CSCE 692 - HOMEWORK 4 (CHAPTER 3)

(100 Points)

Due: NLT 1100 Tuesday, 26 February 2019

**Instructions:**

* Number each page (last name-pg)
* Show your work and clearly indicate your answer

**Book Problems [20 points each]: 3.1, 3.2, 3.3 (Changed from 5th Edition)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Loop: | LD | F2,0(Rx) |  | Latencies beyond single cycle |  |
| I0: | MULTD | F2,F6,F2 |  | Memory LD | +3 |
| I1: | DIVD | F8,F2,F0 |  | Memory SD | +1 |
| I2: | LD | F4,0(Ry) |  | Integer ADD, SUB | +0 |
| I3: | ADDD | F4,F0,F4 |  | Branches | +1 |
| I4: | ADDD | F10,F8,F2 |  | ADDD | +2 |
| I5: | SD | F4,0(Ry) |  | MULTD | +4 |
| I6: | ADDI | Rx,Rx,#8 |  | DIVD | +10 |
| I7: | ADDI | Ry,Ry,#8 |  |  |  |
| I8: | SUB | R20,R4,Rx |  |  |  |
| I9: | BNZ | R20,Loop |  |  |  |
| **Figure 3.47** Code and latencies for Exercises 3.1 through 3.3 | | | | |  |

3.1 [20] <3.1, 3.2> What is the baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.47 if no new instruction's execution could be initiated until the previous instruction's execution had completed? Ignore front-end fetch and decode. Assume for now that execution does not stall for lack of the next instruction, but only one instruction/cycle can be issued. Assume the branch is taken, and that there is a one-cycle branch delay slot.

* **“Just add the execution times”**

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction #** | **Instruction** | **Cycle Start** | **Latency** |
| Loop: | LD | 1 | 3 |
| 0 | MULTD | 5 | 4 |
| 1 | DIVD | 10 | 10 |
| 2 | LD | 21 | 3 |
| 3 | ADDD | 25 | 2 |
| 4 | ADDD | 28 | 2 |
| 5 | SD | 31 | 1 |
| 6 | ADDI | 33 | 0 |
| 7 | ADDI | 34 | 0 |
| 8 | SUB | 35 | 0 |
| 9 | BNZ | 36 | 1 |
| loop | LD | 38 | 3 |

The baseline performance is 37 cycles per loop.

3.2 [20] <3.1, 3.2> Think about what latency numbers really mean-they indicate the number of cycles a given function requires to produce its output, nothing more. If the overall pipeline stalls for the latency cycles of each functional unit, then you are at least guaranteed that any pair of back-to-back instructions (a "producer" followed by a "consumer") will execute correctly. But not all instruction pairs have a producer/ consumer relationship. Sometimes two adjacent instructions have nothing to do with each other. How many cycles would the loop body in the code sequence in Figure 3.48 require if the pipeline detected true data dependences and only stalled on those, rather than blindly stalling everything just because one functional unit is busy? Show the code with <Stall > inserted where necessary to accommodate stated latencies. (Hint: An instruction with latency +2 requires two <Stall> cycles to be inserted into the code sequence. Think of it this way: A one-cycle instruction has latency 1 + 0, meaning zero extra wait states. So, latency 1 + 1 implies one stall cycle; latency 1 + N has N extra stall cycles.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Loop: | LD | F2,0(Rx) |  | Latencies beyond single cycle |  |
| I0: | MULTD | F2,F6,F2 |  | Memory LD | +3 |
| I1: | DIVD | F8,F2,F0 |  | Memory SD | +1 |
| I2: | LD | F4,0(Ry) |  | Integer ADD, SUB | +0 |
| I3: | ADDD | F4,F0,F4 |  | Branches | +1 |
| I4: | ADDD | F10,F8,F2 |  | ADDD | +2 |
| I5: | SD | F4,0(Ry) |  | MULTD | +4 |
| I6: | ADDI | Rx,Rx,#8 |  | DIVD | +10 |
| I7: | ADDI | Ry,Ry,#8 |  |  |  |
| I8: | SUB | R20,R4,Rx |  |  |  |
| I9: | BNZ | R20,Loop |  |  |  |
| **Figure 3.47** Code and latencies for Exercises 3.1 through 3.3 | | | | |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction #** | **Instruction** | **Cycle Start** | **Latency** |
| Loop: | LD | 1 | 3 |
| <stall> | Mult stalling for F2 | | |
| 0 | MULTD | 5 | 4 |
| <stall> | Divide needs F2 | | |
| 1 | DIVD | 10 | 10 |
| 2 | LD | 11 | 3 |
| <stall> | ADDD needs F4 | | |
| 3 | ADDD | 15 | 2 |
| <stall> | ADDD needs F2 from Mult | | |
| 4 | ADDD | 21 | 2 |
| 5 | SD | 22 | 1 |
| 6 | ADDI | 23 | 0 |
| 7 | ADDI | 24 | 0 |
| 8 | SUB | 25 | 0 |
| 9 | BNZ | 26 | 1 |
| loop | LD | 28 | 3 |

By only stalling for true dependencies, the total time for the loop was 27 cycles.

3.3 [20] <3.7> Consider a multiple-issue design. Suppose you have two execution pipelines, each capable of beginning execution of one instruction per cycle, and enough fetch/ decode bandwidth in the front end so that it will not stall your execution. Assume results can be immediately forwarded from one execution unit to another, or to itself. Further assume that the only reason an execution pipeline would stall is to observe a true data dependency. Now how many cycles does the loop require?

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Loop: | LD | F2,0(Rx) |  | Latencies beyond single cycle |  |
| I0: | MULTD | F2,F6,F2 |  | Memory LD | +3 |
| I1: | DIVD | F8,F2,F0 |  | Memory SD | +1 |
| I2: | LD | F4,0(Ry) |  | Integer ADD, SUB | +0 |
| I3: | ADDD | F4,F0,F4 |  | Branches | +1 |
| I4: | ADDD | F10,F8,F2 |  | ADDD | +2 |
| I5: | SD | F4,0(Ry) |  | MULTD | +4 |
| I6: | ADDI | Rx,Rx,#8 |  | DIVD | +10 |
| I7: | ADDI | Ry,Ry,#8 |  |  |  |
| I8: | SUB | R20,R4,Rx |  |  |  |
| I9: | BNZ | R20,Loop |  |  |  |
| **Figure 3.47** Code and latencies for Exercises 3.1 through 3.3  This solution assumes that we must commit instructions in-order, but that does not guarantee in order completion. Also, the latency for the Branch instruction is due to the branch delay slot of one cycle.  Pipeline #1:   |  |  |  |  | | --- | --- | --- | --- | | **Instruction #** | **Instruction** | **Cycle Start** | **Latency** | | Loop: | LD | 1 | 3 | | <stall> | Stalls while loading F2 from LD | | | | 0 | MultD | 5 | 4 | | 1 | DivD | 10 | 10 | | 4 | AddD | 21 | 2 | | 6 | ADDI | 22 | 0 | | 8 | SUB | 23 | 0 | | 9 | BNZ | 24 | 1 | | | | | |  |
| Pipeline #2:   |  |  |  |  | | --- | --- | --- | --- | | **Instruction #** | **Instruction** | **Cycle Start** | **Latency** | | 2 | LD | 10 | 3 | | 3 | ADDD | 14 | 2 | | <stall> | Stalls while waiting for F4 to write into memory | | | | 5 | SD | 21 | 1 | | 7 | ADDI | 23 | 0 | | | | | |  |

By utilizing two separate pipelines, the loop will execute in 25 cycles.

**Additional Problems [20 points each]: 1, 2**

**1.** Here is an unusual loop. List all the dependencies (output, anti, and true) in the following code fragment. Indicate whether the true dependences are loop-carried or not. Finally, rewrite the loop so that it can be executed in parallel (i.e., try to remove the loop dependencies).

for (i=1; i<100; i=i+1) {

a[i] = b[i] + c[i]; /\* S1 \*/   
 b[i] = a[i] + d[i]; /\* S2 \*/

a[i+1] = a[i] + e[i]; /\* S3 \*/

}

* True dependency between S3 and S1 and S2 and S1 with **a[i]** 🡪 not loop carried
* Anti-dependence between S2 and S1 with **b[i]** 🡪 not loop carried
* Output dependence between S3 **a[i+1]** and S1 with **a[i]** 🡪loop carried

Becomes:

for (i=1; i<100; i=i+1) {

a[i] = b[i] + c[i]; /\* S1 \*/   
 b[i] = a[i] + d[i]; /\* S2: \*/

}

a[100] = b[99] + e[99]; /\* S3 – the final iteration of the loop was the only time the value of a[i+1] was not overwritten by the following loop \*/

**2.**  Using the code sequence shown below, show the complete schedule and bookkeeping in the reservation stations/register result status using the Tomasulo approach (print out as many copies of “TomasuloTemplateHW3.pdf” as you need). Show the schedule (i.e., fill in) from the form provided for each clock cycle until complete.

Code Sequence:

L.D F1, 0(R1)

MUL.D F7, F1, F5

ADD.D F1, F1, F5

ADD.D F2, F1, F7

DADDI R1, R2, 8

S.D F2, 0(R1)

Assume the following clock cycle required for each “exec” functional unit:

Loads/Stores 2 cc’s

FP Mult 7 cc’s

FP Add 2 cc’s

Any Integer op 1 cc’s

Note the DADDI (integer adder) is not normally shown in any Tomasulo unit or bookkeeping. For the purposes of this exercise we include it in our bookkeeping as far as the instruction status but do not worry about placing it in a reservation station. We also assume we have a separate CDB for integer and floating point registers.

Shown on attached spreadsheets.