\

Digital Design Lab
EEN 316

Lab #4
Calculator

Nathan Paternoster

Sayan Maity, TA

University of Miami

4/14/14

Abstract

In this lab we had to design and implement a 16-bit calculator. The calculator consists of the main calculator kernel followed by four output registers and tri-state buffers. Therefore we will have to design both the register and tri-state buffer as well. These components will operate on 4 bits each. The calculator will accept an operation code (opc) as an input to determine which function the calculator will perform. Our lab will first be designed and simulated in ModelSim and then downloaded into Quartus and simulated on the Altera DE-2 board.

Table of Contents

Overview

The two components (the 4-bit register and 4-bit tri-state buffer) will be designed algorithmically using the behavioral method. The top-level calculator entity will then be designed using the previous two components with the structural method. The calculator entity will also do the operation calculations within a process. The calculator entity will take 16-bit inputs X and Y and a 5-bit operation code and output a 16-bit result. The result will be divided between four 4-bit registers. The four registers will be fed to a set of four TSBs, which will all output to the calculator's 4-bit output. Only one TSB may be enabled at one time (so we are only able to view one nibble of the result at any time).

The design will then be simulated on ModelSim to ensure that our code will successfully perform the required operations. Once a few instructions have been tested we will write a do-script to be able to simulate these instructions at a future time. The next step will be to download the code into Quartus and the Altera DE-2 board. We will perform the same operations on the DE-2 board to be able to visualize our results.

Objectives

To understand the uses of designing from the top-level down and build from the bottom-level up. To successfully implement and visualize a calculator using VHDL and a digital logic board.

Description

The first step to design this circuit is to design its two components. The register will take four input bits and output four bits. It is a sequential element so it will run on the clock. It will also accept a load line that, when set to '1', will store its input into its internal "storage" signal. The register will store its input to its storage on the rising edge of the clock signal (assuming load is set to '1') and will output data from its storage to the output on the falling edge of the clock. It will be designed using a process with the clock in its sensitivity list (algorithmic behavioral method).

The tri-state buffer (TSB) will also take four input bits and output four bits. It is *not* a sequential element so it will not depend on the clock. Its only other input will be a control (ctr) line that, when set to '1', will send its input to its output, and, when set to '0', will output nothing ("ZZZZ"). It will also be designed using a process with ctr in its sensitivity list (algorithmic behavioral method).

The second step is to design the top-level calculator entity using the structural method. The calculator will accept two 16-bit inputs (X and Y), four control lines, four load lines, clock, operation code (opc), and will output into a 4-bit output. The calculator entity will declare two 16-bit internal signals: result and register output. It will first define a process with X, Y, and the opc as sensitivity list arguments to perform the calculation based on the specified opc. It will place the result of this operation into the result signal. Then four instances of the register component will be declared, and the correct portions of the result placed into each register. These instances will place their outputs into the register output internal signal. Finally, four TSB instances will be declared to connect the registers' outputs to the calculator's output.

Design Synthesis

Operation Code

1 0 0 0 0	Y	Clear when y = 0
1 0 0 0 1	Y + 1	PASS 1 when y = 0
1 0 0 1 1	X + Y	X when y = 0
1 0 1 0 0	NOT Y	
1 0 1 0 1	– Y	
1 0 1 1 1	Y – X	
1 1 0 0 0	Y – 1	PASS –1 when y = 0
1 1 0 0 1	X – Y	
1 1 0 1 1	NOT X	
1 1 1 0 0	X AND Y	
1 1 1 0 1	X OR Y	
1 1 1 1 0	X XOR Y	

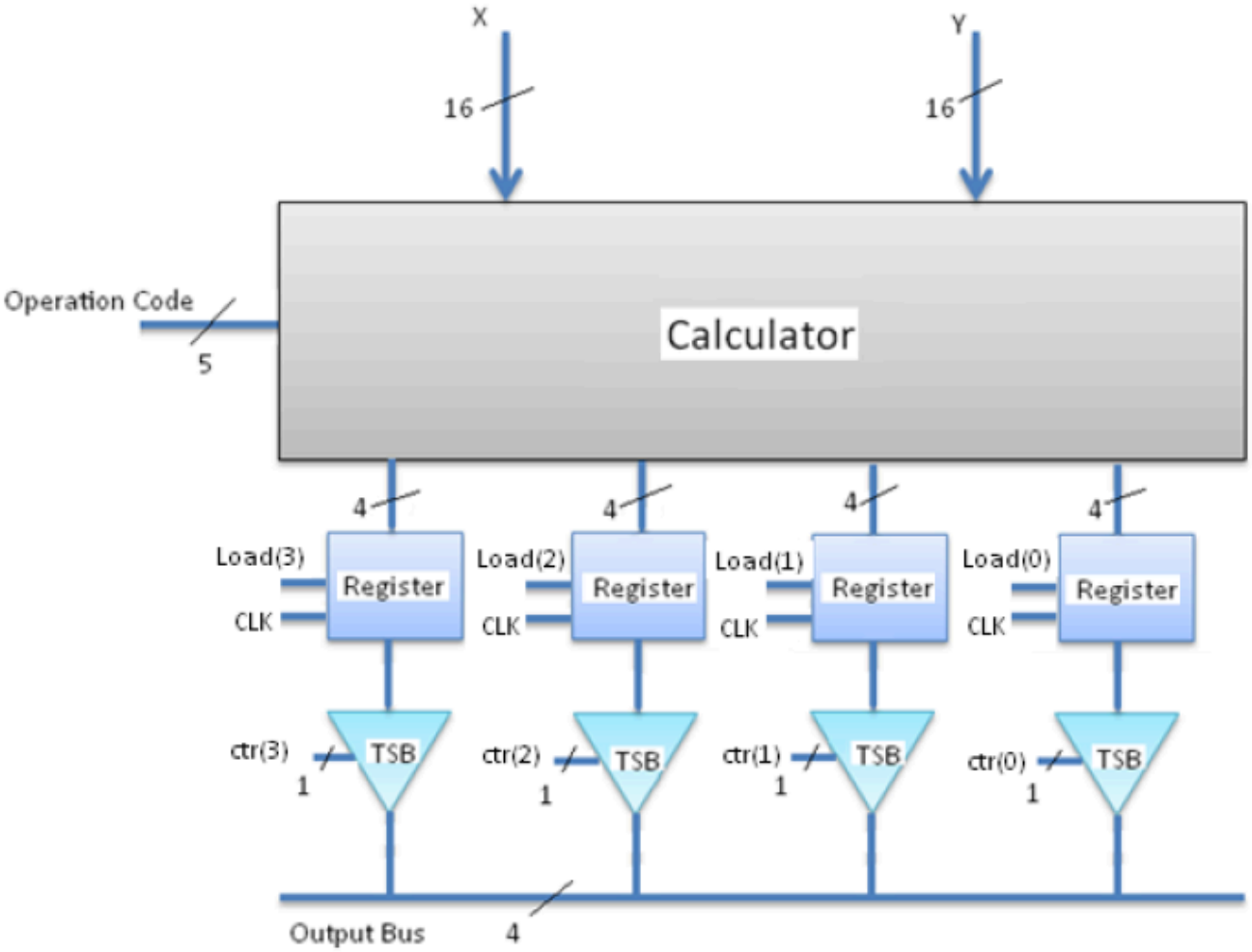
The operations our calculator will support. –Y will be achieved by subtracting Y from 0.

Interface Specification

Name	Bit-Length	Mode	Description
X	16	Input	X operand
Y	16	Input	Y operand
Clk	1	Input	The clock signal for registers
Load	4	Input	Load signals for registers
Ctr	4	Input	Control signals for tri-state buffer
OPC	5	Input	Operation Code
Output	4	Output	Output bus

The ports of the top-level calculator entity.

Complete Logic Diagram



Code

Calculator Top-Level Entity

```
library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;

entity sixteenbitcalculator is
	port(X, Y : in std_logic_vector(15 downto 0);
		clk : in std_logic;
		load, ctr : in std_logic_vector(3 downto 0);
		OPC : in std_logic_vector(4 downto 0);
		output : out std_logic_vector(3 downto 0));
end entity;

architecture structural of sixteenbitcalculator is
	component fourbitreg
		port(input : in std_logic_vector(3 downto 0);
			clk, load : in std_logic;
			output : out std_logic_vector(3 downto 0));
	end component;
	component fourbitbuffer
		port(data : in std_logic_vector(3 downto 0);
			ctr : in std_logic;
			output : out std_logic_vector(3 downto 0));
	end component;
	signal result : std_logic_vector(15 downto 0);
	signal regout : std_logic_vector(15 downto 0);
begin
	calculatorprocess: process(X, Y, OPC)
	begin
		if (OPC = "10000") then
			if (Y = "0") then
				result <= "0000000000000000";
			else
				result <= Y;
			end if;
		elsif (OPC = "10001") then
			if (Y = "0") then
				result <= "0000000000000001";
			else
				result <= Y + "1";
			end if;
		elsif (OPC = "10011") then
			if (Y = "0") then
				result <= X;
```

```
                else
                        result <= X + Y;
                end if;
        elsif (OPC = "10100") then
                result <= NOT Y;
        elsif (OPC = "10101") then
                result <= "0" - Y;
        elsif (OPC = "10111") then
                result <= Y - X;
        elsif (OPC = "11000") then
                if (Y = "0") then
                        result <= "1000000000000000";
                else
                        result <= Y - "1";
                end if;
        elsif (OPC = "11001") then
                result <= X - Y;
        elsif (OPC = "11011") then
                result <= NOT X;
        elsif (OPC = "11100") then
                result <= X AND Y;
        elsif (OPC = "11101") then
                result <= X OR Y;
        elsif (OPC = "11110") then
                result <= X XOR Y;
        end if;
end process;

Reg0 : fourbitreg port map(result(3 downto 0), clk, load(0), regout(3 downto 0));
Reg1 : fourbitreg port map(result(7 downto 4), clk, load(1), regout(7 downto 4));
Reg2 : fourbitreg port map(result(11 downto 8), clk, load(2), regout(11 downto 8));
Reg3 : fourbitreg port map(result(15 downto 12), clk, load(3), regout(15 downto 12));

Buff0 : fourbitbuffer port map(regout(3 downto 0), ctr(0), output);
Buff1 : fourbitbuffer port map(regout(7 downto 4), ctr(1), output);
Buff2 : fourbitbuffer port map(regout(11 downto 8), ctr(2), output);
Buff3 : fourbitbuffer port map(regout(15 downto 12), ctr(3), output);

end architecture;
```

4-bit Register Component

```
library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;

entity fourbitreg is
	port(input : in std_logic_vector(3 downto 0);
		clk, load : in std_logic;
		output : out std_logic_vector(3 downto 0));
end entity;

architecture algorithmic of fourbitreg is
	signal storage : std_logic_vector(3 downto 0);
begin
	regprocess: process(clk)
	begin
		if (clk'event and clk = '1' and load = '1') then
			storage <= input;
		elsif (clk'event and clk = '0') then
			output <= storage;
		end if;
	end process;
end architecture;
```

4-bit Tri-State Buffer Component

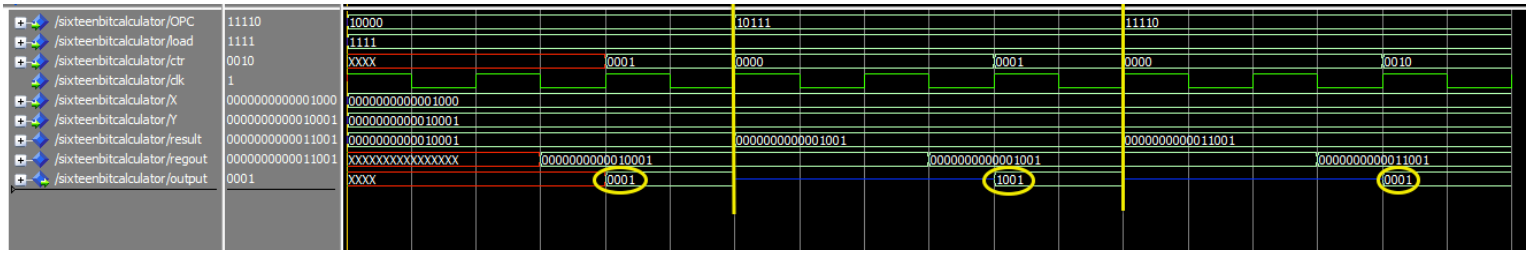
```
library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;

entity fourbitbuffer is
	port(data : in std_logic_vector(3 downto 0);
				ctr : in std_logic;
				output : out std_logic_vector(3 downto 0));
end entity;

architecture algorithmic of fourbitbuffer is
begin
	buffprocess: process(ctr)
	begin
		if (ctr = '1') then
			output <= data;
		elsif (ctr = '0') then
			output <= "ZZZZ";
		end if;
	end process;
end architecture;
```

Results and Simulations

ModelSim



Three op-codes were tested. Our process for simulating started with loading the op-code, loading the X and Y inputs, setting the control lines to 0000, and setting the load lines to 1111. The first clock cycle will load the computations result into our internal signal (result). The second will pass these results into the register output (since the load lines are all set to 1). We will then set the control lines to whichever register we want to put to the output and perform one final clock cycle. The final answer (circled) will be in the output.

Simulation 1: Operation = Y, ctr = 0001

- X = 0000 0000 0000 1000
- Y = 0000 0000 0001 0001
- Output = 0001 (same as last four bits of Y)

Simulation 2: Operation = Y – X, ctr = 0001

- X = 0000 0000 0000 1000 (8)
- Y = 0000 0000 0001 0001 (17)
- Output = 1001 (9)

Simulation 3: Operation = X xor Y, ctr = 0010

- X = 0000 0000 **0000** 1000
- Y = 0000 0000 **0001** 0001
- Output = 0001

Do script

```
vsim -gui work.sixteenbitcalculator

add wave  \
sim:/sixteenbitcalculator/OPC \
sim:/sixteenbitcalculator/load \
sim:/sixteenbitcalculator/ctr \
sim:/sixteenbitcalculator/clk \
sim:/sixteenbitcalculator/X \
sim:/sixteenbitcalculator/Y \
sim:/sixteenbitcalculator/result \
sim:/sixteenbitcalculator/regout \
sim:/sixteenbitcalculator/output \

force -freeze sim:/sixteenbitcalculator/clk 1 0, 0 {50 ns} -r 100
force -freeze sim:/sixteenbitcalculator/Y 0000000000010001 0
force -freeze sim:/sixteenbitcalculator/X 0000000000001000 0
force -freeze sim:/sixteenbitcalculator/load 1111 0

# Simulation 1, Operation = Y
force -freeze sim:/sixteenbitcalculator/OPC 10000 0
run
run
force -freeze sim:/sixteenbitcalculator/ctr 0001 0
run
force -freeze sim:/sixteenbitcalculator/ctr 0000 0

# Simulation 2, Operation = Y - X
force -freeze sim:/sixteenbitcalculator/OPC 10111 0
run
run
force -freeze sim:/sixteenbitcalculator/ctr 0001 0
run
force -freeze sim:/sixteenbitcalculator/ctr 0000 0

# Simulation 3, Operation = X xor Y
force -freeze sim:/sixteenbitcalculator/OPC 11110 0
run
run
force -freeze sim:/sixteenbitcalculator/ctr 0010 0
run
```

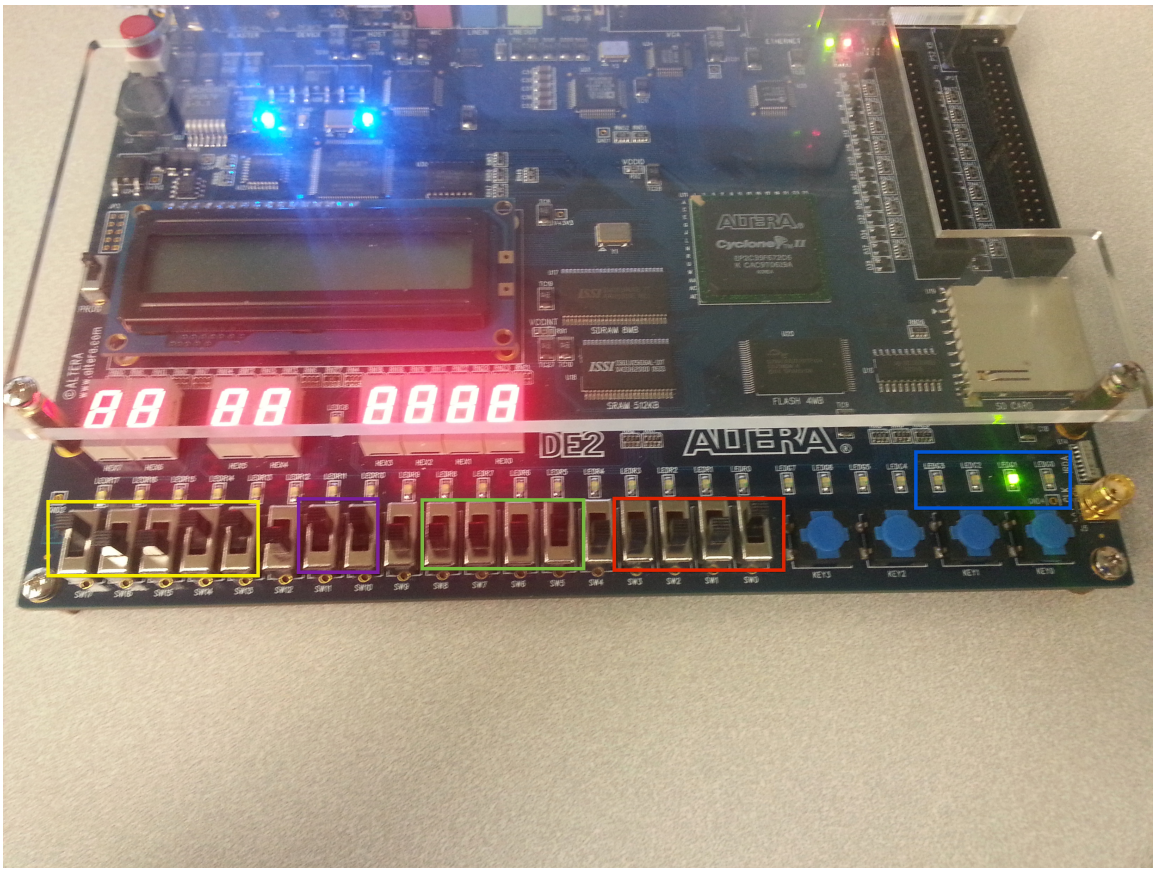
Quartus

To visualize the calculator on the Altera DE-2 board, I had to scale down the inputs to only 4 bits each. This design required only one load and one select line. I performed three calculations to verify our design.

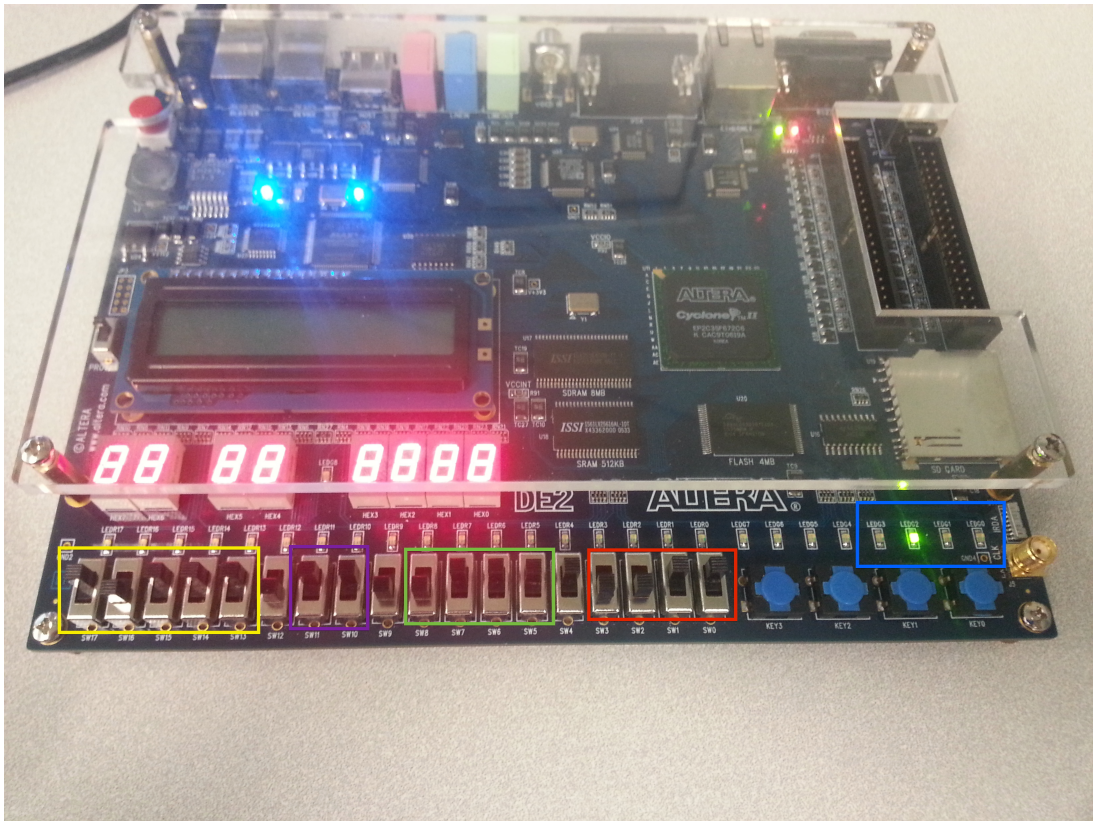
Key

- Yellow – Opcode
- Purple – Load/Select line (always set to 1)
- Green – Y
- Red – X
- Blue – Output

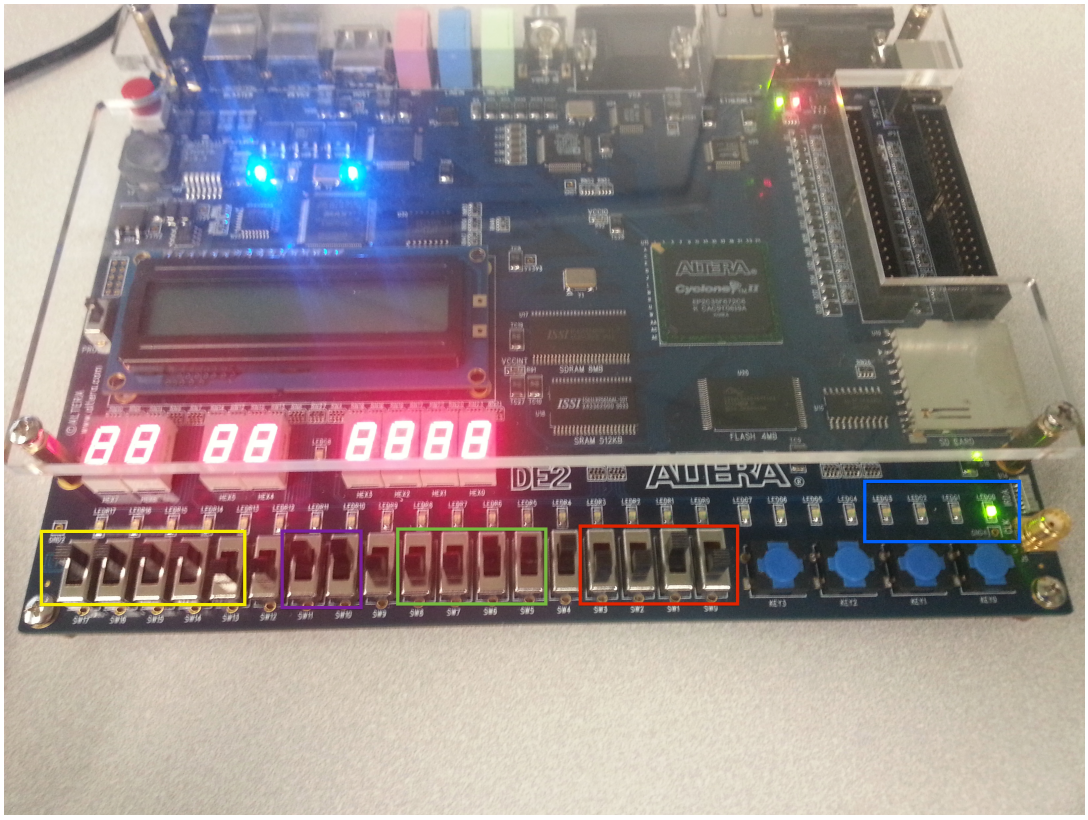
Simulation 1: 1+1 = 2



Simulation 2: 7-3 = 4



Simulation 3: 3 XOR 2 = 1



Conclusion

The calculator was successfully implemented and the computations were shown to be correct on both a waveform simulation and a digital board simulation. I learned how to build more complex digital systems by designing both behavioral and structural entities.

Signed OFF

[Included]