DigiTerm - a terminal window on your PC

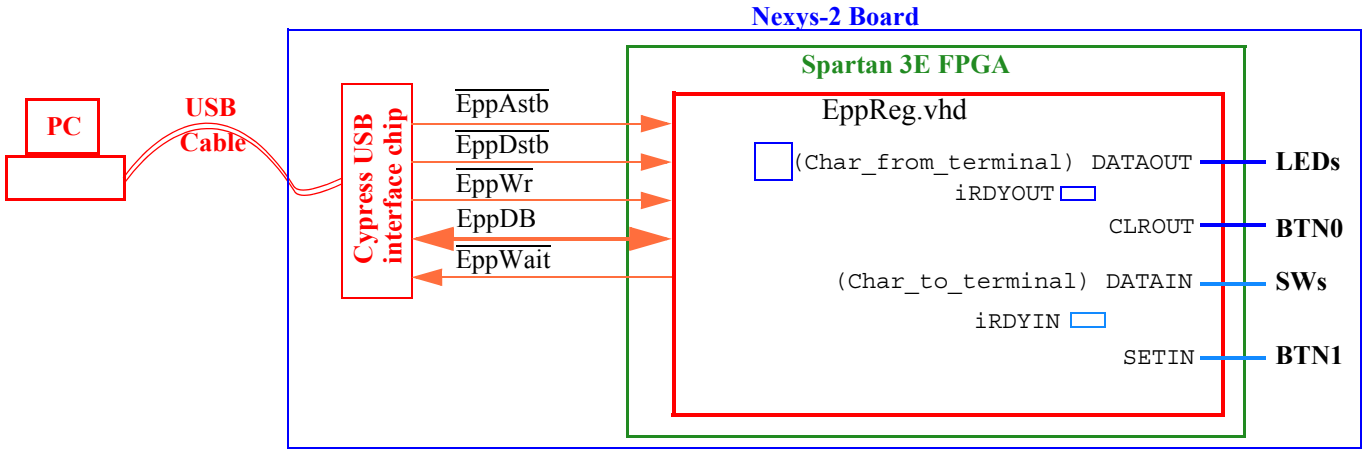
Objective

- To get to know how to use DigiTerm to provide user interface on the PC via an USB link.
- To modify the given example code to make an user interface for the simple divider design.

Introduction

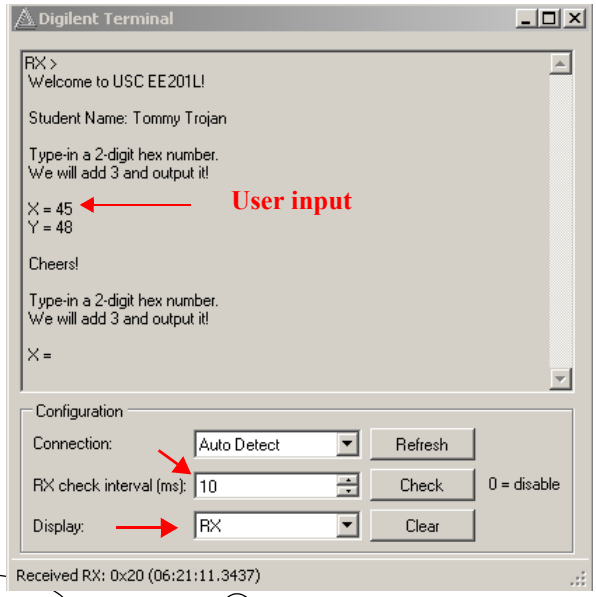
In Spring 2010, Digilent, the supplier of the Nexys-2 board (predecessor to Nexys-3 board), has provided us an utility (`DigilentTerminal.exe together with EppReg.vhd`) which allows us to send and receive ASCII characters between the FPGA on Nexys-2 board and the user PC (a windows PC). The communication is done via a Cypress USB interface chip (`CY7C68013A`) on the Nexys-2 board. Since USB protocol is quite elaborate and complex, we (and Digilent) let the Cypress USB interface chip to take care of it. Between the Cypress chip and the FPGA, a simple Epp interface is created so that the user can easily handle data transfers.

Digilent design demonstrates sending a character (whose ASCII code is set on SWs on the Nexys-2 board) to the PC and receiving a character from the PC (whose ASCII code is displayed on LEDs on the Nexys-3 board). In this demo design (`EppReg.vhd`), the user uses the SETIN button (BTN1 of the Nexys-2) to send a character to the PC. The user uses the CLROUT button (BTN0 of the Nexys-2) to acknowledge receipt of the character from the PC.



We, at USC, have adopted Digilent's core design, improved it, built upon it, and created a simple design having a user core design (in place of a human using BTN1 and BTN0). This design (DigiTerm_example.v) sends out a greeting message on to the PC and interacts with the user. See the screen shot on the side. We designed it originally for the Nexys-2 board in Spring 2010 and converted it to Nexys-3 in Spring 2012

Students can easily change the messages sent out to the PC, prompt for as many user inputs as needed and display as many results as needed.



Short description of the communication part of the `DigiTerm_example.v` design:

Here we assume that you were already taught the Epp protocol. Here we are giving a brief description of the important implementation details. For detailed explanation, please watch the webcast associated with this design and then go through the code.

Using the address strobe, `EppAstb`, the Epp Master writes to a single-bit flip-flop, `rAdr0` a single-bit address (a 1 or a 0). If (`rAdr0 == 0`), then the next data strobe, `EppDstb`, will cause the master to send or receive **data** depending on whether `EppWr` is a 0 or a 1. If (`rAdr0 == 1`), then the next data strobe, `EppDstb`, will cause the master to receive status. At that time, `EppWr` is a 1. During a data read operation by the master ((`EppDstb == 0`)&&(`EppWr == 1`)) depending upon what was deposited in `rAdr0` most recently by the master, we return either `DATAIN` or `Status` to the master.

A little after the beginning of the `EppAstb` or the `EppDstb`, we inactivate `EppWait` (by making it a 1) to allow the master to complete the transaction. When the master is trying to read (`EppWr == 1`), we drive the `EppDB` with the `DATAIN` or `Status` for the entire duration of the `EppDstb` strobe. At all *other* times, the tristate buffer connecting to the `EppDB` is tristated by the `DigiTerm_example.v` design.

When the master is trying to write to us (`EppWr == 0`) (i.e. trying to send us an address or a data), it appears that the Cypress chip keeps the data valid and stable on the `EppDB[7:0]` for the entire period of the `EppAstb` or `EppDstb`. This is evident from the fact that the Digilent example design samples the data almost at the beginning of the `EppAstb` or `EppDstb` and still be able to capture the correct data. However, a more conservative (and hence robust) assumption is that the data on the `EppDB` may not be initially valid and stable. But it has to be valid and stable certainly towards the end of the `EppAstb` or the `EppDstb` pulse. So how do you capture the data at (towards) the end of the control strobe (without know when the end of the strobe is about to arrive)? It is like planing to leave Los Angeles one day before the anticipated big earth quake.

We have come up with an interesting design to capture data towards the end of the control strobe without knowing when the end of the strobe is going to occur. The idea comes from the way they do cockpit recording in an aeroplane to analyze the events leading to a crash in the case of a crash. The cockpit recorder (**the black box**) records say 30 minutes of data continuously (over-writing the previous data) so that most recent 30 minutes of data before the crash will be available. In our design, (which is a deviation from the Digilent design) pipeline registers constantly gather the data on EppDB.

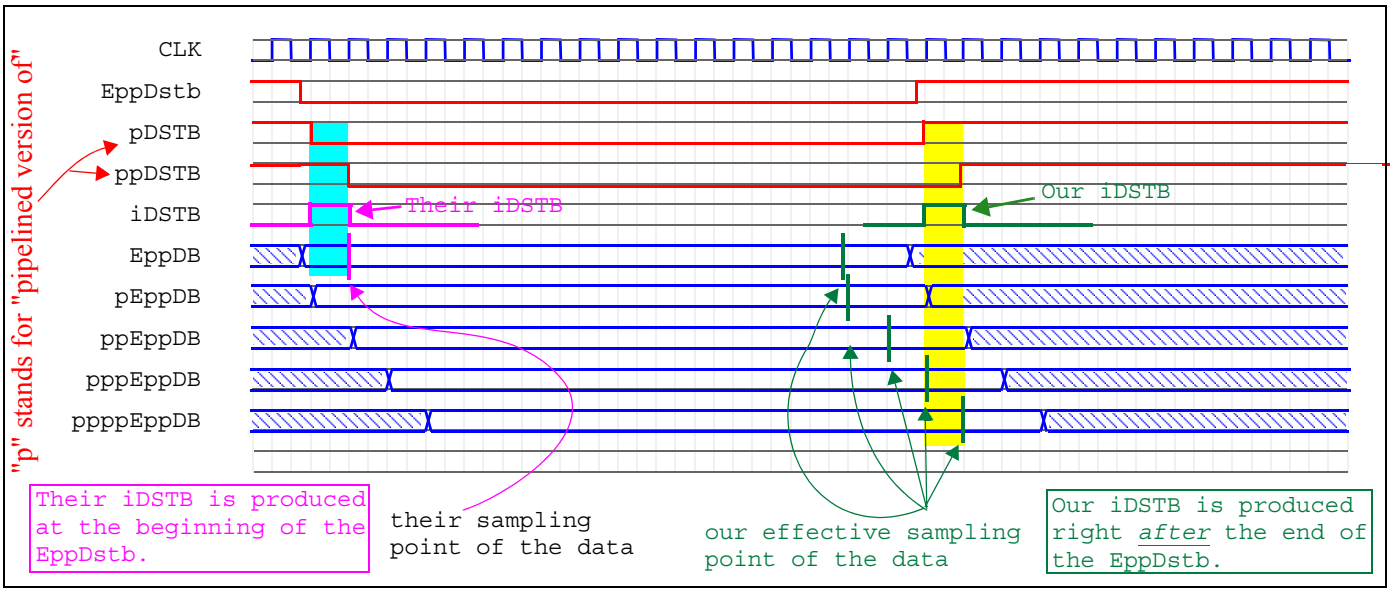
Some significant statements under the clocked always procedural block:

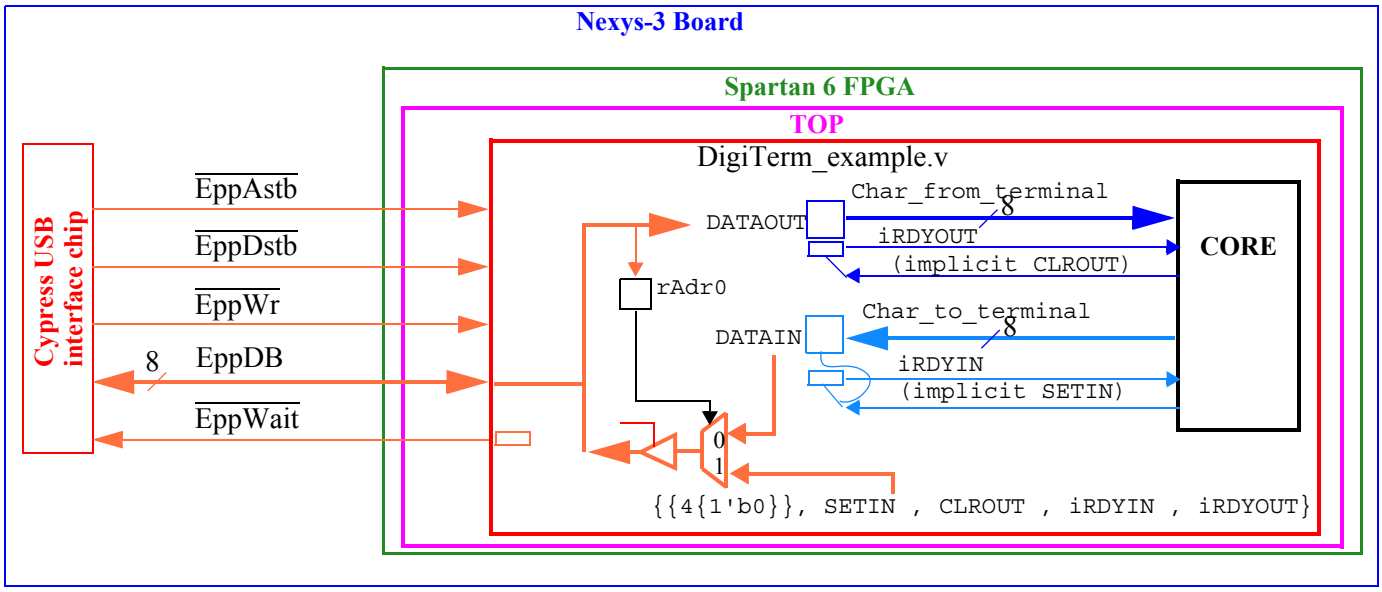
```
pEppDB <= EppDB; ppEppDB <= pEppDB; pppEppDB <= ppEppDB; ppppEppDB <= pppEppDB;
pDSTB  <= EppDstb;   ppDSTB <= pDSTB;
if ((~rAdr0) & (iDSTB) & (~ppppEppWr))  // this is our method
       Char_from_terminal <= ppppEppDB; // this is our method
// if ((~rAdr0) & (iDSTB) & (~EppWr)) // this is their (Digilent's) method
//     Char_from_terminal <= EppDB; // this is their (Digilent's) method
```

Some significant concurrent assign statements:

```
assign iDSTB = ((~ppDSTB) & pDSTB); // we produce a iDSTB pulse, right "after the ending" of pDSTB
// assign iDSTB = (ppDSTB & (~pDSTB)); // They produce a iDSTB, right "after the beginning" of pDSTB
```

Both designs (theirs and ours) worked here because the Cypress USB chip puts out valid data on EppDB pins for the entire EppDSTB active period. However, if we use the Digilent's method, then the iRDYIN is set prematurely at the beginning of the EppDstb pulse letting the user to potentially send the next character on the EppDB before the Epp Master had an opportunity to gather the current character. Hence we needed to qualify the premature signal with the end of the data strobe as in `[((Message_Length != 0) && (iRDYIN == 1'b1) && (pDSTB == 1'b1))]`. But in our design, the condition `[((Message_Length != 0) && (iRDYIN == 1'b1)]` is adequate.





Short description of the user side of the `DigiTerm_example.v` design:

In a way, it is like a hierarchical state machine design. We have a "**`Step`**" counter to keep track of the major steps to be performed. For example:
Step #0: Send out Welcome message.
Step #4: Prompt for input.
Step #5: Echo back the user input.
Step #8: Display results.

The signal "**`Busy`**" acts like a "Flag" between the two state machines (or the two parts of the mega state machine, if you wish to view it that way). The `Step` counter and the `Busy` Flag are cleared on system Reset.
`Step <= 4'b0000; Busy <= 1'b0;`
(`Busy == 1'b1`) means the communication part of the state machine is busy performing the step.
(`Busy == 1'b0`) means next task (next step) can be assigned to the communication part of the state machine.
When (`Busy == 1'b0`), the user state machine sets up the next set of parameters (`message length, message`) and sets `Busy` to **`1`** to alert the communication part of the state machine.

Example:

```
4'h0:  begin Message_Length <= 29; message <= {CR, LF, " Welcome to USC EE201L!", CR, LF, CR, LF}; Busy <= 1'b1; end
```

When (`Busy == 1'b1`), the communication part of the state machine starts sending out character by character to the PC terminal and after sending out the last character (i.e. *after it has been fully transmitted*), it reports to the main state machine by resetting the Busy flag and incrementing the Step counter.

```
if (Message_Length == 0) begin  Busy <= 1'b0;  Step <= Step + 1; end
```

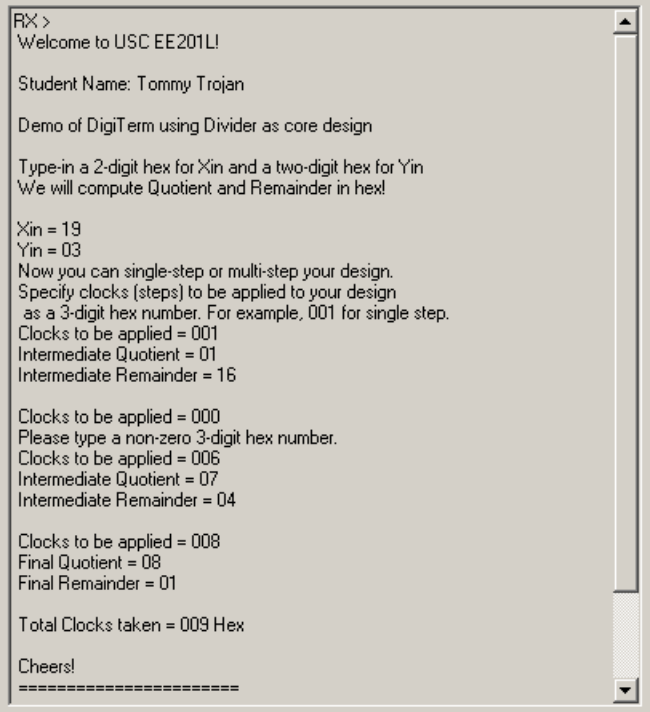
Core Design Instantiation:

The core design is instantiated in the top design responsible for communicating with the DigiTerm. This top design, in a way, acts like a testbench, by conveying the inputs provided by the user (via the DigiTerm) to the core design, giving a `Start` to the core design and then waiting on the `Done` signal, collecting the results, displaying the results on the terminal, and finally giving an `Ack` signal to the core design.

Scope for adding a lot of features:

On the side, you see our divider design with CEN control, instantiated and controlled by digiterm_divider.v design. As you see, the user can single-step or multi-step the core design and have intermediate results displayed. Also we keep track of the number of clocks taken by the core design to complete its operation and report it to the user, very much like a testbench.

Students are encouraged to incorporate DigiTerm in their semester-end projects.



Procedure:

1. Download the `DigiTerm_example_Verilog.zip` and extract files to form a project directory `C:\Xilinx_projects\DigiTerm_example_Verilog`

2. Go through the file, `DigiTerm_example.v`.

3. Program the Nexys-3 with the .bit file provided (`digiterm_example.bit`) using Adept Configuration program. Close the Adept Configuration GUI.

4. Double-click and run the provided `DigilentTerminal.exe`. You will see the terminal window. Test its operation. Close the window.

5. Repeat steps 3 and 4 above with the `digiterm_divider.bit`.

6. If time permits, try to create `digiterm_divider.v` based on `DigiTerm_example.v`. First do not support single/multi-stepping. When you succeed with it, you can add single/multi-stepping and clock counting. We provided `digiterm_divider.v` also. Download `DigiTerm_example_Verilog.zip` and extract files into `C:\Xilinx_projects\DigiTerm_example_Verilog`

Some references for the TAs:

1.Digilent Nexys3 Board Reference Manual
http://digilentinc.com/Data/Products/NEXYS3/Nexys3_rm.pdf
2. Nexys-3 Schematic
http://digilentinc.com/Data/Products/NEXYS3/NEXYS3_sch.pdf
3. Epp protocol
First 4 pages of the **Digilent Parallel Interface Model Reference Manual**
http://www.digilentinc.com/Data/Products/ADEPT/DpimRef%20programmers%20manual.pdf

Celebrate your success!!!
Don't forget this step!