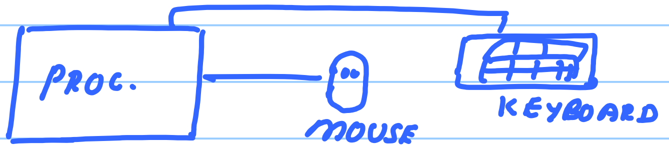
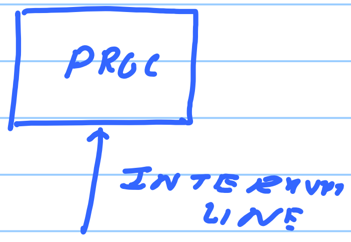
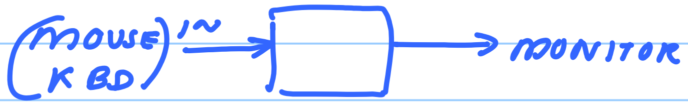
Note Title 19-10-2012

Oct. 19

Interrupts





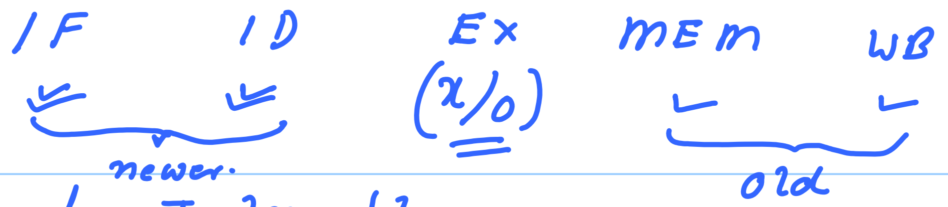


ARM ASSEMBLY :

PORTS → (OUT)

INPUT :

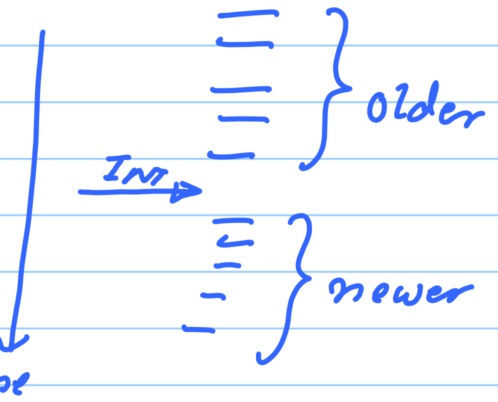
MOUSE → generates a signal → INTERRUPT



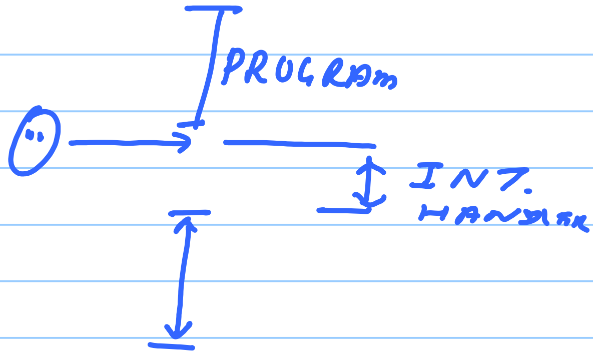
External Interrupt

Internal Interrupt

Precise Interrupt:



Programmer's Perspective:



When the interrupt is processed, (1) all the older instructions need to have completed. (2) no newer instruction should have completed. [PRECISE INTERRUPTS]

Older inst: Instructions that entered the pipeline before the interrupt.

```
        e = 0
      { a = 3;
        b = 4;
        c = a + b;
INT. -> d = *(0x0000 0000); ~ // accessing illegal address
        e = 2;
```

Handle segmentation fault
signal (..){
SIGSEGV
}

$\boxed{C = 7, e = 0}$

IF	ID	EX	M	WB
—	—	—	×	✓

Ensure precise Interrupts:

1) allow older instructions to finish
2) replace all newer instructions with NOPs

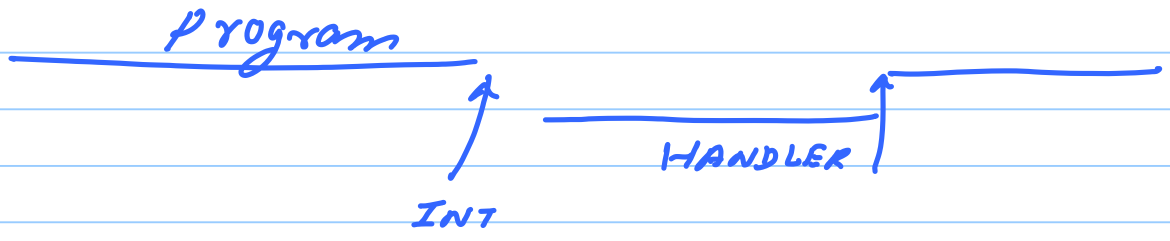
3) For the faulting instruction $\rightarrow$

allow it to go to the end of the

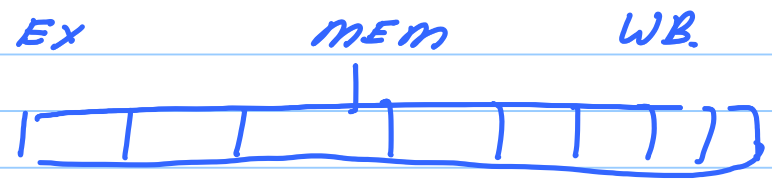
pipeline. → (No side effects like writing to mem. & registers)

4) Once the faulting instruction reaches the end of the pipeline, invoke the interrupt handler. (save registers and next PC)

5) Once the interrupt handler completes, restart program from next PC (restore the value of registers).



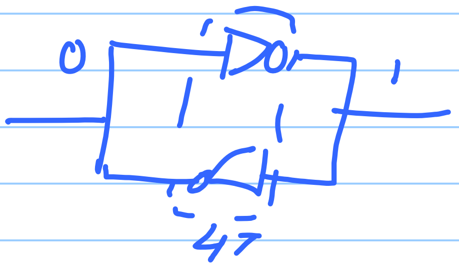
Memory System

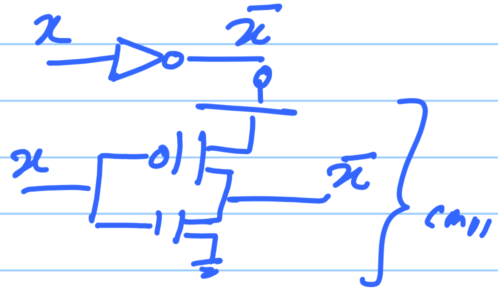


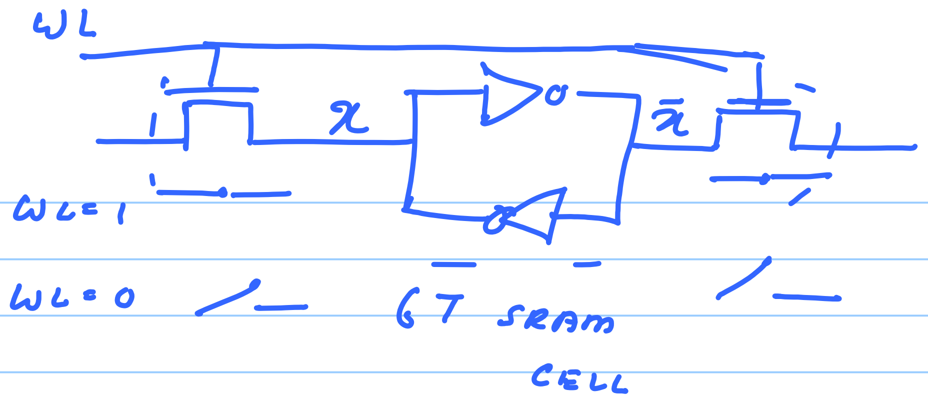
How do we build a memory?

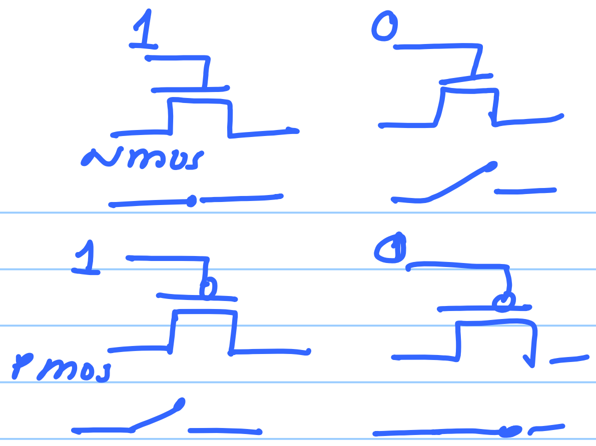
Latch / Flip-flop. (12 + transistors)
fast
Lot of area & power.

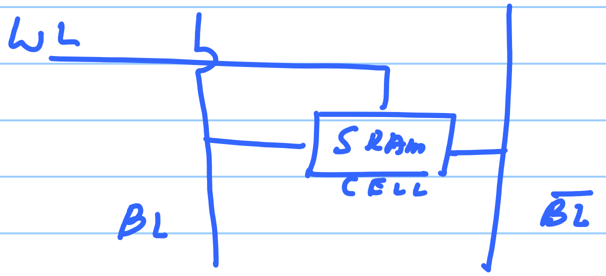
SRAM (Static RAM)

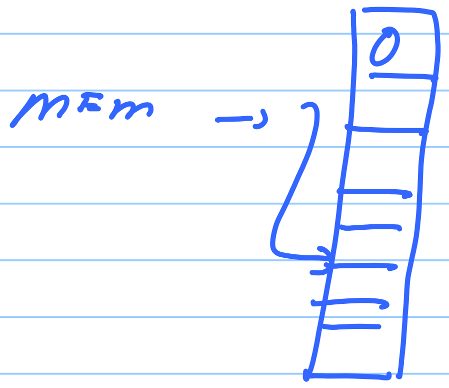




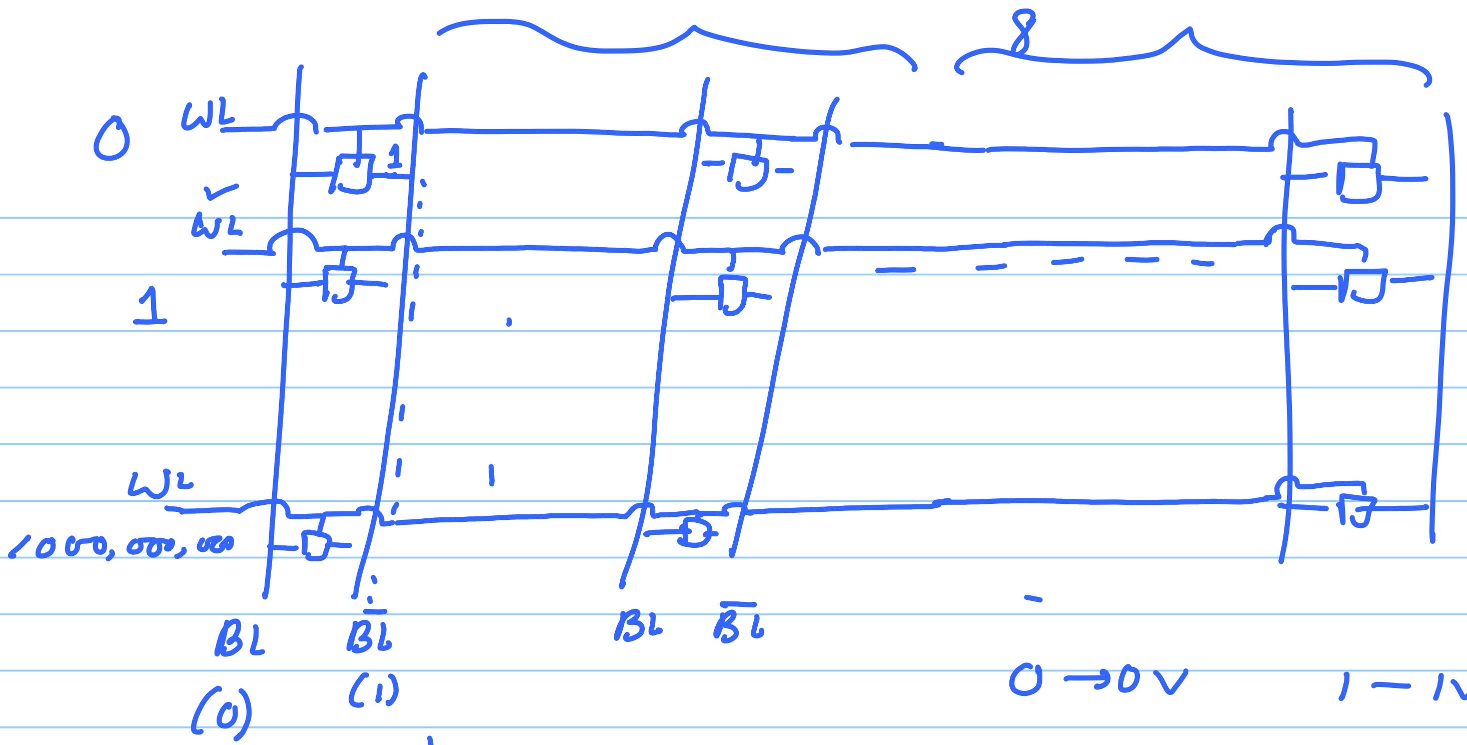

WL
1
x
0
$\bar{x}$
0
WL = 1
WL = 0
6T SRAM
CELL


1
0
Nmos
1
0
Pmos


WL
SRAM
CELL
BL
$\overline{BL}$


MEM
0

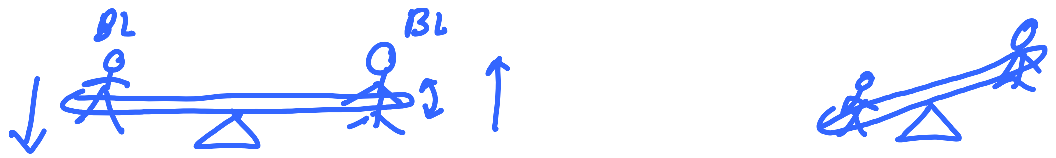
MEM

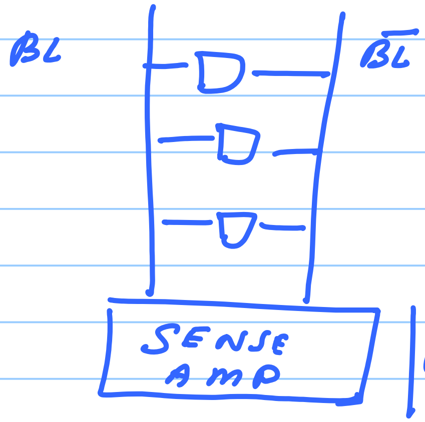


$0 \rightarrow 0V$ $1 - 1V$

1) Precharge BL and $\overline{BL}$ to 0.5 V

2) BL ↓ $\overline{BL}$ ↑



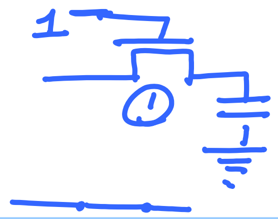


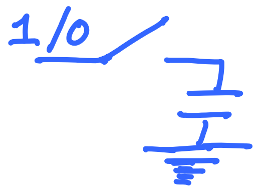
$$|(V(BL) - V(\overline{BL}))| > \delta$$

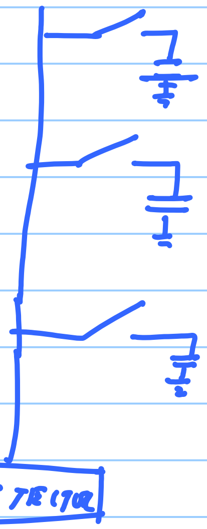
[50mV]

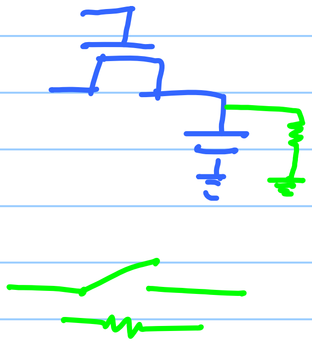
	# Trans/1 bit	Area	speed
Latch	12+	High	High
SRAM	6	Medium	medium
DRAM	1	Low	slow.

DRAM









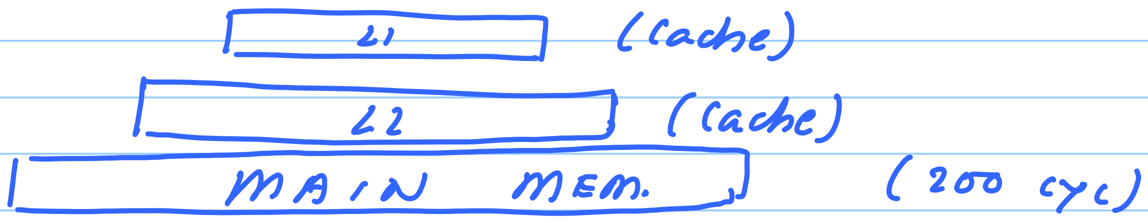
Periodic Refresh: Periodically read the value of every DRAM cell and write it back again.

This ensures that optimum amount of charge across the capacitor is maintained.

Three kinds of memories:

Latch	Pipeline/registers
SRAM (static)	Caches.
DRAM	Main Memory

<u>**MEM**</u>



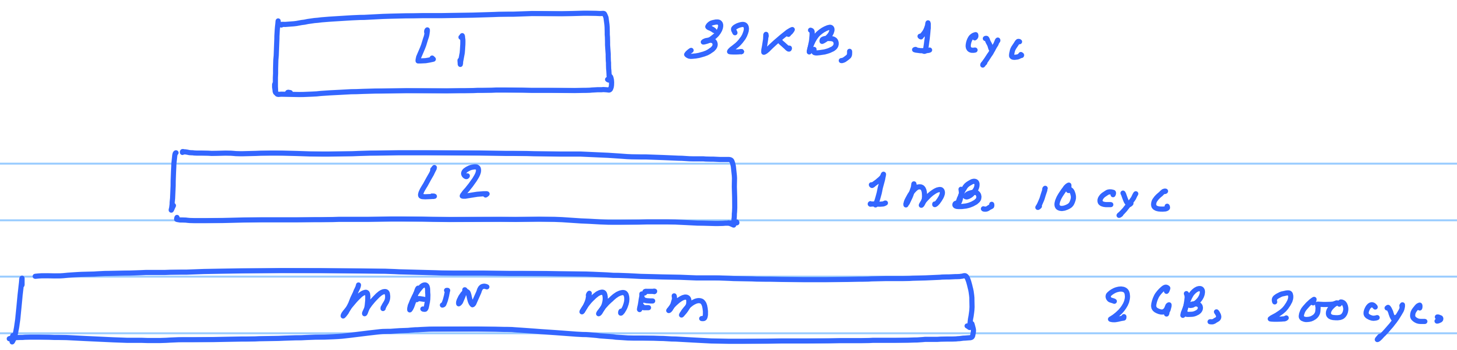
Data (main mem) contains everything.

Data (L1) $\subset$ Data (L2) $\subset$ Data (main mem)
(desk) (shelf) (library)

<u>Behavior</u>

: Temporal locality → tend to access the same piece of data over & over again

Spatial locality → programs tend to access nearby data



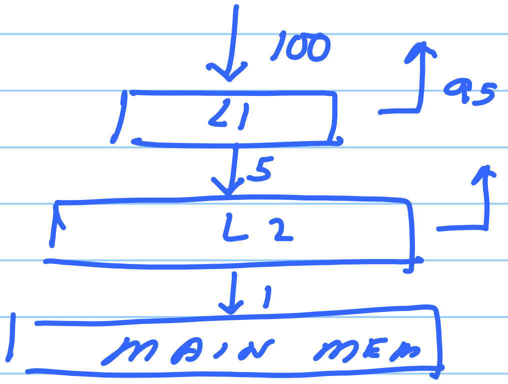
Thumb-Rule : 90% of the data is accessed 10% of time
10% of data " " 90% " "

Hit → Data present in cache
Miss → Not present.

Hit Rate: % of hits.

global hit/miss rate: $\frac{\text{\# of hits/misses}}{\text{\# of memory accesses issued by proc.}}$

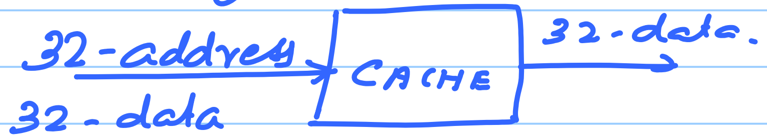
local hit/miss rate: $\frac{\text{\# of hits/misses.}}{\text{\# of accesses to that cache}}$



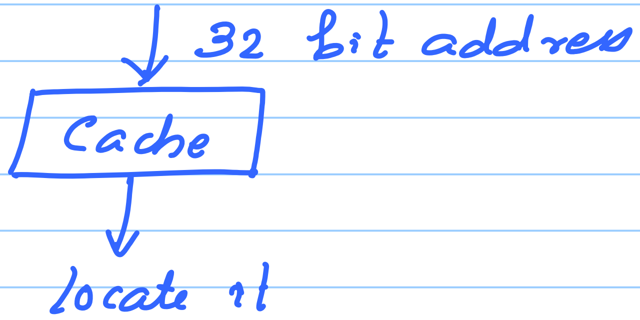
L1 global miss rate: $\frac{5}{100}$ L2 global miss rate: $\frac{1}{100}$

local " " : $\frac{5}{100}$ L2 local " " : $\frac{1}{5}$

How to design a cache:

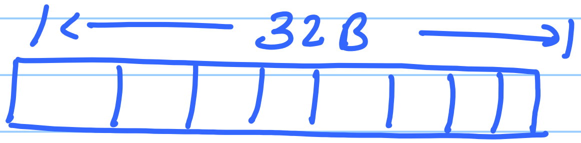


Basic Problem:

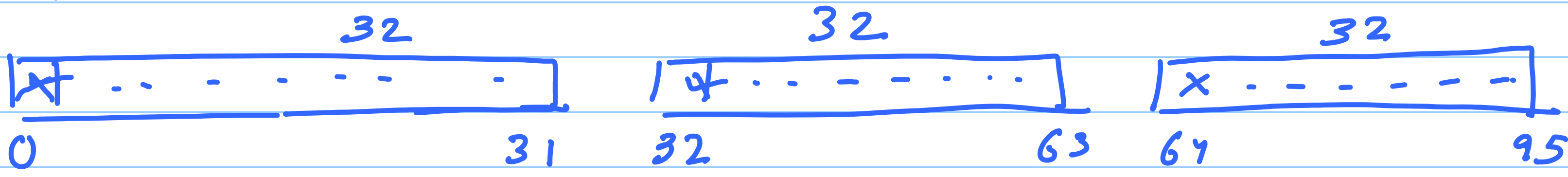


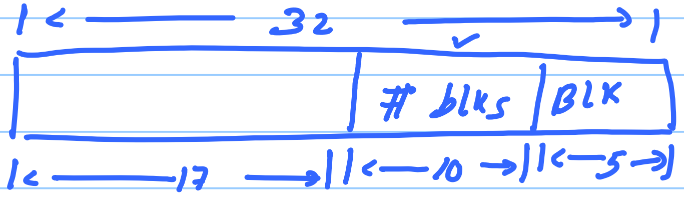
Temporal locality → Multi-level cache

Spatial locality → Create large blocks / line
(32B → 128B)



A[96];





Cache size: 32KB
Block size: 32B

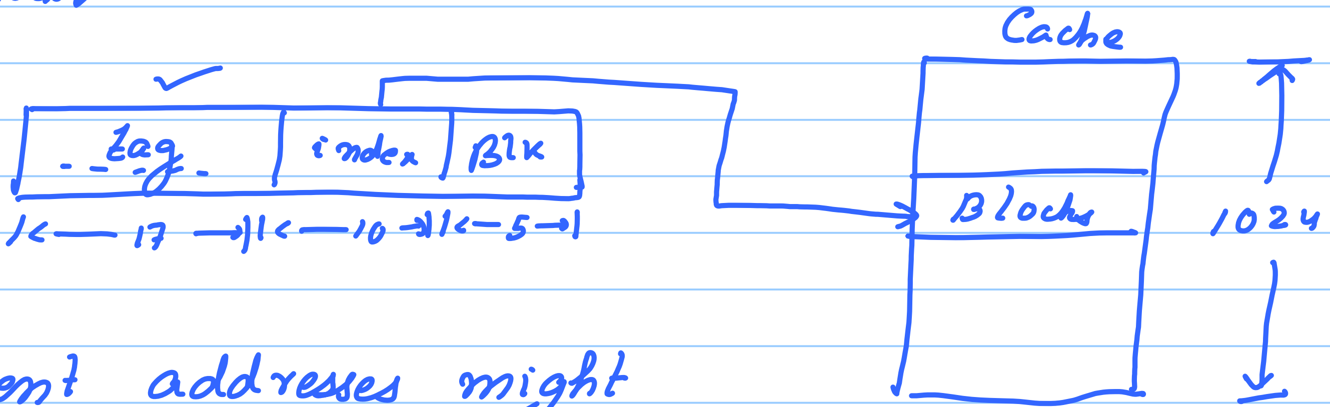
Blocks: 1024

Cache access algorithm:

address A.

$A' = A >> (\text{Blk Size Bits})$

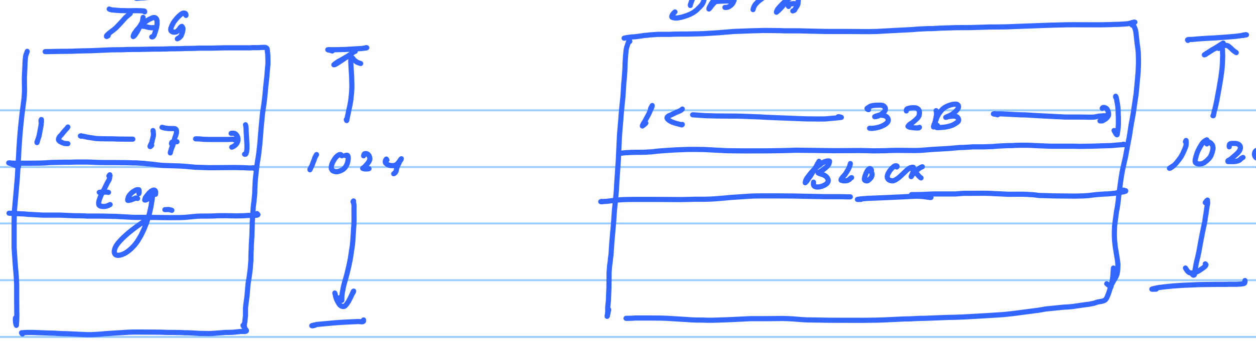
Cache (line/block) number = $A' \,\%\, (\#\text{ blocks})$



Two different addresses might have the same index

[collision]

Cache



Access protocol: address (A)

1) A' = A >> (Blk Size Bits)

2) Index = A' % (# Blocks) ✓

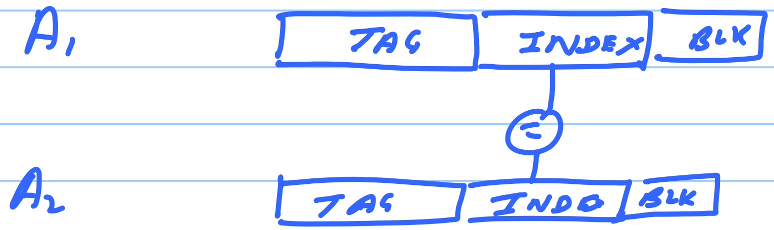
3) tag = A' >> lg(# Blocks)

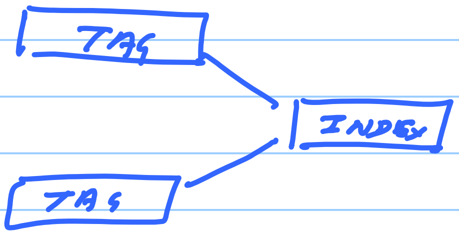
4) if (tag == TAG_ARRAY[index])
   declare hit;

```
            return DATA_ARRAY[index];
    else
        declare miss;
```

+ easy access protocol.
+ simple hashing scheme
− high chance of collisions.

Collision:





Reduce collisions:
Even if the index matches, we do not want a miss.

Reason for a collision:
Every address maps to a single cache line.

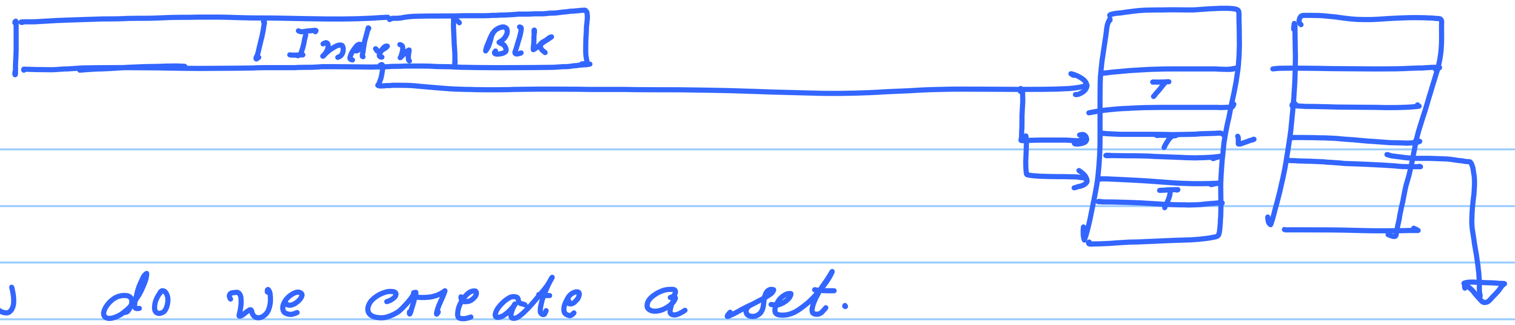
Consider a set of lines: An address can be saved anywhere in a set.

New Algo:

1) $A' = A >> \log(\text{Block\_Size})$

2) $\text{Set Index} = A' \% (\# \text{sets})$

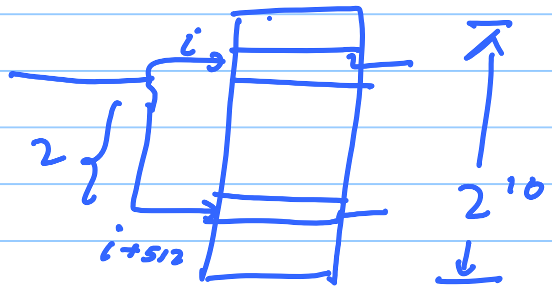
```
3)  tag = computeTag (A')

4)  For. each (index ∈ Sets [setIndex])

        if (tag == TAG-ARRAY [index])
            declare HIT;
            return DATA-ARRAY [index];

        else
            continue;

5)  declare miss
```



How do we create a set.

|Set| = 1

Tag.	Index	Blk
17	10	5



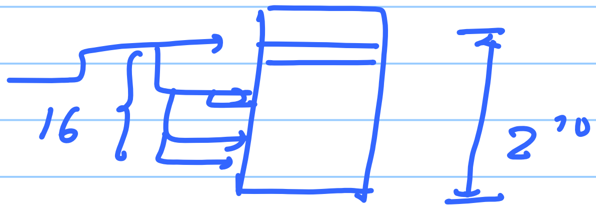
|Set| = 2

Tag	Index	Blk
18	9	5

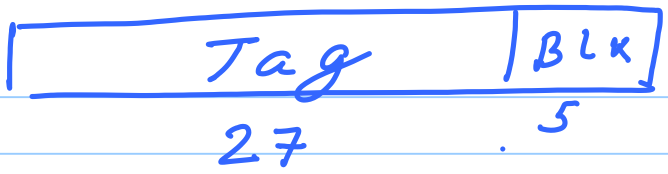
⋮

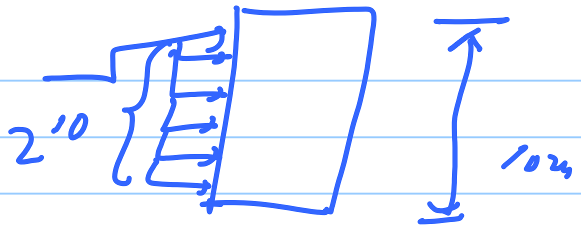
|Set| =16

Tag	Index	Blk
21	6	5



|set| = 1024

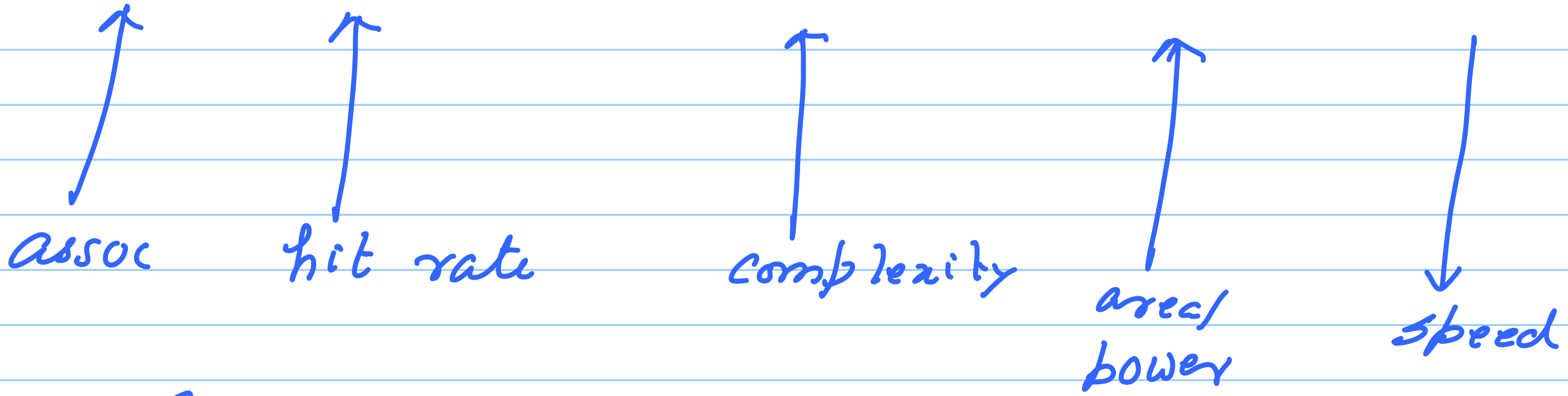


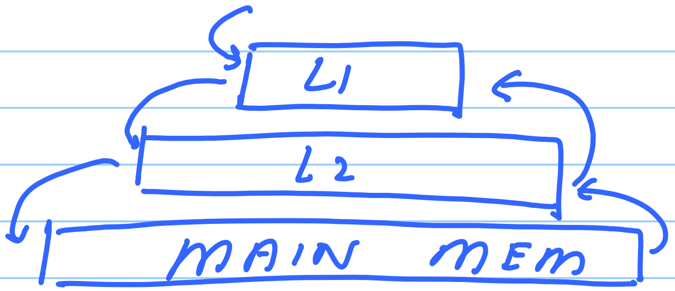


Terminology:

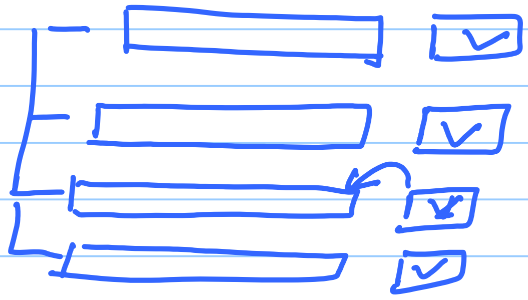
|set| = 1	|set| = k	|set| = # blks
Direct Mapped Cache	k-way set associative cache	fully associative cache.

Tradeoffs.





Set –



V (valid bit) → specifies if the line is valid.

If all the lines in a set are valid, one of them needs to be thrown out (evicted) after a read miss.

$$\begin{tikzcd}
 & \text{after a read miss.} \arrow[dl] \arrow[dr, "\text{LRU}"] & \\
\text{FIFO} & & \text{Least Recently Used}
\end{tikzcd}$$

FIFO

first in first out.

LRU

Least Recently Used

old $\qquad\qquad\qquad\qquad\qquad\qquad$ new.

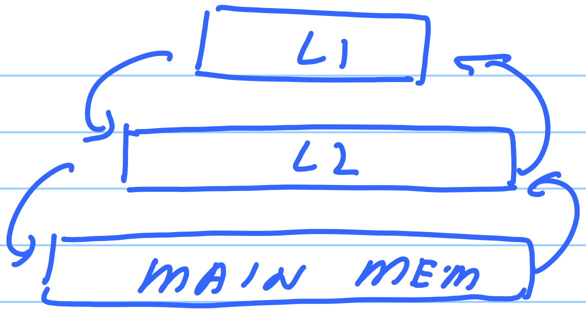
$\underline{1}$ 3 2 4 1 2 $\underline{4}$ 3 $\underline{2}$ $\underline{1}$ $\underline{3}$

FIFO Replacement scheme: $\boxed{1}$

LRU " " : $\boxed{4}$

$\Big\}$ evictions.

Writes:

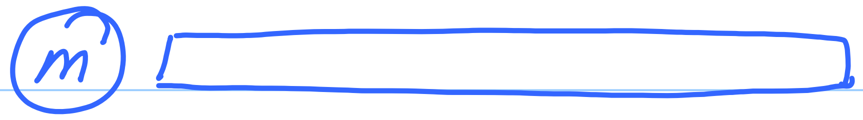


Write Policies:

Write Through: WT
whenever you write to level L_n, you also write to level L_{n+1}

Write-Back: WB
Only write upon an eviction.

Modified bit (m).



If the modified bit is set, write the data to the lower level

	WT		WB
+	write logic is simple	−	complex
−	more writes	+	less writes
+	eviction is cheap	−	eviction is expensive

Misses ↳ { Compulsory

Conflict

Capacity

Fix:
Intelligent compiler/
HW
pre-fetching

increase
associativity

increase cache
size.