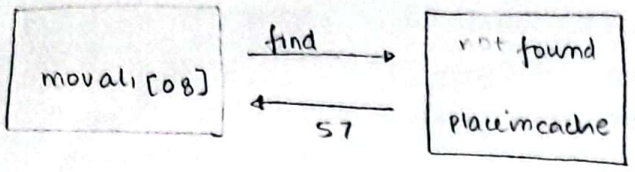
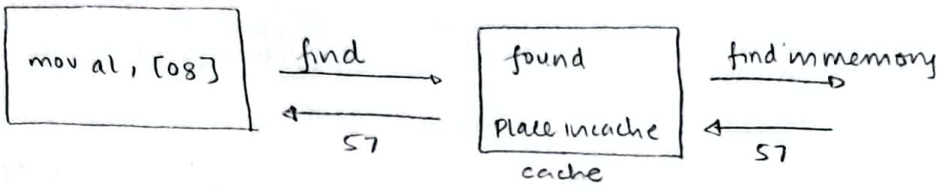
MEMORY FILLING

- a single box represents 1 byte.
- dw is 2 byte so 2 boxes filled, hence in 5566h, 66 first stored and then 55.
- if multiple bytes have string or array, then we go from left to right, so 'abcd',0 → 61,62,63,64,$
- array stored left to right so 11h,22h,30h is stored in the same way.
- for array in word we need 2 bytes. so store in 4 places i.e. 50h, 60h, 70h, 80h will be stored as 0050, 0060, 0070, 0080 from left to right, but each individual element stored in little endian. i.e 50,00, 60,00, 70,00, 80,00
- for word datatype and duplicate is used so first utilize 2 bytes i.e for 3DUP(55h), 56h, 57h it is an array so it will be stored as 55,00, 55,00, 55,00, 56,00, 57,00
- for a negative number in word, take its hexa value and convert it in 2's complement, and then write it as 4 ~~bits~~ 2 bytes. so -22 → -16h → after 2s complement → FFEA so stored in memory will be EA,FF
- for double word take 4 bytes and representation in 8 numbers so 654321ABh will be stored as AB,21,43,65
- LABEL not stored in memory.
- for array in doubleword or dword, each element represented in 8 numbers. so 1,2,3 will be stored as 00000001, 00000002, 00000003 but in little endian so order will be 01,00,00,00, 02,00,00,00, 03,00,00,00

CACHE HIT



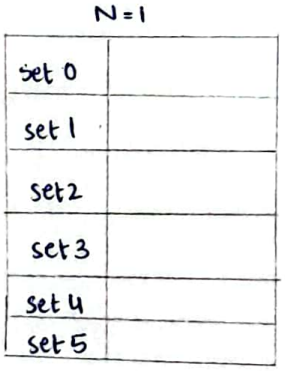

CACHE MISS

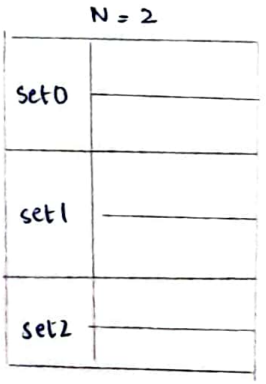


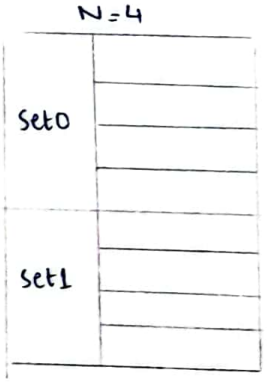

$$\text{miss rate} = \frac{\text{number of misses}}{\text{number of total memory access}}$$

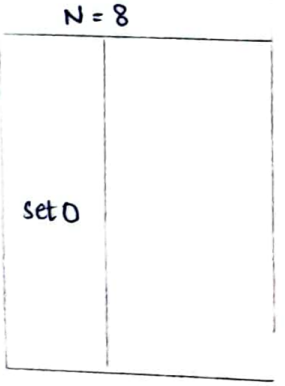
- basic unit of data transfer between memory and cache is called cache block
- size is fixed.
- capacity : number of data bytes in cache → $2^n$
- blocksize : bytes of data brought into cache at once
- number of block : $(B = C/b)$
- degree of associativity (N) : number of blocks in a set
- number of sets $(S = B/N)$ : each memory address maps to exactly one cache set.

* offset bits = $\log_2 b$
* set bits = $\log_2 S$
* address = tag bits − set bits − offset bits




example :

b = 32 bits = 4B
C = 64 bits = 8B
B = 8/4 = 2
S = 2/1 = 2
$\log_2 b = \log_2 4$
offset = 2
$\log_2 S = \log_2 2$
set = 1

| | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 0 | 57 | 98 | 65 | 22 |
| 1 | | | | |

mov al, [09]
* convert 09 into binary.
00001001
offset bits.
* so in al, 98 is stored/moved

| | | |
|---|---|---|
| 0 | | |
| 1 | | |
| 09 | 08 | 57 |
| 2 | 09 | 98 |
| | 0A | 65 |
| | 0B | 22 |

example:

b=4B
C=32B
N=1
B=32/4=8
S=8/1=8

offset bits = $\log_2 4 = 2$
set bits = $\log_2 8 = 3$
address = 8 bits.
tag bits = address - setbits - offsetbits
= 8 - 3 - 2 = 3

1 → 01
2 → 10
3 → 11
4 → 100

| | setbits | V | tagbits | offset bits 00 | 01 | 10 | 11 |
|---|---|---|---|---|---|---|---|
| set 0 | 000 | 1 | 001 000 | B3 AB | 3C 12 | 3F 34 | 3D CD |
| set 1 | 001 | 1 | 001 00-0 | 3C EF | 9F 56 | 77 45 | 89 CD |
| set 2 | 010 | 1 | 000 | 00 | 12 | 35 | 45 |
| set 3 | 011 | 0 | | | | | |
| set 4 | 100 | 0 | | | | | |
| set 5 | 101 | 0 | | | | | |
| set 6 | 110 | | | | | | |
| set 7 | 111 | | | | | | |

1) mov al, [00H] = AB
00h → 000 000 00 cachemiss
2) mov ah, [06H] = 45
06 → 000 001 10 cachemiss
3) mov bl, [0BH] = 45
0B → 000 010 11 cachemiss
4) mov bh, [02H] = 34
02 → 000 000 10 cachehit
5) mov cl, [20H] = B3
20 → 001 000 00 cachemiss
6) mov ch, [25h] = 9F
25 → 001 001 01 cachemiss

example:

b=1B B=4
C = B x b = 4
S = B/N = 4/4 = 1

offset bits = $\log_2 b = \log_2 1 = 0$
set bits = $\log_2 S = \log_2 1 = 0$
tagbits = 8 - 0 - 0 = 8

| set | V | LRU | Tags | Data |
|---|---|---|---|---|
| | 1 | 00 01 10 00 01 10 | 0000 0110 | 06 |
| | 1 | 00 01 10 11 00 | 0010 0011 1110 1111 | 23 EF |
| | 1 | 00 01 10 11 | 1011 0010 | B2 |
| | 1 | 00 01 | 01011001 | 59 |

1. 0x06 → 0000 0110 cachemiss
2. 0xEF → 1110 1111 cachemiss.
3. 0xB2 → 1011 0010 cachemiss.
4. 0x06 → 0000 0110 cachehit
5. 0x59 → 0101 1001 cachemiss.
*the capacity of cache is full, so now the incoming number will replace the [11] row that is the most used
6. 0x23 → 0010 0011 cachemiss

set associative mapping is a combination of both the direct and associative cache

disadvantage of direct mapping: conflict misses are high and hit rate is low.

disadvantage of associative mapping: comparision time for searching a specific block is increased, also compare the tag bits of the search block with every line tag.

example:

b = 2B — offset bits = $\log_2 2 = 1$

C = 16B — set bits = $\log_2 4 = 2$

B = 16/2 = 8 — address = 8

S = B/N = 8/2 = 4 — tagbits = 8 - 2 - 1 = 5

| sets | V | LRU | Tag bits | offset bits 0 | offset bits 1 |
|---|---|---|---|---|---|
| 00 | | | | | |
| 01 | | | | | |
| 10 | | | | | |
| 11 | 1 | 0 | 00011 00000 / 11111 | 06 ~~fe~~ ff | 07 ~~ff~~ ff |
| | 1 | 0 | 10010 | 96 | 97 |

1) 0xFF → 11111 11 1 cachemiss

2) 0x96 → 10010 11 0 cachemiss

3) 0x1E → 00011 11 0 cachemiss

4) 0x97 → 10010 11 1 cachehit

5) 0x07 → 00000 11 1 cachemiss

example:

b = 4B — offset bits = $\log_2 4 = 2$

C = 32B — set bits = $\log_2 2 = 1$

B = 32/4 = 8 — address = 8

S = 8/4 = 2 — tagbits = 8 - 1 - 2 = 5

| | V | LRU | Tagbits | offset bits 00 | 01 | 10 | 11 |
|---|---|---|---|---|---|---|---|
| set 0 | | 00 | 0 | | | | |
| set 1 | 1 | 00 01 / 11 10 00 | 00111 / 00011 | 12 / 27 | 39 / 3F | 75 / 3E | A6 / 6C |
| | 1 | 00 01 / 11 00 | 00110 | 56 | BD | CD | EF |
| | 1 | 00 01 / 11 | 00101 | DE | DF | FD | A6 |
| | 1 | 00 01 / 10 | 00100 | 3E | 9F | 77 | 89 |

1) 0x3D → 00111 1 01 cachemiss

2) 0x35 → 00110 1 01 cachemiss

3) 0x2E → 00101 1 10 cachemiss

4) 0x25 → 00100 1 01 cachemiss

5) 0x1D → 00011 1 01 cachemiss

6) 0x37 → 00110 1 11 cachehit

cache writes

1. hit policy : two methods if data is already present.
   write through and write back.

2. miss policy : two methods if data is not present in the cache
   write allocate, write around (no allocate)

**write through :**

- update cache as well as memory.
- main memory always has the most current copy of data.
- write is slower, every write needs a main memory access.

example → set bits = 3, tagbits = 5, offset bits = 0, address bits = 8, direct mapped

mov [A4], 68

A4 → 10100 100

- assuming data is found so cachehit.
- update both cache and memory with 68

mov al, [0C]

0C → 00001 100

- tagbits not found so update tagbits in cache
- update data in cache as well.

**write back :**

- update cache only
- if tagbits don't match after updating data in cache only before replacing cache block we will write back in main memory.

example → mov [A4], 68

A4 → 10100 100

- tagbits match so replace cache data with 68
- dirty bit will turn 1 as soon as data placed in cache.

mov AL, [0C]

0C → 00001 100

- cachemiss, so we check dirty bit
- dirty bit is 1, so first update A4 memory with 68
  then update cache with new value i.e 99 and change tagbits.
- turn dirty bit with 0.

## Normalization

Transitive: if $X \to Y$ and $Y \to Z$, then $X \to Z$

1st normal form: a table should not contain any multi valued attribute.

| RollNo | Name | Course |
|---|---|---|
| 1 | tab | c/c++ |
| 2 | ash | Java |
| 3 | noor | DBMS |

1NF →

| RollNo | Name | Course |
|---|---|---|
| 1 | tab | c/... |
| 1 | tab | ...c++ |
| 2 | ash | java |
| 3 | noor | DBMS |

∴ primary key is composite that means {RollNo, Course}

2nd normal form: table must be in 1NF and all non-key attributes should be fully functional dependent, hence no partial dependencies.

| CustomerID | StoreID | Location |
|---|---|---|
| 1 | 1 | Delhi |
| 1 | 3 | Mumbai |
| 2 | 1 | Delhi |
| 3 | 2 | Banglore |
| 4 | 3 | Mumbai |

2NF →

| CustomerID | StoreID |
|---|---|
| 1 | 1 |
| 1 | 3 |
| 2 | 1 |
| 3 | 2 |
| 4 | 3 |

| StoreID | Loca... |
|---|---|
| 1 | Delhi |
| 2 | Bangl... |
| 3 | Mumb... |

* these are two different table as now each attribute is dependent on its candidate key for each table, because before it was dependent on part of key.

3rd normal form: table must be in 2NF and there should be no transitive dependency.

| RollNo | State | City |
|---|---|---|
| 1 | punjab | mohali |
| 2 | haryana | ambala |
| 3 | punjab | mohali |
| 4 | haryana | ambala |
| 5 | bihar | patna |

* primary key is {RollNo}
* FD1: RollNo → State
  FD2: State → City (determined by non-key)

↓ 3NF

| RollNo | State |
|---|---|
| 1 | punjab |
| 2 | haryana |
| 3 | punjab |
| 4 | haryana |
| 5 | bihar |

| State | City |
|---|---|
| punjab | mohali |
| haryana | ambala |
| bihar | patna |

BCNF: table should be in 3NF and every right hand side attribute of the functional dependency should depend on the super key of that table.

4NF: it should be in BCNF and should not have any multi-valued dependency.

5NF: it should be in 4NF and cannot be further nonloss decomposed.

* if an attribute is fully as well as partially dependent then also in 2NF a new table is created for partial dependency.

**allocate**

ng data in cache.

mple → mov [0C], 68

0C → 00001 100

- when memory address is on left, then we write
- check tagbits, if cachemiss then bring data from main memory to cache. and updati tagbits

**write around**

- updati datablock in memory.

example → mov [0C], 68

- check tagbits, if cachemiss then updati value of 0C in main memory only.

load (LD) : instruction is used for reading from cache/memory.

store (ST) : instruction is used for writing to cache/memory.

example → b = 4B
C = 16B
address = 8 bits
B = 16/4 = 4
S = B/N = 4/2 = 2

offset bits = $\log_2 b = \log_2 4 = 2$
set bits = $\log_2 S = \log_2 2 = 1$
tag bits = 8 - 1 - 2 = 5

* hits → write back
* miss → write allocate

| set | tag bits | V | LRU | Dirty Bits | Data 00 | 01 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|
| 0 | 00010 | 1 | 0 | 0 | | | | |
| | 01000 | 1 | 1 | 0 | | | | |
| 1 | 00111 | 1 | 1 | 0 | | | | |
| | 00010 | 1 | 0 | 0 | 02 02 dd dd | 02 02 dd dd | 02 02 dd dd | 02 02 dd dd |

| Op | Address | Data (Hex) |
|---|---|---|
| LD | 28 | |
| LD | 44 | |
| ST | 52 | 01 01 01 01 |
| ST | 44 | 02 02 02 02 |
| LD | 24 | |
| ST | 60 | dd dd dd dd |
| ST | 64 | e0 e0 e0 e0 |
| LD | 20 | |
| ST | 16 | 11 11 11 11 |

28 → 00011 100 cachemiss
44 → 00101 100 cachemiss
52 → 00110 100 cachemiss
* 44 → 00101 100 cachehit
24 → 00011 000 cachemiss
60 → 00111 100 cachemiss
64 → 01000 000 cachemiss
20 → 00010 100 cachemiss
16 → 00010 000 cachemiss

* main memory will be updated once a new data comes, and save this before updating data.
* before bringing data to cache replace in memory.
* first replace data in main memory with 02 02 02 02

example : b = 4B
N = 1
S = 4

offset bits = $\log_2 4 = 2$
set bits = $\log_2 4 = 2$
tag bits = $8 - 2 - 2 = 4$

* hits : write through
* miss : write around

| set bits | tag bits | V | | 00 | 01 | 10 | 11 |
|---|---|---|---|---|---|---|---|
| 00 | 0001<br>0010 | 1 | | | | | ~~47~~<br>43 |
| 01 | | | | | | | |
| 10 | 0010 | 1 | | | | | 673 |
| 11 | 0001 | 1 | 00 | 00 | 00 | 00 | 81 |

| Op | Address (Bin) | Data (dec) | |
|---|---|---|---|
| LD | 00 01 11 00 | | cache miss |
| LD | 00 10 10 00 | | cache miss. |
| ST | 00 10 01 00 | 32 | → only updated memory, cache wasn't updated, cache miss. |
| LD | 00 01 00 00 | | cache miss |
| ST | 00 11 01 00 | 642 | → update memory, cache miss. |
| LD | 00 10 10 00 | | cache hit, reading to no changes. |
| LD | 00 01 11 00 | | cache hit, reading |
| ST | 00 10 01 00 | 124 | → update memory, cache miss. |
| LD | 00 10 00 00 | | cache miss. |
| LD | 00 10 10 00 | | → cache hit |

confidence

and

nciple of locality : tendency of a processor to access the same set of memory locations repetitively over a short period of time.

1. spatial locality : if a block of data is required by the processor, then there is a possibility that its neighbours will also be accessed by the processor in future.

2. Temporal locality. if a block of data is used by the processor, there is a large probability that it will be again accessed by the processor.
example → loop iterations, repeated function calls.