University of California, Davis
Department of Electrical and Computer Engineering

EEC180B DIGITAL SYSTEMS II Spring Quarter 2015

LAB 5: Signed Division

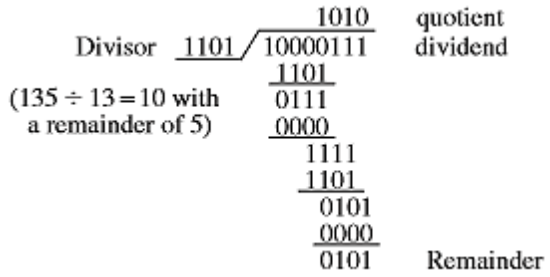
Objective: In this lab, you will design an integer divider and verify its power consumption and performance. A fraction of the grade will depend on optimizing the power consumption and performance of your design, which means you need to design an efficient datapath and controller before writing the Verilog code. Your design should be *synthesizable* but you do not have to download the design to the board.

This is a two-week lab and you are allowed to work with a partner. The partner should be from your lab section. Both partners will get the same score.

I. Signed Division

Design a divider for signed (2's complement) binary numbers that divides a 32-bit **dividend** by a 16-bit **divisor** to give a 16-bit **quotient and** 16-bit remainder.
Although algorithms exist to divide signed numbers directly, they are complicated and take up lot of resources especially on an FPGA. Therefore, we will develop an implementation for dividing unsigned numbers and then negate the dividend, divisors, and results appropriately based on the sign of the dividend and the divisor.

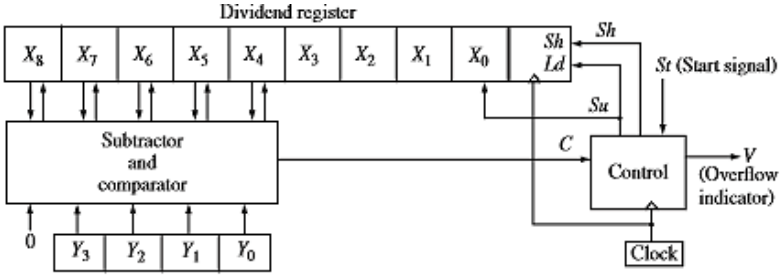
So, what is a good algorithm for dividing unsigned numbers? It turns out that unsigned division can be performed with very simple hardware - counter, shift register, subtractor, negation (2's complement computation) unit, and a finite state machine with a handful of states using the familiar paper-and-pencil algorithm that you learned when you were in elementary school.
Here it is



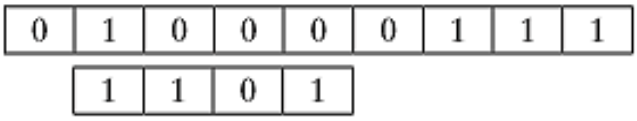
Here is the high-level algorithm for unsigned division.
1. Shift dividend left.

2. Enter the quotient bit by bit into the right end of the dividend register as dividend is shifted left.
3. If divisor is going to be negative, shift dividend one place to the left before subtraction.
4. Subtract to get new dividend.
5. Shift dividend left. If result will be negative, repeat shift then subtract.
6. Perform final shift.
7. If the calculated quotient requires more bits than what is available in the quotient register , an *overflow* has occurred.

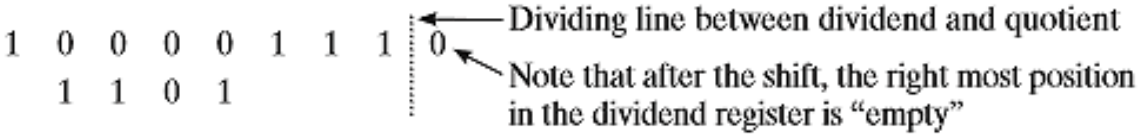
How does it work? The following is a division example for a 9-bit dividend, 4-bit divisor and 4-bit quotient.



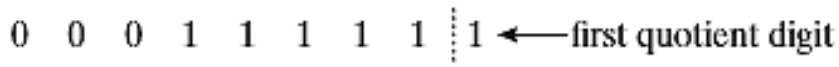
The 9-bit dividend register and the 4-bit divisor register are initialized as follows:



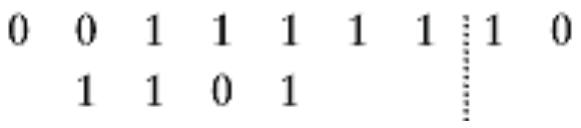
Shift dividend left



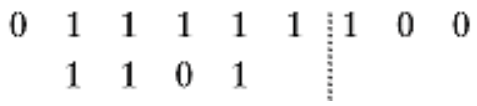
Subtract. First quotient digit of 1 is stored in the unused position of the dividend register:



Shift dividend one place left:

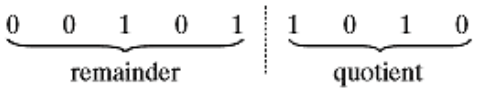


Subtraction would yield a negative result, so shift dividend left again (2nd quotient remains zero):



Subtract. Third quotient digit of 1 is stored in the unused position of the dividend register:

Final shift. Fourth quotient bit is set to 0.



Here is the skeleton for your Verilog code. It should have this interface.

```
module signdiv (CLK, St, Dbus, Quotient, Remainder, V, Rdy, rst);
module  signdiv (quot, rem, done, dbus, clk, rst, start);

output [15:0] Quotient;
output [15:0] Remainder;
output St;
output V; // asserted when there is an overflow, (Extra Credit)
input   [15:0 ] Dbus;
input CLK;
input rst;
input Rdy;

// Your Verilog code here.  Use Behavioral Code.

....

endmodule
```

Principle of Operation

When **St** signal is high, the 16-bit input **Dbus** is used to load the 32-bit dividend and the 16-bit divisor in 3 cycles (on the rising edge of the clock) into the internal registers. When **St** signal goes low, your design should perform the division and when the result is ready, assert the **Rdy** signal and output the quotient and remainder of the division on the **Quotient** and **Remainder** buses.

A reset signal (**rst)** should also be provided to initialize everything especially the state machine to its initial state before commencing the operation.

Design Requirements

1. Your design must be *synthesizable*.
2. A sample testbench is provided with the laboratory exercise. You should use the testbench to verify your design. It is your responsibility to add other testvectors to the testbench to ensure that design works correctly for all legal inputs including positive and negative values (however, you don't have to worry about the divisor being a 0.)
3. Your design should be as efficient as possible. Efficiency of the design is measured in terms of the number of resources (LE's or logic elements and DFFs) used and the clock period reported by the Quartus synthesis tools.
4. You will be estimating the power consumption of the design as well and that is also a measure of how good your design.
5. You have to use the shift and subtract algorithm for this exercise. You cannot use any other algorithm or prebuilt division modules from the Altera's library or elsewhere. You will get a zero if you do so.
6. There is no need to instantiate any components in this exercise, everything you need can be inferred through behavioral constructs (always and assign statements).

What do you need to do?

1. First, make sure you understand the paper and pencil algorithm for unsigned division. As part of prelab workout two examples of division algorithm along the lines of the example described here. The first example is -200/15 and the second is the first four digits of your Student ID by 5. You should clearly indicate what happens in each step.

2. Design the datapath and controller for the design. Use the Booth multiplier that you did in Lab 3 as an example. Draw the state diagram for the controller. This is part of the prelab.

3. Read the testbench carefully and make sure you understand how it works. You can modify it *slightly* if it does not meet your design exactly. For example, adding the reset, or modifying when the inputs are actually applied and resulted sampled

4. Write the Verilog code and simulate your design with the testbench provided to makes sure the design works.

5. Synthesize your design and generate the resource usage and timing reports.

6. Demonstrate the working design to the TA along with the simulation waveforms and synthesis reports. The TA may ask you to modify the testbench to test your design with different input values.

III. Power and Energy

In a CMOS circuit, there are two sources of power consumption – dynamic power that is proportional to the clock frequency and square of the supply voltage and static power that is largely independent of the design in the case of an FPGA. Dynamic power itself has three sources – (a) I/O pads (b) FPGA core and (c) clocking network.

Power is the average energy per unit time is reported in watts (joules/sec). One could reduce the power by performing the same operation very slowly that is spending lot of time to do the same operation. However, that is not a good measure for the *efficiency of a circuit*, because in general we are interested in improving both power consumption and performance of a circuit. Also, in a battery-operated device, we are interested in prolonging the battery life, which means "energy" is a better metric for the efficiency of a circuit.

In general, we are interested in energy per operation, the operation being one 32 bit by 16 bit division operation in this lab exercise. You can compute this by multiplying the average power (reported by the Power Play tool) with the time required to complete a division operation. The latter depends on your specific implementation.

IV. Lab Report

For your lab report, include the following:

- Lab Cover Sheet with signed TA verification.
- Complete Verilog source code for the divider.
- Simulation waveforms of your timing simulations.
- State transition diagram for your multiplier controller state machine.
- Resource report indicating how many FPGA resources were required for each the designs.
- Power consumption report of your division

Extra Credit

If, as a result of a division operation, the quotient contains more bits than are available for storing the quotient, we say that an overflow has occurred. For instance, in the example described above, if the divisor was 7 instead of 13 we should report an overflow because 135/7 will result in a quotient of 19, which cannot be represented with 4 bits.

However, remember that in this design exercise we are dealing with 2's complement numbers. So, the 16-bit quotient includes a sign bit as well, which means the range of quotient values is -2^{15} to $(2^{15}-1)$.

Add a new output called V to your design module. V should be asserted when there is an overflow.

Points Breakdown

Milestone	**Points**
Prelab	**25**
Functional Simulation	**75**
Synthesized Design with detailed timing and resource report	**50**
Power Analysis	**25**
Design Optimization	**25**
Extra Credit	**50**

Testbench

```
module testsdiv;
parameter N = 12;
reg[31:0] dividendarr[1:N];
reg[15:0] divisorarr[1:N];

//inputs to sdiv module should be reg types
reg CLK;
reg St;
reg[15:0] Dbus;
wire[15:0] Quotient;
wire[15:0] Remainder;
wire Rdy;
reg[15:0] Divisor;
reg[31:0] Dividend;
reg[3:0] Count;
reg[31:0] dividendarr_tmp;
reg [15:0] quotientarr[1:N];
reg [15:0] remainderarr[1:N];

integer i;
always
begin
  #10 CLK <= ~CLK;

initial
 begin
//initialization of dividend array
dividendarr[1] = 32'h0000006F;
dividendarr[2] = 32'h07FF00BB;
dividendarr[3] = 32'hFFFFFE08;
```

```
dividendarr[4] = 32'hFF80030A;
dividendarr[5] = 32'h3FFF8000;
dividendarr[6] = 32'h3FFF7FFF;
dividendarr[7] = 32'hC0008000;
dividendarr[8] = 32'hC0008000;
dividendarr[9] = 32'hC0008001;
dividendarr[10] = 32'h00000000;
dividendarr[11] = 32'hFFFFFFFF;
dividendarr[12] = 32'hFFFFFFFF

//initialization of divisor array
divisorarr[1] = 16'h0007;
divisorarr[2] = 16'hE005;
divisorarr[3] = 16'h001E;
divisorarr[4] = 16'hEFFA;
divisorarr[5] = 16'h7FFF;
divisorarr[6] = 16'h7FFF;
divisorarr[7] = 16'h7FFF;
divisorarr[8] = 16'h8000;
divisorarr[9] = 16'h7FFF;
divisorarr[10] = 16'h0001;
divisorarr[11] = 16'h7FFF;
divisorarr[12] = 16'h0000;

//initiaiization of remainder array
remainderarr[1] = 16'h0006;
remainderarr[2] = 16'h00C5;
remainderarr[3] = 16'hFFE8;
remainderarr[4] = 16'hF2F2;
remainderarr[5] = 16'h7FFF;
remainderarr[6] = 16'h7FFE;
remainderarr[7] = 16'h7FFF;
remainderarr[8] = 16'h0000;
remainderarr[9] = 16'h8002;
remainderarr[10] = 16'h0000;
remainderarr[11] = 16'hFFFF;
remainderarr[12] = 16'h0000;


 //initialization of remainder array
  quotientarr[1] = 16'h000F;
  quotientarr[2] = 16'hBFFE;
  quotientarr[3] = 16'hFFF0;
  quotientarr[4] = 16'h07FC;
  quotientarr[5] = 16'h0000;
  quotientarr[6] = 16'h7FFF;
  quotientarr[7] = 16'h0000;
  quotientarr[8] = 16'h7FFF;
  quotientarr[9] = 16'h8001;
  quotientarr[10] =16'h0000;
```

```
  quotientarr[11] =16'h0000;
  quotientarr[12] =16'h0002;

CLK = 0;
Count = 0;
@(posedge CLK);
@(negedge CLK);

for(i = 1 ; i <= N ; i = i + 1)
begin
St = 1'b1;
dividendarr_tmp = dividendarr[i];
Dbus = dividendarr_tmp[31:16];

@(posedge CLK);
  Dbus = dividendarr_tmp[15:0];

@(posedge CLK);

Dbus = divisorarr[i];

St = 1'b0;

Dividend = dividendarr_tmp[31:0];
Divisor = divisorarr[i];

@(posedge Rdy);
Count = i;
if(quotientarr[i] == Quotient)
begin
  $display("quotient[%d] is correct",i);
 end
else
begin
 $display("quotient[%d] is wrong",i);
end
if (remainderarr[i] == Remainder)
begin
$display("remainder[%d] is correct",i);
end
else
begin
$display("remainder[%d] is wrong",i);
end
end
$display("TESTS DONE");
end
signdiv sdiv (CLK, St, Dbus, Quotient, Remainder, V, Rdy);

endmodule // testsdiv
```