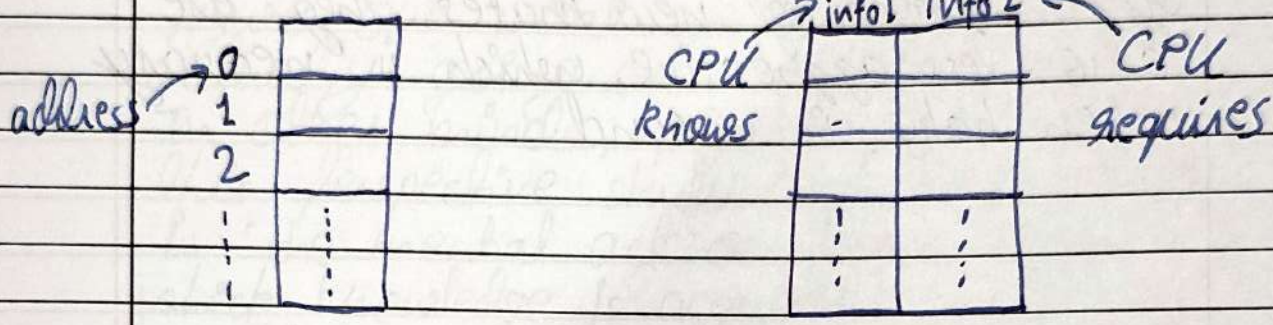
Mine
Date : / /20
Page :

Lecture 25

Associative Memory

- Known as Content Addressable Memory
- Cells do not have addresses.



Addressable Memory Associative Memory

To access the data, CPU will give some info to the memory and in return will get required data.
The required data is searched for in associative memory, unlike Addressable memory where address is directly accessed.

⇒ Each cell in associative memory has a matching logic, this allows each cell to be checked parallely and this memory hence is very fast. (faster than SRAM)

But this also makes associative memory very fast. expensive.

Mine
Date : / /20
Page :

Practically, we try to make all info 1 unique.
But if there is a conflict, the ~~one which occurs first sequentially is returned~~. all matches are sent sequentially (one after another).

Application of associative memory

i) can be used for cache implementation (for fully associative mapping)
ii) for TLB (Translation Lookaside Buffer) (TLB is studied in OS)

Locality of Reference

If CPU requests one address in memory, then that particular address or nearby addiessess will be accessed soon.
It was noticed that during operation of a computer, this statement is very probable to be true.

The quality of computer accessing the memory is called locality of reference
This is of two types
i) Temporal Locality of reference
(Based on time, same address accessed)
ii) Spatial Locality of ref.
(Based on space, nearby address accessed)

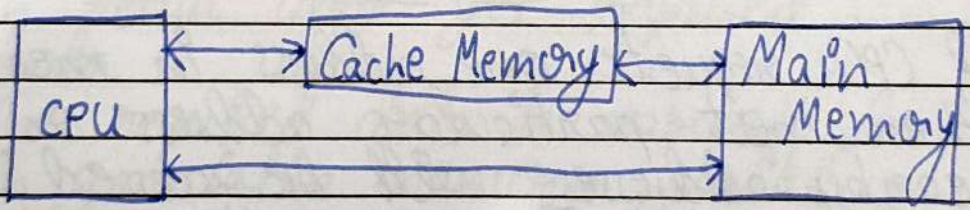
Mine
Date : / /20
Page :

Example of temporal is loop variable
Example of spatial is program instructions.

Cache Memory

Going by the concept of locality of reference, we keep the most accessed parts of main memory in a faster memory called cache memory.

The parts of memory which are most requested are called current demanded locality (or blocks).



Mine
Date : / /20
Page :

Lecture 26

Working of Cache Memory

Cache Hit - If CPU demanded content is present in cache
Cache Miss - Content not present in cache

* Performance is given by Cache Hit Ratio

$$\text{Hit Ratio} = \frac{\text{no of cache hits}}{\text{Total refrences}} = \frac{\text{no of hits}}{\text{no of hits + no of miss}}$$

Miss Ratio = 1 - Hit Ratio

⇒ If there is a cache miss, then when we are fetching that data from main memory, we also fetch a whole block of it into cache memory. (block which contains demanded byte is copied).

Average Memory Access Time

Average Memory Access Time (T_{avg}) = (Hit ratio x Hit time) + (Miss ratio x Miss Time)

T_{avg} = Hit ratio x Hit time + Miss ratio x Miss time

Hit time = Time ~~worst~~ taken for hit
Miss time = Time taken on miss

Mine
Date : / /20
Page :

Types of Cache Accesses

1. Simultaneous Access ⇒
Request for cache & main memory are generated simultaneously.
So CPU accesses them simultaneously and if cache has data then it can access it and request to main memory is discarded.

T_{avg} = Hit ratio × t_{cm} + Miss ratio × t_{mm}

t_{cm} = Time to access cache memory
t_{mm} = Time to access main memory

2. Hierarchical Access ⇒
The faster memory (cache memory) is accessed first and only on miss, main memory is accessed.

T_{avg} = Hit ratio × t_{cm} + Miss ratio × (t_{cm} + t_{mm})

t_{cm} = Time to access cache memory
t_{MM} = Time to access main memory

** Tip for which formula to use when,
If t_{mm} / t_{cm} not given use general formula.
By default assume simultaneous.
If level/hierarchy in ques ⇒ hierarchical access
Else ⇒ simultaneous access
If ques says to ignore initial checking time ⇒ Simultaneous

Mine
Date : / /20
Page :

~~Tavg when locality of~~

Lecture 27

Tavg When ~~locality~~ Locality of reference included

⇒ Simultaneous access ⇒
T_{avg} = Hit ratio × t_{cm} + Miss ratio × t_{block}
where, t_{block} = block size × t_{mm}

This is because when accessed simultaneously, on a miss, $t_{block} \gg t_{mm}$, hence the formulae

⇒ Hierarchical access ⇒
T_{avg} = Hit ratio × t_{cm} + Miss ratio × ($t_{cm} + t_{block}$)
Similar logic to previous allies.
Note how in both formulaes, only t_{mm} is replaced with t_{block}

Tavg when block transfer time is included

⇒ Simultaneous access ⇒
T_{avg} = Hit ratio × t_{cm} + Miss ratio × ($t_{block} + t_{cm}$)

⇒ Hierarchical access ⇒
T_{avg} = Hit ratio × t_{cm} + Miss ratio × ($t_{cm} + t_{block} + t_{cm}$)

Note: if ques mentions miss penalty, use block transfer time included formulae.

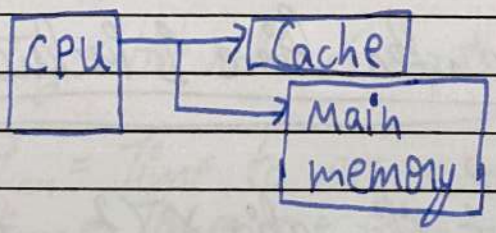
Mine
Date : / /20
Page :

Cache Write

As cache only keeps a copy of the block in main memory.
So when CPU writes same data in cache, block has to be updated in main memory then. This is called write propogation / cache write.

Write propogation
- Write through
- Write back

Write through ⇒ When CPU updates Cache contents, Main memory is also updated at the same time



⇒ Regardless of hit or miss, main memory is always accessed in this case on every write by CPU.

Mine
Date : / /20
Page :

T_{avg} in write through

T_{avg} read = Hit ratio $\times$ t_{cm} + Miss ratio $\times$ t_{mm} [simultaneous

T_{avg} = Hit ratio $\times$ t_{cm} + Miss ratio $\times$ ($t_{cm} + t_{mm}$) [hierarchi-cal

T_{avg} write = max (t_{cm}, t_{mm}) = t_{mm} mostly

T_{avg} = (fraction of read $\times$ T_{avg} read) + (fraction of write $\times$ T_{avg} write)

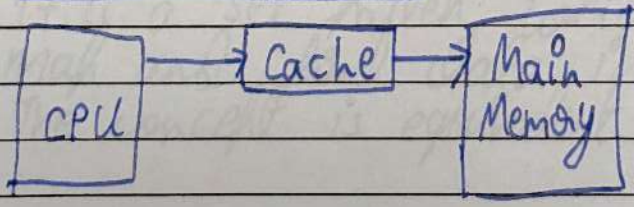
Effective hit ratio

% of time CPU's request is fulfilled only by cache memory. To find effective hit ratio, use T_{avg} formulae with both T_{write} and T_{read} average.

So basically

Effective H.R. $\times$ Hit cost + Effective M.R. $\times$ Miss cost = fraction of read $\times$ T_{avg} read + fraction of write $\times$ T_{avg} write

Write Back Cache



Mine
Date : / /20
Page :

When block in cache is being ~~reptaced~~ replaced, only then the old block is copied back to main memory.

But in this type of cache, if I/O device access the main memory, it won't get the correct data, which is in cache, this is called memory coherence problem.

T_{avg} in Write Back

⇒ Simultaneous :

$t_{avg\ read} / t_{avg\ write} = H.R \times t_{cm} + M.R \times (t_{block} + t_{write\ back})$

⇒ Hierarchial :

$t_{avg\ read} / t_{avg\ write} =$
Hit ratio $\times (t_{cm}) +$ Miss ratio $\times (t_{cm} + t_{block} + t_{write\ back})$

$t_{write\ back} = x \times t_{block}$

x = fraction of dirty block

Mine
Date : / /20
Page :

Lecture 28

Write Allocate vs No Write Allocate

Write Allocate ⇒ The block is loaded on a write miss (both read and write stall) followed by write-hit action.

No Write Allocate ⇒ The block is modified in main memory and not loaded into the cache.

Write Through
↓ uses
No write allocate

Write back
↓ uses
Write allocate.

Cache Mapping

Note : CPU only generates address for main memory.

So when CPU generates addres we use cache mapping to search in cache using address generated for main memory.

It is a set pattern, which is used to map and find data in cache.
The concept is equivalent to hash functions.

Mine
Date : / /20
Page :

Cache Mapping
- Direct Mapping
- Set Associative mapping
- Fully Associative mapping

Direct Mapping

Note : Mapping is always done on blocks.

Suppose the following,
Blocks in cache ⇒ 10 (0-9)
Blocks in main memory ⇒ 100 (0-99)

In direct mapping, example of mapping will be for example,

Cache block location = Memory block location % 10

Or generally,

Cache block location = Memory block location % no. of blok in cache

So a certain block in memory is always mapped to same block in cache.

So direct mapping causes confliction among a group of blocks in main memory.

Mine
Date : / /20
Page :

But the cache itself does not know which block in main memory holds data of.
So each block in cache will hold identification data to store the data about which block from main memory is stored there.
This is called a tag.

So for example, if storing 63 in cache,

63 % 10 = 3 is the cache mapping and 6 is stored as a tag

So, 6 3
Tag — Mapping

This is a common pattern in mapping.
The same concept works in binary numbering system.

Mine
Date : / /20
Page :

Lecture 29

As stated previously, the same idea of tag and cache address from main memory address works in actual systems in binary.

So assume, there are 8 blocks in main memory and 4 blocks in cache memory.

So the process of storing block no 5 in main to cache,

Cache block location = Memory block location % No of blocks in cache

$= 5 \% 4$

$= 1$

$5 = (101)_2$ $\quad 1 = (01)_2$

Therefore, $(5)_{10} = 1\,01$

1 — Tag; 01 — Location in cache memory

This is how the process works in binary

⇒ Note: Blocks do not refer to the unique addresses assigned to in memory. A block is just a continuous chunk of memory, so a single block can have multiple addresses.

For example, if block size is 2 bytes and memory is byte addressable, each block has 2 memory addresses.

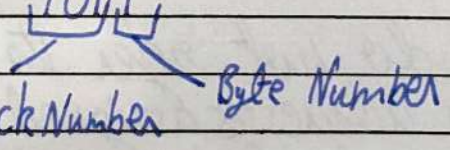
Main memory address to block number

Suppose a block is 2 bytes in a byte addressable memory.

For block number we take the first $\log_2$(no. of blocks) bits from the main memory address for the block number. Here no. of blocks refers to blocks in main memory.
So if there a 8 blocks, each of 2 bytes,

Size of memory address = 4 bits
Size of block no address = 3 bits

So first 3 bits of memory address show the block number.
Suppose the memory address is 1011,
Then,



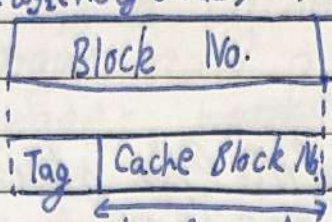
The remaining bits are referred to as byte number.

The block no. is further divided to tag and cache ~~m address~~ block No.

Mine
Date : / /20
Page :

Memory address

$\leftarrow \log_2$(no of blocks) $\rightarrow$ $\leftarrow$ $\log_2$(Block size in bytes) [for byte address] $\rightarrow$



⇒ Note: Cache block and cache line refer to same thing. Also called index field when referring to no. of bit in address form.

⇒ Note: tags are metadata. So total size of all tags is also referred to as metadata of cache memory. Total tag data is also called tag directory.

Mine
Date : / /20
Page :

Lecture 30

Cache Initialization

When computer is initially started, Cache will have garbage value because it is flushed.

This may cause a hit on CPU request to one of the garbage value.
To resolve this, we keep a valid bit for each block of cache and initialize it to zero, now before checking in cache, CPU will check the valid bit.

The valid bits are considered a part of Tag directory.

Write Back Cache Performance Improvement

If CPU only wants to perform read on a cache block, then that block can be ~~read~~ replaced directly w/o write back.

This can be done using Modified bit or dirty bit.

This bit is also a part of Tag directory (other bits can also be mentioned in ques)

Mine
Date : / /20
Page :

So in a cache chip, we have cache storage and a cache controller which manages ~~modi~~ tag, dirty, valid bits of cache and also sends and recieves signals to CPU.

So when talking about cache storage, we don't talk about these metadata bits.
Metadata in cache ⇒ Tag, modified bit, Valid bit

In write back cache,
simultaneous :-

$t_{avg\ read\ or\ write} = H \times t_{cm} + M \times (t_{block} + t_{write\ back})$

~~Way~~ hierarchical :-

$t_{avg\ read\ or\ write} = H \times t_{cm} + M(t_{cm} + t_{block} + t_{write\ back})$

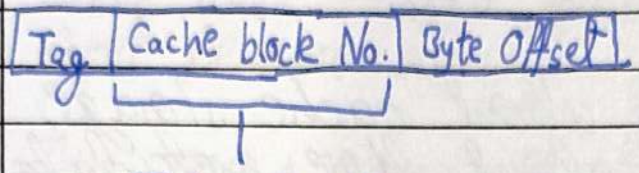
$t_{write\ back}$ = fraction of dirty block $\times$ t_{block}

Mine
Date : / /20
Page:

Lecture - 31

Set Associative Mapping

Memory address

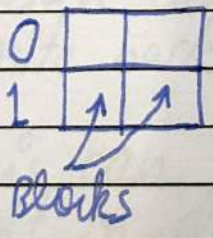


2-way set associative

2 blocks are present per index in cache.

So, Cache block no. = (Main memory block Number) % (No. of sets in cache)

The main reason to do this is so that if CPU is frequently accessing two blocks that and they are mapped to same block in cache the blocks are not frequently replaced



So two tags are present for each set in cache.

Mine
Date : / /20
Page :

We refer to this collection of block to a single address number as a set.

So main memory address has a change in terminology

[Tag | Set Offset | Byte Offset]

The process to get set offset remains same.

Number of blocks per address in cache = Associativity of cache

Number of sets in cache = Number of blocks in cache / Associativity

Note: If block size not given in question. Remember that the size in bits in direct mapped cache of cache block no. and byte offset is $\log_2$ (cache size) or $\log_2$ (cache size / associativity)

[Tag | Cache Block No. | Byte Offset]

$\log_2$ (cache size) bits or $\log_2$ (cache size / associativity)

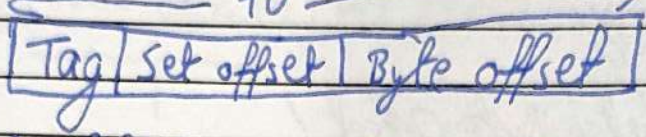
Can also assume block size to be 2^x. Assuming it to 2^x will also help solve ques of associative mapping

Mine
Date : / /20
Page :

For example,

Q) Width of physical address on a machine is 40 bits. The width of tag field in a 512 KB 8-way associative cache is ____ bits?

Sol)



Let block size be 2^x

Byte offset = $\log_2$ (Block size) = x

~~Set offset =~~
~~Number of sets in cache =~~

Number of blocks in cache = $\frac{\text{Cache size}}{\text{Block size}}$

$= \frac{2^9 \times 2^{10} \text{ Bytes}}{2^x \text{ Bytes}}$

$= 2^{19-x}$

Number of sets in cache = $\frac{\text{Number of blocks in cache}}{\text{Associativity}}$

$= \frac{2^{19-x}}{2^3}$

$= 2^{16-x}$

Set offset = $\log_2$ (Number of sets in cache)
= $16-x$ bits

Byte offset + Set offset = $16-x+x$
= 16

Bits for tag = $40 - 16$

Mine
Date : / /20
Page :

Fully Associative Mapping

In fully associative mapping,

associativity = Number of blocks in cache.

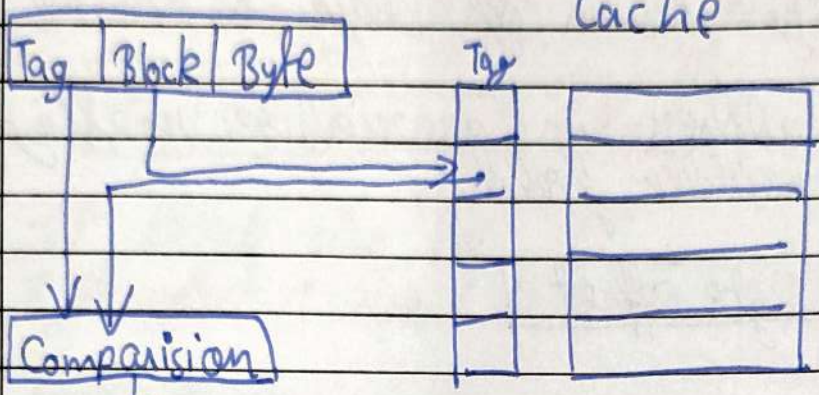
Memory address in associative mapping has no middle field.

Tag	Byte Offset

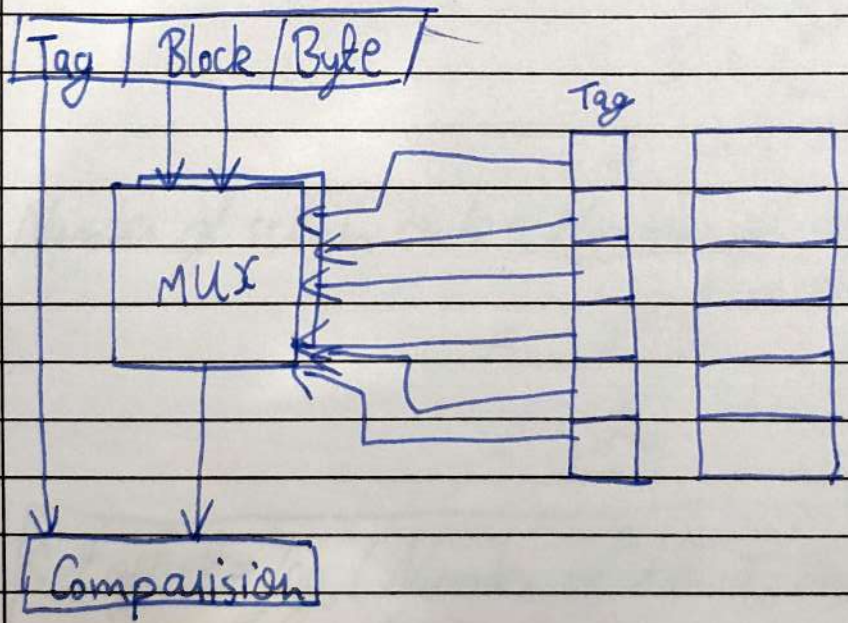
Mine
Date : / /20
Page :

Lecture -32

Checking Hit/Miss in Direct Mapping



Hardware Implementation in Direct Mapping



No. of MUX = Number of bits in tag
Size of MUX = Number of blocks in Cache X 1
Size of Comparator = Number of bits in tag.

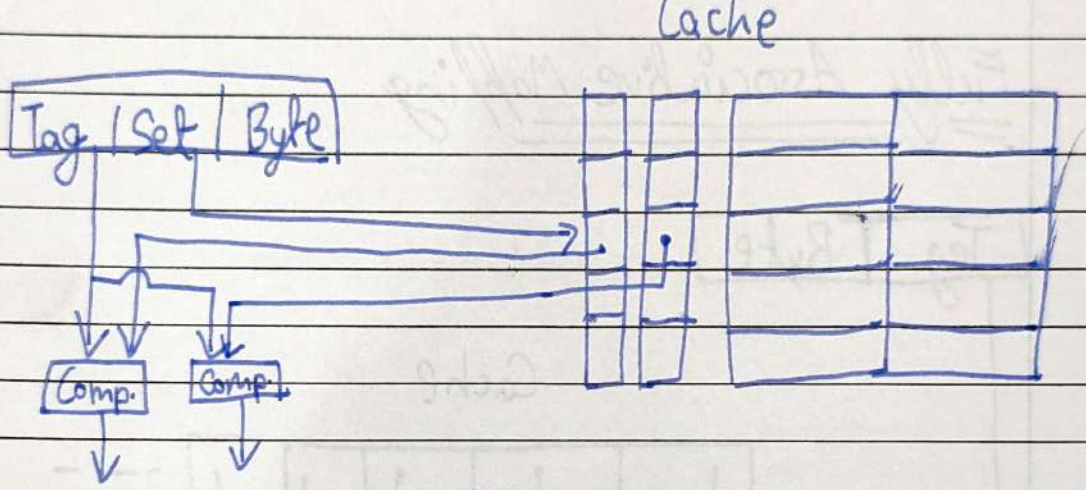
Mine
Date : / /20
Page :

Hit Latency in Direct Mapping

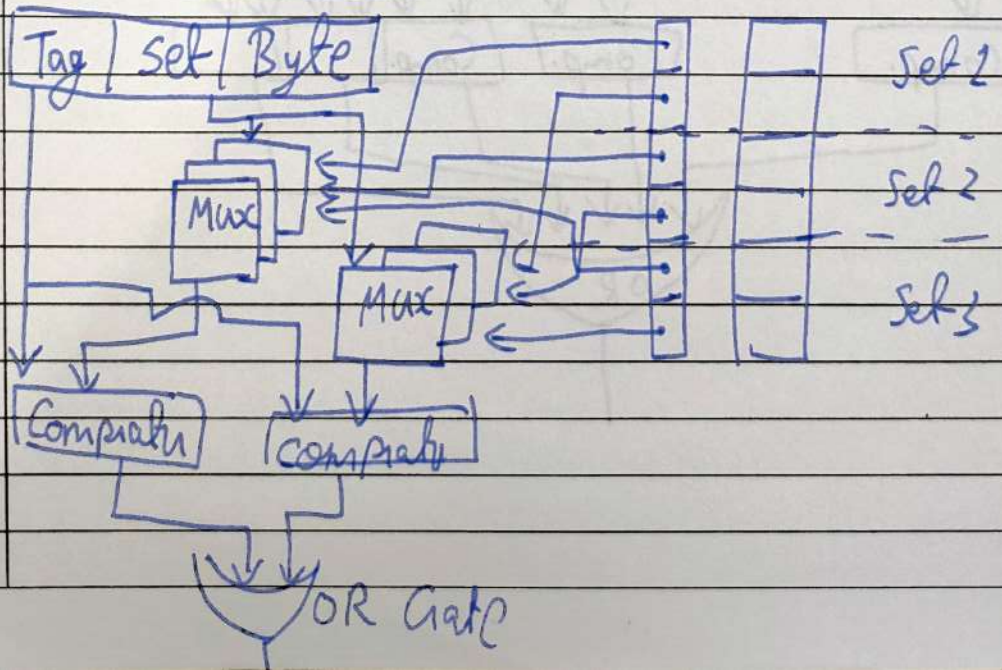
Latency caused to check for a hit or miss in cache.

Hit latency = Mux delay for tag selection + Comparator Delay.

Checking Hit/Miss in Set Associative Mapping



Hardware Implementation

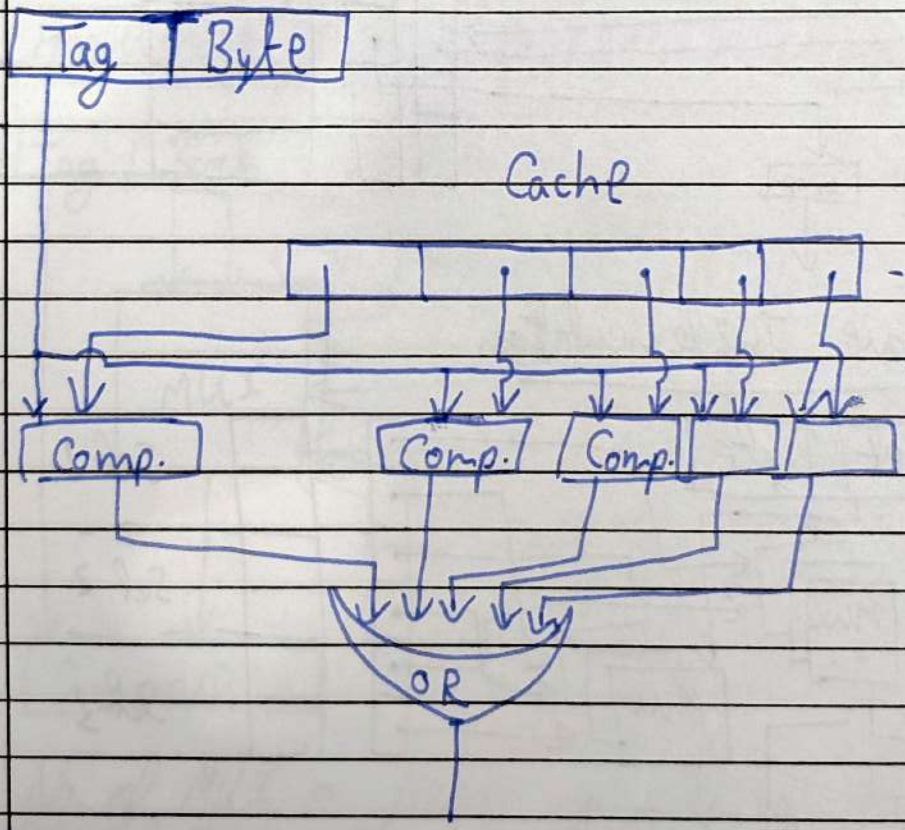


For k-way associative
Size of MUX = No. of sets in cache X 1
No. of MUX for tag selection = k X No. of tag bits
No. of comparators = k
Size of comparator = No. of tag bits.

Hit/miss latency in k-way set associative

Latency = MUX delay for tag selection + Comparator Delay + OR Gate Delay.

Fully Associative Mapping



Mine
Date : / /20
Page :

No. of comparators = No. of blocks in cache
Size of comparators = Tag bits.

Note : Sometimes question will mention a 2x1 Mux rather than an OR Gate. Here 2x1 Mux is used to implement the OR Gate.

Mine
Date : / /20
Page :

Lecture-33

Block Replacement : Direct Mapping

In direct mapping, we don't have a special block replacement policy and replace block if a new block is to be brought in cache.

Block Replacement : Set Associative Mapping

Since multiple blocks are in the same set, we need a policy to know which block is replaced.

Replacement policies

- FIFO (First in First out)
- Optimal ⇒ Keep the block which is used more in the future. (Not practical)
- LRU (Least Recently Used) ⇒ Replace the block which has not been referred for longest time.

These three policies are also used in fully associative cache.

Mine
Date : / /20
Page :

Lecture-34

Cache Misses

- Cold or Compulsory Miss
- Capacity Miss
- Conflict Miss

↓ Check for type of miss in this order

Cold or Compulsory Miss

First time access of a block will always cause a miss

Solution : Increase block size to decrease compulsory Miss

Capacity Miss

If cache is full and hence, miss occurs.

Solution : Increase the cache size.

Conflict Miss

If cache set is full and hence miss occurs due to tag mismatch. (Not a cold or capacity miss)

Solution : Increase the associativity.

Mine
Date : / /20
Page :

Cache Miss Penalty

Time required to bring a missed block from main memory to cache.

Three main info are needed to calculate miss penalty.

- Cycles required to send addres to memory
- Cycles required to access 1 main memory cell
- Cycles required to transfer 1 cell data to cache.

Example-

Cache block size	Memory cell size	Miss Penalty
4 bytes	1 byte	1+(4X10)+(4X1)
4 bytes	2 bytes	1+(2X10)+(2X1)
4 bytes	4 bytes	1+(1X10)+(1X1)

The given info for above is
Cycles to send address = 1 cycle
Cycles to access 1 main memory cell = 10 cycles
Cycles to transfer 1 cell data to cache = 1 cycles

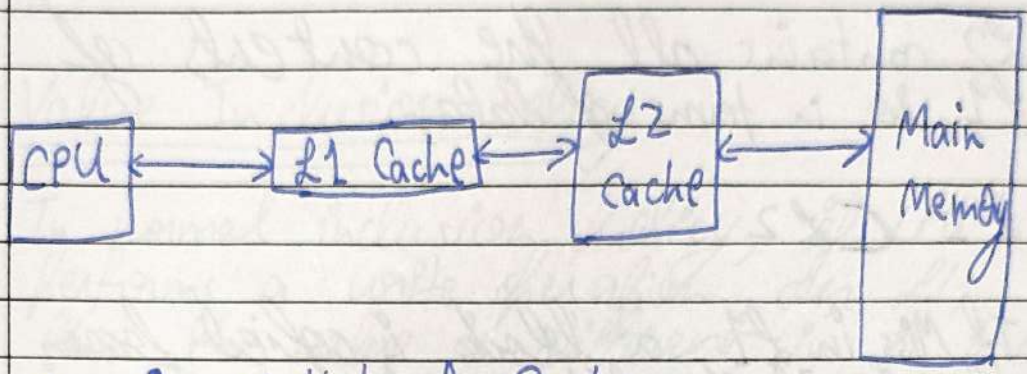
Mine
Date : / /20
Page :

Lecture 35

Goals of Using cache memory

- Minimize Acces Time ⇒ Smaller Cache
- Maximize Hit Rate ⇒ Larger Cache
- Minimize Miss Penalty

} Multi-level Cache.



Tavg in Multilevel Cache

1. Simultaneous Access

$$T_{avg} = H_1 \times t_1 + M_1 [H_2 \times t_2 + M_2 \times t_{mm}]$$

Subscript 1 ⇒ For Level 1
Subscript 2 ⇒ For Level 2

2. Hierarchical Access

$$T_{avg} = H_1 \times t_1 + M_1 [H_2 \times (t_1 + t_2) + M_2 (t_1 + t_2 + t_{mm})]$$

Mine
Date : / /20
Page :

Cache Inclusion Policy

- Inclusion
- Exclusion

Inclusion Policy

L2 contains all the contents of L1. So in terms of data,

$L1 \subset L2$

- If Miss in L1, a block is copied from L2 if it is hit in L2.
  If block is evicted from L1, then no changes in L2.
- If a block is evicted from L2 which is in L1, then send invalidation to L1.
- If miss on both L1 & L2, copy block to both L1 & L2.

Exclusion Policy

It is not necessary that content of L1 cache is also present in L2.

- If miss in L1 and hit in L2, then move block from L2 to L1, and evicted block from L1 is

moved to L2.

* Miss in both L1 & L2, copy block from main mem to L1 only and move evicted block to L2.

L2 hence is called victim cache as it is only taking data from L1.

Value Inclusion Policy

In normal inclusion policy, if CPU performs a write operation on L1, then the value is different in L1 & L2.

To have the same value of blocks in L1 & L2, we use value inclusion policy.

Lec 36 and 37 are practice questions on cache

Mine
Date : / /20
Page :

Lecture - 38

Magnetic Disk

~~In a magnetic disk, there are multiple dis~~
In a hard drive, there are multiple magnetic disks arranged in a stack. Each disk is called a platter. Unlike a optical disk, in a magnetic disk, both surfaces can be used to store data.
So if a drive has 3 platters, we say we have 6 surfaces

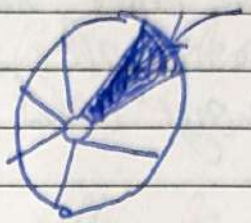
The column that spins all the platters is called spindle

On each surface we have a red read/write head, all heads are attached to an arm, all arms are attached to arm assembly.

The arm assembly is a perpendicular column, arms rotate with assembly to covers the whole surface of platters.

If the arm was fixed only a single line of circle would be read from the platter.

In a magnetic disk,



Sector of a track (referred simply as sector when talking about disk)

Cluster

Disk Capacity

- Constant sector capacity (Variable storage density)
- Variable sector capacity (Constant storage density)

The storage density is the ~~amount~~ amount of data stored per unit surface area. The above difference or inverse effect is caused because sectors near inner radius will have smaller areas compared to sectors near outer radius.

In questions, consider constant sector capacity as this is more widely used.

Sector in disk is the smallest unit of disk which ~~can't~~ can be read or written at once.
i.e for one disk read/write op. is accessing a single sector (Similar to how memory cells work in main memory).

Disk Access Time

Time taken to access 1 sector.

Seek time ⇒ Time to move the arm to correct position.
Rotational Latency ⇒ Time taken to rotate disk to bring sector ~~under head.~~ to the head.
Transfer time ⇒ time required for read/write.

Disk Access time ⇒ seek time + rotational latency + transfer time + additional latency given in question.

Sometimes, we need to calculate seek time using disk scheduling, this is a concept of OS.

* The head is not on top of the ~~head~~ sector, it is in between two sectors, so at the start on one sector and at the

end of another.

* The rotation of platter happens in a single direction.
* The head should be at the start of the sector which is to be under read/write op.

In questions, either we will be given data about head position and rotation time to calculate rotational latency. Or we can use average rotational latency if that is given.

If average rotational latency is also not given, then,

$$\text{rotational latency} = \frac{1 \text{ disk rotation time}}{2}$$

For transfer time:

In 1 rotation of disk = 1 track of data can be transfered.

Then,

$$1 \text{ sector transfer time} = \frac{1 \text{ rotation of disk time}}{\text{No. of sectors per track}}$$

Constant sector capacity	Variable sector capacity
↓	↓
Variable storage density	Constant storage density
↓	↓
Constant angular velocity	Constant linear velocity

Mine
Date : / /20
Page:

Lecture-39

Multiple Sectors Access Time

Sequential Multiple Access

The sectors are on the same track in a continuous manner.

So if we transfer n sectors :

n sector access time = seek time + rotational latency + n * 1 sector access time.

Random multiple Access

Since all the sectors are scattered

n sector access time = n * [seek time + rotational latency + 1 sector transfer time]

Cylinder in Disk

We want to always reduce seek time. (i.e the time to position read/write head).

So if suppose we wanted store a large file, we will store it on a single track. When this track is completely filled, rather than moving

Mine
Date : / /20
Page :

the head, we store next part in the same track on below surface. When that is also full, we use the same track on the next disk. This reduces seek time.

Hence we store data on disk in form of cylinder.

Number of cylinders = No of tracks.