Logic Lab 5: Parity Lab

Group # 2
Erich Wanzek, Nathan Phipps
July 11, 2019

ABSTRACT:

In this lab we created Verilog code to both generate and check parity bits. To elaborate, In order to fully implement our work onto an FPGA Altera board, we first generated minterm expressions pertaining to even, as well as odd, parity generators and checkers. At the top-most level of our code, we put together all of our modules, while creating and implementing additional code modules, in order to fulfill the objectives set forth by our lab rubric. These additional code modules, consisted of modules for seven segment hex displays, and two “2 to 1” multiplexers (1 bit, and 4 bit MUX). From our work, within this lab, we are finally able to offer a full report on the processes, we underwent, as we implemented our parity generators and parity checkers onto an FPGA board. The outline for this report will appear as follows:

1.) **Group Member Contributions:**
   A brief discussion on group member contributions

2.) **Lab Analysis:**
   A.) Analysis of Part 1 from our lab
   B.) Analysis of Part 2 from our lab
   C.) Analysis of Part 3 from our lab
   D.) Analysis of Part 4 from our lab
   E.) Analysis of Part 5 from our lab

3.) **Wave Diagram Pictures:**
   A.) Wave Diagram Pictures of Part 1 from our lab
   B.) Wave Diagram Pictures of Part 2 from our lab
   C.) Wave Diagram Pictures of Part 3 from our lab
   D.) Wave Diagram Pictures of Part 4 from our lab

4.) **Verilog Code:**
   A.) Code from Part 1 from our lab
   B.) Code from Part 2 from our lab
   C.) Code from Part 3 from our lab
   D.) Code from Part 4 from our lab
   E.) Code from Part 5 from our lab

5.) **Test Case Pictures**
   Pictures of the four test cases of the top_level

Group Member Contributions

Throughout the entirety of this lab both group members, Nathan Phipps, and Erich Wanzek, performed equal duties, writing Verilog code, performing wave diagram simulations, compiling data, taking photos and typing this report.

Analysis

Analysis Part 1:

In this part we created an odd parity generator. During this part of the lab we utilized a truth table, in order to generate minterm expressions, which, through the use of boolean algebra, we obtained minimum expressions for this table. From these expressions we were able to implement verilog code that fulfilled our objective to create an odd parity generator. The final expression for the odd parity generator ended up being: **(A⊕B⊕C)'**

Analysis Part 2:

In this part we created an even parity generator. In this part of the lab we, again, utilized a truth table to find minterm expressions, which, through the use of boolean algebra, we obtained minimum expressions. The final expression for the even parity generator, in this case, ended up being the complement of the final expression from the odd parity generator. From these expressions we were able to implement verilog code that fulfilled our objective to create an even parity generator. The final expression for the odd parity generator ended up being: **A⊕B⊕C**

Analysis Part 3:

In Part 3 we wrote a Verilog module for an even parity checker circuit. The received message that is inputted into the even parity checker is 5 bits. The four most significant bits are the data bits; the least significant bit is the parity bit. The even parity checker module uses an instantiation of the even parity generator to generate the even parity bit of the received message. Next the even parity checker module compares this generated bit with the parity bit that was sent with the message. The parity comparison/checking is carried out by XORing the generated parity bit of the message with the received parity bit that came with the transmitted message. After running a functional simulation, we were able to verify that the even parity checker circuit was correct.

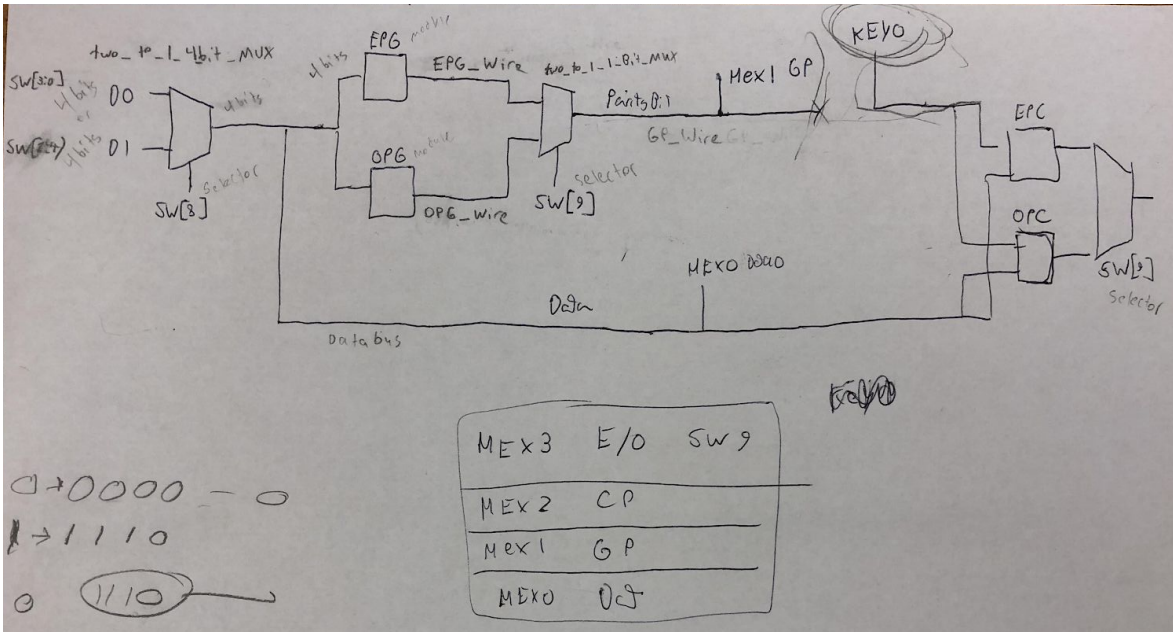
Analysis Part 4:

In Part 4 we wrote a Verilog module for an odd parity checker circuit. The received message that is inputted into the odd parity checker is 5 bits. The four most significant bits are the data bits; the least significant bit is the parity bit. The odd parity checker module uses an instantiation of the odd parity generator to generate the odd parity bit of the received message. Next the odd parity checker module compares this generated bit with the parity bit that was sent with the message. The parity comparison/checking is carried out by XORing the generated parity bit of the message with the received parity bit that came with the transmitted message. After running a functional simulation, we were able to verify that the odd parity checker circuit was correct.

Analysis Part 5:

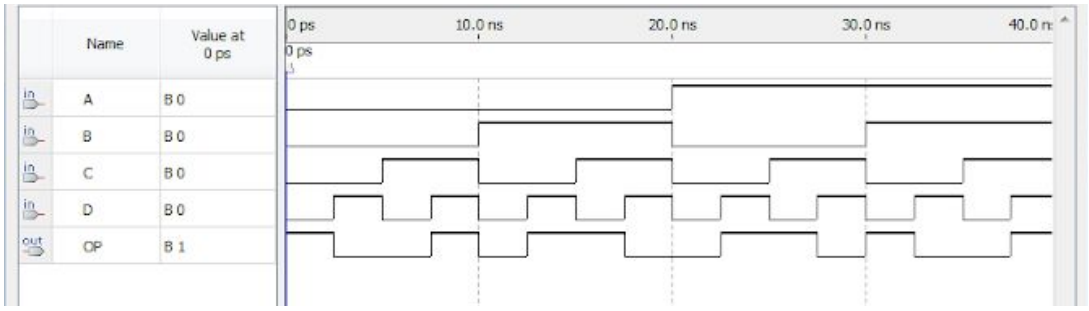
In Part 5 we wrote a Verilog top level module for the parity generator and checker. The Top level tests out the parity generators and checkers by running a 4 bit data message along with its parity through a Verilog circuit that utilizes all the parity generator and checker modules along with two to one 4 bit and 1 bit wide multiplexers. There are two data messages. Data message 0 consists of switches 3 through 0. Data message 1 consists of switches 7 through 4. Switch 8 selects which data message is to be transmitted by utilizing a two to one 4 bit multiplexer. (0 for Data message 0; 1 for Data message 1). The transmitted data message contains the data bits and their associated parity bit. Switch 9 determines whether the parity bit is odd or even parity (0 for Even; 1 for Odd). The selection between odd and even parity is carried out by using a two to one one bit multiplexer.Key 0 is the input parity bit for parity checker. The parity error checker receives the transmitted message and displays its output value on HEX2.The parity bit value is shown on HEX1. The data message is shown on HEX0. An E is displayed on HEX3 for even parity and a 0 is displayed on HEX3 for odd parity. All the switch states are shown on the corresponding LEDRs. The top_level functioned correctly after uploading it on the board and trying out the test cases.

Here is a picture of a schematic for the top level circuit we made to visualize how we coded our top level:

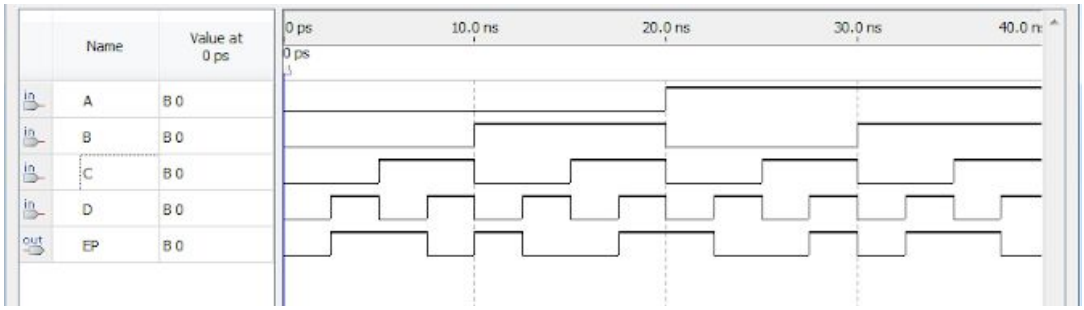


Wave Diagrams

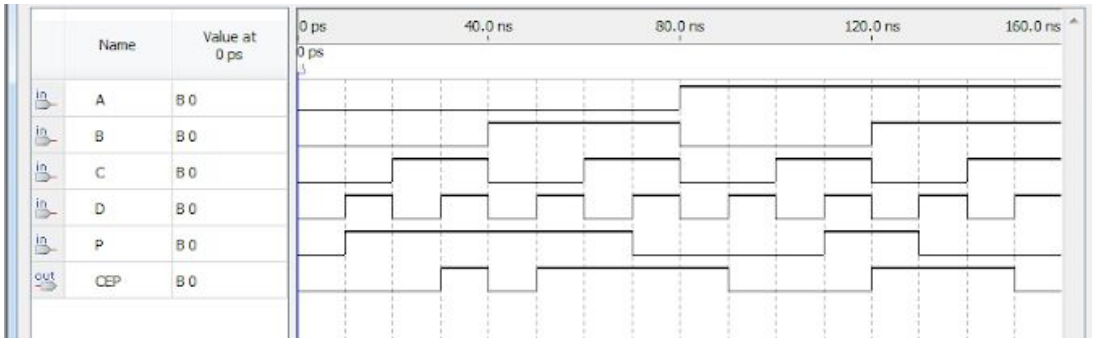
Wave Diagram Pictures of Part 1 from our lab
Odd Parity Generator:



Wave Diagram Pictures of Part 2 from our lab,
Even Parity Generator:



Wave Diagram Pictures of Part 3 from our lab,
Even parity checker:



Wave Diagram Pictures of Part 4 from our lab,
Odd parity checker:

Verilog Codes

Parity Generators

Odd Parity Generator:

```
module odd_parity_generator(OP, A, B ,C, D ); //declares module

        input wire A, B ,C, D; //defines input wires A, B, C, D
        output wire OP; //defines output wire Odd parity
        assign OP = ~(A ^ B ^ C ^ D) ; //Logical expression for odd parity

endmodule //ends module
```

Even Parity Generator:

```
module even_parity_generator(EP, A, B ,C, D ); //declares module

        input wire A, B ,C, D; //defines input wires A, B, C, D
        output wire EP; //defines output wire EP
        assign EP = A ^ B ^ C ^ D ; //assigns logical expression for EP

endmodule //ends module
```

Parity Checkers

Even Parity Checker:

```
module even_parity_checker(CEP, A, B ,C, D, P); //declares module
        input wire A, B ,C, D, P; //defines input wires A, B, C, D, P
        output wire CEP; //defines output wire CEP

        wire EP; //defines the wire EP

        even_parity_generator(EP, A, B ,C, D ); //instantiates the even parity generator to generate the even parity bit to be
checked.

        assign CEP = EP ^ P; //assigns the logical expression to CEP

endmodule //ends module
```

Odd Parity Checker:

```
module odd_parity_checker(COP, A, B ,C, D, P); //declares module
        input wire A, B ,C, D, P; //defines input wires A, B, C, D, P
        output wire COP; //defines output wire COP

        wire EP; //defines wire EP

        odd_parity_generator(OP, A, B ,C, D ); //instantiates the odd parity generator to generate the odd parity bit to be checked

        assign COP = OP ^ P; //assigns logical expression to COP

endmodule //ends module
```

Top Level:

Top Level:

```
module top_level(SW, KEY, HEX0, HEX1, HEX2, HEX3,LEDR); //declares the top_level module
        input wire [9:0]SW; //defines input switches
        input [0:0]KEY; //Key 0 is the input parity bit for parity checker.

        output [9:0]LEDR; //defines output LEDR wires
        output wire [6:0]HEX0; //defines output HEX0 display.
        output wire [6:0]HEX1; //defines output HEX1 display.
        output wire [6:0]HEX2; //defines output HEX2 display.
        output wire [6:0]HEX3; //defines output HEX3 display.
        //Switch 8 selects which data message to transmit.

        assign Parity = SW[9]; // Switch 9 determines whether the parity is odd or even.(0 for Even; 1 for Odd.)
        wire P; //defines wire P
        assign P = KEY [0:0]; // assigns wire P to KEY [0:0]

        wire [3:0]D0 = SW[3:0]; //Data message 0: switches 3:0
        wire [3:0]D1 = SW[7:4]; //Data message 1: switches 7:4


        wire [3:0]Databus; //defines databus wire, 4 bits wide
        wire EPG_wire; //defines even parity generator wire
        wire OPG_wire; //defines odd parity generator wire

        wire GP_wire; //defines generated parity wire

        wire CEP_wire; //defines checked even parity wire
        wire COP_wire; //defines checked odd parity wire

        wire CP_wire; //defines checked parity output wire

        two_to_1_4_bit_mux inst7(Databus[3:0],D0[3:0],D1[3:0],SW[8]); //instantiates a two to 1, 4 bit multiplexer to select the
        data message

        even_parity_generator(EPG_wire, Databus[3], Databus[2] ,Databus[1], Databus[0]); //instantiates the even parity
        generator to generate the even parity bit
        odd_parity_generator(OPG_wire, Databus[3], Databus[2] ,Databus[1], Databus[0]); //instantiates the odd parity generator
        to generate the odd parity bit

        mux_2to1 inst4(GP_wire, SW[9], EPG_wire, OPG_wire); //instantiates a 2 to 1, 1 bit multiplexer to select between the
        generated even parity bit and odd parity bit

        even_parity_checker(CEP_wire,Databus[3], Databus[2] ,Databus[1], Databus[0], P); //checks the recieved data and even
        parity bit
        odd_parity_checker(COP_wire, Databus[3], Databus[2] ,Databus[1], Databus[0], P); //checks the recieved data and odd
        parity bit

        mux_2to1 inst5(CP_wire, SW[9], CEP_wire, COP_wire); //instantiates a two to 1, 1 bit MUX to select between the even bit
        checker and the odd bit checker outputs.


        wire [3:0]X; //defines 4 wires X
        wire [3:0]Y; //defines 4 wires Y
        wire [3:0]Z; //defines 4 wires Z
        wire [3:0]Z0; //defines 4 wires Z0
        wire [3:0]Z1; //defines 4 wires Z1


        assign X = {3'b000, GP_wire}; //concatenates the X wires
        assign Y = {3'b000, CP_wire}; //concatenates the Y wires
```

```
        assign Z0 = {3'b000, Parity}; //concatenates the Z wire
        assign Z1 = {3'b111, ~(Parity)}; //concatenates the Z wire


        two_to_1_4_bit_mux inst6(Z,Z0,Z1,Parity); //instantiates the two to 1, 4 bit MUX


        hex_7seg_bitwise  inst0 (Databus[3:0], HEX0); //calls hex_7seg_bitwise

        hex_7seg_bitwise  inst1 (X, HEX1); //calls hex_7seg_bitwise

        hex_7seg_bitwise  inst2 (Y, HEX2); //calls hex_7seg_bitwise

        hex_7seg_bitwise  inst3 (Z, HEX3); //calls hex_7seg_bitwise

        assign LEDR[9:0] = SW[9:0]; //assigns LEDRS to switches

endmodule //ends module
```

Multiplexers:

2 to 1, “1 bit,” Multiplexer:

```
module mux_2to1(m, select, in0, in1); //declares module

        input wire select, in0, in1; //defines input wires select, in0, in1
        output wire m; //defines output wire
        assign m = (~select & in0) | (select & in1); //assigns logical expression to m

endmodule //ends module
```

2 to 1, “4 bit,” Multiplexer:

```
module two_to_1_4_bit_mux(O,I0,I1,S); //declares module

        input wire [3:0]I0; // Defines Two 4bit wide data inputs
        input wire [3:0]I1; // Defines Two 4bit wide data inputs
        input wire S; // Defines two selctor inputs
        output wire [3:0]O; //Defines output wire

        assign O[0] = (~S&I0[0]) | (S&I1[0]); //assigns logical expression to O[0]
        assign O[1] = (~S&I0[1]) | (S&I1[1]); //assigns logical expression to O[1]
        assign O[2] = (~S&I0[2]) | (S&I1[2]); //assigns logical expression to O[2]
        assign O[3] = (~S&I0[3]) | (S&I1[3]); //assigns logical expression to O[3]

endmodule //ends module
```

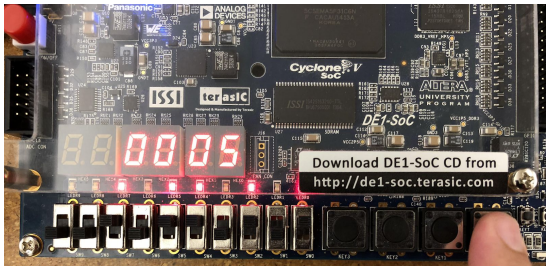
7 Segment Hex Display Module:

Hex Display:

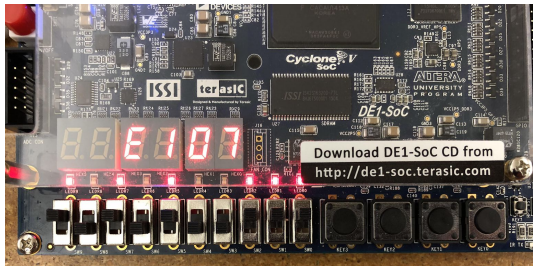
```
module hex_7seg_bitwise (X, segment); //module name 7 segment display, part 1e
        input wire [3:0]X; //defines a wire that's 4 bits wide
        output wire [6:0]segment; //defines an output wire that is 7 bits wide
        // Base for copying and pasting to make typing segments easier: A[3]&B[2]&C[1]&D[0] |
        // S0 = Sum(1,4,11,13)
        // Segment[0] = ~A[3]&~B[2]&~C[1]&D[0] | ~A[3]&B[2]&~C[1]&~D[0] | A[3]&B[2]&~C[1]&D[0] | A[3]&~B[2]&C[1]&D[0];
        assign segment[0] = ~X[3]&~X[2]&~X[1]&X[0] | ~X[3]&X[2]&~X[1]&~X[0] | X[3]&X[2]&~X[1]&X[0] | X[3]&~X[2]&X[1]&X[0];
        // S1 = Sum(5,6,11,12,14,15)
        // Segment[1] = A[3]&B[2]&~D[0] | ~A[3]&B[2]&~C[1]&D[0] | B[2]&C[1]&~D[0] | A[3]&C[1]&D[0];
        assign segment[1] = X[3]&X[2]&~X[0] | ~X[3]&X[2]&~X[1]&X[0] | X[2]&X[1]&~X[0] | X[3]&X[1]&X[0];
        // S2 = Sum(2,12,14,15)
        // Segment[2] = ~A[3]&~B[2]&C[1]&~D[0] | A[3]&B[2]&~D[0] | A[3]&B[2]&C[1];
        assign segment[2] = ~X[3]&~X[2]&X[1]&~X[0] | X[3]&X[2]&~X[0] | X[3]&X[2]&X[1];
        // S3 = Sum(1,4,7,9,10,15)
        // Segment[3] = ~A[3]&B[2]&~C[1]&~D[0] | ~B[2]&~C[1]&D[0] | B[2]&C[1]&D[0] | A[3]&~B[2]&C[1]&~D[0]
        assign segment[3] = ~X[3]&X[2]&~X[1]&~X[0] | ~X[2]&~X[1]&X[0] | X[2]&X[1]&X[0] | X[3]&~X[2]&X[1]&~X[0];
        // S4 = Sum(1,3,4,5,7,9)
        // Segment[4] = ~A[3]&B[2]&~C[1] | ~A[3]&D[0] | ~B[2]&~C[1]&D[0]
        assign segment[4] = ~X[3]&X[2]&~X[1] | ~X[3]&X[0] | ~X[2]&~X[1]&X[0];
        // S5 = Sum(1,2,3,7,13)
        // Segment[5] = ~A[3]&~B[2]&D[0] | ~A[3]&C[1]&D[0] | ~A[3]&~B[2]&C[1] | A[3]&B[2]&~C[1]&D[0]
        assign segment[5] = ~X[3]&~X[2]&X[0] | ~X[3]&X[1]&X[0] | ~X[3]&~X[2]&X[1] | X[3]&X[2]&~X[1]&X[0];
        // S6 = Sum(0,1,7,12)
        // Segment[6] = ~A[3]&~B[2]&~C[1] | ~A[3]&B[2]&C[1]&D[0] | A[3]&B[2]&~C[1]&~D[0]
        assign segment[6] = ~X[3]&~X[2]&~X[1] | ~X[3]&X[2]&X[1]&X[0] | X[3]&X[2]&~X[1]&~X[0];


endmodule //ends module
```

Test Case Pictures

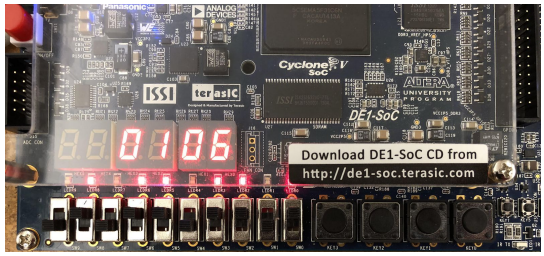
1. Data message 0: 0101; Data message 1: 1011
SW[8] = 0, SW[9]=0, KEY[0]=0;



2. Data message 0: 0111; Data message 1: 1010
SW[8] = 0, SW[9]=1, KEY[0]=1;



3. Data message 0: 1101; Data message 1: 0110
SW[8] = 1, SW[9]=0, KEY[0]=1;



4. Data message 0: 1011; Data message 1: 0010
SW[8] = 1, SW[9]=1, KEY[0]=0;