Computer Architecture and Engineering

CS152 Quiz #4 Solutions

Problem Q4.1: Out-of-Order Scheduling

Problem Q4.1.A

	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	**13**	**15**	**16**	ADD.D	**T2**	**T1**	**F0**
I_4	**2**	**3**	**5**	**17**	ADDI	**T3**	**R2**	**-**
I_5	**3**	**4**	**6**	**18**	L.D	**T4**	**T3**	**-**
I_6	**4**	**7**	**17**	**19**	MUL.D	**T5**	**T4**	**T4**
I_7	**5**	**18**	**20**	**21**	ADD.D	**T6**	**T5**	**T2**

Table Q4.1-1

Common mistakes: Forgetting in-order commit, integer result bypassing, single-ported ROB/register files, issue dependent on writeback of sources and completed decode

Problem Q4.1.B

	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	**13**	**15**	**16**	ADD.D	**T0**	**T1**	**F0**
I_4	**14**	**15**	**17**	**18**	ADDI	**T1**	**R2**	**-**
I_5	**17**	**18**	**19**	**20**	L.D	**T0**	**T1**	**-**
I_6	**19**	**20**	**30**	**31**	MUL.D	**T1**	**T0**	**T0**
I_7	**21**	**31**	**33**	**34**	ADD.D	**T0**	**T1**	**F3**

Table Q4.1-2

Common mistakes: Forgetting in-order commit, not reusing T0 and T1 appropriately

Problem Q4.2: Fetch Pipelines

<table>
<tr><td>PC</td><td>PC Generation</td></tr>
<tr><td>F1</td><td rowspan="2">ICache Access</td></tr>
<tr><td>F2</td></tr>
<tr><td>D1</td><td rowspan="2">Instruction Decode</td></tr>
<tr><td>D2</td></tr>
<tr><td>RN</td><td>Rename/Reorder</td></tr>
<tr><td>RF</td><td>Register File Read</td></tr>
<tr><td>EX</td><td>Integer Execute</td></tr>
</table>

Problem Q4.2.A — Pipelining Subroutine Returns

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

D2

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

RF

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

6

Problem Q4.2.B — Adding a BTB

A subroutine can be called from many different locations and thus a single subroutine return can return to different locations. A BTB holds only the address of the last caller.

Problem Q4.2.C — Adding a Return Stack

Normally, instruction fetch needs to wait until the return instruction finishes the RF stage before the return address is known. With the return stack, as soon as the return instruction is decoded in D2, instruction fetch can begin fetching from the return address. This saves 2 cycles.

A return address is pushed after a JAL/JALR instruction is decoded in D2. A return address is popped after a JR r31 instruction is decoded in D2.

Problem Q4.2.D — Return Stack Operation

```
A: JAL B
A+1:
A+2:
...

B: JR r31
B+1:
B+2:
...
```

instruction	time→																	
A	PC	F1	F2	D1	D2	RN	RF	EX										
A+1		PC	F1	F2	D1	~~D2~~	~~RN~~	~~RF~~	~~EX~~									
A+2			PC	F1	F2	~~D1~~	~~D2~~	~~RN~~	~~RF~~	~~EX~~								
A+3				PC	F1	~~F2~~	~~D1~~	~~D2~~	~~RN~~	~~RF~~	~~EX~~							
A+4					PC	~~F1~~	~~F2~~	~~D1~~	~~D2~~	~~RN~~	~~RF~~	~~EX~~						
B						PC	F1	F2	D1	D2	RN	RF	EX					
B+1							PC	F1	F2	D1	~~D2~~	~~RN~~	~~RF~~	~~EX~~				
B+2								PC	F1	F2	~~D1~~	~~D2~~	~~RN~~	~~RF~~	~~EX~~			
B+3									PC	F1	~~F2~~	~~D1~~	~~D2~~	~~RN~~	~~RF~~	~~EX~~		
B+4										PC	~~F1~~	~~F2~~	~~D1~~	~~D2~~	~~RN~~	~~RF~~	~~EX~~	
A+1											PC	F1	F2	D1	D2	RN	RF	EX

Problem Q4.2.E — Handling Return Address Mispredicts

When a value is popped off the return stack after D2, it is saved for two cycles as part of the pipeline state. After the RF stage of the return instruction, the actual r31 is compared against the predicted return address. If the addresses match, then we are done. Otherwise we mux in the correct program counter at the PC stage and kill the instructions in F1 and F2. Depending on how fast the address comparison is assumed to be, you might also kill the instruction in D1. So there is an additional 2 or 3 cycles on a return mispredict.

Problem Q4.2.F — Further Improving Performance

Ben should add a cache of the most recently encountered return instruction addresses. During F1, the contents of the cache are looked up to see if any entries match the current program counter. If so, then by the end of F1 (instead of D2) we know that we have a return instruction. We can then use the return stack to supply the return address.

Problem Q4.3: Bounds on ILP

Problem Q4.3.A **Maximum ILP**

The solution to this question comes from Little's Law. Every instruction in flight needs a separate destination physical register, unless it does not produce a result. We also need enough physical registers to map the committed architectural registers.

We want to find the minimum number of physical registers for some piece of code to saturate the machine pipelines. The machine cannot fetch and commit more than four instructions per cycle, so we pick the four lowest latency functional units (i.e., the two integer and the two memory) out of the six available, as these will require the least number of instructions in flight to saturate the machine's pipelines.

Instructions allocate a physical register in the decode stage, then need to hold it for at least the issue stage, plus the execute stage(s), plus the writeback stage, plus the commit stage. This is at least 3 cycles in addition to the number of execute stages.

An acceptable answer was:

```
minimum physical = 2 * (3 + 1)       // Two integer instructions
                   2 * (3 + 2)       // Two memory instructions
                   + # architectural registers

                 = 18 + # architectural registers
```

Answers based on this Little's argument received 6/7 points, minus 1 point if the # architectural registers was not included.

We obtain a better solution if we realize that we only need to find one piece of code that saturates the pipelines. Stores and branches do not need to allocate a destination register. Consider the following unrolled loop which zeros a region of memory, written in MIPS assembly code:

```
loop:   sw r0, 0(r1)
        sw r0, 4(r1)
        bne r1, r2, loop
        addu r1, r1, 8      # in delay slot
```

This loop will reach a steady state of one iteration per cycle, and will saturate the machine (four instructions per cycle) while allocating only one physical register per cycle (also freeing one physical register per cycle). The number of physical registers neededfor this loop is only:

```
minimum physical = 4  // Addu instruction.
                   + # architectural registers
```

This possibility got full credit (7/7).

Another possible reading of the question was to find a piece of code that had a different performance bottleneck, e.g., a data dependency, and find the number of physical registers required for it. (This wouldn't actually saturate the machines pipelines). These answers received almost full credit depending on quality of answer.

Common errors:

Many students incorrectly thought that each instruction needs to allocate three physical registers, one for each source operand and one for the destination.

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

The answer does not change, because the FPU would not be used in a solution that minimized the number of physical registers needed.

However, if in part A the assumption had been that code with a performance bottleneck was used, and that included the FPU, then a valid answer would be that the number would change. This was given full credit.

Common errors:

Most errors were due to not understanding the setup in previous part, or not explaining that floating-point units were the longest latency units and hence avoided in part A.

Problem Q4.4.C **Minimum Registers**

There is no minimum as we can always create a scenario where a larger number of physical registers would allow the instruction fetch/decode stages to run ahead and find an instruction that could execute in parallel with preceding instructions. For example, an arbitrarily long serial dependency chain can be followed by an independent instruction:

```
add.d f1, f1, f2
add.d f1, f1, f2
...             # Many similar dependent instructions elided.
add.d f1, f1, f2
sub   r1, r2, r3  # An independent instruction.
```

Given that the fetch stage can proceed faster than the serially dependent instructions can be executed, we can always find a case where more physical registers will lead to faster execution given an infinite ROB.

Problem Q4.4.D **ROB Size**

Yes. Stores and branches do not require new physical registers, and together average around 10-25% of a typical instruction mix. So making the ROB have up to ~30% more entries than the difference between physical and architectural registers will tend to improve performance by allowing larger active instruction windows. Some examples could benefit from even larger ROBs, e.g., code from part A.

Common errors:

Many students assumed that every ROB entry needs a separate physical register (got 1 point for giving this incorrect answer).