University of Colorado, Boulder

Electrical & Computer Engineering 2350

Digital Logic

Mini-CPU

April 1, 2019

by Connor Humiston

Introduction

Average computers today are capable of executing billions of operations every second. The speed with which these processes are completed is becoming more and more important with society's ever-expanding need for quick computation. Thus, it is important to first analyze how these operations work at a primitive level in search of faster procedures before materializing at a larger scale. This project aims to create a miniature central processing unit (CPU) that can perform basic arithmetic, comparison, and logical operations. The processes will be implemented on the MAX 10 (10M50DAF484C7G) Field Programmable Gate Array (FPGA) utilizing Verilog Hardware Description Language (VHDL) and Quartus Prime. The switches, LEDs, and seven segment displays found on the Terasic DE10-LITE board will be used to test each operation programmed in Verilog. The Verilog modules are summarized in a block diagram as illustrated in Figure 1 below. Mode and operation selection truth tables follow in Figure 2.

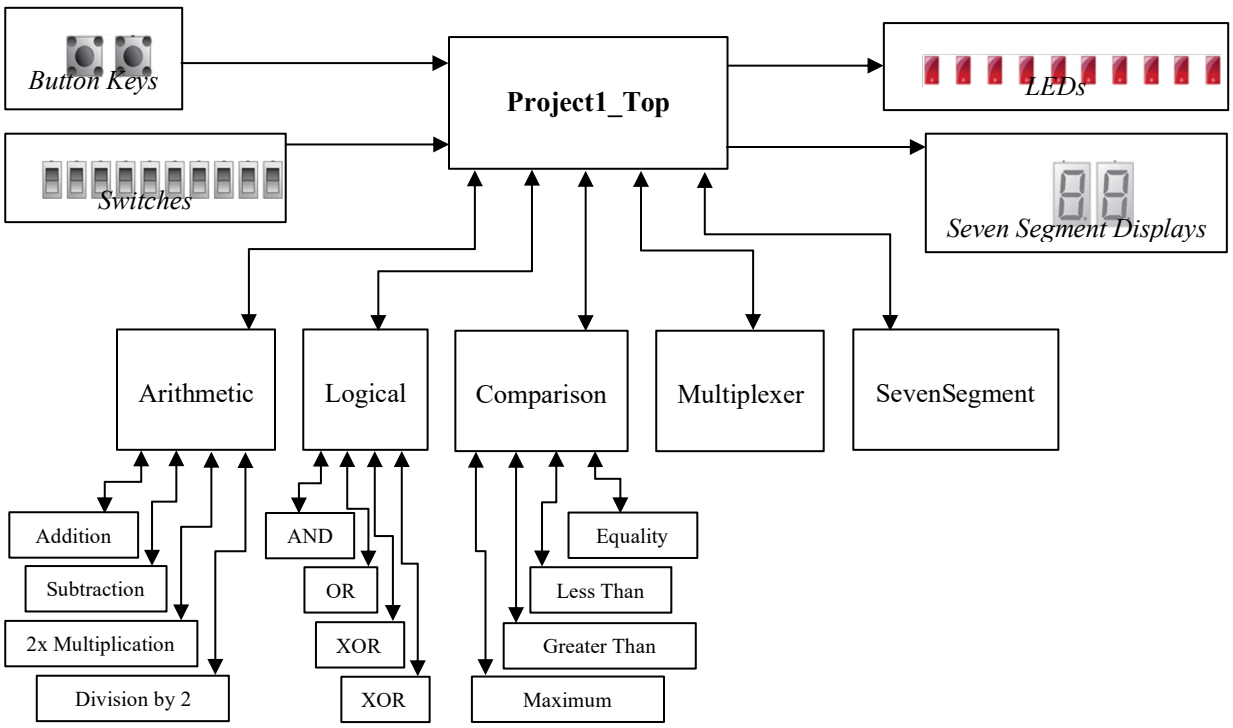

Figure 1: Block diagram (above) including key and switch inputs, project modules and functions, and LED and display outputs

Table 1: Mode Selection

KEY0	KEY1	k	Arithmetic	Logic	Comparison
0	0	00	1	0	0
0	1	01	1	0	0
1	0	10	0	1	0
1	1	11	0	0	1

Table 2: Arithmetic (KEY0 Press for k = 00 or 01)

SW9	SW8	Add	Subtr.	Mult.	Divide
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Table 3: Logic (KEY0 Press for k = 10)

SW9	SW8	AND	OR	XOR	NOT
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Table 4: Comparison (KEY1 Press for k = 11)

SW9	SW8	Equal	Greater	Less	Max
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Figure 2: Truth tables (above) illustrating multiplexer selection of mode and modular operation from button keys and switches

Results

The Verilog code for the following findings can be found in Appendix A.

Project1_Top

As displayed by the block diagram in Figure 1 above, Project1_Top takes in the two buttons and ten switches as inputs, and utilizes the Arithmetic, Logical, Comparison, Multiplexer, and SevenSegment modules to create outputs for the ten LEDs and two seven-segment displays. The top module exploits conditional statements for the button keys to select between the different modes. KEY0 switches between Arithmetic and the other two modules. KEY1 switches between the Comparison and Logic modules. Once a mode is selected, hexadecimal outputs will appear on the seven segment displays and binary outputs will appear on the LEDs. If a carry is produced by the Arithmetic unit, LEDR8 will illuminate and the decimal point on HEX0 will show. Similarly, LEDR9 and the decimal point on HEX1 will indicate if a remainder is produced by the Arithmetic unit.

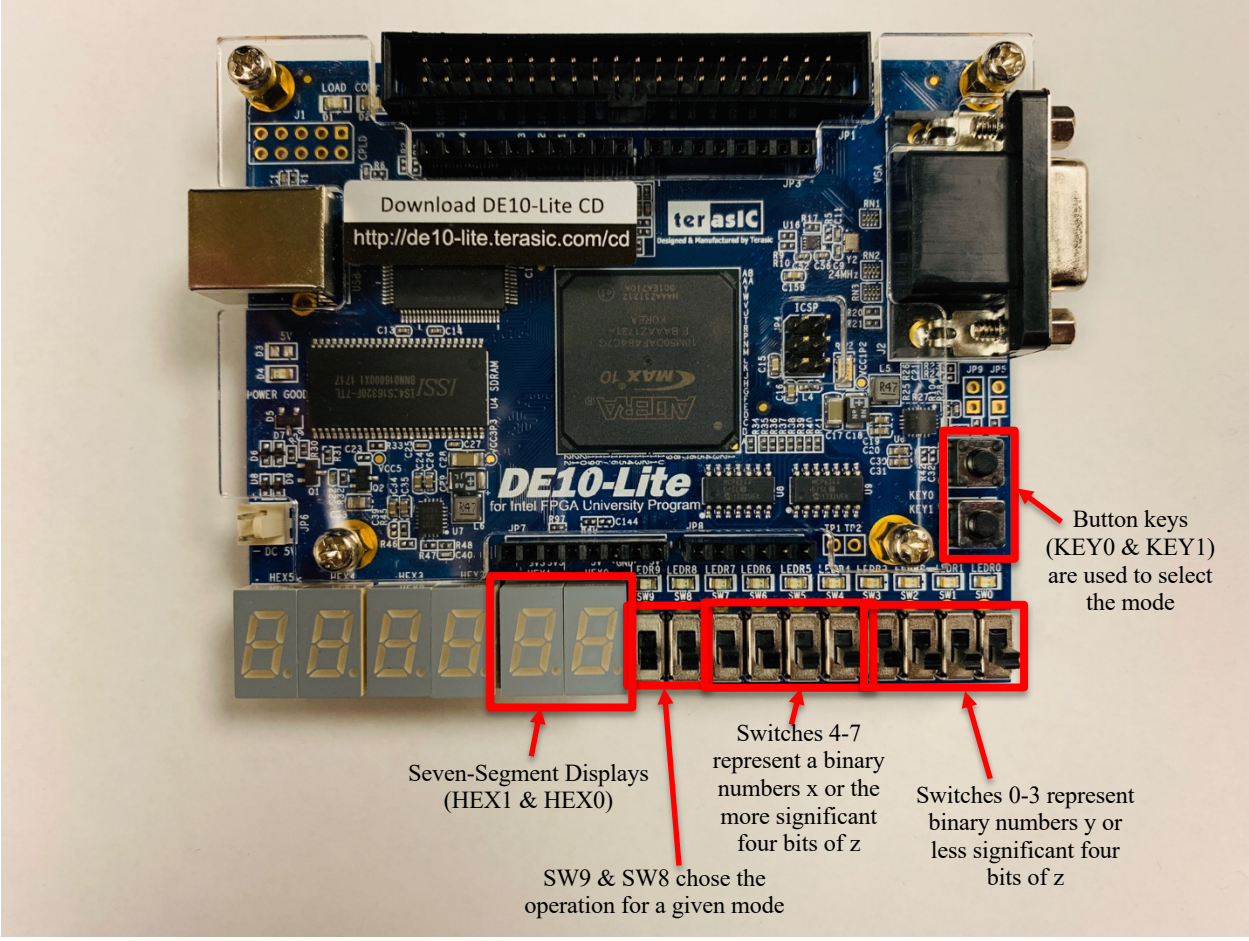

Figure 3: The DE10-Lite board with seven-segment displays, switches, LEDs, and button keys labeled

SevenSegment

This module takes in binary inputs and outputs hexadecimal numbers to the seven-segment display. As shown in Figure 3 below, hexadecimal digits can be created by turning on individual LEDs from the display. The SevenSegment module uses binary values (gfedcba) from the table in Figure 4 to output the correct hexadecimal representation on the display. It is important to note that the LEDs for the seven segment displays on the MAX10 board are actually on for low inputs and off when the input is high, so the code found in Appendix A represents the opposite of that in Figure 4.

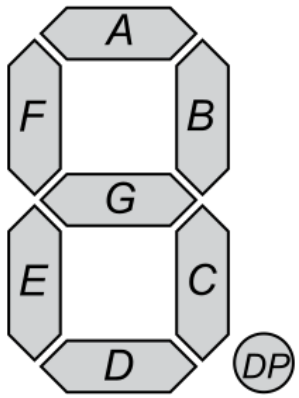

Figure 4: A seven-segment display with labeled individual LEDs

	h	g	f	e	d	c	b	a	hex value
0	0	0	1	1	1	1	1	1	3F
1	0	0	0	0	0	1	1	0	06
2	0	1	0	1	1	0	1	1	5B
3	0	1	0	0	1	1	1	1	4F
4	0	1	1	0	0	1	1	0	66
5	0	1	1	0	1	1	0	1	6D
6	0	1	1	1	1	1	0	1	7D
7	0	0	0	0	0	1	1	1	07
8	0	1	1	1	1	1	1	1	7F
9	0	1	1	0	1	1	1	1	6F
A	0	1	1	1	0	1	1	1	77
b	0	1	1	1	1	1	0	0	7C
C	0	0	1	1	1	0	0	1	39
d	0	1	0	1	1	1	1	0	5E
E	0	1	1	1	1	0	0	1	79
F	0	1	1	1	0	0	0	1	71

Figure 5: Hexadecimal equivalents for each digit to be displayed from binary inputs corresponding to the display's LEDs

Arithmetic

Addition $(x + y)$, subtraction $(x - y)$, multiplication-by-2 $(2 \times z)$, and division-by-2 $(2 \div z)$ operations can be completed with the Arithmetic module. Truth Table 2 in Figure 2 exhibits how the values of switch eight and nine can select which arithmetic operation is desired in a decoder fashion. The carry and remainder bits illuminate the seven-segment decimals and LEDR8 and LEDR9 as described above. Hexadecimal values of $x = 0xA,\ y = 0x7$, and $z = 0xA7$ were chosen to test this module's outputs:

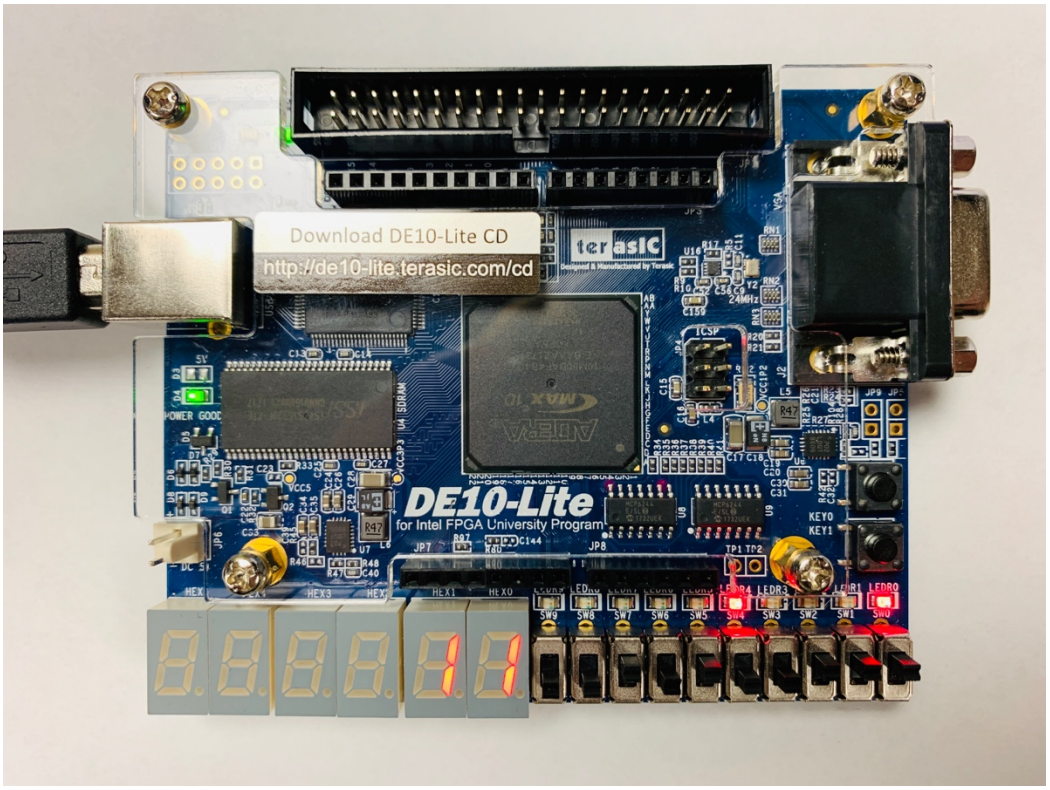

Figure 6: The result of adding $x + y$ where $x = 0xA$ and $y = 0x7$ with binary switch inputs and LED and display outputs

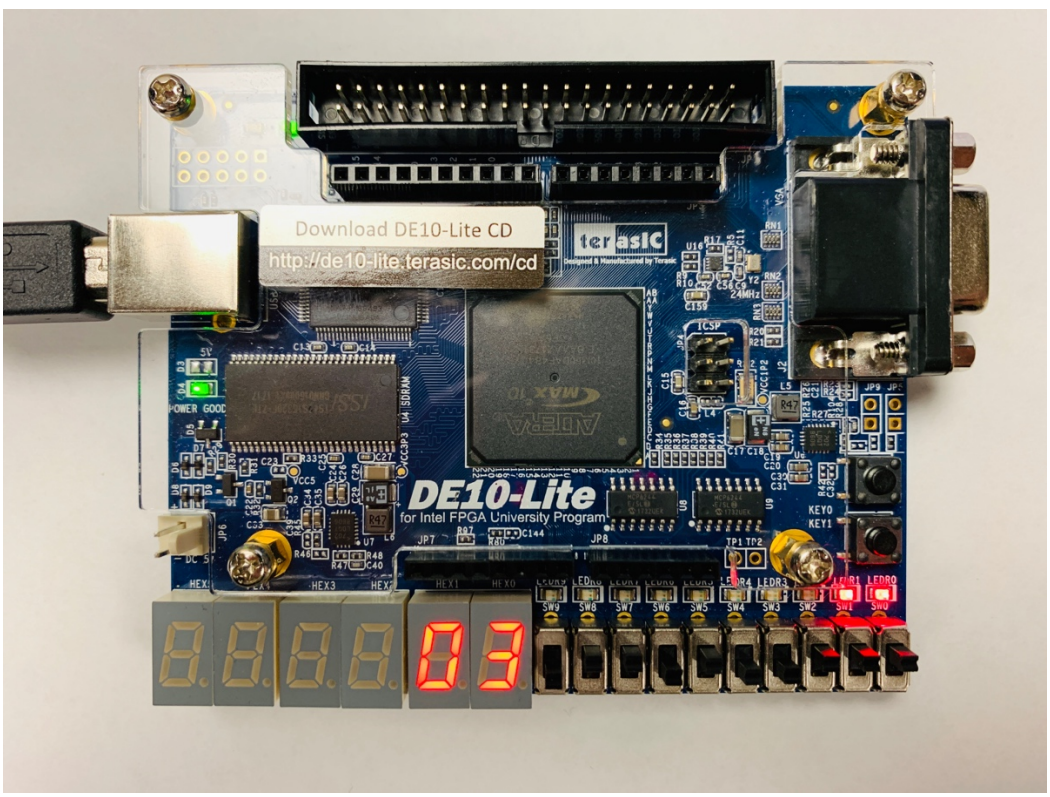

Figure 7: The result of subtracting $x - y$ where $x = 0xA$ and $y = 0x7$ with binary switch inputs and LED and display outputs

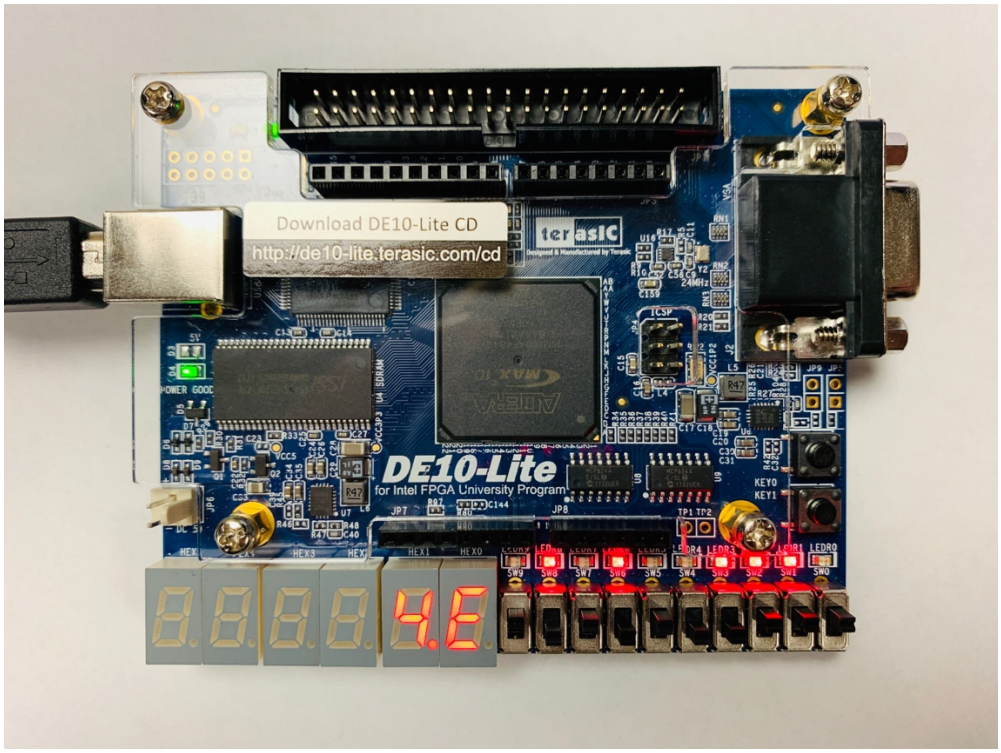

Figure 8: The result of multiplying z by 2 where $z = 0xA7$ with indication of the carry on LEDR8 and the HEX1 decimal point

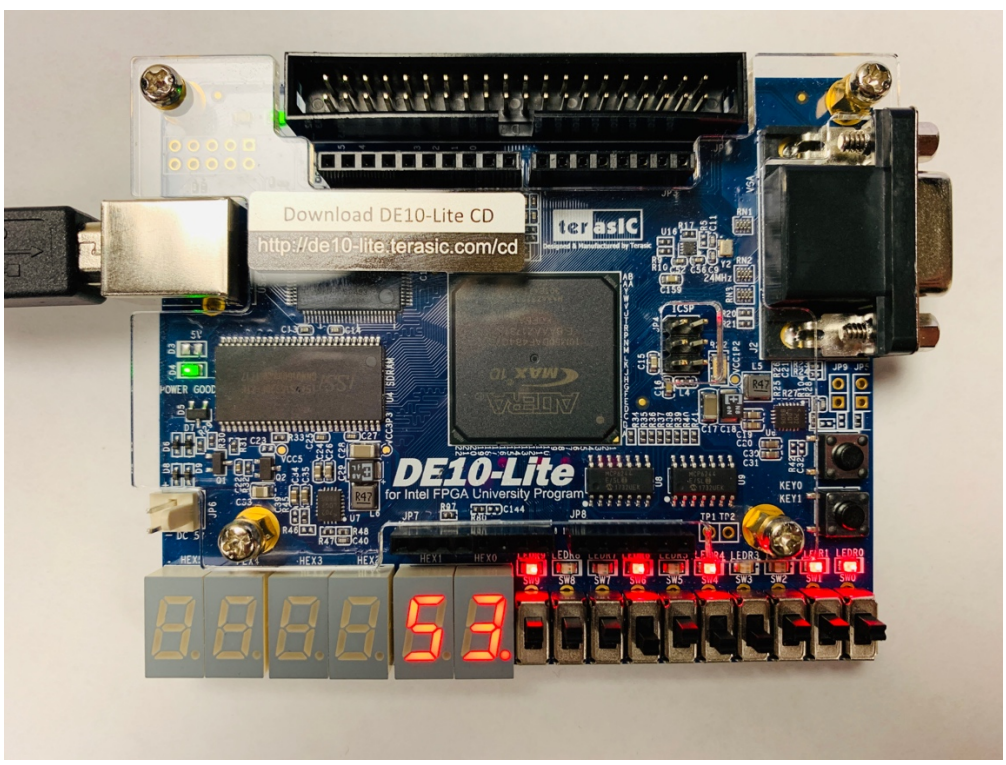

Figure 9: The result of dividing z by 2 where $z = 0xA7$ with indication of the remainder on LEDR9 and the HEX0 decimal

Logical

The logical module includes AND, OR, XOR, and NOT operations that can be selected using SW9 and SW8 according to Table 3 in Figure 2 above. The same hexadecimal values of $x = 0xA$, $y = 0x7$, and $z = 0xA7$ were chosen to test this module's outputs:

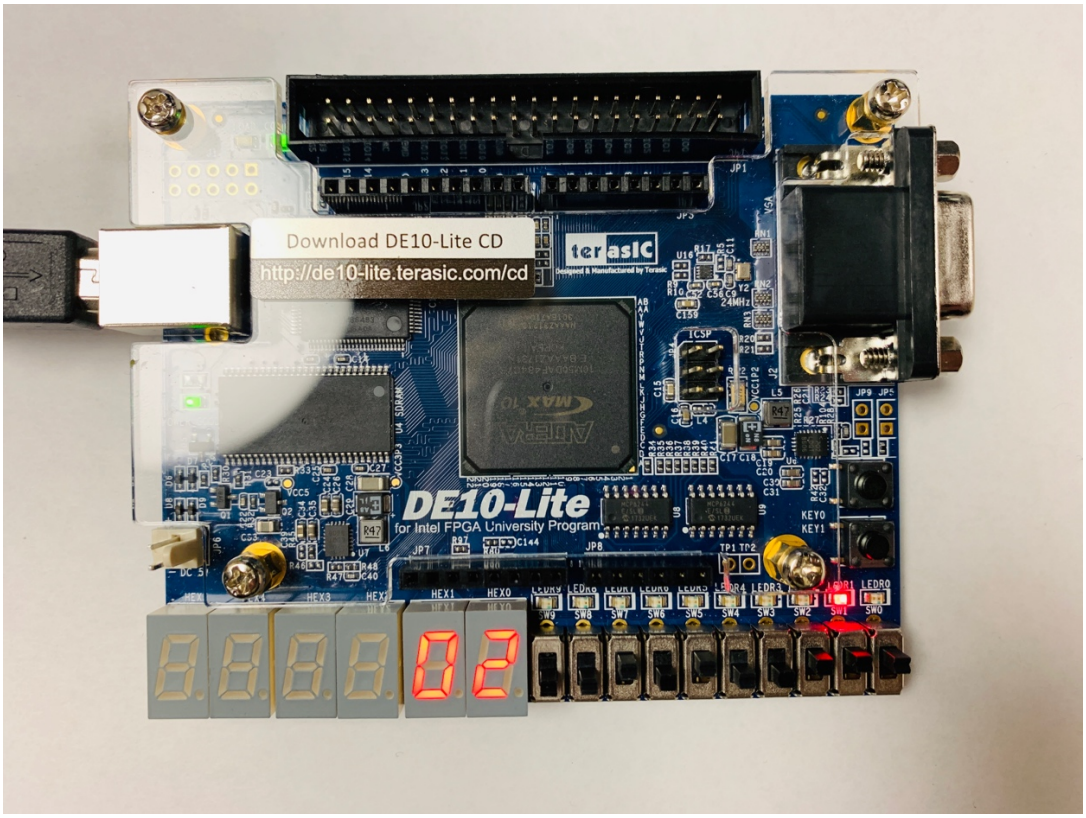

Figure 10: The result of x AND y

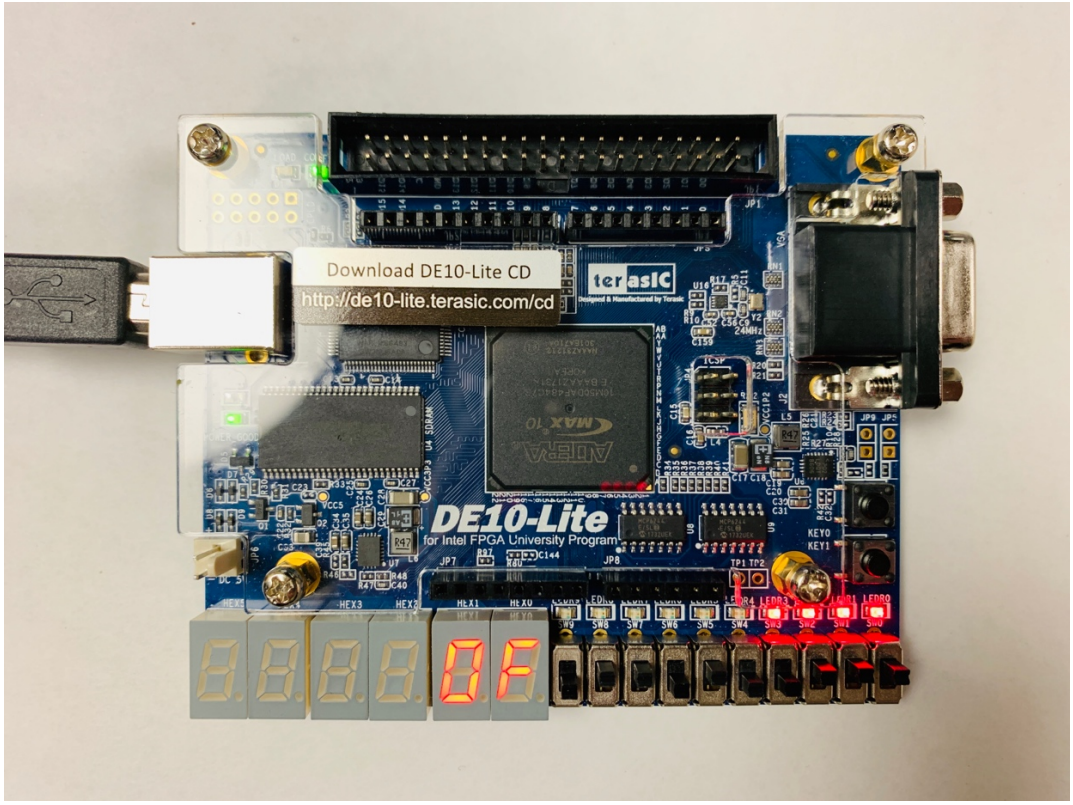

Figure 11: The result of x OR y

Figure 12: The result of x XOR y

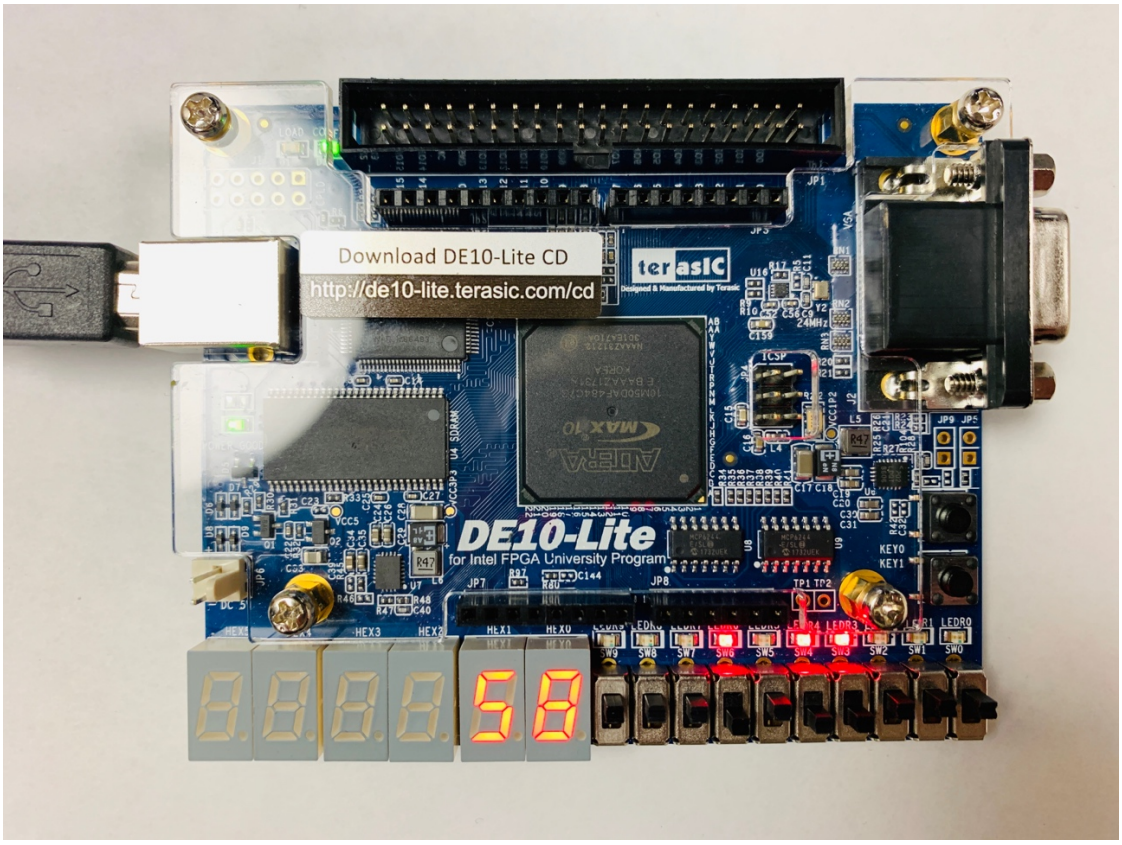

Figure 13: The result of $\overline{z}$ with switch inputs of z

Comparison

The comparison mode includes equal-to, greater-than, less-than, and maximum operations that can be selected using SW9 and SW8 according to Table 4 in Figure 2 above. The same hexadecimal values of $x = 0x8$ and $y = 0x7$ were chosen to test this module's outputs:

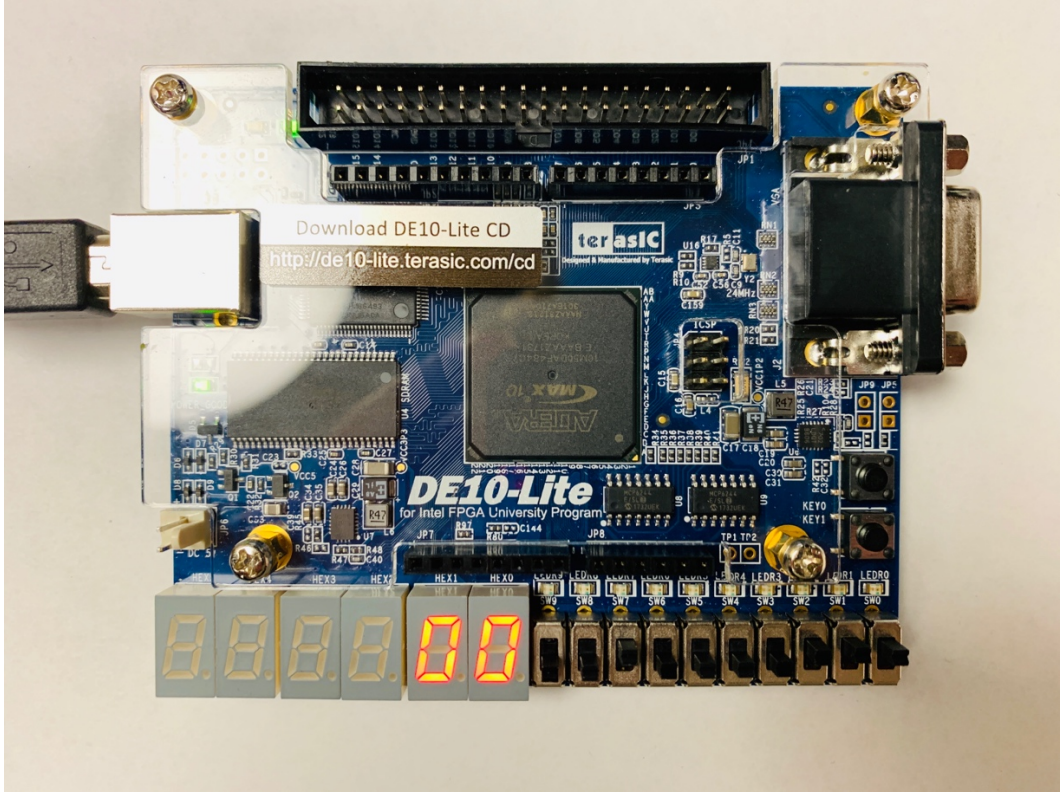

Figure 14: The result of the equal-to comparison ($x = y$) between $x = 0x8$ and $y = 0x7$ with output of 0 (not equal)

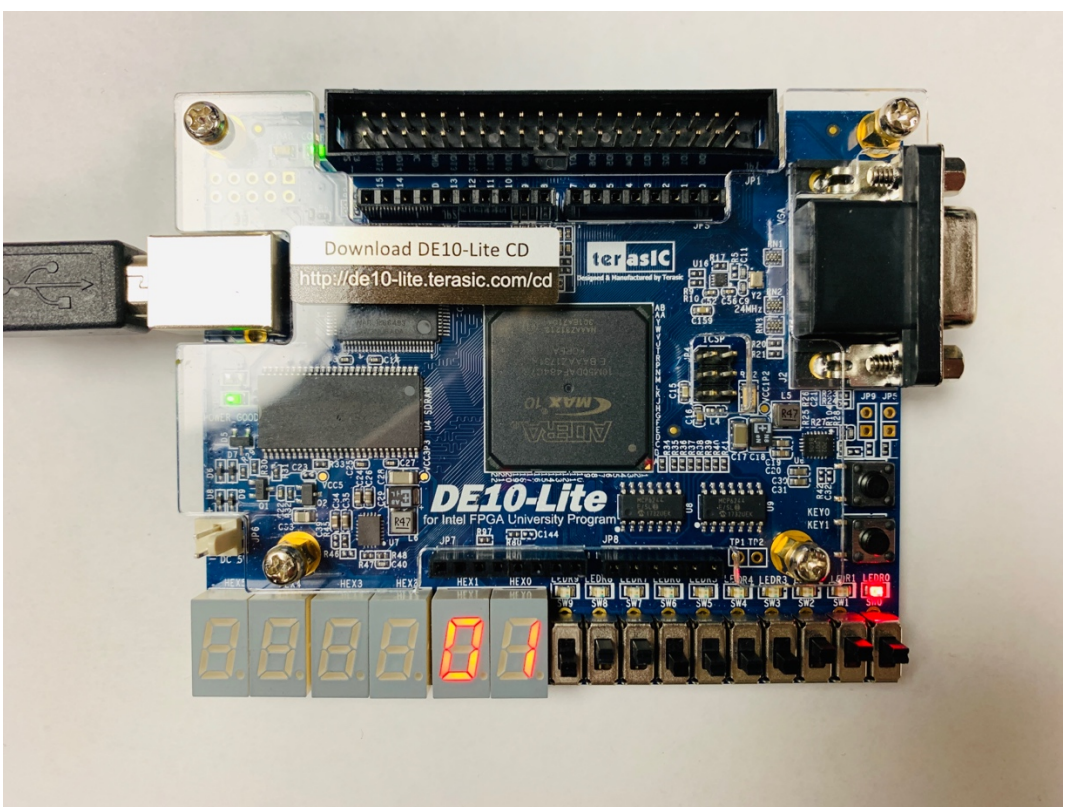

Figure 15: The result of the greater-than comparison ($x > y$) between $x = 0x8$ and $y = 0x7$ with output of 1 (x greater than y)

Figure 16: The result of the less-than comparison ($x < y$) between $x = 0x8$ and $y = 0x7$ with output of 0 (x not greater than y)

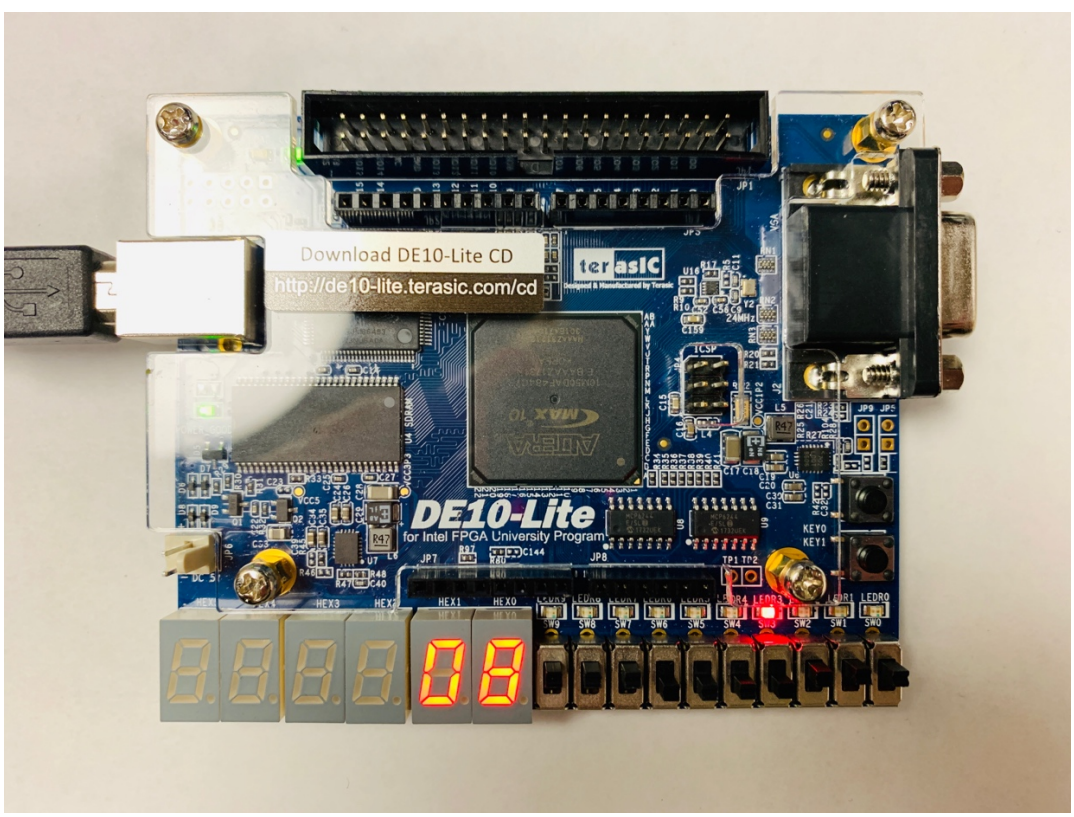

Figure 17: The result of the maximum between $x = 0x8$ and $y = 0x7$ with output of 8 (the maximum of 7 and 8)

Multiplexer

As coded in Appendix A, the multiplexer module simply takes in four inputs and uses two select bits to choose an output. The multiplexer was utilized in other modules to select between inputs, such as in Project1_Top to select which input outputs to the seven-segment display.

Conclusion

The purpose of this project was to create a miniature CPU that can perform basic arithmetic, comparison, and logical operations on the MAX 10 FPGA Verilog and Quartus Prime. Though difficult at first to visualize how each individual logical component or Verilog expression fits in with each other, each operation fell into its right place in the end. The CPU could be improved in the future by implementing a more intuitive way of switching modes with, for instance, a reset button and an incrementation button to navigate the different modes. All in all, it was fascinating to take digital logic ideology and apply it to a small-scale version of something many of us use every day without thinking twice about the detailed work that went into designing it.

Appendix A

Verilog Code

```
// Connor Humiston
// ECEN2350 Digital Logic
// University of Colorado, Boulder

module Project1_Top(KEY, SW, HEX0, HEX1, LEDR); //added LEDR
	input [1:0] KEY;
	input [9:0] SW;
	output [7:0] HEX0, HEX1;
	output [9:0] LEDR;

	wire [9:0] muxout, arithout1, logicout2, compout3, muxout4;
	wire [7:0] compare;
	wire [7:0] arithrslt;
	wire [7:0] logrslt;
	reg K0, K1;

	//x = SW[7:4] aka first nibble
	//y = SW[3:0] aka second nibble

	always @(posedge KEY[0]) //The button keys on my board were very finickey and did
not function correctly on negedge.
	begin
		if (KEY[0] == 1)
			K0 = K0;
		else
			K0 = ~K0;
	end

	always @(posedge KEY[1]) //The button keys on my board were very finickey and did
not function correctly on negedge.
	begin
		if (KEY[1] == 1)
			K1 = K1;
		else
			K1 = ~K1;
	end

	Arithmetic(SW[7:4], SW[3:0], SW[9], SW[8], addrslt, subrslt, multiplyrslt, divrslt,
arithout1[9:0]);
	Logical(SW[7:4], SW[3:0], SW[9], SW[8], andout, orout, exout, notout, logicout2[9:0]);
	Comparison(SW[7:4], SW[3:0], SW[9], SW[8], eqout, grout, lsout, maxout,
compout3[9:0]);
```

```
	Multiplexer mult1(arithout1[9:0], arithout1[9:0], logicout2[9:0], compout3[9:0], K0, K1,
muxout[9:0]);
	SevenSegment sevenout1(muxout[9:0], HEX0[7:0], HEX1[7:0]);

	assign LEDR[9:0] = muxout[9:0];
endmodule
```

```
// Connor Humiston
// ECEN2350 Digital Logic
// University of Colorado, Boulder

module SevenSegment(binaryinput, HEX0, HEX1);
	input [9:0] binaryinput;
	output reg [7:0] HEX0, HEX1;

	always @(binaryinput)
	begin
		case (binaryinput[3:0])
		4'b0000: HEX0[6:0] = 7'b1000000; //if 0 desired, turns on all but g in gfedcba
		4'b0001: HEX0[6:0] = 7'b1111001; //1
		4'b0010: HEX0[6:0] = 7'b0100100; //2
		4'b0011: HEX0[6:0] = 7'b0110000; //3
		4'b0100: HEX0[6:0] = 7'b0011001; //4
		4'b0101: HEX0[6:0] = 7'b0010010; //5
		4'b0110: HEX0[6:0] = 7'b0000010; //6
		4'b0111: HEX0[6:0] = 7'b1111000; //7
		4'b1000: HEX0[6:0] = 7'b0000000; //8
		4'b1001: HEX0[6:0] = 7'b0010000; //9
		4'b1010: HEX0[6:0] = 7'b0001000; //A
		4'b1011: HEX0[6:0] = 7'b0000011; //B
		4'b1100: HEX0[6:0] = 7'b1000110; //C
		4'b1101: HEX0[6:0] = 7'b0100001; //D
		4'b1110: HEX0[6:0] = 7'b0000110; //E
		4'b1111: HEX0[6:0] = 7'b0001110; //F
		default: HEX0[6:0] = 7'b0111111; //default/error signal is a dash
		endcase
	end

	always @(binaryinput)
	begin
		case(binaryinput[7:4])
		4'b0000: HEX1[6:0] = 7'b1000000; //0
		4'b0001: HEX1[6:0] = 7'b1111001; //1
		4'b0010: HEX1[6:0] = 7'b0100100; //2
		4'b0011: HEX1[6:0] = 7'b0110000; //3
		4'b0100: HEX1[6:0] = 7'b0011001; //4
		4'b0101: HEX1[6:0] = 7'b0010010; //5
		4'b0110: HEX1[6:0] = 7'b0000010; //6
		4'b0111: HEX1[6:0] = 7'b1111000; //7
		4'b1000: HEX1[6:0] = 7'b0000000; //8
		4'b1001: HEX1[6:0] = 7'b0010000; //9
		4'b1010: HEX1[6:0] = 7'b0001000; //A
		4'b1011: HEX1[6:0] = 7'b0000011; //B
```

```
		4'b1100: HEX1[6:0] = 7'b1000110; //C
		4'b1101: HEX1[6:0] = 7'b0100001; //D
		4'b1110: HEX1[6:0] = 7'b0000110; //E
		4'b1111: HEX1[6:0] = 7'b0001110; //F
		default: HEX1[6:0] = 7'b0111111; //default/error signal is a dash
		endcase
	end


	always @(binaryinput)
	begin
		case(binaryinput[9])				//for division
		1'b1: HEX0[7] = 1'b0;				//decimal point on
		1'b0: HEX0[7] = 1'b1;				//decimal off
		endcase
	end


	always @(binaryinput)
	begin
		case(binaryinput[8])				//for multiplication
		1'b1: HEX1[7] = 1'b0;				//decimal point on
		1'b0: HEX1[7] = 1'b1;				//decimal off
		endcase
	end
endmodule
```

```
// Connor Humiston
// ECEN2350 Digital Logic
// University of Colorado, Boulder

module Arithmetic(x, y, SW9, SW8, addrslt, subrslt, multiplyrslt, divrslt, arithrslt);
	input [3:0] x, y;
	input SW8, SW9;
	output [9:0] addrslt, subrslt, multiplyrslt, divrslt, arithrslt;

	addern rca(x, y, 0, addrslt);
	full_subtract fs(x, y, 1, subrslt);
	multiply_by_two mbt({x, y}, carrym, multiplyrslt);
	divide_by_two dbt({x, y}, divrslt[9], divrslt[8:0]);

	//for SW9 & SW8
	//00 is addition
	//01 is subtraction
	//10 is multiply by 2
	//11 is divide by 2
	Multiplexer op_arith(addrslt, subrslt, multiplyrslt, divrslt, SW9, SW8, arithrslt);
endmodule


//n-bit ripple carry adder
module addern (x, y, Cin, sum);
	parameter n = 4;
	input [n-1:0] x, y;
	input Cin;
	output [4:0] sum;
	wire [4:0] C;

	assign C[0] = Cin;
	assign sum[n] = C[n];

	genvar i;
	generate
		for (i = 0; i < n; i = i + 1)
		begin: fulladd
			fulladder fa1(x[i], y[i], C[i+1], C[i], sum[i]);
		end
	endgenerate
	assign Cout = C[n];
endmodule


//n-bit full subtractor
```

```
module full_subtract(x, y, Cin, difference);
	parameter n = 4;
	input [n-1:0] x, y;
	input Cin;
	output [4:0] difference;
	wire [4:0] C;

	assign C[0] = Cin;
	assign difference[n] = ~C[n];

	genvar i;
	generate
		for (i = 0; i < n; i = i + 1)
		begin: fullsub
			fulladder fa2(x[i], ~y[i], C[i+1], C[i], difference[i]);
		end
	endgenerate

	assign Cout = C[n];
endmodule


//multiply-by-2
module multiply_by_two(z, carrym, product);
	input [8:0] z;
	output [8:0] product;
	output carrym;

	assign carrym = z[8];
	assign product = z << 1;
endmodule


//divide-by-2
module divide_by_two(z, carryq, quotient);
	input [7:0] z;
	output [9:0] quotient;
	output carryq;

	assign carryq = z[0];
	assign quotient = z >> 1;
endmodule

//full adder
module fulladder(x, y, Cout, Cin, sum);
	input x, y, Cin;
```

```
	output Cout, sum;

	assign sum = x ^ y ^ Cin;
	assign Cout = (x & y) | (y & Cin) | (x & Cin);
endmodule
```

```
// Connor Humiston
// ECEN2350 Digital Logic
// University of Colorado, Boulder

module Logical (x, y, SW9, SW8, andout, orout, xorout, notout, logrslt);
	input [3:0] x, y;
	input SW9, SW8;
	output [9:0] andout, orout, xorout, notout;
	output [9:0]logrslt;

	and_gate and1(x, y, andout);
	or_gate or1(x, y, orout);
	xor_gate exor1(x, y, xorout);
	not_gate not1(x, y, z, notout);

	//for SW9 & SW8
	//00 is for AND
	//01 is for OR
	//10 is for XOR
	//11 is for NOT
	Multiplexer op_log(andout, orout, xorout, notout, SW9, SW8, logrslt);
endmodule


//AND
module and_gate(x, y, andout);
	input [3:0] x, y;
	output [3:0] andout;

	assign andout = x & y;
endmodule


//OR
module or_gate(x, y, orout);
	input [3:0] x, y;
	output [3:0] orout;

	assign orout = x | y;
endmodule


//XOR
module xor_gate(x, y, exorout);
	input [3:0] x, y;
	output[3:0] exorout;
```

```
	assign exout = x ^ y;
endmodule


//NOT
module not_gate(x, y, z, notout);
	input [3:0] x, y;
	output [7:0] z;
	output [7:0] notout;

	assign z = {x, y};
	assign notout = ~z;
endmodule
```

```
// Connor Humiston
// ECEN2350 Digital Logic
// University of Colorado, Boulder

module Comparison(x, y, SW9, SW8, eqrslt, grrslt, lsrslt, mxout, compare);
	input [3:0] x, y;
	input SW9, SW8;
	output [9:0] eqrslt, grrslt, lsrslt, mxout, compare;

	equal eql(x, y, eqrslt);
	greaterthan grt(x, y, grrslt);
	lessthan less(x, y, lsrslt);
	max mx(x, y, mxout);

	//for SW9 & SW8
	//00 is for equal
	//01 is for greater than
	//10 is for less than
	//11 is for max
	Multiplexer op_comp(eqrslt, grrslt, lsrslt, mxout, SW9, SW8, compare);
endmodule


//EQUAL
//outputs 1 if equal and 0 if not equal
module equal(x, y, eqrslt);
	input [3:0] x, y;
	output reg [3:0] eqrslt;

	always @(x, y, eqrslt)
		begin
			if (x == y)
				eqrslt = 1;
			else
			eqrslt = 0;
		end
endmodule


//GREATER
//outputs 1 if x > y and 0 if y >= x
module greaterthan(x, y, grrslt);
	input [3:0] x, y;
	output reg [3:0] grrslt;

	always @(x, y, grrslt)
```

```
		begin
			if (x > y)
				grrslt = 1;
			else
				grrslt = 0;
		end
endmodule


//LESS
//outputs 1 if x < y and 0 if y <= x
module lessthan(x, y, lsrslt);
	input [3:0] x, y;
	output reg [3:0] lsrslt;

	always @(x, y, lsrslt)
		begin
			if (x < y)
				lsrslt = 1;
			else
				lsrslt = 0;
		end
endmodule


//MAX
//outputs x if x is maximum and y if y is maximum
module max(x, y, mxout);
	input [3:0] x, y;
	output reg [3:0] mxout;

	always @(mxout)
		begin
			begin
				if (x[3] == 1 & y[3] == 0)
					mxout = x;
			end
			begin
				if (y[3] == 1 & x[3] == 0)
					mxout = y;
			end
			begin
				if (x[3] == y[3] & x[2] == 1 & y[2] == 0)
					mxout = x;
			end
			begin
```

```
                        if (x[3] == y[3] & x[2] == 0 & y[2] == 1)
                                mxout = y;
                end
                begin
                        if (x[3] == y[3] & x[2] == y[2] & x[1] == 1 & y[1] == 0)
                                mxout = x;
                end
                begin
                        if (x[3] == y[3] & x[2] == y[2] & x[1] == 0 & y[1] == 1)
                                mxout = y;
                end
                begin
                        if (x[3] == y[3] & x[2] == y[2] & x[1] == y[1] & x[0] == 1 & y[0]
== 0)
                                mxout = x;
                end
                begin
                        if (x[3] == y[3] & x[2] == y[2] & x[1] == y[1] & x[0] == 0 &
y[0]== 1)
                                mxout = y;
                end
        end
endmodule
```

```
// Connor Humiston
// ECEN2350 Digital Logic
// University of Colorado, Boulder

module Multiplexer (a, b, c, d, select0, select1, multplxout);
	input [9:0] a, b, c, d;
	input select0, select1;
	output reg [9:0] multplxout;

	always @ (select0, select1)
		case ({select0, select1})
		2'b00: multplxout = a;
		2'b01: multplxout = b;
		2'b10: multplxout = c;
		2'b11: multplxout = d;
	endcase
endmodule
```