Final Project: Clock Dividers and Counters

Logic Lab

Nathan Phipps

July 29, 2019

ABSTRACT:

For the final project I developed code using Verilog to create multiple counters. The counters developed included a “divide by 5” counter, a “divide by 2” counter, and a “divide by 10” counter. Next I created various other counters (a decimal, and a six counter). Utilizing these codes I further fulfilled an objective to create a timer/counter circuit. From my work performed within this project I now offer a full report as pertaining to the project rubric. The outline from my report will be listed as follows

Analysis:

Analysis Part 1:

During this part of the lab I developed code for a “divide by 5 counter” using “if/else” statements nested in “always @ statements.” These “always @” statements were approved by the professor in place of using D-flip-flops. The first “always @” statement contained conditions in the statement’s arguments which used a positive clock edge to trigger the conditions within it. The first condition to be examined, in the first “always @” loop was an “if” statement, which triggered a counting variable (Cnt0). This count variable was incremented until it reached the value 4, all while the variable could not exceed the value of 4, using the operands “!=” (i.e. Cnt0 != 4). Under the circumstance that the count variable is incremented above 4, an else statement was implemented to reset the variable, allowing for the original clock pulse to trigger a cycle, that would be 40 nanoseconds long, **if** attributed to a clock diagram, when the original clock pulse period was every 10 nano-seconds. I say the words “if attributed to a clock diagram,” because the wire that was assigned to the value from Cnt0 was not set as an output clock within my wave diagram, however, it was instead added to a final output which added Cnt0, and Cnt 1 (from a second always @ statement), to produce the correct and final wave diagram simulation output for the “divide by 5 clock.”

In an additional “always @” statement I used a negative clock edge trigger to perform similar functions to the first “always @” statement, however these functions were performed on another variable (Cnt1). The second always @ statement, again allowed me to assign my Cnt1 variable to a wire, which, as mentioned earlier, was added to Cnt0 to create a final output for the “divide by 5 clock.” Again, the final output wire for my divide by 5 clock appeared as follows:

“
```
assign Clock_Output0 = Cnt0[1:1];
assign Clock_Output1 = Cnt1[1:1];
assign Clock_Output = Clock_Output0 | Clock_Output1;
```
“

After creating the divide by 5 counter, I implemented a reduced, but similar code for my divide by 2 counter; this time using only one always @ statement for positive edge clock triggering. Finally, for this part I implemented a top level that instantiated both the “divide by 5” counter, and “the divide by 2” counter, in order to create a divide by 10 counter. Lastly, I performed a wave diagram simulation for the “divide by 2” counter, the “divide by 5 counter,” and the “divide by 10 counter.” My results from these wave diagram simulations agreed with the desired results from the project rubric.

Analysis Part 2:

In this part of the project I created code for both a “decimal counter.” The code utilized several “if/else/else if” statements in order to fully implement these counters. To elaborate, the always @ statement was triggered by a positive edge clock pulse or a negative edge reset and included a condition to initialize at. Other conditions within the if/else statements ranged from setting registers to different values, and incrementing registers. Additionally, several of the if/else statements utilized the “&&” operand to say a certain event happens only if the every triggering term in the if statement’s arguments were met (ex: else if ((rA > 4'd0) && (Count_Direction == 1'b1) && (Start_Stop == 1'b0)). From my work in part 3 it would appear that my “decimal counter” did, in fact, perform correctly.

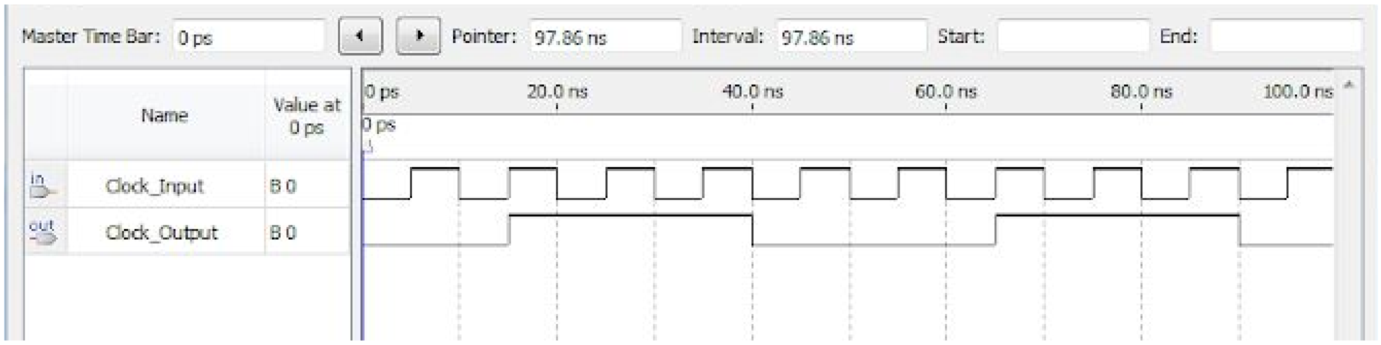
Analysis Part 3:

Within this part of the project, I created a top level and several additional codes necessary to run my top level. One of the additional codes I created was a “six counter.” The “six counter’s” code contained similar code to the “decimal counter,” but with small differences mostly pertaining to start and stop values (i.e. 6 versus 9). Additionally, I reused old code that I developed in a previous lab to help activate the 3 different hex displays. Next, this top level instantiated several clock dividers to bring the clock down from 50 MHz to 0.1Hz. The top level also instantiated both the “decimal counter” and the “six counter” in order to help create a functional timer, with features that included, start, stop, reset, and count direction. All of this timer activity was displayed onto the 3 hex displays. From my results, using my Altera, FPGA, board, I am able to conclude that my code did perform correctly, in accordance with the project guidelines. A video provided in my zip folder, included in my submission to Canvas, displays the final results.

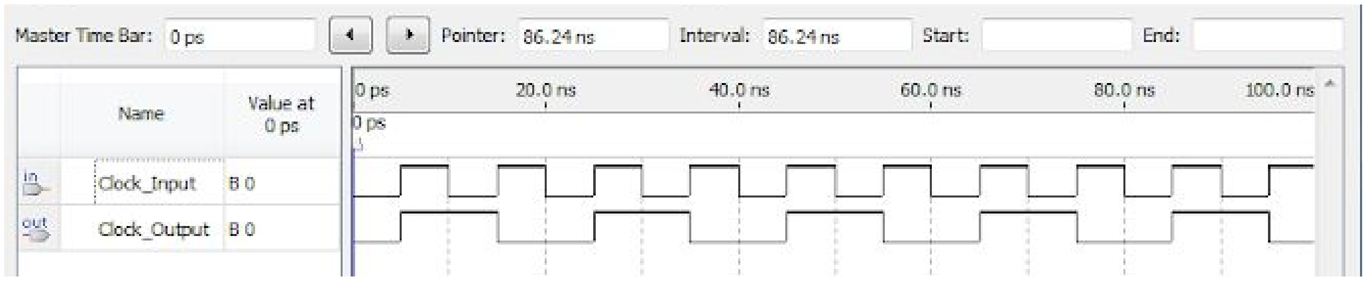
Wave Diagrams

Wave Diagram Pictures of Part 1 from lab:

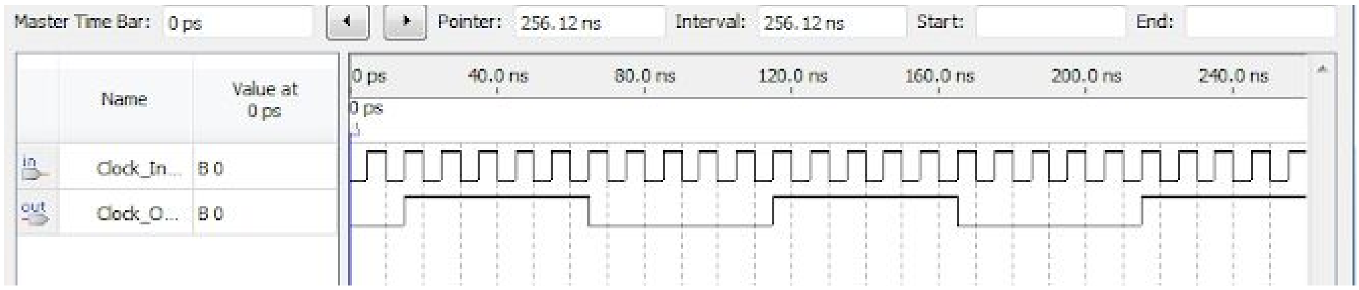
Divide by 5 Wave Diagram Simulation:



Divide by 2 Wave Diagram Simulation:

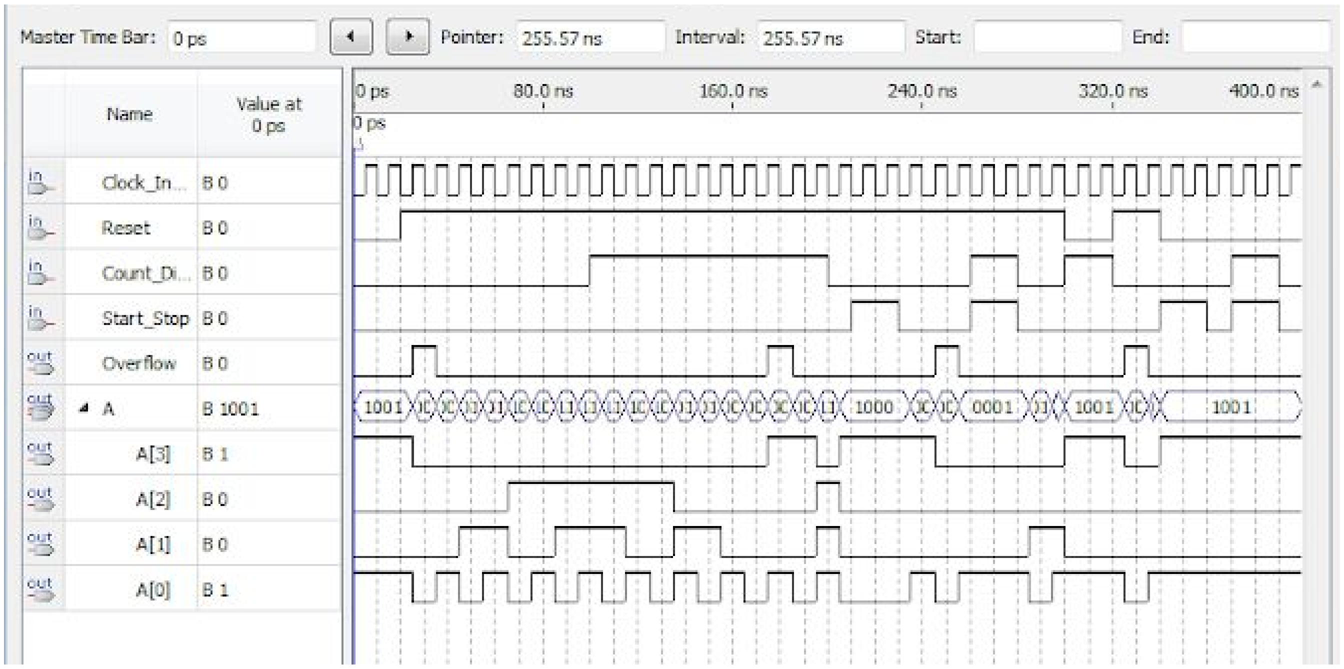


Divide By 10 Wave Diagram Simulation:



Wave Diagram Pictures of Part 2 from the lab:

Decimal Counter:



Verilog Codes

Code from Part 1

Divide By 5 Code:

```
module DivideBy_5(Clock_Input, Clock_Output); //declares module.

        output Clock_Output; //defines Clock_Output.
        input Clock_Input; //defines Clock_Input.

        wire Clock_Output0; //defines wire Clock_Output0.
        wire Clock_Output1; //defines Wire Clock_Output1.

        reg [2:0]Cnt0; //defines register Cnt0.
        reg [2:0]Cnt1; //defines register Cnt1.


        always @(posedge Clock_Input) begin //begins always @ statement for positive edge Clock_Input.
                if(Cnt0 != 4) begin //begins if statement for Cnt0, if statement runs until the counter hits 4, starting at 0.
                        Cnt0 <= Cnt0 + 1; //increments Cnt0 each time the if statement is triggered in the loop.
                        end //ends if statement.

                else Cnt0 <= 0; //if Cnt0 is greater than 4 then the else statement is triggered.
        end //ends else statement.


        always @(negedge Clock_Input) begin //begins always @ statement for negative edge Clock_Input.
                if(Cnt1 != 4) begin //begins if statement for Cnt1, if statement runs until the counter hits 4, starting at 0.
                        Cnt1 <= Cnt1 + 1; //increments Cnt1 each time the if statement is triggered in the loop.
                        end //ends if statement.
                else Cnt1 <= 0; //if Cnt1 is greater than 4 th\en the else statement is triggered.
        end //ends else statement.


        assign Clock_Output0 = Cnt0[1:1]; //assigns Clock_Output0 to Cnt0[1:1].
        assign Clock_Output1 = Cnt1[1:1]; //assigns Clock_Output1 to Cnt1[1:1].
        assign Clock_Output = Clock_Output0 | Clock_Output1; //assigns Clock_Output to Clock_Output0 | Clock_Output1.

endmodule //ends module.
```

Divide by 2 Code:

```
module DivideBy_2(Clock_Input, Clock_Output); //declares module DivideBy_2

        output Clock_Output;//defines Clock_Output.
        input Clock_Input; //defines Clock_Input.

        wire Clock_Output0; //defines wire Clock_Output0.

        reg [2:0]Cnt0; //defines register Cnt0.


        always @(posedge Clock_Input) begin //begins always @ statement for positive edge Clock_Input.
                if(Cnt0 != 1) begin //begins if statement for Cnt0, if statement runs until the counter hits 4, starting at 0.
                        Cnt0 <= Cnt0 + 1; //increments Cnt0 each time the if statement is triggered in the loop.
                        end //ends if statement.

                else Cnt0 <= 0; //if Cnt0 is greater than 4 then the else statement is triggered.
        end //ends else statement.

        assign Clock_Output0 = Cnt0[0:0]; //assigns Clock_Output0 to Cnt0[0:0]
        assign Clock_Output = Clock_Output0; //assigns Clock_Output to Clock_Output0.

endmodule //ends module
```

Divide by 10 Code:

```
module divide10(Clock_Input, Clock_Output); //declares module divide10

        output Clock_Output; //defines Output Clock_Output
        input Clock_Input; //defines input Clock_Input

        wire Clock_A; //defines wire Clock_A

        DivideBy_5(Clock_Input, Clock_A); //instantiates DivideBy_5
        DivideBy_2(Clock_A, Clock_Output); //instantiates DivideBy_5

endmodule //ends module
```

Code from Part 2

Decimal Counter Code:

```
module Count_Decimal(A, Overflow, Clock_Input, Reset, Count_Direction, Start_Stop); //declares module Count_Decimal

        output [3:0]A; //defines an output A, as 4 bits.
        output Overflow; //defines output Overflow.
        input Clock_Input, Reset, Count_Direction, Start_Stop; //defines inputs Clock_Input, Reset, Count_Direction, Start_Stop.

        reg [3:0]rA; //defines a register thats 4 bits.
        reg rOverflow; //defines register rOverflow.
```

```
	assign A = rA; //assigns A to rA.
	assign Overflow = rOverflow; //assigns Overflow to rOverflow.

	always@(posedge Clock_Input or negedge Reset) //starts always @ loop for positive edge of Clock_Input or negative edge eset.
		if (Reset ==1'd0) begin //begins nested if statement, which initializes the registers to the set values.
			rOverflow <= 1'b0; //register rOverflow is less than or equal to binary 0, and is 1 bit.
			rA <= 4'd9; //register rA is less than or equal to decimal 9 in 4 bits.
			end //ends if statement

		else if ((rA == 4'd0) && (Start_Stop == 1'b0) && (Count_Direction == 1'b1)) begin
		//begins nested else if statement, register rA is equal to decimal 0 in 4 bits, and Count_Direction is set to 1, and Start_Stop is 0.
			rOverflow <= 1'b1; //register rOverflow is less than or equal to binary 1, and is 1 bit.
			rA <=4'b1001; //register rA is less than or equal to 9 in binary and is 4 bits.
			end //ends else if statement

		else if ((rA > 4'd0) && (Start_Stop == 1'b0) && (Count_Direction == 1'b1)) begin
		//begins nested else if statement, register rA is greater than decimal 0 in 4 bits, and Count_Direction is set to 1,and Start_Stop is 0.
			rOverflow <= 1'b0; //register rOverflow is less than or equal to binary 0, and is 1 bit.
			rA <= rA - 1'b1; //register rA is less than or equal to rA's previous value minus a one bit 1 in binary.
			end //ends else if statement

		else if ((rA == 4'd9) && (Start_Stop == 1'b0) && (Count_Direction == 1'b0)) begin
		//begins nested else if statement, register rA is equal to decimal 0 in 4 bits, and Count_Direction is set to 0, and Start_Stop is 0.
			rOverflow <= 1'b1; //register rOverflow is less than or equal to binary 1, and is 1 bit.
			rA <=4'b0000; //register rA is less than or equal binary zero and is 4 bits.
			end //ends else if statement

		else if ((rA < 4'd9) && (Start_Stop == 1'b0) && (Count_Direction == 1'b0)) begin
		//begins nested else if statement, register rA is less than decimal 9 in 4 bits, and Count_Direction is set to 0 and is 1bit, and
		//Start_Stop is 0 and is 1bit.
			rOverflow <= 1'b0; //register rOverflow is less than or equal to binary 0, and is 1 bit.
			rA <= rA + 1'b1; //register rA is less than or equal to rA's previous value plus a one bit 1 in binary.
			end //ends else if statement.

		else begin //begins nested else statement
			rOverflow <= 1'b0; //register rOverflow is less than or equal to binary 0, and is 1 bit.
			rA <= rA; //register rA is less than or equal to itself
			end //ends else if statement

endmodule //ends module
```

Code from Part 3:

Six Counter Code:

```
module Count_Six(A, Overflow, Clock_Input, Reset, Count_Direction, Start_Stop); //declares module Count_Six

	output [3:0]A; //defines an output A, as 4 bits.
	output Overflow; //defines an output Overflow.
	input Clock_Input, Reset, Count_Direction, Start_Stop; //defines inputs Clock_Input, Reset, Count_Direction, Start_Stop.

	reg [3:0]rA; //defines a register thats 4 bits.
	reg rOverflow; //defines register rOverflow.

	assign A = rA; //assigns A to rA.
	assign Overflow = rOverflow; //assigns Overflow to rOverflow.

	always@(posedge Clock_Input or negedge Reset) //starts always @ loop for positive edge of Clock_Input or negative edge Reset.
		if (Reset ==1'd0) begin //begins nested if statement, which initializes the registers to the set values.
			rOverflow <= 1'b0; //register rOverflow is less than or equal to binary 0, and is 1 bit.
			rA <= 4'd5; //register rA is less than or equal to decimal 5 in 4 bits.
```

```
                    end //ends if statement

          else if ((rA < 4'd5) && (Start_Stop == 1'b0) && (Count_Direction == 1'b0)) begin
          //begins nested else if statement, register rA must be less than 4 bits decimal 5, and Count_Direction must be equal to 1 bit binary 0,
          //and Start_Stop must equal 1 bit binary 0.
                    rOverflow <= 1'b0; //register rOverflow is less than or equal to binary 0, and is 1 bit.
                    rA <= rA + 1'b1; //register rA is less than or equal to the previous value of rA plus 1 bit binary 1.
                    end //ends if statement

          else if ((rA > 4'd0) && (Start_Stop == 1'b0) && (Count_Direction == 1'b1)) begin
          //begins nested else if statement, register rA must be greater than 4 bits decimal 0, and Count_Direction must be equal to 1 bit
          //binary 1, and Start_Stop must equal 1 bit binary 0.
                    rOverflow <= 1'b0; //register rOverflow is less than or equal to binary 0, and is 1 bit.
                    rA <= rA - 1'b1; //register rA is less than or equal to the previous value of rA minus 1 bit binary 1.
                    end //ends if statement

          else if ((rA == 4'd5) && (Start_Stop == 1'b0) && (Count_Direction == 1'b0)) begin
          //begins nested else if statement, register rA must be equal to 4 bits decimal 5, and Count_Direction must be equal to 1 bit binary 0,
          //and Start_Stop must equal 1 bit binary 0.
                    rOverflow <= 1'b1; //register rOverflow is less than or equal to binary 1, and is 1 bit.
                    rA <=4'b0000; //register rA is less than or equal binary zero and is 4 bits.
                    end //ends if statement

          else if ((rA == 4'd0) && (Start_Stop == 1'b0) && (Count_Direction == 1'b1)) begin
          //begins nested else if statement, register rA must be equal to 4 bits decimal 0, and Count_Direction must be equal to 1 bit binary 1,
          //and Start_Stop must equal 1 bit binary 0.
                    rOverflow <= 1'b1; //register rOverflow is less than or equal to binary 1, and is 1 bit.
                    rA <=4'b0101; //register rA is less than or equal binary 5 and is 4 bits.
                    end //ends if statement.

          else begin //begins nested else statement.
                    rOverflow <= 1'b0; //register rOverflow is less than or equal to 1 bit binary 0.
                    rA <= rA; //register rA is less than or equal rA.
                    end //ends if statement.

endmodule //ends module
```

Top Level Code:

```
module Top_Level_Counter_Circuit(CLOCK_50, SW, HEX0, HEX1, HEX2, LEDR); //declares module Top_Level_Counter_Circuit.

          output [2:0]LEDR; //defines 3 output switches.
          input wire CLOCK_50; //defines input wire CLOCK_50.
          input wire [2:0]SW; //defines 3 input switches.

          wire Clk0, Clk1, Clk2, Clk3, Clk4, Clk5; //defines clocks for cascading into dividers.
          wire [3:0]W0; //defines 4 wires W0.
          wire [3:0]W1; //defines 4 wires W1.
          wire [3:0]W2; //defines 4 wires W2.
          wire Overflow0, Overflow1, Overflow2; //defines wires for Overflow0, Overflow1, Overflow2.
          wire Reset, Count_Direction, Start_Stop; //defines wires for Reset, Count_Direction, Start_Stop.

          output wire [6:0]HEX0; //defines output wires for HEX0.
          output wire [6:0]HEX1; //defines output wires for HEX1.
          output wire [6:0]HEX2; //defines output wires for HEX2.

          assign LEDR[2:0] = SW[2:0]; //assigns LEDRS to Switches.

          assign Reset = SW[0]; //assigns Reset to Switch 0 (SW[0]).
          assign Count_Direction = SW[1]; //assigns Count_Direction to Switch 1 (SW[1]).
          assign Start_Stop = SW[2]; //assigns Start_Stop to Switch 2 (SW[2]).
```

```
        //The Codel below helps to reduce the clock from 50 MHz to 0.1Hz.
        DivideBy_5 inst0(CLOCK_50, Clk0); //instantiates DivideBy_5 inst0.
        divide10   inst1(Clk0, Clk1); //instantiates divide10 inst1.
        divide10 inst2(Clk1, Clk2); //instantiates divide10 inst2.
        divide10 inst3(Clk2, Clk3); //instantiates divide10 inst3.
        divide10 inst4(Clk3, Clk4); //instantiates divide10 inst4.
        divide10 inst5(Clk4, Clk5); //instantiates divide10 inst5.
        divide10 inst6(Clk5, Clk6); //instantiates divide10 inst6.

        hex_7seg_bitwise  inst8 (W0, HEX0); //instantiates hex_7seg_bitwise inst8.
        hex_7seg_bitwise  inst9 (W1, HEX1); //instantiates hex_7seg_bitwise inst9.
        hex_7seg_bitwise  inst10 (W2, HEX2); //instantiates hex_7seg_bitwise inst10.

        Count_Decimal inst11(W0, Overflow0, Clk6, Reset, Count_Direction, Start_Stop); //instantiates Count_Decimal inst11.
        Count_Decimal inst12(W1, Overflow1, Overflow0, Reset, Count_Direction, Start_Stop); //instantiates Count_Decimal inst12.
        Count_Six inst13(W2, Overflow2, Overflow1, Reset, Count_Direction, Start_Stop); //instantiates Count_Decimal inst13.


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
```