Computer Architecture and Engineering
CS152 Quiz #3

March 28th, 2011

Professor Krste Asanović

Name:___<ANSWER KEY>___

This is a closed book, closed notes exam.
80 Minutes
16 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 yet 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 instruction ask you to explain your choice.

Writing name on each sheet	________	1 Points
Question 1	________	17 Points
Question 2	________	20 Points
Question 3	________	12 Points
Question 4	________	30 Points
TOTAL	________	**80 Points**

Question 1: Register Renaming (17 points)

For this question we will analyze the scheduling of the following code on an out-of-order processor.

Written in C, the code is as follows:

```
#define N 1024
int S[N],A[N],B[N],Y[N];

for(int i = 0; i < N; i++)
  S[i] = A[i] * B[i] + Y[i];
```

The code compiles to the following:

```
I1:   addi $1, $0, 1024
I2:   addi $2, $0, 0
   loop:
I3:   ld   $3, A($2)
I4:   ld   $4, B($2)
I5:   mul  $3, $3, $4
I6:   ld   $4, Y($2)
I7:   add  $3, $3, $4
I8:   st   $3, S($2)
I9:   addi $2, $2,  4
I10:  addi $1, $1, -1
I11:  bnez $1, loop
```

A, **B**, **Y**, and **S** are immediates set by the compiler to point to the beginning of the **A**, **B**, **Y**, and **S** arrays. Register \$1 holds the loop's counter (which counts down from 1024 to zero), register \$2 is used to index the arrays, and registers \$3 and \$4 hold temporary variables as needed. For this ISA, register \$0 is read-only and always returns the value zero.

The processor uses a split instruction window/ROB design, with a unified physical register file (similar to the MIPS R10k). A diagram of the processor is shown in Figure 1 below.

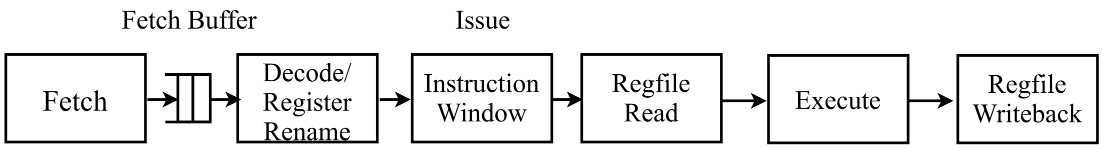

Figure 1. Out-of-order Pipeline with split instruction window and ROB.

All instructions have an effective latency of 3 cycles (dependent instructions can issue 3 cycles after the instruction producing the value issues).

Unfortunately, the ISA being used only has 4 architectural registers - $1,$2,$3, and $4 (remember register $0 is always zero, is read-only, and is not renamed). Also, there are only 8 physical registers (named P1,P2,P3 … through P8).

Q1.A: (9 points)

Complete the following table found on the next page. For each instruction, label the following:

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

The initial state begins with four registers on the free list, in the order of (P1,P2,P3, and P4). The initial state of the retirement rename map table is:

Architectural Register	**Physical Register**
$1	P6
$2	P5
$3	P8
$4	P7

You can assume that there are no instructions in-flight in the beginning.

Complete the following table. This table only covers the first iteration through the loop. The first two instructions have been filled out for you.

ISA Dest. Reg/**Instruction**	**Physical Dest. Register**	**Freed Register**
(\$1) **I1**	P1	P6
(\$2) **I2**	P2	P5
(\$3) **I3**	P3	P8
(\$4) **I4**	P4	P7
(\$3) **I5**	P6	P3
(\$4) **I6**	P5	P4
(\$3) **I7**	P8	P6
(none) **I8**	n/a	n/a
(\$2) **I9**	P7	P2
(\$1) **I10**	P3	P1
(none) **I11**	n/a	n/a

(9 points for 18 entries, so -1/2 for each wrong entry for wrong answers)

Store instructions and branches do not have a destination register.

Registers are freed in program order, so we do not need to know when an instruction is issued or when it writes back. Therefore, P3 is used for I10, *not* P4.

Name _________(answer key)________

Q1.B: Pipeline Stages (8 points)

Circle the correct answer for the following questions.

For some of these questions, multiple answers were generally given full credit if the pipeline could still exhibit correct behavior. Half credit was given if the answer would provide correct execution, but was very sub-optimal in performance.

Which stage allocates entries in the ROB?

`fetch` [✓ `decode`] [✓ `regrename`] [✓ `dispatch`] `issue` `execution` `commit`

Any part of the early in-order pipeline could correctly allocate an entry in the ROB (entries must maintain program order so that instructions are committed in order).

At which stage is an entry in the ROB deallocated?

`fetch` `decode` `regrename` `dispatch` `issue` `execution` [✓ `commit`]

The only correct answer is commit, as the entire purpose of the ROB is to track all in-flight instructions until commit, when it is known the instruction actually executed and will not have to be re-executed or scrapped due to misspeculation.

At which stage is an instruction allocated an entry in the instruction window?

`fetch` [✓ `decode`] [✓ `regrename`] [✓ `dispatch`] `issue` `execution` `commit`

Same answer as allocating ROB entries.

At which stage is an instruction entry in the instruction window deallocated?

`fetch` `decode` `regrename` `dispatch` [✓ `issue`] [✓ `execution`] `commit`

Because issue entries are very expensive, we want to deallocate issue slots as soon as possible, which is when the instruction is issued (this is an advantage to the split inst window/ROB design). However, some pipelines will issue instructions speculatively (e.g., ALU op that is dependent on a speculative load). If a misspeculation occurs, the speculated instruction will have to be re-issued.

Commit is considered incorrect, because otherwise you don't have an instruction window, you have an ROB.

Name ________(answer key)________

At which stage is a store entry allocated in the LD/ST queue?

`fetch` <u>`decode`</u> <u>`regrename`</u> <u>`dispatch`</u> `issue` `execution` `commit`

Entries in the LD/ST queue *must* be in program order to insure the dependency checks between loads and stores are analyzed correctly.

At which stage is a load entry allocated in the LD/ST queue?

`fetch` <u>`decode`</u> <u>`regrename`</u> <u>`dispatch`</u> `issue` `execution` `commit`

Entries in the LD/ST queue *must* be in program order to insure the dependency checks between loads and stores are analyzed correctly.

When is a store performed (i.e., sent to the memory)?

`fetch` `decode` `regrename` `dispatch` `issue` `execution` <u>`commit`</u>

Stores change the architectural state, and thus cannot be sent to memory until the processor *knows* the store will actually be performed. Thus it must wait until commit time (any earlier and it would be speculating that the branch path was correct and no earlier exceptions would occur, etc.).

When is a load performed (i.e., when is data returned that can be used by other dependent instructions)?

`fetch` `decode` `regrename` `dispatch` `issue` <u>`execution`</u> `commit`

(-1/2) for commit (*very* sub-optimal to force loads to be performed in-order at commit time, as one of the biggest advantages of OoO pipelines is allowing loads to run ahead).

Loads do not affect the architectural state, so we can speculate that they are performed before knowing the branch path is correct, no previous exceptions occurred, the load doesn't conflict with earlier stores, etc. Thus we can request from memory the load data and speculatively bypass to other dependent instructions in the Execute stage. Naturally, a lot of undoing may have to occur if a misspeculation occurs, but high performance mandates that loads occur in the Execute stage.

Question 2: Scheduling for Out-of-order Processors (20 points)

The following questions concerns the scheduling of floating-point code on out-of-order processors. For this problem, we will deal with a single-issue out-of-order processor that uses a split instruction window/ROB design, as shown in Figure 2.

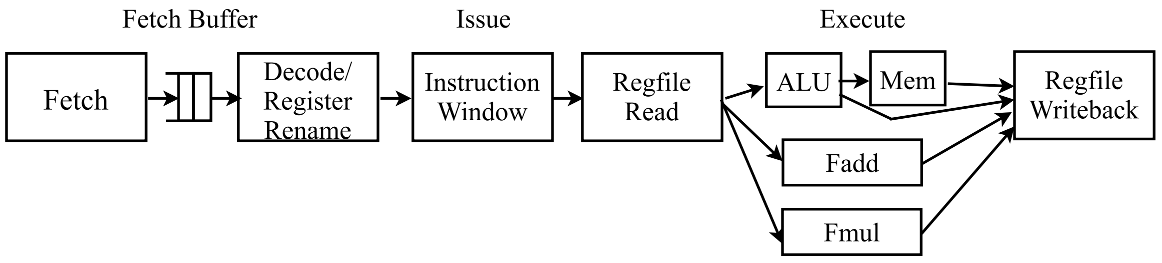

Figure 2. Out-of-order Pipeline with split instruction window and ROB.

The processor contains the following stages:

- Fetch (F), Decode/Rename (D), Issue (I), Regfile Read (R), Execute(X1,X2,...), and Regfile Writeback (W)
- The Execute stage takes a variable number of cycles, depending on the instruction:
    - one cycle for ALU operations (denoted as X1)
    - three cycles for memory operations (X1,X2,X3, which includes the time in the ALU)
    - four cycles for floating-point multiply instructions (X1,X2,X3,X4)
    - two cycles for floating-point add instructions (X1,X2)

You can assume that:

- All functional units are fully pipelined.
- There is no register renaming (not until Q2.B).
- There are two register domains: a set for integer registers (R1,R2,...) and a set for floating-point registers (F1, F2, ...).
- The fetch stage performs perfect branch prediction, and the fetch buffer can hold an infinite number of instructions.
- The issue stage is a buffer of unlimited length that holds instructions waiting to begin execution (aka, the instruction window).
- An instruction will only exit the issue stage if it does not cause a WAR or WAW hazard (in this design, all data is obtained by reading from the register file on instruction issue, so newer instructions must wait on older instructions to read the register file before issuing and thus potentially overwriting the older instructions' sources!).

assumptions, continued:

- Only one instruction can be issued at a time, and if multiple instructions are ready, the oldest will go first.
- An infinite number of instructions may write back to the register file simultaneously.
- The register file bypasses write values to the read ports.
- There is no bypassing between functional units. All operand data is read from the register file.
- Store data is not needed until after the address calculation and can be bypassed from the register file directly to the end of the (X2) stage.

For this problem we will be describing the scheduling of the following code:

```
I1:  ld.d  F1,  A(R1)
I2:  ld.d  F2,  B(R1)
I3:  mul.d F3,  F1, F2
I4:  ld.d  F2,  Y(R1)
I5:  add.d F5,  F3, F2
I6:  st.d  F5,  S(R1)
I7:  addi  R1,  R1, 8
```

A, **B**, **Y**, and **S** are immediates set by the compiler to point to the beginning of the **A**, **B**, **Y**, and **S** arrays. Instructions postfixed (*.d) denote instructions that affect double-precision floating point numbers.

Q2.A: Without Register Renaming (10 points)

Complete Table 1 (found on the following page), indicating which stage each instruction is in for each cycle. There is **no register renaming** available for this question (the *decode* stage still performs work though). Assume all register values are available at the start of the execution of the code. The first two rows have been completed for you.

Q2.B: With Register Renaming (8 points)

Register renaming will take care of some of the hazards that arise in Q2.A.

Given unlimited renaming resources, complete the Table 2 (found on the following page), indicating which stage each instruction is in for each cycle. The first two rows have been completed for you.

Name _________(answer key)________

The code again:

```
I1:  ld.d  F1,  A(R1)
I2:  ld.d  F2,  B(R1)
I3:  mul.d F3,  F1, F2
I4:  ld.d  F2,  Y(R1)
I5:  add.d F5,  F3, F2
I6:  st.d  F5,  S(R1)
I7:  addi  R1,  R1, 8
```

Cycle Inst	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
I_1	F	D	I	R	X1	X2	X3	W												
I_2		F	D	I	R	X1	X2	X3	W											
I_3			F	D	I	I	I	I	R	X1	X2	X3	X4	W						
I_4				F	D	I	I	I	I	R	X1	X2	X3	W						
I_5					F	D	I	I	I	I	I	I	I	R	X1	X2	W			
I_6						F	D	I	I	I	I	I	I	I	R	X1	X2	X3	W	
I_7							F	D	I	I	I	I	I	I	I	R	X1	W		

Table 1: Question 2.A (No register renaming)

Cycle Inst	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
I_1	F	D	I	R	X1	X2	X3	W												
I_2		F	D	I	R	X1	X2	X3	W											
I_3			F	D	I	I	I	I	R	X1	X2	X3	X4	W						
I_4				F	D	I	R	X1	X2	X3	W									
I_5					F	D	I	I	I	I	I	I	I	R	X1	X2	W			
I_6						F	D	I	I	I	I	I	I	I	R	X1	X2	X3	W	
I_7							F	D	I	R	X1	W								

Table 2: Question 2.B (With register renaming)

While in this case register renaming may not appear to be a win because I6 still doesn't finish any faster, register renaming will allow further iterations of the loop to be performed earlier because register renaming breaks the anti-dependencies between loop iterations.

Q2.C: Increasing the width of an OOO processor (2 points)

While you are simulating a new design for the next-generation Intel processor (an aggressively speculating out-of-order superscalar processor), you notice that increasing the width of your design too far causes the *cycles / instruction* to actually ***increase*** (*note:* you are fully simulating the L1 instruction and data caches, as well as simulating all speculation mechanisms). What reason can you supply to your superiors as to why a wider processor actually *hurts* the cycles / instruction metric?

A wider processor allows more instructions to be inflight before a misspeculation is discovered. The speculated instructions could end up polluting the caches or could be using up memory bandwidth that will hurt overall performance.

Some answered that the number of cycles spent rolling back would increase. While the *number* of instructions you kill will increase, the cycles to roll back will be the same (the width increased, so you can roll back more instructions per cycle).

Other students answered "a lot more instructions are inflight and have to be killed". while true, you fetch more instructions so it will be a wash with killing more instructions.

Question 3: Branch Prediction (12 points)

For this problem, we are interested in the following snippet of code:

```
int array[N] = {....};

for(int i = 0; i < N; i++)
  if (array[i] != 0)
    array[i] = array[i] + 1;
```

Using the disassembler we get:

```
   addi $n, $0, N
   addi $i, $0, 0
loop:
   ld   $a, array($i)
   beqz $a, endif
   addi $a, $a, 1
   st   $a, array($i)
endif:
   addi $i, $i,  4
   addi $n, $n, -1
   bnez $n, loop
```

(where **N** is some integer used to specify the size of *array*).

Q3.A: Limited Machine Resources(2 points)

For this piece of code, would it be better to spend hardware resources on branch prediction or register renaming? Why?

Branch prediction. Especially for this code, there are too many dependencies between instructions to allow for register renaming to provide a benefit, and without branch prediction we can not speculate past branches anyways to allow for register renaming to break the anti-dependencies between iterations.

Name ________(answer key)________

Q3.B: Prediction Accuracy (4 points)

The processor that this code runs on uses a 512-entry branch history table (BHT), indexed by PC [10:2]. Each entry in the BHT contains a 2-bit counter, initialized to the 00 state.

Each 2-bit counter works as follows: the state of the 2-bit counter decides whether the branch is predicted taken or not taken, as shown in Table 3. If the branch is actually taken, the counter is incremented (e.g., state 00 becomes state 01). If the branch is not taken, the counter is decremented. The counter saturates at 00 and 11 (a not-taken branch while in the 00 state keeps the 2-bit counter in the 00 state).

State	Prediction
00	not taken
01	not taken
10	taken
11	taken

Table 3. 2-bit counter state table.

If array = {0,1,-3,4,1}, what is the prediction accuracy for the two branches found in the above code for five iterations of the loop, using the 512-entry BHT described above?

With a 512-entry BHT, both branches will safely index different entries in the BHT.

5 iterations

The for loop will be {T,T,T,T,N} (taken, not-taken).
However, the 2-bit counter will predict {N,N,T,T,T} as it takes two cycles to learn the branch is taken. So the predictor is only accurate on 2 out of the 5 predictions.

The if branch will be {T,N,N,N,N}. The 2-bit counter will predict {N,N,N,N,N}. So it is accurate 4 out of 5 times.

-1 point for missing the last iteration that exits the for loop.

BEQZ (if) __(4/5) = 80%__
BNEZ (for loop) __(2/5) = 40%__

Name _________(answer key)________

Q3.C: Prediction Accuracy - Small BHTs (2 points)

If array = {0,1,-3,4,1}, what is the prediction accuracy for the two branches found in the above code for five loop iterations, using a **ONE**-entry BHT (i.e., all branches map to the same two-bit entry).

With a one-entry BHT, both branches will alias to the same entry in the BHT.

The for loop will be {T,T,T,T,N}.
The if branch will be {T,N,N,N,N}.

Here's a table tracking the 2-bit predictor state ("00->01" means that the 2bc started in the 00 state, branch was taken, so it ended up in the 01 state):

	0	1	-3	4	1
if	00 -> 01	10->01	10->01	10->01	10->01
for	01->10	01->10	01->10	01->10	01->00

And here's a table tabulating when the predictor was correct:

	0	1	-3	4	1
if	X	X	X	X	X
for	X	X	X	X	√

We can see that the if branch was 0 for 5, and the for loop branch was 1 out of 5.
-(1/2) points for missing the last iteration of the for loop.

BEQZ (if) __(0/5) = 0%___
BNEZ (for loop) __(1/5) = 20%___

Q3.D: Static Hints (2 points)

For this question, assume that the compiler can specify statically which way the processor should predict the branch will go. If the processor sees a "branch-likely" hint from the compiler, it predicts the branch is taken and does **NOT** update the BHT with this branch (i.e., any branches the compiler can analyze do not pollute the BHT).

Which branches, if any, can the compiler provides hints for, if the input array for the compiler's test runs varies widely (assume the compiler must be fairly confident in the accuracy of a branch to be predicted statically before it labels a branch)?

Circle only one of the following answers and explain:

A. The BEQZ branch (if)

[✓ B. The BNEZ branch (for loop)]

C. Both branches

D. Neither branch

The if branch is data dependent, and the test arrays are specified as being relatively random and thus hard to predict.

The for loop will always be taken except for the last iteration, which makes it easy to predict.

Q3.E: BTBs (2 points)

One of your coworkers suggests adding a BTB (branch target buffer). The BTB can hold two-entries (fully-associative), is indexed by the current PC, and allows the processor's fetch stage to immediately fetch along the target PC's path if the entry tagged by the current PC is found in the BTB.

For this piece of code used in Question 3, and assuming the baseline is a processor with only **one** entry for a BHT, would it be more advantageous to follow your coworker's advice and add a BTB, or would it be better to add static hints from the compiler? *Explain your reasoning.*

Depending on your justifications, the answer could be either the BTB or the static hints.

BTB: The key is to identify that while the actual prediction rates of both would be about the same (especially regarding the for loop), the BTB can redirect the instruction stream much earlier (as it only relies on the current PC), and thus hide a lot of the branch latency. Meanwhile, the BHT/static hints must wait until branch address calculation and instruction decode, which can be many, many cycles later. Thus, the BTB is a win.

-- or --

Static Hints: Some students recognized while both schemes have about the same prediction accuracy, the BTB takes up some area and may not be worth the area/power/timing costs, if that is what we are considering to be the more important metric.

-- or --

Static Hints: Depending upon the interpretation of the interaction of the BTB and BHT, one can make a case for using the one-entry BHT augmented with static hints. The argument is as follows: the BTB can redirect immediately, and will accurately (and quickly) redirect the for loop branch. However, the BHT has the power to overrule the BTB later in the pipeline, if it disagrees. The BHT for this problem has only one entry, and because it is aliased by both the always-taken for loop and the unpredictable if branch, the BHT will provide unpredictable predictions. If the BHT predicts "not-taken" for the for loop, it will undo all of the correct predictions from the BTB, which won't be resolved until the Execute stage (if you assume the BHT is allowed to overrule the BTB).

A real processor will make sure the BHT is much larger (in address coverage) than the BTB to prevent this very issue (although addresses can still alias), because the BHT should technically provide much better prediction abilities than the BTB. It is also possible that the BHT may only overrule the BTB if the BTB predicts not-taken.

----
notes:

-(1/2) points for not recognizing the reduced misspeculation penalty provided by the BTB.

The BTB can have its own prediction bits, so it can invalidate entries it learns are not taken (so it won't always mispredict the if branch is it sees one zero, followed by a bunch of non-zeros).

Question 4: Iron Law for OoO Superscalar Processors (30 points)

Mark whether the following modifications will cause each of the categories to **increase, decrease**, or whether the modification will have **no effect**. You can assume the baseline processor is a standard out-of-order, superscalar processor with register renaming and branch prediction. **Explain your reasoning** to receive full credit.

Assume that in each case the rest of the machine remains unchanged.

Name _________(answer key)________

	Instructions / Program	Seconds / Cycle	Cycles / Instruction
wider instruction issue	no change doesn't affect the ISA	increases a wider instruction issue complicates the wiring of the issue logic	decreases Ideally issuing more instructions per cycle will decrease CPI (i.e., IPC is increasing). This is the goal of widening issue.
more physical registers	no change doesn't affect the ISA	increases larger register file will mean longer to read from and write to. physical register specifiers will also have to be larger, increasing the size of the micro-ops, tag compares, etc.	decreases More physical registers provide more renaming resources, which will help prevent stalls due to running out of available physical registers.
add more entries to the branch target buffer (BTB)	no change doesn't affect the ISA	increases BTB entries are large in size and require an expensive tag comparison - or - no change, if BTB is argued to not be on the critical path	decreases Ideally, adding more entries allows the processor to accurately redirect the PC on a larger number of branches earlier in the pipeline, cutting down on mispredicts and bubbles caused by waiting on the target address calculation.
add static branch hints to the compiler	no change, since we are just adding a new opcode/ type of branch to provide a hint on existing branches - or - increase, as the compiler re-factors and possibly duplicates code to make branches more predictable	increases slightly more complicated decode - or - unchanged, doesn't change logic much and decode is probably not on the critical path, or can assume no changes to ISA	decreases Static hints are added to help the processor better predict branches that can be analyzed at compile-time.
recompile software with a newer, better version of an optimizing compiler	most likely increase many performance optimizations (loop unrolling, register blocking, software pipelining, etc.) will cause the code to inflate - or - however, one could argue that some optimizations, particularly if aiming for "-Os" type optimizations, could shrink the code size, or that optimizing processors will be better at removing redundant code	no change no change is made to the actual hardware	decreases Ideally, your new compiler does a better job than the last one at providing better performance, in the way of hiding instruction latencies (e.g., loop unrolling) or making branches easier to predict, which will affect the measured CPI for a given program. - or - if you argued that your new compiler optimizes for code size, then one could easily imagine the tradeoff was smaller code size for a higher CPI - or - increases, as CPI becomes worse because unoptimized code is easier for hardware to accelerate.

END OF QUIZ