CS-UY 2214 — Recitation 9

Introduction

Complete the following exercises. Unless otherwise specified, put your answers in a plain text file named `recitation9.txt`. Number your solution to each question. When you finish, submit your file on Gradescope. Then, in order to receive credit, you must ask your TA to check your work. Your work should be completed and checked during the recitation session.

You may consult the E20 manual, which is available on Brightspace.

Problems

1. Consider a particular E20 program that, when run, executes instructions in the following amounts before halting:

80%	register instructions, executing in 4 clock cycles each
15%	register instructions with stalls, executing in 10 clock cycles each
5%	memory instructions with stalls, executing in 30 clock cycles each

   What is the overall average CPI of the program?

2. Recall that a *conflict* between instructions results when instructions share a resource (a register, a memory cell, or a functional unit) that may lead to a hazard. A *hazard* is a condition that requires a stall or other mitigating step in order to allow successful pipelined execution of a program.

   Consider the following excerpts of E20 programs. For each excerpt, identify all conflicts among the instructions, including those potentially causing the following hazards:

   - Data conflicts. Identify the type of data conflict (read-after-write, write-after-write, and write-after-read). Specify which instructions (by line number) have the conflict, and specify which register or memory address conflicts. Your answer should include all data conflicts, regardless of whether the conflict would lead to a hazard in practice (which depends on the design of the pipeline).
   - Control hazards. Specify which instruction (by line number) has the hazard.
   - Structural hazards. These don't occur on the E20, so don't worry about them for this exercise.

   If a code excerpt has no conflicts, say so.

   (a)
```
1 movi $1, 5
2 add $2, $1, $1
3 add $3, $1, $1
```

   (b)
```
1 movi $1, 5
2 addi $1, $1, 20
```

(c)

```
1 sw $1, 0($4)
2 movi $5, 40
3 sw $2, 0($4)
```

(d)

```
1 movi $1, 50
2 jr $1
```

(e)

```
1 add $1, $2, $2
2 add $5, $3, $3
3 add $4, $2, $3
```

(f)

```
1 movi $1, 40
2 sw $1, 0($5)
3 movi $1, 50
4 sw $1, 1($5)
```

3. In some cases, it may be possible to reduce stalls in a program by changing the order of instructions. This *re-ordering* may be performed manually by an assembly-language programmer; or automatically by a compiler; or automatically by the processor (in which case it is called *out-of-order execution*). Any re-ordering must preserve the meaning of the original program, i.e. the same result must be achieved in all cases.

   How can we know if a reordering will preserve the meaning? It is usually not safe to reorder two instructions that conflict, i.e. if there is a hazard between them. For example, consider the following excerpt:

```
1     movi $1, 2
2     add $2, $1, $1
3     add $3, $1, $1
```

   There is a read-after-write conflict between instruction 1 and 2; and also a read-after-write conflict between 1 and 3. There is no conflict between 2 and 3. Therefore, we conclude that although we can safely change the order that 2 and 3 execute in, we cannot change the order that 1 and 2 execute in, nor the order that 1 and 3 execute in.

   So, a valid reordering of this program is is $1, 3, 2$. On the other hand, $2, 1, 3$ (due to conflict between 1 and 2) and $3, 1, 2$ (due to conflict betwen 1 and 3) are not valid reorderings.

   Your task is to give all valid re-orderings of each of the code excerpts from the previous question, parts 2a through 2f. Give your answer as a sequence of instruction numbers, as above. If there are multiple valid reorderings, give them all. If there are no valid reorderings, say so.

4. Consider the following E20 assembly program:

```
        lw $1, data($0)      #ram[0]
        jeq $1, $0, skip     #ram[1]
        addi $3, $0, 0       #ram[2]
        addi $2, $1, 5       #ram[3]
        sub $3, $1, $2       #ram[4]
    skip:
        halt                 #ram[5]
    data:
        .fill 10             #ram[6]
```

Assume that the program runs on a 5-stage pipelined E20 processor, as we discussed in class. Assume no forwarding.

(a) Identify all instructions that have potential data and control hazards. For each instruction, indicate the type of hazard; the instruction's memory location; and why it has a hazard.

(b) Rewrite the program so as to eliminate data hazards. Do not reorder the instructions. Instead, insert `nop` instructions appropriately. Keep in mind that you will need to insert enough `nop`s so that the preceding instruction can completely write its data into the register file. It may be helpful to draw a pipeline timing table. Do not address control hazards.

5. At this point, you should have started working on your E20 simulator. In the E20 simulator assignment, we list several "edge cases": situations that require special handling in your code. For example, your code needs to handle the "wrap around" effect caused by register overflow.

   Select another student to work with. Sit with your partner and follow this procedure together:

   (a) Select one of the edge cases listed in the Hints section of the E20 simulator assignment.

   (b) Develop E20 code that will exercise the edge case. For example, if you want to test the handling of register overflow, write a short E20 program that will cause a register to overflow. If you aren't sure how to do this, discuss it with your partner or ask a TA.

   (c) Share your edge case test code with your partner. They will run your test code with their simulator. Determine if the test works, i.e. if their simulator achieves the correct result.

   (d) Submit the edge case you developed for your partner and describe if it helped them.

   (e) Repeat the process, but now your partner develops test code for your simulator.

   Note: *Do not* share the code for your simulator. Share only the E20 code for the test.

   In accordance with the class's policy on academic integrity, please indicate in your project's README file that you developed test cases with your partner.