CPSC 313, Summer 2013 — Final Exam

Date: June 24, 2013; Instructor: Mike Feeley

Q1	/ 10
Q2	/ 10
Q3	/ 20
Q4	/ 12
Q5	/ 10
Q6	/ 10
Q7	/ 10
Q8	/ 3
Q9	/ 9
Q10	/ 6

This is a closed-book exam. No outside notes. Electronic calculators are permitted. You may remove the last two pages of the exam to make the notes contained there easier to use.

Answer in the space provided. Show your work; use the backs of pages if needed. There are **10** questions on **18** pages, totaling **100** marks. You have **2.5 hours** to complete the exam.

STUDENT NUMBER: ____________________

NAME: ____

SIGNATURE: ____

1 **(10 marks)** This question is concerned with the general principles of pipelining, dependencies and hazards.

First, consider two hardware implementations, **S** and **D**, of the same ISA. They have different pipelines: **S**'s pipeline has 5 stages and **D**'s has 7. Answer the following two questions. You must use the term *instruction-level parallelism* in one of your answers below.

1a Give one reason why **S** might execute some programs faster than **D**. Explain carefully.

1b Give one reason why **D** might execute some programs faster than **S**. Explain carefully.

Now, consider the dependencies that exist in the following code snippets. List **every** dependency. For each of them: (a) list the type of dependency and the register or memory location involved (if appropriate); (b) say whether the dependency is a hazard in the Y86 pipeline; and (c) if it is a hazard, explain how the hazard is resolved in the standard Y86-Pipe implementation (with data forwarding and predicting jumps are taken) and how many stalls are required; if it is not a hazard, explain why not.

1c
```
mrmovl  (%eax), %ebx
addl    %ebx, %ebx
```

1d
```
rmmovl  %eax, (%ebx)
mrmovl  (%ebx), %eax
```

1e
```
jle     foo
```

2 **(10 marks)** In the standard Y86-Pipe implementation, the load-use hazard requires one stall to resolve. But there are some forms of this hazard that should not actually require this stall. Consider, for example, this code snippet that performs a memory-to-memory copy, copying a value from the stack into memory at the address in %ebx.

```
popl    %eax
rmmovl  %eax, (%ebx)
```

2a In the standard pipeline, where there is a stall, the new value of register `%eax` is forwarded from the first instruction to the second. List the stage it is forwarded from and say whether it is forwarded from the beginning or end of that stage. List the stage it is forwarded to and say whether it is forward to the beginning or end of that stage.

2b Explain in plain English why the pipeline could be modified to eliminate the stall in this case, but not in the case of certain other load-use hazards.

2c Fixing the pipeline to eliminate this unnecessary stall requires changing the way that register `%eax` is fowarded. List the stage it is forwarded from in the revised pipeline and say whether it is forwarded from the beginning or end of that stage. List the stage it is forwarded to and say whether it is forwarded to the beginning or end of that stage.

2d Now, show the *changes* to the forwarding logic needed to implement this change. Use the simulator's Java syntax. As you did in the simulator, prefix pipeline-stage input registers with the capital letter of stage that reads them and output registers with the lower-case letter of the stage that writes them. Use `get()` and `getValueProduced()` to read pipeline registers and `set()` to write them.

2e What other instructions could have their load-use hazards resolved in exactly this way? List instructions that could be in the first location (the load) and then instructions that could be in the second location (the use).

load instructions:

use instructions:

3 **(20 marks)** In this question you will describe the implementation of a new Y86 instruction that conditionally jumps to an address that is stored in memory: `jxx D(rA)`. This instruction is similar to `jxx Dest` except that the jump-target address is read from memory (`M [D + r[rA]]`) instead of being a constant stored in the instruction. Use `I_JXX_DI` if you need to refer to the new instruction's op-code (i.e., iCd).

For example the following code jumps to address 0x1000.

```
irmovl  $0x1000, %eax   # eax = jump target address
rmmovl  %eax, 4(%ebx)   # store jump target address in memory at address 4+r[ebx]
xorl    %ecx, %ecx      # ALU operation that stores zero in %ecx
je      4(%ebx)         # goto m[4+r[ebx]] if last ALU resulted in 0, which it did
```

As a starting point, consider the following implementation of the standard `jxx Dest` instruction for Y86-PipeMinus that you will recall handles hazards by stalling and does not do data forwarding or jump prediction. *(This is repeated on a back page so that you an remove it.)*

FETCH_select_pc:

```
if (M.iCd.get() == I_JXX && M.cnd.get()==0)
    f.pc.set (M.valP.get());
else
    f.pc.set (F.prPc.get());
```

FETCH:

```
f.iCd.set  (mem.read (f.pc.getValueProduced()+0, 1) [0] .value() >>> 4);
f.iFn.set  (mem.read (f.pc.getValueProduced()+0, 1) [0] .value() & 0xf);
f.valC.set (mem.readIntegerUnaligned (f.pc.getValueProduced()+2));
f.valP.set (f.pc.getValueProduced()+5);
```

FETCH_predict_pc:

```
f.prPc.set (f.valC.getValueProduced());
```

DECODE:

```
d.valA.set (R_NONE);
d.valB.set (R_NONE);
d.dstE.set (R_NONE);
d.dstM.set (R_NONE);
```

EXECUTE:

```
e.cnd.set (read-condition-codes-and-opcode-to-decide-if-jump-is-taken);
```

PIPELINE HAZARD CONTROL:

```
if (D.iCd.get() == I_JXX || E.iCd.get() == I_JXX ) {
    F.stall  = true;
    D.bubble = true;
}
```

Answer these questions about the implementation of the new instruction, `jxx D(rA)`.

3a Provide the following parts of the Y86-PipeMinus implementation of the new instruction (just the new instruction; i.e., ignore all of the other instructions in the ISA just as was done in the example above). Use the same syntax constraints as in Question 2d.

FETCH_select_pc:

FETCH_predict_pc:

EXECUTE:

MEMORY:

PIPELINE HAZARD CONTROL:

3b Now consider the Y86-Pipe implementation of this instruction where your goal is to stall as little as possible by using static jump prediction (only static jump prediction and not using things like jump caches that implement dynamic jump prediction). Describe your strategy in plain English and explain how this strategy differs from the one used for the standard `jxx Dest`, if it does, and why. Be sure to say whether the mis-prediction penalty changes and if so to what and why.

3c Draw a diagram of the pipeline state as the new instruction moves through the pipeline from D to W in the case where the pipeline mis-predicts the jump and then corrects this mis-prediction. Your diagram should have 4 lines.

3d Provide the following parts of the Y86-Pipe implementation of the new instruction. Be careful to prevent a mis-predicted instruction from changing system state (i.e., registers, memory or condition codes).

FETCH_select_pc:

FETCH_predict_pc:

PIPELINE HAZARD CONTROL:

4 **(12 marks)** The following questions relate to the performance of programs running on the standard Y86 Pipe processor with data forwarding and predicting that branches are taken.

4a What two things must you measure to compute a program's throughput on a pipelined processor?

4b Consider the following program running on the standard Y86-Pipe processor (with data forwarding and predicting that conditional jumps are taken).

```
    irmovl  $1, %eax        # eax = 1
    irmovl  $4, %ebx        # ebx = 4
    irmovl  $1000, %ecx     # ecx = 1000
    irmovl  0x2000, %edi    # edi = 0x2000
    irmovl  $0, %esi        # esi = 0
L0: subl    %eax, %ecx      # ecx = ecx - 1
    je      L1              # goto L1 if ecx = 0
    addl    %ebx, %edi      # edi = edi + 4
    mrmovl  (%edi), %ebp    # ebp = m [edi]
    addl    %ebp, %esi      # esi = esi + ebp
    jmp     L0              # goto L0
L1: rmmovl  %esi, (%edi)    # m [edi] = esi
```

What is the CPI (average cycles per instruction) of the loop (i.e., between `L0` and the `jmp L0`, inclusive). Justify your answer and show your work by annotating the code to indicate where stalls occur.

4c Re-write the code starting with L0 to improve its pipeline performance.

4d What is the CPI of your revised code

5 **(10 marks)** Answer the following questions about the configuration of a 8-MB (i.e., 2^{23}-B), 8-way set associative cache with 64-B (i.e., 2^{6}-B) blocks, using 32-bit memory addresses. Recall that only data blocks are included when computing the size of the cache. Assume that address bits are numbered 0 to 31, with 0 the least-significant bit. For questions c-e, give an a bit range (e.g., "bits 3 to 5").

5a How many blocks does this cache have?

5b How many sets does this cache have?

5c Which address bits store the tag?

5d Which address bits store the index?

5e Which address bits store the block offset?

6 **(10 marks)** Fill in the following table describing a program's behaviour in a direct-mapped cache where: **bits 0-3 of the address store the block offset** and **bits 4-7 store the index**. The program performs the memory accesses listed in the table below, and only these, in the order listed. The array `a`, which starts at address `0x1000` is declared as "`int a[16][16];`".

Access	Hex Address	Tag	Index	Hit?	Explain Miss
`a[0][0]`					
`a[2][3]`					
`a[0][2]`					
`a[4][1]`					
`a[0][3]`					
`a[4][1]`					

7 **(10 marks)** In this question you consider the tradeoffs involved in changing a single feature of a cache design at a time. Note that only the listed feature changes (e.g., the overall size of the cache does not change). Carefully answer each question. **Justify your answers.**

7a What is the main benefit of increasing block size?

7b What is the main benefit of decreasing block size?

7c What is the main benefit of increasing associativity?

7d What is the main benefit of a write-back cache compared to a write-through cache with write buffer?

7e What is the main benefit of a write-through cache with write buffer compared to a write-back cache?

8 **(3 marks)** Disk and file system performance.

8a Give a formula for disk-read latency in terms of the following parameters

- N = number of bytes read
- S = maximum seek time (in units of seconds)
- R = rotation speed of disk (in units of rotations per second)
- D = track density (in units of bytes per track)

9 **(9 marks)** In a system that uses the Clock Algorithm for virtual-memory page replacement, explain the purpose of each of the following *page-table-entry (PTE)* bits by describing how they are cleared, set to 1 and read. Say whether the operation is performed by hardware or software, what causes it and why it is done. Each answer can be a single sentence of the form (where "/" indicates alternatives): "Hardware/Software when xxx to indicate/determine that/whether yyy.". For example, if the question asked what set's the PFN, a good answer would be: "Software when a page is faulted from disk into memory to record the physical location of the page for subsequent virtual-to-physical address translations."

9a valid

- set to 0:
- set to 1:
- read:

9b accessed

- set to 0:
- set to 1:
- read:

9c dirty

- set to 0:
- set to 1:

- read:

10 **(6 marks)** File and VM replacement.

10a Briefly explain what LRU replacement is and how it relates to the theoretical optimal replacement algorithm.

10b Give an example of a file-system workload (i.e., a program that reads data a file) where LRU replacement would be a poor choice.

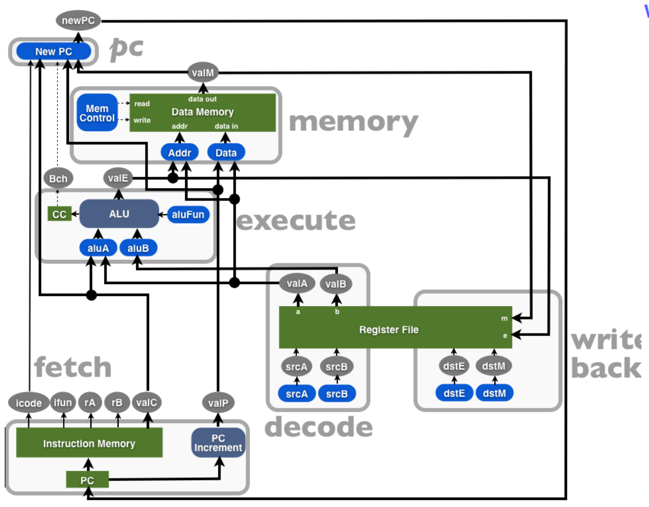
10c Explain how application-level buffering is different from caching and how it can be effective in managing memory used to store disk data in some cases where caching and LRU replacement fail.

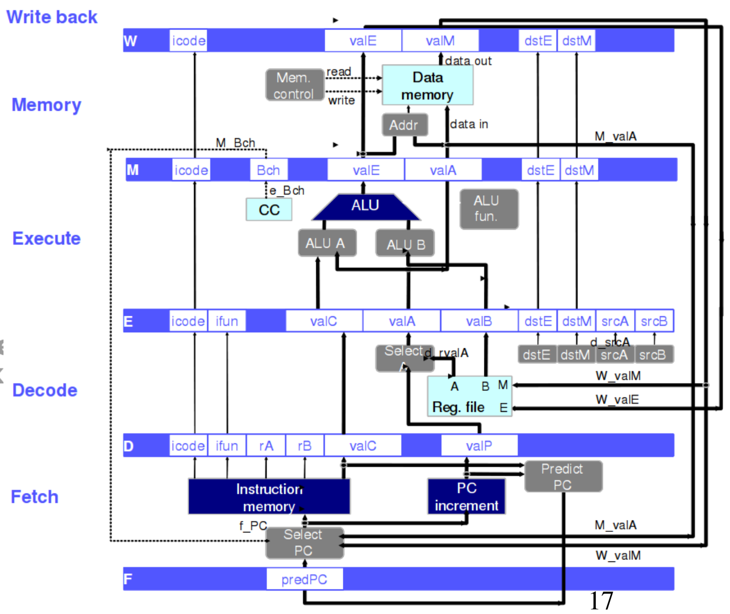
Instructions Encoding

Byte	0		1		2	3	4	5
halt	0	0						
nop	1	0						
rrmovl **rA**, **rB**	2	0	**rA**	**rB**				
cmovXX **rA**, **rB**	2	fn	**rA**	**rB**				
irmovl **V**, **rB**	3	0	F	**rB**	**V**			
rmmovl **rA**, **D**(**rB**)	4	0	**rA**	**rB**	**D**			
mrmovl **D**(**rB**), **rA**	5	0	**rA**	**rB**	**D**			
OPl **rA**, **rB**	6	fn	**rA**	**rB**				
jXX **Dest**	7	fn	**Dest**					
call **Dest**	8	0	**Dest**					
ret	9	0						
pushl **rA**	A	0	**rA**	F				
popl **rA**	B	0	**rA**	F				

Y86 ISA Reference

Instruction	Semantics	Example
rrmovl %rs, %rd	r[rd] $\leftarrow$ r[rs]	rrmovl %eax, %ebx
irmovl $i, %rd	r[rd] $\leftarrow$ i	irmovl $100, %eax
rmmovl %rs, D(%rd)	m[D + r[rd]] $\leftarrow$ r[rs]	rmmovl %eax, 100(%ebx)
mrmovl D(%rs), %rd	r[rd] $\leftarrow$ m[D + r[rs]]	mrmovl 100(%ebx), %eax
addl %rs, %rd	r[rd] $\leftarrow$ r[rd] + r[rs]	addl %eax, %ebx
subl %rs, %rd	r[rd] $\leftarrow$ r[rd] - r[rs]	subl %eax, %ebx
andl %rs, %rd	r[rd] $\leftarrow$ r[rd] & r[rs]	andl %eax, %ebx
xorl %rs, %rd	r[rd] $\leftarrow$ r[rd] $\oplus$ r[rs]	xorl %eax, %ebx
jmp D	goto D	jmp foo
jle D	goto D if last alu result ≤ 0	jle foo
jl D	goto D if last alu result < 0	jl foo
je D	goto D if last alu result $= 0$	je foo
jne D	goto D if last alu result $\neq 0$	jne foo
jge D	goto D if last alu result ≥ 0	jge foo
jg D	goto D if last alu result > 0	jg foo
call D	pushl %esp; jmp D	call foo
ret	popl %esp	ret
pushl %rs	m[r[esp] - 4] $\leftarrow$ r[rs]; r[esp] = r[esp] - 4	pushl %eax
popl %rd	r[rd] $\leftarrow$ m[r[esp]]; r[esp] = r[esp] + 4	popl %eax





FETCH:

```
f.iCd.set  (mem.read (f.pc.getValueProduced()+0, 1) [0] .value() >>> 4);
f.iFn.set  (mem.read (f.pc.getValueProduced()+0, 1) [0] .value() & 0xf);
f.valC.set (mem.readIntegerUnaligned (f.pc.getValueProduced()+2));
f.valP.set (f.pc.getValueProduced()+5);
```

FETCH_predict_pc:

```
f.prPc.set (f.valC.getValueProduced());
```

DECODE:

```
d.valA.set (R_NONE);
d.valB.set (R_NONE);
d.dstE.set (R_NONE);
d.dstM.set (R_NONE);
```

EXECUTE:

```
e.cnd.set (read-condition-codes-and-opcode-to-decide-if-jump-is-taken);
```

MEMORY:

WRITE BACK:

PIPELINE HAZARD CONTROL:

```
if (D.iCd.get() == I_JXX || E.iCd.get() == I_JXX ) {
    F.stall  = true;
    D.bubble = true;
}
```