CS 520: Computer Architecture and Organization
Fall 2015: Prof. Timothy N. Miller
Homework 4
Deadline: Monday, October 26

Problem 1: SHORT ANSWERS

These are all bonus questions. Do them last!

For the following, consider an out-of-order CPU that implements register renaming, along with a reorder buffer to maintain precise architectural state. Moreover, the reorder buffer is unified with the physical register file, and there is a separate architectural file.

Problem 1.1 (2 points)

What factors limit the number of instruction that can be dispatched but not yet retired?

Problem 1.2.1 (2 points)

When an instruction is retired, it is necessary to move its result from the physical register file to the architectural register file. Why?

Problem 1.2.2 (2 points)

When the result is moved, what other status information must be updated in order for future instructions to correctly find their sources?

Problem 1.3 (2 points)

To support register renaming, we implement a Renamed[] array that keeps track of important information used for allocating and deallocating physical registers. For physical register k, and the Boolean value stored in Renamed[k], what does the value indicate when false? What does mean when true?

Problem 1.4 (2 points)

Compare *forwarding* with *completion*. What event leads to forwarding? How does forwarding lead to completion? Which is the most direct precondition for retirement?

Problem 1.5 (2 points)

Consider a multithreaded (SMT) professor, with a single shared ALU, that supports four threads. Any pipeline stage in the ALU may be executing a portion of any instruction from any of the four threads. When an instruction is finished executing and it is time to write the result to the register file, how might this processor distinguish which thread the result belongs to?

Problem 1.6 (2 points)

Consider the following instruction and notice the self-dependency:

```
ADD R3, R4, R3
```

What kind of dependency is this? This dependency can be ignored for static scheduling and software interlocking – Why? How does an out-of-order CPU with renaming handle this dependency?

Problem 2 (50 points)

For this problem, use this code fragment:

```
I1:   MOVI R1, #4000      /* R1 <-- 4000 */
I2:   MOV R2, R1          /* R2 <-- R1 */
I3:   LOAD R4, R2, #9
I4:   SUB R2, R2, R4
I5:   MUL R4, R2, R1
I6:   STORE R5, R2, #11
I7:   LOAD R5, R2, #33
I8:   AND R1, R5, R4
```

Assume that the code fragment is executed on the out-of-order APEX processor with register renaming. Assume for this problem that the APEX ISA defines 6 architectural registers, R0 through R5. The physical registers are numbered P0 trough P10 and as control flows into the code fragment, the contents of the rename table show an identity mapping, that is R0 is mapped to P0, R1 is mapped to P1, R2 is mapped to P2, R3 is mapped to P3, R4 is mapped to P4 and R5 is mapped to P5. The free list, at time 0 contains: P6, P7, P8, P9, P10.

Also assume:

- Fetch and Decode each require one cycle
- An instruction always spends at least one cycle in the IQ. If one has been waiting in the IQ, and its dependencies are satisfied, it can be issued on the same cycle that it receives its input.
- MOV, AND, and SUB have one cycle of latency and are executed in the "ALU"
- LOAD takes 3 cycles in the "LS" unit
- MUL takes 4 cycles in the "M" unit
- You will not run out of IQ entries, and IQs are distributed.

Problem 2.1

Write the renamed code fragment. Don't forget to free physical registers when you can. Add them to the head of the free list (which is therefore a stack structure).

Problem 2.2

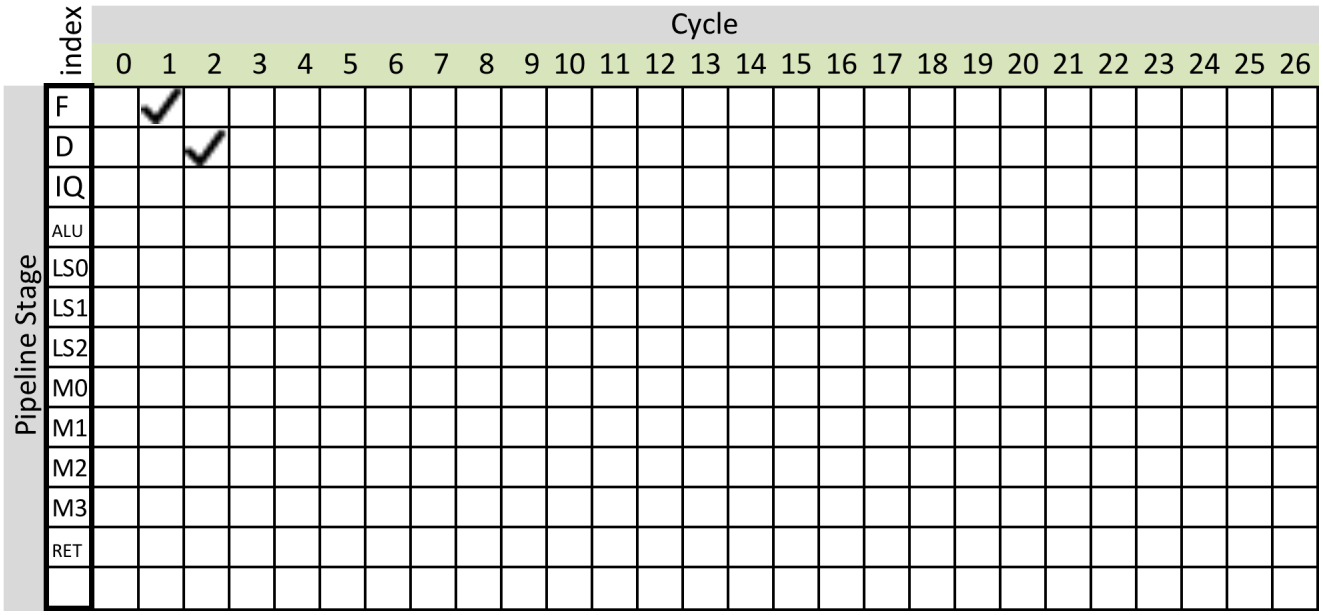
a) Draw a Gantt chart like the one on the next page showing the flow of instructions through the front end pipeline, IQ, and functional unit pipelines. Indicate dependencies being satisfied with arrows.

b) In the same chart, indicate when instructions retire (mark them in the RET stage). An instruction can retire when it is completed, but it can only retire in program order.

c) At the same time, fill out a chart like the one on the following page, indicating changes in RAT, allocated, renamed, and valid flags. Highlight (bold or circle) which entries change each cycle. Column 0 corresponds to the time just prior to the dispatch of I1.

Problem 2.3

List the IQ entry for each instruction and indicate when (which cycles) it is in the queue.


index
Cycle
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
Pipeline Stage
F
D
IQ
ALU
LS0
LS1
LS2
M0
M1
M2
M3
RET

	index	Cycle																											
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
RAT	0	0	✓	✓																								
	1	1																										
	2	2																										
	3	3																										
	4	4																										
	5	5																										
Allocated	0	1																										
	1	1																										
	2	1																										
	3	1																										
	4	1																										
	5	1																										
	6	0																										
	7	0																										
	8	0																										
	9	0																										
	10	0																										
Renamed	0																											
	1																											
	2																											
	3																											
	4																											
	5																											
	6																											
	7																											
	8																											
	9																											
	10																											
Valid	0	1																										
	1	1																										
	2	1																										
	3	1																										
	4	1																										
	5	1																										
	6	0																										
	7	0																										
	8	0																										
	9	0																										
	10	0																										

Problem 3 (50 points)

These problems pertain to two-operand ISAs. See:
http://en.wikipedia.org/wiki/Instruction_set#Number_of_operands
http://randomascii.wordpress.com/2012/12/29/the-surprising-subtleties-of-zeroing-a-register/

For the purposes of this problem, we will use the following syntax for 3-operand ISA instructions:

```
ADD dst, src1, src2
```

And for 2-operand ISA instructions:

```
ADD dst, src2        // dst is also src1;                        dst := dst + src2
MOV dst, src         // copy register contents from src to dst;  dst := src
```

Problem 3.1

Given the instruction sequence below, generate a *minimal* equivalent for 2-operand ISAs. The minimal sequence does not contain any unnecessary MOV instructions that might be used to save registers that would otherwise be overwritten.

```
SUB R1, R2, R3
MUL R4, R1, R2
LDI R1, #100          // R1 := 100; indicates no further dependencies on R1
ADD R5, R5, R4        // NOTE:  When renaming, there are two different R5's here!
AND R6, R5, R4
XOR R7, R6, R5
LDI R6, #100          // R6 := 100; no further dependences on R6
```

Problem 3.2

There are 32 architectural registers, R0 through R31, and they initially rename to P0 through P31 respectively. The free physical register list contains {P32, P33, P34, …, P63}. Show the renamed instructions, corresponding to the *original sequence* in Problem 3.1.

Problem 3.3

Given the same initial conditions as Problem 3.2, show the renamed instructions corresponding to *your answer* to Problem 3.1.

Problem 3.4

Your answer to Problem 3.1 requires MOV instructions. One way to implement a MOV instruction would be to add zero using the integer ALU (i.e. the equivalent of "ADD R1, R2, #0" to implement "MOV R1, R2"). However, this would tie up RoB and VFU resources, add excessive latency, and allocate an extra physical register to hold the same value as another physical register.

Briefly describe an alternative rename policy that reduces and/or eliminates the PRF, RoB, VFU, and latency overheads of the MOV instruction.

Only describe changes to the rename logic (not the rest of the processor), and briefly explain why it reduces those overheads.

Problem 3.5

Based on reading this article:
http://randomascii.wordpress.com/2012/12/29/the-surprising-subtleties-of-zeroing-a-register/
explain how a "ZERO" instruction (which clears a register to zero) might be implemented with minimal overhead in rename logic.