Assignment 3
Due October 17, 2011

You may type or write your answers by hand. If you write by hand, make sure it is clearly presented. **Do not use pencils but ink**. Please put your name and student ID clearly on the submitted assignment.

You may make use of reasonable assumptions of your own for data that might be missing in the problem texts, provided that they are explicitly and clearly stated.

You may submit a partial answer to a problem. The grading will account for this.

Question 0, Feedback (1 pt extra credit)

How many hours did you spend working on this homework assignment?

Question 1 (2 pts)

You are tasked with designing a new processor microarchitecture, and you are trying to figure out how best to allocate your hardware resources. Which of the hardware and software techniques you learned in Chapter 2 should you apply? You have a list of latencies for the functional units and for memory, as well as some representative code. Your boss has been somewhat vague about the performance requirements of your new design, but you know from experience that, all else being equal, faster is usually better. Start with the basics. Figure 1 provides a sequence of instructions and list of latencies.

Loop:	LD	F2,0(Rx)	Memory LD	+4
I2:	DIVD	F8,F2,F0	Memory SD	+1
I3:	MULTD	F2,F6,F2	Integer ADD, SUB	+0
I4:	LD	F4,0(Ry)	Branches	+1
I5:	ADDD	F4,F0,F4	ADDD	+1
I6:	ADDD	F10,F8,F2	MULTD	+5
I7:	ADDI	Rx,Rx,#8	DIV	+12
I8:	ADDI	Ry,Ry,#8		
I9:	SD	F4,0(Ry)		
I10:	SUB	R20,R4,Rx		
I11:	BNZ	R20,Loop		
	(a)		(b)	

Figure 1: Sample code and functional unit latencies for Question 1.

What would be the baseline performance (in cycles, per loop iteration) of the code sequence in Figure 1(a), if no new instruction's execution could be initiated until the previous instruction's execution had completed? Ignore front-end fetch and decode . Assume for now that execution does not stall for lack of the next instruction (*i.e.*, the

instruction cache always hits), but only one instruction can be issued per cycle. Assume the branch is taken, and that there is a one-cycle branch delay slot.

Question 2 (2 pts)

Think about what the latency numbers in Figure 1(b) really mean—they indicate the number of cycles a given function requires to produce its output, nothing more. If the overall pipeline stalls for the latency cycles of each functional unit, then you are at least guaranteed that any pair of back-to-back instructions (a "producer" followed by a "consumer") will execute correctly. But not all instruction pairs have a producer/consumer relationship. Sometimes two adjacent instructions have nothing to do with each other. How many cycles would the loop body in the code sequence in Figure 1(a) require if the pipeline detected true data dependencies and only stalled on those, rather than blindly stalling everything just because any one functional unit is busy? Show the code with <stall> inserted where necessary to accommodate stated latencies. Again, assume that only one instruction can be issued per cycle, that the branch is taken, and that there is a one-cycle branch delay slot. (Hint: An instruction with latency "+2" needs two <stall> cycles to be inserted into the code sequence. Think of it this way: a 1-cycle instruction has latency 1 + 0, meaning zero extra wait states. So latency 1 + 1 implies one stall cycle; latency 1 + *N* has *N* extra stall cycles.)

Question 3 (3 pts)

Computers spend most of their time in loops, so multiple loop iterations are great places to speculatively find more work to keep CPU resources busy. Nothing is ever easy, though; the compiler emitted only one copy of that loop's code, so even though multiple iterations are handling distinct data, they will appear to use the same registers. To keep multiple iterations' register usages from colliding, we rename their registers. Figure 2 shows example code that we would like our hardware to rename.

A compiler could have simply unrolled the loop and used different registers to avoid conflicts, but if we expect our hardware to unroll the loop, it must also do the register renaming. How? Assume your hardware has a pool of temporary registers (call them T registers, and assume there are 64 of them, T0 through T63) that it can substitute for those registers designated by the compiler. This rename hardware is indexed by the src (source) register designation, and the value in the table is the T register of the last destination that targeted that register. (Think of these table values as producers, and the src registers are the consumers; it doesn't much matter where the producer puts its result as long as its consumers can find it.) Consider the code sequence in Figure 2. Every time you see a destination register in the code, substitute the next available T, beginning with T9. Then update all the src registers accordingly, so that true data dependencies are maintained. Show the resulting code. (Hint: See Figure 3.)

```
Loop:  LD      F4,0(Rx)
I2:    MULTD   F2,F0,F2
I3:    DIVD    F8,F4,F2
I4:    LD      F4,0(Ry)
I5:    ADDD    F6,F0,F4
I6:    SUBD    F8,F8,F6
I7:    SD      F8,0(Ry)
```

Figure 2: Sample code for register renaming practice (Question 2).

```
…
I4:    LD      T9,0(Rx)
I5:    ADDD    T10,F0,T9
…
```

Figure 3: Hint: expected output of register renaming.

Question 4 (3 pts)

Assume a five-stage single-pipeline microarchitecture (fetch, decode, execute, memory, write back) and the code in Figure 4. All ops are 1 cycle except LW and SW, which are 1 + 2 cycles, and branches, which are 1 + 1 cycles. There is no forwarding. Show the phases of each instruction per clock cycle for one iteration of the loop. How many clock cycles per loop iteration are lost to branch overhead (*i.e.*, how many cycles do we spend performing branch evaluation without also performing useful work)?

```
Loop:  LW      R3,0(R0)
I2:    LW      R1,0(R3)
I3:    ADDI    R1,R1,#1
I4:    SUB     R4,R3,R2
I5:    SW      R1,0(R3)
I6:    BNZ     R4,Loop
```

Figure 4: Code loop for Question 4.

Question 5 (4 pts)

Let's consider what dynamic scheduling might achieve here. Assume a microarchitecture as shown in Figure 5. Assume that the ALUs can do all arithmetic ops (MULTD, DIVD, ADDD, ADDI, SUB) and branches, and that the Reservation Station can dispatch at most one operation to each functional unit per cycle (one op to each ALU plus one memory op to the LD/ST unit).

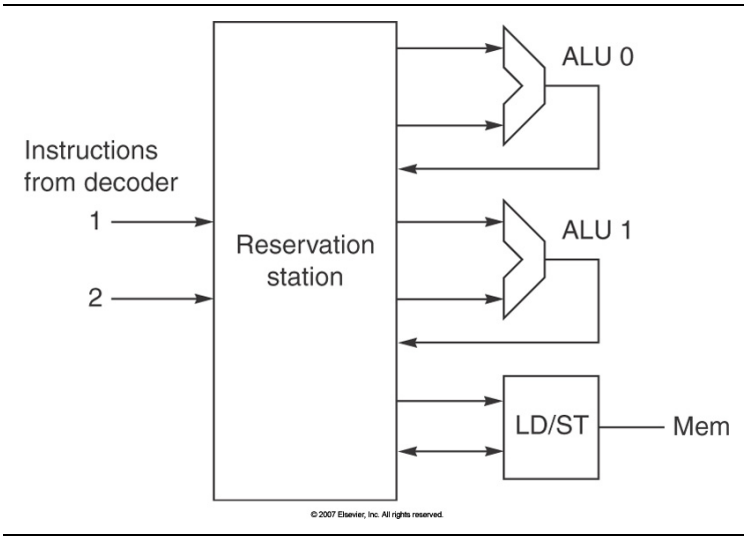

Figure 5: An out-of-order microarchitecture.

a. Suppose all of the instructions from the sequence in Figure 1 are present in the RS, with no renaming having been done. Indicate where register renaming would improve performance. *Hint:* Look for RAW and WAW hazards. Assume the same functional unit latencies as in Figure 1. Assume that functional units are fully pipelined, and that each can start one new operation each cycle.
b. Suppose the register-renamed version of the code from part (a) is resident in the RS in clock cycle *N,* with latencies as given in Figure 1. Show how the RS should dispatch these instructions out-of-order, clock by clock, to obtain optimal performance on this code. Assumptions about the RS in (a) hold here. Also assume that results must be written into the RS before they're available for use; i.e., there is no bypassing, but results written in the first half of the clock cycle can be read in the second half. How many clock cycles does the code sequence take?
c. Part (b) lets the RS try to optimally schedule these instructions. But in reality, the whole instruction sequence of interest is not usually present in the RS. Instead, various events clear the RS, and as a new code sequence streams in from the decoder, the RS must choose to dispatch from among the instructions it has. Suppose that the RS is empty. In cycle 0 the first two register-renamed instructions of this sequence appear in the RS. Assume it takes 1 clock cycle to dispatch any op, and assume functional unit latencies are as they were for Question 1. Further assume that the front end (decoder/register-renamer) will continue to supply two new instructions per clock cycle (and that there is no limit to the number of instructions held by the RS). Show the cycle- by-cycle order of dispatch of the RS. How many clock cycles does this code sequence require now?
d. If you wanted to improve the results of part (c), which would have helped most: (1) another ALU; (2) another LD/ST unit; (3) full bypassing of ALU results to subsequent operations; (4) cutting the longest latency in half? What's the speedup?

Question 6 (6 pts)

Besides studying microarchitecture techniques, to really understand computer architecture you must also program computers. Getting your hands dirty by directly modeling various microarchitectural ideas is better yet. Write a C or Java program to model a (2,1) branch predictor. Your program will read a series of lines from a file named `hw3-history.txt` (available on the course website—see Figure 6).

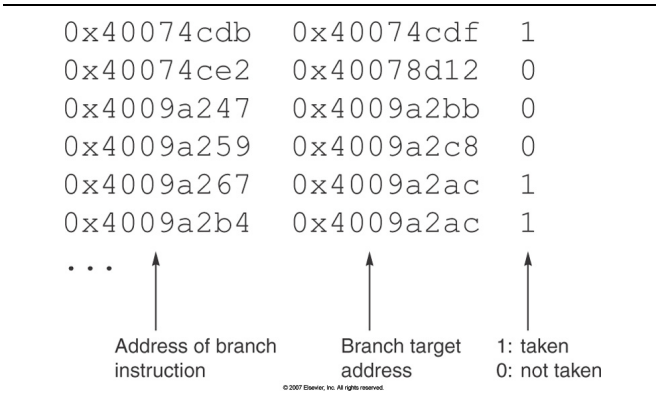

Figure 6: Sample history.txt input file format.

Each line of that file has three data items, separated by tabs. The first datum on each line is the address of the branch instruction in hex. The second datum is the branch target address in hex. The third datum is a 1 or a 0; 1 indicates a taken branch, and 0 indicates not taken. The total number of branches your model will consider is, of course, equal to the number of lines in the file. Assume a direct-mapped BTB, and don't worry about instruction lengths or alignment (*i.e.*, if your BTB has four entries, then branch instructions at 0x0, 0x1, 0x2, and 0x3 will reside in those four entries, but a branch instruction at 0x4 will overwrite BTB[0]). For each line in the input file, your model will read the pair of data values, adjust the various tables per the branch predictor being modeled, and collect key performance statistics. The final output of your program will look like that shown in Figure 7. Make the number of BTB entries in your model a command-line option.

Write a model of a simple four-state branch target buffer with 64 entries.

a. What is the overall hit rate in the BTB? A hit occurs when the same branch is used to access a particular entry twice in a row. A miss occurs when a different branch is used to index the same entry.
b. What is the overall branch misprediction rate on a cold start (the fraction of times a branch was correctly predicted taken or not taken, regardless of whether that prediction "belonged to" the branch being predicted)?
c. Find the most common branch. What was its contribution to the overall number of correct predictions? (Hint: Count the number of times that branch occurs in the history.txt file, then track how each instance of that branch fares within the BTB model.)
d. What is the effect of a cold start versus a warm start? To find out, run the same input data set once to initialize the history table, and then again to collect the new set of statistics.

e. Cold-start the BTB six more times, with BTB sizes 1, 2, 4, 8, 16, and 32. Graph the resulting seven misprediction rates. Also graph the seven hit rates.
f. Submit the well-written, commented source code for your branch target buffer model.

Exercise 2.13 (a)

Number of BTB hits: 54390. Total brs: 55493. Hit rate: 99.8%

Exercise 2.13 (b)

Incorrect predictions: 1562 of 55493, or 2.8%

Exercise 2.13 (c)

<a simple unix command line shell script will give you the most common branch—show how you got it here.>

Most signif. branch seen 15418 times, out of 55493 tot brs ; 27.8%

MS branch = 0x80484ef, correct predictions = 19151 (of 36342 total correct preds) or 52.7%

Exercise 2.13 (d)

Number of hits in BTB: 54390. Total brs: 55493. Hit rate: 99.8%

Incorrect predictions: 1103 of 54493, or 2.0%

Exercise 2.13 (e)

BTB Length	mispredict rate
1	32.91%
2	6.42%
4	0.28%
8	0.23%
16	0.21%
32	0.20%
64	0.20%

Figure 7: Sample program output format.