Computer System Architecture
6.823 Quiz #2
April 1st, 2016
Professors Daniel Sanchez and Joel Emer

Name: _______Solutions_______________

This is a closed book, closed notes exam.
80 Minutes
14 Pages (+2 Scratch)

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.
- Pages 15 and 16 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

Part A	________	20 Points
Part B	________	44 Points
Part C	________	16 Points
Part D	________	20 Points
TOTAL	________	**100 Points**

Part A: Complex Pipelining (20 points)

Ben Bitdiddle is designing a processor with a complex pipeline, shown below:

A	Address (PC) generation
F1	Instruction Fetch Stage 1
F2	Instruction Fetch Stage 2
F3	Instruction Fetch Stage 3
B	Branch Address Calculation / Begin Decode
D	Complete Decode
J	Steer Instructions to Functional Units
R	Register File Read
E	Execute
⋮	Remainder of execution pipeline

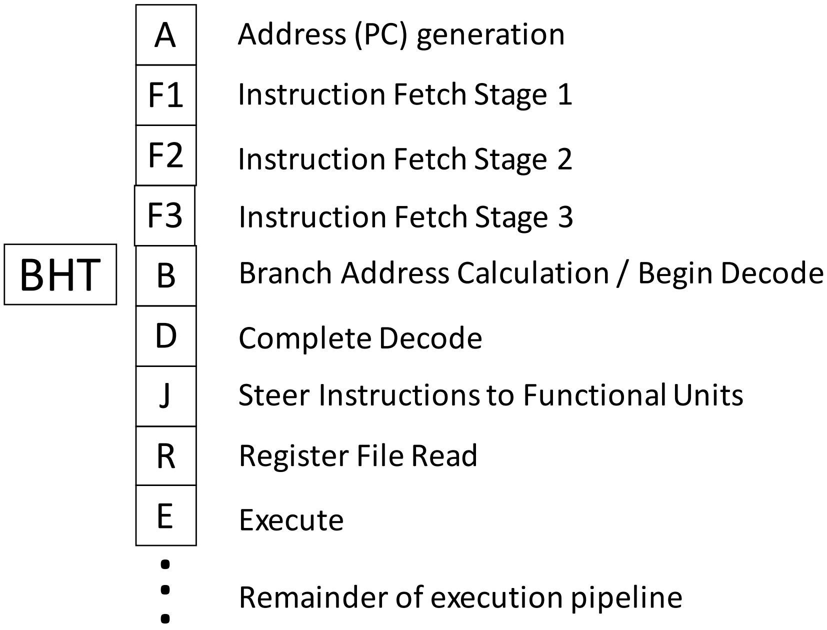
The processor has the following characteristics:

- Issues at most one instruction per cycle.
- Branch addresses are known at the end of the B stage.
- Branch conditions (taken / not taken) are known at the end of the R stage.
- Branches go through the pipeline without any stalls or queueing delays.

For this question, assume there are no control flow instructions other than conditional branches (i.e., no unconditional jumps, jump register, etc).

Question 1 (5 points)

Ben adds a branch history table (BHT) to the pipeline as shown right. With this addition, fetches work as follows. The A stage fetches the instruction at PC+4 by default. In the B stage (Branch Address Calc/Begin Decode), a conditional branch instruction (BEQZ/BNEZ) looks up the BHT. If a branch is predicted to be taken, later instructions are flushed and the PC is redirected to the calculated branch target address.


A
Address (PC) generation
F1
Instruction Fetch Stage 1
F2
Instruction Fetch Stage 2
F3
Instruction Fetch Stage 3
BHT
B
Branch Address Calculation / Begin Decode
D
Complete Decode
J
Steer Instructions to Functional Units
R
Register File Read
E
Execute
Remainder of execution pipeline

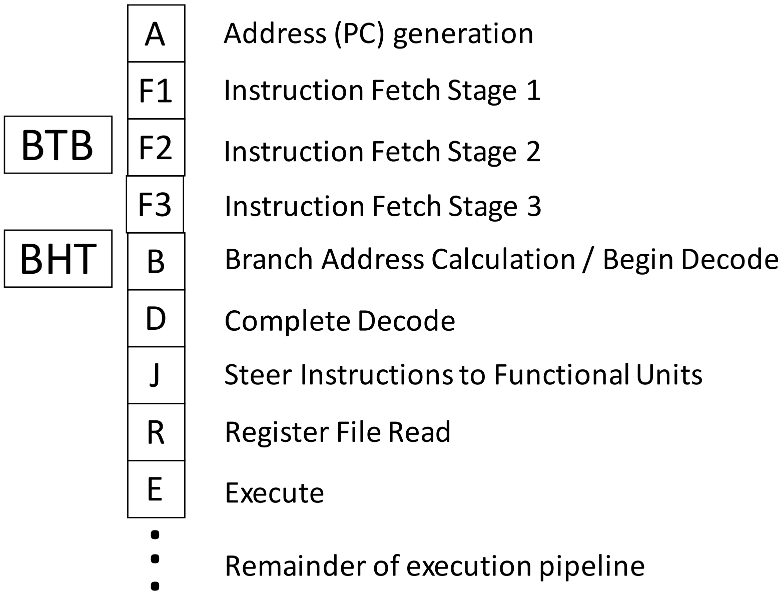
Fill in the table below. First, list all the possible predictions made by the BHT (one per row). Then, fill in each cell with the branch misprediction penalty (in instruction fetches wasted) for each combination of prediction and actual branch outcome.

	Misprediction penalty if branch outcome is	
BHT Prediction	**Taken**	**Not Taken**
Taken	4	7
Not Taken	7	0

Question 2 (10 points)

To improve performance further, Ben decides to add a branch target buffer (BTB) as well. The BTB holds entry_PC, target_PC pairs *for branches predicted to be taken*. Assume that, if the branch is taken, the target_PC predicted by the BTB is always correct for this question (i.e., there is no aliasing).

On a BTB lookup, if there is a match with the current PC, the PC is redirected to the target_PC stored the BTB, unless the PC is redirected by an older instruction.


A
Address (PC) generation
F1
Instruction Fetch Stage 1
BTB
F2
Instruction Fetch Stage 2
F3
Instruction Fetch Stage 3
BHT
B
Branch Address Calculation / Begin Decode
D
Complete Decode
J
Steer Instructions to Functional Units
R
Register File Read
E
Execute
Remainder of execution pipeline

Fill in the table below. First, list all the possible combinations of predictions made by the BTB and BHT (one per row). Then, fill in each cell with the branch misprediction penalty (in instruction fetches wasted) for each combination of predictions and actual branch outcome.

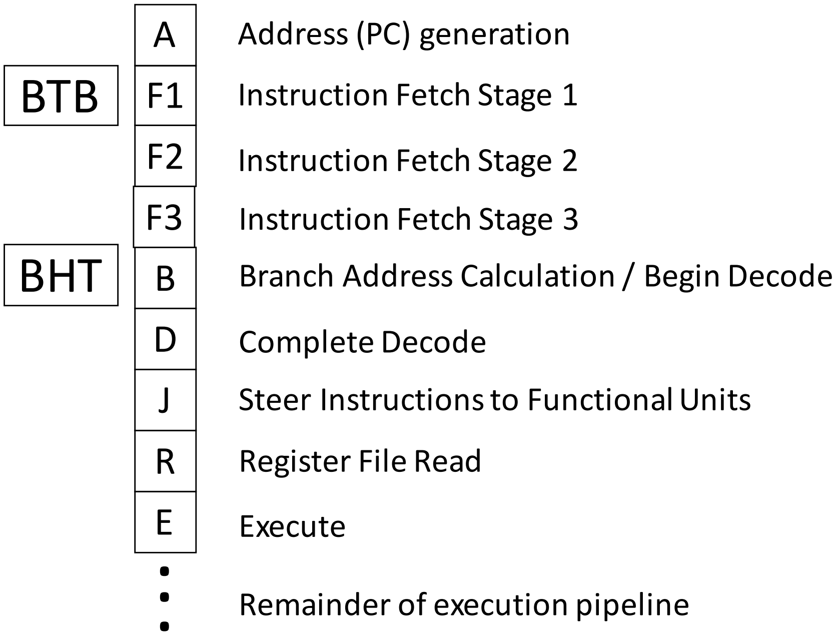
	Misprediction penalty if branch outcome is	
BTB+BHT Predictions	**Taken**	**Not Taken**
BTB: Hit, BHT: Taken	2	7
BTB: Miss, BHT: Taken	4	7
BTB: Miss, BHT: Not Taken	7	0
BTB: Hit, BHT: Not Taken	7	4

Full credit even if last case is missing.

Question 3 (5 points)

Alyssa P. Hacker designs a BTB that can have its prediction in stage F1 (one cycle earlier than Ben's BTB), but is smaller (i.e., has fewer entries) than Ben's BTB, and thus misses more often. Your job is to find when this is a good tradeoff, i.e., when the BTB's lower latency compensates for its lower accuracy.

As in Question 2, the BTB holds entry_PC, target_PC pairs *for branches predicted to be taken*. Assume that, if the branch is taken, the target_PC predicted by the BTB is always correct for this question (i.e., no aliasing).


A
Address (PC) generation
BTB
F1
Instruction Fetch Stage 1
F2
Instruction Fetch Stage 2
F3
Instruction Fetch Stage 3
BHT
B
Branch Address Calculation / Begin Decode
D
Complete Decode
J
Steer Instructions to Functional Units
R
Register File Read
E
Execute
Remainder of execution pipeline

On your benchmark suite, you find that:

- When Ben's BTB hits (i.e., predicts that the branch is taken), the branch is always taken.
- The only inaccuracy incurred by Alyssa's BTB is that, because of its smaller capacity, it suffers from capacity misses, predicting not-taken for a fraction F of the branches that Ben's BTB (correctly) predicts taken.

For what range of values of F is Alyssa's faster but less accurate BTB a better choice?

Based on the specifications in the problem, the only scenarios leading to different latencies in Ben's and Alyssa's case are:

a) BTB hit in Ben's processor and Alyssa's processor: 2 cycle penalty (Ben), 1 cycle penalty (Alyssa).
b) BTB hit in Ben's processor and BTB miss in Alyssa's processor: 2 cycle penalty (Ben), 4 cycle penalty (Alyssa) [note that the branch is actually taken in this case].

All other scenarios result in similar penalties in both Ben's and Alyssa's processors.

We want:
$F \times 4 + (1-F) \times 1 <= 2$
$F <= 1/3$

Part B: Out-of-order Processing (44 points)

Question 1 (30 points)

This question uses the out-of-order machine described in the quiz 2 handout. We describe events that affect the initial state shown in the handout. Label each event with one of the actions listed in the handout. If you pick a label with a blank (_____), you also have to fill in the blank using the choices (i—v) listed below. If you pick R. Illegal action, state why it is an illegal action.

Example: Assume P6 is available. Instruction `I14` is issued and its effective address matches load buffer entry 4.
Answer: (L, iv): Check the correctness of a speculation on memory address and find an incorrect speculation. (You can simply write L, iv)

a) Instruction `I12` is issued and reads store buffer entry `3`.
(B, v): Satisfy a dependence on memory value by bypassing a speculative value

b) Assume `P6` is available. Instruction `I14` is issued and its effective address matches load buffer entry `1`.
(K, iv): Check the correctness of a speculation on memory address and find a correct speculation

c) Instruction `I14` is issued and its effective address does not match any entry in the load buffer.
(K, iv): Check the correctness of a speculation on memory address and find a correct speculation

d) Instruction `I19` hits in the BTB and reads entry `1`.
(E, ii): Satisfy a dependence on PC value by speculation using a dynamic prediction

e) Instruction `I18` updates the global history register from `00010110` to `00101100` (shift in a 0 from the right).
(I, iii): Speculatively update a prediction on branch direction using greedy value management. G: Write a speculative value using greedy value management is also acceptable.

f) Assume all instructions through `I7` have committed. `I8` commits and returns `P12` to the free list.
Q: Commit correctly speculated instruction, and free log associated with greedily updated values

g) Assume all instructions through `I8` have committed. `I9` commits and returns `P1` and `P4` to the free list.
R: Illegal operation. P1 cannot be returned to the free list.

h) Assume all instructions through `I9` have committed. `I10` commits and updates the speculative bit of entry 3 in the store buffer.
P: Commit correctly speculated instruction, and mark lazily updated values as non-speculative

i) Assume the value of `P5` is now available, `I11` is issued and discovers the branch was predicted incorrectly. The global history register is updated from `00010110` to `00000101` (i.e., right shift by 2 bits).
N: Abort speculative action and cleanup greedily managed values

j) Assume the value of `P5` is now available, `I11` is issued and discovers the branch was predicted incorrectly. Entry `4` in the store buffer is cleared.
M: Abort speculative action and cleanup lazily managed values

Question 2 (14 points)

For SW instructions, the pre-dispatch stage checks for empty slots in the store buffer. If it does not find an empty slot, the pipeline stalls. If it finds a slot, that slot is *reserved* for the SW instruction and the instruction proceeds to the dispatch stage and is inserted into the ROB.

The table below lists several statistics for SW instructions as they move through the pipeline.

Frequency of SW instructions	0.2
Average waiting time in ROB before execute	10 cycles
Average execute time (ie. writing to store buffer)	1 cycle
Average waiting time after execute, before commit	9 cycles
Average waiting time in store buffer, after commit	10 cycles

Assume an infinite-size ROB for this question.

a) What is the minimum number of slots required in the store buffer to make sure that SW instructions entering the pre-dispatch stage do not limit the throughput of the system?

Using Little's Law, the minimum size is : 0.2 x (10 + 1 + 9 + 10) = 6

b) On average, for what fraction of time does a *reserved* store buffer slot contain valid data?

(1 + 9 + 10) / (10 + 1 + 9 + 10) = 2 / 3

c) Now, suppose instructions are dispatched to ROB without waiting for a store buffer slot. Instead, a store buffer slot is assigned when a SW instruction is issued from the ROB. What is the minimum number of slots required in the store buffer to ensure that SW instructions do not limit the throughput of the machine?

Using Little's Law: 0.2 x (1 + 9 + 10) = 4

Part C: Reliability (16 points)

Question 1 (10 points)

Indicate whether the following intervals in the lifetime of a bit in a cache are ACE, unACE, or unknown. Assume that reads and writes originate from ACE instructions. Assume that each cache line holds a single byte, and memory is byte-addressed.

	Write-through cache	**Write-back cache**
Fill-to-Read	ACE	ACE
Read-to-Read	ACE	ACE
Write-to-Write	unACE	unACE
Read-to-Write	unACE	unACE
Write-to-Read	ACE	ACE
Write-to-Evict	unACE	ACE

Question 2 (6 points)

What change (increase, decrease, or stay the same, and why) do you expect in the following scenarios, to the:

a) AVF of store buffer when a single-thread out-of-order processor is enhanced to support SMT?
Assuming store buffer size is kept constant: Increases. Utilization of store buffer increases.

b) AVF of write-through cache if miss rate increases?
Decreases. Lines are evicted, so less likely that an error will propagate to an ACE event.

c) AVF of ROB, if branch predictor accuracy decreases?
Decreases. ACE instructions are resident in the ROB for smaller periods of time (instructions are likely to be aborted more on average).

Part D: Multithreading (20 points)

For this problem, we are interested in evaluating the effectiveness of multithreading on the following code, which computes the sum of all the elements of an array.

```
int A[N];
…
int sum = 0;
for (int i = 0; i < N; i++)
  sum += A[i];
```

Here is the corresponding MIPS assembly code:

```
;; Assume:
;; R1 holds value of sum; initialized to 0
;; R2 holds number of iterations remaining; initialized to N
;; R3 holds address of A[i]; initialized to base address of A

loop: ld    R4, 0(R3)
      add   R1, R1, R4
      addi  R2, R2, -1
      addi  R3, R3, 4
      bnez  R2, loop
```

Assume the following:

- The processor can issue one instruction per cycle.
- All instructions except loads execute in a single cycle.
- The end-of-loop branch is always predicted correctly.
- Our system does not have a cache. Each load accesses main memory and takes 30 cycles.
- The load/store unit is fully pipelined.
- After the processor issues a load, it can continue executing instructions until it reaches an instruction that depends on the result of an outstanding load.

Question 1 (5 points)

How many cycles does it take to execute one iteration of the loop in steady-state for a single-threaded processor?

34 cycles

Question 2 (5 points)

Now consider a simple multithreaded pipeline. Threads are switched every cycle using a fixed round-robin schedule. If a thread cannot issue an instruction on its turn, a bubble is inserted into the pipeline.

Each thread executes the code above. What is the minimum number of threads we need to fully utilize the processor? (i.e., no pipeline bubbles in steady state)

30 threads. To hide the latency of the load operation of a particular thread we require at least 29 other threads.

Question 3 (10 points)

Reorder the original code sequence to minimize the number of threads needed to fully utilize the multithreaded pipeline. How many threads do you need?

Original core sequence:

```
loop: ld   R4, 0(R3)
      add  R1, R1, R4
      addi R2, R2, -1
      addi R3, R3, 4
      bnez R2, loop
```

Write reordered code sequence:

```
loop: ld   R4, 0(R3)
      addi R2, R2, -1
      addi R3, R3, 4
      add  R1, R1, R4
      bnez R2, loop
```

The two `addi` instructions are independent of the `ld` instruction, and may be executed without waiting for the result of the load. The cycles at which the different instructions are executed is shown below. To hide the latency between the `ld` and `add` instruction, we require:

$3N + 1 >= 30 + 1$
$N >= 10$

```
loop: ld   R4, 0(R3)          1
      addi R2, R2, -1         1+N
      addi R3, R3, 4          1+2N
      add  R1, R1, R4         1+3N
      bnez R2, loop           1+4N
```

Number of threads needed to fill the pipeline with reordered code: _10_____

Scratch Space

Use these extra pages if you run out of space or for your own personal notes. We will not grade this unless you tell us explicitly in the earlier pages.