CSCE 692 – Appendix C Homework Problems

2nd Lt David Crow – 24 January 2019

(100 points)

Due: NLT 1100 Thursday, 31 January 2019

**Problems:**

**C.1 (a, b, c, d)** [10/20/20/20]

**C.3 (a, b, c, d)** [7/7/7/9]

**Instructions:**

* Put your name at the top, and number each page (last name-pg)
* State the complete problem
* Show your work
* Clearly indicate your answer

**Notes**:

* C.1 (a)
  + Not all data dependences result in hazards. Find ALL data dependences.
  + Do not list “transitive” dependences (those that cross instructions).
* C.1 (b, c, d)
  + Assume branch target is known in ID, but that branch outcome is not decided until the EX stage.
  + When computing how many cycles the loop requires, it is asking for all iterations of the loop, beginning at the first instruction, and continuing until the next valid instruction following the loop is able to begin.
  + When you have to draw a multi-cycle diagram, consider using or printing out copies of the blank template (pipeline\_matrix.xlsx, which is on CANVAS and printed below).

C.1 [10/20/2/20/] <C.2> Using the following code fragment, answer the following questions:

Loop: ld x1,0(x2) ; load x1 from address 0+x2

addi x1,x1,1 ; x1=x1+1

sd x1,0(x2) ; store x1 at address 0+x2

addi x2,x2,4 ; x2=x2+4

sub x4,x3,x2 ; x4=x3-x2

bnez x4,Loop ; branch to Loop if x4!=0

Assume that the initial value of x3 is x2+396.

1. [10] <C.2> Data hazards are caused by data dependences in the code. Whether a dependency causes a hazard depends on the machine implementation (i.e., number of pipeline stages). List all of the data dependences in the code above. Record the register, source instruction, and destination instruction; for example, there is a data dependency for register x1 from the ld to the addi.

|  |  |  |  |
| --- | --- | --- | --- |
| Dependency | Register | Source instruction | Destination instruction |
| 1 | x1 | ld | addi |
| 2 | x1 | addi | sd |
| 3 | x2 | ld | addi |
| 4 | x2 | sd | addi |
| 5 | x2 | addi | sub |
| 6 | x4 | sub | bnez |

In plain English,

1. We must finish loading into x1 before we can add to x1.
2. We must finish adding to x1 before we can store x1 elsewhere.
3. We must finish loading x2’s value elsewhere before we add to x2.
4. We must finish storing data in x2 before we can add to x2.
5. We must finish adding to x2 before we can subtract x2 from x3.
6. We must finish storing a difference in x4 before we can evaluate x4’s value.
7. [20] <C.2> Show the timing of this instruction sequence for the 5-stage RISC pipeline without any forwarding or bypassing hardware but assuming that a register read and a write in the same clock cycle “forwards” through the register file, as shown in Figure C.5. Use a pipeline timing chart like that in Figure C.8. Assume that the branch is handled by flushing the pipeline. If all memory references take 1 cycle, how many cycles does this loop take to execute?

Symbols should be self-explanatory/previously-defined. Here’s the timing chart:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| ld x1,0(x2) | IF | ID | EX | M | WB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| addi x1,x1,1 |  | IF | ID | - | id | EX | M | WB |  |  |  |  |  |  |  |  |  |  |  |  |
| sd x1,0(x2) |  |  | IF | - | - | ID | - | id | EX | M | WB |  |  |  |  |  |  |  |  |  |
| addi x2,x2,4 |  |  |  |  |  | IF | - | - | ID | EX | M | WB |  |  |  |  |  |  |  |  |
| sub x4,x3,x2 |  |  |  |  |  |  |  |  | IF | ID | - | id | EX | M | WB |  |  |  |  |  |
| bnez x4,Loop |  |  |  |  |  |  |  |  |  | IF | - | - | ID | - | id | EX | M | WB |  |  |
| (next instruction) |  |  |  |  |  |  |  |  |  |  |  |  | IF | - | FLUSH | | | | |  |
| ld x1,0(x2) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | IF | ID | EX | M |

Here are the instructions:

1. The ld runs through the pipeline exactly as expected.
2. The addi must wait for the ld to finish loading into x1, so it can’t read x1 until the ld starts its WB.
3. The sd stalls in 4-5 because it can’t decode when the addi is decoding. The sd must wait for the addi to finish writing to x1 before it can read and then store x1.
4. The addi stalls in 7-8 because it can’t decode when the sd is decoding. The sd read x2 way back in 8, so the addi can run without stalls after it starts the decode.
5. The sub must wait for the addi to finish writing to x2. It can then read x2 and continue without stalls.
6. The bnez can’t decode while the sub is decoding. Additionally, it can’t read x4 until 15, because that is the earliest that the sub is not using x4.
7. We grab the next sequential instruction at 13 (which is when the bnez leaves IF). However, at the end of bnez’s id stage, it knows it’s going to branch – as opposed to executing the next sequential instruction. We then flush the incorrectly-fetched instruction, and we need to make sure we flush through all stages of the pipeline.
8. The bnez knows it’s going to follow a branch in 15. It calculates the address that it’s going to branch to during the EX stage; in the M stage, it sends this address to the IF stage; in other words, we can fetch the load when the bnez is in M, which is of course at 17.

We have iterations to complete. Because the end of one iteration and the start of the next iteration are overlapped by , the first iterations each require . Because the end of the last iteration cannot be overlapped with anything, the last iteration requires . In total, then, this loop will execute in .

1. [20] <C.2> Show the timing of this instruction sequence for the 5-stage RISC pipeline with full forwarding and bypassing hardware. Use a pipeline timing chart like that shown in Figure C.8. Assume that the branch is handled by predicting it as not taken. If all memory references take 1 cycle, how many cycles does this loop take to execute?

Here’s the timing chart:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| ld x1,0(x2) | IF | ID | EX | M | WB |  |  |  |  |  |  |  |  |  |
| addi x1,x1,1 |  | IF | ID | id | EX | M | WB |  |  |  |  |  |  |  |
| sd x1,0(x2) |  |  | IF | - | ID | EX | M | WB |  |  |  |  |  |  |
| addi x2,x2,4 |  |  |  |  | IF | ID | EX | M | WB |  |  |  |  |  |
| sub x4,x3,x2 |  |  |  |  |  | IF | ID | EX | M | WB |  |  |  |  |
| bnez x4,Loop |  |  |  |  |  |  | IF | ID | id | EX | M | WB |  |  |
| (next instruction) |  |  |  |  |  |  |  | IF | - | idle | idle | idle | idle |  |
| ld x1,0(x2) |  |  |  |  |  |  |  |  |  | IF | ID | EX | M | WB |

Here are the instructions:

1. The ld runs without stalls.
2. The addi must wait for the ld to finish writing to x1. Because we’re forwarding now, the addi can read x1 immediately after the ld finishes execution.
3. The sd can’t decode while the addi is decoding. Once it starts decoding, it runs without stalls.
4. The addi doesn’t need to stall because the sd only reads x2 and that is accomplished during ID.
5. The sub doesn’t need to stall because the addi forwards x2 to the sub after it finishes executing.
6. The bnez must wait until it receives the value for x4 from the sub before it can decide whether or not to branch. It can then branch (or not) during the execution step (at time 10).
7. We are predicting that the branch is not taken, so we fetch ASAP whatever instruction follows the bnez. However, the branch *is* taken (for all but the last iteration), so this instruction then no-ops for the remainder of the pipeline.
8. We know after step 9 that the bnez will branch to loop, so we can fetch the ld instruction at 10. Because the ld does not depend on the bnez, it runs without stalls.

We again require iterations of the loop. This time, though, the first iterations overlap by . A single iteration requires . In total, then, the whole loop executes in .

1. [20] <C.2> Show the timing of this instruction sequence for the 5-stage RISC pipeline with full forwarding and bypassing hardware. Use a pipeline timing chart like that shown in Figure C.8. Assume that the branch is handled by predicting it as taken. If all memory references take 1 cycle, how many cycles does this loop take to execute?

Here’s the timing chart:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| ld x1,0(x2) | IF | ID | EX | M | WB |  |  |  |  |  |  |  |  |
| addi x1,x1,1 |  | IF | ID | id | EX | M | WB |  |  |  |  |  |  |
| sd x1,0(x2) |  |  | IF | - | ID | EX | M | WB |  |  |  |  |  |
| addi x2,x2,4 |  |  |  |  | IF | ID | EX | M | WB |  |  |  |  |
| sub x4,x3,x2 |  |  |  |  |  | IF | ID | EX | M | WB |  |  |  |
| bnez x4,Loop |  |  |  |  |  |  | IF | ID | id | EX | M | WB |  |
| ld x1,0(x2) |  |  |  |  |  |  |  |  | IF | ID | EX | M | WB |

Here are the instructions (note that the first six are the same as in problem C.1c):

1. The ld runs without stalls.
2. The addi must wait for the ld to finish writing to x1. Because we’re forwarding now, the addi can read x1 immediately after the ld finishes execution.
3. The sd can’t decode while the addi is decoding. Once it starts decoding, it runs without stalls.
4. The addi doesn’t need to stall because the sd only reads x2 and that is accomplished during ID.
5. The sub doesn’t need to stall because the addi forwards x2 to the sub after it finishes executing.
6. The bnez must wait until it receives the value for x4 from the sub before it can decide whether or not to branch. It can then branch (or not) during the execution step (at time 10).
7. Because we are predicting that the branch is taken, we fetch the ld once bnez decodes its instruction and knows *where* to branch. That is, we fetch the ld in 9. The ld then runs without stalls.

The execution time for the iterations is calculated like in the previous problem. This time, though, consecutive iterations of the loop overlap by , so this loop only requires .

C.3 [7/7/7/9] <C.2> We begin with a computer implemented in single-cycle implementation. When the stages are split by functionality, the stages do not require exactly the same amount of time. The original machine had a clock cycle time of 7 ns. After the stages were split, the measured times were: IF, 1 ns; ID, 1.5 ns; EX, 1 ns; MEM, 2 ns; and WB, 1.5 ns. The pipeline register delay is 0.1 ns.

1. [7] What is the clock cycle time of the 5-stage pipelined machine?

When pipelined, we care most about the longest stage (and the delay). Here, the clock cycle time is .

1. [7] If there is a stall every 4 instructions, what is the CPI of the new machine?

We have 5 stages (and thus 5 cycles) and a stall every 4 instructions, so the new CPI is .

1. [7] What is the speedup of the pipelined machine over the single- cycle machine?

The book tells us that . Because , the speedup here for the pipelined machine is .

1. [9] If the pipelined machine had an infinite number of stages, what would its speedup be over the single-cycle machine?

I’m assuming that part b was a standalone, hypothetical problem. In other words, I’m assuming, for the purposes of part d, that we *don’t* have a stall every four instructions.

If we had an infinite number of stages, each stage could be extremely small and thus execute in, essentially, no time. The clock cycle time of the new machine would simply be the pipeline register delay (that is, ). In that case, the speedup would be .