~~tag + valid = 18+2 bits~~

~~∴ total =~~

## Fully associative mapping

If the cache has n blocks, data add can be mapped to any one the block.

Use comparator to check tag bit of each block to that of address of data needed

⇒ (mem block no) % (No. of block in cache)

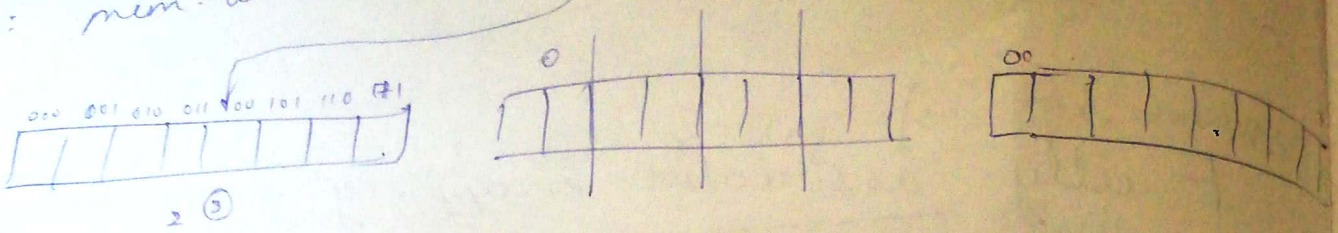
## Set associative

n-way set associative ⇒ a set has n block

⇒ first we need to identify the set & then use tag to find block within that set.

⇒ $\left(\begin{matrix}\text{mm}\\ \text{block}\\ \text{no}\end{matrix}\right) \% \left(\begin{matrix}\text{no. of sets}\\ \text{in cache}\end{matrix}\right)$

eg: mem. add = 12 = 1100



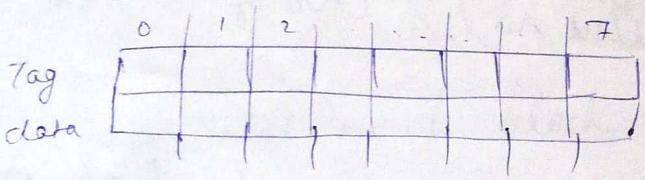

direct map

2-way set associative

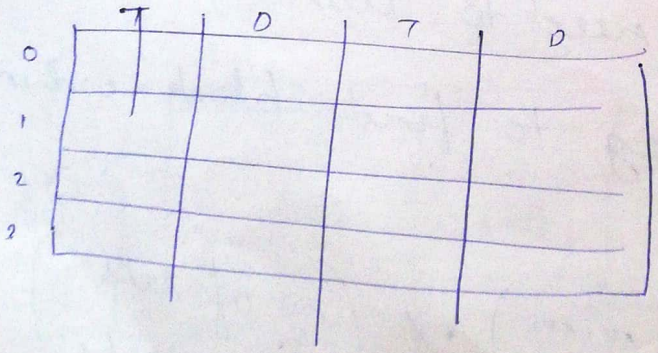
no. of set = 8/2 = 4

∴ 12 % 4 = 0

1-way set ass. with 8 block.




2-way — 8 block ⇒ 4 set




There r 3 cache each with 4 1 word block.

1 is fully assoative

2 is 2 way ass

3 is directed

find no. of misses for each cache for

given seq of block address

0, 8, 0, 6, 8

In 2 way set, if a set [x|y] has 2 data & another data is mapped to it, by least recent principle, the least recently used is replaced (eg: if x is placed after y, y is replaced)

| Address block | hit/miss | Contents of cache block 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|
| 0 | M | m[0] | | | |
| 8 | M | m[8] | update | | |
| 0 | M | m[0] | | | |
| 6 | M | ~~M[2]~~ m[6] | | m[6] | |
| 8 | M | m[8] | update | | |

no of block → for direct mapping

in direct 0 % 4 = 0

8 % 4 = 0

6 % 4 = 2

⇒ for 2 ways – set assoative

| add | hit/miss | set 0 | set 0 | set 1 | set 1 |
|---|---|---|---|---|---|
| 0 | M | m[0] | | | |
| 8 | M | | m[8] | | |
| 0 | H | m[0] | | | |
| 6 | M | | m[6] | | |
| 8 | M | m[8] | | | |

→ replace least recently used
" m[0]
" m[0]

no. of set = 4/2 = 2
(no of block / no. of set)

0 % 2 = 0 ← set
8 % 2 = 0
6 % 2 = 0

→ fully assoiative in a set

place in any block of set 0

direct mapped about set

⇒ for fully assoative

| add | H/M | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|
| 0 | M | m[0] | | | |
| 8 | M | | m[8] | | |
| 0 | H | m[0] | | | |
| 6 | M | | | m[6] | |
| 8 | H | | m[8] | | |

We have to store in sucha way to minimise the misses if they r free block store if there

✗ 8 block ϕ 16 block
no. of set = 4 8

## Handling Cache misses

If an instruction is checked, if its a miss in cache, we check the address of that instruction ( = PC - 4 ) & chick in 1° mem.

already incr by 4
we need prev instr add

## Handling Cache writes

After changing value of say variable n, if we change only the value of n in cache (copied from its original loc in 1° mem during fetching) the cache will have modified value & 1° mem will have old value causing an inconsistancy.

To avoid this, we can use

(i) Write through:

" processor writes to cache & at the same time it writes to memory. takes more time
This causes more delay to processor

(ii) Buffered write

processor writes to buffer instead of writing to i' mem (reducing the time to write to i' mem by processor as buff is faster) & mem writes / updates itself from buffer

Now if buffer is full, it creates a stall

we can avoid this by:

(i) increase buffer size

(ii) keep multiple queues/ buffer

(iii) Write Back / copy back

Update i' mem only if a replacement occurs to corresponding add in' caches

or ~~write~~ copy back ~~add~~ data in cache when power is offed

---

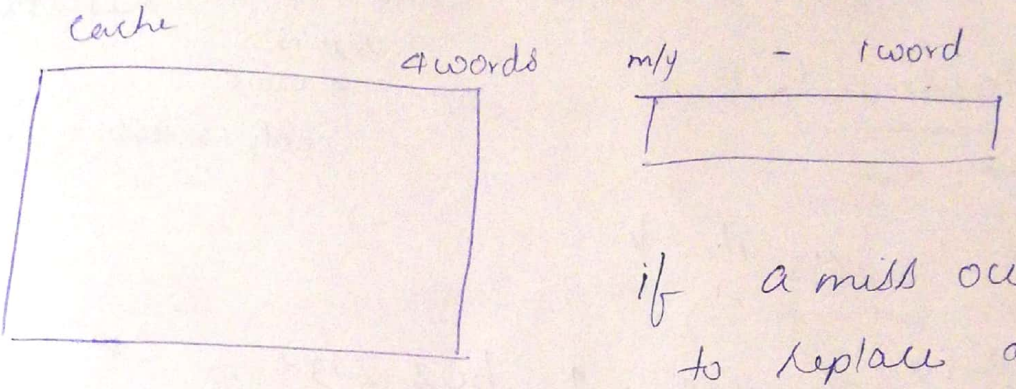
DESIGNING M/Y SYSTEMS

Q. Assume 1 mem bus clk cycle is needed to send address. 15 mem bus clk cycles for each d-ram access initiated

i' mem

1 mem bus clk cycle to send a word of data. If we have a cache block of 4 words & 1 word ~~byte~~ wide bank of d-rams. Calculate miss penalty in terms of mem. bus clk cycle?

A:




if a miss occurs, we have to replace all the 4 words in cache (not that 1 word alone)

if miss happens

- send add to lower mem → 1
- → access 4 words in 1' mem — 15 x 4
- → copy back data → 1 x 4 (word / data bus)

} = 65

if we can access the 4 words seq. if we provide only the starting address

if we have to provide add. of each of 4 word in cache, then

4 x 1 + 15 x 4 + 1 x 4 = 68

Now, if we ↓ cache size, miss penality ↑ but miss rate ↑.

but if we ↑ size of 1° mem (instead of ↓ cache size), then

miss penality = 1 + 15 + 4 × 1 = 20

- 1: d-ram size is 4 word ∴ only 1 access
- 15
- 4 × 1: bus = 1 byte

i it ↓

Now if we ↑ bus size,

miss penality = 1 + 15 + 1 = 17

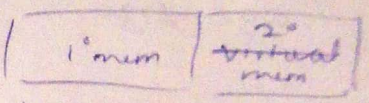
- 1: send one add
- 15: only 1 access
- 1: only 1 access to return data

- ↑ cache size ⟶ ↓ miss rate
- ↓ " ⟶ ↓ miss penality
- ↑ bus ⟶ ↓ miss penality
- ↑ 1° mem size ⟶ ↓ miss penality

# Virtual memory

If RAM is having less size miss rate can be higher along with miss penality

Hence, we attach a part of 2° mem (virtual) to 1° mem so that processor has a larger view. This is called virtual mem



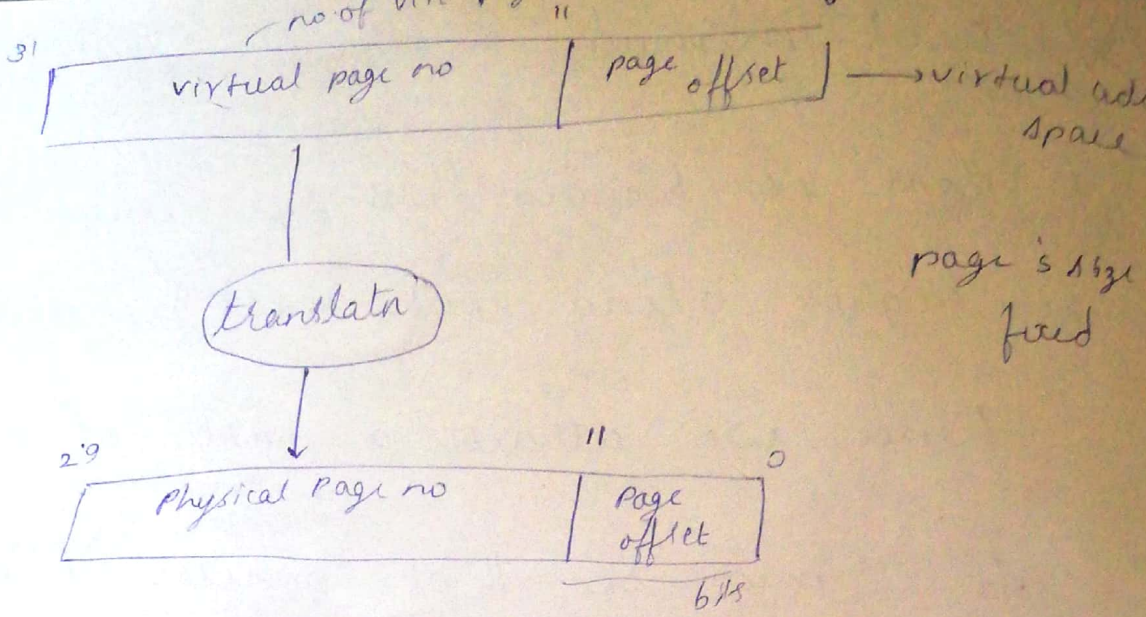

Also, virtual mem allows (virtual mem) sharing of prgms

page: block is called a page

Pagefault: miss is called pagefault

virtual add: add corresponding to virtual space

Address mapping / translation: translation of virtual add to a physical address (1° mem add) using s/w & h/w




page is size is fixed

Page size = $2^{12}$ = 4kB

No. of pages in physical/1° mem = $2^{18}$

No. of pages in virtual mem = $2^{20}$ - large

block

Now, we need to know whether a page is present in 1° mem / 2° mem of virtual mem

∴ we need a page table with each cell having :

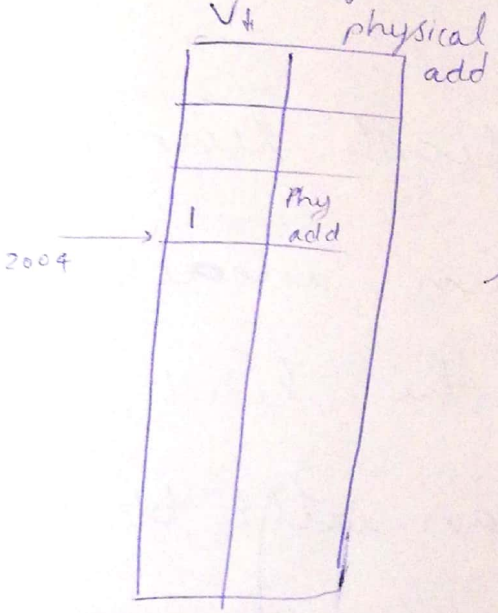
(i) valid bit V=0 ⇒ page fault & page in 2°

V=1 ⇒ page in 1° mem

Once we know the virtual mem add, we use it as an index to search the

page table. If v = 1, the ~~add~~ phy. add. corresponding to it shows page is in 1° mem

if v = 0, ⇒ page in 2° mem, then we use phy. add. adjacent to that index to search 2° mem & update page table by moving data to 1° mem from 2° mem.




index = virtual page no

∴ no. of index in page table

$= 2^{20}$

each prgm has its own page table & page table reg. store the starting add of page table.

⇒ Another type of mapping is called segmentation

⇒ Now, when we have to replace some value in 1° mem when a page fault occur in page table, we use copy back mechanism. (value in 2° mem moved to add. in 1° mem & the value at add n is moved to a space called swap space in 2° mem.

⇒ & we replace the least recently used data from 1° mem incase of a page fault. To know the least recently used block, we keep an extra bit called reference bit R

⇒ When a data is used we set R = 1 but after a time stamp (eg 6ms) its value is cleared saying that it is not a recently used data.

In case of cache, we need to copy back incase of replacement of if data in cache was updated. keep dirty bit = 1 if cache was updated recently & hence we have to copy back

data in cache ≠ data in corresponding 1° mem

~~on~~ before replacing the value in cache in case of miss (if dirty bit = 0, no copy back just replace)

III^rly in case of virtual mem. space in page table dirty bit is set as 1 if data in that table is changed. If it is 1, the we have to copy back the data in that all to corresponding byte in 2° mem before replacing (swaping

Rmk

Page table resides in physical mem.

# Translation look aside buffer.

(TLB)

→ page table in cache

In 1° mem.

| V | D | R | Phy add. |
|---|---|---|---|
| 1 | 0 | 1 | |
| 1 | 0 | 0 | |
| 0 | 0 | 0 | |
| 0 | 1 | 1 | |
| 1 | 0 | 0 | |

We keep cache. to get faster access to page table. Hence we keep only the entries of the recently used index in cache.

∴ TLB

| V | D | R | Phy addr |
|---|---|---|---|
| V | D | R | |

→ recently used index in page table of 1°

Cache TLB can store may be max of 1 or 2 entries. If cache bi miss occurs in this case we replace the recently used ~~if the 2 index is passed to~~

TLB miss ⇒ entry miss not that its a page fault. that may be available in page table. Access page table modify if its page fault & then replace the least recently used entry in TLB

{ 1st TLB if miss goto page table update TLB after modifying page table in case of page fault }