

ECE667

Lab Report #1

Lab Completed: 6/28/11

Due Date: 7/5/11

By:

Larissa Borcz

Sean Ewing

Feiran Lei

Ji Tu

Table of Contents**Page #**

1) Abstract	2
2) Introduction	2
3) Discussion	2
a. Method 1	3
b. Method 2	3
4) Implementation & Testing	3
5) Conclusion	4
6) Appendices	5

Abstract

This project aims to develop a 4-bit UP/DOWN Counter design, in order to fulfill the design requirements specified in Figure 1 & Figure 2, and stated as:

- 1) D[3..0] are the data input,
- 2) Q[3..0] are the data output,
- 3) Counter works synchronically: at the rising edge of CLK, $Q[3..0] = D[3..0]$, if LD = 1; otherwise, Q[3..0] increases by 1 when U/D = 1 and decreases by 1 when U/D = 0.

The design will be created using two methods:

- 1) The traditional way: using truth table, K-map, variable expression and then implemented with D Flip-Flops.
- 2) Using VHDL/Verilog HDL language.

The goal of this laboratory is to introduce the Altera's Quartus II design environment. Designs created for Homework #1 for the 4-bit counter will be entered using the Graphic Editor and the Text Design Tool included in the software package. Each design will be compiled using the compiler, and then simulated using the Waveform Editor.

Introduction

The purpose of this lab is to develop a 4-bit UP/DOWN Counter Design with a VHDL/Verilog program (Altera). Students are to use schematics developed in a graphics editor and the second portion of the lab is to compile, test and repeat (if necessary) code for the same 4-bit counter. Students are to determine the best schematic from their homework and implement the homework into the graphics editor. The code will also come from homework.

Discussion

In this lab we implemented a 4-bit up/down counter. The inputs to the counter are D[3..0], CLK, LD, U/D; and the outputs are Q[3..0]. LD acts as a load: if LD is asserted (LD=1), the inputs D[3..0] are assigned to the outputs Q[3..0]; if LD is not asserted (LD=0), the counter either counts up when U/D=1 or counts down when U/D=0. The counter we designed is

synchronous and is triggered on the positive edge of the CLK signal. The flowchart showing the 4-bit counter's logical operation is shown in Figure 2.

- *Design Method 1: traditional circuit*

Truth table, shown in Figure 3, and K-maps, shown in Figure 4, were used to generate proper logic functions, shown in Figure 5, and then implemented by D-flip flops. The 4-bit up/down sequential counter was designed with 4 D-flip flop to hold the 4 states of the counter. We first started off by writing down the truth table, and then the logic expressions were simplified. From the equations, we can use D-flip flops, XOR, XOR, AND and NOT gates to implement the circuit. QUARTUS II software was used to implement and test of the designing circuit. Simulation results of the circuit and the output waveforms are presented in the appendix. The result obtained matched our laboratory1 specifications. The counter counts up when U/D is 1 and counts down when U/D=0.

- *Design method 2: Using VHDL*

VHDL code was used to generate/simulate the required counter in the lab specification. The codes are shown in Figure 8. The simulation results for the schematic circuit and VHDL code are shown in Figure 9 and Figure 10. The simulation were done in two sections: the first one denotes that U/D is not asserted and therefore the counter should be counting down (0-3.1ns for schematic circuit, and 0-1.8ns for VHDL code); the second section denotes that U/D is asserted, so the counter should be counting up. Also, we confirm that the LD function work well (Q[3..0] should be the same as D[3..0] on the next clock cycle after the LD is asserted).

Implementation & Testing

The design for both the circuit and VHDL, shown in Figure 6, was done entirely with software. The only hardware we used was the PC we used to make the designs. The schematic circuit worked as intended once it was completed, shown in Figure 7. However, there are several procedures to be notice before giving pulses to the input signal for the VHDL simulation. As assigned a different name of module with the name of project, the simulation report cannot go through and shows the errors. After several trials, we eventually found the problem and change

the name of the module and set the VHDL file as the TOP-FILE as well as deleting the schematic file in the library. Then the simulation works and ModelSIM window jumped out.

The simulation for VHDL gave a different diagram at first since the outputs (Q0-Q3) are defined reversely as in the schematic one. Then, after a few modifications to the programming, seen in Figure 8, both of the diagrams have an identical response to the same input signal, shown in Figure 9 & Figure 10.

Conclusion

Overall, due to the hard work by team members on the homework assignment, the actual laboratory implantation went very smoothly. Both the schematic and code in the lab were tested vigorously and proved to be without fault. The most difficult portion of the lab was populating the gates for the schematic. Due to the perceived complexity of the circuit, many, many gates were used in the circuit. Another problem the lab posed was the signal redundancy in the circuit. Often a single signal was routed to multiple gates.

Appendices

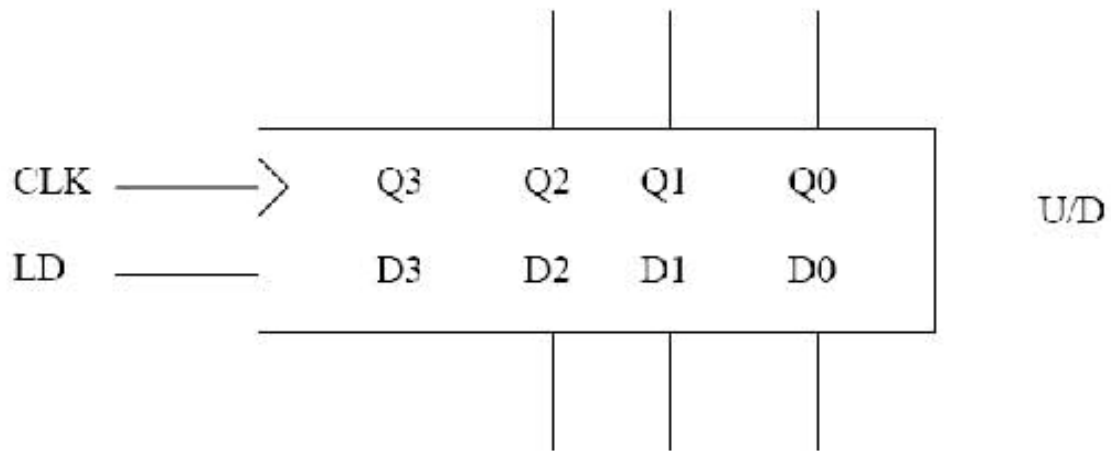


Figure 1: Block Diagram of 4-bit Up/Down Counter

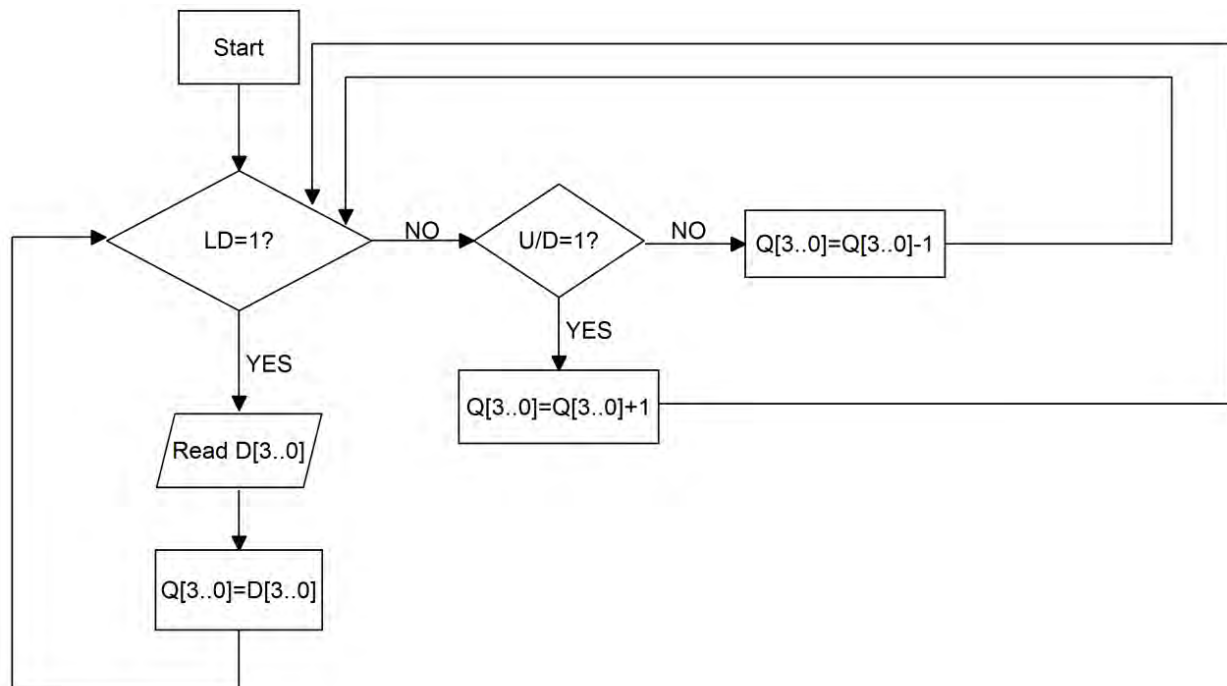


Figure 2: Flowchart of 4-bit Counter Logic

		Present State				Next State			
LD	U/D	q3	q2	q1	q0	Q3	Q2	Q1	Q0
1	X	X	X	X	X	q3	q2	q1	q0
0	0	0	0	0	0	1	1	1	1
0	0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0	1
0	0	0	0	1	1	0	0	1	0
0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	1	0	1	0	0
0	0	0	1	1	0	0	1	0	1
0	0	0	1	1	1	0	1	1	0
0	0	1	0	0	0	0	1	1	1
0	0	1	0	0	1	1	0	0	0
0	0	1	0	1	0	1	0	0	1
0	0	1	0	1	1	1	0	1	0
0	0	1	1	0	0	1	0	1	1
0	0	1	1	0	1	1	1	0	0
0	0	1	1	1	0	1	1	1	0
0	1	0	0	0	0	0	0	0	1
0	1	0	0	0	1	0	0	1	0
0	1	0	0	1	0	0	0	1	1
0	1	0	0	1	1	0	1	0	0
0	1	0	1	0	0	0	1	0	1
0	1	0	1	0	1	0	1	1	0
0	1	0	1	1	0	0	1	1	1
0	1	0	1	1	1	1	0	0	0
0	1	1	0	0	0	1	0	0	1
0	1	1	0	0	1	1	0	1	0
0	1	1	0	1	0	1	0	1	1
0	1	1	0	1	1	1	1	0	0
0	1	1	1	0	0	1	1	0	1
0	1	1	1	0	1	1	1	1	0
0	1	1	1	1	0	1	1	1	1
0	1	1	1	1	1	0	0	0	0

Figure 3: Truth table of 4-bit
Up/Down Counter

Q3

		q2, q1, q0							
		000	001	011	010	100	101	111	110
U/D,	00	1	0	0	0	0	0	0	0
q3	01	0	1	1	1	1	1	1	1
	11	1	1	1	1	1	1	0	1
	10	0	0	0	0	0	0	1	0

$$Q3 = \underline{U/D}q3q2 + \underline{U/D}q3q2 + \underline{U/D}q3q1 + \underline{U/D}q3q0 + q3q1q0 + \underline{U/D}q3q2q1q0 + \underline{U/D}q3q1q2q0$$

Q2

		q2, q1, q0							
		000	001	011	010	100	101	111	110
U/D,	00	1	0	0	0	0	1	1	1
q3	01	1	0	0	0	0	1	1	1
	11	0	0	1	0	1	1	0	1
	10	0	0	1	0	1	1	0	1

$$Q2 = \underline{U/D}q2q1q0 + \underline{U/D}q2q1q0 + \underline{U/D}q2q1 + \underline{U/D}q2q0 + q2q1q0$$

Q1

		q2, q1, q0							
		000	001	011	010	100	101	111	110
U/D,	00	1	0	1	0	1	0	1	0
q3	01	1	0	1	0	1	0	1	0
	11	0	1	0	1	0	1	0	1
	10	0	1	0	1	0	1	0	1

$$Q1 = \underline{U/D}q1q0 + \underline{U/D}q1q0 + \underline{U/D}q1q0 + \underline{U/D}q1q0$$

Q0

		q2, q1, q0							
		000	001	011	010	100	101	111	110
U/D,	00	1	0	0	1	1	0	0	1
q3	01	1	0	0	1	1	0	0	1
	11	1	0	0	1	1	0	0	1
	10	1	0	0	1	1	0	0	1

$$Q0 = q0$$

Figure 4: K-maps of 4-bit
Up/Down Counter

$$\begin{aligned}
 Q_0^+ &= LD \cdot D_0 + \overline{LD} \cdot \overline{Q_0} \\
 Q_1^+ &= LD \cdot D_1 + \overline{LD} \cdot \{UD \cdot [(Q_1 \oplus Q_0) + \overline{UD} \cdot [(Q_1 \oplus Q_0)]] \\
 Q_2^+ &= LD \cdot D_2 + \overline{LD} \cdot \{UD \cdot [(Q_1 \cdot Q_0) \oplus Q_2] + \overline{UD} \cdot [(\overline{Q_1} \cdot \overline{Q_0}) \oplus Q_2]\} \\
 Q_3^+ &= LD \cdot D_3 + \overline{LD} \cdot \{UD \cdot [(Q_2 \cdot Q_1 \cdot Q_0) \oplus Q_3] + \overline{UD} \cdot [(\overline{Q_2} \cdot \overline{Q_1} \cdot \overline{Q_0}) \oplus Q_3]\}
 \end{aligned}$$

Figure 5: State equations derived from the K-map

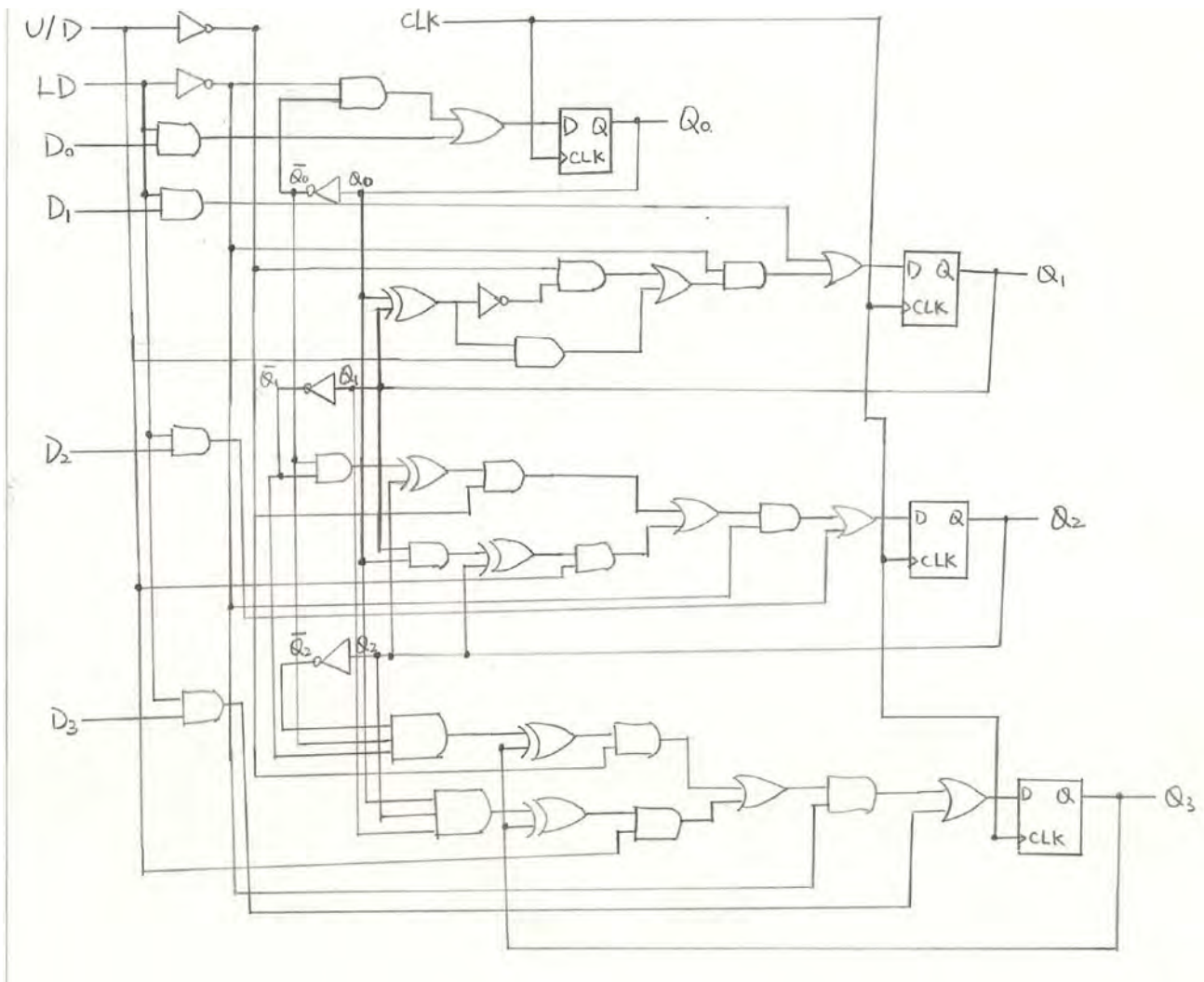


Figure 6: Rough-Draft Design

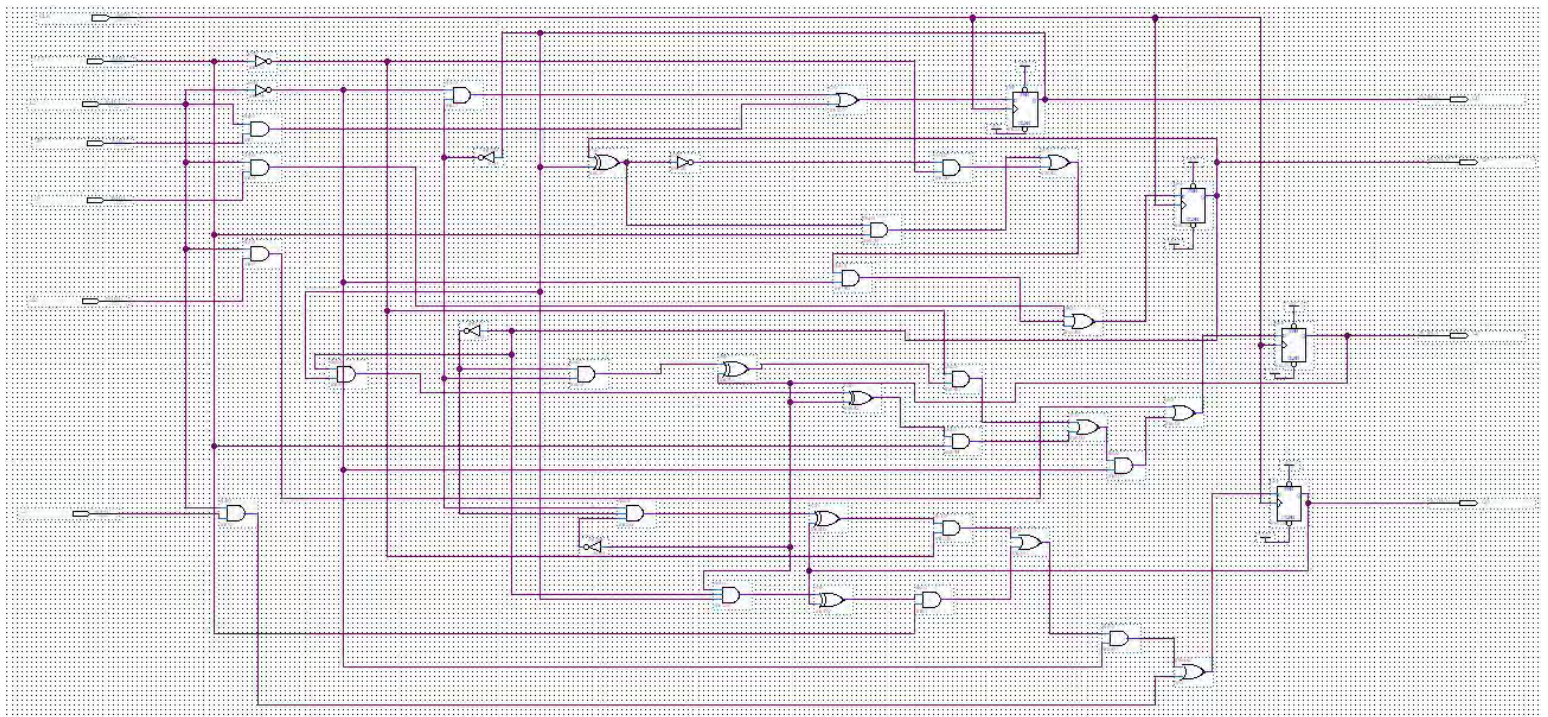


Figure 7: Altera's Quartus II Circuit Design

```

1  -- Design Name: ECE677 Homework #1
2  -- Module Name: Lab 1 - Behavioral
3  -----
4  library IEEE;
5  use IEEE.STD_LOGIC_1164.ALL;
6  use IEEE.STD_LOGIC_ARITH.ALL;
7  use IEEE.STD_LOGIC_UNSIGNED.ALL;
8
9  entity LAB1 is
10     port( D: in std_logic_vector(3 downto 0);
11           CLK: in std_logic;
12           LD: in std_logic;
13           UD: in std_logic;
14           Q: out std_logic_vector(3 downto 0) );
15 end LAB1;
16
17 architecture Behavioral of LAB1 is
18     signal temp: std_logic_vector(3 downto 0);
19 begin
20     process(CLK)
21     begin
22         if (CLK'event and CLK='1') then
23             if LD='1' then
24                 temp <= D;
25             elsif (LD='0' and UD='1') then
26                 temp <= temp + 1;
27             elsif (LD='0' and UD='0') then
28                 temp <= temp - 1;
29             end if;
30         end if;
31     end process;
32     Q <= temp;
33 end Behavioral;

```

Figure 8: 4-bit Up/Down Counter VHDL Code

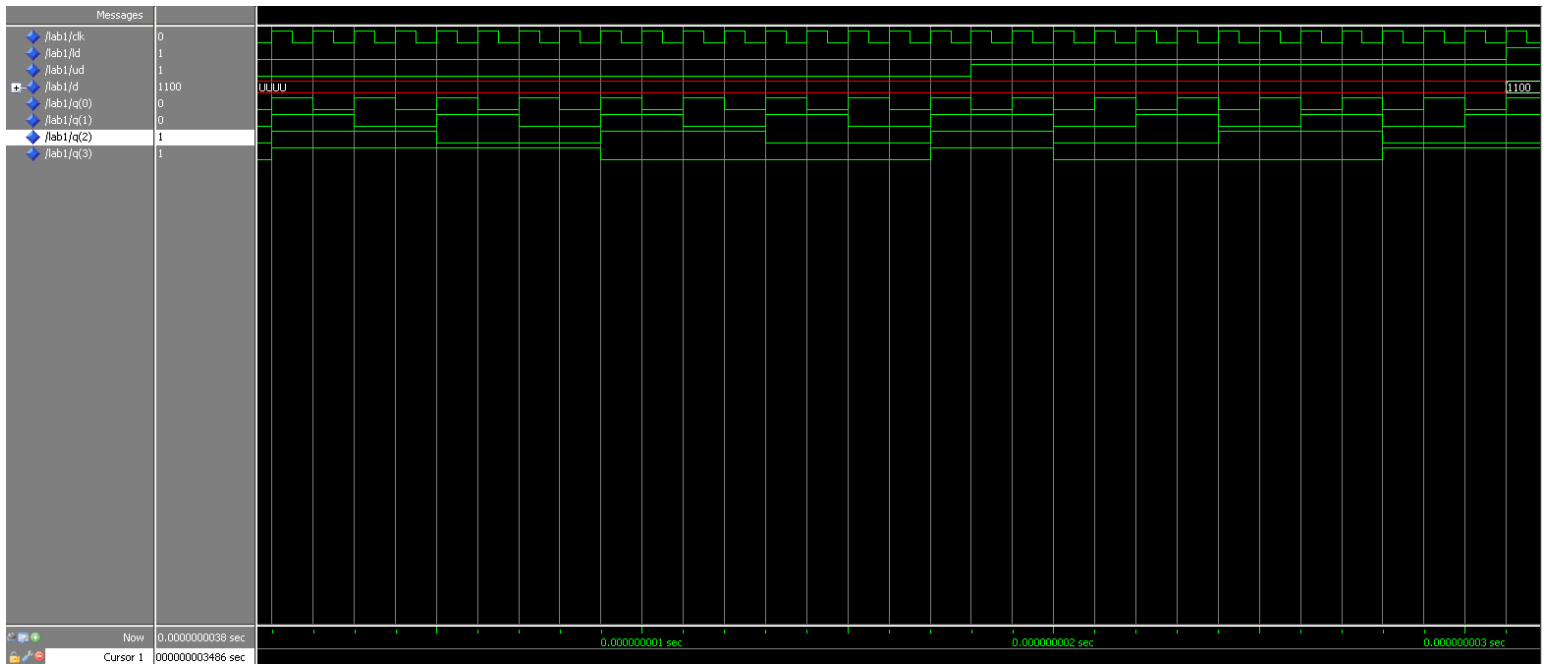


Figure 9: Altera's Quartus II – Method 1 Output

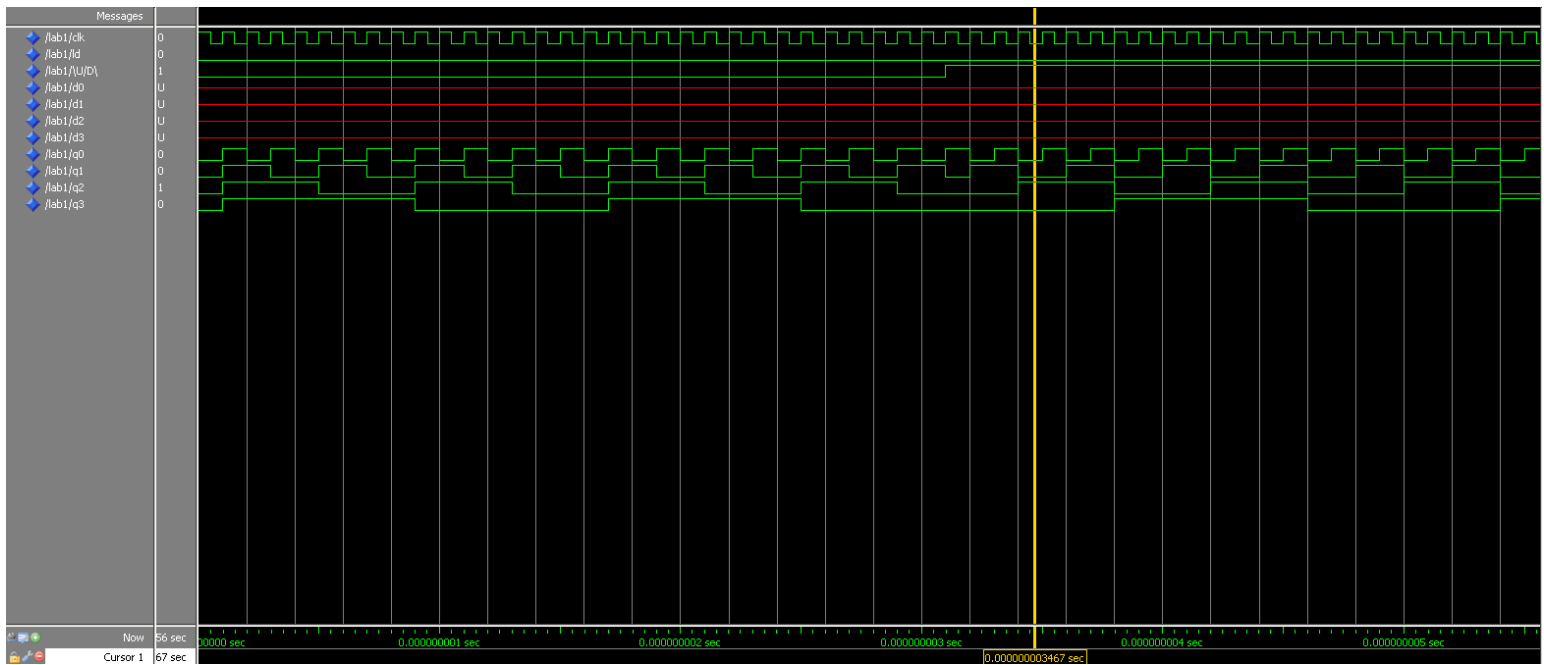
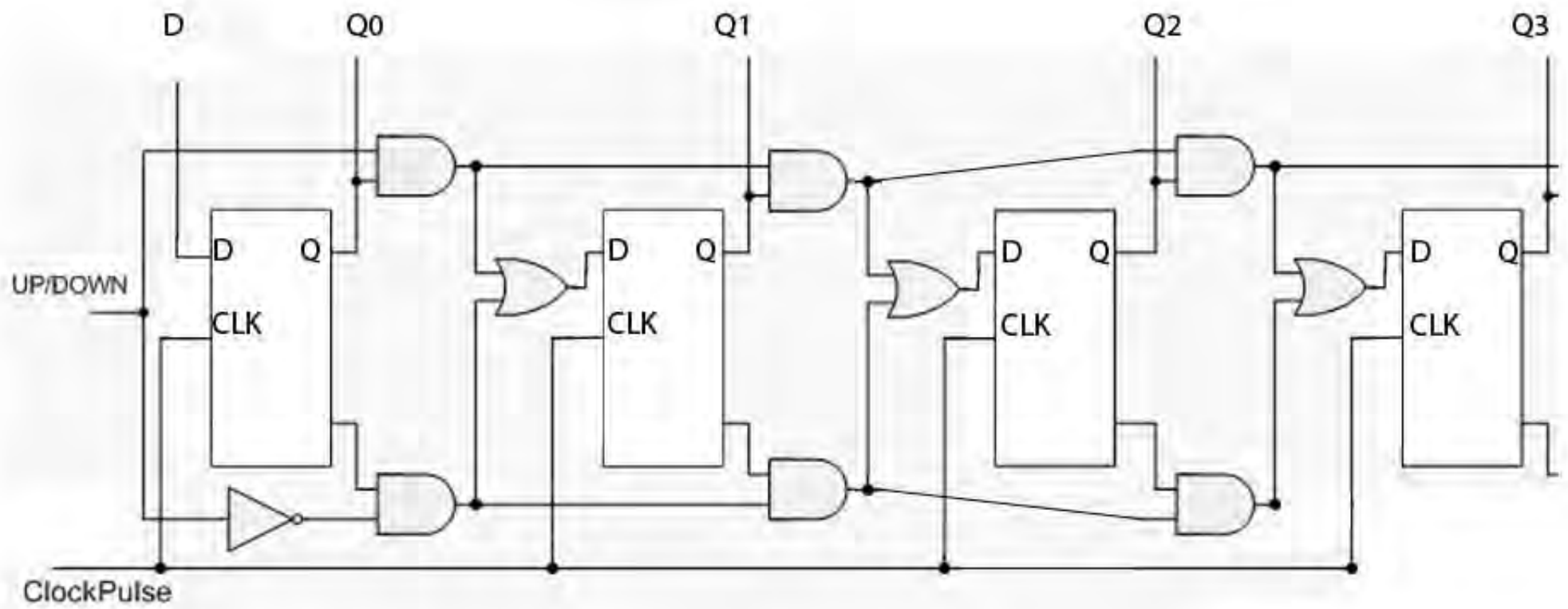
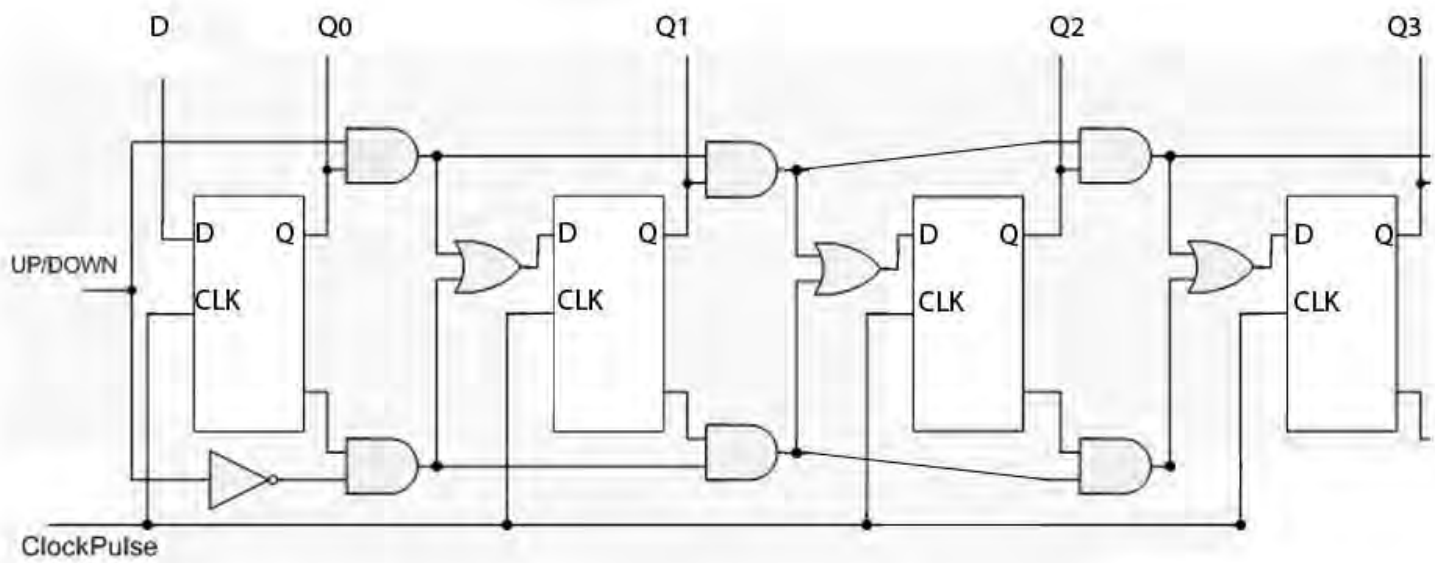
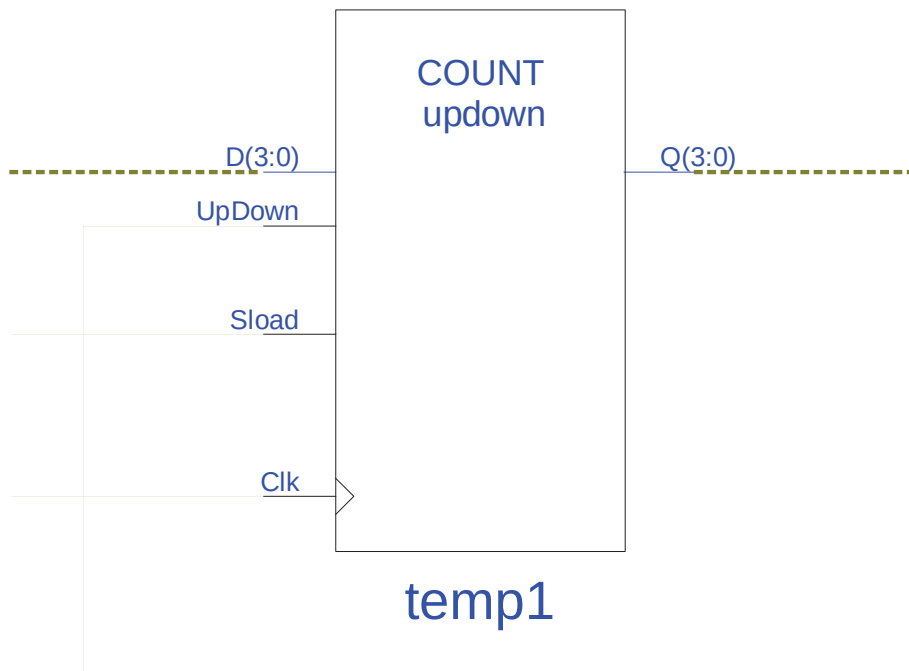


Figure 10: Altera's Quartus II – Method 2 Output

Present State						Next State				Out			
D3	D2	D1	D0	LD	U/D	D3	D2	D1	D0	Q3	Q2	Q1	Q0
0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	0	0	0	0	1	0	0	0	1	0	0	0	1
0	0	0	0	1	X	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0	1	0	0	0	1	0
0	0	0	1	1	X	0	0	0	1	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0	0	0	1
0	0	1	0	0	1	0	0	1	1	0	0	1	1
0	0	1	0	1	X	0	0	1	0	0	0	1	0
0	0	1	1	0	0	0	0	1	0	0	0	1	0
0	0	1	1	0	1	0	1	0	0	0	1	0	0
0	0	1	1	1	X	0	0	1	1	0	0	1	1
0	1	0	0	0	0	0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	X	0	1	0	0	0	1	0	0
0	1	0	1	0	0	0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	1	1	0	0	1	1	0
0	1	0	1	1	X	0	1	0	1	0	1	0	1
0	1	1	0	0	0	0	1	0	1	0	1	0	1
0	1	1	0	0	1	0	1	1	1	0	1	1	1
0	1	1	0	1	X	0	1	1	0	0	1	1	0
0	1	1	1	0	0	0	1	1	0	0	1	1	0
0	1	1	1	0	1	1	0	0	0	1	0	0	0
0	1	1	1	1	X	0	1	1	1	0	1	1	1
1	0	0	0	0	0	0	1	1	1	0	1	1	1
1	0	0	0	0	1	1	0	0	1	1	0	0	1
1	0	0	0	1	X	1	0	0	0	1	0	0	0
1	0	0	1	0	0	1	0	0	0	1	0	0	0
1	0	0	1	0	1	1	0	1	0	1	0	1	0
1	0	0	1	1	X	1	0	0	1	1	0	0	1
1	0	1	0	0	0	1	0	0	1	1	0	0	1
1	0	1	0	0	1	1	0	1	1	1	0	1	1
1	0	1	0	1	X	1	0	1	0	1	0	1	0
1	0	1	1	0	0	1	0	1	0	1	0	1	0
1	0	1	1	0	1	1	1	0	0	1	1	0	0
1	0	1	1	1	X	1	0	1	1	1	0	1	1
1	1	0	0	0	0	1	0	1	1	1	0	1	1
1	1	0	0	0	1	1	1	0	1	1	1	0	1
1	1	0	0	1	X	1	1	0	0	1	1	0	0
1	1	0	1	0	0	1	1	0	0	1	1	0	0
1	1	0	1	0	1	1	1	1	0	1	1	1	0
1	1	0	1	1	X	1	1	0	1	1	1	0	1
1	1	1	0	0	0	1	1	0	1	1	1	0	1
1	1	1	0	0	1	1	1	1	1	1	1	1	1
1	1	1	0	1	X	1	1	1	0	1	1	1	0
1	1	1	1	0	0	1	1	1	0	1	1	1	0
1	1	1	1	0	1	0	0	0	0	0	0	0	0
1	1	1	1	1	X	1	1	1	1	1	1	1	1



COUNTER:1



ECE667

Lab Report #2

Lab Completed: 7/5/11

Due Date: 7/19/11

By:

Larissa Borcz

Sean Ewing

Feiran Lei

Ji Tu

<u>Table of Contents</u>	<u>Page #</u>
1) Abstract	2
2) Introduction	2
3) Discussion	2
a. Column Scanner	2
b. Row Encoder	3
c. Output Data Latch	4
4) Implementation & Testing	5
5) Conclusion	5
6) Appendices	6

Abstract

This experiment focuses on the development and implementation of the first three components of the Keypad Scanner System: 1) Column Scanner, 2) Row Encoder, and 3) Output Data Latch. On activation of the “Enable” terminal the column scanner will sequentially ground one of the four outputs C[3..0], and the corresponding count value will be placed in the two output lines COL[1..0]. When the row encoder detects logic 0 at the inputs (ROW[3..0]), the ZDET output is asserted and the encoded value of the row is placed in the output ROW[1..0]. With the Output Data Latch, when the load signal is asserted, the encoded values of ROW and COL have to be latched. During lab we entered our designs from Homework #2 for the column scanner, row encoder, and output data-latch using Quartus II. We then compiled our designs using the Compiler, and then simulated our designs using the Waveform Editor. We wired up the Altera CPLD, with the purpose to get the CPLD to interface with the ByteBlaster programmer connected to the PC at the lab station. Once this interface was established and the device was configured properly, we found that implementing and changing our designs can be very quick and easy.

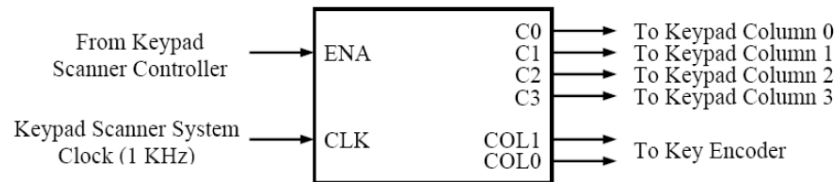
Introduction

The purpose of this lab is to be entering your designs from Homework #2 for the column scanner, row encoder, and output data-latch using Quartus II. Students are to compile, and then simulate your designs using the Waveform Editor. Finally, students are get the CPLD to interface with the ByteBlaster programmer connected to the PC at your lab station. The code will also come from homework.

Discussion

1. Column Scanner

For the first part of this lab, we entered our design for the column scanner as a text description file, using Verilog HDL code. The following block diagram illustrates the required inputs and outputs of the column scanner:



Column scanner block diagram

The column scanner was designed according to the following truth table that was provided in the assignment handout. The following table illustrates the specified requirements:

COLUMN SCANNED	C3	C2	C1	C0	COL(1:0)
0	1	1	1	0	00
1	1	1	0	1	01
2	1	0	1	1	10
3	0	1	1	1	11

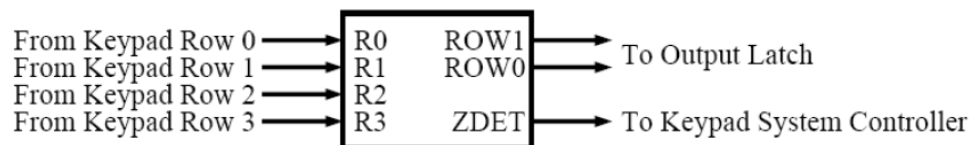
Column scanner truth table

The column scanner is used to ground each column of the keypad at a time in order to allow a key press to be detected at the row outputs. When a key press is detected, the ENA signal to the column scanner will be negated so that the column that contained the pressed key remains grounded. The COL(1:0) output is nothing but a decoded output illustrating which column is currently being scanned. The circuit is quite simple and was easily implemented using Verilog code, seen in Figure #1 of the Appendix.

2. Row Encoder

The next step of this lab was to create the row scanner that was specified in the homework description. The following block diagram illustrates the required input and outputs of the row scanner:

Row Encoder:



Row encoder block diagram

The row detector was designed according to the following truth table that was provided in the assignment handout. The following table illustrates the specified requirements:

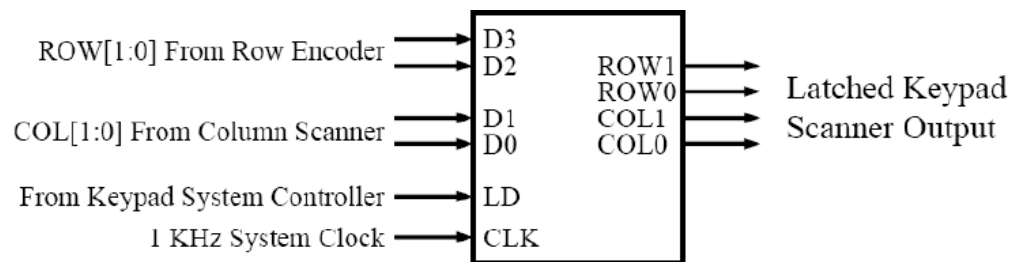
R3 (I)	R2 (I)	R1 (I)	R0 (I)	ROW1 (O)	ROW0 (O)	Encoded Row
1	1	1	0	0	0	0
1	1	0	1	0	1	1
1	0	1	1	1	0	2
0	1	1	1	1	1	3

Row encoder truth table

The row scanner is used to detect when a key has been pressed on the keypad. The row decoder is merely a combinatorial logic circuit that monitors the row outputs on the keypad, and generates an output signal (ZDET) when a zero is detected on a single row. The row detector also places a decoded output of the row number on the ROW(1:0) lines. The output of the COL(1:0) output of the column scanner and the ROW(1:0) outputs of the row decoder along with ZDET are to be used to determine when a key is pressed, and to enable the keypad controller circuit to determine which key was pressed. The logic was implemented using Verilog code, seen in Figure #1 of the Appendix.

3. Output data latch

The final design step was to design a simple data latch to hold the row and column numbers after a successful key press detect sequence. The following block diagram illustrates the required inputs and outputs for the data latch:



Output data latch block diagram

The output data latch is fairly typical of any data latch. If the LD signal is high, the contents of D(3:0) are latched to the ROW(1:0) and COL(1:0) outputs on the next rising clock edge. The latch was implemented using Verilog HDL code, seen in Figure #1 of the Appendix.

The source codes written in the above designs were then simulated by using the QUARTUS II and the generated waveform was attached in the Appendix. The result obtained was similar with the initial expectation.

Implementation & Testing

This design was not implemented in hardware, but was successfully simulated and tested using Altera Quartus II. The simulation was run for the Altera MAX7000S EPM7128SLC84-15 device. The simulation results suggest that this device should be capable of executing the design created in this lab.

To prepare for the implementation of this design into hardware for next week's lab, the Byte Blaster and CPLD were wired into a breadboard and connected to the parallel port of the simulation computer according to the wiring diagram shown in lecture. A "Blank Check" was performed to confirm that the device was connected properly. Initially this "Blank Check" test returned negative results, but upon further examination it was found that the parallel cable of the simulation computer was faulty. Once performed with a functional cable, the device test returned the desired results confirming the correct wiring for all three components: 1) Column Scanner results seen in Figure #2, 2) Row Encoder results seen in Figure #3, and 3) Output Data Latch results seen in Figure #4.

Conclusion

This lab was extremely productive for our team members. The programming and implantation went very smoothly. The column scanner, row encoder, and output data-latch had minimal errors and what errors the team did have were dealt with quickly and effectively. The team functioned on a very high and professional level.

Appendices

```
//HW2 Column_Scanner

module Column_Scanner (ENA, CLK, C,
COL);

input ENA, CLK;
output [3:0] C;
output [1:0] COL;

reg [1:0] COL;
reg [3:0] C;

always @(posedge CLK) begin
    if(ENA) COL=COL+1;
    case (COL)
        2'b00: C=4'b1110;
        2'b01: C=4'b1101;
        2'b10: C=4'b1011;
        2'b11: C=4'b0111;
    endcase
end
endmodule

//HW2 Row Encoder

module Row_Encoder(R, ROW, ZDET);

input [3:0] R;
output [1:0] ROW;
output ZDET;

reg [1:0] ROW;
reg ZDET;

always @ (R) begin //Anytime R changes
run the process
    case (R)
        4'b1110: begin
            ROW=2'b00;
            ZDET=1;
        end
        4'b1101: begin
            ROW=2'b01;
            ZDET=1;
        end
        4'b1011: begin
            ROW=2'b10;
            ZDET=1;
        end
        4'b0111: begin
            ROW=2'b11;
            ZDET=1;
        end
        default: ZDET=0;
    endcase
end
endmodule

//HW2 Output_Data Latch

module Output_Data_Latch (D, LD, CLK,
ROW, COL);

input [3:0] D;
input CLK, LD;
output [1:0] ROW;
output [1:0] COL;

reg [1:0] ROW;
reg [1:0] COL;

always @ (posedge CLK) begin
    if(LD) begin
        ROW<=D[3:2];
        COL<=D[1:0];
    end
end
endmodule
```

Figure #1: VHDL Code for 1) Column Scanner, 2) Row Encoder, and 3) Output Data Latch

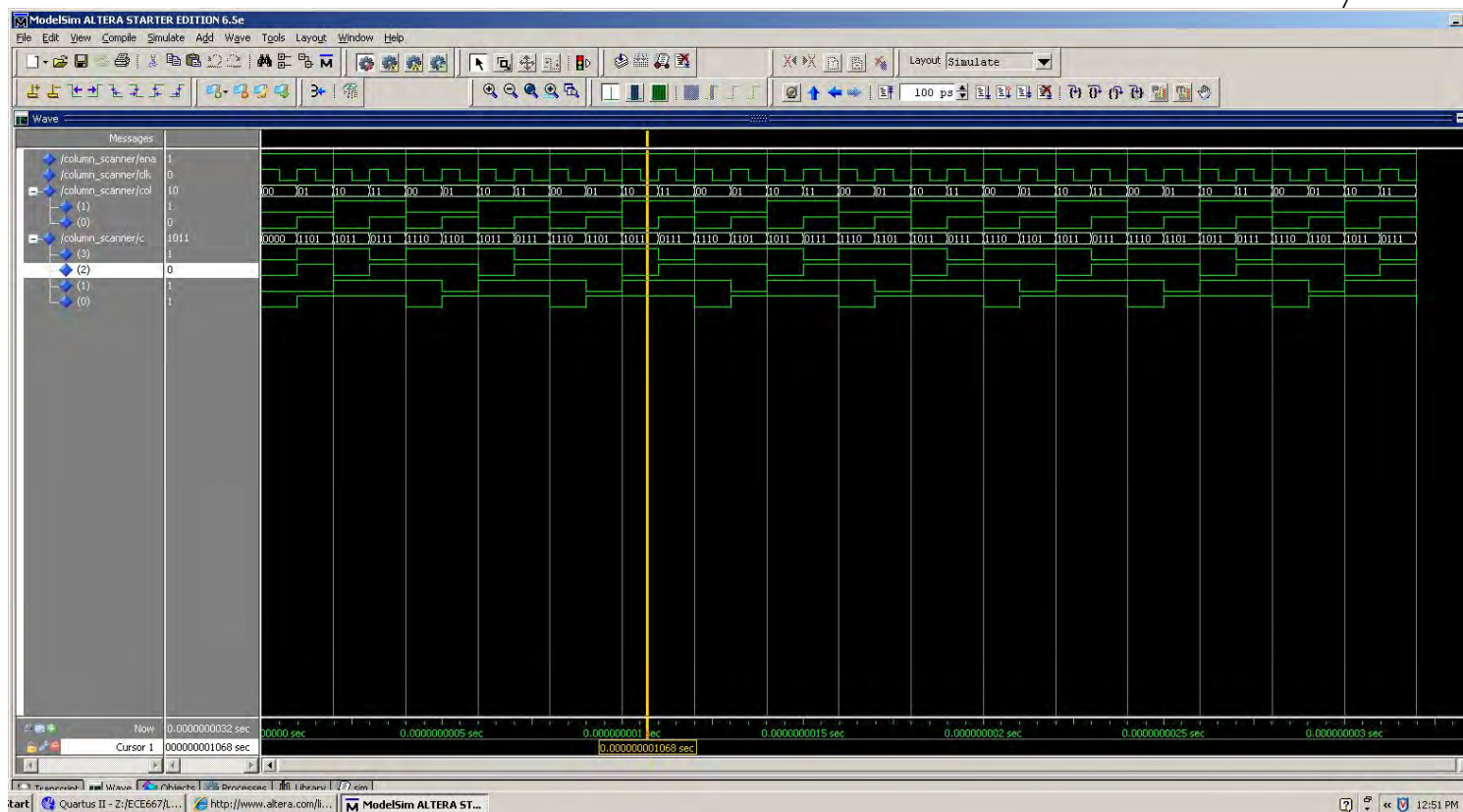


Figure #2: Altera's Quartus II – 1) Column Scanner Output

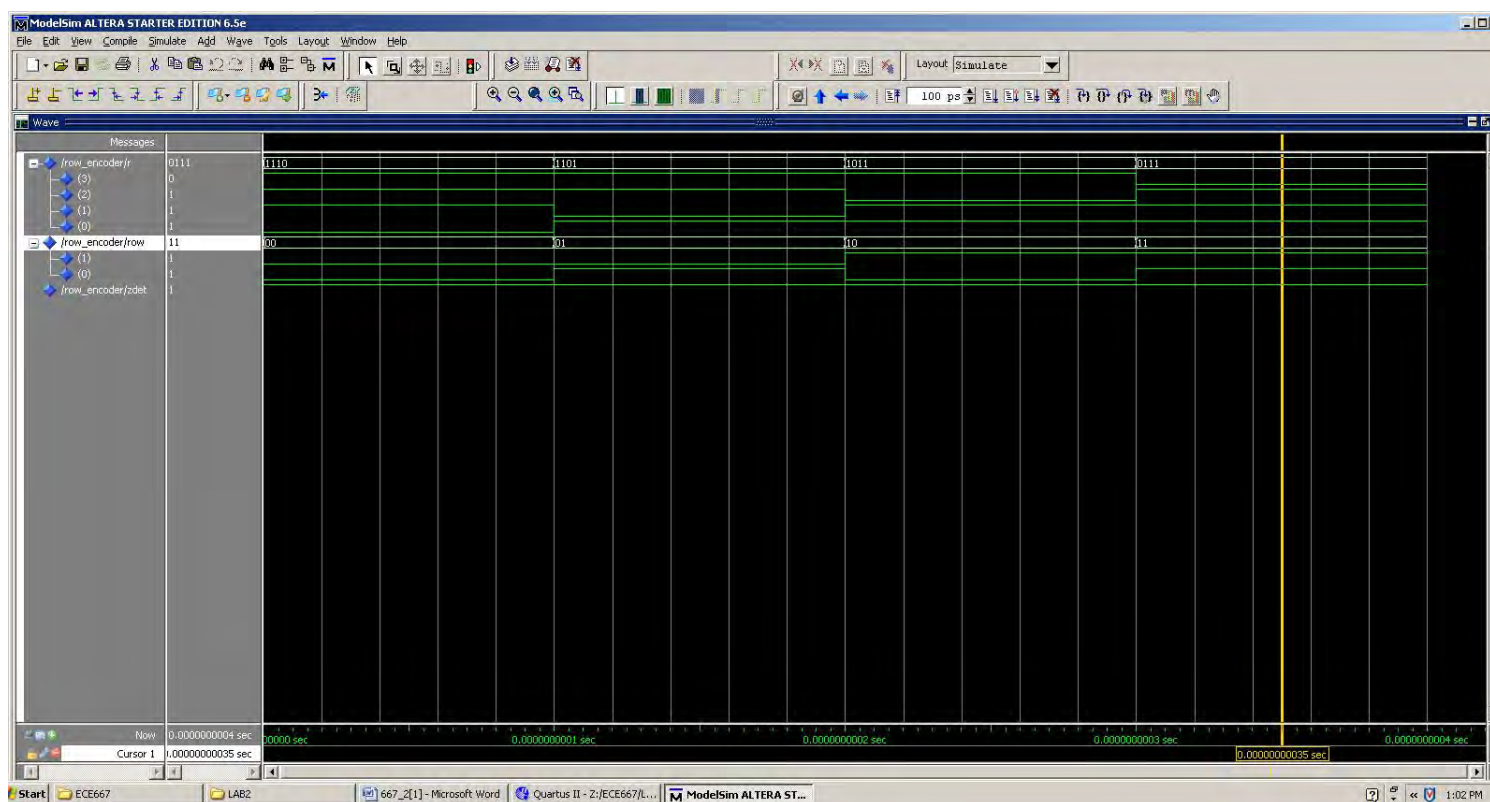


Figure #3: Altera's Quartus II – 2) Row Encoder Output

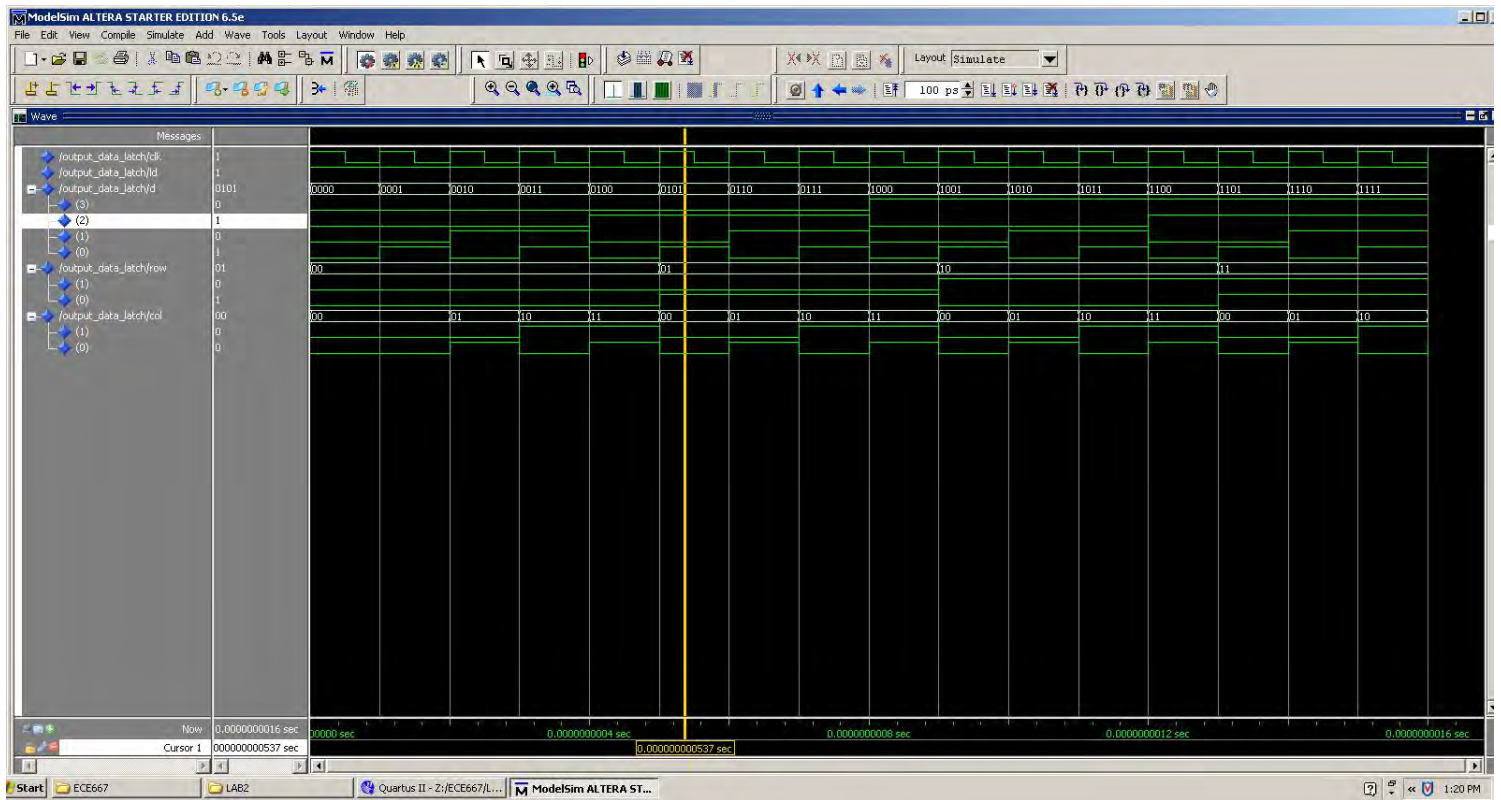


Figure #4: Altera's Quartus II – 3) Output Data Latch Output

```
1  -----
2  -- By: Larissa Borcz
3  -- Project Name: Lab #2
4  -- Module Name:   Homework #2 - Behavioral
5  -- ECE677, MW 12:30pm-1:18pm
6  -----
7
8  -- Column_Scanner
9  LIBRARY ieee;
10 USE ieee.STD_LOGIC_1164.all;
11 USE ieee.STD_LOGIC_UNSIGNED.all;
12
13 entity Column_Scanner is
14     port(ENA, CLK : in std_logic;
15          C : out std_logic_vector (3 downto 0);
16          COL : out std_logic_vector (1 downto 0) );
17 end Column_Scanner;
18
19 architecture archi of Column_Scanner is
20     signal count: STD_LOGIC_VECTOR(1 downto 0);
21     begin
22         process (CLK)
23         begin
24             if (CLK'event and CLK='1') then
25                 if (ENA='1') then
26                     CASE count is
27                         WHEN "00" => C <= "1110";
28                         WHEN "01" => C <= "1101";
29                         WHEN "10" => C <= "1011";
30                         WHEN "11" => C <= "0111";
31                         WHEN others => C <= "1111";
32                     END CASE;
33                     COL <= count+1;
34                 else
35                     COL <= count;
36                 end if;
37             end if;
38         end process;
39     end archi;
40
41 --Row_Encoder
42 LIBRARY ieee;
43 USE ieee.STD_LOGIC_1164.all;
44 USE ieee.STD_LOGIC_UNSIGNED.all;
45
46 entity Row_Encoder is
47     port(R : in std_logic_vector (3 downto 0);
48          ROW : out std_logic_vector (1 downto 0);
49          ZDET : out std_logic );
50 end Row_Encoder;
51
52 architecture archi of Row_Encoder is
53     begin
54         process (R)
55         begin
56             CASE R is
57                 WHEN "1110" => ROW <= "00";
58                 WHEN "1101" => ROW <= "01";
```

```
59         WHEN "1011" => ROW <= "10";
60         WHEN "0111" => ROW <= "11";
61         WHEN others => ROW <= "11";
62     END CASE;
63     ZDET <= NOT(R(3) AND R(2) AND R(1) AND R(0));
64 end process;
65 end archi;
66
67
68
69 -- Output_Data_Latch
70 LIBRARY ieee;
71 USE ieee.STD_LOGIC_1164.all;
72 USE ieee.STD_LOGIC_UNSIGNED.all;
73
74 entity Output_Data_Latch is
75     port(D : in std_logic_vector (3 downto 0));
76         LD : in std_logic;
77         CLK : in std_logic;
78         ROW : out std_logic_vector (1 downto 0);
79         COL : out std_logic_vector (1 downto 0) );
80 end Output_Data_Latch;
81
82 architecture archi of Output_Data_Latch is
83 begin
84     process (CLK)
85     begin
86         if (CLK'event and CLK='1') then
87             if (LD='1') then
88                 ROW <= D(3 downto 2);
89                 COL <= D(1 downto 0);
90             end if;
91         end if;
92     end process;
93 end archi;
```

ECE667

Lab Report #3

Lab Completed: 7/19/11

Due Date: 7/26/11

By:

Larissa Borez

Sean Ewing

Feiran Lei

Ji Tu

<u>Table of Contents</u>	<u>Page #</u>
1) Abstract	2
2) Introduction	3
3) Discussion	3
a. Keypad Scanner System Controller	3
b. Keypad simulation	4
c. Clock Divider	6
4) Implementation & Testing	7
5) Conclusion	7
6) Appendices	8

Abstract

This project aims at the development of the Keypad Scanner System components. The Keypad scanner has 6 components: 1) Column Scanner, 2) Row Encoder, 3) Output Data Latch, 4) Keypad Scanner System Controller, 5) Keypad Simulator, and 6) Clock Divider. In this report, the designing of the keypad scanner system controller, keypad simulator, and clock divider, and implementation of the six components stated above, are focused on.

- Keypad Scanner System Controller: In the keypad scanner system controller setup, upon system power-up, the column scanner should be enabled by setting $CS_ENA = 1$. The column scanner will then continuously scan until the row encoder encounters a low signal on one of the rows ($ZDET = 1$), signaling that a key has been pressed. The system controller immediately disables the column scanner by setting $CS_ENA = 0$, and begins the de-bounce sequence. The system controller is to wait for a period of about 10 ms, and then check the value of ZDET again. If ZDET is low after the delay period, then it is assumed that a key was not pressed. If so, the system resumes scanning. If ZDET is still high, then it is assumed that a valid key press has been detected. Both the VALID, and LATCH_LD signals are to be set high, signaling there was a valid key press, as well as latching in the data specifying which key was actually pressed. The VALID signal is to remain asserted as long as the key remains pressed ($ZDET = 1$). Upon release of the key ($ZDET = 0$), the VALID signal will go low, and the keypad scanner should resume scanning columns and waiting for the next key to be pressed. $VALID \neq \text{NOT } ZDET$, as VALID is asserted only after the debounce period.
- Keypad Simulator: The KEY[3:0] input is used to select one of the 16 keys to simulate. The physical key-press is simulated with the ASSERT signal. When $ASSERT = 0$, the simulation acts as if no keys are being pressed regardless of the key selected. If all of the rows are pulled high through pull-up resistors in the physical keypad, then the R[3:0] outputs should be read as "1111". When $ASSERT = 1$, the simulation acts as if one of the physical keys is pressed. The ROW and COL values are then assigned in relation to the Keypad Simulation table.
- Clock Divider: Reduces a 4MHz signal to a 1 kHz signal.

We then enter the design components for the keypad scanner and create for each of them, inputting the designs for the column scanner, row encoder, output data latch, system controller, keypad simulator, and clock divider. First step is to test the functionality of the design by

attaching the simulation of a keypad to the column outputs and row inputs of the scanner. Next is to enter the design for the 4000-to-1 clock divider, and simulate the component for one full period to make sure that it provides the expected approximate clock frequency. Once it is confirmed that everything is working as expected, the design will be loaded onto the FPGA and tested in hardware.

Introduction

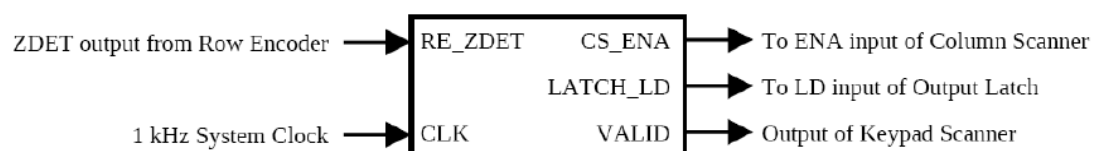
The purpose of this lab is to finish entering the design components for the keypad scanner and create symbols (also known as macros) for each of them. Students will input the designs for the column scanner, row encoder, output data latch, and system controller, you will test the functionality of the keypad scanner you designed by attaching the simulation of a keypad to the column outputs and row inputs of your scanner. Students will also enter in their design(s) for the 4000-to-1 clock divider. Students will confirm that everything is working as expected, load the design onto the FPGA and test everything with hardware.

Discussion

For the first part of the lab, we designed a controller to coordinate the signals of all the components. Then we developed a simulation model of the physical keypad, which will aid in debugging our final design of the keypad scanner. Finally, we designed a clock divider to divide 4MHz signal into specified 1KHz clock signal for the keypad scanner.

1. Keypad Scanner System Controller

The system components have been designed in Lab 2. We will need to design a controller to coordinate the signals in the system, as well as tell the outside world when a valid key press has been detected. The following block diagram illustrates the required inputs and outputs of the controller:

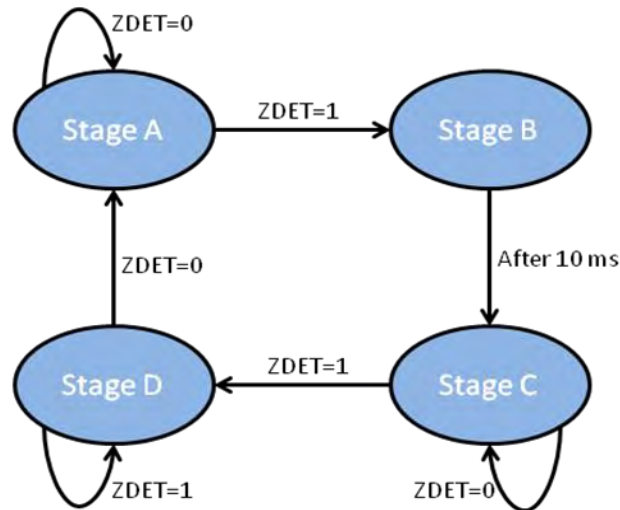


System controller block diagram

The following table and state diagram illustrate the specified requirements:

STATE	RE_ZDET	CS_ENA	LATCH_LD	VAILD	NEXT STATE
A	0	1	0	0	A
A	1	1	0	0	B
B	0	0	X	X	C (if delay of 10 ms)
B	1	0	X	X	C (if delay of 10 ms)
C	0	0	0	0	C
C	1	0	0	0	D
D	0	0	1	1	A
D	1	0	1	1	D

State table of system controller



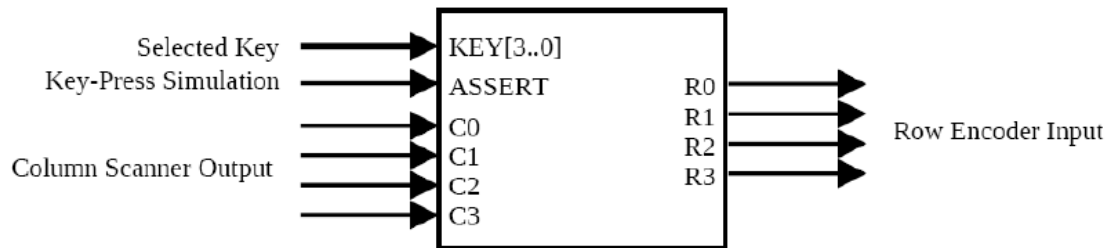
State diagram of system controller

The circuit was easily implemented using Verilog code as seen in Figure #3, in the Appendix.

2. Keypad simulation

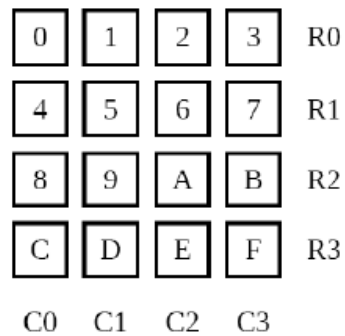
After designing and verifying that all of the system components work individually, we want to see if the entire system work. We developed a simulation model of the physical keypad that will be constructed in the implementation part. The purpose of doing so is to help to debug the final design of the keypad scanner before it is programmed into the CPLD.

The Keypad Scanner takes the Key inputs and if ASSERT is asserted then R[0..3] corresponds to the key pressed. The block diagram for the keypad simulation is shown below:



Keypad simulation block diagram

The key mapping of the keypads will have the following layout:



Layout of the Keypad

The following table illustrates the specified requirements:

ASSERT	Key[3..0]	R[0..3]
0	----	1111
1	0000	1000
1	0001	1000
1	0010	1000
1	0011	1000
1	0100	0100
1	0101	0100
1	0110	0100

1	0111	0100
1	1000	0010
1	1001	0010
1	1010	0010
1	1011	0010
1	1100	0001
1	1101	0001
1	1110	0001
1	1111	0001

State table for the keypad simulation

Then we could derive the equations as:

$$R0 = \text{Key3}' \cdot \text{Key2}' + \text{ASSERT}'$$

$$R1 = \text{Key3}' \cdot \text{Key2} + \text{ASSERT}'$$

$$R2 = \text{Key3} \cdot \text{Key2}' + \text{ASSERT}'$$

$$R3 = \text{Key3} \cdot \text{Key2} + \text{ASSERT}'$$

The circuit was implemented Verilog code as seen in Figure #3, in the Appendix.

3. 4MHz to 1kHz Clock Divider:

The clock divider is to take as input the 4MHz original clock signal and reduce it to a 1kHz clock signal. The clock divider must use the same clock signal on all parts. To accomplish this design we are going to use multiple 4-bit counters to count from 0 to 4000 in order to divide the 4MHz clock signal into a 1kHz clock signal. The counters are arranged in cascade so that when 4000 is reached the output is a pulse to achieve the 1kHz clock speed desired.

The circuit was implemented Verilog code as seen in Figure #3, in the Appendix.

Finally, by putting all components in Lab2 and Lab3 together, we have the schematic circuit for with the Keypad Simulator, seen in Figure #4 in the Appendix, and without the Keypad Simulator, as seen in Figure #5, in the Appendix.

Implementation & Testing

The complete designs (Figure #1 and Figure #2), were successfully simulated and tested using Altera Quartus II (Figure #4 and Figure #5), and then implemented into hardware (Figure #6 and Figure #7). The first time, we failed to run it on the hardware since the wires were loosely connected. LED display remained one without any changes after we loaded the program and pressed the button. After our double checked the wires, the simulation ran for the Altera MAX7000S EPM7128SLC84-15 device. The simulation results suggest that this device should be capable of executing the design created in this lab. These results were then verified by implementing the design into the CPLD and successfully displaying the press key using the 7 segment display.

Conclusion

Throwing all of the programming modules together in Quartus II was very simplistic and designing the modules in schematic form were unremarkable. However, actually getting each module to fully function with the next proved very difficult. Through several different troubleshooting techniques, the students were able to determine what was going wrong with the modules and fix the necessary code. The program was uploaded and the hardware tested per lab procedures.

Appendices

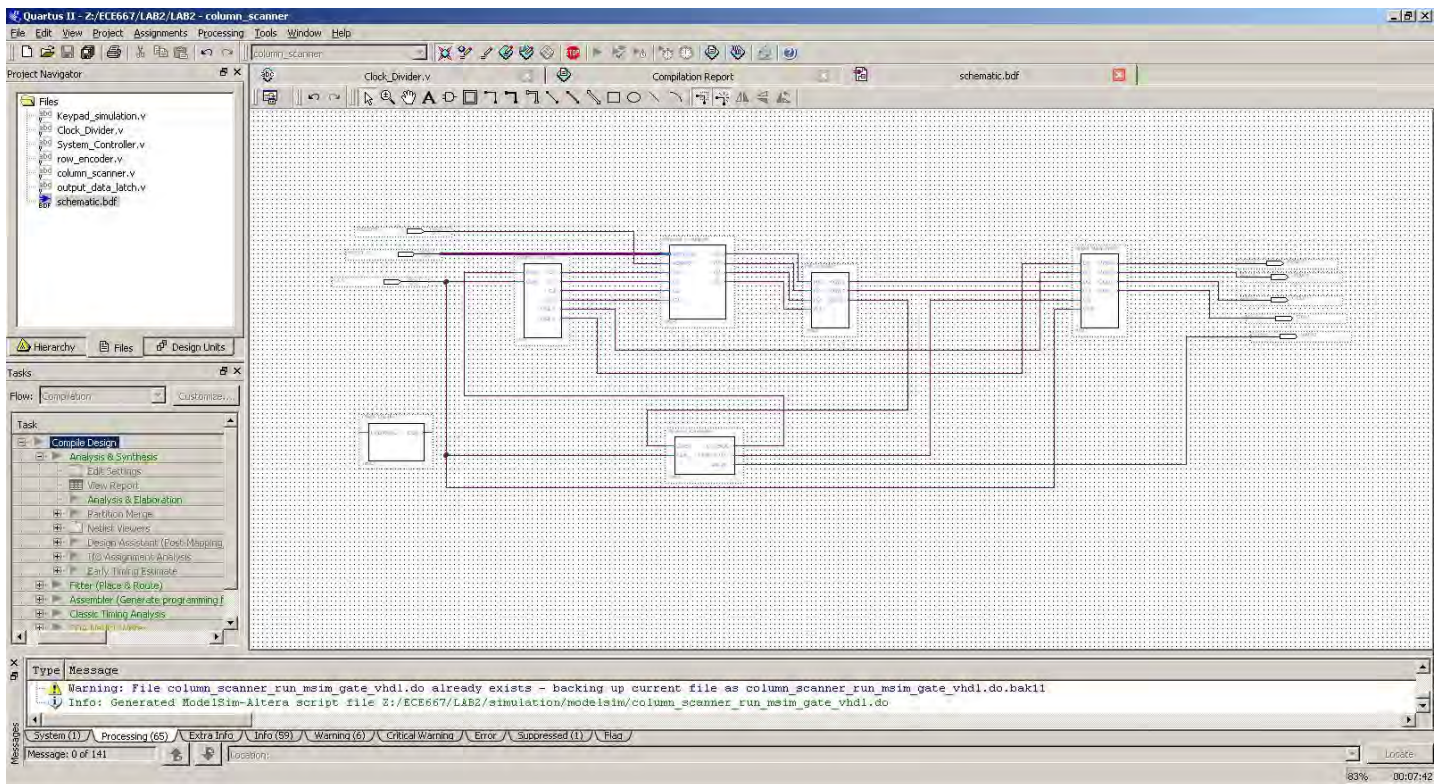


Figure 1: Altera's Quartus II Circuit Design – Keypad w/ Key Simulator

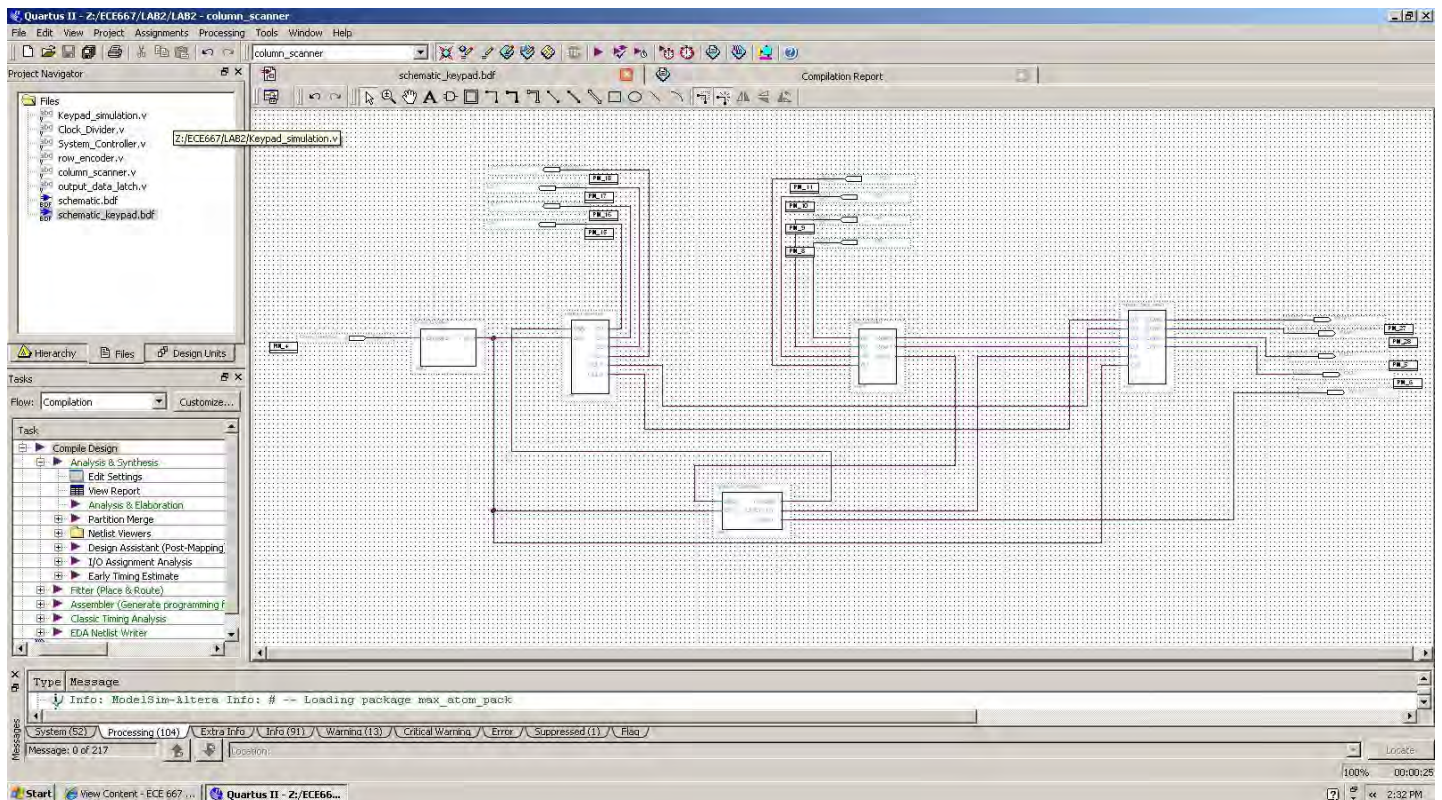


Figure 2: Altera's Quartus II Circuit Design – Keypad w/o Key Simulator

// Keypad Simulator

```

module Keypad_simulation (KEY, ASSERT, C0,C1,C2,C3,
R0,R1,R2,R3);

input[3:0] KEY;
input ASSERT,C0,C1,C2,C3;
output R0,R1,R2,R3;

reg R0,R1,R2,R3;

always @ (ASSERT) begin
    if (!ASSERT) begin R0=1;R1=1;R2=1;R3=1;
end
                        else begin
                        case(KEY)
0: R0=C0;
1: R0=C1;
2: R0=C2;
3: R0=C3;
4: R1=C0;
5: R1=C1;
6: R1=C2;
7: R1=C3;
8: R2=C0;
9: R2=C1;
10: R2=C2;
11: R2=C3;
12: R3=C0;
13: R3=C1;
14: R3=C2;
15: R3=C3;
                        endcase
                        end
end
endmodule

```

// Clock Divider

```

module Clock_Divider (FOURMHz, CLK);
input FOURMHz;
output CLK;

reg CLK;
reg [11:0] counter;
reg state;

always @ (posedge FOURMHz) begin
    case (state)
        0: begin
            CLK=1;
            counter=counter+1;
            if (counter==2000) begin
                state=1;
                counter=0;
            end
        end
        1: begin

```

```

CLK=0;
counter=counter+1;
if (counter==2000) begin
    state=0;
    counter=0;
end
end

```

```

end
default: state=0;

```

```

endcase
end
endmodule

```

// System Controller

```

module System_Controller (ZDET, CLK, CS_ENA, LATCH_LD,
VALID);
input ZDET, CLK;
output CS_ENA, LATCH_LD, VALID;

reg CS_ENA, LATCH_LD, VALID;
reg [1:0] state;
reg [3:0] counter;

always @ (ZDET) begin
    if(ZDET) begin
        CS_ENA=0;
    end
    else CS_ENA=1;
end

always @ (posedge CLK) begin
    case (state)
        0: begin
            if(ZDET) begin
                state=1;
                counter=0;
            end
        end
        1: begin
            counter=counter+1;
            if(counter==9 && ZDET) begin
                counter=0;
                VALID=1;
                LATCH_LD=1;
                state=2;
            end
            else if(counter==9 && !ZDET) begin
                counter=0;
                state=0;
            end
        end
        2: begin
            if(!ZDET) begin
                VALID=0;
                LATCH_LD=0;
                state=0;
            end

```

```

        end
    end
    default: state=0;
endcase
end
endmodule

// Row Encoder
module row_encoder(R0,R1,R2,R3,ROW0,ROW1, ZDET);
input R0,R1,R2,R3;
output ROW0,ROW1;
output ZDET;

reg [3:0] R;
reg [1:0] ROW;
reg ROW0,ROW1;
reg ZDET;

always @ (1) begin //Anytime R changes run the process
    R[0]=R0;
    R[1]=R1;
    R[2]=R2;
    R[3]=R3;
    // R[0:3] or R[3:0]? up/down
    case (R)
        4'b0111: begin
            ROW=2'b11;
            ZDET=1;
        end
        4'b1011: begin
            ROW=2'b10;
            ZDET=1;
        end
        4'b1101: begin
            ROW=2'b01;
            ZDET=1;
        end
        4'b1110: begin
            ROW=2'b00;
            ZDET=1;
        end
        default: ZDET=0;
    endcase
    ROW0=ROW[0];
    ROW1=ROW[1];
end
endmodule

// Column Scanner
module column_scanner (ENA, CLK, C0,C1,C2,C3,
COL1,COL0);
input ENA, CLK;
output C0,C1,C2,C3;
output COL1,COL0;

reg [0:3] C;

reg [1:0] COL;
reg COL1,COL0;
reg C0,C1,C2,C3;

always @(posedge CLK) begin
    if(ENA) COL=COL+1;
    case (COL)
        2'b00: C=4'b0111;
        2'b01: C=4'b1011;
        2'b10: C=4'b1101;
        2'b11: C=4'b1110;
    endcase
    COL0=COL[0];
    COL1=COL[1];
    C0=C[0];
    C1=C[1];
    C2=C[2];
    C3=C[3];
end
endmodule

// Data Latch
module output_data_latch (D0,D1,D2,D3, LD, CLK,
ROW0,ROW1, COL0,COL1);
input D0,D1,D2,D3;
input CLK, LD;
output ROW1,ROW0;
output COL0,COL1;

reg ROW0,ROW1;
reg COL0,COL1;

always @ (posedge CLK) begin
    if(LD) begin
        ROW1=D3;
        ROW0=D2;
        COL1=D1;
        COL0=D0;
    end
end
endmodule

```

Figure 3: Verilog Code – All Components

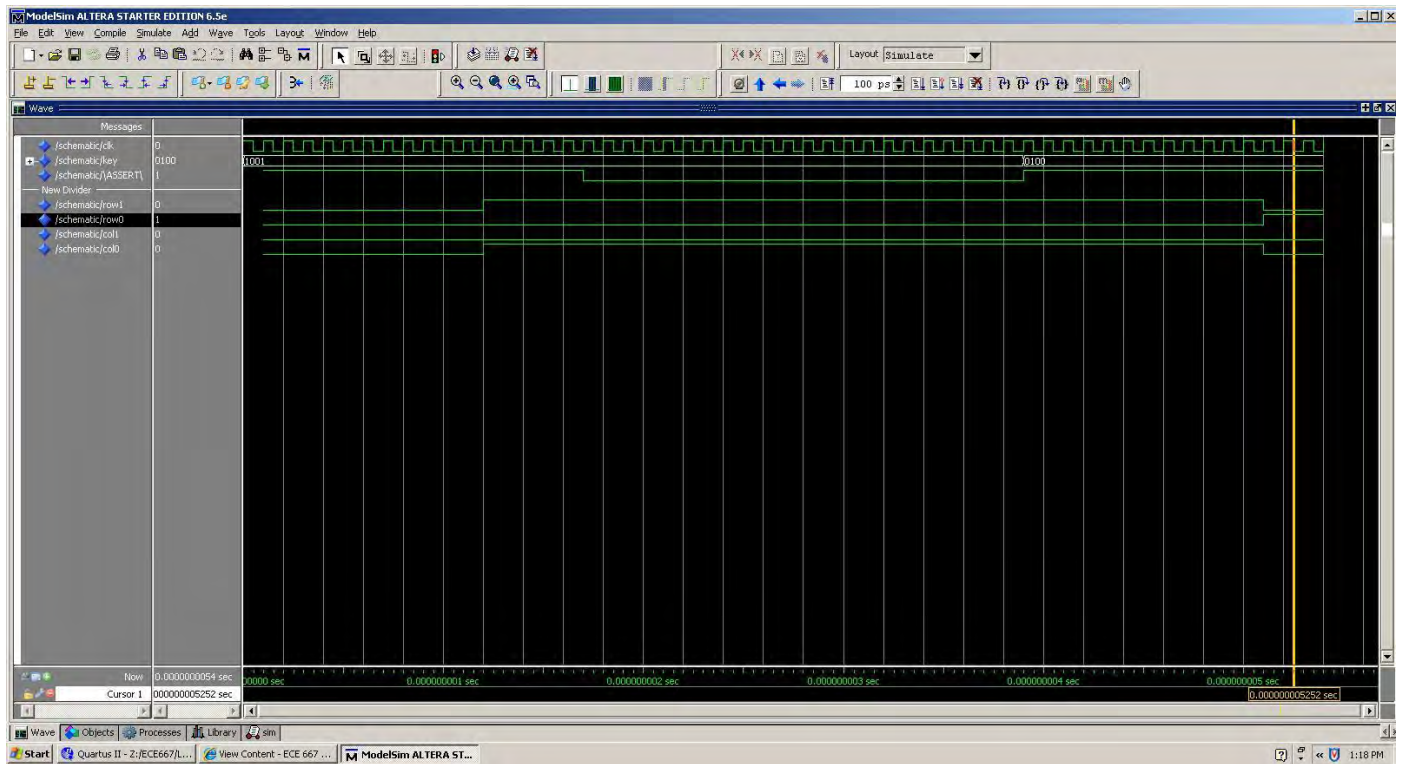


Figure 4: Altera's Quartus II – Method Keypad Waveform Output

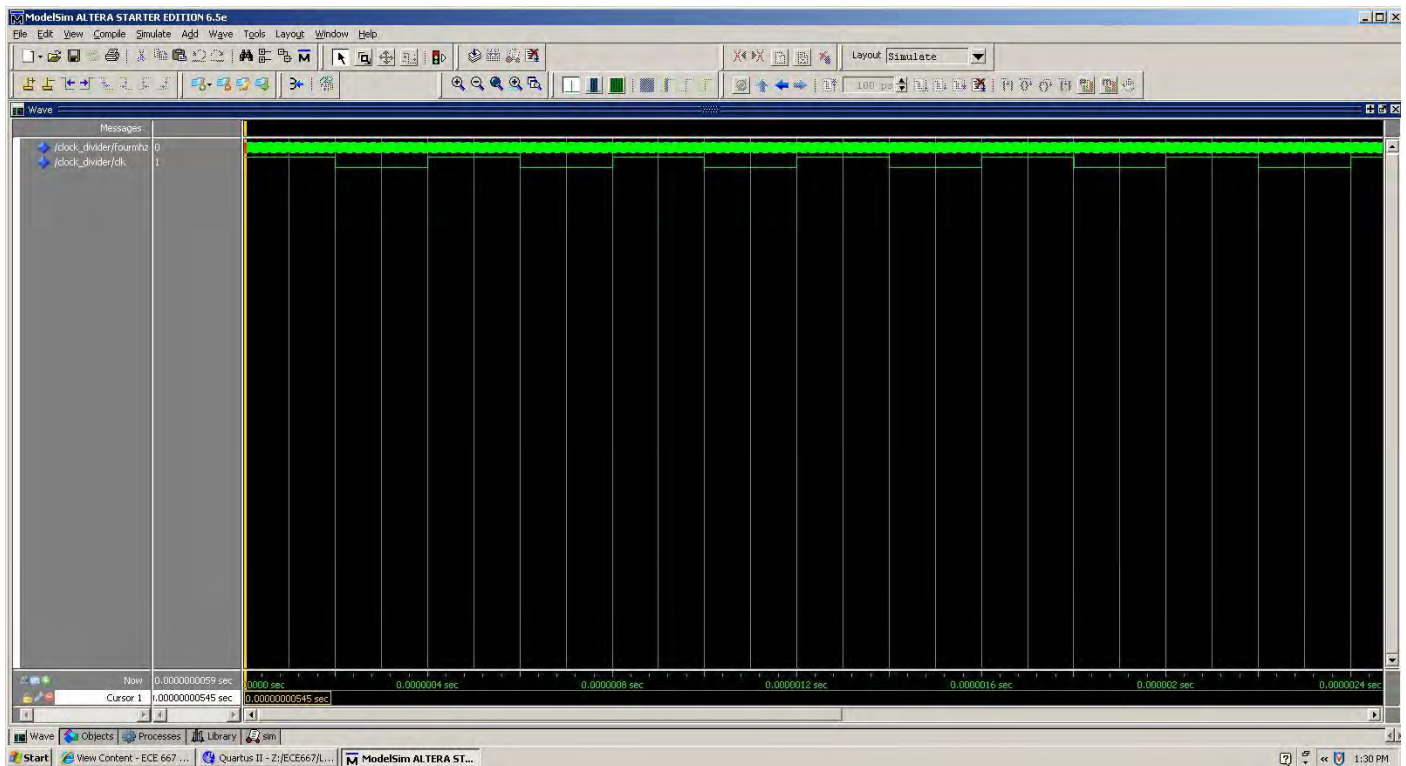


Figure 5: Altera's Quartus II – Method Clock Divider Waveform Output

// Keypad Simulator

```
module Keypad_simulation (KEY, ASSERT, C0,C1,C2,C3,
R0,R1,R2,R3);

input[3:0] KEY;
input ASSERT,C0,C1,C2,C3;
output R0,R1,R2,R3;

reg R0,R1,R2,R3;

always @ (ASSERT) begin
    if (!ASSERT) begin R0=1;R1=1;R2=1;R3=1;
end
```

```
        else begin
            case(KEY)
```

```
                0: R0=C0;
                1: R0=C1;
                2: R0=C2;
                3: R0=C3;
                4: R1=C0;
                5: R1=C1;
                6: R1=C2;
                7: R1=C3;
                8: R2=C0;
                9: R2=C1;
                10: R2=C2;
                11: R2=C3;
                12: R3=C0;
                13: R3=C1;
                14: R3=C2;
                15: R3=C3;
```

```
            endcase
        end
```

```
end
endmodule
```

// Clock Divider

```
module Clock_Divider (FOURMHz, CLK);
input FOURMHz;
output CLK;
```

```
reg CLK;
reg [11:0] counter;
reg state;
```

```
always @ (posedge FOURMHz) begin
    case (state)
```

```
        0: begin
            CLK=1;
            counter=counter+1;
            if (counter==2000) begin
                state=1;
                counter=0;
            end
        end
```

```
    end
    1: begin
```

```
        CLK=0;
        counter=counter+1;
        if (counter==2000) begin
            state=0;
            counter=0;
        end
    end
    default: state=0;
```

```
endcase
```

```
end
endmodule
```

// System Controller

```
module System_Controller (ZDET, CLK, CS_ENA, LATCH_LD,
VALID);
```

```
input ZDET, CLK;
output CS_ENA, LATCH_LD, VALID;
```

```
reg CS_ENA, LATCH_LD, VALID;
reg [1:0] state;
reg [3:0] counter;
```

```
always @ (ZDET) begin
    if(ZDET) begin
        CS_ENA=0;
    end
    else CS_ENA=1;
```

```
end
```

```
always @ (posedge CLK) begin
    case (state)
```

```
        0: begin
```

```
            if(ZDET) begin
                state=1;
                counter=0;
            end
```

```
        end
        1: begin
```

```
            counter=counter+1;
            if(counter==9 && ZDET) begin
                counter=0;
                VALID=1;
                LATCH_LD=1;
                state=2;
            end
```

```
        else if(counter==9 && !ZDET) begin
            counter=0;
            state=0;
        end
```

```
        end
        2: begin
```

```
            if(!ZDET) begin
                VALID=0;
                LATCH_LD=0;
                state=0;
```

```

        end
    end
    default: state=0;
endcase
end
endmodule

// Row Encoder
module row_encoder(R0,R1,R2,R3,ROW0,ROW1, ZDET);
input R0,R1,R2,R3;
output ROW0,ROW1;
output ZDET;

reg [3:0] R;
reg [1:0] ROW;
reg ROW0,ROW1;
reg ZDET;

always @ (1) begin //Anytime R changes run the process
    R[0]=R0;
    R[1]=R1;
    R[2]=R2;
    R[3]=R3;
    // R[0:3] or R[3:0]? up/down
    case (R)
        4'b0111: begin
            ROW=2'b11;
            ZDET=1;
        end
        4'b1011: begin
            ROW=2'b10;
            ZDET=1;
        end
        4'b1101: begin
            ROW=2'b01;
            ZDET=1;
        end
        4'b1110: begin
            ROW=2'b00;
            ZDET=1;
        end
        default: ZDET=0;
    endcase
    ROW0=ROW[0];
    ROW1=ROW[1];
end
endmodule

```

```

// Column Scanner
module column_scanner (ENA, CLK, C0,C1,C2,C3,
COL1,COL0);
input ENA, CLK;
output C0,C1,C2,C3;
output COL1,COL0;

reg [0:3] C;

```

```

reg [1:0] COL;
reg COL1,COL0;
reg C0,C1,C2,C3;

always @(posedge CLK) begin
    if(ENA) COL=COL+1;
    case (COL)
        2'b00: C=4'b0111;
        2'b01: C=4'b1011;
        2'b10: C=4'b1101;
        2'b11: C=4'b1110;
    endcase
    COL0=COL[0];
    COL1=COL[1];
    C0=C[0];
    C1=C[1];
    C2=C[2];
    C3=C[3];
end
endmodule

```

```

// Data Latch
module output_data_latch (D0,D1,D2,D3, LD, CLK,
ROW0,ROW1, COL0,COL1);
input D0,D1,D2,D3;
input CLK, LD;
output ROW1,ROW0;
output COL0,COL1;

reg ROW0,ROW1;
reg COL0,COL1;

always @ (posedge CLK) begin
    if(LD) begin
        ROW1=D3;
        ROW0=D2;
        COL1=D1;
        COL0=D0;
    end
end
endmodule

```

ECE667

Lab Report #4

Lab Completed: 8/2/11

Due Date: 8/9/11

By:

Larissa Borcz

Sean Ewing

Feiran Lei

Ji Tu

<u>Table of Contents</u>	<u>Page #</u>
1) Abstract	2
2) Introduction	2
3) Discussion	2
a. Baud Rate Generator	3
b. Asynchronous Serial Transmitter	3
c. System Controller	4
4) Implementation & Testing	5
5) Conclusion	5
6) Appendices	6

Abstract

This project aims at designing the necessary hardware components to connect a design to the workstation computer through the systems RS232 port, and adding the RS232 transmitter module to the previous design. This involved writing, simulating, and testing three modules: the baud rate generator, the asynchronous serial transmitter, and the transmission controller. The Asynchronous Serial Transmitter is capable of transmitting data at one of four selectable bit-rates. The transmission rate selection will be done through a Baud-Rate Generator, which is essentially a selectable version of the frequency divider designed in the previous homework. The System Controller for the remote control module is used to coordinate the signals from the system components so that the keys pressed on the keypad are transmitted serially through the transmitter. The CPLD, RS232 driver, and the DB-9 connector, were wired together and the DB-9 plug connected to the lab computer in order to receive key presses via the Hyperterminal software. The keypad scanner hardware previously designed is also used.

Introduction

The purpose of this lab is will design the necessary hardware components to connect the students' design to their workstation computer through the systems RS232 port. Students will input the designs for this communication code and will test the functionality of the keypad scanner by displaying the input key on the 7-segment display and also sending the input key to the computer via the RS-232 bus. The Hyperterminal program will display the correct value entered. Students will confirm that everything is working as expected, load the design onto the FPGA and test everything with hardware.

Discussion

In this lab, we continued to use the keypad scanner hardware we have designed in Lab 3, but add some designs of necessary hardware components to connect our design to the workstation computer through the systems RS232 port. We designed an Asynchronous Serial Transmitter which is capable of transmitting data at one of four selectable bit-rates. Transmission rate selection will be done through a Baud-Rate Generator, which is basically just a selectable version of the frequency divider. Finally, we designed a system controller for the remote control

module to coordinate the signals from the system components so that the keys pressed on the keypad are transmitted serially through the transmitter.

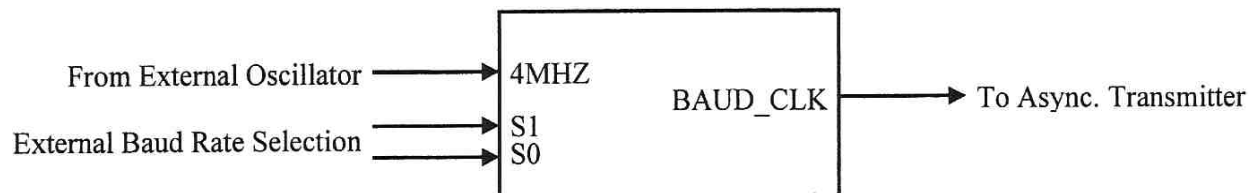
1. Baud Rate Generator

This baud rate generator is similar to our previous clock divider in Lab 3. The data will be transmitted with one of four possible bit rates, and this should be selectable. The following table shows the clock frequencies our circuit generates:

Rate Select	Bit Rate	Clock Freq.
0 (00)	1200 bits/s	1200 Hz
1 (01)	2400 bits/s	2400 Hz
2 (10)	4800 bits/s	4800 Hz
3 (11)	9600 bits/s	9600 Hz

Clock frequencies

The block diagram for this circuit is shown below:

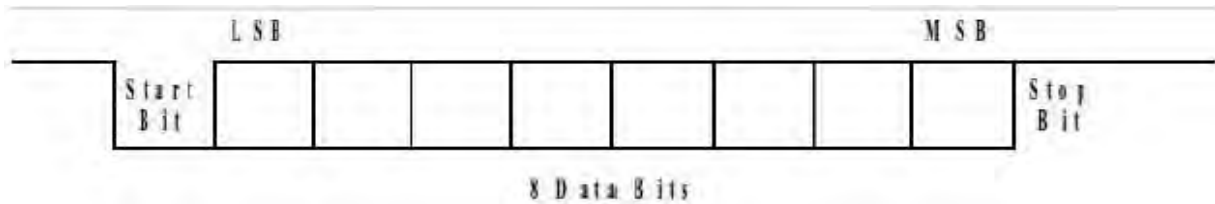


Block diagram for the baud rate generator

Calculations in order to run design the code, error check, and case select were done, as seen in Figure 4 of the Appendix. Case select tests were done for “00”, “01”, “10”, and “11” to confirm the correct waveform for the Baud Rate Generator code, as seen in Figure’s 5, 6, 7, and 8 of the Appendix. The circuit was implemented using Verilog code, as seen in Figure 1 of the Appendix.

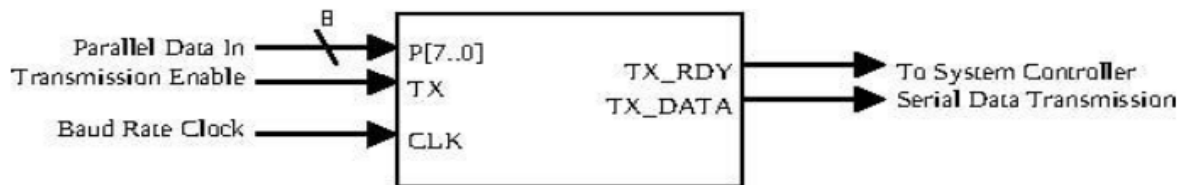
2. Asynchronous Serial Transmitter

After designing a baud rate generator which determines the rate at which the data will be sent, we then designed the asynchronous serial transmitter which can take parallel data from the circuit and transmit this data bit by bit to a serial receiver. Standard RS-232 protocol is used here to transmit the data. The format of the data is shown below:



Format of the data in RS232

The block diagram for the transmitter design is shown below:

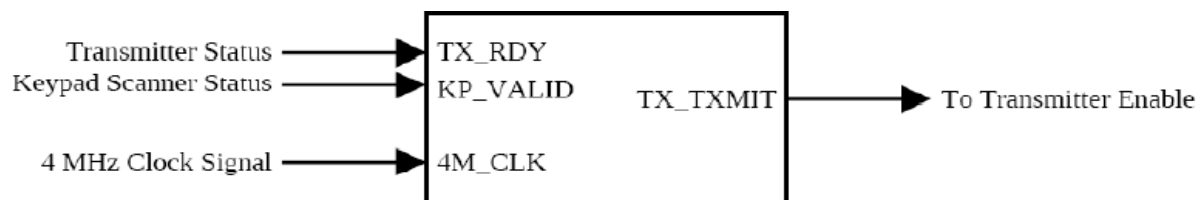


Block diagram for the transmitter

Eight bits of parallel data are placed on lines P(7:0) by first sending out bit P0 for one complete baud clock cycle. In order to make the signal conform to the RS232 protocol, the eight bits of data must be preceded by a start bit and followed by a stop bit. The waveform for the Asynchronous Serial Transmitter was output and checked for verification, as seen in Figure's 9 and 10 of the Appendix. The circuit was implemented using Verilog code, as seen in Figure 1 of the Appendix.

3. System Controller

We would like to combine the keypad scanner and transmitter together, so that when we press a key, the code for that key is sent serially through the transmitter. The block diagram for the controller should look as shown below:



Block diagram for the system controller

When the system begins operation transmission will initially be disabled (TX_TXMIT=0). When the controller detects that a valid key is currently pressed (KP_Valid=1), the controller should then allow one copy of the current key code to be transmitted by enabling the transmitter (TX_TXMIT=1). The waveform for the Serial System Controller was output and checked for verification, as seen in Figure 11 of the Appendix. The circuit was implemented using Verilog

code, as seen in Figure 1 of the Appendix.

Implementation & Testing

This complete design was simulated and tested using Altera Quartus II, as seen in Figure 2 of the Appendix, and then implemented into hardware using the specified pin plan, as seen in Figure 3 of the Appendix. The simulation was run for the Altera MAX7000S EPM7128SLC84-15 device and simulated successfully from beginning. And we connected the MAX233 chip and the RS-232 DB-9 connector to the previous designed schematic. Yet we had a problem after downloading to the device. We always had the correct LED 7 segments display correct while the hyper terminal serial signal delay for only one digit. After few hours debugging, we found the problem resulting from clock divider which is from lab two and needs to be deleted. Then, the circuit gave the correct response in both LED display and Hyperterminal, as seen in Figure 12 of the Appendix. It shows that the circuit works correctly as we expected before. The number of every pressing of a single keypad showed correspondingly on the software.

Conclusion

Throwing all of the programming modules together in Quartus II was very simplistic and designing the modules in schematic form were unremarkable. However, actually getting the entire module to fully function with the next proved very difficult. Our group was able to correctly send the input key to the computer, however, the previous input was sent when we pressed the following key.

For example, a student enters the following key..... 0125678934

Our program would display.....0012567893

All the students in the group went through our code meticulously and could not find a fault anywhere in our code. Eventually, we came to a troubleshooting conclusion that certain modules may be delayed. We removed the 4MHz to 1MHz module for testing and everything worked flawlessly. The program was uploaded and the hardware tested per lab procedures.

Appendices

Figure 1: Verilog Code – All Components

1) Baud Rate Generator:

```

module Baud_Rate_Generator (FOUR_MHZ, S, BAUD_CLK);

input FOUR_MHZ;
input [1:0] S;
output BAUD_CLK;

reg BAUD_CLK;
reg [11:0] counter;
reg state;
reg [11:0] CLK_FREQ;
always @ (posedge FOUR_MHZ) begin
    case (S)                                // Selectable clock frequencies
        0: CLK_FREQ=1667;                    // 1200Hz
        1: CLK_FREQ=833;                     // 2400Hz
        2: CLK_FREQ=417;                     // 4800Hz
        3: CLK_FREQ=208;                     // 9600Hz
    endcase
    case (state)
        0: begin                            // This state gives a high clock signal
            BAUD_CLK=1;
            counter=counter+1;
            if (counter==CLK_FREQ) begin     // Keeps clock in this state for a selected time span
                state=1;
                counter=0;
            end
        end
        1: begin                            // This state gives a low clock signal
            BAUD_CLK=0;
            counter=counter+1;
            if (counter==CLK_FREQ) begin     // Keeps the clock in this state for a selected time span
                state=0;
                counter=0;
            end
        end
        default: state=0;
    endcase
end
endmodule

```

2) Asynchronous Serial Transmitter

```

module Asynchronous_Serial_Transmitter (P, TX, CLK, TX_RDY, TX_DATA);

input [7:0] P;
input TX, CLK;
output TX_RDY;
output TX_DATA;

```

```

reg TX_RDY;
reg TX_DATA;
reg [3:0] data_bit;
reg [7:0] locked_P;
reg start;

always @ (posedge CLK) begin

    if(!TX && !start) begin
        TX_DATA=1;
        TX_RDY=1;
    end
    if(TX) start=1; // Start is held high till transmission is fully executed
    if(start) begin
        case(data_bit)
            0:begin // Start bit
                TX_DATA=0;
                TX_RDY=0;
                locked_P=P;
                data_bit=1;
            end
            1:begin // bit 0
                TX_DATA=locked_P[0];
                data_bit=2;
            end
            2:begin // bit 1
                TX_DATA=locked_P[1];
                data_bit=3;
            end
            3:begin // bit 2
                TX_DATA=locked_P[2];
                data_bit=4;
            end
            4:begin // bit 3
                TX_DATA=locked_P[3];
                data_bit=5;
            end
            5:begin // bit 4
                TX_DATA=locked_P[4];
                data_bit=6;
            end
            6:begin // bit 5
                TX_DATA=locked_P[5];
                data_bit=7;
            end
            7:begin // bit 6
                TX_DATA=locked_P[6];
                data_bit=8;
            end
            8:begin // bit 7
                TX_DATA=locked_P[7];
                data_bit=9;
            end
        endcase
    end
end

```

```

        end
        9:begin                                // stop bit
            TX_DATA=1;
            TX_RDY=1;
            start=0;                            // Transmission is over
            data_bit=0;
        end
        default data_bit=0;
    endcase
end
end
endmodule

```

3) Serial System Controller

```

module Serial_System_Controller (TX_RDY, KP_VALID, CLK, TX_TXMIT);

input TX_RDY, KP_VALID, CLK;
output TX_TXMIT;

reg TX_TXMIT;
reg state;

always @ (posedge CLK) begin
    case(state)
        // State 0 waits for a valid key press
        // and for the transmitter to be ready
        // before activating transmission
        0:begin
            TX_TXMIT=0;
            if (KP_VALID && TX_RDY) begin
                TX_TXMIT=1;
                state=1;
            end
        end
        // State 1 makes sure that the key is
        // un-pressed before allowing
        // it to be transmitted again
        1:begin
            if(!TX_RDY) TX_TXMIT=0;
            if(!KP_VALID) state=0;
        end
        default begin
            TX_TXMIT=0;
            state=0;
        end
    endcase
end
endmodule

```

Figure 2: Altera's Quartus II Circuit Design – All Components

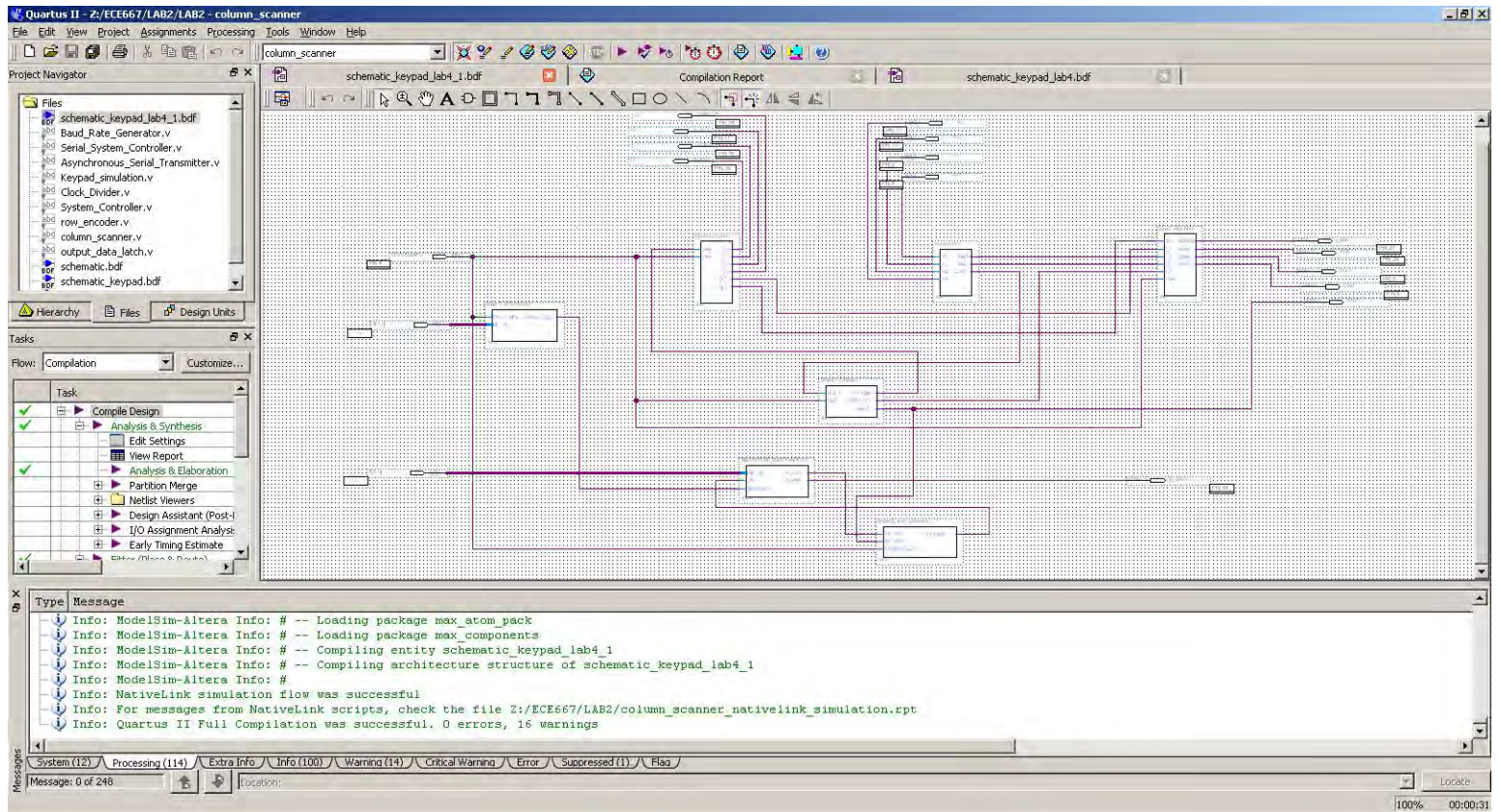


Figure 3: Pin Plan Layout

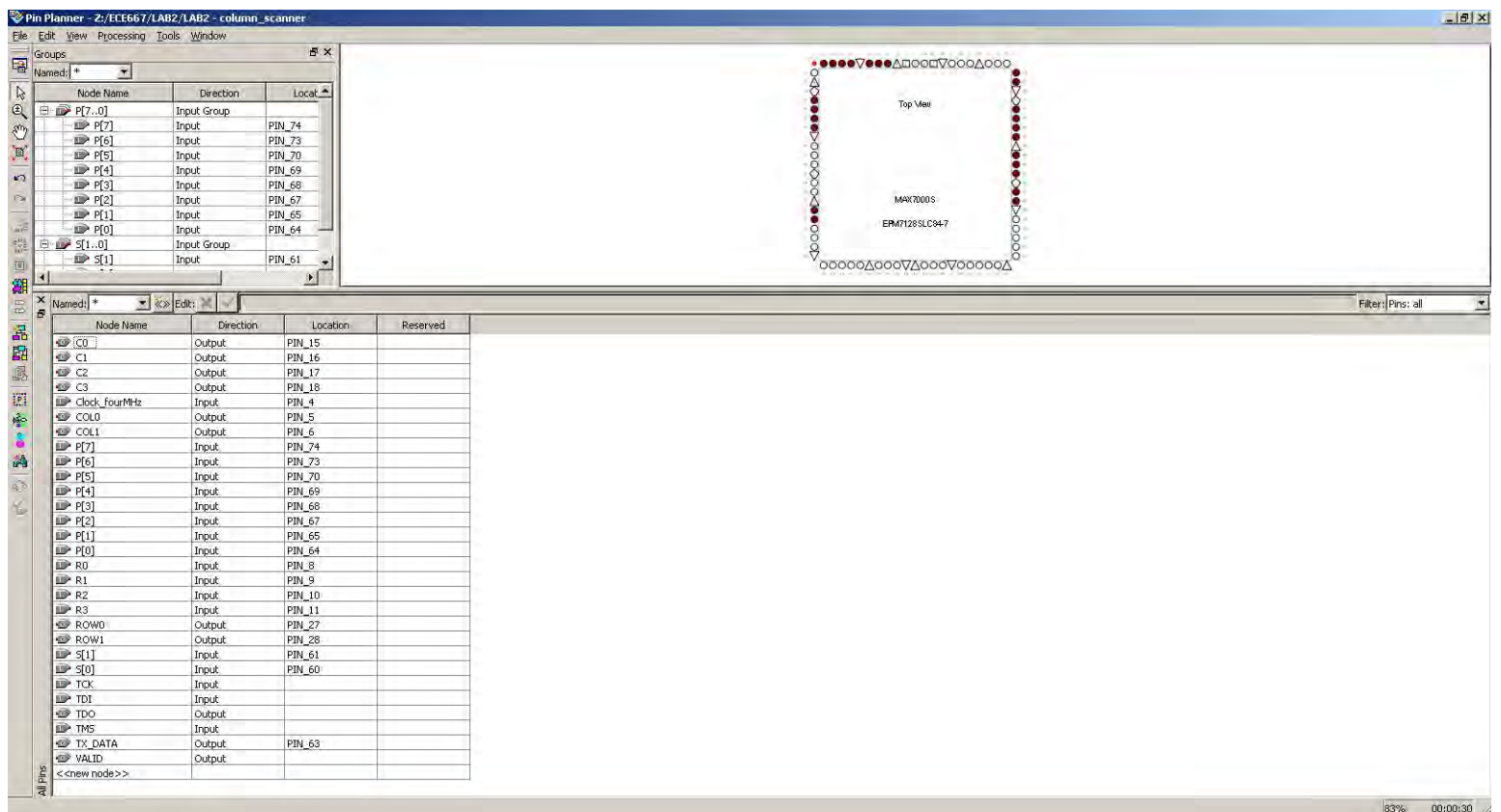


Figure 4: Baud Rate Generator – All Calculations

Baud Rate Generator Error Calculations

Rate Select	Bit Rate (bits/s)	Desired Clock Freq (Hz)	1st Measured Pulse (sec)	2nd Measured Pulse (sec)	Calculated Freq (Hz) = $1/(2nd-1st)$	Error (%) = $(Desired-Calculated)/Desired$	Accuracy (%) = $Calculated/Desired$
0 (00)	1200	1200	0.0004170	0.0012505	1199.7600480	0.02000%	99.98000%
1 (01)	2400	2400	0.0002085	0.0006250	2400.9603842	-0.04002%	100.04002%
2 (10)	4800	4800	0.0001045	0.0003130	4796.1630695	0.07994%	99.92006%
3 (11)	9600	9600	0.00005225	0.00015625	9615.38461538	-0.16026%	100.16026%

Baud Rate Generator Calculations for "case select S":

//selectable clock frequencies			External_Clock (Hz)	Desired_Clock Freq (Hz)	Factor (Clock Divider)	Calculated CLK_FREQ = $External_Clock/Desired_Clock/Factor$
0: CLK_FREQ=1667;	//1200Hz	----->	4000000	1200	2	1666.666667
1: CLK_FREQ=833;	//2400Hz	----->	4000000	2400	2	833.3333333
2: CLK_FREQ=417;	//4800Hz	----->	4000000	4800	2	416.6666667
3: CLK_FREQ=208;	//9600Hz	----->	4000000	9600	2	208.3333333

Figure 5: Baud Rate Generator – Waveform ‘00’

Baud Rate Generator Waveform for Rate Select = ‘00’

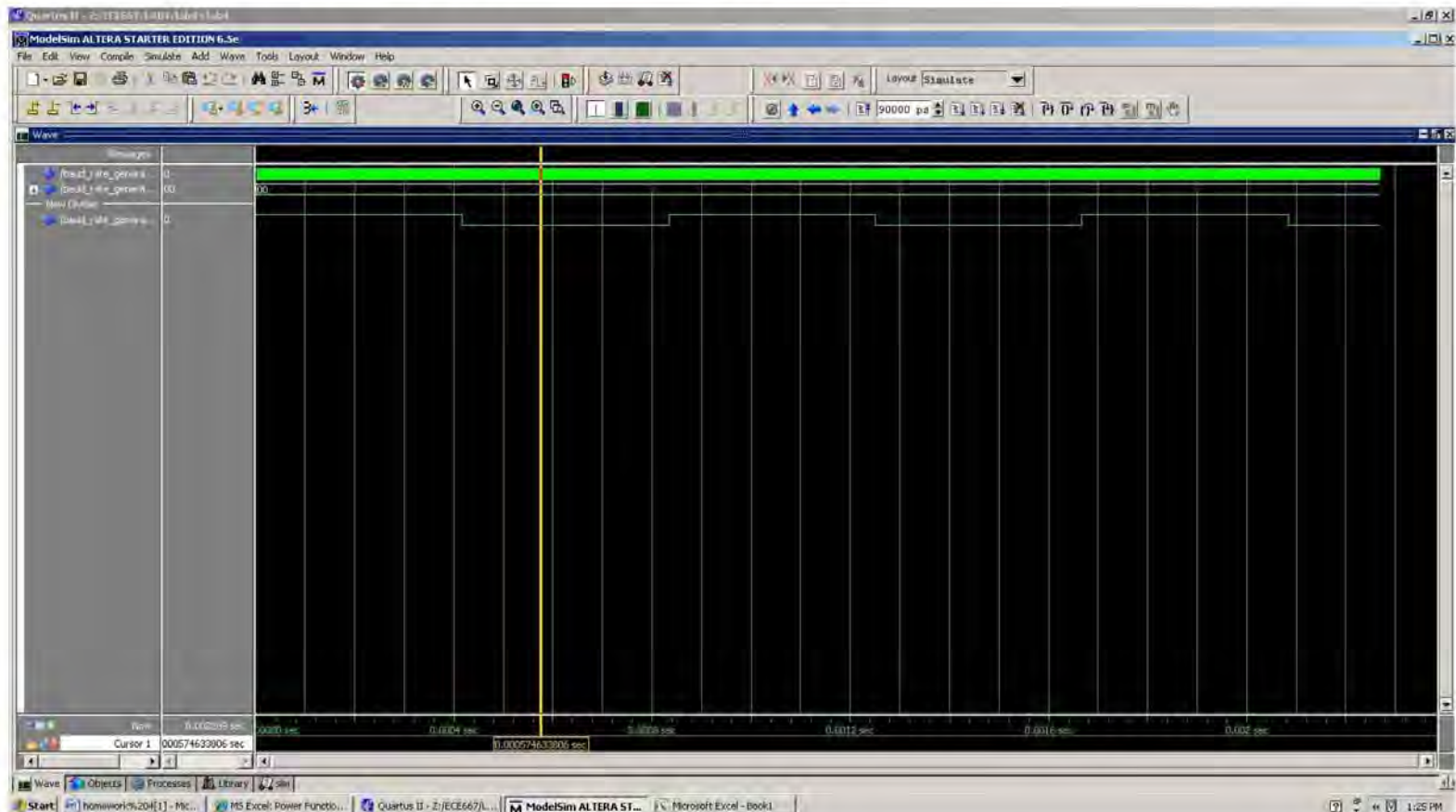


Figure 6: Baud Rate Generator – Waveform ‘01’

Baud Rate Generator Waveform for Rate Select = ‘01’

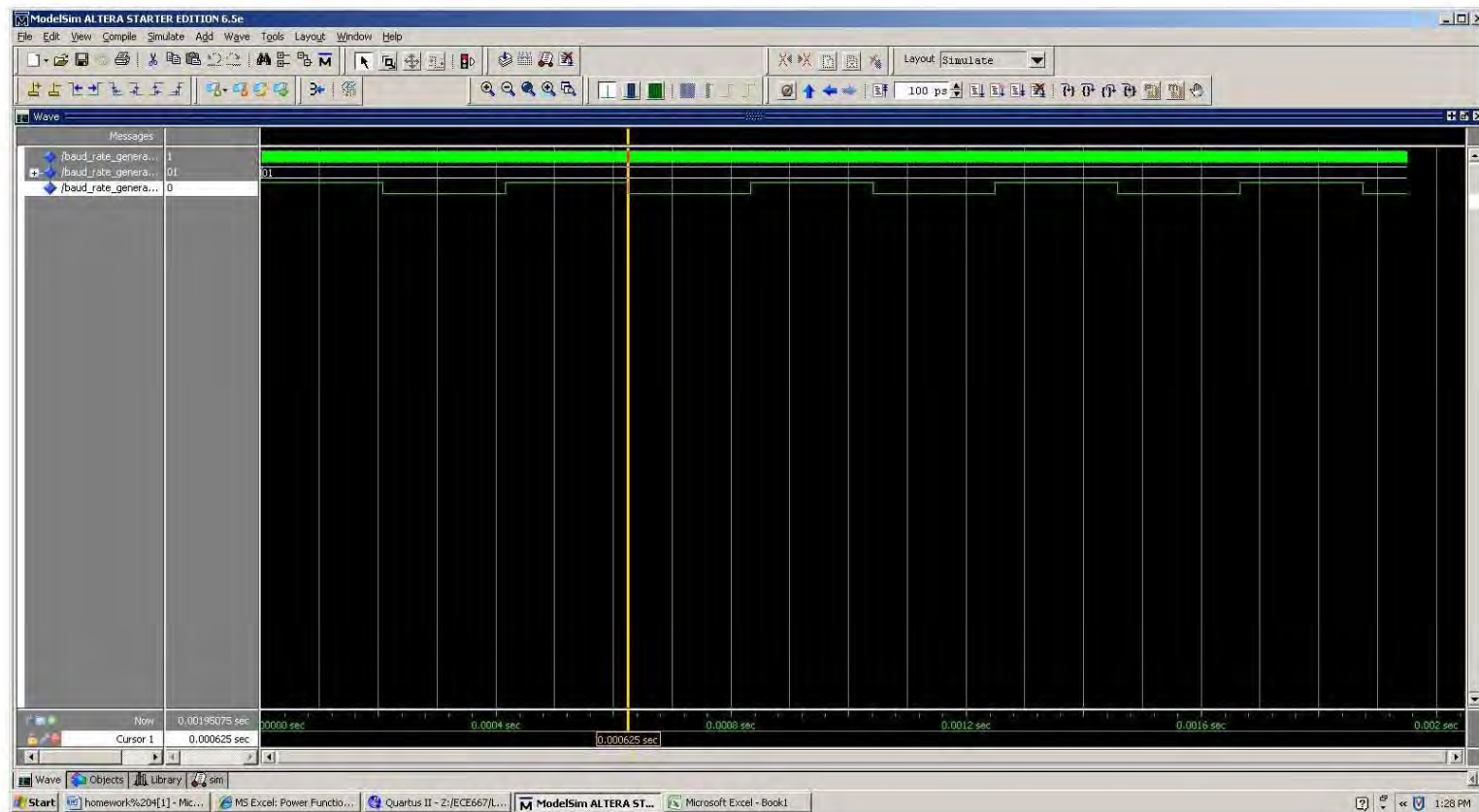


Figure 7: Baud Rate Generator – Waveform ‘10’

Baud Rate Generator Waveform for Rate Select = ‘10’

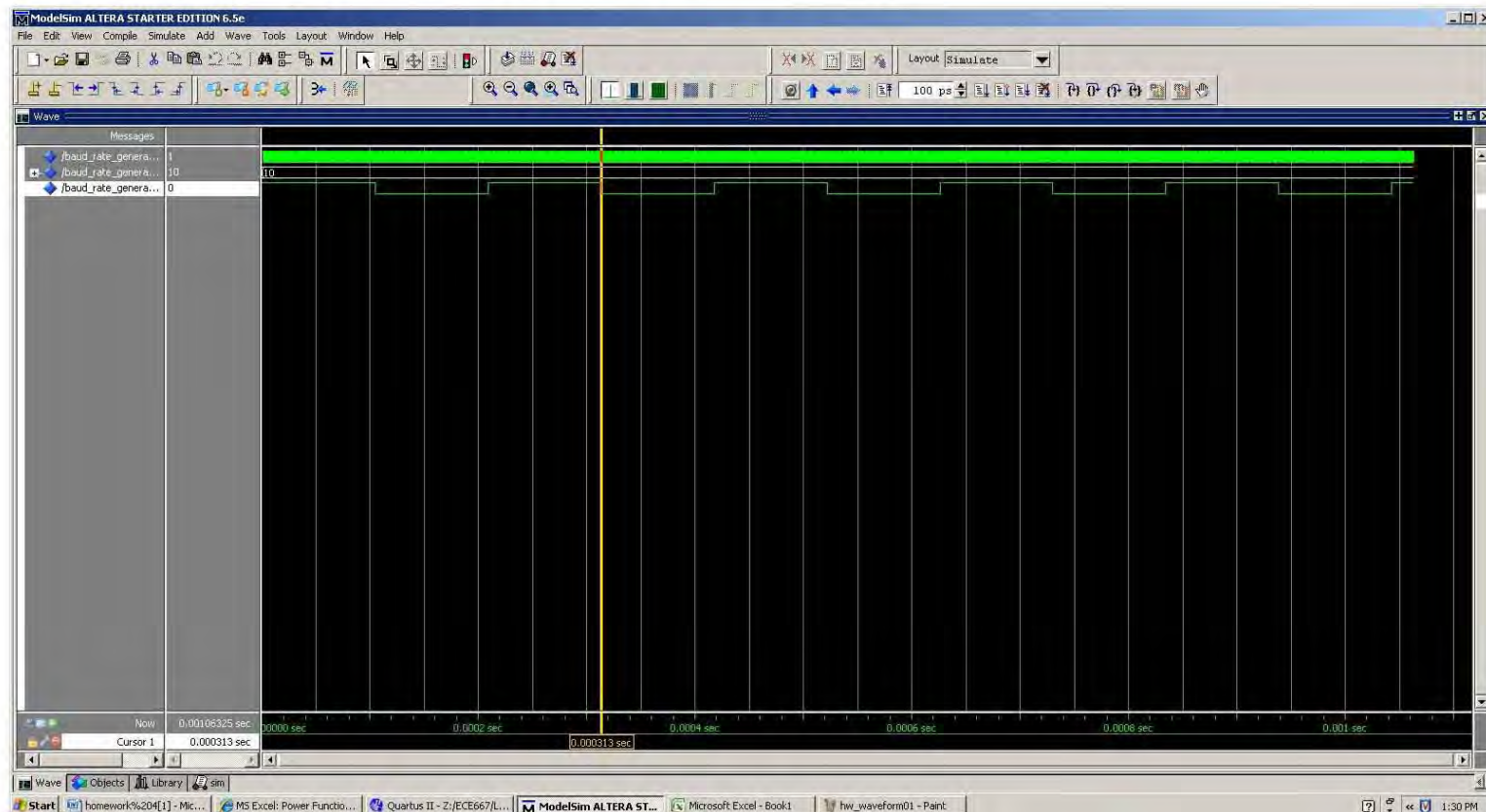


Figure 8: Baud Rate Generator – Waveform ‘11’

Baud Rate Generator Waveform for Rate Select = ‘11’

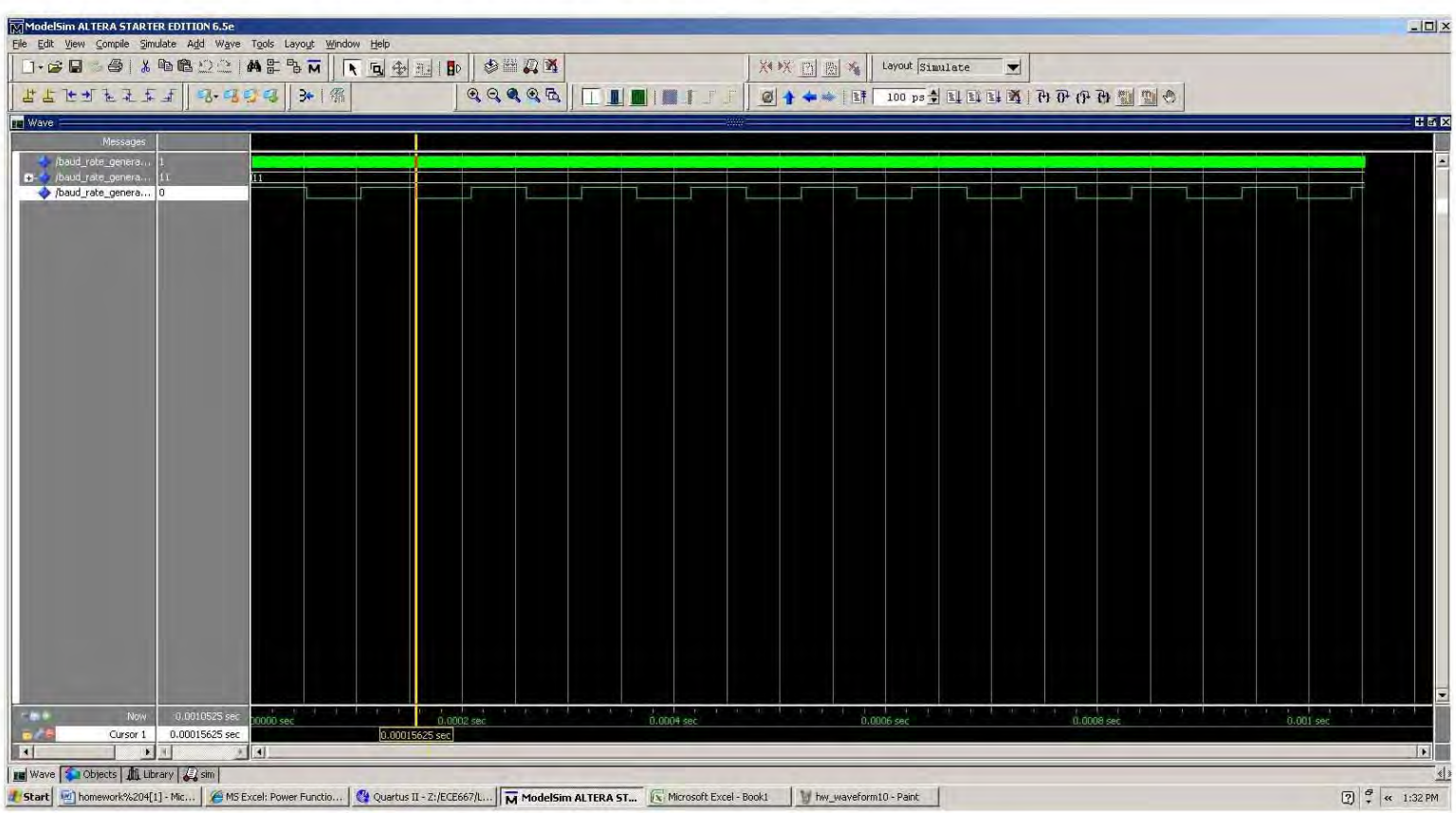


Figure 9: Asynchronous Serial Transmitter – Waveform 1

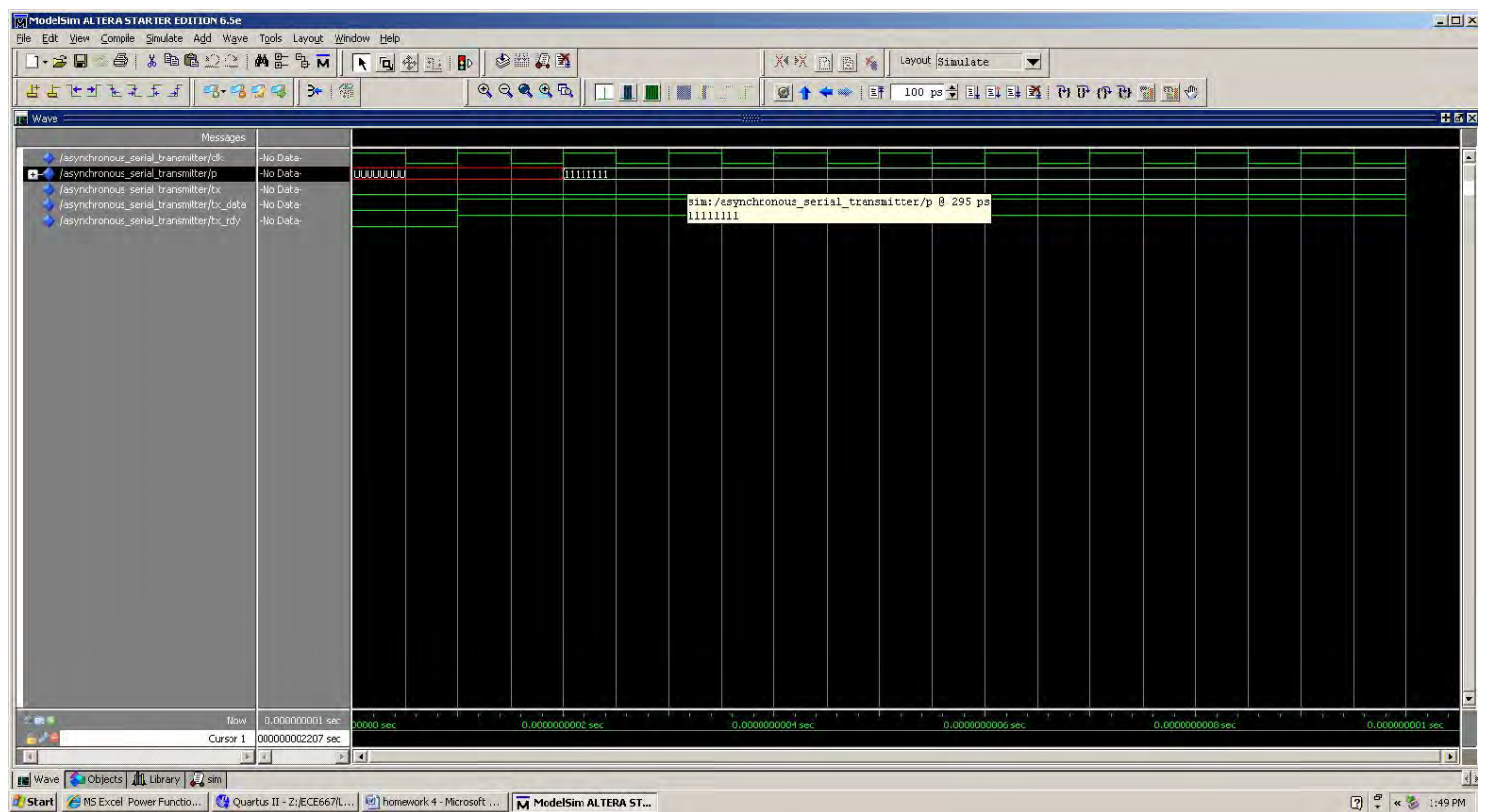


Figure 10: Asynchronous Serial Transmitter – Waveform 2

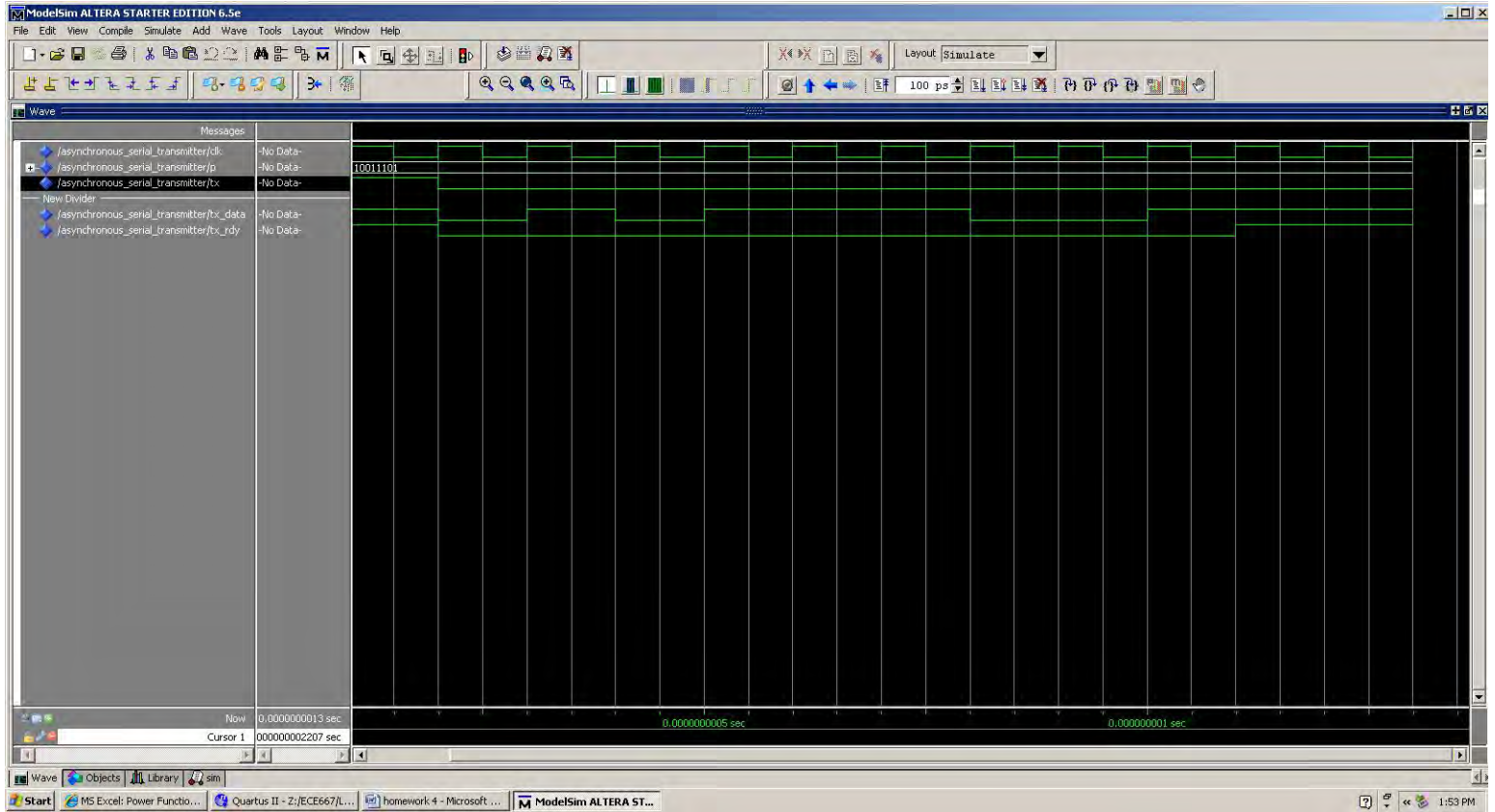


Figure 11: Serial System Controller – Waveform 1

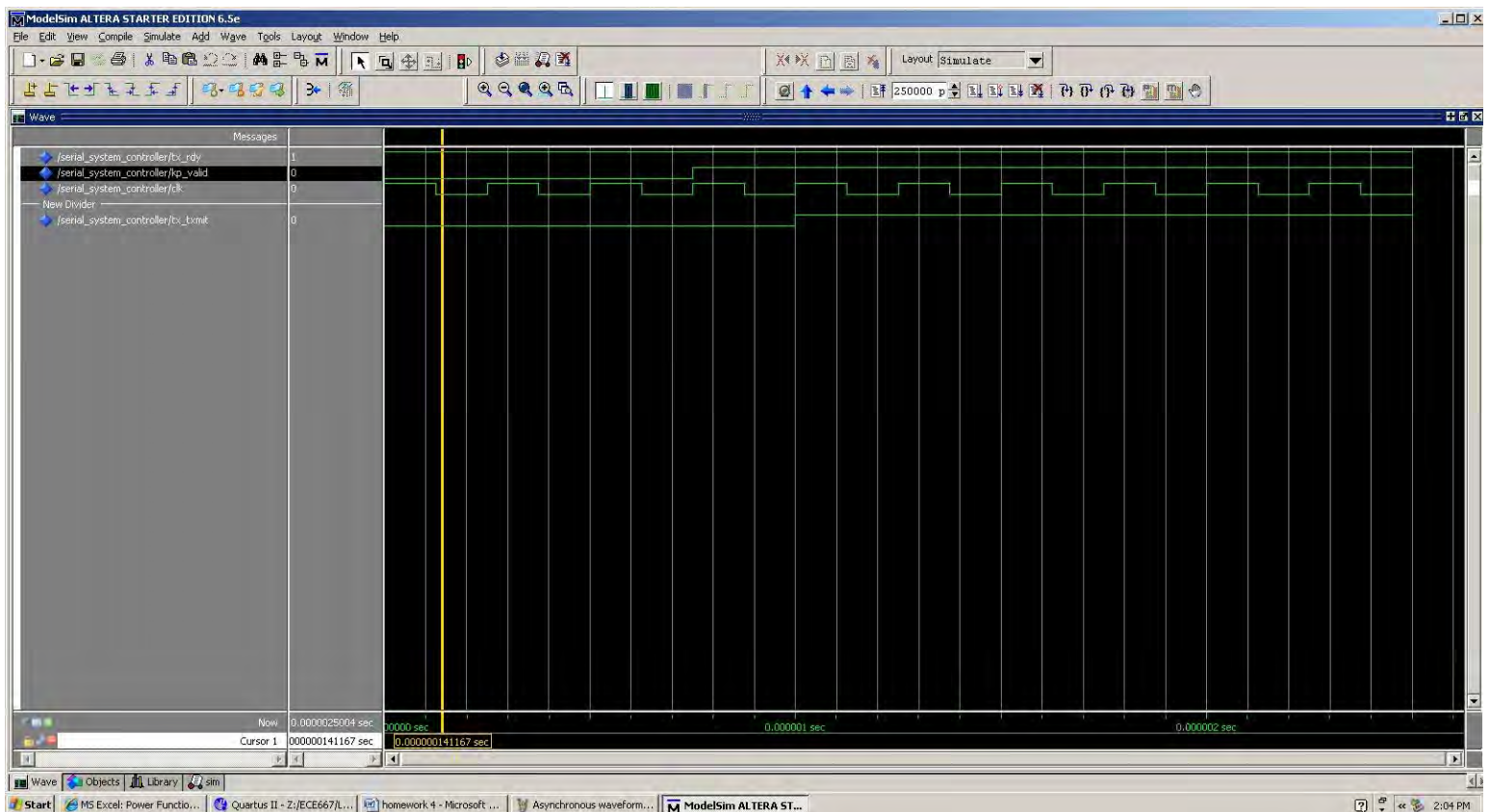
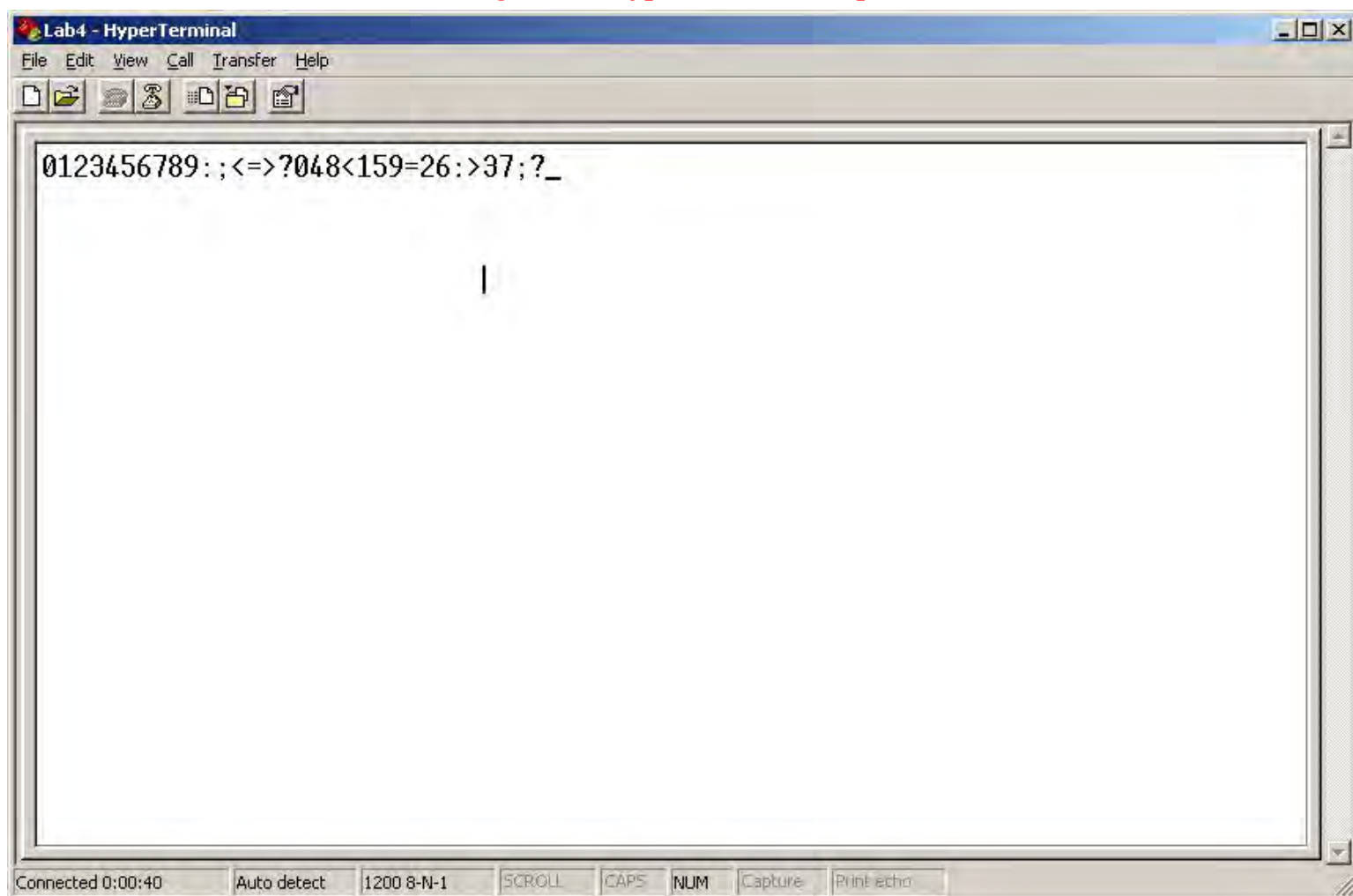


Figure 12: Hyper-terminal Output



ECE667

Homework #4

By:
Larissa Borcz
Sean Ewing
Feiran Lei
Ji Tu

1) Baud Rate Generator:

```
module Baud_Rate_Generator (FOUR_MHZ, S, BAUD_CLK);

input FOUR_MHZ;
input [1:0] S;
output BAUD_CLK;

reg BAUD_CLK;
reg [11:0] counter;
reg state;
reg [11:0] CLK_FREQ;
always @ (posedge FOUR_MHZ) begin
    case (S)                                // Selectable clock frequencies
        0: CLK_FREQ=1667;                  // 1200Hz
        1: CLK_FREQ=833;                   // 2400Hz
        2: CLK_FREQ=417;                   // 4800Hz
        3: CLK_FREQ=208;                   // 9600Hz
    endcase
    case (state)
        0: begin                            // This state gives a high clock signal
            BAUD_CLK=1;
            counter=counter+1;
            if (counter==CLK_FREQ) begin    // Keeps clock in this state for a selected time span
                state=1;
                counter=0;
            end
        end
        1: begin                            // This state gives a low clock signal
            BAUD_CLK=0;
            counter=counter+1;
            if (counter==CLK_FREQ) begin    // Keeps the clock in this state for a selected time span
                state=0;
                counter=0;
            end
        end
        default: state=0;
    endcase
end
endmodule
```

2) Asynchronous Serial Transmitter

```
module Asynchronous_Serial_Transmitter (P, TX, CLK, TX_RDY, TX_DATA);

input [7:0] P;
input TX, CLK;
output TX_RDY;
output TX_DATA;
```

```

reg TX_RDY;
reg TX_DATA;
reg [3:0] data_bit;
reg [7:0] locked_P;
reg start;

always @ (posedge CLK) begin

    if(!TX && !start) begin
        TX_DATA=1;
        TX_RDY=1;
    end
    if(TX) start=1; // Start is held high till transmission is fully executed
    if(start) begin
        case(data_bit)
            0:begin // Start bit
                TX_DATA=0;
                TX_RDY=0;
                locked_P=P;
                data_bit=1;
            end
            1:begin
                TX_DATA=locked_P[0]; // bit 0
                data_bit=2;
            end
            2:begin
                TX_DATA=locked_P[1]; // bit 1
                data_bit=3;
            end
            3:begin
                TX_DATA=locked_P[2]; // bit 2
                data_bit=4;
            end
            4:begin
                TX_DATA=locked_P[3]; // bit 3
                data_bit=5;
            end
            5:begin
                TX_DATA=locked_P[4]; // bit 4
                data_bit=6;
            end
            6:begin
                TX_DATA=locked_P[5]; // bit 5
                data_bit=7;
            end
            7:begin
                TX_DATA=locked_P[6]; // bit 6
                data_bit=8;
            end
            8:begin
                TX_DATA=locked_P[7]; // bit 7
                data_bit=9;
            end
        endcase
    end
end

```

```

        end
        9:begin                                // stop bit
            TX_DATA=1;
            TX_RDY=1;
            start=0;                            // Transmission is over
            data_bit=0;
        end
        default data_bit=0;
    endcase
end
end
endmodule

```

3) Serial System Controller

```

module Serial_System_Controller (TX_RDY, KP_VALID, CLK, TX_TXMIT);

input TX_RDY, KP_VALID, CLK;
output TX_TXMIT;

reg TX_TXMIT;
reg state;

always @ (posedge CLK) begin
    case(state)
        // State 0 waits for a valid key press
        // and for the transmitter to be ready
        // before activating transmission
        0:begin
            TX_TXMIT=0;
            if (KP_VALID && TX_RDY) begin
                TX_TXMIT=1;
                state=1;
            end
        end
        // State 1 makes sure that the key is
        // un-pressed before allowing
        // it to be transmitted again
        1:begin
            if(!TX_RDY) TX_TXMIT=0;
            if(!KP_VALID) state=0;
        end
        default begin
            TX_TXMIT=0;
            state=0;
        end
    endcase
end
endmodule

```

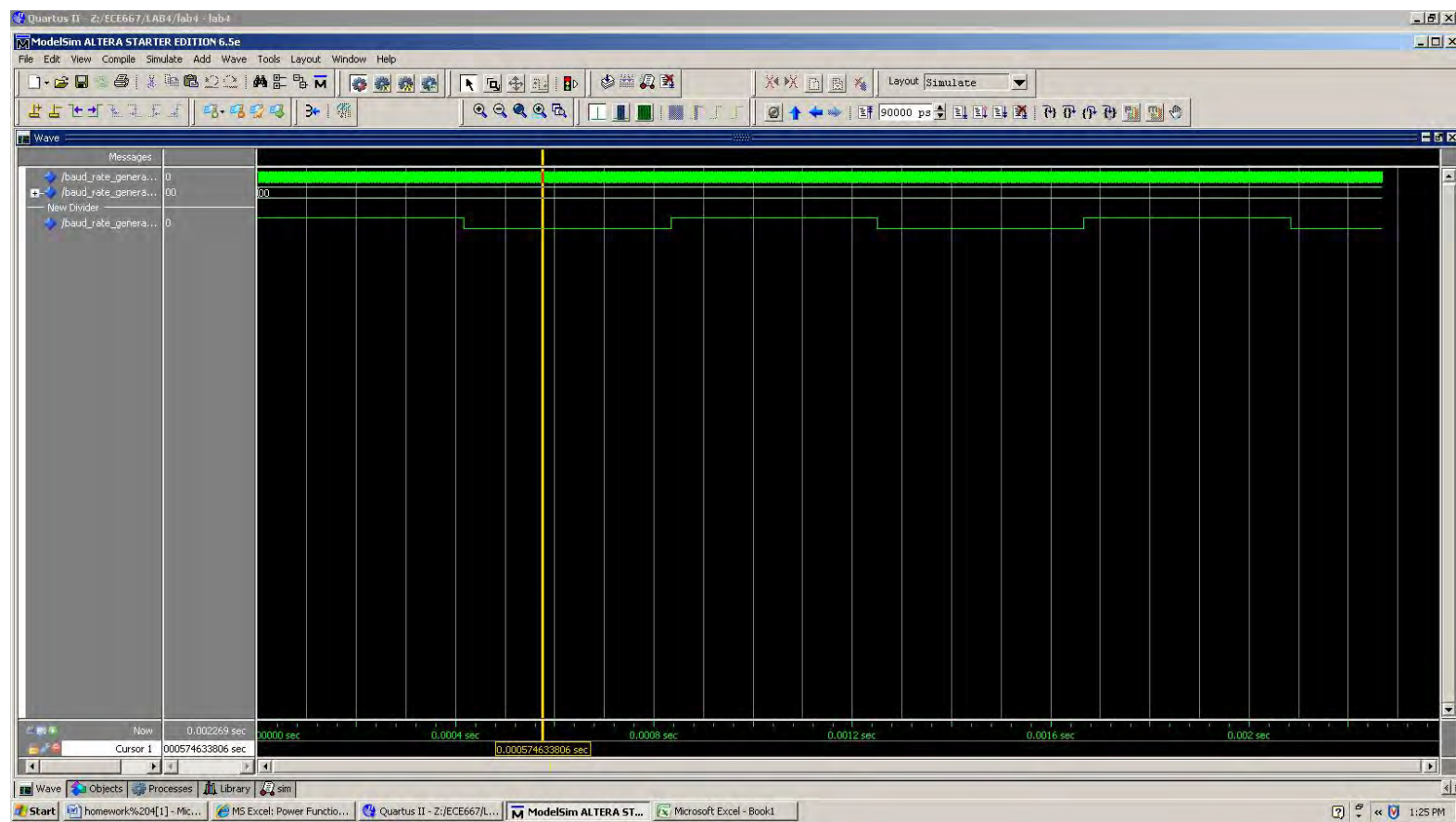
Baud Rate Generator Error Calculations

Rate Select	Bit Rate (bits/s)	Desired Clock Freq (Hz)	1st Measured Pulse (sec)	2nd Measured Pulse (sec)	Calculated Freq (Hz) = 1/(2nd-1st)	Error (%) = (Desired-Calculated)/Desired	Accuracy (%) = Calculated/Desired
0 (00)	1200	1200	0.0004170	0.0012505	1199.7600480	0.02000%	99.98000%
1 (01)	2400	2400	0.0002085	0.0006250	2400.9603842	-0.04002%	100.04002%
2 (10)	4800	4800	0.0001045	0.0003130	4796.1630695	0.07994%	99.92006%
3 (11)	9600	9600	0.00005225	0.00015625	9615.38461538	-0.16026%	100.16026%

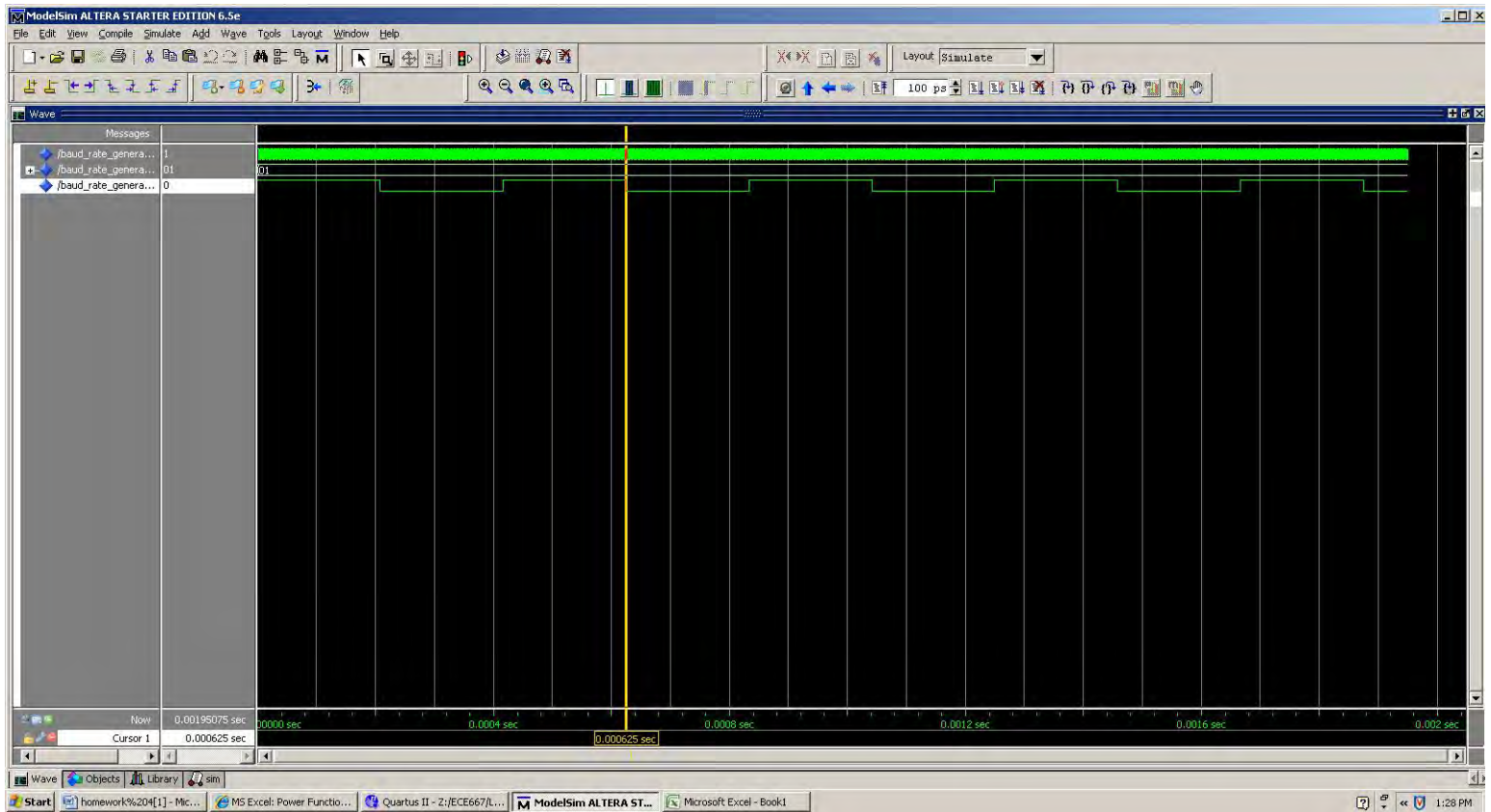
Baud Rate Generator Calculations for "case select S":

//selectable clock frequencies			External_Clock (Hz)	Desired_Clock Freq (Hz)	Factor (Clock Divider)	Calculated CLK_FREQ = External_Clock/Desired_Clock/Factor
0: CLK_FREQ=1667;	//1200Hz	----->	4000000	1200	2	1666.666667
1: CLK_FREQ=833;	//2400Hz	----->	4000000	2400	2	833.3333333
2: CLK_FREQ=417;	//4800Hz	----->	4000000	4800	2	416.6666667
3: CLK_FREQ=208;	//9600Hz	----->	4000000	9600	2	208.3333333

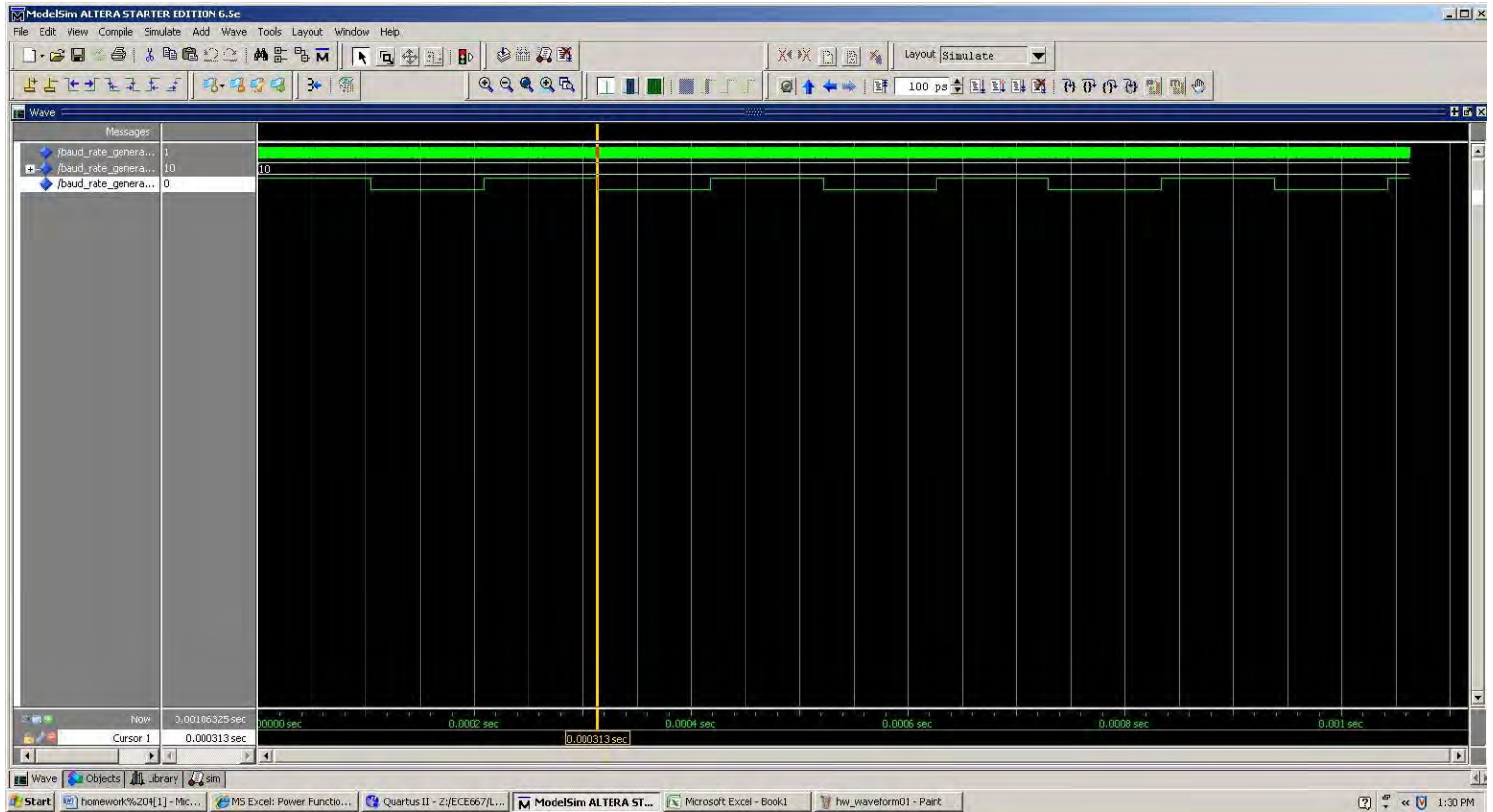
Baud Rate Generator Waveform for Rate Select = ‘00’



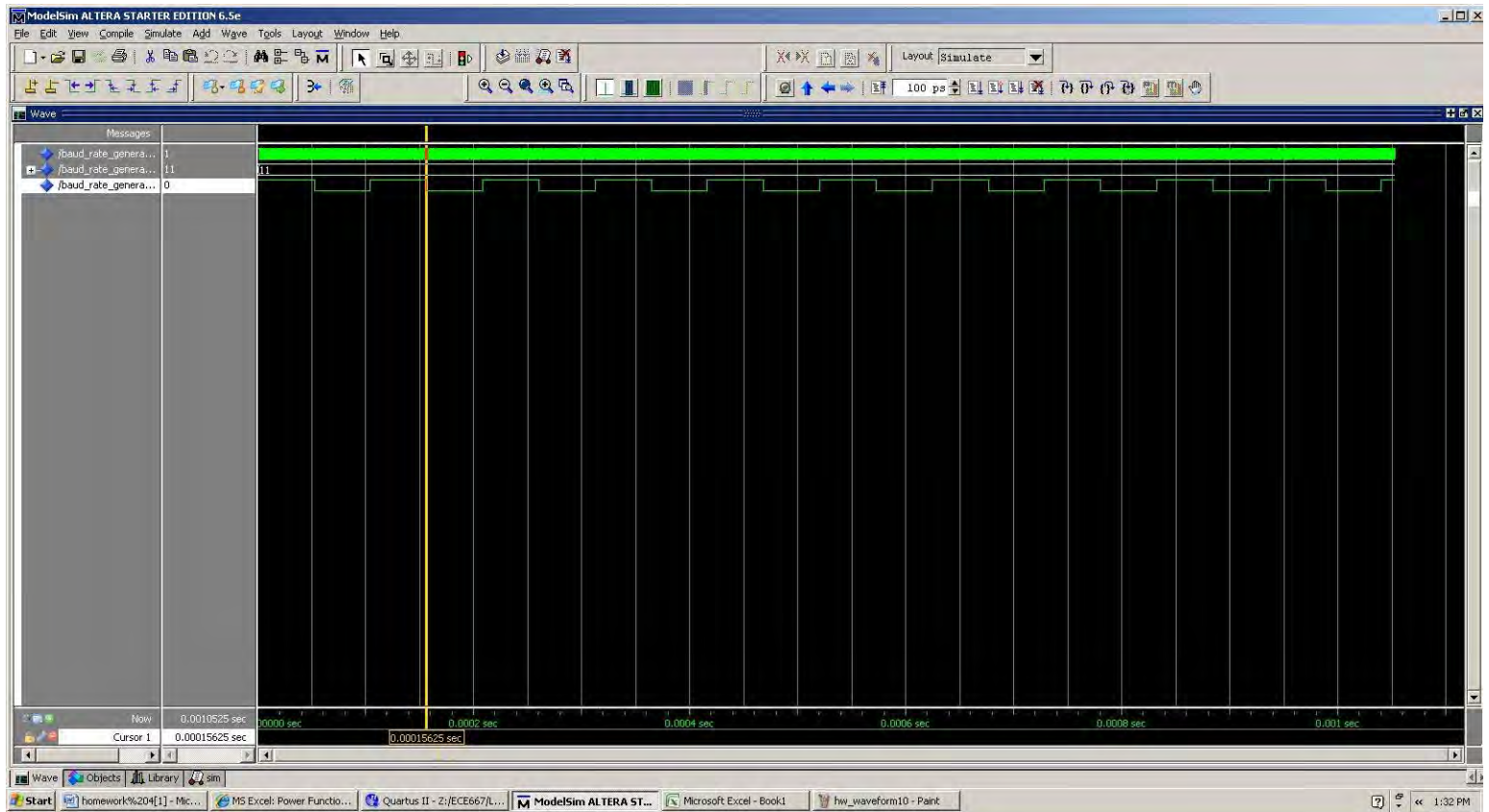
Baud Rate Generator Waveform for Rate Select = '01'



Baud Rate Generator Waveform for Rate Select = '10'



Baud Rate Generator Waveform for Rate Select = '11'



ECE667

Lab Report #5

Lab Completed: 8/16/11

Due Date: 8/22/11

By:

Larissa Borcz

Sean Ewing

Feiran Lei

Ji Tu

<u>Table of Contents</u>	<u>Page #</u>
1) Abstract	2
2) Introduction	2
3) Discussion	3
a. Section #1: Discussion – Lab Set-up	3
i. 1) Initializing and Configuring the PIC	3
ii. 2) Initializing the LCD Display Module	4
iii. 3) Controlling the Display Module	5
b. Section #2: Discussion – Code Set-up	6
4) Implementation & Testing	10
5) Conclusion	10
6) Appendices	11

Abstract

This project aims at writing the necessary code to interface the PIC 16F874 microcontroller with a 16 character by 2 line (16x2) LCD display. The display controller (included on the PCB holding the actual LCD display) requires a specific initialization routine before it will accept data to be displayed. Once this procedure has been executed, display command functions (such as setting a character to display) can be executed by applying specific command codes to the display's controller.

The hardware inside the PIC must be configured to behave the way instructed, before it can interface with an external device. This configuration must be done before any of the functional code can be executed within the PIC, and must be absolutely free of errors in order for the interface to the display work correctly. The code written accomplished three main tasks:

- 1) Initialize and configure the PIC (Internal)
- 2) Initialize the LCD Display Controller (External)
- 3) Send display data to the Display Controller (External).

The software package used for this assignment is MPLAB-IDE by Microchip Technology Inc. It is a free program and can be downloaded from Microchip's web-site: www.microchip.com (under Development Tools).

Introduction

The purpose of this lab is to connect the PIC microcontroller to the 16x2 LCD display, and program the PIC to display a simple message on the screen. Students will be programming the PIC with the modified code file they finished for homework #5. Once programmed, the PIC should automatically display the message you provide when it is powered up. Once completed, students will demonstrate its function to the instructor.

Discussion

Section #1: Discussion – Lab Set-up

The hardware inside the PIC must be configured to behave the way instructed, before it can interface with an external device. This configuration must be done before any of the functional code can be executed within the PIC, and must be absolutely free of errors in order for the interface to the display work correctly. The code written accomplished three main tasks: 1) Initialize and configure the PIC (Internal), 2) Initialize the LCD Display Controller (External), and 3) Send display data to the Display Controller (External).

1) Initializing and Configuring the PIC

To interface with the LCD display two of the available I/O ports on the PIC will be used: I/O Port D (pins RD7:RD0) to connect to DB(7:0) of the LCD display module in order to make the circuit easily testable. Additionally, I/O Port B pins RB2, RB1, and RB0 connect will connect to the remaining E, R/W, and RS pins of the display module, respectively.

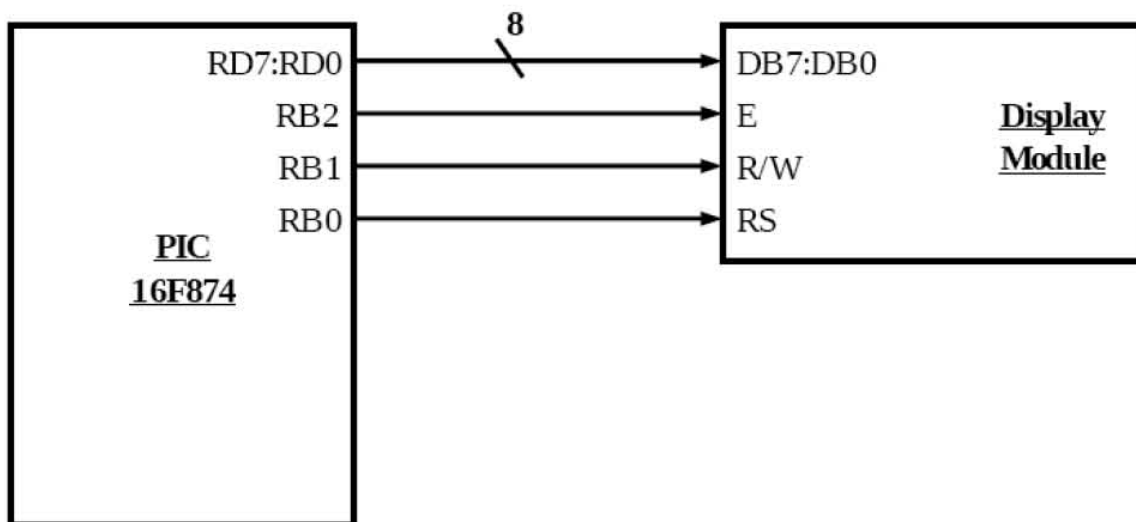


Figure #1: Interface Diagram

I/O Port Configuration:

The full 8 pins on Port D and the 3 pins on Port B will be configured as outputs to the display module. While the Display Controller does contain functions that allow

DB7:DB0 to provide data to the PIC (or any interfaced device), they will not be used so that the interface can be simplified.

Timer 0 Configuration:

During the initialization routine of the LCD display module, several delays will be needed between the applied initialization data and a timer to delay the rate at which characters are displayed; otherwise an overwritten character would change so fast that it would not be able to be seen. To accomplish these delays, a TIMER0 module will be used in the PIC. The TIMER0 module will be configured as a timer, as opposed to a counter, and will have the per-scalar assigned to it.

2) Initializing the LCD Display Module

Before the Display module can accept commands, the module's controller needs to be initialized. The data sheet for the display controller gives the sequence of initialization commands, as well as the binary code for these commands, that need to be applied to the controller at system startup, as well as the specified delays between each initialization command. Each of these commands will be issued at or after the specified delay length to ensure proper operation. The initialization routine is broken up into two sequences. The first of these is the RESET sequence for the LCD Display Controller, and essentially resets the controller to its default state. This sequence waits for 15 ms, and then applies the FUNCTION SET command to the display module. As shown in the initialization flow diagram for the display controller, a total of three (15ms delay – FUNCTION SET) sequences will be needed. Subroutine RESET_LCD accomplishes this with a loop, as shown in the code for this project. The second sequence will initialize the display parameters, and set the options that are used for this lab. Essentially a series of commands are applied to the Display Module to configure these options, similar to the configuration the Ports and Timer modules in the PIC. Since the maximum time it takes for any command on the LCD controller to execute is about 5 ms (from data sheet), our code will wait this long before applying each command. Subroutine INIT_LCD shows the PIC assembly code for the initialization routine.

Since a table lookup is used to get the sequence of commands, the PCLATH must be set to the proper value in the “System Initialization” section of the program, or the table lookup will not function properly.

3) Controlling the Display Module

The commands focused on are the Set Address (DD RAM) command (sets which LCD block to write to), and the Write Data (DD RAM) command (writes an ASCII character to the address specified by the Set Address command).

The Commands are specified as follows:

	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
Set Address	0	0	1	Λ	Λ	Λ	Λ	Λ	Λ	Λ	$\Lambda(6:0)$ address*
Write Data	1	0	D	D	D	D	D	D	D	D	D(7:0) – ASCII char. code

Figure #2: Write Data Commands

The addressing of the LCD locations start at 80h for the first eight characters, and then C0h for the last eight. See the LCD datasheet for more information. A table of 16 characters to be displayed will be created and entered as a part of the program, and will reside in program memory.

Our code will then be a loop of code to accomplish the following:

- (a) Set the address of the next character to write (or, the first address if this is the first time through) by applying the SET ADDRESS command to the LCD module.
- (b) Wait 5 ms for the instruction to execute.
- (c) Get the next character to be displayed from the character table.
- (d) Write the character data to the LCD module with the WRITE DATA command.
- (e) Wait about 1/2 of a second. (delay the next character to be displayed).
- (f) loop back to (a) until all 16 characters are displayed.

The end of the 16th loop should display all 16 characters from the table, displayed from left to right on the LCD display. The enable signal (E on the LCD module, or ENA in the code) from the

Initialization section must be used properly, as the commands will not be latched and executed if the enable signal is not handled properly.

All code will be tested thoroughly using the simulator in MPLAB, with GOTO instructions temporarily added to jump over the delay portions of the program, though this will only verify that the data is being manipulated properly. To test the timing of your delays, a breakpoint will be set at the end of the delay code, the Stopwatch window (Window→Stopwatch) opened, and then the simulator ran (Debug→Run→Run). When the program halts at the breakpoint, the stopwatch will display how long execution took.

Section #2: Discussion – Code Set-up

The code provided to us required that we modify the device configuration section to ensure that the correct registers in the microcontroller contained the correct values. The following assembly code was used to initialize the OPTION register:

```
MOVLW    B'11000111'
ANDWF    OPTION_REG      ;OPTION REGISTER (81H)
                        ; DEFAULT (RESET) = B'11111111'
                        ; <7> DO NOT MODIFY
                        ; <6> DO NOT MODIFY
                        ; * <5> TOCS - TMR0 SOURCE SELECT
                        ;     0=INTERNAL INSTRUCTION CLOCK
                        ;     1=TRANSITION ON RA4/T0CKI
                        ; * <4> TOSE - TMR0 SOURCE EDGE SELECT
                        ;     0=LOW-TO-HIGH ON RA4/T0CKI
                        ;     ;PRESCALAR = '1:256'      ;1=HIGH-TO-LOW ON RA4/T0CKI
                        ;     ; = 256 US PER CNT PULSE
; * <3> PSA - PRESCALAR ASSIGN
                        ;     0=TMR0, 1=WDT
; * <2..0> PS<2..0> - PRESCALAR (P.31)
```

The code above sets the timer 0 clock source for the internal clock, assigns a prescalar to timer 0, and sets the timer 0 prescalar to 256 (the maximum value). The following assembly code is used to configure the PORTB input/output status.

```
BSF    STATUS,RP0 ;BANK 1
MOVLW  B'11111000'
MOVWF  TRISB      ;SET PORT B DATA DIRECTION(1=IN,0=OUT)
```

The value B'11111000' is written to the TRISB register to tell the PIC that the PORTB pins 7-3 will be used as inputs, and the PORTB pins 2-0 will be used as outputs. Note that the bank pointer was switched by setting bits in the status register, since the TRISB register resides in memory bank 1.

The following assembly code is used to configure the PORTD register:

```
BSF    STATUS,RP0 ;BANK 1
MOVLW  B'00000000'
MOVWF  TRISD      ;SET PORT D DATA DIRECTION
```

The value B'00000000' is written to the TRISD register to tell the PIC that all of the PORTD pins will be used as outputs. Note that the bank pointer was switched by setting bits in the status register, since the TRISD register resides in memory bank 1. The flow chart in Figure #3 illustrates the requirements of the LCD display routine that was required:

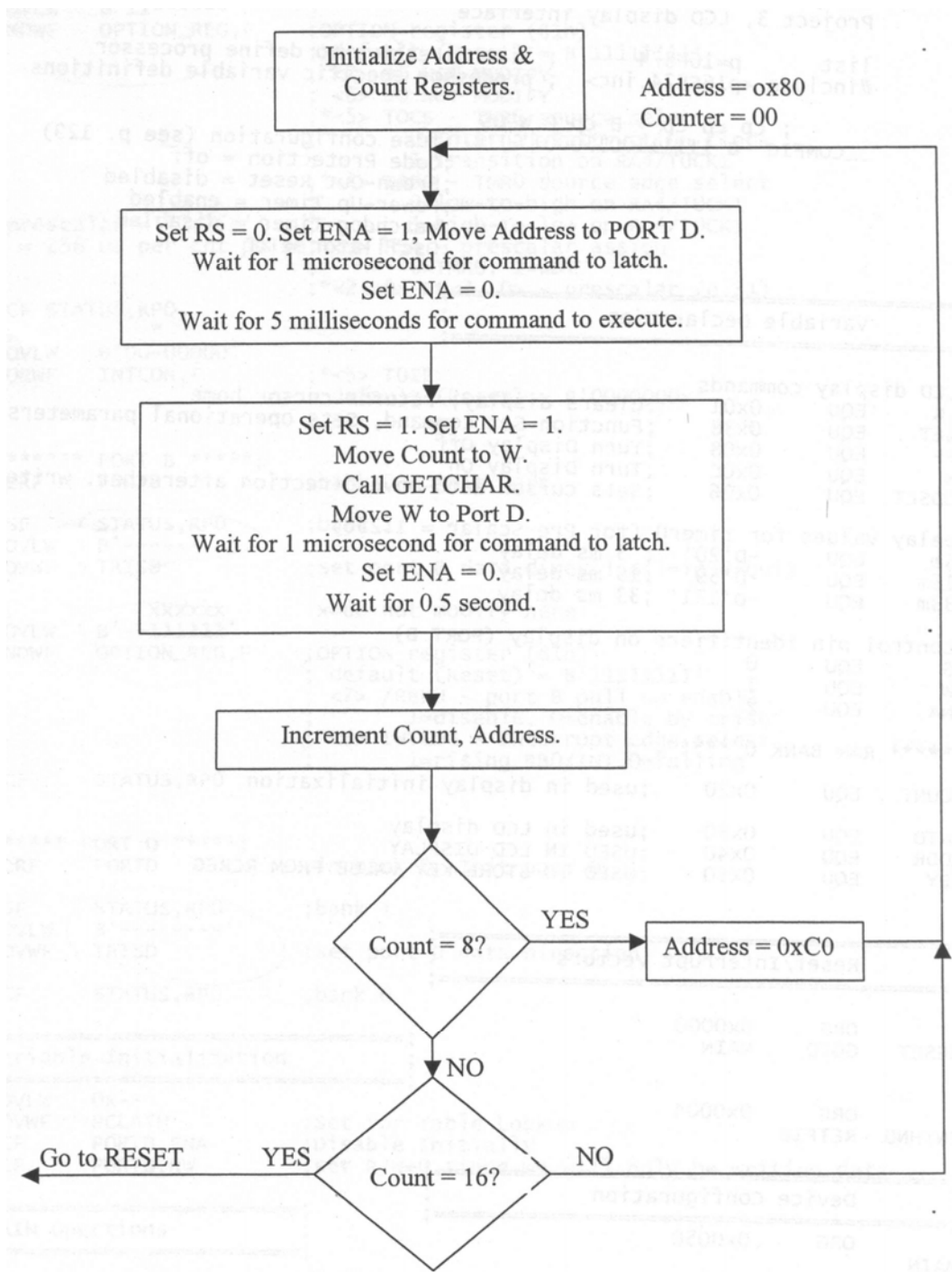


Figure 3 – LCD Display Flow Chart

The following assembly code was developed to accomplish this task:

===== DISPLAY ROUTINE START =====;

```

LCD_DISP          ;<----- Modify here -----
;--Block 1 from Flow Chart
CLRf STATUS

```

```

MOVLW    0x80        ; initialize address
MOVWF    ADDR
CLRF    CNTD        ; clear CNTD

```

;--Block 2 from Flow Chart

```

DRLOOP    BCF    PORTB,RS    ; set RS=0
          BSF    PORTB,ENA    ; set ENA = 1 (Enable Command Latch)
          MOVF    ADDR,W
          MOVWF    PORTD
          NOP                ; wait 1 instruction cycle (1 us) for command to latch
          BCF    PORTB,ENA    ; set ENA=0 (Disable Command Latch)
          MOVLW    D5m        ;
          CALL    DELAY        ; wait 5 ms

```

;--Block 3 from Flow Chart

```

          BSF    PORTB,RS    ; set RS=1
          BSF    PORTB,ENA    ; set ENA = 1 (Enable Command Latch)
          MOVF    CNTD,W        ; Move COUNT to W. Used as index in DISP_COMMAND
          CALL    GET_CHAR    ; get next init.command --> W (table lookup)
          MOVWF    PORTD        ; write command to Port D
          NOP                ; wait 1 us for command to latch
          BCF    PORTB,ENA    ; Set ENA=0
          CALL    WAIT500        ; Delay 500ms

```

;--Block 4 from Flow Chart

```

          INCF    CNTD        ; increment count
          INCF    ADDR        ; increment address

```

;--Block 5 and 6 from Flow Chart

```

EQUALS8    MOVLW    0x08
            SUBWF    CNTD,W
            BTFSS    STATUS,Z    ; Test if Count=8
            GOTO    EQUALS16    ; If NO, check if = 16
            MOVLW    0xC0        ; If Yes, Change Address Value
            MOVWF    ADDR

EQUALS16    MOVLW    0x10
            SUBWF    CNTD,W
            BTFSS    STATUS,Z    ; Test if Count=16
            GOTO    DRLOOP        ; If No, go to
            RETURN                ; If Yes, Return to RESET

```

```

; Here you must provide a full function for implementing the
; writing of characters to the LCD screen. This function must
; perform a single write of your full message to the screen. If
; needed, you are welcome to write additional subroutines. If you
; choose to do so, clearly document them to thier function if
; obvious

```

;-----

RETURN

;===== DISPLAY ROUTINE END =====;

Implementation & Testing

This lab utilized four main hardware components, the PIC16F87X microcontroller, a 16x2 LCD display, an oscillator and a USB programmer. The microcontroller was used to process the code, and it turn was wired to the LCD to display the output of the code, “Brutus was Here!”. The oscillator was wired to the microcontroller to simulate a 4Mhz clock. The USB programmer was wired from the PC to the PIC to assist in debugger. The USB programmer first simulated the code implemented from the PC to the microcontroller, before hard-coding to the PIC. Due to some difficulties our group had to bypass the USB programmer and hard-code directly to the PIC in order to debug. With the code found in the appendix implemented through the previously mentioned hardware configuration, our group was successfully able to achieve the desired result of a scrolling LCD screen displaying the message “Genius Was Here!”. The complete code for all components can be found in the Appendix, Figure #4.

Conclusion

The lab was completed with minimal errors. The biggest issues our team had was learning how MPLAB functions with the PIC chips. Our initial issues were errors within the code which we thought were due to programming problems. This problem was solved by selecting the correct PIC chip and then debugging the software again. Other issues included selecting the correct programmer and setting the “E” line on the LCD module correctly.

Appendices

Figure 4: Code – All Components

```

=====
;
;   Group: Ji Tu, Feiran Lei, Sean C. Ewing, Larissa M Borecz
;
=====
;
;   Lab 5: LCD display interface
;
=====
;
;   This code handles the configuration of the LCD display,
; subroutines for printing to, resetting, and clearing the
; LCD, as well as writing characters to it.
;
=====

=====
;
; Initialization of the environment
;
=====
;
;   list    p=16F874          ; list directive to define processor
;   #include <p16F874.inc>    ; processor specific variable definitions
;
;
;
;
=====
;
; Configuration of the PIC
;
=====
;
;   __CONFIG B'11111100110001';setting configuration word (see PIC16F87X Data Sheet, p. 120)
;
;   ||  | |||
;   ||  | || | ||ww===== Oscillator = XT
;
;   ||  | || |w===== Watchdog Timer = disabled
;
;   ||  | || |w===== Power-Up Timer = enabled
;
;   ||  | || |w===== Brown-Out Reset = disabled
;
;   ww===== Code Protection = off (also bits 5-4)
;
;
;
;
=====
;
;   Constant Declaration
;
=====
;
;LCD display commands
CLR      EQU    0x01          ;Clears display, returns cursor home
FSET     EQU    0x38          ;Function Set command, Sets operational parameters
OFF      EQU    0x08          ;Turn Display Off
ON       EQU    0x0C          ;Turn Display On
MODSET   EQU    0x06          ;Sets cursor auto-move direction after char. write

```

```

;Delay values for timer0 (for Pre-Scalar = 1:256)
D5m      EQU   -D'20'          ; 5 ms delay
D15m     EQU   -D'59'          ;15 ms delay
D33m     EQU   -D'131'         ;33 ms delay

```

```

;control pin identifiers on display (PORT B)
RS        EQU   0
RW        EQU   1
ENA       EQU   2

```

```

=====;
;      Variable Declaration      ;
=====;

```

```

;***** RAM BANK 0 *****;
COUNT    EQU   0x20          ; used in LCD display initialization
CNTD      EQU   0x30          ; used in LCD display
ADDR      EQU   0x40          ; used in LCD display
KEY       EQU   0x50          ; used to store key value from RCREG

```

```

=====;
;      Reset/Interrupt Vectors  ;
=====;
      ORG    0x0000          ; The PIC enters execution here
RESET GOTO  MAIN

      ORG    0x0004          ; The Interrupt vector enters here
INTEND     RETFIE

```

```

=====;
;      Device Configuration     ;
=====;
      ORG    0x0050
MAIN

```

```

;= Timer0 configuration =====;
BSF      STATUS,RP0    ; Set the PIC to use bank 1 registers (see page 18 & 14)
MOVLW    B'11000111'   ;<----- Modify here ----->
; This value will be 'AND'ed with the default value of the OPTION
; register (81h) of the PIC (see next line). The default value on
; reset is B'11111111'. For more information on this register see
; page 19 of the datasheet (on carmen)
;
; You must provide values to set:
; 1) the clock source for the timer

```

```

; 2) the clock edge to count
; 3) the prescalar assignment
; 4) the prescalar value
;
; Fill in bits for the '*' in the command. Do not modify bits 7
; and 6.
;-----
ANDWF    OPTION_REG,F    ; AND the value with OPTION_REG and store in OPTION_REG
BCF      STATUS,RP0      ; Set the PIC to use bank 0 registers (see page 18 & 14)
MOVLW    B'00100000'     ;<----- Modify here -----
; This value will be 'OR'ed with the default value of the INTCON
; register (see next line). The default value of this register on
; reset is B'0000000x'. For more information on this register see
; page 20 of the datasheet (on Carmen)
;
; You must provide a value to:
; 1) Enable the Overflow Interrupt for Timer 0
;
; Fill in bit 5. Do not modify any other bits.
;-----
IORWFINTCON,F    ; OR the value with INTCON and store in INTCON

```

```

;= Port B configuration =====;
CLRF     PORTB          ; Clears Port B Data Latches
BSF      STATUS,RP0     ; Set the PIC to use bank 1 registers (see page 18 & 14)
MOVLW    B'11111000'    ;<----- Modify here -----
; This value will be moved into the PortB data direction register
; (TRISB). Provide a literal value for all the '*' in the command
; to set Port B data direction properly. For more information on
; PortB and TRISB, see page 31 of the datasheet
;
; You must provide values to:
; 1) Set all 8 bits of Port B to the proper data direction
;
; Fill in all 8 bits.
;-----
MOVWF    TRISB          ; Store the value in TRISB
MOVLW    B'00111111'    ;<----- Modify here -----
; This value will be 'AND'ed with the current value of OPTION_REG
; and saved into the OPTION_REG. See page 19 of the datasheet for
; more information
;
; You must provide values to:
; 1) enable/disable PORTB pull-ups
; 2) Set clock edge for interrupt on RB0
;
; Fill in bits 7 and 6. Do not modify any other bits
;-----

```

```

ANDWF    OPTION_REG,F      ; AND the value with OPTION_REG and store in OPTION_REG
BCF      STATUS,RP0        ; Set the PIC to use bank 0 registers (see page 18 & 14)

```

```

;= Port D configuration =====;
CLRF     PORTD              ; Clears Port D Data Latches
BSF      STATUS,RP0         ; Set the PIC to use bank 1 registers (see page 18 & 14)
MOVLW    B'00000000'        ;<----- Modify here -----
                                ; This value will be moved into the PortD data direction register
                                ; (TRISD). Provide a literal value for all the '*' in the command
                                ; to set Port D data direction properly. For more information on
                                ; PortD and TRISD, see page 31 of the datasheet
                                ;
                                ; You must provide values to:
                                ; 1) Set all 8 bits of Port D to the proper data direction
                                ;
                                ; Fill in all 8 bits.
                                ;-----
MOVWF     TRISD              ; Move the value to TRISD
BCF      STATUS,RP0         ; Set the PIC to use bank 0 registers (see page 18 & 14)

```

```

;=====;
;      System Initialization      ;
;=====;

```

```

BCF      PORTB,ENA          ; Initially, we disable the LCD
BCF      PORTB,RW           ; Also, R/W will always be 0, as we will only be writing data

MOVLW    0x05               ;<----- Modify here -----
                                ; This value will be loaded into the PCLATH register (see next line
                                ; of code). We want to setup the PCLATH register so that when
                                ; instructions later on use the PCL as a destination, the correct
                                ; result occurs. Specifically, the GET_CHAR and DISP_COMMAND
                                ; routines make use of such instructions. See page 26 of the
                                ; PIC manual for more information
                                ;
                                ; You must provide values to:
                                ; 1) Set the PCLATH register with the appropriate value to point
                                ;    to the table functions found at the end of the code
                                ;
                                ; Fill in both HEX values
                                ;-----
MOVWF     PCLATH             ; Move the value to PCLATH

```

```

=====;
;      MAIN Operations                      ;
=====;

```

```

CALL RESET_LCD ; Call subroutine to reset the LCD

CALL INIT_LCD  ; Call subroutine to initialize the LCD

CALL LCD_DISP  ; Call subroutine to display text on the LCD

GOTO RESET     ; return to the start of the code

```

```

=====;
;      Functions                          ;
=====;

```

```

RESET_LCD ; resets LCD display -----
; parameters:
;      none

```

```

MOVLW    0x03      ;
MOVWF    COUNT     ; set COUNT = 3

```

```

RLOOP    BCF    PORTB,RS ; set RS=0

```

```

MOVLW    D15m      ;
CALL     DELAY      ; wait 15 ms

```

```

BCF    PORTB,RW ; set R/W = 0

```

```

BSF    PORTB,ENA ; set ENA=1 (Enable Command Latch)

```

```

MOVLW    FSET      ;
MOVWF    PORTD     ;write Function Set command to Port D

```

```

NOP                      ; wait 1 instruction cycle (1 us) for command to latch

```

```

BCF    PORTB,ENA ; set ENA = 0 (Disable Command Latch)

```

```

DECFSZ    COUNT,F      ; Check to see if this has been done 3 times.
GOTO RLOOP              ; if not, do it again
RETURN      ; if so, then LCD controller reset

```

```

-----;

```

```

INIT_LCD    ; initializes LCD display settings/options -----
            ; parameters:
            ;     none

            CLR    COUNT                ; clear COUNT

NEXTCOM     MOVLW    D5m                ;
            CALL    DELAY                ; wait 5 ms

            BSF     PORTB,ENA            ; set ENA = 1 (Enable Command Latch)

            MOVF    COUNT,W              ; Move COUNT to W. Used as index in DISP_COMMAND

            CALL    DISP_COMMAND          ; get next init. command --> W (table lookup)

            MOVWF   PORTD                ; write command to Port D

            NOP                          ; wait 1 us for command to latch

            BCF     PORTB,ENA            ; set ENA=0 (Disable Command Latch)

            INCF    COUNT,F              ; increment command index for table

            MOVLW   0x05                 ;
            SUBWF   COUNT,W              ; Subtract 5 from COUNT and see if it's a 0
            BTFSS   STATUS,Z             ;

            GOTO    NEXTCOM              ; if not, do the next

            MOVLW   D5m                 ;
            CALL    DELAY                ; if so, wait an additional 5 ms

            RETURN                        ; and now Display is ready for operation
;-----

```

```

DELAY              ; delay routine -----
            ; parameters:
            ;     W      :      Moved to TMR0. Routine then waits for overflow

            BCF     INTCON,T0IF          ; Clear Timer0 Interrupt Flag

            MOVWF   TMR0                ; Move W --> TMR0 delay register

WAIT    BTFSS    INTCON,T0IF            ; has the delay period passed?

            GOTO    WAIT                ; no - keep waiting

```

```

RETURN                                ; yes - proceed
;-----

WAIT500                               ; 1/2 second delay routine -----
    ; parameters:
    ; none

    MOVLW    D'15'
    MOVWF    COUNT                    ; set COUNT to 15

HOLDUP    MOVLW    D33m                ;
    CALL    DELAY                    ; wait for 33 ms

    DECFSZ    COUNT,F                ; check to see delay has been called 15 times

    GOTO    HOLDUP                    ; if not, delay again

    RETURN                            ; if so, return
;-----

```

```

;===== DISPLAY ROUTINE START =====;

```

```

LCD_DISP                                ;<----- Modify here -----
;--Block 1 from Flow Chart
    CLRF    STATUS
    MOVLW    0x80                    ; initialize address
    MOVWF    ADDR
    CLRF    CNTD                    ; clear CNTD

;--Block 2 from Flow Chart
DRLOOP    BCF    PORTB,RS            ; set RS=0
    BSF      PORTB,ENA                ; set ENA = 1 (Enable Command Latch)
    MOVF    ADDR,W
    MOVWF    PORTD
    NOP                                ; wait 1 instruction cycle (1 us) for command to latch
    BCF      PORTB,ENA                ; set ENA=0 (Disable Command Latch)
    MOVLW    D5m                      ;
    CALL    DELAY                    ; wait 5 ms

```

;--Block 3 from Flow Chart

```
BSF      PORTB,RS      ; set RS=1
BSF      PORTB,ENA     ; set ENA = 1 (Enable Command Latch)
MOVF  CNTD,W           ; Move COUNT to W. Used as index in DISP_COMMAND
CALL  GET_CHAR        ; get next init. command --> W (table lookup)
MOVWF  PORTD           ; write command to Port D
NOP                                ; wait 1 us for command to latch
BCF  PORTB,ENA         ; Set ENA=0
CALL  WAIT500          ; Delay 500ms
```

;--Block 4 from Flow Chart

```
INCF  CNTD             ; increment count
INCF  ADDR             ; increment address
```

;--Block 5 and 6 from Flow Chart

```
EQUALS8      MOVLW      0x08
              SUBWF     CNTD,W
              BTFSS STATUS,Z      ; Test if Count=8
              GOTO  EQUALS16      ; If NO, check if = 16
              MOVLW      0xC0      ; If Yes, Change Address Value
              MOVWF     ADDR
```

```
EQUALS16      MOVLW      0x10
              SUBWF     CNTD,W
              BTFSS STATUS,Z      ; Test if Count=16
              GOTO  DRLOOP      ; If No, go to
              RETURN              ; If Yes, Return to RESET
```

```
; Here you must provide a full function for implementing the
; writing of characters to the LCD screen. This function must
; perform a single write of your full message to the screen. If
; needed, you are welcome to write additional subroutines. If you
; choose to do so, clearly document them to thier function if
; obvious
;-----
```

RETURN

```
===== DISPLAY ROUTINE END =====;
```

```
=====;
; Interrupt Routines ;
=====;
```

; Used in Lab 6

```

;=====;
;               Tables               ;
;=====;

```

ORG 0x0500 ; Tables are placed in memory block 05 (page 0)

DISP_COMMAND

```

ADDWF PCL,F ; This command adds the offset (placed in W before calling the
; table) to the PCL. Thus, the next line to execute will be
; indicated by the offset number listed to the right of the line.
; Effectively, the number placed in W chooses one of the 5 lines
; below.

```

```

RETLW FSET ; (0)
RETLW CLR ; (1)
RETLW OFF ; (2)
RETLW ON ; (3)
RETLW CLR ; (4)

```

GET_CHAR

```

ADDWF PCL,F ; This table works just like the one above, only larger. Fill the
; table with some sort of pithy message. Points awarded for humor
; value. ;)

```

```

; <----- Modify here ----->
; Fill in the table with your own message

```

```

RETLW A'G' ; (0)
RETLW A'E'
RETLW A'N'
RETLW A'T'
RETLW A'U'
RETLW A'S' ; (5)
RETLW A''
RETLW A'W'
RETLW A'A'
RETLW A'S'
RETLW A'' ; (10)
RETLW A'H'
RETLW A'E'
RETLW A'R'
RETLW A'E'
RETLW A'! ; (15)

```

END

```

;=====
;      Group: Ji Tu, Feiran Lei, Sean C. Ewing, Larissa M Borcz
;=====
;      Lab 5: LCD display interface
;=====
;
;  This code handles the configuration of the LCD display,
; subroutines for printing to, resetting, and clearing the
; LCD, as well as writing characters to it.
;=====

```

```

;=====
; Initialization of the environment
;=====
list    p=16F874          ; list directive to define processor
#include <p16F874.inc>      ; processor specific variable definitions

```

```

;=====
; Configuration of the PIC
;=====
__CONFIG B'11111100110001';setting configuration word (see PIC16F87X Data Sheet, p. 120)
;      ||  | |||
;      ||  | ||ww===== Oscillator = XT
;      ||  | ||
;      ||  | |w===== Watchdog Timer = disabled
;      ||  | |
;      ||  | w===== Power-Up Timer = enabled
;      ||  |
;      ||  w===== Brown-Out Reset = disabled
;      ||
;      ww===== Code Protection = off (also bits 5-4)

```

```

;=====;
;      Constant Declaration          ;
;=====;

```

```

;LCD display commands
CLR      EQU    0x01          ;Clears display, returns cursor home
FSET     EQU    0x38          ;Function Set command, Sets operational parameters
OFF      EQU    0x08          ;Turn Display Off
ON       EQU    0x0C          ;Turn Display On
MODSET   EQU    0x06          ;Sets cursor auto-move direction after char. write

```

;Delay values for timer0 (for Pre-Scalar = 1:256)

D5m EQU -D'20' ; 5 ms delay

D15m EQU -D'59' ;15 ms delay

D33m EQU -D'131' ;33 ms delay

;control pin identifiers on display (PORT B)

RS EQU 0

RW EQU 1

ENA EQU 2

```
=====;
;      Variable Declaration      ;
=====;
```

;***** RAM BANK 0 *****;

COUNT EQU 0x20 ; used in LCD display initialization

CNTD EQU 0x30 ; used in LCD display

ADDR EQU 0x40 ; used in LCD display

KEY EQU 0x50 ; used to store key value from RCREG

```
=====;
;      Reset/Interrupt Vectors  ;
=====;
```

ORG 0x0000 ; The PIC enters execution here
RESET GOTO MAIN

ORG 0x0004 ; The Interrupt vector enters here
INTHND RETFIE

```
=====;
;      Device Configuration     ;
=====;
```

ORG 0x0050
MAIN

;= Timer0 configuration =====;

BSF STATUS,RP0 ; Set the PIC to use bank 1 registers (see page 18 & 14)

MOVLW B'11000111' ;<----- Modify here ----->
; This value will be 'AND'ed with the default value of the OPTION
; register (81h) of the PIC (see next line). The default value on
; reset is B'11111111'. For more information on this register see
; page 19 of the datasheet (on carmen)
;
; You must provide values to set:
; 1) the clock source for the timer

```

; 2) the clock edge to count
; 3) the prescalar assignment
; 4) the prescalar value
;
; Fill in bits for the '*' in the command. Do not modify bits 7
; and 6.
;-----
ANDWF    OPTION_REG,F    ; AND the value with OPTION_REG and store in OPTION_REG
BCF      STATUS,RP0      ; Set the PIC to use bank 0 registers (see page 18 & 14)
MOVLW    B'00100000'     ;<----- Modify here -----
; This value will be 'OR'ed with the default value of the INTCON
; register (see next line). The default value of this register on
; reset is B'0000000x'. For more information on this register see
; page 20 of the datasheet (on Carmen)
;
; You must provide a value to:
; 1) Enable the Overflow Interrupt for Timer 0
;
; Fill in bit 5. Do not modify any other bits.
;-----
IORWFINTCON,F    ; OR the value with INTCON and store in INTCON

```

:= Port B configuration =====;

```

CLRF     PORTB          ; Clears Port B Data Latches
BSF      STATUS,RP0     ; Set the PIC to use bank 1 registers (see page 18 & 14)
MOVLW    B'11111000'    ;<----- Modify here -----
; This value will be moved into the PortB data direction register
; (TRISB). Provide a literal value for all the '*' in the command
; to set Port B data direction properly. For more information on
; PortB and TRISB, see page 31 of the datasheet
;
; You must provide values to:
; 1) Set all 8 bits of Port B to the proper data direction
;
; Fill in all 8 bits.
;-----
MOVWF    TRISB          ; Store the value in TRISB
MOVLW    B'00111111'    ;<----- Modify here -----
; This value will be 'AND'ed with the current value of OPTION_REG
; and saved into the OPTION_REG. See page 19 of the datasheet for
; more information
;
; You must provide values to:
; 1) enable/disable PORTB pull-ups
; 2) Set clock edge for interrupt on RB0
;
; Fill in bits 7 and 6. Do not modify any other bits
;-----

```

```

ANDWF    OPTION_REG,F      ; AND the value with OPTION_REG and store in OPTION_REG
BCF      STATUS,RP0       ; Set the PIC to use bank 0 registers (see page 18 & 14)

```

```

;= Port D configuration =====;
CLRF     PORTD             ; Clears Port D Data Latches
BSF      STATUS,RP0       ; Set the PIC to use bank 1 registers (see page 18 & 14)
MOVLW    B'00000000'      ;<----- Modify here -----
                                ; This value will be moved into the PortD data direction register
                                ; (TRISD). Provide a literal value for all the '*' in the command
                                ; to set Port D data direction properly. For more information on
                                ; PortD and TRISD, see page 31 of the datasheet
                                ;
                                ; You must provide values to:
                                ; 1) Set all 8 bits of Port D to the proper data direction
                                ;
                                ; Fill in all 8 bits.
                                ;-----
MOVWF     TRISD             ; Move the value to TRISD
BCF      STATUS,RP0       ; Set the PIC to use bank 0 registers (see page 18 & 14)

```

```

;=====;
;      System Initialization      ;
;=====;

```

```

BCF      PORTB,ENA        ; Initially, we disable the LCD
BCF      PORTB,RW         ; Also, R/W will always be 0, as we will only be writing data

MOVLW    0x05             ;<----- Modify here -----
                                ; This value will be loaded into the PCLATH register (see next line
                                ; of code). We want to setup the PCLATH register so that when
                                ; instructions later on use the PCL as a destination, the correct
                                ; result occurs. Specifically, the GET_CHAR and DISP_COMMAND
                                ; routines make use of such instructions. See page 26 of the
                                ; PIC manual for more information
                                ;
                                ; You must provide values to:
                                ; 1) Set the PCLATH register with the appropriate value to point
                                ;    to the table functions found at the end of the code
                                ;
                                ; Fill in both HEX values
                                ;-----
MOVWF     PCLATH           ; Move the value to PCLATH

```

```
;=====;
;    MAIN Operations                ;
;=====;
```

```
CALL RESET_LCD    ; Call subroutine to reset the LCD

CALL INIT_LCD     ; Call subroutine to initialize the LCD

CALL LCD_DISP     ; Call subroutine to display text on the LCD

GOTO RESET        ; return to the start of the code
```

```
;=====;
;    Functions                    ;
;=====;
```

```
RESET_LCD    ; resets LCD display -----
              ; parameters:
              ;    none
```

```
MOVLW    0x03    ;
MOVWF    COUNT   ; set COUNT = 3
```

```
RLOOP      BCF    PORTB,RS    ; set RS=0
```

```
MOVLW    D15m    ;
CALL    DELAY     ; wait 15 ms
```

```
BCF    PORTB,RW    ; set R/W = 0
```

```
BSF    PORTB,ENA    ; set ENA=1 (Enable Command Latch)
```

```
MOVLW    FSET     ;
MOVWF    PORTD     ;write Function Set command to Port D
```

```
NOP                ; wait 1 instruction cycle (1 us) for command to latch
```

```
BCF    PORTB,ENA    ; set ENA = 0 (Disable Command Latch)
```

```
DECFSZ    COUNT,F    ; Check to see if this has been done 3 times.
GOTO RLOOP            ; if not, do it again
RETURN      ; if so, then LCD controller reset
```

```
;-----;
```

```

INIT_LCD    ; initializes LCD display settings/options -----
            ; parameters:
            ;      none

            CLRF  COUNT            ; clear COUNT

NEXTCOM     MOVLW    D5m          ;
            CALL  DELAY            ; wait 5 ms

            BSF   PORTB,ENA       ; set ENA = 1 (Enable Command Latch)

            MOVF  COUNT,W         ; Move COUNT to W. Used as index in DISP_COMMAND

            CALL  DISP_COMMAND    ; get next init. command --> W (table lookup)

            MOVWF PORTD           ; write command to Port D

            NOP                    ; wait 1 us for command to latch

            BCF   PORTB,ENA       ; set ENA=0 (Disable Command Latch)

            INCF  COUNT,F         ; increment command index for table

            MOVLW 0x05            ;
            SUBWF COUNT,W         ; Subtract 5 from COUNT and see if it's a 0
            BTFSS STATUS,Z        ;

            GOTO  NEXTCOM         ; if not, do the next

            MOVLW D5m            ;
            CALL  DELAY            ; if so, wait an additional 5 ms

            RETURN                ; and now Display is ready for operation
;-----

```

```

DELAY      ; delay routine -----
            ; parameters:
            ;      W      :      Moved to TMR0. Routine then waits for overflow

            BCF   INTCON,T0IF     ; Clear Timer0 Interrupt Flag

            MOVWF TMR0            ; Move W --> TMR0 delay register

WAIT       BTFSS INTCON,T0IF     ; has the delay period passed?

            GOTO  WAIT            ; no - keep waiting

```

```
        RETURN                ; yes - proceed
;-----
```

```
WAIT500                ; 1/2 second delay routine -----
                        ; parameters:
                        ;      none
```

```
        MOVLW    D'15'
        MOVWF    COUNT                ; set COUNT to 15
```

```
HOLDUP    MOVLW    D33m                ;
        CALL    DELAY                ; wait for 33 ms
```

```
        DECFSZ    COUNT,F                ; check to see delay has been called 15 times
```

```
        GOTO    HOLDUP                ; if not, delay again
```

```
        RETURN                ; if so, return
;-----
```

```
;===== DISPLAY ROUTINE START =====;
```

```
LCD_DISP                ;<----- Modify here -----
;--Block 1 from Flow Chart
```

```
        CLRF    STATUS
        MOVLW    0x80                ; initialize address
        MOVWF    ADDR
        CLRF    CNTD                ; clear CNTD
```

```
;--Block 2 from Flow Chart
```

```
DRLOOP    BCF    PORTB,RS                ; set RS=0
        BSF    PORTB,ENA                ; set ENA = 1 (Enable Command Latch)
        MOVF    ADDR,W
        MOVWF    PORTD
        NOP                ; wait 1 instruction cycle (1 us) for command to latch
        BCF    PORTB,ENA                ; set ENA=0 (Disable Command Latch)
        MOVLW    D5m                ;
        CALL    DELAY                ; wait 5 ms
```

--Block 3 from Flow Chart

```
BSF      PORTB,RS      ; set RS=1
BSF      PORTB,ENA     ; set ENA = 1 (Enable Command Latch)
MOVWF   CNTD,W         ; Move COUNT to W. Used as index in DISP_COMMAND
CALL    GET_CHAR      ; get next init. command --> W (table lookup)
MOVWF   PORTD          ; write command to Port D
NOP      ; wait 1 us for command to latch
BCF     PORTB,ENA      ; Set ENA=0
CALL    WAIT500        ; Delay 500ms
```

--Block 4 from Flow Chart

```
INCF    CNTD          ; increment count
INCF    ADDR          ; increment address
```

--Block 5 and 6 from Flow Chart

```
EQUALS8      MOVLW      0x08
              SUBWF     CNTD,W
              BTFSS    STATUS,Z      ; Test if Count=8
              GOTO     EQUALS16      ; If NO, check if = 16
              MOVLW     0xC0         ; If Yes, Change Address Value
              MOVWF     ADDR

EQUALS16      MOVLW      0x10
              SUBWF     CNTD,W
              BTFSS    STATUS,Z      ; Test if Count=16
              GOTO     DRLOOP        ; If No, go to
              RETURN          ; If Yes, Return to RESET
```

```
; Here you must provide a full function for implementing the
; writing of characters to the LCD screen. This function must
; perform a single write of your full message to the screen. If
; needed, you are welcome to write additional subroutines. If you
; choose to do so, clearly document them to thier function if
; obvious
;-----
```

RETURN

===== DISPLAY ROUTINE END =====;

```
=====;
;      Interrupt Routines      ;
=====;
```

; Used in Lab 6

```
;=====;  
;           Tables           ;  
;=====;
```

ORG 0x0500 ; Tables are placed in memory block 05 (page 0)

DISP_COMMAND

ADDWF PCL,F ; This command adds the offset (placed in W before calling the
; table) to the PCL. Thus, the next line to execute will be
; indicated by the offset number listed to the right of the line.
; Effectively, the number placed in W chooses one of the 5 lines
; below.

RETLW FSET ; (0)
RETLW CLR ; (1)
RETLW OFF ; (2)
RETLW ON ; (3)
RETLW CLR ; (4)

GET_CHAR

ADDWF PCL,F ; This table works just like the one above, only larger. Fill the
; table with some sort of pithy message. Points awarded for humor
; value. ;)

;<----- Modify here ----->
; Fill in the table with your own message

RETLW A'G' ; (0)
RETLW A'E'
RETLW A'N'
RETLW A'T'
RETLW A'U'
RETLW A'S' ; (5)
RETLW A' '
RETLW A'W'
RETLW A'A'
RETLW A'S'
RETLW A' ' ; (10)
RETLW A'H'
RETLW A'E'
RETLW A'R'
RETLW A'E'
RETLW A'!' ; (15)

END