



National University of Sciences and Technology
School of Electrical Engineering and Computer Science

EE-421 DIGITAL SYSTEM DESIGN

Assignment No-1

Date of submission: 07/03/2023

Section: BEE 12-C

Submitted to: Dr. Rehan Ahmed

Name	CMS ID
Nasrun Sithara Ramees	356480
Ahamed Rishard	356482

Building a 4-bit Microprocessor using Verilog HDL

Assignment no.1

EE-421 Digital Systems Design Deadline: 7th March 2023 by 10pm

Contents

1	Administrivia	3
1.1	Learning Objectives	3
1.2	Deliverables and Time-line.....	3
1.3	Marks Distribution	3
2	Introduction	4
3	Arithmetic Logic Unit	4
3.1	Combinational Logic	4
3.2	Circuit Assembly	5
4	Data Registers	5
4.1	Functionality and Time Synchronization	6
4.2	Testing	8
4.3	Output Register	9
4.4	Register De-multiplexing	9
5	Control Logic	10
5.1	Instruction Memory.....	12
5.2	Program Counter	12
5.3	Instruction Encoding.....	13

6	<i>Programming</i>	<i>14</i>
6.1	Fibonacci Numbers	16
7	<i>Concluding Remarks</i>	<i>16</i>

1 Administrivia

1.1 Learning Objectives

This assignment will enable you to,

- Understand the basic principles of computing.
- Understand how micro-architecture of a computer processor is derived from an instruction-set architecture (ISA).
- Understand how a micro-architecture is expressed using Verilog HDL.
- Understand bare metal programming stack.
- Relate your DLD circuits building using discrete ICs with this course.

1.2 Deliverables and Time-line

You are required to deliver and demonstrate the working of this assignment on an FPGA board (you are free to use any board available in the lab). You are required to verify the working of your design thoroughly in simulation before making your way to FPGA board. You are also required to submit a report on LMS showing the simulation snapshots of your final design and the features implemented. A thorough understanding of the assignment is necessary which will help you in oral viva. **Please note that the deadline of this assignment is March 7, 2023 by 10pm.** No submission will be accepted after the deadline, whatsoever! Your demo will be evaluated during the lab time on March 8, 2023.

1.3 Marks Distribution

The following marks distribution is tentative and is subjected to change without any prior notice to the students.

Hardware Demo	Viva	Total Marks
80	20	100

2 Introduction

This assignment will introduce a design of a simple 4-bit number crunching machine (which you may also call as 4-bit Microprocessor) from a customISA using Verilog HDL. In doing so, you'll also create the bare-metal programming of your microprocessor.

The design is broadly divided into three parts, namely - Arithmetic Logic Unit (ALU), Data Registers and Control Logic. You must use the hierarchical design approach, as discussed in the class, and therefore must define the following modules (at-least) and stitch them in the design top. You are free to use any level of abstraction within these modules.

- 4-bit D-type Registers
- 4-bit Binary Full Adder
- 4-bit Binary Counter
- Quad 2-Data Selectors
- 2-Line to 4-Line Decoder
- AND, OR, NOT and XOR Gates
- Parallel Address, Parallel 8-bit I/O EEPROM

3 Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is the most basic and important part of any computing machine. All the arithmetic or logical operations are performed through it. The ALU used in this assignment will be able to perform binary addition as well as 2's complement binary subtraction.

3.1 Combinational Logic

The circuit requirement is to add or subtract two 4-bit numbers and generate a carry. In order to choose between add and subtract operations, we will be

using a selection bit. The boolean equation for such a circuit can be realized as,

$$Out = (A + B)S' \text{ OR } (A - B)S$$

Here “A” and “B” are the two inputs, “Out” is the output and “S”, the selection bit.

Subtraction will be carried out by inverting the bits (i.e. taking 1’s complement) of B and raising the “carry in” of adder to logic 1, in order to add an extra bit which will eventually generate 2’s complement of B.

$$A - B = A + (1's \text{ complement of } B) + 1 (\text{carry in})$$

$$A - B = A + (2's \text{ complement of } B)$$

For inverting the bits we will use XOR gate with one input tied to the selection bit S, and the other to the input bit, such that when selection bit goes 1, the property of XOR, $B \oplus 1 = B'$ can be used and when it is 0 the input passes unaffected $B \oplus 0 = B$.

3.2 Circuit Assembly

The block diagram of ALU is shown in Figure 1.

<i>Selection bit (S)</i>	Function
0	$Out = A + B$
1	$Out = A - B$

4 Data Registers

At least two 4-bit registers are required to hold the data for ALU. The output of these two registers, say R_A and R_B , are directly connected to the two inputs of the ALU, A and B respectively. The output of these registers is always enabled i.e. they are always channeling data into the ALU. However the input to these registers is controlled and data can only enter into them when the input enable bit of R_A and R_B is at logic 1.

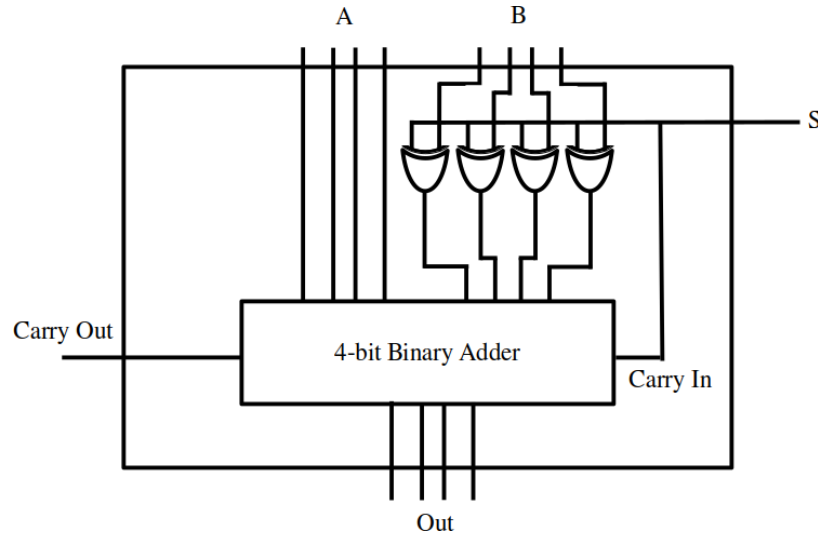


Figure 1: Arithmetic Logic Unit (ALU)

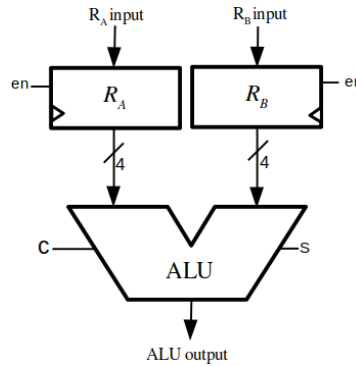


Figure 2: Register-ALU Assembly

4.1 Functionality and Time Synchronization

To start the computation, we need to load some initial values to R_A and R_B . Moreover, we would also like to utilize these registers to save the output of ALU too. In order to achieve this dual functionality, we need to add a 4-bit 1-out-of-2 data MUX before the input of registers. One of the MUX inputs would be connected to ALU's output and the other one to custom input. The selection bit of this MUX, say S_{reg} must be 0 to let the ALU's data pass

through and 1 for custom value. The updated datapath is shown in Figure 3.

A small D flip-flop is also placed adjacent to ALU to store the carry bit into it on positive edge of clock so that we can examine the status of arithmetic operation performed by ALU.

Here the concept of clock is important. We know that registers are made of flip-flops that only store data on the positive edge of clock (assuming the enable signal is 1, otherwise clock edges are in-effective). The small triangle on R_A and R_B in Figure 3 symbolizes input clock.

Let's see an example. Suppose through custom input (in past), 2 was stored in R_A and 3 in R_B . Now, enable of R_A is 1, enable of R_B is 0, selection bit (S) of ALU is 0 and S_{reg} is also 0. At the output of ALU, the sum 5 is present. As soon as the positive edge of clock comes, 5 got stored in R_A (as its enable pin was high) and appears at the output of R_A , the output of ALU becomes 8. But due to "internal gate delays", 8 appears a few nano-seconds after the positive edge had passed and now it cannot enter any register until next positive edge arrives. We can turn down the enable pins of registers to zero, to make clock edges in-effective so that the incoming 8 may not replace previously stored 5.

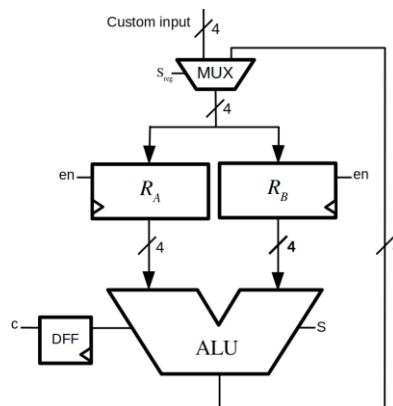


Figure 3: Modified Register-ALU Assembly

4.2 Testing

Let's test our design with a small "swapping" example. Usually when we swap the values of two variables, we use another temporary variable to hold data for sometime,

Initial state: $x = a, y = b$

```
{
    temp = x
    x = y
    y = temp
```

```
}
```

Final state: $x = b, y = a$

However, there is another smart algorithm to swap data of two registers without using any third register,

Initial state: $x = a, y = b$

```
{
    x = x + y
    y = x - y
    x = x - y
```

```
}
```

Final State: $x = b, y = a$

There are four control signals (S_{reg} , en_{R_A} , en_{R_B} and S) and the 4-bit custom input in our hands. First we shall load "a" to R_A and "b" to R_B , after which we will execute the algorithm. Here "a" and "b" are the 4-bit custom inputs.

Function	Control(S_{reg} en_{R_A} en_{R_B} S)	Result(at posedge clk)
Load a to R_A	110x	a stored in R_A
Load b to R_B	101x	b stored in R_B
$R_A = R_A + R_B$	0100	a+b stored in R_A
$R_B = R_A - R_B$	0011	a stored in R_B
$R_A = R_A - R_B$	0101	b stored in R_A

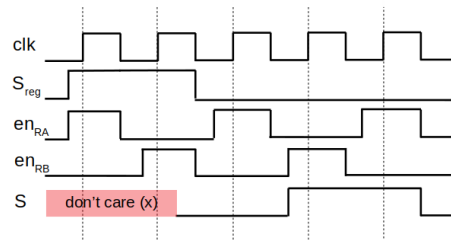


Figure 4: “Swapping” Control Signals Timing diagram

4.3 Output Register

A small addition to our design, the output register R_O . We shall use this register to store the final result after all the processing for the user to refer. The register is directly connected with R_A .

4.4 Register De-multiplexing

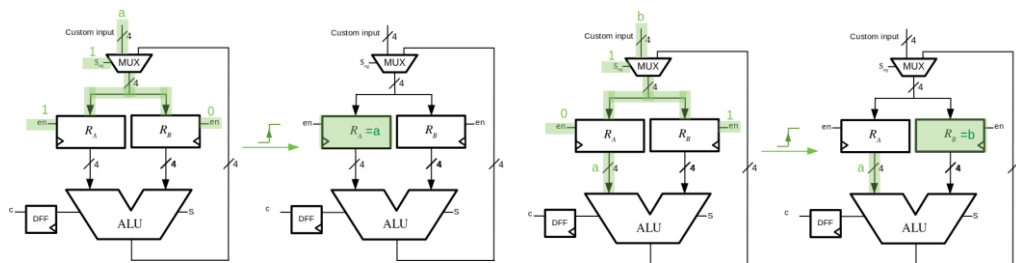
There are now three signals associated with the registers, en_A , en_B and en_O . In order to reduce these signals we will use a 2-to-4 decoder (two inputs and four outputs) with a truth table shown below,

Input (D_0, D_1)	Output (O_0, O_1, O_2, O_3)
00	1000
01	0100
10	0010
11	0001

Now the outputs of decoder are connected such that O_0 to en_A , O_1 is to en_B and O_2 to en_O ,

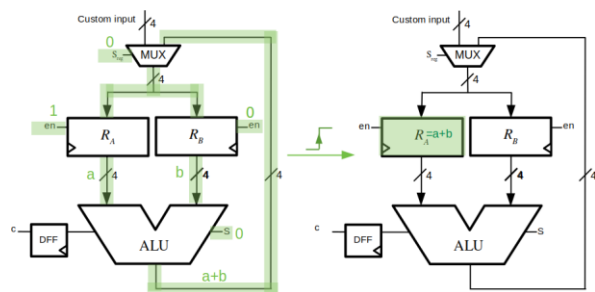
Input (D_0, D_1)	Output
00	R_A input enabled
01	R_B input enabled
10	R_O input enabled
11	no operation

Keep in mind, from now onward we will call the value of D_0D_1 , “address” of the corresponding register.

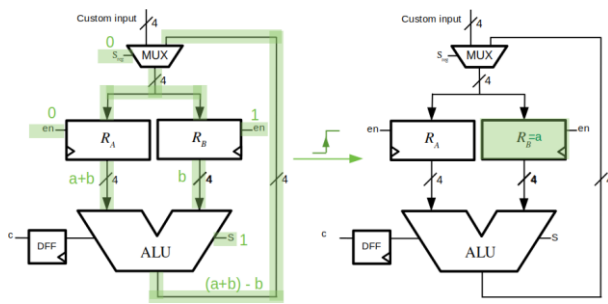


Step 1

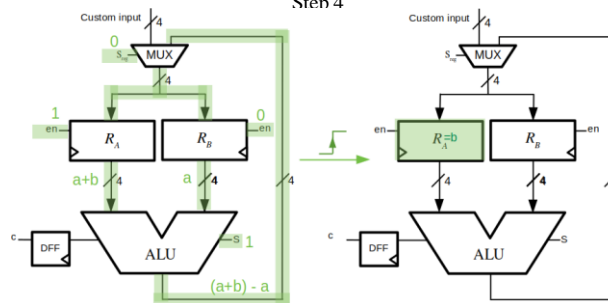
Step 2



Step 3



Step 4



Step 5

5 Control Logic

We cannot apply logic values on the control signals manually, all the time. In order to avoid this inconvenience, we can store the values of control signals

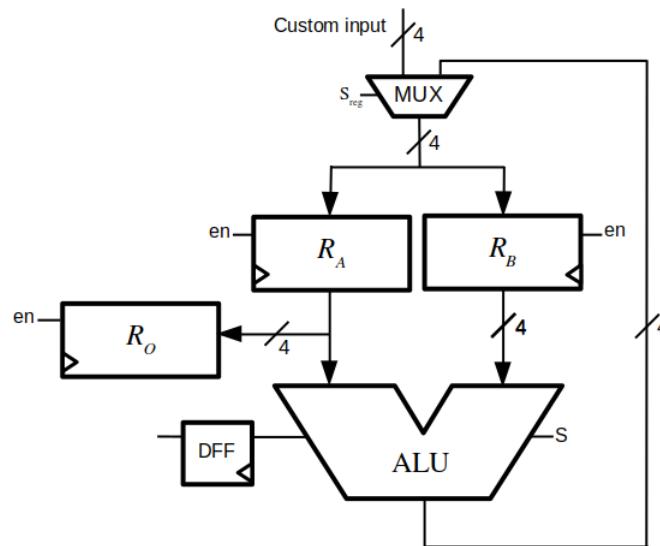


Figure 5: Output Register

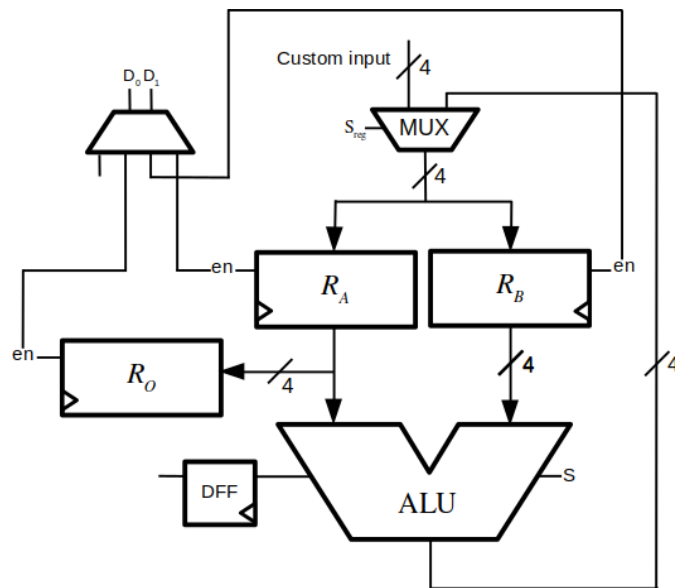


Figure 6: Design with Decoder Incorporated

in a non-volatile memory (ROM or EEPROM) and connect its output lines to our design. Let's call the values of the control signal stored in the memory as Instructions.

5.1 Instruction Memory

Memory is an array (having rows and columns) of cells, which can store binary data (1 or 0). For our design we will talk about those memories having 8 cells (or bits) in each row. Each 8-bit wide row can be accessed by its address starting from zero to (length of array - 1). We will store the set of control signals one in each row. For example in "swapping" test, the set of control signals was 110x, 101x, 0100, 0011, 0101 } We can store 110x in row(0), 101x in row(1) and so on. For an 8-bit row, each set of signals (also called "instruction") will take only four bits, the rest of bits, we don't care.

5.2 Program Counter

In order to set the signals (or instruction) on the control lines of our design before the positive edge of clock arrives, we will be using a 4-bit counter connected to address line of instruction memory, such that initially the value of counter is zero with zeroth row activated. As soon as the positive edge of clock comes, zeroth instruction is executed by our design as well as the counter gets incremented. Now the value of counter is one and row 1 of instruction memory is activated. The incoming positive edge will execute this instruction and the process goes on.

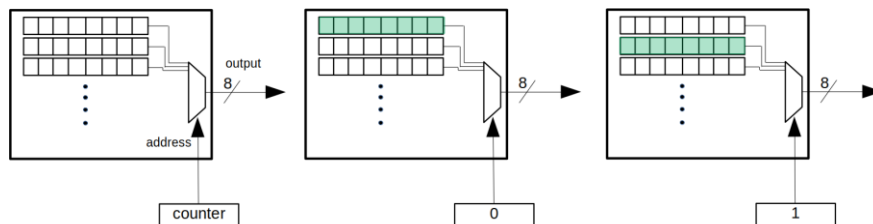


Figure 7: Memory and Program Counter

The program counter must have a 4-bit input and a load control such that when some custom input is applied to 4-bit input of counter and load control is kept high on the arrival of positive edge, the counter stores that custom

input into it and start counting from that specific value on the next edge of clock. We wish to design a combinational logic around this feature of counter.

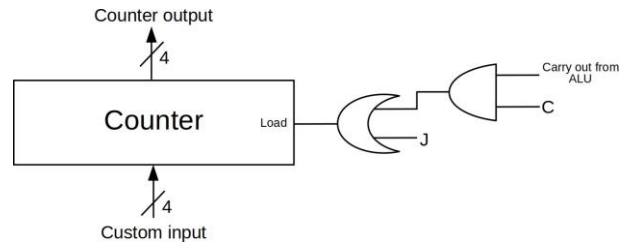


Figure 8: Jump Logic

$$Load = J \text{ OR } (C \text{ AND "Carry out"})$$

Here “J” and “C” are control signals in our hand such that when we apply some custom input to counter and raise J to 1, the counter starts counting from custom input. Similarly, when the carry out from ALU is 1 and we raise C to 1, the counter starts counting from custom input. This phenomenon is known as **“JUMP”** or **“BRANCH”** in computing language.

5.3 Instruction Encoding

Since the rows of instruction memory are 8-bits wide, we wish to adjust our all control signals to these 8-bits. Moreover, our custom input is also supposed to be a part of this instruction. One scheme to adjust all these signals is,

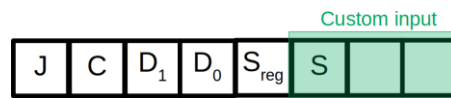


Figure 9: 8-bit wide instruction

1. **Jump (J):** The seventh bit or MSB of the instruction is fixed for Jump. Whenever we set it to 1, the custom input will be loaded in the program counter.

2. **Carry jump (C):** The sixth bit is fixed for “jump if ALU operation produces a carry”. Whenever we set it to 1, the custom input will be loaded in the program counter.
3. **Register Address (D_1D_0):** The fifth and fourth bit of instruction is fixed for register address. The values 00, 01, 10 and 11 corresponds to R_A , R_B , R_O and no register, respectively.
4. **ALU or Custom input (S_{reg}):** The third bit is reserved for multi-plexing ALU or custom input. When it is 0, ALU’s output is directed to registers’ input otherwise, the custom input.
5. **Custom input (immediate):** The last three bits, zeroth, first and second are reserved for custom input. Due to adjustment problem, we have to compromise custom input length (to 3-bits only). The fourth bit is hard wired to zero (ground), which means, we can only load values ranging from 0 to 7 (either in program counter or registers).
6. **ADD or SUB (S):** The second bit of the instruction has dual function. Apart from being part of custom input, it is also connected to ALU selection bit S. This decision is taken based on the observation that when ALU is performing either addition or subtraction, there is no interference of custom input. Similarly, when we are loading some custom input to either registers or counter, we don’t care about ALU processing.

6 Programming

There are 9 functions we can perform with this machine. The functions and the set of control signals for them are,

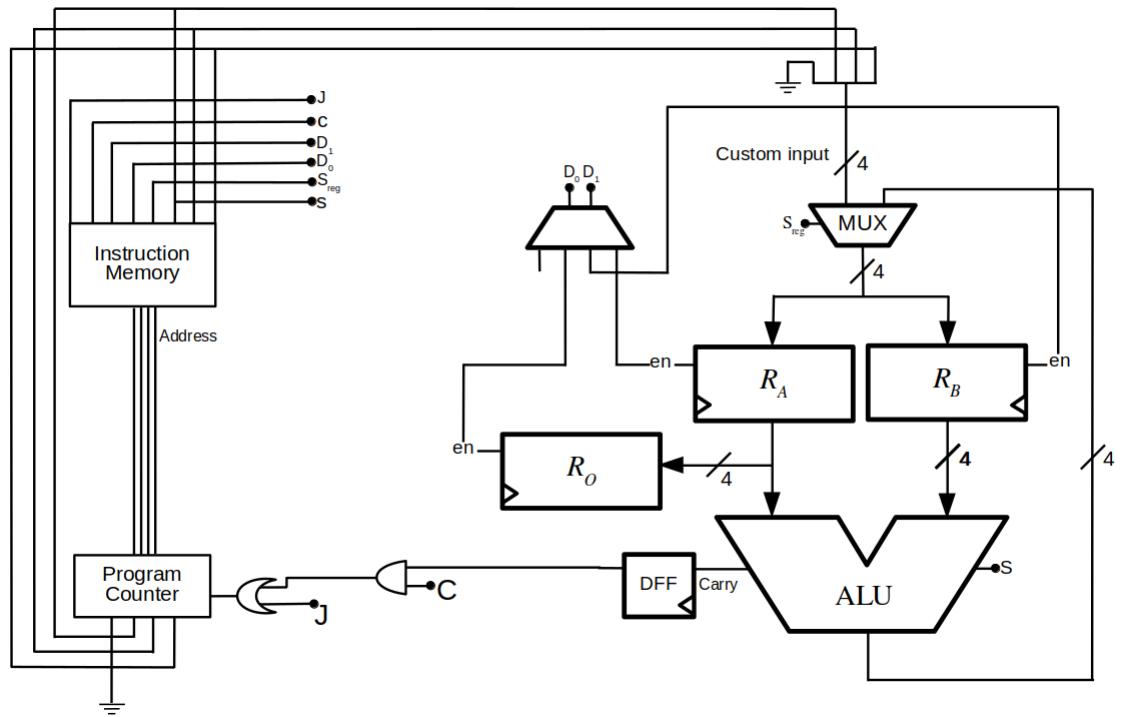


Figure 10: 4-bit Number Crunching Machine

Function	J	C	D1	D0	Sreg	S		
						Custom Input		
$R_A = R_A + R_B$	0	0	0	0	0	0	0	0
$R_B = R_A + R_B$	0	0	0	1	0	0	0	0
$R_A = R_A - R_B$	0	0	0	0	0	1	0	0
$R_B = R_A - R_B$	0	0	0	1	0	1	0	0
$R_O = R_A$	0	0	1	0	0	0	0	0
$R_A = imm$	0	0	0	0	1	imm[2]	imm[1]	imm[0]
$R_B = imm$	0	0	0	1	1	imm[2]	imm[1]	imm[0]
Jump to imm if carry out	0	1	1	1	0	imm[2]	imm[1]	imm[0]
Jump to imm	1	0	1	1	0	imm[2]	imm[1]	imm[0]

6.1 Fibonacci Numbers

Below is a Program that will produce Fibonacci sequence in the output register.

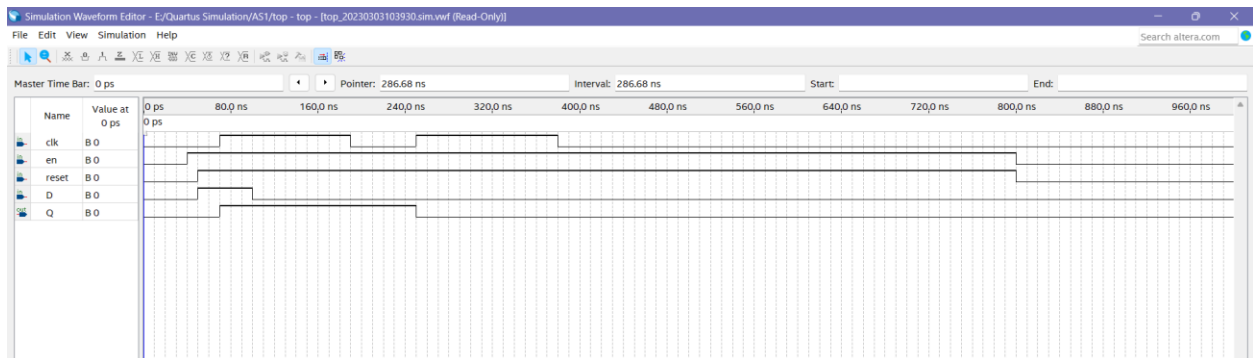
0	$R_A = 0$	00001000	load zero to A
1	$R_B = 1$	00011001	load 1 to B
2	$R_O = R_A$	00100000	push A to output
3	$R_B = R_A + R_B$	00010000	Add A to B
4	jump if carry	01110000	If A+B produce carry jump to start
5	$R_A = R_A + R_B$	00000000	Swap A and B
6	$R_B = R_A - R_B$	00010100	Swap A and B
7	$R_A = R_A - R_B$	00000100	Swap A and B
8	jump to 2	10110010	jump to 3rd instruction

1. Module 1: Single bit D flipflop

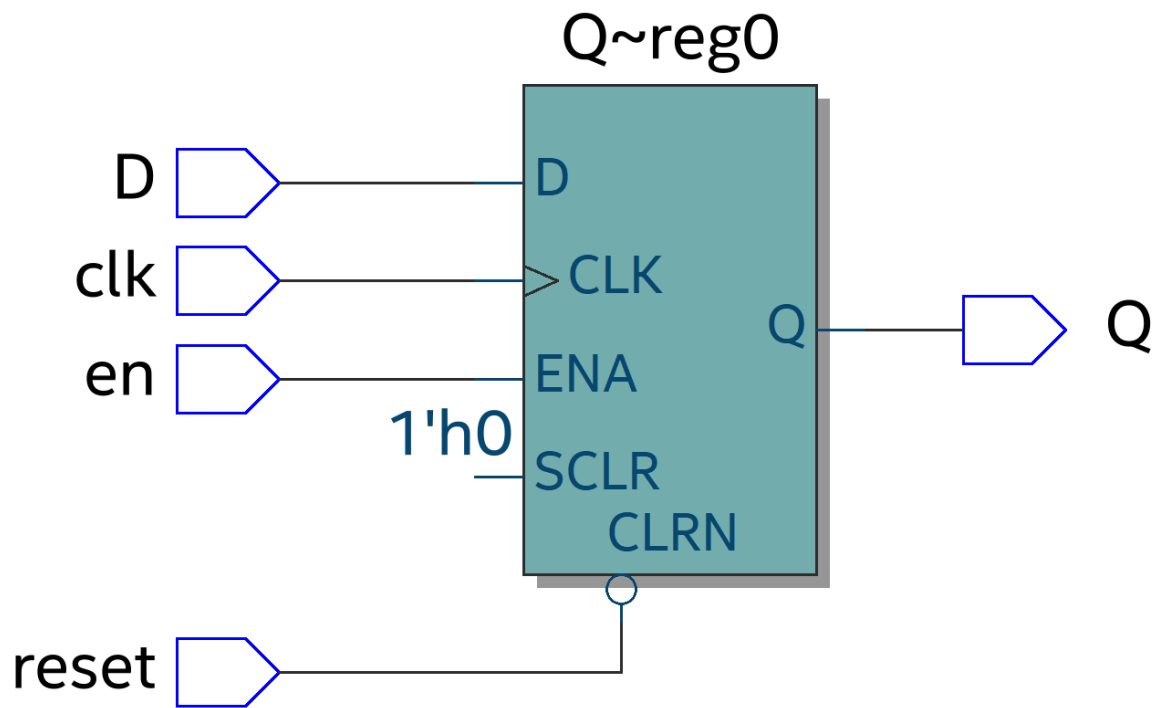
CODE

```
top.v | decoder_2_4.v | Compilation Report - top | ROM3.v | DFF.v | dff_1bit.v x
1 module dff_1bit(D,clk,en,reset,Q);
2
3 input D, reset,en, clk;
4 output Q;
5
6 reg Q;
7
8 //setting the initial value of output to be 0.
9 //initial
10 //begin
11 //Q=1'b0;
12
13 //end
14
15 always @(posedge clk or negedge reset) //Asynchronous reset setting
16 begin
17     if(reset==1'b0)
18         Q<=1'b0;
19     else
20         if(en)
21             Q<=D;
22     end
23 endmodule
24
25
```

SIMULATION

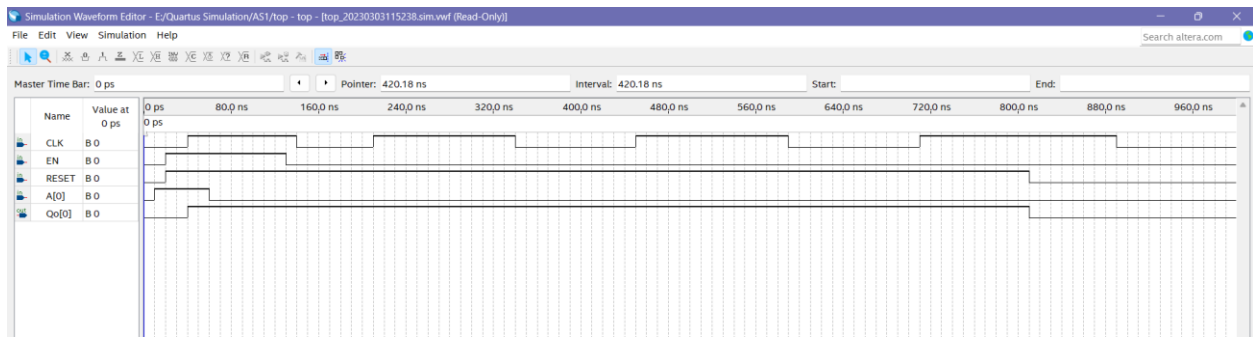
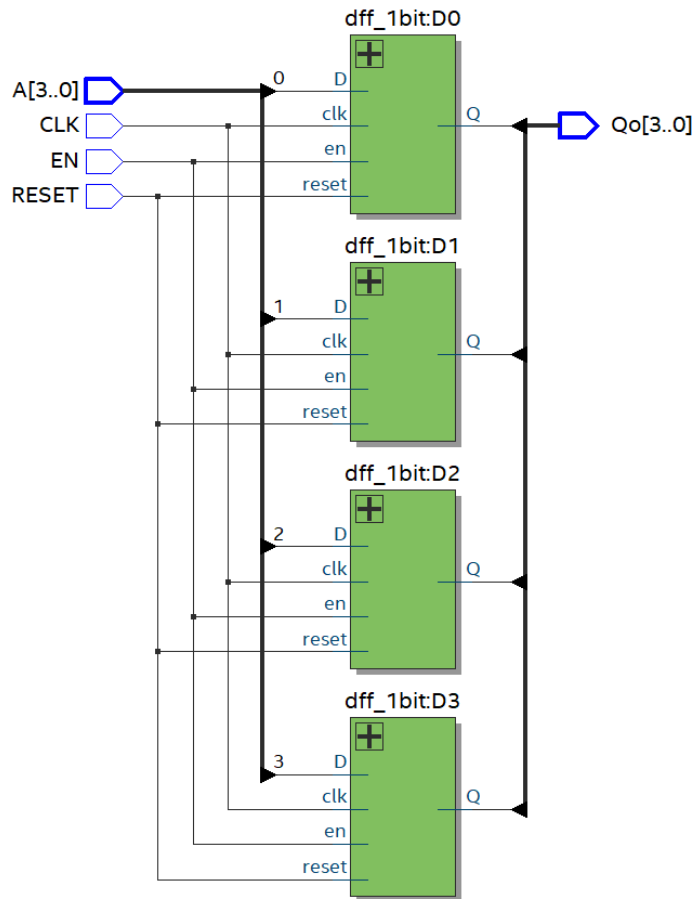


RTL VIEWER



2. Module 2: 4-bit register model

```
1 module D_4_reg(A,CLK,EN,RESET,Qo);  
2  
3   input [3:0] A;  
4   input CLK,EN,RESET;  
5   output reg [3:0] Qo;  
6  
7   wire [3:0] d;  
8  
9   dff_1bit D0(A[0],CLK,EN,RESET,d[0]);  
10  dff_1bit D1(A[1],CLK,EN,RESET,d[1]);  
11  dff_1bit D2(A[2],CLK,EN,RESET,d[2]);  
12  dff_1bit D3(A[3],CLK,EN,RESET,d[3]);  
13  
14  always @(posedge CLK or negedge RESET)  
15  begin  
16  
17    Qo[0]<=d[0];  
18    Qo[1]<=d[1];  
19    Qo[2]<=d[2];  
20    Qo[3]<=d[3];  
21  
22  end  
23 endmodule  
24  
25
```

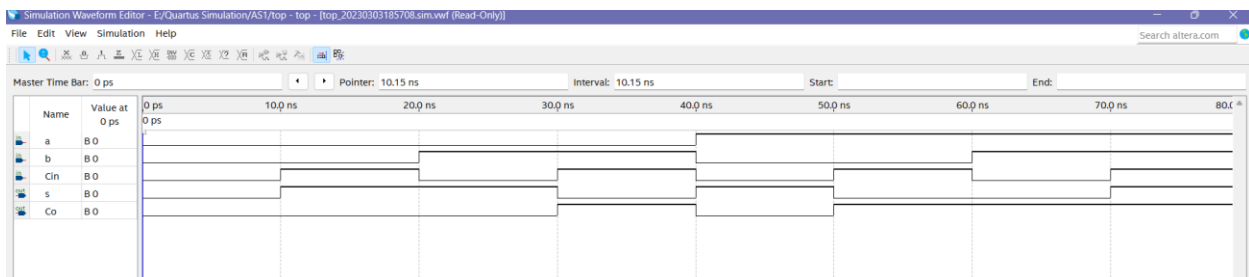


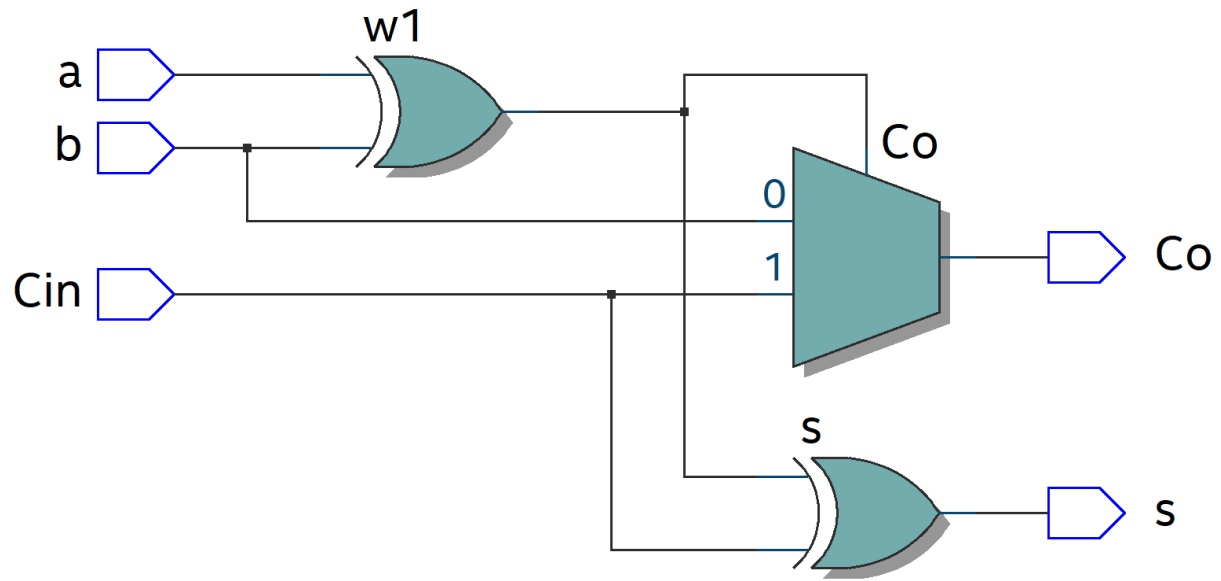
3. Module 3: ALU DESIGN

CODE:

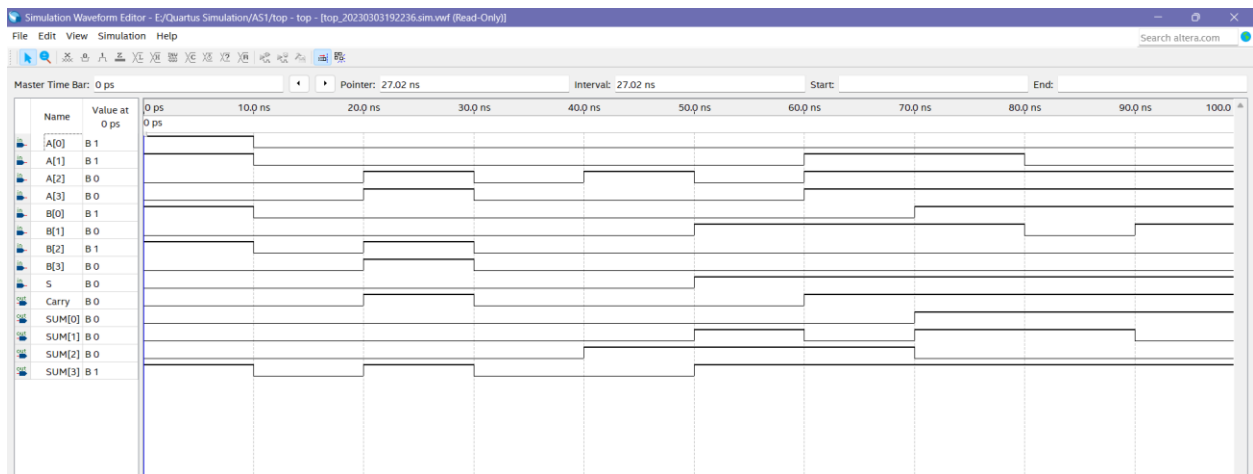
```
1  module ALU_ad_sub(A,B,S,SUM,Carry);
2
3  input [3:0] A;
4  input [3:0] B;
5  input S;
6
7  output [3:0] SUM;
8  output Carry;
9
10 wire B0,B1,B2,B3,C0,C1,C2;
11
12 assign B0=B[0]^S;
13 assign B1=B[1]^S;
14 assign B2=B[2]^S;
15 assign B3=B[3]^S;
16
17 fulladder F0(S,A[0],B0,SUM[0],C0);
18 fulladder F2(C0,A[1],B1,SUM[1],C1);
19 fulladder F3(C1,A[2],B2,SUM[2],C2);
20 fulladder F4(C2,A[3],B3,SUM[3],Carry);
21
22 endmodule
23
```

1 bit testing

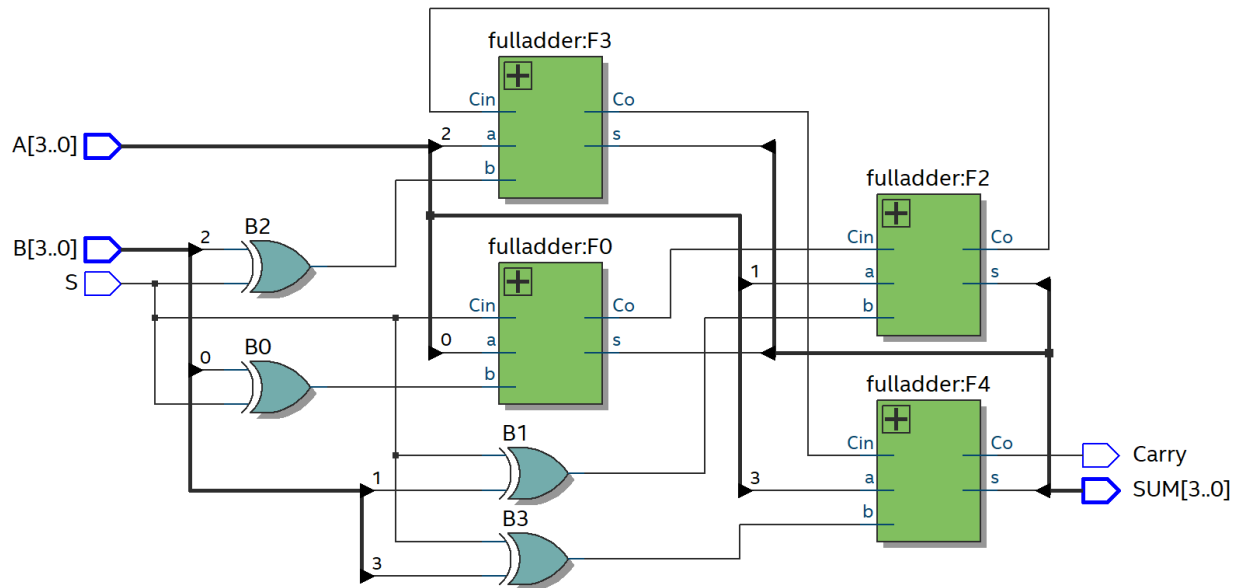




Testing all 4 bits



RTL VIEWER

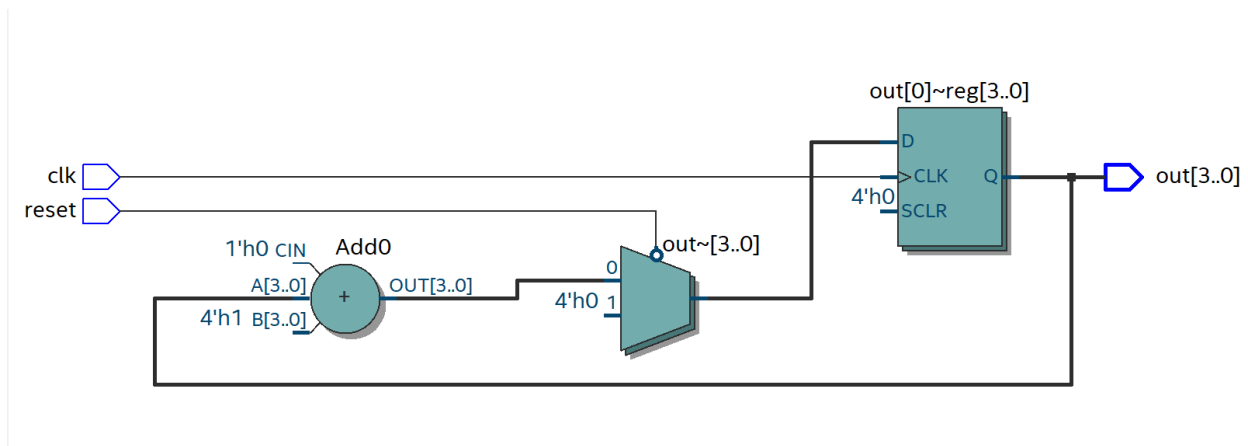


4. Module 4: custom input in counter 4 bit

CODE:

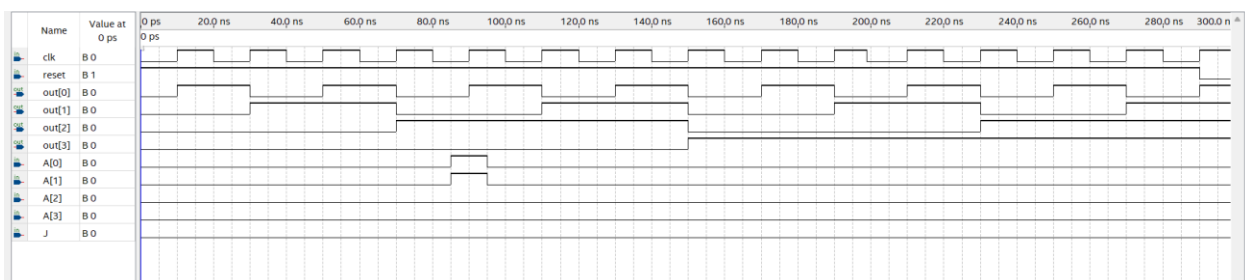
```
1
2
3 module counter(A,clk,J, reset, out);
4
5 input [3:0] A;
6 input clk,J;
7 input reset;
8 output reg [3:0] out;
9 reg [3:0] x;
10 always @(posedge clk)
11 begin
12
13 if(reset==1'b0)
14 out<=0;
15 else if(J==1'b1)
16 begin
17 out<=A;
18
19 end
20 else
21 out<=out+1'b1;
22 end
23 endmodule
24
```

RTL Viewer:

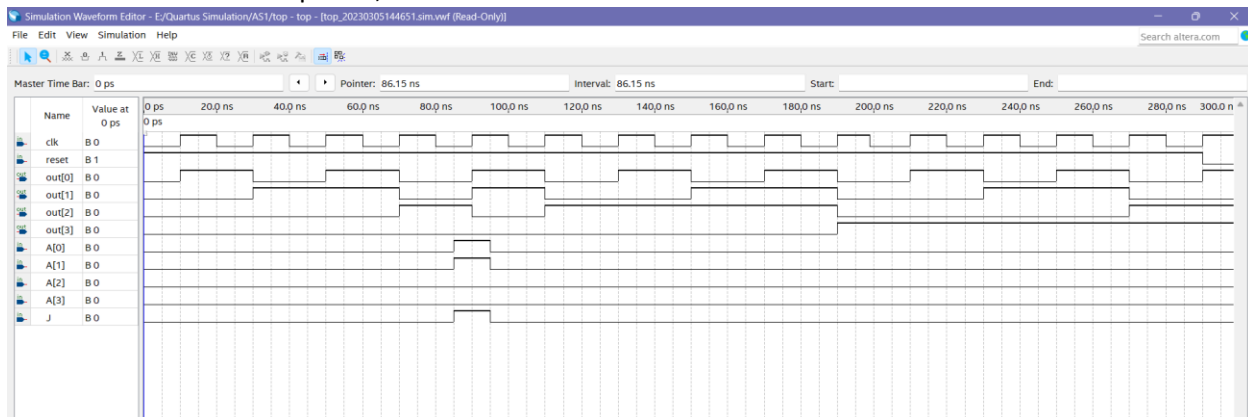


Simulation:

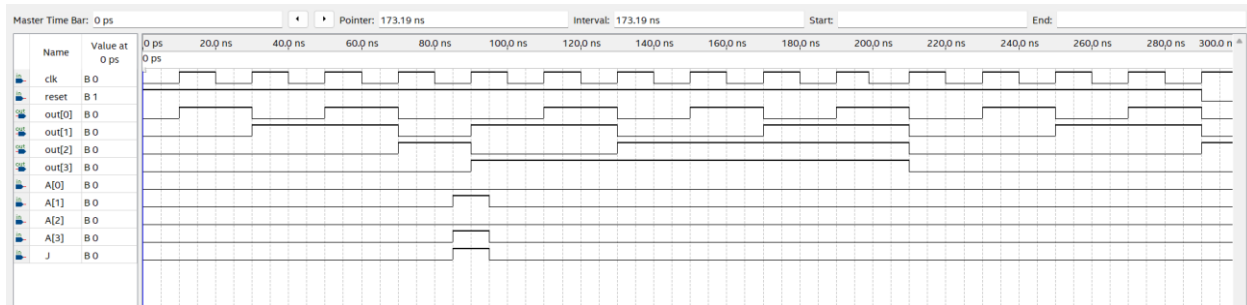
When J=0, Counter counts till 15.(output=out[3:0])



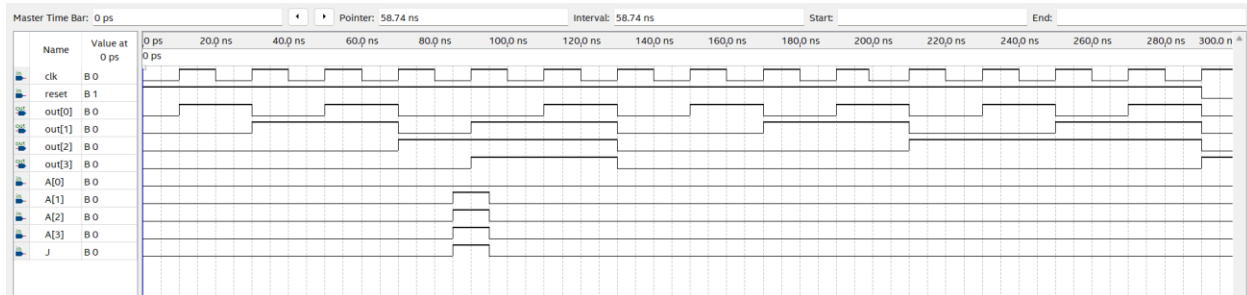
When J=1 and custom input =3, the next count starts from 3.



When J=1 and custom input =10, the next count starts from 10.



When J=1 and custom input =14, the next count starts from 14.



5. Module5:2 by 4-decoder

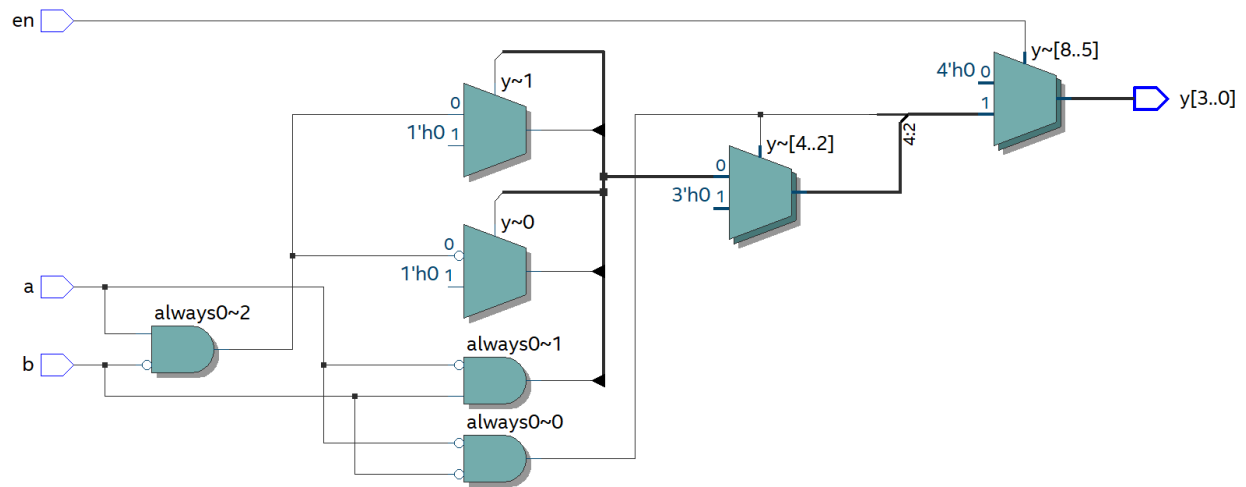
CODE:

```

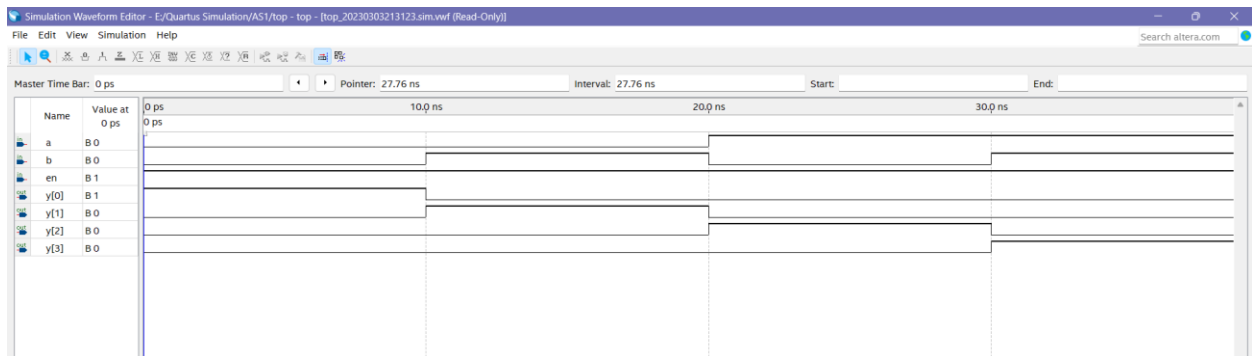
1  module decoder_2_4(a,b,en,y);
2
3  input a,b,en;
4  output reg [3:0] y;
5
6  always @(*)
7  begin
8
9  if(en==1'b1)
10 begin
11 if(a==1'b0 & b==1'b0)
12 y=4'b0001;
13 else if(a==1'b0 & b==1'b1)
14 y=4'b0010;
15 else if(a==1'b1 & b==1'b0)
16 y=4'b0100;
17 else if(a==1'b1 & b==1'b1)
18 y=4'b1000;
19 else
20 y=4'bxxxx;
21 end
22
23 else
24 y=4'b0000;
25
26 end
27 endmodule
28

```

RTL:



SIMULATION:

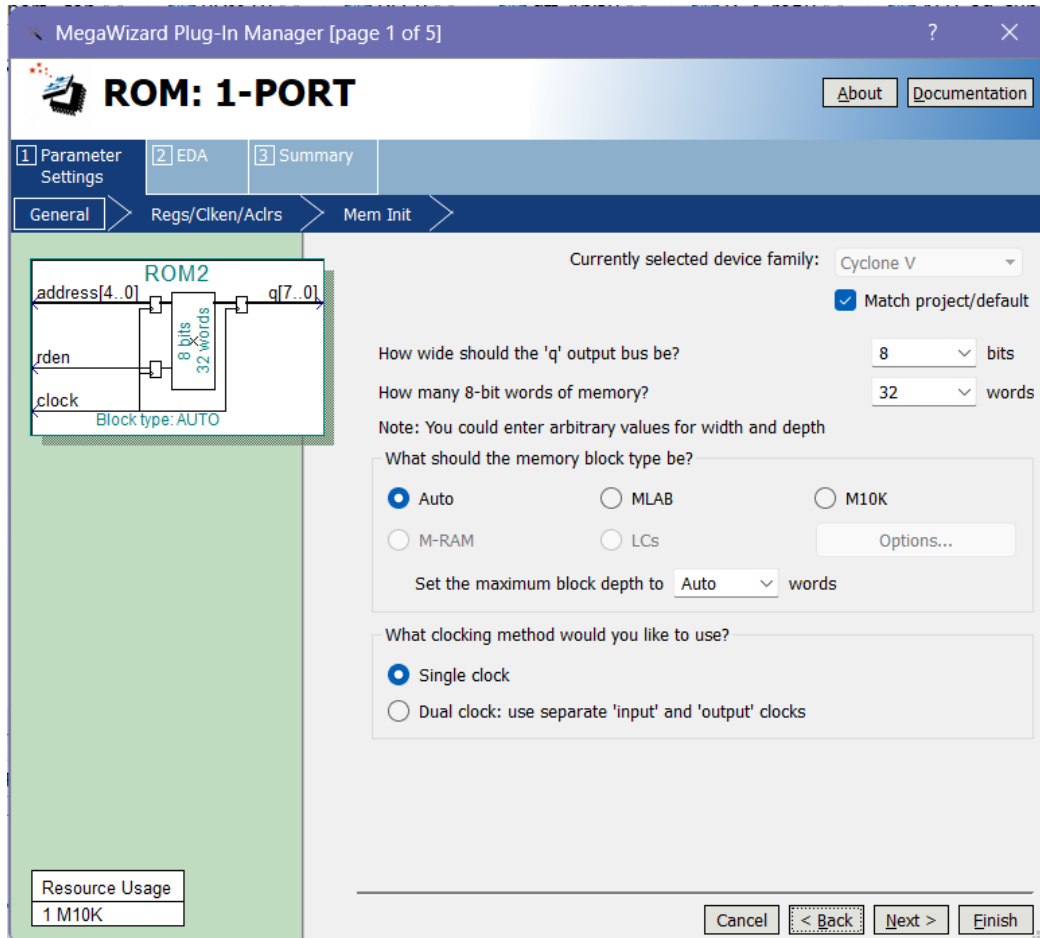


6. Module6: MEMORY – ROM3

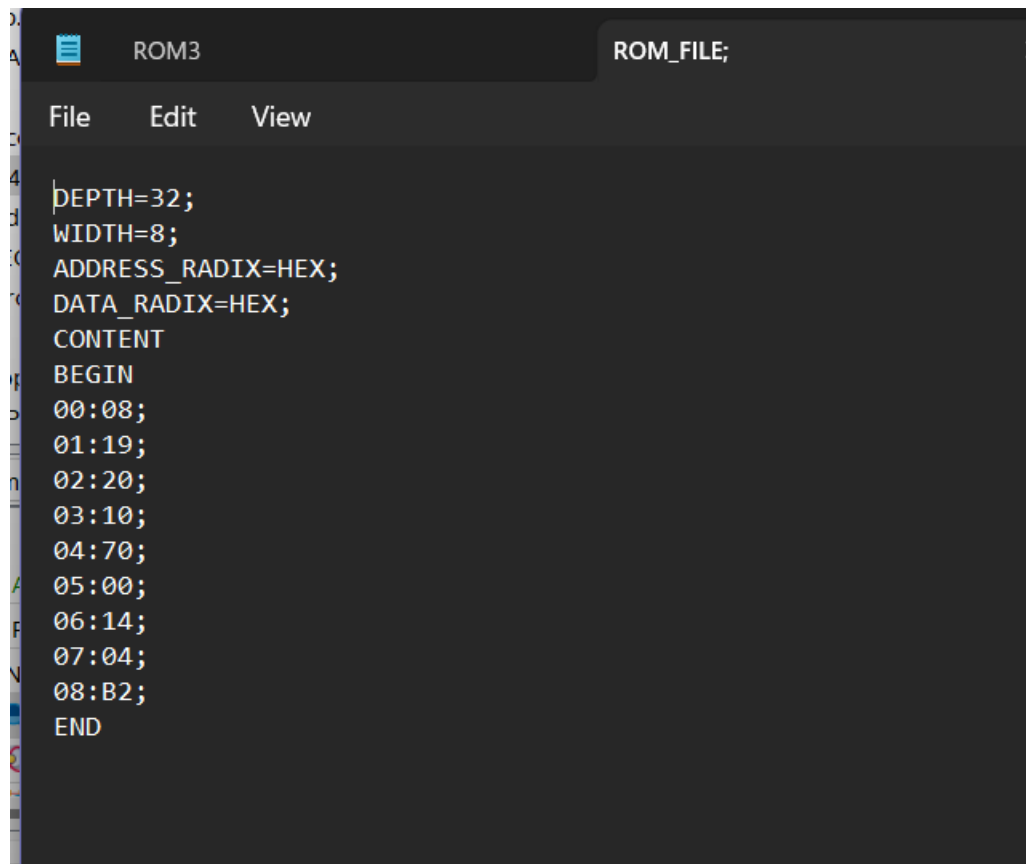
- The ROM module was created using the Quartus IP catalogue as follows.

Step1: Open Megawizard from Tools -> IP catalogue -> On Chip Memory -> single Port Rom.

- We chose single port ROM because we needed a single port output of 8 addresses with 4 bit input.



Step 2: Add the address and the data to be stored in the ROM in a text file and upload it to the ROM.



```
DEPTH=32;  
WIDTH=8;  
ADDRESS_RADIX=HEX;  
DATA_RADIX=HEX;  
CONTENT  
BEGIN  
00:08;  
01:19;  
02:20;  
03:10;  
04:70;  
05:00;  
06:14;  
07:04;  
08:B2;  
END
```

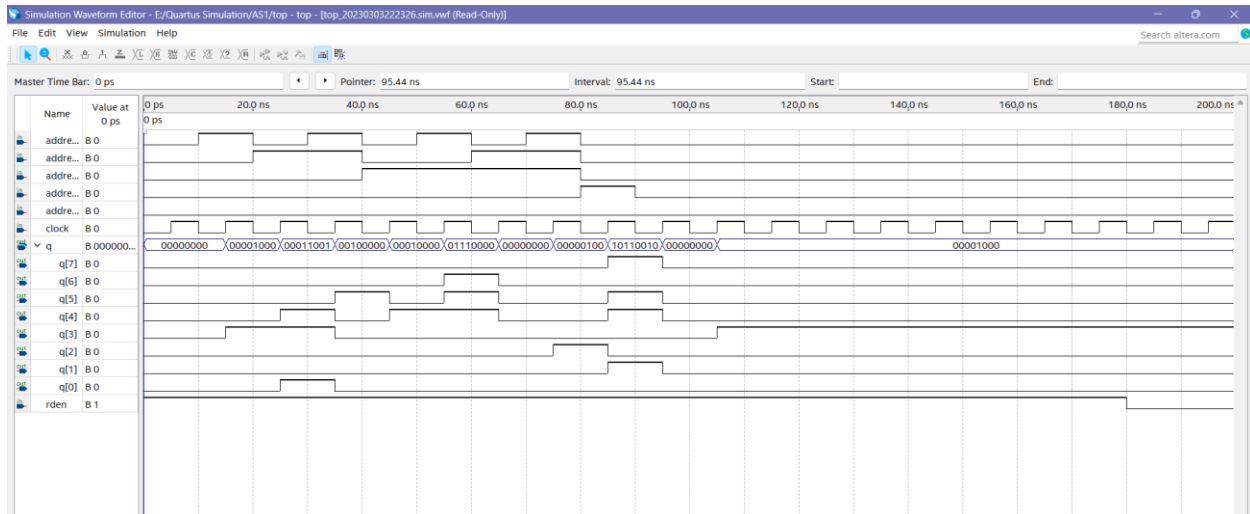
Final CODE:

```
69
70 // synopsys translate_off
71 timescale 1 ps / 1 ps
72 // synopsys translate_on
73 module ROM_MEMEORY (
74     address,
75     clock,
76     rden,
77     q);
78
79     input [4:0] address;
80     input clock;
81     input rden;
82     output [7:0] q;
83
84     `ifndef ALTERA_RESERVED_QIS
85     // synopsys translate_off
86     `endif
87     tri1 clock;
88     tri1 rden;
89     `ifndef ALTERA_RESERVED_QIS
90     // synopsys translate_on
91     `endif
92
93     wire [7:0] sub_wire0;
94     wire [7:0] q = sub_wire0[7:0];
95
96     altsyncram altsyncram_component (
97         .address_a (address),
98         .clock0 (clock),
99         .rden_a (rden),
100         .q_a (sub_wire0),
101         .aclr0 (1'b0),
102         .aclr1 (1'b0),
103         .address_b (1'b1),
104         .addressstall_a (1'b0),
105         .addressstall_b (1'b0),
106         .byteena_a (1'b1),
107         .byteena_b (1'b1),
108         .aclr0 (1'b0),
109         .aclr1 (1'b0),
110         .address_b (1'b1),
111         .addressstall_a (1'b0),
112         .addressstall_b (1'b0),
113         .byteena_a (1'b1),
114         .byteena_b (1'b1),
115         .clock1 (1'b1),
116         .clocken0 (1'b1),
117         .clocken1 (1'b1),
118         .clocken2 (1'b1),
119         .clocken3 (1'b1),
120         .data_a ({8{1'b1}}),
121         .data_b (1'b1),
122         .eccstatus (),
123         .q_b (),
124         .rden_b (1'b1),
125         .wren_a (1'b0),
126         .wren_b (1'b0));
127
128     defparam
129         altsyncram_component.address_aclr_a = "NONE",
130         altsyncram_component.clock_enable_input_a = "BYPASS",
131         altsyncram_component.clock_enable_output_a = "BYPASS",
132         altsyncram_component.init_file = "ROM_FILE.txt",
133         altsyncram_component.intended_device_family = "Cyclone V",
134         altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
135         altsyncram_component.lpm_type = "altsyncram",
136         altsyncram_component.numwords_a = 32,
137         altsyncram_component.operation_mode = "ROM",
138         altsyncram_component.outdata_aclr_a = "NONE",
139         altsyncram_component.outdata_reg_a = "CLOCK0",
140         altsyncram_component.widthad_a = 5,
141         altsyncram_component.width_a = 8,
142         altsyncram_component.width_byteena_a = 1;
143
144 endmodule
```

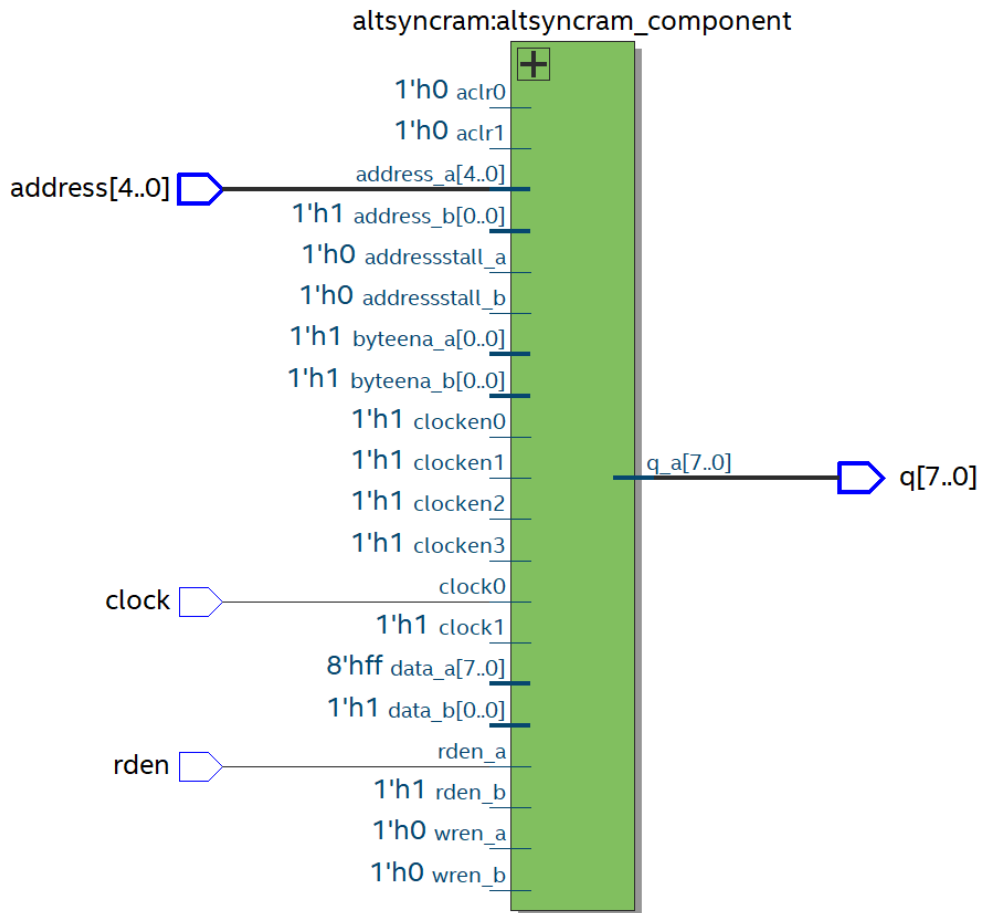
Final Simulation.

- As expected the output from the ROM was given as the stored value after every clock cycle.

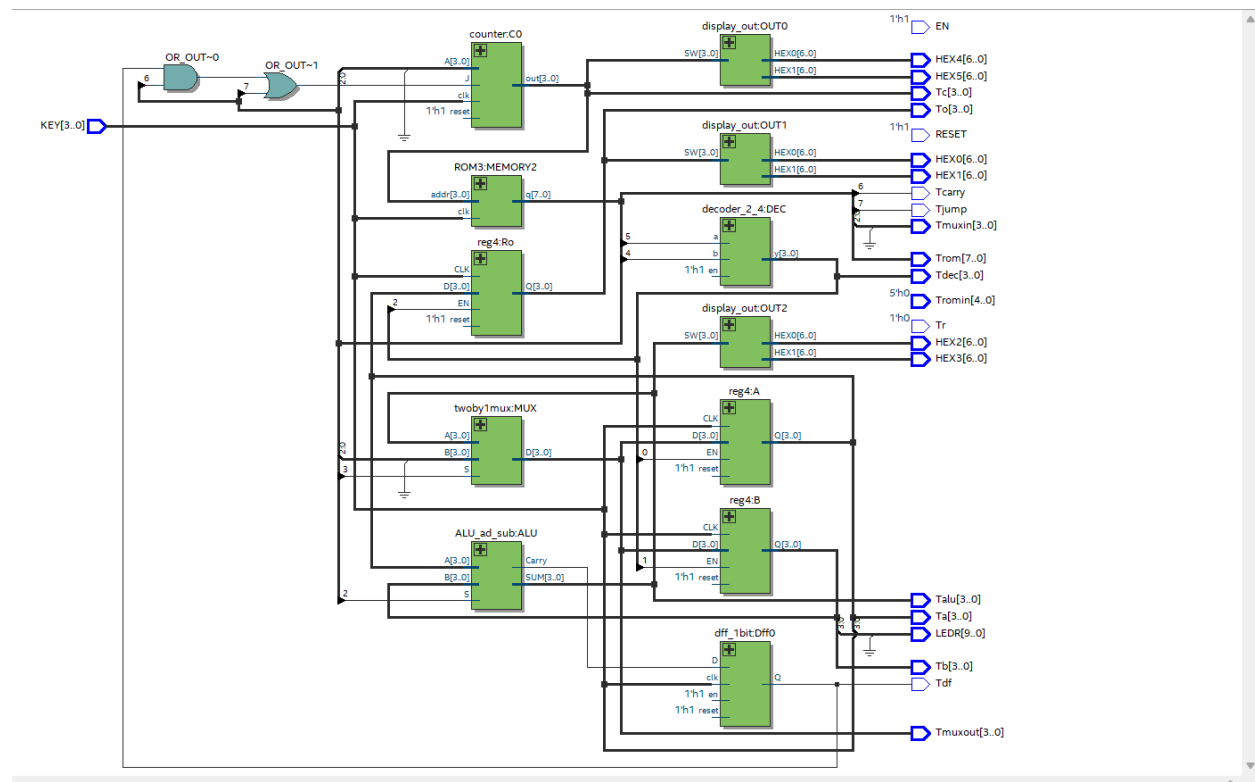
0	$R_A = 0$	00001000	load zero to A
1	$R_B = 1$	00011001	load 1 to B
2	$R_O = R_A$	00100000	push A to output
3	$R_B = R_A + R_B$	00010000	Add A to B
4	jump if carry	01110000	If A+B produce carry jump to start
5	$R_A = R_A + R_B$	00000000	Swap A and B
6	$R_B = R_A - R_B$	00010100	Swap A and B
7	$R_A = R_A - R_B$	00000100	Swap A and B
8	jump to 2	10110010	jump to 3rd instruction



RTL VIEWER



FINAL RTL DESIGN



TESTING

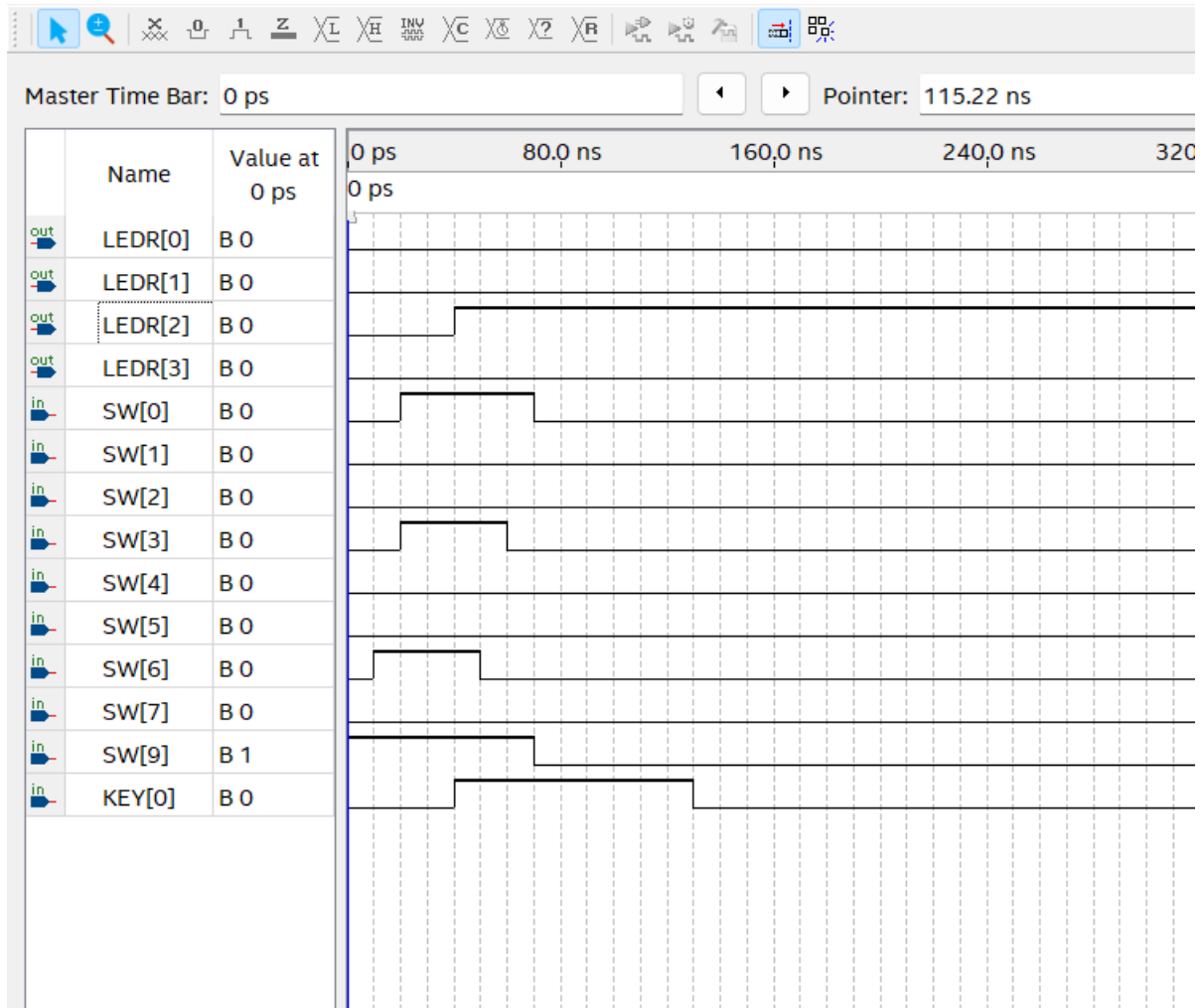
A. Testing MUX module with register A.

KEY[0] assigned as clock.

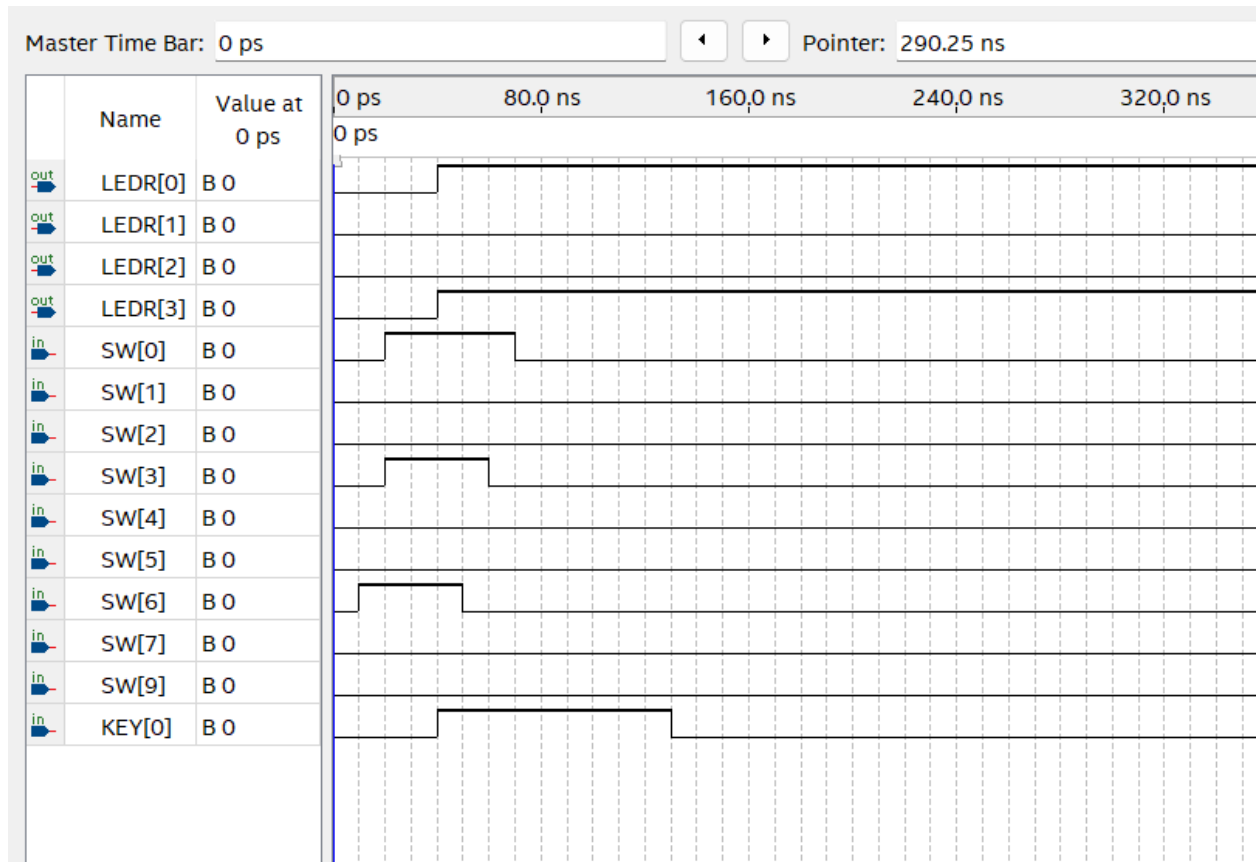
SW[9] assigned as MUX select line.

1. When SW[9]=1 The second input to Mux is selected and passed to the reg A.

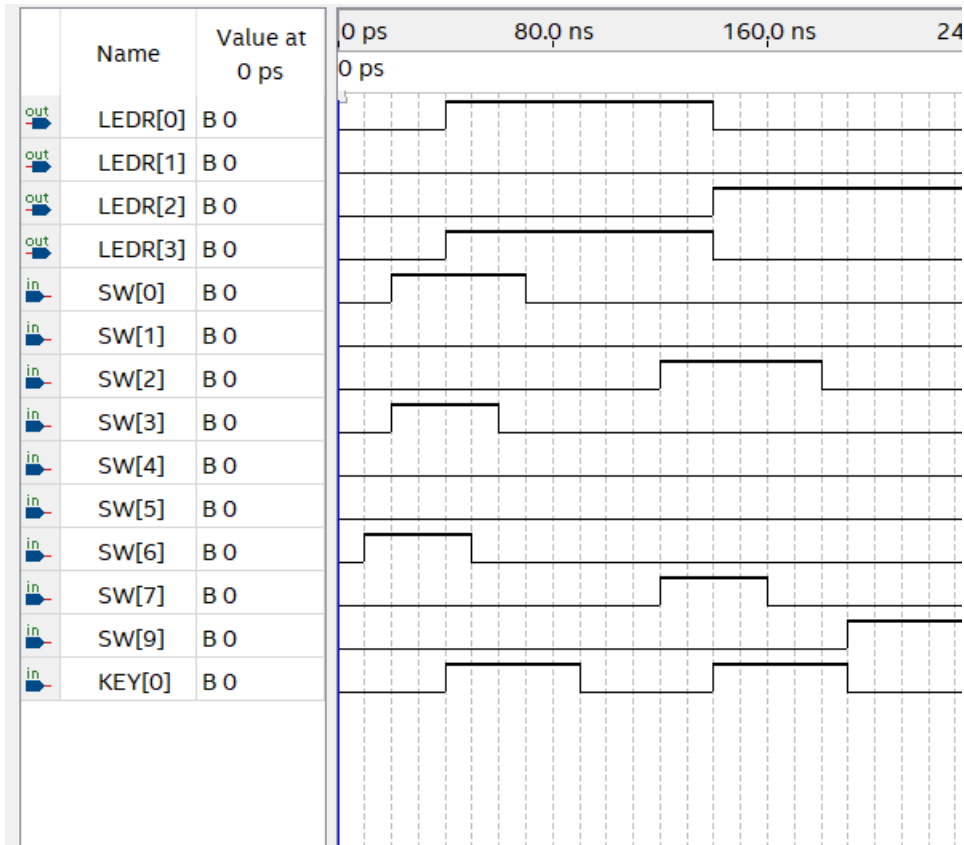
File Edit View Simulation Help



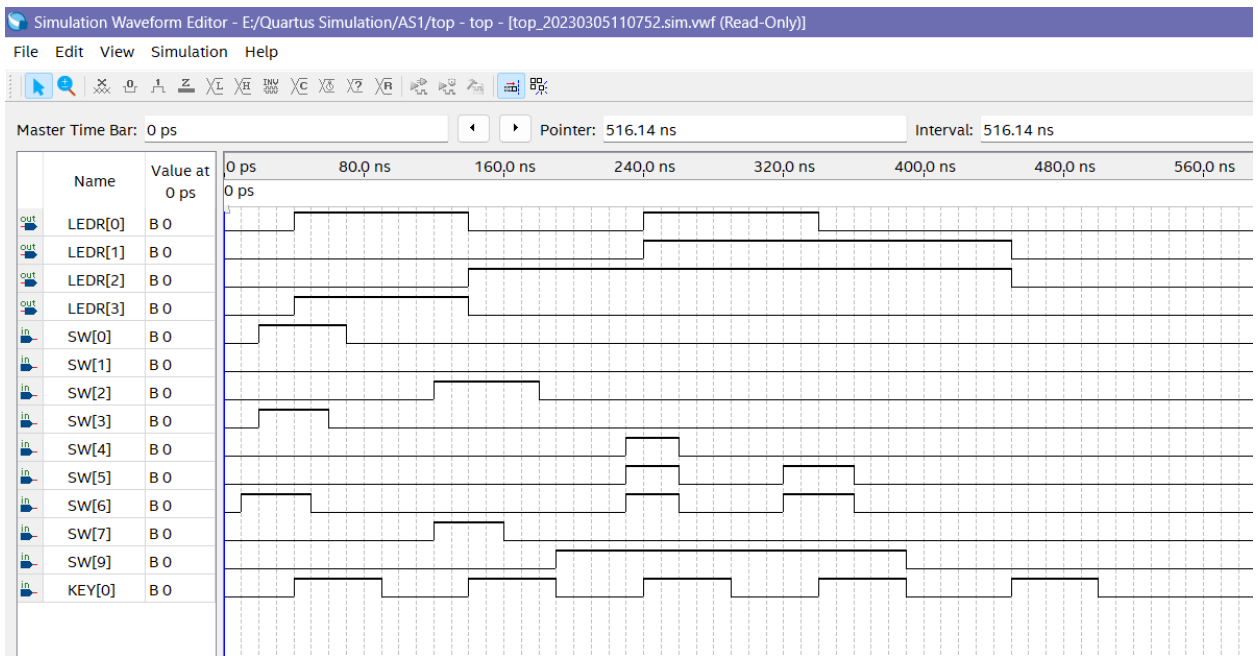
- When SW[9]=0, the first input to MUX is selected and passed to reg A which is value 9.



- When SW[9] = 0, we change the value of MUX input from 9 to 4 which is reflected in the output of reg A as expected in the following clock edge.

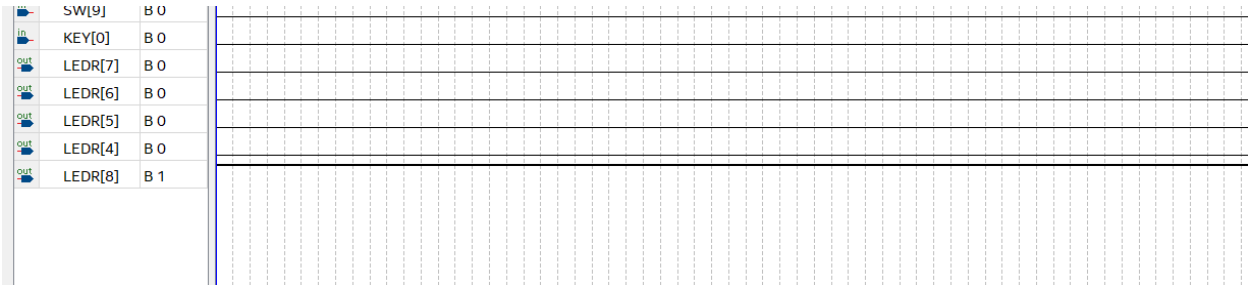


4. When SW[9]=1, the MUX selects the second input as expected which is 7 next 6 as the reg A output.



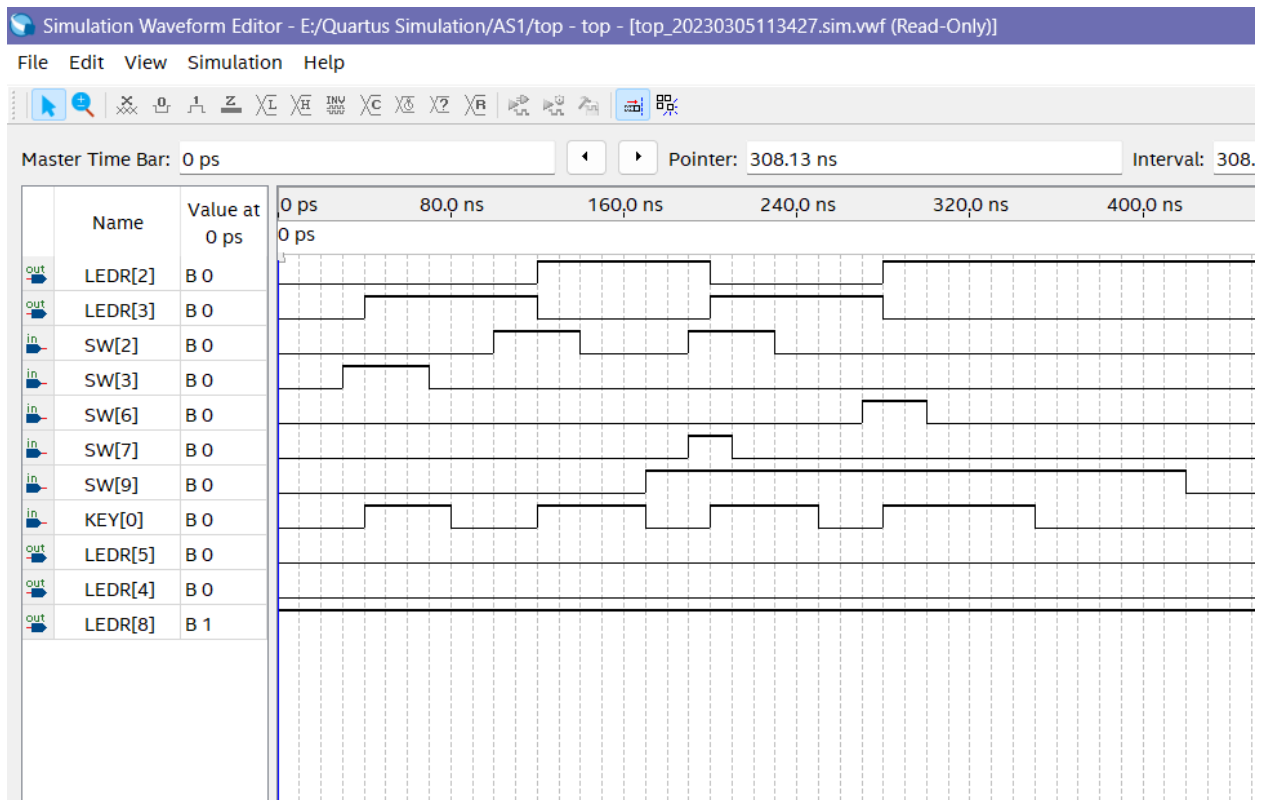
B. Testing our decoder

1. When the input were given as 00 the first out was received at LEDR[8]

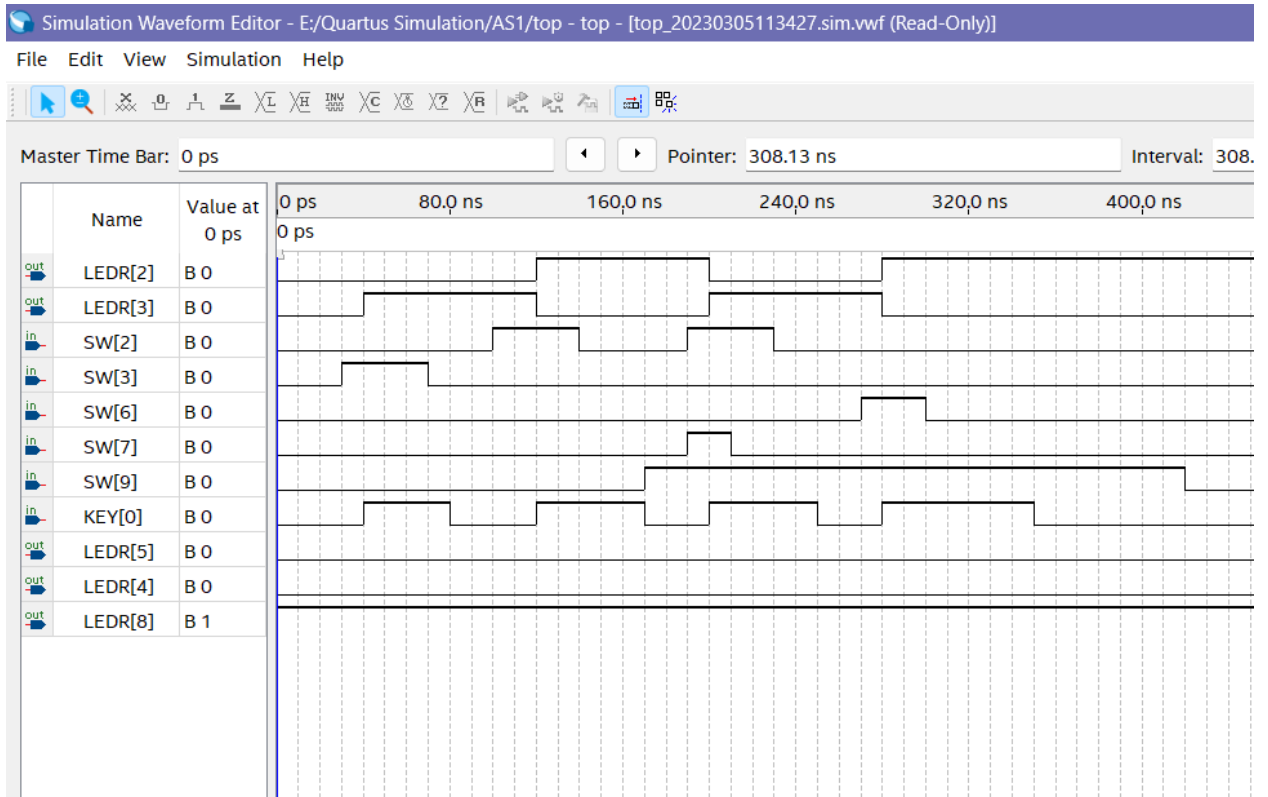


2. Thus, reg A is successfully decoded with the value of the MUX.

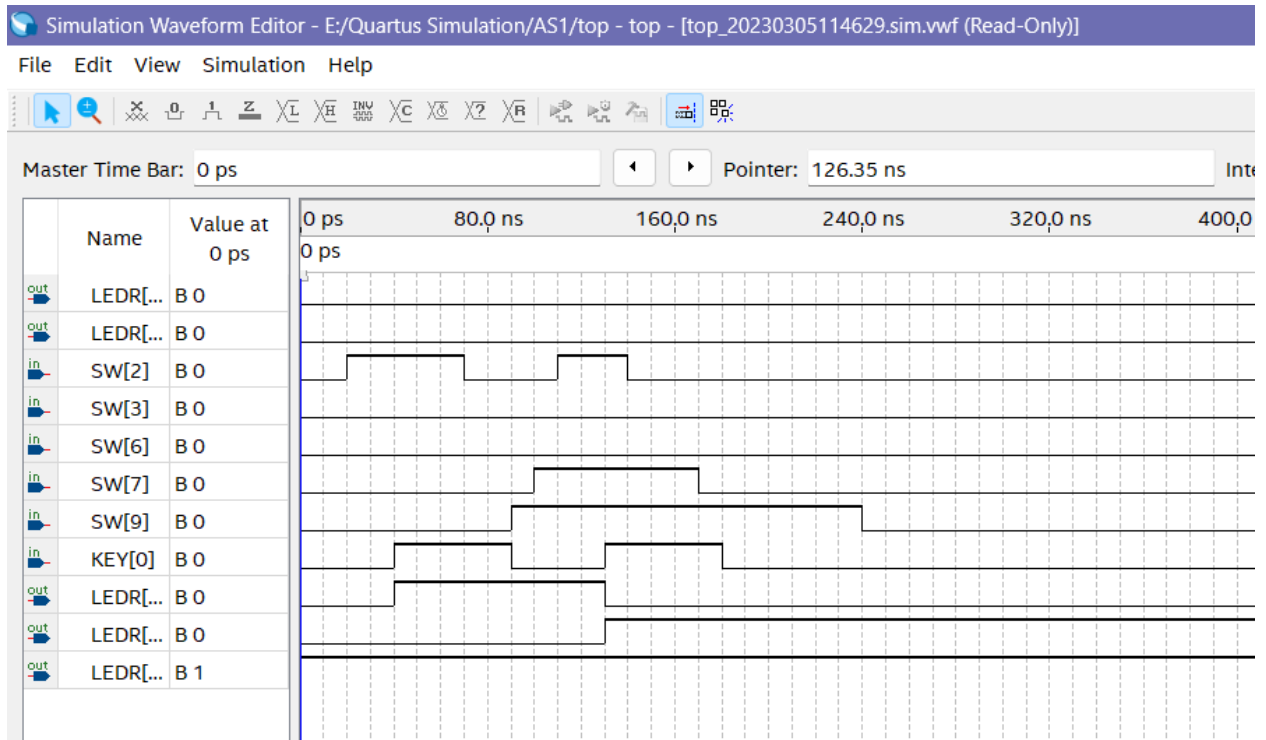
LEDR[2] and LEDR[3] assigned to REGA output. As shown in the below fig only both these LEDRs are working.



3. When input to decoder is given as 01 the output was 1 showing us the REGB is selected as expected. Further when input 4 was passed to the MUX the output was retained in REGB.



Testing SW[9] or MUX select =1 for REGB which worked as expected.



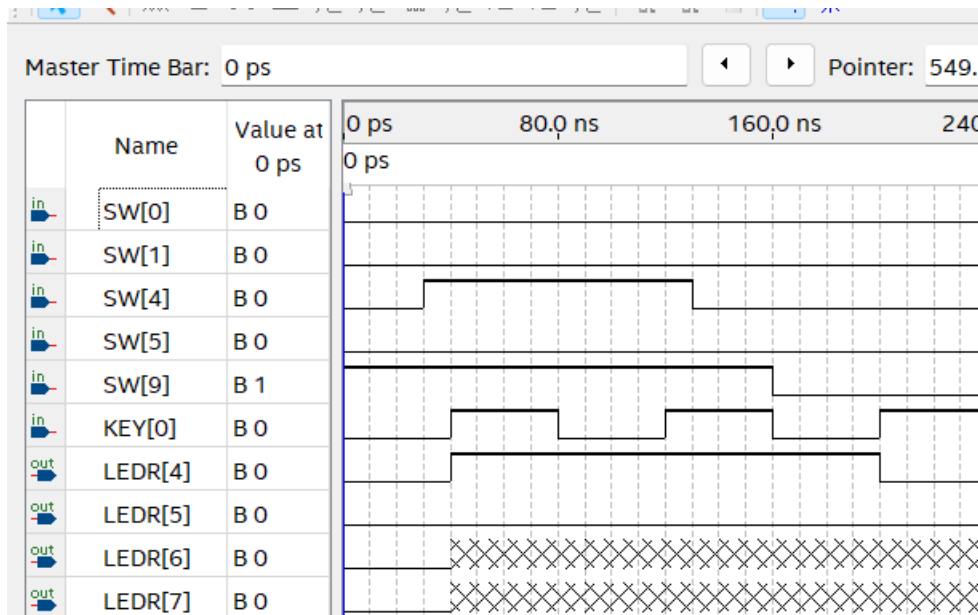
C. Testing the ALU

The testing of the ALU was the most tedious task as it contain two 4 bit inputs which are updated at every clocking cycle. To simplify our process we tested the ALU bit by bit giving one bit input at a time until all the bits gave us consistent results.

1. When REGB is selected and given input 1. The output of the ALU gave us the result 1 as expected.

Here LEDR[7:4] was assigned to output of ALU.

When REGB[0]=SW[4]=1, then A+B=1.



2. Assume we give 4'b1100 and 4'b0011 as the two inputs to MUX and let the select lines be SW[9]. Let the select lines of the decoder be SW[0] and SW[1].

FOR MUX

INPUT SW[9]	OUTPUT
SW[9]=0	1100
SW[9]=1	0011

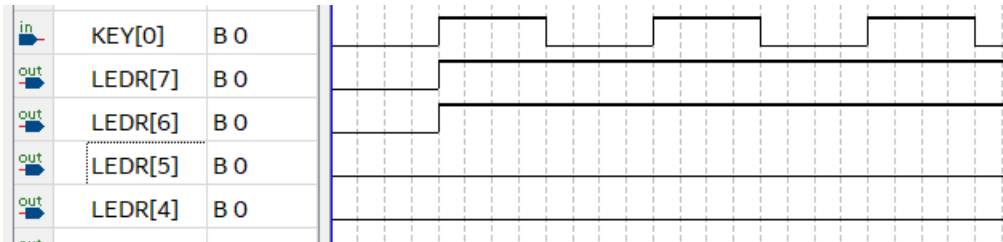
FOR DECODER

SW[0]	SW[1]	OUTPUT ENABLE
0	0	REGA
0	1	REGB
1	0	REG_OUT
1	1	40 xxxx

ADDITION when S=0:

- a. WHEN SW[9]=0 and SW[0]SW[1]=00.

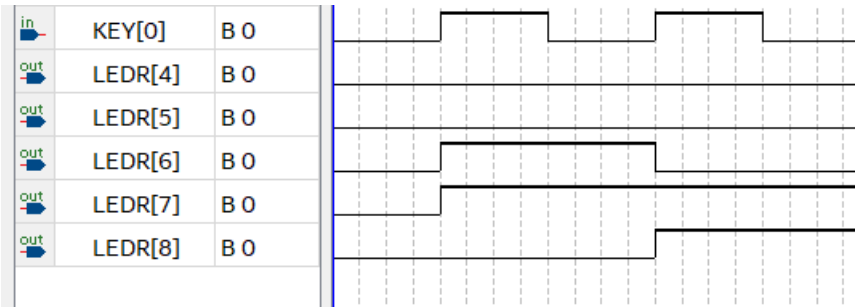
Here REGA should be passed with value of 1100. When added with ALU ALU_OUT=1100.



The output is 1100 as expected as ALU_OUT[3:0]=LEDR[7:4]

- b. WHEN SW[9]=0 and SW[0]SW[1]=01.

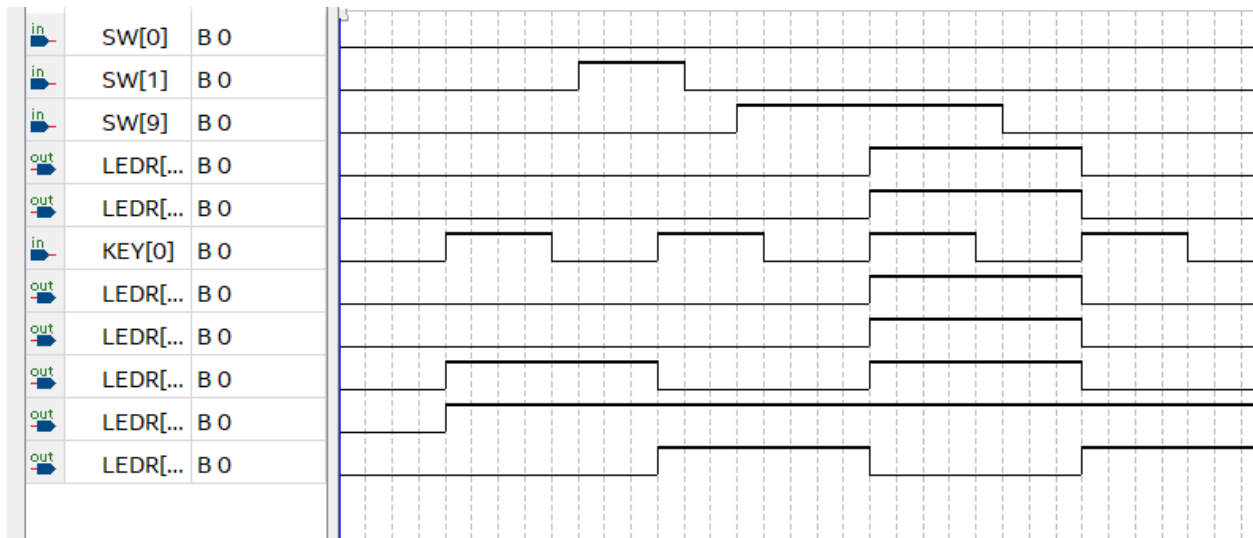
Here REGB should be passed with value of 1100. When added with ALU
ALU_OUT=1100+1100=1000 with CARRY.



In the second clock cycle we can see the output from ALU is 1000 and Carry which is LEDR[8] =1 as expected.

- c. WHEN SW[9]=1 and SW[0]SW[1]=00.

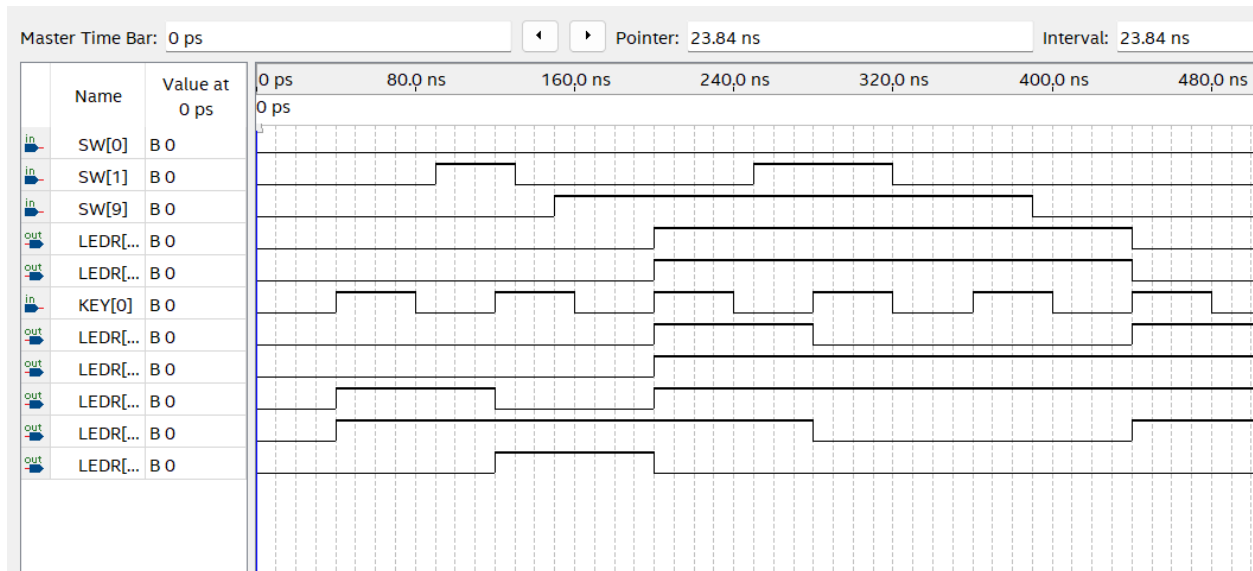
Here REGA should be passed with value of 0011. When added with ALU
ALU_OUT=0011+1100=1111 with CARRY=0.



d. WHEN SW[9]=1 and SW[0]SW[1]=01.

Here REGB should be passed with value of 0011. When added with ALU

ALU_OUT=0011+0011=0110 with CARRY=0.

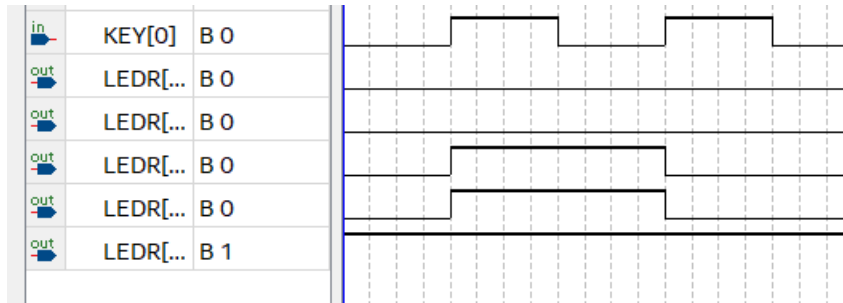


Therefore, we can conclude the ALU is doing the adding operation safely.

SUBTRACTION when S=1:

- a. WHEN SW[9]=0 and SW[0]SW[1]=00.

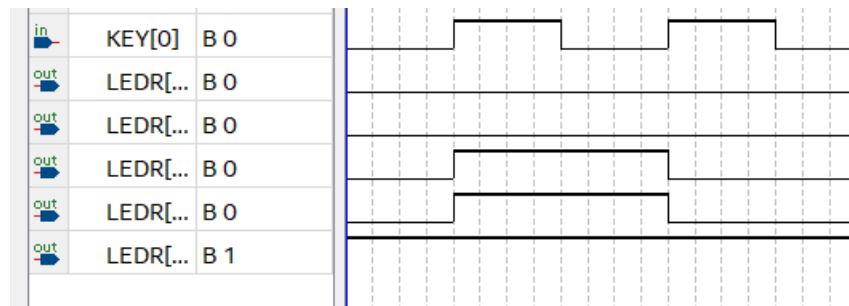
Here REGA should be passed with value of 1100. When subtracted with ALU ALU_OUT=1100 with CARRY =1



The output is 1100 as expected as ALU_OUT[3:0]=LEDR[7:4]

- b. WHEN SW[9]=0 and SW[0]SW[1]=01.

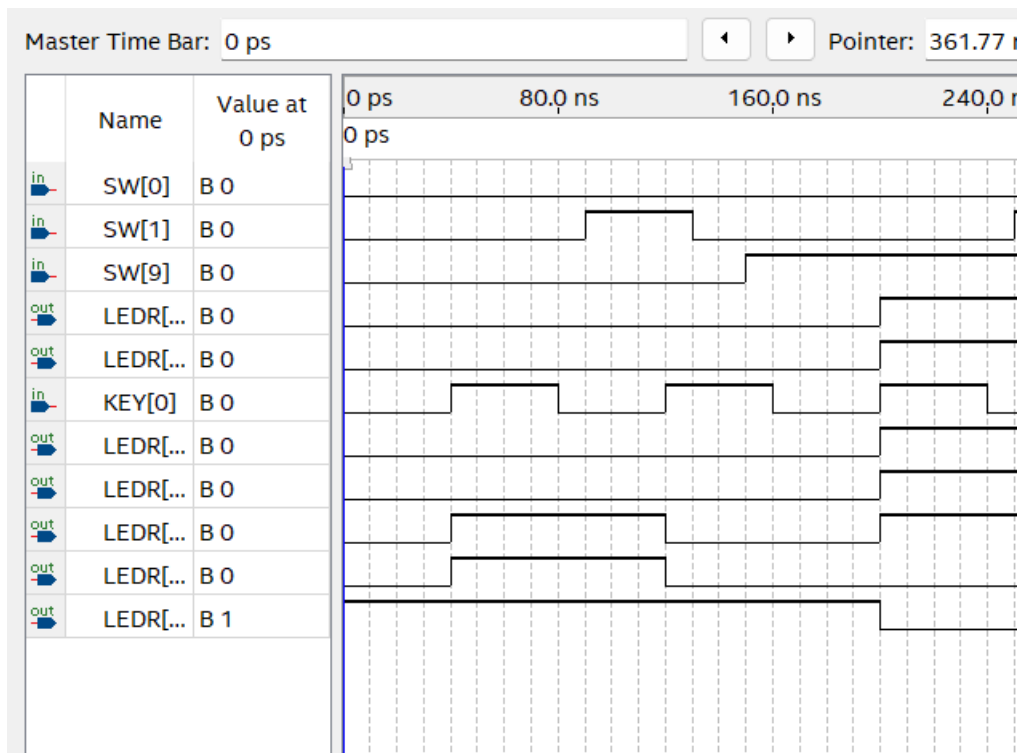
Here REGB should be passed with value of 1100. When subtracted with ALU ALU_OUT=1100-1100=0000 with CARRY=1.



In the second clock cycle we can see the output from ALU is 1000 and Carry which is LEDR[8] =1 as expected.

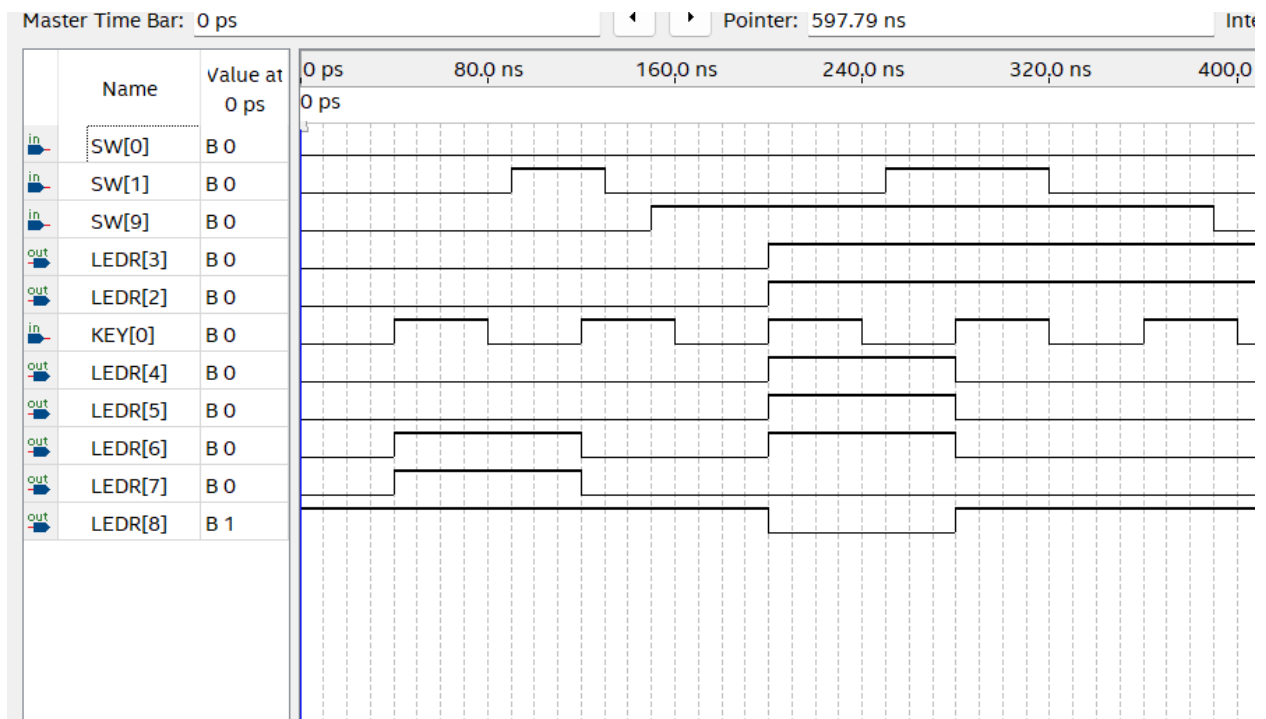
- c. WHEN SW[9]=1 and SW[0]SW[1]=00.

Here REGA should be passed with value of 0011. When subtracted with ALU ALU_OUT=0011-1100=0111 with CARRY=0.



- d. WHEN SW[9]=1 and SW[0]SW[1]=01.

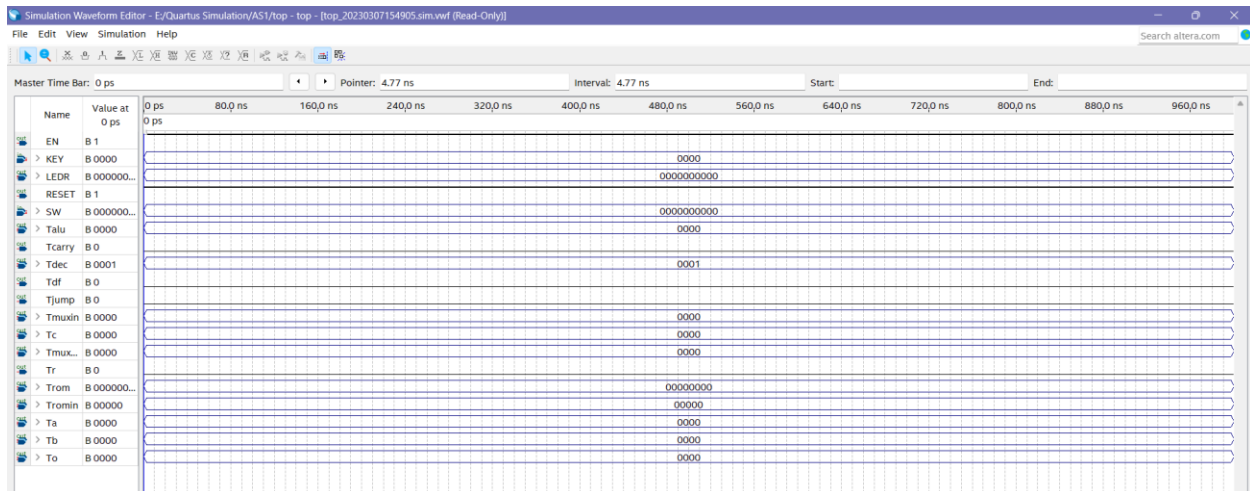
Here REGB should be passed with value of 0011. When subtracted with ALU ALU_OUT=0011-0011=0000 with CARRY=1.



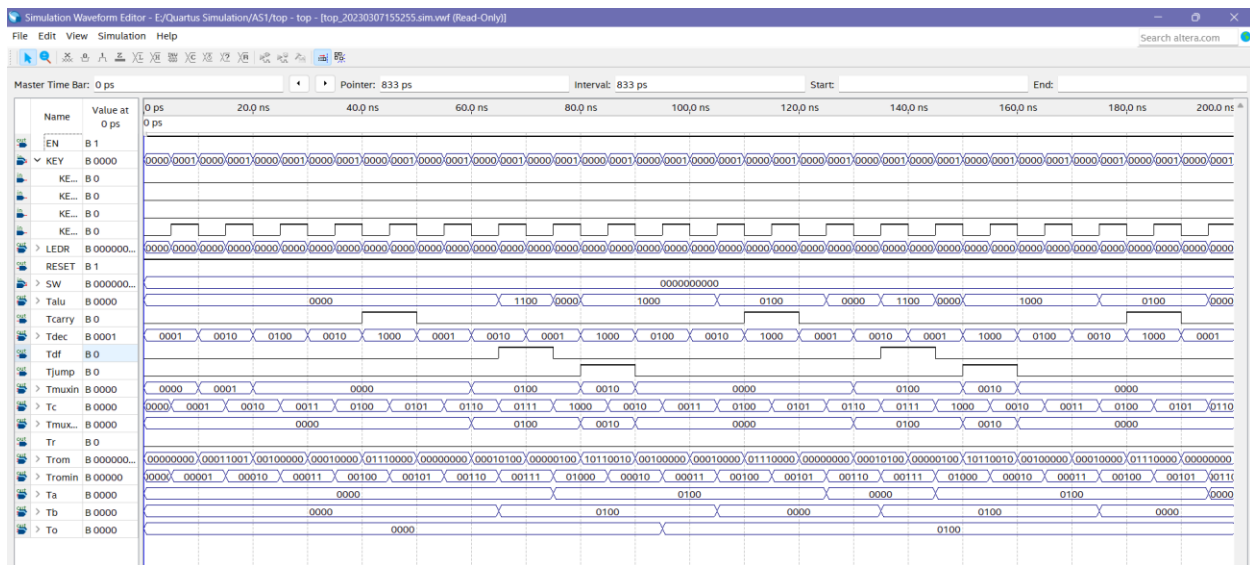
Therefore, we can conclude the ALU is doing the subtraction operation safely.

Final output

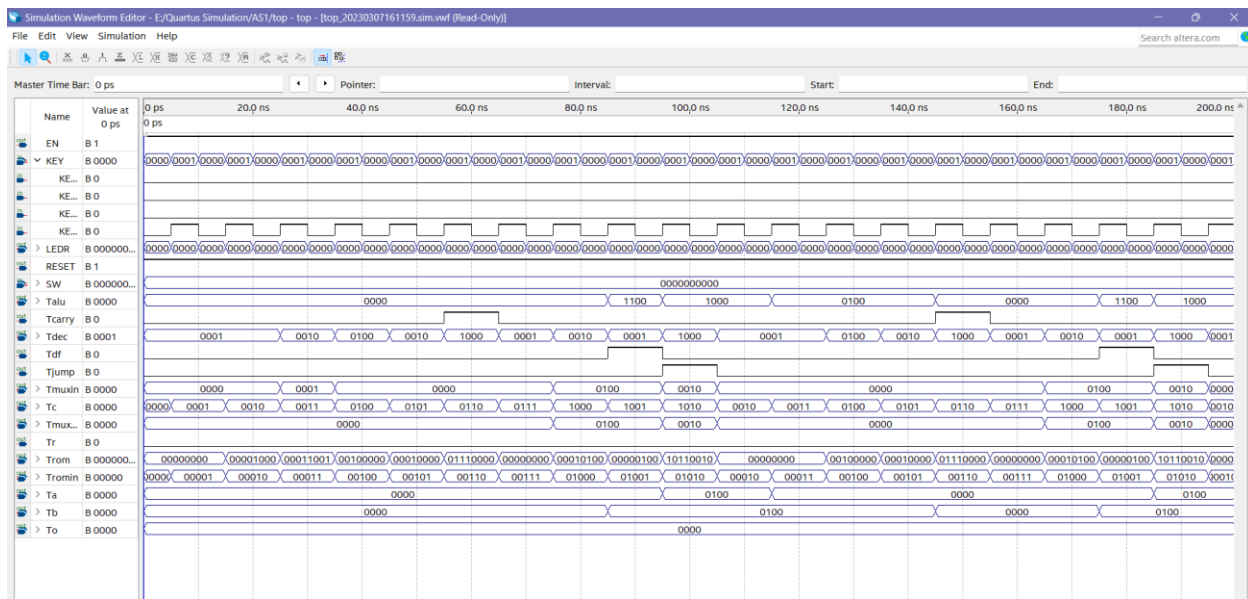
- The first attempt to debug the all the outputs were coming zero. The clock was not given as the input.



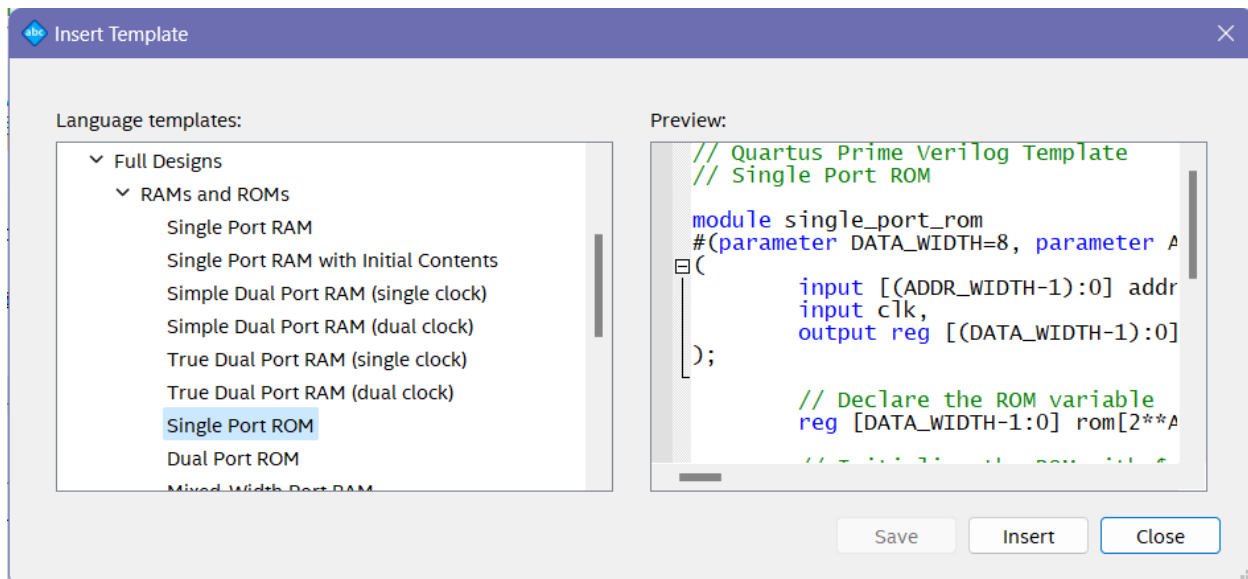
- After giving the clock input the registers started to load the values. But the output values of the registers were 0 and 8. Which is not the expected Fibonacci sequence.



After testing the wire connections, we found out that the **S input** to the ALU was not given instead an always high signal was passed which means the ALU performs only subtraction. After debugging that error, the output was now stuck at 0



After several attempts of debug, we found out that the ROM and the counter should be not working in the same clock cycle for transferring the bits. Therefore, we had to create a new ROM file with a negative edge. This time for more experience we created ROM using templates as follows.

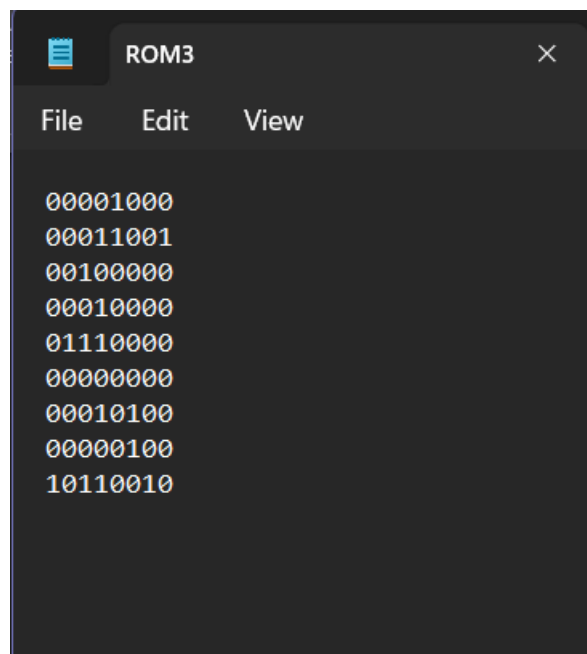


```

3
4 module ROM3
5   #(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=4)
6   (
7     input [(ADDR_WIDTH-1):0] addr,
8     input clk,
9     output reg [(DATA_WIDTH-1):0] q
10  );
11
12    // Declare the ROM variable
13    reg [DATA_WIDTH-1:0] rom[2**ADDR_WIDTH-1:0];
14
15    // Initialize the ROM with $readmemb. Put the memory contents
16    // in the file single_port_rom_init.txt. Without this file,
17    // this design will not compile.
18
19    // See Verilog LRM 1364-2001 Section 17.2.8 for details on the
20    // format of this file, or see the "Using $readmemb and $readmemh"
21    // template later in this section.
22
23    initial
24    begin
25        $readmemb("ROM3.txt", rom);
26    end
27
28    always @ (negedge clk)
29    begin
30        q <= rom[addr];
31    end
32
33 endmodule

```

ROM3.txt file used to store the data in the ROM.



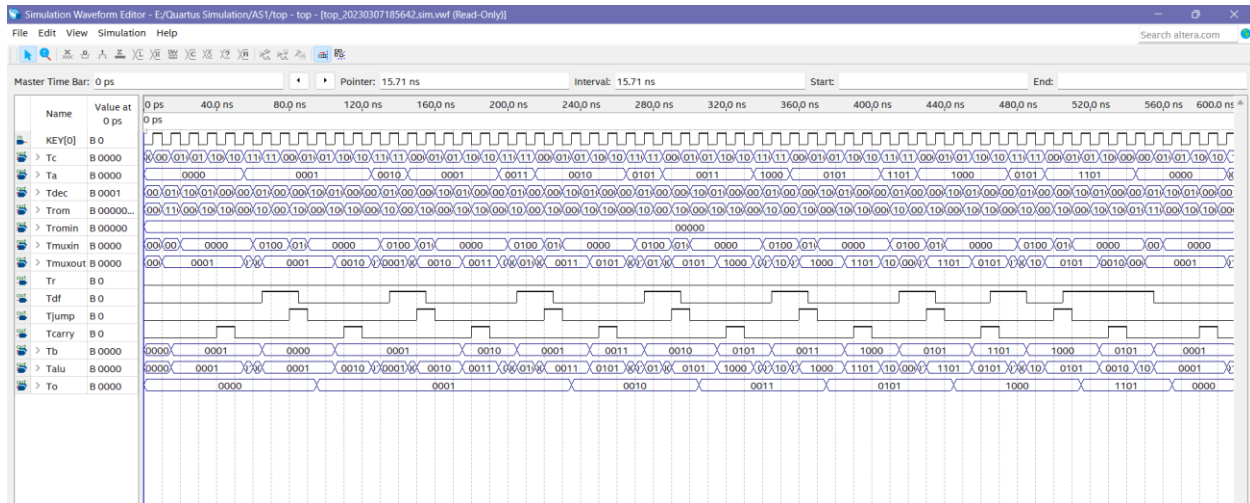
Finally, after debugging and arranging the code the output came as expected in the output register.

The Fibonacci sequence is as follows.

0, 1, 2, 3, 5, 8,

And after the 4-bit maximum value of 13 the output started from 0 again.

The output from output register is To.



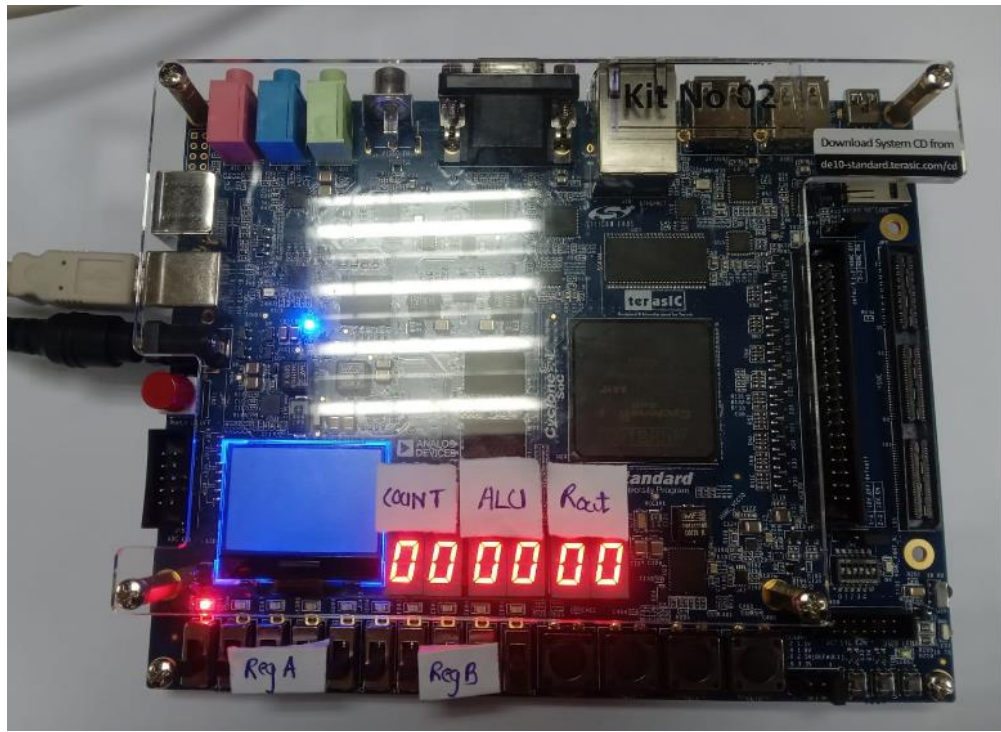
- Tc-counter output
- Talu-ALU output
- Ta-RegA output
- Tb-RegB output
- To-Reg_out output
- Tcarry-Carry output
- Tjump-Jump output
- Tdec-Decoder output
- Tromin-ROM input
- Trom-ROM output
- Tmixin-MUX input
- Tmuxout-MUX output
- Tdf-Dflipflop output

OUTPUT from FPGA

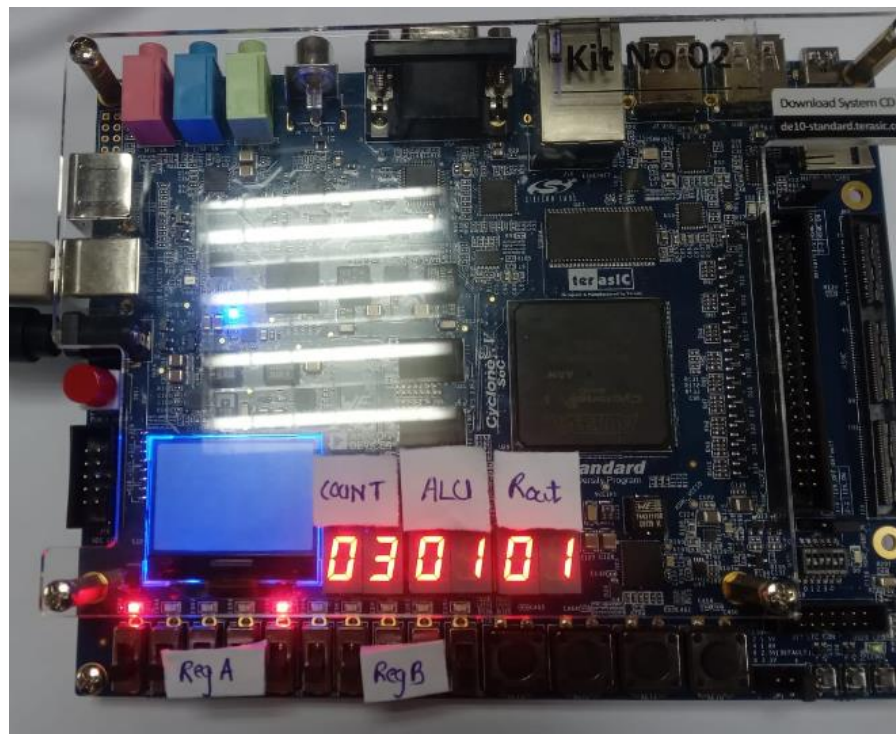
The outputs were assigned as followed:

- REGA- LEDR[8:5]
- REGB- LEDR[3:0]
- CLOCK- LEDR[9]
- ALU- HEX3 and HEX2
- COUNTER- HEX5 and HEX4
- REGo- HEX1 and HEX0

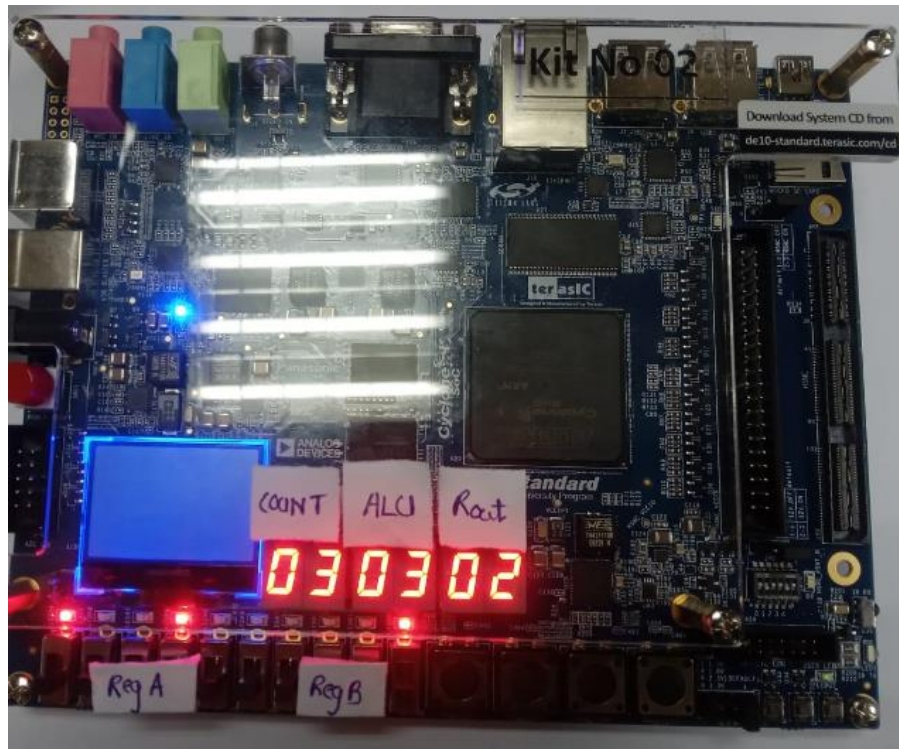
1. Output0



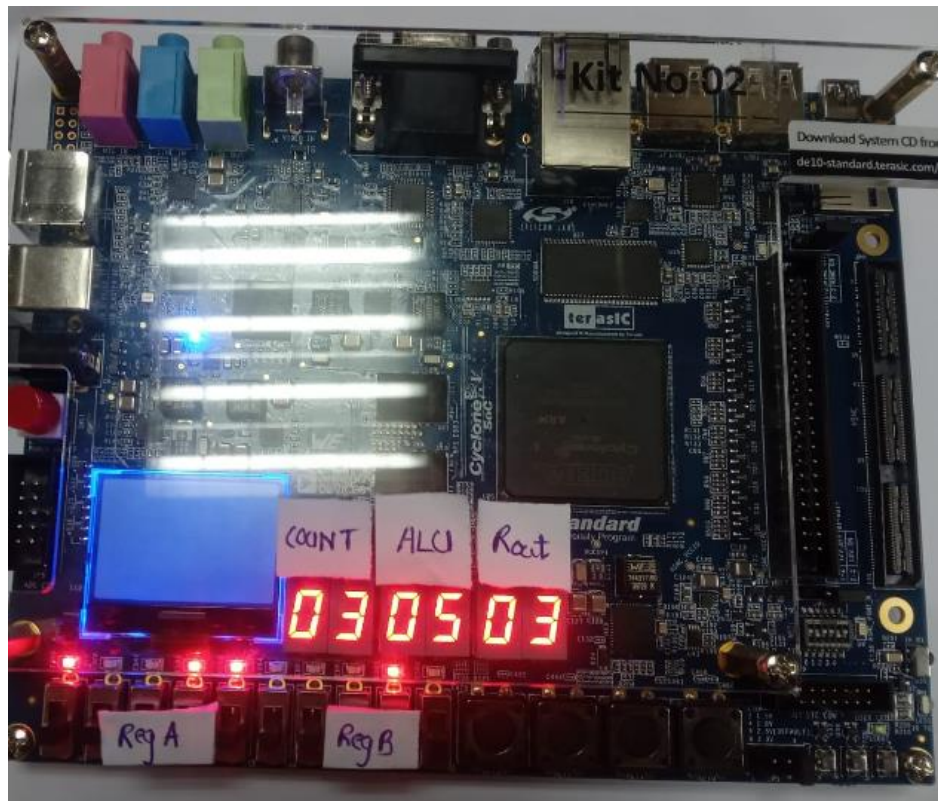
2. Output1



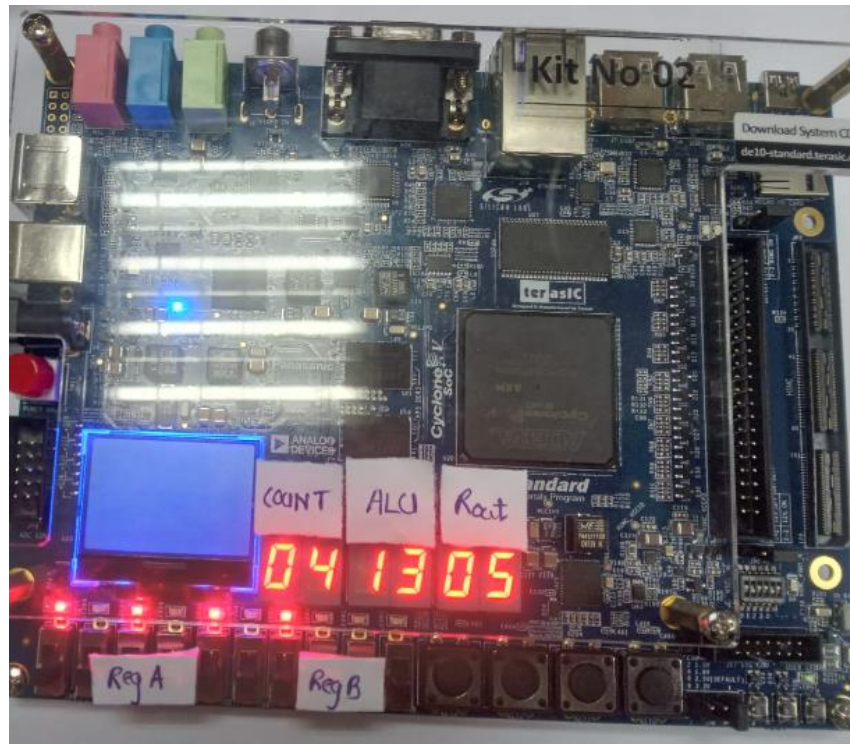
3. Output2



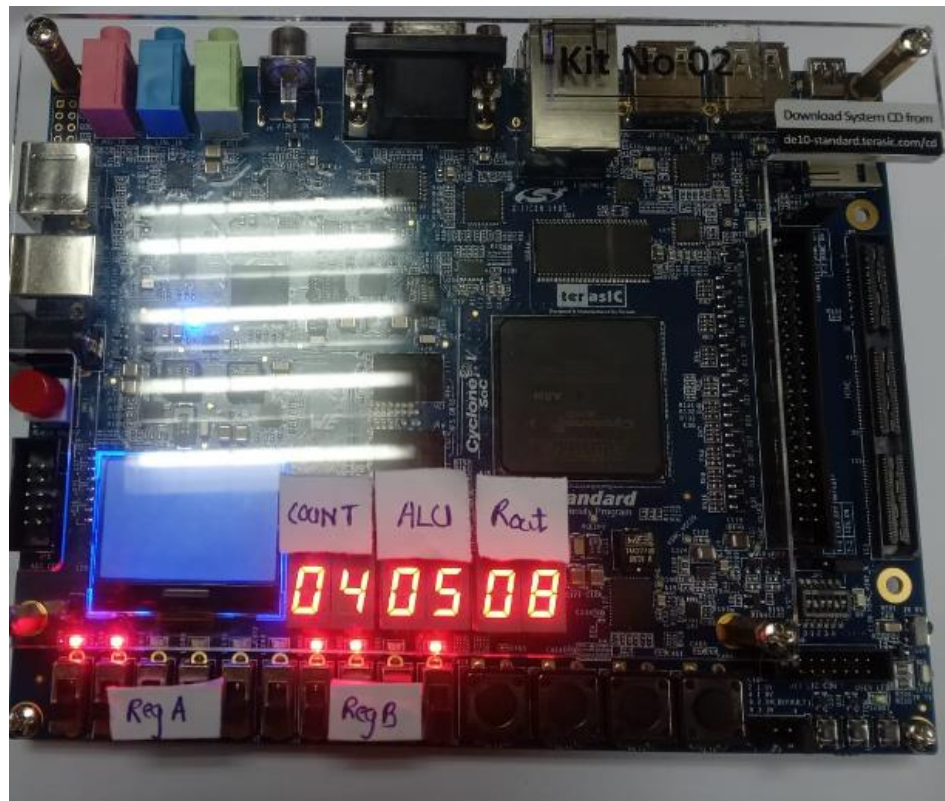
4. Output3



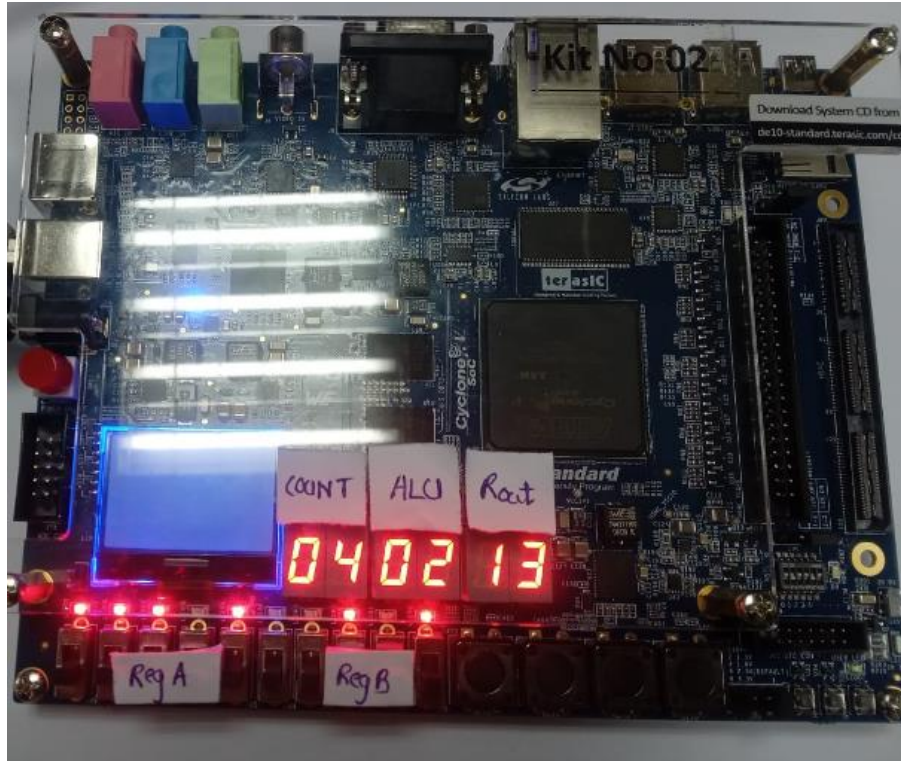
5. Output5



6. Output8



7. Output13



7 Concluding Remarks

In the above assignment we constructed a 4-bit microcontroller using Verilog. In this process we learned how to build each components individually and test them and finally integrate all the models in the final module and debug them. Several issues were faced starting from the EEPROM clocking. The bug was resolved by changing the EEPROM to negative edge. Also minor bugs in the patching of top module also resulted in the delay output but once debugged step by step they were resolved and the output was obtained. **One important lesson learned was that the circuit could be programmed easily but the debugging and simulation took nearly 70% of our time.** Once the simulation was fixed the circuit worked the first time when patched in the FPGA.