# CS2100 Computer Organization

## Tutorial #11: Cache – Draft Answers

11 November – 15 November 2024

1. A byte-addressed machine with a word size of 16 bits and address width of 32 bits has a direct-mapped cache with 16 blocks and a block size of 2 words.

   (a) Given a sequence of memory references as shown below, where each reference is given as a byte address in both decimal and hexadecimal forms, indicate whether the reference is a hit (H) or a miss (M). For a miss, also identify the type of miss it is (i.e., cold, conflict or capacity.) Assume that the cache is initially empty.

| Memory Address | | Hit (H) or Miss (M)? |
|---|---|---|
| (in decimal) | (in hexadecimal) | |
| 4 | 4 | |
| 92 | 5C | |
| 7 | 7 | |
| 146 | 92 | |
| 30 | 1E | |
| 95 | 5F | |
| 176 | B0 | |
| 93 | 5D | |
| 145 | 91 | |
| 264 | 108 | |
| 6 | 6 | |

   (b) Given the above sequence of memory references, fill in the final contents of the cache. Use the notation *M[i]* to denote the word at memory address *i*.

| Index | Tag value | Word 0 | Word 1 |
|---|---|---|---|
| 0 | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |
| 13 | | | |
| 14 | | | |
| 15 | | | |

   (c) Repeat (a) and (b) if the cache is instead a *two-way set associative cache* while having the same block and overall size.

   (d) (d) Repeat (a) and (b) if the cache is instead a *fully associative cache* while having the same block and overall size.

Answer:

(a) Address for 4 bytes and 2 bytes make up a word. So, any two addresses that share the upper 31 bits belong to the same word. A block contains two words. So, any two addresses that share the upper 30 bits are in the same block. Also, the four bits 2–5 form the block index for the 16 block directly mapped cache.

| Memory Address | | | Hit (H) or Miss (M)? |
|---|---|---|---|
| (in decimal) | (in hexadecimal) | (in binary) | |
| 4 | 4 | 0000 . . . 0000 00 00 01 00 | **M(cold)** |
| 92 | 5C | 0000 . . . 0000 01 01 11 00 | **M(cold)** |
| 7 | 7 | 0000 . . . 0000 00 00 01 11 | **H(same block as 0x4)** |
| 146 | 92 | 0000 . . . 0000 10 01 00 10 | **M(cold)** |
| 30 | 1E | 0000 . . . 0000 00 01 11 10 | **M(cold)** |
| 95 | 5F | 0000 . . . 0000 01 01 11 11 | **M(conflict with 0x1e)** |
| 176 | B0 | 0000 . . . 0000 10 11 00 00 | **M(cold)** |
| 93 | 5D | 0000 . . . 0000 01 01 11 01 | **H(same word as 0x5c)** |
| 145 | 91 | 0000 . . . 0000 10 01 00 01 | **H(same block as 0x92)** |
| 264 | 108 | 0000 . . . 0000 00 00 10 00 | **M(cold)** |
| 6 | 6 | 0000 . . . 0000 00 00 01 10 | **H(same word as 0x7)** |

The addresses are converted to binary as shown above. The index field (4 bits) is highlighted in yellow. Note that address 0x92 is a cold miss and not a conflict miss because 0x92 is newly brought in.

(b) Trace backwards.

| Index | Tag value | Word 0 | Word 1 |
|---|---|---|---|
| 0 | | | |
| 1 | 0x00000000 | M[0x4] | M[0x6] |
| 2 | 0x00000004 | M[0x108] | M[0x10a] |
| 3 | | | |
| 4 | 0x00000002 | M[0x90] | M[0x92] |
| 5 | | | |
| 6 | | | |
| 7 | 0x00000001 | M[0x5c] | M[0x5e] |
| 8 | | | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | 0x00000002 | M[0xb0] | M[0xb2] |
| 13 | | | |
| 14 | | | |
| 15 | | | |

(c) With a set size of 2, and the overall size kept the same, the number of sets would be half that of the direct-mapped cache above, i.e., 8 sets. The set index field (3 bits) is highlighted in yellow.

| Memory Address | | | Hit (H) or Miss (M)? |
|---|---|---|---|
| (in decimal) | (in hexadecimal) | (in binary) | |
| 4 | 4 | 0000 . . . 0000 000 0 01 00 | **M(cold)** |
| 92 | 5C | 0000 . . . 0000 010 1 11 00 | **M(cold)** |
| 7 | 7 | 0000 . . . 0000 000 0 01 11 | **H(same block as 0x4)** |
| 146 | 92 | 0000 . . . 0000 100 1 00 10 | **M(cold)** |
| 30 | 1E | 0000 . . . 0000 000 1 11 10 | **M(cold)** |
| 95 | 5F | 0000 . . . 0000 010 1 11 11 | **H(same block as 0x5c)** |
| 176 | B0 | 0000 . . . 0000 101 1 00 00 | **M(cold)** |
| 93 | 5D | 0000 . . . 0000 010 1 11 01 | **H(same word as 0x5c)** |
| 145 | 91 | 0000 . . . 0000 100 1 00 01 | **H(same block as 0x92)** |
| 264 | 108 | 0000 . . . 0000 000 0 10 00 | **M(cold)** |
| 6 | 6 | 0000 . . . 0000 000 0 01 10 | **H(same word as 0x7)** |

| Set Index | Set 0 | | | Set 2 | | |
|---|---|---|---|---|---|---|
| | Tag | Word 0 | Word 1 | Tag | Word 0 | Word 1 |
| 0 | | | | | | |
| 1 | 0x00000000 | M[0x4] | M[0x6] | | | |
| 2 | 0x00000008 | M[0x108] | M[0x10a] | | | |
| 3 | | | | | | |
| 4 | 0x00000004 | M[0x90] | M[0x92] | 0x00000005 | M[0xb0] | M[0xb2] |
| 5 | | | | | | |
| 6 | | | | | | |
| 7 | 0x00000002 | M[0x5c] | M[0x5e] | 0x00000000 | M[0x1c] | M[0x1e] |

(d) For fully associative caches, there is no index. All bits up to the block offset will be part of the tag, as highlighted in green below. The cache is of the same size, hence can accommodate 16 blocks.

| Memory Address | | | Hit (H) or Miss (M)? |
|---|---|---|---|
| (in decimal) | (in hexadecimal) | (in binary) | |
| 4 | 4 | 0000 . . . 0000 0000 01 00 | **M(cold)** |
| 92 | 5C | 0000 . . . 0000 0101 11 00 | **M(cold)** |
| 7 | 7 | 0000 . . . 0000 0000 01 11 | **H(same block as 0x4)** |
| 146 | 92 | 0000 . . . 0000 1001 00 10 | **M(cold)** |
| 30 | 1E | 0000 . . . 0000 0001 11 10 | **M(cold)** |
| 95 | 5F | 0000 . . . 0000 0101 11 11 | **H(same block as 0x5c)** |
| 176 | B0 | 0000 . . . 0000 1011 00 00 | **M(cold)** |
| 93 | 5D | 0000 . . . 0000 0101 11 01 | **H(same word as 0x5c)** |
| 145 | 91 | 0000 . . . 0000 1001 00 01 | **H(same block as 0x92)** |
| 264 | 108 | 0000 . . . 0000 0000 10 00 | **M(cold)** |
| 6 | 6 | 0000 . . . 0000 0000 01 10 | **H(same word as 0x7)** |

| Tag value | Word 0 | Word 1 |
|---|---|---|
| 0x00000001 | M[0x4] | M[0x6] |
| 0x00000017 | M[0x5c] | M[0x5e] |
| 0x00000024 | M[0x90] | M[0x92] |
| 0x00000007 | M[0x1c] | M[0x1e] |
| 0x0000002c | M[0xb0] | M[0xb2] |
| 0x00000042 | M[0x108] | M[0x10a] |

2. A machine with byte addresses and a word size of 32 bits and address width of 32 bits has a direct-mapped data cache with 4 blocks each consisting of 2 words.

   (a) Given the MIPS program below, and assuming that array A starts at memory location 0x1000 while array B starts at memory location 0x4010. Fill in the first 10 address requests seen at the data cache and indicate whether the reference is a hit (H) or a miss (M). Assume that the cache is initially empty.

```
start:
      la   $s0, A               #PC=0x100
      la   $s1, B               #PC=0x104
      li   $t0, 1               #PC=0x108
loop:
      slt  $t1, $t0, 1000       #PC=0x10c
      beq  $t1, $zero, end_loop #PC=0x110
      sll  $t2, $t0, 2          #PC=0x114
      add  $t3, $s0, $t2        #PC=0x118
      lw   $a0, 0($t3)          #PC=0x11c
      add  $t4, $s1, $t2        #PC=0x120
      lw   $a1, 0($t4)          #PC=0x124
      add  $v0, $a0, $a1        #PC=0x128
      sw   $v0, -4($t3)         #PC=0x12c
      addi $t0, $t0, 1          #PC=0x130
      j    loop                 #PC=0x134
end_loop:
```

   (b) Given the above program, fill in the final contents of the cache. Use the notation *M[i]* to denote the word at memory address *i*. (*i* may be in hexadecimal.)

   (c) What is the total number of data cache memory references, hits and misses after the execution of the above MIPS program?

Answer:

(a) `$t0` starts at 1. `$t2 = $t0 * 4`. So the first `lw` is from address `$t3 = $s0 + $t2 = 0x1004`. Next, `lw` is from `$t4 = $s1 + $t2 = 0x4014`. This is then followed by a `sw` to `-4($t3) = 0x1000`. This completes one iteration and `$t0` is incremented by 1. So we have the following sequence:

```
lw from  0x1004 - cold miss, index = 0, tag = 0x80
lw from  0x4014 - cold miss, index = 2, tag = 0x200
sw to    0x1000 - index = 0, tag = 0x80, HIT!
lw from  0x1008 - cold miss, index = 1, tag = 0x80
lw from  0x4018 - cold miss, index = 3, tag = 0x200
sw to    0x1004 - index = 0, tag = 0x80, HIT!
lw from  0x100c - index = 1, tag = 0x80, HIT!
lw from  0x401c - index = 3, tag = 0x200, HIT!
sw to    0x1008 - index = 1, tag = 0x80, HIT!
lw from  0x1010 - cold miss, index = 2, tag = 0x80
    . . .
```

(b) The trick is to work backwards. The loop goes from iteration 1 to 999 (inclusive). At iteration $i$, the 3 accesses are to:

`lw` from 0x1000 + $4i$
`lw` from 0x4010 + $4i$
`sw` from 0x1000 + $4(i - 1)$

In particular, for iteration 999, they are:

1. `lw` from 0x1000 + 4 * 999 = 0x1f9c (index = 3, tag = 0xfc)
2. `lw` from 0x4010 + 4 * 999 = 0x4fac (index = 1, tag = 0x27d)
3. `sw` to 0x1000 + 4 * 998 = 0x1f98 (index = 3, tag = 0xfc)

We can fill these into the final content of the cache. Since this is the last iteration, we can be sure that it has displaced whatever was in the cache index and remained there till the end.

| Index | Tag value | Word 0 | Word 1 |
|---|---|---|---|
| 0 | | | |
| 1 | 0x27d | M[0x4fa8] | M[0x4fac] |
| 2 | | | |
| 3 | 0xfc | M[0x1f98] | M[0x1f9c] |

For iteration 998, they are:

1. `lw` from 0x1000 + 4 * 998 = 0x1f98 (index = 3, tag = 0xfc)
2. `lw` from 0x4010 + 4 * 998 = 0x4fa8 (index = 1, tag = 0x27d)
3. `sw` to 0x1000 + 4 * 997 = 0x1f94 (index = 2, tag = 0xfc)

| Index | Tag value | Word 0 | Word 1 |
|---|---|---|---|
| 0 | | | |
| 1 | 0x27d | M[0x4fa8] | M[0x4fac] |
| 2 | 0xfc | M[0x1f90] | M[0x1f94] |
| 3 | 0xfc | M[0x1f98] | M[0x1f9c] |

For iteration 997, they are:

1. `lw` from 0x1000 + 4 * 997 = 0x1f94 (index = 2, tag = 0xfc)
2. `lw` from 0x4010 + 4 * 997 = 0x4fa4 (index = 1, tag = 0x27d)
3. `sw` to 0x1000 + 4 * 996 = 0x1f98 (index = 3, tag = 0xfc)

| Index | Tag value | Word 0 | Word 1 |
|---|---|---|---|
| 0 | 0x27d | M[0x4fa0] | M[0x4fa4] |
| 1 | 0x27d | M[0x4fa8] | M[0x4fac] |
| 2 | 0xfc | M[0x1f90] | M[0x1f94] |
| 3 | 0xfc | M[0x1f98] | M[0x1f9c] |

This would be the final state of the cache because we are working backwards through the iterations, these would be what remains, having displaced the earlier blocks in the same index.

(c) There are two `lw` followed by a `sw` in each iteration. We start with the simple one first. The `sw` basically hits on the blocks brought in by the first `lw`. Hence, executed 999 times, it will always hit. The first `lw` starts with cache index 0, and due to the way the address is incremented will cycle through 1, 2, 3, then back to 0. The second `lw` starts with index 2 and will cycle through 3, 0, 1, 2, etc. The key here is that the two does not interfere with each other. The two `lw`'s will get to "walk through" each word of the blocks they brought in before suffering a cold miss. So other than the first iteration, each will suffer 1 cold miss, then enjoy 1 hit before missing again since a block consists of 2 words. Hence each will have 998/2 hits and (998/2 + 1) misses (the '1' because of the first iteration).

Therefore, total number of data cache memory references = 3 * 998 = 2997.

Total number of hits = $\left(\frac{998}{2}\right)$ * 2 + 999 = 1498.

Total number of misses = $\left(\frac{998}{2}+1\right)$ * 2 = 1000. Or a miss rate of $\frac{1000}{2997}$ = 33.37%.

3. Suppose a (32 bit) MIPS processor has an instruction cache that is also direct mapped with 4 blocks each consisting of 2 words. Using the same program in Question 2, answer the following:

(a) Fill in the final contents of this cache assuming that start is at location 0x100 and each of the assembler pseudo-instructions (like `la` and `li`) also occupy 32 bits. You can use "PC=..." to indicate the word content of the cache.

(b) What is the miss rate in this case?

Answer:

(a) Working backwards, we get the following trace:

PC = 0x110 (index = 2, tag = 0x8)
PC = 0x10c (index = 1, tag = 0x8)
PC = 0x134 (index = 2, tag = 0x9)
PC = 0x130 (index = 2, tag = 0x9)
PC = 0x12c (index = 1, tag = 0x9)
PC = 0x128 (index = 1, tag = 0x9)
PC = 0x124 (index = 0, tag = 0x9)
PC = 0x120 (index = 0, tag = 0x9)
PC = 0x11c (index = 3, tag = 0x8)
PC = 0x118 (index = 3, tag = 0x8)
PC = 0x114 (index = 2, tag = 0x8)
PC = 0x110 (index = 2, tag = 0x8) – (repeat)

| Index | Tag value | Word 0 | Word 1 |
|---|---|---|---|
| 0 | 0x9 | PC=0x120 | PC=0x124 |
| 1 | 0x8 | PC=0x108 | PC=0x10c |
| 2 | 0x8 | PC=0x110 | PC=0x114 |
| 3 | 0x8 | PC=0x118 | PC=0x11c |

(b) We work forward in the following way until we get to a steady state.
Initialization:
1. PC = 0x100 (index = 0, tag = 0x8) cold miss
2. PC = 0x104 (index = 0, tag = 0x8) hit
3. PC = 0x108 (index = 1, tag = 0x8) cold miss

So we have 2 misses and 1 hit. `$t0` = 1 (iteration 0)
1. PC = 0x10c (index = 1, tag = 0x8) hit
2. PC = 0x110 (index = 2, tag = 0x8) cold miss
3. PC = 0x114 (index = 2, tag = 0x8) hit
4. PC = 0x118 (index = 3, tag = 0x8) cold miss
5. PC = 0x11c (index = 3, tag = 0x8) hit
6. PC = 0x120 (index = 0, tag = 0x9) cold miss
7. PC = 0x124 (index = 0, tag = 0x9) hit
8. PC = 0x128 (index = 1, tag = 0x9) cold miss
9. PC = 0x12c (index = 1, tag = 0x9) hit
10. PC = 0x130 (index = 2, tag = 0x9) cold miss
11. PC = 0x134(index = 2, tag = 0x9) hit

So we get 5 cold misses and 6 hits for this iteration. `$t0` = 2 (iteration 1)
1. PC = 0x10c (index = 1, tag = 0x8) conflict miss (since set 1 is now occupied by a block of tag 0x9. Furthermore, all subsequent misses are conflict misses due to the same reason.)
2. PC = 0x110 (index = 2, tag = 0x8) miss
3. PC = 0x114 (index = 2, tag = 0x8) hit
4. PC = 0x118 (index = 3, tag = 0x8) hit! (because this index is occupied by the same block brought in by the previous iteration and was not evicted.)
5. PC = 0x11c (index = 3, tag = 0x8) hit!
6. PC = 0x120 (index = 0, tag = 0x9) hit! (same reasoning as above.)
7. PC = 0x124 (index = 0, tag = 0x9) hit!
8. PC = 0x128 (index = 1, tag = 0x9) miss
9. PC = 0x12c (index = 1, tag = 0x9) hit
10. PC = 0x130 (index = 2, tag = 0x9) miss
11. PC = 0x134(index = 2, tag = 0x9) hit

So we get 4 cold misses and 7 hits for this iteration.
`$t0` = 3 (iteration 2) to `$t0` = 999 (iteration 998), we have the same as above (the steady state). Only the blocks in 1 and 2 suffers conflicts from different tags. The rest all hits. In other words, we get 998 * (4 misses and 7 hits).

Finally, when `$t0` = 1000, only the first two instructions (both misses) gets executed.

So if we sum it up, we get a total of (2 + 5 + 998 * 4 + 2) = 4,001 misses, and (1 + 6 + 998*7) = 6,993 hits, giving us a miss rate of $\frac{4001}{(4001+6993)}$ = 36.39%

4. Using your knowledge of Boolean circuits, produce a circuit (the simpler the better) that will check the Valid and Tag bits of a cache block to determine if it is a hit (1) or a miss (0). For simplicity, assume that tags are 20 bits long. Limit each gate to at most 4 inputs. Do not use adders. Suggestion: Use Logisim for easier replications.

   Answer: The idea is to use the XOR gate to compare the respective bits. In Figure 4, "Tx" are the respective tag bits in the cache, while "Ax" are the corresponding bits in the address. A hit is when all the cache bits are equal to the corresponding bits of the address. In addition, the entry must be valid, i.e., Valid == 1. The XOR will be '1' if the bits are *different*. We use a tree of ORs to aggregate the results and finally a NOR so that we will get a '1' on a hit and zero on a miss.

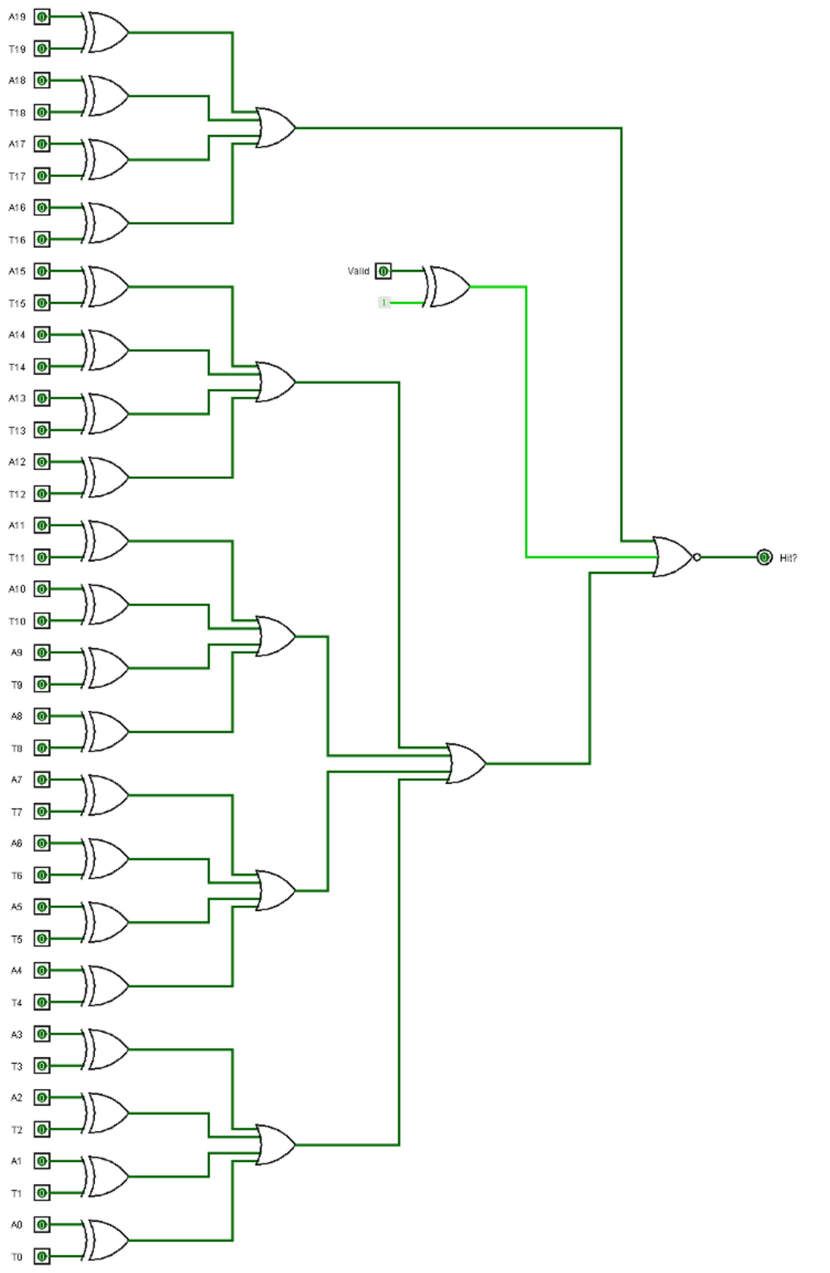

Figure 1: Circuit for question 4

5. **For Fun** ☺: Which was the first processor with a data cache?

   Answer: This is typically considered to be the IBM system/360 model 85. If interested, you can read the article here: `https://dl.acm.org/doi/10.1147/sj.71.0015`. It was published way back in 1968 ☺.