

# EE201L

## Divider design

Objective: To introduce to students

- RTL coding style for state machine and datapath coding
- Testbench with a “task”
- debouncing mechanical Push Buttons and generating DPB, SCEN, MCEN, CCEN
- Single-stepping and Multi-stepping using the push-button debounce unit

### References (for the TAs, not for students):

1. Nexys-3 board reference manual (Nexys3\_rm.pdf) and schematic

<http://digilentinc.com/Products/Detail.cfm?NavPath=2,400,897&Prod=NEXYS3>

[http://digilentinc.com/Data/Products/NEXYS3/Nexys3\\_rm.pdf](http://digilentinc.com/Data/Products/NEXYS3/Nexys3_rm.pdf)

[http://digilentinc.com/Data/Products/NEXYS3/NEXYS3\\_sch.pdf](http://digilentinc.com/Data/Products/NEXYS3/NEXYS3_sch.pdf)

### 2. Epp protocol

First 4 pages of the **Digilent Parallel Interface Model Reference Manual**

<http://www.digilentinc.com/Data/Products/ADEPT/DpimRef%20programmers%20manual.pdf>

### Files provided:

A zip file is provided containing source files for four sample designs in four folders. *Please read the notes at the top of each file to get to know important aspects of the design to note.*

1. ee201\_divider\_simple
2. ee201\_divider\_with\_debounce
3. ee201\_divider\_with\_single-step
4. ee201\_divider\_with\_VIO\_multi\_step

A short description of each of the above 3 designs follows.

### 3. ee201\_divider\_simple:

Points to note:

The datapath elements shall be inferred by the synthesis tool. So we do not code OFL explicitly. See the diagram on the next page.

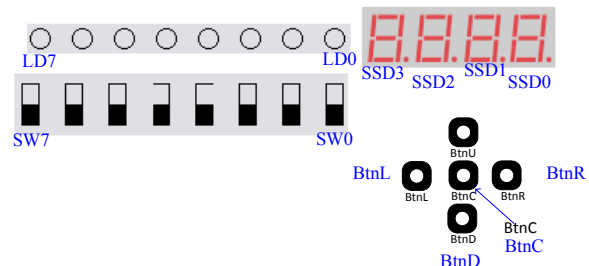
The datapath and the control unit can be combined in one **case** statement under clock as shown in divider\_combined\_cu\_dpu.v. Notice the lines on the side which avoid unnecessary recirculating muxes.

We have also provided another file:

divider\_separate\_cu\_dpu.v.

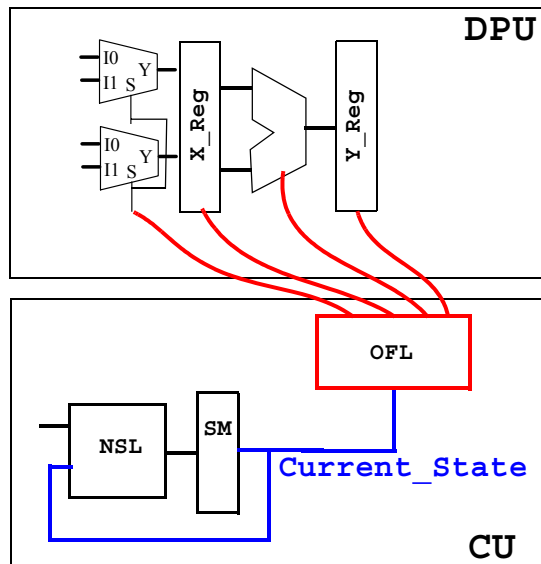
Extract from divider\_combined\_cu\_dpu.v

```
begin : CU_n_DU
  if (Reset)
    begin
      state <= INITIAL;
      X <= 4'bXXXX; // 4'bXXXX to avoid
      Y <= 4'bXXXX; // recirculating mux
      Quotient <= 4'bXXXX; // controlled by Reset
    end
  else
```



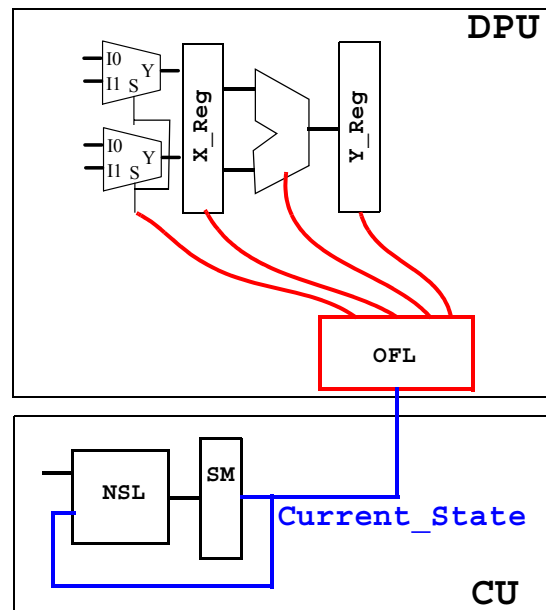
## Traditional division between DPU and CU

OFL (combinational logic) is in the CU.



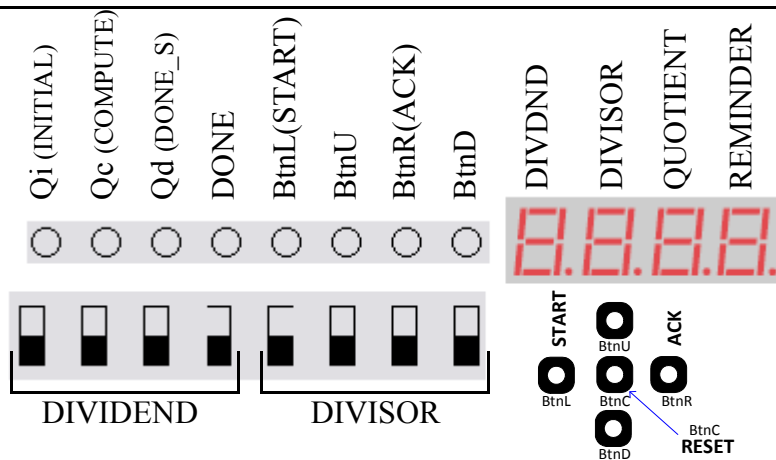
## Division between DPU and CU for HDL coding

OFL (combinational logic) is moved to DPU. It is NOT coded explicitly. The OFL is implicit in the DPU's RTL in the CASE statement.



### ee201\_divider\_simple

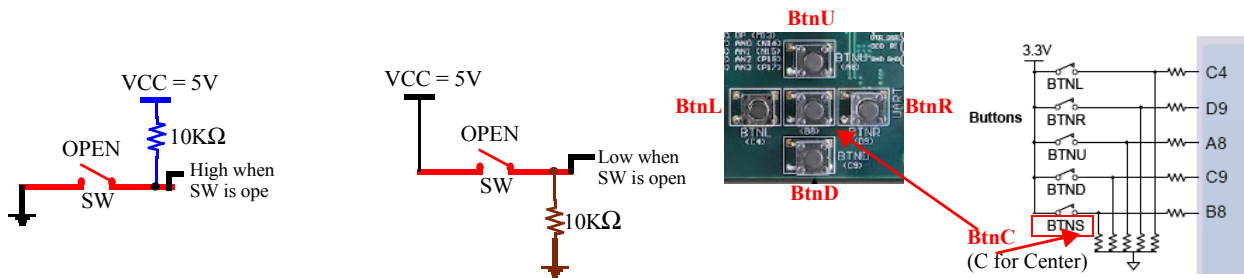
Go through the files and download the provided bit file and test.



### Questions for the ee201\_divider\_simple design:

- What happens if you divide by zero? Is the behavior of the quotient digit display on SSD1 different if you attempt to divide 3 by 0 vs. if you attempt to divide F by 0. How about 0 divided by 0?
- If you improve the divider design to move from compute state to done state if X is equal or less than Y (instead of the current X less than Y), will the above behavior change? Does your answer to Q#1 above change?
- Why does the behavior of the next design (**ee201\_divider\_with\_debounce**) appear to be quite different from this design for division by zero? Is it just appearance only or is it really different? Note: Look at the rate at which sysclk runs in both designs

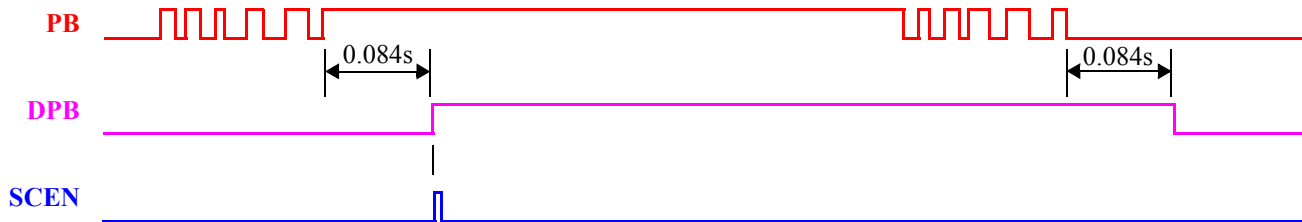
#### 4. Bouncing of mechanical Switches and Push Buttons:



Works for TTL and CMOS logics

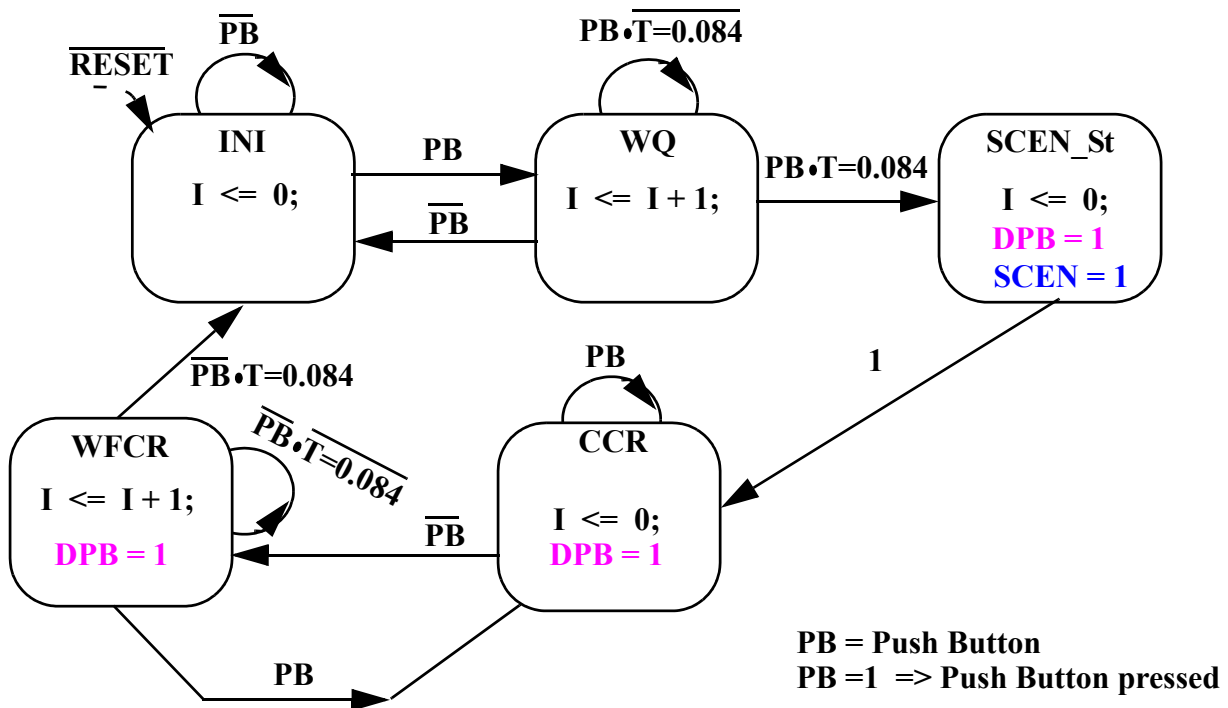
Works for CMOS but not TTL logic

Buttons on Nexys-3:  
When pressed, they produce high.



#### 5. Debouncing State Machines:

Debouncing State Machine (To start with just produce **DPB** and **SCEN**)



**WQ = Wait for a Quarter Second (actually 0.084ms)**

**SCEN = Single Clock Enable**

(enable the RTL transfer operation and/or state transfer operation for one clock of the 100 MHz system clock)

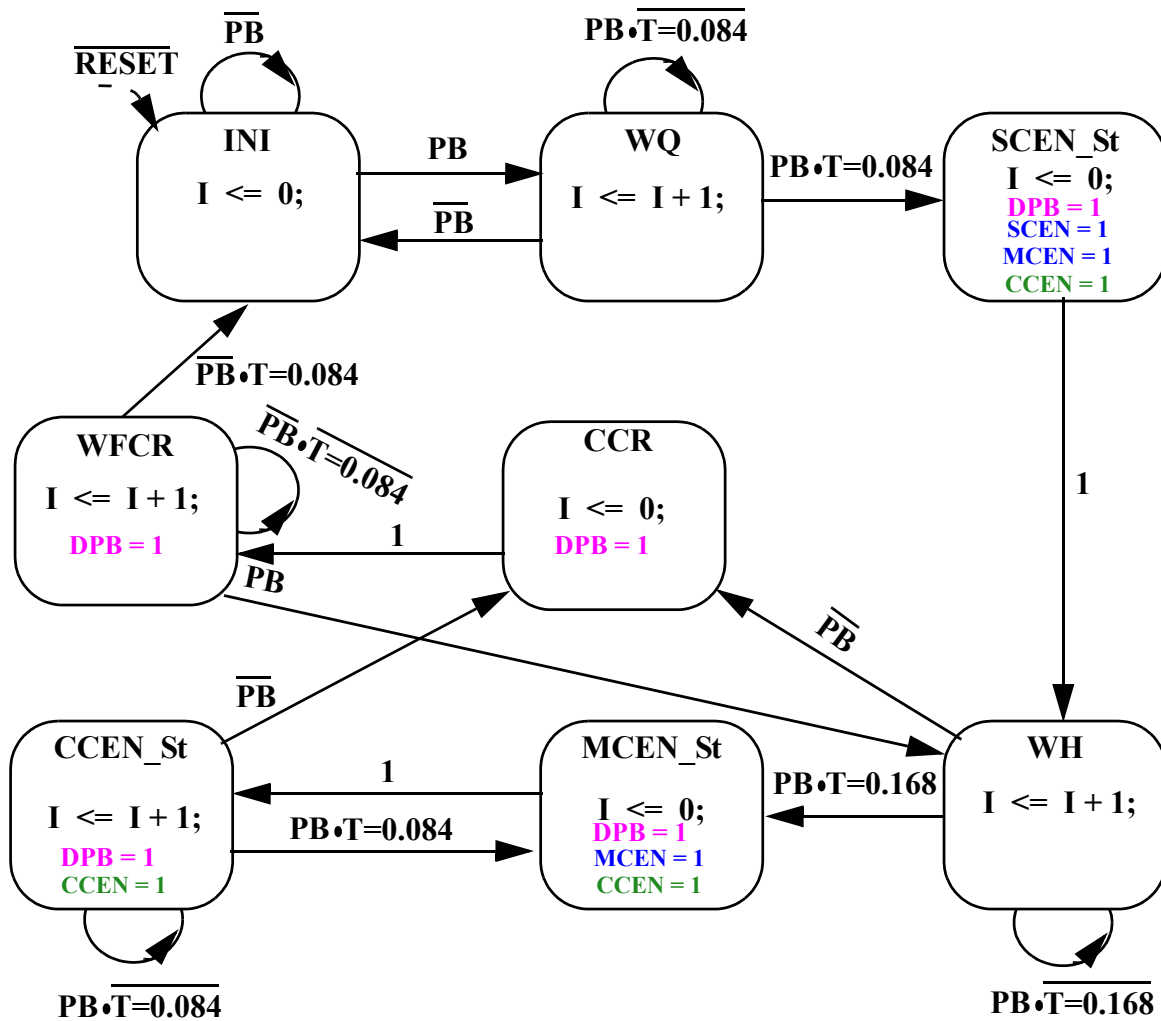
**CCR - Clear Counter**

**WFCR = Wait For Complete Release**

**DPB = 1 in all states except for INI and WQ states.**

**SCEN = 1 in SCEN\_St only. Hence SCEN is a single-clock wide pulse.**

## Debouncing State Machine (Now produce MCEN and CCEN besides DPB and SCEN)



**MCEN = Multiple Clock Enable (of course with 0.084 sec. gap)**

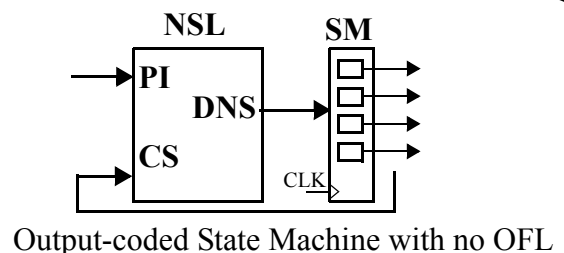
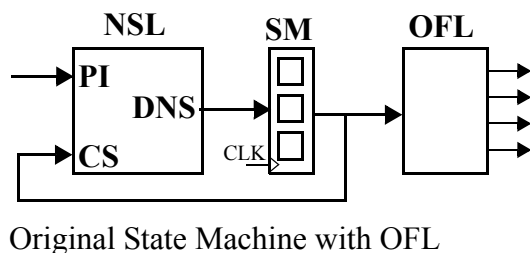
**CCEN = Continuous Clock Enable (with no gap)**

**MCEN is active in SCEN\_St and MCEN\_St.**

**CCEN is active in SCEN\_St, MCEN\_St, and CCEN\_St states.**

## 6. How to produce glitch-free outputs from a state machine:

Earlier, in class, we showed how easily glitches are produced by a combinational logic such as a mux or an equality checker. If we can avoid the OFL (Output Function Logic) in a Moore kind of state machine by cleverly coding symbolic states using output coding, then the output control signals come out of state flip-flops and they will be glitch free!



## 7. ee201\_divider\_with\_debounce:

Let us go through the debouncer design, ee201\_debounce\_DPB\_SCEN\_CCEN\_MCEN.v. It debounces a given push button and produces 4 outputs: DPB, SCEN, CCEN, MCEN.

Output coding (for the states in the state machine) is used to produce glitch free outputs.

State Name	State	DPB	SCEN	MCEN	CCEN	TB1	TB0
initial	INI	0	0	0	0	0	0
wait quarter	WQ	0	0	0	0	0	1
SCEN_state	SCEN_st	1	1	1	1	-	-
wait half	WH	1	0	0	0	0	0
MCEN_state	MCEN_st	1	0	1	1	-	0
CCEN_state	CCEN_st	1	0	0	1	-	-
MCEN_cont	MCEN_st	1	0	1	1	-	1
Counter Clear	CCR	1	0	0	0	0	1
WFCR_state	WFCR	1	0	0	0	1	-

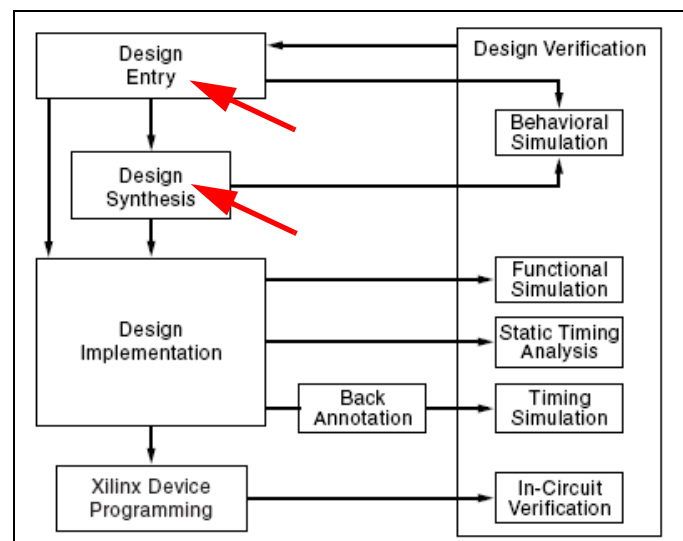
TB1 and TB0 are the tie-breakers to break aliasing in output codes.

One more state added to improve the utility of the earlier MCEN.

ISE => Help => Software Manuals => Click on Design Synthesis in the diagram (copy shown on the side) => XST User guide => Search for FSM Encoding

As shown here, we used verilog attributes to enforce our output coding. Through these attributes, we are informing the tool-vendor (Xilinx here) that we want the tool to honor and retain our user encoding.

It is possible to set FSM Encoding option under ISE => Synthesis XST => Properties => HDL options => FSM Encoding Algorithm = User. But this will apply to the entire design!



```
(* fsm_encoding = "user" *)
reg [5:0] state;
```

Verilog attributes are placed in parentheses between asterisks. Another example:

```
(* full_case, parallel_case *)
case (state)
```

### FSM Encoding Algorithm Verilog Syntax Example

Place FSM Encoding Algorithm immediately before the module or signal declaration:

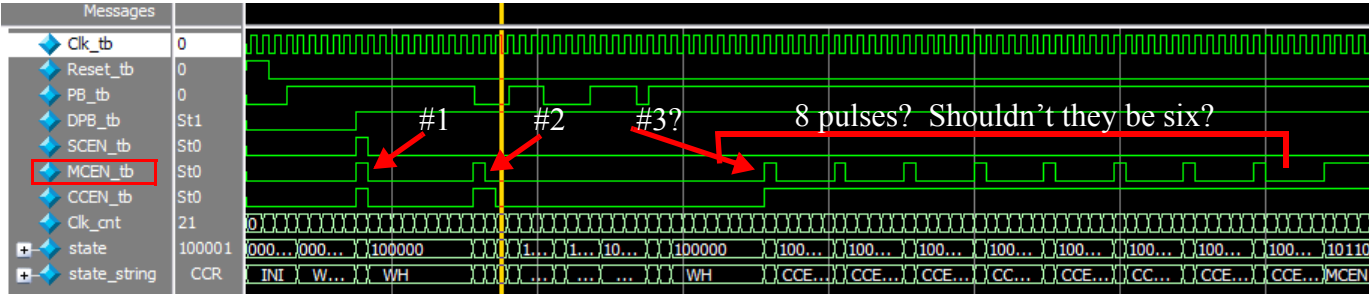
```
(* fsm_encoding = "{auto|one-hot
|compact|sequential|gray|johnson|speed1|user}" *)
```

The default is **auto**.

Read the code (ee201\_debounce\_DPB\_SCEN\_CCEN\_MCEN.v) and complete the state diagram on the next page. Simulate it using ee201\_debounce\_DPB\_SCEN\_CCEN\_MCEN\_tb.v for 9 us.

Notice that, the testbench has instantiated the UUT with N\_dc of 4 in the generic map

```
ee201_debouncer #(.N_dc(4)) ee201_debouncer_1
    (.CLK(Clk_tb), .RESET(Reset_tb), .PB(PB_tb),
     .DPB(DPB_tb), .SCEN(SCEN_tb), .MCEN(MCEN_tb),
     .CCEN(CCEN_tb));
```



A simple (rather construed) example of the SCEN pulse of debouncer is as follows. Suppose, we are running short of the buttons on the board and we wish to use a single button (BtnL) both as a START button and an ACK button, Then DPB pulse does not help as our divider is running at full speed (100MHz) and one operation of the BtnL (say 0.2 sec) will be considered as several hundred thousands of these START and ACK operations. So when you let the BtnL go, you can not tell whether the state machine is waiting in the Initial state or Done state! But with SCEN, only one-clock wide pulse per operation is applied to the circuitry!

**ee201\_divider\_with\_debounce**

Go through the files and download the provided bit file and test.

Note that, unlike in the earlier design, (**ee201\_divider\_simple**), we run the core divider in this design at the **full speed of 100Mhz**.

Q<sub>i</sub> (INITIAL)    Q<sub>c</sub> (COMPUTE)    Q<sub>d</sub> (DONE\_S)    DONE    BtnL (START/ACK)    BtnU    BtnR    BtnD

DIVIDEND    DIVISOR

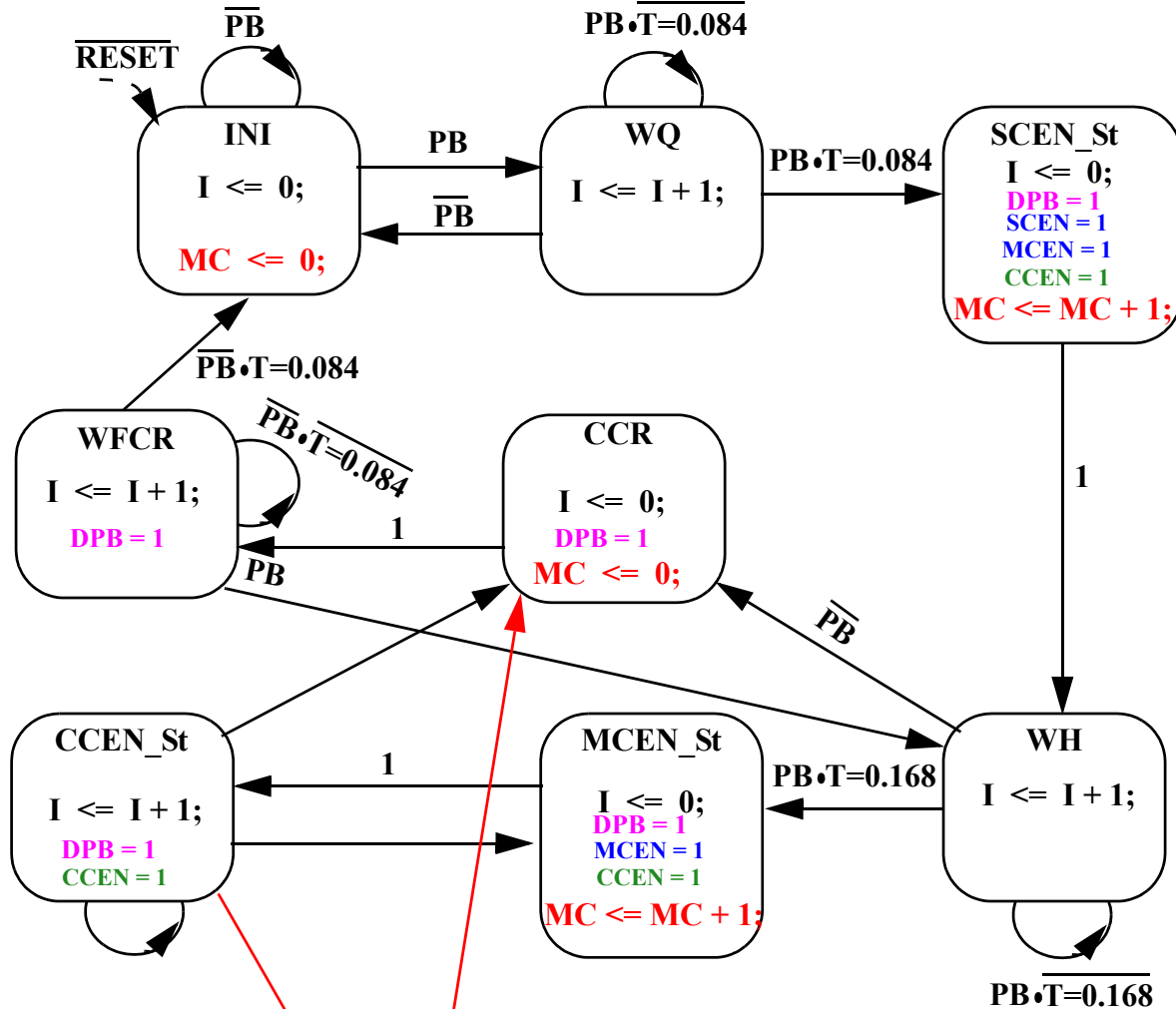
DIVIDND    DIVISOR    QUOTIENT    REMINDER

START/ACK    BtnL    BtnU    BtnR    BtnD    BtnC RESET

Questions on the debouncer and the divider with debouncer:

1. Briefly explain why the N\_dc parameter was changed to 4 during simulation (from the actual value of 25 for synthesis and implementation). Use words such as “inefficient”, “wasteful”, “readability of waveform”, etc.
2. When you simulate, zoom into the area of above waveform extract and arrive at your answer for the above question in the waveform extract (why do we see 8 more pulses on MCEN after already seeing two pulses).
3. Did we use the DPB (Debounced Push-Button) pulse or SCEN (Single-Clock enable) pulse to act as the Start signal and the Acknowledge signal? Could we have used anyone of them?

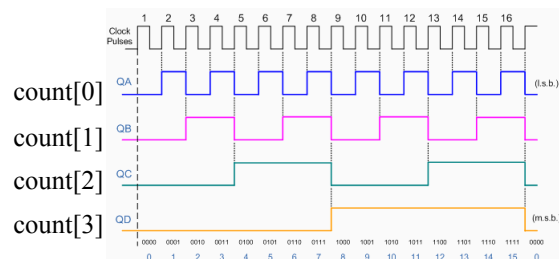
Complete the Debouncing State Machine with the added state MCEN\_cont  
Complete the missing state transition conditions and also any RTL in the state MCEN\_Cont



MC stands for MCEN count.  
After certain count of MCN,  
control is transferred to  
MCEN\_Cont.

MCEN\_Continuous state  
Here MCEN behaves like CCEN.  
See the output coding table given before.

Nexys3 board clock is at 100MHz.  
100MHz frequency corresponds to 10ns clock period.



count[3] becomes 1 after 8 ( $=2^3$ ) clocks of the CLOCK.  
count[2] becomes 1 after 8 ( $=2^3$ ) clocks of the CLOCK.  
 $2^3$  clocks each of 10 ns make 0.084 sec. Hence  $T_1 = 0.084$  sec  
 $2^4$  clocks each of 10 ns make 0.168 sec. Hence  $T_2 = 0.168$  sec

Names of the students submitting:

- 1.
- 2.

## 8. Single-stepping:

Single-stepping and break-point setting are used in software or hardware debugging. Here we wish to show a hardware debugging mechanism involving single-stepping and multi-stepping, which will lead to setting break points. This will be useful particularly when you are interfacing your design with an external system which can not be simulated and proven in simulation. Also sometimes there will be simulation/synthesis mismatches and this helps in debugging in those situations. In later labs, we will also show you chipscope to gather hardware signal activity at full speed. Chipscope is essentially a logic analyzer placed inside the FPGA chip to sample and gather signals and show them to us on the PC monitor as waveforms or state listings.

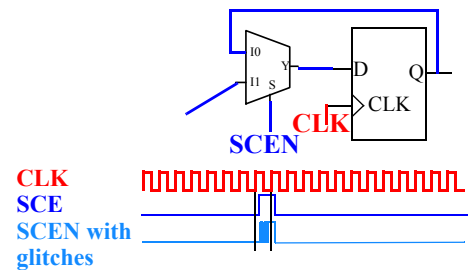
Let us first talk about single-stepping. Most common idea is to apply one clock pulse at a time whenever the single-step PB is pressed. One can think of using a clean (glitch-free) pulse such as DPB as the clock to the system. However the problem in FPGA is to put this derived clock on global routing resources in FPGA.

### Spartan-6 FPGA Clocking Resources

[http://www.xilinx.com/support/documentation/user\\_guides/ug382.pdf](http://www.xilinx.com/support/documentation/user_guides/ug382.pdf)

If we can not use the global routing resources for our DPB, then this DPB reaches different registers in our design at different times and the relative skew (difference in the arrival times of these clock pulses) causes the circuit to fail. For example consider a simple right-shift register, with progressively delayed clock sent to the right-side flip-flops.

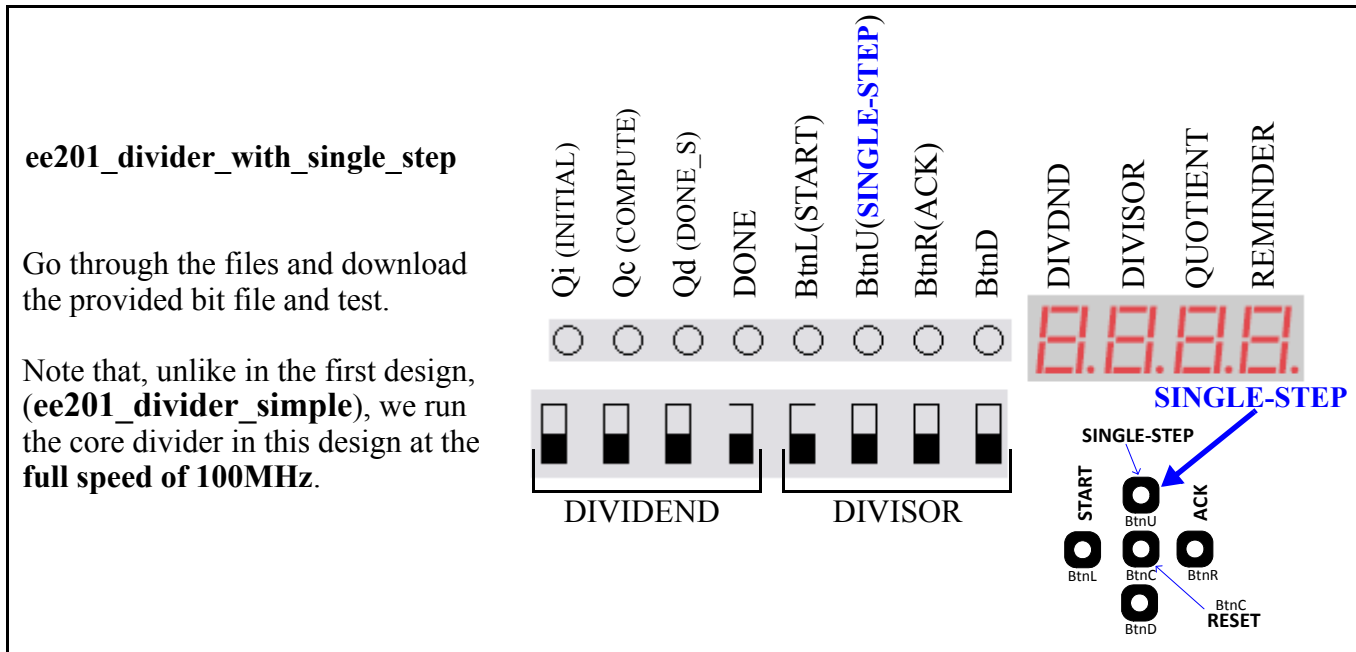
Multi-stepping occurs and the shift register fails. Hence we designed a better way to implement single-stepping. We do not use DPB or SCEN as “the clock” but we use SCEN as the clock enable. SCEN stands for Single Clock Enable and it is nominally equal in width to a single clock cycle. Since it is the clock enable and control the data-recirculating mux, even if SCEN has some glitches, they do not hurt the circuit operation. The glitches are in the beginning of the clock and die down by the end of the clock. It is the responsibility of the STA (Static Timing Analyzer, which is part of any synthesis tool) to make sure that the glitches die down before the arrival of the next clock-edge. So, if the circuit passed timing-design, we can be assured that the glitches do not hurt our circuit.



Single-stepping is not a complete solution for debugging as very often, we need thousands or millions of clocks needed before the suspected malfunctioning part of the circuit behavior can be encountered. For example, a real-time clock (a wall-clock) may misbehave at the roll-over from 23:59:59 to 00:00:00. So, it is a good idea to produce MCEN and CCEN. We can easily modify the above state diagram to terminate the CCEN or MCEN to force the debounce state machine go back to initial state under any break-point condition (such as time = 23:59:59).



## 9. ee201\_divider\_with\_single\_step



Here, in the compute state, we single-step the division operation using the SCEN produced out of BtnU. Notice the following aspects of the design.

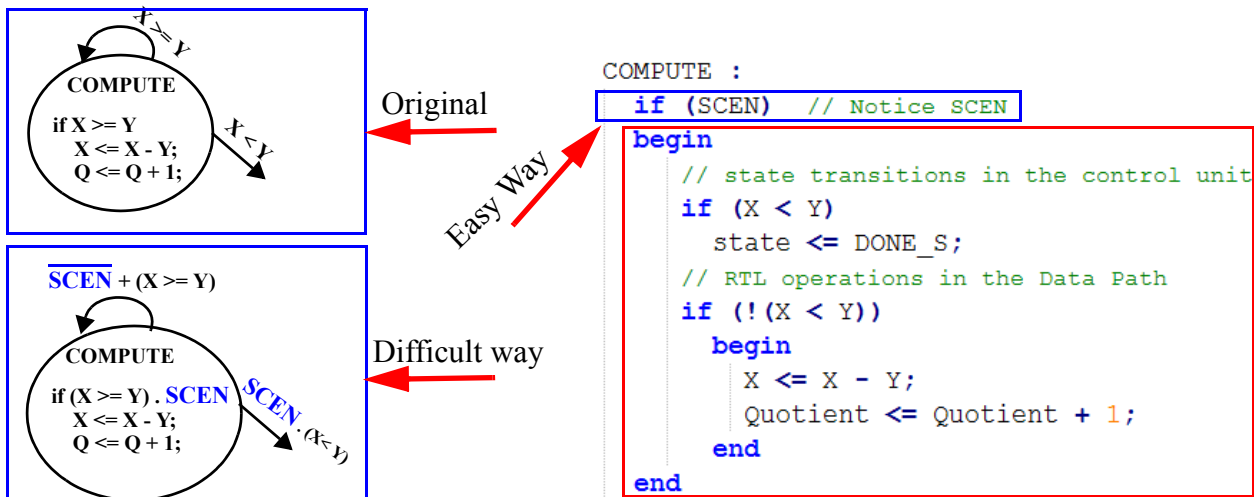
A. The divider and the divider instantiation have a new port pin called SCEN for the top-level design to generate and pass SCEN pulses (Single-Clock-wide clock enable pulses) (more accurately data-enable pulses as the clock itself is not inhibited).

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

B. Single-Step Control can easily be exercised on selected states such as the compute state in the divider as shown below. The “if (SCEN)” clause before “begin” ensures that

- (i) all state transformations from the COMPUTE state and
- (ii) all data transformations with-in the compute state,

are under the control of SCEN. We do not have to rewrite the state diagram as shown below.



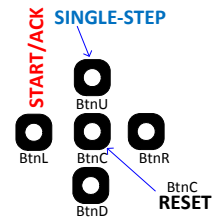
## Questions on ee201\_divider\_with\_single\_step:

A. Is it possible to use SCEN to control one state (or a few states), MCEN to control another state, and further CCEN to control yet another state? When we say “control a state” here, we mean control the RTL operations in the state and also the state-transitions going away from the state (excluding looping-around state transitions). If we are not going away from the state (because of absence of the SCEN pulse) then we will remain in the state, whether originally there is a loop-around state-transition or not.

B. Can we choose to place **all three states** of the divider design under single-stepping control and *simultaneously* combine Start and Ack under one button (say BtnL)?

Is this just not possible or it works if we produce a BtnL\_SCEN and use it as START as well as ACK, or ...?

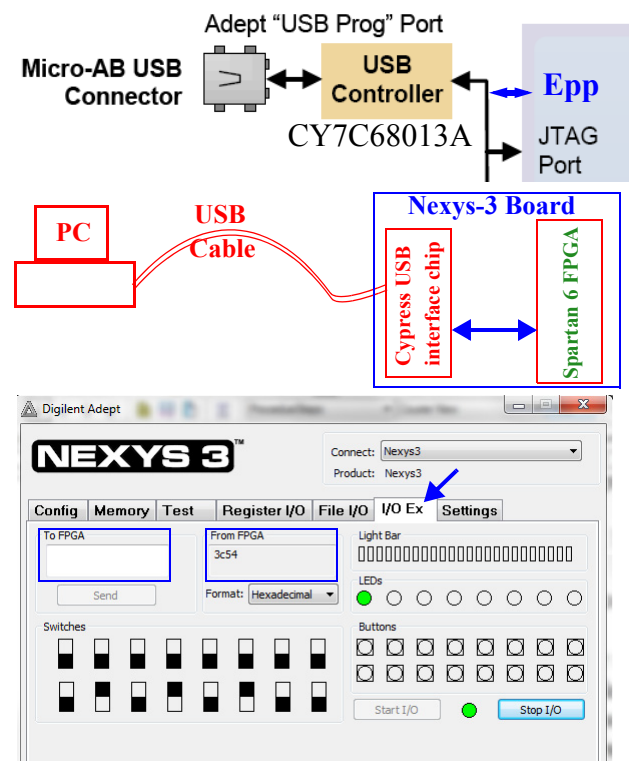
Can you press two buttons exactly at the same time to 10ns or 5ns accuracy? Even if you press at the same time to that accuracy, can you guarantee that they bounce for the same length of time and the two instances of the debouncing state machine would produce their respective SCEN pulses at the same time?



C. We took time to design output-coded state machine with no OFL at all, there by avoiding any glitches in the SCEN, MCEN, etc. Are glitches really harmful in our design or we have just shown a way to produce glitch-free outputs?

## 10. Epp Interface on Nexys-3 board for communication with the PC

Epp stands for Enhanced Parallel Port. Epp interface was used to interface PC-XT to a dot-matrix printer in 1980's. ([http://en.wikipedia.org/wiki/IEEE\\_1284](http://en.wikipedia.org/wiki/IEEE_1284)). It is a very simple interface and easy to use though obsolete. Since USB interface is a fairly complex interface, and since current PCs do not support Epp interface, Cypress (<http://www.cypress.com>) offer USB interface chips, which convert the USB interface to simple interfaces such as Epp. Since Epp is a very simple interface, Digilent chose to use a Cypress interface chip CY7C68013A on their Nexys-3 boards to convert communications from and to a PC on USB to Epp for us to deal with on the FPGA. Digilent has also provided the Adept software to run on the PC and communicate with the Cypress chip on USB. They have also provided IOExpansion.vhd (which we translated to Verilog for EE201L as IOExpansion.v) for instantiation in FPGA-side designer's top file. The UCF file for such project should include the pins associated with the Epp interface. On the Adept GUI, the Register I/O, the File I/O and the I/O Ex. Here, in this lab we explore the I/O Ex tab, which we refer to as Virtual I/O. We call it Virtual I/O as it adds several addition Switches, Push Buttons, LEDs on the GUI to the limited number of these on the Nexys-3 board. Besides these, we can exchange 32-bit data using the two boxes labeled as “To FPGA” and “From FPGA”. After instantiating the IO Expansion module (defined in IOExpansion.v), the user logic can send data to the LEDs in the I/O Ex tab very much in the same fashion as he would send data to the LEDs on the Nexys-3 board. He can read from data on the Virtual switches very much like he reads switches on the Nexys-3 board. The Adept GUI uses program driven I/O (not interrupt driven I/O) to exchange data. It sends and receives data very frequently.

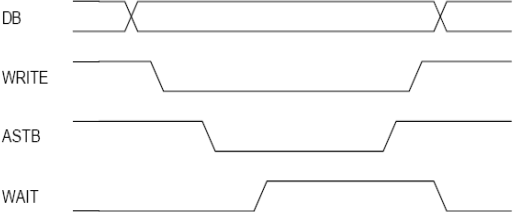


A short extract from Digilent manuals:

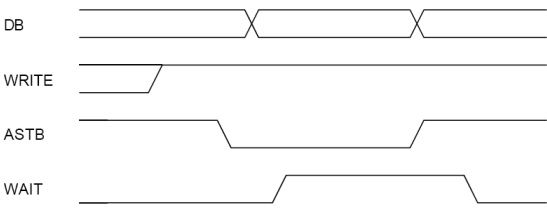
Net "EppAstb" LOC = H1  
Net "EppDstb" LOC = K4  
Net "EppWait" LOC = C2  
Net "EppWr" LOC = F5

Some of the Epp pins in the .ucf file.

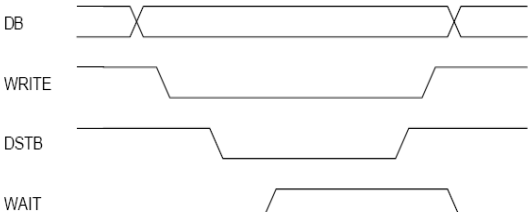
Address Write



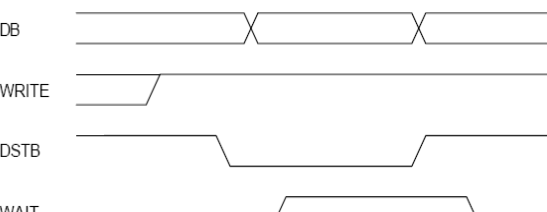
Address Read (The Address Read perhaps is not used in Adept software)



Data Write



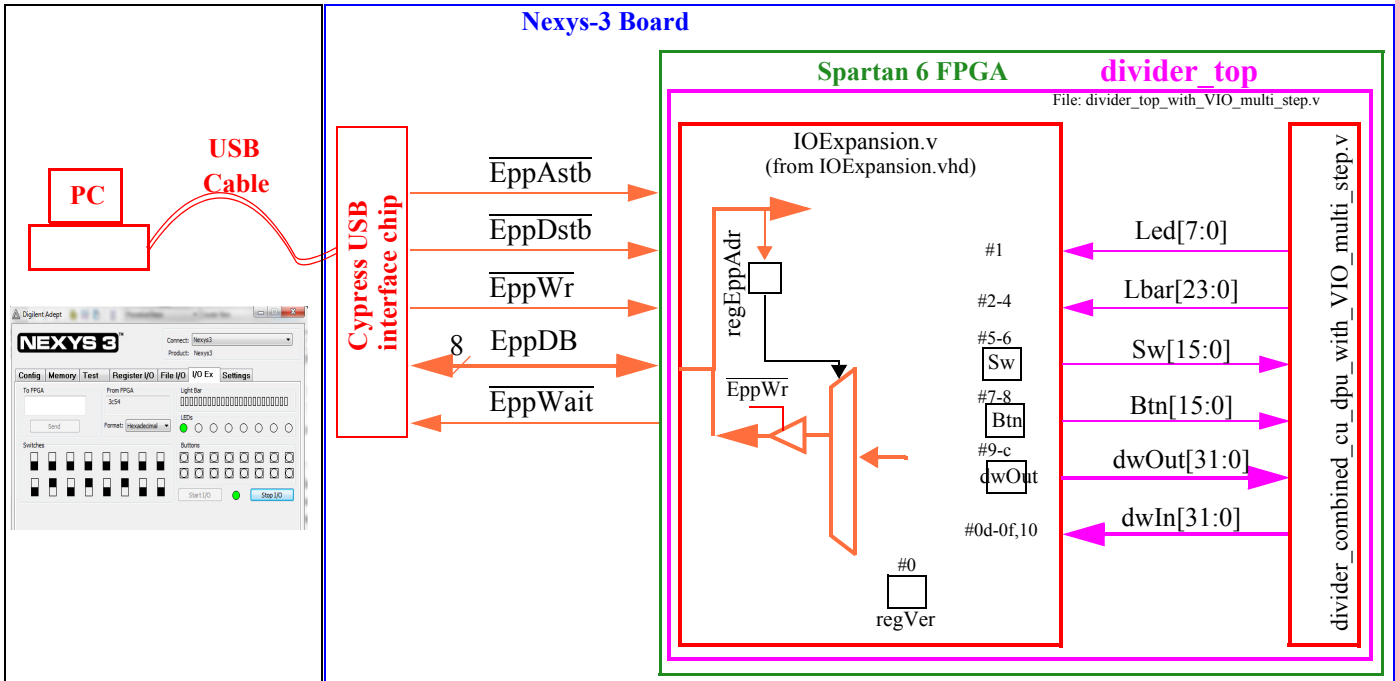
Data Read



The following signals make up the interface:

Name	Source	Description
DB0 – DB7	bidir	Data bus. The host is the source during write cycles and the peripheral is the source during read cycles.
WRITE	host	Transfer direction control. High = read. Low = write
ASTB	host	Address strobe. Causes data to be read or written to the address register
DSTB	host	Data strobe. Causes data to be read or written to a data register
WAIT	peripheral	Synchronization signal used to indicate when the peripheral is read to accept data or has data available.

Instead of viewing this as a low active wait, it may be easier to view it as a high-active GOT signal. Notice that the Epp protocol implements the full (4-way) handshake.



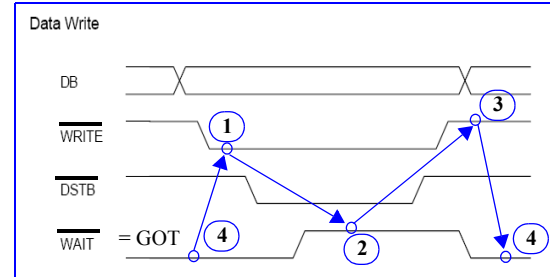
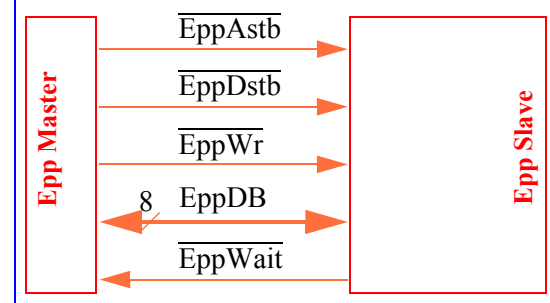
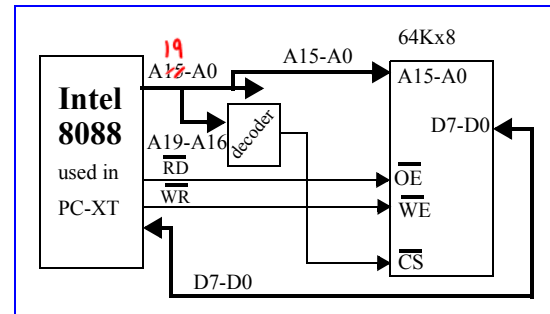
## 11. Epp Protocol in short

Let us understand a simple processor to memory interface. The processor puts out address and a control strobe ( $\overline{RD}$  or  $\overline{WR}$ ) and exchanges data with the addressed location in memory. But this involves several signal wires. In the case of (simplified) 8088 to memory interface, we need 20 address lines ( $A_{19}-A_0$ ), 8 data lines ( $D_7-D_0$ ), two control strobes ( $\overline{RD}$  and  $\overline{WR}$ ). In the case of Epp, it is desired that the interface has less wires. So instead of separate address lines and separate data lines, Epp uses one set of 8 data lines which carry data if  $\overline{DStb}$  is active and address if  $\overline{AStb}$  is active. Since these strobes carry timing information also, we do not need the strobes to carry timing information. Instead, in Epp, we have  $\overline{WR}$  line, which can be viewed as indicating the direction of transfer ( $\overline{WR} = 0$  means write and  $\overline{WR} = 1$  means read). In microprocessor-memory interface, we have a READY line (not shown on the side) which allows a slow memory to request for more time to respond. Here we have  $\overline{wait}$  to implement a 4-way handshake protocol

(1 Take it (2 Got it (3 I see you got it (4 I see that you saw that you got it ).

How does the Epp protocol provide for multiple data transfers in the absence of an address bus?

The Epp master can convey an address first using  $DB[7:0]$  and  $\overline{AStb}$  and then later data using  $DB[7:0]$  and  $\overline{DStb}$  for *that* addressed location. Oh, it means we can at most exchange 256 bytes because using 8-bit address you can only generate  $2^8 = 256$  addresses! No, it is not like that. The Epp master and the Epp slave can have a common understanding that the master always sends address in two parts, high part followed by low part before data is transmitted. This increases the number of bytes that can be transferred to 64KB ( $2^{16} = 65536 = 64K$ ).



```
// EPP Address register
always @(posedge EppAstb)
begin
    if (!EppWr)
        regEppAdr <= EppDB;
end
```

Extract of **IOExpansion.v**

Epp Address Register is written at the end of the Epp Address Strobe because Epp Write control line is low indicating intent to write.

Adept 2.0 or higher running on your PC control the Cypress interface chip and causes communication between the Adept GUI on your PV display and the Cypress chip.

The Cypress chip is the Epp master, which drives the three control lines:

**EppAstb**: Epp Address Strobe (active low, ending edge is posedge),

**EppDStb**: Epp Data Strobe (active low, ending edge is posedge),

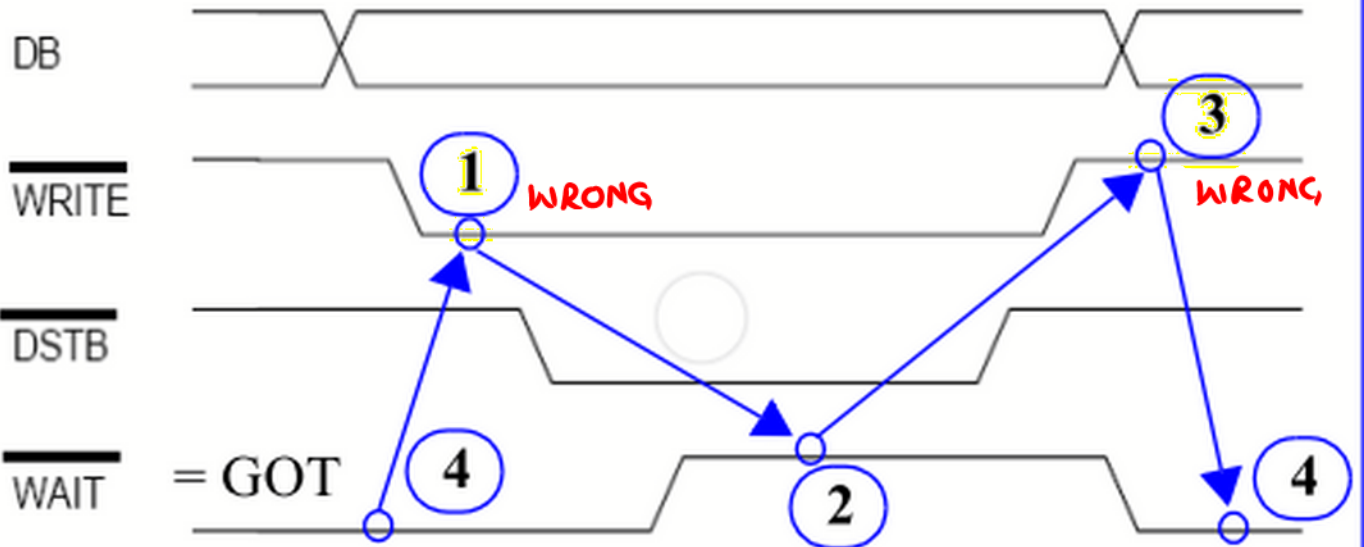
**EppWr**: Epp Write Control (active low, low means intent to write, high means intent to read).

The **EppDB** is the Epp 8-bit data bus. During an active address or data strobe, Epp master drives data if write is true ( $EppWr = 0$ ) else slave drives data if read is true ( $EppWr = 1$ ).

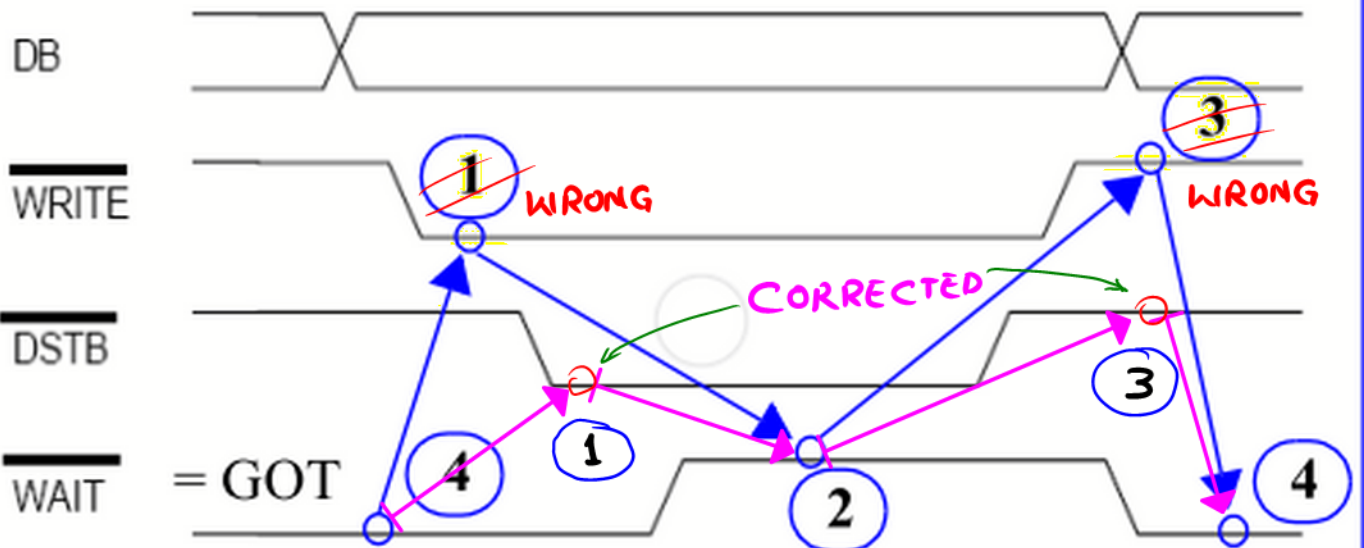
Active-low **WAIT** (= active-high GOT) acts like a hand-shake signal between the two parties.

Address Read Cycle is not implemented in Adept Virtual I/O protocol.

## Data Write



## Data Write

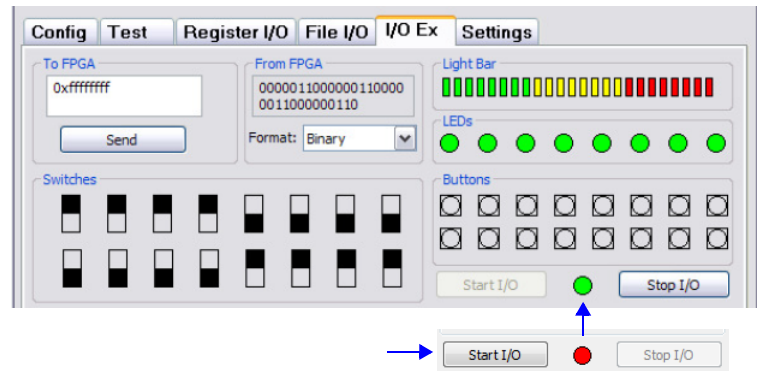


## 12. Adept Application User's Manual.pdf

Please refer to the Adept User's manual on your PC (Start => All Programs => Digilent => Adept => Adept Application User's Manual.pdf. Extract of it is shown on the side.

## 13. ee201\_divider\_with\_VIO\_multi\_step (VIO = Virtual I/O)

Here we are interfacing to the virtual I/O in Adept 2.0. The file, **IOExpansion.vhd**, provided by Digilent, implements the Epp slave-side address and data registers in FPGA. We translated the same to Verilog. The file is called **IOExpansion.v**. Note that now, the UCF file needs to have pins associated with Epp to talk to the Cypress USB interface chip.



You can control the system board using the expanded virtual I/O controls on the I/O Ex tab,

The expanded I/O controls include 16 switches, 16 buttons, 8 LEDs, 24 individual light bars, and the ability to send and receive a 32-bit value.

Adept connects to the I/O Ex configuration when you press the Start I/O button. A special handshaking register is checked to verify that the I/O Ex design is active in the FPGA. If it is valid, the I/O Ex status light turns green. If it isn't, the light turns yellow, but the I/O registers are still polled.

The I/O Ex connection can be stopped any time by pressing the Stop I/O button, switching to a different tab, or changing the connected device.

Switches are flipped and buttons pressed by clicking on the graphics.

The next two pages show the utilization of I/O resources on the Nexys-3 board and in the Virtual I/O GUI.

One important difference between the debouncer user earlier (ee201\_debounce\_DPB\_SCEN\_CCEN\_MCEN.v) and the debouncer used in this part (ee201\_debounce\_DPB\_SCEN\_CCEN\_MCEN\_r1.v) is that here (in\_r1 version), we have increased the time gap between consecutive **MCEN** pulses to **1.342** sec. Hence instantiations of the debouncer uses here an **N\_dc** parameter of 28:

```
ee201_debouncer #(.N_dc(28)) ee201_debouncer_2
```

### Task to be performed

Download the .zip file provided to you into your C:\xilinx\_projects\ directory and extract files to form C:\Xilinx\_projects\ee201\_divider\_verilog directory with 4 sub-folders:

1. ee201\_divider\_simple
2. ee201\_divider\_with\_debounce
3. ee201\_divider\_with\_single-step
4. ee201\_divider\_with\_VIO\_multi\_step

All the four folders have verilog source files, .ucf source file, a .bit file (with TAs\_ prefix) of the completed design . After reading the code, you can download the .bit file to the Nexys-3 board and operate the divider.

The bit files provided to you have a "TAs\_" prefix so that you do not overwrite when you compile the sample designs to get practice in forming a xilinx project and implementing the same.

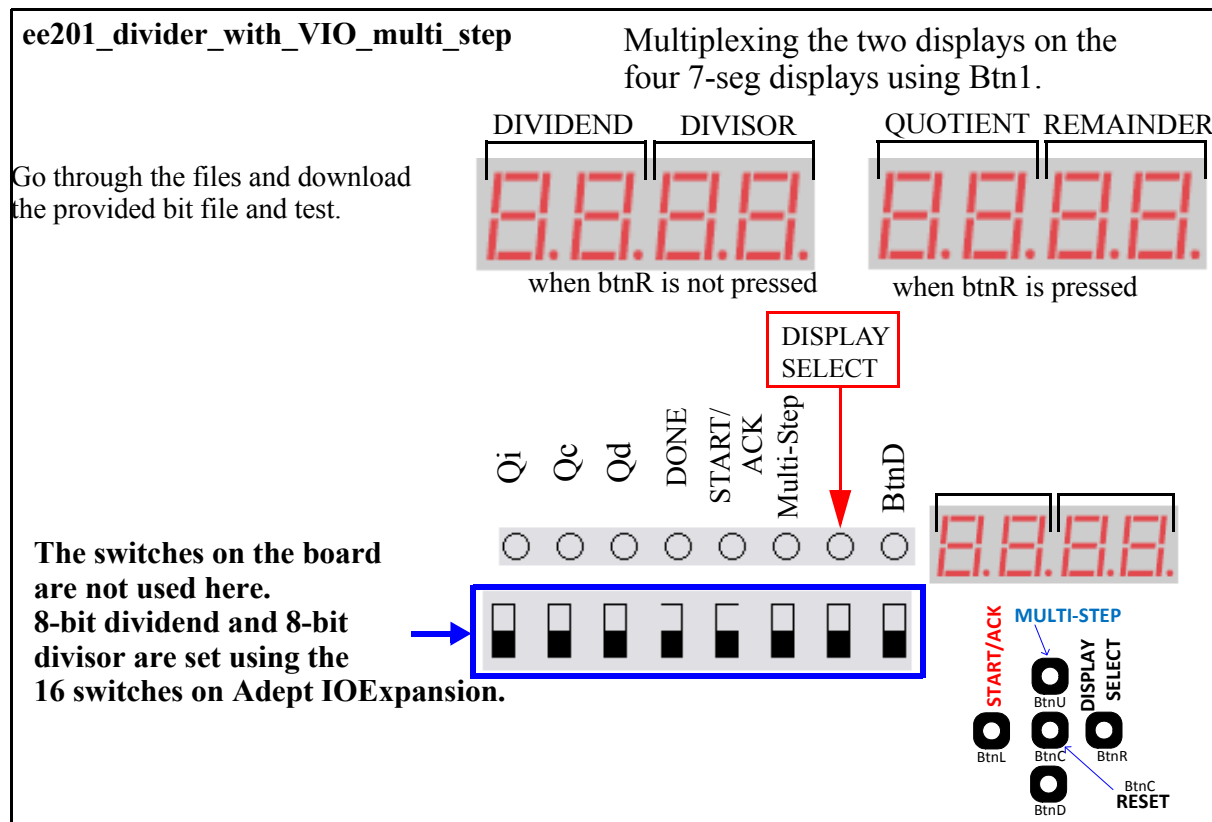
When you are done, you will submit a report to your TA with your answers to questions posted under first three designs. No questions are posted for this last design.

## 14. Celebrate your success!!!

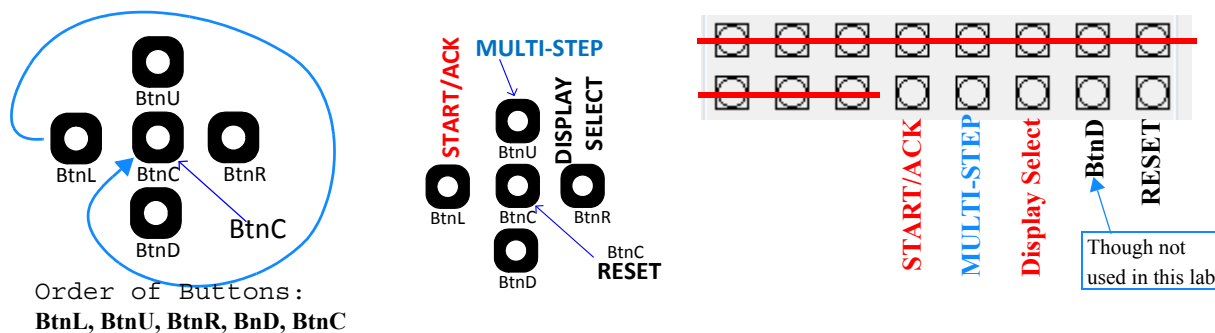
Don't forget this step!



Adept Virtual I/O resource utilization for this part of the lab (unused resources are crossed out):



The push button mapping is as follows:



## ee201\_divider\_with\_VIO\_multi\_step

In this example 58H is the dividend and 04H is the divisor.  
The quotient is 16H and the remainder is 00H.

Can be used to send the Dividend (Xin) and the Divisor (Yin)

**NEXYS 3™**

Connect: Nexys3  
Product: Nexys3

**Dividend, Divisor, Quotient, Remainder**

Config Memory Test Register I/O File I/O I/O Ex Settings

To FPGA: 0x5804  
From FPGA: 58041600  
Format: Hexadecimal

Send

Replica of Nexys-3 LEDs

Light Bar

LEDs: Qd Done

Buttons

Start I/O Stop I/O

Can be used to set Divisor

Can be used to set Dividend

Replica of Nexys-3 Buttons

Note: The 16 switches shall be in down (off) position, if you want to send Xin, Yin via the To FPGA facility. Similarly, if you want to use the switches to send Xin, Yin, send 0x0 through the To FPGA box.

```
assign lbar_to_VIO[7:0] = {8{BtnR_combined}};
```

