Computer Architecture and Engineering

CS152 Quiz #4

April 10, 2008

Professor Krste Asanovic

Name:_____

This is a closed book, closed notes exam.

80 Minutes

10 Pages

Notes:

- **Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.**
- **Please carefully state any assumptions you make.**
- **Please write your name on every page in the quiz.**
- **You must not discuss a quiz's contents with students who have not yet taken the quiz. If you have inadvertently been exposed to the quiz prior to taking it, you must tell the instructor or TA.**
- **You will get no credit for selecting multiple choice answers without giving explanations if the instructions ask you to explain your choice.**

Writing name on each sheet	**_____**	**1 Point**
Question 1	**_____**	**22 Points**
Question 2	**_____**	**30 Points**
Question 3	**_____**	**27 Points**
TOTAL	**_____**	**80 Points**

Problem Q4.1: Out-of-Order Scheduling 22 points

This problem deals with an out-of-order single-issue processor that is based on the basic MIPS pipeline and has a floating-point unit. The FPU has one adder, one multiplier, and one load/store unit. The adder has a **two-**cycle latency and is fully pipelined. The multiplier has a **ten-**cycle latency and is fully pipelined. Assume that loads and stores take **one** cycle (plus **one** cycle for write-back for loads).

There are 4 floating-point registers, `F0-F3`. These are separate from the integer registers. There is a single write-back port to each register file and also to the ROB. In the case of a write-back conflict, the older instruction writes back first. Floating-point instructions (including loads writing floating-point registers) must spend **one** cycle in the write-back stage **before** their result can be used. Integer results are available for bypass **the next cycle after issue**.

To maximize number of instructions that can be in the pipeline, register renaming is used. The decode stage can add up to **one** instruction per cycle to the re-order buffer (ROB).

The instructions are committed in order and only **one** instruction may be committed per cycle. The earliest time an instruction can be committed is **one cycle after write back**.

For the following questions, we will evaluate the performance of the code segment in Figure Q4.1-A.

I_1	`L.D`	`F1, 5(R2)`
I_2	`MUL.D`	`F2, F1, F0`
I_3	`ADD.D`	`F3, F2, F0`
I_4	`ADDI`	`R2, R2, 8`
I_5	`L.D`	`F1, 5(R2)`
I_6	`MUL.D`	`F2, F1, F1`
I_7	`ADD.D`	`F2, F2, F3`

Figure Q4.1-A

Problem Q4.1.A **11 POINTS**

For this question, we will consider an ideal case where we have **unlimited** hardware resources for renaming registers. Assume that you have an **infinite ROB**.

Your job is to complete Table Q4.1-1. Fill in the cycle numbers for when each instruction enters the ROB, issues, writes back, and commits. Also fill in the new register names for each instruction, where applicable. Since we have an infinite supply of register names, you should use a new register name each time a register is written (T0, T1, T2, … etc). Keep in mind that after a register has been renamed, subsequent instructions that refer to that register need to refer instead to the new register name.

	Time				OP	Dest	Src1	Src2
	Decode → ROB	**Issued**	**WB**	**Committed**				
I_1	-1	0	1	2	L.D	T0	R2	-
I_2	0	2	12	13	MUL.D	T1	T0	F0
I_3	1				ADD.D			
I_4					ADDI			-
I_5					L.D			-
I_6					MUL.D			
I_7					ADD.D			

Table Q4.1-1

Problem Q4.1.B **11 POINTS**

For this question, assume that you have a **two-entry ROB** (i.e. only T0 and T1 tags can be used). A ROB entry can be reused **one cycle** after the instruction using it commits.

Your job is to complete Table Q4.1-2. Fill in the cycle numbers for when each instruction enters the ROB, issues, writes back, and commits. Also fill in the new register names for each instruction, where applicable.

	Time				OP	Dest	Src1	Src2
	Decode → ROB	**Issued**	**WB**	**Committed**				
I_1	-1	0	1	2	L.D	T0	R2	-
I_2	0	2	12	13	MUL.D	T1	T0	F0
I_3	3				ADD.D			
I_4					ADDI			-
I_5					L.D			-
I_6					MUL.D			
I_7					ADD.D			

Table Q4.1-2

Problem Q4.2: Fetch Pipelines 30 points

Ben is designing a deeply-pipelined single-issue in-order MIPS processor. The first half of his pipeline is as follows:

PC	PC Generation
F1	ICache Access
F2	
D1	Instruction Decode
D2	
RN	Instruction Dispatch
RF	Register File Read
EX	Integer Execute

There are no branch delay slots and currently there is **no** branch prediction hardware (instructions are fetched sequentially unless the PC is redirected by a later pipeline stage). Subroutine calls use **`JAL/JALR`** (jump and link). These instructions write the return address (PC+4) into the link register (r31). Subroutine returns use **`JR r31`**. Assume that PC Generation takes a whole cycle and that you cannot bypass anything into the end of the PC Generation phase.

Problem Q4.2.A 6 POINTS - Pipelining Subroutine Returns

Immediately after what pipeline stage does the processor know that it is executing a subroutine return instruction?

Immediately after what pipeline stage does the processor know the subroutine return address?

How many pipeline bubbles are required when executing a subroutine return?

Problem Q4.2.B **4 POINTS - Adding a BTB**

Louis Reasoner suggests adding a Branch Target Buffer (BTB) to speed up subroutine returns. A BTB entry stores the last target address of the instruction at a given PC. Why doesn't a standard BTB work well for predicting subroutine returns?

Problem Q4.2.C **5 POINTS - Adding a Return Stack**

Instead of a BTB, Ben decides to add a return stack to his processor pipeline. This return stack records the return addresses of the *N* most recent subroutine calls. This return stack takes no time to access (it is always presenting a return address). Return addresses are pushed onto the stack at the end of the decode stage if the instruction is a call (**`JAL/JALR`**).

Explain how this return stack can speed up subroutine returns. Describe for which instructions and in which pipeline stages return addresses are popped off the stack.

C

Problem Q4.2.D **5 POINTS - Return Stack Operation**

Fill in the pipeline diagram below corresponding to the execution of the following code on the return stack machine:

```
A: JAL B
A+1:
A+2:
…

B: JR r31
B+1:
B+2:
…
```

Make sure to indicate the instruction that is being executed. The first two instructions are illustrated below. The crossed out stages indicate that the instruction was killed during those cycles.

instruction	time→																	
A	PC	F1	F2	D1	D2	RN	RF	EX										
A+1		PC	F1	F2	D1	~~D2~~	~~RN~~	~~RF~~	~~EX~~									

Problem Q4.2.E **4 POINTS - Handling Return Address Mispredicts**

If the return address prediction is wrong, how is this detected? How does the processor recover, and how many cycles are lost (relative to a correct prediction)?

Problem Q4.2.F **6 POINTS - Further Improving Performance**

Describe a hardware structure that Ben could add, in addition to the return stack, to improve the performance of return instructions so that there is usually only a one-cycle pipeline bubble when executing subroutine returns (assume that the structure takes a full cycle to access).

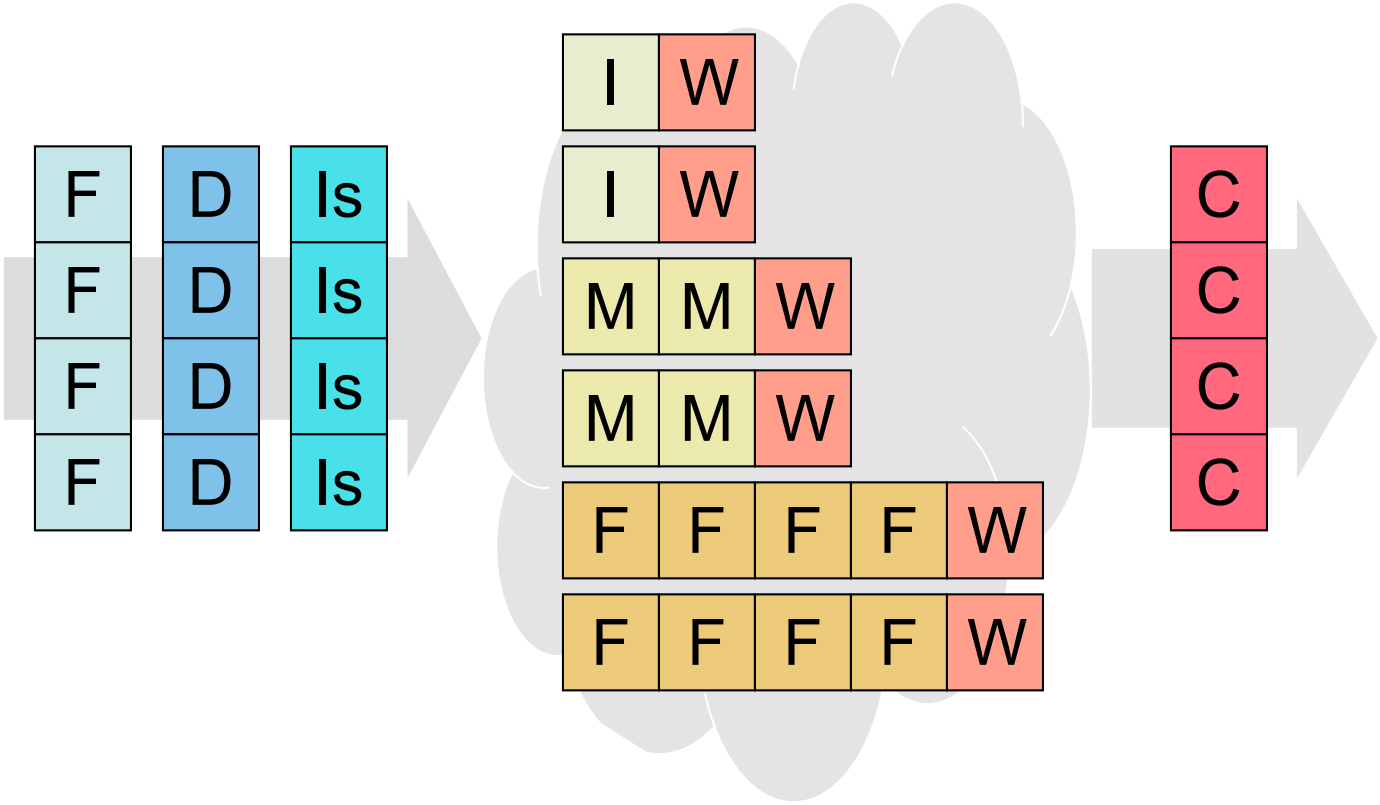
Problem Q4.4: Bounds on ILP 27 points

Consider a superscalar machine with a unified physical register file, i.e., the reorder buffer (ROB) stores only tags not data values. The ROB has infinite entries, and there are an infinite number of tags available for use in renaming.

The machine fetches, decodes, issues and commits instructions in-order, but allows instructions to proceed through the functional units out of order. Different functional units have different latencies as described below. The processor may fetch, decode, issue, or commit up to four instructions on any given cycle.

The pipeline of the machine is illustrated below. The machine has fetch (F), decode (D) and issue (IS) units. The functional units are: a single-cycle latency integer ALU (I), a fully-pipelined two-cycle memory access unit (M) to a perfect memory, and a fully-pipelined four-cycle FPU (F). During the writeback stage (W) the busy bits of instructions in the ROB are updated, allowing instructions whose sources are ready to begin execution on the next cycle. Instructions must commit (C) in order.

During the issue stage an instruction is allocated a new physical register for its result, and its source registers are translated into physical registers using the rename table. Physical registers are freed when a later write to the same architectural register commits. The freed physical register can be used by the decode stage in the next cycle following the commit.



Problem Q4.4.A **7 POINTS - Maximum ILP**

What is the minimum number of registers required for the machine's pipelines to be capable of becoming completely saturated on *some* code sequence? 'Saturated' means that it would be impossible to sustain a higher throughput. Explain.

Problem Q4.4.B **4 POINTS - Maximum ILP with no FPU**

Does the answer to Q4.4A change if we remove the floating-point unit from the machine? Explain why the minimum does or does not change.

Problem Q4.4.C **8 POINTS - Minimum Registers**

Does there exist a **minimum** number of physical registers that would prevent the registers from ever becoming a performance bottleneck, for *any* piece of code. If so, give the number of physical registers and explain. If not, explain why the number required is not bounded.

Problem Q4.4.D **8 POINTS - ROB size**

Would there be any purpose to having more reorder buffer entries than the difference between the number of physical registers and the number of architectural registers? Explain.