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

Date: October 2016; Instructor: Mike Feeley

1 (5 marks) The classic RISC pipeline we are studying consists of 5 stages. *Briefly* explain the role of each stage in plain English (i.e., without referring to details such as register names).

FETCH: Load instruction from memory, determine its length and format, and separate it into individual pieces.
DECODE: Determine register numbers that will be read from and written to the register file. Read values from register file.
EXECUTE: Do math. Set condition codes. Determine whether conditional jumps are taken or whether conditional moves move.
MEMORY: Read or write data operands from/to memory.
WRITE BACK: Write values to register file.

2 (6 marks) Consider the following questions about pipeline organization that examine the relative order of each stage. **Include in your answer a Y86 instruction that could not be implemented if the specified ordering was reversed.**

2a Why is the memory stage after the execute stage?

So that execute can perform address calculations. If E was before M then, `mrmovl` and `rmmovl` could not use base plus displacement addressing and `pushl` would be able to write to `esp` minus 4.

2b Why is the execute stage after the decode stage?

So that we can perform math on register values. If E was before D, we non of the ALU instructions (e.g., `addl`) would work.

2c Why is the write-back stage after the memory stage?

So that we can write a value from memory into a register. If W as before M, `mrmovl` and `popl` would not work.

3 (5 marks) RISC instruction sets do not allow an ALU operation (e.g., addl) to read from memory.

3a Explain how the structure of the pipeline leads to this restriction.

The memory stage comes after the execute stage and so by the time we've read a value from memory it is too late to do math on it.

3b Describe how the pipeline could be modified to lift this restriction. You may not change the number of pipeline stages or their basic function. Your revised pipeline does not need to be able to implement every Y86 instruction (in fact it would not be able to) but it must be able to execute instructions like:

```
addl (%eax), %ebx    # r[ebx] = r[ebx] + m[r[eax]]
```

Swap the order of M and E so that M comes first.

3c This revised pipeline must place some restrictions on the revised ISA that it implements. One of the impacts is on `mrmovl` and `rmmovl`. Describe the problem and modify the instructions so that they will work with the new pipeline. Note that the revised versions need not be as powerful as the original ones, but they must still load and store between memory and a register.

We must eliminate base plus displacement addressing. The only addressing mode allowed would be like this `mrmovl (%eax), %ebx`.

4 **(8 marks)** Consider the following five-stage pipeline with stage delays (including overheads) of 34 ps, 42 ps, 75 ps, 50 ps and 18 ps. *(NOTE: To simplify the math, you can give answers like 1/100 instead of 0.01 or 1+2+3 instead of 6. Be sure to include units.)*

4a What is the maximum clock rate (i.e., fastest) acceptable for this pipeline?

(1 cycle) / (75 ps) = 1000/75 Ghz

4b What is the maximum throughput of this pipeline?

1/75 instructions per pico second.

4c What, if anything, might cause the actual throughput of programs to be lower than this maximum?

Pipeline bubbles caused by data or control hazards.

4d What is the minimum instruction latency of this pipeline?

75 * 5 ps

5 **(10 marks)** Consider the following piece of Y86 assembly code.

```
[0]  addl    %eax, %ebx
[1]  irmovl  $1, %eax
[2]  mrmovl  %(ebx), %ebx
[3]  addl    %ebx, %eax
```

5a List **all** of the data dependencies present in this code. Describe each dependency carefully. Indicate its type (causal, output or anti/alias) and the instructions and registers or memory locations involved.

1. anti between 0 and 1 on `%eax`
2. anti between 0 and 2 on `%ebx`
3. anti between 0 and 3 on `%eax`
4. output between 1 and 3 on `%eax`
5. output between 0 and 2 on `%ebx`
6. causal between 0 and 2 on `%ebx`
7. causal between 1 and 3 on `%eax`
8. causal between 2 and 3 on `%ebx`

5b List all of the data hazards present in this code for the Y86, five-stage pipeline.

The three causal dependencies.

5c For each hazard, indicate the total number of bubbles added by the Pipe-Minus implementation.

1. 2 for causal between 0 and 2 on `%ebx`
2. no stalls for causal between 1 and 3 on `%eax`, because 3 was stalled when 2 was stalled waiting for 0 (above)
3. 3 for causal between 2 and 3 on `%ebx`

5d For each hazard, indicate the total number of bubbles added by the Pipe implementation.

1 for causal between 2 and 3 on `%ebx`; this is a load-use hazard

6 **(10 marks)** Describe the implementation of the following new instruction for Y86 Seq as you did in Homework 2. This instruction is similar to `mrmovl` except that it adds 4 to `rB` and does not have a static-displacement operand. It would be useful, for example, for iterating over an array of integers.

Syntax:

```
mrmovincl (rB), rA
```

Semantics:

```
r[rA] <= m[r[rB]]
r[rB] <= r[rB] + 4
```

Memory Layout:

5	F	rA	rB

Describe each stage using a relaxed syntax similar as shown below. The Fetch and PC Update stages are complete. List only the code that would be added for this instruction.

Fetch:

```
f.iCd  = m[F.pc] >> 4
f.iFn  = m[F.pc] & 0xf
f.rA   = m[F.pc+1] >> 4
f.rB   = m[F.pc+1] & 0xf
f.valP = F.pc + 2
```

Decode:

```
d.srcA = R_NONE
d.srcB = D.rB
d.dstE = D.rB
d.dstM = D.rA
d.valB = r[d.srcB]
```

Execute:

```
e.aluA   = 4
e.aluB   = E.valB
e.aluFun = A_ADD
e.valE   = 4 + E.valB
```

Memory:

```
m.valM = m4[M.valB]
```

Write Back:

```
r[W.dstE] = W.valE
r[W.dstM] = W.valM
```

PC Update (pseudo stage):

```
w.pc = W.valP
```

7 **(6 marks)** Write Y86 assembly code that computes the sum of an array of integers where the address of the array is stored in `%ebx` and the length of the array is in `%ecx`. Place the sum in `%eax`.

```
     irmovl $0, %eax
     irmovl $1, %edi
     irmovl $4, %esi
     andl   %ecx, %ecx     # set CC for n
     jle    L1             # goto L1 if i<=0
L0:  mrmovl (%ebx), %edx   # edx = *a
     addl   %edx, %eax     # eax = eax + *a
     addl   %esi, %ebx     # a++
     subl   %edi, %ecx     # n--
     jg     L0             # goto L0 if n > 0
L1:
```

8 **(8 marks)** Answer these questions by saying what the specified feature is and why it is important. Your explanation must refer to specific architectural feature(s) of the processor or memory hierarchy that are designed to execute programs faster when the specified feature exists.

8a spatial locality

Spatial locality exists when memory accesses are clustered together to nearby memory addresses. Caches exploit spatial locality by storing data in multi-byte blocks.

8b temporal locality

Temporal locality exists when the same memory location is accessed repeatedly. Caches exploit temporal locality by retaining recently accessed blocks in the cache.

8c instruction-level parallelism

Instruction-level parallelism exists between a pair of instructions when there are no dependencies between them and thus their execution order does not matter. Pipelined (and super-scalar) processor architectures exploit instruction-level parallelism.

8d thread-level parallelism

Thread-level parallelism exists when programs explicitly indicate that threads of execution can be executed in parallel by either directly or indirectly creating multiple threads that can execute concurrently. Multi-core processors (and hyper-threading) exploit thread-level parallelism.

9 (4 marks) Polymorphic dispatch common to object-oriented languages like Java is implemented using a double-indirect call instruction that reads an address from memory and then jumps to it. Explain why it is challenging to implement such an instruction without stalling in a pipelined processor.

The address of the next instruction to execute after the call is not know to the pipeline until the call instruction exits the memory stage and so predicting which instructions should be F, D, and E at this point is impossible without retaining execution history.

10 (8 marks) Consider the following instruction-execution frequencies for a program running on the standard *Y86 Pipe* processor. The table shows, for example, that 7% of all instructions executed read the value of a register immediately after the preceding instruction modified that register by writing into it a value that came from memory.

7%	read register immediately after an instruction writes into that register a value it reads from memory
6%	read register immediately after an instruction writes into that register a value computed in Execute
12%	conditional jump that is taken
8%	conditional jump that is *not* taken
5%	`call`
5%	`ret`
57%	the remaining introduce no bubbles

Your answers can be in the form of a formula using the specific numbers involved. Show your work.

10a What is the CPI (average cycles per instruction) for this execution?

$CPI = 1 + 0.07 + 0.08 \times 2 + 0.05 \times 3 = 1.38$

This formula accounts for the bubbles associated with load-use hazards, mis-predicted jumps, and return instructions, respectively.

10b What is the throughput of this execution on a 3-Ghz processor (i.e., 3×10^9 cycles per second)?

$T = \frac{clock\_rate}{CPI} = \frac{3 \times 10^9}{1.38} \frac{instructions}{second}$

11 (14 marks) In this question you will consider adding a new stage to *Y86 Pipe* between D and E called M0 and renaming the existing M stage to M1. The resulting pipeline thus looks like this F-D-M0-E-M1-W. For the first five questions, M0 does all memory reads and M1 does all memory writes (thus allowing ALU instructions to operate on values read from memory; e.g., "`addl (%eax), %ebx`". Of course, this does require dropping the base-plus-displacement addressing mode for these instructions, but we will ignore this.

11a This design introduces a new hazard (for the new memory-ALU instructions it enables). Describe it.

There is now a causal hazard on memory if two instructions access the same memory location, the first one is a write and the second a read, and they are not separated by at least two other instructions.

11b When this hazard can be resolved by data forwarding, indicate the stage *to* which data is forwarded and state whether it is forward to the beginning or end of that stage.

End of M0.

11c When this hazard can be resolved by data forwarding, indicate the stage or stages *from* which data is forwarded and state whether it is forwarded from the beginning or end of those stage(s).

End of E and end of M1.

11d Does this hazard ever require stalling? Explain.

No. The value needed by the memory read is known by the end of E and it is not needed until the end of M0.

11e Does this new design improve the performance (i.e., reduce stalls) for any of the original *Y86* instructions? Explain.

Yes. It eliminates the stall associated with the load-use hazard because the load from memory will now be performed in M0 and can thus be forwarded to D without stalling.

11f Finally, consider a different design in which M0 and M1 can both read and write memory (and additions to the instruction set to fully utilize this capability). Does this new design introduce the possibility of new hazards? Carefully explain what they are.

Yes. It introduces the possibility of hazards on both *output* and *anti* dependencies for access to memory. Two instructions could be writing to the same memory location, one in M0 and one in M1, which creates an output-dependency hazard. One instruction could be writing a location in M0 that is being read by another instruction in M1, which creates an anti-dependency hazard.

12 (10 marks) Consider how to handle the hazards associated with the following new instruction added to *Y86 Pipe* (the one discussed in class, not the modified one you considered in the previous question).

```
cmrmovlXX D(rB), rA
```

Its `iCd` is `I_CMR` and it has the following semantics where XX is one of (le, l, e, ne, ge, g):

```
r[rA] <= m[D +r[rB]] if condition codes indicate that condition XX holds
```

When answering this question, use a simplified syntax for accessing pipeline-stage registers that leaves out `.get()` and `.set()` etc. (but be sure to correctly capitalize the pipeline id). For example you can say things like `if E.valC == W.valP {d.srcA = 0}`. Recall that `cnd` indicates whether the XX condition holds for an instruction (1 if it holds; 0 if it does not).

12a Give the complete data-forwarding logic for `d.srcA` for this instruction (and only this one) as it would appear in the DECODE stage (just `srcA`).

```
if (d.srcA != R_NONE) {
    if (d.srcA == M.dstM && M.cnd)
        d.valA = m.valM;
    else if (d.srcA == W.dstM && W.cnd)
        d.valA = W.valM;
    else
        d.valA = r[d.srcA];
}
```

12b Give the complete pipeline-control logic for this instruction (and only this one) that would stall the pipeline when and if necessary.

```
if (d.srcA != R_NONE && d.srcA == E.dstM && e.cnd) {
    f.stall = true;
    d.stall = true;
    e.bubble = true;
}
```

13 (15 marks) Consider the following Y86 code that contains three data hazards and two control hazards that cause bubbles in *Y86 Pipe*. Note that the program's inputs are `n` and the array `a`; and its output is `s`.

```
[01]        irmovl $1, %eax
[02]        irmovl $4, %ebx
[03]        irmovl $s, %ecx       # ecx = &s
[04]        irmovl $n, %edx       # edx = &n
[05]        irmovl $a, %edi       # temp_a = &a[0] = a
[06]        mrmovl (%edx), %edx   # temp_n = n
[07] LOOP:  subl   %eax, %edx     # temp_n = temp_n - 1
[08]        jl     DONE           # goto DONE if temp_n < 0
[09]        mrmovl (%edi), %esi   # esi = *temp_a
[10]        andl   %esi, %esi     # set CC for *temp_a
[11]        jl     L0             # goto L0 if *temp_a < 0
[12]        mrmovl (%ecx), %ebp   # temp_s = s
[13]        addl   %esi, %ebp     # temp_s = temp_s + *temp_a
[14]        rmmovl %ebp, (%ecx)   # s = temp_s
[15] L0:    addl   %ebx, %edi     # temp_a = temp_a + 4
[16]        jmp    LOOP
[17] DONE:  ...                   # more instructions

.pos 0x1000
s:  .long 0
n:  .long 4
a:  .long 1
    .long 2
    .long 3
    .long 4
```

13a Identify the three data hazards that cause bubbles by giving the line numbers involved.

1. Instructions 6 and 7 on `%edx`
2. Instructions 9 and 10 on `%esi`
3. Instructions 12 and 13 on `%ebp`

13b Rearrange the code to eliminate as many data-hazard bubbles as possible for *this particular input*; you may not add or change any instructions. You may describe your change with line numbers or by annotating the code; just be sure its clear what you mean.

1. Move instruction 6 up so that it is between 4 and 5.
2. Move instruction 12 up so that it is between 9 and 10.

Code replicated for convenience:

```
[01]        irmovl $1, %eax
[02]        irmovl $4, %ebx
[03]        irmovl $s, %ecx       # ecx = &s
[04]        irmovl $n, %edx       # edx = &n
[05]        irmovl $a, %edi       # temp_a = &a[0] = a
[06]        mrmovl (%edx), %edx   # temp_n = n
[07] LOOP:  subl   %eax, %edx     # temp_n = temp_n - 1
[08]        jl     DONE           # goto DONE if temp_n < 0
[09]        mrmovl (%edi), %esi   # esi = *temp_a
[10]        andl   %esi, %esi     # set CC for *temp_a
[11]        jl     L0             # goto L0 if *temp_a < 0
[12]        mrmovl (%ecx), %ebp   # temp_s = s
[13]        addl   %esi, %ebp     # temp_s = temp_s + *temp_a
[14]        rmmovl %ebp, (%ecx)   # s = temp_s
[15] L0:    addl   %ebx, %edi     # temp_a = temp_a + 4
[16]        jmp    LOOP
[17] DONE:  ...                   # more instructions
```

13c Identify the two control hazards that sometimes cause bubbles by giving their line numbers.

Instructions 8 and 11.

13d Rewrite the code to eliminate as many total bubbles (data- and control-hazards) as possible for *this input*. In this case you *are* permitted to change or add instructions. If you re-write the entire piece of code (not required), clearly indicate where you made changes and why you made them.

```
        irmovl $1, %eax
        irmovl $4, %ebx
        irmovl $s, %ecx       # ecx = &s
        irmovl $n, %edx       # edx = &n
        irmovl $a, %edi       # temp_a = &a[0] = a
        mrmovl (%edx), %edx   # temp_n = n
        jmp    L1             # goto L1
LOOP:   mrmovl (%edi), %esi   # esi = *temp_a
        mrmovl (%ecx), %ebp   # temp_s = s
        andl   %esi, %esi     # set CC for *temp_a
        jge    L2             # goto L2 if *temp_a >= 0
        jmp    L0             # goto L0 if *temp_a <= 0
L2:     addl   %esi, %ebp     # temp_s = temp_s + *temp_a
        rmmovl %ebp, (%ecx)   # s = temp_s
L0:     addl   %ebx, %edi     # temp_a = temp_a + 4
L1:     subl   %eax, %edx     # temp_n = temp_n - 1
        jge    LOOP           # goto LOOP if temp_n >= 0
DONE:   ...                   # more instructions
```

13e If you did this right, your solution will not work equally well (i.e., the same number of bubbles) if the input array contains negative numbers. Explain why and then carefully explain whether it is possible to modify the CPU-pipeline implementation so that this piece of code works nearly as well for arrays that contain all negative values — without changing the number of bubbles for arrays with all positive values — for sufficiently large arrays.

Since conditional jumps are predicted to be taken, the jump we are using (i.e, `jge`) is taken (and thus correctly predicted) only for positive array elements. If dynamic jump prediction where used, however, and all of the array elements were negative, after the pipeline saw a sufficient number of mis-predictions at this instruction, it would change its strategy to start predicting not-taken, which is the right choice when array elements are negative.

14 **(8 marks)** Answer these questions that consider changing an existing cache hierarchy to achieve a specific goal. In each case, describe a single change that you think would be most effective in achieving the stated goal. If there are multiple changes that you think would be equally effective, just describe one of them. Justify your answers.

14a Improve the performance of programs that have significant spatial locality.

Increase block size. Bigger blocks will yield fewer compulsory misses (and perhaps fewer capacity misses) if spatial locality extends beyond the boundary of the original blocks.

14b Improve the performance of programs that have a significant number of capacity misses.

Increase cache size. Larger caches store more data and thus may result if fewer misses to blocks that are replaced from the cache to make room for other data.

14c Improve the performance of programs that have a significant number of conflict misses.

Increase set-associativity. Larger sets allow the cache to store more lines with the same index simultaneously and thus avoid misses that are caused when a line in a set is replaced to make room for another line with the same set index.

14d Improve the performance of programs that have a significant number of cache hits.

Decrease cache size (or decrease set-associativity). Cache hit latency is higher for bigger caches and for caches with more associativity.