Chapter 5 Solutions: For More Practice

5.4 Fetching, reading registers, and writing the destination register takes a total of 300ps for both floating point add/subtract and multiply/divide. Thus, floating point add/subtract takes 300ps + 400ps = 700ps, and floating point multiply/divide takes 300ps + 600ps = 900ps.

5.5 The single-cycle critical path is floating point multiply/divide, so the cycle time is 900ps.

5.6 The weighted average of the cycle times for the different instructions types is:

$(600\text{ps} \times 0.3) + (550\text{ps} \times 0.15) + (400\text{ps} \times 0.25) + (350\text{ps} \times 0.10) + (200\text{ps} \times 0.05) + (700\text{ps} \times 0.05) + (900\text{ps} \times 0.10) = 532.5\text{ps}$.

5.15 If Regdst = 0, all R-format instructions will not work properly because we will specify the wrong register to write. If ALUSrc = 0, then all I-format instructions except branch will not work because we will not be able to get the sign-extended 16 bits into the ALU. If MemtoReg = 0, then loads will not work. If Zero = 0, the branch instruction will never branch, even when it should.

5.16 If Regdst = 1, loads will not work (bad destination). If ALUSrc = 1, then all R-format instructions as well as branch will not work properly because the second register read will not get into the ALU. If MemtoReg = 1, then all R-format instructions will not write the correct data into the register file. If Zero = 1, all branches will be taken all of the time, even if they shouldn't be.

5.17 If RegDst = 0, all R-format instructions will be unable to write into the proper register (rd). If MemtoReg = 0 or IorD = 0, then loads will not work. If ALUSrcA = 0, none of the instructions will work properly because they all require A to be operated on by the ALU at some point.

5.18 If RegDst = 1, load instructions will not work. If MemtoReg = 1, all R-format instructions will not work. If IorD = 1, no instructions will work because the PC will never be used to get an instruction. If ALUSrcA = 1, the program counter will not be properly incremented by 4 and essentially all instructions will not work.

5.19 No additions to the datapath are required. A new row should be added to the truth table in Figure 5.18. The new control is similar to load word because we want to use the ALU to add the immediate to a register (and thus RegDst = 0, ALUSrc = 1, ALUOp = 00). The new control is also similar to an R-format instruction because we want to write the result of the ALU into a register (and thus MemtoReg = 0, RegWrite = 1) and of course we aren't branching or using memory (Branch = 0, MemRead = 0, MemWrite = 0).

5.20 We already have a way to change the PC based on the specified address (using the datapath for the jump instruction), but we'll need a way to put PC + 4 into register `$ra` (31), and this will require changing the datapath. We can expand the multiplexor controlled by RegDst to include 31 as a new input. We can expand the multiplexor controlled by MemToReg to have PC + 4 as an input. Because we expand these multiplexors, the entire columns for RegDst and MemtoReg must change appropriately in the truth table. The `jal` instruction doesn't use the ALU, so ALUSrc and ALUOp can be don't cares. We'll have Jump = 1, RegWrite = 1, and we aren't branching or using memory (Branch = 0, MemRead = 0, MemWrite = 0).

5.21 One possible solution is to add a new control signal called "Invzero" that selects whether Zero or inverted Zero is an input to the AND gate used for choosing what the new PC should be (thus a new multiplexor is introduced). The new control signal Invzero would be a don't care whenever the control signal Branch is zero. Many other solutions are possible.

Note to instructors: Sometimes different solutions submitted by students will in fact be identical (e.g., one student draws a multiplexor, the other draws gates, but in fact they both implemented it the same way—just at different levels of abstraction). You may want to look for solutions that you can use to emphasize this point to your students. For example, an alternative solution would be to use Zero as an input to a new multiplexor that selects between Branch and a new control signal Bnequal. In this case, there would be no don't cares because if we aren't branching, both Branch and Bnequal need to be zero. Some students may present this exact solution in gate form instead of in mux form.

5.22 No changes are needed in the datapath. The new variant is just like `lw` except that the ALU will use the Read data 2 input instead of the sign-extended immediate. Of course the instruction format will need to change: the register write will need to be specified by the rd field instead of the rt field. However, all needed paths are already in place for the sake of R-format instructions. To modify the control, we simply need to add a new row to the existing truth table. For the new instruction, RegDst = 1, ALUSrc = 0, MemtoReg = 1, RegWrite = 1, MemtoReg = 1, MemWrite = 0, ALUop = 00.

5.23 There are no temporary registers that can be made to hold intermediate (temporary) data. There are no intermediate edges to clock them. There is no state machine to keep track of whether rs has been updated or both rs and rt have been updated. In essence, we can't order things.

5.24 No solution provided.

5.25 No solution provided.

5.26 The key is recognizing that we no longer have to go through the ALU and then to memory. We would not want to add zero using the ALU; instead we want to provide a path directly from the Read data 1 output of the register file to the read/write address lines of the memory (assuming the instruction format does not change). The output of the ALU would no longer connect to memory. The control does not need to change, but some of the control signals now are don't cares.

5.40 No solution provided.

5.41 The existing datapath is insufficient. As described in the solution to Exercise 5.6, we'll need to expand the two multiplexors controlled by RegDst and MemtoReg. The execution steps would be

Instruction fetch (unchanged)

Instruction decode and register fetch (unchanged)

`jal: Reg[31] = PC; PC = PC [31-28] || (IR[25-0]<<2);`

Note that in this case we are writing the PC after it has already been incremented by 4 (in the Instruction fetch step) into register `$ra` (thus the datapath requires that PC be an input to the MemtoReg multiplexor and 31 needs to be an input to the RegDst multiplexor). We need to modify existing states to show proper values of RegDst and MemtoReg and add a new state that performs the `jal` instruction (and then returns to state 0) via PCWrite, PCSource = 10, RegWrite, and appropriate values for MemtoReg and RegDst.

5.42 The solution is dependent on the instruction format chosen, and there are many implementation options. One approach for the format of the `swap` instruction might be to name in both the rs and rd fields one of the two registers being swapped. The other register would be named in rt. The outputs of A and B in Figure 5.28 should both be directed to the multiplexor controlled by MemtoReg. The MemtoReg control signal now becomes 2 bits (e.g., `00 = write ALUout, 01 = write MDR, 10 = write A, 11 = write B`). Because the existing multicycle control of the datapath causes A to be written on every clock cycle (see page 326, step 2), we should add a new state to those in Figure 5.42 for clock cycle 3 of the `swap` instruction that accomplishes `Reg[IR[20-16]] = A`, that is `rt = rs`, and add another new state for clock cycle 4 of the `swap` instruction that accomplishes `Reg[IR[15-11]] = B`, that is `rd = rt`. Because we wrote the `swap` instruction with rs and rd referring to the same register, the step in cycle 4 does `rs = rt`, completing the `swap`. In the new state for cycle 3, we need control signals set as RegDst = 1, RegWrite, MemtoReg = 11, and in the new state for cycle 4, we need RegDst = 0, RegWrite, MemtoReg = 10. In Figure 5.38, state 4 should be revised to have MemtoReg = 01 and state 7 should have MemtoReg = 00.

5.43 There are a variety of implementation options. In state 0 the PC is incremented by 4, so we must subtract 4 from the PC and put this value in a register. This requires two new states (10 and 11). There would be an arrow from state 1 to state 10 labeled Op = 'wai', and there should be unconditional arrows from state 10 to state 11 and from state 11 to state 0. State 10 accomplishes `ALUout = PC - 4 (ALUSrcA=0, ALUSrcB=01, and ALUOp=01)` and state 11 accomplishes `Reg[IR[20-16]]=ALUout (RegDst=0, MemtoReg=0, RegWrite)`. An alternative solution relies on the idea that `wai` could be handled like a branch instruction. If a –1 occupied the 16-bit immediate value field of a `wai` instruction, then calculation of the branch target address actually computes the PC of the `wai` instruction itself. This clever solution reduces the number of needed new control states by one.

5.44 The jump memory instruction behaves much like a load word until the memory is read. The data coming out of memory needs to be deposited into the PC. This will require a new input to the multiplexor controlled by PCSource. We add a new state coming off of state 3 that checks for Op = 'jump memory' (and modify the transition to state 4 to ensure Op = 'lw'). The new state has PCWrite, PCSource = 11. The transitions to states 2 and 3 need to also include Op = 'jump memory'.

5.45 The instruction will require using the ALU twice. A multiplexor should be added to permit the ALU to use the previous result produced by the ALU (stored in ALUOut) as an input for the second addition. Note that changes will also be needed for register read. The instruction can be implemented in 5 cycles by reading the third register source during the cycle in which the first addition is taking place. Clearly a multiplexor and a new control signal will be needed for the inputs to one of the register read ports.

5.46 The rt field of the jump register instruction contains 0. So, along with rs we read \$0. To get rs to the output of the ALU, we can perform a dummy addition of rs + \$0. We already have a path from ALUOut to the PC. We can use this path for loading the PC with rs. The finite state machine will need a new state for when the opcode is 000000 and the function code is 001000. The current condition for entering state 6 needs to be modified to make sure the function code is not 001000. In the new state, we want the appropriate combination of states 0 and 6: ALUSrcA = 1, ALUSrcB = 00, PCWrite, PCSource = 00, and of course we make sure an addition is performed with ALUOp = 00.

5.47 The instruction mix is

Instruction class	Frequency
Loads	26%
Stores	10%
R-type	49%
Jump/branch	15%

The CPIs for each machine by instruction class are

Instruction class	M1 CPIs	M2 CPIs	M3 CPIs
Loads	5	4	3
Stores	4	4	3
R-type	4	3	3
Jump/branch	3	3	3
Effective CPI	4.11	3.36	3

$$\text{MIPS}_{M1} = \frac{500}{4.11} = 121.7$$

$$\text{MIPS}_{M2} = \frac{400}{3.36} = 119.0$$

$$\text{MIPS}_{M3} = \frac{250}{3.0} = 83.3$$

So the original machine M1 is fastest for this mix. M3 would be the fastest for an instruction mix that contained only loads.

5.48 Here is a possible code sequence:

```
      beq  $t3, $zero, done
move: lw   $t4, 0($t1)
      sw   $t4, 0($t2)
      addi $t1, $t1, 4
      addi $t2, $t2, 4
      addi $t3, $t3, -1
      bne  $t3, $zero, move
done:   ...
```

The number of cycles required is determined by computing the number of instructions required. To copy 100 words we'll perform 100 loads, 100 stores, 300 adds, and 101 branches. We then use the number of cycles for each instruction to arrive at 500 cycles for the loads, 400 cycles for the stores, 1200 cycles for the adds, and 303 cycles for the branches. Grand total is 2403 cycles.

5.52 No solution provided.

5.54 No solution provided.

5.55 We need to change Dispatch 1 and Dispatch 2 and introduce a new entry (perhaps inserted after SW2) with a label of "ADDI2" and a register control of "Write ALU rt," which is similar to "Write ALU" except that it uses the rt field instead of rd. You could argue that further changes should be made to existing labels to make things easier to understand.

5.56 No solution provided.