Chapter 2 Solutions

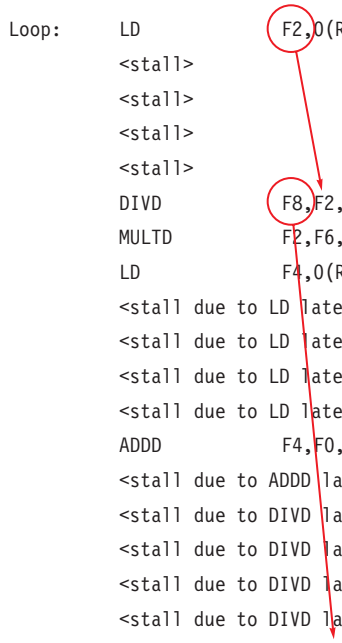
Case Study 1: Exploring the Impact of Microarchitectural Techniques

2.1 The baseline performance (in cycles, per loop iteration) of the code sequence in Figure 2.35, if no new instruction's execution could be initiated until the previous instruction's execution had completed, is 40. How did I come up with that number? 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 L.2 Baseline performance (in cycles, per loop iteration) of the code sequence in Figure 2.35.

2.2 How many cycles would the loop body in the code sequence in Figure 2.35 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 L.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 L.3 Number of cycles required by the loop body in the code sequence in Figure 2.35.

2.3 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? The answer is 22, as shown in Figure L.4. The LD goes first, as before, and the DIVD must wait for it through 4 extra latency cycles. After the DIVD comes the MULTD, which can run in the second pipe along with the DIVD, since there's no dependency between them. (Note that they both need the same input, F2, and they must both wait on F2's readiness, but there is no constraint between them.) The LD following the MULTD does not depend on the DIVD nor the MULTD, so had this been a superscalar-order-3 machine, that LD could conceivably have been executed concurrently with the DIVD and the MULTD. Since this problem posited a two-execution-pipe machine, the LD executes in the cycle following the DIVD/MULTD. The loop overhead instructions at the loop's bottom also exhibit some potential for concurrency because they do not depend on any long-latency instructions.

	Execution pipe 0		Execution pipe 1
Loop:	**LD F2,0(Rx)**	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	DIVD F8,F2,F0	;	**MULTD F2,F6,F2**
	LD F4,0(Ry)	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	<stall for LD latency>	;	<nop>
	ADD F4,F0,F4	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	ADDD F10,F8,F2	;	**ADDI Rx,Rx,#8**
	ADDI Ry,Ry,#8	;	**SD F4,0(Ry)**
	SUB R20,R4,Rx	;	**BNZ R20,Loop**
	<nop>	;	**<stall due to BNZ>**
	cycles per loop iter 22		

Figure L.4 Number of cycles required per loop.

2.4 Possible answers:

1. If an interrupt occurs between N and $N + 1$, then $N + 1$ must not have been allowed to write its results to any permanent architectural state. Alternatively, it might be permissible to delay the interrupt until $N + 1$ completes.
2. If N and $N + 1$ happen to target the same register or architectural state (say, memory), then allowing N to overwrite what $N + 1$ wrote would be wrong.
3. N might be a long floating-point op that eventually traps. $N + 1$ cannot be allowed to change arch state in case N is to be retried.

Long-latency ops are at highest risk of being passed by a subsequent op. The `DIVD` instr will complete long after the `LD F4,0(Ry)`, for example.

2.5 Figure L.5 demonstrates one possible way to reorder the instructions to improve the performance of the code in Figure 2.35. The number of cycles that this reordered code takes is 20.

	Execution pipe 0		Execution pipe 1		
Loop:	LD F2,0(Rx)	;	LD F4,0(Ry)		
	<stall for LD latency>	;	<stall for LD latency>		
	<stall for LD latency>	;	<stall for LD latency>		
	<stall for LD latency>	;	<stall for LD latency>		
	<stall for LD latency>	;	<stall for LD latency>		
	DIVD F8,F2,F0	;	ADDD F4,F0,F4		
	MULTD F2,F6,F2	;	<stall due to ADDD latency>		
	<stall due to DIVD latency>	;	SD F4,0(Ry)		
	<stall due to DIVD latency>	;	<nop>	#ops:	11
	<stall due to DIVD latency>	;	<nop>	#nops:	(20 × 2) – 11 = 29
	<stall due to DIVD latency>	;	ADDI Rx,Rx,#8		
	<stall due to DIVD latency>	;	ADDI Ry,Ry,#8		
	<stall due to DIVD latency>	;	<nop>		
	<stall due to DIVD latency>	;	<nop>		
	<stall due to DIVD latency>	;	<nop>		
	<stall due to DIVD latency>	;	<nop>		
	<stall due to DIVD latency>	;	<nop>		
	<stall due to DIVD latency>	;	SUB R20,R4,Rx		
	ADDD F10,F8,F2	;	BNZ R20,Loop		
	<nop>	;	<stall due to BNZ>		
	cycles per loop iter 20				

Figure L.5 Number of cycles taken by reordered code.

2.6 a. Fraction of all cycles, counting both pipes, wasted in the reordered code shown in Figure L.5:

11 ops out of 2x20 opportunities.
1 – 11/40 = 1 – 0.275
= 0.725

b. Results of hand-unrolling two iterations of the loop from code shown in Figure L.6:

	Execution pipe 0		Execution pipe 1
Loop:	LD F2,0(Rx)	;	LD F4,0(Ry)
	LD F2,0(Rx)	;	LD F4,0(Ry)
	<stall for LD latency>	;	<stall for LD latency>
	<stall for LD latency>	;	<stall for LD latency>
	<stall for LD latency>	;	<stall for LD latency>
	DIVD F8,F2,F0	;	ADDD F4,F0,F4
	DIVD F8,F2,F0	;	ADDD F4,F0,F4
	MULTD F2,F0,F2	;	SD F4,0(Ry)
	MULTD F2,F6,F2	;	SD F4,0(Ry)
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	ADDI Rx,Rx,#16
	<stall due to DIVD latency>	;	ADDI Ry,Ry,#16
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	<stall due to DIVD latency>	;	<nop>
	ADDD F10,F8,F2	;	SUB R20,R4,Rx
	ADDD F10,F8,F2	;	BNZ R20,Loop
	<nop>	;	<stall due to BNZ>
	cycles per loop iter 22		

Figure L.6 Hand-unrolling two iterations of the loop from code shown in Figure L.5.

c. $$\text{Speedup} = \frac{\text{exec time w/o enhancement}}{\text{exec time with enhancement}}$$

Speedup = 20 / (22/2)
= 1.82

2.7 Consider the code sequence in Figure 2.36. Every time you see a destination register in the code, substitute the next available T, beginning with T9. Then update all the src (source) registers accordingly, so that true data dependencies are maintained. Show the resulting code. (*Hint:* See Figure 2.37.)

Loop:	LD	T9,0(Rx)
I0:	MULTD	T10,F0,T2
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)

Figure L.7 Register renaming.