Computer Architecture
HW#3 Solution

3.1

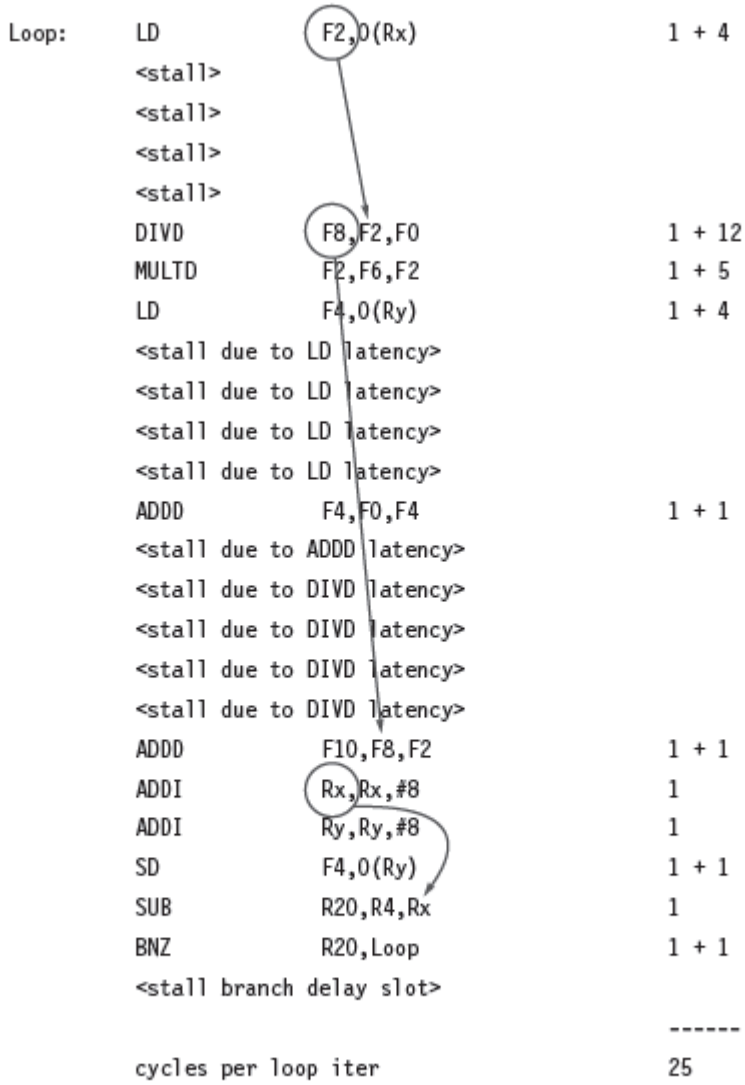
The baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.48, if no new instruction's execution could be initiated until the previous instruction's execution had completed, is 40. See Figure S.2. Each instruction requires one clock cycle of execution (a clock cycle in 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.

3.3

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 S.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 S.4 Number of cycles required per loop.

3.4

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.

3.5

	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: $(20 \times 2) - 11 = 29$
	<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 S.5 Number of cycles taken by reordered code.

3.6

a. Fraction of all cycles, counting both pipes, wasted in the reordered code shown in Figure S.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 S.6:

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

Speedup = 20 / (22/2)
= 1.82

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

3.10

An example of an event that, in the presence of self-draining pipelines, could disrupt the pipelining and yield wrong results is shown in Figure S.10.

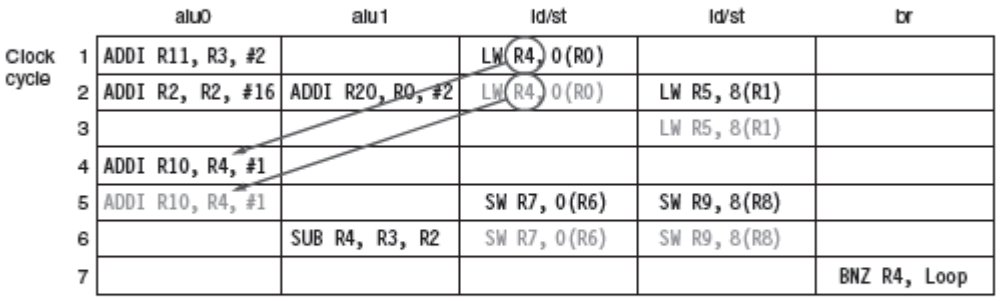


Figure S.10 Example of an event that yields wrong results. What could go wrong with this? If an interrupt is taken between clock cycles 1 and 4, then the results of the LW at cycle 2 will end up in R1, instead of the LW at cycle 1. Bank stalls and ECC stalls will cause the same effect—pipes will drain, and the last writer wins, a classic WAW hazard. All other "intermediate" results are lost.

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.

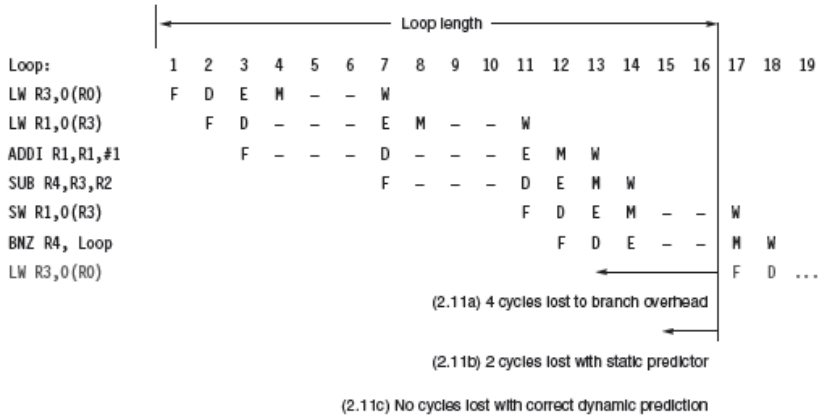


Figure S.11 Phases of each instruction per clock cycle for one iteration of the loop.

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.

3.14

a.

Iteration	Instruction	Issues at	Executes/ Memory	Write CDB at	Comment
1	L.D F2,0(R1)	1	2	3	First issue
1	MUL.D F4,F2,F0	2	4	19	Wait for F2 Mult rs [3–4] Mult use [5–18]
1	L.D F6,0(R2)	3	4	5	Ldbuf [4]
1	ADD.D F6,F4,F6	4	20	30	Wait for F4 Add rs [5–20] Add use [21–29]
1	S.D F6,0(R2)	5	31		Wait for F6 Stbuf1 [6–31]
1	DADDIU R1,R1,#8	6	7	8	
1	DADDIU R2,R2,#8	7	8	9	
1	DSLTU R3,R1,R4	8	9	10	
1	BNEZ R3,foo	9	11		Wait for R3
2	L.D F2,0(R1)	10	12	13	Wait for BNEZ Ldbuf [11–12]
2	MUL.D F4,F2,F0	11	~~14~~ 19	34	Wait for F2 Mult busy Mult rs [12–19] Mult use [20–33]
2	L.D F6,0(R2)	12	13	14	Ldbuf [13]
2	ADD.D F6,F4,F6	13	35	45	Wait for F4 Add rs [14–35] Add use [36–44]
2	S.D F6,0(R2)	14	46		Wait for F6 Stbuf [15–46]
2	DADDIU R1,R1,#8	15	16	17	
2	DADDIU R2,R2,#8	16	17	18	
2	DSLTU R3,R1,R4	17	18	20	
2	BNEZ R3,foo	18	20		Wait for R3
3	L.D F2,0(R1)	19	21	22	Wait for BNEZ Ldbuf [20–21]
3	MUL.D F4,F2,F0	20	~~23~~ 34	49	Wait for F2 Mult busy Mult rs [21–34] Mult use [35–48]
3	L.D F6,0(R2)	21	22	23	Ldbuf [22]
3	ADD.D F6,F4,F6	22	50	60	Wait for F4 Add rs [23–49] Add use [51–59]
3	S.D F6,0(R2)	23	55		Wait for F6 Stbuf [24–55]
3	DADDIU R1,R1,#8	24	25	26	
3	DADDIU R2,R2,#8	25	26	27	
3	DSLTU R3,R1,R4	26	27	28	
3	BNEZ R3,foo	27	29		Wait for R3

b.

Iteration	Instruction	Issues at	Executes/ Memory	Write CDB at	Comment
1	L.D F2,0(R1)	1	2	3	
1	MUL.D F4,F2,F0	1	4	19	Wait for F2 Mult rs [2–4] Mult use [5]
1	L.D F6,0(R2)	2	3	4	Ldbuf [3]
1	ADD.D F6,F4,F6	2	20	30	Wait for F4 Add rs [3–20] Add use [21]
1	S.D F6,0(R2)	3	31		Wait for F6 Stbuf [4–31]
1	DADDIU R1,R1,#8	3	4	5	
1	DADDIU R2,R2,#8	4	5	6	
1	DSLTU R3,R1,R4	4	6	7	INT busy INT rs [5–6]
1	BNEZ R3,foo	5	7		INT busy INT rs [6–7]
2	L.D F2,0(R1)	6	8	9	Wait for BEQZ
2	MUL.D F4,F2,F0	6	10	25	Wait for F2 Mult rs [7–10] Mult use [11]
2	L.D F6,0(R2)	7	9	10	INT busy INT rs [8–9]
2	ADD.D F6,F4,F6	7	26	36	Wait for F4 Add RS [8–26] Add use [27]
2	S.D F6,0(R2)	8	37		Wait for F6
2	DADDIU R1,R1,#8	8	10	11	INT busy INT rs [8–10]
2	DADDIU R2,R2,#8	9	11	12	INT busy INT rs [10–11]
2	DSLTU R3,R1,R4	9	12	13	INT busy INT rs [10–12]
2	BNEZ R3,foo	10	14		Wait for R3
3	L.D F2,0(R1)	11	15	16	Wait for BNEZ
3	MUL.D F4,F2,F0	11	17	32	Wait for F2 Mult rs [12–17] Mult use [17]
3	L.D F6,0(R2)	12	16	17	INT busy INT rs [13–16]
3	ADD.D F6,F4,F6	12	33	43	Wait for F4 Add rs [13–33] Add use [33]
3	S.D F6,0(R2)	14	44		Wait for F6 INT rs full in 15
3	DADDIU R1,R1,#8	15	17		INT rs full and busy INT rs [17]
3	DADDIU R2,R2,#8	16	18		INT rs full and busy INT rs [18]
3	DSLTU R3,R1,R4	20	21		INT rs full
3	BNEZ R3,foo	21	22		INT rs full