Design of a Simple Pipeline

Objective

To design and implement a simple pipelined system (other than CPU).

It is important to obtain a deep understanding of the basic concepts of pipelining such as data-stationary control, forwarding, and stalling. Since the textbook has presented a completed design of the pipelined CPU, it does not provide an opportunity for students to arrive at the basic design of a new pipelined system by themselves. It is hoped that this lab provides such an opportunity.

Introduction

In this lab the operations to be performed (the instructions to be executed) are very simple. Using a pipelined system, a series of such simple instructions are to be executed very much like in a CPU. We need to take care of data dependencies by designing appropriate forwarding unit (FU) and hazard detection and stalling unit (HDU).

The design has 5 pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execution Stage 1 (EX1), Execution Stage 2 (EX2), and Write Back Stage (WB). In EX1 we have a subtractor to subtract a constant of 3 (given A, it produces A - 3). In EX2, we have an adder to add a constant of 4 (given A, it produces A + 4). Using these resources, we can perform a MOV or a SUB3 or a ADD4 or even a ADD1. The ADD1 operation requires performing both operations, SUB3 and ADD4. The MOV operation requires neither operation.

Datapath

As stated in the introduction, we can perform four operations (besides a NOP) namely MOV, SUB3, ADD4, and ADD1. Figure 1 serves as a block diagram for this lab. The subtract_3 operation in EX1 is implemented by actually tying a MINUS_THREE constant (1111_1111_1111_1101) to one of the two inputs of a regular adder. Similarly the add_4 operation in EX2 is implemented by actually tying a PLUS_FOUR constant (0000_0000_0000_0100) to one of the two inputs of a regular adder. The ADD1 operation is performed by subtracting 3 in EX1 and then adding 4 in EX2. The MOV operation is performed by neither subtracting 3 nor adding 4 to the given data. Since the SUB3 instruction does not require any processing by the EX2 stage and similarly the ADD4 instruction does not require any processing by the EX1 stage, (and further since the MOV instruction does not require any processing by either EX1 or EX2 stages), we have two result-selection muxes, (R1-Mux and R2-Mux) at the output of the adders in EX1 and EX2 stages to bypass the subtract_3 and/or add_4 operations, as needed. Here the register file has one read port and one separate write port. It is an internally-forwarding register file. Actually we are providing the register file from lab 7 Part 1 for this part also. It has three read ports and one write port. You can ignore two out of the three read ports.

Instruction format: The most significant 4 bits [31:28] are used for opcode which is one-hot coded. The result register address (RA) is specified by the 4 bits [7:4] and the source register address (XA) is specified by the 4 bits [3:0]. Rest of the 20 bits [27:8] are all zeros.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Opcode</th>
<th>MSD</th>
<th>32-bit instruction in hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td></td>
<td>MOV</td>
<td>SUB3</td>
<td>ADD4</td>
</tr>
<tr>
<td>MOV $R, $X;</td>
<td>($R) &lt;= ($X)</td>
<td>0 0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>SUB3 $R, $X;</td>
<td>($R) &lt;= ($X) - 3</td>
<td>0 1 0</td>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>ADD4 $R, $X;</td>
<td>($R) &lt;= ($X) + 4</td>
<td>0 0 1</td>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>ADD1 $R, $X;</td>
<td>($R) &lt;= ($X) + 1</td>
<td>0 0 0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

© Copyright 2011 Gandhi Puvvada
1. Complete all missing connections to the Reg. File. Also complete the RA (Result Addrees) connection in ID stage (ID_RA).
2. Complete all five enable (EN) controls on the pipeline registers (including PC).
3. Complete the skip controls (SKIP1, SKIP2).
4. Draw the logic for the HDU, FU1, and FU2, producing STALL, PRIORITY, FORW1, FORW2.

LAB 7 Part 3 Block Diagram

Fig. 1
Initial contents of the Register File

<table>
<thead>
<tr>
<th>Register id</th>
<th>id in hex</th>
<th>Content in hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register 00</td>
<td>0</td>
<td>0001h &lt;= Unlike MIPS, the register 0 is like any other register here</td>
</tr>
<tr>
<td>Register 01</td>
<td>1</td>
<td>0002h</td>
</tr>
<tr>
<td>Register 02</td>
<td>2</td>
<td>0004h</td>
</tr>
<tr>
<td>Register 03</td>
<td>3</td>
<td>0008h</td>
</tr>
<tr>
<td>Register 04</td>
<td>4</td>
<td>0010h</td>
</tr>
<tr>
<td>Register 05</td>
<td>5</td>
<td>0020h</td>
</tr>
<tr>
<td>Register 06</td>
<td>6</td>
<td>0040h</td>
</tr>
<tr>
<td>Register 07</td>
<td>7</td>
<td>0080h</td>
</tr>
<tr>
<td>Register 08</td>
<td>8</td>
<td>0100h</td>
</tr>
<tr>
<td>Register 09</td>
<td>9</td>
<td>0200h</td>
</tr>
<tr>
<td>Register 10</td>
<td>A</td>
<td>0400h</td>
</tr>
<tr>
<td>Register 11</td>
<td>B</td>
<td>0800h</td>
</tr>
<tr>
<td>Register 12</td>
<td>C</td>
<td>1000h</td>
</tr>
<tr>
<td>Register 13</td>
<td>D</td>
<td>2000h</td>
</tr>
<tr>
<td>Register 14</td>
<td>E</td>
<td>FFF8h</td>
</tr>
<tr>
<td>Register 15</td>
<td>F</td>
<td>FFFFh</td>
</tr>
</tbody>
</table>

Control:

Please consider the following questions before designing and implementing your control. You need to think who can wait for forwarding data latest until when and who can provide forwarding data earliest by when. May be MOV is in some aspects like SUB3 because it is capable of helping subsequent instruction(s) when it is in EX2. May be MOV is in some aspects like ADD4 instruction because it can postpone receiving help until it reaches EX2.

Note: Separate pages are provided at the end of this assignment for submitting your answers to these questions.

Question 1 Can an instruction postpone receiving forwarding data until it reaches EX2 stage? If an instruction can postpone receiving forwarding help until reaching EX2, would it still try to receive help while it is in EX1 (may be because the donor instruction can not wait)?

Question 2 Are there occasions where you end up stalling an instruction because you could not provide the needed forwarding data to it in EX1?

Question 3 Can an instruction in EX2 provide forwarding help to an instruction in EX1? Or is it too early for any instruction to start providing forwarding help while it is in EX2?

Question 4 Which instructions need to wait until they reach WB stage for them to provide forwarding help? And why they need to wait until then?

Question 5 Priority in Forwarding: Recall that, in the MIPS pipelined CPU design, if the instructions in both MEM stage and also in WB stage are willing to provide forwarding help to the instruction in EX stage, we exercise priority and accept help from (MEM/WB) stage. Do we have such a situation here? If so, explain with an example instruction sequence.

Question 6 Normally forwarding is done at the beginning of a clock so that the recipient instruction can process the information during the clock. However, sometimes, it may make sense (as in Lab 7 Part 1) to forward information at the end of the clock, if the receiving instruction is not performing any operation on the data received. Can an instruction such as ADD4 or ADD1 in EX2 provide forwarding help to an ADD4 or MOV instruction in EX1 towards the end of the clock? If yes, did you provide such an arrangement in your design? If not, is it desirable to provide such an arrangement? Does it cost extra? Does it avoid any stalls, thereby improving the pipeline performance? Or is it that the particular help we plan to offer at the end of the clock will anyway be available at the beginning of the next clock and it is just one and the same (one and the same, whether you provide data at the end of the current clock or at the beginning of the next clock)? Explain briefly.

Question 7 Also answer question #3 from Spring 2002 Final Exam (slightly modified for this lab) included later in this assignment. Answer this question before you attempt the second part of this lab (ee457_lab7_P3_EX1_EX2_merged).

Note: The original Spring 2002 question did not have the MOV instruction. It has only SUB3, ADD4, ADD1, and of course the NOP. Now we modified the question to include the MOV instruction also.
Fill-in the following to understand stalling needs and forwarding opportunities

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Receiving forwarding help</th>
<th>Providing forwarding help</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insists on receiving in EX1</td>
<td>Doesn’t mind receiving in EX1</td>
</tr>
<tr>
<td>MOV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Based on the above, if an instruction is dependent on a senior instruction which is not just above (just above = just before), there is never a need to stall the dependent instruction. True / False

If the dependent instruction is either __________ or __________, then it needs help at the beginning of the clock when it is in EX1 as it needs to process the data using SUB3 in EX1. And if the senior instruction (donor instruction) is just above it (just before it) in EX2 stage, and if it is either __________ or __________, then it can't help at the beginning of the clock, as it is still producing the data using the ADD4 in EX2. Hence this dependency hazard should be detected when the dependent instruction is in the ID stage and should be stalled. The stall is for ________________ (just 1 clock, minimum for 1 clock).

__________ (Unlike/Like) the MIPS 5 stage pipeline, where the instructions ______________ (have only one source register / can have two source registers), here the instructions ______________ (have only one source register / can have two source registers). Hence it ______________ (is / isn’t) possible to stall the dependent instruction in EX1 stage instead of the ID stage.

Draw logic to go into HDU and FU2

```
HDU

STALL

FU2

FORW2
```
Draw logic to go into FU1 as per the diagram on page 2

Redesign the logic to go into FU1 if the arrangement of the forwarding muxes is changed as shown

Any advantage of the NEW_FU1 over the original FU? Note: If logic is reduced, then it is cheaper and faster!
Instruction Stream in the TestBench

It is important to go through and understand the instruction stream (given in the testbench) to be run on your CPU design before you start debugging. You need to know what behavior you are expecting. Please go through ee457_lab7_P3_tb.v after you download files for the first subpart of this lab. The instructions loaded in the instruction memory are specified there and they are fully explained. The instruction stream is common to all four subparts.

What you have to do

1. There are four subparts to this Lab#7 Part3. The four subparts are:
   
<table>
<thead>
<tr>
<th>Subpart</th>
<th>File Name Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lab 7 Part 3 subpart#1</td>
<td>ee457_lab7_P3</td>
</tr>
<tr>
<td>Lab 7 Part 3 subpart#2</td>
<td>ee457_lab7_P3_with_EX1_EX2_merged</td>
</tr>
<tr>
<td>Lab 7 Part 3 subpart#3</td>
<td>ee457_lab7_P3_RTL_Coding_Style</td>
</tr>
<tr>
<td>Lab 7 Part 3 subpart#4</td>
<td>ee457_lab7_P3_RTL_Coding_Style_with_EX1_EX2_merged</td>
</tr>
</tbody>
</table>

2. First we do Lab7 Part 3 subpart#1.

3. Create a folder C:\ModelSim_projects\ee457_lab7_P3 (under your C:\ModelSim_projects). Download the .zip file provided (ee457_lab7_P3.zip) and extract the .v and .do files and place them in the above directory.

4. Create a modelsim project with the project name ee457_lab7_P3. Choose ee457_lab7_P3 for the project directory.

5. Add all verilog files to the project.

6. Edit (in Notepad++) and complete the ee457_lab7_P3.v.

7. Go through ee457_lab7_P3_tb.v and understand the instruction stream used for testing. Compile all Verilog files.

8. Start simulation by selecting ee457_lab7_P3_tb.v. Unselect "Enable optimization".

9. Use the given .do file "ee457_lab7_P3_wave.do" to set up the waveform (command: do ee457_lab7_P3_wave.do).

10. Select the Memories tab in the workspace and double click on the reg_file to display its contents in the right pane.

11. Initially the data content of the memory is displayed as xxxx. You can simulate for a very short time (say 1ns) (run 1ns) to display the actual initial contents. Steps 7-11 can be performed by using the ee457_lab7_P3.do file (do ee457_lab7_P3.do).

12. Run the simulation for 1199ns more (run 1199ns) (total 1200ns).

13. Verify the final register contents. Look at the waveform to see if any signals are misbehaving. Look at the TimeSpace.txt file produced and placed in the project directory. Use Notepad++ to look at this file as wordpad refuses to open this file as it is still being controlled/updated by ModelSim. (On Windows-XP, I could open the TimeSpace.txt file in Notepad++ while simulation is not yet done. If Windows-7 does not allow you to open the file while simulation is going on, please end the simulation, inspect the file and restart the simulation again.) Perform incremental simulation to find errors. Use restart -f to start the simulation again from 0ns. Now run for short lengths of time examining the register file contents, the waveforms, and the TimeSpace.txt.
14. After finishing all debugging, compile the revised .v file, again restart (restart -f), run for 1ns, examine register contents using the command-line command at VSIM> prompt: "examine -radix hex UUT/REG_FILE/reg_file". Copy the response in the transcript window into "RF_Content_Lab7_P3.txt". Further run simulation for 1199ns more (run 1199ns) and again examine register contents using command "examine -radix hex UUT/REG_FILE/reg_file". Copy the response in the transcript window again into "RF_Content_Lab7_P3.txt".

15. Now we do Lab 7 Part 3 subpart#2 (ee457_lab7_P3_with_EX1_EX2_merged)

16. Finish answering Question 7.3 at the end of this assignment and complete the diagram on page 14/14.

17. Download ee457_lab7_P3_with_EX1_EX2_merged.zip, extract files, and place the .v and .do files in C:\ModelSim_projects\ee457_lab7_P3_with_EX1_EX2_merged. Create a ModelSim project with the name ee457_lab7_P3_with_EX1_EX2_merged. Add the verilog files to it. Note: The file, ee457_lab7_components.v is identical to the file with the same name provided in previous subpart. And the file ee457_lab7_P3_tb.v is nearly identical to the testbench file of the previous subpart.

18. Complete ee457_lab7_P3_with_EX1_EX2_merged.v file.

19. Compile all verilog files.

20. Start simulation of the module ee457_lab7_P3_tb. Unselect "Enable optimization".

21. Run the do file ee457_lab7_P3_with_EX1_EX2_merged_wave.do to setup the waveform.

22. Run for 1ns, examine register contents using command "examine -radix hex UUT/REG_FILE/reg_file" Steps 19-22 can be performed by using the ee457_lab7_P3_with_EX1_EX2_merged.do file (do file ee457_lab7_P3_with_EX1_EX2_merged.do).

23. Now run for 1199ns more and again examine the register contents and the waveforms. Debug.

24. Copy the two sets of register file contents into file "RF_Content_Lab7_P3_EX1_EX2_merged.txt".

25. For Lab7 Part 3 subpart#3 (ee457_lab7_P3_RTL_Coding_Style) and Lab7 Part 3 subpart#4 (ee457_lab7_P3_RTL_Coding_Style_with_EX1_EX2_merged), please see a separate handout.

Sub part 3 is a demonstration of the sub part 1 in RTL coding style. Sub part 3 is an exercise asking you to recode the earlier sub part #2 in RTL coding style. The coding in sub parts 1 and 2 is structural coding using too many small components. This is not the desirable way of coding. Besides showing RTL coding style, these two sub parts try to place emphasis on appropriate use of Blocking and Non-Blocking assignments in RTL coding, paying attention to the fact that you need to produce an intermediate variable before you attempt to consume it, and that the combinational paths in a pipeline may go across stages. These sub parts also show that an intermediate signal (in the NSL combinational logic) generated in a clocked procedural block should not be accessed outside that procedural block either for viewing in waveform or for actual logic coding.

26. General Guidelines

27. Start early and seek help early if needed.

28. Finally submit online (through your unix account) (submission commands specified separately) one set of files for a team of two students:

29. You need to use the file names exactly as stated and follow the submission procedure exactly as specified. We use Unix script files to automate grading.

30. Non-working lab submission: In simulation, it will be evident if your lab is not working. We discourage you from submitting a non-working lab. If you want to submit a non-working lab, each member of your team needs to send an email to all lab graders (with a copy to all TAs) stating in the subject line, "EE457 Non-working lab submission request" and obtain an approval from one of them. Submitting a non-working lab or partial lab without such approval is interpreted as an intention to cheat. Sorry to say all this, but this makes sure that the system works well.

What you have to turn-in

On-line (one submission for a team of 2 students)

Please turn in the following (three separate submissions):

1. submit -user ee457lab -tag puvvada_lab7_p3 ee457_lab7_P3.v RF_Content_Lab7_P3.txt TimeSpace.txt names.txt

2. submit -user ee457lab -tag puvvada_lab7_p3_merged ee457_lab7_P3_with_EX1_EX2_merged.v RF_Content_Lab7_P3_EX1_EX2_merged.txt TimeSpace.txt names.txt

Paper submission (individual effort, each student separately)

Please complete, staple together, and submit pages 2/14, 4/14, 5/14 and also pages 8/14 to 14/14.

Questions (individual effort, paper submission, submit pages 2/14, 4/14, 5/14 and also pages 8/14 to 14/14)

Please consider the following questions before implementing and designing your control. You need to think who can wait for forwarding data latest until when and who can provide forwarding data earliest by when.

Q 1
Can an instruction postpone receiving forwarding data until it reaches EX2 stage? If an instruction can postpone receiving forwarding help until reaching EX2, would it still try to receive help while it is in EX1 (may be because the donor instruction can not wait)?

Q 2
Are there occasions where you end up stalling an instruction because you could not provide the needed forwarding data to it in EX1?

Q 3
Can an instruction in EX2 provide forwarding help to an instruction behind it? Or is it too early for any instruction to start providing forwarding help while it is in EX2?

Q 4
Which instructions need to wait until they reach WB stage for them to provide forwarding help? And why they need to wait until then?
Q 5
Priority in Forwarding: Recall that, in the MIPS pipelined CPU design, if the instructions in both MEM stage and also in WB stage are willing to provide forwarding help to the instruction in EX stage, we exercise priority and accept help from (MEM/WB) stage. Do we have such a situation here? If so, explain with an example instruction sequence.

Q 6
Normally forwarding is done at the beginning of a clock so that the recipient instruction can process the information during the clock. However, sometimes it may make sense (as in Lab 7 Part 1) to forward information at the end of the clock. Can an instruction such as ADD4 or ADD1 in EX2 provide forwarding help to an ADD4 or MOV instruction in EX1 towards the end of the clock? If yes, did you provide such an arrangement in your design? If not, is it desirable to provide such an arrangement? Does it cost extra? Does it avoid any stalls, thereby improving the pipeline performance? Or is it that the particular help we plan to offer at the end of the clock will anyway be available at the beginning of the next clock and it is just one and the same (one and the same, whether you provide data at the end of the current clock or at the beginning of the next clock)? Explain briefly.
Q 7  The following is a slightly modified version of the Q#3 from Spring 2002 Final Exam. Please answer this as part of this lab questions.

7.1  Suppose the current design is working at 500 MHz (clock period = 2 ns). Due to VLSI technology improvements, you can either (1) double the clock rate to 1 GHz (clock period = 1 ns) or (2) keep the clock rate at the same level (500 MHz) and combine IF and ID stages into one stage called IFID and also the EX1 and EX2 stages into one stage called EX12 stage. Circle your choice and explain.
   a) Both options are equally good   (b) Option 1 is better than option 2   (c) Option 2 is better than option 1

7.2  Given below are four flip-flop hook-ups and five statements describing their operation. You need to find a matching statement for each of the hook-ups.

(a) Once SET, it remains set.
(b) Once RESET, it remains reset.
(c) If it is currently SET, it will RESET on the next clock.
(d) If it is currently RESET, it will SET on the next clock.
(e) none of the above

Matching statement____  Matching statement____  Matching statement____  Matching statement____
Let us go back to the original (slow) VLSI technology. Let us still combine the two stages EX1 and EX2 into one stage **EX12** as shown in the *incomplete* design on page 14/14. The SUB3 and ADD4 instructions require *only one of two resources* in EX12 and take *only one clock* to pass through EX12. MOV and NOP do not need any computation. Only the **ADD1** instruction requires both subtract_three and add_four operations and takes **two-clocks** through EX12. At that time we need to stall the entire pipe including WB stage for one clock so that the ADD1 completes using EX12 stage.

7.3.1 Explain why do we need to stall the WB stage also and why cannot we send a bubble into the WB stage. Use the sequence on this page *(where the entire pipeline is stalled during clock 2)* to explain.

---

ADD4 $5, $6 ; ($5) <= ($6) + 4
ADD1 $4, $5 ; ($4) <= ($5) + 1
SUB3 $3, $5 ; ($3) <= ($5) - 3
ADD4 $1, $2 ; ($1) <= ($2) + 4

---

Pipe Stalled

<table>
<thead>
<tr>
<th>Clock</th>
<th>IF</th>
<th>ID</th>
<th>EX12</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock 1</td>
<td>ADD4</td>
<td>SUB3</td>
<td>ADD1</td>
<td>ADD4</td>
</tr>
<tr>
<td>Clock 2</td>
<td>ADD4</td>
<td>SUB3</td>
<td>ADD1</td>
<td>ADD4</td>
</tr>
<tr>
<td>Clock 3</td>
<td>ADD4</td>
<td>SUB3</td>
<td>ADD1</td>
<td></td>
</tr>
<tr>
<td>Clock 4</td>
<td>ADD4</td>
<td>SUB3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

7.3.2 Notice that in the *incomplete* design, we have provided a flip-flop in the **EX12** stage to help you *stall the entire pipe for one clock* *(no more no less)* when **ADD1** is passing through the **EX12** stage. Complete the design after answering the following questions.

- Do you need to stall the pipe to resolve any *dependency* problem?  **Yes / No**
  Explain. Notice that we have removed the hazard detection unit.

---

- Notice that
  -- we removed one of the two comparators
  -- we removed the forwarding mux X2_Mux and FU2
  -- we removed the prioritization mux.
  Explain why it is appropriate to remove these.
7.4 Performance: Carefully compare the original Lab 7 Part 3 subpart 1 design with the design in the above 7.3 (Lab 7 Part 3 subpart 2). Both are running at 500MHz. Does one of them perform always better? Or depending on the code any one of them could perform better? Decide after considering the two code sequences below and completing the time-space diagrams on this page.

A sequence of **dependent** ADD4's

<table>
<thead>
<tr>
<th>Code A running on Lab #7 Part 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clock 1</strong></td>
</tr>
<tr>
<td>ADD4 $5, $6 ; ($5) &lt;= ($6) + 4; ADD4#1</td>
</tr>
<tr>
<td>SUB3 $4, $5 ; ($4) &lt;= ($5) - 3; SUB3#2</td>
</tr>
<tr>
<td>ADD4 $3, $4 ; ($3) &lt;= ($4) + 4; ADD4#3</td>
</tr>
</tbody>
</table>

B sequence of **independent** ADD1's

<table>
<thead>
<tr>
<th>Code B running on Lab #7 Part 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clock 1</strong></td>
</tr>
<tr>
<td>ADD1 $15, $10 ; ($15) &lt;= ($10) + 1; ADD1#1</td>
</tr>
<tr>
<td>ADD1 $13, $14 ; ($13) &lt;= ($14) + 1; ADD1#2</td>
</tr>
<tr>
<td>ADD1 $11, $12 ; ($11) &lt;= ($12) + 1; ADD1#3</td>
</tr>
</tbody>
</table>

Code A running on design in section 7.3 above

| **Clock 1** | **IF** | **ID** | **EX12** | **WB** |
| ADD4#3 SUB3#2 ADD4#1 |

Code B running on design in section 7.3 above

| **Clock 1** | **IF** | **ID** | **EX12** | **WB** |
| ADD1#3 ADD1#2 ADD1#1 |

One of the two designs is always better. **TRUE / FALSE**. Explain.
7.5 Multicycle implementation:

7.5.1 The Datapath below is complete. Complete the state diagram and produce the two outputs, R_Write and PC_EN (draw combinational logic necessary to produce R_Write and PC_EN).

7.5.2 The design below corresponds to
   (i) the Lab 7 P3 Subpart #1 design (with separate EX1 and EX2 stages) only
   (ii) the design in Lab 7 P3 Subpart #2 design (with EX1 and EX2 merged) only
   (iii) both the above two designs.

7.5.3 If an IR (Instruction register) is available, PC can be incremented early. TRUE / FALSE

Performance improves if we add IR. TRUE / FALSE

Figure for Question 7.5
LAB 7 P 3 with EX1 and EX2 merged Block Diagram

1. Complete the missing connections to the register file.
2. Design the forwarding unit. Generate SKIP1 and SKIP2 signals.
3. Use the flip-flop in EX12 stage to get one extra clock for ADD1 instruction.
4. Control the EN (ENABLE) control signal on PC and the three stage registers IF/ID, ID/EX12, and EX12/WB.

Figure for Question 7.3