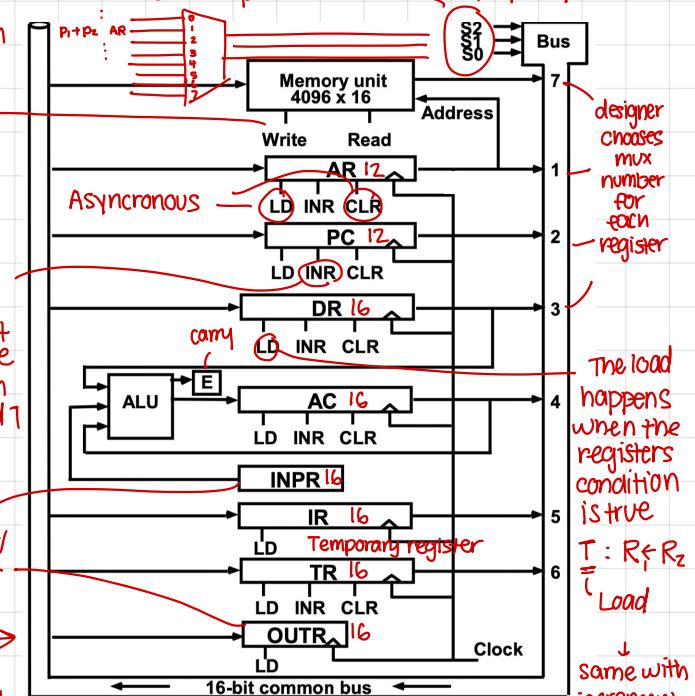
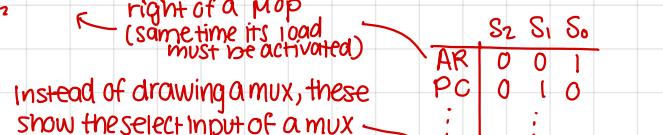
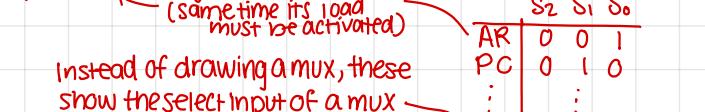
Input/output Instructions



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

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

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



\* always include size of each data line

steps 1-3 done :

1. determining data path sizes
2. determining registers
3. Instruction formatting

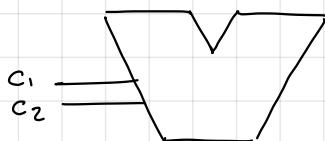
T:  $R_1 \leftarrow R_1 + 1$   
 and CLR  
 $R_x \leftarrow 0$   
 if said MOPs  
 do not exist,  
 we connect to ground

- 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

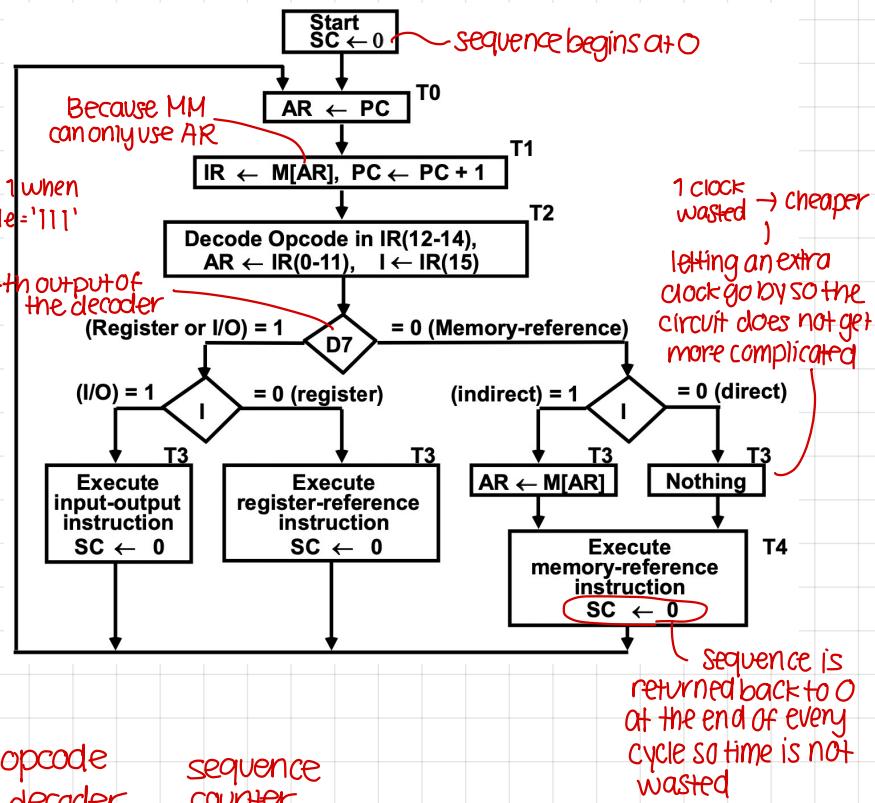


	C1 C2	DITs
AND	0 0	DITs
ADD	0 1	DITs

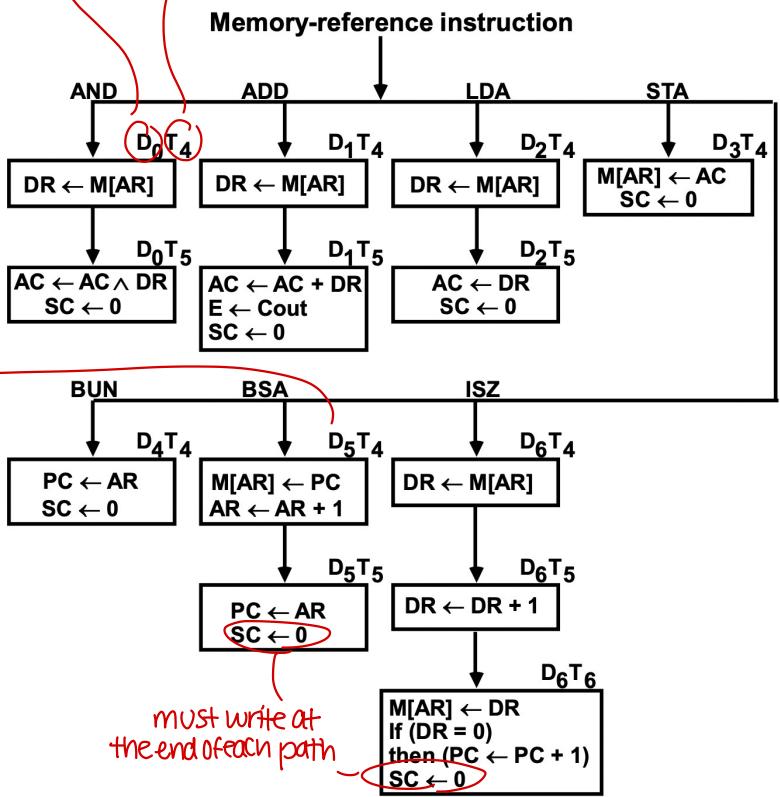
↳ we OR the conditions in which we want to add

- 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

D<sub>5</sub> must be ANDed with T<sub>4</sub> otherwise all operations wired to T<sub>4</sub> will run each cycle at the same time



opcode decoder  
sequence counter decoder



### Commonly used special purpose registers :

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

# \* تقييم الأداء - مراجعة و ملخص \*

## Performance Evaluation

- 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  $\rightsquigarrow$  CPU A and Computer B  $\rightsquigarrow$  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
  - ↳ only valid if it's a computer running only the one program (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

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

$$\text{CPI} : \frac{\text{# of clocks}}{\text{# of instructions}}$$

$$\text{IPC} : \frac{\text{# of instructions}}{\text{# of clocks}}$$

- ★ calculate the CPI of your own computer
- ↳ look at ISA and 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

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

same program takes less instructions, but MIPS is lower

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

$$\begin{aligned} \text{MIPS}_A &= 2 \leftarrow k \text{ instructions} \\ \text{MIPS}_B &= 3.5 \leftarrow 2k \text{ instructions} \end{aligned}$$

one program

### • Amdahl's law

- ↳ if  $\alpha$  percent of a program has sped up by  $n$ , the speed up of the program is:

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

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

unchanged part of the program

percentage of the program that has sped up

speed up of said program

# \* جلسه بیست و پنجم - سهشنبه ۲ خرداد \*

سوالات محاسبه کارایی

مثال ۱ . دریک کامپیوتری مسئله از دستورات زیر پس از بهینه سازی معده شدنک ۱۰٪ دستورات INT بیان افزایش سرعت حفظ است. لذا دستور ADD بینام ADDINT ساخته شد که آنرا است. مسخن کن ADD همراه هستند.

$$T_1 \approx n \cdot CPI \times \frac{1}{f}$$

۱۰۰ دستورات قبل از بهینه سازی

$$T_2 \approx n' \cdot CPI' \times \frac{1}{f}$$

۹۸ دستورات بعد از بهینه سازی

$$CPI_1 = \left( \frac{۲۰}{۱۰۰} \times ۱۰ \right) + \left( \frac{۳۰}{۱۰۰} \times ۱۸ \right) + \left( \frac{۱۰}{۱۰۰} \times ۲۰ \right) + \left( \frac{۴۰}{۱۰۰} \times ۸ \right) = ۱۰/۸$$

$$CPI_2 = \left( \frac{۱۷}{۹۸} \times ۱۰ \right) + \left( \frac{۲۱}{۹۸} \times ۱۸ \right) + \left( \frac{۱}{۹۸} \times ۲۰ \right) + \left( \frac{۴}{۹۸} \times ۸ \right) = ۱۰/۳۰۶$$

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

$$\text{speed up} = \frac{T_2}{T_1} = \frac{100 \times 10/8 \times ۱/۸}{98 \times 10/306 \times ۱/۸} = 1/0^{۳۰۶}$$

دستور ADDINT که قبل از دستورات اجرایی می شود  
حالا در اینجا می شود

\* در این سوال،  $\alpha$  را درصدی از نرم افزار که نتیجه گرفته را دعیق نمایم چون که علاوه خطای برنامه با این تغییر، موضع می شود.

۲ تفاوتی بعد از بهینه سازی درست است نه برای  $\alpha$   
اگر با وزن  $\frac{۱۵۵}{۹۸}$  ۱۵۵ کافیانه اعمال را حساب کنیم درست می شود.

مثال ۲ : دریک کامپیوتر دستورات کوافلی را ۲ برابر سریع کرده اند که این دستورات ۱۰٪ کل دستورات برنامه ها

$$f = ۳۵\% = ۰/۳۵$$

$$n = ۲ برابر = ۲$$

$$\text{Amdahl's law} : \frac{1}{(1-\alpha) + \frac{\alpha}{n}} = \frac{1}{(1-0/35) + \frac{0/35}{2}} = \boxed{\frac{1}{1/21} \text{ سریع}}$$

مثال ۳ : صورتی پس از بررسی ISA، بهینه سازی انجام داده اند بطوری که واندر CPI دستورات را ۱۰٪ درصد بعید دهنده می سازند. میزان سریعی؟

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

CPI  $\Rightarrow$  درصد بهتر شدن

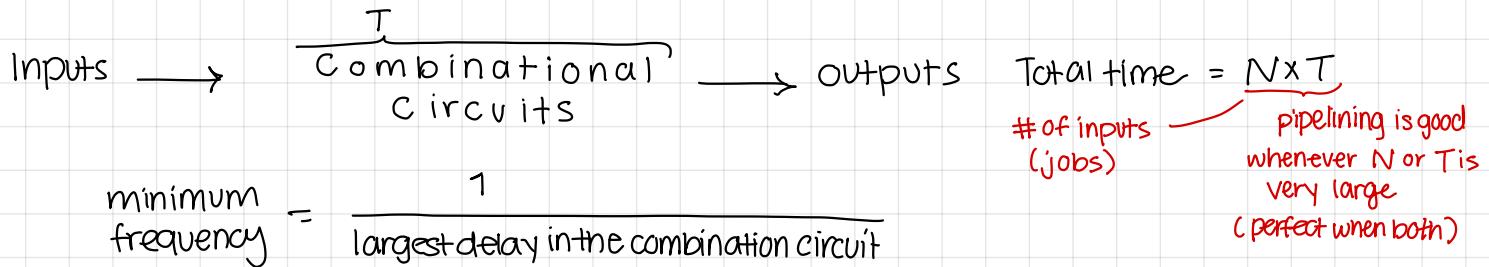
فرکانس ۱۰٪ درصد  
بیشتر شدن

$$\text{Speedup} = \frac{t_{old}}{t_{new}} = \frac{\text{Exec old}}{\text{Exec new}} = \frac{\alpha \times CPI_{old} \times f_{new}}{\alpha \times CPI_{new} \times f_{old}} = \frac{1}{0.6} \times \frac{0.2}{1} = \frac{1}{3}$$

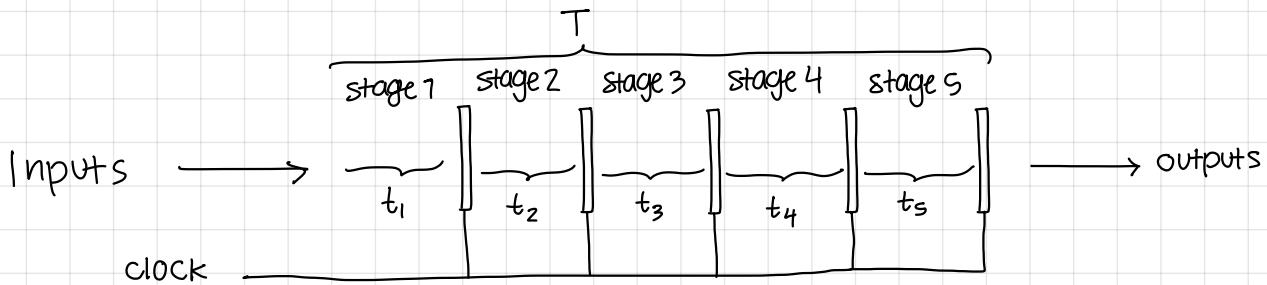
# Pipelining

"رسنے دست کرنے" 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



- Without pipelining,  $N$  inputs will take  $T \times N$  time to be executed.



$$\text{Total time} = (5 + (N-1)) \times t$$

↳ 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:

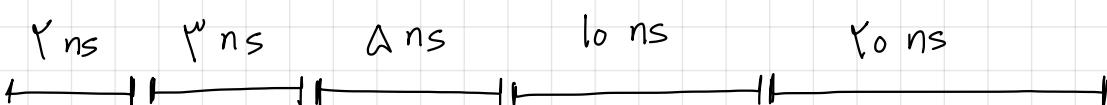
$$\textcircled{1} \text{ Speed up} = \frac{N \times T}{(K + (N-1)) \times t}$$

② speedup when  $N \rightarrow \infty$   $\approx \frac{I}{t}$

③ speed up when  $N \rightarrow \infty$  and  $t_i$  of the stages are equal  $\approx \frac{T}{\frac{T}{K}} = K \sim$  equals the number of stages

(more stages  $\rightarrow$  more speedup)

**مثال:** فرض کنہ 1000 کا input داریم و اس کا مرحلہ ہی صورت زیریاست۔ میں ان تسلیع کیں؟



$$\text{Total Time} = \frac{1000 \times 1\text{ ns}}{(A+999) \times 1\text{ ns}} = \frac{1000}{1000} \approx 1\text{ ns}$$

1/98 (.)

K (2)

٢) الف

△(L)

علمی  $\rightarrow$  تنهایی تسریع اینزاره تعداد مراحل است  
که  $N$  به اینزاره کافی بزرگ باشد ( $\infty$ )

- The Von Neumann algorithm is happening  $\infty$  times  $\rightarrow$  we can pipeline
  - $\hookrightarrow$  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 ( $\infty$ )
- 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
- $\hookrightarrow$  the  $n^{\text{th}}$  instruction is done at time  $n+4$  **number of instructions**

CLOCK	1	2	3	4	5	6	7	8	...	T
IF	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>					
ID		I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>				
OR			I <sub>1</sub>	( <span style="color: red;">○</span> )	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>		
EXE				I <sub>1</sub>	( <span style="color: red;">○</span> )	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	
WB					I <sub>1</sub>	( <span style="color: red;">○</span> )	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>

when we face hazards like  
 data dependencies, "bubbles"  
 are created in our pipeline

$\hookrightarrow$  become vacant stages

$\hookrightarrow$  if executing takes  
 more than one clock  
 we can divide into  
 stages  $\text{exe}_1 \dots \text{exe}_m$

- There are cases where we can't start stages of an instruction without the previous instruction completing (hazards)

$\hookrightarrow$  we can pause the dependent instruction  $\rightarrow$  stall

$\hookrightarrow$  creates "bubbles"  $\rightarrow$  vacant stages resurface in every stage after

- where else can we use pipelining in the computer?

Cache X  $\rightarrow$  no spatial or temporal locality  
 ALU  $\checkmark$   $\rightarrow$  ex. array multiplier

- In the best case, the computer's speed is multiplied by 5

$\hookrightarrow$  not realistic because there are hazards

$\hookrightarrow$  superscalar  
 computers

$\hookrightarrow$  stage after every  
 AND or every addition

### Solvable issues

#### Hazards in pipelining

$\hookrightarrow$  ex. when we use writeback

- ① data dependency  $\rightarrow$  when there is data dependency between instructions

$\hookrightarrow$  possible solution: Instruction reorder - moving dependent instructions

$\hookrightarrow$  done both by the  
 programmer and compiler

- ② resource conflict  $\rightarrow$  when both instructions require access to a common resource

$\hookrightarrow$  both instructions needing the ALU

$\hookrightarrow$  possible solution: adding new resources (ex. separate adder for a register)

$\hookrightarrow$  costly (\$)

- ③ when we have jmp/goto instructions

$\hookrightarrow$  On average of every 6 instructions, one of them contains jmp ( $1/6$ )

$\hookrightarrow$  worse case is when we have conditional jumps since the next instruction is determined by flags

$\hookrightarrow$  possible solutions:  $\begin{cases} \text{Take one randomly - take the jmp's if or else randomly} \\ \text{Interleave execution - do both the if and else before hand} \end{cases}$

$\hookrightarrow$  Branch predictor - guesses which one by holding the history

$\hookrightarrow$  if a guess worked  
 once it'll be used again

$\hookrightarrow$  Branch target buffer (BTB) - buffer that stores whether it was if or else before

$\hookrightarrow$  Loop buffer - buffer containing the n most recently fetched instructions (for loops)

$\hookrightarrow$   $I_2 \neq I_{i+1}$

$\hookrightarrow$  "jmp"

$\hookrightarrow$  still don't know if  
 we go to  $I_2$  when  
 we operand  
 read in  $I_1$

\* خرداد ۹ می - پیش‌نیوی معا

## Input/Output Organization

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

- Separate buses for data, address, control, and status for each I/O.

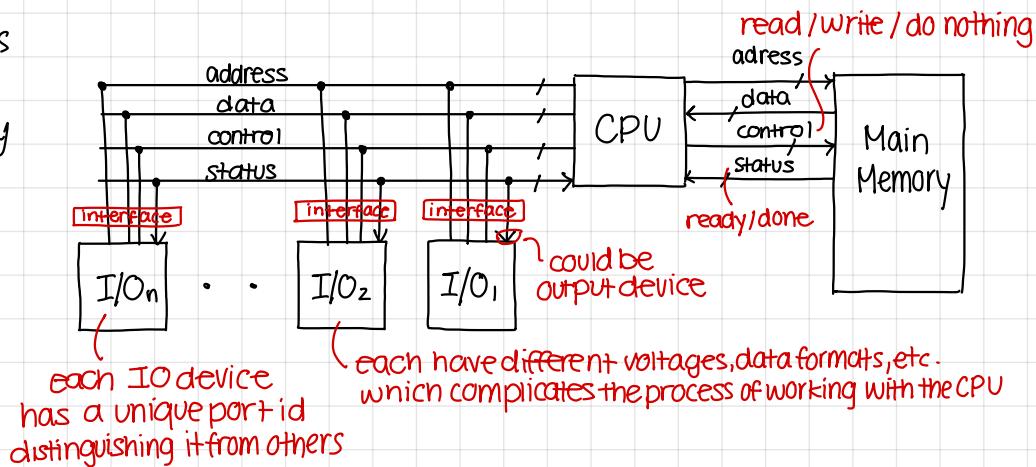
↳ the Main memory and

I/O devices are treated as separate entities

↳ not practical - too many buses going into the CPU

> hard to manage

having this many pins can cause error



- Common buses for address, data, control and status

↳ the Main memory is being treated as an I/O device

↳ we read and write from I/O devices using memory addresses

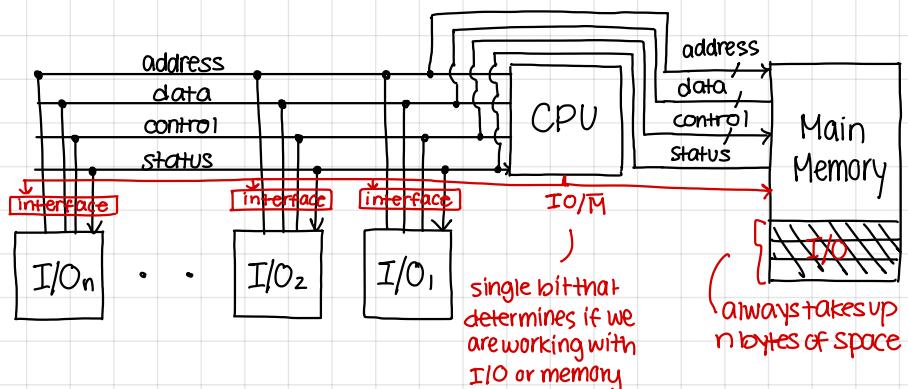
↳ wastes n bytes of space in the memory for I/O devices

↳ memory is being used unnecessarily each time we use an I/O

↳ solvable by adding an IO/M bit

↳ also won't take up the extra space anymore

memory mapped I/O



- Separate buses for control and status, and common buses for address and data

↳ a common bus is used for both

I/O and memory, and each one

knows if that address is for them

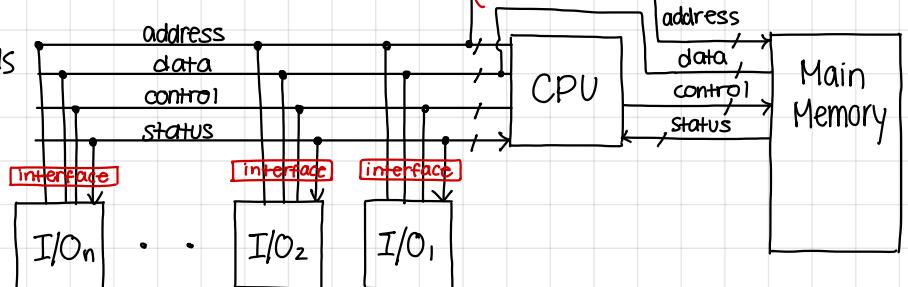
based on the separately activated controls

↳ requires more instructions in our ISA to get/give data from our I/O devices

ISA { IN (port) }  
          { OUT (port) }

↳ in mapped memory I/O devices could be accessed with the same read/write controls

Isolated I/O



- 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

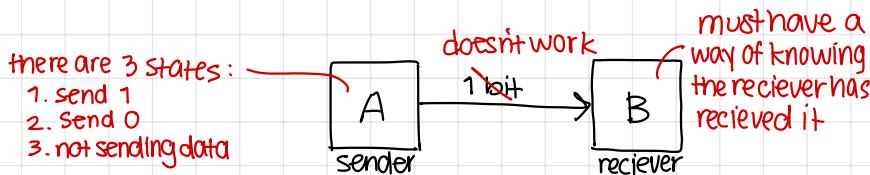
↳ interfaces have 3 main tasks

↳ added part on motherboard  
(not our concern)

- Adjusting voltage levels of the I/O device
- Creates a common data format design between the I/O and CPU
  - ↳ the interface is responsible for making this change
- Increases speed between I/O and CPU

isolating the CPU from the I/O device

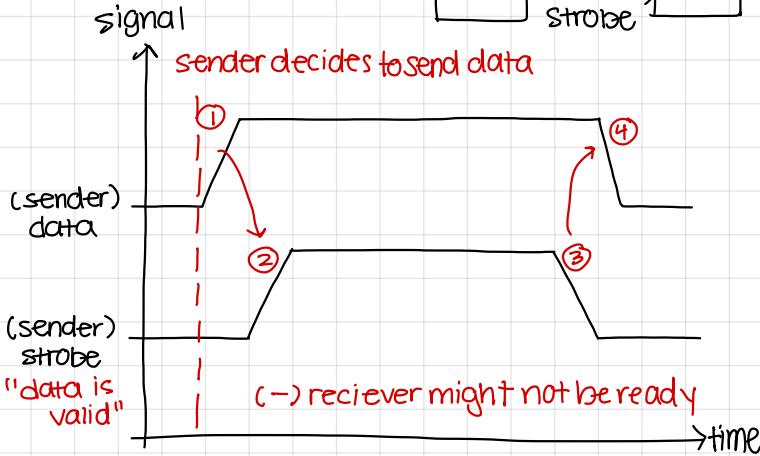
- Data needs to be transferred between the CPU and I/O devices
  - ↳ the devices don't necessarily have a clock, and we want to use minimum wires
- A single bit cannot be used to transfer a bit of data



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

### ① Strobed data transfer

(a) Sender controlled

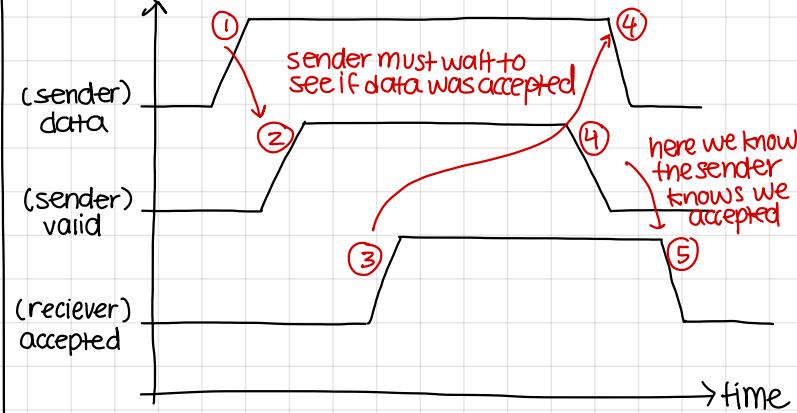


### ③ Handshaking data transfer

(a) Sender controlled

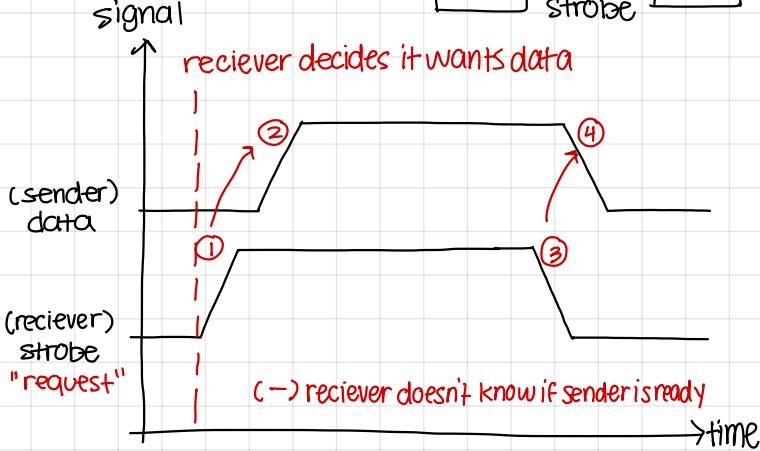
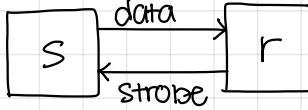


signal



### ② Strobed data transfer

(b) Receiver controlled

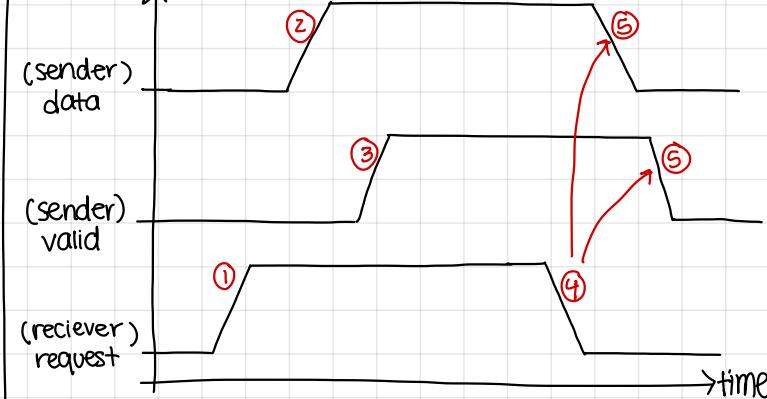


### ④ Handshaking data transfer

(b) Receiver controlled



signal



- When the CPU wants to read data:

↳ the CPU sends an address to be read from memory  
↳ memory sends back data and shows it's 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

\* دخواج اتوماتیک - پریس و تامین مالی \*

## Direct Memory 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 receive 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
  - If the program being run is high critical, the DMA will not be acknowledged

DMA:

- ① Burst transfer → transferring 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

$\text{Mem}[x] = \text{key scanf}()$  ⇒ 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  
↳ original pc is stored somewhere to come back to

only practical in multi tasking (multiple program) computers

- Must have a way of returning back to the program once input is entered
  - ① polling - continuously checks if the input was entered ex. while  $\text{rest} < > \text{null}$
  - 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 it's 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
3. vectored interrupt

look up table

the I/O device itself determines the ISR address after Interrupt Acknowledge

