Chapter 3 Solutions

Case Study 1: Exploring the Impact of Microarchitectural Techniques

3.7

Consider the code sequence in Figure 3.48. 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 3.49.)

```
addi T0, x1, x1
addi T1, T0, T0
addi x1, T1, T1
```

Value in X1 should be 40

Figure S.5 Value of X1 when the sequence has been executed.

3.9

See Figure S.5.

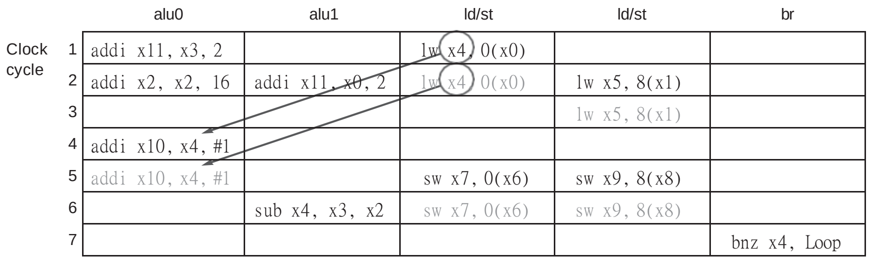
3.10

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.6.

3.11

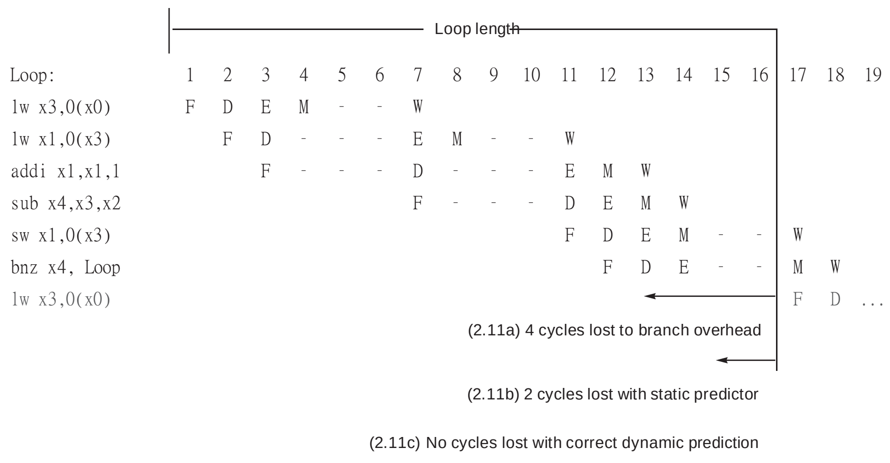
See Figure S.7. The convention is that an instruction does not enter the execution phase until all of its operands are ready. So, the first instruction, `ld x1,0(x0)`, marches through its first three stages (F, D, E) but that M stage that comes next requires the usual cycle plus two more for latency. Until the data from a `ld` is available at the execution unit, any subsequent instructions (especially that `addi x1,x1,#1`, which depends on the 2nd `ld`) cannot enter the E stage, and must therefore stall at the D stage.

a. Four 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 `ld x1` can't begin until the branch has completed.



Clock cycle	alu0	alu1	ld/st	ld/st	br
1	addi x11, x3, 2		lw x4, 0(x0)		
2	addi x2, x2, 16	addi x11, x0, 2	lw x4, 0(x0)	lw x5, 8(x1)	
3				lw x5, 8(x1)	
4	addi x10, x4, #1				
5	addi x10, x4, #1		sw x7, 0(x6)	sw x9, 8(x8)	
6		sub x4, x3, x2	sw x7, 0(x6)	sw x9, 8(x8)	
7					bnz x4, Loop

Figure S.6 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.


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

Loop:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
lw x3,0(x0)	F	D	E	M	-	-	W												
lw x1,0(x3)		F	D	-	-	-	E	M	-	-	W								
addi x1,x1,1			F	-	-	-	D	-	-	-	E	M	W						
sub x4,x3,x2							F	-	-	-	D	E	M	W					
sw x1,0(x3)											F	D	E	M	-	-	W		
bnz x4, Loop												F	D	E	-	-	M	W	
lw x3,0(x0)																	F	D	...

Figure S.7 Phases of each instruction per clock cycle for one iteration of the loop.

b. Two 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.

3.12

a. See Figure S.8.

b. See Figure S.9. The number of clock cycles taken by the code sequence is 25.

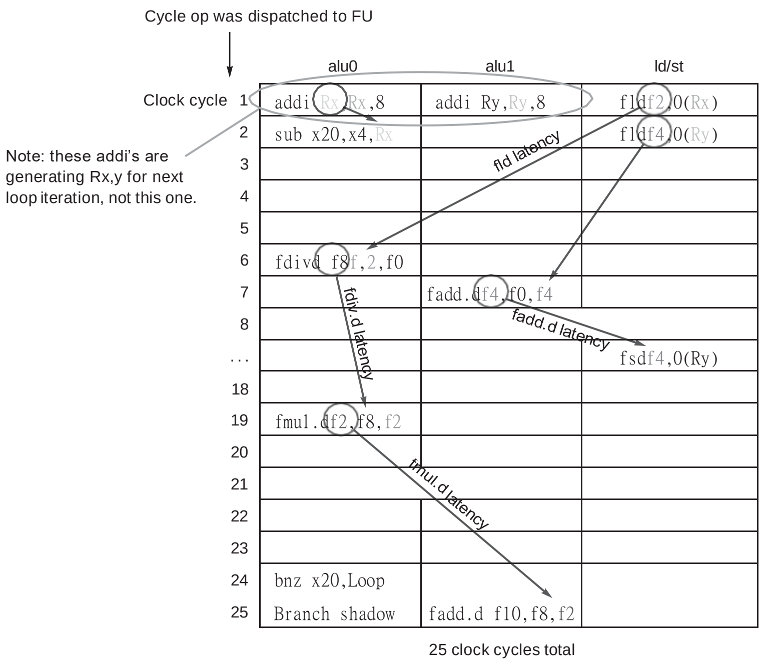
c. See Figures S.10 and S.11. The bold instructions are those instructions that are present in the RS, and ready for dispatch. Think of this exercise from the Reservation Station's point of view: at any given clock cycle, it can only "see" the instructions that were previously written into it, that have not already dispatched. From that pool, the RS's job is to identify and dispatch the two eligible instructions that will most boost machine performance.

d. See Figure S.12.
   1. Another ALU: 0% improvement
   2. Another LD/ST unit: 0% improvement
   3. Full bypassing: critical path is `fld` –> `fdiv.d` –> `fmult.d` –> `fadd.d`. Bypassing would save 1 cycle from latency of each, so 4 cycles total.
   4. Cutting longest latency in half: divider is longest at 12 cycles. This would save 6 cycles total.

e. See Figure S.13.

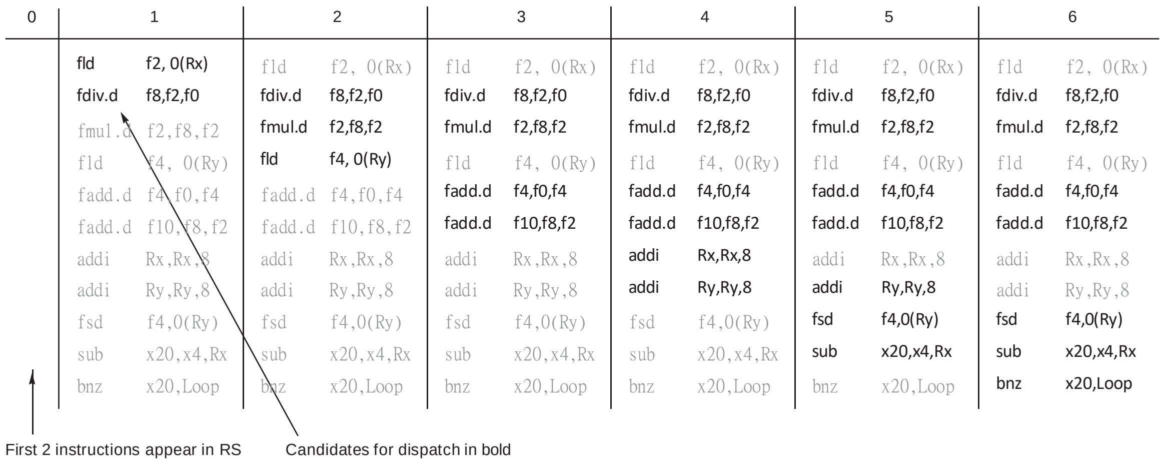
```
fld         f2,0(Rx)
fdiv.d      f8,f2,f0
fmul.d      f2,f8,f2     ; reg renaming doesn’ t really help here, due to
                         ; true data dependencies on F8 and F2
fld         F4,0(Ry)     ; this LD is independent of the previous 3
                         ; instrs and can be performed earlier than
                         ; pgm order. It feeds the next ADDD, and ADDD
                         ; feeds the SD below. But there’ s a true data
                         ; dependency chain through all, so no benefit
fadd.d      f4,f0,f4
fadd.d      f10,f8,f2    ; This ADDD still has to wait for DIVD latency,
                         ; no matter what you call their rendezvous reg
addi        Rx,Rx,#8     ; rename for next loop iteration
addi        Ry,Ry,#8     ; rename for next loop iteration
fsd         f4,0(Ry)     ; This SD can start when the ADDD’ s latency has
                         ; transpired. With reg renaming, doesn’ t have
                         ; to wait until the LD of (a different) F4 has
                         ; completed.
sub         x20,x4,Rx
bnz         x20,Loop
```

Figure S.8 Instructions in code where register renaming improves performance.


Cycle op was dispatched to FU
Clock cycle
Note: these addi’s are generating Rx,y for next loop iteration, not this one.
fld latency
fdiv.d latency
fadd.d latency
fmul.d latency
25 clock cycles total

Clock cycle	alu0	alu1	ld/st
1	addi Rx,Rx,8	addi Ry,Ry,8	fld f2,0(Rx)
2	sub x20,x4,Rx		fld f4,0(Ry)
3			
4			
5			
6	fdivd f8,f2,f0		
7		fadd.d f4,f0,f4	
8			
...			fsd f4,0(Ry)
18			
19	fmul.d f2,f8,f2		
20			
21			
22			
23			
24	bnz x20,Loop		
25	Branch shadow	fadd.d f10,f8,f2	

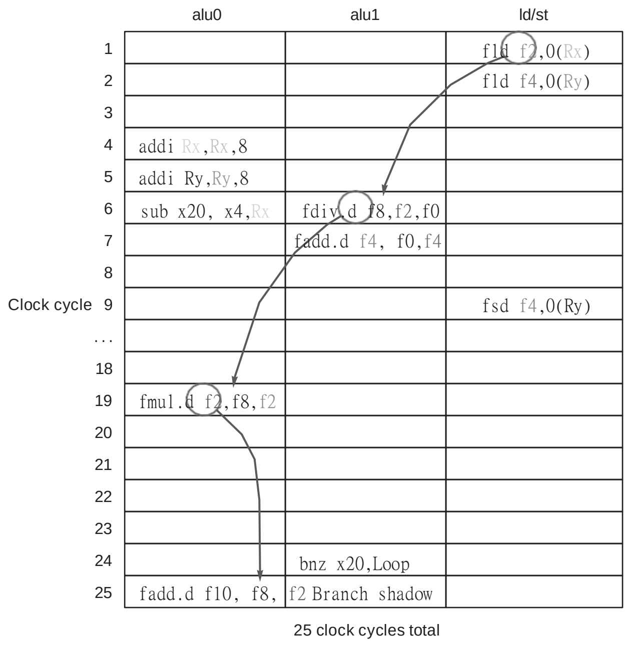
Figure S.9 Number of clock cycles taken by the code sequence.



0	1	2	3	4	5	6
	fld f2, 0(Rx)	fld f2, 0(Rx)	fld f2, 0(Rx)	fld f2, 0(Rx)	fld f2, 0(Rx)	fld f2, 0(Rx)
	fdiv.d f8,f2,f0	**fdiv.d f8,f2,f0**	**fdiv.d f8,f2,f0**	**fdiv.d f8,f2,f0**	**fdiv.d f8,f2,f0**	**fdiv.d f8,f2,f0**
	fmul.d f2,f8,f2	**fmul.d f2,f8,f2**	**fmul.d f2,f8,f2**	**fmul.d f2,f8,f2**	**fmul.d f2,f8,f2**	**fmul.d f2,f8,f2**
	fld f4, 0(Ry)	**fld f4, 0(Ry)**	fld f4, 0(Ry)	fld f4, 0(Ry)	fld f4, 0(Ry)	fld f4, 0(Ry)
	fadd.d f4,f0,f4	fadd.d f4,f0,f4	**fadd.d f4,f0,f4**	**fadd.d f4,f0,f4**	**fadd.d f4,f0,f4**	**fadd.d f4,f0,f4**
	fadd.d f10,f8,f2	fadd.d f10,f8,f2	**fadd.d f10,f8,f2**	**fadd.d f10,f8,f2**	**fadd.d f10,f8,f2**	**fadd.d f10,f8,f2**
	addi Rx,Rx,8	addi Rx,Rx,8	addi Rx,Rx,8	**addi Rx,Rx,8**	addi Rx,Rx,8	addi Rx,Rx,8
	addi Ry,Ry,8	addi Ry,Ry,8	addi Ry,Ry,8	**addi Ry,Ry,8**	**addi Ry,Ry,8**	addi Ry,Ry,8
	fsd f4,0(Ry)	fsd f4,0(Ry)	fsd f4,0(Ry)	fsd f4,0(Ry)	**fsd f4,0(Ry)**	**fsd f4,0(Ry)**
	sub x20,x4,Rx	sub x20,x4,Rx	sub x20,x4,Rx	sub x20,x4,Rx	**sub x20,x4,Rx**	**sub x20,x4,Rx**
	bnz x20,Loop	bnz x20,Loop	bnz x20,Loop	bnz x20,Loop	bnz x20,Loop	**bnz x20,Loop**

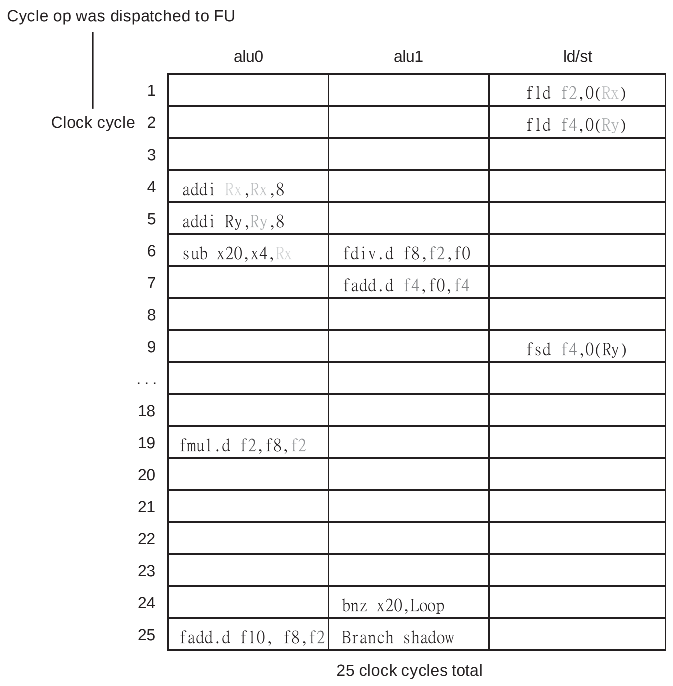
First 2 instructions appear in RS
Candidates for dispatch in bold

Figure S.10 Candidates for dispatch.



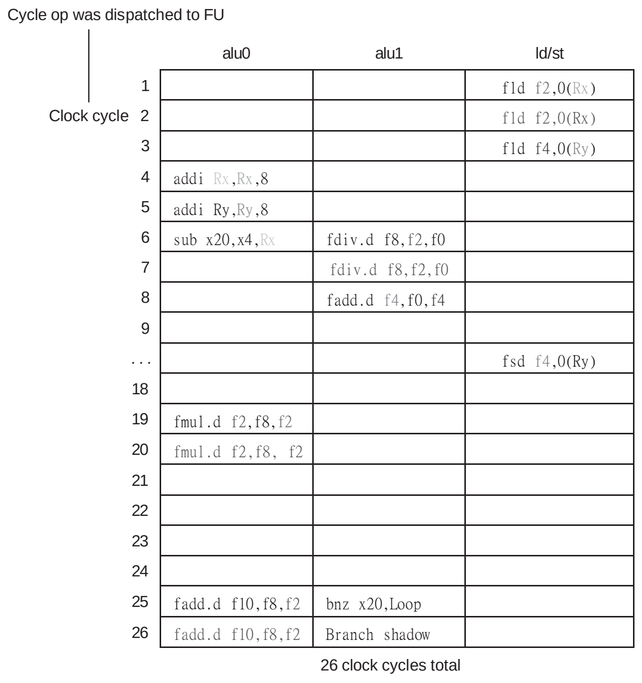
Clock cycle	alu0	alu1	ld/st
1			fld f2,0(Rx)
2			fld f4,0(Ry)
3			
4	addi Rx,Rx,8		
5	addi Ry,Ry,8		
6	sub x20, x4,Rx	fdiv.d f8,f2,f0	
7		fadd.d f4, f0,f4	
8			
9			fsd f4,0(Ry)
. . .			
18			
19	fmul.d f2,f8,f2		
20			
21			
22			
23			
24		bnz x20,Loop	
25	fadd.d f10, f8, f2	Branch shadow	

Figure S.11 5 Number of clock cycles required.


Cycle op was dispatched to FU
Clock cycle
25 clock cycles total

	alu0	alu1	ld/st
1			fld f2,0(Rx)
2			fld f4,0(Ry)
3			
4	addi Rx,Rx,8		
5	addi Ry,Ry,8		
6	sub x20,x4,Rx	fdiv.d f8,f2,f0	
7		fadd.d f4,f0,f4	
8			
9			fsd f4,0(Ry)
. . .			
18			
19	fmul.d f2,f8,f2		
20			
21			
22			
23			
24		bnz x20,Loop	
25	fadd.d f10, f8,f2	Branch shadow	

Figure S.12 Speedup is (execution time without enhancement)/(execution time with enhancement) **5** 25/(25 **2** 6) **5** 1.316.


Cycle op was dispatched to FU
Clock cycle
26 clock cycles total

	alu0	alu1	ld/st
1			fld f2,0(Rx)
2			fld f2,0(Rx)
3			fld f4,0(Ry)
4	addi Rx,Rx,8		
5	addi Ry,Ry,8		
6	sub x20,x4,Rx	fdiv.d f8,f2,f0	
7		fdiv.d f8,f2,f0	
8		fadd.d f4,f0,f4	
9			
. . .			fsd f4,0(Ry)
18			
19	fmul.d f2,f8,f2		
20	fmul.d f2,f8, f2		
21			
22			
23			
24			
25	fadd.d f10,f8,f2	bnz x20,Loop	
26	fadd.d f10,f8,f2	Branch shadow	

Figure S.13 Number of clock cycles required to do two loops' worth of work. Critical path is LD -> DIVD -> MULTD -> ADDD. If RS schedules 2nd loop's critical LD in cycle 2, then loop 2's critical dependency chain will be the same length as loop 1's is. Since we're not functional-unit-limited for this code, only one extra clock cycle is needed.

3.13

Processor A:

Cycle	Slot 1	Slot 2	Notes
1	fld x1(thread 0)	fld x1(thread 1)	threads 1,2 stalled until cycle 5
2	fld x1 (thread 2)	fld x1 (thread 3)	threads 3,4 stalled until cycle 6
3	stall	stall	
4	stall	stall	
5	fld x2(thread 0)	fld x2(thread 1)	threads 1,2 stalled until cycle 9
6	fld x2 (thread 2)	fld x2 (thread 3)	threads 3,4 stalled until cycle 10
7	stall	stall	
8	stall	stall	
…			
33	beq (thread 0)	beq (thread 1)	threads 1,2 stalled until cycle 37
34	beq (thread 2)	beq (thread 3)	threads 3,4 stalled until cycle 38
35	stall	stall	
36	stall	stall	
…			
65	addi (thread 0)	addi (thread 1)	
66	addi (thread 2)	addi (thread 3)	
67	blt (thread 0)	blt (thread 1)	
68	blt (thread 2)	blt (thread 3)	
69	stall	stall	
70	stall	stall	first iteration ends
71			second iteration begins
140			second iteration ends

Processor B:

Cycle	Slot 1	Slot 2	Slot 3	Slot 4	Notes
1	fld x1 (th 0)	fld x2 (th 0)	LD x3 (th 0)	LD x4 (th 0)	
2	fld x1 (th 1)	fld x2 (th 1)	fld x3 (th 1)	fld x4 (th 1)	
3	fld x1 (th 2)	fld x2 (th 2)	fld x3 (th 2)	fld x4 (th 2)	
4	fld x1 (th 3)	fld x2 (th 3)	fld x3 (th 3)	fld x4 (th 3)	
5	fld x5 (th 0)	fld x6 (th 0)	fld x7 (th 0)	fld x8 (th 0)	
6	fld x5 (th 1)	fld x6 (th 1)	fld x7 (th 1)	fld x8 (th 1)	
7	fld x5 (th 2)	fld x6 (th 2)	fld x7 (th 2)	fld x8 (th 2)	
8	fld x5 (th 3)	fld x6 (th 3)	fld x7 (th 3)	fld x8 (th 3)	
9	beq (th 0)				first beq of each thread
10	beq (th 1)				

11	beq (th 2)	
12	beq (th 3)	
13	beq (th 0)	second beq if each thread
14	beq (th 1)	
15	beq (th 2)	
16	beq (th 3)	
…		
41	addi (th 0)	
42	addi (th 1)	
43	addi (th 2)	
44	addi (th 3)	
45	blt (th 0)	
46	blt (th 1)	
47	blt (th 2)	
48	blt (th 3)	end of first iteration
…		
96		second iteration ends

Processor C:

Cycle	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6	Slot 7	Slot 8	Notes
1	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	
2	stall								
3	stall								
4	stall								
5	beq x1 (th0)								
6	stall								
7	stall								
8	stall								
9	beq x2 (th0)								
10	stall								
11	stall								
12	stall								
…									
37	addi (th 0)								

38	blt (th0)								
39	stall								
40	stall								
41	stall								
42	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	fld (th 0)	start of second iteration (th 0)
43	stall								
44	stall								
45	stall								
…									
83	fld (th 1)	fld (th 2)	fld (th 3)	fld (th 4)	fld (th 5)	fld (th 6)	fld (th 7)	fld (th 8)	start of first iteration (th 1)
…									
328									end of second iteration (th 3)

3.18

For this problem we are given the base CPI without branch stalls. From this we can compute the number of stalls given by no BTB and with the BTB: CPI_{noBTB} and CPI_{BTB} and the resulting speedup given by the BTB:

$$\text{Speedup} = \frac{CPI_{noBTB}}{CPI_{BTB}} = \frac{CPI_{base} + Stalls_{base}}{CPI_{base} + Stalls_{BTB}}$$

$$Stalls_{noBTB} = 15\% \times 2 = 0.30$$

To compute Stalls$_{BTB}$, consider the following table:

BTB result	BTB prediction	Frequency (per instruction)	Penalty (cycles)
Miss		15% × 10% = 1.5%	3
Hit	Correct	15% × 90% × 90% = 12.1%	0
Hit	Incorrect	15% × 90% × 10% = 1.3%	4

Therefore:

$$\text{Stalls}_{\text{BTB}} = (1.5\% \times 3) + (12.1\% \times 0) + (1.3\% \times 4) = 1.2$$

$$\text{Speed up} = \frac{1.0 + 0.30}{1.0 + 0.097} = 1.2$$