EECS 470 *Midterm Exam -ANSWERS*

Winter 2012

Name: _____KEY__________________________________ unique name: ________________

Sign the honor code:

I have neither given nor received aid on this exam nor observed anyone else doing so.

____

Scores:

Page #	Points
2	**/20**
3	**/11**
4	**/13**
5	**/15**
6	**/10**
7	**/16**
8 & 9	**/15**
Total	***/100***

NOTES:

- Open book and Open notes
- Calculators are allowed, but no PDAs, Portables, Cell phones, etc.
- Don't spend too much time on any one problem.
- You have about 120 minutes for the exam.
- There are ***9*** pages including this one.
- Be sure to show work and explain what you've done when asked to do so.
- **There are two "answer areas" for the last problem. Clearly mark which one you want graded or we will grade the first one.**

Multiple choice/fill in the blank **[20 points, -2 per blank or wrong answer]**

a. Say a processor with a 32-bit address space (byte addressed) has an L1 cache with a 16KB data store which is 4-way associative. If the cache uses 32-byte blocks, there are ___128_____ sets in the cache and the tag store is ***512 / [✓ 1280] / 2968 / 4096 / 5552*** bytes (including only the tags themselves).

b. Over time, devices and wires have gotten smaller. This has generally resulted in faster devices (transistors) and slower wires. The wire delay is proportional to the "RC" delay. This increased delay in smaller wires is because if the wires get narrower the ***resistance / [✓ capacitance] / inductance*** of a *wire of fixed length* goes ***up / [✓ down]*** only slightly while the ***[✓ resistance] / capacitance / inductance*** of that wire goes ***[✓ up] / down*** significantly.

c. Consider an add instruction. In the original Tomasulo's algorithm, the ARF is updated only when the add ***[✓ finishes execution and the RAT points to the add] / finishes execution / commits / the RAT points to the instruction, and the instruction is at the head of the ROB***.

In the "P6" algorithm, the ARF is updated only when the add ***finishes execution and the RAT points to the add / finishes execution / [✓ commits] / the RAT points to the add, and the add is at the head of the ROB***.

d. If, in the "R10K" algorithm, the RAT had only one write port; you could only have one instruction that writes to a register ***[✓ dispatch] / start execution / complete execution / retire*** per cycle.

e. In the "R10K" algorithm, we will free ***all PRF entries / no PRF entries / [✓ those PRF entries which aren't pointed to by the RRAT] / those PRF entries in the RAT that are overwritten by the RRAT*** when a branch mispredict occurs.

f. For the original implementation of Tomasulo's algorithm, the primary source of high-latency instructions were ***store instructions / [✓ floating point instructions] / integer square root instructions***.

g. In the "P6" algorithm, the RAT points to a ***[✓ reorder buffer entry] / a reservation station / a physical register / an execution unit***.

h. "Energy constrained computing" refers to devices where the primary concern is related to ***the device getting too hot / [✓ the device using battery power too quickly] / the device needing a higher voltage than we can easily supply***.

i. A CMOV instruction is commonly used to eliminate a ***[✓ branch] / load / store / floating point*** instruction.

Short answer [24 points]

Consider the pipeline you were to implement for your third programming assignment, but assume that the structural hazard has been removed. A given program consists of 20% loads, 10% stores, 10% branches and 60% ALU operations. If 40% of the branches are not-taken and 40% of all instructions are dependent on the instruction in front of them, what is the expected CPI of the processor on this program? Show you work. **[5 points]**

Every instruction takes 1 cycle, plus additional penalties for Loads (1 cycle MEM->EX forwarding) and mispredicted branches (3 cycles squashing IF,ID,EX)

CPI = 1 + (1 cycle) * (20% loads) * (40% dependent) + (3 cycles) * (10% branches) * (60% mispredict)
= 1 + .08 + .18
= 1.26

1. Consider the case of self-modifying code[1] in the context a processor which implements a standard 5-stage pipeline. **[6 points]**
   a. In a single sentence, describe the hazard that self-modifying code creates. **[3]**

      Instructions currently in flight (already fetched) could be overwritten, causing the pipeline to execute incorrect instructions. (One point for each / any of: A "Read-After-Write" hazard in instruction memory / PC / IF).

   b. Briefly explain how you would ***detect*** the issue. Specifically, what would you be looking for in which stage(s)? **[3]**

      Each stage should snoop on the Proc2DMem signal (from the MEM stage), whenever it is valid, the address should be compared against the stage's PC.

[1] Self modifying code is when a program modifies a memory location and then executes that memory location.

2. **Say that** in the “R10K” algorithm, you have 64 RoB entries, 8 RS entries, 96 physical registers and the ISA supports 32 architected registers. How many bits would you need for the RRAT (just the pointers in the table, not valid bits, the free list or anything else)? Show your work. **[4 points]**

   32 architected registers => 32 table entries
   96 physical registers => ceil(log2(96)) = 7 bits to uniquely address them.
   32 * 7 = 224 bits

3. Write a Verilog module which implements a 4-bit modulo[2] up-counter with enable and synchronous reset. It takes three inputs: clock (clk), enable (en) and reset (reset). Its only output is the four bit count (Q[3:0]). You are to follow the Verilog coding guidelines for the course. Highly inefficient code will receive fewer points. **[9 points]**

```
module counter(
    input            clock,
    input            reset,
    input            en,
    output reg [3:0] Q,
);

//synopsys sync_set_reset "reset"
always @(posedge clock) begin
    if (reset)
        Q <= `SD 0;
    else if (en)
        Q <= `SD Q + 4'b1;
end

endmodule
```

[2] “modulo” means that the counter wraps around. So this counter, while enabled, should count (0, 1, 2, … 14, 15, 0, 1, etc.). It should reset to 0.

Longer answer [41 points]

1. Consider the following pseudo-assembly code:

```
           r3=1000
           r5=0
           r6=0
     bob:  r1=MEM[r3+0]                 // THE LOAD
           if(r1>0) goto TWO            // Branch 1 – Notice the target!
           r5=r5+1
           r6=r6+r1
           r6=r6+r5
           r6=r6/2
     ONE:  if(r1>=0) goto TWO           // Branch 2
           r6=r6+1
     TWO:  r3=r3+4
           if(r5<50000) goto bob        // Branch 3
```

“bob” has an address of 0x100. The predictors all use the least significant bits of the PC other than the word-offset. Predictors and patterns are all initialized to all zeros (not taken).

You are to consider how different branch predictors will behave on this code under different circumstances.

- **Case 1:** The data loaded from memory is 1 the first time, 0 the second, 1 the third, 0 the forth and follows that pattern forever (1, 0, 1, 0, 1, 0, etc.)
- **Case 2:** The data loaded from memory is 0 the first time, -1 the second, -1 the third, -1 the forth and follows that pattern forever (0, -1 -1, -1, 0, -1, -1, -1, etc.)
- **Case 3:** The data loaded is (1, 0, -1, 1, 0, -1, 1, 0, -1, etc.)

You are now to consider 2 branch predictors:

- **Predictor 1:** A PC-based predictor with 8 entries each a 1 bit predictor.
- **Predictor 2:** A local pattern history predictor. The BHT has 16 entries, each with 2 bits of history. The predictors are each 1 bit.

What are the expected prediction *rates* for each of the following (percentage of time right)? Your answers must be correct within 1.0%. **[15 points, -1 per wrong or blank box, min 0]**

	Case 1		**Case 2**		**Case 3**	
	Predictor 1	*Predictor 2*	*Predictor 1*	*Predictor 2*	*Predictor 1*	*Predictor 2*
Branch 1	50	100	0	75	33	66
Branch 2	100	100	50	75	0	50
Branch 3	50	100	0	100	33	100

2. Given the following design changes to a simple out-of-order pipeline (and assuming no other changes to the pipeline or workload), what would be the effect on the i) number of instructions committed (N_{inst}), ii) the cycles-per-instruction (CPI), iii) clock period (t_{clk}), and iv) silicon area cost (A_{cost}). For each possible effect, indicate one of the following: no change (Ø), equal or greater (↑), equal or less (↓), or not enough information to determine (?). Provide the best answer. For a few of the boxes, we are looking for a short (one sentence) written explanation.
[10 points, -.5 per wrong or blank answer, -1 per wrong or blank explanation]

Design Change	N_{inst}	**CPI**	t_{clk}	A_{cost}
Increase the number of ROB entries	Ø	↓	↑	↑
Increase the number of architected registers	↓	*1 ↓	↑	↑
Change from the original Tomasulo's algorithm to the "P6" scheme.	Ø	*2 ? (↓or ↑)	↑Ø	↑
Implement early branch resolution	Ø	↓	↑Ø	↑
Improve T_{CPU} by using CMOVs to remove certain branches.	*3 ↑	↓	↑Ø	Ø↑

In the case where more than one answer is given, any of those answers were accepted. For the *2 question, we took any of those 3 answers (though ? is the best answer) but only gave credit on *2 if you identified both cases.

Provide a short (one sentence) explanation of your answer for the "*ed" boxes.

*1: There are fewer spills and fills.

*2: Better CPI due to branches stalling instructions after them, worse due to adding RoB as structural hazard.

*3: replacing a branch with a CMOV often involves adding other instructions.

3. Consider a set of code where there are two classes of instructions.
   - *"Short"* instructions are not dependent on any other instruction and can execute in 3 cycles.
   - *"Long"* instructions are not dependent on any other instruction and can execute in 20 cycles.
   - *"Dependent"* instructions are (only) dependent on the instruction in front of them and take 3 cycles to execute.

   Say you have a machine which can issue one instruction per cycle, finish execution of one instruction per cycle, and retire one instruction per cycle. This machine implements what we have called the "P6" algorithm and it keeps an instruction in its RS until that instruction has finished executing. The machine has an RS size of 16 and a RoB size of 64. You may assume the machine otherwise has unlimited resources (execution units etc.) Show your work! **[16 points]**

   a. What is the best CPI this machine could achieve if the program being run consisted of only "*long*" instructions? **[4]**
   In steady state, 16 instructions will enter RS, first one will stall for 4 cycles, then all 16 will leave without stalling. Next set of 16 will do the same thing. So 20/16=1.25

   b. What is the best CPI this machine could achieve if the program being run consisted of only "*dependent*" instructions? **[4]**
   Each has to wait to start until the one in front of it is done. 3.

   c. What is the best CPI this machine could achieve if the program being run consisted of groups of 200 instructions, where the first 199 were "dependent" and the last was "short". Assume there are a large number of these groups. (So the code is 199 dependent, 1 short, 199 dependent, 1 short, etc.) **[4]**
   You'd get groups of 200 instructions, each of which takes 600 cycles to execute. However, each group could overlap while both were in the system. RS will be limiting factor, so overlap will be for 15*3 or 45 cycles. So (600-45)/200=2.775

   d. What is the best CPI this machine could achieve if the program being run consisted of groups of 3 instructions, where the first 2 were "dependent" and the last was "short". Assume there are a large number of these groups. (So the code is 2 dependent, 1 short, 2 dependent, 1 short, etc.) **[4]**
   1. Can easily work on three different dependency chains at the same time.

Consider the following state of a machine implementing what we've called the "R10K" algorithm. **[15 points]**

RAT	
Arch Reg #	Phy. Reg #
0	0
1	6
2	7
3	12
4	8

ROB					
Buffer Number	PC	Executed?	Dest. PRN	Dest ARN	
0	20	N	5	1	← HEAD
1	24	Y	12	3	
2	28	N	6	1	
3	32	Y	--	--	
4	36	Y	7	2	
5	40	N	8	4	←TAIL
6					
7					
8					

RRAT	
Arch Reg #	Phy. Reg #
0	0
1	1
2	2
3	3
4	4

RS							
RS#	Op Type	Op1 Ready?	Op1 PRN/value	Op2 Ready?	Op2 PRN/value	Dest PRN	ROB
0	+	N	5	Y	13	6	2
1	*	Y	6	Y	7	5	0
2	+	N	6	Y	4	8	5
3							
4							

PRF			
Phy Reg #	Value	Free	Valid
0	4	N	Y
1	5	N	Y
2	6	N	Y
3	7	N	Y
4	8	N	Y
5	9	N	N
6	1	N	N
7	21	N	Y
8	3	N	N
9	4	Y	N
10	5	Y	N
11	6	Y	N
12	13	N	Y

KEY:

- **Op1 PRN/value** is the value of the first argument if "Op1 ready?" is yes; otherwise it is the Physical Register Number that is being waited upon.
- **Op2 PRN/value** is the same as above but for the second argument.
- **Dest. PRN** is the destination Physical Register Number.
- **Dest. ARN** is the destination Architectural Register Number.
- **ROB** is the associated ROB entry for this instruction.
- **Free/Valid** indicates if the PRF entry is currently available for allocation and if the valid in it is valid. *A free entry should be marked as invalid*.

Say that the instruction in ROB #3 is a branch and it was mis-predicted: The next PC should have been 100. Say that the instruction in memory location 100 is R2=R0+R4 and that the instruction in 104 is R0=R2+R3. Update the machine to the state where the branch has left the RoB, and the instructions at memory 100 has been dispatched and executed, but not committed while the one at location 40 has been dispatched but not executed. When faced with an arbitrary decision, just be sure to make a legal choice. ***Be sure to update the head and tail pointers!***

On the following page is an extra copy of this problem. You may use this one or the one on the next page but be sure to cross out (with a BIG X) the one you don't want graded.

<u>This is a copy of the previous state. You may use this one or the previous one but be sure to cross out (with a BIG X) the one you don't want graded.</u>

RAT	
Arch Reg #	**Phy. Reg #**
0	0->11
1	6
2	7->10
3	12
4	8->4

ROB					
Buffer Number	**PC**	**Executed?**	**Dest. PRN**	**Dest ARN**	
~~0~~	~~20~~	~~N~~	~~5~~	~~1~~	← ~~HEAD~~
~~1~~	~~24~~	~~Y~~	~~12~~	~~3~~	
~~2~~	~~28~~	~~N~~	~~6~~	~~1~~	
~~3~~	~~32~~	~~Y~~	~~--~~	~~--~~	
~~4~~	~~36~~	~~Y~~	~~7~~	~~2~~	
~~5~~	~~40~~	~~N~~	~~8~~	~~4~~	←~~TAIL~~
6	100	Y	10	2	← HEAD
7	104	N	11	0	←TAIL
8					

RRAT	
Arch Reg #	**Phy. Reg #**
0	0
1	1->6
2	2
3	3->12
4	4

RS							
RS#	**Op Type**	**Op1 Ready?**	**Op1 PRN/value**	**Op2 Ready?**	**Op2 PRN/value**	**Dest PRN**	**ROB**
0	~~+~~	~~N~~	~~5~~	~~Y~~	~~13~~	~~6~~	~~2~~
1	~~*~~	~~Y~~	~~6~~	~~Y~~	~~7~~	~~5~~	~~0~~
2	~~+~~	~~N~~	~~6~~	~~Y~~	~~4~~	~~8~~	~~5~~
3	~~+~~	~~Y~~	~~4~~	~~Y~~	~~8~~	~~10~~	~~6~~
4	+	Y	12	Y	13	11	7

KEY:

- **Op1 PRN/value** is the value of the first argument if "Op1 ready?" is yes; otherwise it is the Physical Register Number that is being waited upon.
- **Op2 PRN/value** is the same as above but for the second argument.
- **Dest. PRN** is the destination Physical Register Number.
- **Dest. ARN** is the destination Architectural Register Number.
- **ROB** is the associated ROB entry for this instruction.
- **Free/Valid** indicates if the PRF entry is currently available for allocation and if the valid in it is valid. <u>*A free entry should be marked as invalid*</u>.

PRF			
Phy Reg #	**Value**	**Free**	**Valid**
0	4	N	Y
1	5	N->Y	Y->N
2	6	N	Y
3	7	N->Y	Y->N
4	8	N	Y
5	42	N->Y	N
6	55	N	N->Y
7	21	N->Y	Y->N
8	3	N->Y	N->N
9	4	Y	N
10	5	Y->N	N->Y
11	6	Y->N	N
12	13	N	Y

Say that the instruction in ROB #3 is a branch and it was mis-predicted: The next PC should have been 100. Say that the instruction in memory location 100 is R2=R0+R4 and that the instruction in 104 is R0=R2+R3. Update the machine to the state where the branch has left the RoB, and the instructions at memory 100 has been <u>dispatched</u> and <u>executed</u>, but not committed while the one at location 40 has been <u>dispatched</u> but <u>not executed</u>. When faced with an arbitrary decision, just be sure to make a legal choice. ***Be sure to update the head and tail pointers!***

Note: Free/crossed out values don't matter.