Computer Architecture and Engineering

CS152 Quiz #4

April 7, 2009

Professor Krste Asanovic

Name:___Answer Key___

This is a closed book, closed notes exam.

80 Minutes

9 Pages

Notes:

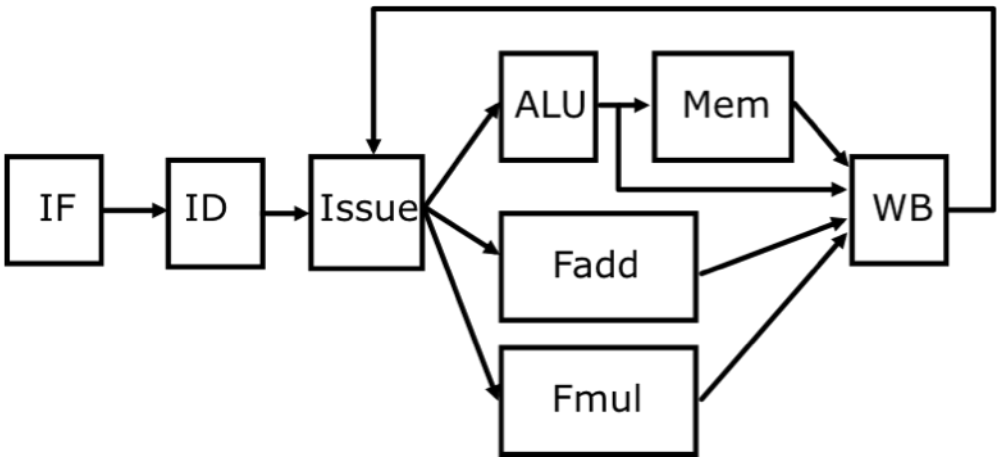
- **Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.**
- **Please carefully state any assumptions you make.**
- **Please write your name on every page in the quiz.**
- **You must not discuss a quiz's contents with students who have not yet taken 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	_____	21 Points
Question 2	_____	20 Points
Question 3	_____	18 Points
Question 4	_____	20 Points
TOTAL	_____	80 Points

NAME:_____

Problem Q4.1: Scheduling 21 points

In this problem we will examine the execution of a code segment on a single-issue out-of-order processor. It is like the one shown in lecture and reproduced below:



You can assume that:

- All functional units are pipelined
- ALU operations take 1 cycle
- Memory operations take 3 cycles (includes time in ALU)
- Floating-point add instructions take 3 cycles
- Floating-point multiply instructions take 5 cycles
- There is no register renaming (not until Q4.1.C)
- Instructions are fetched and decoded in order
- The issue stage is a buffer of unlimited length that holds instructions waiting to start execution
- An instruction will only enter the issue stage if it does not cause a WAR or WAW hazard
- Only one instruction can be issued at a time, and in the case multiple instructions are ready, the oldest one will go first

To simplify book-keeping for this problem we will only track instructions during a few critical stages:

- **I -** When an instruction enters the issue stage
- **E -** When an instruction starts execution (enters FU)
- **C -** When an instruction completes execution (enters WB, results available this cycle)

For this problem we will be using the following code:

```
I1   L.D    F2, 0(R1)
I2   ADD.D  F1, F1, F2
I3   MUL.D  F4, F3, F3
I4   ADDI   R1, R1, 8
I5   L.D    F2, 0(R1)
I6   ADD.D  F1, F2, F4
```

NAME:_____

Problem Q4.1.A **8 points**

Fill in the following table to indicate how the given code will execute. Assume all register values are available at the start of execution. The first two rows have been completed for you. (2 points per row)

Inst \ Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
I_1	I	E			C															
I_2		I			E			C												
I_3			I	E					C											
I_4					I	E	C													
I_5								I	E			C								
I_6									I			E			C					

Common mistakes: Not reading all the assumptions on the previous page. Single-issue means two instructions can't start at the same time, and an instruction can't enter the issue stage if it will cause a WAR hazard.

2 points per row, and errors did not propagate. Each row was graded on the assumption that the rows above were correct (even if they weren't).

Problem Q4.1.B **5 points**

Why can't an instruction enter the issue stage if it might cause a WAR or WAW hazard even if the earlier instruction causing the hazard has already started execution? (Hint: feel free to use the code from the previous page under adversarial conditions as an example).

WAR – If for some reason register values aren't latched at the start of the execution stage, letting an instruction into the issue stage that creates a WAR hazard with it could cause that instruction to later read the wrong values. If it did latch the register values at the start of execution it would be hard to provide precise exceptions. Using the code from the problem, if I_1 has already started execution and I_4 is allowed to enter the issue buffer there could be a problem. If I_4 completes before I_1, and it turns out that I_1 faults and needs to be re-executed, it will now be reading the wrong R1.

WAW – If the two instructions have different latencies the wrong value could be stored. If a long latency instruction has started execution, and an another instruction that causes a WAW hazard with it is allowed into the issue stage, the new instruction could finish before the old instruction, causing what should be the older value to overwrite the new value.

NAME:_____

Problem Q4.1.C **8 points**

With renaming we can take care of many of the issues in Q4.1.A. Given unlimited renaming resources, fill the table below to indicate how the same code would run with renaming.

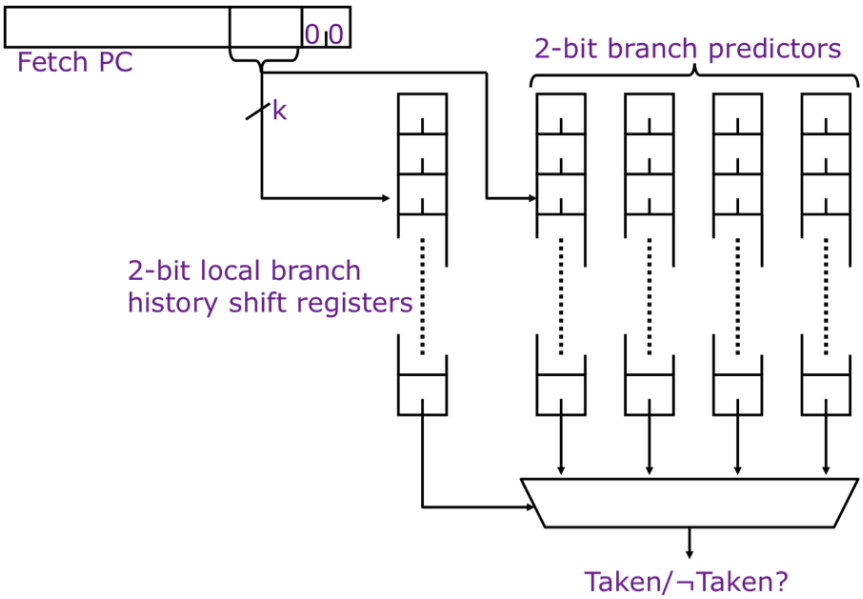
Inst \ Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
I_1	I	E			C															
I_2		I			E			C												
I_3			I	E					C											
I_4				I		E	C													
I_5					I		E			C										
I_6						I				E			C							

Same grading, and mistakes as part A.

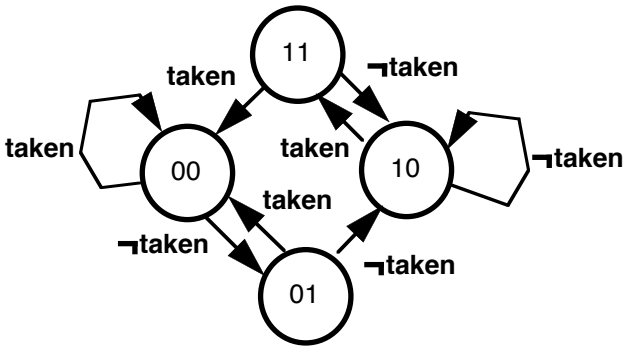
NAME:_____

Problem Q4.2: Branch Prediction 20 points

The two-bit predictor shown in class was used for the Branch History Table (BHT) entries because after training, it would only mispredict for one iteration of a loop (the loop exit) instead of two (loop exit plus first iteration of next execution). If we make a two-level predictor addressed by the local branch history (not global history like in class), we can achieve zero mispredicts for short loops. The figure below shows the scheme for two bits of local branch history.



The last two outcomes of each branch are tracked in a two-bit shift register (1 for taken, 0 for not). The local history is used to address which two-bit predictor to use. That predictor is the same as the one on the problem set and reproduced below. Remember in state 1X we will guess not taken and in state 0X we will guess taken.



NAME:_____

Problem Q4.2.A **8** points

In this problem we will track the results of just one branch. The surrounding code is given below, but every time it is run the loop ends up taking two iterations.

```
LOOP:     LW    R1, 0(R2)
          ADDI  R2, R2, 4
          BNEQZ R1, LOOP
```

Complete the following table (a few iterations have already been done). For the branch predictor columns you only have to write the value when it changes.

System State	Branch Predictor					Branch Behavior	
R1	History	Way 00	Way 01	Way 10	Way 11	Predicted	Actual
36	00	10	10	10	10	N	T
27	01	11				N	T
0	11		11			N	N
25	10					N	T
43	01			11		N	T
0	11		00			N	N
67	10					N	T
69	01			00		T	T
0	11					N	N

Problem Q4.2.B **6** points

After training, using two bits of local history per branch is able to achieve 100% accuracy on loops that always repeat two times. Can it always achieve 100% accuracy on loops that always repeat only once? Why?

Yes. Actually just one bit of local state per branch could get 100% accuracy (after training). After training, when the history bit is 0, it will know to branch, and when it is 1 it will know not to. If two bits of local stage per branch that will correspond to 01 and 10.

Problem Q4.2.C **6** points

To achieve 100% accuracy on loops with more iterations, what costs (performance and hardware) are there? How will they scale?

To handle loops that always repeat N times, each entry in the BHT it will need N bits to store the local branch history and 2^N 2-bit predictors. The training time (in terms of branch predictions) will increase because there is more predictor state. Because the hardware requirements grow exponentially, this does not scale well and this technique should not be used for large loops.

NAME:_____

Problem Q4.3: Importance of Features 18 points

For the following snippets of code, select the single architectural feature that will *most* improve the performance of the code. Explain your choice, including description of why the other features will not improve performance as much and your assumptions about the machine design. The features you have to choose from are: out-of-order issue with renaming, branch prediction, and superscalar execution. Loads are marked whether they hit or miss in the cache.

Problem Q4.3.A 6 points

```
ADD.D F0, F1, F8
ADD.D F2, F3, F8
ADD.D F4, F5, F8
ADD.D F6, F7, F8
```

Circle one:

- Out-of-Order Issue with Renaming
- Branch Prediction
- Superscalar

Superscalar because the instructions have no dependencies so they could be issued in parallel (superscalar could deliver speedup).

The other two techniques will waste hardware because they will offer no speedup. Out-of-Order with renaming will not help because there are no dependencies. Branch prediction is useless since there are no branches.

Problem Q4.3.B 6 points

```
loop: ADD R3 R4 R0
      LD R4, 8(R4) # cache hit
      BNEQZ R4, LOOP
```

Circle one:

- Out-of-Order Issue with Renaming
- Branch Prediction
- Superscalar

Branch prediction is necessary with a tight loop to prevent bubbles in the pipeline.

Superscalars will be limited not only by the branches, but also the WAR and RAW hazards. They will limit how much ILP it can achieve.

Out-of-order with renaming will be limited by the branches. Renaming will take care of the WAR hazard, but the RAW hazard will still limit how much improvement is possible.

NAME:_____

Problem Q4.3.C **6 points**

```
LD   R1 0(R2)   # cache miss
ADD  R2 R1 R1
LD   R1 0(R3)   # cache hit
LD   R3 0(R4)   # cache hit
ADD  R3 R1 R3
ADD  R1 R2 R3
```

Circle one:

- Out-of-Order Issue with Renaming
- Branch Prediction
- Superscalar

Out-of-order with renaming will let the third, fourth, and fifth instruction run while the cache miss is being handled. When the miss completes it only needs to do the second and sixth instruction.

Branch prediction won't help with getting around the cache miss and there are enough hazards to limit a superscalar.

Common mistakes for all parts: To assume superscalar only has disjoint functional units. It can have multiple copies of each (i.e. multiple ALU's).

NAME:_____

Problem Q4.4: Lifetime of Physical Registers 20 points

For this problem, consider an out-of-order CPU that uses a unified physical register file for renaming.

Problem Q4.4.A **7 points**

During class, it was determined that a physical register mapped to a given architectural register can be reused (added to the free list) when the next instruction in program order that writes the same architectural register commits. Why can't it be freed earlier?

(2/7 points) Otherwise there could be instructions still dependent on that result and if the register is freed it could be overwritten prematurely. If a following instruction (say instruction A) commits to the same architectural register it is impossible for there to be any uncommitted instructions that still depend on the old value. If an instruction depends on the old value of that architectural register, it must occur before instruction A in program order. Since commits are done in order, when instruction A commits, all instructions that depend on the old value must have also committed.

Even if there aren't any following instructions that haven't executed, it could still go wrong during an exception. If the next write to the same architectural register hasn't committed and an exception happens, that old value will be needed to correctly roll back state so dependent following instructions can re-execute with the correct data.

Problem Q4.4.B **13 points**

With the renaming scheme shown in class, a new physical register is allocated to an instruction during the decode stage when it enters the reorder buffer and it does not free that physical register until the case described in Q4.4.A. This results in instructions sometimes holding physical registers for a long time, which requires a large number of physical registers. To reduce the number of physical registers, instructions can wait until they issue into execution to be assigned a physical register for their destination.

When an instruction enters the reorder buffer, it is assigned a *virtual* physical register number for its destination. The virtual physical register number is just a tag used to track dependencies until an instruction is assigned a real physical register at issue time. The rename table now holds either virtual or real physical register numbers depending on whether the producer instruction has entered execution or not. Following instructions use the rename table to rename their source registers to either virtual or real physical register numbers. When an instruction executes, it is assigned a physical register, and this is broadcast so the appropriate virtual register tags in the rename table and the issue window can be updated to now point to real physical registers.

NAME:_____

i) By reducing the number of physical registers, it is now possible for this design to deadlock. How could this happen? (7 points)

Consider the following code:

```
LD   R1, 0(R2)      # long latency cache miss
ADD  R3, R1, R7
SUB  R4, R1, R8
# many independent instructions
```

When the load misses, the add and the sub will wait in the issue stage. The independent instructions will all issue, complete, and consume all the physical registers. They will not be able to commit (and free physical registers) because earlier instructions in program order haven't committed. When the load finishes, it will free a physical register. If that register is given to the sub, then the sub instruction will complete and hold that physical register, so now everything is blocking on the add.

Some students correctly identified that an early instruction could block and later instructions could complete but be unable to free, but the description above describes in detail how the instructions that are waiting never can get executed. If the earlier instruction is waiting on something, when that completes it will free up an instruction so those early instructions will not be held up anymore. If the allocation scheme gave the physical register freed by the load to the add, the system also wouldn't deadlock.

ii) Propose a solution to fix the problem in part i). (6 points)

When selecting instructions to issue, if you select the oldest ready instruction you will avoid the case in part i. There is a slight subtlety that might prevent this from working every time. Using the code above, if the instruction the add is blocking on doesn't free a register (store or branch), then when it completes it won't get the register it needs to execute. However instructions shouldn't be dependent on branches or stores unless there is some complex memory queue issue.

Another way to solve the problem is to squash the newest instructions in program order to free up registers. To do this, the CPU will need to roll back like on a branch mispredict.

Alternatively pointers and counters could be used to track the state to guarantee that there are enough physical registers to not deadlock. If there won't be, precedence is given to instructions older in program order. This is described in detail in the original virtual physical register paper.

Try reading the original paper and it should be decently understandable after CS 152 (link only works on campus):
http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00650557

END OF QUIZ