5.5 No additions to the datapath are required. A new row should be added to the truth table in Figure 5.20. 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).

5.14 The key is to understand that it is the length of the longest path in the combinational logic that is determining the cycle time. Essentially, we compute the length of the longest path for each instruction and then must take the one with maximum value. At present, the `lw` instruction is providing the longest path of length 8 ns.

a. The changes do not increase any paths beyond current maximums. Thus the cycle time is still 8 ns.

b. Consider the `beq` instruction. We now have a path which requires 10 ns to compute the branch address (X + Y = 10 ns), and this is maximum. Thus, the cycle time increases to 10 ns.

c. You might be tempted to again conclude that the branch instruction will require X + Y = 9 ns for branch address computation. This is not quite correct, however, because the second adder (requiring Y time) has two inputs, one of which is not available until after the instruction is read (the 16 immediate bits), and this takes 2 ns. Thus, the actual maximum length path is again 10 ns, and the cycle time is 10 ns.

5.15 The existing datapath is sufficient. The execution steps for `addi` would be

Instruction fetch (unchanged)

Instruction decode and register fetch (unchanged)

Execution: `ALUOut = A + sign-extend (IR[15-0]);`

Instruction Completion: `Reg[IR[20-16]] = ALUOut`

The first three steps are identical to those performed for memory access instructions, so we can use them (we must add Op = 'addi' to the transition from state 1 to state 2). The fourth step is different and not like any existing step; thus we must add a new state to our finite state machine (i.e., state 10), a transition from state 2 to state 10 if Op='addi', and a transition from state 10 back to state 0. In state 10 we have RegDst = 0, RegWrite, MemtoReg = 0.

5.22 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.

5.23 Many solutions are possible. In all of them, a multiplexor will be needed as well as a new control signal (e.g., RegRead) to select which register is going to be read (i.e., using `IR[25-11]` or `IR[20-16]`). One simple solution is simply to add a write signal to A and break up state 1 into two states, in which A and B are read. It is possible to avoid adding the write signal to A if B is read first. Then A is read and RegRead is held stable (because A always writes). Alternatively, you could decide to read A first because it may be needed to calculate an address. You could then postpone reading B until state 2 and avoid adding an extra cycle for the load and store instructions. An extra cycle would be needed for the branch and R-type instructions.