Only Problem Set Two will be graded. Turn in only Problem Set Two on November 2, 2016 (Wednesday) by 5:00 pm.
Submit homework via gradescope <gradescope.com> as a PDF. Other formats may or may not be viewed and evaluated accurately.

Problem Set One

1. Hennessy & Patterson (5th Ed.) C.13 (Scoreboards)
2. Hennessy & Patterson (5th Ed) 3.1
3. Hennessy & Patterson (5th Ed) 3.3
4. Hennessy & Patterson (5th Ed) 3.5
5. Hennessy & Patterson (5th Ed) 3.7
6. Hennessy & Patterson (5th Ed) 3.10
7. Hennessy & Patterson (5th Ed) 3.17
8. Hennessy & Patterson (5th Ed) 3.18
Problem Set Two

1. Structural Hazards in Floating-Point Pipeline

The figure above shows a floating point pipeline that can execute three types of floating point instructions, *add*, *mult*, and *div*. Addition takes 4 cycles to finish, while multiply takes 7 and divide takes 11. More interestingly, during execution all three instructions need to use two pieces of common hardware, namely, *X* and *Y*, thus giving rise to possible structural hazards. The execution stages during which these two common hardware components are utilized for each instruction is given below, with “*x*” denoting the use of component *X*, “*y*” the use of component *Y*, and “0” no use of either *X* or *Y* in that stage.

Add: 0x0y
Mult: x0y0y0x
Div: xy00xy00x0y

In this pipeline design, the conflict resolution for the possible structural hazard is performed once and for all at the ID stage. In other words, if the instruction at the ID stage conflicts with an instruction in execution, it is blocked until the cycle at which it can be issued with no conflict. Once the execution starts, the instruction does not stall.

(Part A) A technique was outlined in your book for dealing with the structural hazards of register write in the MIPS floating point pipeline. Please first briefly describe that technique, then furthermore generalize it so as to resolve the aforementioned resource conflicts. More specifically, you need to address the type and the number of hardware component(s) that need to be employed, as well as the approach for handling structural hazards using such component(s).
(Part B) The instructions listed in the following cases are assumed to be data independent. For each case, please identify the minimum latency (in terms of clock cycles) induced by the aforementioned structural hazards between any two adjacent instructions.

1. *add* followed by *multiply*

2. *multiply* followed by *divide*

3. *add* followed by *divide*

4. *divide* followed by *add*

5. *multiply* followed by *add* followed by *multiply*

6. *divide* followed by *add* followed by *multiply*
(Part C) Noticing that structural hazards have caused a large number of stalls between adjacent floating-point instructions, one engineer in the design team has proposed to insert a no-op stage (which obviously uses neither the X nor the Y component), during the execution of mult so as to reduce the impact of structural hazards. The engineer, however, is not quite confident of being able to persuade the manager to adopt this idea, since this no-op will increase the execution time of multiply instructions by 1 cycle. Please help the engineer to evaluate her idea, by determining whether inserting a no-op is helpful in reducing the number of stalls caused by structural hazard. If you think it is not helpful, please state the reasons; if you think it is helpful, please show the position where the no-op should be inserted, and furthermore show instruction sequences that can benefit from this no-op.
2. Advanced Scoreboarding

One of the mechanisms to take advantage of ILP, the scoreboard, does not have a mechanism to deal with WAR and WAW hazards effectively. When an instruction completes execution, if it would cause an anti-dependence, it is stalled from writing until the previous instruction reads its operands. If precise exceptions are required, then the previous instruction must finish execution. Compared to Tomasulo’s algorithm and register renaming schemes, the scoreboard is much worse because it stalls on all anti-dependencies.

At Chips Ahoy Inc., you design a custom Scoreboard-based processor, Chewy, with a very simple co-processor that does your exception handling. Since the co-processor does exception handling, you can freeze and resume work on the main processor’s functional units; nonetheless, you don’t want to lose precise exception capabilities. Of course, implementing strict precise exceptions with a scoreboard is tricky since it would require in-order commit, so you are content with simulating precise exception semantics instead.

Your simulations show that the programs that Chewy is designed to execute are filled with WAR & WAW dependencies, and you need Chewy to run faster. You don’t want to add the complexities of register renaming or in-order commit, so you ask a consultant, Alice, what to do.

Alice proposes that adding a certain number of shared result registers will solve the problem. Your boss is skeptical that it can help.

One of the issues that Alice’s proposal raises is the de-allocation strategy for these result registers. Some of your employees want to add reference counters to each of the result registers, but the CEO, being very stingy, does not want to add such complicated hardware. “If the result registers alone can’t entirely solve the problem, we will bear any remaining performance hit,” he says.

(Part A) What is the rule the original scoreboard (the one discussed in class and described in the textbook) follows to prevent WAR hazards? Use the following code to explain your answer. (You may assume timing information as discussed in your textbook, i.e. long divisions, short additions, and medium-length multiplications.)

```
div F2, F6, F8
add F4, F2, F0
add F0, F6, F8
mul F10, F0, F0
```
(Part B) What do you think Alice’s solution is? How can the result registers be used to allow the scoreboard to proceed faster on WAR dependencies? Use the code from Part A to explain your answer. Address the following aspects:

- Where does a dependent instruction obtain its input from in this “improved” architecture?
- Do the result registers ever get written to the architectural registers? If not, why not? If they do, do the architectural registers get updated with the same frequency as in the original scoreboard or less or more frequently, or never?

(Part C) To handle WAW dependencies, the standard scoreboard prevents an instruction from issuing until the previous instruction has written its result. Can the result registers be used to prevent the scoreboard from stalling on WAW dependencies? If not, describe an instruction sequence and the scoreboard state (you do not need to write out all 3 tables, just describe the pertinent values) that must still stall. If it can, describe how the result registers can be used to prevent stalls on WAW dependencies with a WAW code sample.
3. Application-Specific Tomasulo’s Processor

A custom processor company is utilizing Tomasulo’s processor with register renaming as a backbone for its custom solutions. Given a particular application, as defined by its program, the company customizes the Tomasulo processor by analyzing the program and incorporating just the right number of functional units, reservation stations, and physical registers in order to enable it to operate in full speed at steady state while ensuring no resource is wasted by lying idle in steady state. The execution latencies for integer units, memory units, FP adders, and FP multipliers are 1, 2, 2 and 4, respectively. Assume that an unlimited number of simultaneous memory accesses are allowed.

Consider the following three loops. Each one of them reads values, manipulates values, and subsequently stores them into various location(s). For each of the loops below, first, identify the instruction schedules at steady state which fully utilize the given reservation stations, and also identify the minimum number of functional units, reservation stations and physical FP registers for these cases. Assume that no branch mis-predictions, no exceptions and no cache misses arise while each loop is executing.

(A)
LOOP:

L.D F0, 0(R1)
ADD.D F1, F0, F0
S.D F1, 100(R1)
DADDIU R1, R1, -8
BNZ R1, LOOP; branches if R1 is not zero
(B)
LOOP:
L.D    F0, 0(R1)
L.D    F1, 100(R1)
ADD.D  F2, F0, F1
S.D    F2, 0(R2)
ADD.D  F3, F2, 50
S.D    F3, 0(R3)
DADDIU R1, R1, -8
DADDIU R2, R2, 8
DADDIU R3, R3, 16
BNZ    R1, LOOP; branches if R1 is not zero

(C)
LOOP:
L.D    F0, 0(R1)
L.D    F1, 100(R1)
MULT.D F2, F0, F1
S.D    F2, 0(R2)
S.D    F1, 0(R3)
DADDIU R1, R1, -8
DADDIU R2, R2, 8
DADDIU R3, R3, 16
BNZ    R1, LOOP; branches if R1 is not zero
4. Compiling Branches

You’re writing smart compilers to optimize for branch prediction. Ignore branches which may be used to construct the for loop. Do not worry about them neither in terms of predictor accuracy accounting nor in terms of branch history construction. Your code looks something like:

```python
for i=0 to 9999:
    if (i%4 == 0 || i%4 == 3) { // Branch 1. Word Address 0x1000
        Block A = 9 instructions
    }
    if (i%4 == 1 || i%4 == 2) { // Branch 2. Word Address 0x100A
        Block B = 11 instructions
    }
    if (i%4 < 2) { // Branch 3. Word Address 0x1016
        Block C = 10 instructions
    }
```

This means that the access pattern for Branch 1 is TNNT, Branch 2 is NTTN, and Branch 3 is TTNN. Note that the branches and their associated blocks are independent of each other. You need to optimize this code for two different branch prediction schemes:

**Local Predictor**: This uses the 2-bit LSB of the branch word address to index into a table of 2-bit saturating counter predictors that start Strongly Taken.

**Global Predictor**: This predictor uses a 2-bit history to index into a table of 2-bit saturating counter predictors that start Strongly Taken. Note that no address bits are used! The history starts NN (not taken, not taken).

*(Part A)* Given the block of code shown above as is, please identify the prediction accuracy of the local predictor overall as well as the prediction accuracy of each of the 3 branches.
(Part B) Please identify the prediction accuracy of the global predictor overall as well as the prediction accuracy of each of the 3 branches.

(Part C) Which predictor accuracies are affected by re-arranging the branches? For each affected predictor, what is the optimal order for the 3 branches, and what is the improved accuracy?

(Part D) Instead of re-arranging, you want to explore which predictor accuracies are affected by inserting NOOPs. For each affected predictor, what is the optimal placement of NOOPs, and what is the improved accuracy?