CPE 324-03: Advanced Digital Logic Design Laboratory

Lab 5

Single Button Texter

Submitted by: Hannah Kelley

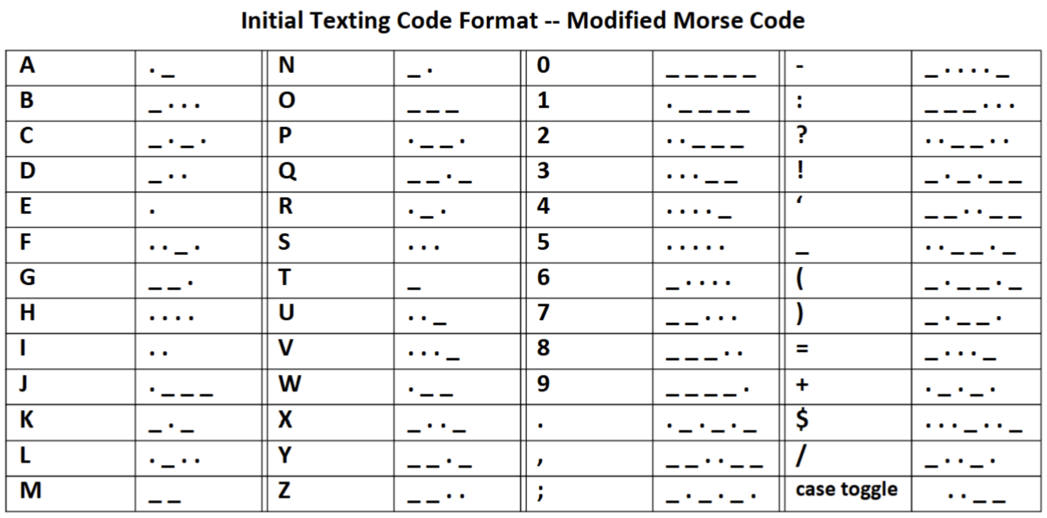
Date of Experiment: February 10, 12, and 17, 2026

Report Deadline: February 19, 2026

Demonstration Deadline: February 17, 2026

1 Introduction

The objective of this laboratory is to apply finite state machine (FSM) design techniques to the development of a Single Button Texter module intended for use as a Human–Machine Interface (HMI) in computing and communication systems. The module is designed to interpret user input generated by toggling a single mechanical switch, effectively decoding an on–off keying scheme based on a modified version of Morse Code. These input patterns are translated into plain text characters corresponding to the ASCII subset of the Unicode standard. The modified Morse Code used in this lab and an example string of Morse Code characters can be found in Figure 1.



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	··__

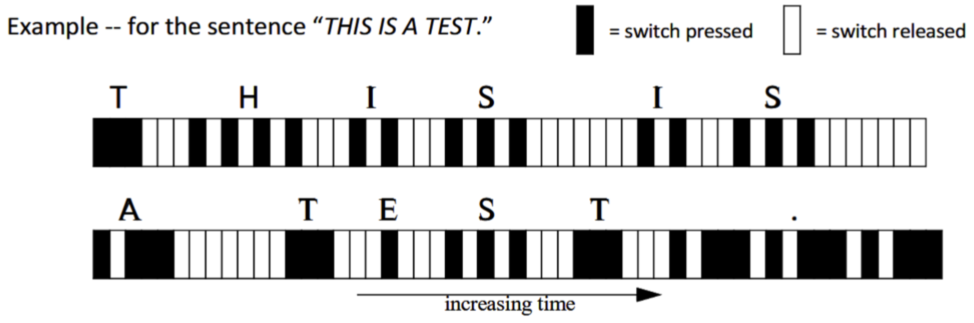


Figure 1: Modified Morse Code Lookup Table and Example

2 Experiment Description

In this experiment, a template project was modified to implement the Morse Code decoding. The template project was complete expect for the texter_control.v file. This Verilog file was completed by adapting the Algorithmic State Machine graph found in Figure 2 into Verilog code using the principles of finite state machines. The full Verliog code found in texter_control.v can be found in Appendix A.

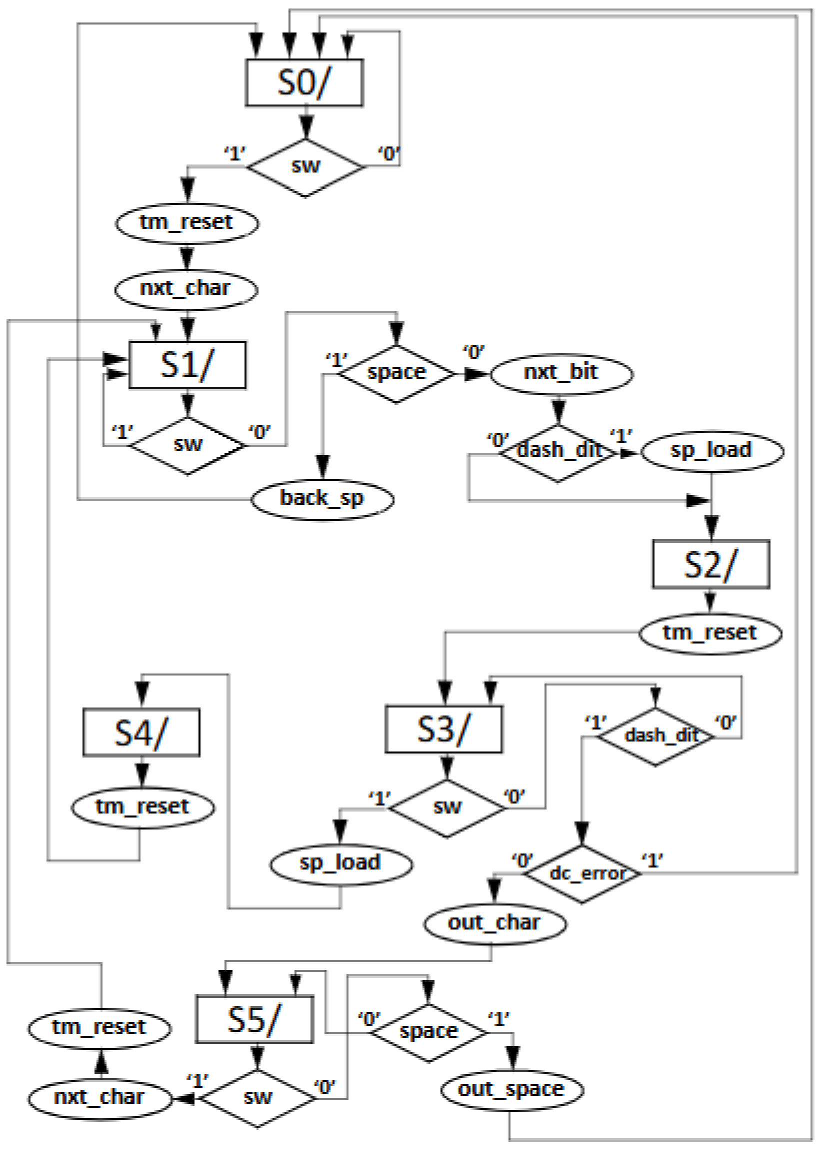


Figure 2: Algorithm State Machine Graph

To ensure the code was implemented in such a way that would not cause unexpected results or interactions with other modules, the top-level design was considered as shown in Figure 3.

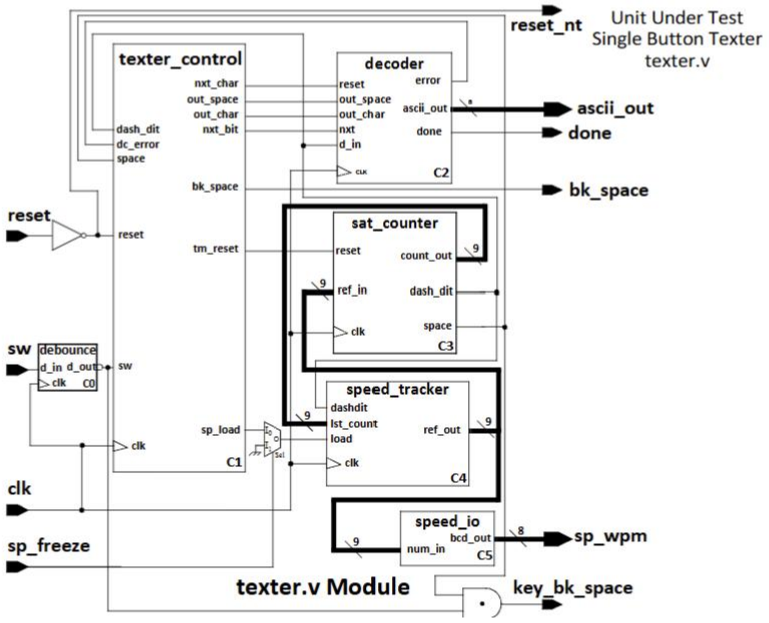


Figure 3: Single Button Texter Top-Level Design

In this top-level design, it is important to note that a portion of the keys and LEDs have been implemented to have certain functions in modules pre-programmed in the template. These functions are shown in Tables 1 and 2.

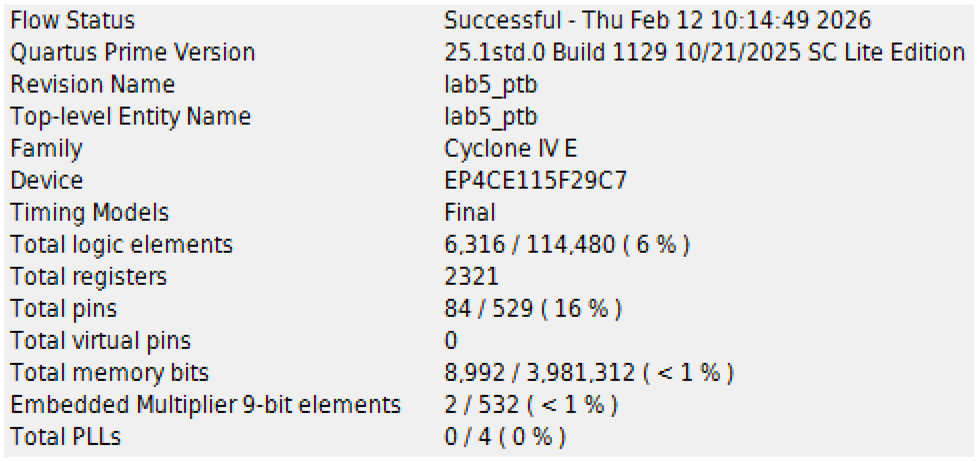
Key	Functionality
KEY0	Decrease the sending speed of text in automated text mode
KEY1	Increase the sending speed of text in automated text mode
KEY2	Toggle mode
KEY3	Display and testbench reset

Table 1: Key Functionality

LED(s)	Functionality
LEDG0	Turn on when SW is pressed
LEDG0-G7	Turn on when SW is held for a long time to indicate backspace
LEDR0-R7	Toggle to indicate relative speed of text in automated text mode
HEX0-1	Indicated approximate texting speed in words per minute
LEC Screen	Indicate decoded text output

Table 2: LED Functionality

After the Verilog code was implemented and incorporated into the overall project, the project was compiled and implemented on the DE2-115 FPGA. The compilation results and FPGA resource allocation can be found in Figure 4.



Flow Status	Successful - Thu Feb 12 10:14:49 2026
Quartus Prime Version	25.1std.0 Build 1129 10/21/2025 SC Lite Edition
Revision Name	lab5_ptb
Top-level Entity Name	lab5_ptb
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	6,316 / 114,480 (6 %)
Total registers	2321
Total pins	84 / 529 (16 %)
Total virtual pins	0
Total memory bits	8,992 / 3,981,312 (< 1 %)
Embedded Multiplier 9-bit elements	2 / 532 (< 1 %)
Total PLLs	0 / 4 (0 %)

Figure 4: FPGA Resource Allocation

3 Results

3.1 Experiment Results

The functionality of the Singel Button Texter was tested in two ways: in automated mode with a pre-programmed stream of inputs and in manual mode.

In automated mode, the pre-programmed stream of bits went through a cycle of inputs. The texter started with lowercase letters a-z, then numbers 0-9, then special characters. After the special characters, the input stream shifted to a pre-programmed message. Partial results of this input stream can be found in Figures 5 and 6.

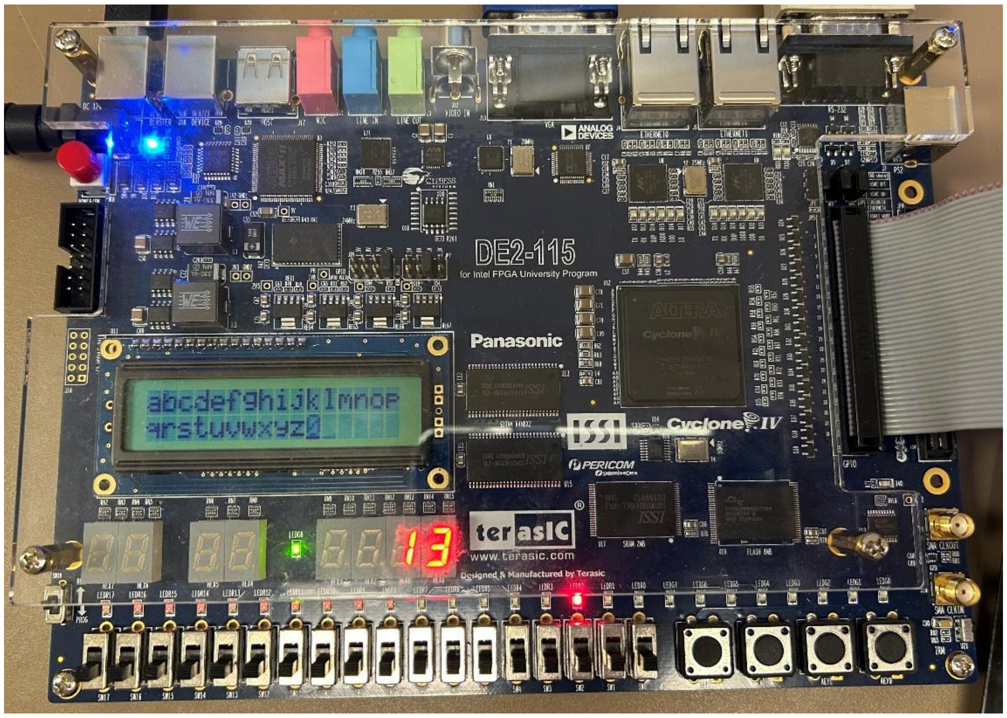
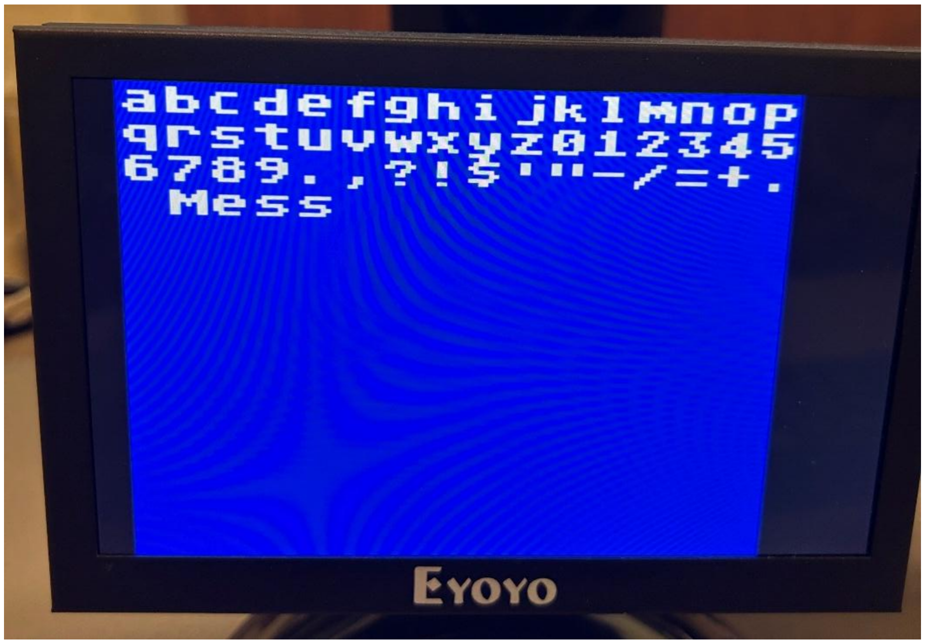


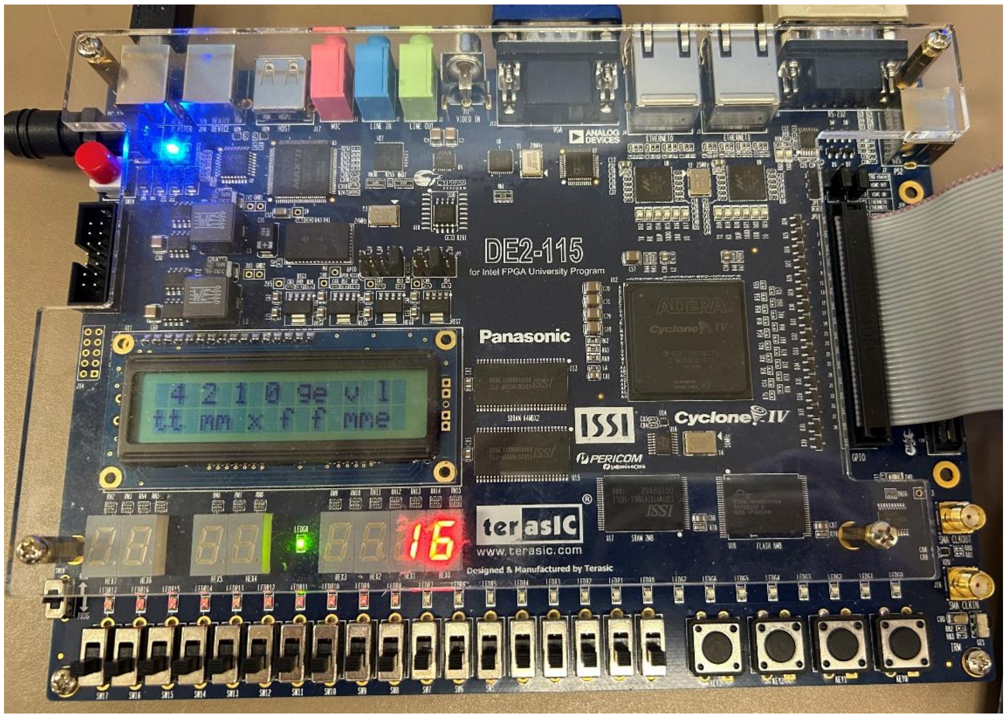
Figure 5: Automated Mode LCD Results



```
abcdefghijklmnop
qrstuvwxyz012345
6789.,?!$'"-/=+.
 Mess
```

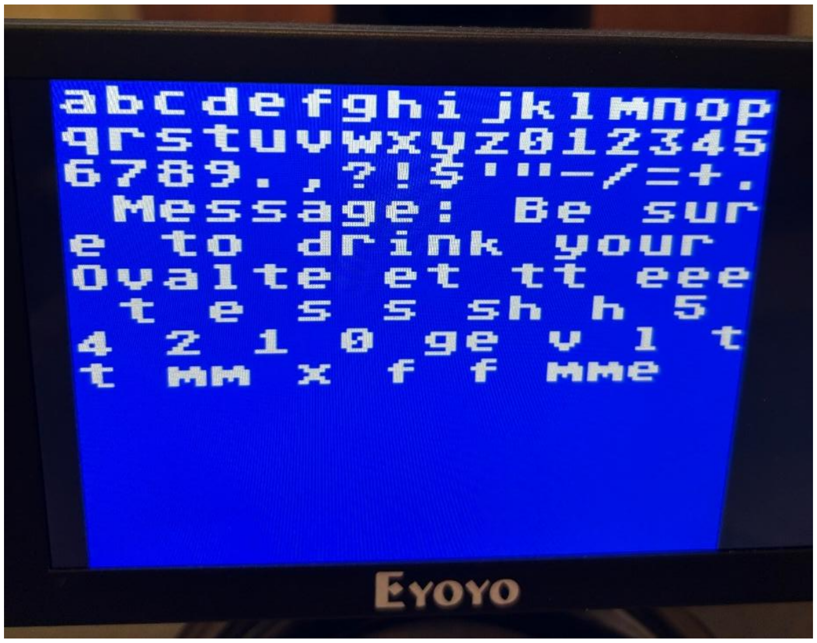
Figure 6: Automated Mode VGA Partial Results

In manual mode, my name Hannah Kelley, was attempted as an input. I could not get the timing correct, however, so I abandoned that attempt and instead tried the numbers 0-5 in reverse order. After getting the timing down for the numbers, I attempted a random set of letters. Figures 7 and 8 shows my attempts at Morse Code after confirming functionality with the automated message.



```
 4 2 1 0 ge v l
tt mm x f f mme
```

Figure 7: Manual Mode LCD Results



```
abcdefghi jklmnop
qrstuvwxyz012345
6789.,?!$'"-/=+.
 Message: Be sur
e to drink your
Ovalte et tt eee
 t e s s sh h 5
4 2 1 0 ge v l t
t mm x f f mme
```

Figure 8: Manual Mode VGA Results

3.2 Post-Lab Questions

After the lab concluded, the following questions were considered:

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?**

   Synchronous sequential digital design is characterized by the use of a common clock signal shared by all storage elements in the system. This global clock synchronizes state changes, allowing the circuit to operate in a predictable and reliable manner. Key characteristics of synchronous design include the use of sequential logic elements (such as flip-flops), well-defined timing analysis, modular design practices, and the frequent use of finite state machines to model system behavior.

   The control path is responsible for directing the operation of the system by generating control signals that determine how data moves and which operations are performed. In contrast, the data path contains the hardware that processes data, including arithmetic, logical, and storage components.

   Clocking requirements specify the conditions under which the system operates correctly, most notably the clock frequency. The clock period must be long enough to accommodate worst-case propagation delays and setup times of sequential elements. Reliable operation also depends on minimizing clock skew and variations in clock arrival times across the circuit.
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 function correctly.

Warning (10030): *Net "clear_seq.data_a" at usend.v(21) has no driver or initial value, using a default initial value '0'.* This warning appeared multiple times for different signals across various files. It indicates that a signal is not explicitly driven and therefore defaults to a logic value of 0. In this design, the default value was sufficient and did not affect the intended operation of the circuit. Since the system functioned as expected during simulation and synthesis, this warning was safely ignored.

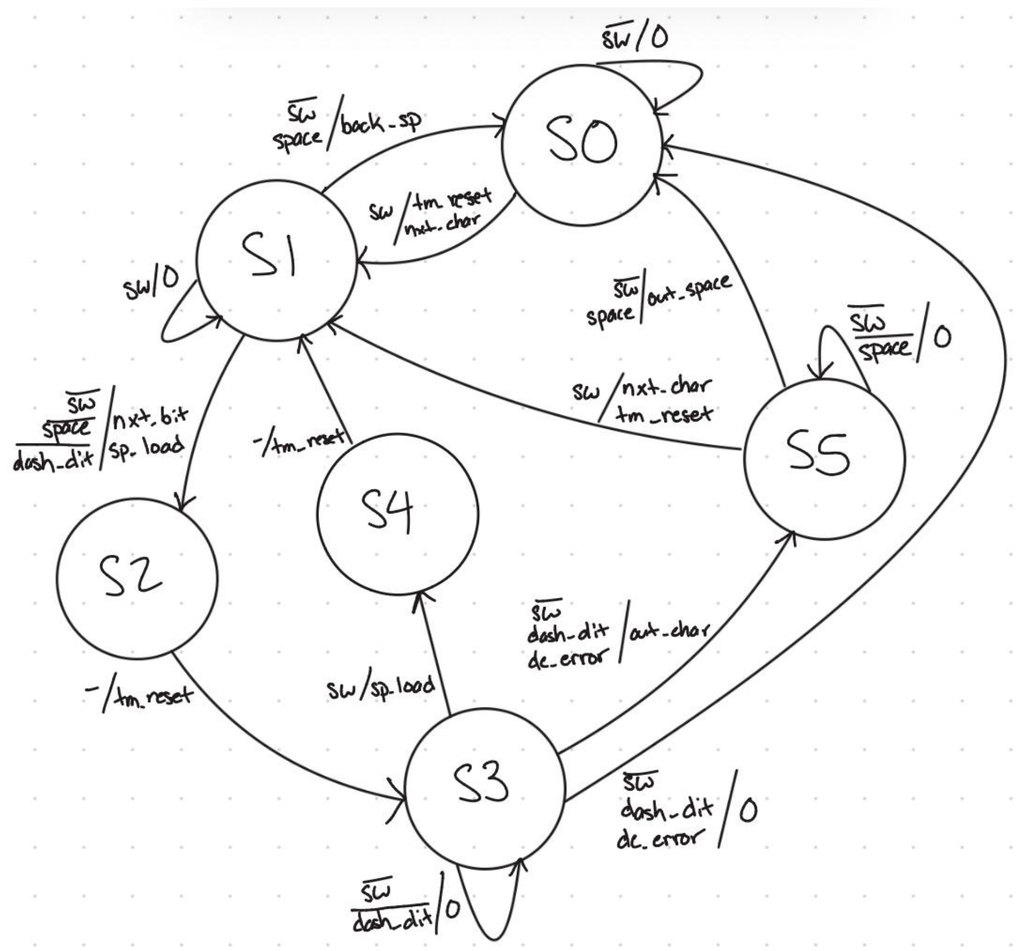
Warning (10236): *Number of processors has not been specified which may cause overloading on shared machines.* This warning relates only to compilation performance and not to the synthesized hardware. Because parallel compilation was not required for this design and functionality is unaffected by the number of processors used during compilation, this warning was ignored.

Warning (10230): *Verilog HDL assignment warning at usend.v(33): truncated value with size 32 to match size of target (13).* This warning indicates that a larger value was truncated to fit into a smaller signal width, which could potentially cause data loss. In this case, the truncated bits were all zeros, meaning no meaningful information was lost. Therefore, the behavior of the design remained unchanged, and the warning could be ignored.

Warning (292013): *Feature LogicLock is only available with a valid subscription license.* This warning indicates that a licensed feature was referenced by the software but not enabled. Since the LogicLock feature was not required or used in the design, this warning had no impact on functionality and was ignored.

Warning (15714): *Some pins have incomplete I/O assignments. Refer to the I/O Assignment Warnings report for details.* This warning suggests that not all I/O pins were explicitly assigned. In this case, the I/O assignments were provided as part of the lab template and were not modified. Because the design synthesized and functioned correctly, the warning did not indicate a functional issue and was ignored..

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.**

   S0 is the initial state of the FSM. In this state, the system waits for the button to be pressed while maintaining all timing counters in a reset or inactive condition. No input is being evaluated in this state.

   S1 is entered after button activity is detected. This state evaluates the measured timing information to determine whether the input corresponds to a dot, dash, space, or an invalid character. The decision logic in this state relies on counter thresholds provided by the timing IP core elements.

   S2 is used to store the decoded dot or dash into the current symbol register. This state updates the internal representation of the character being constructed before transitioning to a spacing or continuation state.

S3 measures the duration of the gap following a button release. Short gaps indicate that additional symbols belonging to the same character are expected, while longer gaps signify the end of a character.

S4 performs the translation of the accumulated dot/dash sequence into its corresponding ASCII character. Once decoding is complete, the character is made available to the output interface.

S5 handles the finalization of the character output and resets internal registers and counters in preparation for the next character entry, after which the FSM returns to the idle state.

While reduction in the number of states used in this FSM could be done and preserve functionality, significant changes would need to be made. Consolidating states 2 and 4 and states 1 and 5 could be possible, though it would take significant time and effort to ensure no errors were made.

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?**
   A traditional simulation testbench operates entirely in software, applying precisely controlled inputs while allowing full visibility of internal signals. In contrast, the physical testbench used in this laboratory evaluates the design in real hardware, using actual input devices and system clocks to observe real-time behavior. The primary advantage of a physical testbench is that it reveals non-ideal effects such as switch bounce, timing variations, and hardware delays that are often not captured in simulation. This leads to more realistic validation of the design, particularly for human–machine interfaces. However, physical testing is more difficult to debug and less repeatable, since internal signals are harder to observe and inputs may vary between tests. Simulation-only testbenches offer easier debugging, repeatability, and exhaustive testing, but may miss hardware-specific issues. Developing a physical testbench presents challenges including additional hardware integration, signal synchronization, and the complexity of automating tests in a real-world environment.
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.

Choosing between hardware and software implementations involves balancing performance, flexibility, and complexity. Hardware designs offer high processing speed, deterministic real-time behavior, and efficient energy use for fixed tasks, but they are more difficult to modify and debug. Software implementations on microcontrollers are easier to develop and update, and are more flexible as requirements change, but they may suffer from lower performance, non-deterministic timing, and higher power usage for time-critical functions. As a result, hardware is typically favored for deterministic, high-speed tasks, while software is better suited for adaptable and evolving system functionality.

4 Conclusion

The Single Button Texter module was successfully designed, implemented, and tested using finite state machine techniques. The system reliably interpreted timed button inputs and converted them into valid ASCII characters while maintaining synchronous operation on the FPGA platform. Testing within the physical test bench environment confirmed correct decoding behavior across a range of input patterns, including boundary and error conditions. This laboratory reinforced the importance of careful state design, timing measurement, and systematic verification in the development of reliable digital human–machine interfaces.

Appendix A – Full Verilog Code for texter_control.v

```
// Single Button Texter -- texter control module
// -- texter_control.v file
// (c) 2/5/2026 B. Earl Wells, University of Alabama in Huntsville
// all rights reserved -- for academic use only.
//
// This is the main texter_control module for the single button texter.
// This module should implement the design that was described by the
// ASM Chart that was presented in the CPE 322 class.
// The design is to be reset to state S0 whenever the reset signal is
// at a logic high. Whenever the reset signal is at a logic low it
// should fully implement the state machine specified by the SM chart.
// The clock signal is assumed to be a 50% duty cycle clock that is the
// same clock that drives the other functional units that
// are being controlled by this module (50 Mhz on the DE2-115). The other
// input and output signals are all assumed to be active high. The
// input signal, sw, is controlled by the user and the other inputs
// are status inputs that come directly from the functional units in the
// data path that are being controlled. These inputs will change their states
// in direct response to your controlling outputs that you provide. The
// outputs are all assumed to be active for a one clock duration as
// indicated by the ASM chart.

module texter_control(input clk, reset, sw, space, dash_dit, dc_error,
   output reg nxt_bit, nxt_char, out_char, out_space, tm_reset,
   sp_load, back_sp);

  reg [2:0] state;
  reg [2:0] next_state;

  localparam S0 = 3'd0, S1 = 3'd1, S2 = 3'd2, S3 = 3'd3, S4 = 3'd4, S5 = 3'd5;

  // all 6 states case statement (S0 - S5)
  always @(state, reset, sw, space, dash_dit, dc_error) begin
            nxt_bit = 0;
            nxt_char = 0;
            out_char = 0;
            out_space = 0;
            tm_reset = 0;
            sp_load = 0;
            back_sp = 0;

            next_state = state;

      case (state)
        0: begin
```

```
    if (sw) begin
      tm_reset  = 1;
      nxt_char  = 1;
      next_state = 1;
    end
  end

  1: begin
    if (sw) begin
      next_state = 1;
    end else begin
      if (space) begin
        back_sp   = 1;
        next_state = 0;
      end else begin
        nxt_bit = 1;
        if (dash_dit) begin
        sp_load = 1;
      end
        next_state = 2;
      end
    end
  end

  2: begin
    tm_reset = 1;
    next_state = 3;
  end

  3: begin
    if (sw) begin
      sp_load   = 1;
      next_state = 4;
    end else begin
      if (!dash_dit) begin
        next_state = 3;
      end else begin
        if (dc_error) begin
          next_state = 0;
        end else begin
          out_char = 1;
          next_state = 5;
        end
      end
    end
  end
```

```
      4: begin
        tm_reset   = 1;
        next_state = 1;
      end

      5: begin
        if (sw) begin
          nxt_char  = 1;
          tm_reset  = 1;
          next_state = 1;
        end else if (space) begin
          out_space = 1;
          next_state = 0;
        end else begin
          next_state = 5;
        end
      end
    endcase
  end

 always @(posedge clk) begin
  if (reset) begin
			state <= 0;
		end
		else begin
			state <= next_state;
		end
 end

endmodule
```