MS108: Computer System 1

Spring 2013

Homework #2

Due: Two Weeks from Assignment

TA: Haopeng Liu

Course Email: ms108_2013@163.com

Collaboration Policy

These homework sets will be extremely valuable as tools for learning the material and for doing well on the midterm and final. You are required to obey the following rules:
(a) Each student should write out their solution independently and in their own words.
(b) Same applies to programming assignments – you should do your own coding.
Above all, make sure that you understand the solution to these homework problems. They really are assigned to help you understand the material and be prepared for the types of problems on the midterm and final!

Q1. Scoreboarding vs. Tomasulo's Algorithm

A shortcoming of the scoreboard approach occurs when multiple functional units that share input buses are waiting for a single result. The units cannot start simultaneously, but must serialize. This is not true in Tomasulo's algorithm. Give a code sequence that uses no more than 10 instructions and shows this problem. Assume the same hardware configuration as in the lectures. Indicate where Tomasulo's algorithm can continue, but the scoreboard approach must stall. Assume the following latencies.

Instruction Producing Result	Instruction Using Result	Latency in Clock Cycles
FP ALU op	Another FP Alu Op	3
FP ALU op	Store double	2
Load Double	FP ALU Op	1
Load Double	Store Double	1

Q2. Branch Prediction

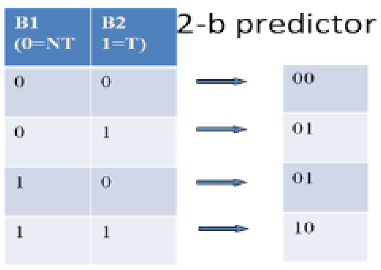
Suppose we have a program with the following sequence of statements. It has three branches as indicated by B1, B2 and B3.

…
if (a<b) then a=2*a; # branch B1
if (c>b) then c=c-b; # branch B2
if (a>c) then a=a-b; # branch B3

…
The instruction sequence corresponding to the above statements is shown in Fig. 1 in assembly language. In Fig. 1, register R1 is used for the variable a, R2 for b and R3 for c. R4 is a register to store temporary results. We maintain a (m,n) predictor for each branch and the predictor for the branch B3 is illustrated in the following table (Fig. 2).

```
        …
S1:     SUB  R4, R1, R2  ;   R4=R1-R2
B1:     BGE  R4, S2      ;   if R4≥0, then branch to S2 (B1 branch)
        ADD  R1, R1, R1  ;   R1=R1+R1
S2:     SUB  R4, R3, R2  ;   R4=R3-R2
B2:     BLE  R4, S3      ;   if R4≤0, then branch to S3 (B2 branch)
        SUB  R3, R3, R2  ;   R3=R3-R2
S3:     SUB  R4, R1, R3  ;   R4=R1-R3
B3:     BLE  R4, S4      ;   if R4≤0, then branch to S4 (B3 branch)
        SUB  R1, R1, R2  ;   R1=R1-R2
S4:     …
```

Fig. 1: The sequence of instructions



B1 (0=NT	B2 1=T)		2-b predictor
0	0	→	00
0	1	→	01
1	0	→	01
1	1	→	10

Fig. 2: The (m,n) predictor for branch B3

a) For the (m, n) predictor, what is the parameter m and what is the parameter n?

b) Suppose at certain time instance, the variables a=26, b=50 and c=46. The program counter (PC) points the first instruction at label S1 (SUB R4, R1, R2). Suppose the state of predictor of B3 at the time instance is shown in Fig. 2. According to this predictor, what prediction will be made for the branch B3 (TAKEN or NOT TAKEN)? Explain the reason.

c) Follow the conditions of the question b. When the program has just finished the execution of the branch B3 (i.e., PC becomes more than B3), what will be the state of the predictor of B3?

d) Suppose we can use up to 10000 bits for dynamic branch prediction using this (m,n) predictor scheme. How many entries can we hold in the cache at most? Assume the number of entries is a power of 2, and each entry corresponds to a different instruction address. (Hint: m and n are determined in question a)

Q3. Code Scheduling

The following code calculates the floating-point expression E = A + B + C * D, where the memory addresses of A, B, C, D, and E are stored in R1, R2, R3, R4, and R5, respectively:

```
L.S F0, 0(R1)
L.S F1, 0(R2)
ADD.S F0, F0, F1
L.S F2, 0(R3)
L.S F3, 0(R4)
MUL.S F2, F2, F3
ADD.S F0, F0, F2
S.S F0, 0(R5)
```

X.S means performing operation X on single-precision data. For example, L.S means loading a single- precision number. Consider the memory load/store unit as a functional unit in parallel with the FP ALU units. Assume the following latencies:

Load → FP ALU: 2 cycles
FP multiplication → FP ALU: 4 cycles
FP multiplication → Store: 3 cycles
FP addition → FP ALU: 2cycles
FP addition → Store: 2cycles

As an example, if a load instruction (e.g., L.S) is issued in cycle 1, an ALU instruction that uses the load instruction's result can be issued in cycle 4 and proceed in the pipeline without stalls. Assume that all functional units are fully pipelined and ignore any write back conflicts on the register file.

a) Suppose we execute the above code segment on an in-order pipelined machine. Show the stalls we need to insert between instructions, and calculate the number of cycles this code sequence would take to execute (i.e., the number of cycles between the issue of the first load instruction and the issue of the final store, inclusive). For simplicity, you can think of the "issue" cycle (or the cycle in which an instruction is issued) as the cycle immediately before an instruction enters its execution functional unit for execution.

b) Reorder the instructions in the code sequence to minimize the execution time. Show the new instruction sequence, and indicate the position and number of stalls we need to insert between instructions.

Q4. VLIW

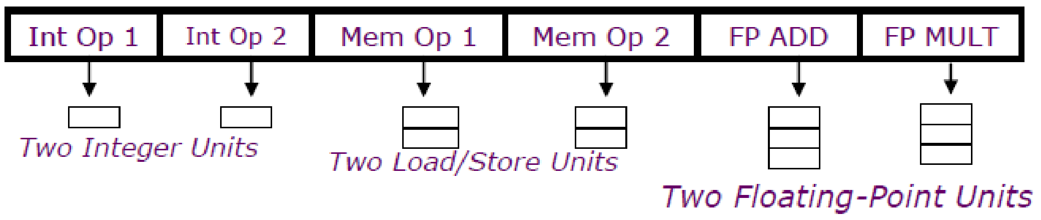
The program we will use for this problem is listed below (In all questions, you should assume that arrays A, B and C do not overlap in memory).

```
              C code
for (i=80; i>0; i--) {
    A[i] = A[i] * B[i];
    C[i] = C[i] + A[i];
}
```

In this problem, we will deal with the code sample on a VLIW machine. Because the software keeps track of the dependences among instructions and schedules them in the long instruction words, our VLIW machine does not use interlocks for hazard prevention. The machine does not use forwarding, and the result of an operation is written to the register file immediately after it has gone through the corresponding execution functional unit. Our machine has six execution units:

- 2 integer ALU units, also used for branch operations. The latency between an integer ALU operation and any other operation is 1 cycle.
- 2 memory units, fully pipelined. Each unit can perform either a store or a load. The latency between a memory operation and any other operation is 2 cycles.
- 2 FP units, fully pipelined. One unit can perform **`fadd`** operations, the other **`fmul`** operations. The latency between an FP ALU operation and any other operation is 3 cycles.

As an example, if an integer ALU operation enters the integer ALU unit in cycle 1, another operation using the integer ALU operation's result can enter its execution functional unit in cycle 3 and proceed in the pipeline without stalls.
Below is a diagram of our VLIW machine:



The loop in the program can be translated to the following operations. Suppose r1, r2 and r3 initially contain the addresses of A[80], B[80], and C[80], and r4 initially contains the number 80.

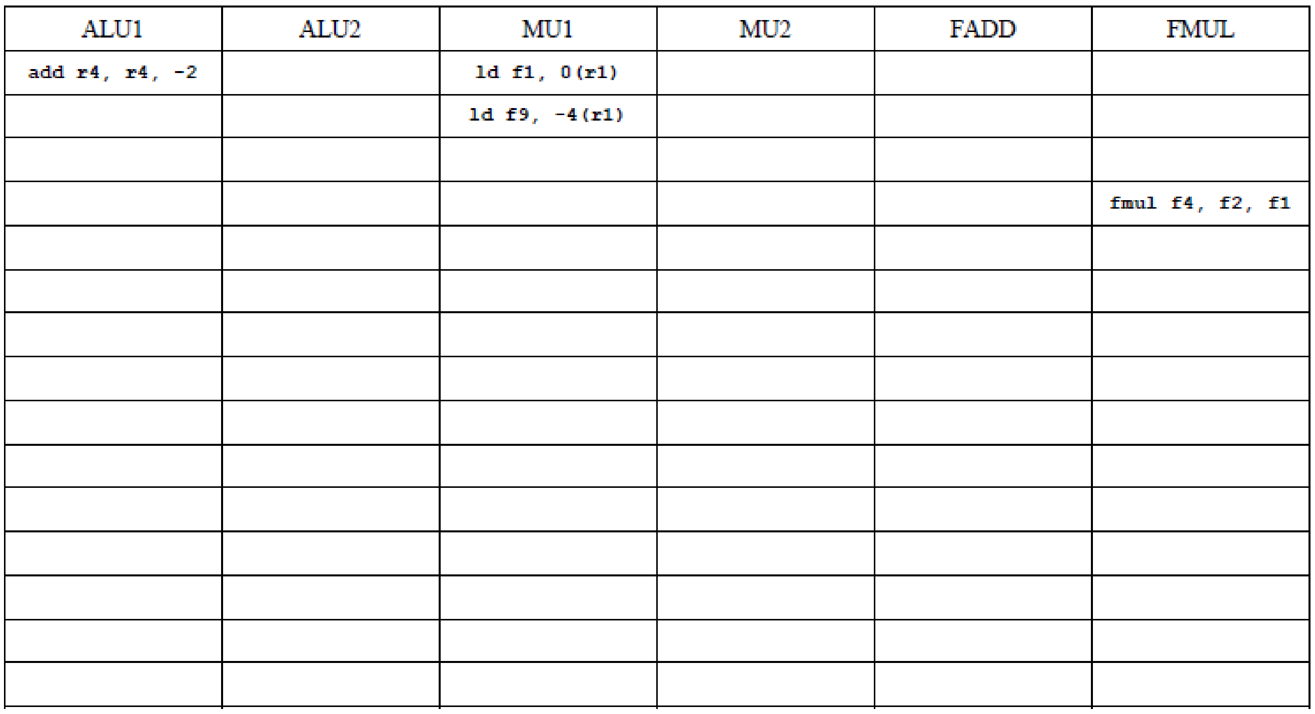
```
loop:    1.  ld f1, 0(r1)         ; f1 = A[i]
         2.  ld f2, 0(r2)         ; f2 = B[i]
         3.  fmul f4, f2, f1      ; f4 = f1 * f2
         4.  st f4, 0(r1)         ; A[i] = f4
         5.  ld f3, 0(r3)         ; f3 = C[i]
         6.  fadd f5, f4, f3      ; f5 = f4 + f3
         7.  st f5, 0(r3)         ; C[i] = f5
         8.  add r1, r1, -4
         9.  add r2, r2, -4
        10.  add r3, r3, -4
        11.  add r4, r4, -1       ; i--
        12.  bnez r4, loop        ; loop
```

a) Table 6.a shows our program rewritten for our VLIW machine, with some operations missing (operations **2, 6** and **7**). We have adjusted and rearranged the operations to let them execute as soon as they possibly can, while ensuring program correctness. Assume the VLIW machine's processor reads operands for the operations before issuing them into the execution functional unit. Hence, combining operations such as 1 and 8 in one instruction word is possible. Please fill in missing operations in Table 6.a. (Note, you may not need all the rows).

ALU1	ALU2	MU1	MU2	FADD	FMUL
`add r1, r1, -4`	`add r2, r2, -4`	`ld f1, 0(r1)`			
`add r3, r3, -4`	`add r4, r4, -1`	`ld f3, 0(r3)`			
					`fmul f4, f2, f1`
			`st f4, 4(r1)`		
	`bnez r4, loop`				

Table 6.a: VLIW Program

b) Assume the VLIW processor has 32 FP registers (f0 – f31). If we unrolled the loop once (combine two iterations to be one larger iteration), would that give us better performance? How shall we (or the compiler) write the VLIW instructions to accomplish the best performance? Write the instructions in Table 6.b.



ALU1	ALU2	MU1	MU2	FADD	FMUL
add r4, r4, -2		ld f1, 0(r1)			
		ld f9, -4(r1)			
					fmul f4, f2, f1

Table 6.b: VLIW Program

Q5. Simple Cache

a) Here is a string of address references given as word addresses: 1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17. Assume a direct mapped cache with 16 one-word blocks that is initially empty, label each reference in the list as a hit or miss and show the final contents of the cache.

b) Using the same reference string, show the hits and misses and final cache contents for a direct mapped cache with four-word blocks and a total size of 16 words.

c) Using the same reference string, show the hits and misses and final cache contents for a two-way set associative cache with one-word blocks and a total size of 16 words. Assume LRU replacement.

d) Using the same reference string, show the hits and misses and final cache contents for a fully associative cache with one-word blocks and a total size of 16 words. Assume LRU replacement.

Q6. SimpleScalar Assignment

In this problem, we will begin to use the “sim-bpred” simulator that allows the user to explore various branch prediction algorithms via functional simulation. This simulator can study basic branch predictors like the ones that we studied in class: “predict nottaken”, “predict taken”, “bimodal” (2-bit counters), “two-level”, or a combination of “bimodal” and “two-level.”

Problem Statement:
This problem involves a design space exploration study. Recall that 2-bit saturating counters are preferable to 1-bit counters because they provide some hysteresis for the branch decisions. However, using 2-bit counters requires more space thus leaving less room for entries in the predictor. In this problem, we want to explore varying the number of bits in these counters (1-bit or 2-bits) vs. the number of entries in the branch predictor. The point of this assignment is to determine whether the extra storage space is better spent on more entries or more and if there is a “crossover point” where one option becomes more efficient than the other. This will require some minor changes to the bpred.c code to support the 1-bit counter option. The 2-bit counter is the default so this will run “out-of-the-box”. The changes are pretty minimal after you find where to make them.

Choose three of the benchmarks to run with sim-bpred and then simulate with various values of “-bpred”. You may want to write a script to automate the simulation process. Make sure that you are correctly running the benchmarks. Try at least the bimodal predictor and at least two2-level predictors (say, GAp and PAg).

What to report?
The above will require a fair amount of simulations, with about 1-2 minutes required per simulation. You will have three benchmarks, three base predictor configurations (bimodal and two 2-level predictors), each with the 1-bit vs. 2-bit counter simulations. Then you will have to perform enough sizing analysis to determine what decision makes the most sense.

After you have finished debugging, you may want to “make clean” and then change the compiler optimization level to “-O3” to help speed up your simulations. Generate charts that graphically depict which configuration gives the best predictor accuracy for the least amount of hardware and comment on your results. Clearly state which predictors you used (you may want to draw the diagrams) and how many bits are used in the various places.