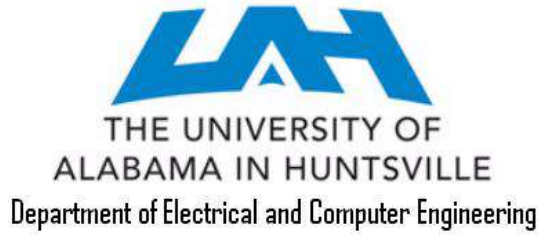


CPE 324 Advanced Logic Design Laboratory

Laboratory Assignment #5

Single Button Texter

(7% of Final Grade)

The focus of this laboratory is for you to utilize finite state machine techniques to develop a *Single Button Texter* module that could be used as a Human Machine Interface (HMI) for computing and communication devices. The texting module should in effect decode on/off keying made by toggling a simple switch to generate plain text whose input pattern corresponds to a modified version of International Morse Code. The plain text will correspond to the ASCII subset of Unicode characters. This Single Button Texter module is to be formulated as a dedicated synchronous digital design. The functionality of this design is to be tested by integrating it into a physical test bench environment which has already been pre-designed to allow for real-time manual and automated tests to be performed to verify the overall operation of the texter.

Verilog HDL Implementation

It is assumed that you are working as part of a team of engineers that are to design the *Single Button Texter* that would form the basis for a Human Machine Interface (HMI) device for simple text message communication such as Short Message Service (SMS) systems. The goal is to develop a digital hardware representation in Verilog HDL that can be evaluated using the DE2-115 FPGA platform where, once tested, it can be easily ported into Application-Specific Integrated Circuit, ASIC, technology. The texter is to be designed to allow SMS-style text messages to be sent by pressing and releasing a single button. The goal is to create a systems where such messages could be sent without the need to look at a screen or keypad. To evaluate the prototype system, an actual momentarily closed button type switch is to be used but if the device were mass produced in large volumes it is believed that the button could actually be an ubiquitous sensor that could be rhythmically varied between two states.

- The development an optimal input method for a human to convey low level language information using on/off switching that could be applied to sending simple messages in the English language is suitable for much research. A basic trade study in this area revealed that the International Morse Code had been used in the past for this purpose. It had the attributes of being a variable length code with the more frequently used character elements requiring less time to transfer than the less frequently used ones. The drawbacks included a significant time overhead that occurs between elements, characters, and words as well as the need to send data in a fairly rhythmical manner. With this in mind, a modified form of the Morse code was adapted for this implementation allowing a framework through which other

codes could be implemented in the future if they were found to be more beneficial in terms of ease of human entry, data rate or accuracy. Morse Code has the following properties.

- Each character is represented as a unique sequence of one or more dots(.) and dashes(_).
- Dots represent the short period of time, dashes represents the longer period of time. This coding sequence is shown below in Figure 1
- The dot time is equal to one unit of time, the dash time is equal to three units of time.
- The time between successive dot/dashes in a character coding sequence is one unit of time.
- There should be a total of three units of time between successive characters in a word (or character groups).
- There are seven units of time between words (or groups of character groups)
- The base unit of time, $\mathbf{T_{unit}}$ depends on the speed that the text is sent. It is given by the equation

$$\mathbf{T_{unit}} = 1200 / S$$

where **S** is the speed in words per minute, and $\mathbf{T_{unit}}$ is the unit time in milliseconds.

Initial Texting Code Format -- Modified Morse Code

A	._	**N**	_.	**0**	_____	**-**	_...._
B	_...	**O**	___	**1**	._____	**:**	___...
C	_._.	**P**	.__.	**2**	.._____	**?**	..__..
D	_..	**Q**	__._	**3**	..._____	**!**	_._.__
E	.	**R**	._.	**4**	...._	**‘**	__..__
F	.._.	**S**	...	**5**	.....	**_**	..__._
G	__.	**T**	_	**6**	_....	**(**	_.__._
H	....	**U**	.._	**7**	__...	**)**	_.__.
I	..	**V**	..._	**8**	___..	**=**	_..._
J	.___	**W**	.__	**9**	____.	**+**	._._.
K	_._	**X**	_.._	**.**	._._._	**$**	..._.._
L	._..	**Y**	__._	**,**	__..__	**/**	_.._.
M	__	**Z**	__..	**;**	_._._.	**case toggle**	..__

Example -- for the sentence "*THIS IS A TEST*."

= switch pressed = switch released

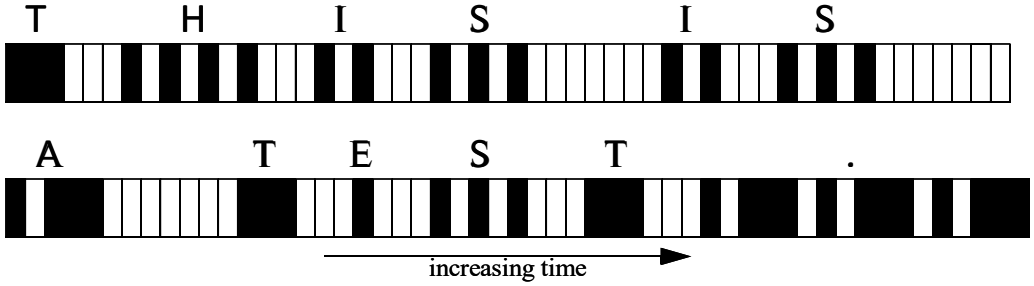


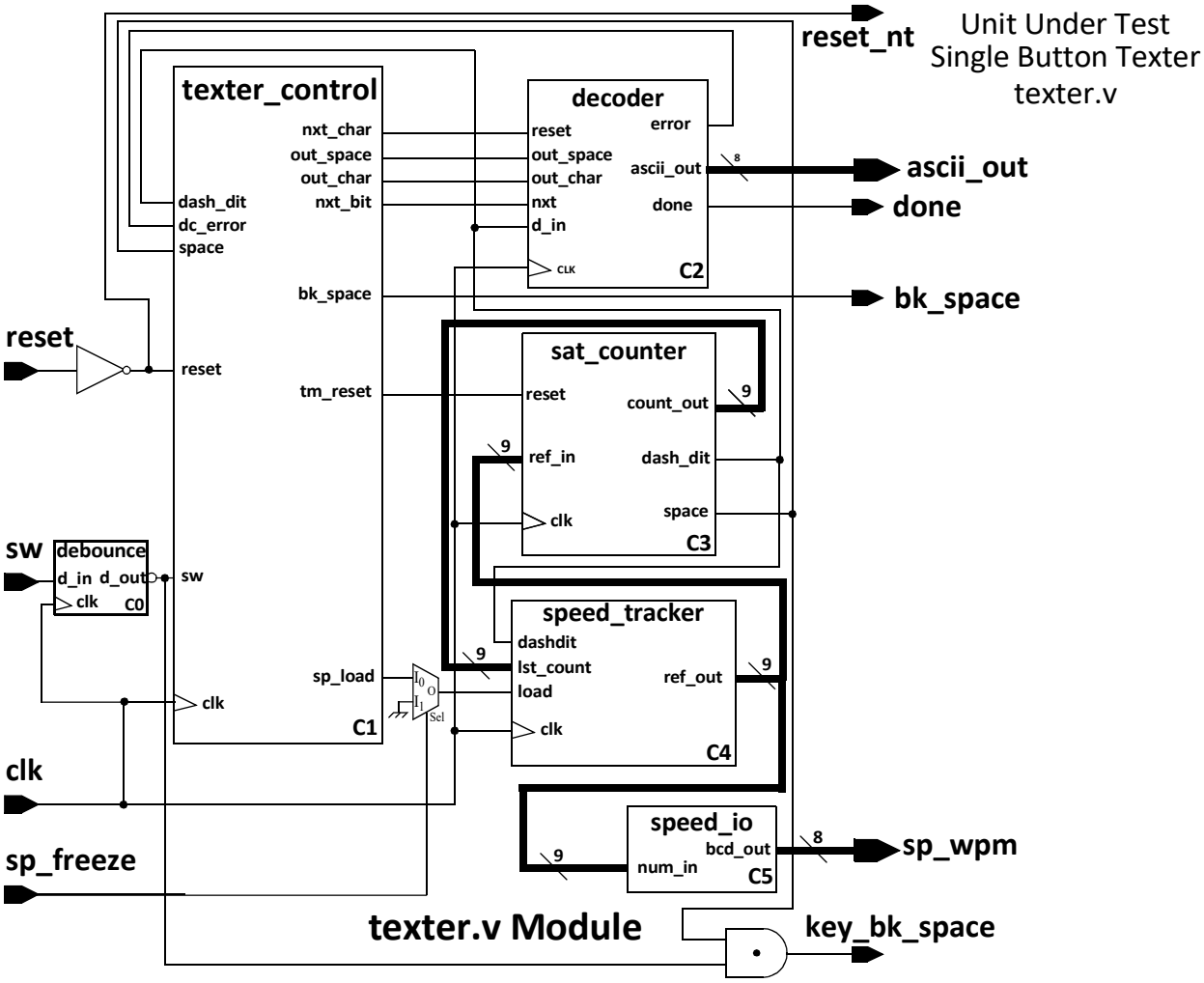
Figure 1: Texting Code Format

Figure 2 illustrates the high level design for the texting module that was developed by the mythical design team. The major components which compose the data path portion of the system include the decoder, sat_counter, and speed_tracker. These modules have already been designed, separately implemented, and tested at the unit level. These components are to be monitored and controlled by the main texter_control module. The texter_control module, represents the main control element within the design. It is to be developed by you in behavioral Verilog using standard finite state machine techniques.

The *decoder* module is used to shift in the dot/dash bits (dots represented by 0, dashes represented by 1) and to match the bit pattern with a valid Morse Code character. It then outputs the ASCII code representation of the character. The *sat_counter* is a saturation counter that times the period that the button is down and judges whether the button press is a dot or dash. It is also used to time the period the button is not pressed and if it reaches its full value, it holds it and declares that a space should be generated to separate words from one another.

The *speed_tracker* module produces the reference number of clock cycles that is used by the sat_counter module. For the case where a key is pressed then if saturation counter is greater than this reference count then it is a dash, if it is less than or equal this count a dot is declared. The *speed_io* module monitors the reference count produced by the *speed tracker* to output (in BCD) the estimated speed in words per minute (WPM) that is being sent. These modules have been created by the design team and are part of the IP core library. Detailed information can be obtained by viewing the source files.

Please note that the speed tracker is disabled by disconnecting the *sp_load* output signal from the *texter_control* module and placing on the *load* input of the *speed_tracker* module a logic 0 level on it instead. Normal speed tracking mode occurs when a logic 0 is placed on *sp_freeze* input of the texter module which connects the *texter_control* module's *sp_load* output to the *load* input of the *speed_tracker* through a 2-to-1 MUX. When this *sp_freeze* input is set to a logic 1 then speed_tracker module's *ref_out* speed that is used by the *sat_counter* is held constant.




```
// Single Button Texter -- main Single Button Texter module
// -- texter.v file
// (c) 2/5/2026 B. Earl Wells, University of Alabama in Huntsville
// all rights reserved -- for academic use only.
module texter(input reset,clk,sw,sp_freeze, output bk_space,done,
   key_bk_space,reset_nt, output [7:0] ascii_out,sp_wpm);

   // internal nodes and busses
   wire n0, n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11;
   wire [8:0] b0,b1;

   assign reset_nt = n0;
   assign n0 = ~reset;
   assign key_bk_space = n1 & n2;

   debounce C0(.clk(clk),.d_in(sw),.d_out(n1));

   texter_control C1(.clk(clk),.reset(n0),.sw(n1),.space(n2),
      .dash_dit(n3),.dc_error(n4),.nxt_bit(n5),.nxt_char(n6),
      .out_char(n7),.out_space(n8),.tm_reset(n9),.sp_load(n10),
      .back_sp(bk_space));

   texter_control C1(.clk(clk),.reset(n0),.sw(n1),.space(n2),
      .dash_dit(n3),.dc_error(n4),.nxt_bit(n5),.nxt_char(n6),
      .out_char(n7),.out_space(n8),.tm_reset(n9),.sp_load(n10),
      .back_sp(bk_space));

   decoder C2(.clk(clk), .d_in(n3),.nxt(n5),.reset(n6),.out_char(n7),
      .out_space(n8),.error(n4),.done(done),.ascii_out(ascii_out));

   sat_counter C3(.clk(clk),.reset(n9),.ref_in(b0),.dash_dit(n3),
      .space(n2),.count_out(b1));

   assign n11 = (sp_freeze) ? 0 : n10; // 2-to-1 MUX

   speed_tracker C4(.clk(clk),.load(n11),.dashdit(n3),
      .lst_count(b1),.ref_out(b0));

   speed_io C5(.num_in(b0),.bcd_out(sp_wpm));

endmodule
```

Figure 2: Single Button Texter Logic (Top-Level)

The *texter_control* module is to coordinate all activities of the *Single Button Texter* design. Your assignment is to implement this module as a finite state machine. An Algorithmic State Machine, ASM, Chart for this module has been suggested and is shown in Figure 3.

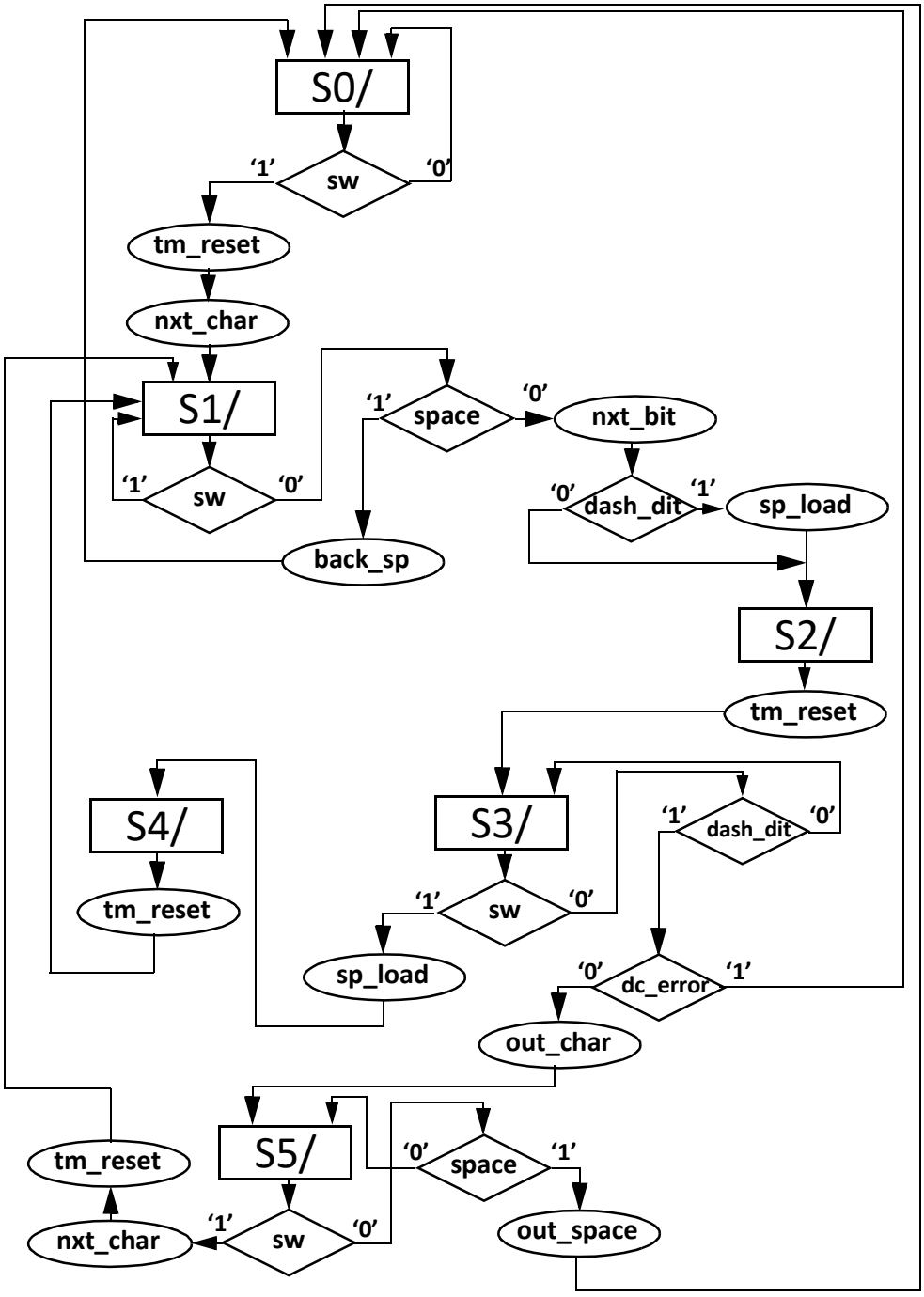


Figure 3: texter_control Algorithmic State Machine Representation

The ASM assumes that each transition occurs on an active edge of a clock. In this case you are to assume that the active edge is the positive clock edge.

Physical Test Bench

A *test bench* is an environment used to verify that the component being designed functions as intended. It gets its name from the traditional method of testing hardware and software in the laboratory where the physical prototype of the design is placed on a test bench where it can be easily accessed by manual testing equipment (such as oscilloscopes, multimeters, etc.). In Verilog HDL, test benches are often virtual in nature allowing the functionality of the unit under test to be verified by the use of simulation. In the case of this lab, the complexity of the waveforms make simulation difficult and very time consuming so the test bench is actually a physical one that is designed to drive and monitor in real time the IP component to be tested.

In general, a test bench (virtual or physical) is an environment that is constructed that will allow the component being designed to be tested using a collection of testing tools in which the suite of these tools is often designed specifically for the unit being tested. In FPGAs these testing tools may be other hardware component modules that were created to drive and receive data from the component module that is being designed. A test bench must provide the following functionality.

- An interface with the component that is to be tested: The product component that is being designed must be able to be incorporated into the test bench. In Verilog HDL this is often done by instantiating it as a structural component.
- Stimulus Generation Mechanism: The test bench must drive the component being tested with the necessary stimulus to adequately exercise it over the desired range of functionality. In this laboratory, the test bench allows the stimulus to be manually generated or it can be generated automatically from the switch driver module.
- Verification Procedures: The test bench environment should support the verification that the input stimulus has produced the desired output. This can be done in a manner that directly involves the human test engineer (visual displays such as waveforms, observable outputs , etc.) or fully automated procedures that provide design verification with minimal impact from the user. In this laboratory the verification procedure is to be performed by the human engineer.
- Acceptance Criteria: The basic design acceptance criteria that indicates the component under test has met the slated design goals.

In this laboratory the component that is to be tested is the *texter* module and its subcomponents. Stimulus generation is performed through either the manual generation of texting patterns (morse code), or by utilizing the externally created *sw_driver* IP core module that was designed to send a prerecorded morse code texting pattern at various speeds. This module produces an output that can be used to drive the main input switch of the texter module in a manner that implements the correct morse code sequences at varying speeds (7, 10, 13, 15, 18, 20, 25 and 30 WPM). Selection between the manual and automated entry of switch input to the texter module that is being tested is performed by a specially designed multiplexer component, *sw_mux*, that is activated by another switch. The *usend*, *lcd*, *vga*, *tone_gen*, *bintohex*, and modules discrete *LED* outputs are used to provide the visual and audio output of our tests. The verification procedures rely upon a human observer utilizing these outputs to verify the correct functionality of the design. The acceptance criteria would involve a set of design experiments that would fully exercise the texter module over its full scope of operation. Such criteria should test for all allowable outputs of the system, over all range of operating speeds and should utilize both manual and automated texting. During the operation of the test bench the main component being tested (the *texter* module in this case) may have to be redesigned to meet the acceptance criteria. Figure 4 shows the external connections to the DE2-115 that are needed to interface to the external switch button, drive the PC's serial terminal, SPP bluetooth interface, DE2-115's LCD, external VGA monitor, wireless AM tone element. Figure 5 is a block diagram of the physical test bench, *lab5_ptb.v,*to be used to drive the texter module.

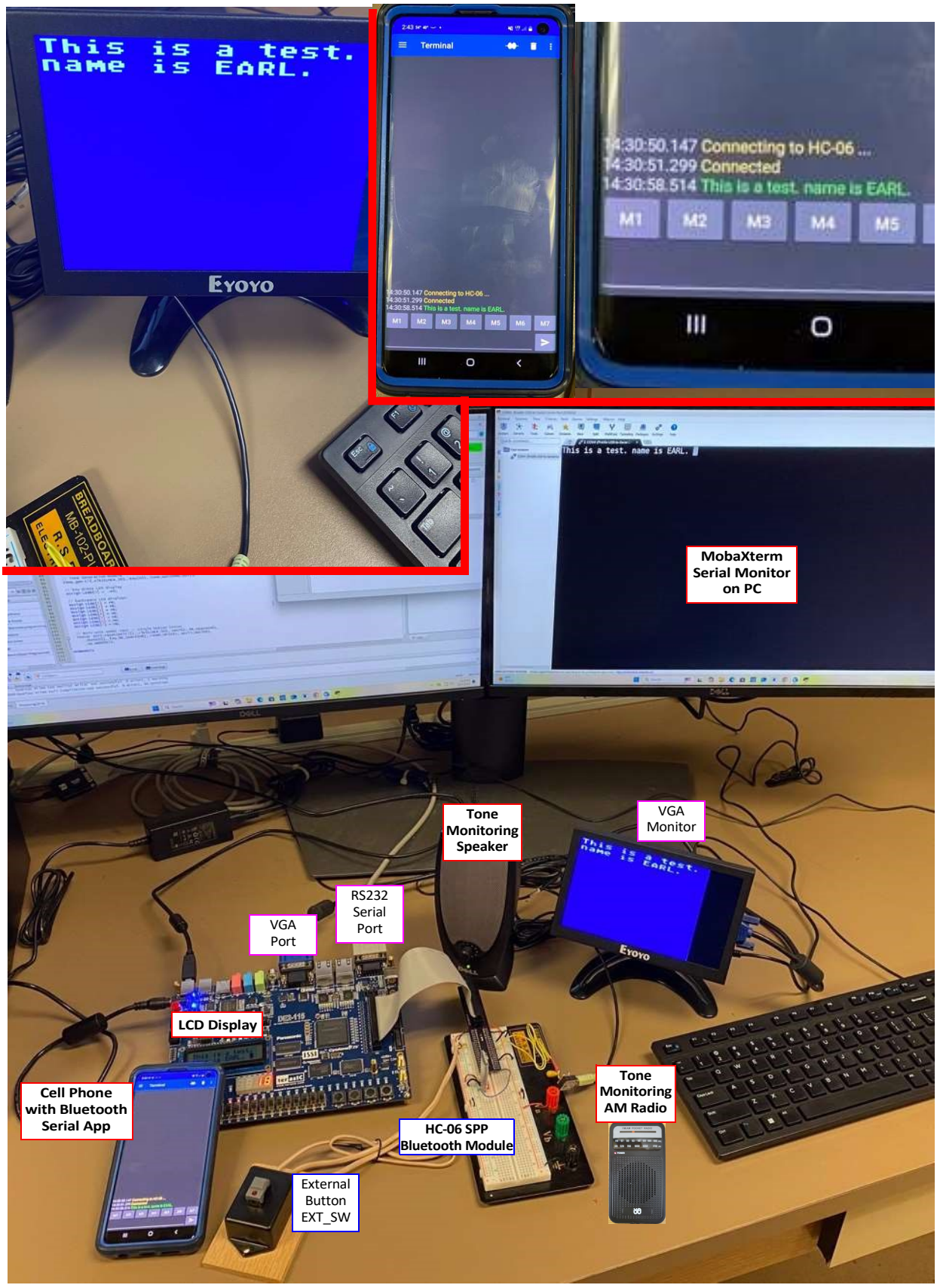
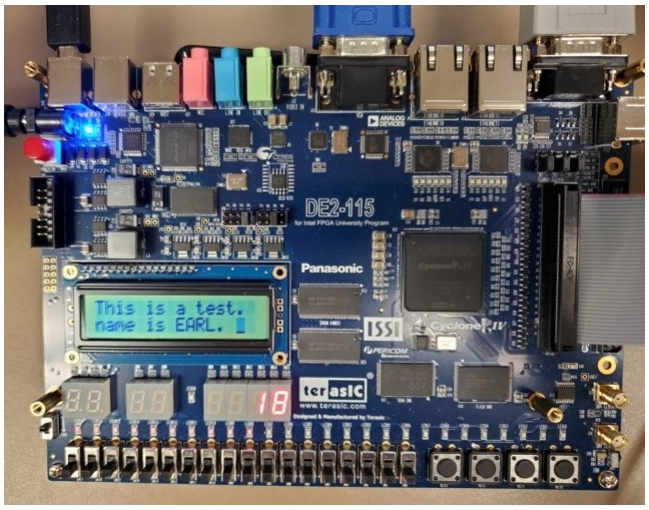
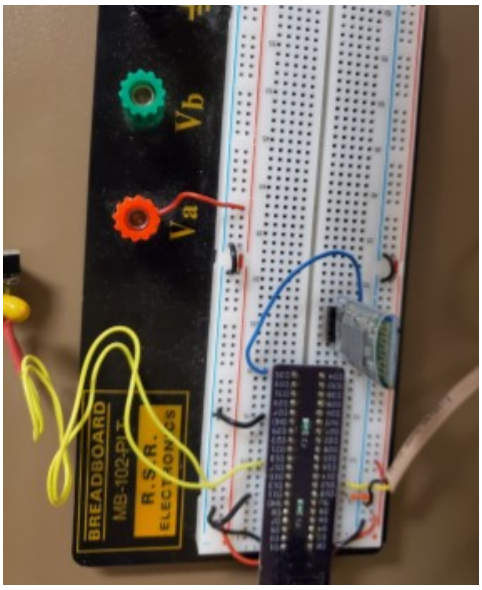


Figure 4a: Testbench Setup





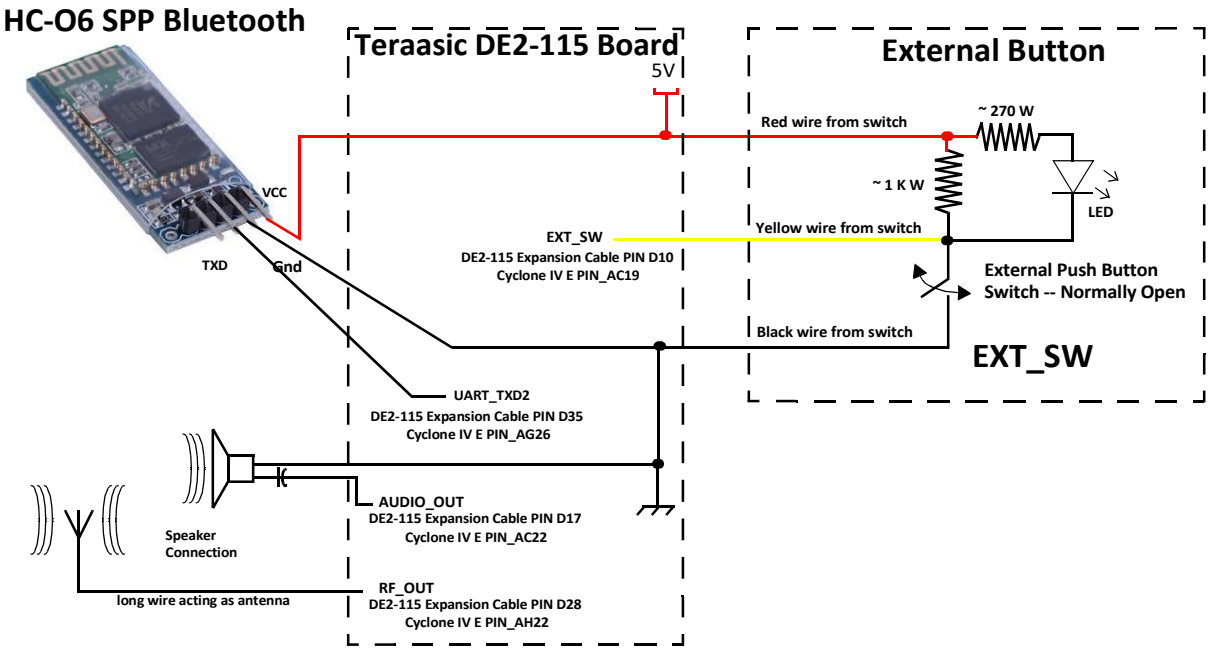
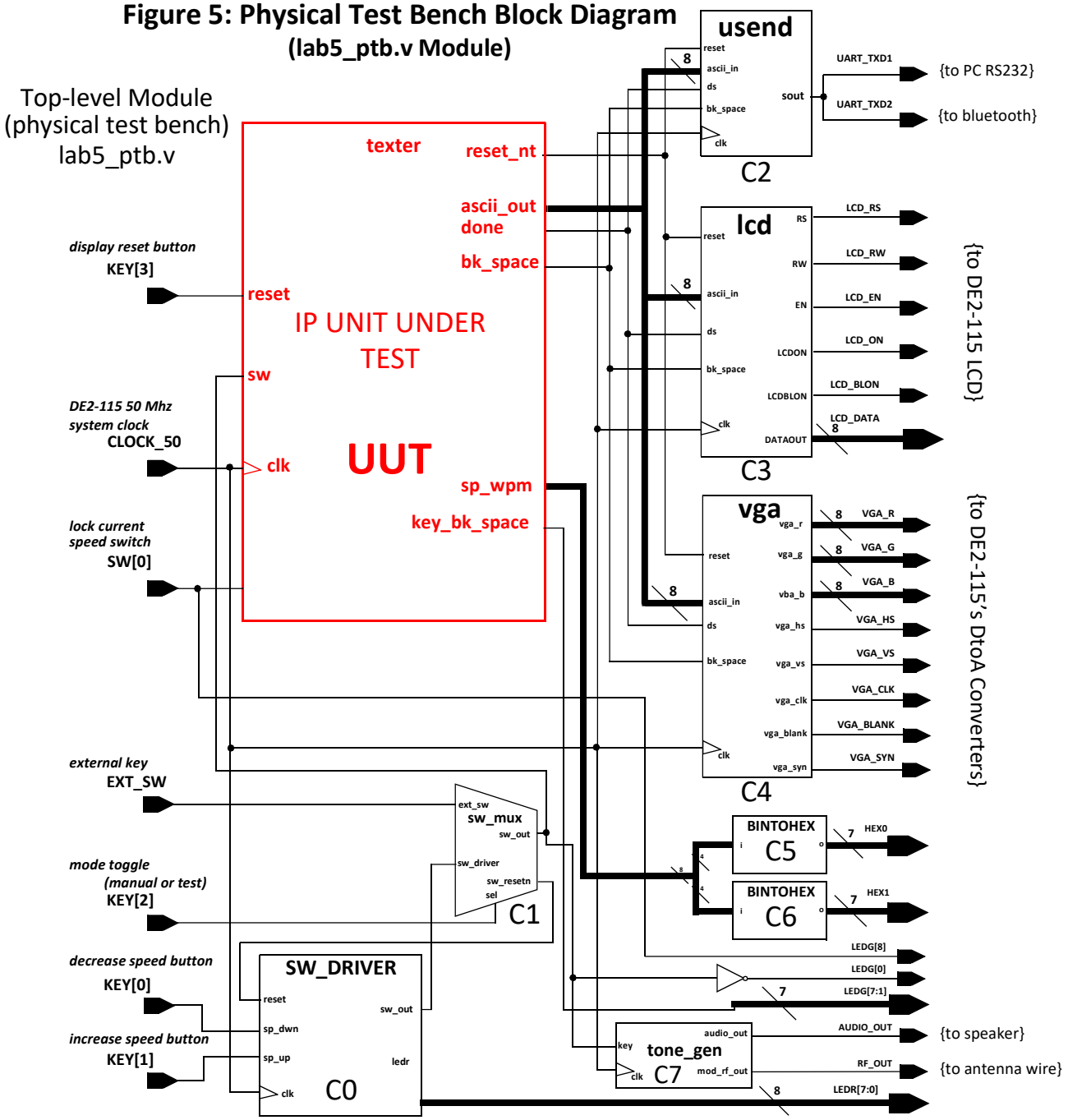


Figure 4b: Testbench Setup and External Connections

Figure 5: Physical Test Bench Block Diagram (lab5_ptb.v Module)




```
module lab5_ptb(input EXT_SW, CLOCK_50, input [0:0] SW, input [3:0] KEY,
   output UART_TXD1,UART_TXD2,VGA_BLANK,VGA_CLK,VGA_VS,VGA_HS,
   output LCD_BLON,LCD_ON,LCD_EN,LCD_RW,LCD_RS,VGA_SYNC,AUDIO_OUT,RF_OUT,
   output [6:0] HEX0,HEX1,
   output [7:0] LCD_DATA,LEDR,VGA_B,VGA_G,VGA_R, output [8:0] LEDG);

   // internal wires and busses
   wire n0, n1, n2, n3, n4, n5, n6, n7;
   wire [7:0] b0, b1;

   // switch driver module
   sw_driver C0(.clk(CLOCK_50),.sp_up(KEY[1]),.sp_dwn(KEY[0]),
      .reset(n0),.sw_out(n1),.ledr(LEDR));

   // switch multiplexer module
   sw_mux C1(.i0(EXT_SW),.i1(n1),.sel(KEY[2]),.sw_resetn(n0),
      .sw_out(n5));

   // RS-232 Asynchonous serial port output module
   usend C2 (.clk(CLOCK_50),.reset(n2),.ds(n3),.back_sp(n4),
             .ascii_in(b0),.sout(n7));
   assign UART_TXD1 = n7;
   assign UART_TXD2 = n7;

   // LCD display module
   lcd C3(.clk(CLOCK_50),.reset(n2),.ds(n3),.back_sp(n4),
      .ascii_in(b0),.rs(LCD_RS),.rw(LCD_RW),.en(LCD_EN),
      .lcdon(LCD_ON),.lcdblon(LCD_BLON),.data_out(LCD_DATA));

   // VGA display module
   vga C4(.clk(CLOCK_50),.reset(n2),.ds(n3),.back_sp(n4),
      .ascii_in(b0),.vga_hs(VGA_HS),.vga_vs(VGA_VS),
      .vga_clk(VGA_CLK),.vga_blank(VGA_BLANK),.vga_syn(VGA_SYNC),
      .vga_b(VGA_B),.vga_g(VGA_G),.vga_r(VGA_R));

   // BCD to Hex modules
   bintohex C5(.i(b1[3:0]),.o(HEX0));
   bintohex C6(.i(b1[7:4]),.o(HEX1));

   // Tone Generation Module
   tone_gen C7(.clk(CLOCK_50),.key(n5),.audio_out(AUDIO_OUT),
      .mod_rf_out(RF_OUT));

   // key press LED display
   assign LEDG[0] =  ~n5;

   // backspace LED displays
   assign LEDG[1] = n6;
   assign LEDG[2] = n6;
   assign LEDG[3] = n6;
   assign LEDG[4] = n6;
   assign LEDG[5] = n6;
   assign LEDG[6] = n6;
   assign LEDG[7] = n6;

   // indicate if speed tracking is on by turning on LEDG[8]
   assign LEDG[8] = ~SW[0];

   // main unit under test -- single button texter
   texter UUT(.reset(KEY[3]),.clk(CLOCK_50),.sw(n5),.sp_freeze(SW[0]),.bk_space(n4),
      .done(n3),.key_bk_space(n6),.reset_nt(n2),.ascii_out(b0),
      .sp_wpm(b1));

endmodule
```

DE2-115 Pin Assignments

Node Name	Direction	Location
AUDIO_OUT	Output	PIN_AC22
CLOCK_50	Input	PIN_Y2
EXT_SW	Input	PIN_AC19
HEX0[6]	Output	PIN_H22
HEX0[5]	Output	PIN_J22
HEX0[4]	Output	PIN_L25
HEX0[3]	Output	PIN_L26
HEX0[2]	Output	PIN_E17
HEX0[1]	Output	PIN_F22
HEX0[0]	Output	PIN_G18
HEX1[6]	Output	PIN_U24
HEX1[5]	Output	PIN_U23
HEX1[4]	Output	PIN_W25
HEX1[3]	Output	PIN_W22
HEX1[2]	Output	PIN_W21
HEX1[1]	Output	PIN_Y22
HEX1[0]	Output	PIN_M24
KEY[3]	Input	PIN_R24
KEY[2]	Input	PIN_N21
KEY[1]	Input	PIN_M21
KEY[0]	Input	PIN_M23
LCD_BLON	Output	PIN_L6
LCD_DATA[7]	Output	PIN_M5
LCD_DATA[6]	Output	PIN_M3
LCD_DATA[5]	Output	PIN_K2
LCD_DATA[4]	Output	PIN_K1
LCD_DATA[3]	Output	PIN_K7
LCD_DATA[2]	Output	PIN_L2
LCD_DATA[1]	Output	PIN_L1
LCD_DATA[0]	Output	PIN_L3
LCD_EN	Output	PIN_L4
LCD_ON	Output	PIN_L5
LCD_RS	Output	PIN_M2
LCD_RW	Output	PIN_M1
LEDG[8]	Output	PIN_F17
LEDG[7]	Output	PIN_G21
LEDG[6]	Output	PIN_G22
LEDG[5]	Output	PIN_G20
LEDG[4]	Output	PIN_H21
LEDG[3]	Output	PIN_E24
LEDG[2]	Output	PIN_E25
LEDG[1]	Output	PIN_E22
LEDG[0]	Output	PIN_E21
LEDR[7]	Output	PIN_H19
LEDR[6]	Output	PIN_J19
LEDR[5]	Output	PIN_E18
LEDR[4]	Output	PIN_F18
LEDR[3]	Output	PIN_F21
LEDR[2]	Output	PIN_E19
LEDR[1]	Output	PIN_F19
LEDR[0]	Output	PIN_G19
RF_OUT	Output	PIN_AH22
SW[0]	Input	PIN_AB28
UART_TXD1	Output	PIN_G9
UART_TXD2	Output	PIN_AG26
VGA_B[7]	Output	PIN_D12
VGA_B[6]	Output	PIN_D11
VGA_B[5]	Output	PIN_C12
VGA_B[4]	Output	PIN_A11
VGA_B[3]	Output	PIN_B11
VGA_B[2]	Output	PIN_C11
VGA_B[1]	Output	PIN_A10
VGA_B[0]	Output	PIN_B10
VGA_BLANK	Output	PIN_F11
VGA_CLK	Output	PIN_A12
VGA_G[7]	Output	PIN_C9
VGA_G[6]	Output	PIN_F10
VGA_G[5]	Output	PIN_B8
VGA_G[4]	Output	PIN_C8
VGA_G[3]	Output	PIN_H12
VGA_G[2]	Output	PIN_F8
VGA_G[1]	Output	PIN_G11
VGA_G[0]	Output	PIN_G8
VGA_HS	Output	PIN_G13
VGA_R[7]	Output	PIN_H10
VGA_R[6]	Output	PIN_H8
VGA_R[5]	Output	PIN_J12
VGA_R[4]	Output	PIN_G10
VGA_R[3]	Output	PIN_F12
VGA_R[2]	Output	PIN_D10
VGA_R[1]	Output	PIN_E11
VGA_R[0]	Output	PIN_E12
VGA_SYNC	Output	PIN_C10
VGA_VS	Output	PIN_C13

Figure 6: Physical Test Bench DE2-115 Pin Assignments

Assignment

Complete your design of the *texter_control* module and fully integrate it into the overall design (including the physical test bench portion of the design). Demonstrate that the design works correctly to your laboratory instructor both by varying the speed of the automated message as well as sending your name manually using the external switch button. Note that this laboratory introduces you to a large number of I/O devices that you can use to view the output of the *texter* module to verify the correctness of your *texter_control* module implementation. The purpose for including all these modules in the testbench was primarily to illustrate the ease at which new output devices can be added if the interface to them is either designed in a similar manner or if they are designed in a way that they all adhere to an existing standard. In this physical testbench the interface between the texter module and the output devices such as the LCD, VGA, and serial all utilize the same simple custom interface that allowing these devices to respond to the texter model simultaneously.

To verify your design of the *texter_control* module, though you only need to use <u>one</u> output device, such as the LCD that is built-in to the DE2-115 of the physical test bench.

Post Laboratory Questions

Upon completion of this assignment you should answer the following post laboratory questions and include your answers as part of your individual laboratory report.

1. What are the major characteristics of synchronous sequential digital design?

   What are the differences between the control path and data path? What are the clocking requirements?

2. During compilation, there were many warning messages. List at least five different types of messages that you observed. For each of these messages, specify what you perceive to be their meaning. Also specify why these messages could be ignored and the design still functioned correctly.

3. Create an equivalent Extended State Diagram that has the same functionality as the Algorithmic State Machine representation of the texter_control module that was presented in Figure 3 of this lab. You should adhere to the rules of Extended State Diagrams, that was presented in the CPE 322 class. (Note that the power point slides for both Extended State Diagrams and Algorithmic State Machine representations have been placed under the lab5 folder on Canvas).

4. Based upon your understanding of the Algorithmic State Machine representation of the texter_control module and the functionality of the other IP core elements that make up the texter module, explain the function of each of the six states in the system shown in Figure 3.

   Do you think it would be possible to reduce the number of states and still preserve the functionality? Explain your answer.

5. In what ways does this physical testbench differ from traditional test benches that are used for simulations?

   What are the advantages and disadvantages to utilizing the physical testbench approach over that of a simulation-only approach that utilizes a traditional testbench?

   What would be the challenges associated with creating this testbench?

6. This design could also have been implemented in software using commercially available embedded instruction set processor (micro-controller) environment (Arduino, Raspberry PI, etc.). What are the general design trade-offs one must consider when choosing when components should be implemented in hardware or software? Consider such items as performance requirements (processing speed), the ease of implementation, the ability to expand and alter the system as new requirements emerge, the ability to execute in a real-time/deterministic manner, the energy utilization, etc.