E85 Lab 5: Digital Building Blocks

E85 Spring 2016
Due: 3/9/16

Overview:

This lab focuses on the creation of digital systems using a hierarchical building block approach. In this lab you will learn how to implement ideas into circuits rapidly using building blocks. You will learn how to test a building block completely using a self-checking testbench. Finally you will create a complex building block in Verilog and test its operation both in simulation and physically.

What to turn in:

1. Your answers to the questions in part A
2. Your testvectors and answers to the questions in Part B
3. Your code for your module and simple testbench for Part C
4. The waves for the requested OT/P case for Part C
5. Answers to the verification questions in Part C
6. The testvector file used with the self-checking testbench in Part C

Feedback:

Let us know:
What went well in this assignment?
Were there points of confusion?
How long did you need to work on this assignment?
Part A – Building Blocks

(1) Create the following function using a 4:1 Multiplexer and no additional gates:

\[ Y = AB + \overline{A} \overline{B} \]

(2) Create a circuit that checks the value on a 32-bit address bus, if the address is 0x00000001 then it saves the value on a 32-bit data bus to a register on the next clock edge.

First draw the circuit as building blocks, then write a short piece of Verilog code to implement your circuit. If you collapse the hierarchy of your circuit your code should be very short.

(3) Create a 32-bit counter that can count either up (0, 1, 2, 3…) or down by 1 (3, 2, 1, 0 …) You may only use one adder that does not have a carry in.

Write a Verilog code that makes the circuit.

(4) Let A and B be 32-bit numbers. Create a circuit that computes:

First clock edge: \( B = A - B \)

Second clock edge: \( B = A \& B \) (second clock)

Third clock edge: \( B = A \mid B \)

Fourth clock edge: \( B = A + B \)

You do not care what happens after the 4th cycle. The result (B) should change on each clock cycle. You may assume both A and B have something interesting in them to start with. Use only one ALU.

Only draw the circuit that performs the computation. You do not need to write the any Verilog code. You are welcome to use higher level building blocks as long as their function is clear.
Part B – Tutorial on Self-Checking Testbenches

In previous labs you have seen how to create simple testbenches and have performed ad-hoc testing to verify that our modules work.

When you are creating a building block that you wish to reuse for all time, it is wise to fully check the function of the block.

If the block contains more than just a few bits this can quickly become impossible for you to accomplish using waves and the techniques we have used up to this point.

In this tutorial we will use a fully programmatic self-checking testbench to fully test an 8-bit ALU.

Before writing the testbench we must of course create the module. In this case it is an ALU, the heart of a computer.

The ALU we will create is shown above. It is a 4-function ALU which takes arguments that are N bits in width. This figure above is 5.17 in the text. The authors have assumed the ALU is 32-bit in certain places but we will not make this assumption with our code.
module alu #(parameter width = 32)
    (input logic [width-1:0] a, b,
    input logic [1:0] ALUControl,
    output logic [width-1:0] Result,
    output logic [3:0] ALUFlags);

logic Cout;
logic [width-1:0] sum, bchoice, iResult;

// 2's Complement add/subtract
assign bchoice = ALUControl[0] ? ~b : b;
assign {Cout, sum} = bchoice + a + ALUControl[0];

// MUX
always_comb
    casez(ALUControl[1:0])
        2'b0?: iResult = sum;
        2'b10: iResult = a & b;
        2'b11: iResult = a | b;
    endcase

// N, Z, C, V Flags
assign ALUFlags[0] = (~(a[width-1] ^ b[width-1] ^ ALUControl[0]))
    & (~ALUControl[1]) & (sum[width-1] ^ a[width-1]);
assign ALUFlags[1] = Cout & (~ALUControl[1]);
assign ALUFlags[2] = (iResult[width-1:0] == 0);
assign ALUFlags[3] = iResult[width-1];

// Connect internal node to output
assign Result = iResult;
endmodule

We can see in the above module that it nicely matches the circuit drawn. We have defined the module parametrically to allow for any width. To match the diagram we have set the default width to be 32-bits. To avoid any possibility of this design causing the synthesizer to create more than one adder we have written only a single line with addition. Although there are two addition signs we must trust (but verify) that the synthesizer will be smart enough to use the carry-in for the single bit.

Although it is possible to create use the output Result internally in a Verilog module. This is not possible in VHDL, so I generally avoid using lines marked at “outputs” as “inputs” and vice versa when writing HDL.

Get familiar with the ALU code above, and convince yourself that it makes the ALU circuit shown in the previous page before continuing.
We will now create a testbench to check that our ALU is bug free. Bugs do get into hardware: Intel’s original Pentium had an error in the floating point unit. Intel had to recall the affected CPUs!

We will test the 8-bit ALU fully. With 8-bits, 2 operands, and 4 functions we must check $256^2 \times 4 = 262,144$ combinations. If you could check one combination every second visually it would only take you a little over 3 days, or 9 days if you work only 8 hours a day. For a 32-bit adder it would take you 1000 years. At a billion checks per second, a computer would finish in just over 30 seconds. Clearly we should leave this to a computer.

To start we will build the Verilog module to check the ALU, you can get the full code here:

http://pages.hmc.edu/bbryce/E85S16/files/testbenchALU.sv

Let’s walk through the code piece by piece.

```verilog
module testbenchALU();
    logic clk, reset;
    logic [7:0] a, b, Result, expectedResult;
    logic [3:0] ALUFlags, expectedFlags;
    logic [1:0] ALUControl;
    alu #8 dut(a, b, ALUControl, Result, ALUFlags);

As usual in a test bench we start with the internal signals we will need and the signals we will feed to the device under test (DUT). The items we will want to check are prefixed with expected. So we will be comparing the Result to the expectedResult. The ALU we create here is 8-bits because of the #8 parameter that is passed.

    //Format of testvectors is: a_b_ALUcontrol_expectedResult_expectedFlags
    // bit per signal 8_8_2_8_4 = 30
    logic [29:0] testvectors[255*255*4 + 1];
    logic [31:0] index, testpassed;

    The next block of code creates an array of buses called testvectors. It is 30-bits wide and contains enough slots to hold every test we might want and 1 more. We will be loading all of the tests we want to do into the testvectors array! To look up individual tests we will need a counter, which we have called index. We also want to keep track of how many tests we have passed, so we create a signal called testpassed.

    Our next step is to create a clock:

    // generate clock with 10 ns period
    always begin
        clk <= 1; # 5; clk <= 0; # 5;
    end

Our module under test (dut) here is actually purely combinational so in principle a clock is not actually needed but for most testbenches of this sort you would want one to cause the contents of registers to update. Here we choose to use a fairly standard form.
With the clock setup and running we can now setup our testing:

//initial setup
initial
begin
  $readmemb("alutest.tv", testvectors);
  index = 0;
  testspassed = 0;
  //reset isn't really used, but kept for FSM type modules
  reset = 1;
  #26;
  reset = 0;
end

Generally this starts with an initial block like the one above. This is run once at the start of the simulation. We have seen this before in simpler testbenches. The important new item here is the $readmemb(). This is a macro that reads the files in the quotes as binary and puts the result into testvectors. This is what loads all the data we need to check.

The format of our alutest.tv is: a_b_ALUcontrol_expectedResult_expectedFlags. The first five lines of our test file are:

1000000_1000000_00_00000000_0111
1000000_1000000_01_00000000_0110
1000000_1000000_10_10000000_1000
1000000_1000000_11_10000000_1000
1000000_1000000_00_00000001_0011

Using the above as a template write 4 testvectors that you think would be interesting to test. Save the result as INITIALS_alutest.tv.

In the file we have separated each field by an underscore character. This is ignored in Verilog but helps us to be able to read the file.

Now that the tests are loaded into testvectors we can apply the tests to our module and check its response.

  //change inputs after positive clk edge
  always @(posedge clk)
  begin
    #1; //wait until just after clock
    {a, b, ALUControl, expectedResult, expectedFlags} = testvectors[index];
  end

If we change the inputs exactly on the edge of the clock it might be confusing if it is 0 or 1 at that edge. So we simulate the propagation of the signal from some theoretical flip-flop with the #1 delay. At each rising edge of the clock we assign the value in the testvector currently under test to the internal signals that match it. Note that the width of the bracketed quantity exactly matches the width of the testvector. If we had used hex (and $readmemh) in our testvector file we would have to be more careful with the alignment as 30 is not a multiple of 4; this is common error!
Our last step is to check the results against what we expect.

The long statements make the above a little harder to read than the previous code blocks. Reading it in an external editor from the full .sv file may help.

```verilog
//check outputs at negative clock edge
always @(negedge clk)
    if (~reset) // skip checking during reset
        begin
            if (Result !== expectedResult)
                begin
                    $display("Result error at index: %d, {a, b, ALUControl} = {%b, %b, %b}", index, a, b, ALUControl);
                    $display("expected/got: %b,%b", expectedResult, Result);
                end
            if (ALUFlags !== expectedFlags)
                begin
                    $display("Flags error at index: %d, {a, b, ALUControl} = {%b, %b, %b}", index, a, b, ALUControl);
                    $display("expected/got: %b,%b", expectedFlags, ALUFlags);
                end
        end
        if (Result === expectedResult & ALUFlags === expectedFlags)
            testspassed = testspassed + 1;
            index = index + 1;
    //end testing when invalid input in testvector
    if(testvectors[index][0] === 1'bx)
        begin
            $display("Tests attempted: %d", index);
            $display("Tests passed: %d",testspassed);
            $stop;
        end
end
```

Essentially this code checks that the expected outputs are correct at the negative edges of the clock. It does not do any checking while reset = 1. If something is not correct it prints warning messages to the console that explain the error.

The statement: if(testvectors[index][0] === 1'bx) checks if the next testvector that would be run is valid. It does this with the foreknowledge that the initial contents of the register would be all Xs (unknown). We could check the vector is 30 Xs but we know here that we will never supply any vectors with X values so checking the first bit ([0]) is adequate. The $stop macro halts the simulation. There is also the $finish macro which also exits the simulation.

Finally note that one can inspect internal signals in instanced modules by using the dot operator.

For instance dut.Cout would give us access to the internal signal Cout of the ALU instance dut.
Download module and testbench codes and create a ModelSim project.

http://pages.hmc.edu/bbryce/E85S16/files/alu.sv

http://pages.hmc.edu/bbryce/E85S16/files/testbenchALU.sv

Modify the testbench code to read the short testvector file you created earlier. Run the simulation with waves for your short testvector file and note how many of your tests passed. If any failed correct your testvector file.

If you modified the the #1 delay in the posedge block, to be #8 would the testbench work? Why or why not?

Download alutest.tv:

http://pages.hmc.edu/bbryce/E85S16/files/alutest.tv

This file was generated by the following python program:

http://pages.hmc.edu/bbryce/E85S16/files/gen.txt

(Change the extension to .py if you wish to use it, the pages server does not allow this to be served with the *.py extension).

You do not need to include the .tv file in the ModelSim project just have it in the same directory as the testbench. If ModelSim give an error not being able to find the .tv file, provide the full path to the file rather than just the filename in the loading macro.

Modify the testbench to read the downloaded file and run the simulation for 3,000,000 time units to verify that the ALU passes all the tests.
Part C – Create a Pulse Wave Modulation Module

Pulse width modulation (PWM) is a very common peripheral available on microcontrollers and systems-on-a-chip (SoC). In the datasheet for the microcontroller used on the board we designed in lab 1 you will see PWM listed as a feature that makes use of counters. For this part of the lab we will design a hardware PWM module, and use it to modulate the brightness of an LED.

PWM can be characterized by a period (use label: \( P \)) and on time (use label: \( OT \)). For our design let the PWM maximum period be \( 2^{32} \) clocks, and thus settable as a 32-bit value. The time the PWM pin is logic high is the on time. For our design this will also be settable as a 32-bit value. Let the output of the PWM be called \( PWMOut \). Store the values of \( P \) and \( OT \) in registers that utilized enable pins.

Three periods of the \( PWMOut \) signal are shown below. The horizontal axis is time, the high value is logic 1.

First, draw a circuit using building blocks that will create the \( PWMOut \) signal shown.

Next, write Verilog code to implement the modules you drew in your circuit. Create the overall PWM with a single parent module:

```verilog
module pwm(input logic clk, reset, enableP, enableOT,
            input logic [31:0] OT, P,
            output logic PWMout);
```

Test your PWM module with a self-checking testbench. Although we checked all cases in the tutorial for this lab you do not need to check all cases, check a reasonable number.

Write a simple testbench to check the basic function of your PWM module.

Plot the waves: \( CLK \), \( PWMOut \), \( OT \) and \( P \) for your PWM module for \( OT = 2 \) and \( P = 5 \).

The output should be high for exactly 2 clocks low for 3 clocks repeatedly for this condition.

Download the following testbench and verify your pwm module using at least 4 reasonable tests:

http://pages.hmc.edu/bbryce/E85S16/files/testbenchPWM.sv

Does your module pass the tests?
Use the following wrapper code to your .sv file to allow you to physically test the PWM module on the DE2 board:

```verilog
module pwmlight(input logic [17:0] SW, 
               input logic CLOCK_50, 
               output logic [0:0] LEDR);

logic reset, clk, light, enableOT, enableP;
logic [17:0] SSW, S1;
// Synchronizers for switches
always_ff @(posedge clk)
   begin
      S1 <= SW;
      SSW <= S1;
   end
assign LEDR[0] = light;
assign clk = CLOCK_50;
//Reset is 11, enableOT is 10 enableP is 01, 00 does nothing
assign reset = SSW[17] & SSW[16];
assign enableOT = SSW[17] & ~SSW[16];
assign enableP = ~SSW[17] & SSW[16];

// Create the PWM module
// Use middle bits to create both large and small count values
pwm pwmLED(clk, reset, enableP, enableOT, {6'b0, SSW[15:0], 10'b0},
           {6'b0, SSW[15:0], 10'b0}, light);
endmodule
```

You can change the top module for your project by going to Assignments->Settings->General

Change the value of Top-Level entity to the wrapper (pwmlight).

Synthesize the code and program the DE2 with your PWM module. Do not forget to load those pin assignments!

Notice the wrapper code uses some of the bits in the middle of the 32-bit input.

Verify that your PWM output allows you to:

Blink the LED at different periods and for different on times. Are you able to see it blink?

 Allows you to create a dimming effect when the period of blinking is too fast to see with the human eye. Are you able to get a dimming effect?

To check the blinking try setting P to SW[15] high only, and OT to SW[14] high only.

To check the dimming try setting SW[7:0] = 1 for P and then set OT to various smaller values.

Extra credit (10%):

Create a module that contains the PWM that makes the LED pulse in brightness instead of just blinking on and off, you do not need any external settings other than perhaps reset.

Run your test physically.