CS M151B Homework 9

Charles Zhang

March 7, 2022

Problem 1

***I Amdahl-ighted with Tradeoffs*: Given the following problems, suggest one solution and give one drawback of the solution. Be brief, but specific.**

<u>EXAMPLE</u>
Problem: long memory latencies
Solution: ***Caches***
Drawback: ***when the cache misses, the latency becomes worse due to the cache access latency.***

Problem: too many capacity misses in the data cache
Solution: Expand the size of the data cache
Drawback: Longer hit times

Problem: too many control hazards
Solution: Resolve the branch addresses earlier in the pipeline
Drawback: Creates more data hazards in the case that the branch instruction is dependent on a prior instruction

Problem: our carry lookahead adder is too slow
Solution: Use a hybrid design with carry-select/ripple carry to reduce delays generated by increased fan-ins
Drawback: Higher cost of design and silicon

Problem: we want to be able to use a larger immediate field in the MIPS ISA
Solution: Increase the size of all instructions
Drawback: Unnecessarily bloats the size of many instructions, resulting in more use of instruction memory

Problem: the execution time of our CPU with a single-cycle datapath is too high
Solution: Pipeline the datapath
Drawback: Results in a much higher complexity of the pipeline, including the introduction of hazards

Problem 2

***Hazard a Guess?*: Assume you are using the 5-stage pipelined MIPS processor, with a three-cycle branch penalty. Further assume that we always use predict not taken. Consider the following instruction sequence, where the `bne` is taken once, and then not taken once (so 7 instructions will be executed total):**

```
Loop:   lw $t0, 512($t0)
        lw $t1, 64($t0)
        bne $s0, $t1, Loop
        sw $s1, 128($t0)
```

Assuming that the pipeline is empty before the first instruction:

a) Suppose we do not have any data forwarding hardware – we stall on data hazards. The register file is still written in the first half of a cycle and read in the second half of a cycle, so there is no hazard from `WB` to `ID`. Calculate the number of cycles that this sequence of instructions would take.

A three-cycle branch penalty tells us the branch resolves in `MEM`. The second `lw` is dependent on the first `lw`. The `bne` is dependent on the second `lw`. Knowing this, we can say that this sequence takes:

$$\boxed{\text{21 cycles}}$$

b) How many cycles would this sequence of instructions take with data forwarding hardware?

Data forwarding would allow dependent instructions following `lw`s to execute one cycle earlier, while the `lw` is in `MEM` rather than `WB`. As a result, this sequence takes:

$$\boxed{\text{17 cycles}}$$

Problem 3

***More $ More Problems*: Find the data cache hit or miss stats for a given set of addresses. The data cache is a 1KB, direct mapped cache with 64-byte blocks. Find the hit/miss behavior of the cache for a given byte address stream, and label misses as compulsory, capacity, or conflict misses. All blocks in the cache are initially invalid.**

Since each cache block is 64B = 2^6B, we know that each address has 6 offset bits. In addition, we know that the next 4 bits are index bits because $\frac{2^{10}}{2^6} = 2^4$.

Address in Binary	Cache Hit or Miss	Cache Miss Type
...0011011010000	Miss	Compulsory
...0001011100000	Miss	Compulsory
...0000011010000	Miss	Compulsory
...0011011100000	Miss	Conflict
...0001011100000	Miss	Conflict
...0000011010000	Hit	N/A
...0011011100000	Miss	Conflict

Problem 4

***The Trouble with TLBs*: Consider an architecture with 32-bit virtual addresses and 1GB of physical memory. Pages are 32KB and we have a TLB with 64 sets that is 8-way set associative. The data and instruction caches are 8KB with 16B block sizes and are direct mapped – and they are both virtually indexed and physically tagged. Assume that every page mapping (in the TLB or page table) requires 1 extra bit for storing protection information. Answer the following:**

a) How many pages of virtual memory can fit in physical memory at a time?

$$\frac{\text{1GB}}{\text{32KB/page}} = \frac{2^{30}}{2^5 \times 2^{10}} \text{ pages} = \boxed{2^{15} \text{ pages}}$$

b) How large (in bytes) is the page table?

We know there can be 2^{15} pages in physical memory, which means there are 2^{15} PTEs in the page table. A PTE is composed of a single protection bit (as stated by the problem), and a PPN. Since there is 1GB = 2^{30}B of addressable space in physical memory, a physical address must be 30 bits long. 15 bits are accounted for by the page offset, which means the PPN must be 15 bits long. Therefore, the total page table size is:

$$16 \text{ bits/PTE} \times 2^{15} \text{ PTEs} = 2\text{B/PTE} \times 2^{15} \text{ PTEs} = \boxed{2^{16}\text{B}}$$

c) What fraction of the total number of page translations can fit in the TLB?

The TLB has 64 sets, which means it can store 2^6 address translations. This gives a fraction of:

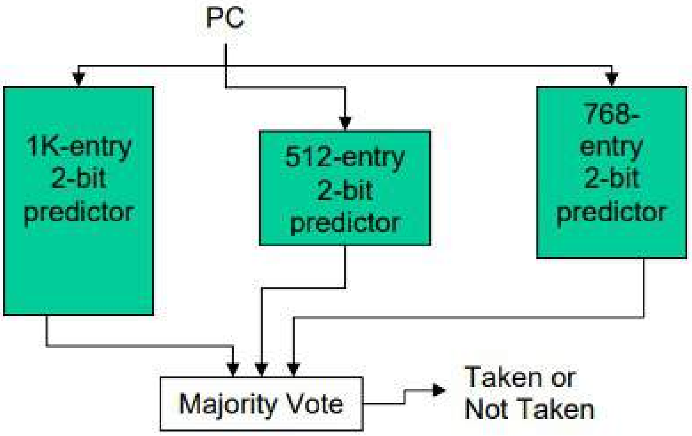
$$\frac{2^6}{2^{15}} = \boxed{\frac{1}{2^9}}$$

d) What bits of a virtual address will be used for the index to the TLB? Specify this as a range of bits – i.e. bits 4 to 28 will be used as the index. The least significant bit is labeled 0 and the most significant bit is labeled 31

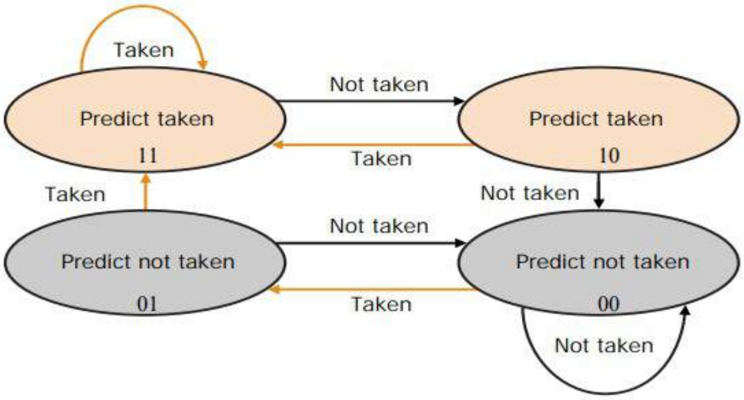
The TLB is indexed by the VPN, which consists of the top bits of the VA, specifically, those bits that are not in the offset. Since the page size is 2^{15}B, we know the least significant 15 bits act as the page offset. Therefore, the virtual page number is made up of bits 16 through 31.

Problem 5

A Branch Too Far: One difficulty in designing a branch predictor is trying to avoid cases where two PCs with very different branch behavior index to the same entry of a 2-bit branch predictor. This is called destructive aliasing. One way around this is to use multiple 2-bit branch predictors with different sizes. This way, if two PCs index to the same entry in one predictor, they will not likely index to the same entry in the other predictor. We will evaluate a scheme with three 2-bit branch predictors – each with a different number of entries. The three predictors will be accessed in parallel, and the majority decision of the predictors will be chosen. So if two predictors say taken and the other predictor says _not taken_, the majority decision will be _taken_. The scheme looks like this:



Each predictor has the following FSM:

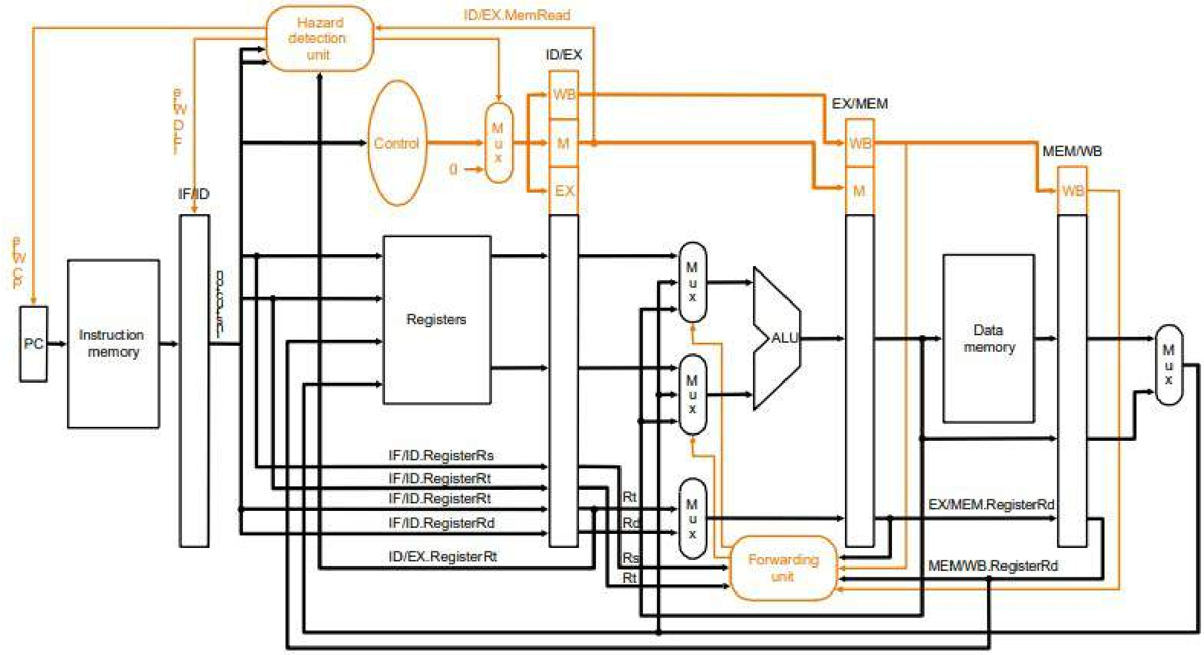


Evaluate the performance of this prediction scheme on the following sequence of PCs. The table shows the address of the branch and the actual direction of the branch (taken or not taken). You get to fill in whether or not the branch predictor would guess correctly or not. Each node of the FSM is marked with the 2-bit value representing that state. Assume that all predictors are initialized to `00`. To find an index into a predictor, assume we use the simplified branch indexing formula: `index = PC % predictor_size`. The symbol `%` represents the modulo operator. `predictor_size` will be different according to the predictor.

PC	Actual Direction	Correctly Predicted?
128	Taken	No
640	Not Taken	Yes
1152	Not Taken	Yes
128	Taken	No
640	Taken	No
1152	Not Taken	No
128	Taken	No
640	Not Taken	Yes
1152	Not Taken	Yes
128	Taken	No
640	Taken	No
1152	Not Taken	Yes

Problem 6

***With Friends Like These...*: Consider the scalar pipeline we have explored in class:**



a) Suppose 10% of instructions are stores, 15% are branches, 25% are loads, and the rest are R-type. 30% of all loads are followed by a dependent instruction. We have full forwarding hardware on this architecture. We use a predict not taken branch prediction policy and there is a 2 cycle branch penalty. This means that the PC is updated at the end of the `EX` stage – after the comparison is made in the ALU. One third of all branches are taken. There is an instruction cache with a single cycle latency and a miss rate of 10% and a data cache with a single cycle latency and a miss rate of 20%. We have an L2 cache that misses 5% – it has a 10 cycle latency – and memory has a 100 cycle latency. Find the TCPI for this architecture.

$$\text{TCPI} = \text{BCPI} + \text{MCPI}$$

$$\text{BCPI} = \text{Base CPI} + \text{Load-Use Penalty} + \text{Branch Misprediction Penalty}$$

$$\text{BCPI} = 1 + (0.25)(0.3)(1) + (0.15)(0.66)(2) = 1.273$$

$$\text{MCPI} = \text{I-cache Miss Penalty} + \text{D-cache Miss Penalty}$$

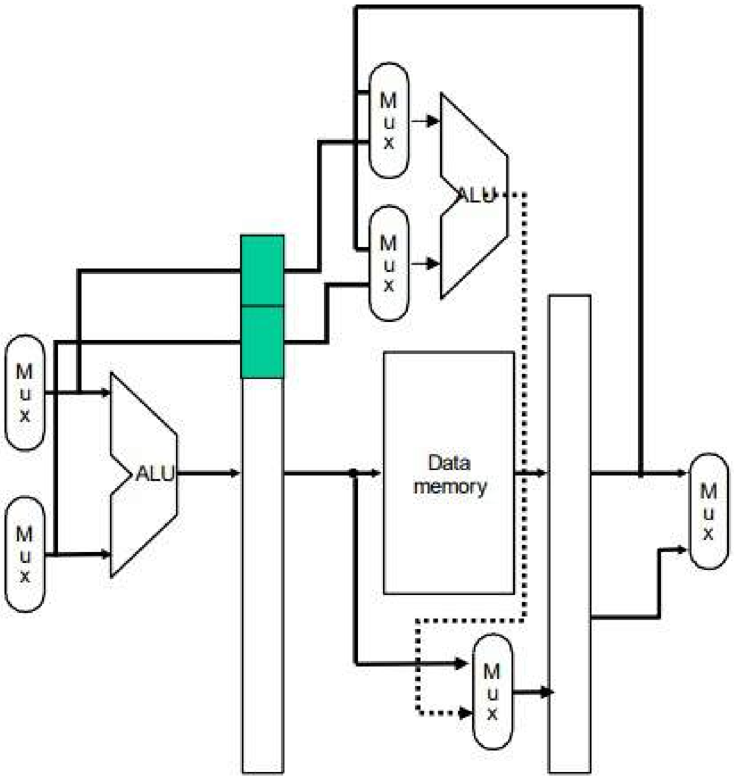
$$\text{I-cache Miss Penalty} = (1)(0.1)(10 + (0.05)(100)) = 1.5$$

$$\text{D-cache Miss Penalty} = (0.35)(0.2)(10 + (0.05)(100)) = 1.05$$

$$\text{MCPI} = 1.5 + 1.05 = 2.55$$

$$\text{TCPI} = 1.27 + 2.55 = \boxed{3.82}$$

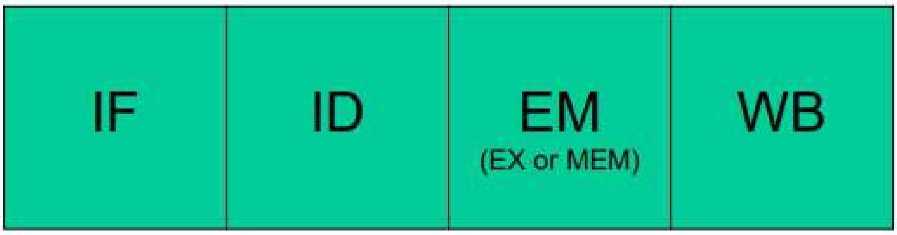
b) Your friend has a flash of brilliance – "I know a way to get rid of stalls in this pipeline. The reason we have to stall now is because a load can have a dependent instruction follow it through the pipeline, and we cannot forward the load's data until the end of the `MEM` stage – but the dependent instruction needs it at the beginning of the `EX` stage. So what if we add another ALU that recomputes what we did in `EX` if the instruction before it is a load and it is dependent on the load?" This ALU will be in the memory stage of the pipeline as shown below in this simplified picture:



Is your friend right or wrong? If they are wrong, give an example of when we would still need to stall.

They are wrong. Although the new design may fix the load-use hazard, there would still be a need to stall in the case where there is a cache miss in the I-cache or D-cache, as both misses would take more than a cycle to recover. As a result, following instructions would need to be stalled until the instruction/data is fetched from the L2 cache or memory.

c) Another friend offers an alternative – using the original pipeline from part `a`, let's get rid of base + displacement addressing for loads and stores. Loads and stores can only use register addressing now. This will allow us to combine `EX` and `MEM` into one stage (called `EM`) and avoid the need to stall entirely. Instructions will either use the ALU or memory – but not both. There is still forwarding hardware, but now we only need to forward from the `EM/WB` latch to the `EM` stage ALU. The pipeline will now be:



Suppose that four fifths of loads actually use base + displacement addressing (i.e. they have a non-zero displacement), which means that these loads will need to have `add` instructions before them to do their effective address computation. Half of stores use base + displacement addressing, and these will also need to be replaced with an `add` plus the store instruction. This modification has no impact on the branch penalty or the instruction cache miss rate.

Is your friend right or wrong – will this eliminate all stalls? If they are wrong, give an example of when we would still need to stall.

The same issue as in part `b` still exists. Stalls would need to occur on cache misses.

d) A third friend has a different idea (it may be time for you to get new friends who don't talk about architecture all the time). Forget about trying to eliminate hazards – she says we should just use superpipelining and get a win on cycle time. Take the original architecture from part `a` – ignore the suggestions from `b` and `c` – and assume that the stages have the following latencies:

Stage	Latency (in picoseconds)
`IF`	200
`ID`	100
`EX`	200
`MEM`	200
`WB`	100

Your friend suggests a way to cut the `IF`, `EX`, and `MEM` stages in half – just increase the pipeline depth and make each of these stages into two stages. So your pipeline would now have `IF1`, `IF2`, `ID`, `EX1`, `EX2`, `MEM1`, `MEM2`, and `WB` stages – each of which would have 100 picosecond latency. Your friend also finds a way to do full forwarding between stages – even in the ALU – but loads are still a problem. In fact, load stalls will increase now because of this increase in pipeline depth. To help you figure out the new # of pipeline stalls from load data hazards, use the following table:

% of Loads	Distance of the Next Dependent Instruction
30%	1 cycle
20%	Exactly 2 cycles later
20%	Exactly 3 cycles later
10%	Exactly 4 cycles later
10%	Exactly 5 cycles later
5%	Exactly 6 cycles later
5%	Exactly 7 or more cycles later

So this means that 30% of loads are immediately followed by a dependent (i.e. 1 cycle later), 20% of loads have a dependent exactly 2 cycles later, 20% have a dependent 3 cycles later, and so on. These classifications are completely disjoint – the 20% of loads that have a dependent 2 cycles later do NOT have dependents 1 cycle later. Find the TCPI of this new architecture.

We note that extending the pipeline has no impact on the MCPI. In addition, the load-use hazard now affects loads with dependent instructions up to 2 cycles later. Also, assuming branches are resolved in `EX2`, we now have a branch misprediction penalty of 4 cycles. As a result:

$$\text{TCPI} = \text{BCPI} + 2.55$$

$$\text{BCPI} = \text{Base CPI} + \text{Load-Use Penalty} + \text{Branch Misprediction Penalty}$$

$$\text{BCPI} = 1 + (0.25)(0.5)(2) + (0.15)(0.66)(4) = 1.65$$

$$\text{TCPI} = 1.65 + 2.55 = \boxed{4.2}$$