CS146: Computer Architecture
Fall 2019
Homework #2
Due October 2, 2019 (Wednesday) Evening

---

1. **Pipelining**

Consider the code segment below. Assume that full bypassing/forwarding has been implemented. Assume that the initial value of register R23 is much bigger than the initial value of register R20. Assume that all memory references take a single cycle. **Assume that both load-use hazards and branch delay slots are hidden using delay slots.** You may- not reorder instructions to fill such slots, but if a subsequent instruction is independent and is properly positioned, you may assume that it fills the slot. Otherwise, fill slots with additional no-ops as needed.

```
LOOP:   lw    R10, X(R20)
        lw    R11, Y(R20)
        subu  R10, R10, R11
        sw    Z(R20), R10
        addiu R20, R20, 4
        subu  R5, R23, R20
        bnez  R5, LOOP
        nop                  ; 1 delay slot
```

a. (5 points) Draw a pipeline diagram of 2 iterations of its execution on a standard 5-stage MIPS pipeline (for clarity, use graph paper or a computer). Assume that the branch is resolved using an ID control point. In the box below, write the total number of cycles required to complete 2 iterations of the loop.

Cycles =	

b. (15 points) On the second grid page that follows, draw a pipeline diagram of 2 iterations of the loop on the pipeline below. Note that the loop has a single branch delay slot nop included – you may need to add more. You cannot assume anything about the program's register usage before or after this code segment. Fill in the boxes below.

Pipeline Branch Delay =	
Pipeline Load Delay =	
Cycles =	

Pipeline:

IF1	IF2	ID	RF	EX1	EX2	M1	M2	WB					
	IF1	IF2	ID	RF	EX1	EX2	M1	M2	WB				
		IF1	IF2	ID	RF	EX1	EX2	M1	M2	WB			
			IF1	IF2	ID	RF	EX1	EX2	M1	M2	WB		
				IF1	IF2	ID	RF	EX1	EX2	M1	M2	WB	
					IF1	IF2	ID	RF	EX1	EX2	M1	M2	WB

IF1: Begin Instruction Fetch
IF2: Complete Instruction Fetch
ID: Instruction Decode
RF: Register Fetch
EX1: ALU operation execution begins. Branch target calculation finishes. Memory address calculation. Branch condition resolution calculation begins.
EX2: Branch condition resolution finishes. Finish ALU ops. (But branch and memory address calculations finish in a single cycle).
M1: First part of memory access, TLB access.
M2: Second part of memory access, Data sent to memory for stores OR returned from memory for loads.
WB: Write back results to register file

2. **Scoreboarding**

For the code sequence shown below, draw a pipeline diagram of how instructions would issue in a machine using scoreboarding as discussed in class. Use the execution mix and scoreboard structure as given in the class example. Assume that the FP Add unit has 4 EX phases, the FP Multiply unit has 7 EX phases, and divide has 24 EX phases. FP Adds, Subtracts, and Multiplies are fully-pipelined, while divide operations are NOT pipelined.

```
LD    F6, 12(R2)
LD    F2, 16(R3)
ADDD  F0, F2, F4
DIVD  F10, F0, F6
SUBD  F8, F6, F2
ADDI  R2, R2, 8
ADDI  R3, R3, 16
ADDD  F6, F8, F2
```

3. **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 hardware configuration from Figure A.51, for the scoreboard, and Figure 3.2 for Tomasulo's scheme. 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

4. **SimpleScalar Problem**

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."

Here is the list of SPEC2000 benchmarks that we will run for this assignment. Pick at least three to report your results for.

Benchmark	Instructions	Sim-bpred Time (sec)
perlbmk	205927571	80
ammp	45838888	21
gcc	96761516	39
mcf	187806891	91
parser	269103374	108

a) How to run the benchmarks?

I have made a few scripts that will help you with this assignment. These are ~cs146/run_scripts/ on the FAS system. There are two scripts that you will fine here – "run_spec2k.pl" and "extract_results.pl". These two perl scripts have all of the information that you will need to run the benchmarks. You will just need to setup your account and configure the script for this assignment.

Steps
1) Go to your home directory and make two directories "run_scripts" and "results" (ie. mkdir run_scripts). This can be in the root of your home directory or somewhere else. For example you may have a subdirectory cs146 where you will put all of your cs146 coursework. You should also have your "simplesim-3.0" directory in here.
2) Copy the scripts from ~cs146/run_scripts to this directory.
3) First edit "run_spec2k.pl". There are a few things to note in this script. First, the third definition of @BENCH_LIST lists the benchmarks that the script will run. Edit this to add or subtract benchmarks that you would like to run. Now skip all the way to near the end of the script where it says $HOME_DIR = "…". Change this directory to where you did "mkdir run_scripts" above. You can also change the binary that you are simulating with by changing the $SIM_BIN variable – in the base script it is set for "sim-fast" – you will want to change this to "sim-bpred" for this assignment. Now you can just execute "./run_spec2k.pl" to run the simulations for this assignment. All of the results will be stored in your "results" directory with the subdirectory of the benchmark names.
4) Make similar changes to the "extract_spec2k.pl" file. This file will help you automatically extract the result data from your results directory. Change "@STAT_LIST" to print out additional stats (for example the branch predictor hit rates, etc). Just enter the new simplescalar stat name into the list.

b) 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" (this can be changed by adding parameters to the $CONFIG variable in the script – better yet use a foreach statement to cycle through all of your combinations – make sure that you generate different result files as well -for each configuration). Use the Table on the previous page to make sure that you are correctly running the benchmarks. Try at least the bimodal predictor and at least two 2-level predictors (say, GAp and PAg).

c) 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), and 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.

Even though each simulation only takes a couple of minutes, you will obviously want to automate this process with provided scripts. Let me or Kim know if you need any help with the perl scripts – again, the changes required should be minimal.

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.