**CS161 – Fall 2016**
**Homework Assignment 3**

**Due: Friday, November 18th @ 11:59pm**
**This assignment may be completed in pairs. Please Indicate both names when submitting the assignment and upload in PDF format to iLearn**

**1.** Consider the following loop.

```
loop: SUBI  R1, R1, #1
      LW    R3,0(R2)
      LW    R4,4(R2)
      MUL   R5, R3, R4
      ADD   R3, R5, R6
      ADDI  R2, R2, #8
      BNE   R1, R0,loop
      ADD   R10, R11, R12
```

a) Identify all data dependencies (potential data hazards) in the given code snippet. Assume the loop takes exactly one iteration to complete. Specify if the data dependence is RAW, WAW or WAR.

b) Assume a 5-stage pipeline (IF ID EX MEM WB) without any forwarding or bypassing hardware, but with support for a register read and write in the same cycle. Also assume that branches are resolved in the ID stage and handled by stalling the pipeline. All stages take 1 cycle. Again, the loop takes one iteration to complete. Which dependencies from part (a) cause stalls? How many cycles does the loop take to execute?

**2.** Consider the following loop.

```
loop: LW    R1,0(R1)
      AND   R1,R1,R2
      LW    R1,0(R1)
      LW    R1,0(R1)
      BEQ   R1,R0,loop
```

Assume that perfect branch prediction is used (no stalls due to control hazards), that there are no delay slots, and that the pipeline has full forwarding support. Also assume that many iterations of this loop are executed before the loop exits.

a) Show a pipeline execution diagram for the third iteration of this loop, from the cycle in which we fetch the first instruction of that iteration up to (but not including) the cycle in which we can fetch the first instruction of the next iteration. Show all instructions that are in the pipeline during these cycles (not just those from the third iteration).

b) How often (as a percentage of all cycles) do we have a cycle in which all five-pipeline stages are doing useful work?

**3.** This exercise is intended to help you understand the relationship between delay slots, control hazards, and branch execution in a pipelined processor. In this exercise, we assume that the following MIPS code is executed on a pipelined processor with **a 5-stage pipeline**, **full forwarding**, and a **predict-taken branch predictor**:

```
        LW    R2,0(R1)
label1: BEQ   R2,R0,label2   # not taken once, then taken
        LW    R3,0(R2)
        BEQ   R3,R0,label1   # taken
        ADD   R1,R3,R1
label2: SW    R1,0(R2)
```

a) Draw the pipeline execution diagram for this code, assuming there are no delay slots and that branches execute in the EX stage.

b) Repeat (a), but assume that delay slots are used. In the given code, the instruction that follows the branch is now the delay slot instruction for that branch.

c) One way to move the branch resolution one stage earlier (ID stage) is to not need an ALU operation in conditional branches. The branch instructions would be **"bez rd,label"** and **"bnez rd,label"**, and it would branch if the register has and does not have a zero value, respectively. Change this code to use these branch instructions instead of beq. You can assume that register R8 is available for you to use as a temporary register, and that an seq (set if equal) R-type instruction can be used.

Section 4.8 describes how the severity of control hazards can be reduced by moving branch execution into the ID stage. This approach involves a dedicated comparator in the ID stage, as shown in Figure 4.62 (final page). However, this approach potentially adds to the latency of the ID stage, and requires additional forwarding logic and hazard detection.

d) Using the first branch instruction in the given code as an example, describe the hazard detection logic needed to support branch execution in the ID stage as in Figure 4.62 (final page). Which type of hazard is this new logic supposed to detect?

e) For the given code, what is the speedup achieved by moving branch execution into the ID stage? Explain your answer. In your speedup calculation, assume that the additional comparison in the ID stage does not affect clock cycle time and that delay slots are not used.

f) Using the first branch instruction in the given code as an example, describe the forwarding support that must be added to support branch execution in the ID stage. Compare the complexity of this new forwarding unit to the complexity of the existing forwarding unit in Figure 4.62 (final page).

**4.** The importance of having a good branch predictor depends on how often conditional branches are executed. Together with branch predictor accuracy, this will determine how much time is spent stalling due to mispredicted branches. In this exercise, assume that the breakdown of dynamic instructions into various instruction categories is as follows:

| R-Type | BEQ | J | LW | SW |
|---|---|---|---|---|
| 40% | 25% | 5% | 25% | 5% |

Also, assume the following branch predictor accuracies:

| Always-Taken | Always-Not-Taken | 2-Bit |
|---|---|---|
| 45% | 55% | 85% |

a) Stall cycles due to mispredicted branches increase the CPI. What is the extra CPI due to mispredicted branches with the always-taken predictor? Assume that branch outcomes are determined in the EX stage, that there are no data hazards, and that no delay slots are used.

b) Repeat (a) for the "always-not-taken" predictor.

c) Repeat (a) for for the 2-bit predictor.

d) With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaces a branch instruction with an ALU instruction? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

e) With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaced each branch instruction with two ALU instructions? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

f) Some branch instructions are much more predictable than others. If we know that 80% of all executed branch instructions are easy-to-predict loop-back branches that are always predicted correctly, what is the accuracy of the 2-bit predictor on the remaining 20% of the branch instructions?

**5.** This exercise examines the accuracy of various branch predictors for the following repeating pattern (e.g., in a loop) of branch outcomes: T, NT, T, T, NT

a) What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

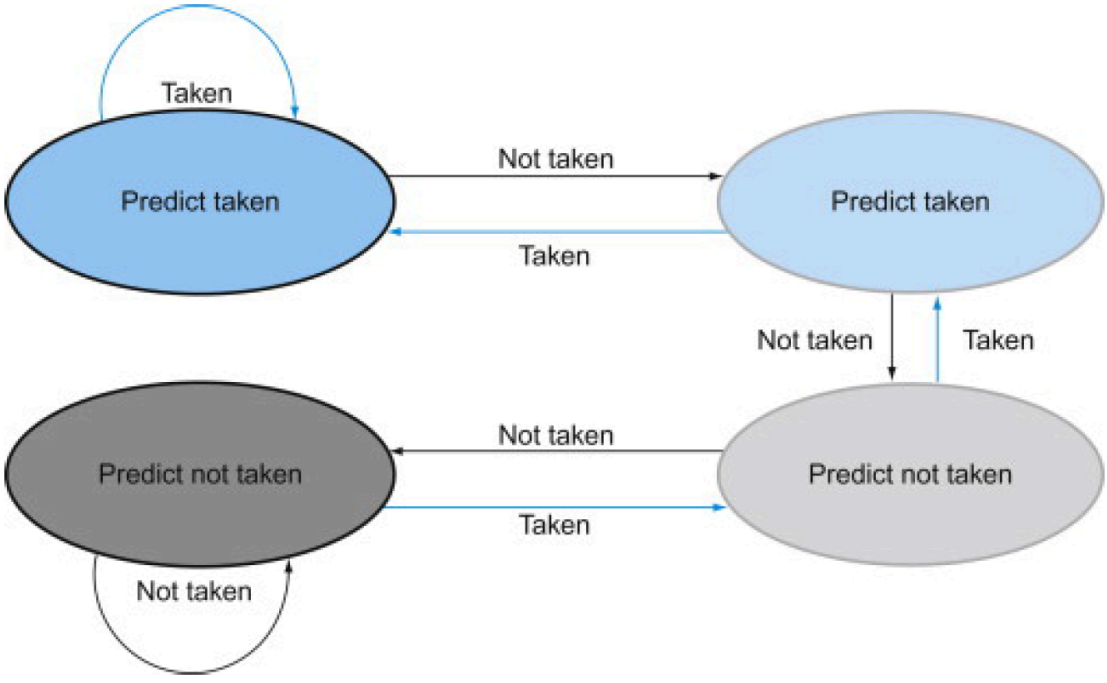
b) What is the accuracy of the two-bit predictor for the first 4 branches in this pattern, assuming that the predictor starts off in the bottom left state from the figure below (predict not taken)?

c) What is the accuracy of the two-bit predictor if this pattern is repeated forever?

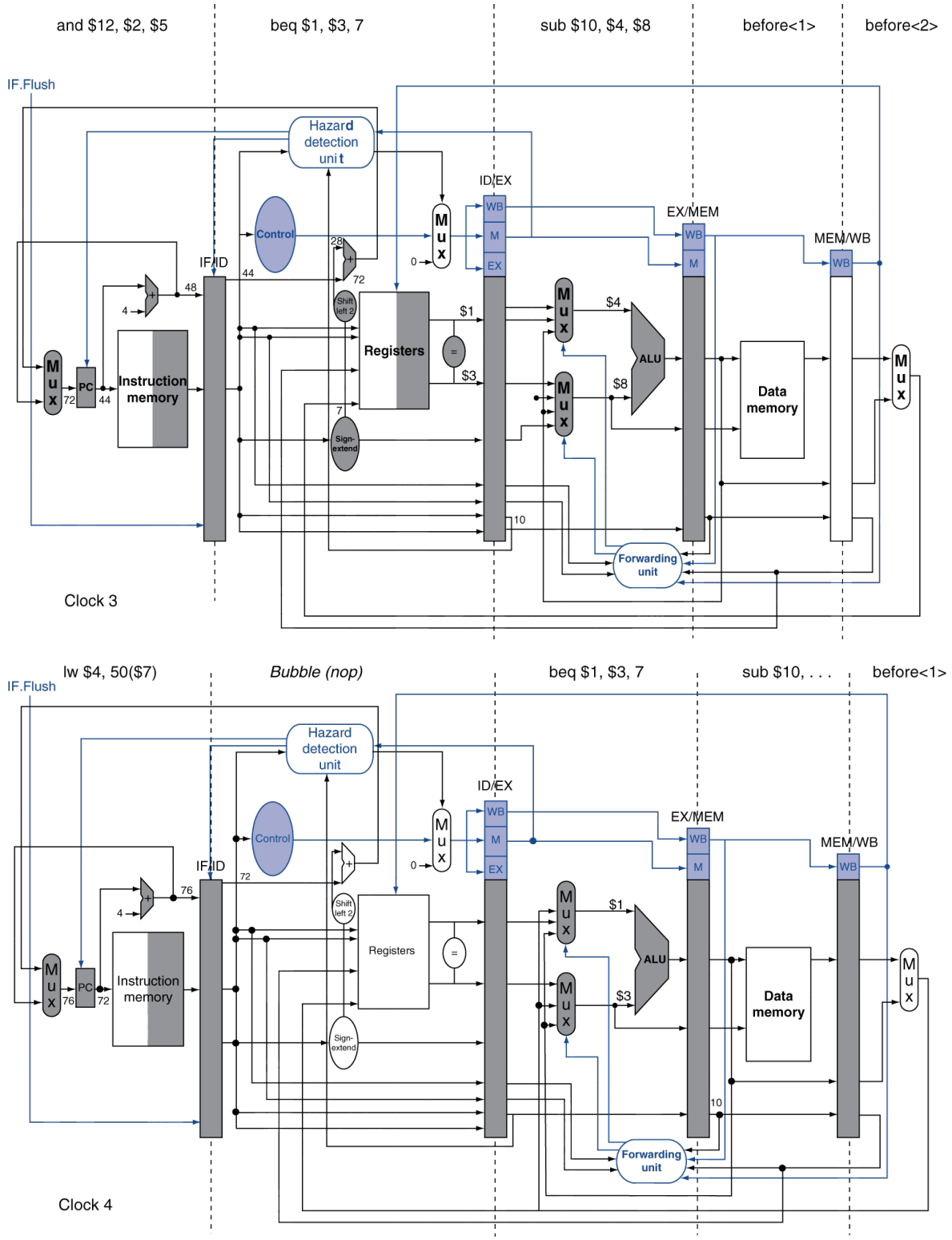
d) Design a predictor that would achieve a perfect accuracy if this pattern is repeated forever. You predictor should be a sequential circuit with one output that provides a prediction (1 for taken, 0 for not taken) and no inputs other than the clock and the control signal that indicates that the instruction is a conditional branch.

e) What is the accuracy of your predictor from (d) if it is given a repeating pattern that is the exact opposite of this one?

f) Repeat (d), but now your predictor should be able to eventually (after a warm-up period during which it can make wrong predictions) start perfectly predicting both this pattern and its opposite. Your predictor should have an input that tells it what the real outcome was. Hint: this input lets your predictor determine which of the two repeating patterns it is given.



# Figure 4.62


and $12, $2, $5
beq $1, $3, 7
sub $10, $4, $8
before<1>
before<2>
IF.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
WB
M
EX
IF/ID
Shift left 2
Registers
ALU
Data memory
Instruction memory
PC
Sign-extend
Forwarding unit
Clock 3
lw $4, 50($7)
Bubble (nop)
beq $1, $3, 7
sub $10, . . .
before<1>
Clock 4