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.

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.

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

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

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

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.

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?

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

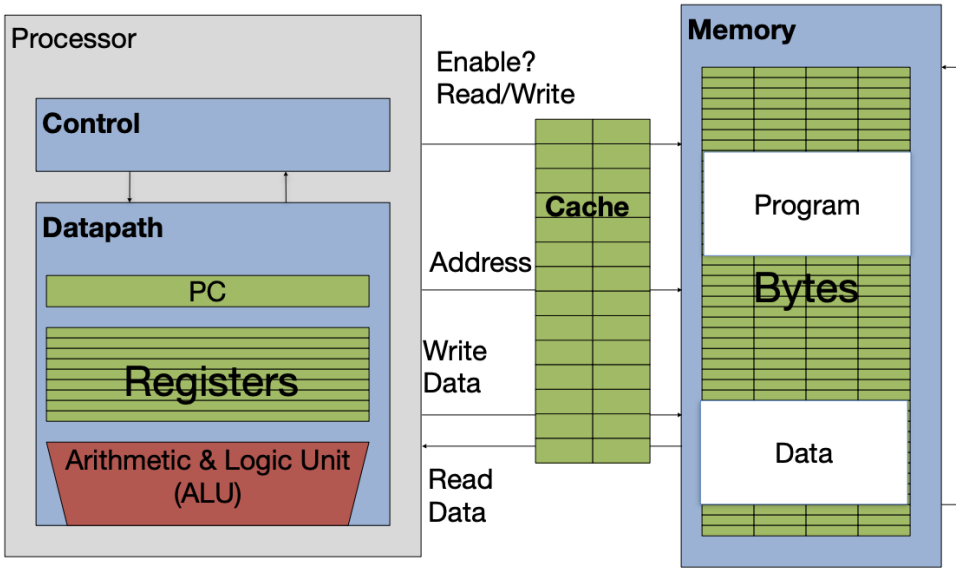
2.3 How many cycles does it take to execute the compiled program with our 5-stage pipelined CPU? Ignore the first 4 cycles where the pipeline isn't completely filled yet for now (i.e. start counting from when the first instruction is in its WB stage).

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

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:

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

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?

3.2 Which bits are our tag bits? What about our offset?

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`		

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.

Note: There are many different ways of fixing misses. The name of the miss doesn't necessarily tell us the best way to reduce the number of misses.

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?

5.2 What is the T:I:O breakdown?

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

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