THE UNIVERSITY OF ALABAMA IN HUNTSVILLE
Department of Electrical and Computer Engineering

CPE 324 Advanced Logic Design Laboratory

Laboratory Assignment #10

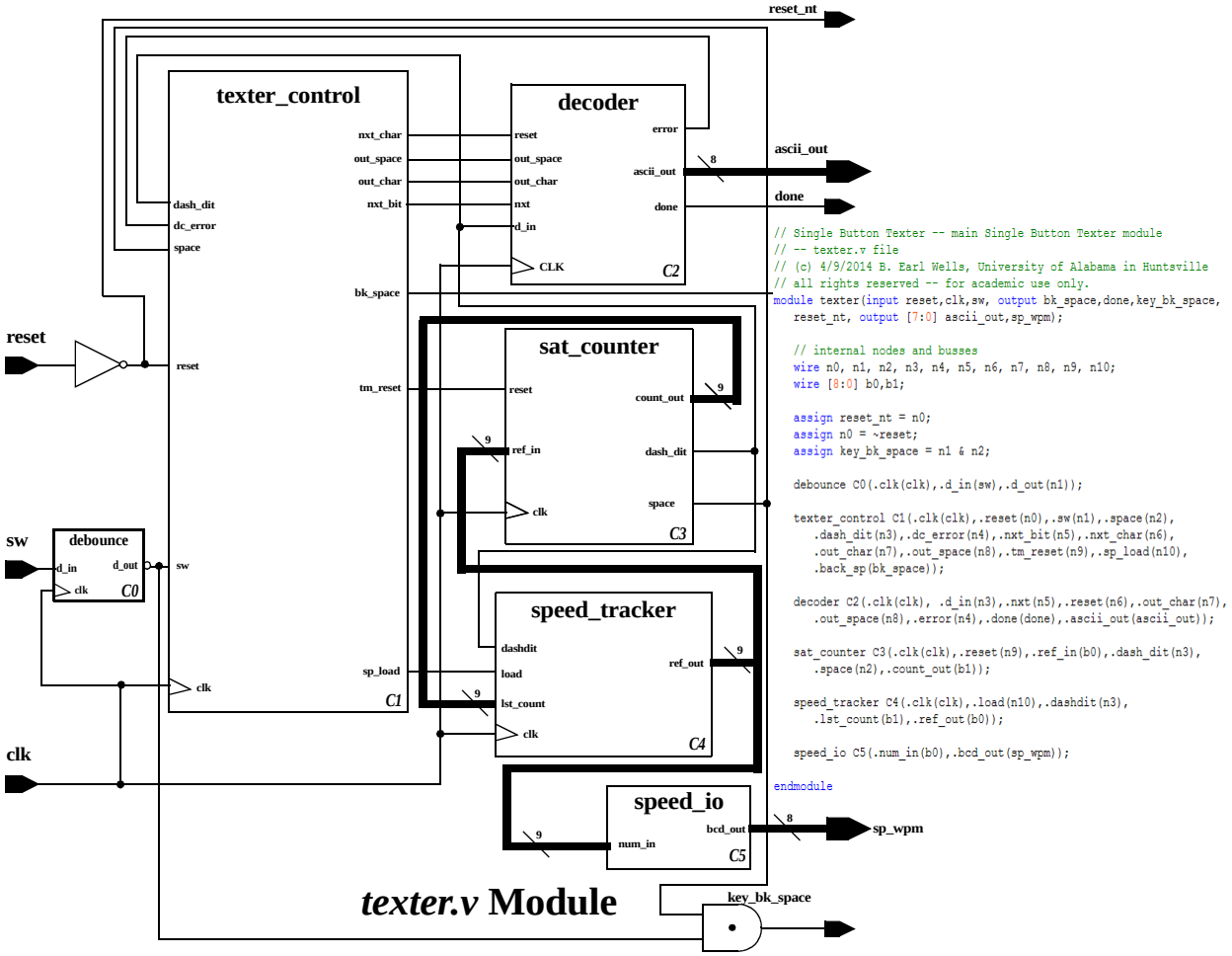
Single Button Texter

(15 % of Final Grade)

The purpose of this laboratory project is to give each student the opportunity to understand the basic trade-offs associated when choosing whether to implement a design in hardware or software. Students will be required to evaluate the trade-offs associated with creating a design entirely in hardware or implementing the design using a combination of software and hardware through the use of a soft-core embedded processor. It is assumed that the digital system is prototyped in a System on a Programable Chip, SoPC, environment.

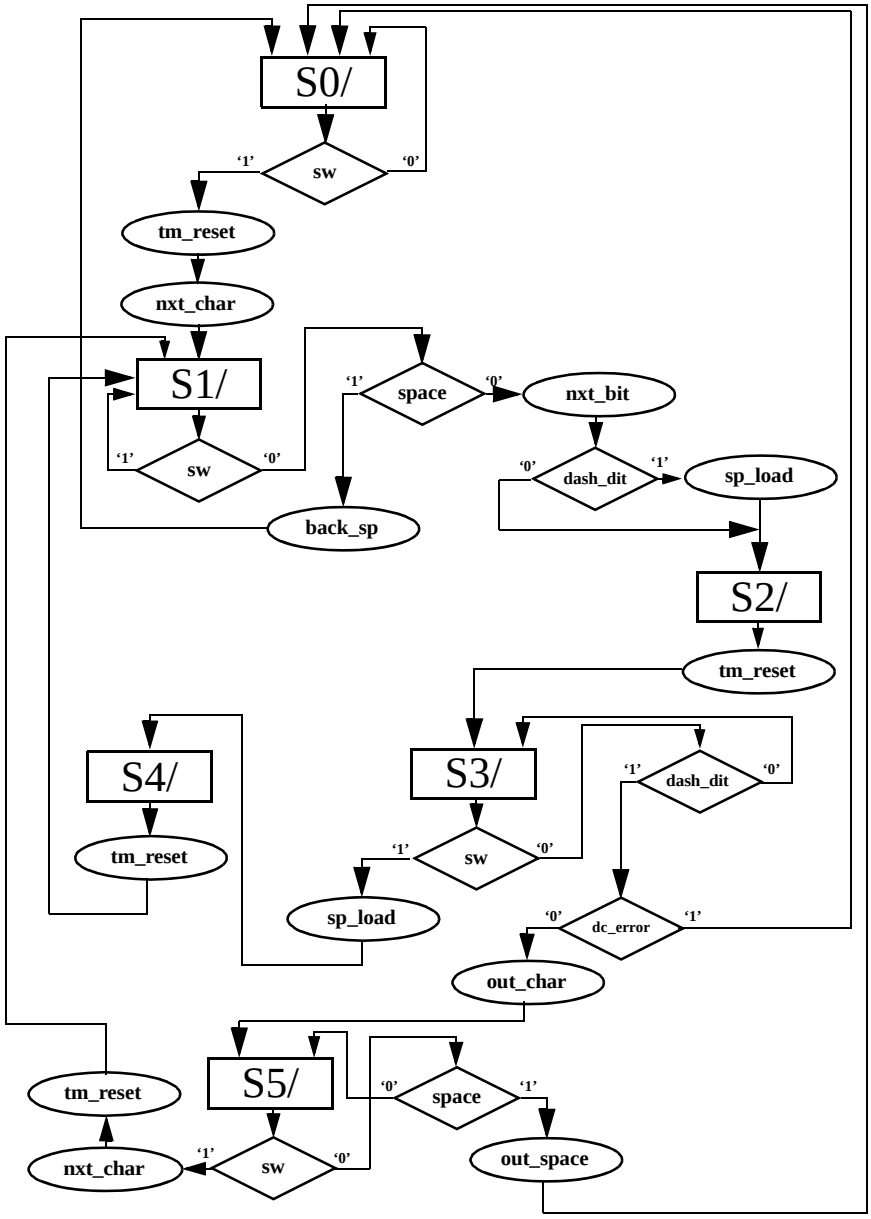
Part I: Complete Verilog HDL Implementation (Rapid Prototyping)

It is assumed that you are working as part of a team of engineers that are to design the *Single Button Texter*. The purpose of this texter is to allow someone to send text messages using a single button using an ubiquitous sending device without the need to look at a keypad. The design team has met and formulated the following high-level design as discussed in the CPE 322 class.




```
// Single Button Texter -- main Single Button Texter module
// -- texter.v file
// (c) 4/9/2014 B. Earl Wells, University of Alabama in Huntsville
// all rights reserved -- for academic use only.
module texter(input reset,clk,sw, 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;
   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));

   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));

   speed_tracker C4(.clk(clk),.load(n10),.dashdit(n3),
      .lst_count(b1),.ref_out(b0));

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

endmodule
```

The design for the *texter_control* module which is described using the following ASM Chart shown below. This represents the major control section for the overall system. The final design has been implement and can be tested using an Altera/Teriasic DE-2 platform. .



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

Background Information

Good news, your design team has completed the preliminary design and the unit tests of all of the functional components! The component files are named *texter.v, sw_driver.v, testbench.v,* and *i_o.v.* Note that the *lab10.v* file contains the highest-level representation and should be the top level of the design. These files can be found on the CPE 324 course Canvas site. The messaging protocol that has been adopted is a modified form of the Morse code which is outlined in the figure and table shown below**.**

Each character is made up of one or more dots(.) or dashes (_) as shown in the table.

A dot represents the basic unit of time that a switch is pressed, the dash represents the long period of time that a switch is pressed.

The time duration of a dot is one unit pulse.

The time duration of a dash is three unit pulses.

The time between each dot/dash that make up each morse code character is one unit pulse.

The time between consecutive characters is three unit pulses.

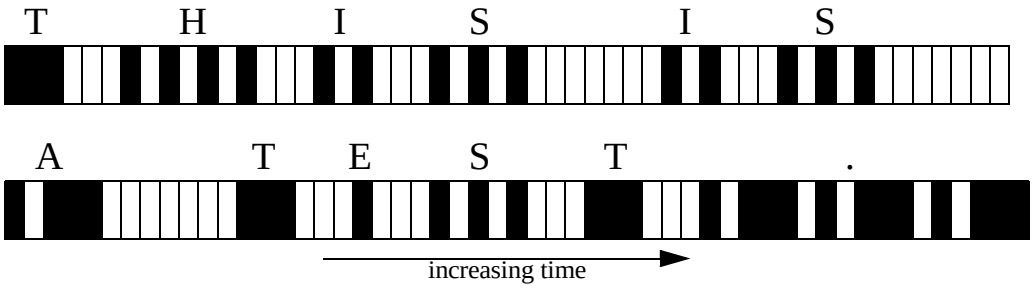
The time between words (or groups of characters) is seven unit pulses.

Proposed Texting Code -- Modified Morse Code

A	. _	L	. _ . .	W	. _ _	7	_ _ . . .
B	_ . . .	M	_ _	X	_ . . _	8	_ _ _ . .
C	_ . _ .	N	_ .	Y	_ . _ _	9	_ _ _ _ .
D	_ . .	O	_ _ _	Z	_ _ . .	.	. _ . _ . _
E	.	P	. _ _ .	0	_ _ _ _ _	?	. . _ _ . .
F	. . _ .	Q	_ _ . _	1	. _ _ _ _	,	_ _ . . _ _
G	_ _ .	R	. _ .	2	. . _ _ _	;	_ . _ . _ .
H	. . . .	S	. . .	3	. . . _ _	:	_ _ _ . . .
I	. .	T	_	4	. . . . _	--	_ . . . . _
J	. _ _ _	U	. . _	5	. . . . .	$	. . . _ . . _
K	_ . _	V	. . . _	6	_	case toggle	. . _ _

= switch pressed = switch released

Example -- for the sentence "*this is a test*."



Physical Test Bench

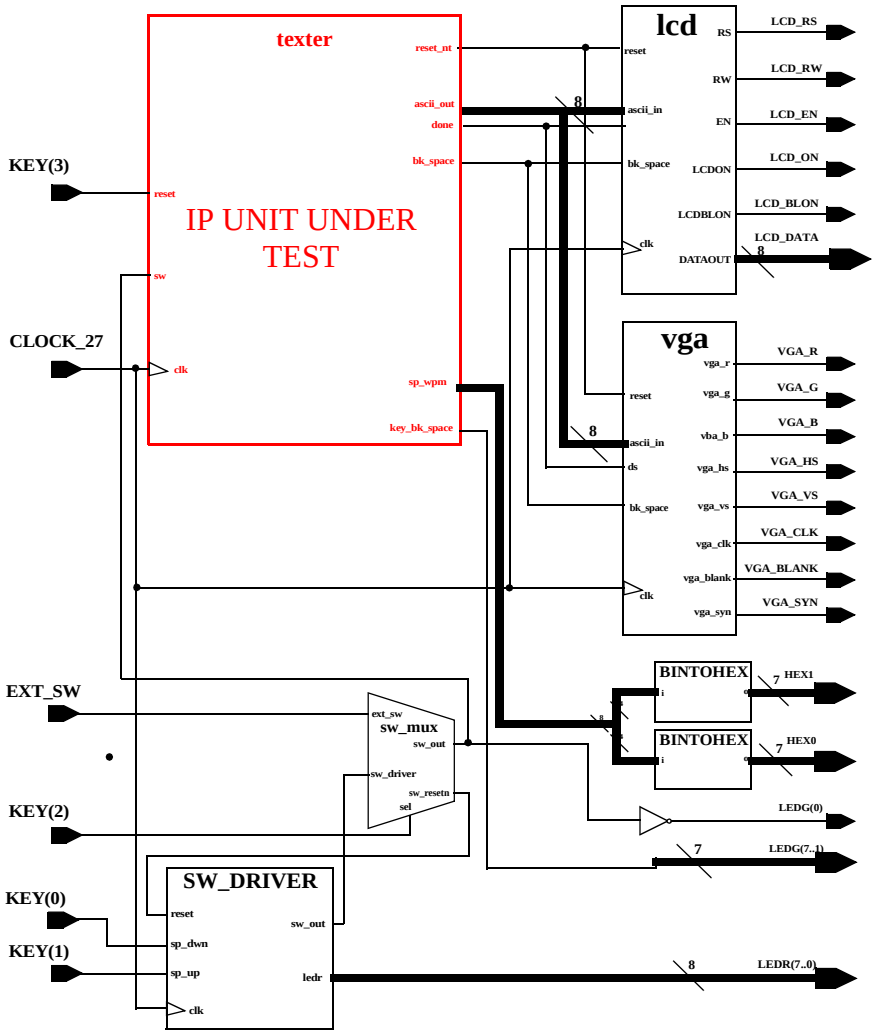
A *test bench* is a environment used to verify that the component being designed functions as intended. It gets its name from the traditional method of testing hardware and software in the laboratory where the physical prototype of the design is placed on a test bench where it can be easily accessed 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 so the test bench is actually a physical one that is designed to drive and monitor the IP component to be tested in real time.

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. In general the 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 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. The stimulus generation mechanism is manual generation of texting patterns (morse code), or an automated generation of texting patterns that utilizes the *sw_driver* software core. 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 *lcd*, *vga*, *bintohex*, and discrete *LED* outputs are used to provide the visual output of our tests. The verification procedures rely upon a human observer utilizing these outputs to demonstrate 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 probably 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. The next figure is a block diagram of the test bench for the texter control module. The file *testbench.v* contains the structural design for this module

testbench.v Module



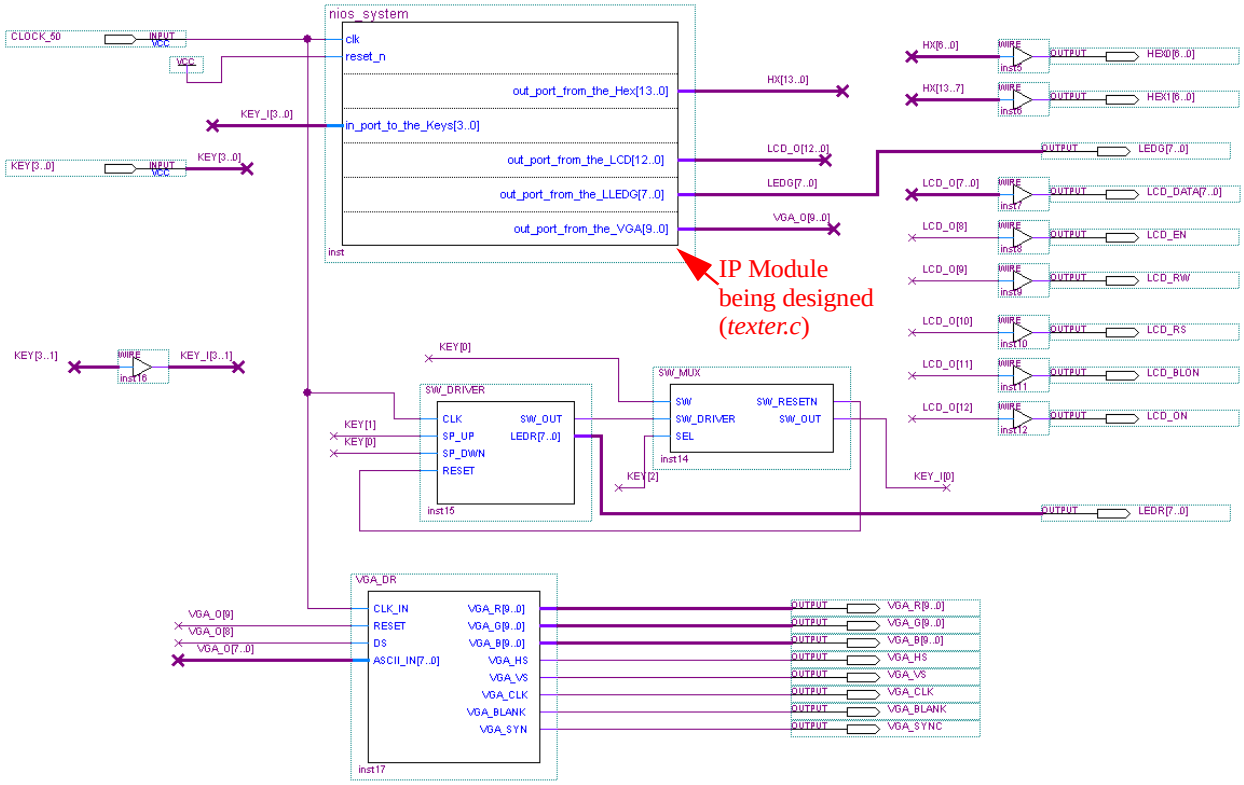

```
module lab10(input EXT_SW, CLOCK_27, input [3:0] KEY,
   output VGA_BLANK, VGA_CLK, VGA_VS, VGA_HS, LCD_BLON, LCD_ON,
   output LCD_EN, LCD_RW, LCD_RS, VGA_SYNC,
   output [6:0] HEX0, HEX1,
   output [7:0] LCD_DATA, LEDG, LEDR,
   output [9:0] VGA_B, VGA_G, VGA_R);

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

   // switch driver module
   sw_driver C0(.clk(CLOCK_27),.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));

   // LCD display module
   lcd C2(.clk(CLOCK_27),.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 C3(.clk(CLOCK_27),.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));
// switch multiplexer module
sw_mux C1(.i0(EXT_SW),.i1(n1),.sel(KEY[2]),.sw_resetn(n0),
    .sw_out(n5));
```

```
// LCD display module
lcd C2(.clk(CLOCK_27),.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 C3(.clk(CLOCK_27),.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 C4(.i(b1[3:0]),.o(HEX0));
bintohex C5(.i(b1[7:4]),.o(HEX1));

// 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;

// main unit under test -- single button texter
texter UUT(.reset(KEY[3]),.clk(CLOCK_27),.sw(n5),.bk_space(n4),
   .done(n3),.key_bk_space(n6),.reset_nt(n2),.ascii_out(b0),
   .sp_wpm(b1));
endmodule
```

Part II: Alternative Implementation (NIOS II Soft-Core ISP)

It is assumed that while you have been developing the texter module for the hardware design of the single button texter that another group within your company is simultaneously investigating the feasibility of utilizing a commercially available embedded instruction set processor. The processor that has been chosen to demonstrate this design is the basic Altera's NIOS IIe softcore processor. This processor incorporates only the basic architectural features and is to be clocked at 50 Mhz and reset once after the FPGA's configuration. This NIOS IIe processor is implemented using encrypted Verilog HDL which is automatically generated using the Altera Qsys high-level synthesis utility that is part of the Quartus II Design package. Once created, the NIOS processor can be interconnected to other hardware within the Cyclone II FPGA just like any other component. This makes it possible to integrate it with user-created hardware. Before the NIOS processor component is generated, users have the option to create internal components or they can specify external memory mapped I/O ports that can be used to communicate with other peripheral components. The latter approach was the one taken by the NIOS II soft-core processor team to implement the texter. They chose to implement the texter entirely in ANSII C software using a polling mechanism. In creating an equivalent test bench module they incorporated the same automated switch driver (*sw_driver*), switch multiplexer (*sw_mux*), and a modified version of the VGA driver (*vga_dr*). The bcd display (*bintohex*) functionality needed to output the texting speed was performed within the NIOS soft-core processor along with the LCD interface logic. The Quartus II top-level diagram is shown in the figure below.

Equivalent *lab10.v* Module that utilizes a Single NIOS II soft-core IP processor



The software portion of the design was written in standard ASCII C and was compiled utilizing the standard *gnu* tool chain. It was downloaded (after the FPGA was configured) and executed using another tool that utilized the JTAG I/O capability of the DE-2 board. The results also met the acceptance criteria that was set for the test bench.The *texter.c* file can be accessed on the CPE 324 course Canvas site.

This represents an extreme case where as much functionality as possible has been placed in the software with the minimum functionality being placed in the hardware. This type of software utilizes the serialized encoding and communication strategy that is often called **bit banging**. In bit banging the software itself is responsible for setting and sampling the states of its memory mapped digital inputs and outputs directly. It is responsible for maintaining all of the timing, synchronization and system protocol parameters. The advantage of bit banging is that hardware cost can be minimized significantly but the overall expandability, portability and functionality of this type of software is often very limited.

At the end of this evaluation process, the major attributes of the two designs can be compared to one another to fully explore the design trade-offs associated with these two extremes within the hardware/software design space. To do this a simple design was created that utilized the basic texter module (implemented in the first case as an application-specific Verilog HDL module and in the second by utilizing a NIOS IIe processor and an LCD interface -- implemented in hardware in the first design, and in software in the second). In other words, the automated switch driver, switch multiplexer, and the VGA interface were removed for both designs when determining the following statistics.

Verilog HDL only Implementation (resource utilization):

Logic Cells	FF (logic Registers)	Memory Bits	M4Ks	LUT-only LCs	Register-Only LCs	LUT/ Register LCs
991	302	0	0	689	47	255

Number of pins: 48
FPGA Configuration time: 1.9 Minutes
Maximum Sending Speed > 1000 WPM

NIOS II ISP Implementation (resource utilization):

Logic Cells	FF (logic Registers)	Memory Bits	M4Ks	LUT-only LCs	Register-Only LCs	LUT/ Register LCs
1522	825	75776	22	697	138	687

Number of pins: 48
Program size: 4,240 Bytes
FPGA Configuration time: 5.5 Minutes
Program Compilation time: 30 seconds
Maximum Sending Speed > 1000 WPM

Note: No actual Implementation of the design is required for this lab but this resource usage information is needed in order to answer the Post Laboratory Questions.

Post Laboratory Questions

Students are also expected to answer the following questions that reflect upon their laboratory experiences. The answers to these questions should be part of their laboratory report.

1. Compare the two implementations discussed in the previous section based upon the overall FPGA hardware resource utilization. From the FPGA's perspective (not the DE-2s) how many complete representations of the texter/LCD interface modules could you put in a single Cyclone II EP2C35F672C6 Device before the needed resources would exceed the FPGA's capability? If this resource was not the limiting factor what resource would be the next limiting factor and how many texter/LCD interface modules could you place in the Cyclone II EP2C35F672C6 if the LCD hardware itself was not a limiting resource?

2. From a debug perspective which implementation is easier to implement and debug if a change is to be made to the basic texter module? Justify your answer.

3. Examining the software implementation file *texter.c* what is the primary method by which data is sent and received from external devices? What would happen if the PLL in the FPGA was used to double the clock frequency?

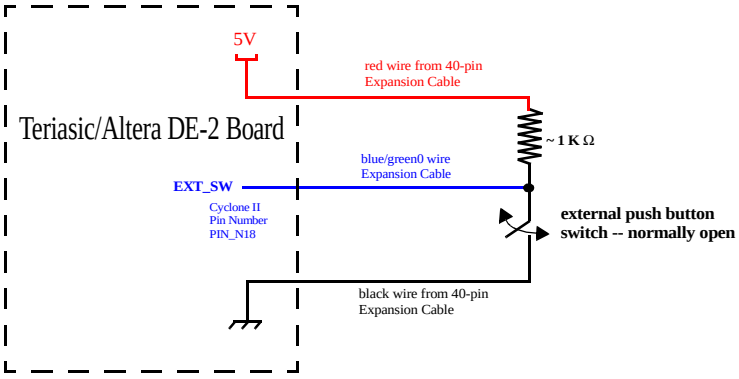
4. In the NIOS II full test bench implementation, why do you think that the VGA functionality was placed in hardware but the LCD can be implemented in software?

5. In general what are the design trade-offs for implementing components in hardware or software in a hybrid system?

6. Explain the issues involved in making a decision on which design is better in the case of this lab and when one should migrate functionality from hardware to software and vice-a-versa

DE-2 Setup

To manually operate the texter requires the inclusion of a separate switch that is connected to the blue/green0 wire of the 40-pin expansion cable as shown below.



Note that the switch is connected across the 5V VCC and the common point ground using a pull-up resister. A complete listing of the DE-2 Cyclone II pin assignments is shown below:

Node Name	Direction	Location
CLOCK_27	Input	PIN_D13
EXT_SW	Input	PIN_N18
HEX0[6]	Output	PIN_V13
HEX0[5]	Output	PIN_V14
HEX0[4]	Output	PIN_AE11
HEX0[3]	Output	PIN_AD11
HEX0[2]	Output	PIN_AC12
HEX0[1]	Output	PIN_AB12
HEX0[0]	Output	PIN_AF10
HEX1[6]	Output	PIN_AB24
HEX1[5]	Output	PIN_AA23
HEX1[4]	Output	PIN_AA24
HEX1[3]	Output	PIN_Y22
HEX1[2]	Output	PIN_W21
HEX1[1]	Output	PIN_V21
HEX1[0]	Output	PIN_V20
KEY[3]	Input	PIN_W26
KEY[2]	Input	PIN_P23
KEY[1]	Input	PIN_N23
KEY[0]	Input	PIN_G26
LCD_BLON	Output	PIN_K2
LCD_DATA[7]	Output	PIN_H3
LCD_DATA[6]	Output	PIN_H4
LCD_DATA[5]	Output	PIN_J3
LCD_DATA[4]	Output	PIN_J4
LCD_DATA[3]	Output	PIN_H2
LCD_DATA[2]	Output	PIN_H1
LCD_DATA[1]	Output	PIN_J2
LCD_DATA[0]	Output	PIN_J1
LCD_EN	Output	PIN_K3
LCD_ON	Output	PIN_L4
LCD_RS	Output	PIN_K1
LCD_RW	Output	PIN_K4
LEDG[7]	Output	PIN_Y18
LEDG[6]	Output	PIN_AA20
LEDG[5]	Output	PIN_U17
LEDG[4]	Output	PIN_U18
LEDG[3]	Output	PIN_V18
LEDG[2]	Output	PIN_W19
LEDG[1]	Output	PIN_AF22
LEDG[0]	Output	PIN_AE22
LEDR[7]	Output	PIN_AC21
LEDR[6]	Output	PIN_AD21
LEDR[5]	Output	PIN_AD23
LEDR[4]	Output	PIN_AD22
LEDR[3]	Output	PIN_AC22
LEDR[2]	Output	PIN_AB21
LEDR[1]	Output	PIN_AF23
LEDR[0]	Output	PIN_AE23

Node Name	Direction	Location
VGA_B[9]	Output	PIN_B12
VGA_B[8]	Output	PIN_C12
VGA_B[7]	Output	PIN_B11
VGA_B[6]	Output	PIN_C11
VGA_B[5]	Output	PIN_J11
VGA_B[4]	Output	PIN_J10
VGA_B[3]	Output	PIN_G12
VGA_B[2]	Output	PIN_F12
VGA_B[1]	Output	PIN_J14
VGA_B[0]	Output	PIN_J13
VGA_BLANK	Output	PIN_D6
VGA_CLK	Output	PIN_B8
VGA_G[9]	Output	PIN_D12
VGA_G[8]	Output	PIN_E12
VGA_G[7]	Output	PIN_D11
VGA_G[6]	Output	PIN_G11
VGA_G[5]	Output	PIN_A10
VGA_G[4]	Output	PIN_B10
VGA_G[3]	Output	PIN_D10
VGA_G[2]	Output	PIN_C10
VGA_G[1]	Output	PIN_A9
VGA_G[0]	Output	PIN_B9
VGA_HS	Output	PIN_A7
VGA_R[9]	Output	PIN_E10
VGA_R[8]	Output	PIN_F11
VGA_R[7]	Output	PIN_H12
VGA_R[6]	Output	PIN_H11
VGA_R[5]	Output	PIN_A8
VGA_R[4]	Output	PIN_C9
VGA_R[3]	Output	PIN_D9
VGA_R[2]	Output	PIN_G10
VGA_R[1]	Output	PIN_F10
VGA_R[0]	Output	PIN_C8
VGA_SYNC	Output	PIN_B7
VGA_VS	Output	PIN_D8