CSCI 654 – Advanced Computer Architecture
Homework 3

Due: October 3, 2016

Alexander Powell

1. **3.3**

   Using a multiple-issue design and two execution pipelines only stalling for true data dependencies, the loop requires 22 cycles to complete. This performance gain is achieved by separating the instructions into the two pipelines based on where the dependencies are. Each step/stall is shown in the following list of operations.

Cycle #	Pipeline 1	Pipeline 2
1	LD F2, 0(RX)	<waiting>
2	<stall>	<waiting>
3	<stall>	<waiting>
4	<stall>	<waiting>
5	<stall>	<waiting>
6	DIVD F8, F2, F0	MULTD F2, F6, F2
7	LD F4, 0(Ry)	<waiting>
8	<stall>	<waiting>
9	<stall>	<waiting>
10	<stall>	<waiting>
11	<stall>	<waiting>
12	ADDD F4, F0, F4	<waiting>
13	<stall>	<waiting>
14	<stall>	<waiting>
15	<stall>	<waiting>
16	<stall>	<waiting>
17	<stall>	<waiting>
18	<stall>	<waiting>
19	ADDD F10, F8, F2	ADDI Rx, Ry, #8
20	ADDI Ry, Ry, #8	SD F4, 0(Ry)
21	SUB R20, R4, Rx	BNZ R20, Loop
22	<waiting>	<stall>

2. **3.5**

   The following table shows the execution pipeline order after the instructions have been reorded. By doing this, we are able to get the number of cycles per loop down to 20.

Cycle #	Pipeline 1	Pipeline 2
1	LD F2, 0(RX)	LD F4, 0(Ry)
2	<stall>	<stall>
3	<stall>	<stall>
4	<stall>	<stall>
5	<stall>	<stall>
6	DIVD F8, F2, F0	ADDD F4, F0, F4
7	MULTD F2, F6, F2	<stall>
8	<stall>	SD F4, 0(Ry)
9	<stall>	<stall>
10	<stall>	ADDI RX, RX, #8
11	<stall>	ADDI Ry, Ry, #8
12	<stall>	SUB R20, R4, RX
13	<stall>	<waiting>
14	<stall>	<waiting>
15	<stall>	<waiting>
16	<stall>	<waiting>
17	<stall>	<waiting>
18	<stall>	<waiting>
19	ADDD F10, F8, F2	BNZ R20, Loop
20	<waiting>	<stall>

3. **3.7**

The following table shows the resulting code after the register renames has been performed.

Loop:	LD T9, 0(RX)
I0:	MULTD T10, F0, F2
I1:	DIVD T11, T9, T10
I2:	LD T12, 0(Ry)
I3:	ADDD T13, F0, T12
I4:	SUBD T14, T11, T13
I5:	SD T14, 0(Ry)

4. **3.11**

To make the next table, easier to read, the instructions will be refered to by their order. For example, I0 refers to **LW R3, 0(R0)** and I1 refers to **LW R1, 0(R3)**. Also, fetch, decode, execute, memory and writeback will be denoted with F, D, E, M, and W respectively. So, for example, I0-F means the first instruction is being fetched in that cycle.

Cycle #	Instruction Stages
1	I0-F
2	I0-D, I1-F
3	I0-E, I1-D, I2-F
4	I0-M, I1-stall, I2-stall
5	I0-stall, I1-stall, I2-stall
6	I0-stall, I1-stall, I2-stall
7	I0-W, I1-E, I2-D, I3-F
8	I1-M, I2-stall, I3-stall
9	I1-stall, I2-stall, I3-stall
10	I1-stall, I2-stall, I3-stall
11	I1-W, I2-E, I3-D, I4-F
12	I2-M, I3-E, I4-D, I5-F
13	I2-W, I3-M, I4-E, I5-D
14	I3-W, I4-M, I5-E
15	I4-stall, I5-stall
16	I4-stall, I5-stall

(a) From the table above, we can see that 4 cycles are lost on each iteration to branch overhead.

(b) Assuming a static branch predictor, 2 cycles will be wasted on branch overhead. This is because the fetch/decode stages still must be performec before any prediction can be made.

(c) Using a dynamic branch predictor, no cycles will be lost to branch overhead when a correct prediction is made.

5. **3.12**

(a) From figure 3.48, the following instructions would have improved the performance if register naming was used:

```
I2, I4, I5, I6, I7
```

(b) If the register-renames version of the code from (a) is resident in the reservation station in clock cycle N with the given latencies, the code sequence requires 25 clock cycles per iteration.

(c) The following table shows the cycle-by-cycle order dispatch of the reservation station. Again, the code still requires 25 clock cycles.

Cycle #	Instruction Stages
1	I0, I1
2	I1, I2, I3
3	I1, I2, I4, I5
4	I1, I2, I4, I5, I6, I7
5	I1, I2, I4, I5, I7, I8, I9
6	I1, I2, I4, I5, I8, I9, I10

(d) If you want to improve the results of part (c), it would help the most to cut the longest instruction latency in half. The instruction with the longest latency is DIVD of 12 cycles. By choosing this method you would save 6 clock cycles per iteration.