CPSC 313, Winter 2016 Term 1 — Midterm **Solution**

Date: October 24, 2016; Instructor: Mike Feeley

1 **(8 marks)** **RISC Pipeline and Instruction Design.** For the first three questions, compare and contrast the two pipeline stages listed by identifying a similarity between them and then examining this similarity by clearly describing (a) each stage's unique role and (b) a way that one stage depends on the other. For example, in the student test-taking pipeline, if I listed the stages *Study* and *Take Test*, you might say: "The similarity is the ability to answer a question. *Study*'s role is to develop that ability and *Take Test*'s role is to measure the ability. *Take Test* depends on *Study* in that the better you study the higher the measured ability." Good answers will be two short sentences, one for (a) and one for (b).

1a Fetch and Decode

Fetch gets register numbers and decode gets their values. Decode depends on fetch to get the register numbers.

1b Execute and Memory

Execute does math and memory reads and writes to memory. Memory depends on execute to compute memory addresses.

1c Decode and Write Back

Decode sets dstE and dstM and Write Back writes values into these registers. Write Back depends on Decode to set these register numbers.

Now, answer the next question about the difference between CISC and RISC instruction sets.

1d Recall that Intel's x86 ISA (i.e., IA32) includes instructions like "`addl 100(%eax), %ebx`", which adds a value from memory to a register, and Y86 and other RISC ISAs don't. Carefully explain how the classic 5-stage RISC pipeline we studied restricts RISC ISAs in this way.

Either: (a) execute is required to compute address and so it can't add two operands or (b) execute comes before memory and so there the operand value is not yet known when the instruction is in execute and thus it can not be added.

2 **(7 marks)** **Pipeline Performance.**

2a Give a function $t(i, r, c)$ that computes the total running time of a program given:

i the total number of instructions executed by the program

r the clock rate

c CPI

$t(i, r, c) =$ $\boxed{i * c/r}$

2b Under what circumstances, if any, would adding a pipeline stage increase clock rate?

Splitting the slowest stage into two stages lowers the maximum stage delay and thus the clock can tick faster.

2c Under what circumstances, if any, would adding a pipeline stage increase CPI?

Adding a stage may increase the number of pipeline stalls.

3 **(8 marks)** **Parallelism, Dependencies and Hazards.**

Consider the following Y86 assembly code.

```
[1]  pop %eax              # pop stack top into %eax
[2]  irmovl $1, %ebx       # %ebx = 1
[3]  addl %eax, %ebx       # %ebx = %ebx + %eax
[4]  mrmovl (%ebx), %eax   # %eax = M[%ebx]
[5]  push %eax             # push %eax onto the stack
```

List all of the *causal* dependencies that exist in this code by listing (a) the two line numbers involved, (b) the affected operand (e.g., register name), and (c) the number of stalls that would be required to resolve the hazard (i.e., stalls *added* for this hazard) in *Pipe-Minus* (i.e., without data forwarding).

1. 1-3, eax, 0 stalls (because of 2-3 stall)
2. 1-5, esp, 0 stalls
3. 2-3, ebx, 3 stalls
4. 3-4, ebx, 3 stalls
5. 4-5, eax, 3 stalls

4 (9 marks) Data Forwarding.

4a For each of the hazards listed in the previous question, describe how it is resolved in *Pipe* (i.e., with data forwarding) by listing (a) what is forwarded from where and (b) how many stalls are now required to resolve the hazard.

1. 1-3, eax: forward valM from M; no stalls
2. 1-5, esp: no forwarding needed; no stalls
3. 2-3, ebx: forward valE from E; no stalls
4. 3-4, ebx: forward valE from E; no stalls
5. 4-5, eax: forward valM from M; one stall

4b The load-use hazard requires one stall to resolve. Could this stall be avoided by forwarding to E instead of D? Carefully critique this idea.

Yes it could, but doing so would increase the propagational delay of E because you would be forwarding from the end of M to the beginning of E, which might in turn require lowering the clock rate.

4c Consider the following specific example of the load-use hazard.

```
mrmovl (%ebx), %eax    # %eax = M[%ebx]
push %eax              # push %eax onto the stack
```

Could the pipeline's data-forwarding mechanism be modified to avoid stalling in this case? If so, carefully explain the changes you would have to make to do so (in plain English; pseudo-code not required).

Note that `mrmovl`'s `m.valM` in this case is not needed by the `push` instruction until it itself is in M. And so, you can allow `push` to proceed through E without stalling and with the wrong value of `valA` and then forward `W.valM` to `m.valA` when `mrmovl` is in W and `push` is in M. Doing this requires adding an additional forwarding path from W to M, checking the op-codes in W and M as well as comparing `W.dstM` to `M.srcA` both in the forwarding path and also in the pipeline control logic so that you don't stall/bubble in this case. The same is true if the second instruction is `rmmovl`.

5 (8 marks) Jump Prediction.

5a Which of the following two techniques for jump prediction is better: (a) predicting that jumps are not taken or (b) predicting that a jump will be taken if and only if it was taken the last time it executed? Carefully justify your answer.

Predicting that jumps are not taken is better, because it is more important to provide the compiler with predictable behaviour than it is to adjust behaviour dynamically. Consider, for example, the case of loops. The compiler will know when generating the code for a loop whether the loop's conditional jump will be taken or not taken. If it knows that jumps are predicted not taken, then it will construct the loop so that the conditional jump is used to test the exit condition. This jump will predict correctly every time through the loop but the last time. But, if the ISA does not disclose to the compiler what the normal prediction behaviour is, then the compiler will be unable to use its static knowledge of the program to reduce mis-predictions in this way. Dynamic prediction is good, but it should be secondary to static prediction.

5b Carefully describe the potential benefit and drawback of moving the computation of `cnd` (aka `bch`) from E to F.

Doing this would eliminate the need for jump prediction for condition jumps, because when the jump is in D, the fetch stage can set the PC like this: `pc = D.cnd? D.valC: d.valP`.

However, this creates a data hazard on the condition codes that would now be written in E and read in F; a situation much like the one we have for the register file which is written in W and read in D. It will thus be necessary to stall a jump instruction in F that follows an ALU instruction, until the ALU instruction is in W and the condition codes have been set. Consider, for example, this program.

```
    addl  %eax, %ebx
    jle   foo
```

The `jle` must stall in F three times until `addl` is in W because `addl` sets the condition codes in E and `jle` must read the new value of the condition codes in order to compute $f.cnd$.

6 (4 marks) Architecting for Parallelism.

6a Is instruction-level parallelism important for good pipeline performance? Why or why not?

Yes, because the lack of instruction-level parallelism can create hazards the might then require stalls to be resolved.

6b Is thread-level parallelism important for good pipeline performance? Why or why not?

No, because the granularity of parallelism here is an entire stream of instructions associated with a thread, but the pipeline is design to exploit parallelism among individual instructions at a much finer granularity.

7 (6 marks) Memory Hierarchy Characteristics and Tradeoffs.

7a Carefully explain why locality is important for good cache performance by indicating how caches exploit both temporal and spatial locality.

Caches work by placing data in fast memories automatically as a side effect of other accesses. When a program exhibits temporal locality, it accesses the same address multiple times: the first access misses in the cache but brings the address into the cache so that the subsequent accesses are hits. When a program exhibits spatial locality, it accesses different addresses that may reside in the same cache block (i.e., a contiguous chuck of several bytes from memory). The first access to a cache block is a miss but brings the entire block containing the referenced address into the cache so that subsequent accesses to other addresses in the block are hits.

7b Give one benefit of increasing cache block size (and keeping all other cache parameters constant)?

It will reduce compulsory misses for programs with spatial locality, because one miss will bring in more bytes and so the program will access new cache blocks less often. Consider, for example, reading a 256-byte array sequentially: if blocks are 64-B blocks there will be 4 misses but in a cache with 256-B blocks there will be only one.

7c Give one drawback of increasing cache size (and keeping all other cache parameters constant)?

Big caches are slower and so the latency of a cache hit will increase. This is, for example, the difference between the L1 and L2 caches. L1 is small and very fast. L2 is bigger, but slower.