

## 6.004 Worksheet Questions

### L16 – Caches

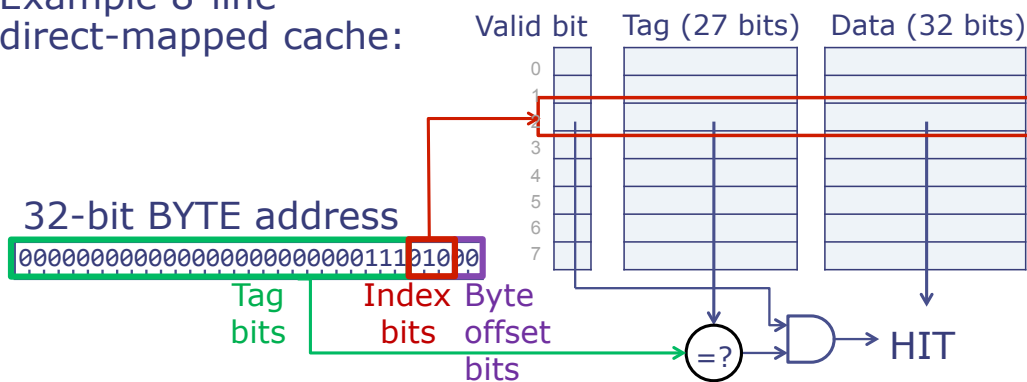
Keep the most often-used data in a small, fast SRAM (often local to CPU chip). The reason this strategy works: LOCALITY.

- *Temporal locality: If a location has been accessed recently, it is likely to be accessed (reused) soon*
- *Spatial locality: If a location has been accessed recently, it is likely that nearby locations will be accessed soon*

$$\text{AMAT(Average Memory Access Time)} = \text{HitTime} + \text{MissRatio} * \text{MissPenalty}$$

## Direct-Mapped Caches

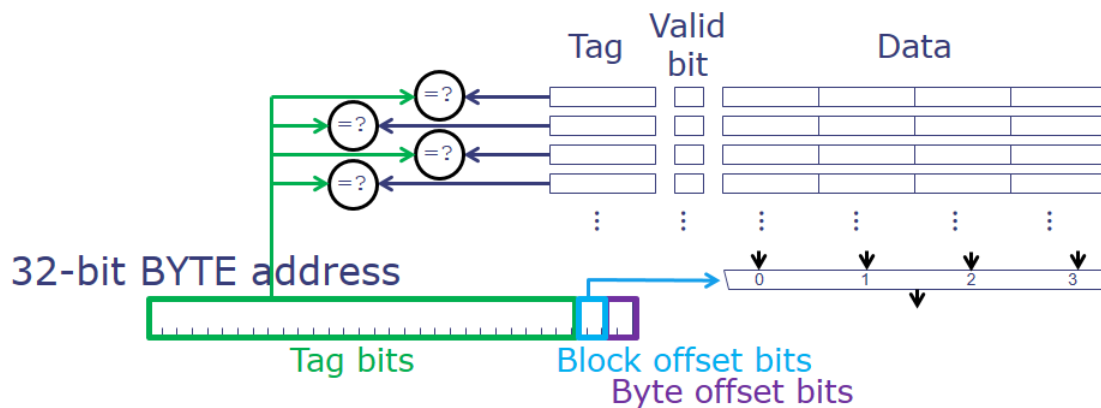
- Each word in memory maps into a single cache line
- Access (for cache with  $2^W$  lines):
  - Index into cache with  $W$  address bits (the **index bits**)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, HIT
- Example 8-line direct-mapped cache:



# Fully-Associative Cache

Opposite extreme: Any address can be in any location

- No cache index!
- **Flexible** (no conflict misses)
- **Expensive**: Must compare tags of all entries in parallel to find matching one



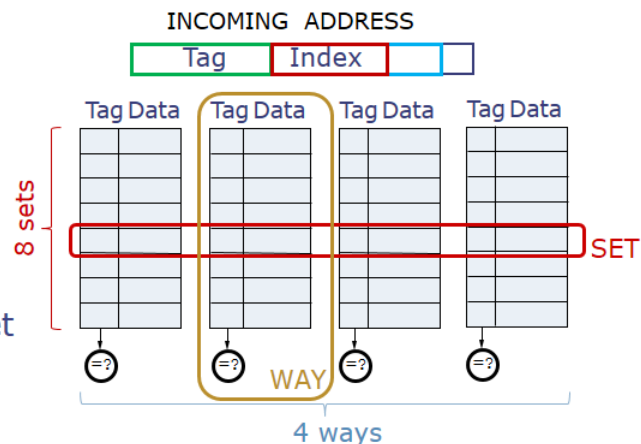
# N-way Set-Associative Cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses

- Nomenclature:

- # Rows = # Sets
- # Columns = # Ways
- Set size = #ways  
= "set associativity"  
(e.g., 4-way → 4 lines/set)

- Each address maps to only one set, but can be in any way within the set
- Tags from all ways are checked in parallel



- Fully-associative cache: Extreme case with a single set and as many ways as cache lines

## Cache summary

How to break up address:

- Bottom 2 bits of memory address are always 00 (for word alignment) – these 2 bits are called the byte offset.
- Number of block offset bits =  $\log_2$  (words per block)
- Number of index bits =  $\log_2$  (number of sets (rows))

Cache structure:

- In a direct mapped cache, number of sets = number of rows = number of cache lines
- In a set-associative cache, number of sets = number of rows = number of cache lines / # of ways (e.g., 2-way set associative cache has 2 cache lines in a set. A cache line = a block so it may contain multiple words).
- In a fully-associative cache, number of sets = 1 so there are no index bits.

### Problem 1 (parts A and B are from worksheet 15)

The RISC-V Engineering Team is working on the design of a cache. They've decided that the cache will have a **total of  $2^{10} = 1024$  data words**, but are still thinking about the other aspects of the cache architecture.

First assume the team chooses to build a direct-mapped cache with a block size of 4 words.

(A) Please answer the following questions:

**Number of lines in the cache: 256**

In a direct-mapped cache, each line has one block. If each block has 4 words and we want a total of 1024 words, we must have  $(1024 \text{ words}) / (4 \text{ words/block}) / (1 \text{ block/line}) = 256$  cache lines.

**Number of bits in the tag field for each cache entry: 20**

With 256 cache lines, we need 8 bits to index into them (index bits). This gives us one block. Within each block, we have 4 words. Each word is 4 bytes. Thus, within a block, we need 2 bits to index into the block and retrieve the word (block offset) and then 2 bits to index into the word to retrieve an individual byte (byte offset). Since memory addresses are 32 bits wide, we have  $32 - 8 - 2 - 2 = 20$  bits remaining for the tag.

(B) This cache takes 2 clock cycles to determine if a memory access is a hit or a miss and, if it's a hit, return data to the processor. If the access is a miss, the cache takes 20 additional clock cycles to fill the cache line and return the requested word to the processor. If the hit rate is 90%, what is the processor's average memory access time in clock cycles?

**Average memory access time assuming 90% hit rate (clock cycles): 4**

$$2 + (1 - 90\%) * 20 = 4$$

Now assume the team chooses to build a 2-way set-associative write-back cache with a block size of 4 words. The total number of data words in the entire cache is still 1024. The cache uses a LRU replacement strategy.

(C) Please answer the following questions:

$$1024 / 4 \text{ words} / 2 \text{ way} = 128 \text{ lines}$$

**Address bits used as block offset: A[3:2]**

**Address bits used as cache line index: A[10:4]**

**Address bits used for tag comparison: A[31:11]**

(D) To implement the LRU replacement strategy this cache requires some additional state for each set. How many state bits are required for each set?

**1 bit to select which of the two ways was most recently used for each set**

**Number of state bits needed for each set for LRU: 1**

To test this set-associative cache, the team runs the benchmark code shown on the right. The code sums the elements of a 16-element array. The first instruction of the code is at location 0x0 and the first element of the array is at location 0x10000. Assume that the cache is empty when execution starts and remember *the cache has a block size of 4 words*.

(E) How many instruction misses will occur when running the benchmark?

**Number of instruction misses when running the benchmark: 3**

Note: this explanation (which covers both 2E and 2F) is similar to that for problem 1E.

As we see in part (C), for each memory access, we look at bits [10:4] of the address to determine which index in the cache that particular memory address maps to. Remember that both instructions and the array are stored in memory. Thus, each instruction fetch is a memory access and is also cached along with all “normal” memory accesses in the form of lw instructions.

```
. = 0x0
mv x3, x0 // index
mv x1, x0 // sum
// x4 = 0x10000
lui x4, 0x10

L: add x5, x4, x3
   lw x2, 0(x5)
   add x1, x1, x2
   addi x3, x3, 4
   slti x2, x3, 64
   bnez x2, L
   unimp // halt

. = 0x10000
A: .word 0x1
   .word 0x2
   ...
   .word 0xF
   .word 0x10
```

In addition, note that each block in this cache contains four (4) words. Therefore, when we fetch a new line into the cache, we will be adding four instructions/array values into our cache. Because our instructions are mostly executed linearly and because our array is accessed in a strictly linear fashion, this pattern allows us to take huge advantage of spatial locality to reduce the number of instruction/data fetches that we must execute.

We start from the top with the mv instruction at address 0x0. Bits [10:4] = 0b0. Therefore, we check index 0 of the cache for the mv instruction. However, since our cache starts empty, there is nothing there (the valid bit == 0). This is a cache miss, so we load a cache line into that slot in way 1 of our cache. Since each line contains four words, we load in the instructions at addresses [0x0, 0x4, 0x8, 0xC] – i.e. the next three instructions in addition to the current instruction. Thus, the next three instructions are all cache hits. The fifth instruction, lw, is at address 0x10. Bits[10:4] = 0b1, so this instruction is cached at index 1. The pattern should be evident: every fourth instruction is cached at the next cache index.

We fetch and cache the lw instruction next at index 0b0100. The lw instruction accesses address 0x10000 (the first element of the array). Bits [10:4] of 0x10000 are 0b0. This is the first data read miss. Note that we have already loaded in something in our cache at that index – the first set of four instructions. However, since our cache has two (2) ways, we can use the other way to load in another line of 4 words of array values. Thus, we only have a data fetch miss every fourth iteration of the loop L.

Since there are 16 array values and nine (9) instructions, we have that there are  $16 // 4 = 4$  data misses and  $9 // 4 = 3$  instruction misses.

Here is the final cache picture:

Index	Contents							
	Way 0				Way 1			
0000 00	mv	mv	lui	add	Mem[0x100 00]	Mem[0x100 04]	Mem[0x100 08]	Mem[0x100 0C]
0000 01	lw	add	addi	slti	Mem[0x100 10]	Mem[0x100 14]	Mem[0x100 18]	Mem[0x100 1C]
0000 10	bnez	unimp	-	-	Mem[0x100 20]	Mem[0x100 24]	Mem[0x100 28]	Mem[0x100 2C]
0000 11	Mem[0x100 30]	Mem[0x100 34]	Mem[0x100 38]	Mem[0x100 3C]				

(F) How many data misses (i.e., misses caused by the memory access from the LD instruction) will occur when running the benchmark?

**Number of data misses when running the benchmark: 4**

(G) What's the exact hit rate when the complete benchmark is executed?

**Benchmark hit rate: 109/116**

# instruction fetches = 3 instructions before + 6 instructions/iteration of loop\*16 iterations+1 instruction after =100

# data fetches = 16

Misses = 7

Hits = 116-7=109



Since this is a 2-way cache, the instructions and data will never conflict (they will be stored in two different ways even if they have the same line index). Thus, at steady-state, there is a 100% hit ratio.

**Steady-state hit ratio (%): 100%**



### Problem 3 ★

Consider a 2-way set-associative cache where each way has 4 cache lines with a **block size of 2 words**. Each cache line includes a valid bit (V) and a dirty bit (D), which is used to implement a write-back strategy. The replacement policy is least-recently-used (LRU). The cache is used for both instruction fetch and data (LD,ST) accesses. Please use this cache when answering questions (A) through (D).

- (A) Using this cache, a particular benchmark program experiences an average memory access time (AMAT) of 1.3 cycles. The access time on a cache hit is 1 cycle; the miss penalty (i.e., additional access time) is 10 cycles. What is the hit ratio when running the benchmark program? You can express your answer as a formula if you wish:

**Hit ratio for benchmark program: 0.97**

$$\text{AMAT} = \text{hit\_time} + (1 - \text{hit\_ratio}) * \text{miss\_penalty}$$

$$1.3 = 1 + (1 - \text{HR}) * 10$$

- (B) The circuitry for this cache uses various address bits as the block offset, cache line index and tag field. Please indicate which address bits A[31:0] are used for each purpose by placing a “B” in each address bit used for the block offset, “L” in each address bit used for the cache line index, and “T” in each address bit used for the tag field.

**Fill in each box with “B”, “L”, or “T”**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	L	L	B	0	0

Block Size = 2Words = 8Byte: Block Offset 1bit, Byte Offset 2bit (2bits are 2'b00)  
 4 lines/each way = 4 sets cache: Index 2bits  
 Remaining bits are all tag bits: 27 bits

- (C) This cache needs room to store new data and based on the LRU replacement policy has chosen the cache line whose information is shown to the right for replacement. Since the current contents of that line are marked as dirty (D = 1), the cache must write some information back to main memory. What is the address of each memory location to be written? Please give each address in hex.

Way: 0  
 Cache line index: 3  
 Valid bit (V): 1  
 Dirty bit (D): 1  
 Tag field: 0x123

**Addresses of each location to be written (in hex): 0x2478, 0x247C**

Block offset: 1-bit value: 0b1 or 0b0; Byte offset: 2-bit value 0b00

Index: 2-bit 3 = 0b11

Tag: 27-bit 0x123 = 0b000 0000 0000 0000 0001 0010 0011

Address = {27'b tag, 2'b index, 1'b block offset, 2'b byte offset}

1) Block Offset (0b0): 0b000...00\_0010\_0100\_0111\_1000 = 0x2478

2) Block Offset (0b1): 0b000...00\_0010\_0100\_0111\_1100 = 0x247C

- (D) This cache is used to run the following benchmark program. The code starts at memory address 0; the array referenced by the code has its first element at memory address 0x200. First determine the number of memory accesses (both instruction and data) made during each iteration through the loop. Then estimate the steady-state average hit ratio for the program, i.e., the average hit ratio after many iterations through the loop.

```

. = 0
    mv x3, x0          // byte index into array
    mv x1, x0          // initialize checksum accumulator
loop:
    lw x2, 0x200(x3)    // load next element of array
    slli x1, x1, 1      // shift checksum
    addi x1, x1, 1       // increment checksum
    add x1, x1, x2       // include data value in checksum
    addi x3, x3, 4       // byte index of next array element
    slti x2, x3, 1000    // process 250 entries
    bnez x2, loop
    unimp               // halt

. = 0x200
array:
    ... array contents here ...

```

In steady state:

- Each loop fetches 7 instructions and 1 data (lw).
- Loop starts at instruction address 0x8=0b1000
- If you calculate the index bits for every two instructions (since each cache line contains two words) in the loop, you will see that none of them overlap. In addition, they can all be inserted into Way 0 of the cache. Thus, there are no instruction cache misses!

The other way of our cache can be used to load data from our array starting at address 0x200. Every time we encounter a new element of the array not in the cache, our cache load request will actually fetch two words (i.e. to contiguous elements in our array); thus, in the next iteration of our loop, we will not need to fetch again, as the next element is already in our cache.

Thus, the data fetch instruction only (lw) misses every other iteration of the while loop. Since there are 8 total fetches per while loop iteration, we see that one out of every  $2*8=16$  instructions is a miss.

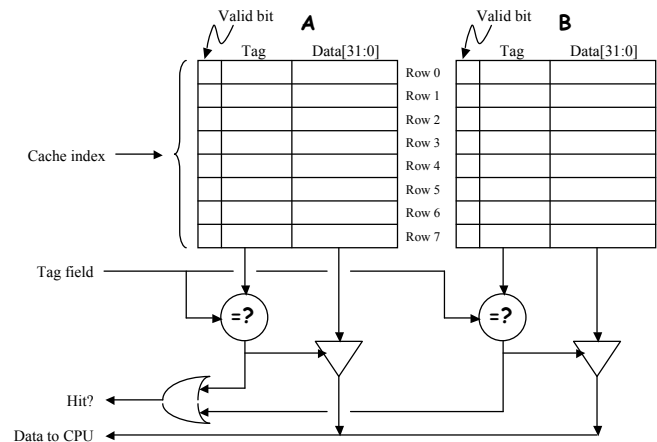
Steady-state hit ratio =  $(8-0.5)/8 = 15/16$

**Number of memory accesses made during each iteration of the loop: 8**

**Estimated steady-state average hit ratio: 15/16**

#### Problem 4

Consider the diagram to the right for a 2-way set associative cache to be used with our RISC-V processor. Each cache line holds a single 32-bit word of data along with its associated tag and valid bit (0 when the cache line is invalid, 1 when the cache line is valid).



- (A) The RISC-V produces 32-bit byte addresses, A[31:0]. To ensure the best cache performance, which address bits should be used for the cache index? For the tag field?

**address bits used for cache index: A[4:2]**

**address bits used for tag field: A[31:5]**

As always, 2 bits for byte offset (both 0)  
 1 word per block implies block offset = 0 bits (no need to select)  
 8 lines/way implies index = 3 bits  
 Remaining 27 bits are tag bits.

- (B) Suppose the processor does a read of location 0x5678. Identify which cache location(s) would be checked to see if that location is in the cache. For each location specify the cache section (A or B) and row number (0 through 7). E.g., **3A** for row 3, section A. If there is a cache hit on this access what would be the contents of the tag data for the cache line that holds the data for this location?

**cache location(s) checked on access to 0x5678: 6A and 6B**

**cache tag data on hit for location 0x5678 (hex): 0x2B3**

0x5678 in binary: 0b 0101 0110 011 1 1000  
 Line:  $3'b110 = 6$   
 Tag: 0b0010\_1011\_0011 = 0x2B3

Remember, all ways are checked in parallel when searching a set-associative cache.

- (C) Assume that checking the cache on each read takes 1 cycle and that refilling the cache on a miss takes an *additional* 8 cycles. If we wanted the *average* access time over many reads to be 1.1 cycles, what is the minimum hit ratio the cache must achieve during that period of time? You needn't simplify your answer.

$$1.1 = 1 + (1-HR)*8$$

**minimum hit ratio for 1.1 cycle average access time: 79/80**

(D) Estimate the approximate cache hit ratio for the following program. Assume the cache is empty before execution begins (all the valid bits are 0) and that an LRU replacement strategy is used. Remember the cache is used for both instruction and data (LD) accesses.

```

        . = 0
        addi x4, x0, 0x100
        mv x1, x0
        lui x2, 1           // x2 = 0x1000
loop:   lw x3, 0(x4)
        addi x4, x4, 4
        add x1, x1, x3
        addi x2, x2, -1
        bnez x2, loop
        sw x1, 0x100(x0)
        unimp              // halt

        . = 0x100
source:
        . = . + 0x4000 // Set source to 0x100, reserve 0x1000 words

```

**approximate hit ratio: 5/6**

The first instruction in the loop (lw) is at 0xC=0b1000. Extracting the index bits from this address, we see that the instructions within the loop body (lw - bnez) occupy lines 3-7 of one of the ways in the cache.

Thus, one way is always free for data. Each iteration causes a data fetch that is a cache miss (since there is only a single word per cache line -> no spatial locality). Thus, each iteration there are 5 instruction fetches (hits) and 1 data fetch (always a miss) .

(E) After the program of part (D) has finished execution what information is stored in row 4 of the cache? Give the addresses for the two locations that are cached (one in each of the sections) or briefly explain why that information can't be determined.

**Addresses whose data is cached in “Row 4”: 0x10 and 0x40F0**

(@ 0x10) addi x4,x4,4: Mapped to Row 4

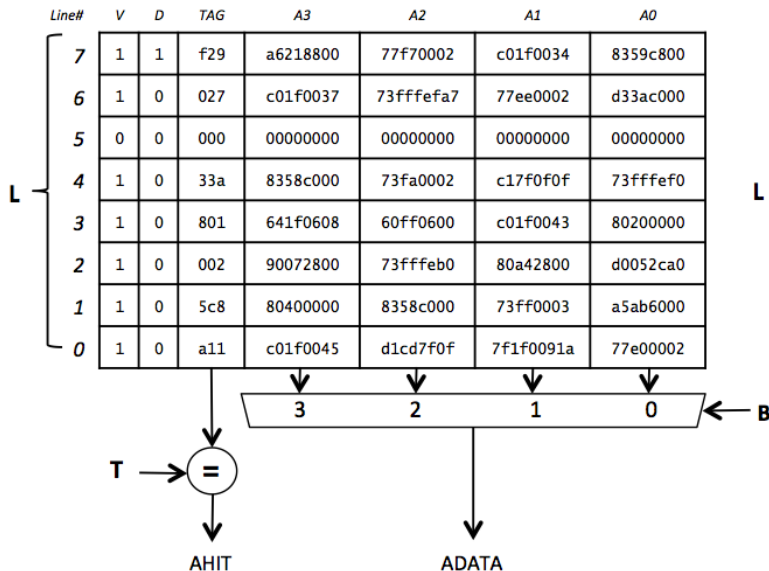
Data access ends at 0x4100 - 4 (Starts from 0x100 and does 0x1000 times: next element at +4)  
= 0x40FC @ Row 7

...  
**0x40F0 @ Row 4**  
 0x40F4 @ Row 5  
 0x40F8 @ Row 6  
 0x40FC @ Row 7  
 ...

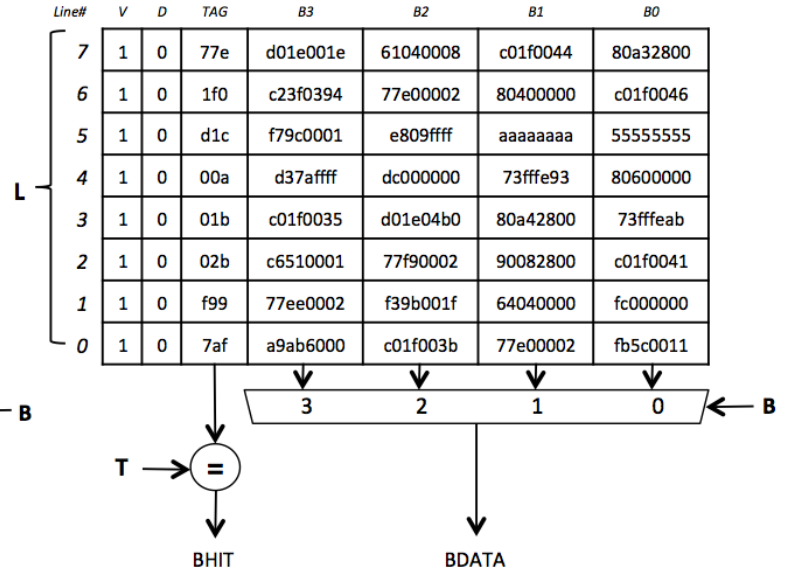
## Problem 5 ★

A standard unpipelined RISC-V is connected to a 2-way set-associative cache containing 8 sets, with a block size of 4 32-bit words. The cache uses a LRU replacement strategy. At a particular point during execution, a snapshot is taken of the cache contents, which are shown below. All values are in hex; **assume that any hex digits not shown are 0.**

Way #1



Way #2



- (A) The cache uses bits from the 32-bit byte address produced by the processor to select the appropriate set (L), as input to the tag comparisons (T) and to select the appropriate word from the data block (B). For correct and optimal performance what are the appropriate portions of the address to use for L, T and B? Express your answer in the form “A[N:M]” for N and M in the range 0 to 31, or write “CAN’T TELL”.

4 bytes/word -> 2-bit byte offset (as always)

4 word (16 Byte) cache block -> 2-bit block offset

8 rows per way (8-set cache) -> 3-bit index

Address -> {25'b tag, 3'b index, 2'b block offset, 2'b byte offset}

Address bits to use for L: **A[6:4]**

Address bits to use for T: **A[31:7]**

Address bits to use for B: **A[3:2]**

- (B) For the following addresses, if the contents of the specified location appear in the cache, give the location’s 32-bit contents in hex (determined by using the appropriate value from the cache). If the contents of the specified location are NOT in the cache, write “MISS”.

0xA1100: Tag 0x1422; Line 0; Word 0 (B0 or A0)

Contents of location 0xA1100 (in hex) or “MISS”: **MISS**

0x548: Tag 0x00A; Line 4; Word 2 (B2 or A2)

Contents of location 0x548 (in hex) or “MISS”: **0xDC000000**

- (C) Ignoring the current contents of the cache, is it possible for the contents of locations 0x0 and 0x1000 to both be present in the cache simultaneously?

2-ways, so yes, even though their index bits match

Locations 0x0 and 0x1000 present simultaneously (circle one): **YES** ... NO

- (D) Give a one-sentence explanation of how the D bit got set to 1 for Line #7 of Way #1. At what point should the D bit be reset to 0?

**One sentence explanation**

ST updated a value of a word in that line in cache, but hasn't yet been written back to memory (Dirty). It will be reset when that line is evicted from the cache and written to memory.

- (E) The following code snippet sums the elements of the 32-element integer array X. Assume this code is executing on a RISC-V processor with a cache architecture as described above and that, initially, the cache is empty, i.e., all the V bits have been set to 0. Compute the hit ratio as this program runs until it executes the *unimp* instruction, a total of  $2 + (6 \times 32) + 1 = 195$  instruction fetches and 32 data accesses.

**Hit ratio: 216/227**

```
. = 0
mv x4, x0          // loop counter
mv x1, x0          // accumulated sum

loop:
    slli x2, x4, 2   // convert loop counter to byte offset
    lw x3, 0x100(x2) // load next value from array
    add x1, x1, x3    // add value to sum
    addi x4, x4, 1    // increment loop counter
    slti x2, x4, 32   // finished with all 32 elements?
    bnez x2, loop     // nope, keep going

    unimp            // all done, sum in x1

. = 0x100
X: .word 1           // the 32-element integer array X
   .word 2
   ...
   .word 32
```

Cache stores 4 words per line: First four instructions (mv, mv, slli, lw) map to cache index “0” and the other 4 instructions (add, addi, slti, bnez) map to cache index “1”. Unimp is mapped to index “3”

For 32 words in the array, 4 are grouped into a cache block and stored in another way of the cache (total 8 lines)

3 instruction misses and 8 data misses (Compulsory – initially cache empty)

No more misses after that.

Total instruction fetches:  $2 + 6 \times (32) + 1 = 195$ , total data fetches: 32

$(195 + 32 - 3 - 8) / (195 + 32)$

## Problem 6 ★

After his geek hit single *I Hit the Line*, renegade singer Johnny Cache has decided he'd better actually learn how a cache works. He bought three RISC-V processors, identical except for their cache architectures:

- **Proc1** has a 64-line direct-mapped cache
- **Proc2** has a 2-way set associative cache, LRU, with a total of 64 lines
- **Proc3** has a 4-way set associative cache, LRU, with a total of 64 lines

Note that each cache has the same total capacity: 64 lines, each holding a single 32-bit **word** of data or instruction. All three machines use the same cache for data and instructions fetched from main memory.

Johnny has written a simple test progr

```
// Try a little cache benchmark
// Assume x7 = 0x2000 (data region A)
// Assume x8 = 0x3000 (data region B)
// Assume x9 = 16 (size of data regions in BYTES!)

. = 0x1000                                // start program here
P:   addi x6, x0, 1000                      // outer loop count
Q:   mv x3, x9                             // Loop index i (array offset)
R:   addi x3, x3, -4                        // i = i-1
      addi x9, x3, x7                      // x9 = address of A[i]
      addi x10, x3, x8                    // x10 = address of B[i]
      lw x1, 0(x9)                        // read A[i]
      lw x2, 0(x10)                       // read B[i]
      bnez x3, R                          // repeat many times
      addi x6, x6, -1
      bnez x6, Q
      unimp                               // halt
```

Johnny runs his program on each processor, and finds that one processor model outperforms the other two.

(A) Which processor model gets the highest hit ratio on the above benchmark?

3 regions: 0x1000 (instructions), 0x2000 (array A), 0x3000 (array B)

Circle one: Proc1 Proc2 **Proc3**

(B) Johnny changes the value of **B** in his program to **0x2000** (same as **A**), and finds a substantial improvement in the hit rate attained by one of the processor models (approaching 100%).

Which model shows this marked improvement?

2 regions (array A is now equal to array B), so 2 ways is sufficient.

Circle one: Proc1 **Proc2** Proc3

(C) Finally, Johnny moves the code region to 0x0 and the two data regions **A**, and **B** each to **0x0**, and sets **x9** to **64**. What is the TOTAL number of cache misses that will occur executing this version of the program on each of the processor models?

One region (code data) needs to fetch  $64/4 = 16$  elements once in all caches ("compulsory misses")

TOTAL cache misses running on Proc1: **16**; Proc2: **16**; Proc3: **16**

### Problem 7 ★

- (A) We would like to design a cache with an AMAT (average memory access time) of 1.5 cycles. Accessing our cache takes 1 cycle, and on a miss, it takes an *additional* 10 cycles to retrieve the data from main memory and update the cache. What does our hit ratio need to be in order to achieve the target AMAT?

$$\text{AMAT} = \text{HitTime} + \text{MissPenalty} * \text{MissRatio}$$

$$1.5 \text{ cycles} = 1 \text{ cycle} + (10 \text{ cycles}) * (1 - \text{HitRatio})$$

$$.05 = 1 - \text{HitRatio}$$

$$\text{Hit ratio} = \underline{\quad 0.95 \quad}$$

We choose to implement a 2-way set-associative cache with a block size of 4 (i.e. 4 words per line). The number of sets in the cache is 4. Assume that addresses and data words are 32 bits.

- (B) To ensure the best cache performance, which address bits should be used for the block offset, the cache index, and the tag field?

Address bits used for byte offset: A[   1   :   0   ]

Address bits used for block offset: A[   3   :   2   ]

Address bits used for cache index: A[   5   :   4   ]

Address bits used for tag field: A[  31  :   6  ]

- (C) Assuming the cache uses a writeback policy, what is the total number of bits per cache line?

Please show your work for partial credit.

$$\text{Bits per line} = 1 \text{ Valid bit} + 1 \text{ Dirty bit} + 26 \text{ Tag bits} + 4 * 32 \text{ Data bits}$$

$$= 28 + 128 \text{ bits}$$

$$= 156 \text{ bits}$$

$$\underline{\quad 156 \quad} \text{ bits}$$



We want to analyze the performance of this cache on the following assembly program, which iterates through a 1000-word array A and sets each element to  $A[i] = -A[i]$ . The base address of array A is 0x3000.

```
. = 0x100 // The following code starts at address 0x100
```

```
// Assume the following registers are initialized:
// x1=0 (loop index)
// x2=1000 (number of array elements)
// x3=0x3000 (base address of array A)
```

```
loop:
    slli x5, x1, 2    // x5 = byte offset of the ith element
    add x6, x5, x3    // x6 = address of A[i]
    lw x7, 0(x6)      // x7 = A[i]
    sub x7, x0, x7    // x7 = -A[i]
    sw x7, 0(x6)      // store A[i] = -A[i]
    addi x1, x1, 1    // increment i
    blt x1, x2, loop  // continue looping
```

(D) Below is the cache state the first time the program is about to enter the loop at loop. Assume that the cache uses a least-recently used (LRU) replacement policy, and that all cache lines in Way 1 are currently the least-recently used. Mark up the cache below to indicate the state of the cache immediately after one loop iteration (after executing the blt instruction for the first time). You do not need to specify the value of data words, but do specify the values of D (dirty bit), V (valid bit), and Tag.

#### Way 0

D	V	Tag	Word 0	Word 1	Word 2	Word 3
0 1	1	0x0 0xC0				
0	1	0x0				
0	1	0x0				
-	0	-				

#### Way 1

D	V	Tag	Word 0	Word 1	Word 2	Word 3
- 0	0 1	- 0x4				
- 0	0 1	- 0x4				
-	0	-				
-	0	-				

(F) How many instruction fetches and data accesses occur per iteration of the loop?

Number of instruction fetches: 7

Number of data accesses: 2

(G) After the program has been running for many loop iterations, what is the steady-state hit ratio for instruction fetches and data accesses?

All instructions fetches fit in Way 1. They are never replaced by data since they are always accessed more recently than the previous set of data. So in steady state, there is an instruction hit rate of 100%.

The data array is large enough that in steady state every time you access a new block of data you get 1 miss. However, when you get the miss, you bring in 4 consecutive data elements, each of which is accessed twice through a lw followed by a sw. So for every 8 data accesses, you get 1 miss, or a hit rate of 7/8.

Steady-state hit ratio for instruction fetches: 1

Steady-state hit ratio for data accesses: 7/8

## Problem 8

Anne and Ben just learned about caches in 6.004 and decided to design their own.

- (A) They would like to design a cache with an AMAT (average memory access time) of 3 cycles. Accessing the cache should take 1 cycle, and on a miss, it should take an *additional* 16 cycles to retrieve the data from main memory, update the cache, and return the requested word to the processor. What should their hit ratio be in order to achieve the target AMAT?

Hit ratio:   7/8  

Ben suggests implementing a 2-way set-associative cache with a block size of 4 (i.e. 4 words per line). The number of sets in the cache is 8. Assume that addresses and data words are 32 bits wide.

- (B) To ensure the best cache performance, which address bits should be used for the block offset, the cache index, and the tag field?

Address bits used for byte offset: A[  1   :   0  ]

Address bits used for block offset: A[  3   :   2  ]

Address bits used for cache index: A[  6   :   4  ]

Address bits used for tag field: A[ 31  :   7  ]

- (C) Anne agrees with the appropriateness of a 2-way set-associative implementation, but she suspects that a larger block size might result in a higher hit rate. Suppose the block size of the cache is doubled to 8. If the total number of data words in the cache remains unchanged, how would the number of cache lines change?

Change in # of cache lines (select one of the choices below):

UNCHANGED ... +1 ... -1 ... 2x ... **0.5x** ... CAN'T TELL

Ultimately, they choose to implement Ben's 2-way, 4-block cache. Below is a snapshot of the cache during the execution of some unknown code. The column labeled *Word x* corresponds to

the  $x^{\text{th}}$  word of the block. The V bit specifies whether or not the line is valid, and the D bit specifies whether or not the line is dirty.

	V	D	Tag	Word 0	Word 1	Word 2	Word 3
set 0	1	0	0x32	0x0A	0x1A	0x2A	0x3A
set 1	1	0	0x32	0x4B	0x5B	0x6B	0x7B
set 2	1	0	0x32	0x3C	0x2C	0x1C	0x0C
set 3	1	0	0x32	0x7D	0x6D	0x5D	0x4D
set 4	1	1	0x50	0x33	0x23	0x13	0x03
set 5	1	1	0x50	0x44	0x34	0x24	0x14
set 6	0	0	0x43	0x55	0x65	0x75	0x85
set 7	1	0	0x66	0x66	0x76	0x86	0x96

V	D	Tag	Word 0	Word 1	Word 2	Word 3
1	0	0x33	0x80	0x81	0x82	0x83
1	1	0x33	0xB4	0xB5	0xB6	0xB7
1	0	0x95	0xC3	0xC2	0xC1	0xC0
1	0	0x95	0xD3	0xD4	0xD5	0xD6
1	0	0x22	0x89	0x88	0x87	0x86
1	0	0xA0	0x92	0x93	0x94	0x95
1	1	0x37	0xF5	0xF6	0xF7	0xF8
1	1	0x18	0xA7	0xA8	0xA9	0xAA

(D) Would a load request to address 0x193C result in a hit or a miss? If it results in a hit, specify what value is returned; if it is a miss, write N/A.

Hit / Miss : Hit

Returned value if hit or N/A if miss: 0x4D

Anne and Ben want to analyze the performance of this cache on the following assembly program, which calculates the first 256 terms in the Fibonacci sequence and stores them in an array A. The base address of array A is 0x3000.

```
// Assume the following registers are initialized:
// x1 = 0 (initial loop index)
// x2 = 256 - 2 = 254 (number of Fibonacci elements to calculate)
// x3 = 0x3000 (base address of array A)

. = 0x100 // The following code starts at address 0x100
fibonacci:
    li x4, 1          // x4 = 1 (second element in sequence)
    sw x0, 0(x3)      // A[0] = 0
    sw x4, 4(x3)      // A[1] = 1
loop:
    slli x4, x1, 2    // x4 = byte offset of the ith element
    add x5, x4, x3    // x5 = address of A[i]
    lw x6, 0(x5)      // x6 = A[i]
    lw x7, 4(x5)      // x7 = A[i+1]
    add x6, x6, x7    // x6 = A[i] + A[i+1]
    sw x6, 8(x5)      // A[i+2] = x6
    addi x1, x1, 1    // increment i
    blt x1, x2, loop // continue looping
```

Answer the following questions about the behavior of the cache during execution of the above code. Assume that the cache uses a least recently used (LRU) replacement policy, that the cache is initially empty, and that all cache lines in Way 0 are currently the least-recently used.

(E) How many instruction fetches and data accesses occur per iteration of the loop?

Number of instruction fetches: 8

Number of data accesses: 3

(F) After the program has been running for many loop iterations, what is the steady-state hit ratio for instruction fetches and data accesses? *Hint: Note that in steady state each array element is accessed in multiple loop iterations.*

Steady-state hit ratio for instruction fetches: 1

Steady-state hit ratio for data accesses: 11/12

### Problem 9 (From Past Quizzes)

Dilvina and Saniel are analyzing 4.006 grade statistics and are performing some hefty calculations, so they suspect that a cache could improve their system's performance.

- (A) They are considering using a 2-way set-associative cache with a block size of 4 (i.e. 4 words per line). The cache can store a total of 64 words. Assume that addresses and data words are 32 bits wide. To properly make use of locality, which address bits should be used for the block offset, the cache index, and the tag field?

Address bits used for byte offset: A[  1   :   0  ]

Address bits used for *tag* field: A[ 31  :   7 ]

Address bits used for *block offset*: A[  3  :   2 ]

Address bits used for cache *index*: A[  6  :   4 ]

Block size of 4 → 2 bits for block offset  
64 words / 4 words per block = 16 blocks  
16 blocks / 2 ways = 8 sets → 3 bits for cache index  
 $32 - 3 - 2 - 2 = 25$  bits of tag

- (B) If Dilvina and Saniel instead used a direct-mapped cache with the same total capacity (64 words) and same block size (4 words), how would the following parameters in their system change?

Change in # of cache lines (select one of the choices below):

UNCHANGED ... +1 ... -1 ... **2x** ... 0.5x ... CAN'T TELL

Accepted Unchanged or 2x as solutions because 2-way cache also has  $\frac{1}{2}$  the number of sets as the direct mapped cache, but the number of cache lines is actually the same.

Change in # of bits in tag field (select one of the choices below):

UNCHANGED ... +1 ... **-1** ... 2x ... 0.5x ... CAN'T TELL

Block size of 4 → 2 bits for block offset  
64 words / 4 words per block = 16 blocks → 4 bits for cache line index  
 $32 - 4 - 2 - 2 = 24$  bits of tag

Ultimately, they decided that the 2-way set associative cache would probably have better performance for their application, so the remainder of the problem will be considering a 2-way set associative cache. Below is a snapshot of this cache during the execution of some unknown code. **V is the valid bit and D is the dirty bit of each set.**

Way 0							
			Word				
V	D	Tag	0	1	2	3	
0	1	0	0x32	0x0A	0x1A	0x2A	0x3A
1	1	0	0x32	0x4B	0x5B	0x6B	0x7B
2	1	0	0x32	0x3C	0x2C	0x1C	0x0C
3	1	0	0x32	0x7D	0x6D	0x5D	0x4D
4	1	1	0x50	0x33	0x23	0x13	0x03
5	1	1	0x50	0x44	0x34	0x24	0x14
6	0	0	0x43	0x55	0x65	0x75	0x85
7	1	0	0x66	0x66	0x76	0x86	0x96

Way 1							
			Word				
V	D	Tag	0	1	2	3	
0	1	0	0x33	0x80	0x81	0x82	0x83
1	1	0	0x33	0xB4	0xB5	0xB6	0xB7
2	1	0	0x95	0xC3	0xC2	0xC1	0xC0
3	1	0	0x95	0xD3	0xD4	0xD5	0xD6
4	1	1	0x22	0x89	0x88	0x87	0x86
5	1	1	0xA0	0x92	0x93	0x94	0x95
6	0	0	0x37	0xF5	0xF6	0xF7	0xF8
7	1	0	0x18	0xA7	0xA8	0xA9	0xAA

(C) Would the following memory accesses result in a hit or a miss? **If it results in a hit, specify what value is returned; if it is a miss, explain why in a few words or by showing your work.**

32-Bit Byte Address: 0x4AB4

Line index: 3

Tag: 0x 95

Block offset: 1

Returned value if hit / Explanation if miss: 0xD4

32-Bit Byte Address: 0x21E0

Line index: 6

Tag: 0x 43

Block offset: 0

Returned value if hit / Explanation if miss: miss, valid bit is 0

Dilvina and Sanieel want to analyze the performance of this same cache on the following assembly program, which averages the quiz 2 scores for the 225 students in 4.006. The students' scores are stored in an array A whose base address is at 0x3000.

```
// Assume the following registers are initialized:
// x1 = 0 (loop index)
// x2 = 225 (number of 4.006 students)
// x3 = 0x3000 (base address of array A)
// x6 = 0 (used for summing)

. = 0x0 // The following code starts at address 0x0
sum_loop:
    slli x4, x1, 2           // x4 = 4*i
    add x5, x4, x3           // x5 = address of A[i]
    lw x5, 0(x5)             // x5 = A[i]
    add x6, x6, x5           // x6 = sum(A[i:0])
    addi x1, x1, 1           // increment i
    blt x1, x2, sum_loop     // continue looping
divide_by_n:
    // divide by 225 here
```

Answer the following questions about the behavior of the cache during execution of the above code. Assume that the cache uses a least recently used (LRU) replacement policy, that the cache is initially empty, and that all cache lines in Way 0 are currently the least-recently used.

(D) How many instruction fetches and data accesses occur per iteration of the loop?

**Number of instruction fetches per loop iteration: 6**

**Number of data accesses per loop iteration: 1**

(E) What is the hit ratio for all memory accesses (both instruction fetches and data accesses) the first time through the loop?

First instruction misses, next 3 hit because block size is 4. Fifth instruction misses, sixth instruction hits. Data access is a miss.

**First loop iteration hit ratio: 4/7**

(F) After the program has been running for many loop iterations, what is the steady-state hit ratio for all memory accesses (both instruction fetches and data accesses)?

All instructions hit. Data access misses once every 4 iterations. So 1 miss per for iterations or 27/28 hit rate.

**Steady-state hit ratio: 27/28**

(G) Dilvina and Sanieel want to use the cache above in their memory hierarchy between the CPU and main memory. The cache takes 5 cycles to determine if a memory access is a miss or a hit, and the main memory in their system takes an additional 140 cycles for each memory access that reaches it. What is the steady-state AMAT of their memory system during the test that they ran above?



$$\begin{aligned} \text{AMAT} &= \text{hit\_time} + (1-\text{HR})(\text{miss penalty}) \\ &= 5 + (1/28)(140) = \end{aligned}$$

**AMAT:**   10