Greatest Common Divisor

1. Synopsis:

The purpose of this lab is to teach RTL design as well as some techniques in the “top” design. Given two 8-bit unsigned numbers, we calculate their greatest common divisor (GCD). We will implement Euclid’s Algorithm for computing GCD as a Mealy machine using one-hot state encoding. We will then implement the design on Xilinx FPGA board. Here, we learn RTL method of design in Verilog (state machine design and data path design in Verilog) and also understanding single-stepping.

2. Introduction:

Prelab: If the GCD algorithm was not discussed in your class, please watch a short webcast on “Small system design example - GCD” GCD.avi GCD.pdf .
Before coming to the lab you are required to complete the state diagram provided in the prelab.

Part 1: In part 1, you will implement the core design in Verilog, create a test fixture, and simulate using Modelsim.

Part 2: In part 2, you are provided with an incomplete top level design with a number of TODO sections. These include exercising single-step clock enable control. You will complete the top level design, download to the board, and verify the results.

When you finish this lab, you are expected to be confident in RTL coding of any given core task and designing a suitable top with single-stepping capability if needed.

3. Algorithm:

Following is the Euclid’s algorithm to compute GCD of two numbers, A and B. The algorithm was reported by J. Stein in 1967 and is available on the web at http://www.nist.gov/dads/HTML/binaryGCD.html (link is perhaps inactive) and also at http://en.wikipedia.org/wiki/Binary_GCD_algorithm.

1. if A = B, then GCD(A,B) = A
2. else if A is less than B, swap A and B
3. else if A and B are even, then the GCD(A,B) = 2 x GCD(A/2,B/2)
4. else if A is odd and B is even, then GCD(A,B) = GCD(A,B/2)
5. else if A is even and B is odd, then GCD(A,B) = GCD(A/2,B)
6. else if both are odd, then GCD(A,B) = GCD((A-B),B)

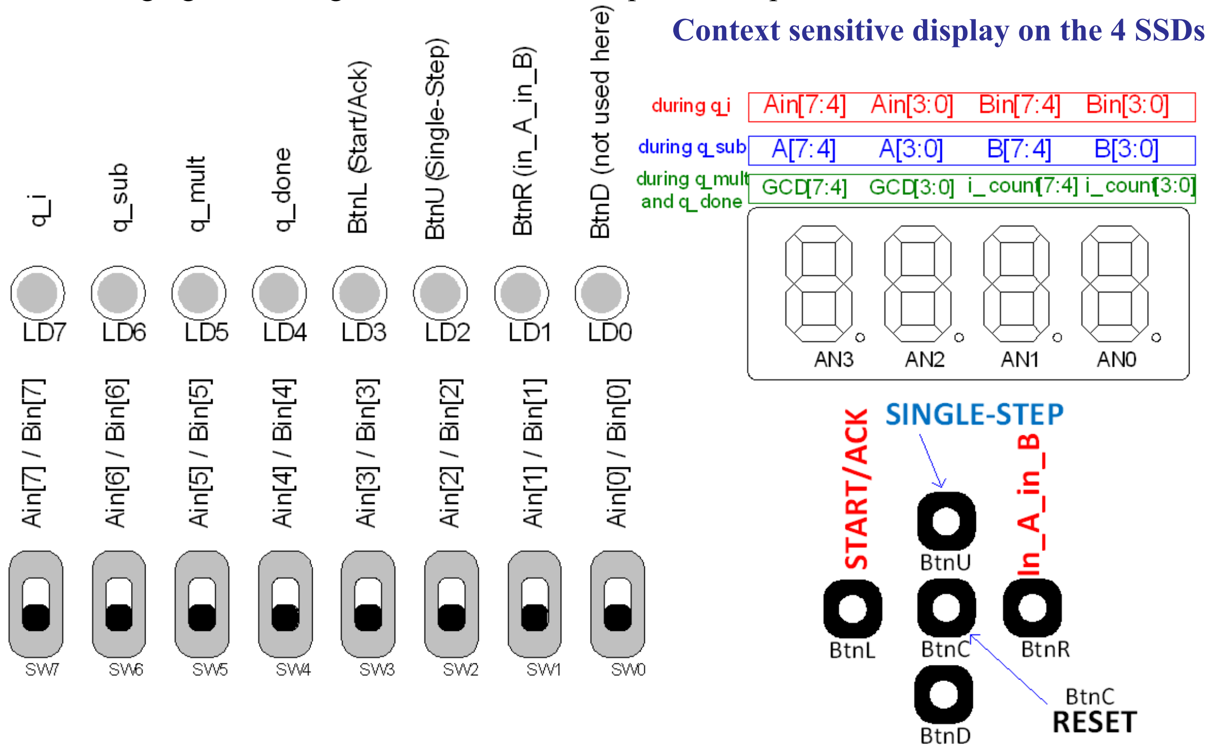
4. Description of the circuit:

In this design, you will implement a state machine along with a data path unit. This circuit will compute the greatest common divisor of two **8-bit unsigned** numbers, **A** and **B**, and output the **GCD** as the result. The eight switches on the board are used to enter the two numbers, one at a time. Though we have 16 switches on the Nexys-4 board, we will use the right 8 switches only (as per the original design of this lab on Nexys-3). Similarly, though we have 8 SSDs on the Nexys-4 board, we will use the right 4 SSDs following our original design of this lab on the Nexys-3 board. The core design will compute the **GCD** when signaled to start and output the result for displaying on the seven segment display.

We have two state machines here, a 2-state state machine in the top and a 4-state state machine in the core. The 2-state top state machine allows us to deposit 8-bit values in two registers, **Ain** and **Bin** , in the top using the same 8 switches, **{Sw7, Sw6, Sw5, Sw4, Sw3, Sw2, Sw1, Sw0}**.

At **RESET** the state machine in the top design goes to (**A_bar_slash_B == 0**) state and awaits insertion of **Ain** . At the *first* press of **BtnR** the value on the right 8 switches will be stored as **Ain** in the *top*, and the state changes to (**A_bar_slash_B == 1**) state. At the *second press of the same button*, **BtnR**, the value on the switches will be stored as **Bin** in the *top*. The (**A_bar_slash_B**) state flips on every operation of the **BtnR**. So, if one enters a wrong value for one of the two, he can cycle through and reenter both values again. These two values will be displayed on the seven segment display. **SSD3** and **SSD2** show the two digits of **Ain** and **SSD1** and **SSD0** show the two digits of **Bin**.

At **RESET** the state machine in the core design goes to **INITIAL** state and awaits **START** signal. **BtnL** is used to signal the core state machine to **START** the computation. Four LEDs, **LD7-LD4**, indicate the current state of the core state machine. When the computation is completed, the value of **A** and **B** on the seven segment display will be replace by the result of computation. **SSD3** and **SSD2** will be used to show the two digits of the **GCD** of **A** and **B**. **BtnL** (which was used to signal **START**) is also used to signal acknowledging of noticing the results of the completed computation.



5. Prelab:

Q 5. 1: Note that in the top level design, we use a `hex-to-ssd` conversion unit to convert the 4-bit binary numbers to their corresponding seven segment code, which is then sent to one of the **`SSDs`**. In this **`GCD`** design, **`A`**, **`B`** and **`GCD`** are 8-bit numbers, instead of 4-bit numbers. To show a 8-bit numbers on a pair of **`SSDs`** do you think we display the numbers in hex digits or decimal (**`BCD`**)? (2 pts)

To display on 4 **`SSDs`**, are you using four `hex-to-ssd` conversion units, or ...(2 pts)

Q 5. 2: Follow the Euclid's algorithm and compute the GCD of the following pairs of numbers. Report your step by step computation.(6 pts = 3 + 3)

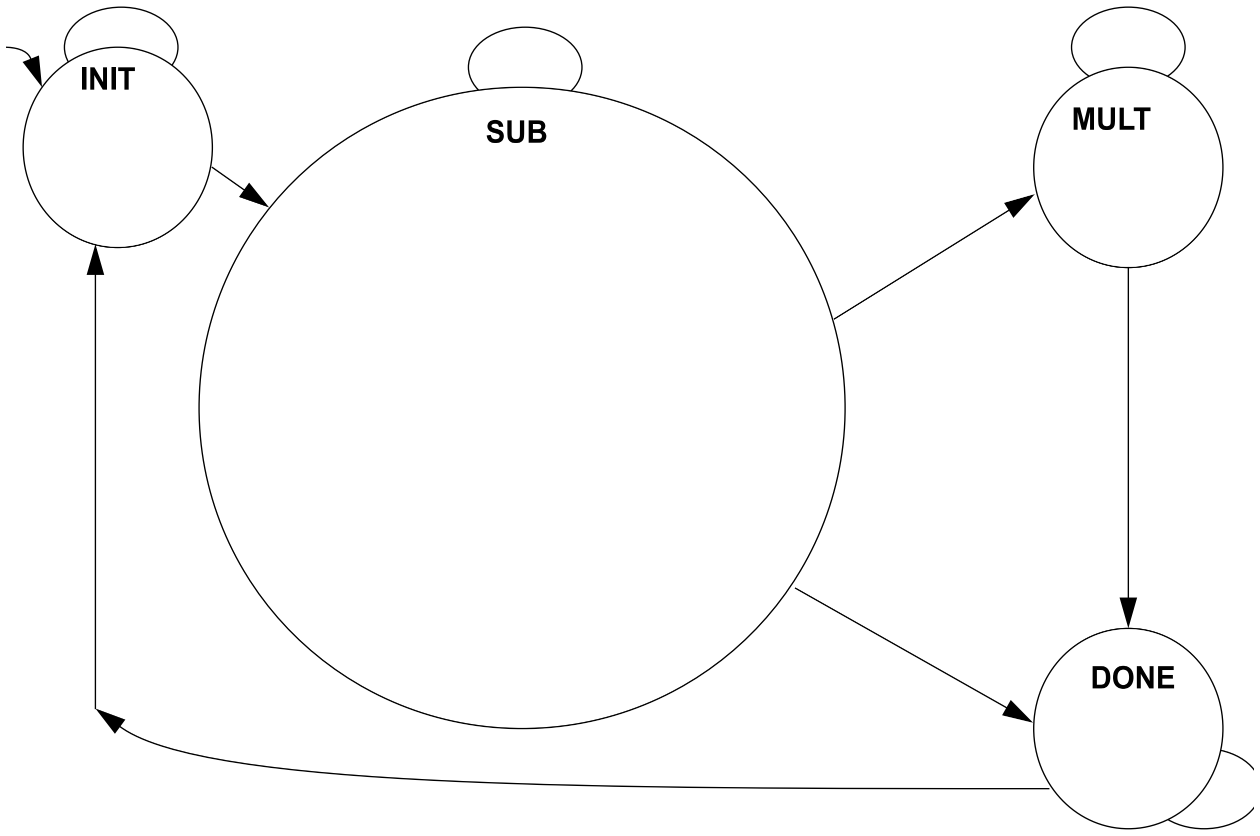
(a) GCD(36,24)

State: Initial	A = 36	B = 24	i = 0
State: Sub	A =	B =	i =
State:	A =	B =	i =
State:	A =	B =	i =
State:	A =	B =	i =
State:	A =	B =	i =
State:	A =	B =	i =
State:	GCD =	i =	
State:	GCD =	i =	
State:	GCD =	i =	

(b) GCD(5, 15):

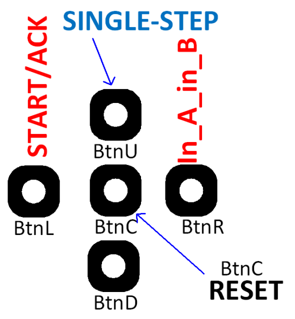
State: Initial	A = 5	B = 15	i = 0
State:	A =	B =	i =
State:	A =	B =	i =
State:	A =	B =	i =
State:	A =	B =	i =
State:	GCD =	i =	

Q 5. 3: If the GCD algorithm was not discussed in your class, please watch a short webcast on "Small system design example - GCD" GCD.avi GCD.pdf . Understand the state machine of the GCD computation thoroughly and complete the following state diagram.(5 pts)

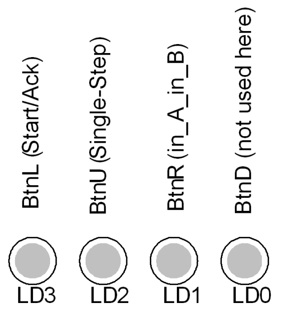


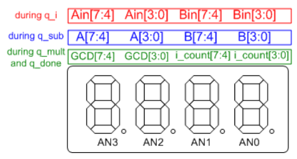
6. Procedure:

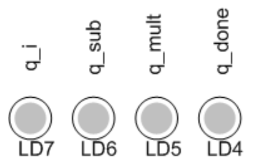
Part 0: Preview the completed design

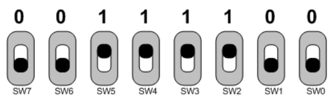


6.1 Download the zip file `ee354L_GCD.zip`. Extract to form a directory `ee354L_GCD` under your `C:\Xilinx_projects`. Open the subdirectory `ee354L_GCD`. Download the bit file `TAs_ee354_gcd_top.bit` to your Nexys 4 board. Operate the switches and buttons and observe LEDs and SSDs to see how your design will behave when completed for the following Ain and Bin.
Ain = **60 (00111100)= 3C hex** and Bin = **84 (01010100) = 54 hex**.

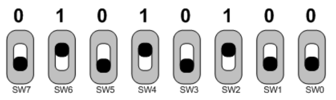








Ain = 60 (00111100)= 3C hex



Bin = 84 (01010100) = 54 hex

```
LD7 is ON to indicate q_i state
1.  on reset                      0 0  0 0  Ain, Bin
2.  set sw[7:0]= 00111100
    and press BtnR (in_AB_pulse)  3 C  0 0  Ain, Bin
3.  set sw[7:0]= 01010100
    and press BtnR (in_AB_pulse)  3 C  5 4  Ain, Bin
Until now we are in q_i state and LD7 is on. Observe
that we now move to q_sub state and the LD6 glows.
4.  press BtnL (start_ack)
    to start                      3 C  5 4  A, B
5.  press BtnU to single-step     5 4  3 C  A, B swapped
6.  press BtnU to single-step     2 A  1 E  A, B halved, i = 1
7.  press BtnU to single-step     1 5  0 F  A, B halved, i = 2
8.  press BtnU to single-step     0 6  0 F  A-B, B
9.  press BtnU to single-step     0 F  0 6  A, B swapped
10. press BtnU to single-step     0 F  0 3  A, B/2
11. press BtnU to single-step     0 C  0 3  A-B, B
12. press BtnU to single-step     0 6  0 3  A/2, B
13. press BtnU to single-step     0 3  0 3  A/2, B
Until now we are in q_sub state and LD6 is on.
Observe that we now move to q_mult state and the LD5 glows.
14. press BtnU to single-step     0 3  0 2  GCD, i_count
15. press BtnU to single-step     0 6  0 1  GCD, i_count
Until now we are in q_mult state and LD5 is on.
Observe that we now move to q_done state and the LD4 glows.
16. press BtnU to single-step     0 C  0 0  GCD, i_count
17. press BtnU to single-step <= nothing new happens as
    the state machine is waiting for ACK.
18. press BtnL (start_ack)
    to ACK                        3 C  5 4  Ain, Bin
Now we are back in q_i state and LD7 glows.
```

Note: The .bit files provided by us bear "**`TAs_`**" prefix. They have the dot points on SSDs lit to differentiate from the .bit files created by the students.

6.2 Also, please go through the **`divider_verilog`** design example provided to you in previous labs.

Part 1: Core design and simulation

6.3 Open the directory `C:\Xilinx_projects\ee354L_GCD`. Open the core design file `ee354_GCD.v` (in Notepad++ preferably). This (`ee354_GCD.v`) is a 'bare bones' module for the GCD core design. Use this file to implement the state machine you completed in the prelab along with the required data path. Recall that the control unit and data path can be coded in the same `always` block. Follow the guidelines below when coding your state machine.

- Name your always and initial blocks. It makes debugging easier. After begin type a colon (**:**) and give a name.
  ```
  always @ (posedge Clk, posedge Reset)
    begin : my_GCD
  ```

- Data registers, which usually need not be initialized at reset, should be assigned X's at reset in order to avoid the tool from inferring an unnecessary re-circulating multiplexer with each such register.
  ```
  if(Reset)
    begin
       state <= I;
       i_count <= 8'bx;
       A <= 8'bx;
       B <= 8'bx;
       AB_GCD <= 8'bx;
    end
  ```

- Although we are coding data path unit together with the control unit, you should separate state transitions from data transitions (under each case branch) by suitable comments to make it more readable.

6.4 Once you have completed the core design, open the test fixture, `ee354_GCD_tb.v`, and complete the missing lines. To do so, you are required to **compute** the *minimum* amount of wait time required for each set of inputs (hint: minimum wait time for each set of inputs depends upon the number of cycles it will take your state machine to compute the GCD for that specific input data set). Compare the results of your simulation with the prelab and show your waveforms to your TA.

Part 2: Top-level design

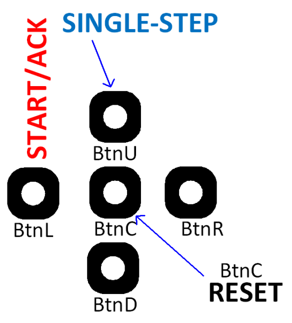
This top design is based on the top design for the divider design example presented to you earlier.

6.5 As a first step, get acquainted with the debouncer design. Refer to the section on debouncer and single-stepping in that example design of a divider provided to you earlier. Please read the state diagram for the debouncer and also read the code `ee354_debounce_DPB_SCEN_CCEN_MCEN_r1.v`. Notice that, in this revised code `ee354_debounce_DPB_SCEN_CCEN_MCEN_r1.v.` , we have increased the time gap between consecutive **`MCEN`** pulses to **`1.342`** sec.

6.6 Read the following two top designs of the divider example design: **`divider_top_with_debounce.v`** and **`divider_top_with_single_step.v`**.

6.7 The following may help you with writing some of the lines in your `ee354_GCD_top.v`
Suppose we combine both the above TOP designs together to make one TOP called, say, **`divider_top_with_debounce_and_single_step.v`** . And, say, the button assignment is as shown on the side. Then the instantiation lines responsible (a) for generating single-clock wide pulses for **`Start_Ack_SCEN`** to work as the **`START`** signal and also later as the **`ACK`** signal and (b) for generating the **`SCEN`** signal to work as single-step-control signal, would look like this:

`divider_top_with_debounce_and_single_step.v`
`(combination of two given designs)`



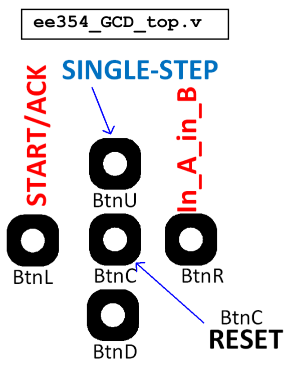
```
ee354_debouncer #(.N_dc(25)) ee354_debouncer_2
       (.CLK(sys_clk), .RESET(Reset), .PB(BtnL), .DPB( ),
       .SCEN(Start_Ack_SCEN), .MCEN( ), .CCEN( ));

ee354_debouncer #(.N_dc(25)) ee354_debouncer_0
       (.CLK(sys_clk), .RESET(Reset), .PB(BtnU), .DPB( ),
       .SCEN(SCEN), .MCEN( ), .CCEN( ));

       // instantiate the core divider design. Note the .SCEN(SCEN)
divider divider_1(.Xin(Xin), .Yin(Yin), .Start(Start_Ack_SCEN),
       .Ack(Start_Ack_SCEN), .Clk(sys_clk), .Reset(Reset),
       .SCEN(SCEN), .Done(Done), .Quotient(Quotient),
       .Remainder(Remainder), .Qi(Qi), .Qc(Qc), .Qd(Qd) );
```

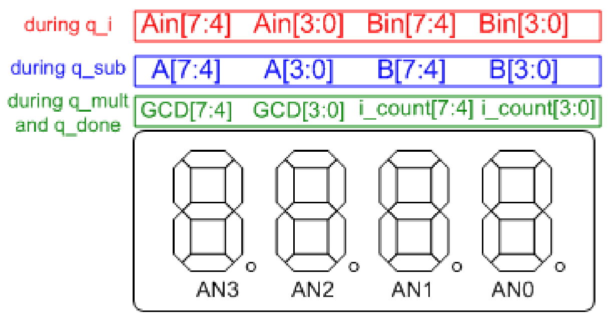
The debouncer design here (in the GCD design) may be slightly different from the debouncer design given with the divider design. Here (in `_r1` version), we have increased the time gap between consecutive **`MCEN`** pulses to **`1.342`** sec. Hence instantiations of the debouncer similar to the above will use here an `N_dc` parameter of `28`: `ee354_debouncer #(.N_dc(`**`28`**`)) ee354_debouncer_2`

6.8 In our `ee354_GCD_top.v`, we intend to use **BtnL** and **BtnU** in a fashion similar to the above divider design.



In addition, we use **BtnR** for depositing **Ain** and **Bin** *in two registers in the top* (note: in the *top*, not in the *core*) called **Ain** and **Bin**. We have only 8 switches on the previous Nexys-3 board. But we need to provide *two* 8-bit operands to the core design. So we have two registers, called **Ain** and **Bin** in the *top* design, and we deposit an 8-bit value from the switches into one of these two registers, one at a time. We set **Ain** value on the switches, press **BtnR** once. Then set the **Bin** value on the same switches, press **BtnR** the second time. This requires that we have a flip-flop to remember if the **BtnR** was pressed once already. So we declared a FF (`reg A_bar_slash_B;`), initialized it under reset (`A_bar_slash_B <= 1'b0;`), and toggled it (`A_bar_slash_B <= ~ A_bar_slash_B;`) every time the **BtnR** was pressed. So, do you think you can use the *bare* **BtnR** signal for toggling or do you need to debounce and produce a single-clock-wide pulse, called, say, **in_AB_Pulse**? Read the incomplete `ee354_GCD_top.v` and complete the "TODO" section related to this. We have a number of "TODO" sections in `ee354_GCD_top.v` for your to complete. Use search/find function in your editor to locate them.

6.9 In another "TODO" section, you are required to write a small piece of code that will assign different values to the SSDs depending on the current state. In the initial state, the SSDs will display the **Ain** and **Bin** being conveyed to the core design. These are the values *deposited* in the **Ain** and **Bin** registers (and not the *raw* values of the switches. In the subtract state, the SSDs will display the changing values of **A** and **B**. In the multiply and done states, the SSDs will display the **GCD** value and 2's counter value.


during q_i
Ain[7:4] Ain[3:0] Bin[7:4] Bin[3:0]
during q_sub
A[7:4] A[3:0] B[7:4] B[3:0]
during q_mult and q_done
GCD[7:4] GCD[3:0] i_count[7:4] i_count[3:0]
AN3
AN2
AN1
AN0

To assign data values to the SSD, you can use the conditional continuous assignment operator (also known as tertiary assignment operator) as shown in the example below (this implements a 2:1 mux):

```
assign y = s ? i1 : i0;
```

Note: You can define a 4 to 1 mux using a nested operations. looking like:

```
assign y = s1 ? (s0 ? i3: i2): (s0 ? i1: i0);
```

6.10 Make sure that you do not display dot points on the SSDs. This is to differentiate your .bit file from the TA's .bit file!

6.11 After having completed all "TODO" section in the top design, synthesize, implement, and download your design to the board. Demonstrate it to your TA. Celebrate your success!!!

7. Lab Report:

Name:________________________	Date: ________________________
Lab Session: __________________	TA's Signature: __________________

For TAs: Pre-lab (15): _____ Implementation (30+10): _____ Report (out of 45): ___
Comments:

Q 7. 1: Attach the files or submit them online as directed by your TA/instructor (`ee354_GCD.v, ee354_GCD_top.v`).

Q 7. 2: What is the frequency of the clock you used in the top design to capture the value of the switches into registers A and B? (2 pts) ________________
Assume that whenever a user presses BtnR, he or she holds the button down for about 0.25 seconds. Based on this 0.25 seconds depression time, compute the number of times A or B values are registered at the press of the corresponding buttons. (6pts)

Q 7. 3: Here we used a single flip-flop (or flag) called `A_bar_slash_B` to make it possible to accept `Ain` first followed by `Bin` next. Suppose, in another design, you need to capture 16 items (item 0 to item 15), would you use 16 flip-flops/flags or will you be able to manage with a small counter? Explain. (7pts)