

LAB 7: Processor Design in VHDL

Objective: In this lab, you will design, simulate, synthesize and test a simple microprocessor. You will synthesize your processor using the Xilinx libraries and download and test your design on a Xilinx demo board.

I. Processor Specifications

This simple processor will have an 8-bit address bus and an 8-bit bi-directional data bus for accessing both instructions and data. A *simplified* block diagram of the processor is given in Figure 1. The EPROM and RAM chips have been wire-wrapped onto the Xilinx FPGA demo board. You will need to design your processor so that the read and write operations satisfy the timing specifications of these external memory chips.

The instruction set and op codes for the processor will be defined. However, the internal architecture of the processor is left for you to design. The processor can have a *multi-cycle implementation*, in which the processor executes a single instruction at a time, or it can have a *pipelined implementation*, in which the processor executes several instructions concurrently. The block diagram given in Figure 1 assumes a multi-cycle implementation. For a pipelined design, you will need to modify the data path by adding pipeline registers at the appropriate places. Your processor, whether multi-cycle or pipelined, must execute the programs that are provided to you in the simulation test bench and in the EPROM. That is, you can not modify the programs by changing the instruction sequence in order to accommodate your design.

The instruction set can be divided into the following basic groups:

- **Load and Store** instructions, including **load immediate**, **load** and **store**, move data between memory and the register file.
- **Computational** instructions, including **add**, **increment**, **negate** and **subtract**, perform ALU operations on register values and write the results back into the register file.
- **Branch** instructions, including **branch**, **branch if zero** and **branch if negative**, change the instruction execution sequence of a program.

The instruction format and opcodes are shown in Table 1, which also summarizes the operation of each instruction.

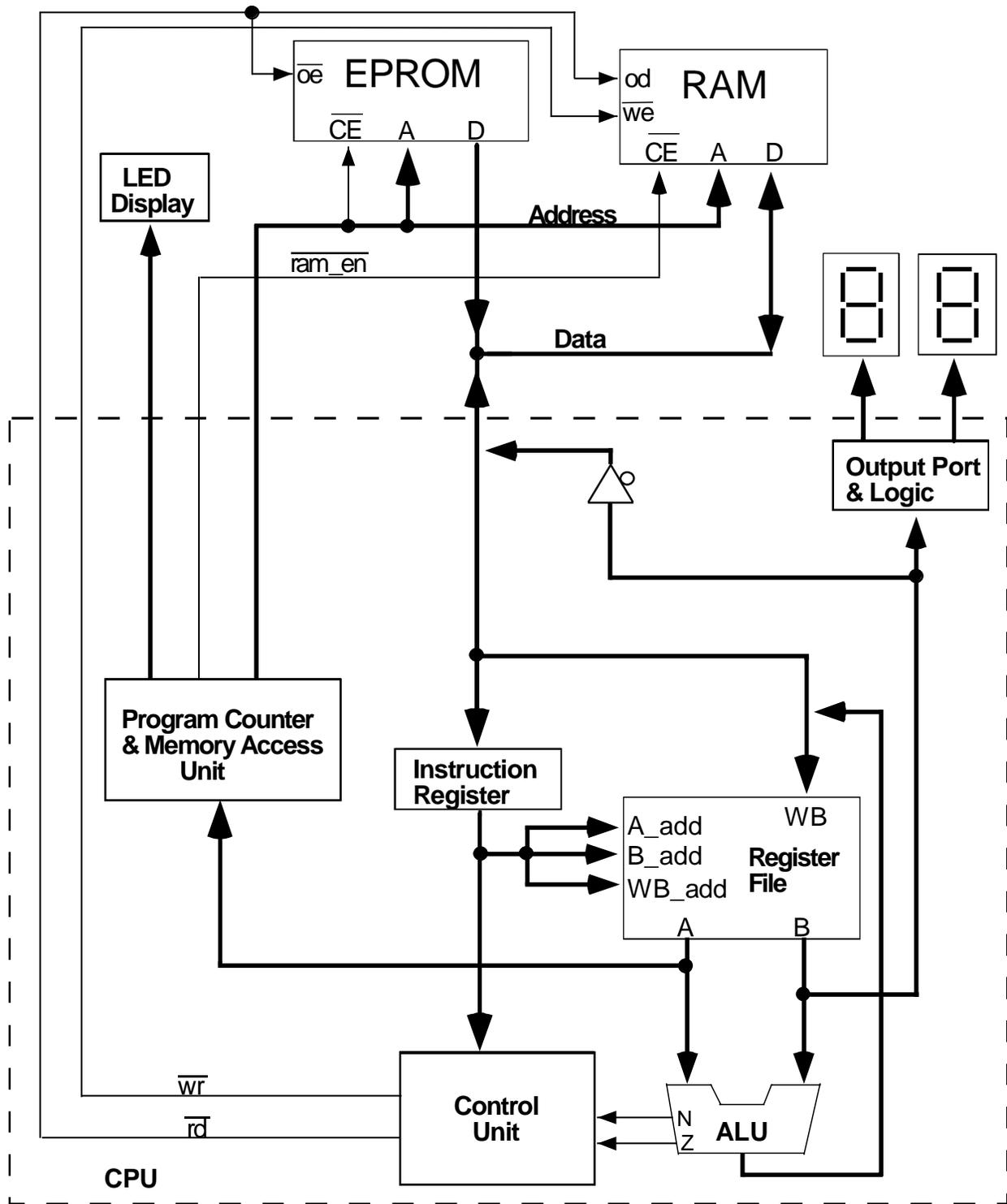


Figure 1: Processor Block Diagram

Instruction	Mnemonic		Instruction Format			Operation
			(Bits 7-0)			
No operation	NOP		0000	Xx	xx	
Load Immediate (Note 1)	LDI	Rb	0001	Rb	xx	$Rb \leftarrow M[PC+1]$ $PC \leftarrow PC+1$
Load	LD	Rb, Ra	0010	Rb	Ra	$Rb \leftarrow M[Ra]$
Store	ST	Rb, Ra	0011	Rb	Ra	$M[Ra] \leftarrow Rb$
Add	ADD	Rb, Ra	0100	Rb	Ra	$Rb \leftarrow Rb+Ra$
Increment	INC	Rb, Ra	0101	Rb	Ra	$Rb \leftarrow Ra+1$
Negate	NEG	Rb, Ra	0110	Rb	Ra	$Rb \leftarrow -Ra$
Subtract	SUB	Rb, Ra	0111	Rb	Ra	$Rb \leftarrow Rb-Ra$
Branch	BR	Ra	1000	xx	Ra	$PC \leftarrow Ra$
Branch if zero	BRZ	Ra	1001	xx	Ra	$PC \leftarrow Ra$ if Z=1
Branch if negative	BRN	Ra	1010	xx	Ra	$PC \leftarrow Ra$ if N=1

Table 1: Instruction format and op code specification

Note 1 The next byte in memory after the LDI instruction is loaded into Rb. The PC is then incremented so that the instruction following the data byte will be the next instruction fetched.

II. Processor Operation

The execution steps and the execution time can vary for the different instruction types. For example, the branch instructions may only require a few clock cycles to execute whereas the store instruction may require several more clock cycles in order to meet all the memory device's timing specifications. A general description of the execution steps required for each instruction is given below. This description is only intended to provide a basic understanding of how the processor will operate. It is not required that your design follows these steps precisely.

1. Instruction Fetch

The first execution step of *any* instruction is the instruction fetch. The contents of the program counter (PC) are placed on the address bus and the read signal, /rd, is asserted so that the memory module puts the next instruction onto the data bus. Generally the instruction will be read from the EPROM, but it is also possible to fetch instructions from the RAM. After the instruction has been placed on the data bus, it must be clocked into the Instruction Register (IR) on a rising clock edge. The PC can be incremented on the same clock edge to point to the next instruction in memory.

$$IR \leftarrow M[PC]; PC \leftarrow PC + 1;$$

2. Instruction Decode

In order to execute any instruction, the processor must first "decode" the instruction in order to determine which actions to perform. The operand fields, Ra and Rb, are used to access specific registers in the Register File. These fields can be used directly as the A_add and B_add inputs to the Register File. Rb also specifies the write-back, or destination, address, WB_add.

$$A \leftarrow \text{RF}[Ra]; B \leftarrow \text{RF}[Rb];$$

The op code will be decoded in the Control Unit in order to generate the proper sequence of control signals. The op code will be used as a state machine input in order to determine the output control signals and the next state. The Decode stage may not require a full clock cycle, depending on where you choose to insert registers in the data path.

3. Computation, Branch Completion, or Memory access

The third execution step varies depending on the type of instruction being executed. Thus, instead of a single execution step, there are actually three different steps depending on whether the instruction is an ALU instruction, a branch instruction, or memory-access instruction.

Computation: Perform the ALU operation on the operand or operands specified by the instruction. The ALU instructions are **add**, **increment**, **negate** and **subtract**. When any ALU instruction is executed, the status flags, N and Z, must be clocked into a two-bit status register. The status register will only be updated following an ALU instruction and will not be affected by other instructions.

Branch completion: If the branch condition is satisfied, clock the branch target address into the PC. The N and Z flags, contained in the status register, are used to determine the outcome of the **BRN** and **BRZ** instructions, respectively.

Memory access: Place the address on the address bus. For a store instruction, place the data on the data bus. For the store instruction, the address must be placed on the address bus and satisfy the address set-up time (t_{AW}) before the write signal, /wr, can be asserted low. Thus, you will probably need to allow one clock cycle for the address set-up time.

4. Memory Access Completion or Write-back

The final execution step again varies depending on the instruction type.

Write-back: Store the ALU result in the destination register specified by the Rb field of the instruction.

Memory access completion: Once the setup times have been met, perform the memory access and then satisfy the hold times. For a memory write (i.e. store) operation, the /wr

signal must satisfy the write pulse width (t_{WP}) specification, which will probably require a clock cycle. The address and data must be held on their respective buses after the /wr signal goes high in order to meet the address hold time (t_{WR}) and the data hold time (t_{DH}). These hold times will most likely require another clock cycle.

III. Memory and I/O Interface

The data sheets for the AM2111-1 RAM chips and the 27256 EPROM will be available to you in the lab. The requirements for the EPROM and RAM read cycles are straightforward. The EPROM has an access time of 250 ns and the RAM has an access time of 500 ns (worst-case). These requirements limit the maximum clock speed at which your processor can run. The timing requirements for the RAM write cycle are the most difficult to satisfy due to the various set-up and hold time requirements. A simple implementation of the memory write operation is to use one clock cycle for address set-up time, one clock cycle for the write enable pulse, and one clock cycle for the address and data hold times.

The processor will access memory through an 8-bit address bus and can therefore address 256 bytes of memory. EPROM is located at addresses 0-7F hex, while RAM is at addresses 80-FF hex. The output port, PR, is memory-mapped to location FF hex. Thus, writing to location FF hex will store the data in RAM *and* in the PR register. The contents of the PR register should always be displayed in hexadecimal on two 7-segment displays. Data can be read from RAM location FF hex just as from any other RAM location since the PR port is strictly an output port.

The Xilinx FPGA demo board also has a bank of 8 active-low LEDs, which should be used to display the inverted memory address bus. When you are testing your processor, the LED display will provide useful debugging information.

It is very important that you avoid bus conflicts on the memory data bus. The processor's external data bus must be a bi-directional bus with tri-state capability. The processor, EPROM and RAM must be controlled such that only one device drives the data bus at any time. Whenever one of these devices is not driving the data bus, its output bus should be tri-stated.

IV. ALU

Based on the instruction set of the processor, the ALU must perform four operations: add, increment, negate (i.e. two's complement) and subtract. The inputs to the ALU consist of two 8-bit operands, A and B, and control signals. The output of the ALU consists of an 8-bit result and two status flags, Z and N.

The ALU is essentially a combinational circuit. The only register associated with the ALU is the two-bit status register for storing the N and Z flags. This register should be updated with the status signals N and Z that result from any of the four ALU operations. Otherwise, the register should hold its value. (Note: this status register can actually reside in the Control Unit.)

It is up to you to define the control signals to control the ALU operations. One possibility is to

define an individual control signal for each of the four operations. When the controller decodes an ALU instruction, it would assert the associated control signal, ADD, INC, NEG or SUB, while the other three control signals would be de-asserted. This implies that only one of the control signals will be asserted at a time. The following table illustrates the ALU outputs for each of the four ALU operations.

ADD	INC	NEG	SUB	OUTPUT	Operation
1	0	0	0	A+B	add
0	1	0	0	A+1	increment
0	0	1	0	-A	two's complement
0	0	0	1	B-A	subtract
0	0	0	0	Don't care	no operation

The ALU status signals, Z and N, reflect the status of the ALU output. The Z, or Zero, signal is asserted high when the ALU output is zero. The N, or Negative, signal should be asserted high when the output is negative. The most significant bit of the ALU output indicates the sign of the two's complement number. Note that the Z and N signals are only required to be valid during an ALU operation. If the ALU output is a "don't care", then the status signals Z and N are also "don't cares" since they will only be clocked into the status register at the end of a valid ALU operation.

V. Entity and Interface Definition

The entity declaration and the attribute file define the external interface of the processor. The attribute file, `cpu.attr`, is listed in the Appendix of this handout and will be available in the course `/afs directory, /afs/ece/classes/eec180b/lab7/`. The entity declaration is shown below. The `leds`, `Mdata` and `Maddr` vectors are specified from MSB to LSB, meaning that index 7 corresponds to the MSB and index 0 to the LSB. The `msb_seg` and `lsb_seg` vectors are specified from seven-segment display drivers a to g where index 0 corresponds to display driver a, the top segment, and index 6 corresponds to driver g, the middle segment. Segments b through f are ordered clockwise from the upper right corner to the upper left corner of the seven-segment display. The reset signal is an active-low signal that is generated by a push-button switch on the Xilinx demo board. The `ram_en_bar`, `wr_bar`, and `rd_bar` signals are all active-low output signals.

```
entity cpu is
port (clk, reset : in std_logic;
      Mdata : inout std_logic_vector(7 downto 0);
      Maddr : out std_logic_vector(7 downto 0);
      ram_en_bar : out std_logic;
      wr_bar : out std_logic;
      rd_bar : out std_logic;
      leds : out std_logic_vector(7 downto 0);
      msb_seg, lsb_seg : out std_logic_vector(0 to 6));
end cpu;
```

VI. Lab Requirements

There is no pre-lab required for this assignment. This lab has three separate parts – design, simulation and synthesis - which have specific due dates.

You may work with a partner on this lab. That is, you may work with *one* other person in the class. Cooperative effort or sharing work among larger groups of students or using material from previous quarters is cheating and is strictly prohibited.

Part I - Data path and controller design (100 points)

- a) Draw the block diagram showing all registers, multiplexers, tri-state gates, etc., which must be controlled. The purpose is not to do a detailed gate-level schematic, but to identify the various hardware components that will require control signals.
- b) Define **all** of the input and output signals that will be needed to implement your processor.
- c) Draw a SM chart or a state graph of the finite state machine that tests the input signals and generates the proper sequence of output signals. Draw a timing waveform showing the sequence of control signals asserted for each of the following instructions: Load Immediate, Load, Store, Add, Branch and Branch if zero. Show how many clock cycles are required for each of these instructions and which control signals are asserted in each clock cycle. For example, for the first clock cycle of every instruction, show the control signals which must be asserted to load the IR and increment the PC. Your SM chart and timing waveforms should be *neat* and fully documented. Use grid paper to draw your timing waveforms.

Due: Friday, May 28 by 6:00pm in the 180B homework box in 2131 Engineering II.

(Note: you may want to keep a copy of your work to use in your VHDL coding. It may take up to a week for the TA to grade and return your work.)

Part II - VHDL modeling and functional simulation (100 points)

- a) Write a *synthesizable* VHDL description of your processor, based on your design in Part I.
- b) Use the testbench program, testbench.vhd, along with the package my_fun.vhd to functionally simulate your design. The testbench.vhd and my_fun.vhd files are available in the /afs/ece/classes/eec180b/lab7 directory. These files model the EPROM and RAM components as well as the clock and reset signals. The EPROM is loaded with the example program shown below which calculates the Fibonacci sequence.

Fibonacci number generator test program

<u>Address</u>	<u>Binary Data</u>	<u>Hex</u>	<u>Instruction</u>	<u>Comment</u>
00	00010000	10	-- ldi r0	; r0 <- 0 (first Fib. number)

01		00000000	00	-- 0	
02		00010100	14	-- ldi r1	; r1 <- 80 (RAM address)
03		10000000	80	-- 80	
04		00110001	31	-- st r0,r1	; M[80] <- 0
05		01010000	50	-- inc r0,r0	; r0 <- 1 (second Fib. number)
06		01010101	55	-- inc r1,r1	; r1 <- 81 (RAM address)
07		00110001	31	-- st r0,r1	; M[81] <- 1
08		00010000	10	-- ldi r0	; r0 <- 80 (RAM address)
09		10000000	80	-- 80	
0a		00010100	14	-- ldi r1	; r1 <- 81 (RAM address)
0b		10000001	81	-- 81	
0c	L	00101000	28	-- ld r2,r0	; r2 <- lower Fib. number
0d		00101101	2d	-- ld r3,r1	; r3 <- higher Fib. number
0e		01001011	4b	-- add r2,r3	; r2 <- next Fib. number
0f		00011100	1c	-- ldi r3	; r3 <- branch address X
10		00011010	1a	-- 1a	; address X
11		10100011	a3	-- brn r3	; if N, done - exit loop by branching to X
12		00000000	00	-- nop	; branch delay slot
13		01010000	50	-- inc r0,r0	; increment memory pointer
14		01010101	55	-- inc r1,r1	; increment memory pointer
15		00111001	39	-- st r2,r1	; store new Fib. number in RAM buffer
16		00011100	1c	-- ldi r3	; r3 <- branch address L
17		00001100	0c	-- 0c	; address L
18		10000011	83	-- br r3	; jump to L (loop)
19		00000000	00	-- nop	; branch delay slot
1a	X	00010000	10	-- ldi r0	; r0 <- 80
1b		10000000	80	-- 80	
1c		01110100	74	-- sub r1,r0	; r1 has address of last Fib. number
1d		01010101	55	-- inc r1,r1	; get number of values stored in buffer
1e		00010000	10	-- ldi r0	; r0 <- 8f (RAM address)
1f		10001111	8f	-- 8f	
20		00110100	34	-- st r1,r0	; M[8f] <- number of Fib. values in buffer
21	P	00010000	10	-- ldi r0	; r0 <- 80 (beginning of RAM buffer)
22		10000000	80	-- 80	
23		00010100	14	-- ldi r1	; r1 <- 8f (address of number of values)
24		10001111	8f	-- 8f	
25		00100101	25	-- ld r1,r1	; r1 <- M[8f] (number of values in buffer)
26		01100101	65	-- neg r1,r1	; r1 <- -r1 (two's complement)
27		00011000	18	-- ldi r2	; r2 <- display port address
28		11111111	ff	-- ff	
29	Q	00101100	2c	-- ld r3,r0	; r3 <- Fib. number from RAM buffer
2a		00111110	3e	-- st r3,r2	; display Fib. number on 7-segment displays
2b		00011100	1c	-- ldi r3	; r3 <- branch address Z

2c	00110110	36	-- 36	; address Z
2d	01010000	50	-- inc r0,r0	; increment RAM buffer pointer
2e	01010101	55	-- inc r1,r1	; increment loop counter
2f	00000000	00	-- nop	; delay for Z, N status flags to be set
30	10010011	93	-- brz r3	; if r1=0, branch to Z (exit loop)
31	00000000	00	-- nop	; branch delay slot
32	00011100	1c	-- ldi r3	; r3 <- branch address Q (top of loop)
33	00101001	29	-- 29	; address Q
34	10000011	83	-- br r3	; branch to Q
35	00000000	00	-- nop	; branch delay slot
36	Z 00011100	1c	-- ldi r3	; r3 <- branch address P
37	00100001	21	-- 21	; address P
38	10000011	83	-- br r3	; jump to P to re-display Fib. numbers
39	00000000	00	-- nop	; branch delay slot

You should trace all the signals in the entity of your processor as well as important registers and control signals. In the Synopsys debugger, vhdldb, you can use the cd command to change the working directory to different levels of the design hierarchy in order to trace signals on that level of hierarchy. You can double-click on a signal that you want to trace and then click the Trace button. After you have specified the signals to trace once, you can save the log file as a simulation script for future simulations. An example simulation script is shown below. Once you have specified all the signals that you would like to trace, you simply enter run commands, specifying the number of nanoseconds, i.e. run 10000. You should verify that the waveforms produced by the simulator are in **agreement** with the timing waveforms that you specified in Part I.

Example simulation script:

```

trace clk
trace reset
trace Mdata
trace Maddr
trace ram_en_bar
trace wr_bar
trace rd_bar
trace leds
trace msb_seg
trace lsb_seg
cd UUT
trace IR
trace PC
trace PR
cd U1
trace RF

```

c) Demonstrate your functional simulation to your TA and have him sign a verification sheet.

- d) Turn in your verification sheet along with your VHDL source code. Each of your modules must be well documented so that someone else can easily understand your design.

DUE: Thursday, June 10, 6:00 pm in the 180B homework box in 2131 Engineering II.

Part III - Synthesis to the Xilinx library and downloading to the demo board (50 points)

- a) Once you have verified your design through simulation, synthesize the design using the Xilinx libraries. An example synthesis script, lab7.scr, is provided in the course /afs directory, /afs/ece/classes/eec180b/lab7. You will need to modify this script to account for your module names, architecture names, etc. Check your revised script carefully to be sure that it matches with your file names, entity names, architecture names and that it analyzes and elaborates all of the source files in your design.
- a) Download the bit file to a Xilinx demo board that has an EPROM containing the Fibonacci program. Verify that your processor works correctly. Demonstrate your processor to your TA and have him sign a verification sheet.

DUE: Thursday, June 10, 6:00 pm in the 180B homework box in 2131 Engineering II.

VII. Hints

In order to facilitate the debugging and synthesis of your VHDL model you should follow the guidelines given below:

1. Keep the VHDL model simple. Make sure you know what each VHDL construct you use is likely to produce in hardware. You can accomplish a lot by just "if-then-else" and case statements. You don't need fancy loops, generate statements, or procedures for this exercise.
2. Remember that wait statements infer registers. Use them carefully. Also recall that incompletely specified "if-then-elsif" and case statements infer latches. *Avoid* generating level-sensitive latches!
3. Do not initialize signals in entity or declaration statements, as these initializations cannot be synthesized.
4. Model the various signals using the appropriate data types. You should primarily use std_logic and std_logic_vector. Do not use types indiscreetly. If you model a bus using integer type, make sure that you specify an integer range based on the desired width of the bus.
5. The control unit is the crucial aspect of your design. After you draw the SM chart or state graph, spend some time analyzing it and discussing it with your partner. Make sure all the control signals that you need are there and are assigned correctly in each clock cycle.

6. Make sure that all registers and counters are clocked directly by the system clock rather than by state machine output signals. Your state machine should produce *enable* signals that allow specific registers or counters to be updated on a rising clock edge.

VIII. Extra Credit Option

1. Pipelined processor with data forwarding (50 points)

Design and simulate a pipelined processor with data forwarding which executes the given instruction set. The execution of ALU and branch instructions should be fully overlapped, so that a new instruction is started on each clock cycle. You may need to have pipeline stalls in the load and store operations.

- Draw a detailed data path showing the pipeline registers.
- Explain how you implemented data forwarding.
- Verify your design by functional simulation and demonstrate it to your TA.
- Synthesize your design for the Xilinx libraries. Download and verify your processor on a Xilinx demo board. Have your TA verify your working design.
- Submit your VHDL source code along with the verification sheets and design documentation.

DUE: Thursday, June 10, 6:00 pm in the 180B homework box in 2131 Engineering II.

APPENDIX

cpu.attr

```
set_pad_type -no_clock find(port,"reset");
set_attribute {reset} "pad_location" -type string "P56"
set_attribute find(port,"clk") "pad_location" -type string "P72"
set_attribute find(port,"lsb_seg<0>") "pad_location" -type string "P49"
set_attribute find(port,"lsb_seg<1>") "pad_location" -type string "P48"
set_attribute find(port,"lsb_seg<2>") "pad_location" -type string "P47"
set_attribute find(port,"lsb_seg<3>") "pad_location" -type string "P46"
set_attribute find(port,"lsb_seg<4>") "pad_location" -type string "P45"
set_attribute find(port,"lsb_seg<5>") "pad_location" -type string "P50"
set_attribute find(port,"lsb_seg<6>") "pad_location" -type string "P51"
set_attribute find(port,"msb_seg<0>") "pad_location" -type string "P39"
set_attribute find(port,"msb_seg<1>") "pad_location" -type string "P38"
set_attribute find(port,"msb_seg<2>") "pad_location" -type string "P36"
set_attribute find(port,"msb_seg<3>") "pad_location" -type string "P35"
set_attribute find(port,"msb_seg<4>") "pad_location" -type string "P29"
set_attribute find(port,"msb_seg<5>") "pad_location" -type string "P40"
set_attribute find(port,"msb_seg<6>") "pad_location" -type string "P44"
set_attribute find(port,"leds<0>") "pad_location" -type string "P60"
set_attribute find(port,"leds<1>") "pad_location" -type string "P59"
set_attribute find(port,"leds<2>") "pad_location" -type string "P58"
set_attribute find(port,"leds<3>") "pad_location" -type string "P57"
set_attribute find(port,"leds<4>") "pad_location" -type string "P66"
set_attribute find(port,"leds<5>") "pad_location" -type string "P65"
set_attribute find(port,"leds<6>") "pad_location" -type string "P62"
set_attribute find(port,"leds<7>") "pad_location" -type string "P61"
set_attribute find(port,"Maddr<0>") "pad_location" -type string "P3"
set_attribute find(port,"Maddr<1>") "pad_location" -type string "P4"
set_attribute find(port,"Maddr<2>") "pad_location" -type string "P5"
set_attribute find(port,"Maddr<3>") "pad_location" -type string "P6"
set_attribute find(port,"Maddr<4>") "pad_location" -type string "P7"
set_attribute find(port,"Maddr<5>") "pad_location" -type string "P8"
set_attribute find(port,"Maddr<6>") "pad_location" -type string "P9"
set_attribute find(port,"Maddr<7>") "pad_location" -type string "P10"
set_attribute find(port,"Mdata<0>") "pad_location" -type string "P77"
set_attribute find(port,"Mdata<1>") "pad_location" -type string "P78"
set_attribute find(port,"Mdata<2>") "pad_location" -type string "P79"
set_attribute find(port,"Mdata<3>") "pad_location" -type string "P80"
set_attribute find(port,"Mdata<4>") "pad_location" -type string "P81"
set_attribute find(port,"Mdata<5>") "pad_location" -type string "P82"
set_attribute find(port,"Mdata<6>") "pad_location" -type string "P83"
set_attribute find(port,"Mdata<7>") "pad_location" -type string "P84"
set_attribute find(port,"rd_bar") "pad_location" -type string "P67"
set_attribute find(port,"wr_bar") "pad_location" -type string "P68"
set_attribute find(port,"ram_en_bar") "pad_location" -type string "P70"
```

my_fun.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

package my_fun is
type ma is range 0 to 127;
type memory is array (ma) of bit_vector(7 downto 0);
function get_int(val:std_logic_vector(6 downto 0)) return ma;
constant hiZ : std_logic_vector(7 downto 0) := "ZZZZZZZZ";
constant ROM: memory:= (
    x"10",
    x"00",
    x"14",
    x"80",
    x"31",
    x"50",
    x"55",
    x"31",
    x"10",
    x"80",
    x"14",
    x"81",
    x"28",
    x"2d",
    x"4b",
    x"1c",
    x"1a",
    x"a3",
    x"00",
    x"50",
    x"55",
    x"39",
    x"1c",
    x"0c",
    x"83",
    x"00",
    x"10",
    x"80",
    x"74",
    x"55",
    x"10",
    x"8f",
    x"34",
    x"10",
    x"80",
    x"14",
    x"8f",
    x"25",
    x"65",
    x"18",
    x"ff",
    x"2c",
    x"3e",
    x"1c",
    x"36",
    x"50",
```

```

        x"55",
        x"00",
        x"93",
        x"00",
        x"1c",
        x"29",
        x"83",
        x"00",
        x"1c",
        x"21",
        x"83",
        others => x"00");
end my_fun;

package body my_fun is
function get_int(val:std_logic_vector(6 downto 0)) return ma is
variable temp : ma;
begin
    temp := 0;
    for i in 0 to 6 loop
        if (val(i)='1') then
            temp := temp + 2**i;
        end if;
    end loop;
    return(temp);
end; -- function
end my_fun;

```

testbench.vhd

```
-- Testbench

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
use WORK.my_fun.all;

entity tb is
end tb;

architecture test of tb is

component cpu
    port (clk, reset : in std_logic;
          Mdata : inout std_logic_vector(7 downto 0);
          Maddr : out std_logic_vector(7 downto 0);
          ram_en_bar, wr_bar, rd_bar : out std_logic;
          leds : out std_logic_vector(7 downto 0);
          msb_seg, lsb_seg : out std_logic_vector(0 to 6));
end component;

signal clk, reset, ram_en_bar, wr_bar, rd_bar : std_logic;
signal Mdata, Maddr, leds : std_logic_vector(7 downto 0);
signal msb_seg, lsb_seg : std_logic_vector(0 to 6);
signal RAM : memory;

begin

UUT : cpu
    port map(clk=>clk, reset=>reset, Mdata=>Mdata, Maddr=>Maddr,
             ram_en_bar=>ram_en_bar, wr_bar=>wr_bar, rd_bar=>rd_bar,
             leds=>leds, msb_seg=>msb_seg, lsb_seg=>lsb_seg);

driver : process
begin
    reset <= '0';
    wait for 60 ns;
    reset <= '1';
    wait;
end process;

EPROM : process(Maddr, rd_bar)
begin
    if ((Maddr(7)='0') and (rd_bar='0')) then
        Mdata <= To_StdLogicVector(ROM(get_int(Maddr(6 downto 0))));
    else
        Mdata <= hiZ;
    end if;
end process;

DRAM : process(Maddr, ram_en_bar, wr_bar, rd_bar)
begin
    if ((Maddr(7)='1') and (ram_en_bar='0')) then
        if (rd_bar='0') then
            Mdata <= To_StdLogicVector(RAM(get_int(Maddr(6 downto 0))));
        end if;
    end if;
end process;
```

```

        elsif wr_bar'event and wr_bar='1' then
            RAM(get_int(Maddr(6 downto 0))) <= To_bitvector(Mdata);
        end if;
    else
        Mdata <= hiZ;

    end if;
end process;

clkgen : process
begin
    clk <= '1';
    wait for 20 ns;
    clk <= '0';
    wait for 20 ns;
end process;

end test;

configuration tb_cfg of tb is
for test
end for;
end tb_cfg;

```

lab7.scr

```
/* ===== */
/* Lab 7 synthesis script file
/* ===== */

/* To execute this DC Shell script, from command line, type: */
/*   dc_shell -f lab7.scr > lab7.log & */

top_design_name = lab7.vhd
top_ent_name = cpu
output_name = cpu
attr_name = cpu

remove_design -all

/* Analyze and elaborate design files in hierarchical order. */
/* I.e. Modules which are referenced in other files first. */
/* Packages, low-level modules, finally the top-level design. */

analyze -format vhdl my_pkg.vhd /* package */

/* analyze the source file. */
/* Elaborate the entity. */

analyze -format vhdl regfile.vhd
elaborate reg_file /* entity */

analyze -format vhdl alu.vhd
elaborate ALU /* entity */

analyze -format vhdl control.vhd
elaborate control /* entity */

analyze -format vhdl top_design_name
elaborate top_ent_name /* entity */

/* set the current design to the top-level design file. */

current_design top_ent_name
link

set_attribute {top_ent_name} "part" -type string "4005epc84-3"
set_operating_conditions "WCCOM"
set_wire_load "4005e-3_avg"
set_port_is_pad find(design,top_ent_name)
include attr_name + ".attr"
uniquify
insert_pads
check_design
compile -ungroup_all
check_design
replace_fpga
check_design
write -f db -output output_name + ".db"
write -f xnf -output output_name + ".sxnf"
quit
```