EEL 6764 Principles of Computer Architecture Homework #4

1 Answers are attached at the end.

2 There are many possible answers to this question. One answer is that register renaming is not necessary as the code does not contain any hazards.

3 Straightforward.

4 Branch prediction

(a) 1-bit predictor. Red slots on third row show 8 total mis-predictions.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
	T,	T,	N,	T,	N,	T,	T,	T,	T,	N,	T,	T,	N
predictor	N,	T,	T,	N,	T,	N,	T,	T,	T,	T,	N,	T,	T

(b) 2-bit predictor. Red slots on third row show 4 total mis-predictions.

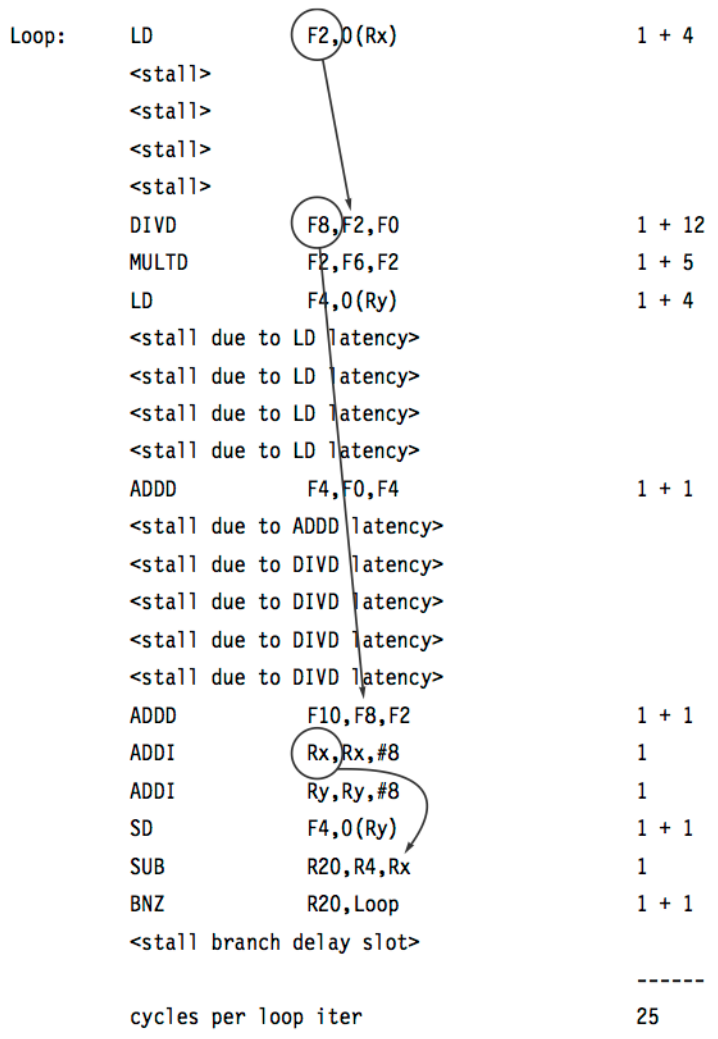
Index	1	2	3	4	5	6	7	8	9	10	11	12	13
	T,	T,	N,	T,	N,	T,	T,	T,	T,	N,	T,	T,	N
predictor	10,	11,	11,	10,	11,	10,	11,	11,	11,	11,	10,	11,	11

3.1 The baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.48, if no new instruction's execution could be initiated until the previous instruction's execution had completed, is 40. See Figure S.2. Each instruction requires one clock cycle of execution (a clock cycle in which that instruction, and only that instruction, is occupying the execution units; since every instruction must execute, the loop will take at least that many clock cycles). To that base number, we add the extra latency cycles. Don't forget the branch shadow cycle.

```
Loop:   LD       F2,0(Rx)     1 + 4
        DIVD     F8,F2,F0     1 + 12
        MULTD    F2,F6,F2     1 + 5
        LD       F4,0(Ry)     1 + 4
        ADDD     F4,F0,F4     1 + 1
        ADDD     F10,F8,F2    1 + 1
        ADDI     Rx,Rx,#8     1
        ADDI     Ry,Ry,#8     1
        SD       F4,0(Ry)     1 + 1
        SUB      R20,R4,Rx    1
        BNZ      R20,Loop     1 + 1
                              ____
        cycles per loop iter  40
```

Figure S.2 Baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.48.

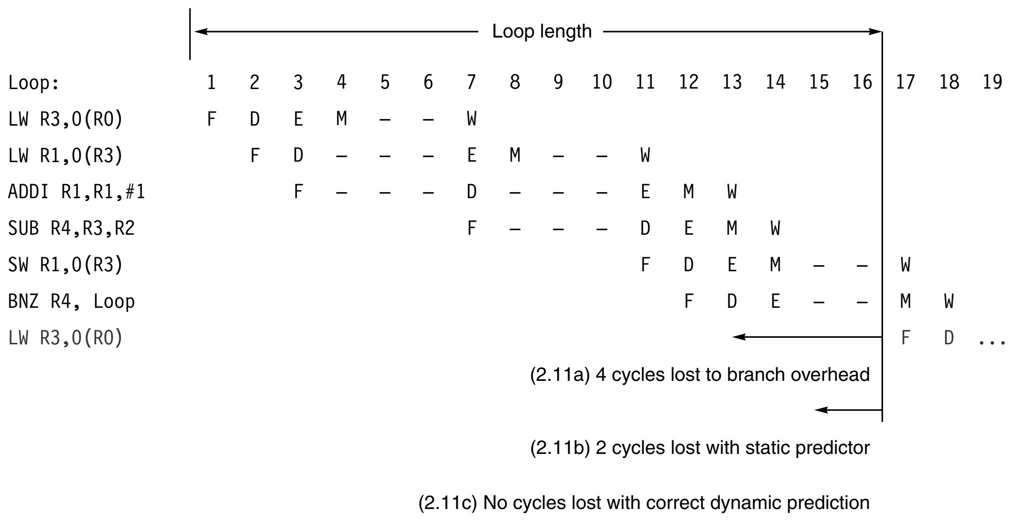
3.2 How many cycles would the loop body in the code sequence in Figure 3.48 require if the pipeline detected true data dependencies and only stalled on those, rather than blindly stalling everything just because one functional unit is busy? The answer is 25, as shown in Figure S.3. Remember, the point of the extra latency cycles is to allow an instruction to complete whatever actions it needs, in order to produce its correct output. Until that output is ready, no dependent instructions can be executed. So the first LD must stall the next instruction for three clock cycles. The MULTD produces a result for its successor, and therefore must stall 4 more clocks, and so on.



```
Loop:   LD      F2,0(Rx)                          1 + 4
        <stall>
        <stall>
        <stall>
        <stall>
        DIVD    F8,F2,F0                          1 + 12
        MULTD   F2,F6,F2                          1 + 5
        LD      F4,0(Ry)                          1 + 4
        <stall due to LD latency>
        <stall due to LD latency>
        <stall due to LD latency>
        <stall due to LD latency>
        ADDD    F4,F0,F4                          1 + 1
        <stall due to ADDD latency>
        <stall due to DIVD latency>
        <stall due to DIVD latency>
        <stall due to DIVD latency>
        <stall due to DIVD latency>
        ADDD    F10,F8,F2                         1 + 1
        ADDI    Rx,Rx,#8                          1
        ADDI    Ry,Ry,#8                          1
        SD      F4,0(Ry)                          1 + 1
        SUB     R20,R4,Rx                         1
        BNZ     R20,Loop                          1 + 1
        <stall branch delay slot>
                                                  ------
        cycles per loop iter                      25
```

Figure S.3 Number of cycles required by the loop body in the code sequence in Figure 3.48.

3.11 See Figure S.11. The convention is that an instruction does not enter the execution phase until all of its operands are ready. So the first instruction, `LW R3,0(R0)`, marches through its first three stages (F, D, E) but that M stage that comes next requires the usual cycle plus two more for latency. Until the data from a `LD` is available at the execution unit, any subsequent instructions (especially that `ADDI R1, R1, #1`, which depends on the 2nd `LW`) cannot enter the E stage, and must therefore stall at the D stage.



Loop:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
LW R3,0(R0)	F	D	E	M	–	–	W												
LW R1,0(R3)		F	D	–	–	–	E	M	–	–	W								
ADDI R1,R1,#1			F	–	–	–	D	–	–	–	E	M	W						
SUB R4,R3,R2							F	–	–	–	D	E	M	W					
SW R1,0(R3)											F	D	E	M	–	–	W		
BNZ R4, Loop												F	D	E	–	–	M	W	
LW R3,0(R0)																	F	D	...

Loop length
(2.11a) 4 cycles lost to branch overhead
(2.11b) 2 cycles lost with static predictor
(2.11c) No cycles lost with correct dynamic prediction

Figure S.11 Phases of each instruction per clock cycle for one iteration of the loop.

a. 4 cycles lost to branch overhead. Without bypassing, the results of the `SUB` instruction are not available until the `SUB`'s W stage. That tacks on an extra 4 clock cycles at the end of the loop, because the next loop's `LW R1` can't begin until the branch has completed.

b. 2 cycles lost w/ static predictor. A static branch predictor may have a heuristic like "if branch target is a negative offset, assume it's a loop edge, and loops are usually taken branches." But we still had to fetch and decode the branch to see that, so we still lose 2 clock cycles here.

c. No cycles lost w/ correct dynamic prediction. A dynamic branch predictor remembers that when the branch instruction was fetched in the past, it eventually turned out to be a branch, and this branch was taken. So a "predicted taken" will occur in the same cycle as the branch is fetched, and the next fetch after that will be to the presumed target. If correct, we've saved all of the latency cycles seen in 3.11 (a) and 3.11 (b). If not, we have some cleaning up to do.