EE 2700 Project 3 – Microprocessor Controller

In this project, you will finish the microprocessor you started in Project 2 by adding the control logic responsible for asserting the control signals at the right times. You may work individually or in teams of two.

Design.

The controller must reset in a state that (a) puts the PC on the address bus and (b) asserts the read signal. The first opcode of the first instruction will then be available on the internal bus. Based on that opcode (and possibly the carry flag), the state machine should transition to the first state of a sequence that will execute that instruction. After execution is complete, the state machine must put the PC back on the bus to fetch the next opcode.

Implement your controller with a state machine using a behavioral VHDL model. You may write a test bench for the controller if it is helpful, but it is not required. If you used a schematic in Project 2, create a schematic symbol for your controller and integrate it with the rest of your CPU. If you used a structural VHDL model in Project 2, add a component declaration and an instance of the controller to your model. You will need to remove the control signals from the port list (e.g. ld_pc, ld_mar, inc_pc, fetch, wr, op, etc.) as these will now come from the controller.

External to the CPU, the read, write and reset signals are required to be active low. If you have designed your CPU so that these signals are active high, add inverters to your top-level design between the (active low) ports and the (active high) internal signals.

Create a test bench for the CPU. Modify your test bench first by including a reference to the unsigned std_logic library:

```
use IEEE.std_logic_unsigned.all;
```

Second, declare the opcodes for your instruction set in the architecture block of the test bench. You may use the following constant declarations as an example:

```
constant op_halt:  std_logic_vector(7 downto 0) := X"00";
constant op_jmp:   std_logic_vector(7 downto 0) := X"01";
constant op_jc:    std_logic_vector(7 downto 0) := X"02";
constant op_jnc:   std_logic_vector(7 downto 0) := X"03";
constant op_ldi:   std_logic_vector(7 downto 0) := X"10";
constant op_addi:  std_logic_vector(7 downto 0) := X"11";
constant op_adci:  std_logic_vector(7 downto 0) := X"12";
constant op_xori:  std_logic_vector(7 downto 0) := X"13";
constant op_ldm:   std_logic_vector(7 downto 0) := X"20";
constant op_addm:  std_logic_vector(7 downto 0) := X"21";
constant op_adcm:  std_logic_vector(7 downto 0) := X"22";
constant op_xorm:  std_logic_vector(7 downto 0) := X"23";
constant op_stm:   std_logic_vector(7 downto 0) := X"30";
```

If necessary, modify the values of these constants to match your choice of opcodes. Third, declare a memory in the test bench and initialize it so it contains a program for your CPU to execute. Do this by including the VHDL code shown below. (Note: this program computes 2 numbers, 145 (91_{16}) and 232 ($E8_{16}$), then uses Euclid's algorithm to find their greatest common factor.)

```
type ram_type is array (0 to 255) of STD_LOGIC_VECTOR (7 downto 0);
signal ram: ram_type := (
   op_ldi,  X"9E", -- 00 LDI  9E
   op_addi, X"AA", -- 02 ADDI AA (A = 48, C=1)
   op_stm,  X"3F", -- 04 STM  3F
   op_adcm, X"3F", -- 06 ADCM 3F (A = 91, C = 0)
   op_stm,  X"3F", -- 08 STM  3F
   op_adci, X"7B", -- 0A ADCI 7B (A = 0C, C = 1)
   op_xorm, X"3F", -- 0C XORM 3F (A = 9D, C = 1)
   op_adci, X"4A", -- 0E ADCI 4A (A = E8, C = 0)
   op_stm,  X"3E", -- 10 STM  3E
   op_ldm,  X"3F", -- 12 LDM  3F
   op_xori, X"FF", -- 14 XORI FF
   op_addm, X"3E", -- 16 ADDM 3E (C if [3E]>[3F])
   op_jc,   X"28", -- 18 JC   28
   op_ldm,  X"3E", -- 1A LDM  3E
   op_xori, X"FF", -- 1C XORI FF
   op_addm, X"3F", -- 1E ADDM 3F (C if [3F]>[3E])
   op_jnc,  X"2E", -- 20 JNC  2E (done)
   op_addi, X"01", -- 22 ADDI 01
   op_stm,  X"3F", -- 24 STM  3F
   op_jmp,  X"12", -- 26 JMP  12 (top of loop)
   op_addi, X"01", -- 28 ADDI 01
   op_stm,  X"3E", -- 2A STM  3E
   op_jmp,  X"12", -- 2C JMP  12 (top of loop)
   op_halt,        -- 2E HALT
   others=>X"FF" );
```

Finally, model the memory by adding the following VHDL code after the first BEGIN:

```
process (clk)
begin
  if rising_edge(clk) then
    if wr_L = '0' then
      ram(conv_integer(address))<=data;
    end if;
  end if;
end process;

data <= ram(conv_integer(address)) when rd_L = '0' else "ZZZZZZZZ";
```

Add a process that briefly asserts reset, then generates at least 135 clock pulses. Simulate your test bench. If it is correct, the memory at address 63 ($3F_{16}$) will contain 29 ($1D_{16}$). Print the last page of the simulation (making sure to show the values on the address and data busses during the final two write cycles).

Switch ISE to implementation mode and open the project properties. Select a device large enough to hold the CPU (e.g. Spartan 3E – XC3S500). Synthesize the design and print the first 2 pages of the synthesis report.

Turn in the top level schematic or structural VHDL module, the VHDL code for the controller, the test bench, the simulation results and the synthesis report. (Note: submissions that lack the synthesis report will be accepted, but there will be a 20% penalty.)

Design Guidelines

All the flip-flops in the design that have resets must use the same global asynchronous reset (or preset). This is the reset which starts the system. No internally generated asynchronous resets are allowed.

All flip-flops and registers must use a single, global clock. No internally generated clocks are allowed.

The design should not have internal bidirectional busses. (The external bidirectional data bus is created at the I/O port.)