SOLUTIONS for PRACTICE PROBLEMS

4) 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) 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).

6) 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.

7) 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.

8) We use the same datapath, so the immediate field shift will be done inside the ALU.
1. Instruction fetch step: This is the same (IR <= Memory[PC]; PC <= PC + 4)

2. Instruction decode step: We don't really need to read any register in this stage if we know that the instruction in hand is a lui, but we will not know this before the end of this cycle. It is tempting to read the immediate field into the ALU to start shifting next cycle, but we don't yet know what the instruction is. So we have to perform the same way as the standard machine does.
A <= 0 ($r0); B <= $rt; ALUOut <= PC + (sign-extend(immediate field));

3. Execution: Only now we know that we have a lui. We have to use the ALU to shift left the low-order 16 bits of input 2 of the multiplexor. (The sign extension is useless, and sign bits will be flushed out during the shift process.)
ALUOut <= {IR[15-0],16(0)}

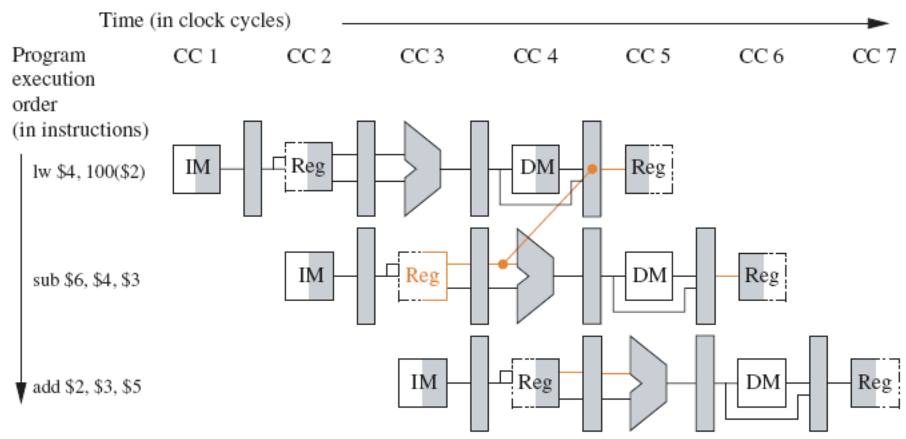
4. Instruction completion: Reg[IR[20-16]] = ALUOut.

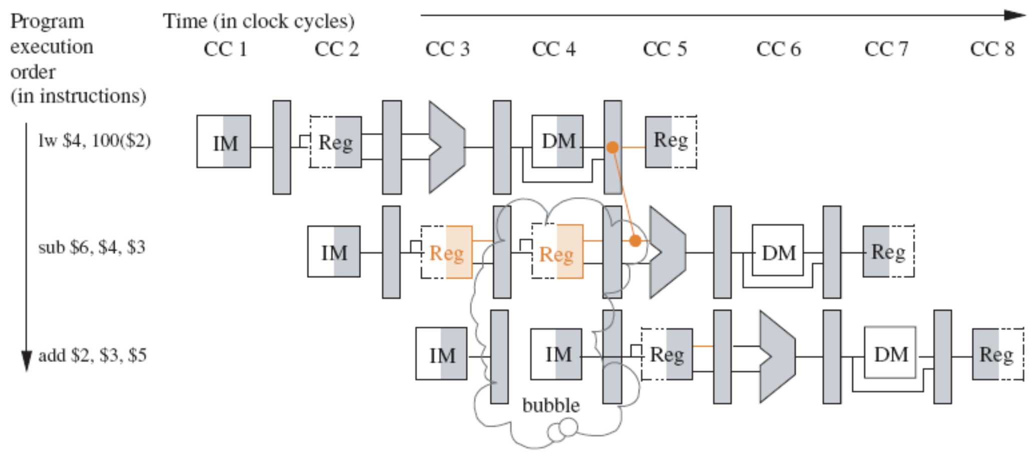
The first two cycles are identical to the FSM of Figure 5.38. By the end of the second cycle the FSM will recognize the opcode. We add the Op='lui', a new transition condition from state 1 to a new state 10. In this state we perform the left shifting of the immediate field: ALUSrcA = x, ALUSrcB = 10, ALUOp = 11 (assume this means left shift of ALUSrcB). State 10 corresponds to cycle 3. Cycle 4 will be translated into a new state 11, in which RegDst = 0, RegWrite, MemtoReg = 0. State 11 will make the transition back to state 0 after completion.

As shown above the instruction execution takes 4 cycles.

9) There is a data dependency through $3 between the first instruction and each subsequent instruction. There is a data dependency through $6 between the lw instruction and the last instruction. For a five-stage pipeline as shown in Figure 6.7, the data dependencies between the first instruction and each subsequent instruction can be resolved by using forwarding. The data dependency between the load and the last add instruction cannot be resolved by using forwarding.

10) It will take 8 cycles to execute this code, including a bubble of 1 cycle due to the dependency between the lw and sub instructions.


Time (in clock cycles)
Program execution order (in instructions)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
lw $4, 100($2)
sub $6, $4, $3
add $2, $3, $5
IM
Reg
DM
Reg


Program execution order (in instructions)
Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
lw $4, 100($2)
sub $6, $4, $3
add $2, $3, $5
IM
Reg
DM
Reg
bubble