EEL2020 Digital Design Lab Report
Sem II AY 2023-24

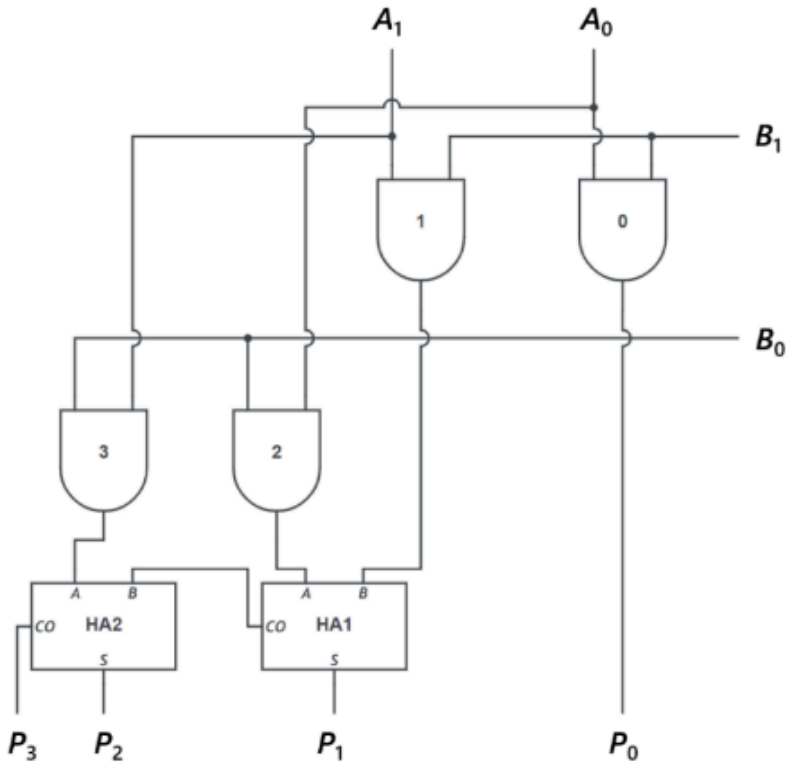
Experiment No.	**:**	**06**
Name	**:**	**Maulik Desai**
Roll No.	**:**	**B22CS033**
Partner Name (Partner Roll No.)	**:**	**Jay Mehta (B22CS034)**

Part 1

Objective

(i) Design, simulate and implement a 2-bit multiplier using half adders and AND gates

Logic Design



Expected circuit diagram

We are implementing a two bit multiplier using AND gates and half adders. The inputs in the depicted circuit are two two-bit inputs, A and B. The logic for the above expected circuit is as follows.

A two-bit multiplier can be implemented as follows:

		A_1	A_0
	×	B_1	B_0
		A_0B_1	A_0B_0
+	A_1B_1	A_1B_0	x
C_2	$A_1B_1 + C_1$	$A_0B_1 + A_1B_0$	A_0B_0

A one-bit multiplication can be implemented using a two-input AND gate.
The 2-bit multiplication output be a 4-bit product ($P_3P_2P_1P_0$) would involve

- $P_0 = A_0B_0$: Simple product of LSBs (Can be implemented with one AND gate)
- P_1 = Addition of A_0B_1 and A_1B_0 (can be implemented with a half adder)
- P_2 = Addition of A_1B_1 with carry of the previous HA (can be implemented with another half adder)
- P_3 = Carry out of the second HA, if any.

Source Description

- Design source

The source file is _2bitMulti.v which has two modules, a half adder module which performs simple addition of two one bit inputs to give out two outputs, sum and carry. The other module is the two bit multiplier module which uses instances of the half adder modules along with AND gates to implement the above mentioned logic design.

- Constraint file

The PYNQ XDC file was updated with the following changes:

Ports (from Verilog module)	**Designation** (Input/Output)	PYNQ Component Type (Button/LED/Switch etc. along with number, eg. LD01, BTN2, etc.)	Pin Configuration (from the PYNQ User Manual)
P[3]	Output	LED3	M14
P[2]	Output	LED2	N16
P[1]	Output	LED1	P14
P[0]	Output	LED0	R14

The RPI XDC file was updated with the following changes:

Ports (from Verilog module)	**Designation** (Input/Output)	RPI Component Type (Button/LED/Switch etc. along with number, eg. LD01, BTN2, etc.)	Pin Configuration (from the RPI User Manual)
B[0]	Input	RPIO_18	C20
B[1]	Input	RPIO_19	Y8
A[2]	Input	RPIO_20	A20
A[3]	Input	RPIO_26	W9

- Simulation source

We have used a test bench which tests for the following cases (each case runs for 10ns):-

1) 0*2 = 0
   (00*10=0000)

2) 3*1 = 3
   (11*01=0011)

3) 2*3=6
   (10*11=0110)

4) 3*0=0
   (11*00=0000)

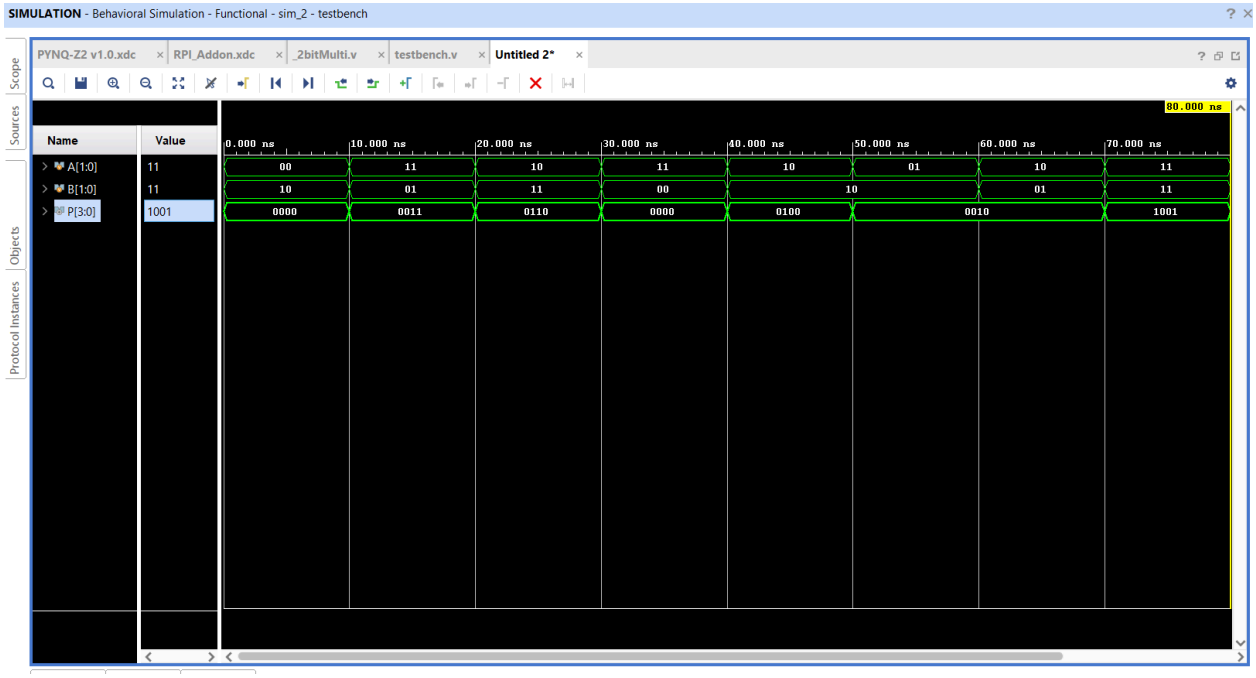
5) 2*2=4
   (10*10=0100)

6) 1*2=2
   (01*10=0010)

7) 2*1=2
   (10*01=0010)

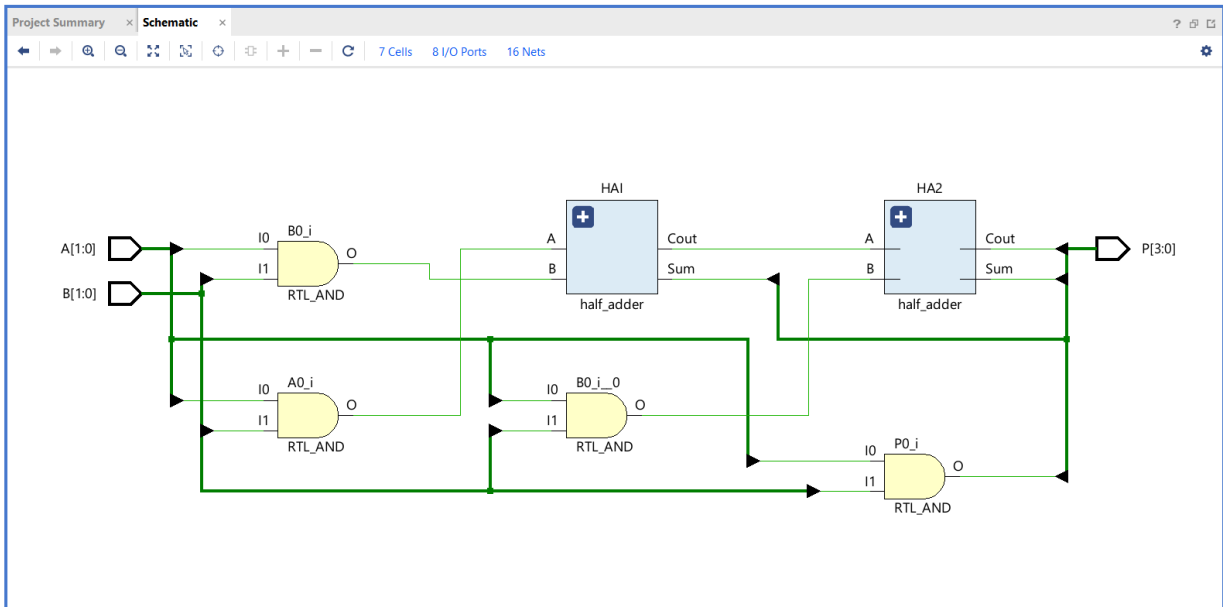
8) 3*3=9
   (11*11=1001)

Simulation Results (Timing diagram)



All the test cases mentioned above can be verified from the timing diagram.

Elaborated Design (from VIVADO)



We can see that the elaborate design matches the expected logic design

Codes:

1. **_2bitMulti.v**

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 11.03.2024 14:06:07
// Design Name:
// Module Name: two_bit_multiplier
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module half_adder (input A, input B, output Sum, output Cout);
assign {Cout, Sum} = A+B;
endmodule

module two_bit_multiplier (input [1:0] A, B, output [3:0] P);
wire C;
assign P[0] = A[0] & B[0];
half_adder HAI ((A[1] & B[0]), (A[0] & B[1]), P[1], C);
half_adder HA2 (C, (A[1] & B[1]), P[2], P[3]);
endmodule
```

---

2. **testbench.v**

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 11.03.2024 14:27:30
// Design Name:
// Module Name: testbench
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module testbench();
reg [1:0] A, B;
wire [3:0] P;
two_bit_multiplier tb(.A(A), .B(B),.P(P));

initial begin
// Test 1 -- 0*2 = 0
A=2'b00;
B =2'b10;
#10

// Test 2 -- 3*1 = 3
A=2'b11;
B=2'b01;
#10

// Test 3 -- 2*3 = 6
```

```
A=2'b10;
B=2'b11;
#10

// Test 4 -- 3*0 = 0
A=2'b11;
B=2'b00;
#10

// Test 5 -- 2*2 = 4
A=2'b10;
B=2'b10;
#10

// Test 6 -- 1*2 = 2
A=2'b01;
B=2'b10;
#10

// Test 7 -- 2*1 = 2
A=2'b10;
B=2'b01;
#10

// Test 8 -- 3*3 = 9
A=2'b11;
B=2'b11;
#10

$finish;
end
endmodule
```

Part 2

Objective

(ii) Design and simulate an Arithmetic Logic Unit (ALU) that performs eight operations selected using operation codes

Logic Design

An Arithmetic Logic Unit (ALU) is a fundamental component of a CPU (Central Processing Unit) that performs arithmetic and logical operations on operands. It is responsible for executing operations like addition, subtraction, AND, OR, etc., depending on the instruction provided by the control unit. The instruction from the control unit can be provided in the form of an operation code.
The operation table is shown below:-

Opcode			Operation
0	0	0	Reset
0	0	1	A+B
0	1	0	A-B
0	1	1	A AND B
1	0	0	A OR B
1	0	1	A XOR B
1	1	0	A XNOR B
1	1	1	Preset

Source Description

- Design source

The source file is ALU.v which includes the ALU module which implements the above suggested arithmetic logic unit.

- Simulation source

We have used a test bench which tests for all the possible cases (each case runs for 10ns):-
I implemented the ALU on two separate input cases:-

1. A = 9 (1001), B = 14 (1110)

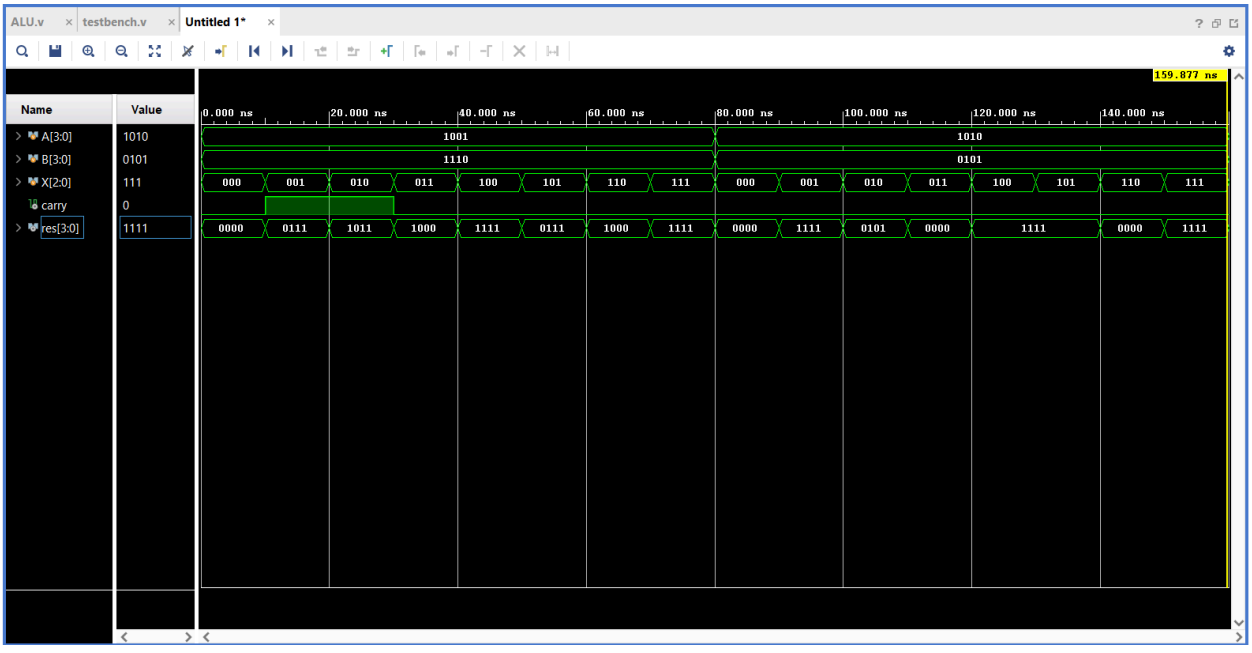
Input			Output				
			Carry	Res			
0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	1
0	1	0	1	1	0	1	1
0	1	1	0	1	0	0	0
1	0	0	0	1	1	1	1
1	0	1	0	0	1	1	1
1	1	0	0	1	0	0	0
1	1	1	0	1	1	1	1

2. A = 10 (1010), B = 5 (0101)

Input			Output				
			Carry	Res			
0	0	0	0	0	0	0	0
0	0	1	0	1	1	1	1
0	1	0	0	0	1	0	1
0	1	1	0	0	0	0	0
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	0	0	0	0
1	1	1	0	1	1	1	1

Simulation Results (Timing diagram)

The validity of the above truth table can be verified from the timing diagram.



Codes:

3. **ALU.v**

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 11.03.2024 14:51:06
// Design Name:
// Module Name: ALU
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module ALU(input [3:0]A, [3:0]B, [2:0]X, output reg [3:0]res, reg carry);
always @(*) begin
case(X)
3'b000: {carry,res} = {0, 4'b0000};
3'b001: {carry,res} = A+B;
3'b010: {carry,res} = A-B;
3'b011: {carry,res} = {0, A&B};
3'b100: {carry,res} = {0, A|B};
3'b101: {carry,res} = {0, A^B};
3'b110: {carry,res} = {0, ~(A^B)};
3'b111: {carry,res} = {0, 4'b1111};
endcase
end
endmodule
```

4. **testbench.v**

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 11.03.2024 15:01:04
// Design Name:
// Module Name: testbench
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module testbench();
reg [3:0]A;
reg [3:0]B;
reg [2:0]X;
wire carry;
wire [3:0]res;
ALU uu(.A(A), .B(B), .X(X), .carry(carry), .res(res));

initial begin
A = 4'b1001; B = 4'b1110;
X = 3'b000;#10;
X = 3'b001;#10;
X = 3'b010;#10;
X = 3'b011;#10;
X = 3'b100;#10;
```

```
X = 3'b101;#10;
X = 3'b110;#10;
X = 3'b111;#10;

A = 4'b1010; B = 4'b0101;
X = 3'b000;#10;
X = 3'b001;#10;
X = 3'b010;#10;
X = 3'b011;#10;
X = 3'b100;#10;
X = 3'b101;#10;
X = 3'b110;#10;
X = 3'b111;#10;

$finish;
end
endmodule
```
