

- 2.1 [10] <1.8, 2.1, 2.2> What would be 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? 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.
- 2.2 [10] <1.8, 2.1, 2.2> 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 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? Show the code with <stall> inserted where necessary to accommodate stated latencies. (*Hint*: An instruction with latency “+2” needs 2 <stall> cycles to be inserted into the code sequence. Think of it this way: a 1-cycle instruction has latency 1 + 0, meaning zero extra wait states. So latency 1 + 1 implies 1 stall cycle; latency 1 + *N* has *N* extra stall cycles.)

		Latencies beyond single cycle
Loop:	LD F2,0(Rx)	Memory LD +4
I0:	DIVD F8,F2,F0	Memory SD +1
I1:	MULTD F2,F6,F2	Integer ADD, SUB +0
I2:	LD F4,0(Ry)	Branches +1
I3:	ADD F4,F0,F4	ADD +1
I4:	ADD F10,F8,F2	MULTD +5
I5:	ADDI Rx,Rx,#8	DIVD +12
I6:	ADDI Ry,Ry,#8	
I7:	SD F4,0(Ry)	
I8:	SUB R20,R4,Rx	
I9:	BNZ R20,Loop	

Figure 2.35 Code and latencies for Exercises 2.1 through 2.6.

## Re-ordered code of Problem 2.5



- 2.6 [10/10] <2.1, 2.2> Every cycle that does not initiate a new operation in a pipe is a lost opportunity, in the sense that your hardware is not “living up to its potential.”
- [10] <2.1, 2.2> In your reordered code from Exercise 2.5, what fraction of all cycles, counting both pipes, were wasted (did not initiate a new op)?
  - [10] <2.1, 2.2> Loop unrolling is one standard compiler technique for finding more parallelism in code, in order to minimize the lost opportunities for performance. Hand-unroll two iterations of the loop in your reordered code from Exercise 2.5.
  - [10] <2.1, 2.2> What speedup did you obtain? (For this exercise, just color the  $N + 1$  iteration’s instructions green to distinguish them from the  $N$ th iteration’s; if you were actually unrolling the loop you would have to reassign registers to prevent collisions between the iterations.)
- 2.7 [15] <2.1> Computers spend most of their time in loops, so multiple loop iterations are great places to speculatively find more work to keep CPU resources busy. Nothing is ever easy, though; the compiler emitted only one copy of that loop’s code, so even though multiple iterations are handling distinct data, they will appear to use the same registers. To keep multiple iterations’ register usages from colliding, we rename their registers. Figure 2.36 shows example code that we would like our hardware to rename.

A compiler could have simply unrolled the loop and used different registers to avoid conflicts, but if we expect our hardware to unroll the loop, it must also do the register renaming. How? Assume your hardware has a pool of temporary registers (call them T registers, and assume there are 64 of them, T0 through T63) that it can substitute for those registers designated by the compiler. This rename hardware is indexed by the src (source) register designation, and the value in the table is the T register of the last destination that targeted that register. (Think of these table values as producers, and the src registers are the consumers; it doesn't much matter where the producer puts its result as long as its consumers can find it.) 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 registers accordingly, so that true data dependencies are maintained. Show the resulting code. (*Hint: See Figure 2.37.*)

---

```

Loop: LD    F4,0(Rx)
I0:  MULTD F2,F0,F2
I1:  DIVD  F8,F4,F2
I2:  LD    F4,0(Ry)
I3:  ADDD  F6,F0,F4
I4:  SUBD  F8,F8,F6
I5:  SD    F8,0(Ry)

```

---

**Figure 2.36** Sample code for register renaming practice.

---

```

I0:  LD    T9,0(Rx)
I1:  MULTD T10,F0,T9
. . .

```

---

**Figure 2.37** Hint: expected output of register renaming.