# Chapter 3 Solutions

## Case Study 1: Exploring the Impact of Microarchitectural Techniques

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.
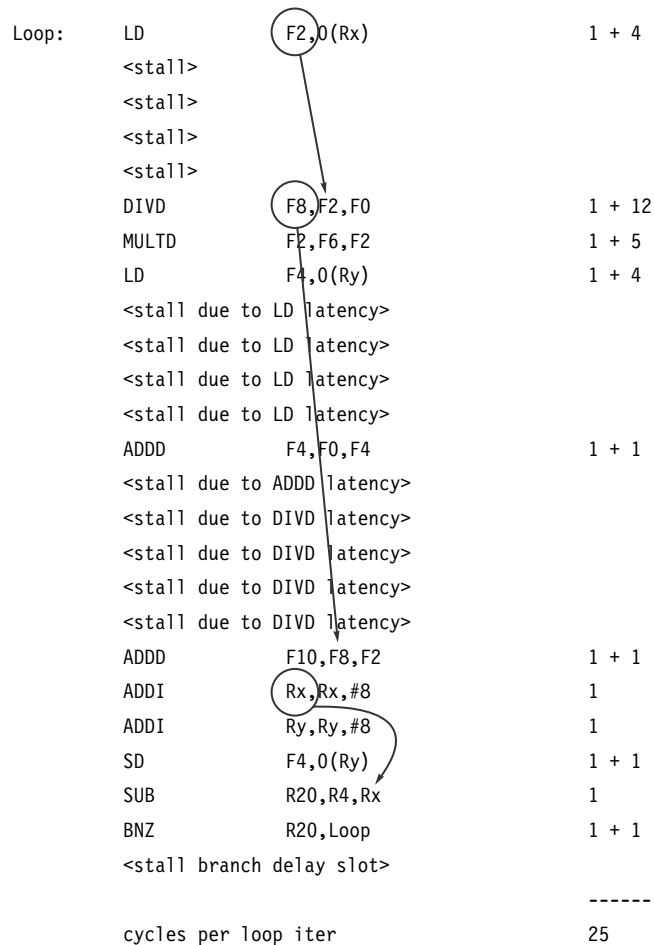
```
Loop:     LD              F2,0(Rx)                    1 + 4
          <stall>
          <stall>
          <stall>
          <stall>
          DIVD            F8,F2,F0                    1 + 12
          MULTD           F2,F6,F2                    1 + 5
          LD              F4,0(Ry)                    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,

| | 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.**

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.

3.4   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.

3.5 Figure S.5 demonstrates one possible way to reorder the instructions to improve the performance of the code in Figure 3.48. 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 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. Speedup $= \dfrac{\text{exec time w/o enhancement}}{\text{exec time with enhancement}}$

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.**

3.7    Consider the code sequence in Figure 3.49. 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 3.50.)

| | | |
|---|---|---|
| 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 S.7  Register renaming.**

3.8 See Figure S.8. The rename table has arbitrary values at clock cycle $N-1$. Look at the next two instructions (I0 and I1): I0 targets the F1 register, and I1 will write the F4 register. This means that in clock cycle $N$, the rename table will have had its entries 1 and 4 overwritten with the next available Temp register designators. I0 gets renamed first, so it gets the first T reg (9). I1 then gets renamed to T10. In clock cycle $N$, instructions I2 and I3 come along; I2 will overwrite F6, and I3 will write F0. This means the rename table's entry 6 gets 11 (the next available T reg), and rename table entry 0 is written to the T reg after that (12). In principle, you don't have to allocate T regs sequentially, but it's much easier in hardware if you do.
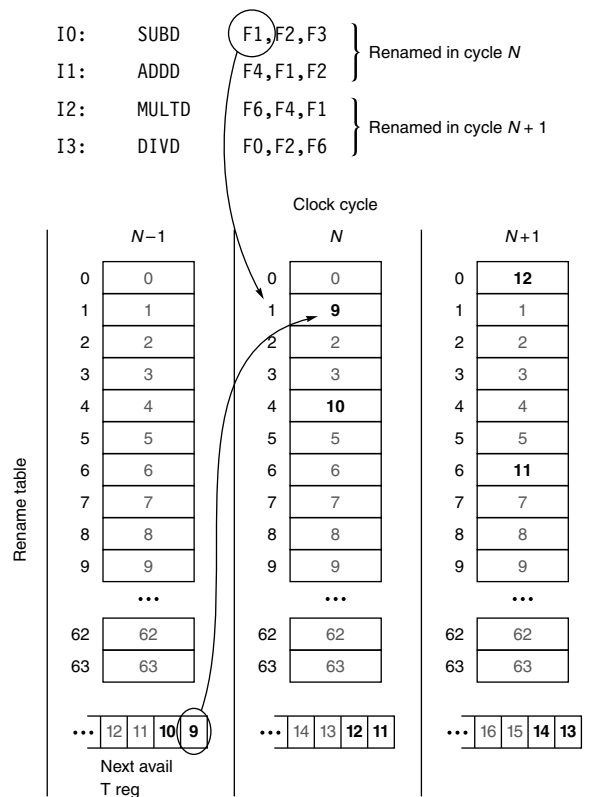


**Figure S.8** Cycle-by-cycle state of the rename table for every instruction of the code in Figure 3.51.

3.9 See Figure S.9.

| ADD | R1, R1, R1; | 5 + 5 –> 10 |
| ADD | R1, R1, R1; | 10 + 10 –> 20 |
| ADD | R1, R1, R1; | 20 + 20 –> 40 |

**Figure S.9** Value of R1 when the sequence has been executed.

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.

| | alu0 | alu1 | ld/st | ld/st | br |
|---|---|---|---|---|---|
| Clock cycle 1 | ADDI R11, R3, #2 | | LW R4, 0(R0) | | |
| 2 | ADDI R2, R2, #16 | ADDI R20, R0, #2 | LW R4, 0(R0) | LW R5, 8(R1) | |
| 3 | | | | LW R5, 8(R1) | |
| 4 | ADDI R10, R4, #1 | | | | |
| 5 | ADDI R10, R4, #1 | | SW R7, 0(R6) | SW R9, 8(R8) | |
| 6 | | SUB R4, R3, R2 | SW R7, 0(R6) | SW R9, 8(R8) | |
| 7 | | | | | BNZ R4, Loop |

**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.
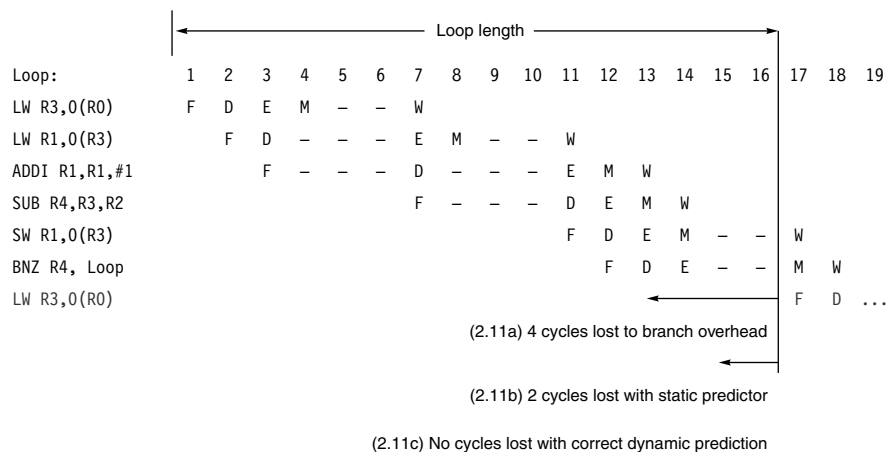


|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Loop: | | | | | | | | | | | | | | | | | | | |
| LW R3,0(R0) | F | D | E | M | – | – | W | | | | | | | | | | | | |
| LW R1,0(R3)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

3.12 a. See Figure S.12.

```
LD          F2,0(Rx)
DIVD        F8,F2,F0
MULTD       F2,F8,F2      ; reg renaming doesn't really help here, due to
                          ; true data dependencies on F8 and F2
LD          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
ADDD        F4,F0,F4
ADDD        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
SD          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         R20,R4,Rx
BNZ         R20,Loop
```

**Figure S.12 Instructions in code where register renaming improves performance.**

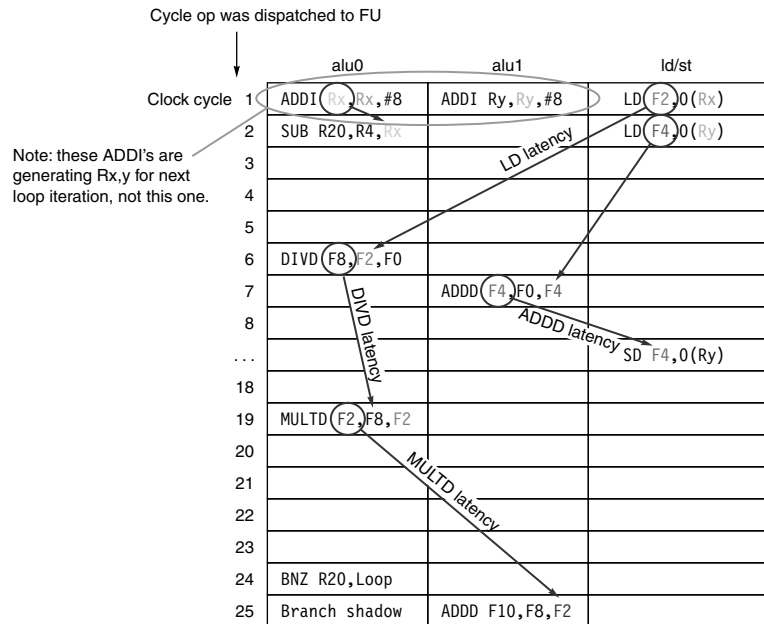b. See Figure S.13. The number of clock cycles taken by the code sequence is 25.



**Figure S.13** **Number of clock cycles taken by the code sequence.**

c. See Figures S.14 and S.15. 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.



**Figure S.14** **Candidates for dispatch.**
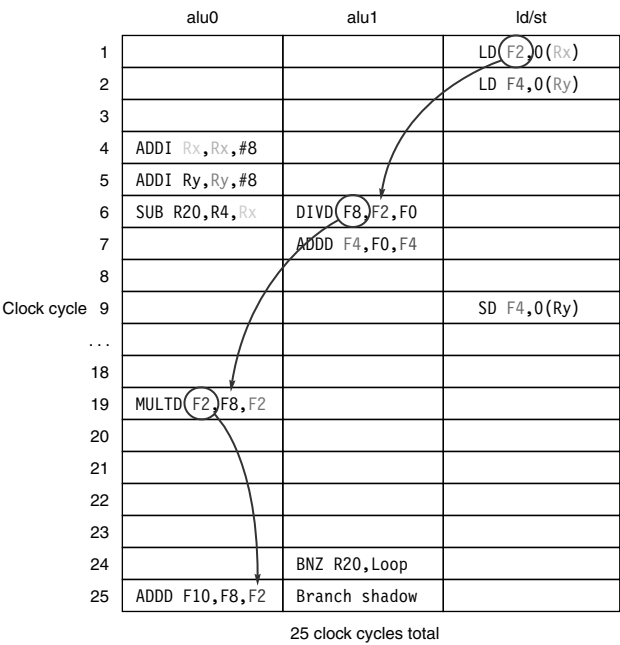
**Figure S.15  Number of clock cycles required.**
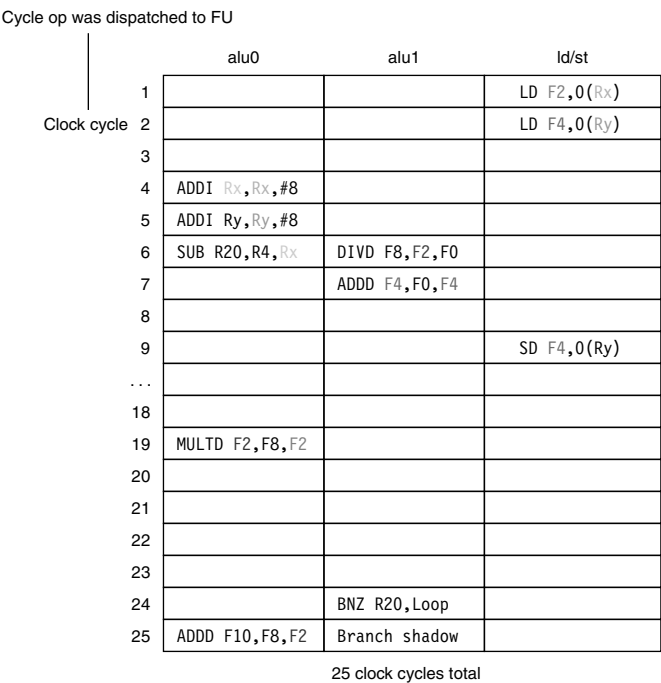
d.  See Figure S.16.



**Figure S.16 Speedup is (execution time without enhancement)/(execution time with enhancement) = 25/(25 − 6) = 1.316.**

1. Another ALU: 0% improvement

2. Another LD/ST unit: 0% improvement

3. Full bypassing: critical path is LD -> Div -> MULT -> ADDD. 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.17.

Cycle op was dispatched to FU

| | alu0 | alu1 | ld/st |
|---|---|---|---|
| 1 | | | LD F2,0(Rx) |
| 2 | | | LD F2,0(Rx) |
| 3 | | | LD F4,0(Ry) |
| 4 | ADDI Rx,Rx,#8 | | |
| 5 | ADDI Ry,Ry,#8 | | |
| 6 | SUB R20,R4,Rx | DIVD F8,F2,F0 | |
| 7 | | DIVD F8,F2,F0 | |
| 8 | | ADDD F4,F0,F4 | |
| 9 | | | |
| ... | | | SD F4,0(Ry) |
| 18 | | | |
| 19 | MULTD F2,F8,F2 | | |
| 20 | MULTD F2,F8,F2 | | |
| 21 | | | |
| 22 | | | |
| 23 | | | |
| 24 | | | |
| 25 | ADDD F10,F8,F2 | BNZ R20,Loop | |
| 26 | ADDD F10,F8,F2 | Branch shadow | |

Clock cycle is labeled to the left of cycle 2.

26 clock cycles total

**Figure S.17 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.

## Exercises

3.13   a.   See Figure S.18.

| Clock cycle | Unscheduled code | | Scheduled code | |
|:---:|---|---|---|---|
| 1 | DADDIU | R4,R1,#800 | DADDIU | R4,R1,#800 |
| 2 | L.D | F2,0(R1) | L.D | F2,0(R1) |
| 3 | stall | | L.D | F6,0(R2) |
| 4 | MUL.D | F4,F2,F0 | MUL.D | F4,F2,F0 |
| 5 | L.D | F6,0(R2) | DADDIU | R1,R1,#8 |
| 6 | stall | | DADDIU | R2,R2,#8 |
|  | stall | | DSLTU | R3,R1,R4 |
|  | stall | | stall | |
|  | stall | | stall | |
| 7 | ADD.D | F6,F4,F6 | ADD.D | F6,F4,F6 |
| 8 | stall | | stall | |
| 9 | stall | | stall | |
| 10 | stall | | BNEZ | R3,foo |
| 11 | S.D | F6,0(R2) | S.D | F6,-8(R2) |
| 12 | DADDIU | R1,R1,#8 | | |
| 13 | DADDIU | R2,R2,#8 | | |
| 14 | DSLTU | R3,R1,R4 | | |
| 15 | stall | | | |
| 16 | BNEZ | R3,foo | | |
| 17 | stall | | | |

**Figure S.18 The execution time per element for the unscheduled code is 16 clock cycles and for the scheduled code is 10 clock cycles.** This is 60% faster, so the clock must be 60% faster for the unscheduled code to match the performance of the scheduled code on the original hardware.

b.   See Figure S.19.

| Clock cycle | Scheduled code | |
|:---:|---|---|
| 1 | DADDIU | R4,R1,#800 |
| 2 | L.D | F2,0(R1) |
| 3 | L.D | F6,0(R2) |
| 4 | MUL.D | F4,F2,F0 |

**Figure S.19 The code must be unrolled three times to eliminate stalls after scheduling.**

| | | |
|---|---|---|
| 5 | L.D | F2,8(R1) |
| 6 | L.D | F10,8(R2) |
| 7 | MUL.D | F8,F2,F0 |
| 8 | L.D | F2,8(R1) |
| 9 | L.D | F14,8(R2) |
| 10 | MUL.D | F12,F2,F0 |
| 11 | ADD.D | F6,F4,F6 |
| 12 | DADDIU | R1,R1,#24 |
| 13 | ADD.D | F10,F8,F10 |
| 14 | DADDIU | R2,R2,#24 |
| 15 | DSLTU | R3,R1,R4 |
| 16 | ADD.D | F14,F12,F14 |
| 17 | S.D | F6,-24(R2) |
| 18 | S.D | F10,-16(R2) |
| 19 | BNEZ | R3,foo |
| 20 | S.D | F14,-8(R2) |

**Figure S.19** *Continued*

c. See Figures S.20 and S.21.

Unrolled 6 times:

| Cycle | Memory reference 1 | Memory reference 2 | FP operation 1 | FP operation 2 | Integer operation/branch |
|---|---|---|---|---|---|
| 1 | L.D F1,0(R1) | L.D F2,8(R1) | | | |
| 2 | L.D F3,16(R1) | L.D F4,24(R1) | | | |
| 3 | L.D F5,32(R1) | L.D F6,40(R1) | MUL.D F1,F1,F0 | MUL.D F2,F2,F0 | |
| 4 | L.D F7,0(R2) | L.D F8,8(R2) | MUL.D F3,F3,F0 | MUL.D F4,F4,F0 | |
| 5 | L.D F9,16(R2) | L.D F10,24(R2) | MUL.D F5,F5,F0 | MUL.D F6,F6,F0 | |
| 6 | L.D F11,32(R2) | L.D F12,40(R2) | | | |
| 7 | | | | | DADDIU R1,R1,48 |
| 8 | | | | | DADDIU R2,R2,48 |

**Figure S.20  15 cycles for 34 operations, yielding 2.67 issues per clock, with a VLIW efficiency of 34 operations for 75 slots = 45.3%.** This schedule requires 12 floating-point registers.

| | | | | |
|---|---|---|---|---|
| 9 | | | ADD.D F7,F7,F1 | ADD.D F8,F8,F2 | |
| 10 | | | ADD.D F9,F9,F3 | ADD.D F10,F10,F4 | |
| 11 | | | ADD.D F11,F11,F5 | ADD.D F12,F12,F6 | |
| 12 | | | | | DSLTU R3,R1,R4 |
| 13 | S.D F7,-48(R2) | S.D F8,-40(R2) | | | |
| 14 | S.D F9,-32(R2) | S.D F10,-24(R2) | | | |
| 15 | S.D F11,-16(R2) | S.D F12,-8(R2) | | | BNEZ R3,foo |

**Figure S.20** *Continued*

Unrolled 10 times:

| Cycle | Memory reference 1 | Memory reference 2 | FP operation 1 | FP operation 2 | Integer operation/branch |
|---|---|---|---|---|---|
| 1 | L.D F1,0(R1) | L.D F2,8(R1) | | | |
| 2 | L.D F3,16(R1) | L.D F4,24(R1) | | | |
| 3 | L.D F5,32(R1) | L.D F6,40(R1) | MUL.D F1,F1,F0 | MUL.D F2,F2,F0 | |
| 4 | L.D F7,48(R1) | L.D F8,56(R1) | MUL.D F3,F3,F0 | MUL.D F4,F4,F0 | |
| 5 | L.D F9,64(R1) | L.D F10,72(R1) | MUL.D F5,F5,F0 | MUL.D F6,F6,F0 | |
| 6 | L.D F11,0(R2) | L.D F12,8(R2) | MUL.D F7,F7,F0 | MUL.D F8,F8,F0 | |
| 7 | L.D F13,16(R2) | L.D F14,24(R2) | MUL.D F9,F9,F0 | MUL.D F10,F10,F0 | DADDIU R1,R1,48 |
| 8 | L.D F15,32(R2) | L.D F16,40(R2) | | | DADDIU R2,R2,48 |
| 9 | L.D F17,48(R2) | L.D F18,56(R2) | ADD.D F11,F11,F1 | ADD.D F12,F12,F2 | |
| 10 | L.D F19,64(R2) | L.D F20,72(R2) | ADD.D F13,F13,F3 | ADD.D F14,F14,F4 | |
| 11 | | | ADD.D F15,F15,F5 | ADD.D F16,F16,F6 | |
| 12 | | | ADD.D F17,F17,F7 | ADD.D F18,F18,F8 | DSLTU R3,R1,R4 |
| 13 | S.D F11,-80(R2) | S.D F12,-72(R2) | ADD.D F19,F19,F9 | ADD.D F20,F20,F10 | |
| 14 | S.D F13,-64(R2) | S.D F14,-56(R2) | | | |
| 15 | S.D F15,-48(R2) | S.D F16,-40(R2) | | | |
| 16 | S.D F17,-32(R2) | S.D F18,-24(R2) | | | |
| 17 | S.D F19,-16(R2) | S.D F20,-8(R2) | | | BNEZ R3,foo |

**Figure S.21  17 cycles for 54 operations, yielding 3.18 issues per clock, with a VLIW efficiency of 54 operations for 85 slots = 63.5%.** This schedule requires 20 floating-point registers.

3.14    a.    See Figure S.22.

| 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] |

**Figure S.22  Solution for exercise 3.14a.**

| | | | | | |
|---|---|---|---|---|---|
| 3 | S.D F6,0(R2) | 23 | 55 | | Wait for F6<br>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 |

**Figure S.22** *Continued*

b. See Figure S.23.

| Iteration | Instruction | Issues at | Executes/<br>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<br>Mult rs [2–4]<br>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<br>Add rs [3–20]<br>Add use [21] |
| 1 | S.D F6,0(R2) | 3 | 31 | | Wait for F6<br>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<br>INT rs [5–6] |
| 1 | BNEZ R3,foo | 5 | 7 | | INT busy<br>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<br>Mult rs [7–10]<br>Mult use [11] |
| 2 | L.D F6,0(R2) | 7 | 9 | 10 | INT busy<br>INT rs [8–9] |
| 2 | ADD.D F6,F4,F6 | 7 | 26 | 36 | Wait for F4<br>Add RS [8–26]<br>Add use [27] |
| 2 | S.D F6,0(R2) | 8 | 37 | | Wait for F6 |
| 2 | DADDIU R1,R1,#8 | 8 | 10 | 11 | INT busy<br>INT rs [8–10] |

**Figure S.23** **Solution for exercise 3.14b.**

| 2 | DADDIU R2,R2,#8 | 9 | 11 | 12 | INT busy<br>INT rs [10–11] |
|---|---|---|---|---|---|
| 2 | DSLTU R3,R1,R4 | 9 | 12 | 13 | INT busy<br>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<br>Mult rs [12–17]<br>Mult use [17] |
| 3 | L.D F6,0(R2) | 12 | 16 | 17 | INT busy<br>INT rs [13–16] |
| 3 | ADD.D F6,F4,F6 | 12 | 33 | 43 | Wait for F4<br>Add rs [13–33]<br>Add use [33] |
| 3 | S.D F6,0(R2) | 14 | 44 | | Wait for F6<br>INT rs full in 15 |
| 3 | DADDIU R1,R1,#8 | 15 | 17 | | INT rs full and busy<br>INT rs [17] |
| 3 | DADDIU R2,R2,#8 | 16 | 18 | | INT rs full and busy<br>INT rs [18] |
| 3 | DSLTU R3,R1,R4 | 20 | 21 | | INT rs full |
| 3 | BNEZ R3,foo | 21 | 22 | | INT rs full |

**Figure S.23** *Continued*

3.15 See Figure S.24.

| Instruction | Issues at | Executes/Memory | Write CDB at |
|---|---|---|---|
| ADD.D F2,F4,F6 | 1 | 2 | 12 |
| ADD R1,R1,R2 | 2 | 3 | 4 |
| ADD R1,R1,R2 | 3 | 5 | 6 |
| ADD R1,R1,R2 | 4 | 7 | 8 |
| ADD R1,R1,R2 | 5 | 9 | 10 |
| ADD R1,R1,R2 | 6 | 11 | 12 (CDB conflict) |

**Figure S.24** Solution for exercise 3.15.

3.16    See Figures S.25 and S.26.

Correlating Predictor

| Branch PC mod 4 | Entry | Prediction | Outcome | Mispredict? | Table Update |
|---|---|---|---|---|---|
| 2 | 4 | T | T | no | none |
| 3 | 6 | NT | NT | no | change to "NT" |
| 1 | 2 | NT | NT | no | none |
| 3 | 7 | NT | NT | no | none |
| 1 | 3 | T | NT | yes | change to "T with one misprediction" |
| 2 | 4 | T | T | no | none |
| 1 | 3 | T | NT | yes | change to "NT" |
| 2 | 4 | T | T | no | none |
| 3 | 7 | NT | T | yes | change to "NT with one misprediction" |

**Figure S.25  Individual branch outcomes, in order of execution.  Misprediction rate = 3/9 = .33.**

Local Predictor

| Branch PC mod 2 | Entry | Prediction | Outcome | Mispredict? | Table Update |
|---|---|---|---|---|---|
| 0 | 0 | T | T | no | change to "T" |
| 1 | 4 | T | NT | yes | change to "T with one misprediction" |
| 1 | 1 | NT | NT | no | none |
| 1 | 3 | T | NT | yes | change to "T with one misprediction" |
| 1 | 3 | T | NT | yes | change to "NT" |
| 0 | 0 | T | T | no | none |
| 1 | 3 | NT | NT | no | none |
| 0 | 0 | T | T | no | none |
| 1 | 5 | T | T | no | change to "T" |

**Figure S.26  Individual branch outcomes, in order of execution.  Misprediction rate = 3/9 = .33.**

3.17 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:

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

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

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

| BTB Result | BTB Prediction | Frequency (Per Instruction) | Penalty (Cycles) |
|---|---|---|---|
| Miss | | $15\% \times 10\% = 1.5\%$ | 3 |
| Hit | Correct | $15\% \times 90\% \times 90\% = 12.1\%$ | 0 |
| Hit | Incorrect | $15\% \times 90\% \times 10\% = 1.3\%$ | 4 |

**Figure S.27 Weighted penalties for possible branch outcomes.**

Therefore:

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

$$Speedup = \frac{1.0 + 0.30}{1.0 + 0.097} = 1.2$$

3.18 a. Storing the target instruction of an unconditional branch effectively removes one instruction. If there is a BTB hit in instruction fetch and the target instruction is available, then that instruction is fed into decode in place of the branch instruction. The penalty is –1 cycle. In other words, it is a performance gain of 1 cycle.

b. If the BTB stores only the target address of an unconditional branch, fetch has to retrieve the new instruction. This gives us a CPI term of $5\% \times (90\% \times 0 + 10\% \times 2)$ of 0.01. The term represents the CPI for unconditional branches (weighted by their frequency of 5%). If the BTB stores the target instruction instead, the CPI term becomes $5\% \times (90\% \times (–1) + 10\% \times 2)$ or –0.035. The negative sign denotes that it reduces the overall CPI value. The hit percentage to just break even is simply 20%.