Name _____

Computer Architecture and Engineering
CS152 Quiz #3
March 19th, 2013
Professor Krste Asanović

Name: _____<ANSWER KEY>_____

This is a closed book, closed notes exam.
80 Minutes
11 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 other students who have not taken the quiz. If you have inadvertently been exposed to a 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	_____	41 Points
Question 2	_____	18 Points
Question 3	_____	20 Points
TOTAL	_____	80 Points

Name _____

Question 1: Out-of-order Processors [41 points]

For this question, we will use the following DAXPY code to understand how out-of-order processors behave. Written in C, the code is as follows:

```
void daxpy(double a, double* x, double* y, int n) {
  for (int i=0; i<n; i++)
    y[i] = y[i] + a * x[i];
}
```

Assuming that **a** is stored in **f4**, pointer **x** is stored in **x5**, pointer **y** is stored in **x6**, and **n** is stored in **x7**, the code compiles to the following inner loop:

```
loop:
  fld  f1, 0(x5)
  fld  f2, 0(x6)
  fmul f3, f1, f4
  fadd f1, f2, f3
  fsd  f1, 0(x6)
  addi x5, x5, 8
  addi x6, x6, 8
  addi x7, x7, -1
  bne  x7, x0, loop
```

Name _____

Q1.A RAW, WAR, WAW Hazards [6 points]

Draw arrows to mark all *register* RAW, WAR, WAW hazards *between instructions*. Count how many hazards you have marked and write the total count at the end of each column. We have expanded the loop to spot data hazards across loop iterations.

RAW Hazards

```
fld  f1, 0(x5) [1]
fld  f2, 0(x6) [1]
fmul f3, f1, f4 [1]
fadd f1, f2, f3 [1]
fsd  f1, 0(x6)
addi x5, x5, 8 [2]
addi x6, x6, 8 [2]
addi x7, x7, -1 [2]
bne  x7, x0, loop
fld  f1, 0(x5) [1]
fld  f2, 0(x6) [1]
fmul f3, f1, f4 [1]
fadd f1, f2, f3 [1]
fsd  f1, 0(x6)
addi x5, x5, 8
addi x6, x6, 8
addi x7, x7, -1 [1]
bne  x7, x0, loop
```

# of RAW Hazards: __15__

WAR Hazards

```
fld  f1, 0(x5) [1]
fld  f2, 0(x6) [1]
fmul f3, f1, f4 [1]
fadd f1, f2, f3 [2]
fsd  f1, 0(x6) [2]
addi x5, x5, 8 [1]
addi x6, x6, 8 [1]
addi x7, x7, -1[1]
bne  x7, x0, loop[1]
fld  f1, 0(x5) [1]
fld  f2, 0(x6) [1]
fmul f3, f1, f4 [1]
fadd f1, f2, f3
fsd  f1, 0(x6) [1]
addi x5, x5, 8
addi x6, x6, 8
addi x7, x7, -1
bne  x7, x0, loop
```

# of WAR Hazards: __15__

WAW Hazards

```
fld  f1, 0(x5) [1]
fld  f2, 0(x6) [1]
fmul f3, f1, f4 [1]
fadd f1, f2, f3 [1]
fsd  f1, 0(x6)
addi x5, x5, 8 [1]
addi x6, x6, 8 [1]
addi x7, x7, -1 [1]
bne  x7, x0, loop
fld  f1, 0(x5) [1]
fld  f2, 0(x6)
fmul f3, f1, f4
fadd f1, f2, f3
fsd  f1, 0(x6)
addi x5, x5, 8
addi x6, x6, 8
addi x7, x7, -1
bne  x7, x0, loop
```

# of WAW Hazards: ___8___

Q1.B Register Renaming and Data Hazards [3 points]
Which of the data hazards from above would register renaming remove? Explain.

RAW WAR WAW

WAR, WAW hazards are not real dependences. With the destination register renamed for instructions that writes to a register, all subsequent instructions that read from that destination register will read the right value (gets rid of the WAR hazard). Similarly, all subsequent instructions that write to the same destination register doesn't risk clobbering the data as it writes to a different physical register (gets rid of the WAW hazard).

Name _____

Q1.C Impact of Register Renaming on Performance [5 points]
Suppose we run the DAXPY code on two out-of-order processors, one with register renaming and one without. Both processors have unlimited reordering resources, perfect memory disambiguation, perfect branch prediction, and infinite superscalar issue width. On which out-of-order processor would the DAXPY code perform better? Explain your reasoning.

With register renaming, we are able to overcome WAR and WAW hazards and get the next iteration in flight.

-1 if you didn't mention getting the next iteration in flight
-3 if you didn't talk about getting rid of WAR and WAW hazards
-5 if you didn't pick the machine with register renaming

Name _____

Q1.D Register renaming [6 points]

Complete the following table, assuming that both **x** registers and **f** registers are renamed from the same pool of unified physical registers. The initial map table and free list is given below. Assume that the free list is organized as a FIFO – a physical register is popped off the top of list when allocated, and a physical register is added back to the bottom of the list when reclaimed. For each instruction, label the following: -1 for incorrect box

- which physical register gets assigned to the instruction as a destination
- upon commit, which physical register gets added back to the free list

Architectural Register	Physical Register
f1	P11
f2	P22
f3	P3
f4	P7
x5	P9
x6	P13
x7	P14

Free List
P15
P16
P17
P18
P19
P20
P21
P1
P2
P4
P5
P6
P8
P10

Instruction	ISA Destination Register	Physical Destination Register	Freed Register
`fld f1, 0(x5)`	f1	P15	P11
`fld f2, 0(x6)`	f2	P16	P22
`fmul f3, f1, f4`	f3	P17	P3
`fadd f1, f2, f3`	f1	P18	P15
`fsd f1, 0(x6)`	n/a	n/a	n/a
`addi x5, x5, 8`	x5	P19	P9
`addi x6, x6, 8`	x6	P20	P13
`addi x7, x7, -1`	x7	P21	P14
`bne x7, x0, loop`	n/a	n/a	n/a
`fld f1, 0(x5)`	f1	P1	P18
`fld f2, 0(x6)`	f2	P2	P16
`fmul f3, f1, f4`	f3	P4	P17
`fadd f1, f2, f3`	f1	P5	P1
`fsd f1, 0(x6)`	n/a	n/a	n/a
`addi x5, x5, 8`	x5	P6	P19
`addi x6, x6, 8`	x6	P8	P20
`addi x7, x7, -1`	x7	P10	P21
`bne x7, x0, loop`	n/a	n/a	n/a

Name _____

Q1.E Precise Exceptions [5 points]
Describe how precise exceptions are implemented in out-of-order processors with unified physical register files. Include descriptions of how the map table and free lists are maintained.

Commit needs to happen in-order. Once an exception is detected, the ROB is sequentially unrolled backwards starting from the tail pointer. During this process, the map table is rolled back to an older mapping, and the allocated physical register is put back to the free list.

-1 if you didn't mention ROB
-1 if you weren't specific about how the map table and free lists are maintained
-2 if you didn't tell how to maintain the map table
-2 if you didn't specify how to update the free list
-1 if you purposed snapshotting every cycle, but didn't correctly specify how to maintain free list

Q1.F Exceptions in Action [6 points]
Suppose the second **`fsd`** instruction (the instruction that is in **bold** in the previous page) triggered an exception. Show the state of the map table and free list before jumping into the exception handler. Assume that the machine is flushed when the commit pointer of the ROB moves to the faulting instruction.
-1 for incorrect box

Architectural Register	Physical Register
f1	P5
f2	P2
f3	P4
f4	P7
x5	P19
x6	P20
x7	P21

Free List
P11
P22
P3
P15
P9
P13
P14
P18
P16
P17
P1
P10
P8
P6

Name _____

Q1.G Lifetime of Physical Registers [5 points]
When can we free a physical register? Explain.

When next write of same architecture register commits.

-1 if said when last reader commits, since due to exceptions that can happen after the last reader commits.
-1 if didn't mention "commit".
No points if you said you can free a physical register once an instruction has committed.

Q1.H Impact of Unrolling on Performance [5 points]
Could unrolling the loop in software, as shown below, help performance? Could it hurt performance? Assume that **n** is always an even number, and hence the loop doesn't need patch-up code for the cases where **n** is an odd number. Explain your reasoning.

```
loop:
  fld  f1, 0(x5)
  fld  f2, 0(x6)
  fld  f5, 8(x5)
  fld  f6, 8(x6)
  fmul f3, f1, f4
  fadd f1, f2, f3
  fmul f7, f5, f4
  fadd f5, f7, f6
  fsd  f1, 0(x6)
  fsd  f5, 8(x6)
  addi x5, x5, 16
  addi x6, x6, 16
  addi x7, x7, -2
  bne  x7, x0, loop
```

Loop unrolling does help performance, as unrolling reduces loop overhead (bookkeeping). This lets the processor to put more load/store instructions in the ROB, and hence helps performance. Many people mentioned that unrolling will let the processor execute the next iteration earlier, but if you think about it, without unrolling, the out-of-order processor is still able to kick out the next iteration earlier. Some people also said that by unrolling, there are fewer branches in the code, but note that this branch is an easy one to predict. The benefit of unrolling is more about reducing the instruction count rather than reducing the number of branches to predict.

Name _____

Question 2: Limits on Throughput [18 points]

Suppose we are running the following assembly codes on an out-of-order processor with unlimited renaming registers and reordering resources, perfect memory disambiguation, perfect prediction, and infinite superscalar issue width. Floating-point add instructions have a 5-cycle latency, floating-point multiply takes 8 cycles, and integer instructions take one cycle. Assume that all loads/stores hit in the cache, and take 2 cycles to execute. Also assume that load values are forwarded from the speculative store buffer; in this case, the load instruction can execute one cycle after the store instruction. For each of the following code sequences, in the steady state, how many *floating-point operations* will execute *per cycle*?

Q2.A [6 points]

C Code	Variable Mapping	Assembly Code

`for (int i=0; i<n; i++)`
`  y[i] = y[i] + a * x[i];`
	`f3: a`
	`x1: x pointer`
	`x2: y pointer`
	`x3: i`
		`loop:`
		`  fld f1, 0(x1)`
		`  fld f2, 0(x2)`
		`  fmul f1, f1, f3`
		`  fadd f1, f1, f2`
		`  fsd f1, 0(x2)`
		`  addi x1, x1, 8`
		`  addi x2, x2, 8`
		`  addi x3, x3, -1`
		`  bne x3, x0, loop`

Critical path: addi (previous iteration) → addi (1 cycle latency)
# of floating-point ops: 2

2 floating-point operations per cycle

-2 if 5 FLOP/cycle (i.e., counted floating point load/store instructions)

Name _____

Q2.B [6 points]

C Code	Variable Mapping	Assembly Code

`for (int i=0; i<n; i++)`
`  sum = sum + x[i];`
	`f2: sum`
	`x1: x pointer`
	`x2: i`
		`loop:`
		`  fld f1, 0(x1)`
		`  fadd f2, f2, f1`
		`  addi x1, x1, 8`
		`  addi x2, x2, -1`
		`  bne x2, x0, loop`

Critical path: fadd (previous iteration) → fadd (5 cycle latency)
# of floating-point ops: 1

1/5 floating-point operations per cycle

-2 if 2/5 FLOP/cycle (i.e., counted floating point load/store instructions)

Q2.C [6 points]

C Code	Variable Mapping	Assembly Code

`for (int i=0; i<n; i++)`
`  a[i+1] = a[i] * b[i];`
	`x1: a pointer`
	`x2: b pointer`
	`x3: i`
		`loop:`
		`  fld f1, 0(x1)`
		`  fld f2, 0(x2)`
		`  fmul f1, f1, f2`
		`  fsd f1, 8(x1)`
		`  addi x1, x1, 8`
		`  addi x2, x2, 8`
		`  addi x3, x3, -1`
		`  bne x3, x0, loop`

Critical path: fld → fmul → fsd (previous iteration) → fld (8+2+1 cycle latency)
# of floating-point ops: 1

1/11 floating-point operations per cycle

-2 if 1/12 FLOP/cycle (i.e., counted 2 cycles for fld)
-2 if 4/11 FLOP/cycle (i.e., counted floating point load/store instructions)
-4 if 4/12 FLOP/cycle (i.e., made both mistakes)

Name _____

Question 3: Potpourri [20 points]

Q3.A [5 points] Why is the issue window usually sized smaller than the ROB?

Issue window smaller than ROB.

Q3.B [5 points] Why do architects put more attention on the accuracy of branch predictors in a superscalar out-of-order processor than in a 5-stage in-order processor?

- branch mispredicts are major source of performance loss in OoO machines, much more than 5-stage in-order processors.
- Lots of speculative work can be wasted
- Incorrect speculation can take resources away from correct path code

-1 if not listed

Name _____

Q3.C [5 points] Suppose we speculatively issue an instruction that depends on a load value assuming the load will hit in the cache. How can we detect misspeculation at commit time by re-executing the load? Explain.

- assumed instructions behind load keep executing with wrong "miss" value
- check load value against value reloaded at commit and squash pipe if different
- checking for miss at commit time doesn't work, as intervening store commit can cause eviction between correct speculation at load commit.

Q3.D [5 points] Suppose we bypass load values from the speculative store buffer. If the load address hits in both the store buffer and the cache, which one should we use: load data from forwarded from the store buffer or the load data from the cache?

- we use youngest store that is older than the load
- this would come from either cache or store buffer depending the order of load & stores in store buffer

-2 if didn't include cache as potential source but otherwise correct
-4 if didn't consider relative age of loads & stores