Digital Systems Design

Engr 378

Lab 2

Arithmetic Circuit. Behavioral, Dataflow, and Structural Descriptions

Date:
2/23/21

Grade:

By:
Aaron Luong and Richard Couto

Problem Analysis

We need to have Verilog code for a simple, four-bit arithmetic circuit so we can perform 4 unique operations if given two input numbers, a carry bit, as well as a state selection. This will involve modeling the logic gates and other logic components that would provide correct and reliable output. Then those gates and components will be transferred into Verilog. This will be tested with a testbench module in ModelSim to verify that the code delivers the expected waveforms and outputs. The following are the expected functions to be performed.

S1	S0	Cin=0	Cin=1
0	0	$F = A + B$ *(Addition w/o carry)*	$F = A + B + 1$ *(Addition with carry)*
0	1	$F = A + \overline{B}$	$F = A + \overline{B} + 1$ *(Substract: A-B)*
1	0	$F = \overline{A} + B$	$F = \overline{A} + B + 1$ *(Substract: B-A)*
1	1	$F = \overline{B}$ *(1's Complement of B)*	$F = \overline{B} + 1$ *(2's Complement of B)*

Fig 1: Arithmetic Function

Hardware Design

Two methods were used to come to a design for this arithmetic circuit. One which began with K-maps and another by observing the question. It was apparent an adder was needed as well that the Cin value was a toggle and inverted out outputs. A's inversion was connected to S1 and B's was connected to S0, which laid the groundwork for which values would be connected into 2x1 mux before the desired 4-bit A and B are fed into our final Full Adder. The K-map plan had several 3-input AND gates which all fed into a single or gate that would give an output a single bit at a time with a separate, adder-like circuit to account for the carry.

Upon writing some code with Verilog, and understanding that there are pre-existing modules present, we elected to adopt the more intuitive design alongside some advice from the professor. Here we continued to use a mux, but a 4x1 instead of a 2x1, and here we will have Full adders that deliver the output on a 4bit line into the Mux and allow that to be driven by our S1 and S0. All three designs are shown below.

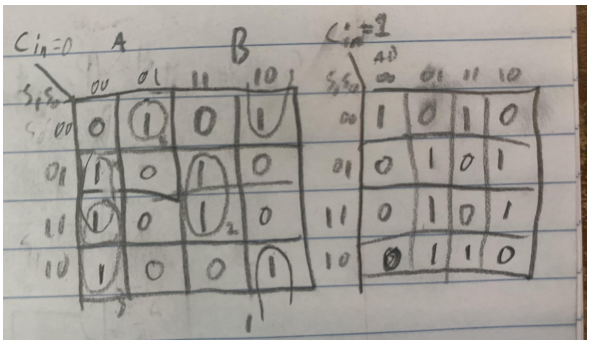

Fig 2: K-map

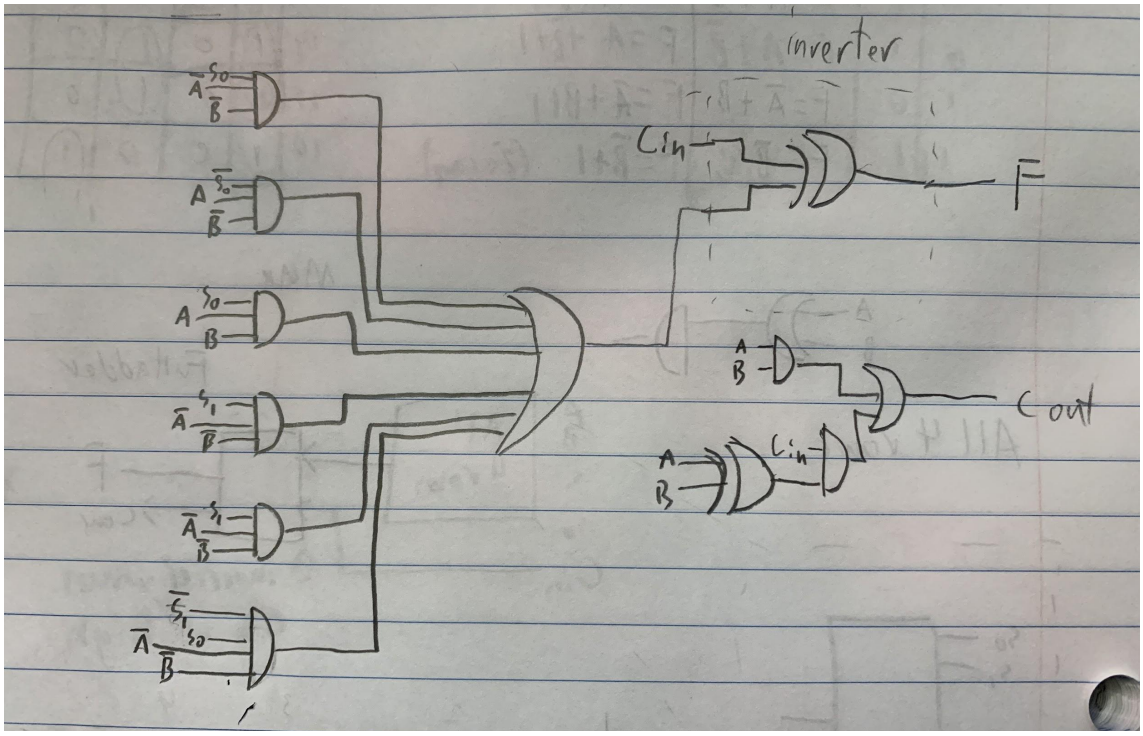

Fig 3: K-map Circuit

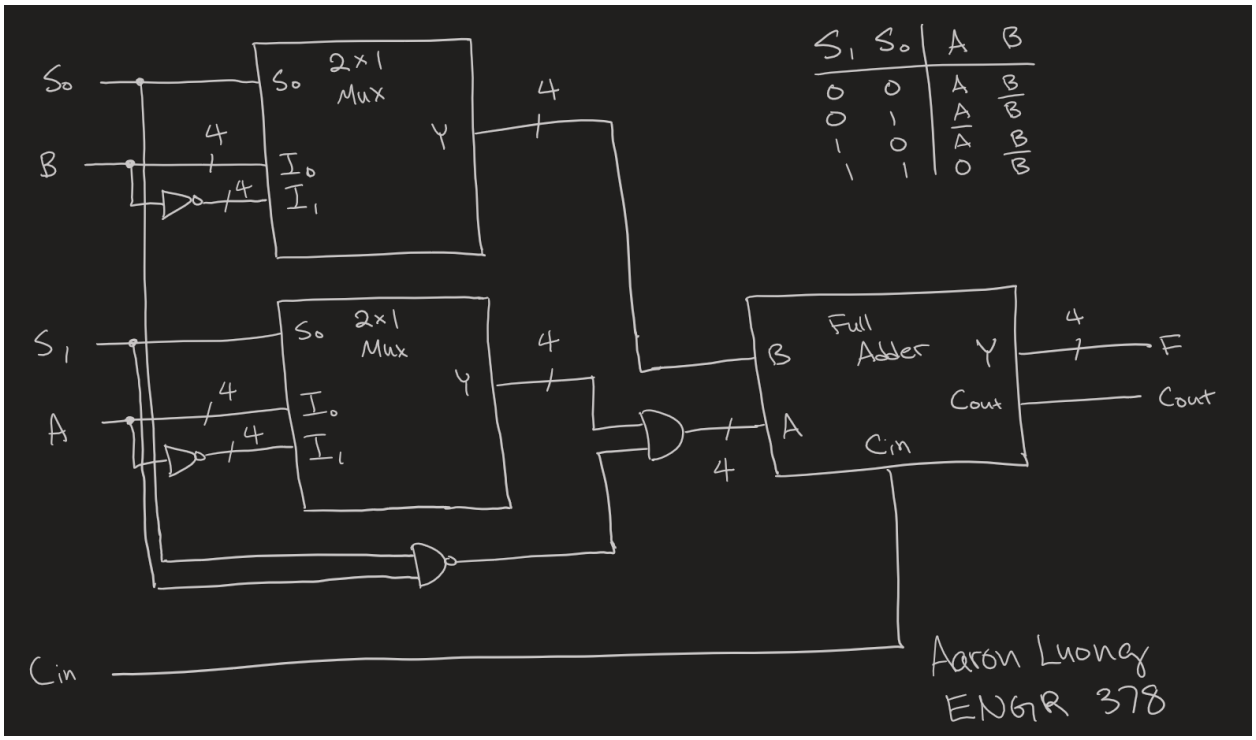

Fig 4: Two 2x1 Muxs and Full Adder

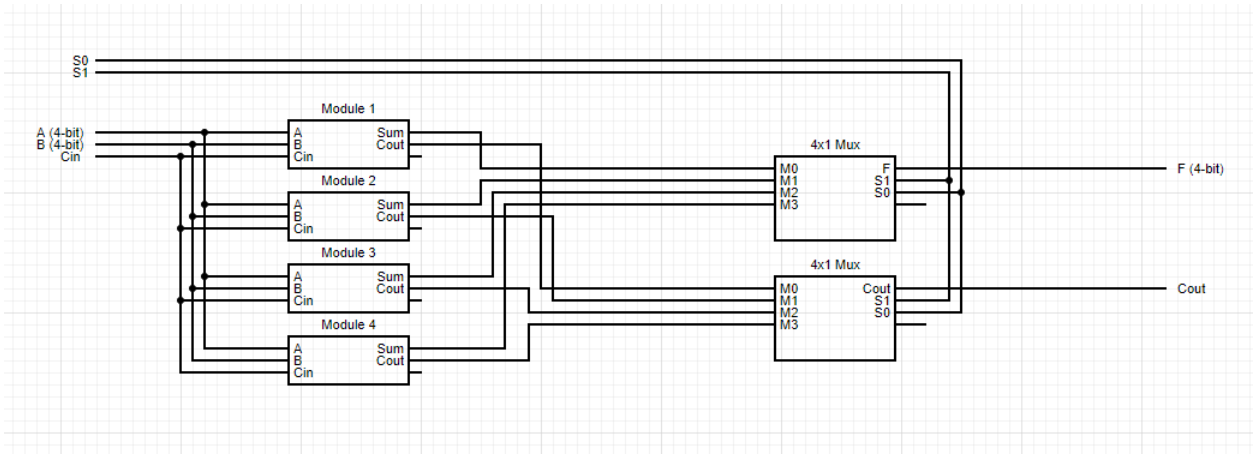

Fig 5: Final Design

Verilog Modeling

We began by breaking up the first 2 modules between us as we played with Verilog to construct full adders without errors. They all needed A, B, Cin, S1, S0, and Cout to be declared with their respective input and output label. All the modules look roughly the same but the part going into the full adder is different depending on what specifications the modules need to work as intended. Below is module one which does a standard addition of A, B, and Cin.

Modules:

```
module module1(a, b, cin, sum, cout);
  input [3:0] a, b;
  input cin;
  output [3:0] sum;
  output cout;
  FullAdder Add1(a, b, cin, sum, cout);
endmodule
```

As these modules use the FullAdder module, it is best to explain how this works. This module adds inputs A, B, and Cin together. We were planning on using gates here, but since you have to end up with a one-bit number by anding two four-bit numbers and a one-bit number, we were unsure if we can get the correct results for Cout.

Full Adder:

```
module FullAdder(a, b, cin, sum, cout);
  input [3:0] a, b;
  input cin;
  output [3:0] sum;
  output cout;
  assign {cout, sum} = a + b + cin;
endmodule
```

For the Mux, there are two different versions. One that takes in four-bit values and one that takes in one-bit values. As a result, there are two muxes with similar code other than how the inputs and outputs are declared.

Mux:

```
module mux_4x1c (a, b, c, d, sel, out);
  input a, b, c, d;
  input [1:0] sel;
  output out;

  assign out = sel[1] ? (sel[0]?d:c):(sel[0]?b:a);

endmodule
```

Now finally for the “parent” module, this is where everything comes together to make this program into a black box. This module takes in five inputs and outputs two different values. How this works is after declaring the inputs and outputs, wires are then defined. This is so the program can get the results from each function module without losing or changing the input data that the user has put into the input. With these wires holding the values needed, this data is put into the mux where it will then produce one answer that is four-bits long and one answer that is one-bit long instead of eight different answers.

“Parent” module:

```
module Lab2(a,b,cin,s1,s0,F,cout);
  input [3:0] a,b;
  input s1,s0,cin;
  output [3:0] F;
```

```
  output cout;
  wire [3:0] M1, M2, M3, M4; //values of F from different modules
  wire C1,C2,C3,C4; ////values of Cout from different modules

  module1 Mod1(a, b, cin, M1, C1);
  module2 Mod2(a, b, cin, M2, C2);
  Module3 Mod3(a, b, cin, M3, C3);
  Module4 Mod4(a, b, cin, M4, C4);

  mux_4x1c Carry(C1,C2,C3,C4,{s1,s0},cout);
  mux_4X1s Sum(M1,M2,M3,M4,{s1,s0},F);

endmodule
```

Results/ Verification

```
module Lab2Test();
  reg [3:0] A, B;
  reg Cin, s0, s1;
  wire [3:0] Sum;
  wire Cout;
  parameter time_out = 100;

  initial $monitor (A, B, Cin, s1, s0, Sum, Cout);
  always
  begin
        #10 s1=0; s0=0; Cin=0; A=4'b1001; B=4'b0011;
        #10 s1=0; s0=1; Cin=0; A=4'b1001; B=4'b0011;
        #10 s1=1; s0=0; Cin=0; A=4'b1001; B=4'b0011;
        #10 s1=1; s0=1; Cin=0; A=4'b1001; B=4'b0011;
        #10 s1=0; s0=0; Cin=1; A=4'b1001; B=4'b0011;
        #10 s1=0; s0=1; Cin=1; A=4'b1001; B=4'b0011;
        #10 s1=1; s0=0; Cin=1; A=4'b1001; B=4'b0011;
        #10 s1=1; s0=1; Cin=1; A=4'b1001; B=4'b0011;

  end
  Lab2 test(A, B, Cin, s1, s0, Sum, Cout);

endmodule
```

The test bench is based on the example code given to us from ilearn and the part that has been modified is inside of the always block. Inside of that block, every ten times delays the values of Cin, S0, and S1 are changed to see if the program is working properly. The values of A and B are left unchanged in order to not be confusing when looking at the waveforms to check if everything is working as intended.

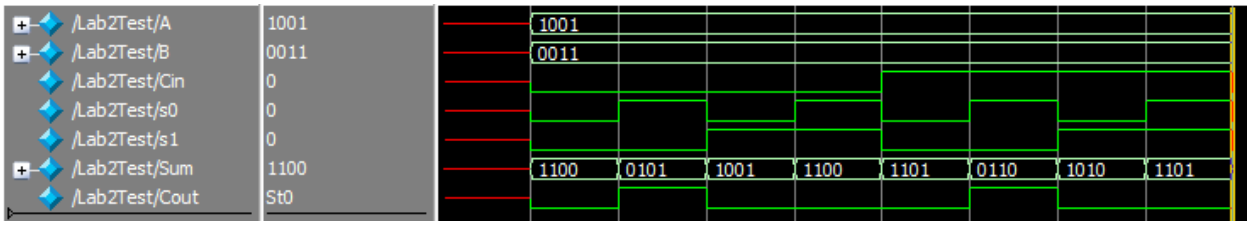

Fig 6: Results of Verilog Code

Here are our 4-bit numbers we used to test the four states to check that the arithmetic was being performed correctly. Our sum displays the value out the output variable while cout monitors the overflow, which only goes high in a single case and is reflected as such.

From the results of the test bench, we got the circuit to work correctly with the adders and the switch inputs to produce a black box that takes in two four-bit inputs and three one-bit inputs and outputs one four-bit output and one one-bit output.

Conclusion and Discussion

The experience for this lab was one of hardship. Learning a somewhat new language with all of its quirks was difficult in the beginning, but it got easier as it went on. The program does work as intended with taking in five inputs and outputting two different outputs. Some problems that we encountered included how to get the program to output the results for two states of the program. The issue for both happened to be syntax errors and were both promptly fixed.

Work Breakdown

For the work, we decided to split the work of the modules in the middle. Both Aaron and Ricky took two modules to work on. From there the work of the Mux was given to Ricky as Aaron was working on the “parent” module to bring all the different pieces of code together. After this, both Aaron and Ricky came together and wrote the testbench to test the code and to troubleshoot anything that doesn’t work as intended.

Prelabs

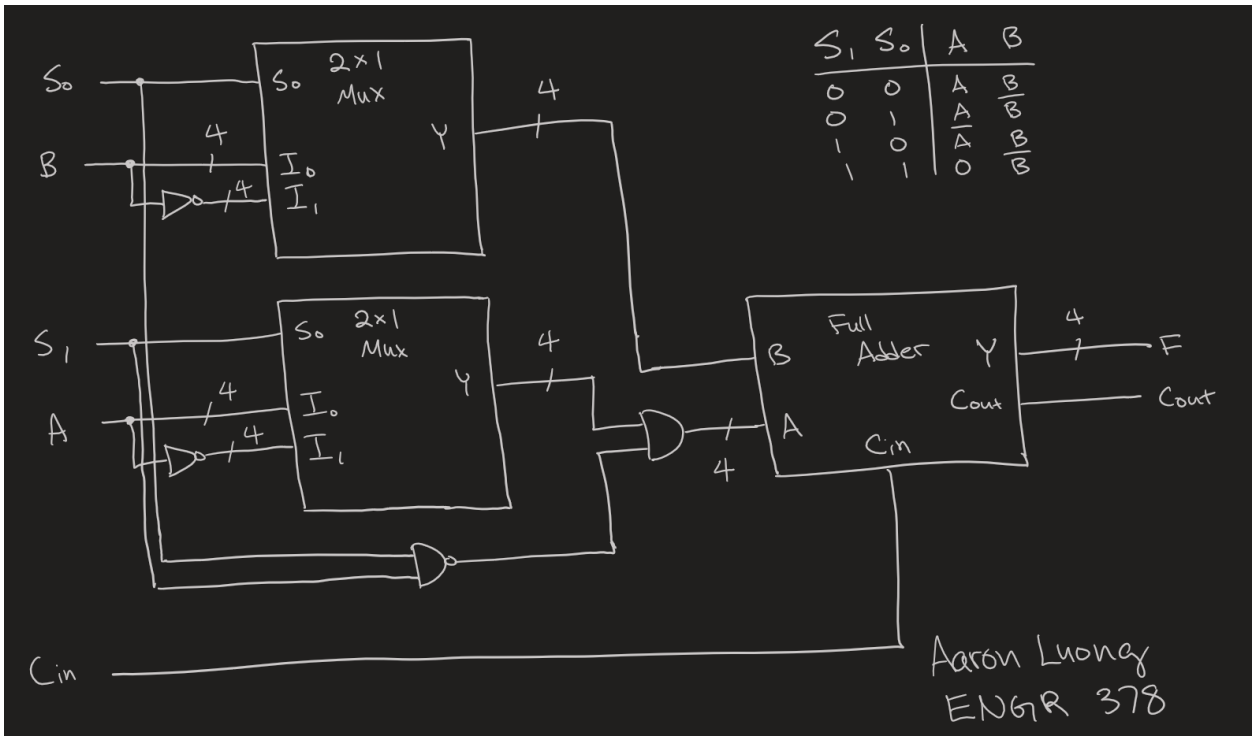

Fig 7: Prelab (Aaron Luong)

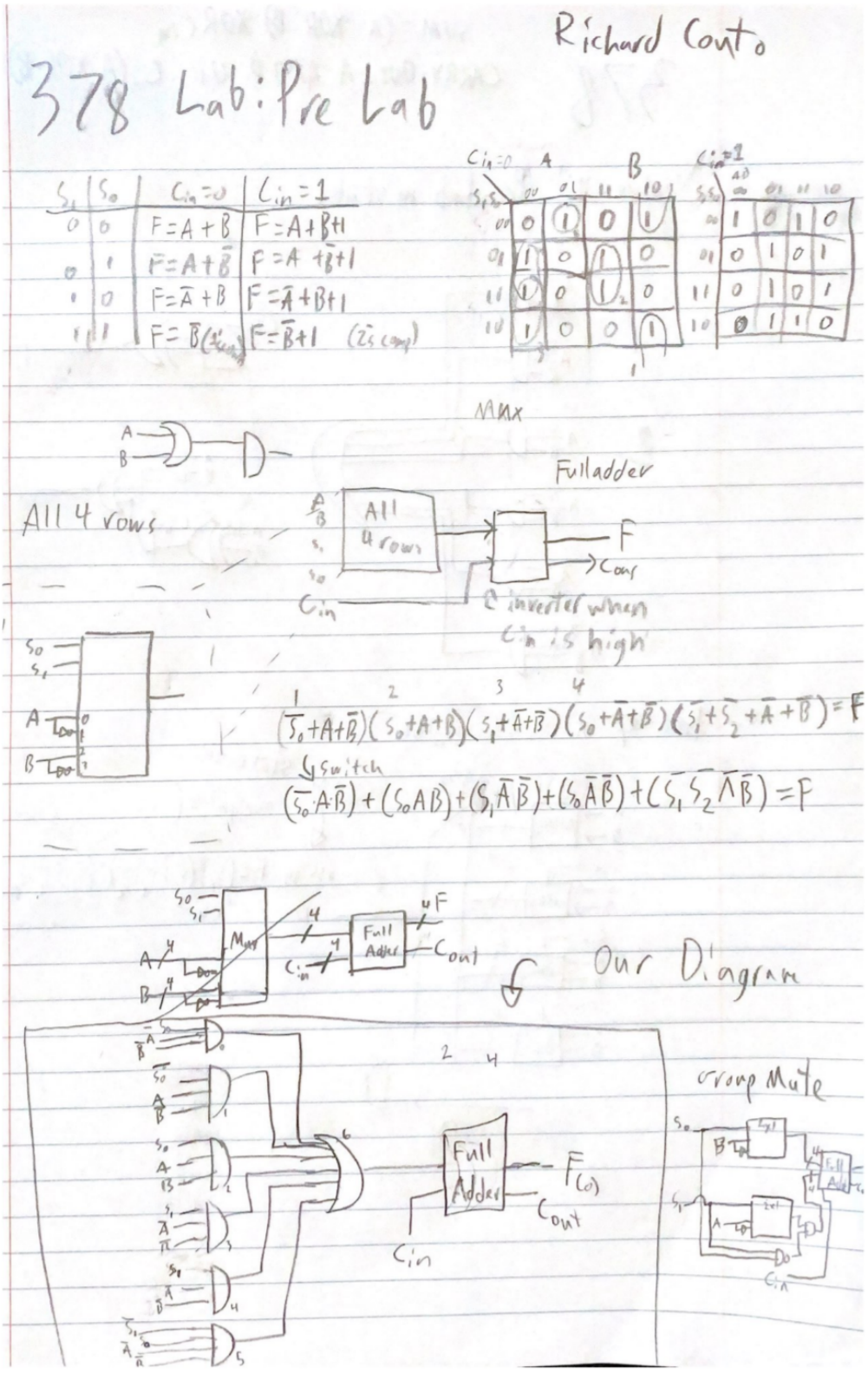

Fig 8: Prelab (Richard Couto)

HDL Source Code

```
module Lab2Test();
  reg [3:0] A, B;
  reg Cin, s0, s1;
  wire [3:0] Sum;
  wire Cout;
  parameter time_out = 100;

  initial $monitor (A, B, Cin, s1, s0, Sum, Cout);
  always
  begin
        #10 s1=0; s0=0; Cin=0; A=4'b1001; B=4'b0011;
        #10 s1=0; s0=1; Cin=0; A=4'b1001; B=4'b0011;
        #10 s1=1; s0=0; Cin=0; A=4'b1001; B=4'b0011;
        #10 s1=1; s0=1; Cin=0; A=4'b1001; B=4'b0011;
        #10 s1=0; s0=0; Cin=1; A=4'b1001; B=4'b0011;
        #10 s1=0; s0=1; Cin=1; A=4'b1001; B=4'b0011;
        #10 s1=1; s0=0; Cin=1; A=4'b1001; B=4'b0011;
        #10 s1=1; s0=1; Cin=1; A=4'b1001; B=4'b0011;

  end
  Lab2 test(A, B, Cin, s1, s0, Sum, Cout);

endmodule

module Lab2(a,b,cin,s1,s0,F,cout);
  input [3:0] a,b;
  input s1,s0,cin;
  output [3:0] F;
  output cout;
  wire [3:0] M1, M2, M3, M4; //values of F from different modules
  wire C1,C2,C3,C4; ////values of Cout from different modules

  module1 Mod1(a, b, cin, M1, C1);
  module2 Mod2(a, b, cin, M2, C2);
  Module3 Mod3(a, b, cin, M3, C3);
  Module4 Mod4(a, b, cin, M4, C4);

  mux_4x1c Carry(C1,C2,C3,C4,{s1,s0},cout);
```

```
  mux_4X1s Sum(M1,M2,M3,M4,{s1,s0},F);

endmodule

module mux_4x1c (a, b, c, d, sel, out);
  input a, b, c, d;
  input [1:0] sel;
  output out;

  assign out = sel[1] ? (sel[0]?d:c):(sel[0]?b:a);

endmodule

module mux_4x1c (a, b, c, d, sel, out);
  input a, b, c, d;
  input [1:0] sel;
  output out;

  assign out = sel[1] ? (sel[0]?d:c):(sel[0]?b:a);

endmodule

module module1(a, b, cin, sum, cout);
  input [3:0] a, b;
  input cin;
  output [3:0] sum;
  output cout;
  FullAdder Add1(a, b, cin, sum, cout);
endmodule

module module2(a, b, cin, sum, cout);
  input [3:0] a, b;
  input cin;
  output [3:0] sum;
  output cout;
  FullAdder Add2(a, ~b, cin, sum, cout);
endmodule

module Module3(a, b, cin, sum, cout);
  input [3:0] a, b;
```

```
  input cin;
  output [3:0] sum;
  output cout;
  FullAdder Add3(~a, b, cin, sum, cout);
endmodule

module Module4(a, b, cin, sum, cout);
  input [3:0] a, b;
  input cin;
  output [3:0] sum;
  output cout;
  FullAdder Add4(0, ~b, cin, sum, cout);
endmodule

module FullAdder(a, b, cin, sum, cout);
  input [3:0] a, b;
  input cin;
  output [3:0] sum;
  output cout;
  assign {cout, sum} = a + b + cin;
endmodule
```