Bruno E. Gracia Villalobos

EE 4513

Assignment # 1

September 2, 2019

A. Short Answer Questions

1. Briefly describe the three design levels that Verilog supports. Provide figures or code to help describe an example of each type of level.

- **Behavioral Level**

  This level allows you the use of Boolean equations to generate hardware. It becomes easier to implement large designs using equations rather than drawing out the circuit. At this level, the synthesizer is left with discretion with what hardware to use to implement the “behavior” or logic equations specified by the engineer. However, this can lead to unnecessary hardware, and is synthesizer specific. Not all behavioral level code is synthesizable, and its primary purpose is to simulate a design; data types can be assigned 1,0,x,z for simulation purposes. Procedural blocks may be used but not necessary.

  Example of behavioral statements:

```
wire y, x1, x2;

assign y = sel ? x1 : x2;

assign z = a | b;
```

- **RTL Level**

  RTL is useful to model a processor-based design with datapath and controller units. Finite state machines are used to control the transfer of data between registers in the datapath unit. A big differentiator over gate level is the use of sequential design with procedural blocks. Most importantly, this level places focus on synthesizable logic.

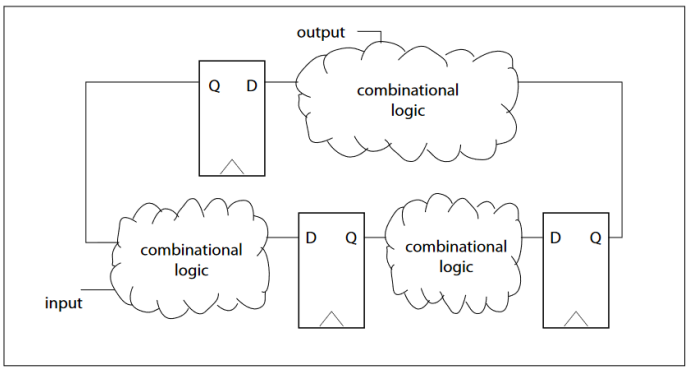


Figure 8-14 A register-transfer system.

(Taken from page 496 of *Modern VLSI Design by Wayne Wolf, 4th edition).*

```
reg q;
wire d;

always@(posedge CLK) begin
```

```
        q <= d;
end
```

- **Gate Level**

  This level has the least abstraction compared to RTL and Behavioral. The circuit is built using primitive gates such as AND, OR, XOR, and Inverters. A key feature of this abstraction level is the use of combinational design. Usually no procedural blocks are used in this level.

```
wire y1, a, b;

and a1(y1, a b);
```

2. Describe the difference in the use of Little Endian and Big Endian as it pertains to its use in Verilog Design.

When defining a bus in Verilog, memory bank, or register array, the endianness of the indexed values must be defined.

```
wire [7:0] bus; // little endian 8-bit bus

wire [0:7] bus; // big endian 8-bit bus
```

In both occasions, the MSB is the index 7, and the LSB is index 0.

3. Describe the three commonly used data types in Verilog:wire, reg, integer. (eg. How are they used, what are their default sizes, etc.

- Wire
  These are really wires in the circuit. They transfer information between structures. The default size of a wire is 1 bit, but they can be defined to be an arbitrary number of wires combined to form a bus. Wires can be used to output data of a module. In the testbench, wires are used to read incoming data from the module. Wires cannot be assigned in a procedural block, and they are only assigned to with concurrent statements.

- Reg
  Registers store information with respect to time and their default size is 1 bit. Compared to a wire, which must be driven continuously to “store” or contain information, a register can hold a bit until changed. Registers are the primary source of logic transfer in a sequential design and therefore can only be altered in a procedural block.

- Integer
  This data type is a signed integer with a default width of 32 bits. Useful for storing constants to be used in arithmetic. For example, when designing a digital clock, an integer can be used to mark the alarm clock time for the logic to check until it reaches that point in time.

4. What are always blocks and initial blocks? (eg. Howare they used in Verilog? How are they executed?)

Always blocks and initial blocks are procedural blocks useful for defining the behavior of the design, and they can also be used for a design specified at RTL.

- Always block

An always block is essentially a perpetual “checker” that evaluates its sensitivity list first before running the code within. Although it is primarily used for sequential design, an always block with a wildcard sensitivity list (*) can create combinational logic such as a multiplexer with case statements—conditional statements can only be used within a procedural block. Above all, the always block lives up to its name and always checks its sensitivity list.

- Initial block
  This block only runs once and is executed during the instantiation of the module. For testbenches, the initial block is the place for sequential statements to test the UUT. Initial blocks are useful to specify initial register conditions in a module. For example, in a FSM design, the initial block can be used to initialize the state register to the reset state.

5. What is the difference between Blocking and Non-Blocking Assignments? In which cases should each one be used? Provide code example or schematic for your explanation.

Both types of assignments matter within a procedural block. Blocking assignments are used to define a sequential design: they are executed in order. Non-blocking assignments are used to give combinational-like characteristics and mimic concurrent assignments: they are ran concurrently after the triggering of the procedural block.

Example of Blocking assignments:

```
reg d, enable, clk;

initial begin
        clk = 0;
        enable = 1;
        d = 1;
end
```

Example of Non-Blocking assignments:

```
reg state, count, next_state;

always@(posedge CLK) begin
        state <= next_state;
        count <= count + 1;
end
```

6. What is the difference between asynchronous and synchronous in Verilog modeling? Provide code example or schematic

Asynchronous designs do not need to respond to the clock—they are evaluated whenever the control signal is sent. On the other hand, synchronous designs do respond to the clock and are only evaluated as defined by setting—active high or active low.

Example asynchronous reset on a 32-bit counter:

```
reg [31:0] counter;
```

```
wire reset;
always@(*) begin         //does not depend on clock
        if(reset) counter = 32'h0000;
        counter = counter + 32'h0001;
end
```

Example synchronous load on a 32-bit counter:

```
reg [31:0] counter;
wire load, [31:0] sync_load;
always@(posedge CLK) begin       //active high design
        if(load) counter = sync_load;
        counter = counter + 32'h0001;
end
```

7. How would you represent 226 in Verilog as an integer number in decimal, hex, and binary, as a real number in decimal or scientific notation, and as a signed/unsigned number?

```
reg [7:0] number; //8 bits for 0-255
reg [15:0] s_number; //to test sign extension
initial begin
        //unsigned numbers
        number = 8'hE2; // in hex
        number = 8'd226; // in decimal
        number = 8'b1110_0010; //in binary

        //signed
        s_number = 9'sb0_1110_0010; //will extend 7 upper bits with 0's

        //real number
        s_number = 226.0;

        //scientific notation
        s_number = 2.26E2;
end
```

B. Model the following simple logic modules using Verilog HDL

1. 16 to 1 MUX

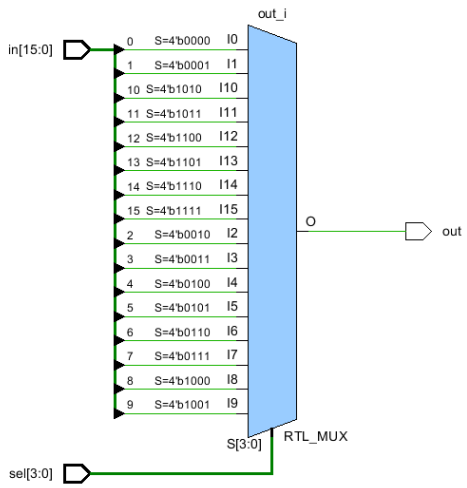


Figure 1: 16 to 1 MUX Schematic

Procedure

A module was created to generate the design for the 16 to 1 MUX as shown in Figure 1. First, the port list was defined to include a 16-bit width wire bus for the input lines to the mux, a 4-bit wire bus input to select the input lines, and a 1-bit output register for the output of the MUX. All ports are little-endian for readability.

```
3  module MUX(
4      input [15:0] in, //16 bit input
5      input [3:0] sel, //log2(16) = 4 bits to select from 16 inputs
6      output reg out //output the 1 bit answer
```

With the port list in place, the next step defined the procedural block to conditionally output the selected input line. An always block with a wild card sensitivity list was used to generate the combinational logic needed to model the MUX and shown in Figure 2. Inside the always block is a case statement that depends on the select lines, and each case selectively outputs the chosen input line through the 1 bit output register. It is important to mention the default case is set to output a Hi-Z to catch both 1'bx and 1'bz cases.

```
9  //procedural block generates combinational logic because of wildcard sensitivity list
10 always@(*) begin
11     case(sel) //case statement to select the input as the output
12         4'h0: out = in[0];
```

Figure 2: Procedural Block for MUX

Observations and Results

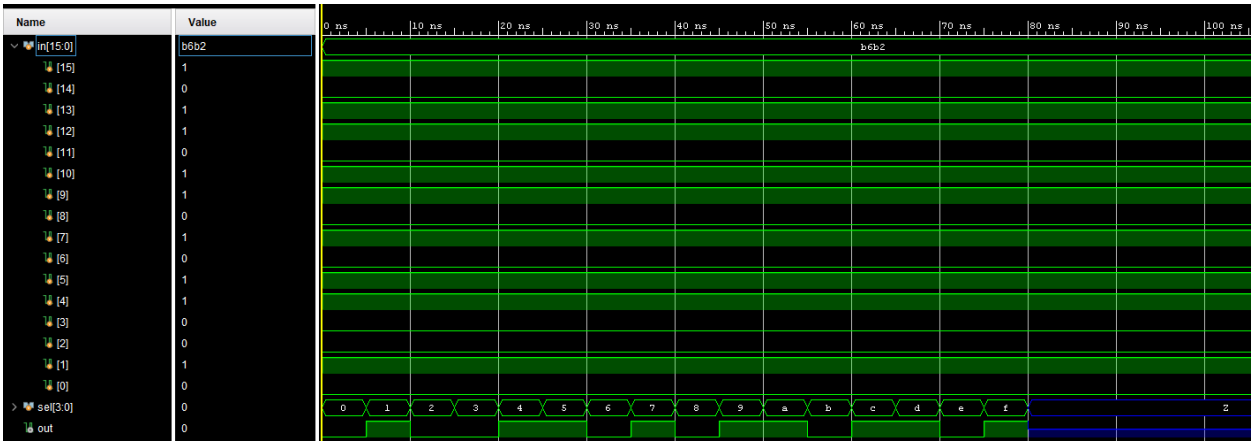
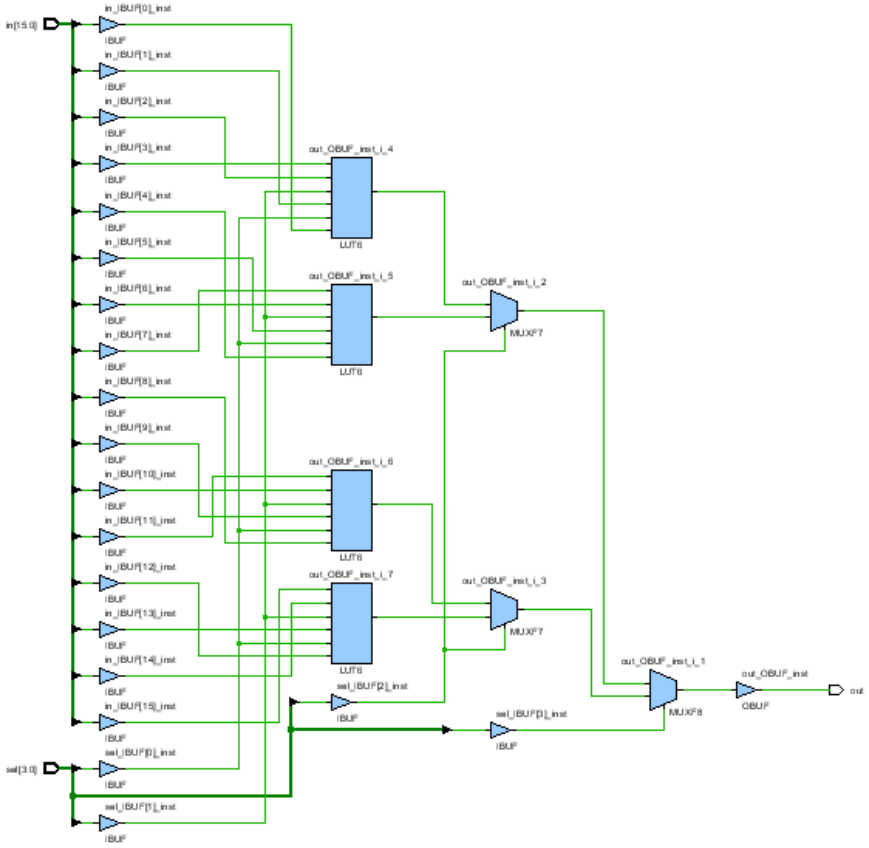


Figure 3: Simulation Results for MUX

An arbitrary 16-bit word is chosen as the inputs to the MUX. Figure 3 displays all the inputs correctly channeled to the output given their respective select lines with 5 ns delays for visibility. The don't care 4'h? input was also tested and shown at the 80ns mark as a hi-Z; this is useful because the MUX should not output any data unless the select lines are explicitly set to the corresponding input.

Conclusion

For a different implementation, the MUX could be defined with the conditional operator '?' as well. However, this approach could be more difficult programmatically and the synthesized result would most likely amount to the same RTL_MUX. Using a 16-bit input bus rather than individually named inputs also give the designer greater flexibility when implementing the module. Above all, the MUX functions as intended and its robust implementation—checking for don't cares—allows the VLSI Engineer to debug its use in a larger design.



Examining the TCL Report from the XST during synthesis helped to understand the implementation better. The highlighted line shown in the report in the appendix helped me to learn the default case of the MUX design to set the output as Hi-Z is not used when synthesized. That is, the behavior of the real MUX in hardware cannot be set to Hi-Z when the input is X or Z. Essentially, understanding the TCL messages from the XST can provide valuable insight into the design.

The Figure above displays the synthesized result. According to the CLB (Configurable Logic Block) specification of the FPGA part used (Artix-7), a 16:1 MUX is implemented with 4, LUT6's as described in the UG474 from Xilinx shown below.

16:1 Multiplexer

Each slice has an F8MUX. F8MUX combines the outputs of F7AMUX and F7BMUX to form a combinatorial function up to 27 inputs (or a 16:1 MUX). Only one 16:1 MUX can be implemented in a slice, as shown in Figure 2-23.

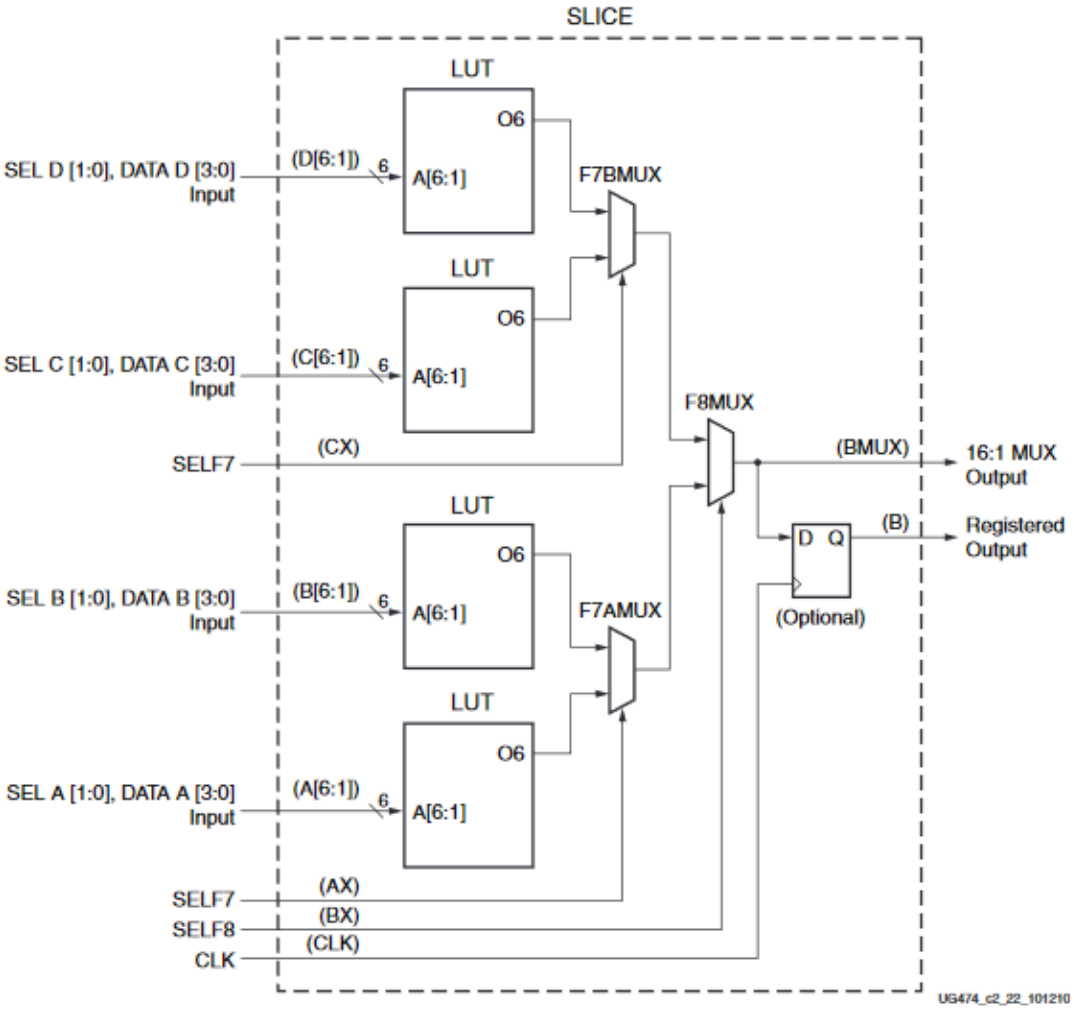


Figure 2-23: **16:1 Multiplexer in a Slice**

2. 3 to 8 Decoder

Procedure

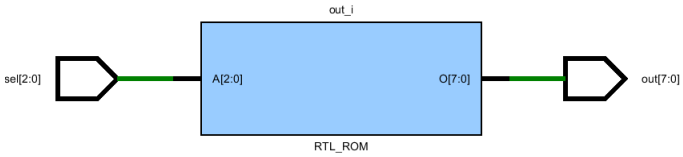


Figure 4: Decoder Schematic

The decoder was designed as a standalone module. The schematic is shown in Figure 4 as an RTL_ROM component due to the nature of its behavior. The port-list was the first thing to be defined, and both a 3-bit input select line wire bus and an 8-bit output register are used. Figure 5 shows how the 3 bit input represents 8 possible combinations.

```
3  module DECODER(
4      input [2:0] sel,    //3 bit select line to select from 8 outputs
5      output reg [7:0] out    //8 bit output
6  );
```

Figure 5: Decoder Port-list

To implement the logic of the Decoder, a procedural always block with a wildcard sensitivity list was used; the wildcard, similarly used for the 16-to-1 MUX, is chosen to create combinational logic. A case statement is triggered on the 3-bit select input, and the 8-bit output is defined using a one-hot assignment respective of the input lines. Here, the ‘d’ configurator is conveniently used to specify the assignment in decimal. Figure 6 showcases the procedural block used to define the decoder.

```
8   //procedural block to create one-hot decoder
9   always@(*) begin //wildcard to create combinational logic
10      case(sel)
11          3'd0: out = 8'd1;
12          3'd1: out = 8'd2;
```

Figure 6: Decoder procedural block logic

Observation and Results

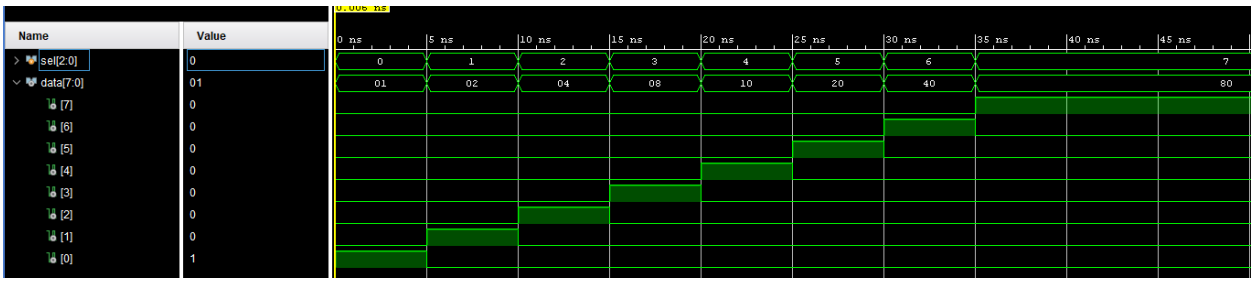


Figure 7: 3-to-8 Decoder Simulation

The testbench instantiates the decoder as the UUT, and both a 3-bit register and 8-bit wire bus are used to interface the inputs and outputs of the module respectively. The procedural initial block tests each of the 8 different input combinations to observe the one-hot outputs of the module within 5ns delays for visibility. Figure 7 displays the simulation results and verifies the design of the module successfully.

Conclusion

The 3-to-8 decoder implementation was straightforward and similar to the MUX; From online research, I found out a decoder can also be named as a de-MUX, and this is apparent in both modules’ use of case statements to selectively control the output. Further, after learning of the

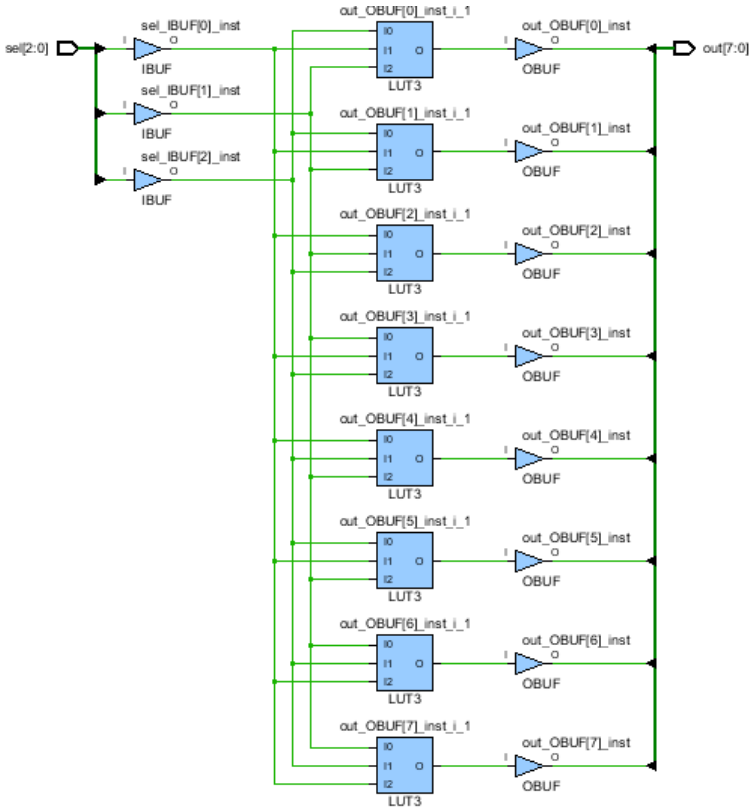
unused default case from the previous XST report of the 16-to-1 MUX, I omitted it from the code in this design.

The RTL Schematic in Figure 4 shows an RTL_ROM block. I understand the decoder serves as a similar function to a ROM, but after further research on Xilinx's XST manual, there are synthesis reports of generating an actual decoder instead. The XST Report in the appendix for this assignment report shows the synthesizer generated an 8-bit input MUX which is also strange to me as shown in Figure 8; the most likely explanation is realizing the use of the Kintex-7 part in this project led the synthesizer to use what is available in that part. However, re-running the synthesis for this module using an Artix-7 part also produced the same results.

```
52 Detailed RTL Component Info :
53 +---Muxes :
54       8 Input      8 Bit        Muxes := 1
55 ------------------------------------------------------------------------------
```

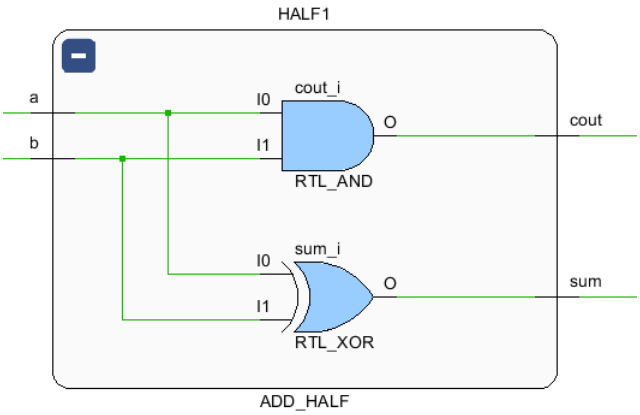
Figure 8: XST Report for Decoder

The realized circuit after synthesis is shown in the figure below.



3. 1-bit Full Adder

Procedure



Designing a 1 bit Full adder consists of a bottom up, structural approach. First, the ADD_HALF module is designed, simulated, and synthesized to verify corrrect operation; its schematic is shown in Figure 9. Next, the FULL_ADDER module is designed. Like the ADD_HALF module, this module is also simulated and synthesized to verify correct operation; its schematic is shown in Figure 10.

Figure 9: ADD_HALF schematic

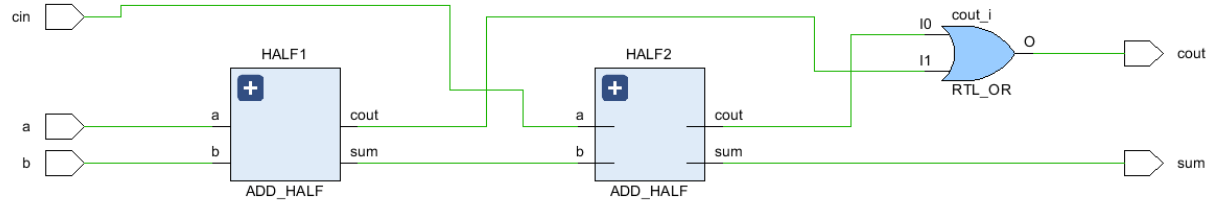


Figure 10: FULL_ADDER Schematic

```
3  module FULL_ADDER(
4      input a,b,cin,
5      output cout,sum
6
7  );
```

Figure 10: FULL_ADDER portlist

To design the half adder, a structural approach was used: both AND and XOR gate primitives are instantiated with the appropriate wires in their port lists to create the half-adder. After simulation proved the desired operation, the module was synthesized to continue with the design process.

Once the ADD_HALF module was done, the next step was to design the FULL_ADDER module. This module is also built with a structural approach using two instantiated ADD_HALF modules and an OR gate for the carry out. Three input ports for the operands and carry in, as well as two output ports for the sum and carry out are all 1-bit in width and are shown in Figure 11.

Observation and Results

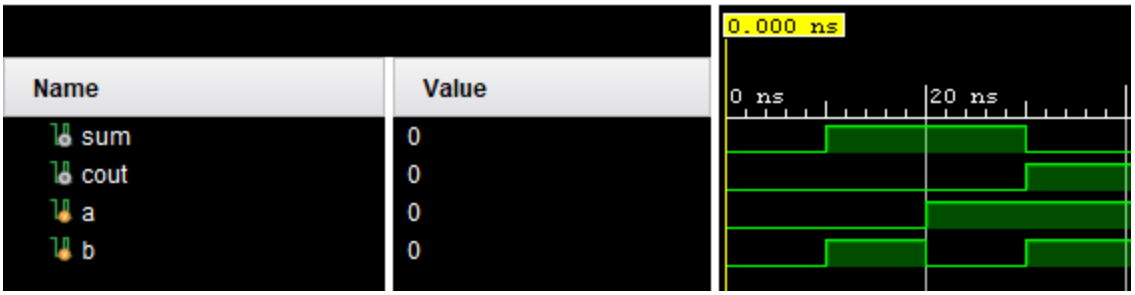


Figure 11: ADD_HALF Simulation

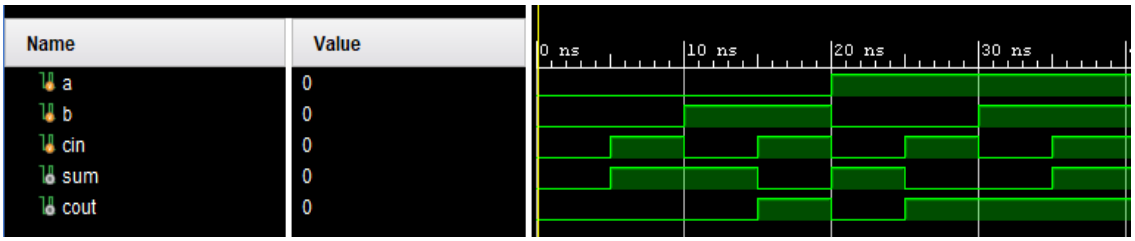


Figure 12: FULL_ADDER Simulation

Figure 12 displays the simulation results from the ADD_HALF module. Given there are 2 input bits, a total of 4 combinations are verified for desired operation. Figure 13 showcases the simulation results from the FULL_ADDER module. Here, three bits are used as inputs, and therefore 8 different combinations are tested with 5 ns delays and verified for desired operation. Both simulations proved successful.

Conclusion

A bottom-up approach is used to create the 1-bit full adder. Although a behavioral method would by far be the easiest, this approach allowed me to learn a little of how the XST synthesizer works. Further, there is more control the amount of gates that are used at this level, and for simulation purposes, it was helpful to analyze the input and output signals of each gate.

To prove the efficiency of my design and to fulfill the requirements of this assignment, the FULL_ADDER module was synthesized. Since the FULL_ADDER module is essentially a function of 5 variables, the synthesizer used 2 LUT3's to implement this function, as well as 3 IBUFs and 2 OBUF's for the three inputs and two outputs respectively. Figure 13 displays the synthesized result.

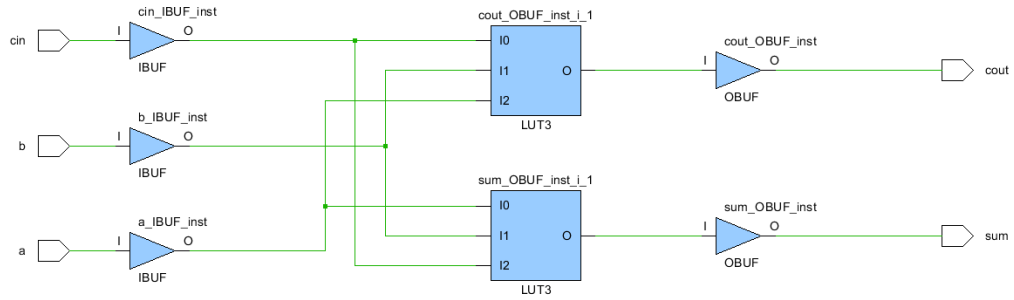


Figure 13: Synthesized FULL_ADDER module

4. D Flip Flop

Procedure

```
2  module DFF(
3      input d, reset, CLK,
4      output reg q
5  );
```

Figure 14: DFF port list

A new module is created to define the behavior of a D Flip Flop with Asynchronous Reset. This time, a structural model is skipped in favor of an easier approach at RTL. This Flip Flop has 1-bit clock, reset, and d inputs and a 1-bit output q as shown in Figure 14. For this matter, the D Flip Flop is designed using an always block with a sensitivity list of both clk and reset positive edges.

Whenever a positive CLK edge is present, the output q follows d; whenever the reset signal rises to logic high, the output q is set to 0. The schematic of the RTL logic is shown in Figure 15 as an RTL_REG_ASYNC, which matches the intended design.

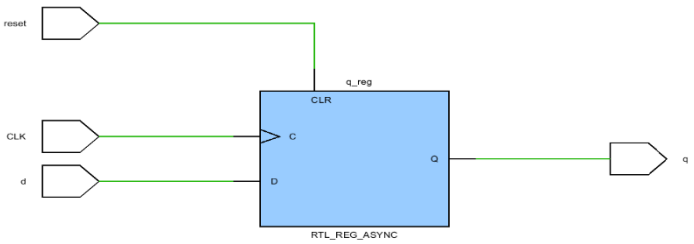


Figure 15: DFF RTL Schematic

Observation and Results

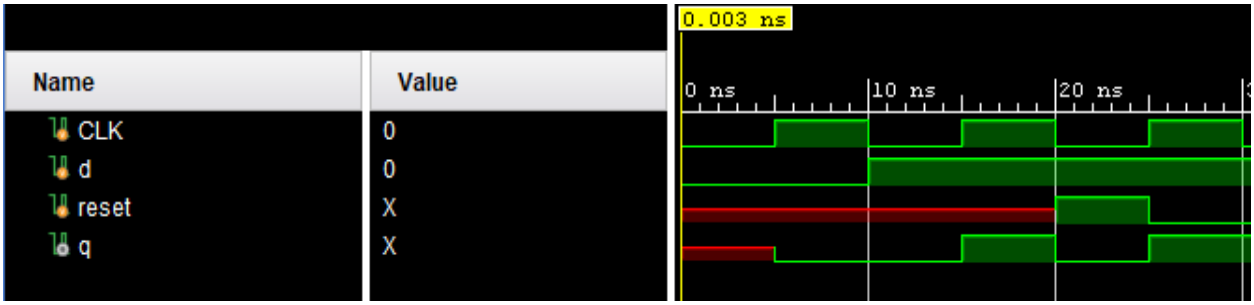


Figure 16: DFF Behavioral Simulation

After developing the module and verifying the RTL schematic for intended design, a testbench tested the module as the UUT (Unit under test) under a behavioral simulation with a 10 MHz clock; Figure 16 displays the simulation. At time t=5ns, q follows d and drops. At time t=10ns, d is set high but q does not follow until the posedge at t=15ns. At time t=20ns, the asynchronous reset sets q low before the active edge of the clock. Finally, the reset signal is set low and q follows d at t=20ns.

Conclusion

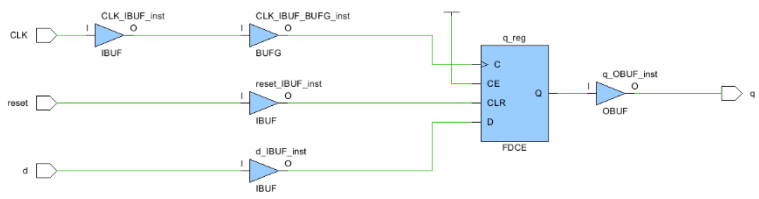


Figure 17: DFF Synthesized Schematic

The D Flip Flop is generated using an RTL design approach. Once the Flop Flop was simulated for desired operation, the module was synthesized as the last step. Figure 17 displays the synthesized schematic of the Flip Flop. The design is successfully synthesized as the Xilinx primitive FDCE with input and output buffers.

C. 4-bit Ripple Carry Adder (RCA) Design

Procedure

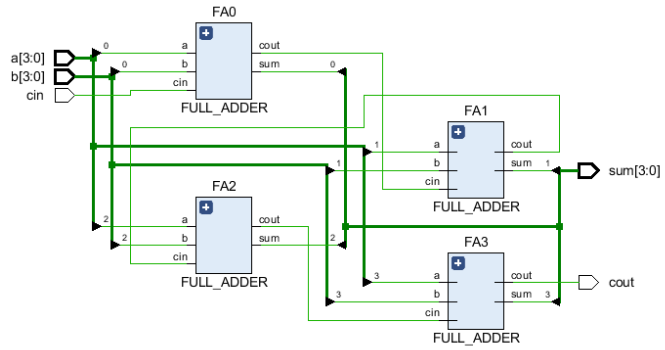


Figure 19: RCA RTL Schematic

```
2  module RCA(
3  //4 bit operands
4  input [3:0] a,
5  input [3:0] b,
6  input cin, //1 bit carry in
7  output cout, //1 bit carry out
8  output [3:0] sum //4 bit sum
```

Figure 18: RCA porlist

The 4-bit RCA is designed using a top-down approach because the nested modules necessary to build this device have been developed and tested already. To begin, the portlist is created to take in two 4-bit operands a and b and a 1-bit cin. For the outputs, the module drives a 4-bit sum bus and a 1-bit cout wire. Next, four FULL_ADDERs are instantiated within the module after creating the portlist as shown in Figure 18. Intermediary wires were also created for the ripple carry between FULL_ADDERs. The realized RTL schematic is shown in Figure 19. Once the observed structural logic of the RCA was verified with the RTL schematic, the test bench was crafted next to test the behavioral operation of the module.

Observation and Results

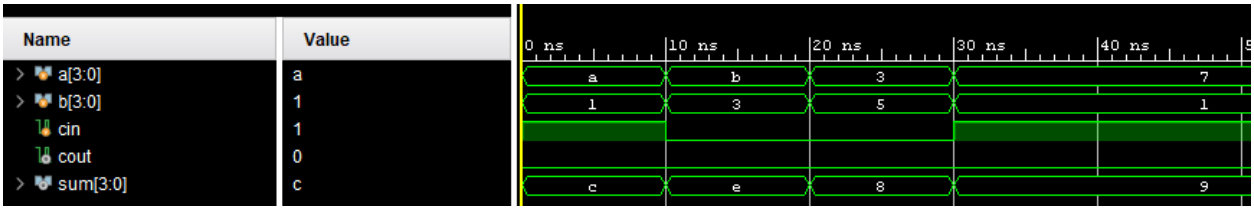


Figure 20: RCA Behavioral Simulation

A new module named RCA_SIM.v is created as the testbench of the module. First, the portlist is copied from the RCA module definition and instantiated as the UUT. Then, the appropriate registers and wires needed to communicate with the UUT are declared in the testbench. Lastly, an initial procedural block defines the operands and carry ins to test the operation of the RCA. Figure 20 displays the simulation in Vivado. Four different test cases are used to verify the operation and the following table lists the inputs (blue) and outputs (green) of the UUT within 5ns delays.

A	B	CIN	SUM	COUT
A	1	1	C	0
B	3	0	E	0
3	5	0	8	0
7	1	1	9	0

Conclusion

The design uses a structural approach due to the low-level specification of half adders and full adders—these are defined using gates. Furthermore, the bottom up design approach consisted of developing the half adder module, instantiating the two half adders and an OR gate to make a full adder, and finally instantiating four full adders to build the ripple carry adder. Above all, these approaches taken to design and test the RCA gave me the oversight of implementing only the necessary hardware for this design compared to a behavioral approach where the XST synthesizer might allocate different resources. Nevertheless, this design does not take advantage of the DSP48 slice inside the Artix-7 FPGA to implement an adder because of the following clause in the Xilinx XST manual:

> Macro implementation on DSP48 blocks is controlled by DSP Utilization Ratio (DSP_UTILIZATION_RATIO) with a default value of **auto**. In **auto** mode, if an adder/subtractor is a part of a more complex macro such as a filter, XST automatically places it on the DSP block. Otherwise, XST implements adders/subtractors using LUTs. Set the value of Use DSP48 (USE_DSP48) to **yes** in order to force XST to push these macros into a DSP48. When

In other words, since the adder is not part of a DSP oriented design such as a filter, it is implemented using LUTs. However, the following attribute can be declared to use this dedicated resource by preceding the module or signal: `(*use_dsp48="{auto|yes|no}"*` The allocated resources for the RCA are demonstrated in the synthesized schematic in Figure 21.

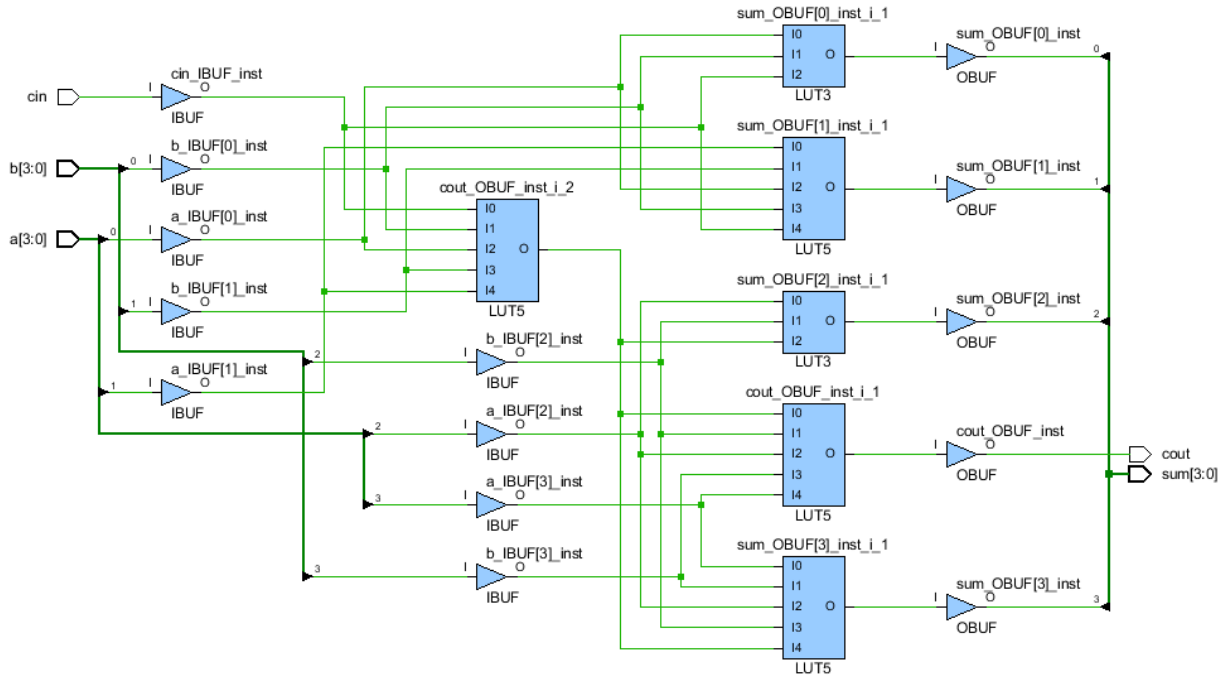


Figure 21: RCA Synthesized Schematic

APPENDICES (Code and XST Reports)

B.1 (16 to 1 MUX)

XST (Xilinx Synthesizer Technology) TCL Report

```
synth_design -rtl -name rtl_1

Command: synth_design -rtl -name rtl_1

Starting synth_design

Using part: xc7k70tfbv676-1

Top: MUX

---------------------------------------------------------------------------------

Starting RTL Elaboration : Time (s): cpu = 00:00:02 ; elapsed = 00:00:03 . Memory (MB): peak = 1099.469 ;
gain = 104.746

---------------------------------------------------------------------------------

INFO: [Synth 8-6157] synthesizing module 'MUX'
[C:/Users/admin/Documents/UTSA/EE4513/HW/HW1/HW1.srcs/sources_1/new/MUX.v:3]

INFO: [Synth 8-226] default block is never used
[C:/Users/admin/Documents/UTSA/EE4513/HW/HW1/HW1.srcs/sources_1/new/MUX.v:11]

INFO: [Synth 8-6155] done synthesizing module 'MUX' (1#1)
[C:/Users/admin/Documents/UTSA/EE4513/HW/HW1/HW1.srcs/sources_1/new/MUX.v:3]
```

Code

```
`timescale 1ns / 1ps

module MUX(

    input [15:0] in, //16 bit input

    input [3:0] sel, //log2(16) = 4 bits to select from 16 inputs

    output reg out //output the 1 bit answer

);

//procedural block generates combinational logic because of wildcard sensitivity list

always@(*) begin

    case(sel) //case statement to select the input as the output

        4'h0: out = in[0];

        4'h1: out = in[1];

        4'h2: out = in[2];

        4'h3: out = in[3];

        4'h4: out = in[4];

        4'h5: out = in[5];

        4'h6: out = in[6];

        4'h7: out = in[7];

        4'h8: out = in[8];

        4'h9: out = in[9];
```

```
        4'hA: out = in[10];
        4'hB: out = in[11];
        4'hC: out = in[12];
        4'hD: out = in[13];
        4'hE: out = in[14];
        4'hF: out = in[15];
    default: out = 1'bz; //disconnect output if input is unknown
    endcase
end

endmodule
```

```
`timescale 1ns / 1ps

module MUX_SIM();
reg [15:0] in; //to send the 16 inputs to the MUX
reg [3:0] sel; //register needed to change inside procedural initial block
wire out; //to receive the answer from the UUT

//instantiate 16 to 1 MUX as the UUT for this testbench
MUX UUT(
.in(in),
.sel(sel),
.out(out)

);

initial begin
    in = 16'b1011_0110_1011_0010; //initialize the inputs to the MUX

    //test every input, 5 ns delay to see the changes
    sel = 4'h0;
    #5 sel = 4'h1;
    #5 sel = 4'h2;
    #5 sel = 4'h3;
    #5 sel = 4'h4;
    #5 sel = 4'h5;
    #5 sel = 4'h6;
```

```
    #5 sel = 4'h7;
    #5 sel = 4'h8;
    #5 sel = 4'h9;

    #5 sel = 4'hA;
    #5 sel = 4'hB;
    #5 sel = 4'hC;
    #5 sel = 4'hD;
    #5 sel = 4'hE;
    #5 sel = 4'hF;

    #5 sel = 4'h?;
end
endmodule
```

B.2 (3-to-8 Decoder Code)

Code

```
`timescale 1ns / 1ps

module DECODER(
    input [2:0] sel,    //3 bit select line to select from 8 outputs
    output reg [7:0] out    //8 bit output
);

//procedural block to create one-hot decoder
always@(*) begin //wildcard to create combinational logic
    case(sel)
        3'd0: out = 8'd1;
        3'd1: out = 8'd2;
        3'd2: out = 8'd4;
        3'd3: out = 8'd8;
        3'd4: out = 8'd16;
        3'd5: out = 8'd32;
        3'd6: out = 8'd64;
        3'd7: out = 8'd128;
    endcase
end
```

```
endmodule
```

```
`timescale 1ns / 1ps
module DECODER_SIM();
reg [2:0] sel; //for selecting the output
wire [7:0] data; //for receiving the output of the decoder

//instantiate the decoder as the UUT
//use named association for better portlist awareness
DECODER UUT(
.sel(sel),
.out(data)
);

//procedural block to begin the testbench
initial begin
    sel = 3'd0;
    #5 sel = 3'd1;
    #5 sel = 3'd2;
    #5 sel = 3'd3;
    #5 sel = 3'd4;
    #5 sel = 3'd5;
    #5 sel = 3'd6;
    #5 sel = 3'd7;


end

endmodule
```

B.3 (1-bit Full Adder Code)

Code

```
`timescale 1ns / 1ps

module FULL_ADDER(
    input a,b,cin,
    output cout,sum
```

```
);

wire half1_cout, half1_sum, half2_cout;

ADD_HALF HALF1(
.cout(half1_cout),
.sum(half1_sum),
.a(a),
.b(b)
);

ADD_HALF HALF2(
.cout(half2_cout),
.sum(sum),
.a(cin),
.b(half1_sum)
);

//for the carry out of both half adders
or(cout, half2_cout, half1_cout);

endmodule
```

```
`timescale 1ns / 1ps
module FULL_ADDER_SIM();
reg a,b,cin;
wire sum,cout;

FULL_ADDER UUT(
.a(a),
.b(b),
.cin(cin),
.sum(sum),
.cout(cout)
);
```

```
initial begin
{a,b,cin} = 3'd0;
#5 {a,b,cin} = 3'd1;
#5 {a,b,cin} = 3'd2;
#5 {a,b,cin} = 3'd3;
#5 {a,b,cin} = 3'd4;
#5 {a,b,cin} = 3'd5;
#5 {a,b,cin} = 3'd6;
#5 {a,b,cin} = 3'd7;


end

endmodule
```

```
`timescale 1ns / 1ps

module ADD_HALF(
    output cout,
    output sum,
    input a,
    input b
);
    xor(sum, a, b);
    and(cout, a, b);
endmodule
```

```
`timescale 1ns / 1ps

module ADD_HALF_SIM();

wire sum, cout;
reg a, b;

ADD_HALF TEST (
.sum(sum),
.cout(cout),
.a(a),
.b(b)
```

```
);



initial begin
    {a, b} = 2'b00;

    #10;
    {a, b} = 2'b01;

    #10;
    {a, b} = 2'b10;

    #10;
    {a, b} = 2'b11;



end

endmodule
```

B.4 (DFF Code)

Code

```
`timescale 1ns / 1ps
module DFF(
    input d, reset, CLK,
    output reg q
);



//posedge reset signifies asynchronous reset
always@(posedge CLK, posedge reset) begin
    if(reset) q <= 1'b0; //if reset set q to 0
    else q <= d;    //q follows d

end

endmodule
```

```
`timescale 1ns / 1ps
```

```
module DFF_SIM();

//The following three lines setup a 1/10ns = 10MHz CLK
reg CLK;
initial CLK = 1'b0; //this block sets up the initial value of the clk
always #5 CLK = ~CLK; //makes the clock tick every 5 ns


reg d, reset; //for sending to the DFF
wire q; //for receiving the output of DFF

//Instantiation of the DFF
DFF UUT(
.d(d),
.CLK(CLK),
.reset(reset),
.q(q)
);

initial begin

//test the q output
d = 1'b0;
#10 d = 1'b1;

//test asynchronous reset
#10 reset = 1'b1;
#5 reset = 1'b0;

#10 d = 1'b1;

end
endmodule
```

C. (RCA Code)

Code

```
`timescale 1ns / 1ps
module RCA(
```

```
//4 bit operands
input [3:0] a,
input [3:0] b,
input cin, //1 bit carry in
output cout, //1 bit carry out
output [3:0] sum //4 bit sum

);
wire [2:0] fa_cout; //intermediary wires to ripple the carry

//instantiate four copies of full-adders
FULL_ADDER FA0(
.a(a[0]),
.b(b[0]),
.cin(cin),
.sum(sum[0]),
.cout(fa_cout[0])
);

FULL_ADDER FA1(
.a(a[1]),
.b(b[1]),
.cin(fa_cout[0]),
.sum(sum[1]),
.cout(fa_cout[1])
);

FULL_ADDER FA2(
.a(a[2]),
.b(b[2]),
.cin(fa_cout[1]),
.sum(sum[2]),
.cout(fa_cout[2])
);

FULL_ADDER FA3(
.a(a[3]),
```

```
.b(b[3]),
.cin(fa_cout[2]),
.sum(sum[3]),
.cout(cout)
);
endmodule
```

```
`timescale 1ns / 1ps
module RCA_SIM();

//for sending to the Ripple carry adder
reg [3:0] a;
reg [3:0] b;
reg cin;

//for receiving from the RCA
wire cout;
wire [3:0] sum;


RCA UUT(
.a(a),
.b(b),
.cin(cin),
.cout(cout),
.sum(sum)
);

initial begin
//add 10 + 1 + 1 = 12
a = 4'hA;
b = 4'h1;
cin = 1'b1;

#10;

a = 4'hB;
b = 4'h3;
```

```
cin = 1'b0;

#10;

a = 4'h3;
b = 4'h5;
cin = 1'b0;

#10;

a = 4'h7;
b = 4'h1;
cin = 1'b1;




end


endmodule
```