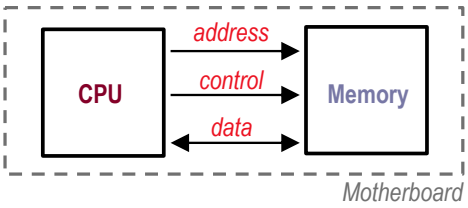
UNT
UNIVERSITY OF NORTH TEXAS

CSCE 5610
Computer System Architecture

Memory Hierarchy

COMPUTER ABSTRACTION

1) **CPU** and **Memory** are located on the motherboard



2) Memory Request Occurs

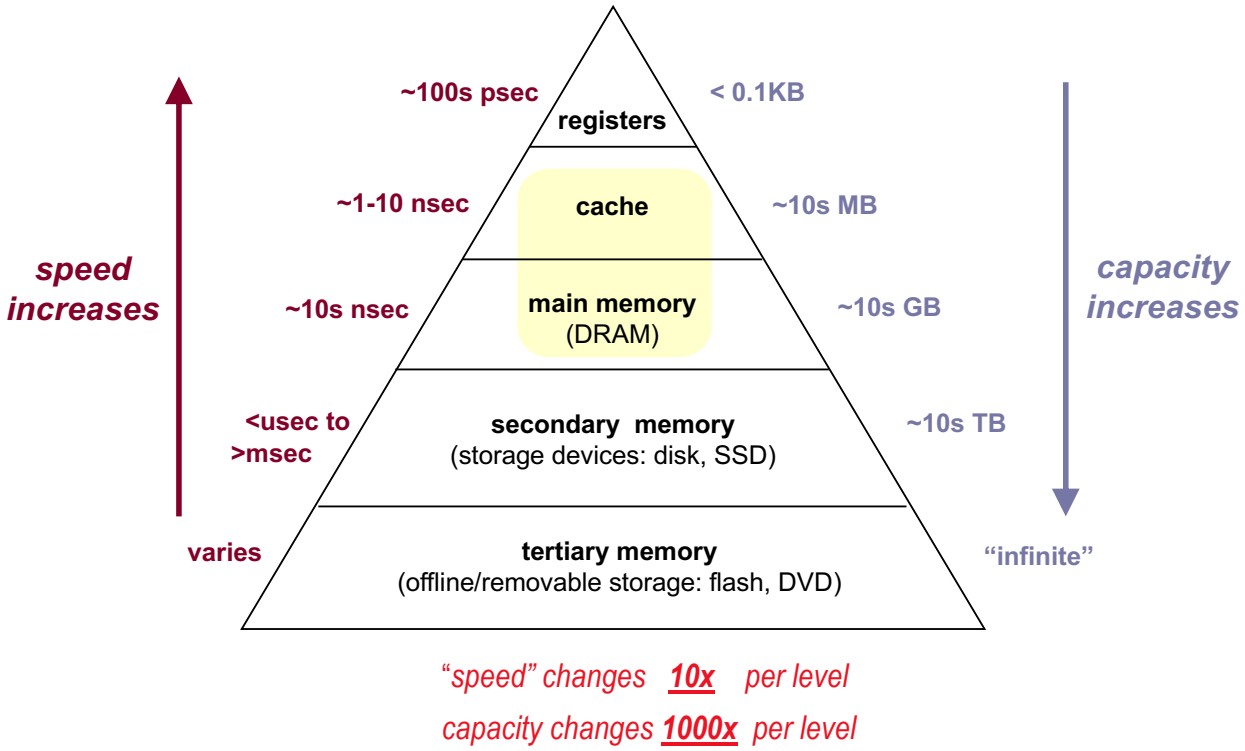
Instruction: Program Counter contains address of next instruction to fetch

Data: (offset + base) address from `lw $t0,offset(base)`

3) **CPU** outputs address bits over **Address Bus** to **Memory** and then asserts **Control Signals** such as *R/W* and *Enable*

4) **Memory** receives **address** and **control**, looks up the data value at that address, and sends the data bits over the **Data Bus** to the **CPU**

MEMORY HIERARCHY



*"speed" changes **10x** per level*

*capacity changes **1000x** per level*

Memory Technology

- **Static RAM (SRAM)**
  - $1000 –$2000 per GB
- **Dynamic RAM (DRAM)**
  - $5 –$10 per GB
- **Magnetic disk**
  - $0.05 –$0.1 per GB
- **Ideal memory**
  - Access time of SRAM
  - Capacity and cost/GB of disk

MEMORY HIERARCHY GOAL

Opposing properties of capacity vs speed:

"Large memories are slow." and "Fast memories are small."

So how do we create a memory that "gives the illusion" of being large, inexpensive, and fast?

with **hierarchy** – leverages locality

- the data at fastest levels is a subset of data at slower levels

with **parallelism** – multiple bit access between levels

- ***word***: 32-bits = 4 Bytes transferred at a time (Reg ⟷ Cache)
- ***block***: 32 to 128 Bytes at a time (Cache ⟷ Main Memory)

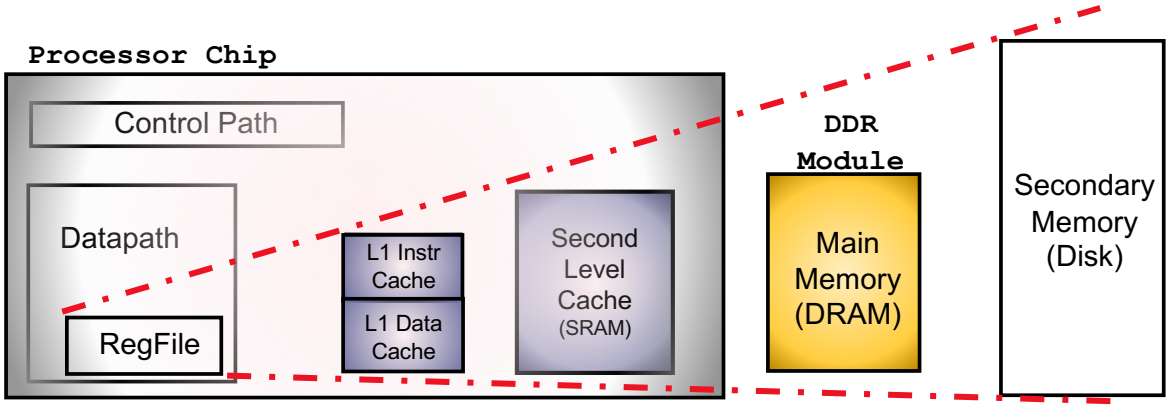
We seek to design hardware such that:

- [t_{access} = average access time] → close to registers
- [*$/bit* = average cost] → close to secondary storage

A TYPICAL MEMORY HIERARCHY

Memory Hierarchy:

Uses ***principle of locality*** to give illusion of memory has the ***cheapest*** **technology per bit**, at the speed offered by the ***fastest*** **technology**



Cost: ***($/bit)***	*highest*	*high*	*high*	*medium*	*lowest*
Speed: ***(clock cycles)***	*1*	*~5*	*~10*	*~50*	*>10,000*

WHY DOES MEMORY HIERARCHY WORK?

Programs access a small part of address space at any time

90/10 rule: 90% of time spent in 10% of the total code (loops)
90% of memory accesses are to 10% of variables in a program

Since memory access is not uniformly distributed during execution and throughout memory range

Can optimize to make the commonly used data fast:
Store the 10% commonly used in fast memory
Store the 90% not frequently used in slow memory

Results in improved performance vs. cost:
The system still has fast access time on average
The system still has low cost per bit on average

The important property of computer programs used here is *locality*.

The Principle of Locality

- It's usually difficult or impossible to figure out what data will be —most frequently accessed before a program actually runs, which makes it hardto know what to store into the small, precious cache memory.
- But in practice, most programs exhibit *locality*, which the cache can take advantage of.
  - — The principle of **temporal locality** says that if a program accesses one memory address, there is a good chance that it will access the same address again.
  - — The principle of **spatial locality** says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.

Temporal Locality in Programs

- Loops are excellent examples of temporal locality in programs.
  - The loop body will be executed many times.
  - The computer will need to access those same few locations of the instruction memory repeatedly.
- For example:

```
Loop: lw    $9, 0($22)
      add   $9, $9, $10
      sw    $9, 0($22)
      addi  $10, $9, -4
```

  - Each instruction will be fetched over and over again, once on every loop iteration.

Temporal Locality in Data

- Programs often access the same variables over and over, especially within loops. Below, sum and i are repeatedly read and written.

```
sum = 0;
for (i = 0; i < MAX; i++)
      sum = sum + f(i);
```

- Commonly-accessed variables can sometimes be kept in registers, but this is not always possible.
  - There are a limited number of registers.
  - There are situations where the data must be kept in memory, as is the case with shared or dynamically-allocated memory.

Spatial Locality in Programs

```
sub   $9, $9, $16
sw    $9, 0($22)
sw    $9, 4($22)
sw    $9, 8($22)
```

- Nearly every program exhibits spatial locality, because instructions are usually executed in sequence—if we execute an instruction at memory location *i*, then we will probably also execute the next instruction, at memory location *i+4*.
- Code fragments such as loops exhibit *both* temporal and spatial locality.

Spatial Locality in Data

- Programs often access data that is stored contiguously.
  - Arrays, like a in the code on the top, are stored in memory contiguously.
  - The individual fields of a record or object like employee are also kept contiguously in memory.
- Can data have both spatial and temporal locality?

```
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + a[i];
```

```
employee.name = "Homer Simpson";
employee.boss = "Mr. Burns";
employee.age = 45;
```

MEMORY HIERARCHY OPERATION

1) A **word** is referenced by **`Processor`** via its Byte Address in **`Main Memory`**

2) Is that address already in **`cache`**?

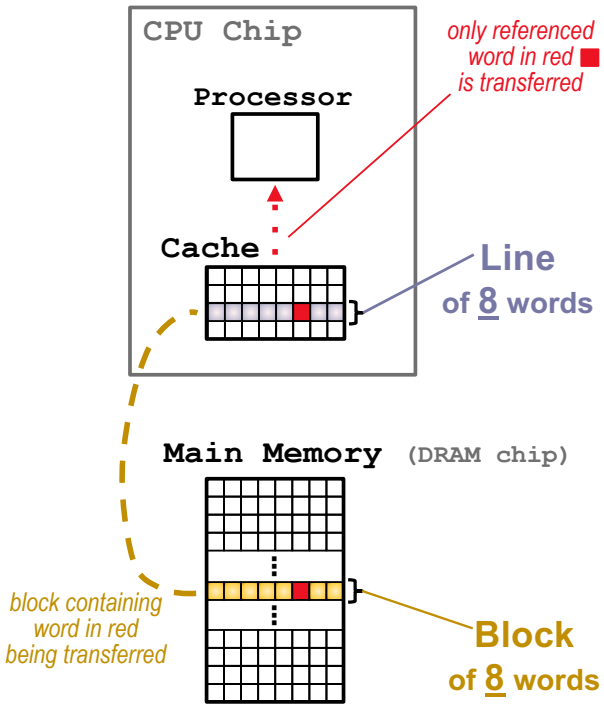
YES: access data from **cache line**

NO: transfer the "**containing block**" of data from memory to cache

Line size and Block size are *always* equal:

4, 8, 32, … or 128 words

captures spatial locality

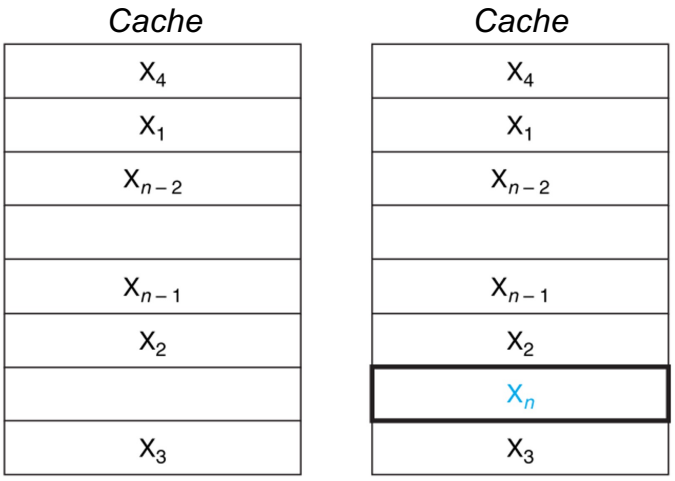


Definitions: Hits and Misses

- A **cache hit** occurs if the cache contains the data that we're looking for. Hits are good, because the cache can return the data much faster than main memory.
- A **cache miss** occurs if the cache does not contain the requested data. This is bad, since the CPU must then wait for the slower main memory.
- There are two basic measurements of cache performance.
  - The **hit rate** is the percentage of memory accesses that are handled by the cache.
  - The **miss rate** (1 -hit rate) is the percentage of accesses that must be handled by the slower main RAM.
- Typical caches have a hit rate of 95% or higher, so in fact most memory accesses will be handled by the cache and will be dramatically faster.
- In future lectures, we'll talk more about cache performance

THE BASICS OF CACHES

Let's assume a very simple cache in which the processor requests only one Word and the blocks also consist of one Word.



a. Before the reference to X_n b. After the reference to X_n

How do we know if the word referenced is in the cache?

If it is in cache, how do we find it?

DIRECT-MAPPED STRATEGY

The simplest way to assign a location in the cache for each word in memory is to assign the cache location based on the address of the word in memory.

Almost all direct-mapped caches use **`mod`** function of address bits to map main memory blocks to their storage location in the cache

Cache line number for storage =

{ [block address] modulo [number of blocks in the cache] }

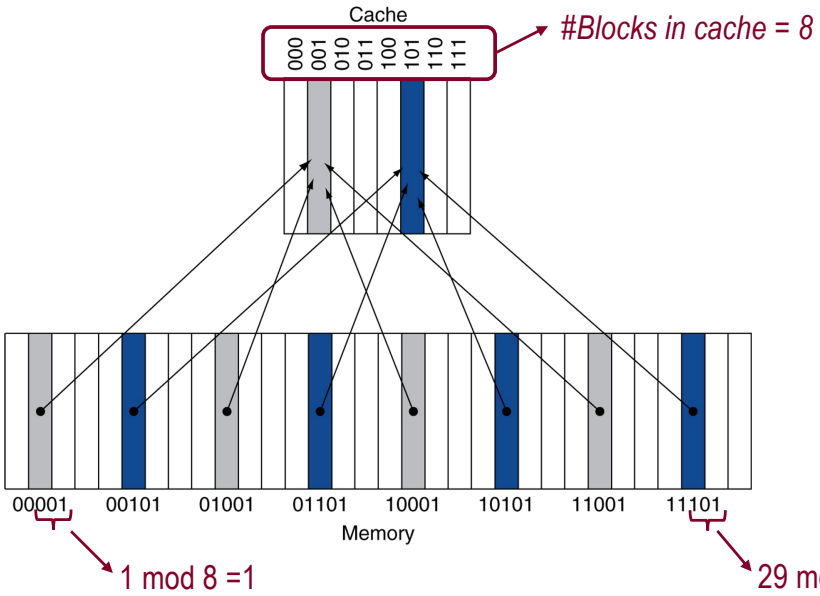
→ **`mod`** function can be realized for free by ignoring some higher address bits!!!

DIRECT-MAPPED CACHE

Location determined by address

Direct mapped: only one choice of cache line to use:

[Block address] mod [#Blocks in cache]



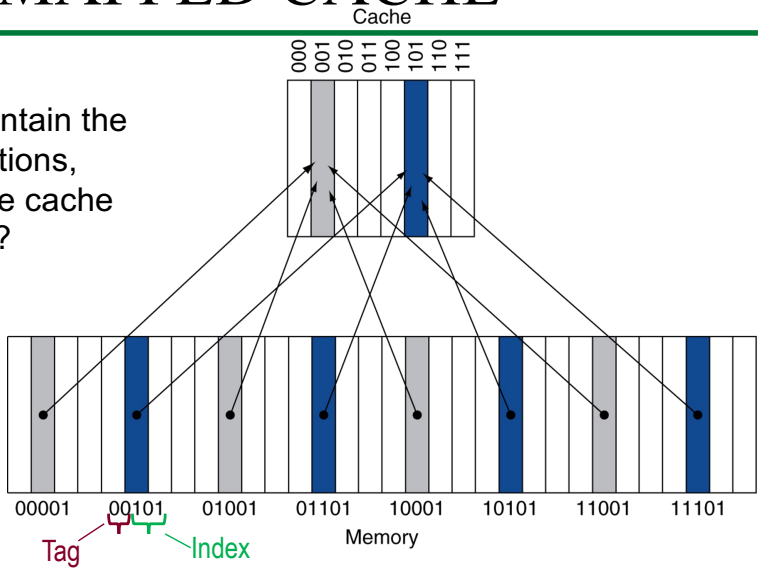
#Blocks is a power of 2

Use low-order address bits to determine bin (cache line)

Here every 8th block of memory maps to same line of cache

DIRECT-MAPPED CACHE

Since each cache location can contain the contents of different memory locations, how can we know if the data in the cache corresponds to a requested word?



Tag contains the address information to identify whether a word in the cache corresponds to a requested word → **tag** only contains the upper portion of the address corresponding to the bits that are not used as an index into the cache.

Valid bit indicates whether an entry contains a valid address. If the bit is not set, there cannot be a match for this block → For instance, when a processor starts up, the cache does not have good data

DIRECT-MAPPED CACHE: EXAMPLE

Given: CPU references addresses {**22**, 26, 22, 26, 16, 3, 16, 18} in that order.

Sought: Indicate contents of cache after each reference and calculate the hit rate.

reference information

Addr (Byte)	Binary addr	Cache line	Hit/Miss
22	10 110 →	110	miss

cache contents

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		

*Mem[10110] = **data** that is stored at address 10110.*

DIRECT-MAPPED CACHE: EXAMPLE

{22, **26**, 22, 26, 16, 3, 16, 18}

reference information

Addr (Byte)	Binary addr	Cache line	Hit/Miss
26	11 010	010	miss

cache contents

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

DIRECT-MAPPED CACHE: EXAMPLE

{22, 26, **22, 26**, 16, 3, 16, 18}

reference information

Addr (Byte)	Binary addr	Cache line	Hit/Miss
22	10 110	110	hit
26	11 010	010	hit

cache contents

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

DIRECT-MAPPED CACHE: EXAMPLE

{22, 26, 22, 26, **16**, **3**, **16**, 18}

reference information

Addr (Byte)	Binary addr	Cache line	Hit/Miss
16	10 000	?	?
3	00 011	?	?
16	10 000	?	?

cache contents

Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

DIRECT-MAPPED CACHE: EXAMPLE

reference information

Addr (Byte)	Binary addr	Cache line	Hit/Miss
18	10 010	?	?

cache contents

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

For this address reference string {22, 26, 22, 26, 16, 3, 16, **18**} on this cache design, a **Hit Ratio (or Hit Rate)** of (3 hits / 8 references) = 0.375 = 37.5% is obtained.

DIRECT-MAPPED CACHE

Let's calculate the total number of bits in a direct-mapped cache:

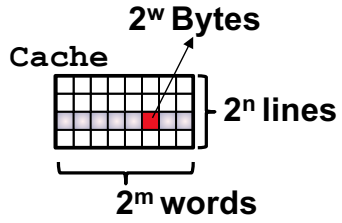
Main Memory size is 2^a Bytes
given memory is *Byte-addressable* → width of address bus = a bits
(for MIPS a=32)

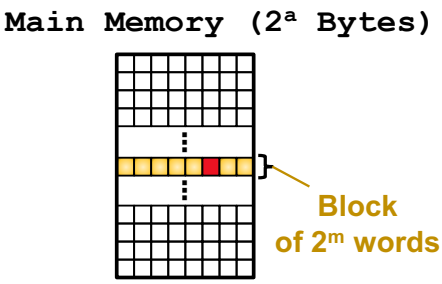
Cache Size is 2^n blocks
blocks in cache often referred to as "lines"
If there are 2^n lines then *n* bits are needed to specify the line index

Block size is 2^m words:
m bits are needed to specify the word index within block
"main memory block size" is *always* the same as "cache line size"

Each Word contains 2^w Bytes
for MIPS w=2 → 32-bit word width

Size of tag field:
t bits are needed to uniquely label each item in cache
t = (a – [n+m+w]) bits = (a – n – m – w) bits
because a=t+n+m+w as entire memory address was decomposed into these fields





DIRECT-MAPPED CACHE

Given:

16MBytes of Byte-addressable main memory
Cache capacity of 64KB = 2^{16} Bytes
Block size is 4 words
Word size is 4 Bytes

How wide is main memory address bus?

2^a = 16MB = (2^4)(2^20) B, thus a=24. Answer: 24 bits

How many blocks are in main memory?

(4 words/block)*(4 Bytes/word)=16 Bytes/block
16MB / (16 Byte/block) = (2^24)/(2^4) blocks = 2^20 blocks. Answer: 1,048,576 blocks

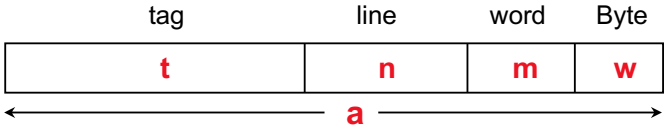
How many lines are in the cache?

Block size in main memory = line size in cache
64KB / (16 Bytes/line) = (2^16)/(2^4) = 2^12 line in cache Answer: 4,096 lines

DIRECT-MAPPED CACHE

Field Encoding:

Every address referenced by the processor contains **a** bits, encoded as:



where 1) Main Memory capacity is 2^**a** Bytes
2) Cache capacity is……2^**n** lines
3) Block/Line size is……2^**m** words
4) Each word contains…… 2^**w** Bytes
5) Tag field width is……**t** = (a – n – m – w) bits

DIRECT-MAPPED CACHE: EXAMPLE

Given: You are tasked to design MIPS-based ultra-lightweight IoT sensor powered only by ambient wifi signals. It has 256 Bytes of main memory, a 64-Byte direct-mapped cache, and block size of 2 words.
Sought: address field encoding.

1) How many bits wide is the address bus?

256=2^a → a=8. Answer: 8 bits

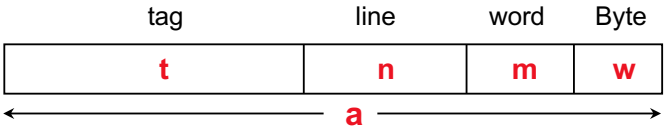
2) How many lines are in the cache?

64 Bytes / (4 Byte/word) = 16 words in cache
(16 words)/(2 words/line)=2^3=8 lines → n=3 Answer: 8 lines

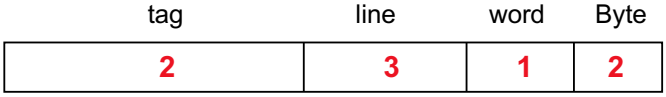
3) What is the bit field encoding for memory addresses?

MIPS has 32-bit word = 4 Bytes so 2^w=4 → w=2.
It is given that 2^m=2 word/block → m=1.
So a=8, n=3, m=1, w=2 thus t=8-3-1-2=2 so t=2.

Substituting into:

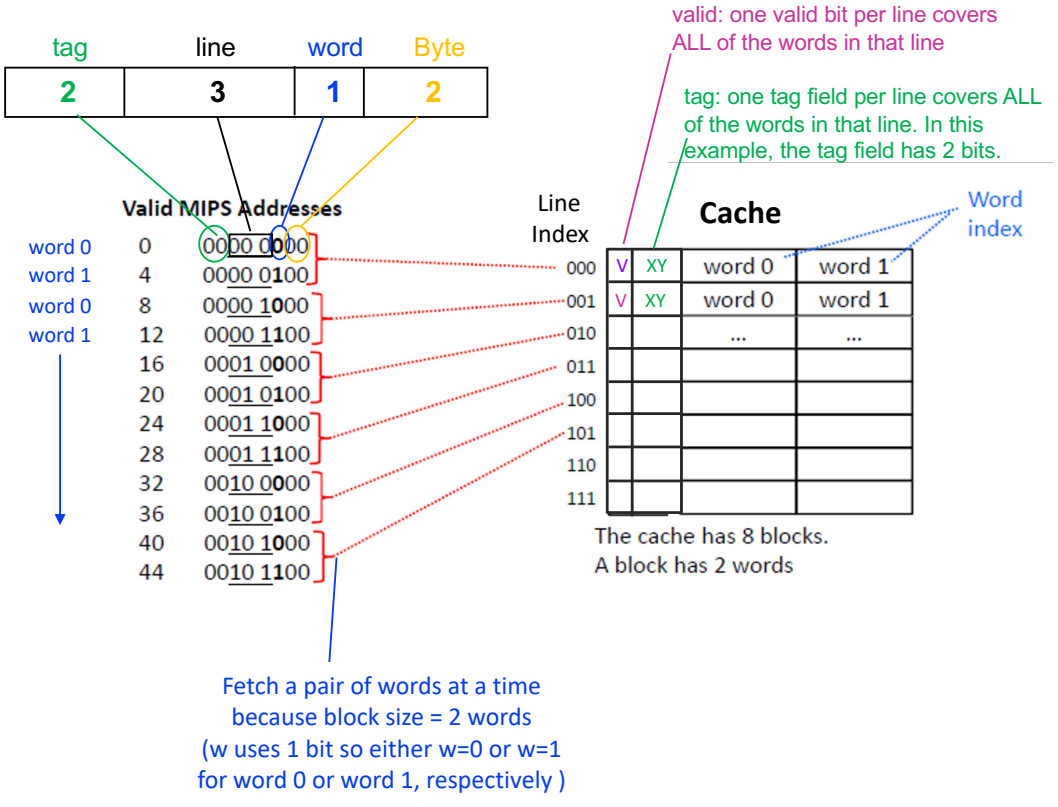


Field Encoding:



→ How are these bits used by the cache?

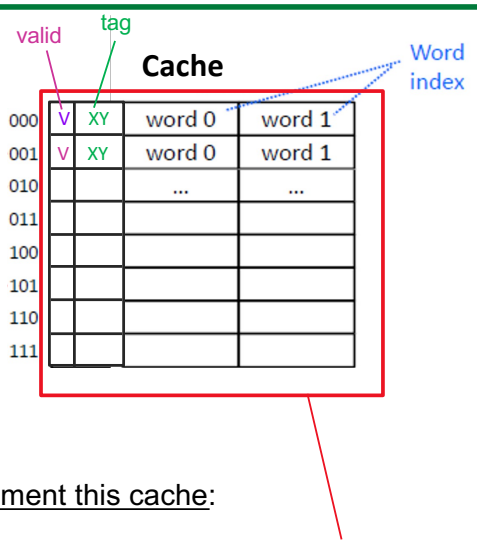
DIRECT-MAPPED CACHE



DIRECT-MAPPED CACHE

Given the cache on the previous slide:

8 lines deep
2 data words per line
Thus, cache capacity = 16 words of data
2-bit tag field per line
1 valid bit per line
32 bits per word
Direct-mapped strategy



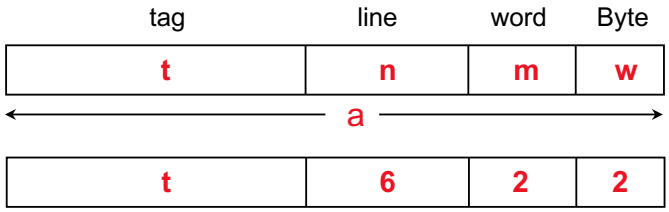
Calculate total number of storage bits needed to implement this cache:
(# cache lines) ×[number of bits required per line] =
8 × [1+2+32+32] = 8*67 = 536 bits inside

DIRECT-MAPPED CACHE

Given: A direct-mapped cache containing 64 lines with block size of 16 Bytes (4 words).
Sought: Which line in the cache would hold the content in main memory address 1200?

We can use the bit-field encoding:

- 64 = 2^n lines in cache → n=6
- It is given that 2^m=4 word/block → m=2
- It is given that 16 Bytes are 4 words so 4 Bytes/word = 2^w=4 → w=2



Memory address 1200 is the Byte address and written in binary 1200 = 10010110000
So according to the field encoding:

… 1	001011	00	00

address 1200 gives contents of the line field = $001011_{two} = 11_{ten}$ → **it resides in block 11_{ten}**

USE OF SPATIAL LOCALITY

Assume main memory has a total size of 16 words → $2^a=16$ →a=4

Assume cache has total size of 4 words and each line holds 2 words
→ 4/2 = 2 lines → $2^n = 2$ → n=1
→Each cache line holds two words → $2^m=2$ → m=1

Assume Word size = 1 Byte so $2^w=1$ → w=0 thus t= a-n-m-w = 4-1-1-0 = 2 bits

tag	line	word
2	1	1

USE OF SPATIAL LOCALITY

Assuming the memory hierarchy described in the previous slides

Memory reference string: bit format is [(**tag1 tag0** *tag*) (**line** *up/down*) (**word** *left/right*)]

{ 0 1 2 3 5 3 4 15 } in decimal

0000 0001 0010 0011 0101 0011 0100 1111 in binary

0 miss

00	Mem[0]	Mem[1]

1 hit

00	Mem[0]	Mem[1]

2 miss

00	Mem[0]	Mem[1]
00	Mem[2]	Mem[3]

3 hit

00	Mem[0]	Mem[1]
00	Mem[2]	Mem[3]

5 miss

~~00~~ 01	~~Mem[0]~~ 4	~~Mem[1]~~ 5
00	Mem[2]	Mem[3]

3 hit

01	Mem[4]	Mem[5]
00	Mem[2]	Mem[3]

hit even though not seen previously... thanks to spatial locality!

4 hit

01	Mem[4]	Mem[5]
00	Mem[2]	Mem[3]

15 miss

~~01~~ 11	Mem[4]	~~Mem[5]~~ 15
~~00~~	~~Mem[2]~~ 14	~~Mem[3]~~

8 requests, 4 misses → h=4/8=0.5=50% → m=1-h=0.5=50%

TYPES OF CACHE MISSES

Compulsory Miss:

always miss very first access to a block → small impact in practice as access a billion per second

Solution: *prefetching* → a block of data is brought into the cache before it is actually referenced.

Capacity Miss:

cache cannot contain all blocks accessed by the program

Solution: increase cache size (may increase access time)

Conflict Miss:

multiple memory locations mapped to the same cache location

Solution 1: increase cache size (may increase access time)

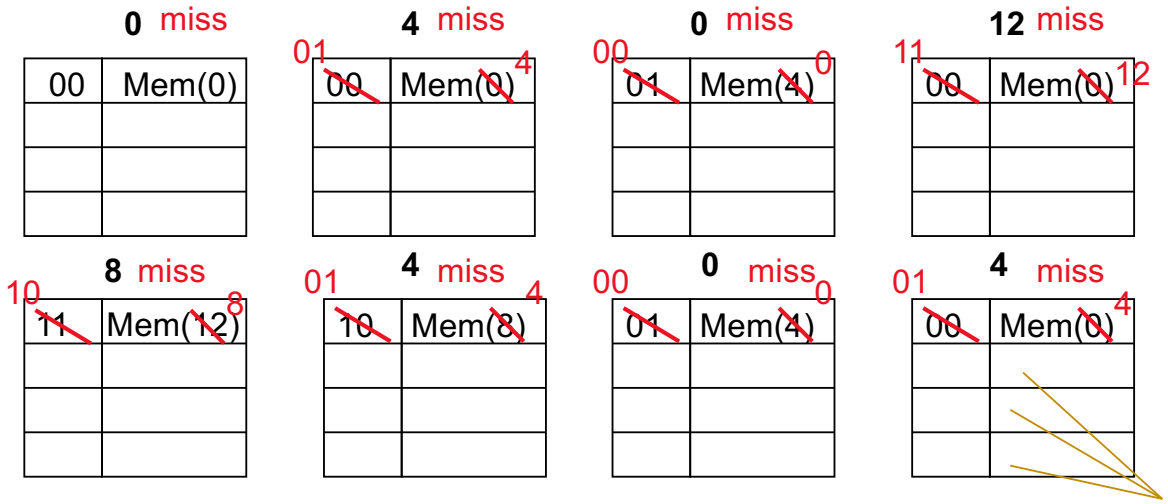
Solution 2: increase associativity (may increase access time)

DIRECT-MAPPED CACHE: CONFLICT MISS

{ 0 4 0 12 8 4 0 4 } in decimal

0000 0100 0000 1100 1000 0100 0000 0100 in binary

Cache size = 4 words, 4-bit memory address, 1 word per block, 1 Byte per word



→ 8 requests, 8 misses → hit rate = 0

available but unused

Ping-pong effect due to *conflict misses*:
i.e. two memory locations that map into the same cache line

→ *solve with associativity* →

ASSOCIATIVITY

We seek more flexible block placement:

In a **Direct Mapped cache** a memory block maps to exactly one cache line ***only***

At the other extreme, could allow a memory block to be mapped to ***any*** cache line → so-called **Fully Associative cache**

A compromise is to divide the cache into **sets** each of which consists of *k* **ways** (so-called **k-way Set Associative cache**)

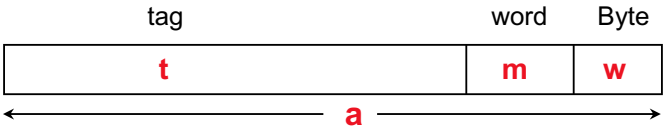
Give ***k choices/chances*** for each block of memory to place in cache

Each memory block maps to a unique set (specified by the index field) and can be placed in any "way" of that "set" → there are k choices in the cache for each block

FULLY-ASSOCIATIVE CACHE

Field Encoding:

Every address referenced by processor contains **a** bits, encoded as:



where 1) Main Memory capacity is 2^**a** Bytes
2) Each word contains……. 2^**w** Bytes
3) Block size is……...2^**m** words
4) Tag field width is……**t** = (a – m – w) bits

Size of fully associative cache is not part of encoding, because there are no direct addresses to cache lines.

FULLY-ASSOCIATIVE CACHE

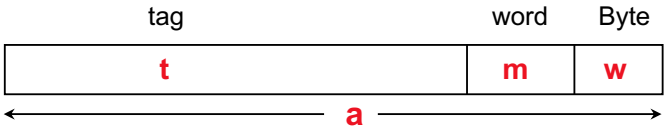
Given: MIPS-based tablet system has 4GB Bytes of main memory, a single-level of fully associative-mapped cache, and block size of 32 words.

Sought: What is the bit-field encoding for memory addresses?

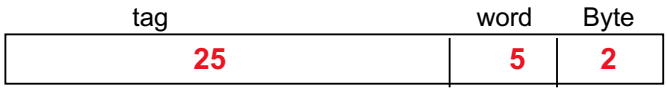
Solution:

MIPS has 32-bit word = 4 Bytes so 2^w=4 → w=2.
It is given that 2^m=32 word/block → m=5.
4GB main memory = (2^2)(2^30)B → a=32,
m=5, w=2 thus t=32-5-2=25 so t=25.

Substituting into:



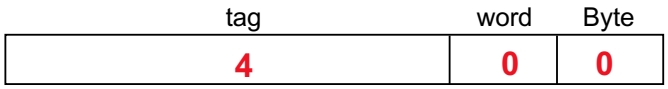
Answer:



FULLY-ASSOCIATIVE CACHE

{ 0 4 0 12 8 4 0 4 } in decimal
0000 0100 0000 1100 1000 0100 0000 0100 in binary

Cache size = 4 words, 4-bit memory address, 1 word per block, 1 Byte per word



0 miss

0000	Mem(0)

4 miss

0000	Mem(0)
0100	Mem(4)

0 hit

0000	Mem(0)
0100	Mem(4)

12 miss

0000	Mem(0)
0100	Mem(4)
1100	Mem(12)

8 miss

0000	Mem(0)
0100	Mem(4)
1100	Mem(12)
1000	Mem(8)

4 hit

0000	Mem(0)
0100	Mem(4)
1100	Mem(12)
1000	Mem(8)

0 hit

0000	Mem(0)
0100	Mem(4)
1100	Mem(12)
1000	Mem(8)

4 hit

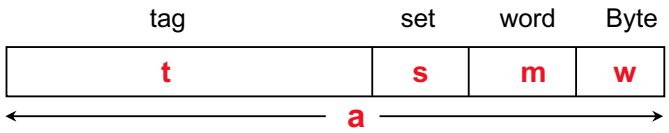
0000	Mem(0)
0100	Mem(4)
1100	Mem(12)
1000	Mem(8)

→ 8 requests, 4 misses → hit rate = 50%

K-WAY SET ASSOCIATIVE CACHE

Field Encoding:

Every address referenced by processor contains **a** bits, encoded as:



where
1) Main Memory capacity is 2^**a** Bytes
2) Each word contains……. 2^**w** Bytes
3) Block size is……....2^**m** words
4) The cache contains.....……2^**s** sets
5) Each set contains……**k** lines
6) Tag field width is……**t** = (a – s – m – w) bits

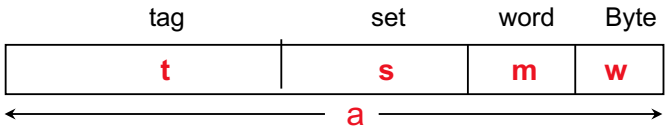
SET ASSOCIATIVE CACHE

Given: An embedded 8-bit microcontroller has 16 Bytes of main memory, a 2-way set associative-mapped cache containing 4 lines, and a block size of 1 word.

Sought: Show bit-field encoding for memory addresses.

Solution:
1) 16B =2^4B memory → a=4
2) 8-bit word = 1 Byte so 2^w=1 → w=0.
3) It is given that 2^m=1 word/block → m=0.
4) 2-way → k=2 line/set. So, the number of sets = (2^2 lines)/(2 line/set)=2^1 sets→s=1.
5) So, t=a-s-m-w= 4-1-0-0 → t=3.

Substituting into:



Answer:

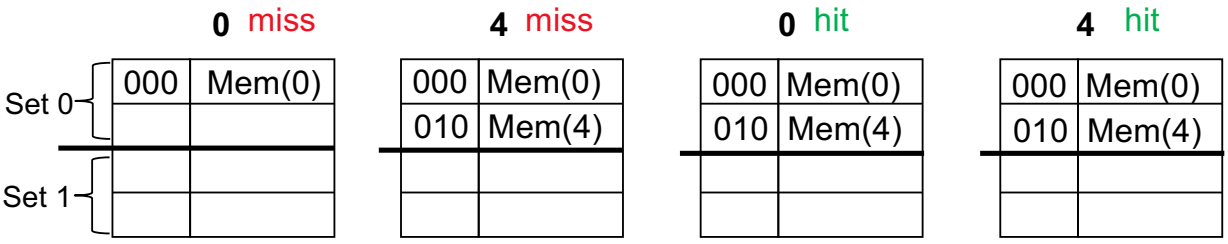
m and w not part of the encoding because they are zero bits wide

SET ASSOCIATIVE CACHE

{ 0 4 0 4 0 4 0 4 } in decimal
0000 0100 0000 0100 0000 0100 0000 0100 in binary

Cache size = 4 words with a 2-way set-associative map, 4-bit memory address, 1 word/block, 1 Byte/word

tag	set	word	Byte
3	1	0	0



8 requests, 2 misses … compare to direct-mapped miss rate=8/8

REPLACEMENT POLICIES

What occurs when Cache is full?

- move blocks to next level of cache but which one?

1) Direct Mapped caches

no replacement policy needed

no alternatives possible because memory address is directly mapped to cache line index

whenever a miss occurs then line is replaced, no matter how recent thus jeopardizing temporal locality

2) Fully Associative and Set Associative Caches

need to decide which block to replace (keep ones likely to be used in cache)

Least Recently Used (LRU): looks backwards in time to predict future accesses

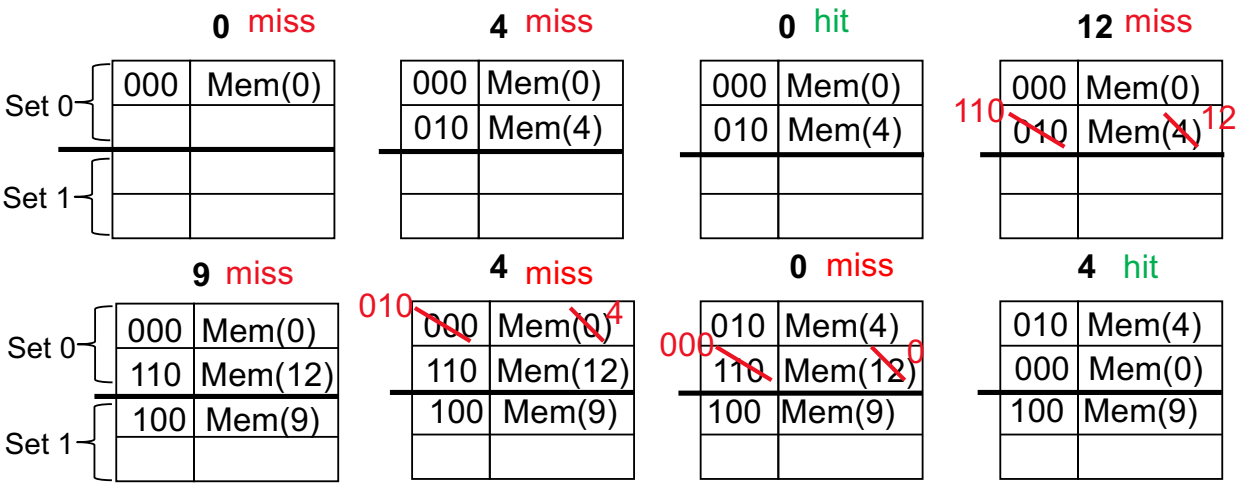
Least Frequently Used (LFU): counts occurrence of reference without regards to time of reference.

SET ASSOCIATIVE CACHE WITH LRU

{ 0 4 0 12 9 4 0 4 } in decimal
0000 0100 0000 1100 1001 0100 0000 0100 in binary

Cache size = 4 words with a 2-way set-associative map with LRU replacement policy, 4-bit memory address, 1 word/block, 1 Byte/word

tag	set	word	Byte
3	1	0	0



→ 8 requests, 6 misses → hit rate = 25%

REPLACEMENT POLICIES

Least Recently Used (LRU) Policy

Looks backwards in time as some prediction of future accesses
assumes temporal locality is prevalent

Hardware Implementation:

Provide a hardware counter that is incremented with each reference to a cache line

Example: 4 lines of associativity → 2-bit counter

1) reset counter for line accessed to 0
2) increment counter of other line
3) When miss occurs, replace line with highest count

REPLACEMENT POLICIES

Least Frequently Used (LFU) Policy

When capacity miss occurs replace line with fewest references so far in the string

Hardware Implementation:

Provide a hardware counter for each line that is only incremented if that cache line is accessed

When miss occurs, replace the line with the lowest count

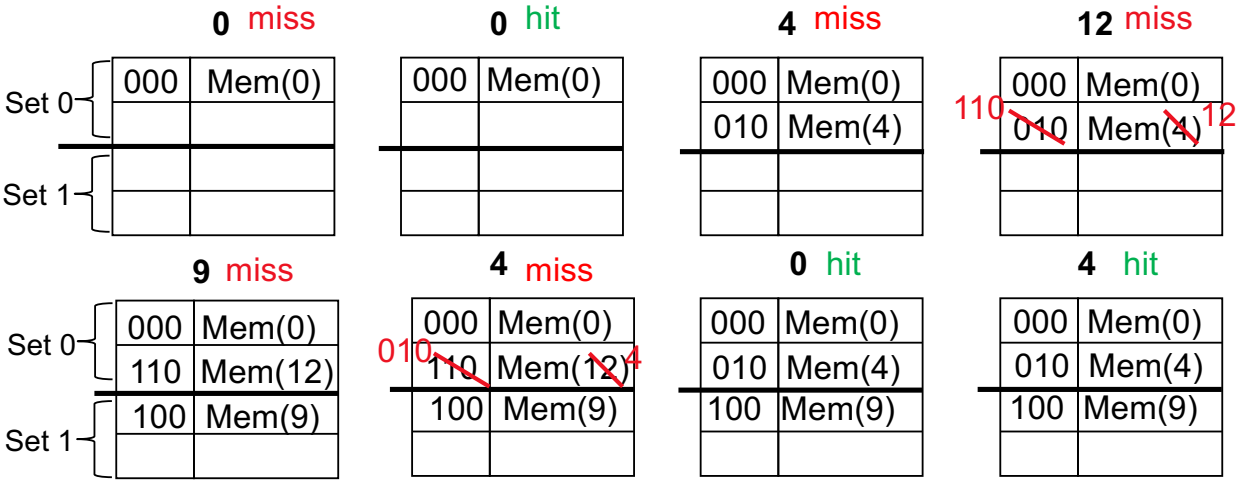
SET ASSOCIATIVE CACHE WITH LFU

{ 0 0 4 12 9 4 0 4 } in decimal

0000 0000 0000 1100 1001 0100 0000 0100 in binary

Cache size = 4 words with a 2-way set-associative map with LFU replacement policy, 4-bit memory address, 1 word/block, 1 Byte/word

tag	set	word	Byte
3	1	0	0



→ 8 requests, 5 misses → hit rate = 37.5%

CACHE PERFORMANCE

CPU Time: Clock cycles that the CPU spends executing the program plus the clock cycles that the CPU spends waiting for the memory system.

CPU Time = (CPU execution clock cycles + Memory-stall clock cycles) × Clock cycle time

Memory-Stall Clock Cycles: sum of the stall cycles coming from memory access operations

Memory-Stall Clock Cycles = (Memory Access/Program) × Miss rate × Miss Penalty

OR

Memory-Stall Clock Cycles = (Instructions/Program) × (Misses/Instruction) × Miss Penalty

CACHE PERFORMANCE: EXAMPLE

Given: Assume a processor with a CPI of 2 is running a program with 36% load and store instructions.

Sought: If the instruction cache and data cache has a miss rate of 2% and 4%, respectively, and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never missed.

Solution: CPU Time = (CPU execution clock cycles + Memory-stall clock cycles) × Clock cycle time

Memory-Stall Clock Cycles = (Instructions/Program) × (Misses/Instruction) × Miss Penalty

1) CPU execution clock cycles = Instructions × CPI = 2 × Instructions

2) Memory stall clock cycles = Instruction miss cycles + Data miss cycles
= (Instructions × 0.02 × 100) + (0.36 × Instructions × 0.04 × 100)
= (2 × Instructions) + (1.44 × Instructions) = 3.44 × Instructions

3) CPU Time = (2 × Instructions + 3.44 × Instructions) × Clock cycle time
= 5.44 × Instructions × Clock cycle time

4) Speed-up with a Perfect Cache = CPU Time with Stalls/CPU time with perfect caches
= (5.44 Instructions × Clock cycle time) / (2 × Instructions × Clock cycle time)
= 5.44/2 = 2.72

CACHE PERFORMANCE: AMAT

Average Memory Access Time (AMAT) metric is used to include the effect of the time to access data for both hits and misses on performance

AMAT= Time for a hit + (Miss rate × Miss Penalty)

AMAT can be calculated based on *seconds* or *# of clocks.*

→ AMAT in cache layer ***n*** = Miss Penalty in cache layer ***n-1***

AMAT: EXAMPLE

Given: Assume a processor with a 1ns clock cycle time, a miss penalty of 20 clock cycles, a miss rate of 0.05 misses per instruction, and a cache access time (including hit detection) of 1 clock cycle.

Sought: Assuming the read and write miss penalties are the same and ignoring other write stalls, find the AMAT for the processor.

Solution: AMAT= Time for a hit + (Miss rate × Miss Penalty)

1) Time for a hit = cache access time = 1 clock cycles × 1ns clock cycle time = 1ns

2) Miss Penalty = 20 clock cycles × 1ns clock cycle time = 20ns

3) AMAT = 1ns + (0.05 × 20 ns) = 2 ns or 2 clock cycles