

**Department of Computer and Software Engineering – ITU****SE201L: Digital Logic Design Lab**

<b>Course Instructor:</b> MS Aqsa Khalid	<b>Dated:</b> 18 – 09 – 2024
<b>Lab Engineer:</b> Muhammad Kashif	<b>Semester:</b> Fall 2024
<b>Batch:</b> BSSE 23 A	

**LAB 5      Dataflow Modeling in Verilog: Implementation of Adders  
and Subtractors**

<b>Name</b>	<b>Roll Number</b>	<b>Lab Marks</b>
Zunaira Abdul Aziz	BSSE23058	

Checked on: \_\_\_\_\_

Signature: \_\_\_\_\_

# Dataflow Modeling in Verilog: Implementation of Adders and Subtractors

## 5.1. Introduction

This lab exercise familiarizes students with dataflow modeling style in Verilog. In addition to that, adders and subtractors are studied and their implementation using Verilog dataflow modeling and hierarchical design techniques is carried out.

## 5.2. Objectives

This lab will enable students to achieve the following:

- Familiarize with Verilog syntax for dataflow modeling style
- Understand the working of half and full adders and subtractors
- Implement half and full adders using Verilog dataflow modeling
- Implement hierarchically multiple-bit adders and subtractors

## 5.3. Conduct of Lab

1. This lab experiment has to be performed using the ModelSim PE Student Edition, installed in Embedded Lab PCs.
2. Bring printout of this lab manual when you come to perform the lab.
3. You can work and get evaluated in groups of two. However, manual submission has to be separate.
4. If there is difficulty in understanding any aspect of the lab, please seek help from the lab engineer or the TA.
5. If a lab task contains an instruction to show working to lab engineer, make sure that the lab engineer evaluates and marks on your manual for that task. If your manual is unmarked for this task, it can result in marks deduction.
6. Complete the lab within the allocated time. Late submissions will be marked zero.
7. Print the codes and screenshots of simulation results of all tasks and attach these to your lab manual. Submit complete manual to the lab engineer no later than 24 hours after the lab.

## 5.4. Theory and Background

### 5.4.1. Binary adders

A *binary adder* is a digital logic circuit that performs addition of binary numbers. The most basic binary adder that adds two 1-bit numbers is called a *half adder*. *Full adder* is a 1-bit binary adder that adds three 1-bit numbers. All multiple-bit binary adders can be constructed using only half adders or full adders.

A half adder has two 1-bit inputs, *A* and *B*. It has two outputs, *Sum* and *Carry*. The output functions are given below:

$$\text{Sum} = A \oplus B$$

$$\text{Carry} = A \cdot B$$

A full adder has three 1-bit inputs *A*, *B* and *Cin*. Like a half adder, it also has two outputs, *Sum* and *Carry*, whose Boolean functions are given below:

$$\text{Sum} = A \oplus B \oplus \text{Cin}$$

$$\text{Carry} = A \cdot B + \text{Cin} \cdot (A \oplus B)$$

### 5.4.2. Binary subtractors

A *binary subtractor* is a digital logic circuit that subtracts two binary numbers. The most basic type of binary subtractor is a 1-bit *half subtractor*. A half subtractor can subtract a 1-bit number from another 1-bit number. A *full subtractor* is a 1-bit binary subtractor that subtracts two 1-bit numbers from a 1-bit number. All multiple binary subtractors can be constructed using only half subtractors or full subtractors.

A half subtractor has two inputs,  $A$  and  $B$ , and two outputs, *Difference* and *Borrow*. The half subtractor perform  $A-B$  and its output functions are as follows:

$$\text{Difference} = A \oplus B$$

$$\text{Borrow} = \bar{A}B$$

A full subtractor has three inputs,  $A$ ,  $B$  and  $\text{Bin}$ , and two outputs, *Difference* and *Borrow*. The full subtractor performs  $A-B-\text{Bin}$  and its output functions are as follows:

$$\text{Difference} = A \oplus B \oplus \text{Bin}$$

$$\text{Borrow} = \bar{A}B + \text{Bin} \cdot (\overline{A \oplus B})$$

### 5.4.3. Dataflow modeling in Verilog

*Dataflow modeling* in Verilog has a higher level of abstraction in syntax as compared to gate-level (structural) modeling. As the name implies, this type of modeling is based on the flow of data in a circuit.

Let us create a half adder module in Verilog and discuss its each line of code to understand the dataflow modeling syntax:

```
module half_adder (a, b, sum, carry);
  input a, b;
  output sum, carry;
  assign sum = a^b;
  assign carry = a&b;
endmodule
```

1. *module half\_adder (a, b, sum, carry);* This line declares a module, named half\_adder, with 4 ports, a, b, sum and carry.
2. *input a, b;* This line declares ports a and b as inputs.
3. *output sum, carry;* This line declares ports sum and carry as outputs.
4. *assign sum = a^b;* This line XORs a and b by using the XOR operator '^' and assigns the output to sum. Notice that, here, we are not concerning ourselves with the connections between the gates; we have just assigned a value (data obtained by manipulating data from two inputs) to our output. Table 5.1 lists a number of operators that can be used in dataflow modeling style.

**Table 5.1: Operators used in dataflow modeling**

Symbol	Operation
&	Bitwise AND
	Bitwise OR
~	Bitwise NOT
^	Bitwise XOR
+	Binary addition
-	Binary subtraction
>	Greater than
<	Less than
==	Equal to
{ }	Concatenation
? :	Conditional

In this lab, we will only use the logical and arithmetic operators; the relational operators will be used in coming labs.

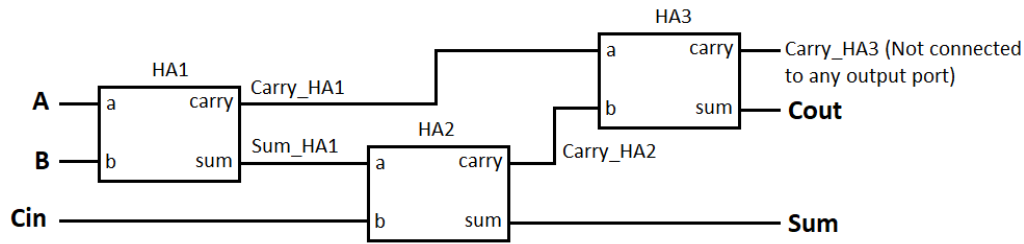
5. *assign carry = a&b;* This line assigns the AND of a and b to carry.
6. *endmodule* This line marks the end of half\_adder module.

#### 5.4.3.1. Hierarchical design in Verilog

In Verilog, we can re-use modules inside other modules by instantiation. This method of design is called *hierarchical design*. You must already be familiar with instantiation inside a test bench. Instantiation in a hierarchical design is no different from that form of instantiation. Let us look at an example of hierarchical design.

A 1-bit full adder can be constructed by using a few half adders and some inter-connections between them. A 1-bit full adder adds three 1-bit numbers a, b and C\_in. So using one half adder we will add a and b. This will give two 1-bit values, a sum and a carry. We will then use a second half adder to add the sum from the first half adder and Cin. This will again give a sum

and a carry. The sum from the second half adder will be our final sum bit. We will use another half adder to add the carries from previous two half-additions. This will again generate a sum and a carry. The sum of this third half-addition will be our final carry. The carry of this third half-addition will always be zero as addition of three 1-bit numbers cannot have more than two bits in the result. This whole setup is shown in the figure below:



**Figure 5.1: Full adder using half adders**

Verilog code using the hierarchy according to the above diagram is given below:

```
module full_adder (A, B, Cin, Sum, Cout);
    input A, B, Cin;
    output Sum, Cout;
    wire Sum_HA1, Cout_HA1, Cout_HA2, Cout_HA3;
    half_adder HA1(.a(A), .b(B), .sum(Sum_HA1), .carry(Cout_HA1));
    half_adder HA2(.a(Sum_HA1), .b(Cin), .sum(Sum), .carry(Cout_HA2));
    half_adder HA3(.a(Cout_HA1), .b(Cout_HA2), .sum(Cout), .carry(Cout_HA3));
endmodule
```

#### 5.4.3.2. Vectors in Verilog

The number of bits in data type *nets* and *regs* can be specified and such declarations of these data types are called *vectors*. Until now, we have only worked with *scalars* that were 1-bit wide. The standard syntax to declare a vector is:

```
Data_type [msb: lsb] Variable_name;
```

Below are a few examples of vector declaration:

```
output [1:0] a;
reg [3:1] b;
wire [2:-2] c;
```

*a* is declared a 2-bit wide *input*, *b* is declared a 3-bit wide *reg*, *c* is declared as a 5-bit *wire*. Vectors can be accessed bit-wise, or as a subset of indices, or as a whole as shown below:

```
assign a[1] = 1'b0;
b[3:2] = 2'b01;
assign c = 5'b10101;
```

In the first line, the MSB of *a* is assigned 1-bit value 0. In the second line, the higher two bits of *b* are loaded with 2-bit value 01. In third line, *c* is assigned a 5-bit value 10101.

## 5.5. Lab Tasks

### 5.5.1. Task 1: Full subtractor [Marks: 10]

1. Use the full subtractor equations given in 5.4.2 to implement a full subtractor module named “sub\_logical”. Use only the logical operators given in Table 5.1. [3]

```
module sub_logical (A, B, Bin, Bout, Diff);
    input A, B, Bin;
    output Bout, Diff;
    assign Diff = A ^ B ^ Bin;
    assign Bout = (~A & Bin) | (~A & B) | (B & Bin);
endmodule
```

2. Implement the full subtractor in another module named “sub\_arith” using only the arithmetic operators from Table 5.1.

```
module sub_arith (A, B, Bin, Bout, Diff);  
    input A, B, Bin;  
    output Bout, Diff;  
    assign Diff = A - B - Bin;  
    assign Bout = (A < B) | ((A - B) < Bin);  
endmodule
```

3. Write a test bench to test your sub\_logical module for all the input combinations. Fill the truth table in Table 5.2 according to the timing diagrams thus obtained: [2]

```
module test_sub_logical;  
    reg A, B, Bin;  
    wire Bout, Diff;  
    sub_logical logical(.A(A), .B(B), .Bin(Bin), .Bout(Bout), .Diff(Diff));  
    initial begin  
        A=0;  
        B=0;  
        Bin=0;  
        #100  
        A=0;  
        B=0;  
        Bin=1;  
        #100  
        A=0;  
        B=1;  
        Bin=0;  
        #100  
        A=0;  
        B=1;  
        Bin=1;  
        #100  
        A=1;  
        B=0;  
        Bin=0;  
        #100  
        A=1;  
        B=0;  
        Bin=1;  
        #100  
        A=1;  
        B=1;  
        Bin=0;  
        #100  
        A=1;  
        B=1;  
        Bin=1;  
    end  
endmodule
```

**Table 5.2: Truth table of full subtractor**

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

0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

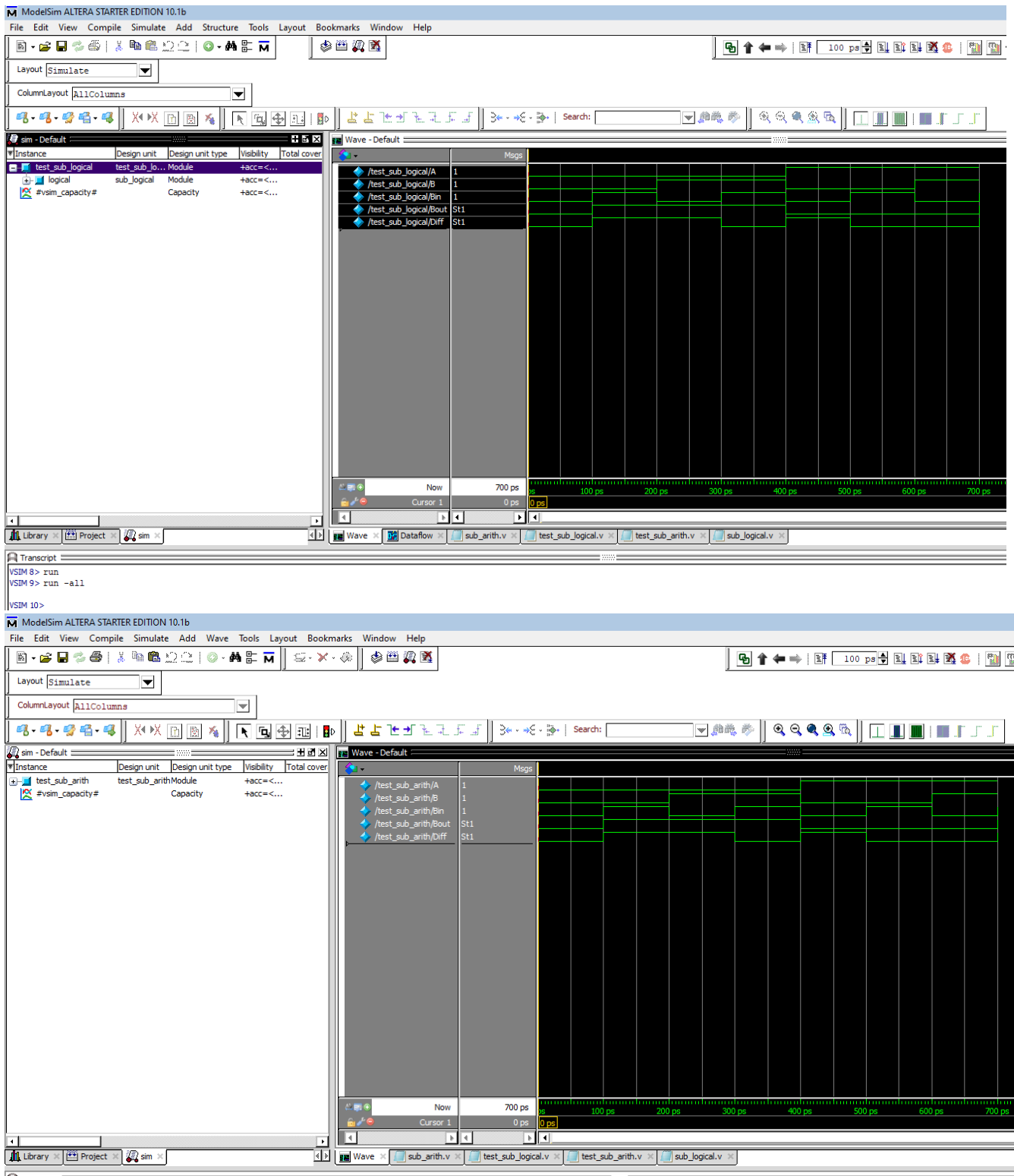
4. Verify the working of sub\_arith module by changing the instantiation in the same test bench. [2]

```

module test_sub_logical;
  reg A, B, Bin;
  wire Bout, Diff;
  sub_arith arith(.A(A), .B(B), .Bin(Bin), .Bout(Bout), .Diff(Diff));
  initial begin
    A=0;
    B=0;
    Bin=0;
    #100
    A=0;
    B=0;
    Bin=1;
    #100
    A=0;
    B=1;
    Bin=0;
    #100
    A=0;
    B=1;
    Bin=1;
    #100
    A=1;
    B=0;
    Bin=0;
    #100
    A=1;
    B=0;
    Bin=1;
    #100
    A=1;
    B=1;
    Bin=0;
    #100
    A=1;
    B=1;
    Bin=1;
  end
endmodule

```

5. Take clear screenshots of the code of all modules of this task and the simulation timing waveforms, and paste them in a WORD file such that all fit on a single page. Make sure that screenshots are legible. [3]



6. Show code and simulation results to the lab engineer to obtain credit.

### 5.5.2. Task 2: 2-bit adder module [Marks: 10]

1. In this task, you are required to design a 2-bit adder using full adders hierarchically. Your 2-bit adder should have two 2-bit inputs A and B, one 1-bit input Carry\_in, one 2-bit output Sum and one 1-bit output Carry\_out. In the space given below, draw a block diagram of your design. Use a rectangular block for the full adder. [5]

2. Write Verilog code according to your proposed block diagram. Use the following skeleton code as a starter: [3]

```
module Full_Adder (A,B,C,Carry_in, Sum, Carry_out);
    input A,
    input B,
    input Carry_in,
    output Sum,
    output Carry_out
    assign Sum = A ^ B ^ Carry_in;
    assign Carry_out = (A & B) | (A & Carry_in) | (B & Carry_in);
endmodule
```

```
module Adder_2bit (A, B, Carry_in, Sum, Carry_out);
    input [1:0] A, B;
    input Carry_in;
    output [2:0] Sum;    should be [1:0]
    output Carry_out;
    wire Carry_FA;
    Full_Adder firstAdder (.A(A[0]),.B(B[0]),.Carry_in(Carry_in),.Sum(Sum[0]),.Carry_out(Carry_FA));
    Full_Adder secondAdder (.A(A[1]),.B(B[1]),.Carry_in(Carry_FA),.Sum(Sum[1]),.Carry_out(Carry_out));
endmodule
```

3. Write a test bench for your 2-bit adder to test for the input values of A and B given in Table 5.3. Fill in the outputs in the table accordingly: [2]

```
module test_Adder_2bit;
    reg [1:0] A;
    reg [1:0] B;
    reg Carry_in;
    wire [2:0] Sum;    also here [1:0]
    wire Carry_out;
    Adder_2bit full_adder (.A(A),.B(B),.Carry_in(Carry_in),.Sum(Sum),.Carry_out(Carry_out));
    initial begin
```



```

A = 2'b00; B = 2'b00; Carry_in = 1;
#100
A = 2'b00; B = 2'b00; Carry_in = 0;
#100
A = 2'b01; B = 2'b10; Carry_in = 1;
#100
A = 2'b01; B = 2'b10; Carry_in = 0;
#100
A = 2'b11; B = 2'b11; Carry_in = 1;
#100
A = 2'b11; B = 2'b11; Carry_in = 0;
#100
end
endmodule

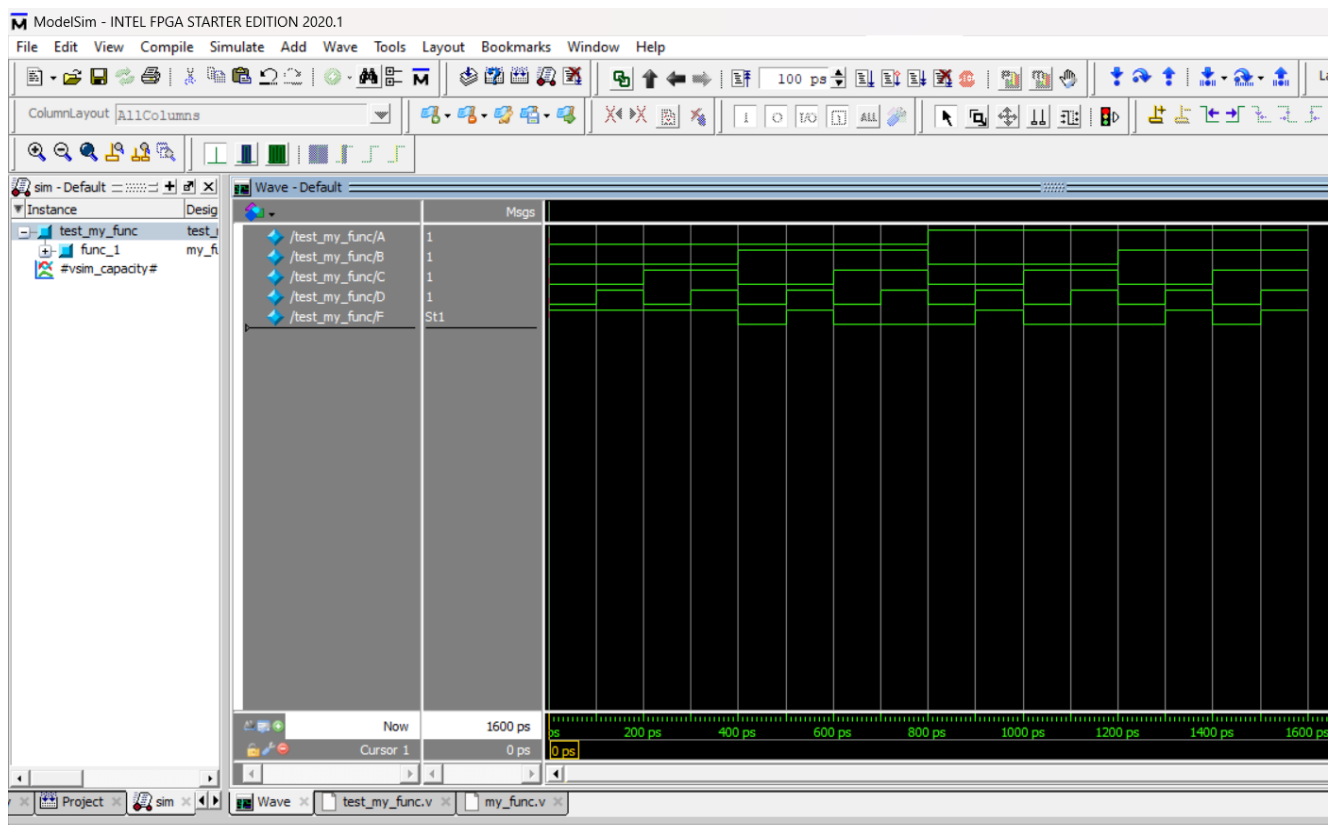
```

These sum values should be 2 bits sorry for the mistake

Table 5.3: Test inputs for 2-bit adder

A	B	Carry_in	Carry_out	Sum
2'b00	2'b00	1	0	001
2'b00	2'b00	0	0	000
2'b01	2'b10	1	0	011
2'b01	2'b10	0	0	011
2'b11	2'b11	1	1	111
2'b11	2'b11	0	1	110

- Take clear screenshots of the code of all modules of this task and the simulation timing waveforms, and paste them in your WORD file such that all fit on no more than two pages. Make sure that screenshots are legible.



- Show the timing waveforms and code to the lab engineer to obtain credit.

### 5.5.3. Task 3: 2-bit subtractor module [Marks: 10]

1. In this task, you are required to design a 2-bit subtractor using full subtractors hierarchically. Your 2-bit subtractor should have two 2-bit inputs A and B, one 1-bit input Borrow\_in, one 2-bit output Difference and one 1-bit output Borrow\_out. In the space given below, draw a block diagram of your design using rectangular blocks for full subtractors. [4]

2. Write Verilog code according to your block diagram. Use the given skeleton code as a starter: [2]

```
module Full_Subtractor (A,B,Borrow_in, Difference,Borrow_out);
    input A,
    input B,
    input Borrow_in,
    output Difference,
    output Borrow_out;
    assign Difference = A ^ B ^ Borrow_in;
    assign Borrow_out = (~A & B) | (~A & Borrow_in) | (B & Borrow_in);
endmodule

module Subtractor_2bit(A,B,Borrow_in, Difference, Borrow_out);
    input [1:0] A,B;
    input Borrow_in,
    output [2:0] Difference,    also [1:0]
    output Borrow_out;
    wire Borrow_FS;
    Full_Subtractor
    firstsubtractor(.A(A[0]),.B(B[0]),.Borrow_in(Borrow_in),.Difference(Difference[0]),.Borrow_out(Borrow_FS));
    Full_Subtractor
    secondsubtractor(.A(A[1]),.B(B[1]),.Borrow_in(Borrow_FS),.Difference(Difference[1]),.Borrow_out(Borrow_out));
endmodule
```

3. Write a test bench for your 2-bit subtractor to test for the input values of A and B as given in Table 5.4. Fill in the table with the corresponding outputs. [4]

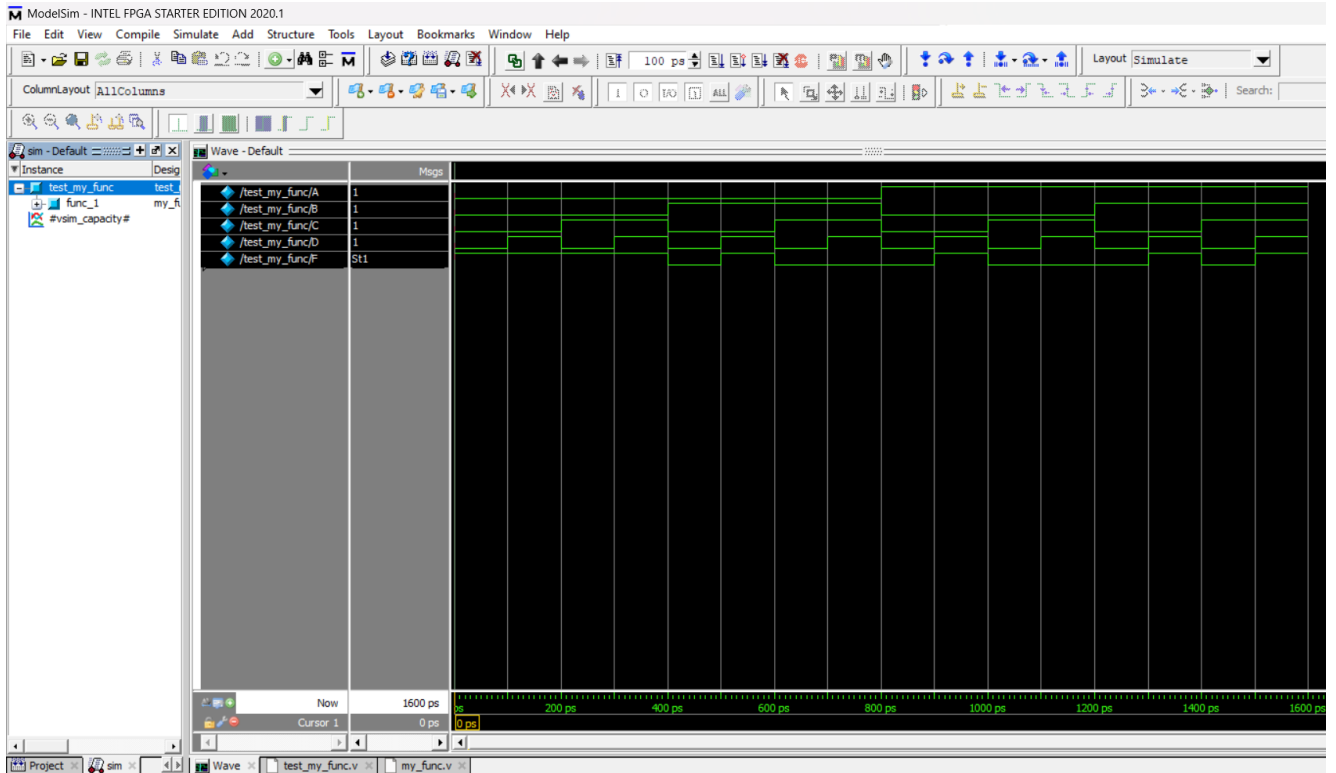
**Table 5.4: Test inputs for 2-bit subtractor**

A	B	Borrow_in	Borrow_out	Difference
2'b00	2'b00	1	0	001

the difference value should be 2 bit not 3 bit

2'b00	2'b00	0	0	000
2'b01	2'b10	1	0	111
2'b01	2'b10	0	0	111
2'b11	2'b11	1	1	000
2'b11	2'b11	0	1	001

4. Take clear screenshots of the code of all modules of this task and the simulation timing waveforms, and paste them in your WORD file such that all fit on no more than two pages. Make sure that screenshots are legible.



5.

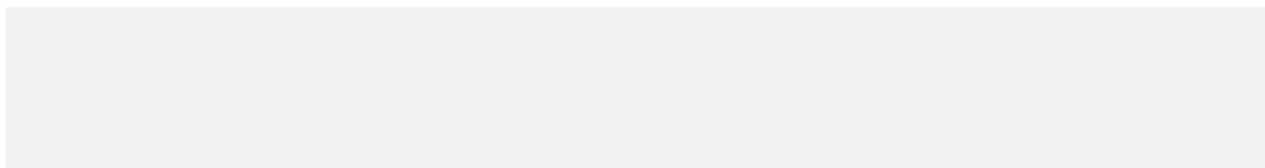
#### 5.5.4. Analysis [Marks: 5]

1. Using the binary subtraction operator (-) and concatenation ({}), write Verilog code to implement subtraction between two 2-bit numbers A and B. [2]

2. Why are vectors used in Verilog? [1]

3. What is meant by hierarchical design in a Verilog code? [1]

4. Does your proposed diagram give the subtracted result in two's complement form? Verify with the help of one example from the Table 5.4. [1]



# Assessment Rubric

## Method:

Lab report evaluation and instructor observation during lab sessions.

Performance	CLO	Able to complete the tasks over 80% (4-5)	Able to complete the tasks 50 – 80% (2-3)	Tasks completion below 50% (0-1)	Marks
1. Teamwork	1	Actively engages and cooperates with other group members in an effective manner	Cooperates with other group members in a reasonable manner	Distracts or discourages other group members from conducting the experiments	
2. Laboratory safety and disciplinary rules	1	Observes lab safety rules; handles the development board and other components with care and adheres to the lab disciplinary guidelines aptly	Observes safety rules and disciplinary guidelines with minor deviations	Disregards lab safety and disciplinary rules	
3. Realization of experiment	3	Conceptually understands the topic under study and develops the experimental setup accordingly	Needs guidance to understand the purpose of the experiment and to develop the required setup	Incapable of understanding the purpose of the experiment and consequently fails to develop the required setup	
4. Conducting experiment	3	Sets up hardware/software properly according to the requirement of experiment and examines the output carefully	Makes minor errors in hardware/software setup and observation of output	Unable to set up experimental setup, and perform the procedure of experiment	
5. Data collection	3	Completes data collection from the experiment setup by giving proper inputs and observing the outputs, complies with the instructions regarding data entry in manual	Completes data collection with minor errors and enters data in lab report with slight deviations from provided guidelines	Fails at collecting data by giving proper inputs and observing output states of experiment setup, unable to fill the lab report properly	
6. Data analysis	3	Analyzes the data obtained from experiment thoroughly and accurately verifies it with theoretical understanding, accounts for any discrepancy in data from theory with sound explanation, where asked	Analyzes data with minor error and correlates it with theoretical values reasonably. Attempts to account for any discrepancy in data from theory	Unable to establish the relationship between practical and theoretical values and lacks the theoretical understanding to explain any discrepancy in data	
7. Computer use	3	Successfully uses lab PC and internet to look for relevant datasheets, carry out calculations, or verify results using simulation	Requires assistance in looking for IC datasheets and carrying out calculation and simulation tasks	Does not know how to use computer to look up datasheets or carry out calculation and simulation tasks	
				<b>Total</b> (out of 35)	

```

module Full_Adder (
    input A,
    input B,
    input Carry_in,
    output Sum,
    output Carry_out
);
    assign Sum = A ^ B ^ Carry_in;
    assign Carry_out = (A & B) | (A & Carry_in) | (B & Carry_in);
endmodule

```

```

module Adder_2bit (
    input [1:0] A,
    input [1:0] B,
    input Carry_in,
    output [1:0] Sum,
    output Carry_out
);
    wire Carry_FA;

    Full_Adder firstAdder
    (.A(A[0]), .B(B[0]), .Carry_in(Carry_in), .Sum(Sum[0]), .Carry_out(Carry_FA));

    Full_Adder secondAdder
    (.A(A[1]), .B(B[1]), .Carry_in(Carry_FA), .Sum(Sum[1]), .Carry_out(Carry_out));
endmodule

```

```

module test_Adder_2bit;

```

```
reg [1:0] A;  
reg [1:0] B;  
reg Carry_in;  
wire [1:0] Sum;  
wire Carry_out;
```

```
    Adder_2bit full_adder  
    (.A(A), .B(B), .Carry_in(Carry_in), .Sum(Sum), .Carry_out(Carry_out));
```

```
initial begin  
    A = 2'b00; B = 2'b00; Carry_in = 1;  
    #100;  
    A = 2'b00; B = 2'b00; Carry_in = 0;  
    #100;  
    A = 2'b01; B = 2'b10; Carry_in = 1;  
    #100;  
    A = 2'b01; B = 2'b10; Carry_in = 0;  
    #100;  
    A = 2'b11; B = 2'b11; Carry_in = 1;  
    #100;  
    A = 2'b11; B = 2'b11; Carry_in = 0;  
    #100;  
end
```

```
endmodule
```

```
module Full_Subtractor (
```

```

    input A,
    input B,
    input Borrow_in,
    output Difference,
    output Borrow_out
);
    assign Difference = A ^ B ^ Borrow_in;
    assign Borrow_out = (~A & B) | (~A & Borrow_in) | (B & Borrow_in);
endmodule

```

```

module Subtractor_2bit (
    input [1:0] A,
    input [1:0] B,
    input Borrow_in,
    output [1:0] Difference,
    output Borrow_out
);
    wire Borrow_FS;

    Full_Subtractor firstsubtractor (
        .A(A[0]),
        .B(B[0]),
        .Borrow_in(Borrow_in),
        .Difference(Difference[0]),
        .Borrow_out(Borrow_FS)
    );

```



```

Full_Subtractor secondsubtractor (
    .A(A[1]),
    .B(B[1]),
    .Borrow_in(Borrow_FS),
    .Difference(Difference[1]),
    .Borrow_out(Borrow_out)
);
endmodule

```

```

module test_Subtractor_2bit;

    reg [1:0] A;
    reg [1:0] B;
    reg Borrow_in;
    wire [1:0] Difference;
    wire Borrow_out;

    Subtractor_2bit uut (
        .A(A),
        .B(B),
        .Borrow_in(Borrow_in),
        .Difference(Difference),
        .Borrow_out(Borrow_out)
    );

    initial begin

        A = 2'b00; B = 2'b00; Borrow_in = 1; #100;

        A = 2'b00; B = 2'b00; Borrow_in = 0; #100;
    end

```

A = 2'b01; B = 2'b10; Borrow\_in = 1; #100;

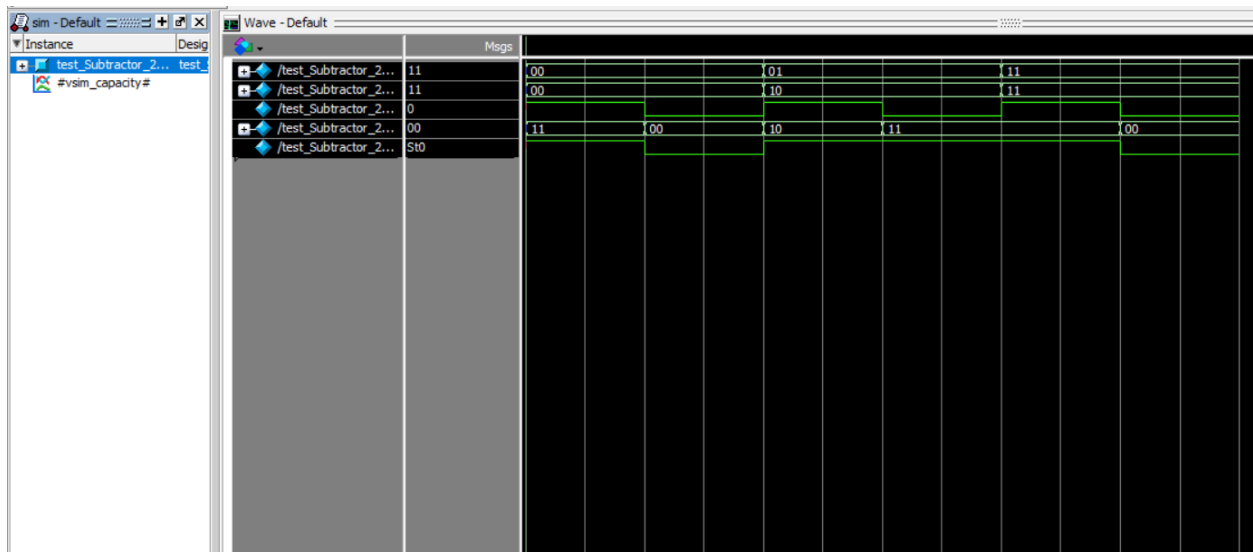
A = 2'b01; B = 2'b10; Borrow\_in = 0; #100;

A = 2'b11; B = 2'b11; Borrow\_in = 1; #100;

A = 2'b11; B = 2'b11; Borrow\_in = 0; #100;

end

endmodule



A	B	<u>Borrow_in</u>	<u>Borrow_out</u>	Difference
2'b00	2'b00	1	1	11
2'b00	2'b00	0	0	00
2'b01	2'b10	1	1	10
2'b01	2'b10	0	1	11
2'b11	2'b11	1	1	11
2'b11	2'b11	0	0	00

Table 5.3: Test inputs for 2-bit adder

A	B	<u>Carry_in</u>	<u>Carry_out</u>	Sum
2'b00	2'b00	1	0	01
2'b00	2'b00	0	0	00
2'b01	2'b10	1	1	00
2'b01	2'b10	0	0	11
2'b11	2'b11	1	1	11
2'b11	2'b11	0	1	10