Bruno E. Gracia Villalobos

EE 4513

Assignment # 4

1 to 4-bit Subtractor and 4-bit Arithmetic Logic Unit Design

September 21, 2019

Part A1: 1-bit Subtractor Design

Procedure

A	B	Bin	Diff	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Figure 1: 1-bit Subtractor Truth Table

Bin\AB	00	01	11	10
0	0	1	0	1
1	1	0	1	0

Figure 2: Diff K-map

Bin\AB	00	01	11	10
0	0	1	0	0
1	1	1	1	0

Figure 3: Bout K-map

First, the 1-bit subtractor is designed at RTL. Given the truth table shown in Figure 1, two Karnaugh Maps (K-maps) are derived for each of the subtractor's outputs: the difference and the "carry out". For the difference of the subtractor Diff, Figure 2 solves the Boolean equation to be:

$$Diff = A'B'Bin + A'BBin' + ABBin + AB'Bin$$

As for the carry out Bout, Figure 3 solves the Boolean equation to be:

$$Bout = A'B + A'Bin + BBin$$

Given the abstraction level, the logic equations for the outputs are the only necessary statements needed to define the 1-bit subtractor. Now, a module for the 1-bit subtractor is created with the Verilog HDL.

The portlist reflects the inputs and outputs of the truth table shown in Figure 1.

```
1  `timescale 1ns / 1ps
2  module SUBTRACTOR(
3  input A, B, Bin,
4
5  output Diff, Bout
6      );
7
8  assign Diff = ~A & ~B & Bin | ~A & B & ~Bin | A & B & Bin | A & ~B & ~Bin;
9  assign Bout = ~A & B | ~A & Bin | B & Bin;
10
11
12 endmodule
```

Figure 4: 1-bit subtractor module

The only code necessary to define the operation of the 1-bit subtractor are two assign statements for deriving Diff and Bout. Figure 4 above displays the full code for the 1-bit subtractor module; bitwise operators are used for Verilog to implicitly choose AND, NOT, and OR gates. It is important to mention parentheses are omitted due to the precedence of ~ before

& and before |. The following section seeks to observe the realization of the module in schematic form before and after synthesis, as well as the behavioral simulation of the module.

Observations & Results

The 1-bit subtractor is now elaborated using Vivado's "Elaborated Design" workflow section, which essentially is an interpretation of the Verilog HDL before synthesis at RTL. Given the explicit Boolean equations used to define the behavior of the subtractor, Figure 5 displays the expected output of the design: an AND, OR, and INV network; this step is crucial in verifying the desired interpretation of the HDL developed.

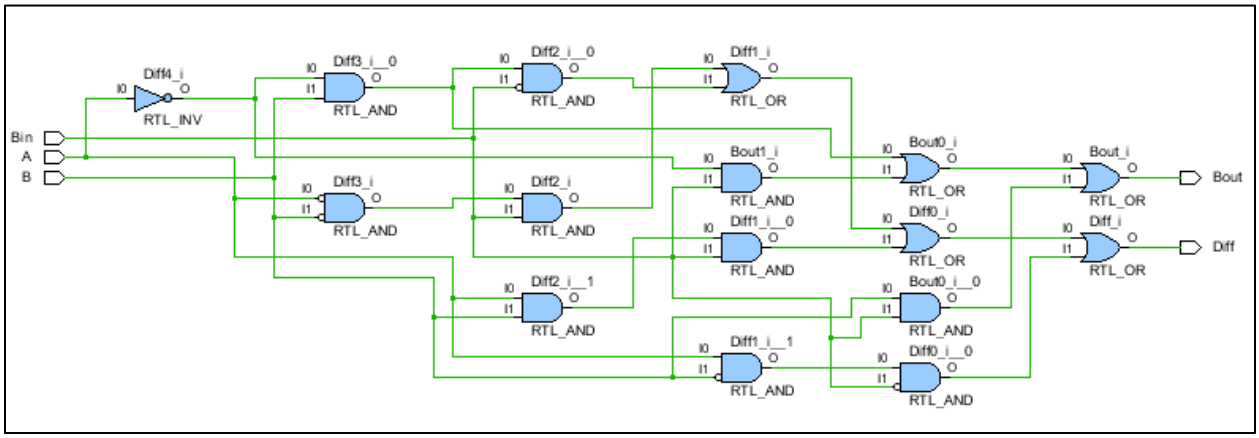


Figure 2: 1-bit subtractor elaborated design

Now, a testbench is created to verify the desired operation of the 1-bit subtractor. Given there are three 1-bit inputs, a total of 2^3=8 combinations are tested. Figure 5 below presents the results of the simulation of the module as the Unit Under Test (UUT).

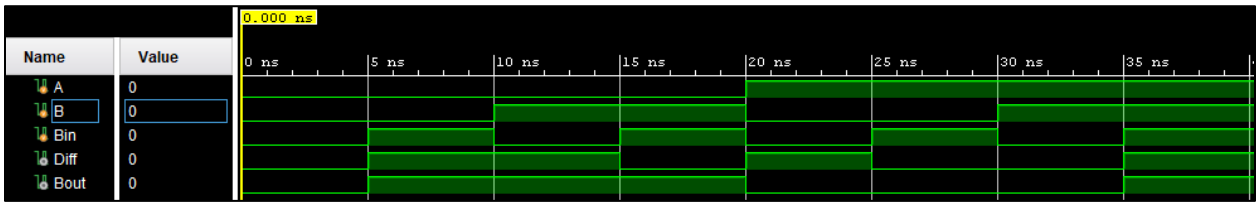


Figure 3: 1-bit subtractor behavioral simulation

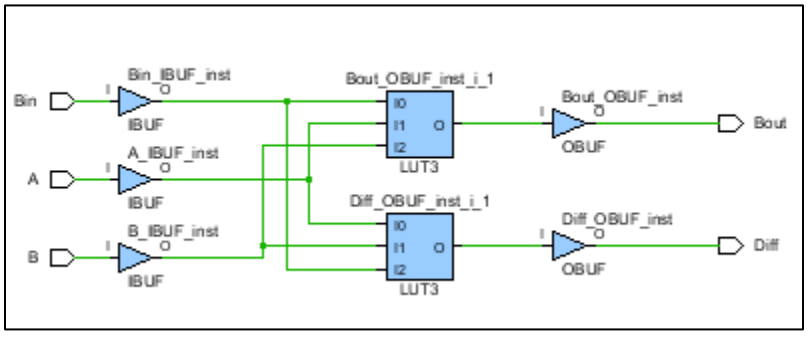


Figure 4: 1-bit subtractor synthesized design

As the simulation can attest, each possible combination corresponds to the expected output. With the elaborated design and simulation in check, the last step is to analyze the module's synthesis by the XST as displayed in Figure 6. The schematic showcases the use of 2-LUT3's due to two outputs having three inputs each; the IBUF's and OBUF's are there to communicate with the IOB's (Input-output blocks) of the FPGA.

Conclusion

The 1-bit subtractor was developed with a bottom-up approach at RTL. First the corresponding outputs were calculated with k-maps to derive minimized SOP Boolean equations. Second, the Verilog HDL is developed to implement the realized Boolean equations using assign statements for both Diff and Bout outputs. Third, the design is realized with Vivado's preliminary code interpretation in schematic form and verified for desired logic. Fourth, the design is simulated at the behavioral level to verify the correct operation of the subtractor. Lastly, the design is synthesized with XST and presented in schematic form.

The mentioned steps are followed in a successive manner to better formulate a verifiable design process. At any given step, if errors are to arise, the design can be retraced to the previous step to fix them before proceeding to the next. Above all, the bottom-up approach is useful to realize simple designs such as AND-OR-INV networks that will serve as submodules to a bigger design—in this case, the 1-bit subtractor. The following section entails of the development of a 4-bit subtractor using the 1-bit subtractor as a building block.

Part A2: 4-bit Subtractor Design

```
`timescale 1ns / 1ps
module SUBTRACTOR4(
    input [3:0] A,
    input [3:0] B,
    input Bin,
    output [3:0] Diff,
    output Bout
    );

    wire [2:0] carries;

SUBTRACTOR S0(
.A(A[0]),
.B(B[0]),
.Bin(Bin),

.Diff(Diff[0]),
.Bout(carries[0])
);
```

Figure 5: 4-bit subtractor portlist and structural design

Designing the 1-bit subtractor is a crucial step before developing the 4-bit subtractor due to its inherent purpose as a building block. Using a bottom-up approach for this design, the verified 1-bit subtractor can now be thought of as "fail-safe" to focus now on the verification of the 4-bit subtractor. In other words, the main logic of the 4-bit subtractor has been built, and now the construction of the design will be a more streamlined process. The following section explains this process for the 4-bit subtractor module.

Procedure

The module for the 4-bit subtractor is created using a structural approach. The internal logic of the subtractor consists of four 1-bit subtractor instantiations with internal wires to connect the ripple-carry chain of Bout carries. For the portlist, two 4-bit A and B operands as well as a preliminary Bin carry bit serve as inputs—one 4-bit difference output and a single bit carry out serve as outputs. Given the plain design, a snippet of the module is showcased in Figure 4 to the left. The next section explores the elaborated design, behavioral simulation, and synthesized result.

Observation and Results

First, similar to the 1-bit subtractor design process, the elaborated design of the 4-bit subtractor is presented to check for desired interpretation of the Verilog HDL as shown in Figure 6. As expected, the schematic showcases the structural approach used to build the module: four instantiated 1-bit subtractors with internal wires for the ripple-carry chain.

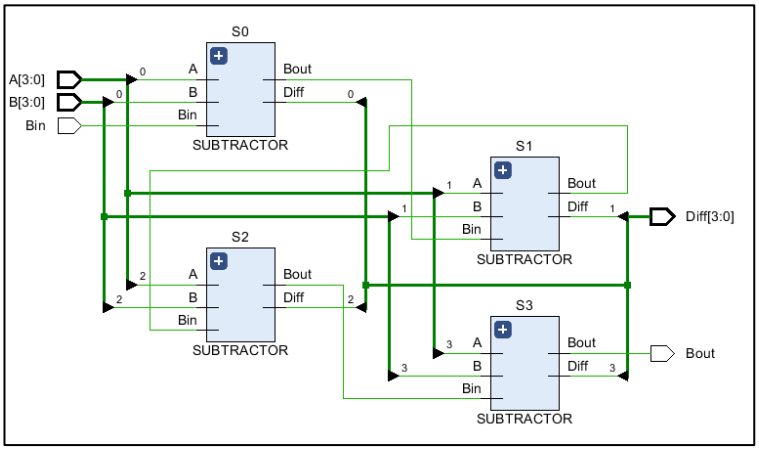


Figure 6: 4-bit subtractor elaborated design

Now, a behavioral simulation is conducted to verify the desired operation of the 4-bit subtractor. A testbench is created to instantiate the UUT, and an initial block used to specify the test cases. Compared to the 1-bit subtractor, only four cases are tested due to the increased complexity of verifying all the possible combinations = 2^4 * 2^4 * 2 = 512. As a result, Figure 7 showcases the four individual cases that include an overflow case as a valid benchmark for desired operation.

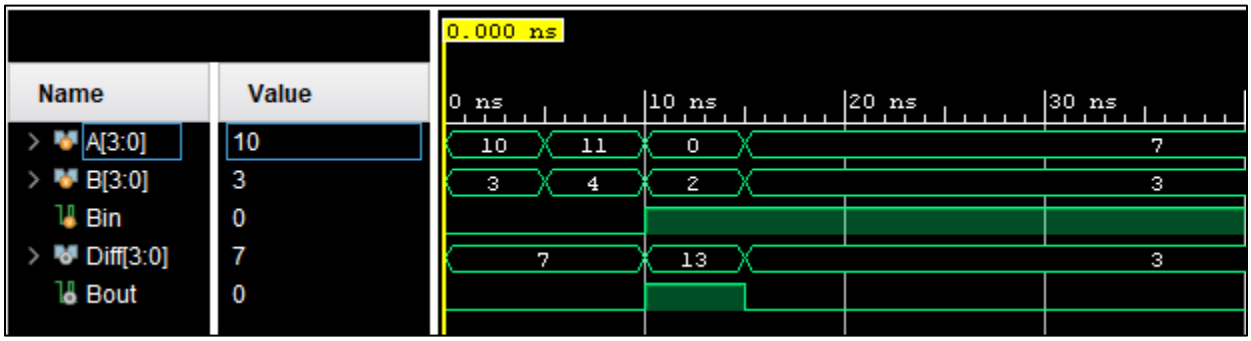


Figure 8: 4-bit subtractor behavioral simulation

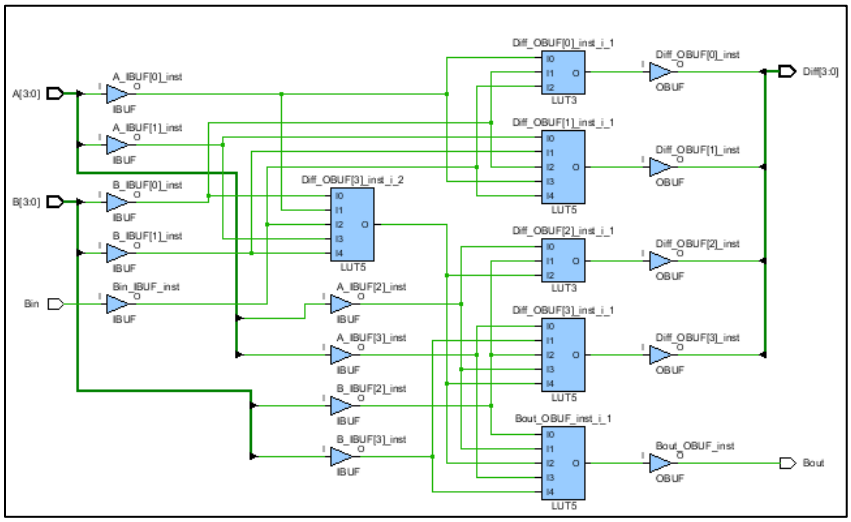


Figure 7: 4-bit subtractor synthesized design

With Figure 7 the module can be inferred to be working as desired, and the synthesized result from the XST is now presented next. In the high level, the 4-bit subtractor being now a 9-bit input with a 5-bit output function, the XST must improvise to implement a logic of such magnitude. Therefore, in addition to the utilized LUT3s in the 1-bit subtractor, now four LUT5s are used to define the rest of the logic shown in Figure 8.

Conclusion

The 4-bit subtractor is developed using a bottom-up approach at a structural level to better define the desired operation and to make use of the verified 1-bit subtractor. This approach proves that RTL is not always the best level to design a device in, and for this case, a structural approach provided an advantage as a fast way to develop the design. Further, it simplified the amount of testing done at a behavioral level due to the previous verification of the 1-bit subtractor. Above all, the higher one climbs in the bottom-up approach hierarchy, the faster the design development seems to be.

Part A2: 4-bit ALU Design

In contrast with the both subtractors' designs, the Arithmetic Logic Unit (ALU) is designed at a behavioral level of abstraction to ease the implementation. With a 3-bit opcode input, the ALU can carry out eight different operations as listed in Figure 8. Although more control is given to the Vivado interpreter and XST with a behavioral design, there would be a great increase in complexity when designing a submodule for each operation. The following section explains of the design procedure of the 4-bit ALU.

Control	Instruction	Operation
000	Add	Output <= A + B + Cin; Cout contains the carry
001	Sub	Output <= A – B - Cin; Cout contains the borrow
010	Or	Output <= A or B
011	And	Output <= A and B
100	Shl	Output <= A[2:0] & '0'
101	Shr	Output <= '0' & A[3:1]
110	Rol	Output <= A[2:0] & A[3]
111	Ror	Output <= A[0] & A [3:1]

Figure 9: ALU opcodes

Procedure

Due to a behavioral design, the development of the ALU consisted of two steps: create the portlist and create a procedural block for assessing the opcodes. First, the portlist creation is presented. For a typical ALU's arithmetic operations, there are two operands and one output; This design entails of a 4 bit architecture, and therefore the operands and output are 4-bits in length. To select the operation to carry out, a 3-bit opcode control signal establishes an input in the ALU. There is also a Cin input and Cout output available as the carry bit in addition and subtraction operations. Figure 10 displays the portlist of the ALU.

```
1   `timescale 1ns / 1ps
2   module ALU(
3   input [3:0] A,
4   input [3:0] B,
5   input Cin,
6   input [2:0] opcode,
7
8   output reg [3:0] ans,
9   output Cout
10
11  );
```

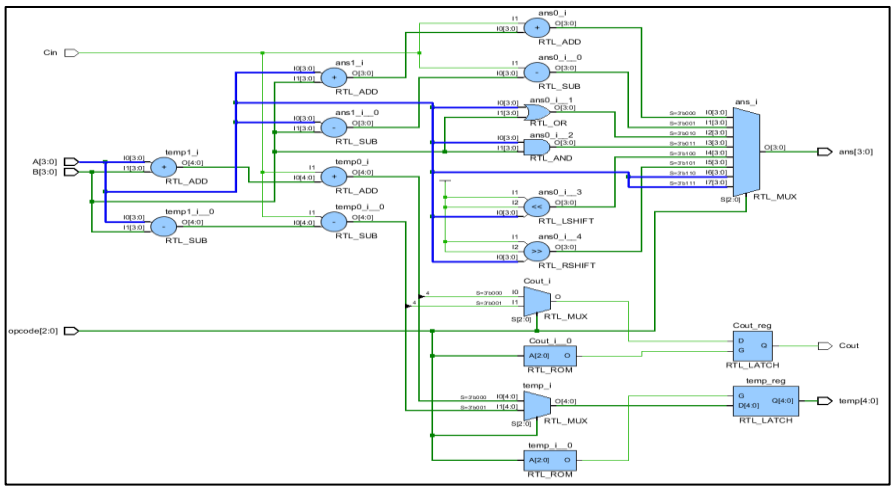
Figure 10: ALU port-list

With the portlist in place, the next step is to create a procedural always block to define the behavior of the ALU. Inside the block is a case statement to select from the opcode control signal the desired operation to carry out; the 4-bit output is transferred to register "ans" as shown in Figure 10. In this case, the sensitivity list consists of the wildcard asterisk to create a combinational design with the block. Now with the Verilog HDL completed, the next section explores the elaborated design, behavioral simulation, and synthesis results.

Observation and Results

The ALU is elaborated at RTL to verify the desired interpretation by Vivado. So far, the logic seems accurate except for I6 and I7 to the RTL_MUX that multiplexes the output register O: The A operand is simply shorted to these pins and is not being rotated left or right as opcodes 6 and 7 require. Figure 11 shows the elaborated design.

Figure 11: 4-bit ALU elaborated design



Despite the shorted input to the MUX, let's check the behavioral simulation for verifying the ALU's desired operation. A testbench is created to instantiate the 4-bit ALU as the UUT, and all eight combinations are tested. Figure 12 below showcases the ALU's output with the given test cases; the desired operation is verified.

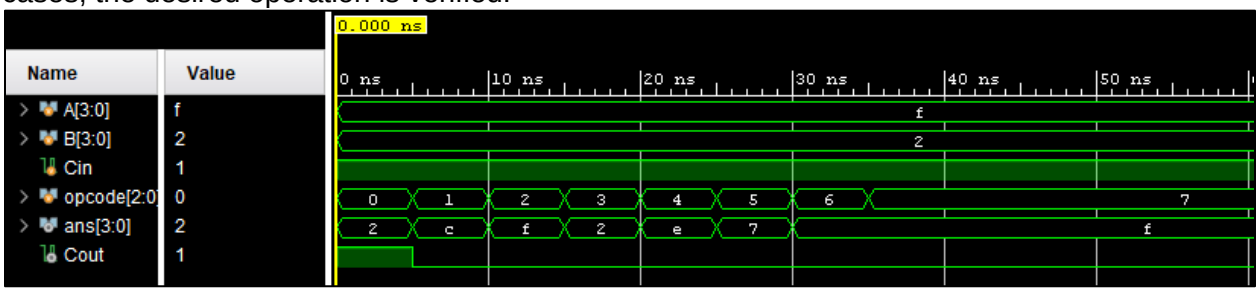
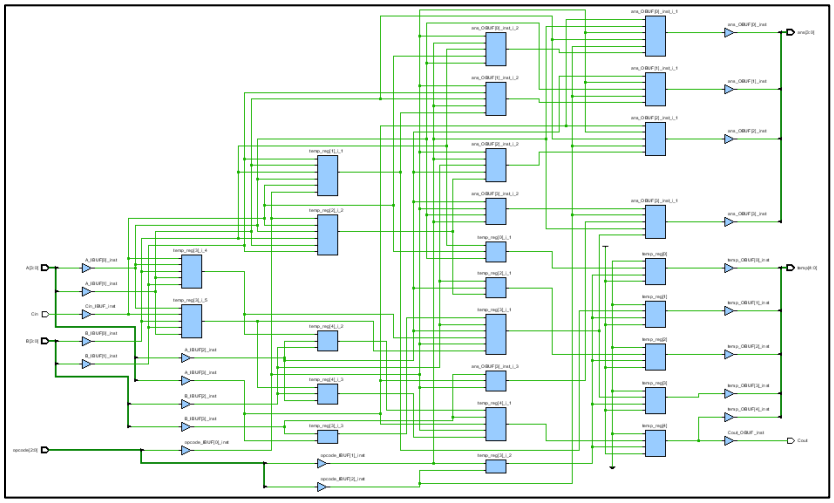


Figure 12: 4-bit ALU behavioral simulation



Since the behavioral simulation indeed verified the desired operation of the ALU, the next step is to now compare the synthesized design by XST to the elaborated design by the interpreted Verilog HDL—perhaps there is a reason why the input is shorted to the MUX. After synthesizing the module, the synthesized schematic is analyzed. Figure 13 showcases the result from the XST.

Figure 13: 4-bit ALU Synthesized result

As the schematic shows, there is no sign of shorts, but it is hard to tell due to the number of LUTs. Maybe there is an internal rewiring of the input ports to be switched in significance to implement the rotate left and right operations. After all, it is only necessary to rewire the LSB to the MSB and the MSB to the LSB for the rotate left and rotate right operations respectively.

Conclusion

Designing the 4-bit ALU was straightforward using a behavioral approach; it saved time over creating an individual module for each of the eight operations. However, there was an issue with establishing the carry out at first—a temporary 5-bit register had to be used to be able to "hold" the carry bit, then the carry out bit was assigned to the MSB of the temporary 5-bit register. On the same token, the ALU could have used a mix of both structural design and behavioral design to take advantage of the previous Labs' full adder designs.

The elaborated design did not truly explain what happened to the rotate left and right operations. Indeed, it must be due to the data width of the A operand, for the schematic can only represent a bus as one thick wire, and therefore the switched significance of the wire bits must be hidden. Further, coming up with an explanation by examining the synthesized result proved futile do to the complexity of the LUT network. The question is left unanswered and left for the reader to explain.

CODE

SUBTRACTOR.V

```
`timescale 1ns / 1ps

module SUBTRACTOR(

input A, B, Bin,


output Diff, Bout

   );


assign Diff = ~A & ~B & Bin | ~A & B & ~Bin | A & B & Bin | A & ~B & ~Bin;

assign Bout = ~A & B | ~A & Bin | B & Bin;



endmodule
```

SUBRACTOR_SIM.V

```
`timescale 1ns / 1ps

module SUBTRACTOR_SIM();
```

```
reg A, B, Bin;
wire Diff, Bout;

SUBTRACTOR UUT(
.A(A),
.B(B),
.Bin(Bin),

.Diff(Diff),
.Bout(Bout)
);



initial begin
{A, B, Bin} = 3'b000;
#5 {A, B, Bin} = 3'b001;
#5 {A, B, Bin} = 3'b010;
#5 {A, B, Bin} = 3'b011;

#5 {A, B, Bin} = 3'b100;
#5 {A, B, Bin} = 3'b101;
#5 {A, B, Bin} = 3'b110;
#5 {A, B, Bin} = 3'b111;

end

endmodule
```

SUBRACTOR4.V

```
`timescale 1ns / 1ps
module SUBTRACTOR4(
    input [3:0] A,
```

```
    input [3:0] B,
    input Bin,
    output [3:0] Diff,
    output Bout
    );

    wire [2:0] carries;

SUBTRACTOR S0(
.A(A[0]),
.B(B[0]),
.Bin(Bin),

.Diff(Diff[0]),
.Bout(carries[0])
);

SUBTRACTOR S1(
.A(A[1]),
.B(B[1]),
.Bin(carries[0]),

.Diff(Diff[1]),
.Bout(carries[1])
);

SUBTRACTOR S2(
.A(A[2]),
.B(B[2]),
.Bin(carries[1]),
```

```
.Diff(Diff[2]),
.Bout(carries[2])
);

SUBTRACTOR S3(
.A(A[3]),
.B(B[3]),
.Bin(carries[2]),

.Diff(Diff[3]),
.Bout(Bout)
);

Endmodule
```

SUBTRACTOR4_SIM.V

```
`timescale 1ns / 1ps
module SUBTRACTOR4_SIM();

reg [3:0] A,B;
reg Bin;

wire [3:0] Diff;
wire Bout;

SUBTRACTOR4 UUT(
.A(A),
.B(B),
.Bin(Bin),

.Diff(Diff),
.Bout(Bout)
```

```
);


initial begin
{Bin, A, B} = 9'h0_A_3;
#5 {Bin, A, B} = 9'h0_B_4;
#5 {Bin, A, B} = 9'h1_0_2;
#5 {Bin, A, B} = 9'h1_7_3;

end
endmodule
```

ALU.V

```
`timescale 1ns / 1ps
module ALU(
input [3:0] A,
input [3:0] B,
input Cin,
input [2:0] opcode,

output reg [3:0] ans,
output reg [4:0] temp,
output reg Cout

);

always@(*) begin
case(opcode)
3'b000: begin
        ans = A+B+Cin;
        temp = A+B+Cin;
        Cout = temp[4];
```

```
        end
3'b001: begin
            ans = A-B-Cin;
            temp = A-B-Cin;
            Cout = temp[4];
        end
3'b010: ans = A|B;
3'b011: ans = A&B;
3'b100: ans = A << 1; //shift left logical
3'b101: ans = A >> 1; //shift right logical
3'b110: ans = {A[2:0], A[3]}; //rotate left
3'b111: ans = {A[0], A[3:1]}; //rotate right
default: ans = 4'h0;

endcase




end

endmodule
```

ALU_SIM.V

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

reg [3:0] A, B;
reg Cin;
reg [2:0] opcode;

wire [3:0] ans;
//wire [4:0] temp;
```

```
wire Cout;

ALU UUT(
.A(A),
.B(B),
.Cin(Cin),
.opcode(opcode),

.ans(ans),
//.temp(temp),
.Cout(Cout)
);

initial begin
A = 4'hF;
B = 4'h2;
Cin = 1'b1;

opcode = 3'b000;
#5 opcode = 3'b001;
#5 opcode = 3'b010;
#5 opcode = 3'b011;
#5 opcode = 3'b100;
#5 opcode = 3'b101;
#5 opcode = 3'b110;
#5 opcode = 3'b111;

end

endmodule
```