Lab 8: Multicycle Processor (Part 1)

Introduction

In this lab and the next, you will design and build your own multicycle MIPS processor! Your processor should match the design from the text reprinted below (Figure 5.28 of *Computer Organization and Design*). It is divided into four units: the *cunit* (control), *eunit* (execution), *iunit* (instructions) and *munit* (memory). Note that the munit contains the shared memory used to hold both data and instructions. Also note that the cunit comprises both the decoder using OP(5:0) and the ALUCONTROL logic taking ALUOP(1:0) and the FUNCT code from the low bits of the INSTRUCTION. The units have the inputs and outputs listed in Figures 2–5.

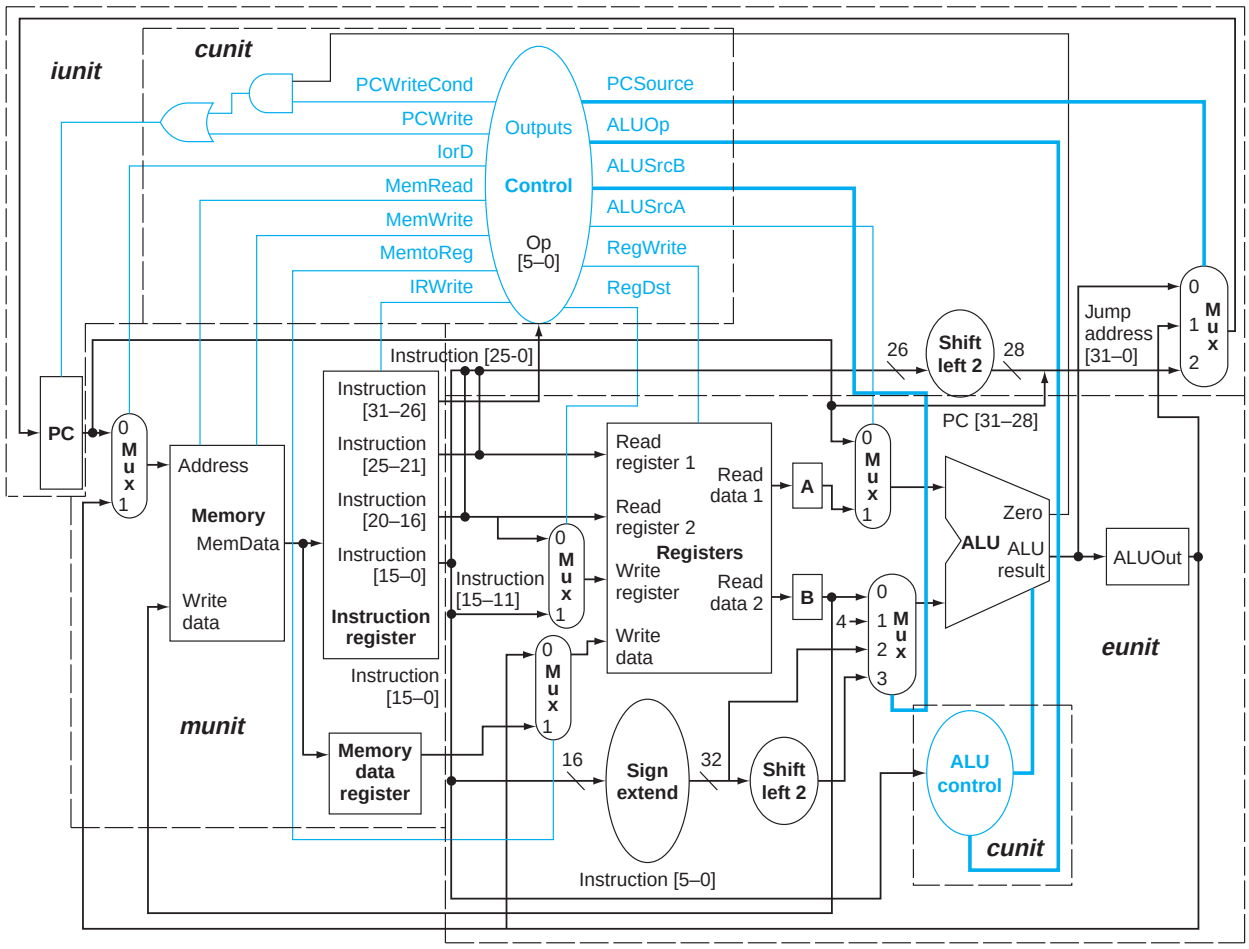

FIGURE 1 Multicycle Processor Datapath.

Cunit Interface Signals					
CLK		Input	REGWRITE		Output
RESET		Input	REGDST		Output
OP	(5:0)	Input	IORD		Output
FUNCT	(3:0)	Input	MEMREAD		Output
ZERO		Input	MEMWRITE		Output

FIGURE 2 Cunit Interface Signals.

Cunit Interface Signals					
PCSOURCE	(1:0)	Output	MEMTOREG		Output
ALUCONTROL	(3:0)	Output	IRWRITE		Output
ALUSRCB	(1:0)	Output	PCENABLE		Output
ALUSRCA		Output			

FIGURE 2 Cunit Interface Signals.

Iunit Interface Signals		
Clk		Input
Reset		Input
Instruction	[31:0]	Input
ALUResult	[31:0]	Input
ALUOut	[31:0]	Input
PCEnable		Input
PCSource	[1:0]	Input
PC	[31:0]	output

FIGURE 3 Iunit Interface Signals.

Munit Interface Signals					
CLK		input	MEMREAD		input
RESET		input	MEMWRITE		input
PC	(31:0)	input	IRWRITE		input
ALUOUT	(31:0)	input	INSTRUCTION	(31:0)	output
WRITEDATA	(31:0)	input	MEMORYDATA	(31:0)	output
IORD		input			

FIGURE 4 Munit Interface signals.

Eunit Interface Signals					
CLK		input	ALUSRCA		input
RESET		input	ALUSRCB	(1:0)	input
INSTRUCTION	(31:0)	input	ALUCONTROL	(3:0)	input
MEMORYDATA	(31:0)	input	ZERO		output
PC	(31:0)	input	ALURESULT	(31:0)	output
REGDST		input	ALUOUT	(31:0)	output
MEMTOREG		input	WRITEDATA	(31:0)	Output
REGWRITE		input			

FIGURE 5 Eunit Interface signals.

Your task in this lab will be to design and test a microcoded state machine for the cunit and to work out a test program for the processor as a whole. In the next lab, you will design the remaining parts of the multicycle processor, the eunit, munit and iunit, and put all four units together and test them. You will be much more on your own to complete these labs than you have been in the past, but may reuse any of your hardware from previous labs.

Test Program

Your first task will be to prepare a simple test program, shown below, to test all of the instructions. The program doesn't do anything terribly interesting, but will prove your processor is working correctly if it ends in an infinite loop at `done` with the proper value in $t4. Translate the program to machine language and predict the values of

major signals after each cycle. You will use this program in the next lab as you debug your processor. Note that the constants 42, 2C and 28 are all in hexadecimal, not decimal. Start the program at address 0 in memory. Recall that branches are counted relative to the next instruction, so the beq branches back by –7 instructions (FFF9).

```
                        addi  $t0, $0, 42
                        j     later
earlier:                addi  $t1, $0, 4
                        sub   $t2, $t0, $t1
                        or    $t3, $t2, $t0
                        sw    $t3, 2C($0)
                        lw    $t4, 28($t1)
done:                   j     done
later:                  beq   $0, $0, earlier
```

What result should be in $t4 when the program reaches done?

Before debugging your multicycle processor, you will need to have a good idea of what to expect out of the processor. Complete Figure 12 at the end of this lab showing the values of the FSM state, PC, INSTRUCTION, SRCA, SRCB, ALURESULT and ZERO at each cycle for your program. The FSM states (0 through 9) are numbered as in Figure 5.38 of *Computer Organization and Design*; addi requires two additional states, A and B, as shown in Figure 7. When Figure 7 indicates a don't care, you can leave a ? in Figure 12 indicating the result is implementation-dependent. Notice that the instruction code is fetched during state 0 and therefore not updated until state 1 of each instruction.

Cunit Design

The cunit is the most complex part of the multicycle processor. It should take the inputs and produce the outputs described in the unit overview previously. On RESET, the cunit should start at State 0. The cunit should support the instructions from Figure 5.38 plus addi. The state transition table is shown in Figure 7. Recall that SEQ of 00 means next state, 10 means Dispatch 1, 11 means Dispatch 2, and 01 means Fetch, as defined in Figure 5.7.3 in CD Section 5.7 on the Companion CD for *Computer Organization and Design*.

Design your controller using a microcode sequencer, as shown in Figure 6. The microcode storage is a ROM taking a 4-bit state as input and producing 18 bits of output.

Translate the state transition table in Figure 7 into a series of 5-nibble hexadecimal values (treat the upper two bits as 0) that can be entered as the contents of the microcode ROM.

The address select logic of the Microcode Sequencer is shown in Figure 8. The dispatch tables are stored in the ROMs that determine next addresses based on OP(5:0). For example, the Dispatch2 ROM should go to state 3 on a lw (OP(5:0)=23_{16}) or state 5 on OP(5:0)=$2B_{16}$).

In Xilinx ISE 4.2, a powerful tool, CORE generator, can be used to set up the memory elements such as RAMs, ROMs and Register files. Additionally, elements such as bus multiplexers can be generated using CORE generator. These elements are useful for selecting the correct address for the ROM or for selecting data into the ALU for different cases. For example, here you will probably need a 4-to-1 multiplexer to select an address with a width of 4 bits for the ROM (which has a depth of 16 and a width of 18 bits for each word) where your microcode is stored. To do so, choose Project→New Source, select the source type to be Coregen IP and name your multiplexer, for example, as "mux4_4bits". When the CORE generator is launched, the working interface is shown as in Figure 9. You can use it to generate different kind of modules, but we mainly use the Basic Elements category that includes memory elements, multiplexers, registers, shifters and pipelining. In this case, double click the Bus Multiplexer and choose it as the component type. Then you will be asked to enter the component name, number of input buses and bus width, etc. Make sure it is a Non Registered version. On the left side, a diagram of the symbol will be shown. When finished, you can click Generate on the left lower corner, and a multiplexer component will be generated and listed in the Generated Modules in the CORE generator. At the same time, a symbol with the same name will be available to use for in the schematic editor.

Next you can set up two ROMs for storing the dispatch tables and one ROM for storing the microcode. Before generating a ROM, you need to use Memory Editor in the CORE generator interface to input the content of the ROM. Choose Tools→Memory Editor, and the interface of Figure 10 is shown. In the case of the microcode ROM,

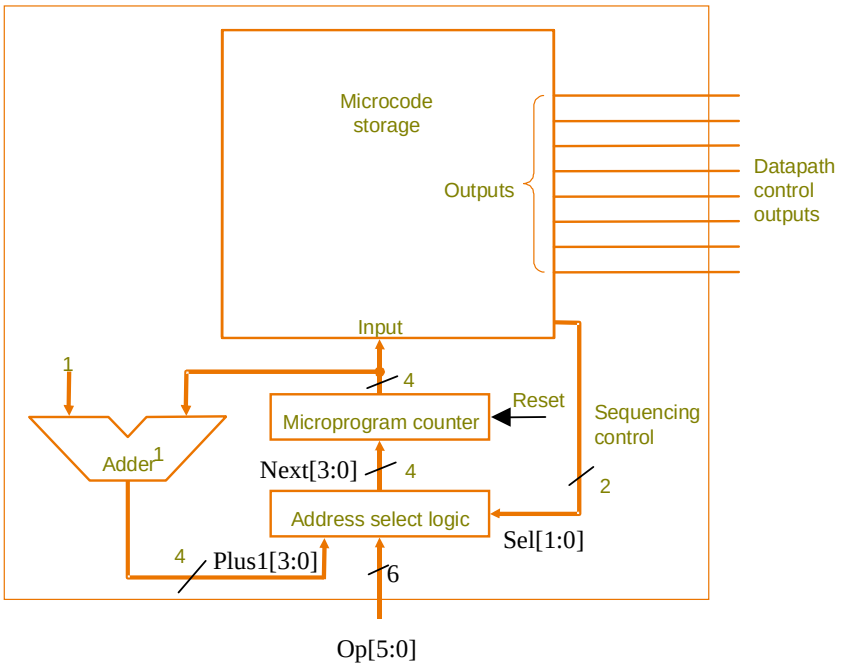

FIGURE 6 Microcode Sequencer.

STATE	ALUOP(1:0)	ALUSRCA	ALUSRCB(1:0)	REGWRITE	REGDST	MEMTOREG	IORD	MEMREAD	MEMWRITE	IRWRITE	PCSOURCE(1:0)	PCWRITE	PCWRITECOND	SEQ(1:0)	ROM Contents
0	00	0	01	0	X	x	0	1	0	1	00	1	0	00	02148
1	00	0	11	0	X	x	x	0	0	0	xx	0	0	10	
2	00	1	10	0	X	x	x	0	0	0	xx	0	0	11	
3	xx	X	xx	0	X	x	1	1	0	0	xx	0	0	00	
4	xx	X	xx	1	0	1	x	0	0	0	xx	0	0	01	
5	xx	X	xx	0	X	x	1	0	1	0	xx	0	0	01	
6	10	1	00	0	X	x	x	0	0	0	xx	0	0	00	
7	xx	X	xx	1	1	0	x	0	0	0	xx	0	0	01	
8	01	1	00	0	X	x	x	0	0	0	01	0	1	01	
9	xx	X	xx	0	X	x	x	0	0	0	10	1	0	01	
A	00	1	10	0	X	x	x	0	0	0	xx	0	0	00	
B	xx	X	xx	1	0	0	x	0	0	0	xx	0	0	01	

FIGURE 7 Microcode ROM Contents.

you need 16 words (although you only used 12 words, you need 4-bit address to access the ROM), and each word is 18 bits wide. When you finish entering all the microcodes at the corresponding addresses, select File→Save as "controlrom"; a *controlrom_controlrom.coe* file will be generated at the same time. This file can be imported later to set up the complete 16×18 ROM memory block. Now close the Memory Editor. You should also use memory editor to input the content for the dispatch ROMs.

Next, you are ready to set up the ROM module. In the CORE generator interface, select Basic Elements→Memory Elements, then select Distributed Memory in the content window by double clicking. Go through different options for the setup. For example, set Component Name to "controlrom", and set Depth and Data Width with the same values as you used for the "controlrom" memory block in the Memory Editor. Check

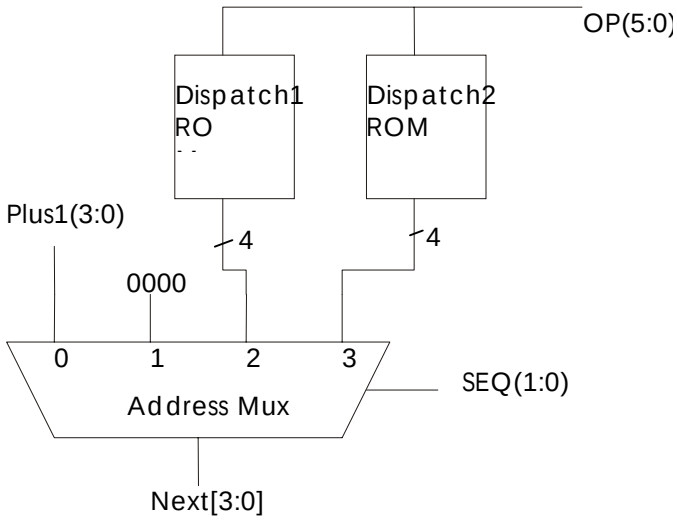

FIGURE 8 Address Select Logic.

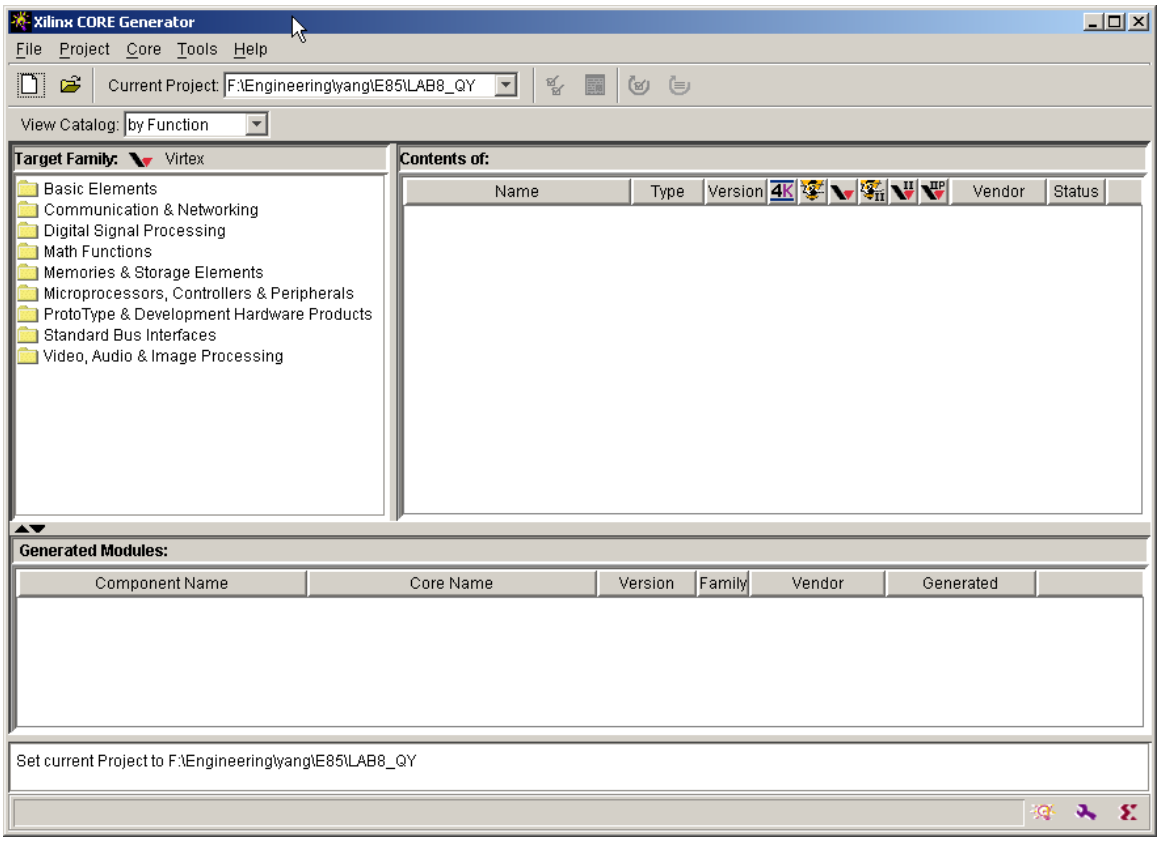

FIGURE 9 Xilinx CORE Generator Interface.

Non Registered. This is shown in Figure 11 below (although this illustration was captured using a different name for the component).

When this is done, click Next; you will need to load Initial Contents of the memory. Click Load Coefficients to find the *.coe* files you just generated from the Memory Editor to store the contents of your ROM. Now you complete the setting of the ROM by clicking Generate button. A ROM called “controlrom” with stored microcodes is created, and a symbol with the same name is also ready to use for schematic files.

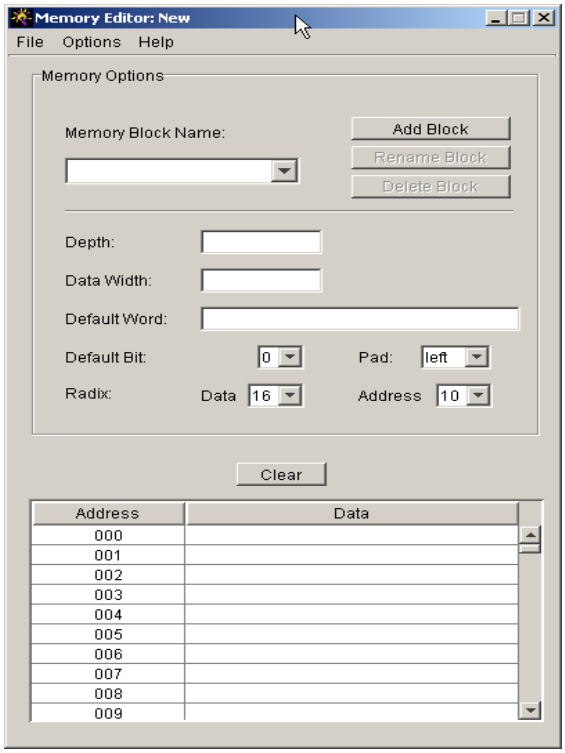

FIGURE 10 Memory Editor Interface.

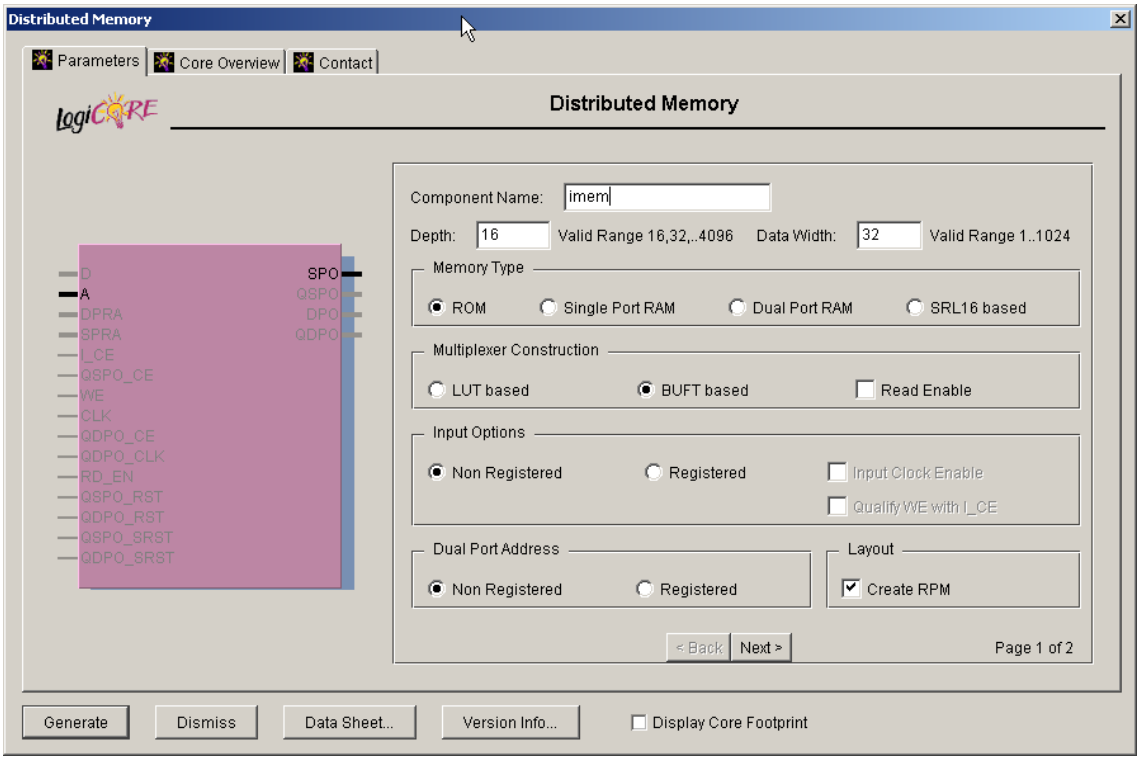

FIGURE 11 Set up the ROM as Distributed Memory.

You can go through a similar procedure to setup your ROMs to store dispatch1 and dispatch2. Both dispatch ROMs are 4-bits wide and addressed by OP(5:0); so a depth of 64 is available, although you only use a few of the addresses.

In addition to the address select logic and microcode ROM, the microprogram counter in Figure 6 is neeeded to interface between the address logic and the microcode ROM. It not only delivers the address from the select logic to the ROM, but also makes this address available for the calculation of the next address (Plus1, or add address with 1). In addition you should be able to reset the state of the counter. You may use a simple data register to implement the microprogram counter. Explore the CORE generator to setup your register properly.

Now all the major elements are ready for your cunit. Create a schematic for your cunit. The schematic should include the microcode sequencer from Figure 6 along with logic to compute ALUCONTROL (3:0) from ALUOP(1:0) (you can copy this part from your Lab 2) and additional logic to compute PCENABLE from PCWRITE, PCWRITECOND and ZERO.

There are a number of ways to connect the microcode ROM output bus to the individual outputs. One of the simpler ways is to draw 18 bus taps and hook each bit of the bus to a buffer, which in turn can drive the output with the appropriate name.

When you have completed your schematics, simulate the cunit. Develop testbench waveform for the CLK, RESET, OP, FUNCT and ZERO inputs based on Figure 12. Make sure you reset the system in the first clock cycle. Print out your waveforms showing all of the control outputs at each state. Make sure the outputs match your expectations. (You haven't written these expectations, so you'll have to work them out as you go by looking at the current state and Figure 7.) If you find any errors, debug your circuit.

Cycle	RESET	PC	INSTRUCTION	FSM State	SRCA	SRCB	ALU-RESULT	ZERO
0	1	00	0	0	00	04	04	0
1	1	00	0	0	00	04	04	0
2	0	04	addi 20080042	1	04	108	10C	0
3	0	04	addi 20080042	A	00	42	42	0
4	0	04	addi 20080042	B	??	??	??	?
5	0	04	addi 20080042	0	04	04	08	0
6	0	08	j 08000008	1	08	20	28	0
7	0	08	j 08000008	9	??	??	??	?
8	0	20	j 08000008	0	20	04		0
9	0	24	beq 1000fff9	1	24	FFFFFFE4	08	0
10	0	24	beq 1000fff9	8	00	00	00	1
11	0	08	beq 1000fff9		08	04	0C	0
12	0	0C	addi 20090004					0
13	0	0C	addi 20090004	A	00	04	04	0

FIGURE 12 Expected Instruction Trace.

Cycle	RESET	PC	INSTRUCTION	FSM State	SRCA	SRCB	ALU-RESULT	ZERO
14	0	0C	addi 20090004	B	??	??	??	?
15	0							
16	0							
17	0							
18	0							
19	0	10	sub 01095022	0	10	04	14	0
20	0							
21	0							
22	0							
23	0							
24	0	18	sw ac0b002c	1	18	B0	C8	0
25	0							
26	0							
27	0							
28	0	1C	lw 8d2c0028	1	1C	A0	BC	0
29	0							
30	0							
31	0	1C	lw 8d2c0028	4	??	??	??	?
32	0	1C	lw 8d2c0028	0	1C	04	20	0
33	0	20	j 08000007	1	20	1C	3C	0
34	0	20	j 08000007	9	??	??	??	?
35	0	1C	j 08000007	0	1C	04	20	0
36	0	20	j 08000007	1	20	1C	3C	0
37	0	20	j 08000007	9	??	??	??	?

FIGURE 12 Expected Instruction Trace. (Continued)