Computer System Architecture
6.823 Quiz #2
April 5th, 2019

Name: _____SOLUTIONS_____

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

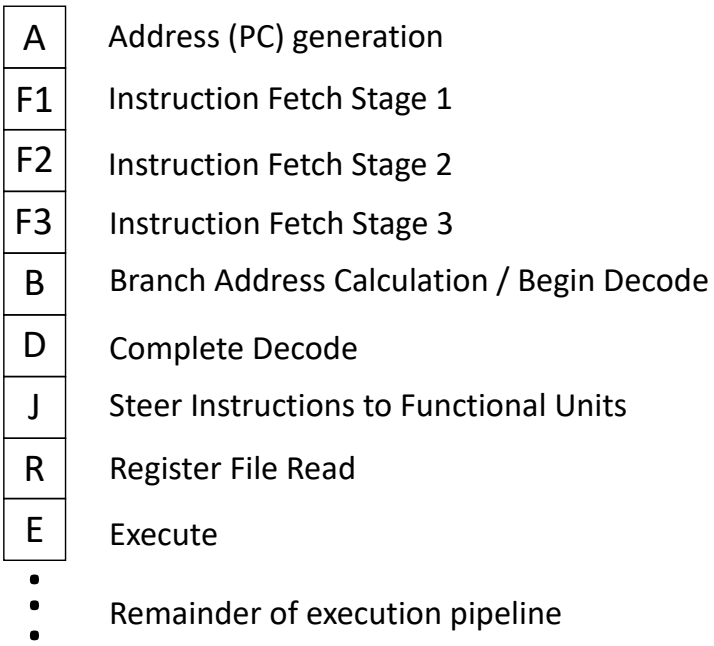
Notes:

- 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.
- Show your work to receive full credit.
- 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 17 and 18 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

Part A	_____	28 Points
Part B	_____	30 Points
Part C	_____	20 Points
Part D	_____	22 Points
TOTAL	_____	**100 Points**

Part A: Branch Prediction (28 points)

Consider a processor with the following pipeline stages:



The processor has the following characteristics:

- The A (Address generation) stage fetches the instruction at address PC+4.
- The branch target address is known at the end of the B stage.
- The branch condition is known at the end of R stage. If the branch was mispredicted, the processor squashes all previous instructions in stages A to J.

To analyze the performance of this processor, we will use the following program:

```
int count = 0;
for(int i = 0; i <= 1000000; i++)
{
    if(A[i] == 0)  //Branch B1
    {
        count++;
    }

    if(B[i] == 0)  //Branch B2
    {
        count--;
    }
}     //Branch LP
```

The assembly code for this program is as follows. Assume that registers `R4` and `R5` hold the base addresses of arrays A and B, respectively, and that `R6` holds the value of `count`.

```
Address                Instruction
0x1000                 ANDI R1, 0
0x1004          BEGIN: LW   R2, 0(R4)
0x1008                 LW   R3, 0(R5)
0x100C             B1: BNEZ R2, B2
0x1010                 ADDI R6, R6, 1
0x1014             B2: BNEZ R3, END
0x1018                 SUBI R6, R6, 1
0x101C            END: ADDI R4, R4, 4
0x1020                 ADDI R5, R5, 4
0x1024                 ADDI R1, R1, 1
0x1028                 SUBI R7, R1, 1000000
0x102C             LP: BNEZ R7, BEGIN
```

For the following questions, Array A contains a repeating pattern of [0, 0, 1], and B contains a repeating pattern of [0, 0, 0, 1]:

A = [0, 0, 1, 0, 0, 1, ... 0, 0, 1]
B = [0, 0, 0, 1, 0, 0, 0, 1, ... 0, 0, 0, 1]

Question 1 (6 points)

In steady state, What is the average number of cycles lost per iteration for each of the following branches?
Average cycles lost = (how often branch is taken) * (branch mispredict penalty)

(a) B1

(1/3) * 7 = 7/3

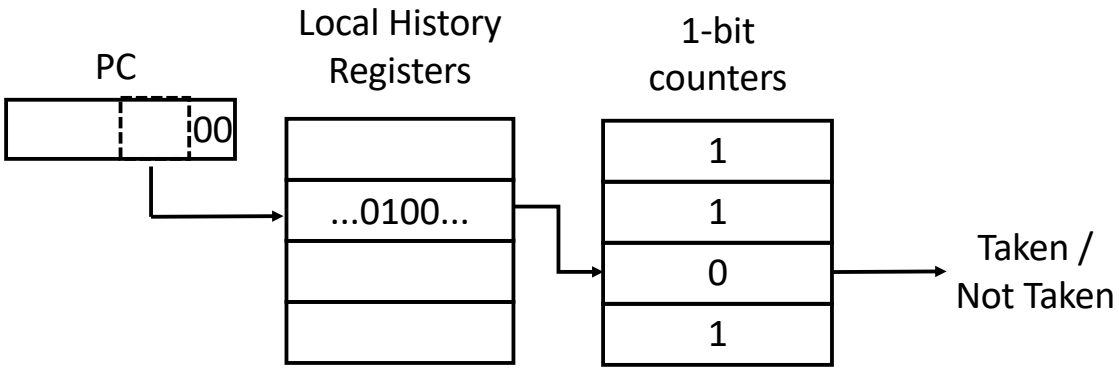
(b) B2

(1/4) * 7 = 7/4

(c) LP

1 * 7 = 7

Ben Bitdiddle adds a two-level branch predictor to the **B stage** to improve branch prediction accuracy, as shown in the figure below.



The predictor consists of a set of **local history registers** indexed by lower bits of the PC (excluding the least significant 2 bits since instructions are 4B-aligned). Each local history register contains the last several outcomes of a given branch. Every time a new branch is encountered, a 1 is shifted in from the right for taken branches and 0 for non-taken branches. Each local history register value is used to index into a table of 1-bit counters. Each counter predicts taken if the given counter entry is 1, and predicts not taken if it is 0.

With this addition, fetches work as follows. The A stage fetches PC+4, like before. If the fetched instruction is a branch, the B stage then looks up the two-level branch predictor with the branch PC. If the predictor predicts taken, all following instructions in the pipeline in stages A to F3 are squashed, and the PC is redirected to the calculated branch address.

Question 2 (4 points)

Ben first wants to map each branch in his code to a distinct local history registers by using the least significant bits of the PC. With the given indexing scheme, what is the **smallest** size of the local history registers table (i.e., the number of local history registers) required to ensure that no two branches map to the same local history register? Note the table size must be a power of two.

Table size must be big enough to prevent aliasing between B1 (0x100C) and LP (0x102C).

=> Need 4 bits => 2^4 = 16

Question 3 (4 points)

Ben wants to decide how many bits each local history register should have. What is the **minimum** number of bits required to achieve perfect prediction in steady state? Remember that A = [0, 0, 1, 0, 0, 1, ... 0, 0, 1] and B = [0, 0, 0, 1, 0, 0, 0, 1, ... 0, 0, 0, 1].

We can start with the least # of bits that covers at least one pattern:
3 bits => Does not work for pattern 100 (need to predict 1 for B1, and 0 for B2)
4 bits => Does not work for pattern 0100 (predict 1 for B1, predict 0 for B2)
5 bits => Does not work for pattern 00100 (predict 1 for B1, predict 0 for B2)
Lastly, 6 bits is guaranteed to work since none of the patterns of B1 overlap with those of B2 when you consider the 3 possible patterns of B1 (001001, 010010, 100100)

Question 4 (4 points)

Alyssa P. Hacker adds a Branch Target Buffer (BTB) to the **F1** stage to further improve performance. The BTB holds a mapping of the branch PC to the target PC for branches that it predicts to be taken. Assume that, if the branch is taken, the target PC predicted by the BTB is always correct (i.e., there is no aliasing).

With this addition, fetches work as follows. As before, the A stage fetches PC+4. The BTB is looked up in the F1 stage. Upon a hit, the BTB redirects control flow to the target PC and squashes the following instruction in the A stage. The B stage redirects control flow with the two-level predictor as explained in Question 2.

In the table below, fill in how many cycles are lost to branches for each scenario.

BTB prediction	**2-Level Predictor prediction**	**Cycles lost to branches if:**	
		Taken	**Not taken**
Taken	Taken	1	7
Taken	Not Taken	7	4
Not Taken	Taken	4	7
Not Taken	Not Taken	7	0

Question 5 (5 points)

Assume that for the B2 branch, the BTB always hits. Using the combination of BTB and the two-level predictor we designed in Questions 2-4, how many cycles are lost, on average per loop iteration, due to the **B2 branch**? (Full credit will be given for a correct formula that depends on the values from your answers to questions 2-4, even if those are not correct.)

Average cycles lost = (ratio of taken branches). * (cycles lost to branch if branch is taken) + (ratio of not taken branches) * (cycles lost to branch if branch is not taken)
= (1/4) * (1) + (3/4) * (4) = 3.25 cycles per iteration

Question 6 (5 points)

Consider the case where B now contains the repeating pattern [1, 0, 0, 1], while array A stays the same. That is, the array contents are:

A = [0, 0, 1, 0, 0, 1, ... 0, 0, 1]
B = [1, 0, 0, 1, 1, 0, 0, 1, ... 1, 0, 0, 1]

Does our two-level predictor still work as well as it did on the previous pattern? Why or why not? Explain.

Yes. Like in Question 4, none of the 3 patterns of A (001001, 010010, 100100) overlap with those of B if we use 6 bits. In fact, we can get away with 5 bits here.

Part B: Out-of-order Execution (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—vii) listed below. If you pick "R. Illegal action", state why it is an illegal action. If in doubt, state your assumptions.

Example: Instruction `I17` hits in the BTB and reads entry `1`.
Answer: (E, ii): Satisfy a dependence on PC value by speculation using a dynamic prediction. (You can simply write E, ii)

a) Assume the value of physical register `P7` is now available, which is `2004`. Instruction `I14` is issued, finds a matching address on instruction `I15` in the load buffer, and aborts instruction `I15`.

Since this problem was vaguely worded in terms of what happens when instruction `I15` aborts, we gave full credit for the following answers:
(L, iv), (M), (N)

b) Instruction `I8` finishes execution and writes result to physical register `P2`.

(G): Write a speculative value using greedy data management

c) Assume physical register P6 becomes available and holds a value of ~~0~~1. Instruction `I13` executes and finds that the branch is indeed not taken.

(K,iii): Check the correctness of a speculation on branch direction and find a correct speculation

d) Assume all instructions up to `I6` commit. `I7` commits and adds physical register `P9` to the free list.

(Q): Commit correctly speculated instruction, and free log associated with greedily updated values

e) Assume instruction `I13` is a mispredicted branch. Upon detecting the misprediction, the rename table is restored to a previous snapshot.

(N): Abort speculative action and cleanup greedily managed values

f) Assume instruction `I8` finishes execution and the value of `P2` becomes available. Instruction `I9` is issued and reads physical registers `P1` and `P2`.

(B): Satisfy a dependence on Register value by bypassing a speculative value

g) Assume instruction `I16` is decoded and found to be a branch. The branch is predicted taken, and the global history register is updated from `10011010` to `00110101` (shift in a `1` from the right).

(I): Speculatively update a prediction on branch direction using greedy value management

h) Assume instruction `I16` writes to register `R4`. `I16` is dispatched, grabs a new physical register `P11` from the free list, and updates the rename table entry of `R4` to `P11`.

(G): Write speculative value using greedy data management

i) Assume the value of physical register `P2` becomes available. `I9` issues, finishes execution, and frees physical register `P1`.

(R): Illegal operation. P1 is not freed until I9 commits.

j) Assume entry `1` of the BTB becomes invalid. Instruction `I18` is fetched from address `0x3c`.

(E): Satisfy a dependence on branch direction by speculation using a dynamic prediction.
OR
(D): Satisfy a dependence on branch direction by speculation using a static prediction.

Part C: Out-Of-Order Processor Design (20 points)

You are given an out-of-order processor with unlimited decode, issue, commit bandwidth. The processor's ISA has 16 architectural registers. To achieve an efficient design, you are asked to calculate the average occupancy of various structures for different implementation alternatives. We will use the following code:

```
while(true) {
     i = i + 1
     B[i] = A[i]
     sum += A[i] * 2
}
```

The loop can be unrolled (thus eliminating branches) and translated into the following instruction sequence, with six instructions per iteration:

```
I0    addi   r1, r1, #4
I1    addi   r2, r2, #4
I2    lw     r3, 0(r1)
I3    sw     r3, 0(r2)
I4    muli   r3, r3, #2
I5    add    r4, r4, r3
I6    addi   r1, r1, #4
I7    addi   r2, r2, #4
I8    lw     r3, 0(r1)
I9    sw     r3, 0(r2)
I10   muli   r3, r3, #2
I11   add    r4, r4, r3
                ⋮
```

Below are two different diagrams that show the cycles at which instructions are decoded, issued, and committed in steady state (use the one you find more convenient). First, the following table shows these cycles for the instructions in the Nth loop iteration:

Instruction Number	Opcode	Decode	Issue	Commit
6N	addi	2N	2N+1	2N+6
6N+1	addi	2N	2N+1	2N+6
6N+2	lw	2N	2N+3	2N+6
6N+3	sw	2N+1	2N+5	2N+6
6N+4	muli	2N+1	2N+4	2N+7
6N+5	add	2N+1	2N+5	2N+7

For example, instruction I8 (lw) is decoded at cycle 2, issued at cycle 5, and committed at cycle 8.

Second, the waterfall diagram below describes how the instructions are scheduled in steady state.

Time:	2N	2N+1	2N+2	2N+3	2N+4	2N+5	2N+6	2N+7	2N+8	2N+9
I6N (addi)	D	I					C			
I6N+1 (addi)	D	I					C			
I6N+2 (lw)	D			I			C			
I6N+3 (sw)		D				I	C			
I6N+4 (muli)		D			I			C		
I6N+5 (add)		D				I		C		

Hint: To answer these questions, you do not need to derive the instruction scheduling for more iterations**.**

Question 1 (5 points)

Assume that we have an data-in-ROB design that works as follows:

- At decode stage: an instruction is decoded and written to the ROB. The instruction grabs an ROB entry at the beginning of the cycle.
- At issue stage: the instruction enters the execution pipeline.
- At commit stage: the instruction leaves the ROB at the end of the cycle.

On average, how many ROB entries are used in steady state?

Use Little's Law.
Throughput = 3 instructions per cycle
Average Latency = (7 + 7 + 7 + 6 + 7 + 7) / 6 = 41/6 cycles
N = T * L = 3 * (7 + 7 + 7 + 6 + 7 + 7) / 6 = 20.5 entries

Question 2 (5 points)

To simplify the ROB implementation, we introduce a separate, smaller issue queue that holds instructions waiting to be issued:

- At decode stage: an instruction is decoded. The instruction grabs an ROB entry as well as an entry in the issue queue at the beginning of the cycle.
- At issue stage: the instruction leaves the issue queue at the end of the cycle.
- At commit stage: the instruction leaves the ROB at the end of the cycle.

On average, how many issue queue entries are used in steady state?

Throughput = 3 instructions per cycle
Average Latency = (2 + 2 + 4 + 5 + 4 + 5) / 6 = 22/6 cycles
N = T * L = 3 * (2 + 2 + 4 + 5 + 4 + 5) / 6 = 11 entries

Question 3 (5 points)

We change our ROB from the data-in-ROB design to having a unified physical register file that holds both speculative and non-speculative register values. We use an ISA that has **16 architectural registers**. Instructions interact with the unified register file as follows:

- At decode stage: an instruction is decoded. At the beginning of the cycle, the instruction grabs an ROB entry and also grabs a free physical register from the free list.
- At issue stage: the instruction is issued.
- At commit stage: the instruction leaves the ROB and releases the previously mapped physical register at the end of the commit stage.

In steady state, how many physical registers are in use on average?

Note that store instructions do not allocate physical registers.

Throughput = 2.5 instructions per cycle
Average Latency = (7 + 7 + 7 + 7 + 7) / 5 = 7 cycles
N = T * L = 2.5 * 7 = 17.5 registers.

We also need to take into account 16 physical registers that are initially mapped.
Hence, total physical registers = 16 + 17.5 = 33.5

Question 4 (5 points)

We design our processor to have a unified load-store buffer that holds both pending loads and stores. Assume that on average stores occupy the buffer for 5 cycles, and loads occupy the buffer for 3 cycles. In steady state, how many load-store buffer entries are in use on average?

Calculate number of entries for stores and loads separately, then add them.

Throughput of loads = 0.5 loads per cycle
N_load = 0.5 * 3 = 1.5
Throughput of stores = 0.5 stores per cycle
N_store = 0.5 * 5 = 2.5

Thus, total comes to 4 entries

Part D: Multithreading (22 points)

In this part, we will investigate the tradeoffs between different fetch policies for a processor that supports simultaneous multithreading (SMT).

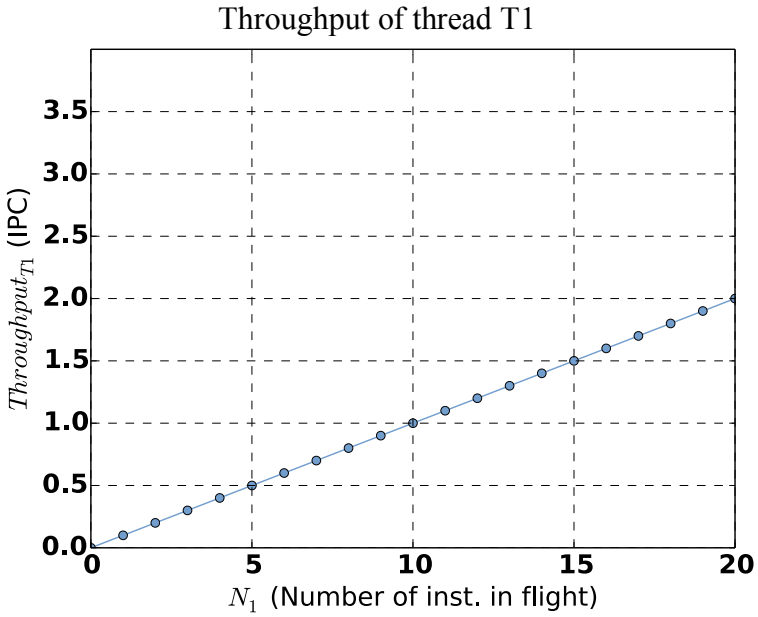
Ben Bitdiddle is given a superscalar, out-of-order processor with support for SMT that has a 20-entry ROB. Ben wishes to run 2 threads T1 and T2 on this machine while maximizing the machine's aggregate throughput, measured in terms of **committed instructions per cycle (IPC)**. To aid his decision, he first runs each thread in isolation on the machine to see the relationship between the number of instructions in flight and the throughput.

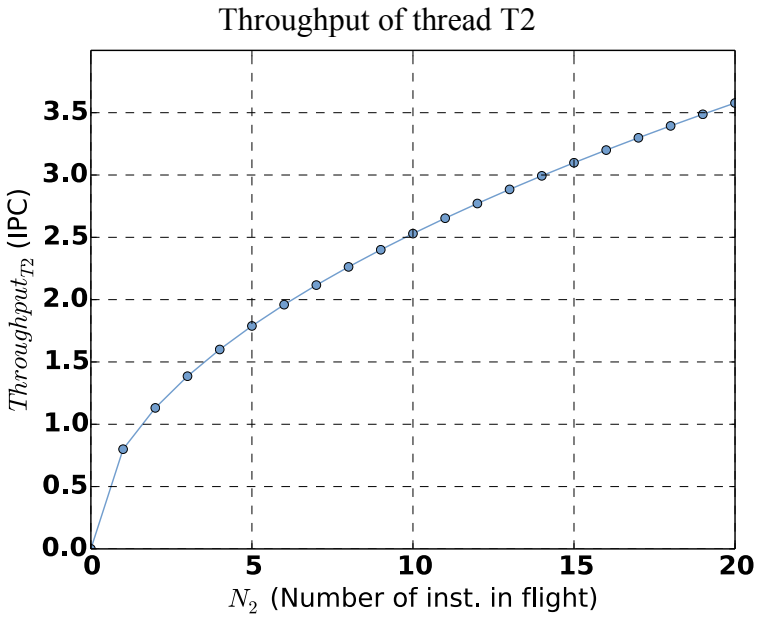
Ben finds that each thread's throughput can be analytically expressed as follows, where N_i is the number of instructions in flight for thread T_i:

$$Throughput_{T1} = 0.1 \times N_1$$
$$Throughput_{T2} = 0.8 \times \sqrt{N_2}$$
$$N_1 + N_2 = 20$$

In the questions below, assume that the only resource both threads contend on is the ROB, and that both threads always have instructions ready to be issued into the ROB. Thus, the aggregate throughput of the machine depends only on how the threads share ROB entries.

For your convenience, we provide plots that visualize the throughput profiles:





We also provide a table of each thread's throughput for different numbers of instructions in flight:

N_1	$Throughput_{T1}$
0	0
1	0.1
2	0.2
3	0.3
4	0.4
5	0.5
6	0.6
7	0.7
8	0.8
9	0.9
10	1
11	1.1
12	1.2
13	1.3
14	1.4
15	1.5
16	1.6
17	1.7
18	1.8
19	1.9
20	2

N_2	$Throughput_{T2}$
0	0
1	0.8
2	1.13
3	1.39
4	1.6
5	1.79
6	1.96
7	2.12
8	2.26
9	2.4
10	2.53
11	2.65
12	2.77
13	2.88
14	2.99
15	3.09
16	3.2
17	3.29
18	3.39
19	3.49
20	3.58

Question 1 (8 points)

Ben's initial instinct is to use a round-robin policy. The round-robin policy alternates which thread to fetch instructions from at every cycle.

a) Which thread would have more instructions in flight with the round-robin policy? Explain. (4 points)

Round-robin will eventually equalize the throughput of two threads, since at steady state the commit throughput of two threads will be the same as the fetch throughput.

b) Calculate the machine's aggregate throughput with the round-robin policy. (4 points)

T1 = T2 = 1.6 when N1 = 16 and N2 = 4. Thus, aggregate throughput = 3.2 IPC

Question 2 (8 points)

Alyssa proposes to use the ICOUNT policy. Recall that ICOUNT issues instructions into the ROB from the thread with the fewest instructions in flight.

a) Calculate the machine's aggregate throughput with the ICOUNT policy. (4 points)

ICOUNT policy implies that entries from each thread are adjusted until they occupy the same number of ROB entries. Thus, we calculate the aggregate throughput when N1 = N2 = 10, which is T1 + T2 = 1 + 2.53 = 3.53

b) Fill in the table below to indicate how the number of instructions in flight for each thread and the aggregate throughput change with the ICOUNT policy compared to round-robin. For each row, place a check mark on the increase or decrease column. (4 points)

	Increase	**Decrease**
Number of Inst. in flight for T1		√
Number of Inst. in flight for T2	√	
Aggregate throughput	√	

Question 3 (6 points)

Do either round-robin or ICOUNT maximize the machine's aggregate throughput? If so, explain why this is. If not, how should the machine allocate ROB entries among threads to achieve maximum throughput? Describe the general strategy, then calculate the maximum aggregate throughput with the new allocation policy.

The optimal policy given two thread profiles that are monotonically increasing is to find a point where the derivatives of the two are equal:

$$\frac{d}{dN_1}(0.1 \times N_1) = \frac{d}{dN_2}(0.8\sqrt{N_2})$$

We could build such a system by greedily allocating more entries to one thread until the marginal gain in throughput for that thread equals the marginal loss of throughput in the other thread.

Solving for above, we get N1 = 4 and N2 = 16, thus giving us an aggregate throughput of 3.6 IPC

Maximum Aggregate Throughput: _____3.6_____

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.