CS 61C Fall 2022 — Caches — Discussion 9

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 We cannot use a 1KB cache in a 32-bit system because it's too small and cannot contain all possible addresses.

False. The purpose of the cache is not to hold every possible piece of memory at the same time, but rather to hold some parts of it only, so a 1KB cache is not "too small".

1.2 If a piece of data is both in the cache and in memory, reading it from cache is faster than reading from memory.

True. The cache is smaller and faster than memory.

1.3 Caches see an immediate improvement in memory access time at program execution.

False. A cache starts off 'cold', and required loading in values in blocks at first directly from memory, forcing compulsory misses. This can be somewhat alleviated by the use of a hardware prefetcher, that uses the current pattern of misses to predict and prefetch data that may be accessed later on.

1.4 Any cache miss that occurs when the cache is full is a capacity miss.

False. When the cache is full, you can still get compulsory misses (when a block of data is put in the cache for the first time).

1.5 Increasing cache size by adding more blocks always improves (increases) hit rate for all programs.

False. Whether this improves the hit rate for a given program depends on the characteristics of the program. As an example, it is possible for a program that only consists of a loop that runs through an array once to have each access be separated by more than one block (say, the block size is 8B, but we have an integer array and accessing every fourth element, so our access are separated by 16B). This makes every miss a compulsory miss, and there is no way for us to reduce the number of compulsory misses just by adding more blocks to our cache.

1.6 Decreasing block size for a cache to increase the number of blocks held by the cache improves the program speed for all programs.

False. This question is similar to the one above, in that the answer to it depends on the program that is running. If we have a program with a for loop that loops through continuous memory (like an array), having a bigger blocks size and fewer blocks might be helpful, as the single blocks will holds more continuous data. For example, lets say cache A has 10 lines and a block size of 8 bytes, while cache B has 20 lines with a a block size of 4 bytes and the array we loop through has 80 characters. Cache A in this case will have 10 cache misses and 70 hits, while Cache B will have 20 misses and 60 hits.

2 Hazards Review: CPI

Consider the following C program, which is compiled into RISC-V code on the right:

```
int func(int* arr) {
    int sum = 0;
    for(int i = 0; i < 10; i++) {
        sum += arr[i];
    }
    arr[0] = sum / 8;
    return sum;
}
```

```
    # a0: arr
    1. add t0, x0, x0  # t0: sum
    2. add t1, x0, x0  # t1: i
    3. addi t3, x0, 10
loop:
    4. bge t1, t3, end
    5. slli t2, t1, 2
    6. addi t2, t2, a0
    7. lw t2, 0(t2)
    8. add t0, t0, t2
    9. addi t1, t1, 1
    10.jal x0, loop
end:
    11.srai t2, t0, 3
    12.sw t2, 0(a0)
    13.add a0, t0, x0
    14.jalr x0, ra, 0
```

Assume that we are using the 5-stage pipeline from before with no optimizations (no forwarding, no double-pumping, and no branch prediction), but we'll always fetch the correct next instruction for jump instructions.

2.1 How many stalls do we need for a data hazard between two adjacent instructions? How many penalty cycles do we need to stall for a control hazard?

For data hazards, the second instruction's ID stage needs to wait until after the first instruction's WB stage to execute, resulting in 3 stalled cycles. For control hazards, if the branch is taken, then the second instruction's IF stage needs to wait until after the first instruction's MEM stage to execute, resulting in 3 stalled cycles.

2.2 How many (RISC-V) instructions are executed through the execution of the compiled program?

We start with the three instructions before the loop, then execute the instructions between loop and end 10 times, then execute the bge instruction once more before finishing with the four instructions at the end, resulting in a total of $3+10\times 7+1+4 = 78$ instructions.

2.3 How many cycles does it take to execute the compiled program with our 5-stage pipelined CPU? Ignore the first 4 cycles where the pipeline isn't completely filled yet for now (i.e. start counting from when the first instruction is in its WB stage).

Once the pipeline is completely filled, each instruction will take only one cycle, except for stalls, so let's analyze where we need to stall in the program.

3→4: data hazard on t3, 3-cycle stall executed once. (There is also a data hazard on t1 between 2→4, but we don't need to stall extra for it since we're already stalling for 3→4).

5→6: data hazard on t2, 3-cycle stall executed 10 times.

6→7: data hazard on t2, 3-cycle stall executed 10 times.

7→8: data hazard on t2, 3-cycle stall executed 10 times.

9→4: data hazard on t1, 2-cycle stall executed 10 times. The execution here will always be 9→10→4, where t1 is used two instructions after being set, so we only need to stall two cycles each time.

4→11: control hazard! Whenever the branch is taken, we need to stall for 3 cycles, and this happens only once.

11→12: data hazard on t2, 3-cycle stall executed once.

Therefore, the total stall cycles we need are $3 + (3 + 3 + 3 + 2) \times 10 + 3 + 3 = 119$ cycles. Add one cycle per instruction for executing the instructions normally to get $119 + 78 = 197$ total cycles.

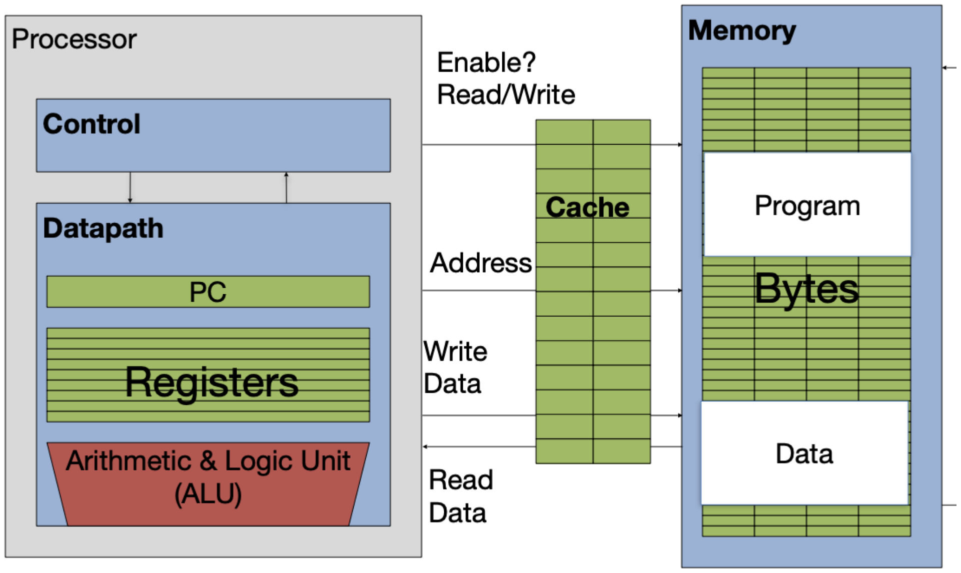
2.4 Based on the previous two parts, what is the CPI (cycles per instruction) of our CPU on this compiled program?

Add the first 4 cycles where the pipeline is being filled up to our answer in 2.3: $\frac{197+4}{78} = 2.58$

For additional review, you may attempt 2.3 and 2.4 with forwarding.

3 Understanding T/I/O

We use caches to make our access to data faster. When working with main memory (RAM), the main problem faced is the fact that access to the main memory is very slow. In fact, modern processors take about 100 instructions cycles or more to access the main memory, meaning memory accesses become the bottleneck of our programs. Caches help fix this problem for us - they hold a portion of the data in main memory, that we might access again later on. They are closer to the processor in the memory hierarchy, and thus accessing a cache is much faster than accessing the main memory.



As seen above, the access to cache is the middle step between the CPU asking for a memory bit, and the actual main memory access - if the data is not found in the cache, only then is main memory accessed. This way unnecessary trips to main memory are avoided. One important detail is that caches are much smaller in size than main memory - this is why we have to be efficient in what we hold in caches. When we are saving data in caches, we need to be as efficient as possible. In order to do this, we make use of locality. We have two different kinds of locality to consider.

> **Temporal Locality:** If we have accessed a piece of information recently, it is possible that we will access it again. So, we hold this data in the cache.
> **Spatial Locality:** If we have accessed a memory location recently, it is probable that we will access the neighbouring addresses as well. So, we also keep the neighbouring addresses within the cache. An example is array accesses - if we access the 0th element of an array, it is probable we will also access the 1st one.

Note that caches hold the data in blocks that have a size equal to the block size of the cache.

When working with caches, we have to be able to break down the memory addresses we work with to understand where they fit into our caches. There are three fields:

Tag - Used to distinguish different blocks that use the same index. Number of bits: (# of bits in memory address) - Index Bits - Offset Bits
Index - The set that this piece of memory will be placed in. Number of bits: $\log_2$(# of indices)
Offset - The location of the byte in the block. Number of bits: $\log_2$(size of block)

Given these definitions, the following is true:

$\log_2(\text{memory size}) = \text{address bit-width} = \#\text{ tag bits} + \#\text{ index bits} + \#\text{ offset bits}$

Another useful equality to remember is:

$$\text{cache size} = \text{block size} * \text{num blocks}$$

$\boxed{3.1}$ Assume we have a direct-mapped byte-addressed cache with capacity 32B and block size of 8B. Of the 32 bits in each address, which bits do we use to find the index of the cache to use?

We can determine the number of index bits we need from the number of sets our cache has. Since our cache is direct-mapped, the number of sets is the same as the number of blocks, so we just need to figure out how many blocks our cache has. Using the equality from above, we see that num blocks = cache size/block size, so our cache has $32/8 = 4$ blocks. We need $\log_2(4) = 2$ bits to differentiate the 4 blocks, so we have 2 index bits.
In order to determine where exactly the index bits are, we need to calculate the number of offset bits and tag bits we have. The number of offset bits is just dependent on the block size, so since our blocks are size 8B, we need $\log_2(8) = 3$ bits to differentiate the 8 bytes in the block, so we have 3 offset bits.
our offset bits take up the least significant bits, with the index bits being the set of next most significant bits. Denoting the most significant bit (MSB, on the left) as 31 and the least significant bit (LSB, on the right) as 0, having 3 offset bits means our index bits start at bit 3, and thus we use bits 3 and 4 as the index bits.

$\boxed{3.2}$ Which bits are our tag bits? What about our offset?

The offset (in this case) is the 3 least significant bits, so reusing the convention from the previous quesiton, the offset bits are bits 0, 1, and 2. Our tag is the remaining high-order bits, so our tag bits are bits 5-31.

3.3 Classify each of the following byte memory accesses as a cache hit (H), cache miss (M), or cache miss with replacement (R). Tip: Drawing out the cache can help you see the replacements more clearly.

Address	T/I/O	Hit, Miss, Replace
`0x00000004`		
`0x00000005`		
`0x00000068`		
`0x000000C8`		
`0x00000068`		
`0x000000DD`		
`0x00000045`		
`0x00000004`		
`0x000000C8`		

Ignore miss types (compulsory/conflict/capacity) until Q4.

```
0x00000004     Tag 0, Index 0, Offset 4: M, Compulsory
0x00000005     Tag 0, Index 0, Offset 5: H
0x00000068     Tag 3, Index 1, Offset 0: M, Compulsory
0x000000C8     Tag 6, Index 1, Offset 0: R, Compulsory
0x00000068     Tag 3, Index 1, Offset 0: R, Conflict
0x000000DD     Tag 6, Index 3, Offset 5: M, Compulsory
0x00000045     Tag 2, Index 0, Offset 5: R, Compulsory
0x00000004     Tag 0, Index 0, Offset 4: R, Capacity
0x000000C8     Tag 6, Index 1, Offset 0: R, Capacity
```

Note that the M and R distinction here is for student understanding, and that the cache doesn't behave differently for these cases.

4 The 3 C's of Cache Misses

4.1 Go back to question 3 and classify each M and R as one of the 3 types of misses described below:

1. Compulsory: First time you ask the cache for a certain block. A miss that must occur when you first bring in a block. Reduce compulsory misses by having longer cache lines (bigger blocks), which bring in the surrounding addresses along with our requested data. Can also pre-fetch blocks beforehand using a hardware prefetcher (a special circuit that tries to guess the next few blocks that you will want).
2. Conflict: Occurs if, hypothetically, you went through the **entire** string of accesses with a fully associative cache and wouldn't have missed for that specific access. Increasing the associativity or improving the replacement policy would remove the miss. Note that Conflict misses tend to occur with inefficient cache usage, compared to Capacity misses.
3. Capacity: Capacity misses are independent of the associativity of your cache. If you hypothetically ran the **entire** string of memory accesses with a fully

associative cache of the same size as your cache, and it was a miss for that specific access, then this miss is a capacity miss. The only way to remove the miss is to increase the cache capacity.

Note: The test you can use to see if a miss is a conflict miss is the same as the test you can use to see if a miss is a capacity miss.

See solutions for Q3.

5 Code Analysis

Given the follow chunk of code, analyze the hit rate given that we have a byte-addressed computer with a total memory of **1 MiB**. It also features a **16 KiB** Direct-Mapped cache with **1 KiB** blocks. Assume that your cache begins cold.

```
#define NUM_INTS 8192   // 2^13
int A[NUM_INTS];        // A lives at 0x10000
int i, total = 0;
for (i = 0; i < NUM_INTS; i += 128) {
    A[i] = i;           // Line 1
}
for (i = 0; i < NUM_INTS; i += 128) {
    total += A[i];      // Line 2
}
```

5.1 How many bits make up a memory address on this computer?

We take $\log_2(1\text{ MiB}) = \log_2(2^{20}) = 20$.

5.2 What is the T:I:O breakdown?

Offset $= \log_2(1\text{ KiB} = \log_2(2^{10}) = 10$
Index $= \log_2(\frac{16\text{ KiB}}{1\text{ KiB}}) = \log_2(16) = 4$
Tag $= 20 - 4 - 10 = 6$

5.3 Calculate the cache hit rate for the line marked Line 1:

The integer accesses are $4 * 128 = 512$ bytes apart, which means there are 2 accesses per block. The first accesses in each block is a compulsory cache miss, but the second is a hit because `A[i]` and A[i+128] are in the same cache block. Thus, we end up with a hit rate of **50%**.

5.4 Calculate the cache hit rate for the line marked Line 2:

The size of A is $8192 * 4 = 2^{15}$ bytes. This is exactly twice the size of our cache. At the end of Line 1, we have the second half of A inside our cache, but Line 2 starts with the first half of A. Thus, we cannot reuse any of the cache data brought in from Line 1 and must start from the beginning. Thus our hit rate is the same as Line 1 since we access memory in the same exact way as Line 1. We don't have to consider cache hits for total, as the compiler will most likely store it in a register. Thus, we end up with a hit rate of **50%**.