California State University, Long Beach
Computer Engineering and Computer Science Department
CECS 361 - Digital Design Techniques and Verification
Lab 4 - Iterative Divider

Objectives:

- To familiarize yourself with creating Finite State Machines
- To learn the skills to verify your design via simulation at every stage
- To familiarize students with implementing HDL designs for a target technology

Theory:

Finite state machines (FSMs) are used to model sequential logic that transits among a finite number of internal states. FSM designs are modeled with the least possible amount of states to perform the required function, as less states take less time. The transitions through states occur as functions of the present state of the machine and the current inputs, whereas the outputs at any given state are functions of either the present state (Moore machine) or functions of the present state and present inputs (Mealy machine). This lab assignment will help solidify the concepts covered in "Topic 4: Functional Design Verification."

Assignment:

You are to implement an iterative divider circuit that takes two unsigned numbers *A* and *B*, where *A* is the dividend and *B* is the divisor. The result produces a quotient *Q* and a remainder *R.* Thus, A = B x Q + R. The algorithm to be implemented can be explained by **Figure 1** below.

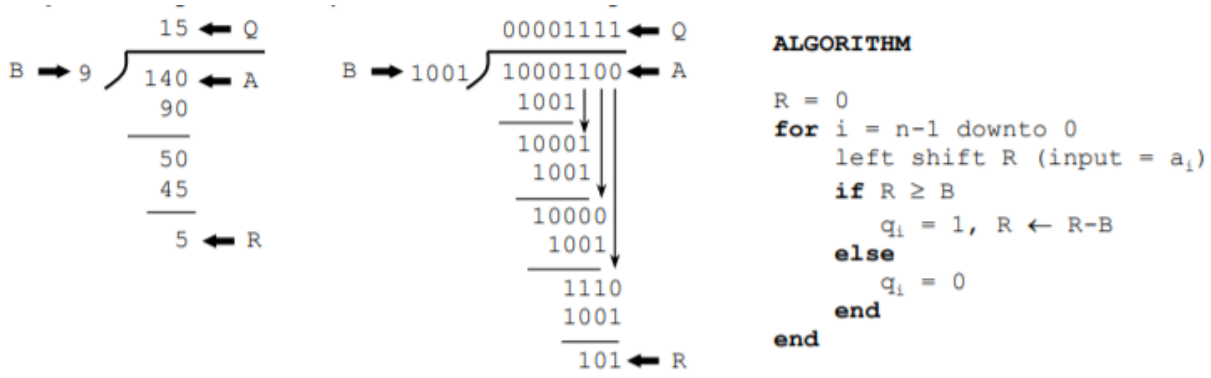

Figure 1. Iterative Division Algorithm

Input *A* must be a 6-bit wide and *B* must be 4-bit wide. A clearable left shift register will store the value *R,* which holds the remainder. A division is started when an input, *E,* receives a value of '1'. This triggers a capture of inputs *A* and *B,* and after every clock cycle the circuit will either

shift in the next bit of *A* or shift in the next bit of *A* and subtract *B.* The output signal *done* will indicate that the division operation has been successfully completed, and that the values of *Q* and *R* are valid. **Figure 2** and **Figure 3** below detail the architectural design and the FSM states to be implemented.

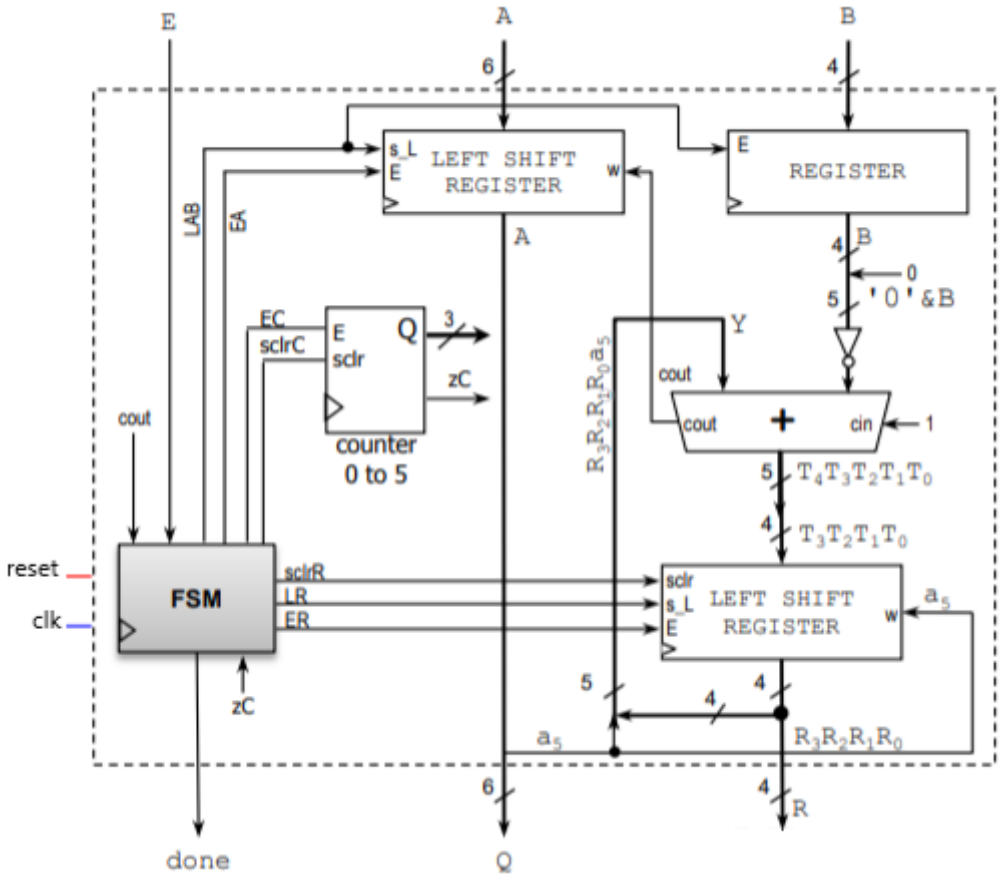

Figure 2. Architectural Schematic

Note:

- The left shift registers will be provided to you. The left shift register on the bottom right of the schematic includes an input port, *sclr*, that clears the register outputs when *E* = 1 and *sclr* = 1.
- The modulo-6 counter includes a synchronous input, *sclr*, that clears the count when *E* = 1 and *sclr* = 1. The output *zC* is driven high when the counter reaches a value of '5'.
- Every sequential component has *clk* and *reset* inputs.

Hint #1: If the output variable is not explicitly defined in a state, then set it to '0' at the beginning of the state.

Hint #2: Create a reset state as well, where all the output values are set to '0'.

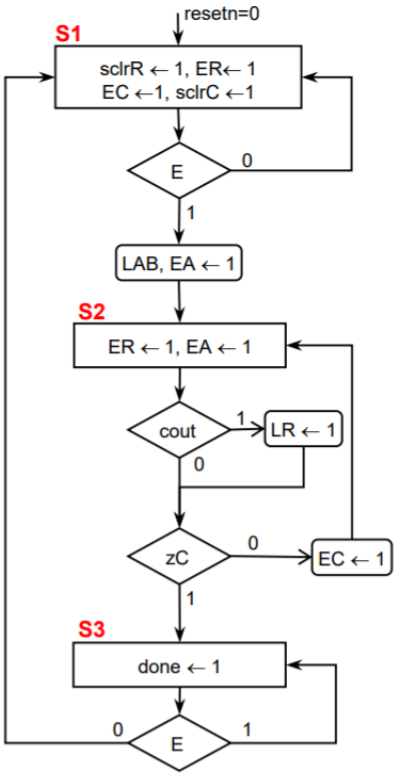

Figure 3. Finite State Machine Design

Steps:

1. Create a project and name it lab4.
2. Select the Nexys A7-100T (xc7a100tcsg324-1) board when prompted to select a target device.
3. Start by adding the left shift register modules from BeachBoard (LShift_Reg.v and LShift_Reg_Clr.v). These registers are used for storing the calculated values of *Q* and *R*, respectively.
4. Create a new module and name it *Register*. Implement a 4-bit register with a load enable signal. Your module must have the following inputs/outputs:
    a. clk: 1-bit input driven by the on-board 100 MHz clock
    b. reset: 1-bit input driven by an on-board switch that serves as the global reset
    c. E: 1-bit input that serves as the load enable for the register
    d. D: 4-bit input that serves as the data to be loaded when E = 1
    e. Q: 4-bit output of the register
5. Verify the functionality of your *Register* module by performing a simple simulation.
6. Create a new module and name it *FA_5bit*. Implement a Full Adder circuit that adds two 5-bit inputs. Your module must have the following inputs/outputs:
    a. A: 5-bit input that serves as operand A

   b. B: 5-bit input that serves as operand B
   c. cin: 1-bit input that serves as the carry-in
   d. Sum: 5-bit output that serves as the result of the addition of the full adder
   e. cout: 1-bit output that serves as the carry-out of the full adder
7. Verify the functionality of your *FA_5bit* module by performing a simple simulation.
8. Create a new module and name it *Counter*. Implement a modulo 6 counter, meaning that your counter will count up to 5 before cycling again. Your module must have the following inputs/outputs:
   a. clk: 1-bit input driven by the on-board 100 MHz clock
   b. reset: 1-bit input driven by an on-board switch that serves as the global reset
   c. E: 1-bit input that will enable the counting behavior
   d. sclr: 1-bit input that will clear the value of the counter when slcr = 1 and E = 1
   e. Q: 3-bit output that hold the current value of the counter
   f. zC: 1-bit output that is asserted when the counter reaches a value of '5'
9. Verify the functionality of your *Counter* module by performing a simple simulation.
10. Create a new module and name it *FSM*. This module will serve as the control unit to your circuit. The state diagram for this module is shown in **Figure 3**. Your module must have the following inputs/outputs:
    a. clk: 1-bit input driven by the on-board 100 MHz clock
    b. reset: 1-bit input driven by an on-board switch that serves as the global reset
    c. E: 1-bit input driven by an on-board switch, signaling the start of a division operation
    d. cout: 1-bit input driven by the *cout* output of *FA_5bit*
    e. zC: 1-bit input driven by the *zC* output of *Counter*
    f. LAB: 1-bit output that drives the s_L input of *LShift_Reg*, as well as the E input of *Register*
    g. EA: 1-bit output that drives the E input of *LShift_Reg*
    h. EC: 1-bit output that drives the E input of *Counter*
    i. sclrC: 1-bit output that drives the sclr input of *Counter*
    j. sclrR: 1-bit output that drives the sclr input of *LShift_Reg_Clr*
    k. LR: 1-bit output that drives the s_L input of *LShift_Reg_Clr*
    l. ER: 1-bit output that drives the E input of *LShift_Reg_Clr*
    m. done: 1-bit output that will indicate when the division operation has been completed
11. Verify the functionality of your *FSM* module through simulation. Ensure that the output signals match the state diagram at every state transition.
12. Create a new module and name it *Iter_Div*. Implement the circuit in **Figure 2** structurally by creating instances of all of your design modules and making the necessary interconnections. Note the following design features:

a. The output Q of *Register* is concatenated with an MSB '0' to create a 5-bit value, then it is inverted before it is connected into input B of *FA_5bit*.
b. Input A of *FA_5bit* is driven by a concatenation of output Q of *LShift_Reg_Clr* as the MSB 4-bits and output Q[5] of *LShift_Reg* as the LSB, thus creating a 5-bit value.
c. Input cin of *FA_5bit* is hardwired to a value of '1'

13. Verify the functionality of your *Iter_Div* module through an extensive simulation. Clearly define your test cases and print the result of each test case in the console by using the following **$display** function after the successful completion of each test case:

```
Initial begin
        $timeformat(-9, 1, " ns", 9);
        // Test cases go here
        .
        .
        .
        // Test case 1 done
        $display("time=%t A=%2d B=%2d || done=%0d Q=%2d R=%2d",
              $time, A, B, done, Q, R);
        .
        .
        .
        // Test case 2 done
        $display("time=%t A=%2d B=%2d || done=%0d Q=%2d R=%2d",
              $time, A, B, done, Q, R);
        .
        .
        .
endmodule
```

It is recommended that you become familiar with using loops in Verilog to create test cases in an iterative approach. You may also use the **$urandom** function to create random unsigned values for *A* and *B*.

14. Add the NexysA7-100T.xdc constraints file from Beachboard into your project. The constraints file maps the various inputs and outputs of your top module to the components for which they are meant to drive. Modify the constraints file to make the following connections on your board:
    a. clk: connect to the on-board 100MHz clock
    b. reset: connect to Switch 0 (pin J15)
    c. E: connect to Switch 1 (pin L16)

   d. A: connect to Switches 7 through 2 (pins R13 – M13)
   e. B: connect to Switches 11 through 8 (pins T13 – T8)
   f. done: connect to LED 15 (pin V11)
   g. Q: connect to LEDs 5 through 0 (pins V17 – H17)
   h. R: connect to LEDs 10 through 7 (pins U14 – U16)

15. Generate the bitstream. Under the Flow Navigator pane, click on **Generate Bitstream** to generate the bitstream that will map your design to the FPGA. You will be prompted to run the synthesis and implementation steps that are required to generate the bitstream. Leave all the prompts with the default selections and continue.
16. Program your device. Connect your Nexys board to a USB port on your computer and power it on. Connect to the board by clicking on **Program and Debug > Open Hardware Manager > Open Target > Auto Connect** to automatically connect to your board. Once connection has been achieved, click on **Program Device** to program the board with the bitstream that you have generated. Observe behavior of your design on the board.

Lab Deliverables:

Submit a brief report including the snapshots of your simulation waveforms, and the snapshots of your running board. Also, you need to demonstrate your work and the completed steps in the demo time. Upload your solutions (one .zip file) including a lab report (.pdf file) and LShift_Reg and LShift_Reg_Clr modules, Register module and testbench, FA_5bit module and testbench, Counter module and testbench, FSM module and testbench, and Iter_Div module and testbench (.v files) to the Dropbox labeled Lab 4.

Submission (Report & Verilog files) due date: Nov 2nd 2020, 11:30PM

Demo: Nov 3rd 2020, 6:00PM - 7:15PM & Nov 5th 2020, 6:00PM - 7:15PM