6.004 Recitation Problems
L15 – Caches

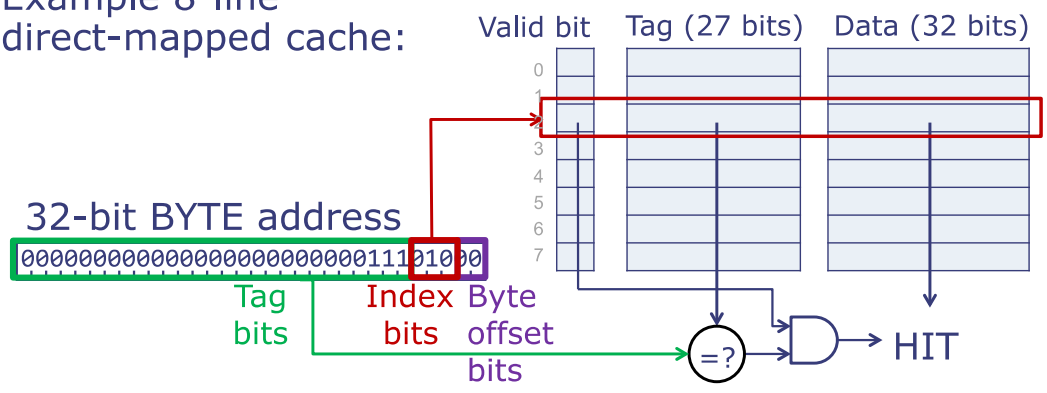
Keep the most often-used data in a small, fast SRAM (often local to CPU chip). The reason this strategy works: LOCALITY.

- *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*

AMAT(Average Memory Access Time) = HitTime + MissRatio * 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:



April 9, 2020 MIT 6.004 Spring 2020 L15-8

Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indices → 6 index bits

Read Mem[0x400C]

0100 0000 0000 1100

TAG: 0x40
INDEX: 0x3
BYTE OFFSET: 0x0

HIT, DATA 0x42424242

Would 0x4008 hit?

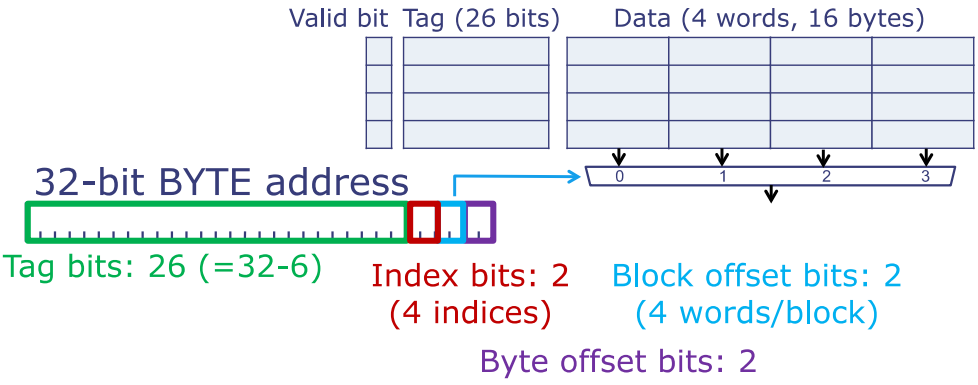
INDEX: 0x2 ⇒ tag mismatch
⇒ MISS

	Valid bit	Tag (24 bits)	Data (32 bits)
0	1	0x000058	0xDEADBEEF
1	1	0x000058	0x00000000
2	1	0x000058	0x00000007
3	1	0x000040	0x42424242
4	0	0x000007	0x6FBA2381
⋮	⋮	⋮	⋮
63	1	0x000058	0xF7324A32

Part of the address (index bits) is encoded in the location
Tag + Index bits unambiguously identify the data's address

Block Size

- Take advantage of spatial locality: Store multiple words per data line
  - Always fetch entire block (multiple words) from memory
  - Another advantage: Reduces size of tag memory!
  - Potential disadvantage: Fewer indices in the cache
- Example: 4-block, 16-word direct-mapped cache



Problem 1. ★

(A) The timing for a particular cache is as follows: checking the cache takes 1 cycle. If there's a hit the data is returned to the CPU at the end of the first cycle. If there's a miss, it takes 10 *additional* cycles to retrieve the word from main memory, store it in the cache, and return it to the CPU. If we want an average memory access time of 1.4 cycles, what is the minimum possible value for the cache's hit ratio?

Minimum possible value of hit ratio: _0.96_

AMAT = HitTime + MissRatio * MissPenalty
1.4=1+(1-HitRatio) * 10
=> HitRatio=0.96

(B) If the cache block size, i.e., words/cache line, is doubled but the total number of data words in the cache is unchanged, how will the following cache parameters change? Please circle the best answer.

# of block offset bits: UNCHANGED … +1 … -1 … 2x … 0.5x … CAN'T TELL

# of tag bits: UNCHANGED … +1 … -1 … 2x … 0.5x … CAN'T TELL

# of cache lines: UNCHANGED … +1 … -1 … 2x … 0.5x … CAN'T TELL

The block offset bits tell us how far into the retrieved block we need to index to get the requested data. Because the block size doubled, there are twice as many indices needed, so we need to add an extra offset bit.

Because the block size is doubled, we would store twice as many data words, but the problem stipulates that the total number of data words in the cache is unchanged. Therefore, the number of cache lines must be cut in half since each cache line is now twice as big.

Because the number of cache lines has halved, we need one fewer index bits. In effect, we traded one index bit for one offset bit, so the total number of tag bits remains unchanged.

Consider a direct-mapped cache with 64 total data words with 1 word/cache line. This cache architecture is used for parts (C) through (F).

(C) If cache line number 5 is valid and its tag field has the value 0x1234, what is the address in main memory of the data word currently residing in cache line 5?

Main memory address of data word in cache line 5: 0x123414

The memory address has 32 bits total which will be divided up into bits for the tag, index, block offset and byte offset.

Since there are 64 total words with 1 word/cache line, there must be 64 cache lines. The index bits must fully index into these 64 lines. This requires 6 bits. Since the address maps to cache line 5, we know that the 6 index bits must be 000101.

Each cache line holds one block which is specified to be one word (32 bits or 4 bytes). Since there is only one word in each block, we need no bits (i.e. 0 bits) for this offset.

Since each byte has its own address, we need four possible offsets into each block. Thus, the two bits are sufficient for the byte offset.

This leaves 24 bits for the tag, which is specified to be 0x001234.

Tag: 24 bits, Index: 6 bits (000101), Block offset: 0 bits, Byte offset: 2 bits (00)

The program shown on the right repeatedly executes an inner loop that sums the 16 elements of an array that is stored starting in location 0x310.

The program is executed for many iterations, then a measurement of the cache statistics is made during one iteration through all the code, i.e., starting with the execution of the instruction labeled `outer_loop:` until just before the next time that instruction is executed.

```
  . = 0             // tell assembler to start at
                    // address 0
outer_loop:
  addi x4, x0, 16   // initialize loop index J
  mv x1, x0         // x1 holds sum, initially 0

loop:               // add up elements in array
  subi x4, x4, 1    // decrement index
  slli x2, x4, 2    // convert to byte offset
  lw x3, 0x310(x2)  // load value from A[J]
  add x1, x1, x3    // add to sum
  bne x4, x0, loop  // loop until all words are summed

  j outer_loop      // perform test again!
```

(D) In total, how many instruction fetches occur during one complete iteration of the outer loop? How many data reads?

Number of instruction fetches: 83

Instruction fetch = 2+5*16+1=83
2 instructions before `loop` label are executed once. The 5 instructions in the loop are executed 16 times, and `j outer_loop` is executed once.

Number of data reads: 16

The inner loop runs for 16 iterations. lw is called once per iteration.

(E) How many instruction fetch misses occur during one complete iteration of the outer loop? How many data read misses? Hint: remember that the array starts at address 0x310.

Number of instruction fetch misses: ____4____

Number of data read misses: ____4____

As we see in part (C), For each memory access, we look at bits [7:2] 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.

Also note that this program has been executing for multiple cycles already. Thus, many slots in the cache are already populated. We need to figure out which ones will result in cache misses. As we walk through the program for the first time, we will not count initializing the cache contents as cache misses.

We start from the top with the addi instruction at address 0x0. Bits [7:2] = 0b0. Therefore, the addi instruction is cached at index 0. As we will see later, this instruction was already present in the cache. This is not a cache miss. The next instruction, mv, is at address 0x04. Bits[7:2] = 0b1, so this instruction is cached at index 1. The next instruction, subi, is at address 0x08. Bits[7:2] = 0b10, so this instruction is cached at index 2. The pattern should be evident: each subsequent instruction is cached at the next cache index.

We fetch and cache the lw instruction next at index 0b0100. The lw instruction accesses address 0x310 (the first element of the array). Bits [7:2] of 0x310 are 0b100. But the cache line at index 0b0100 already holds the lw instruction which must be evicted. This is the first data read miss. The next two add and bne instructions are cached in indices 0b0101 and 0b0110 respectively.

On the next iteration of the inner loop, the subi and slli instructions are already cached. The lw instruction is not. This is the first instruction fetch miss. We fetch the instruction and cache it into index 0b0100, evicting the previously cached memory contents at address 0x310 from the last iteration. Next, we look at the next element of the array at address 0x314. This gets cached at index 0b101, evicting the cached add instruction (the second data read miss). Almost immediately, we need to fetch the add instruction again (the second instruction fetch miss). This evicts the memory contents of 0x314 which we just cached. At this point, observe the caching pattern. All cached instructions in indices 0b0100 through 0b10011 (the cache indices containing the elements of the array) will be evicted once, replaced with array elements, and ultimately restored in a subsequent iteration. This leads to 4 instruction fetch misses and 4 data read misses.

As a final note, notice that on the next iteration of the outer loop, most of the cache indices will already contain the correct values which is the case if this code has been running for a while. This is why we did not treat all the cache initializations as cache misses – because previous runs had already set the cache.

Index	Contents
0000	`addi`
0001	`mv`
0010	`subi`

0011	slli
0100	lw Mem[0x310] lw
0101	add Mem[0x314] add
0110	bne Mem[0x318] bne
0111	j outer_loop Mem[0x31C] j outer_loop
1000	Mem[0x320]
1001	Mem[0x324]
1010	Mem[0x328]
…	…

(F) What is the hit ratio measured after one complete iteration of the outer loop?

Hit ratio: **91/99**

The total number of memory accesses is the sum of the instruction fetches and data reads (83 + 16 = 99). There were 4 instruction fetch misses and four data read misses, so the total number of cache hits is 83 – 4 + 16 – 4 = 91. The hit ratio is therefore 91/99.

Problem 2.

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 words) / (4 words/block) / (1 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 words/2 way = 128 lines

Address bits used as byte 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*.

```
. = 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
```

(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.

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			
000000	mv	mv	lui	add	Mem[0x10000]	Mem[0x10004]	Mem[0x10008]	Mem[0x1000C]
000001	lw	add	addi	slti	Mem[0x10010]	Mem[0x10014]	Mem[0x10018]	Mem[0x1001C]
000010	bnez	unimp	-	-	Mem[0x10020]	Mem[0x10024]	Mem[0x10028]	Mem[0x1002C]
000011	Mem[0x10030]	Mem[0x10034]	Mem[0x10038]	Mem[0x1003C]				

(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

Problem 3.

Assume, the program shown on the right is being run on a RISC-V processor with a cache with the following parameters:

- **2-way set-associative**
- **block size of 2**, i.e., 2 data words are stored in each cache line
- total number of data words in the cache is **32**
- **LRU** replacement strategy

```
. = 0x240          // start of program
test:
  addi x4, x0, 16 // initialize loop index J
                  // to size of array
  mv x1, x0       // x1: sum

loop:             // add up elements in array
  subi x4, x4, 1  // decrement index
  slli x2, x4, 2  // convert to byte offset
  lw x3, 0x420(x2)// load value from A[J]
  add x1, x1, x3  // add to sum
  bnez x4, loop   // loop N times

  j test          // perform test again!

// allocate space to hold array
. = 0x420
A: .word A[0]
   .word A[1]
   …
```

(A) The cache will divide the 32-bit address supplied by the processor into four fields: 2 bits of byte offset, B bits of block offset, L bits of cache line index, and T bits of tag field. Based on the cache parameters given above, what are the appropriate values for B, L, and T?

2 words per line; log_2(2) = 1

value for B: 1

8 lines per ways

value for L: 3

value for T: 32-6=26

(B) If the SLLI instruction is resident in a cache line, what will be its cache line index? the value of the tag field for the cache?

Instruction address = base + instruction_index * 4 = 0x240 + 3*4 = 0x24C

0x24C=0b00000000000000000000001001_001_100

Cache line index for SLLI when resident in cache: 1

Tag field for SLLI when resident in cache: 0x9

(C) Given that the code begins at address 0x240 and the array begins at address 0x420, and that there are 16 elements in the array as shown in the code above, list *all* the values j ($0 \le j < 16$) where the location holding the value A[j] will map to the same cache line index as the `SLLI` instruction in the program.

The array addresses that overlap are:

0x448=0b00000000000000000000010001_001_000

and

0x44C=0b00000000000000000000010001_001_100

These correspond to the 10th and 11th elements of the array.

List all j where A[j] have the same cache line index as SLLI: 10, 11

(D) If the outer loop is run many times, give the steady-state hit ratio for the cache, i.e., assume that the number of compulsory misses as the cache is first filled are insignificant compared to the number of hits and misses during execution.

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 4. ★

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

AMAT = hit_time + (1 - hit_ratio) * miss_penalty
1.3 = 1 + (1-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.

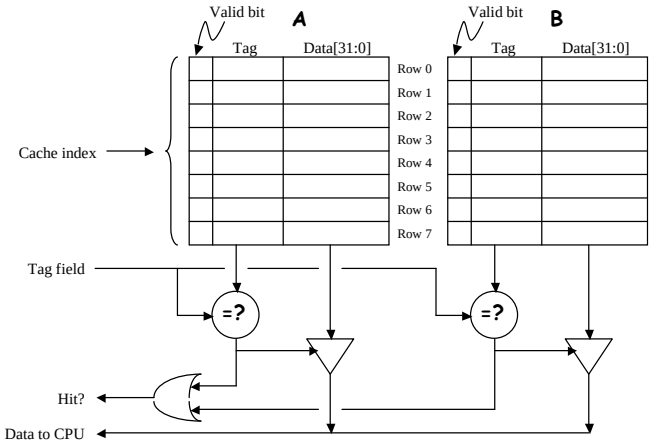
Steady-state hit ratio = (8-0.5)/8 = 15/16

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

Estimated steady-state average hit ratio: 15/16

Problem 5.

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 10**00
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 amiss) .

(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

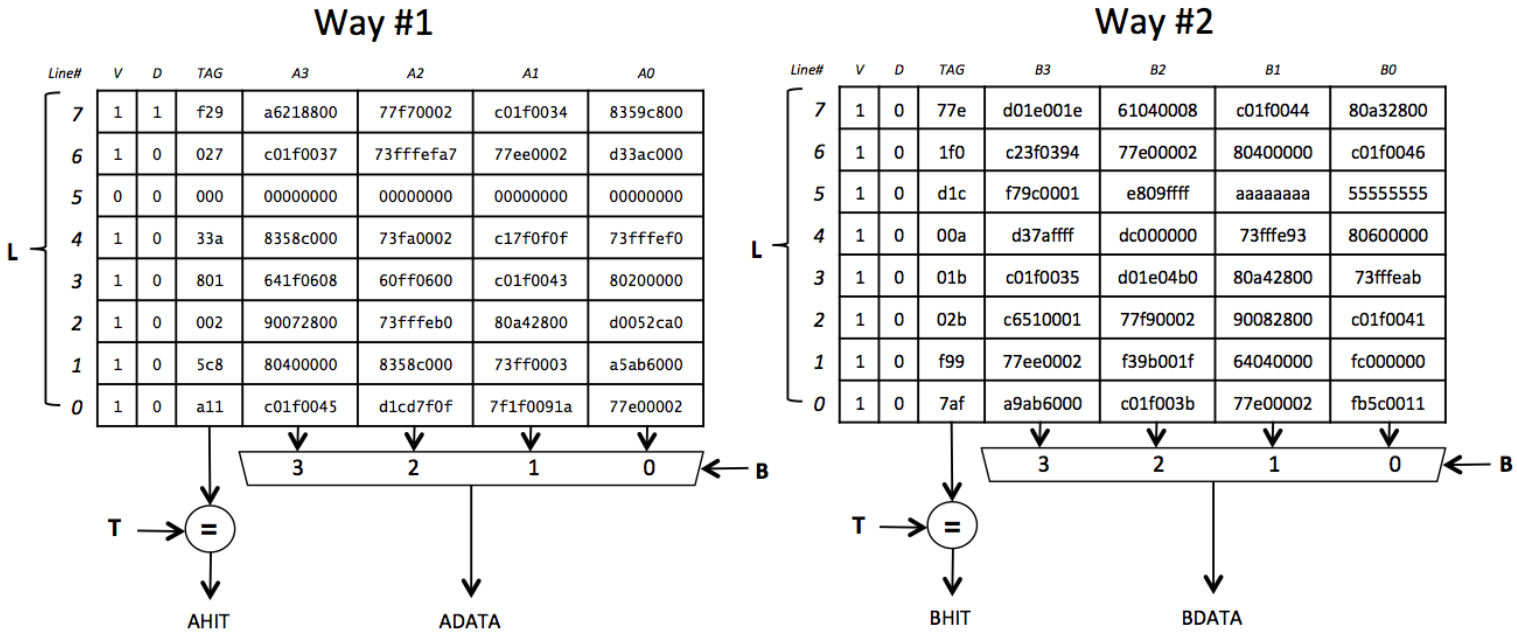
(@ 0x10) addi x4,x4,4: Mapped to Row 4

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 6. ★

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**.



(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*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*(32) + 1 = 195, total data fetches: 32

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

Problem 7. ★

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
      addi x6, x6, -1         // repeat many times
      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