32 Bit SIngle Cycle MIPS Processor Report

EL-6463 ADVANCED COMPUTER HARDWARE DESIGN

By

Raj Kamal (rk3098)

Pankaj Subhash Yewale (psy231)

Utkarsh Gautam (ug277)

Albin Gomes (ag4806)

Sagar Panchal (sap586)

1. Introduction

The CPU or Central Processing Unit is the hardware in any Computer System that performs the tasks based on the instructions given by the computer programs. This report gives the description of the project on the design of NYU-6463 Processor. NYU-6463 is a 32-bit single cycle MIPS processor. The processor is designed using Xilinx ISE Design Suite and simulations are performed on ModelSim.

1.1 Instruction types

NYU-6463 Processor has three instruction types:

a) R-Type for arithmetic instructions
b) I-Type for immediate value operations, load and store instructions
c) J-Type for Jump instructions

The three instruction types are shown in the table below.

Opcode (6-bits)	Rs (5-bits)	Rt (5-bits)	Rd (5-bits)	Shamt (5-bits)	Funct (6-bits)

Opcode (6-bits)	Rs (5-bits)	Rt (5-bits)	Address/Immediate (16-bits)

Opcode (6-bits)	Address (26-bits)

Each of three instructions are 32 bit instructions and they defines the type of instructions as well as other information like operand values and destination where the results are to be stored. Each of the instruction fields has its own importance and the tasks to be performed by the processor depends on the data in these instruction fields.

The description regarding each instruction fields is given in the table below.

Field	Description
Opcode	6-bit primary operation code
Rd	5-bit specifier for the destination register
Rs	5-bit specifier for the source register
Rt	5-bit specifier for the target (source/destination) register
Address/Immediate	16-bit signed immediate used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacement
Address	26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address
Shamt	5-bit shift amount
Funct	6-bit function field used to specify functions within the primary opcode

2.2 Instruction Set

To perform different tasks depending on different instructions, a specific set of instructions are defined, which are supported by the NYU-6463 Processor. Any operation on the processor is carried out using these set of instruction. The list of the instruction set, supported by NYU-6463 Processor is given in the table below.

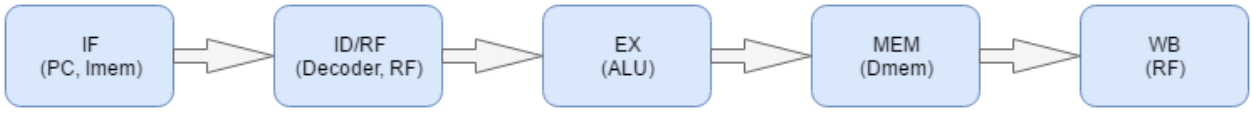
Mnemonic	Description	Type	Opcode (Hex)	Func (Hex)	Operation
ADD	Add Registers	R	00	10	Rd = Rs + Rt
ADDI	Add Immediate	I	01		Rt = Rs + SignExt(Imm)
SUB	Subtract Registers	R	00	11	Rd = Rs - Rt
SUBI	Subtract Immediate	I	02		Rt = Rs - SignExt(Imm)
AND	Register bitwise And	R	00	12	Rd = Rs & Rt
ANDI	And Immediate	I	03		Rt = Rs & SignExt(Imm)
OR	Register bitwise OR	R	00	13	Rd = Rs | Rt
NOR	Register bitwise NOR	R	00	14	Rd = !(Rs | Rt)
ORI	OR Immediate	I	04		Rt = Rs | SignExt(Imm)
SHL	Shift Left by immediate bits	I	05		Rt = Rs << Imm
SHR	Shift Right by immediate bits	I	06		Rt = Rs >> Imm
LW	Load Word	I	07		Rt ← Mem[SignExt(Imm) + Rs]
SW	Store Word	I	08		Mem[SignExt(Imm) + Rs] ← Rt
BLT	Branch if less than	I	09		If (Rs < Rt) then PC = PC + 1 + Imm
BEQ	Branch if equal	I	0A		If (Rs = = Rt) then PC = PC + 1 + Imm
BNE	Branch if not equal	I	0B		If (Rs != Rt) then PC = PC + 1 + Imm
JMP	Jump	J	0C		PC = {(PC + 1)[31:26], address}
HAL	Halt	J	3F		

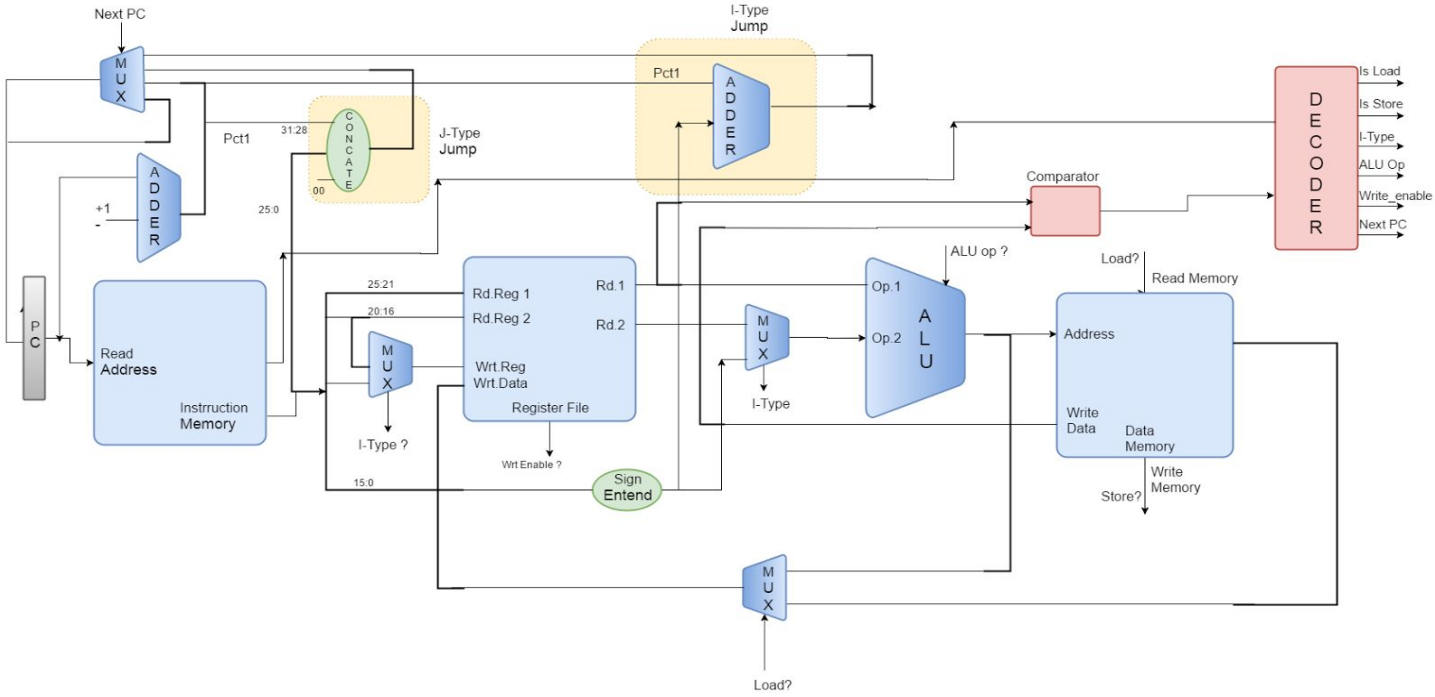
2.3 Working of the processor

The designed processor is a single cycle processor i.e. it performs instruction fetch, decode, execution, memory and write-back, all in a single cycle. Each instruction is divided into different fields as shown in the table above. The Opcode field is used to determine the type of the instruction. Which type of the task to be performed by the ALU depends on the type of the instruction. The fields Rd, Rs and Rt are used to address the Registers File. The Register File reads the respective address and gives the data in the address as the output. These are the data on which the operations are performed by the ALU. This is the point where the Control Unit comes to play its role. Depending on the data in the instruction fields, the Control Unit decides whether to compute the memory address or to perform the arithmetic or comparison operation. Depending on the Instruction decoded, the results from the ALU are directed appropriately. If the task performed is arithmetic, the ALU result is stored into a register. If the task is load or store, the ALU result is then used to address the data memory. Finally, the ALU result or the memory value is written back to the Register File.

2. Block Diagram and Components

2.1 Block Diagram





2.2 Components

PC:

PC (Program Counter) is a 32 bit register that holds the address of next instruction to be executed.

Instruction Memory:

Instruction Memory is initialized to contain the program to be executed.

Register File:

This block contains 32 32-bit registers. The register file supports two independent register reads and one register write in one clock cycle. 5 bits are used to address the register file.

ALU:
This block performs operations such as addition, subtraction, comparison, etc. It uses the control signals generated by the Decode Unit, as well as the data from the registers or from the instruction directly. It computes data that can be written into one of the registers (including PC).

Data Memory:
The data memory stores the data and is accessed using load word and store word instructions.

Comparator:
Comparator takes two inputs from the register file and determines whether one of the input is greater than the other or both the inputs are equal.

Decoder:
This block takes as input some or all of the 32 bits of the instruction, and computes the proper control signals to be utilized for other blocks. These signals are generated based on the type and the content of the instruction being executed.

Sign Extend:
Sign Extend is a component that is designed specially. It accepts one input, 16 bits wide, and gives one output. The output is 32 bits wide and it depends on the MSB of 16 bits wide input. If the MSB of the input is '1', then the output is created with all 16 MSBs equal to '1', or 'FFFF' concatenated with the input. If the MSB of the input is '0', then the output will be same as input, with 16 MSBs equal to '0'.

3. Simulations

Simulation is carried out using ModelSim and the screenshots were captured. The screenshots were captured at three checkpoints. Firstly, the ALU was designed. Before moving further, the simulation was carried out and screenshots were captured. Secondly, the Decoder Unit was designed, simulated and captured in the form of screen shots. The achievement of above two checkpoints was very important from design point of view. After the two checkpoints were achieved, the remaining design of processor was completed. The complete design of the processor was then simulated and the screen shots were captured at different stages of the operation.

The following sections shows the simulation screenshots, of ALU, Decoder Unit and finally, the complete processor.

3.1 Simulation of each components

32 bit Adder

The 32 bit adder is designed and the basic components used for adder includes NOT gate, AND gate and Or gate, which combines to form Half adder and Full adder, which further combines to form a complete 32 bit adder. Functional simulation and timing simulations are carried out and the screen shots were captured for some values.

Functional Simulation:

1) Test case 1:

Inputs	:	op_1	=	AABBCCDD
		op_2	=	11FF2233
Outputs	:	sum_out	=	BCBAEF10
		carry	=	0

2) Test case 2:

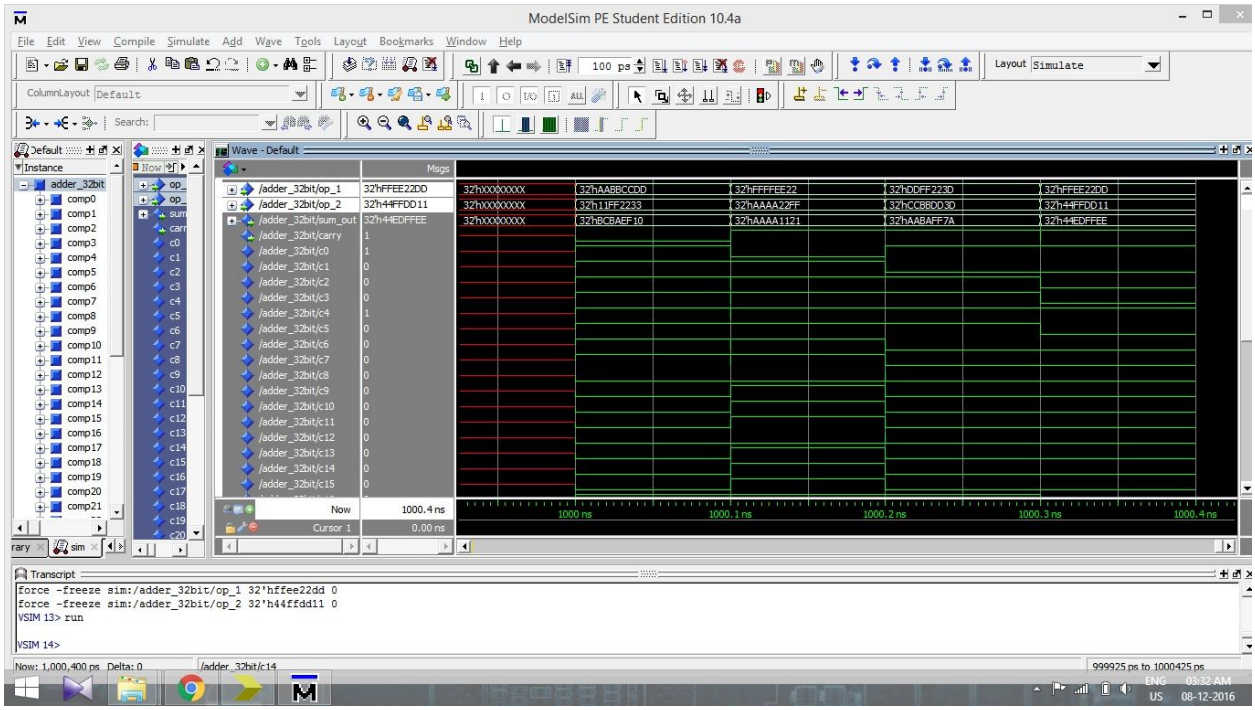
Inputs	:	op_1	=	FFFFEE22
		op_2	=	AAAA22FF
Outputs	:	sum_out	=	AAAA1121
		carry	=	1

3) Test case 3:

Inputs	:	op_1	=	DDFF223D
		op_2	=	CCBBDD3D
Outputs	:	sum_out	=	AABAFF7A
		carry	=	1

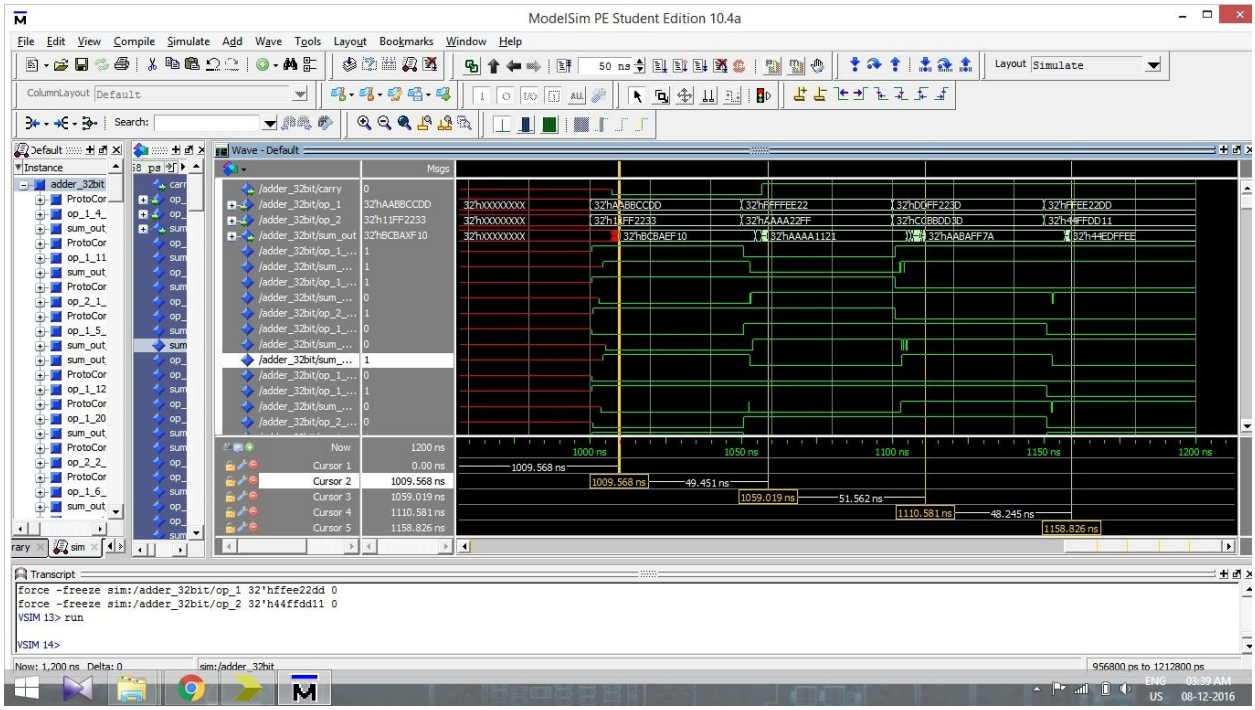
4) Test case 4:

Inputs	:	op_1	=	FFEE22DD
		op_2	=	44FFDD11
Outputs	:	sum_out	=	44EDFFEE
		carry	=	1



Timing Simulation:

For timing simulations, same values were used as the one used for Functional simulations. We can see the same outputs and same values for the carry bits as in the functional simulations, but with a delay as shown with the horizontal bars. The screenshot of the timing simulation is shown below.



Comparator:

In the comparator, the two inputs are given as input1 and input2. If the reset bit is active high or '1', then the output becomes '0'. Else if the reset bit it is active low or '0', and if both the inputs are same, the output bit shows the value '1', else if input1 is greater than input2, output bit shows the value '2' or if the input1 is less than input2, the output bit shows the value '3'.
The functional and timing simulations are carried out for the above logic, and two test cases are implemented to prove the logic.

1) Test case 1:

Input1 = FFEE2211
Input2 = FFEE2200
rst = 0
Output = 2

Input1 = FFEE2211
Input2 = FFEE2244
rst = 0
Output = 3

Input1 = FFEE2211
Input2 = FFEE2211
rst = 0
Output = 1

Input1 = FFEE2211
Input2 = FFEE2211
rst = 1
Output = 0

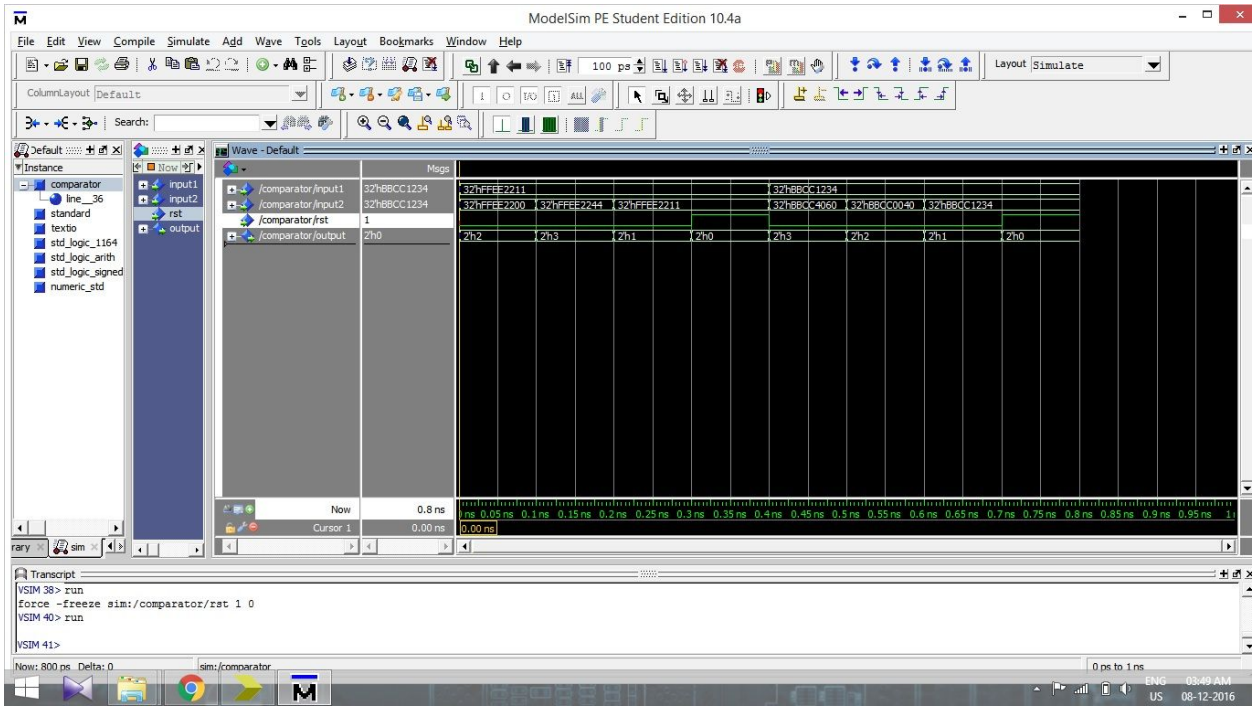
2) Test case 2:

Input1 = BBCC1234
Input2 = BBCC4060
rst = 0
Output = 3

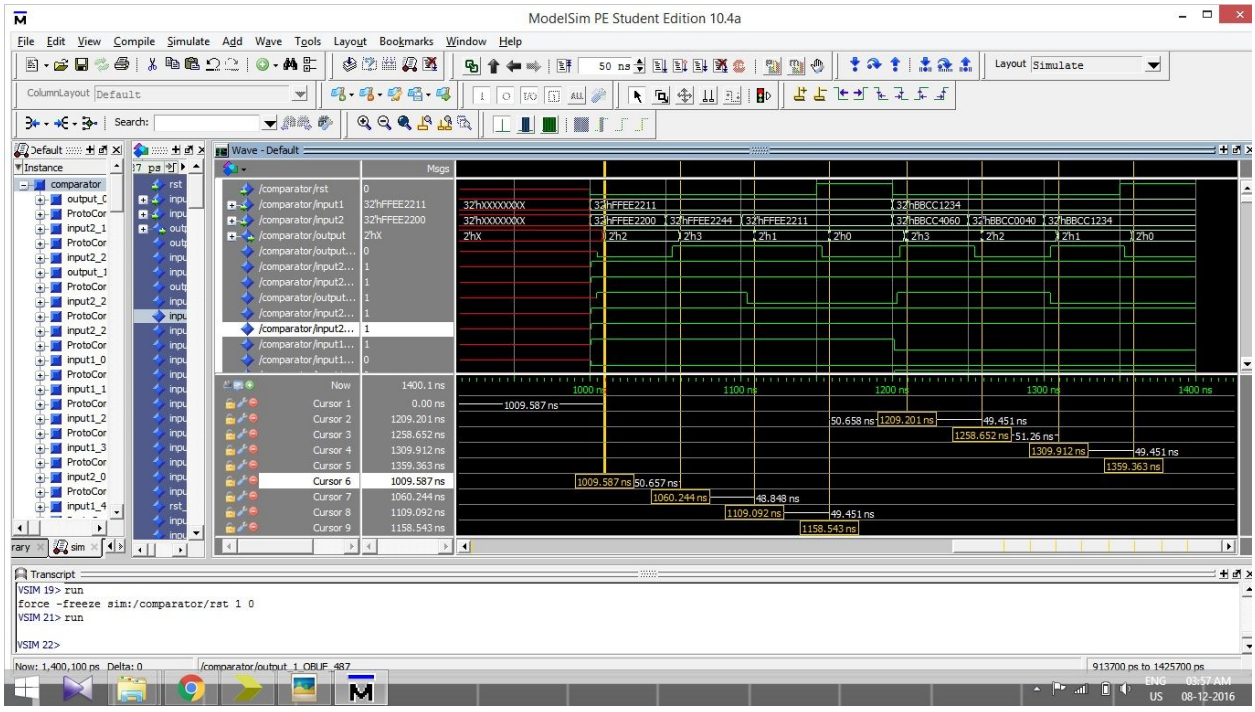
Input1 = BBCC1234
Input2 = BBCC0040
rst = 0
Output = 2

Input1 = BBCC1234
Input2 = BBCC1234
rst = 0
Output = 1

Input1 = BBCC1234
Input2 = BBCC1234
rst = 1
Output = 0



The above figure shows the functional simulation of a Comparator. The inputs and test cases as described above are taken for simulation. The same values are used for timing simulation also. Note the delays in timing simulation screenshot below. The delays are shown by multiple horizontal bars.

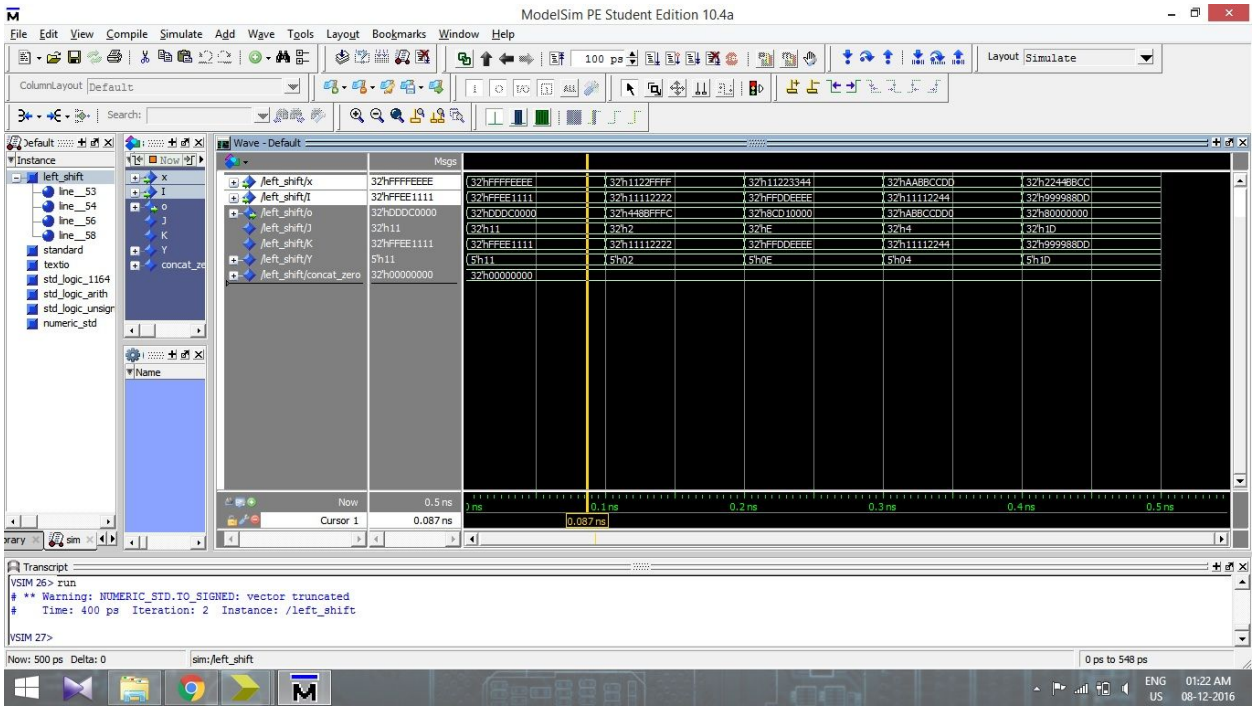


Left and Right Shift:

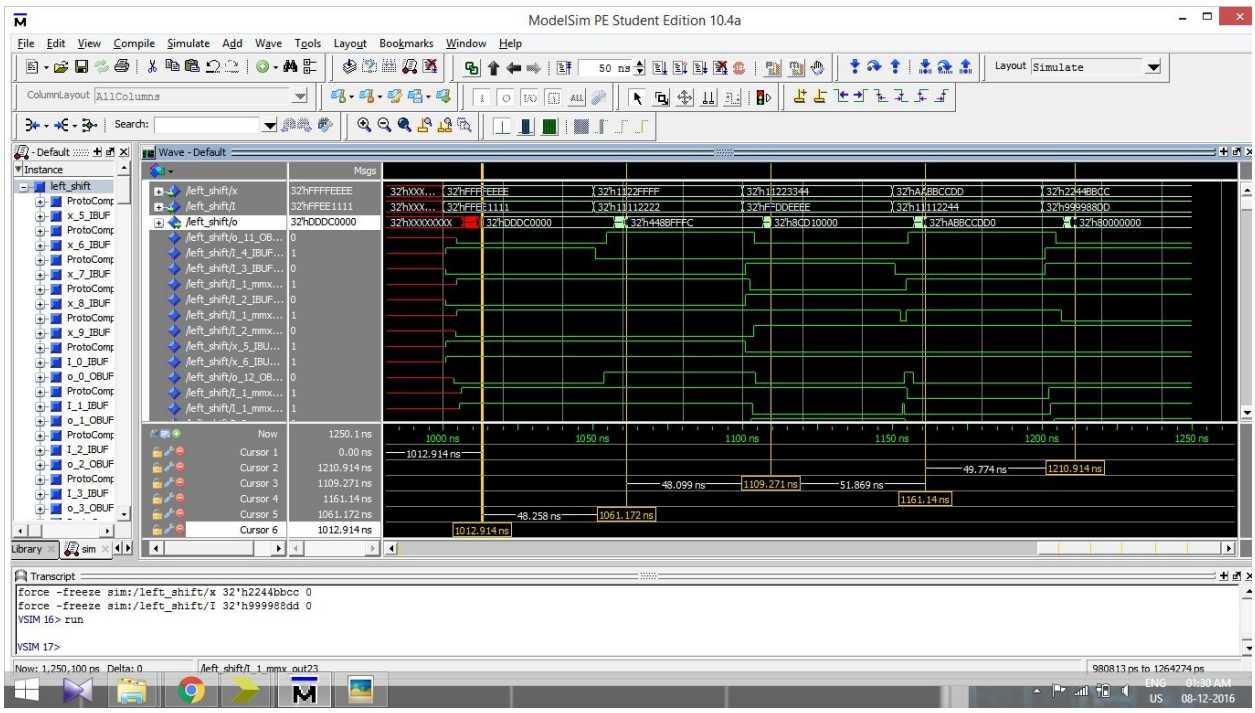
For left and right shifts, there are 2 inputs. One of the inputs (x) is to be shifted by number of bits that is obtained by certain operation on input 2 (I). The input 2 is converted from hexadecimal to decimal, and then divided by 32. The remainder is equal to the bits to be shifted for input 1. The reason behind the division by 32 is that the rotation does not exceeds 32 bits.

Left Shift:

1) Test case 1:
Input 1 = FFFFEEEE
Input 2 = FFEE1111
Output = DDDC0000 (Output left shifted by 17 bits)
2) Test case 2:
Input 1 = 1122FFFF
Input 2 = 11112222
Output = 448BFFFC (Output left shifted by 2 bits)
3) Test case 3:
Input 1 = 11223344
Input 2 = FFDDEEEE
Output = 8CD10000 (Output left shifted by 14 bits)
4) Test case 4:
Input 1 = AABBCCDD
Input 2 = 11112244
Output = ABBCCDD0 (Output left shifted by 4 bits)
5) Test case 5:
Input 1 = 2244BBCC
Input 2 = 999988DD
Output = 80000000 (Output left shifted by 29 bits)



Functional Simulation of left shift



Timing Simulation of left shift

Right Shift:

1) Test case 1:

   Input 1 = FFFFEEEE

   Input 2 = 00007FFF

   Output = DDDC0000 (Output right shifted by 17 bits)

2) Test case 2:

   Input 1 = 1122FFFF

   Input 2 = 11112222

   Output = 0448BFFF (Output right shifted by 2 bits)

3) Test case 3:

   Input 1 = 11223344

   Input 2 = FFDDEEEE

   Output = 00004488 (Output right shifted by 14 bits)

4) Test case 4:

   Input 1 = AABBCCDD

   Input 2 = 11112244

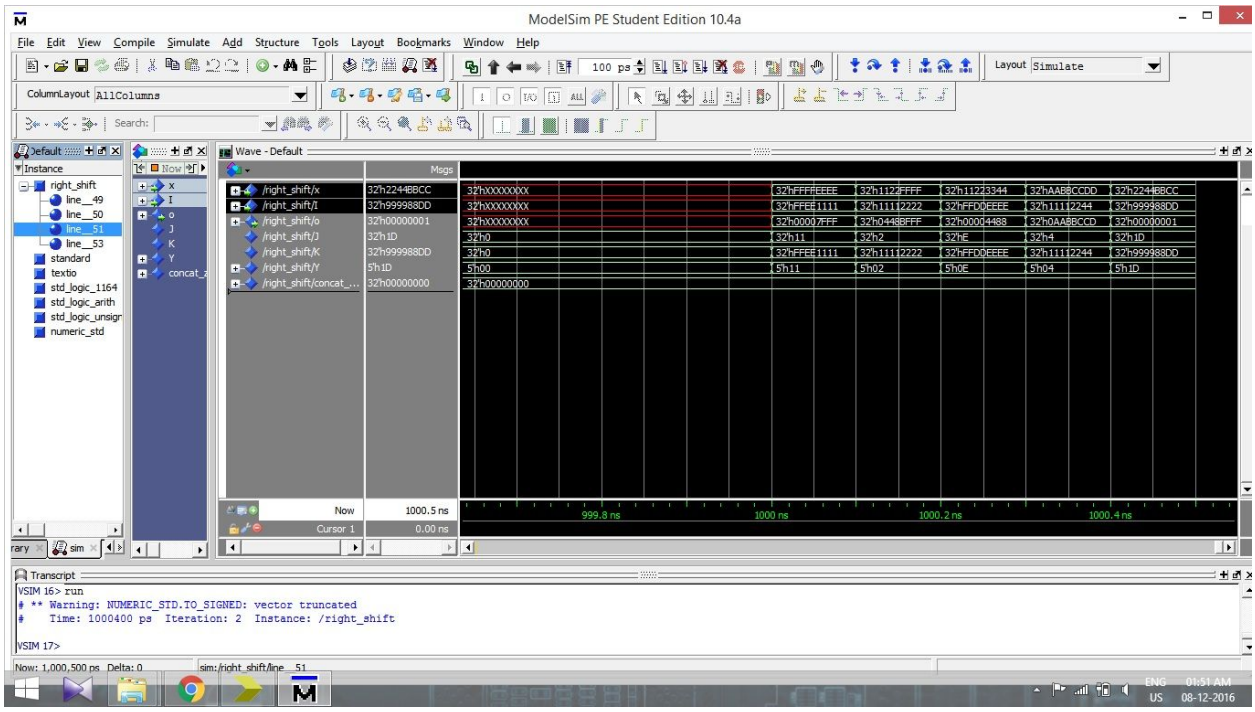
   Output = 0AABBCCD (Output right shifted by 4 bits)

5) Test case 5:

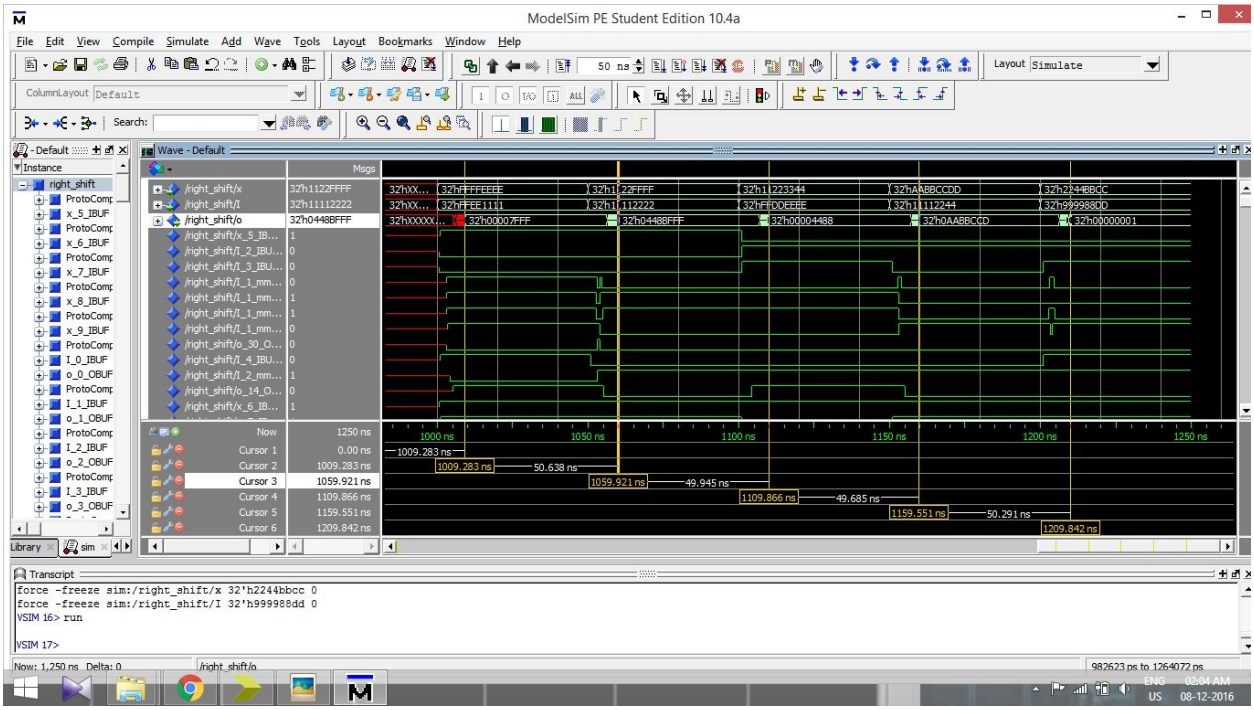
   Input 1 = 2244BBCC

   Input 2 = 999988DD

   Output = 00000001 (Output right shifted by 29 bits)



Functional simulation of Right Shift



Timing simulation of Right Shift

MUX 5 bit 2x1:

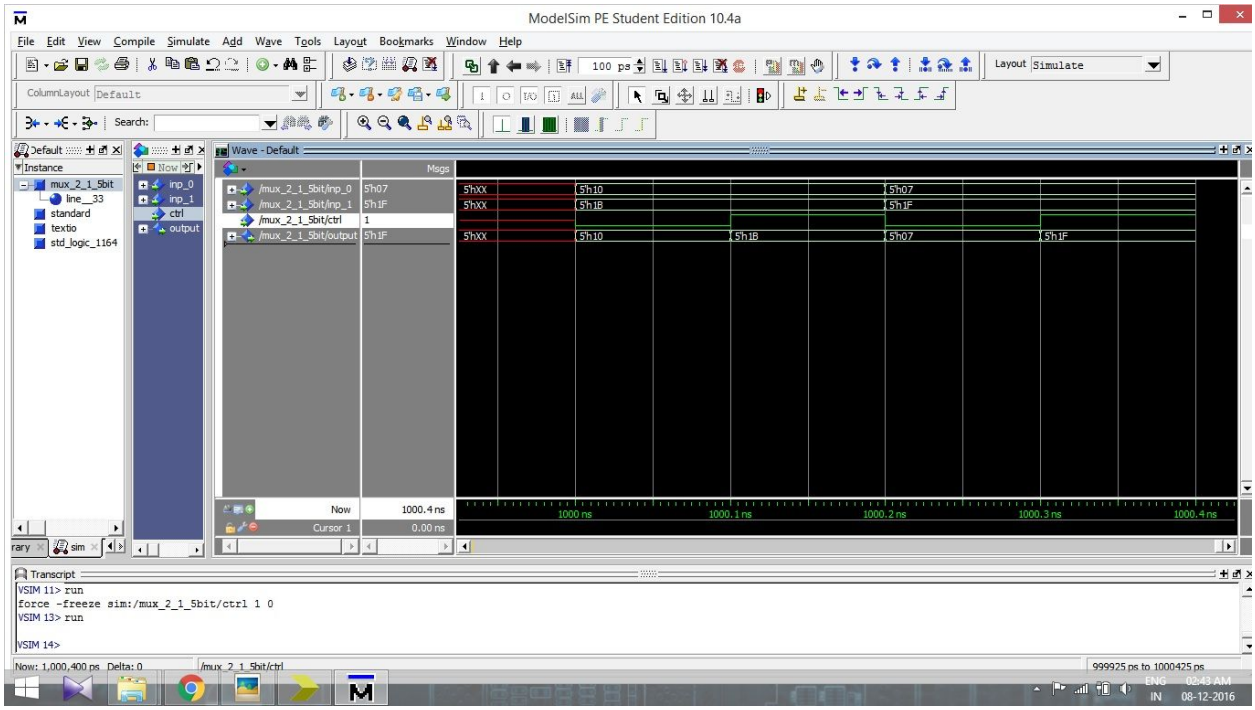
A 5 bit MUX is designed with 2x1 selection. Two 5 bits inputs (inp_0 and inp_1) are given, and a control input is to be selected between active high or '1' and active low or '0'. If the control is set as active low or '0', the output shows the exact same value as inp_0, otherwise, if the control is '1', then the output shows exactly same as inp_1.

1) Test case 1:

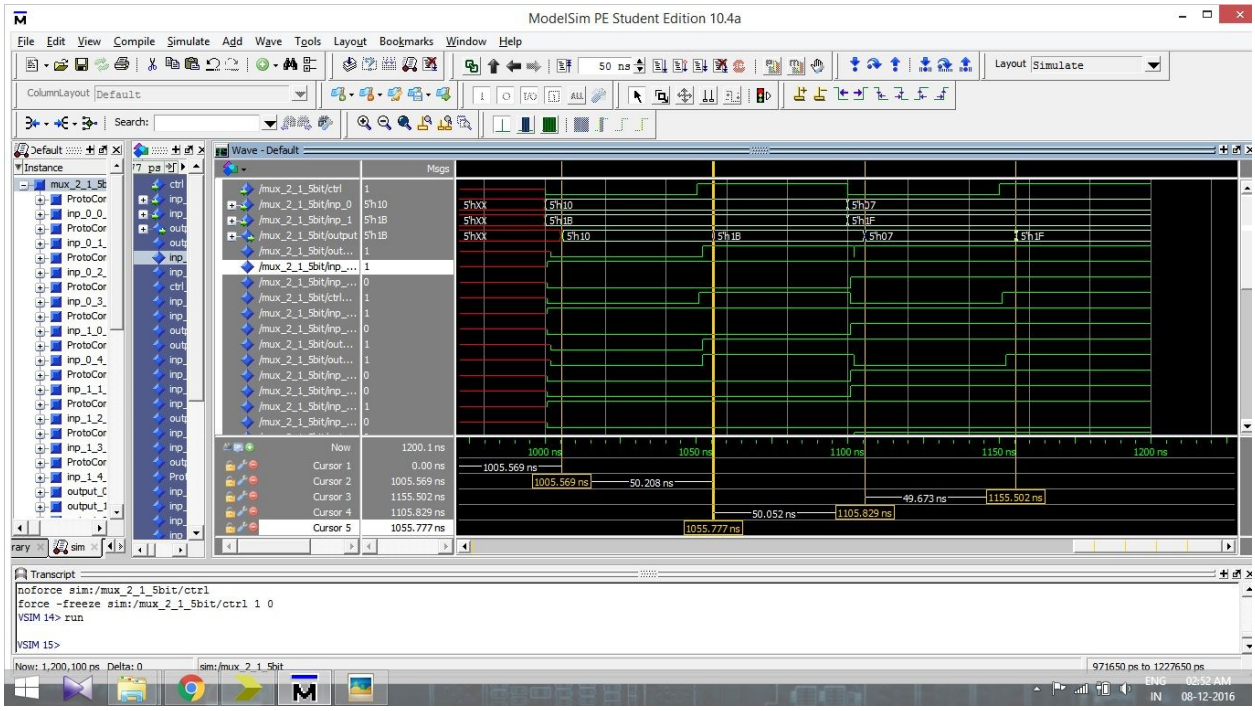
Inp_0	=	h`10			
Inp_1	=	h`1B			
Ctrl	=	0	Ctrl	=	1
Output	=	h`10	Ctrl	=	h`1B

2) Test case 1:

Inp_0	=	h`07			
Inp_1	=	h`1F			
Ctrl	=	0	Ctrl	=	1
Output	=	h`07	Output	=	h`1F



Functional simulation of MUX 5 bit 2x1



Timing simulation of MUX 5 bit 2x1

MUX 32 bit 2x1:

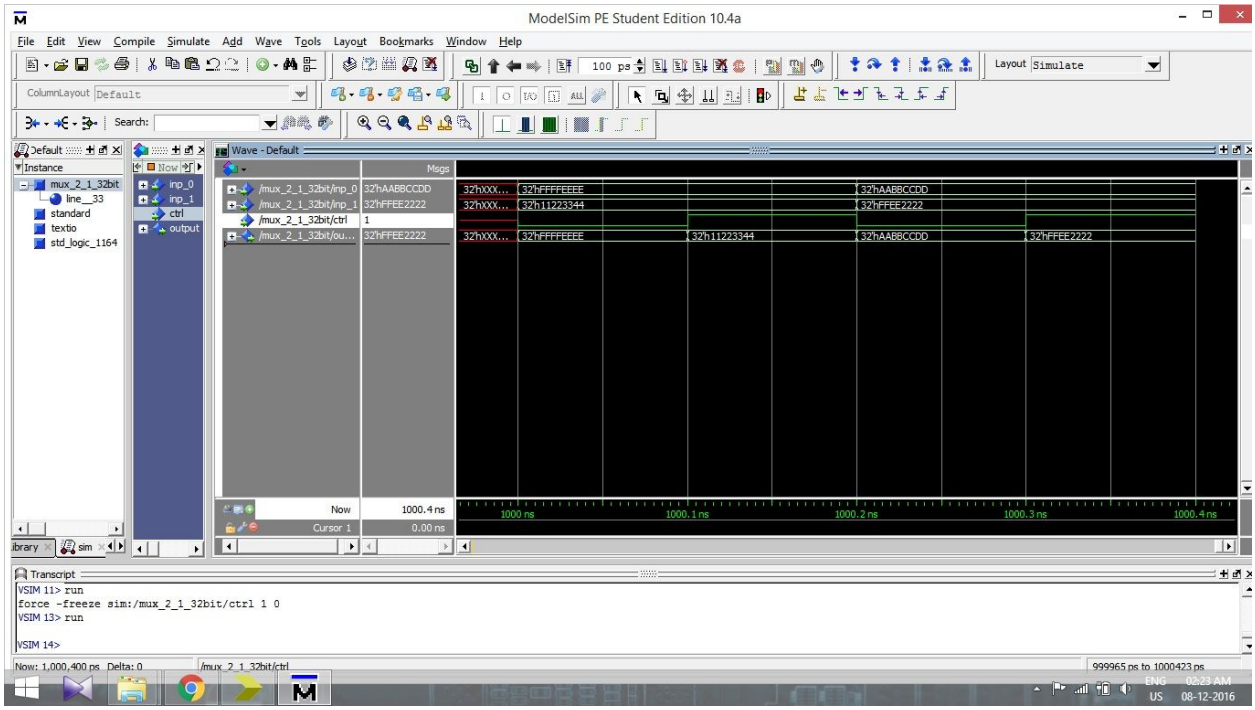
A similar MUX for 32 bits input is designed with 2x1 selection. If the control input is active low or '0', the output becomes equal to inp_0, otherwise, if control input is active high or '1', the output becomes equal to inp_1.

1) Test case 1:

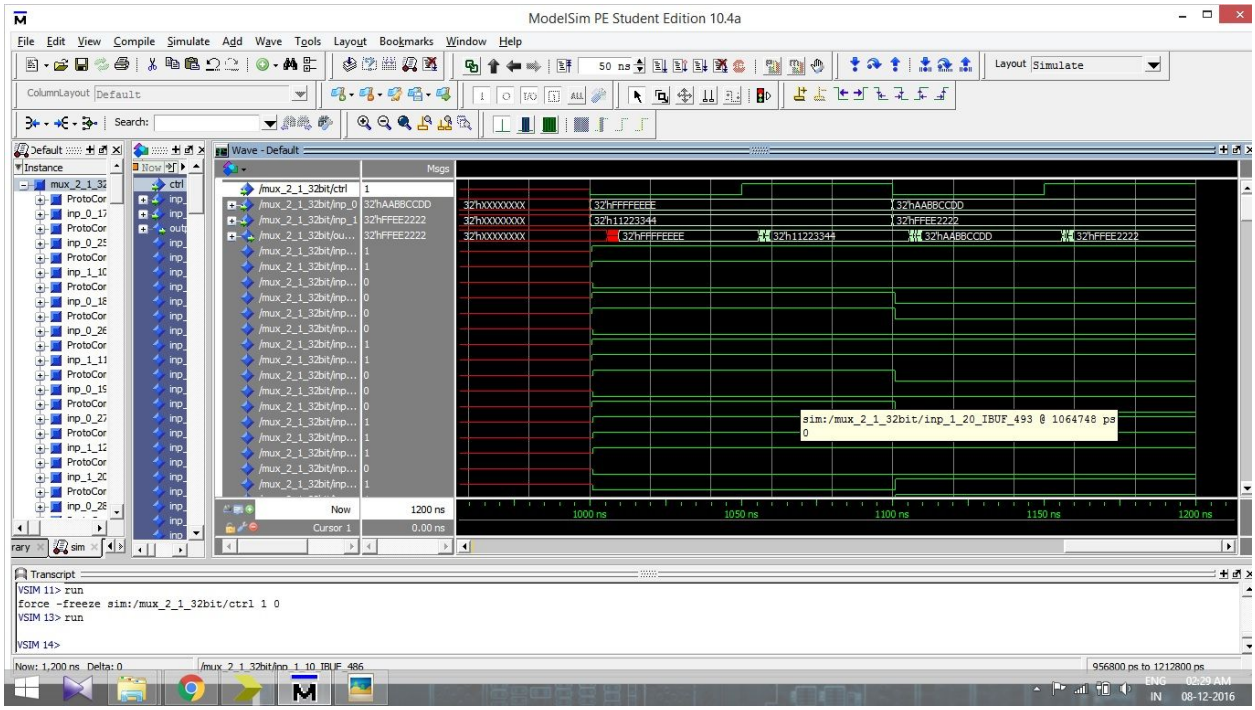
Inp_0	=	FFFFEEEE			
Inp_1	=	11223344			
Ctrl	=	0	Ctrl	=	1
Output	=	FFFFEEEE	Output	=	11223344

2) Test case 2:

Inp_0	=	AABBCCDD			
Inp_1	=	FFEE2222			
Ctrl	=	0	Ctrl	=	1
Output	=	AABBCCDD	Output	=	FFEE2222



Functional simulation of MUX 32 bit 2x1



Timing simulation of MUX 32 bit 2x1

MUX 32 bit 4x1:

One more 32 bit MUX is designed with 4x1 selection. Here, four different inputs are given, and the output is equal to one of the inputs depending on the control input. The timing and functional simulations are carried out for two test cases as shown below.

1) Test case 1:

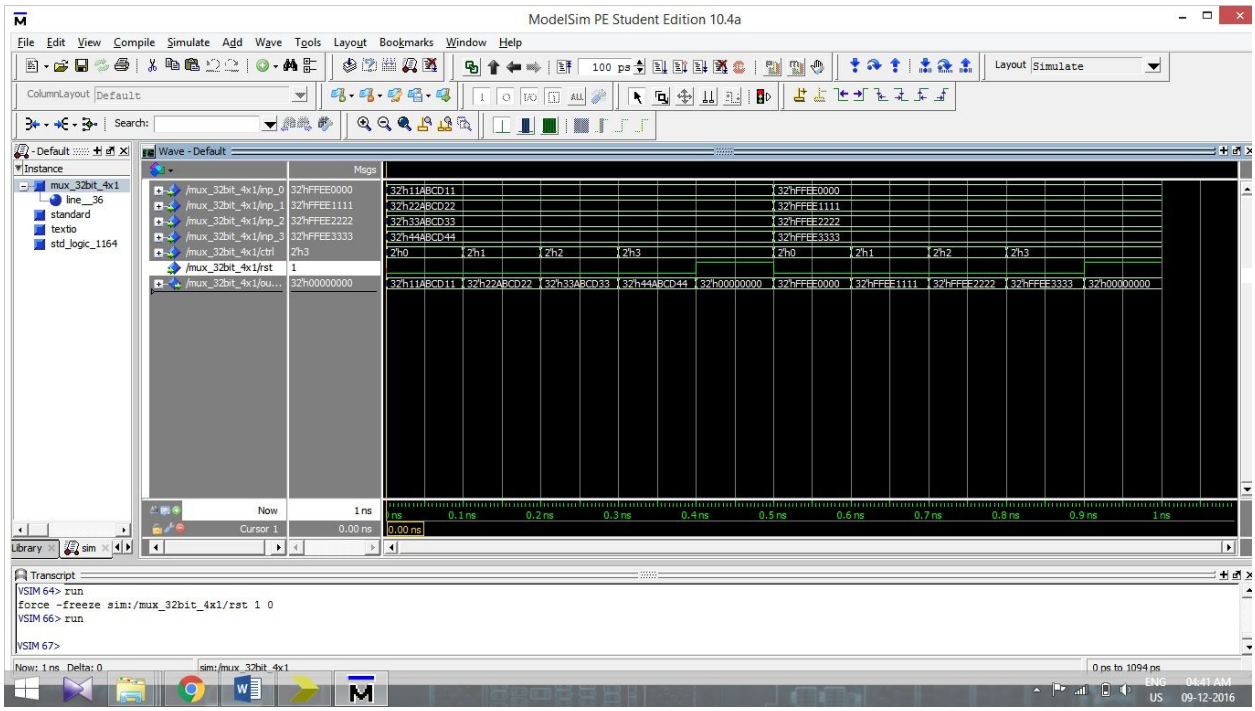
Inp_0 = 11ABCD11
Inp_1 = 22ABCD22
Inp_2 = 33ABCD33
Inp_3 = 44ABCD44
Reset = 0

Ctrl	=	0	output	=	11ABCD11
Ctrl	=	1	output	=	22ABCD22
Ctrl	=	2	output	=	33ABCD33
Ctrl	=	3	output	=	44ABCD44
Reset	=	1	output	=	00000000

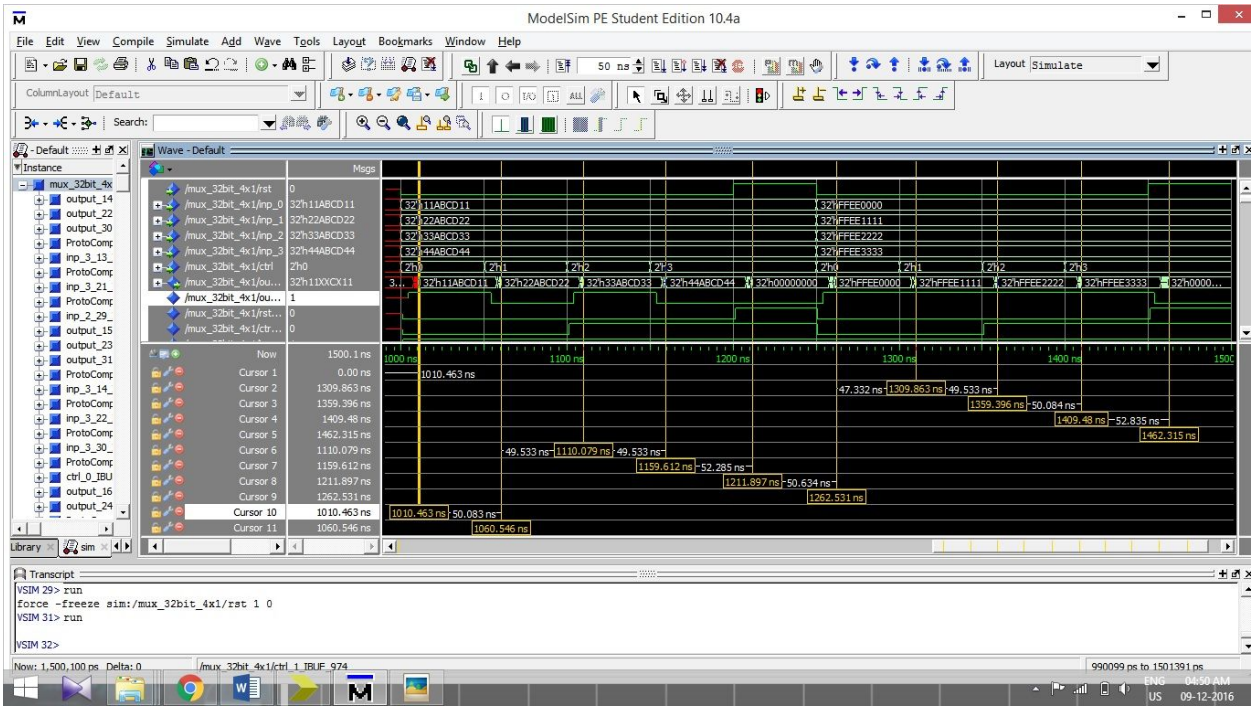
2) Test case 1:

Inp_0 = FFEE0000
Inp_1 = FFEE1111
Inp_2 = FFEE2222
Inp_3 = FFEE3333
Reset = 0

Ctrl	=	0	output	=	FFEE0000
Ctrl	=	1	output	=	FFEE1111
Ctrl	=	2	output	=	FFEE2222
Ctrl	=	3	output	=	FFEE3333
Reset	=	1	output	=	00000000



Functional simulation of MUX 32 bit 4x1



Timing simulation of MUX 32 bit 4x1

Sign Extend:

Sign_extend is the component that is designed specially. It accepts one input, 16 bits wide, and gives one output. The output is 32 bits wide and the output depends on the MSB of 16 bits wide input. If the MSB of the input is '1', then the output is created with all 16 MSBs equal to '1', or 'FFFF' in hexadecimal form. If the MSB of the input is '0', then the output will be same as input, with 16 MSBs equal to '0'. The simulations, both functional and timing are carried out, with three different inputs as follows.

1) Test case 1:

Input = h`ABCD (1010101111001101)

Output = h`FFFFABCD

Here, the MSB of the 16 bit input is '1'. So, the MSBs of the output becomes '1' and the output thus becomes "FFFFABCD" in hexadecimal.

2) Test case 2:

Input = h`4321 (0100001100100001)

Output = h`00004321

Here, the MSB of the 16 bit input is '0'. So, the MSBs of the output becomes '0' and the output thus becomes "00004321" in hexadecimal. Note that the output is same as the input, but resembled in 32 bits form.

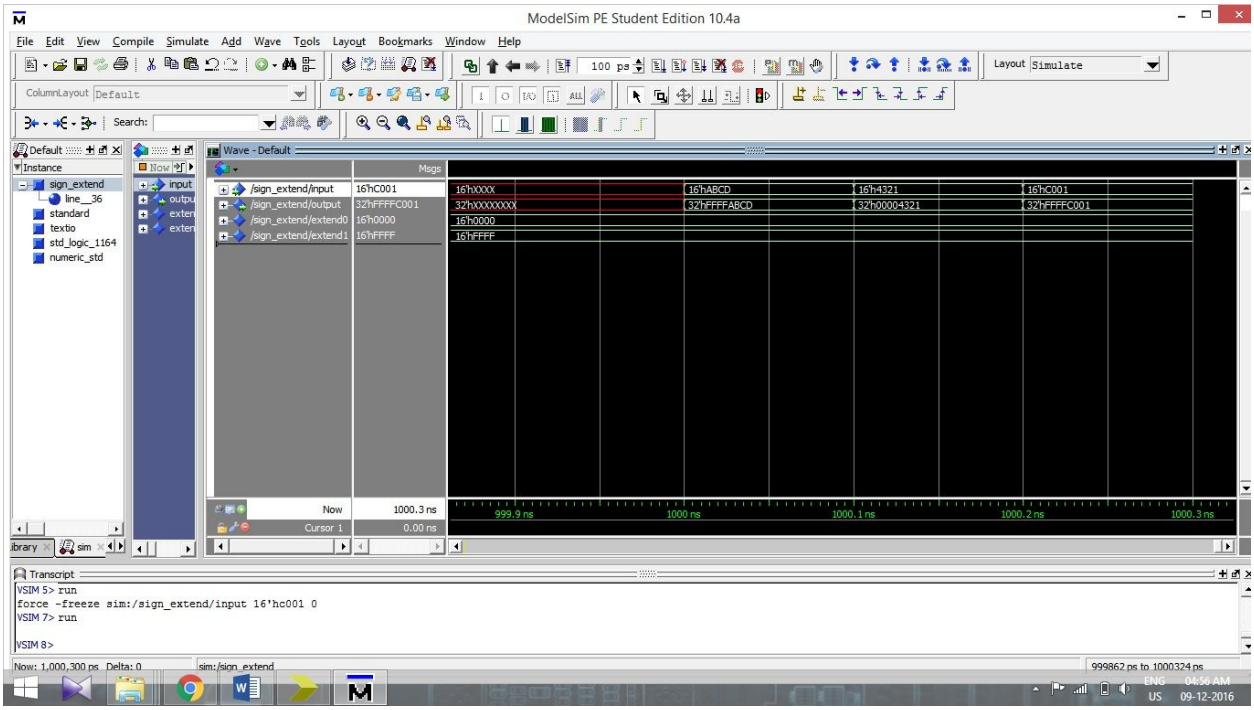
3) Test case 3:

Input = h`C001 (1100000000000001)

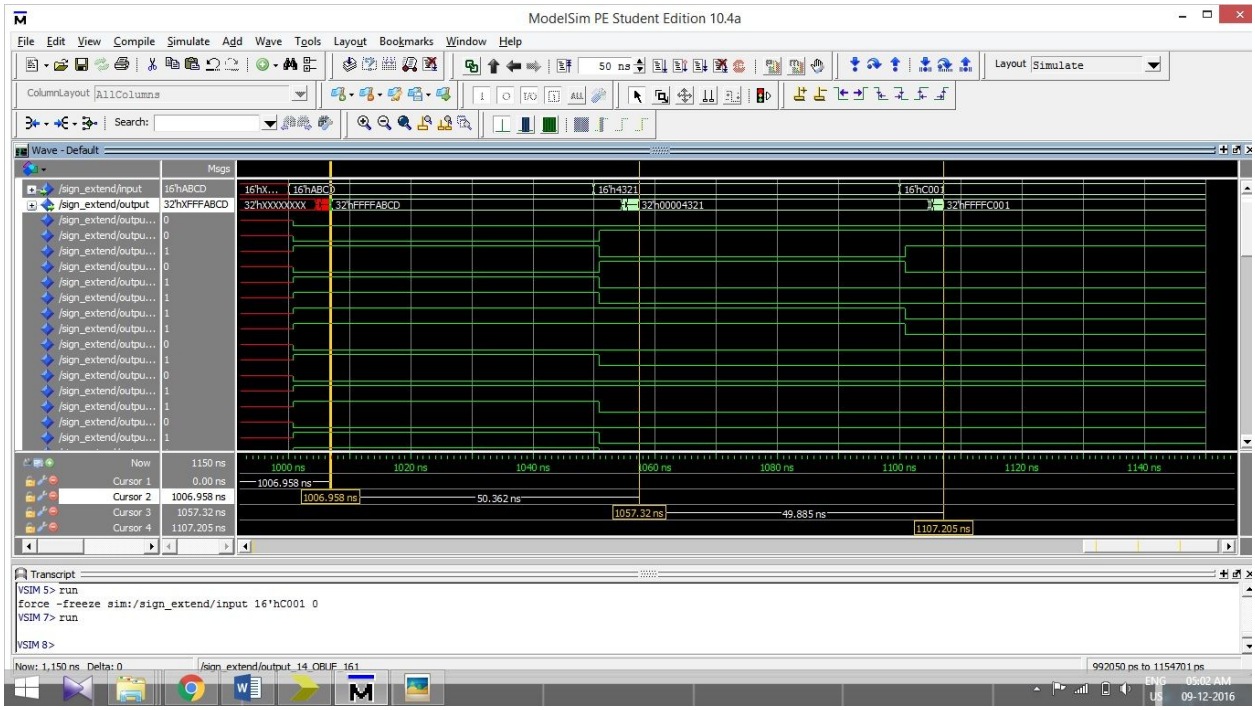
Output = h`FFFFC001

Here, the MSB of the 16 bit input is '1'. So, the MSBs of the output becomes '1' and the output thus becomes "FFFFC001" in hexadecimal.

The functional and the timing simulation are carried out and the screen shots are shown below. The results of both the simulations are same, except in the timing simulation, the results are shown with significant delays. The delays are shown with the help of horizontal bars.



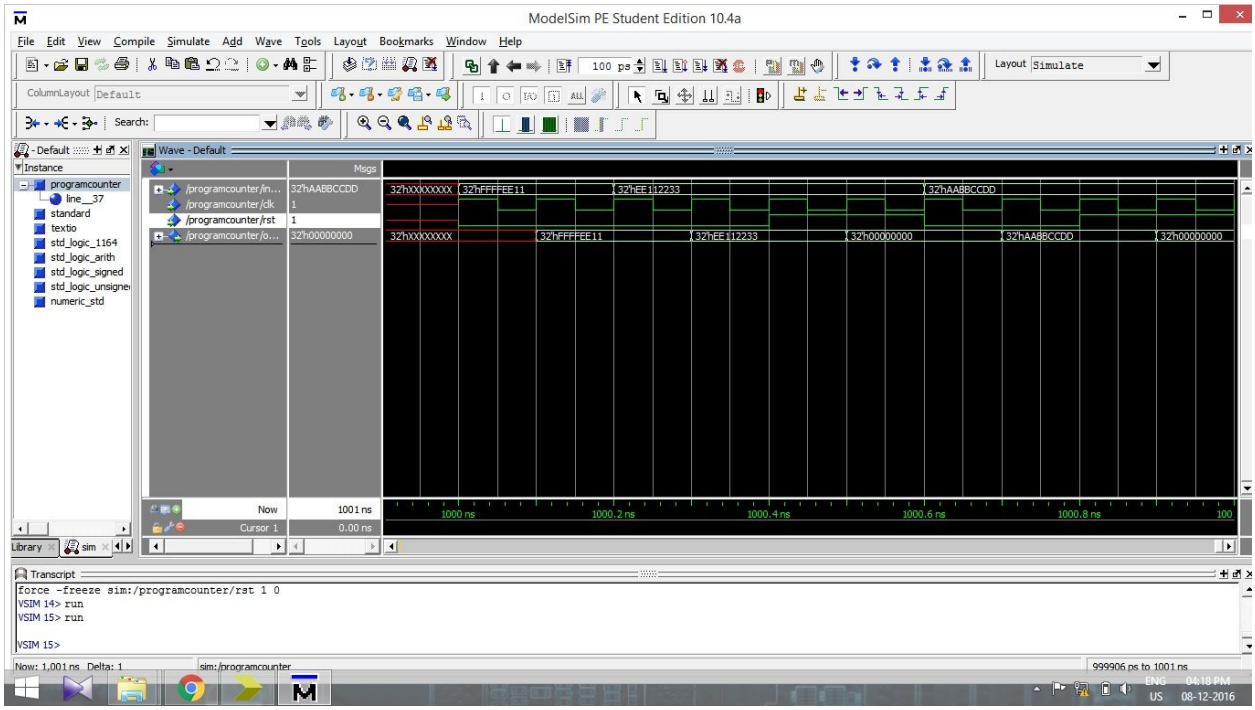
Functional simulation of Sign_extend



Timing simulation of Sign_extend

Program Counter:

In the Program Counter, the input_pc is 32 bits wide, and the output_pc is also 32 bits wide. With each rising edge of the clock, the output_pc shows the value that is equal to the input_pc unless the reset bit (rst) is not equal to '1'. Whenever the rst bit is set to active high or '1', then the value in output_pc is readjust to '00000000'. The simulation of Program counter is done and the screen shot is shown below.



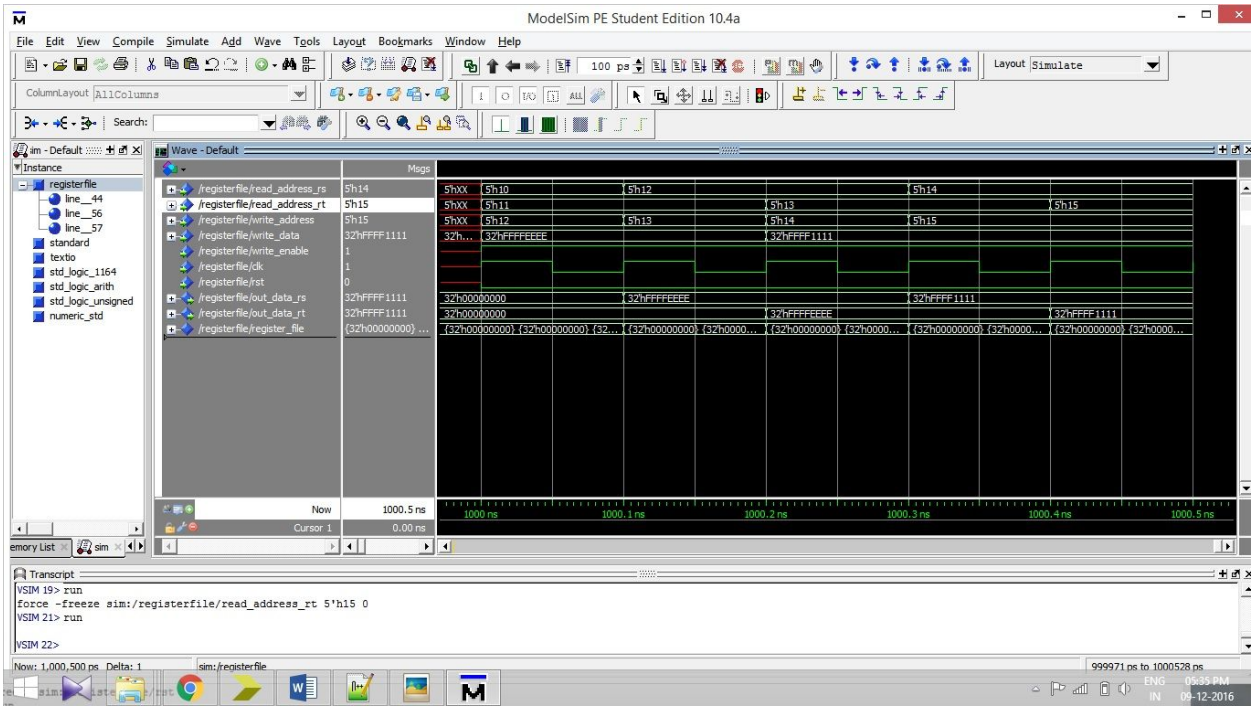
Simulation of Program Counter with 3 different values. Note that the output shows exact value as the input unless rst bit is not '1'. With rst bit becoming '1', the output_pc becomes '00000000'.

Register File:

The register file is designed with two read addresses (read_address_rs and read_address_rt) and a write address as inputs. All the three addresses are 5 bits wide. Then, the data to be written, or write_data, which is 32 bits wide, is to be entered as input. This 32 bits wide data is written at the address specified by the write address. Now, the output can be read by changing the read addresses in the inputs. Note that the conditions are set for the data to be written. The write enabled bit should be active high or set to '1' to make the writing operation. Also note that all the operations are being done with the positive cycle of the clock.

Simulation:

For the simulation, the read_address_rs and read_address_rt were set to h`10 and h`11 respectively and the write_address was set to h`12. The write_data was set to h`FFFFEEEE. Thus, with the positive cycle of the clock, the data, h`FFFFEEEE would be written on the address h`12. Now, for the next cycle, the read_address_rs is set to h`12 and the write_address is set to h`13. So the output_data_rs will show the data h`FFFFEEEE written on the address h`12 and the same data will be written on the new address h`13. Further, the read_address_rt is set to h`13 and the data h`FFFFEEEE is to be written on the new address h`14. So the output_data_rt will show the data h`FFFFEEEE and the new data h`FFFF1111 is written on the new address h`14. For the next cycle, the read_address_rs is set to h`14, and the write_address is set to h`15. Thus, the output_data_rs will be equal to the new data h`FFFF1111 and data is written on the address h`15. Finally, the read_address_rt is set to h`15 and for the next cycle, the output_data_rt will resemble the data h`FFFF1111 from the address h`15.



Functional simulation for the Register File. Note the changes in the output data, when the input_read_addresses and the write_address are changed.

3.2 Simulation of ALU

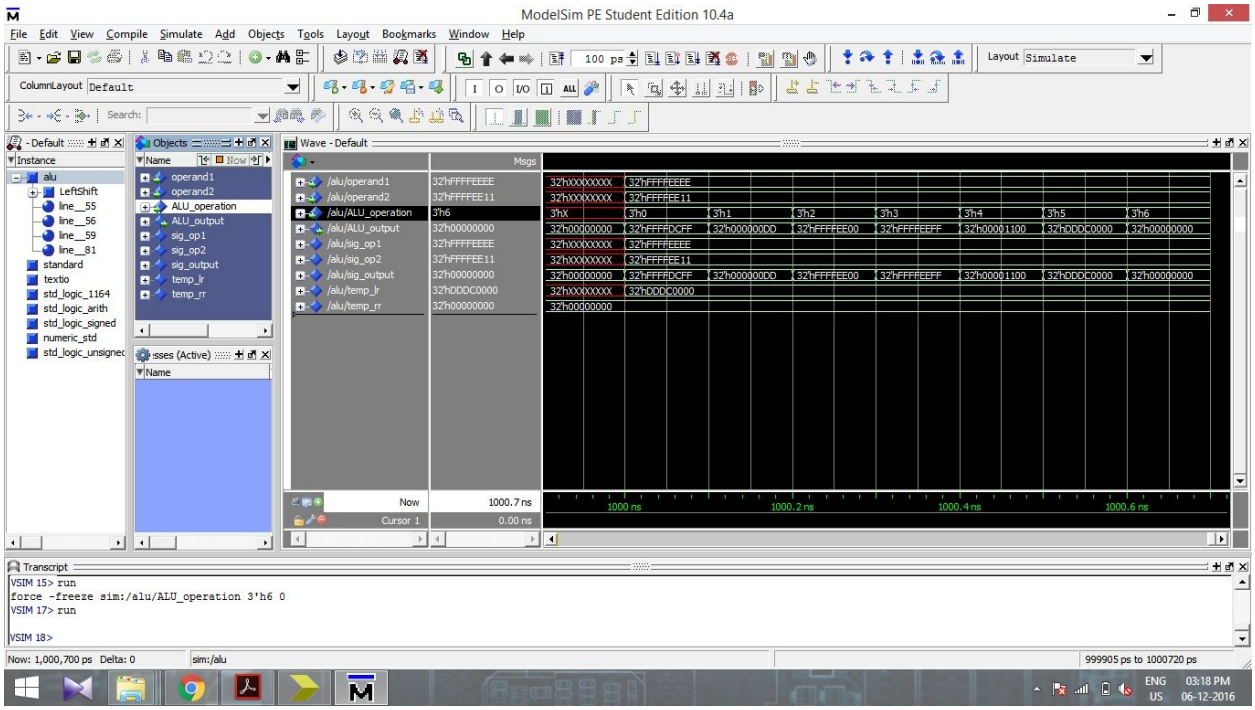
3.2.1 Functional Simulation

Test case 1:

Operand 1 : FFFFEEEE

Operand 2 : FFFFEE11

ALU_operation	: 0	ALU_output	: FFFFDCFF
ALU_operation	: 1	ALU_output	: 000000DD
ALU_operation	: 2	ALU_output	: FFFFEE00
ALU_operation	: 3	ALU_output	: FFFFEEFF
ALU_operation	: 4	ALU_output	: 00001100
ALU_operation	: 5	ALU_output	: DDDC0000
ALU_operation	: 6	ALU_output	: 00000000

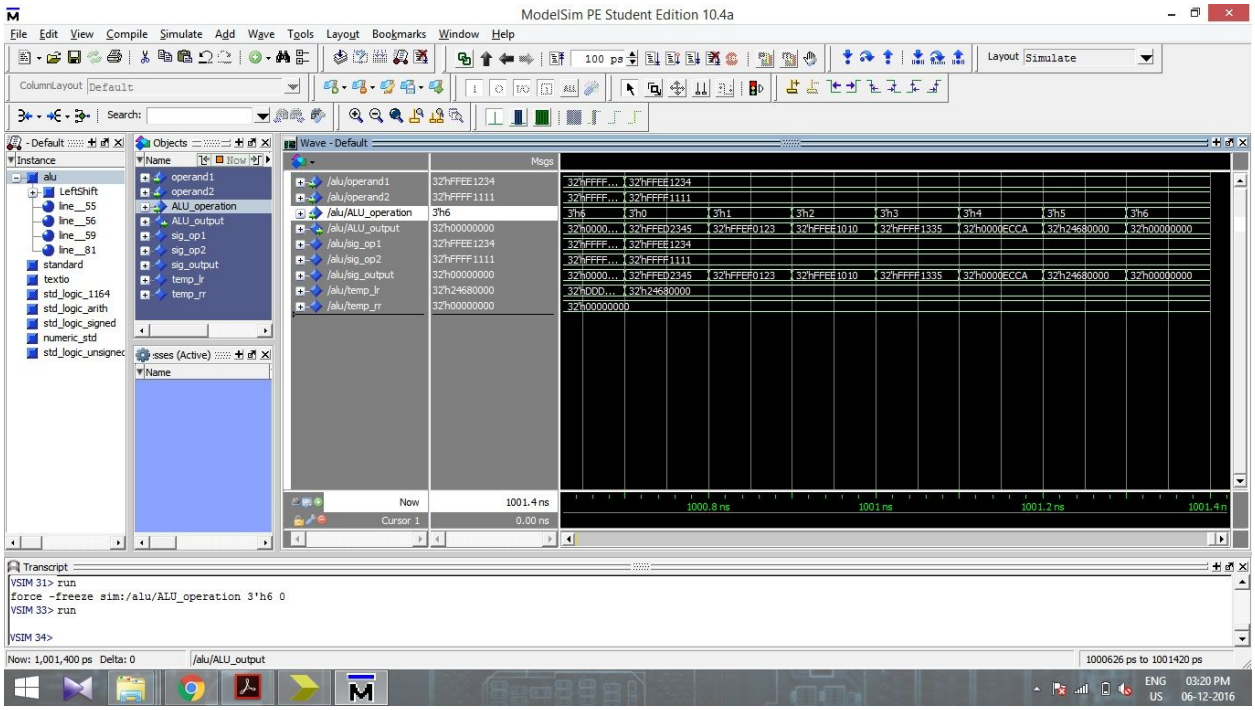


Test case 2:

Operand 1 : FFEE1234

Operand 2 : FFFF1111

ALU_operation	: 0	ALU_output : FFED2345
ALU_operation	: 1	ALU_output : FFEF0123
ALU_operation	: 2	ALU_output : FFEE1010
ALU_operation	: 3	ALU_output : FFFF1335
ALU_operation	: 4	ALU_output : 0000ECCA
ALU_operation	: 5	ALU_output : 24680000
ALU_operation	: 6	ALU_output : 00000000

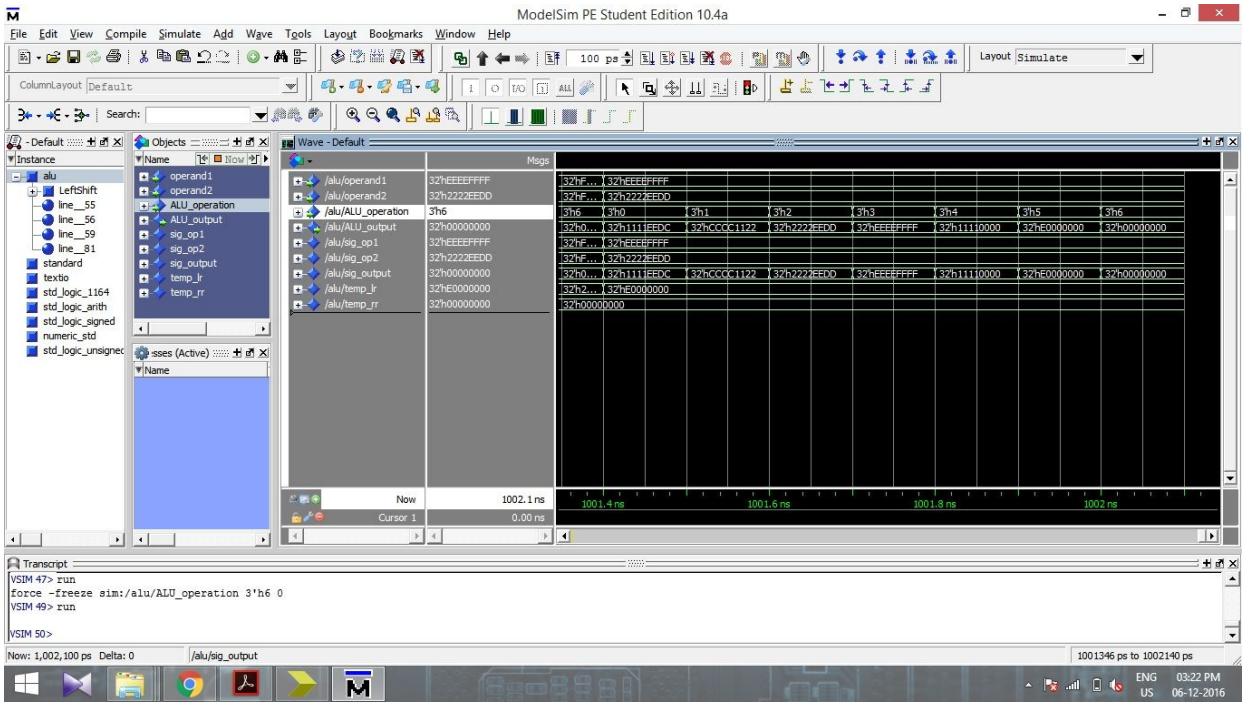


Test case 3:

Operand 1 : EEEEFFFF

Operand 2 : 2222EEDD

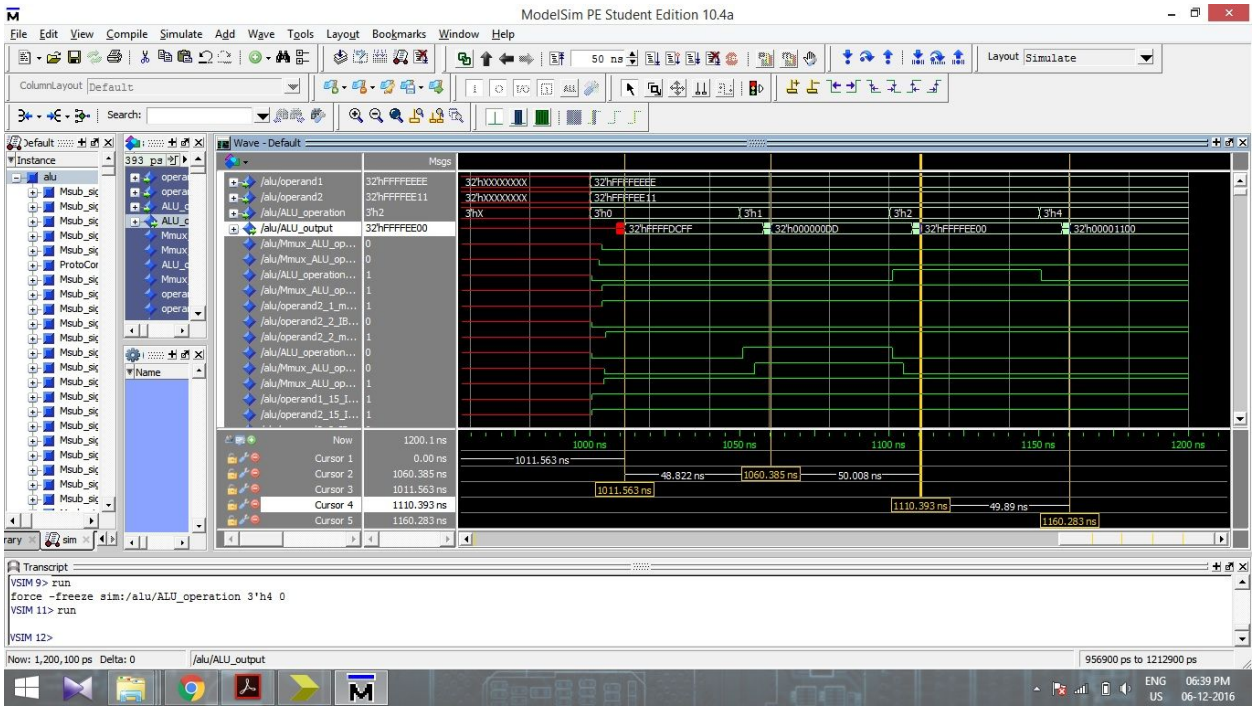
ALU_operation	: 0	ALU_output	: 1111EEDC
ALU_operation	: 1	ALU_output	: CCCC1122
ALU_operation	: 2	ALU_output	: 2222EEDD
ALU_operation	: 3	ALU_output	: EEEEFFFF
ALU_operation	: 4	ALU_output	: 11110000
ALU_operation	: 5	ALU_output	: 00000000
ALU_operation	: 6	ALU_output	: 00000000



3.2.2 Timing Simulation

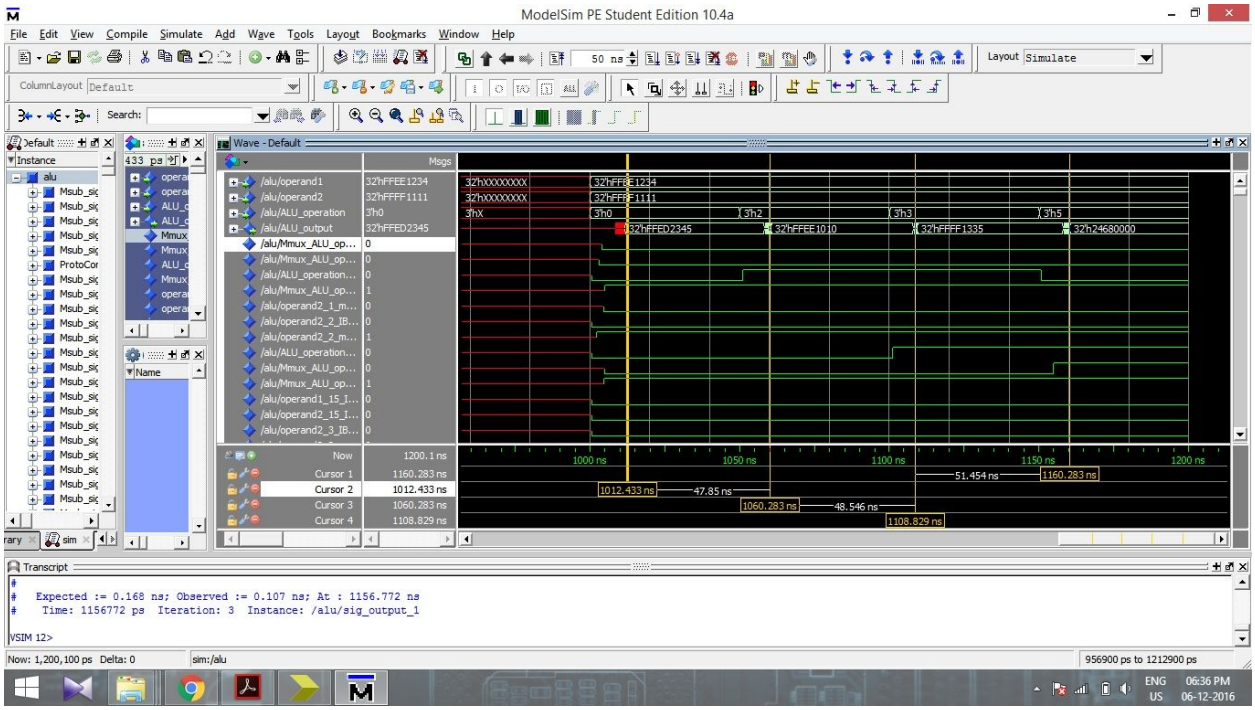
The timing simulations are carried for the same inputs for which Functional simulations were carried out. The inputs (operand values) and the respective simulation screenshots for three test cases are shown below.

Test case 1: Operand 1 : FFFFEEEE
Operand 2 : FFFFEE11

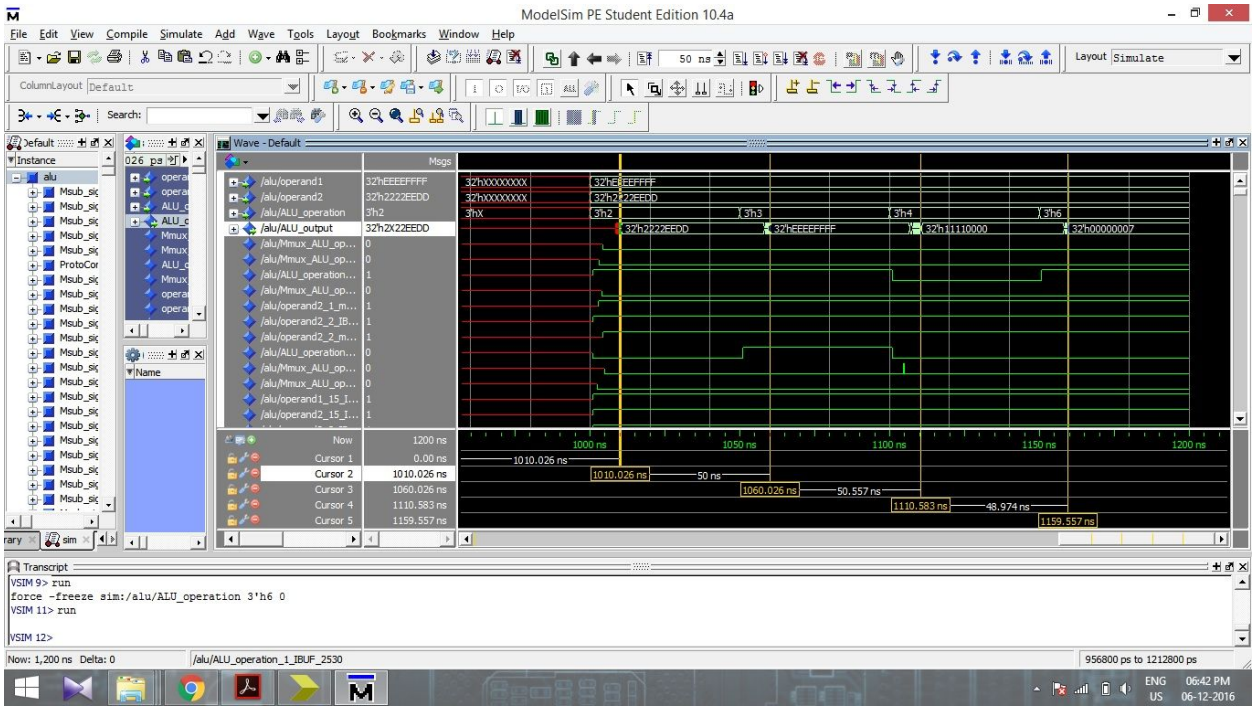


Test case 2: Operand 1 : FFEE1234

Operand 2 : FFFF1111

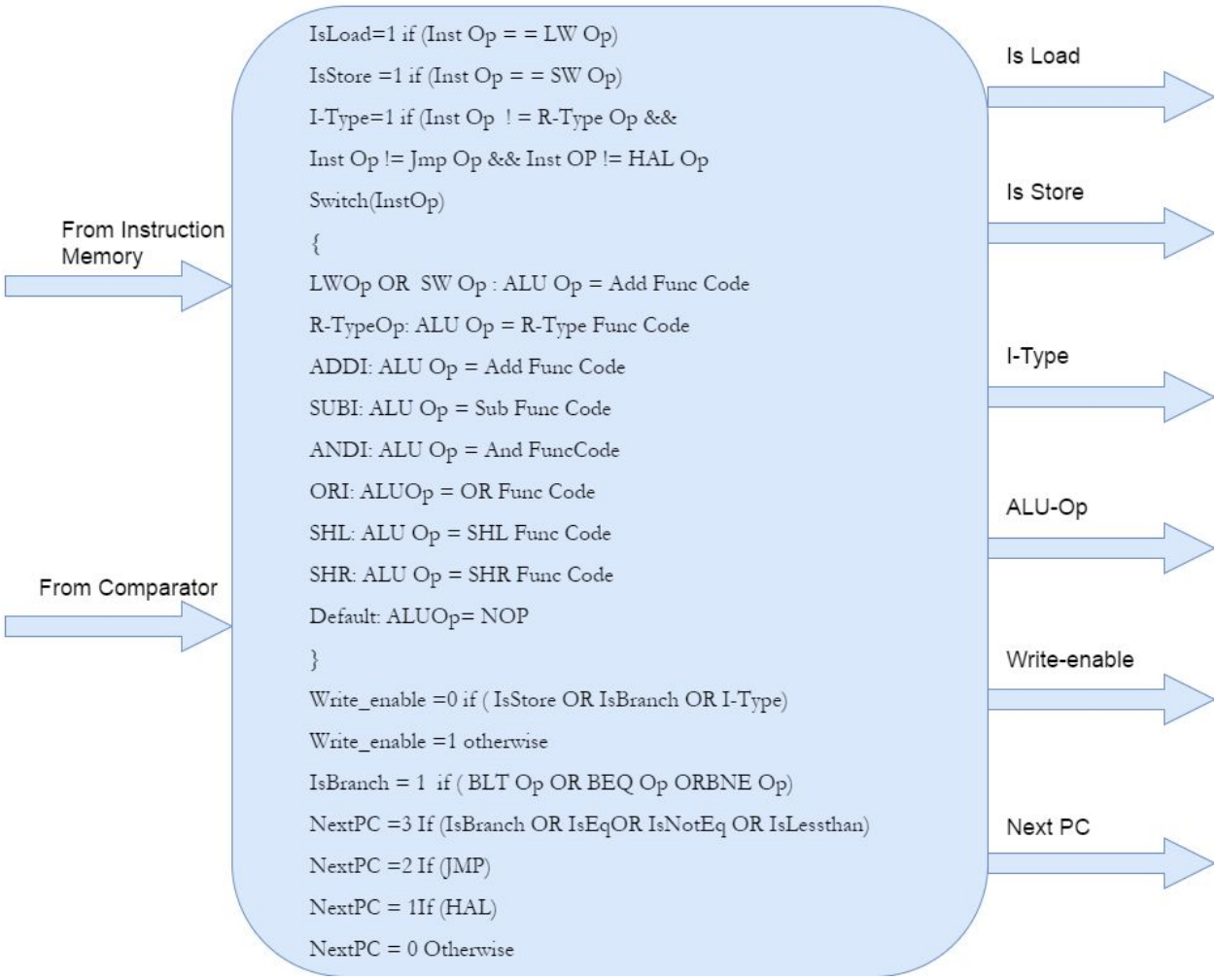


Test case 3: Operand 1 : EEEEFFFF
Operand 2 : 2222EEDD



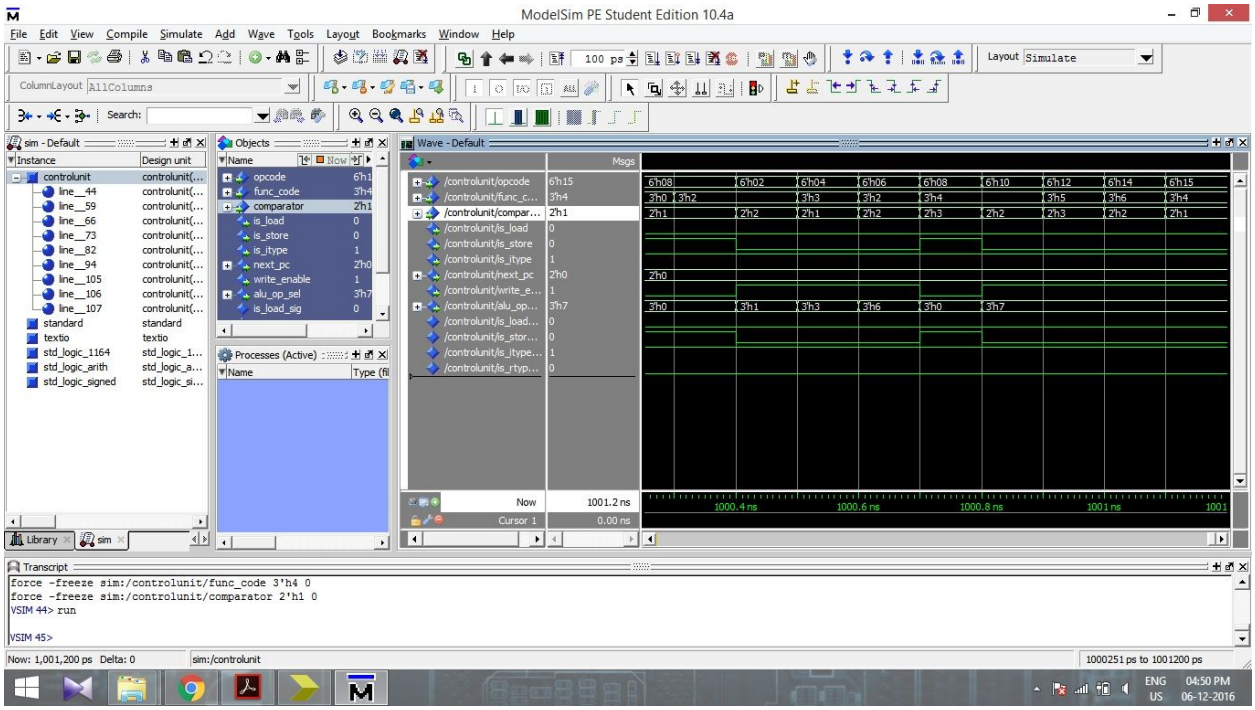
3.3 Simulation of Decoder Unit

The Decoder unit was designed and the design and working algorithm is shown in the block below. We can see two inputs to the decoder, one from Instruction memory and other from the Comparator. The output from the Decoder Unit are used to select the different components like store,load,write enable, etc. The functional and timing simulations were carried out for random values of opcode, func_code and comparator. The results are shown in the screen shots below.

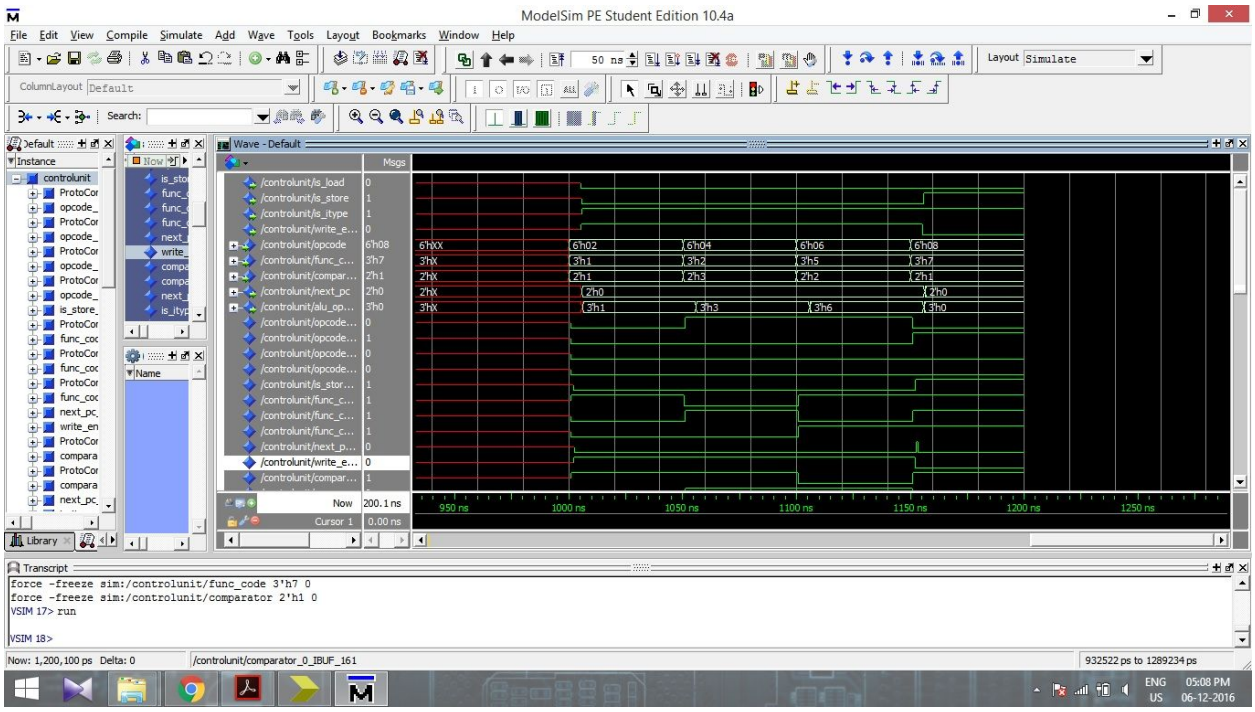


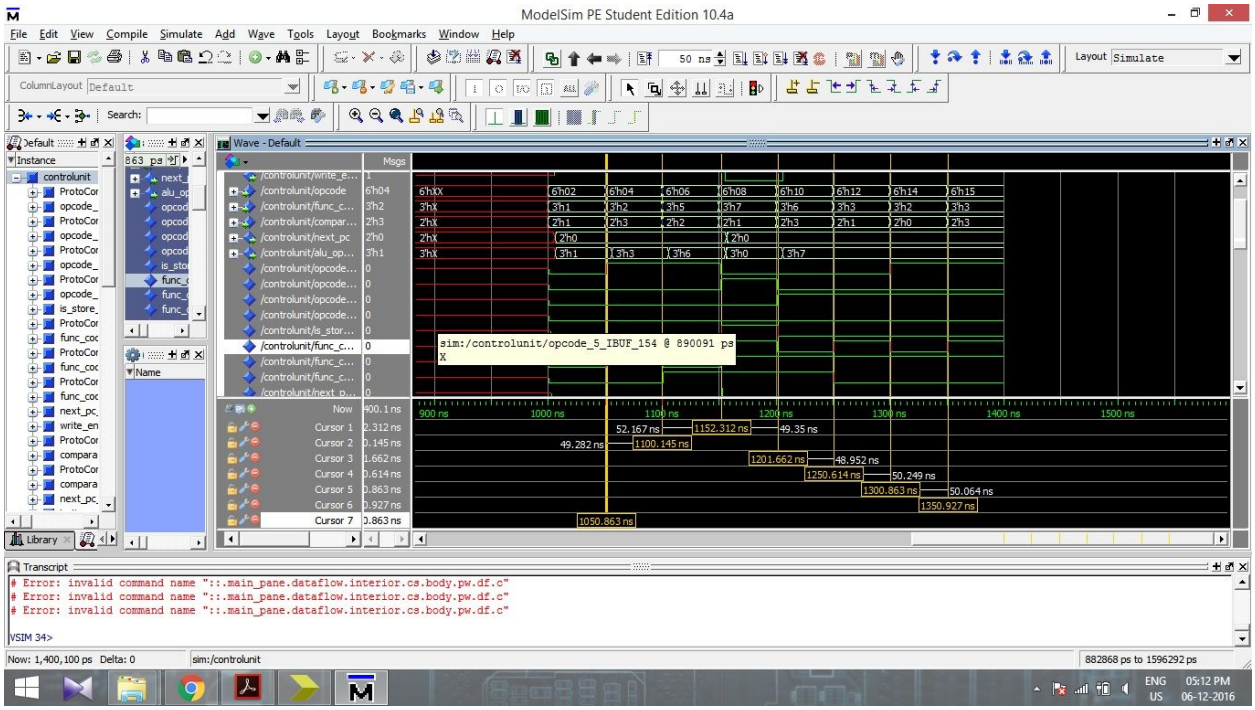
Block Diagram of Decoder Unit

3.3.1 Functional Simulation



3.3.2 Timing Simulations




ModelSim PE Student Edition 10.4a
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Wave - Default
/controlunit/opcode
/controlunit/func_c...
/controlunit/compar...
/controlunit/next_pc
/controlunit/alu_op...
/controlunit/write_e...
/controlunit/opcode...
/controlunit/is_stor...
/controlunit/func_c...
sim:/controlunit/opcode_5_IBUF_154 @ 890091 ps
X
Cursor 1 2.312 ns
Cursor 2 0.145 ns
Cursor 3 1.662 ns
Cursor 4 0.614 ns
Cursor 5 0.863 ns
Cursor 6 0.927 ns
Cursor 7 0.863 ns
52.167 ns
1152.312 ns
49.35 ns
49.282 ns
1100.145 ns
1201.662 ns
48.952 ns
1250.614 ns
50.249 ns
1300.863 ns
50.064 ns
1350.927 ns
1050.863 ns
Transcript
Error: invalid command name "::.main_pane.dataflow.interior.cs.body.pw.df.c"
Error: invalid command name "::.main_pane.dataflow.interior.cs.body.pw.df.c"
Error: invalid command name "::.main_pane.dataflow.interior.cs.body.pw.df.c"
VSIM 34>
Now: 1,400,100 ps Delta: 0
sim:/controlunit
882868 ps to 1596292 ps
ENG US
05:12 PM
06-12-2016

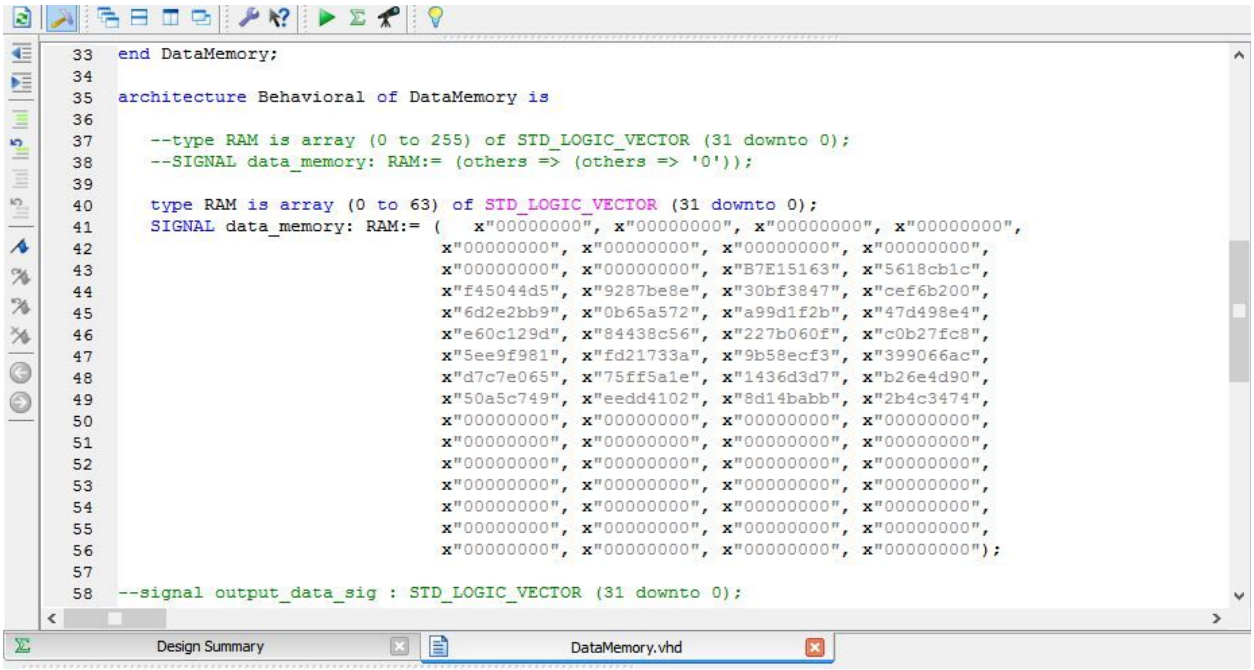
4 Simulation of Complete Design

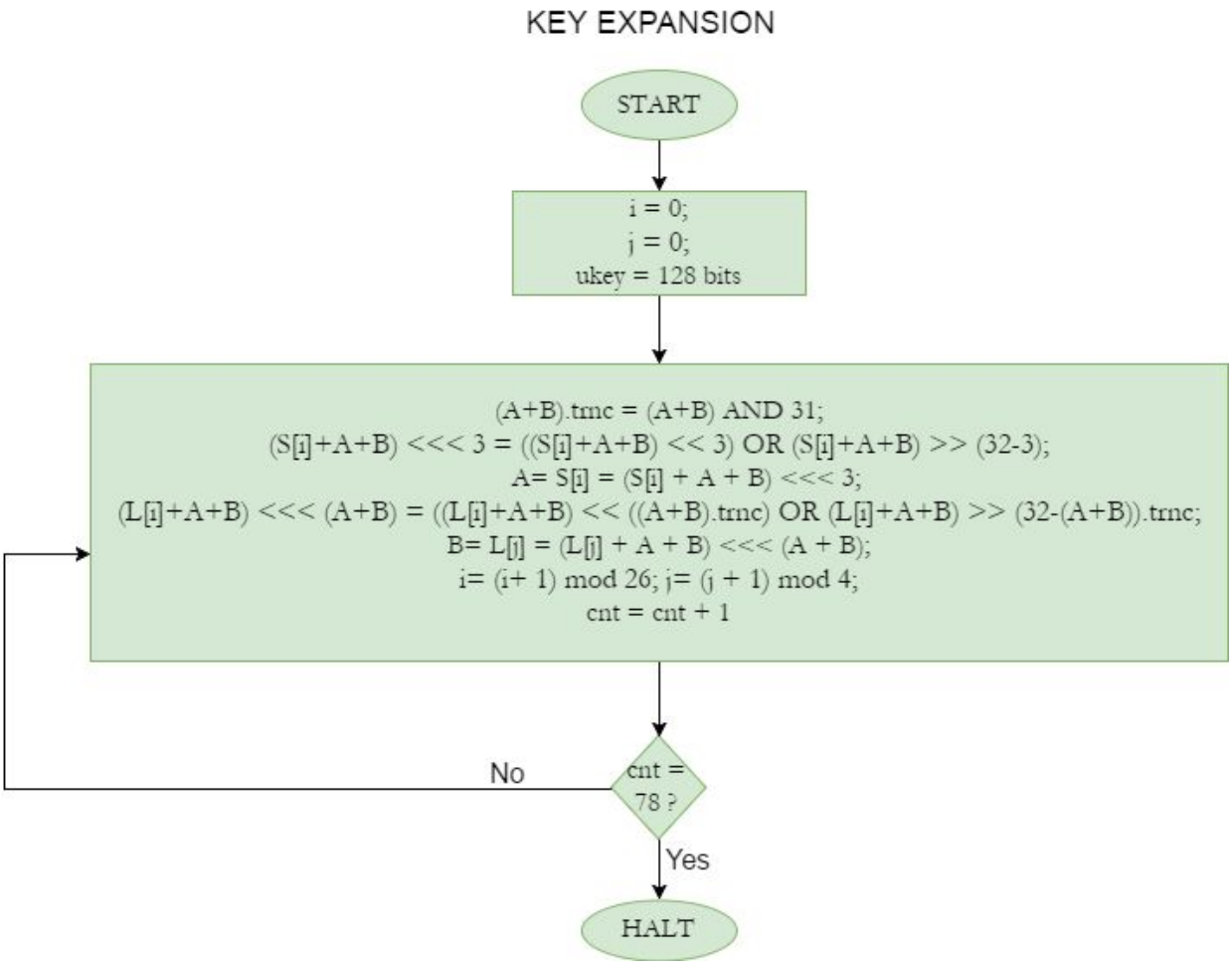
4.0 High Level Description of Implementation of RC5

The implementation of RC5 has been carried out successfully on the designed processor. All the three cycles of RC5 were implemented successfully. The algorithm of all three parts of RC5 is shown in the form of flow chart. The following sections includes the highly descriptive algorithmic flowcharts, for all the sections of RC5, including Encryption, Decryption and Key-expansion. Moreover, the initial S-array, as declared in the program, is shown in the screenshot below

4.1 Key Expansion

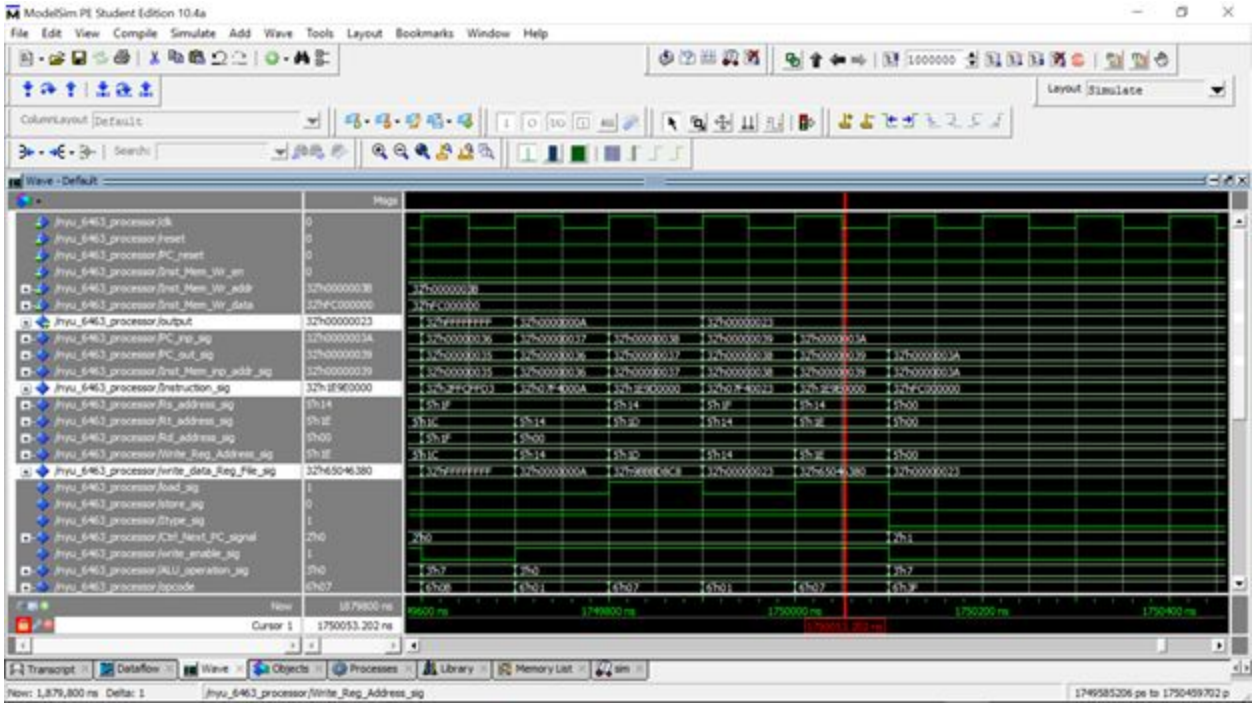
Initial S-array Data Memory



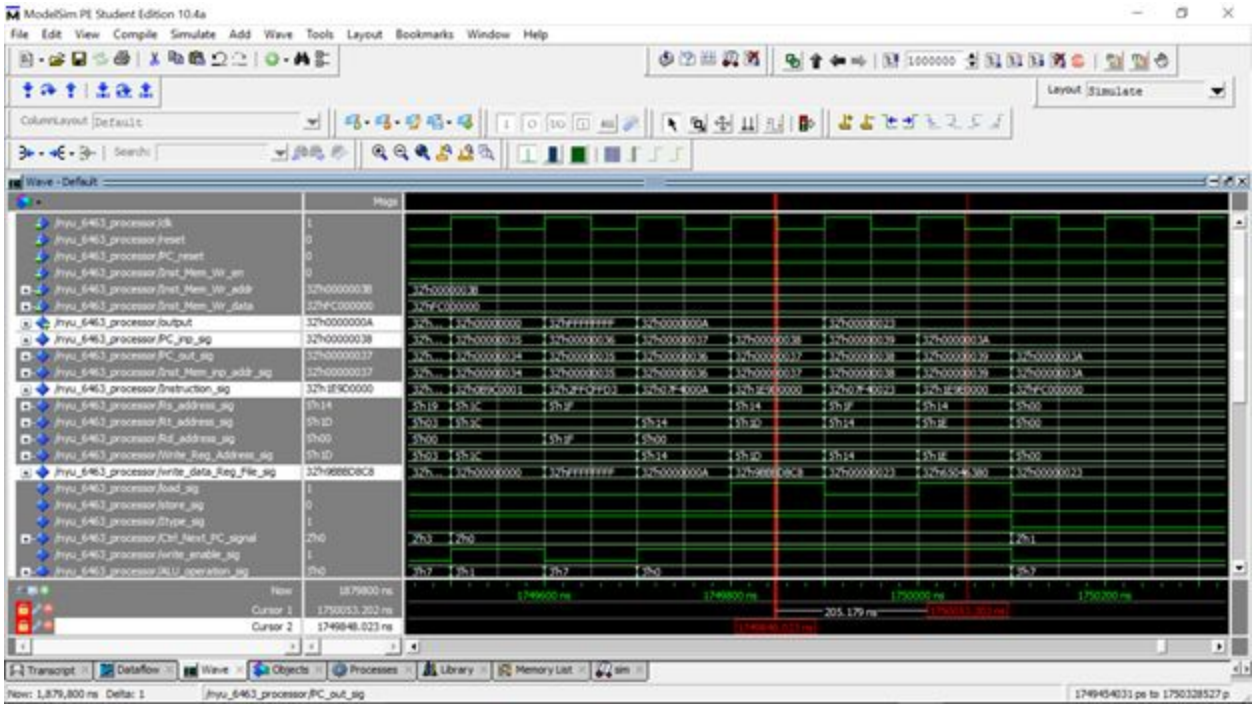

KEY EXPANSION
START
i = 0;
j = 0;
ukey = 128 bits
(A+B).trnc = (A+B) AND 31;
(S[i]+A+B) <<< 3 = ((S[i]+A+B) << 3) OR (S[i]+A+B) >> (32-3);
A= S[i] = (S[i] + A + B) <<< 3;
(L[i]+A+B) <<< (A+B) = ((L[i]+A+B) << ((A+B).trnc) OR (L[i]+A+B) >> (32-(A+B)).trnc;
B= L[j] = (L[j] + A + B) <<< (A + B);
i= (i+ 1) mod 26; j= (j + 1) mod 4;
cnt = cnt + 1
cnt = 78 ?
No
Yes
HALT

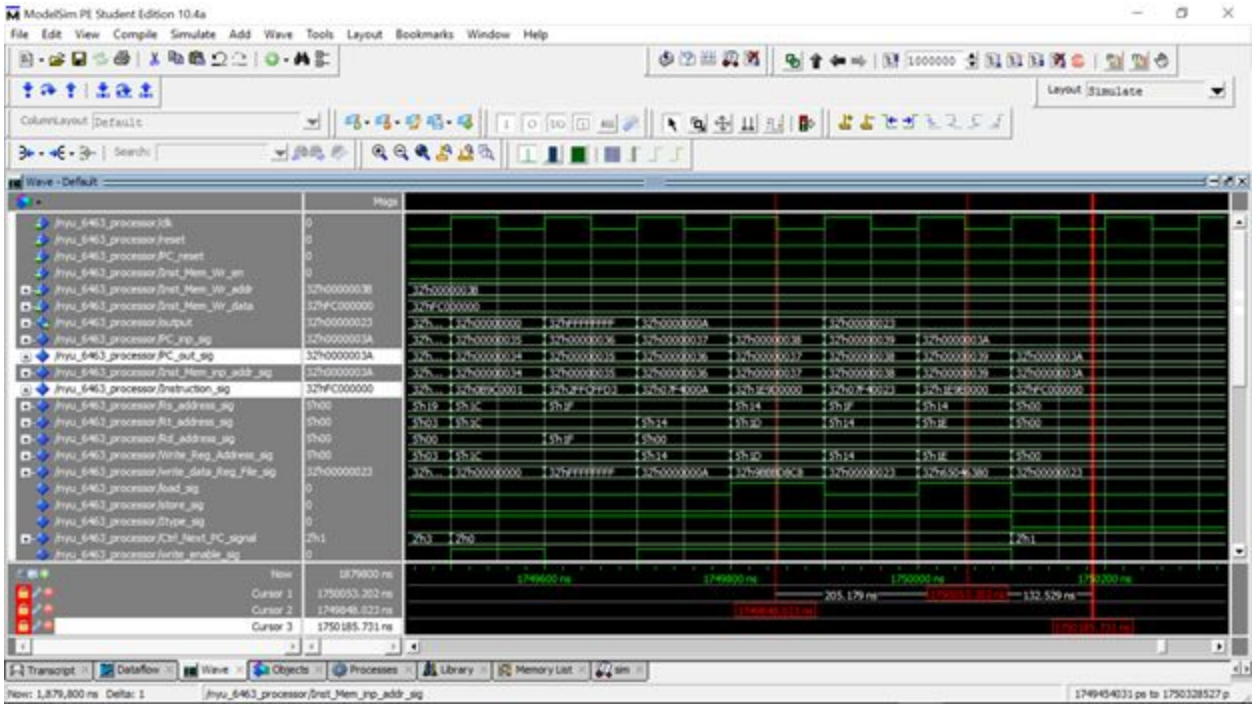
Functional Simulation:

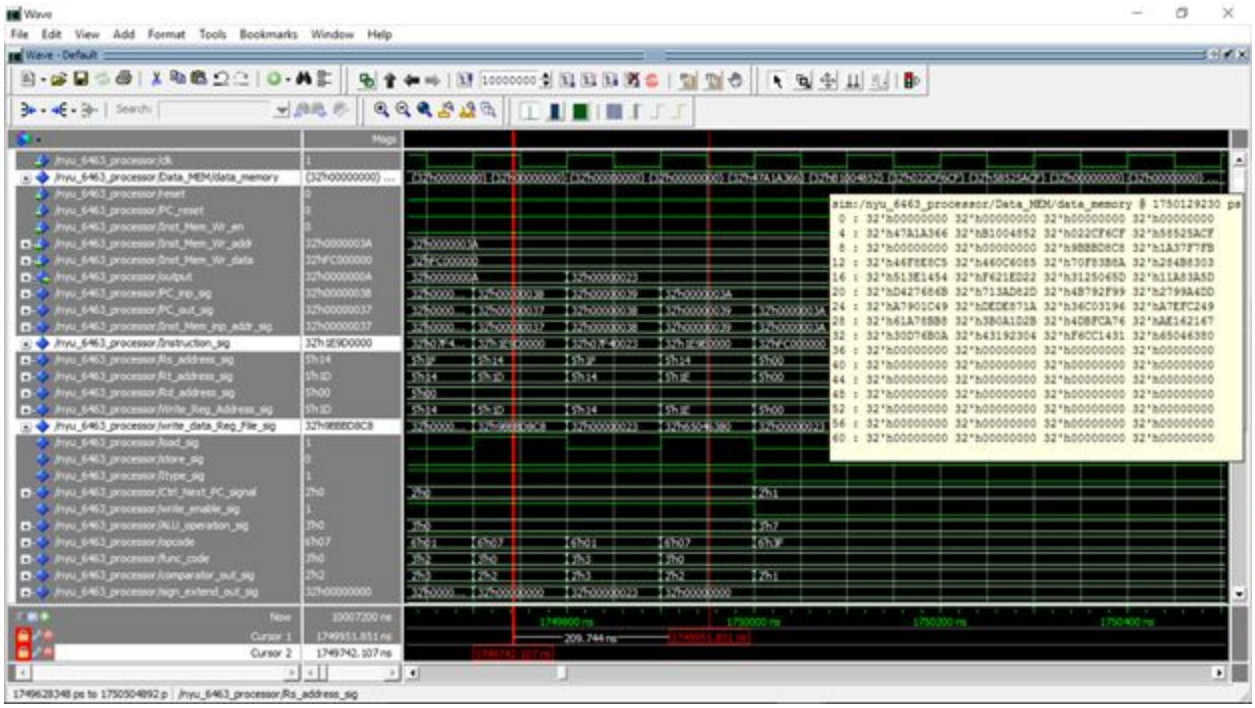
S25



S0

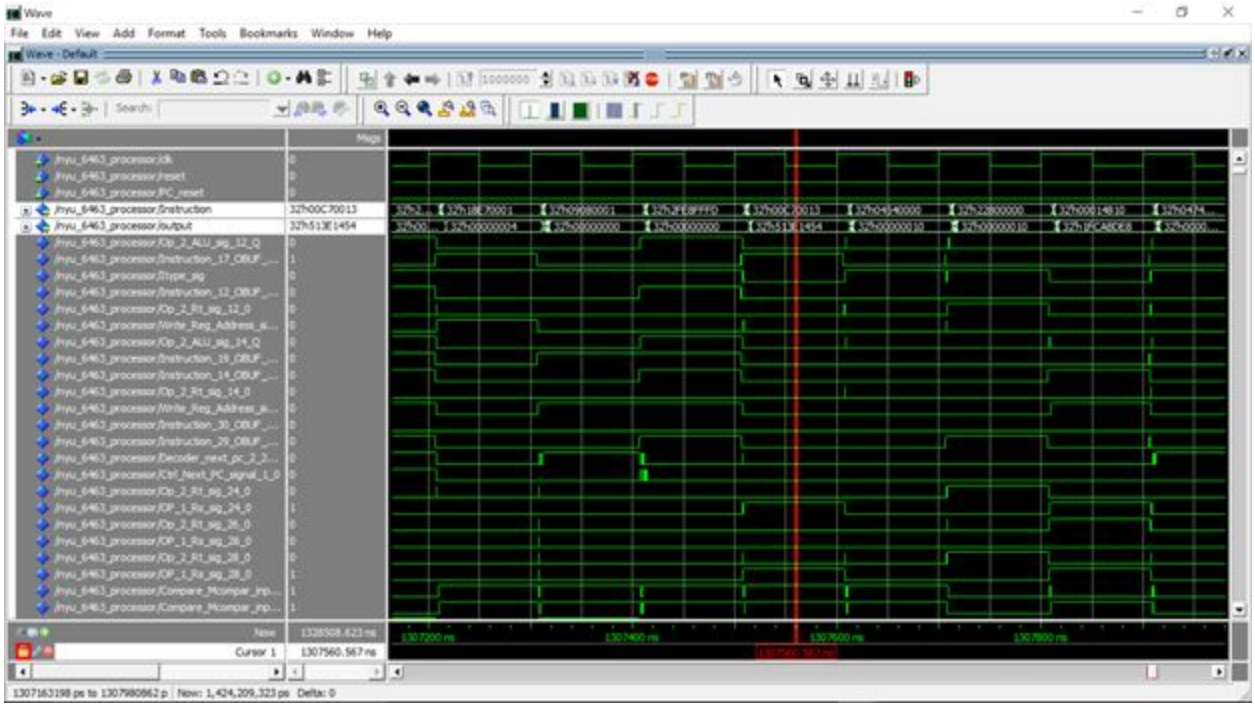


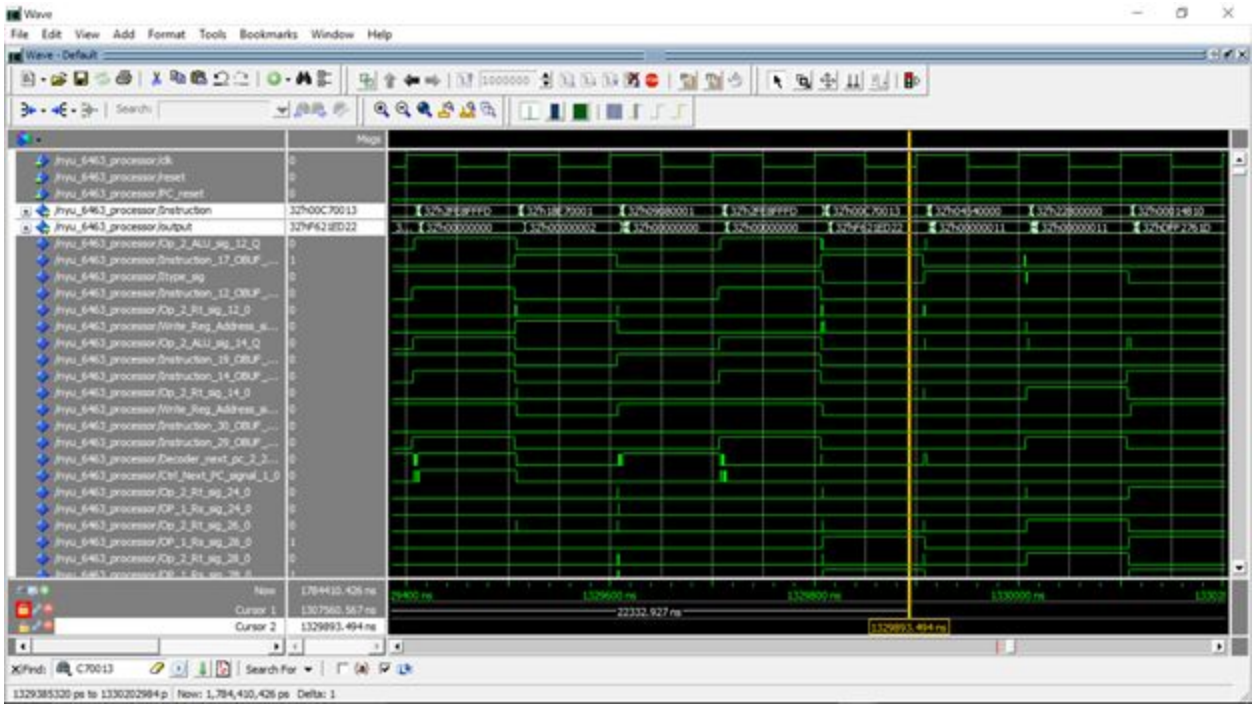

ModelSim PE Student Edition 10.4a
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Wave - Default
Transcript
Dataflow
Wave
Objects
Processes
Library
Memory List
sim


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default

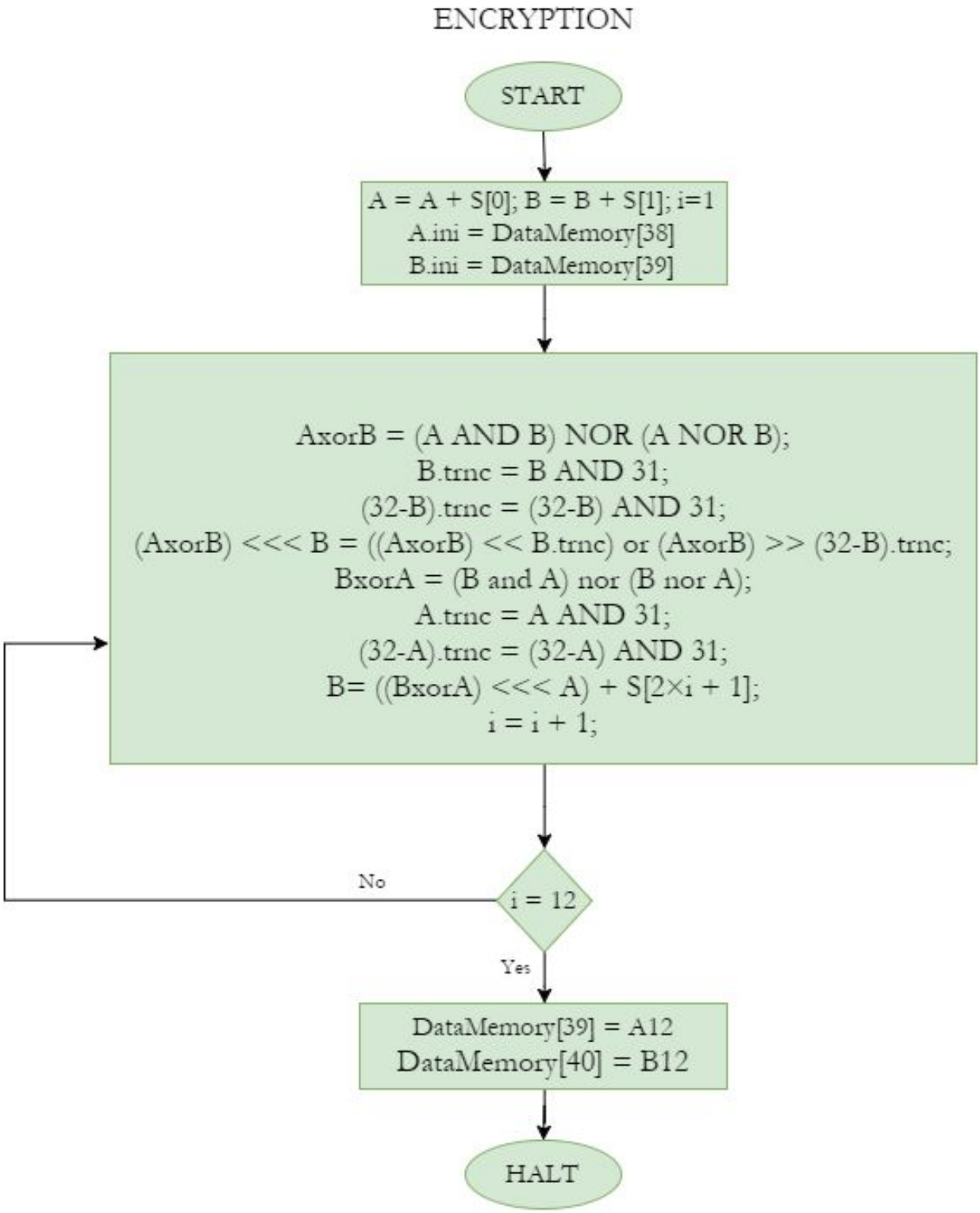
Timing Simulations

Timing simulations were carried and screenshots of intermediate values were captured. The screenshots are shown in the below images.



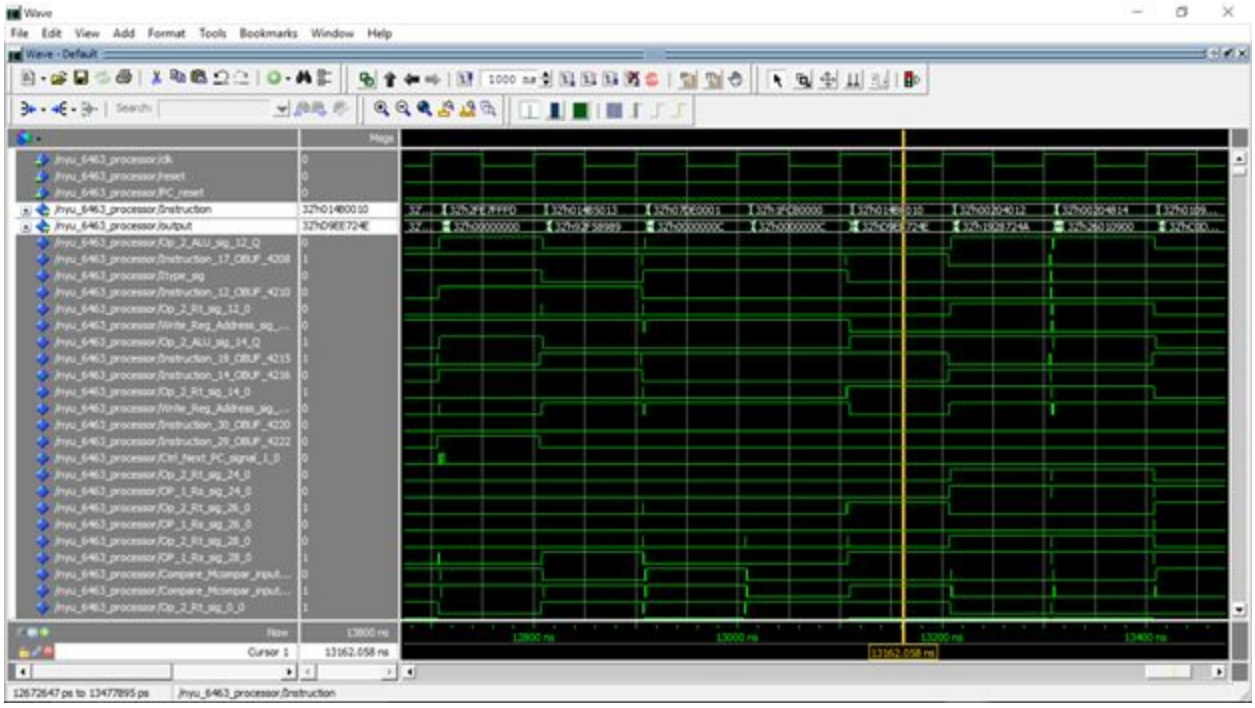


4.2 Encryption

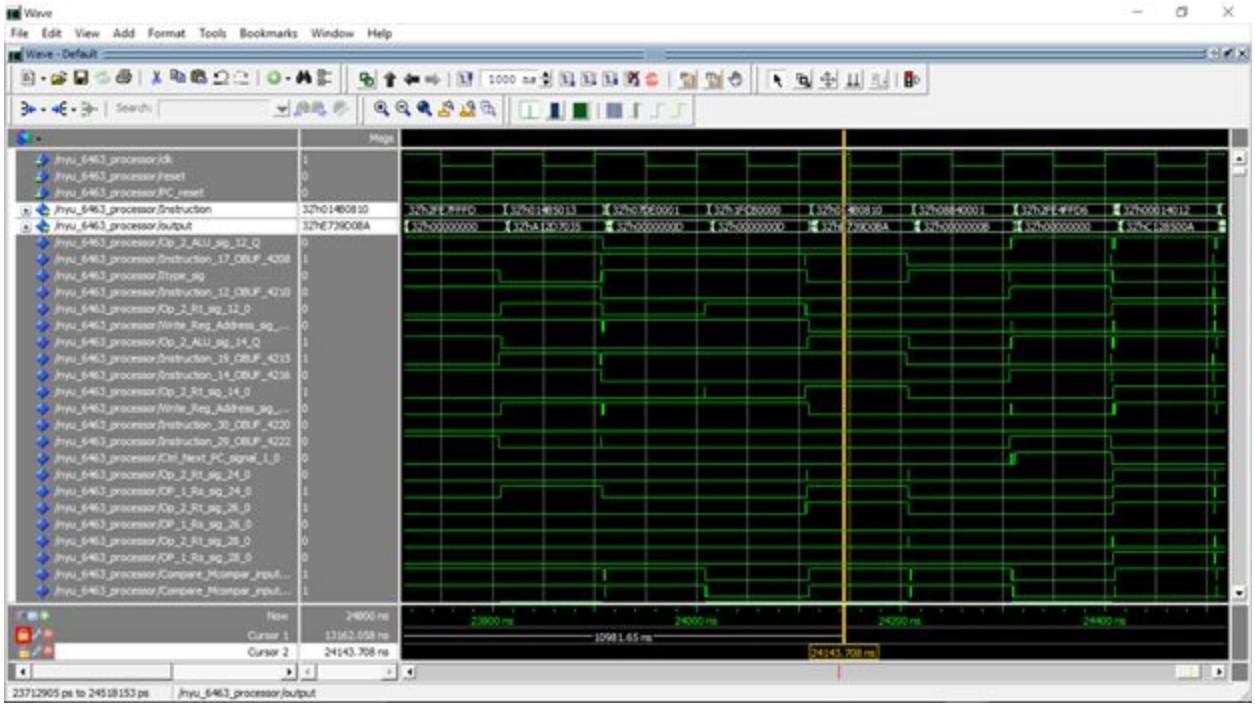


4.2.1 Simulation results

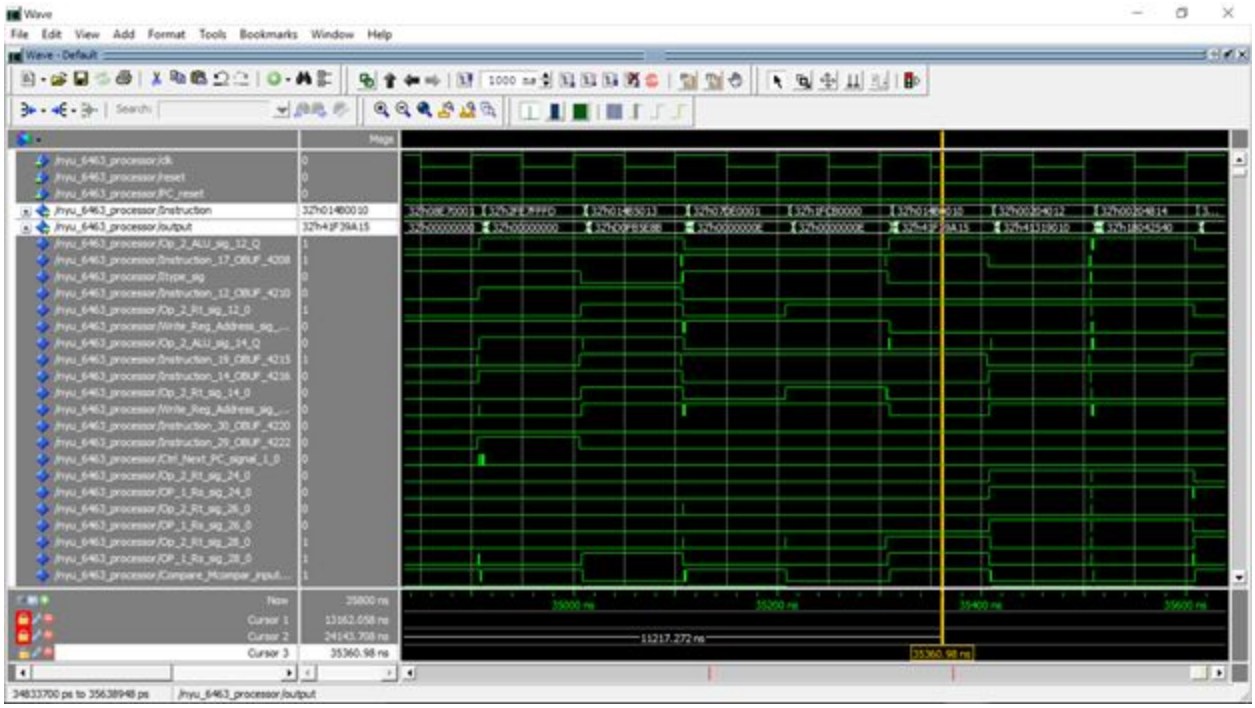
A1



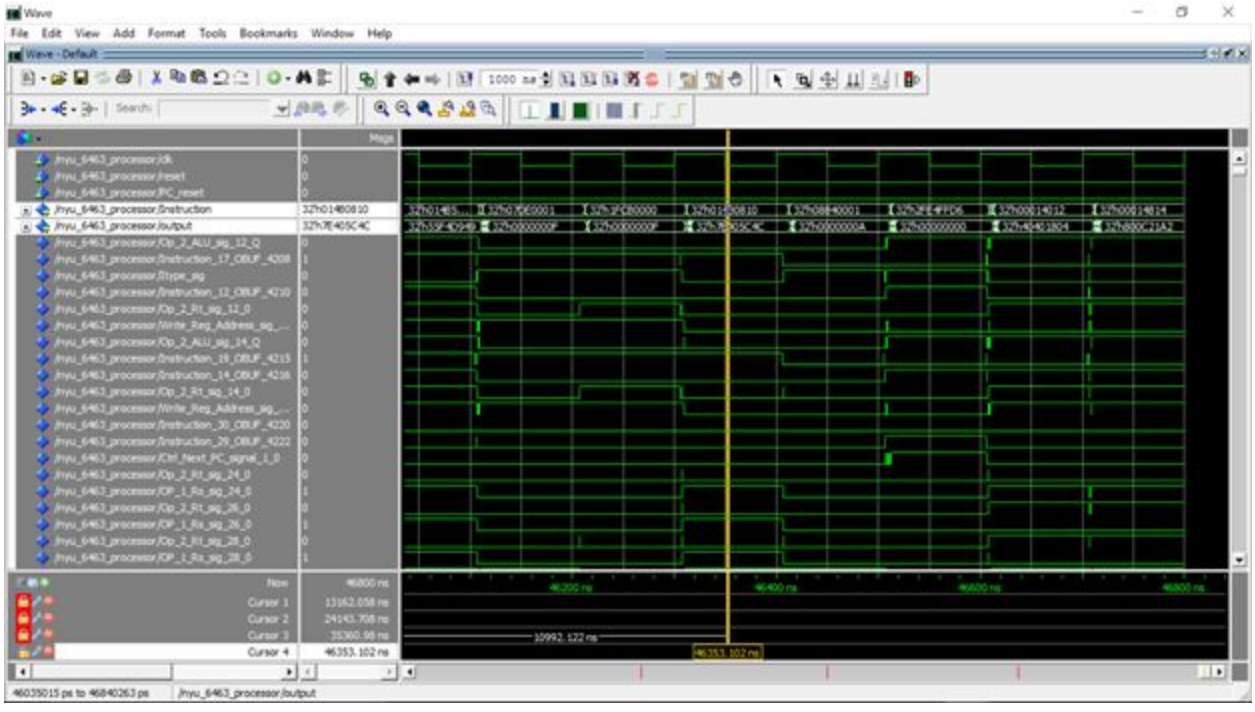
B1



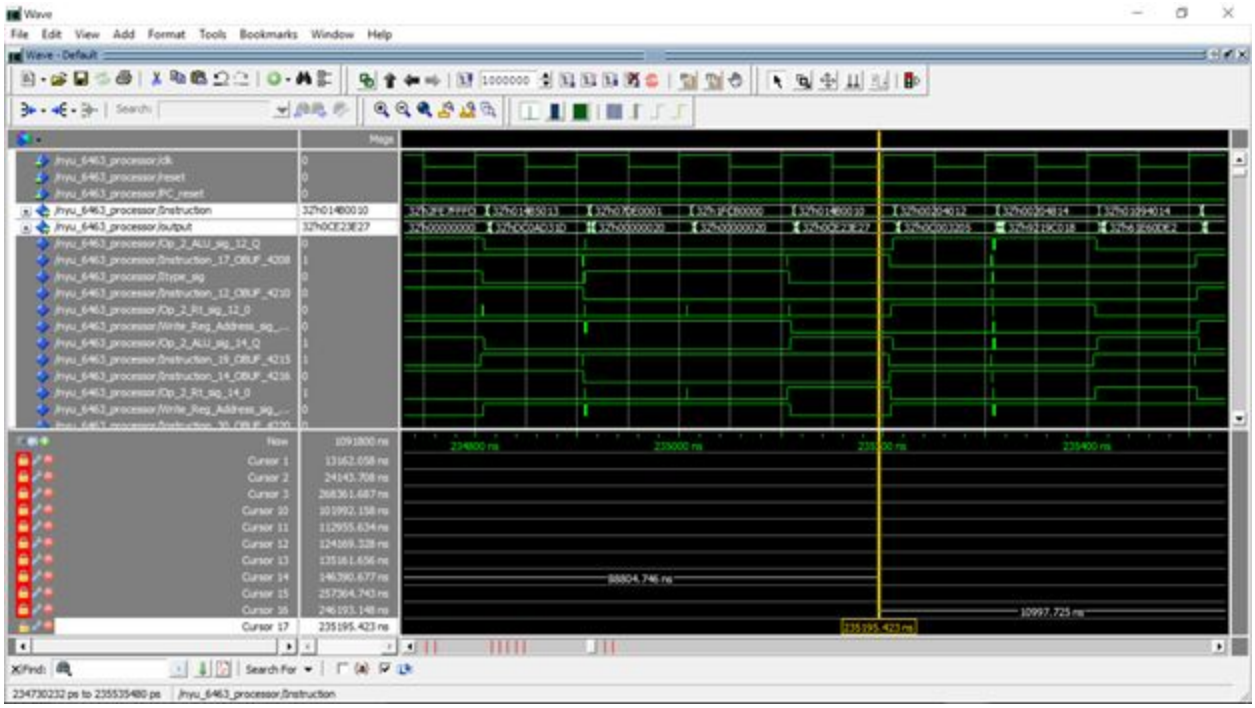
A2


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
/nyu_6463_processor/clk
/nyu_6463_processor/reset
/nyu_6463_processor/PC_reset
/nyu_6463_processor/Instruction
/nyu_6463_processor/output
Now
Cursor 1
Cursor 2
Cursor 3
11217.272 ns
/nyu_6463_processor/output

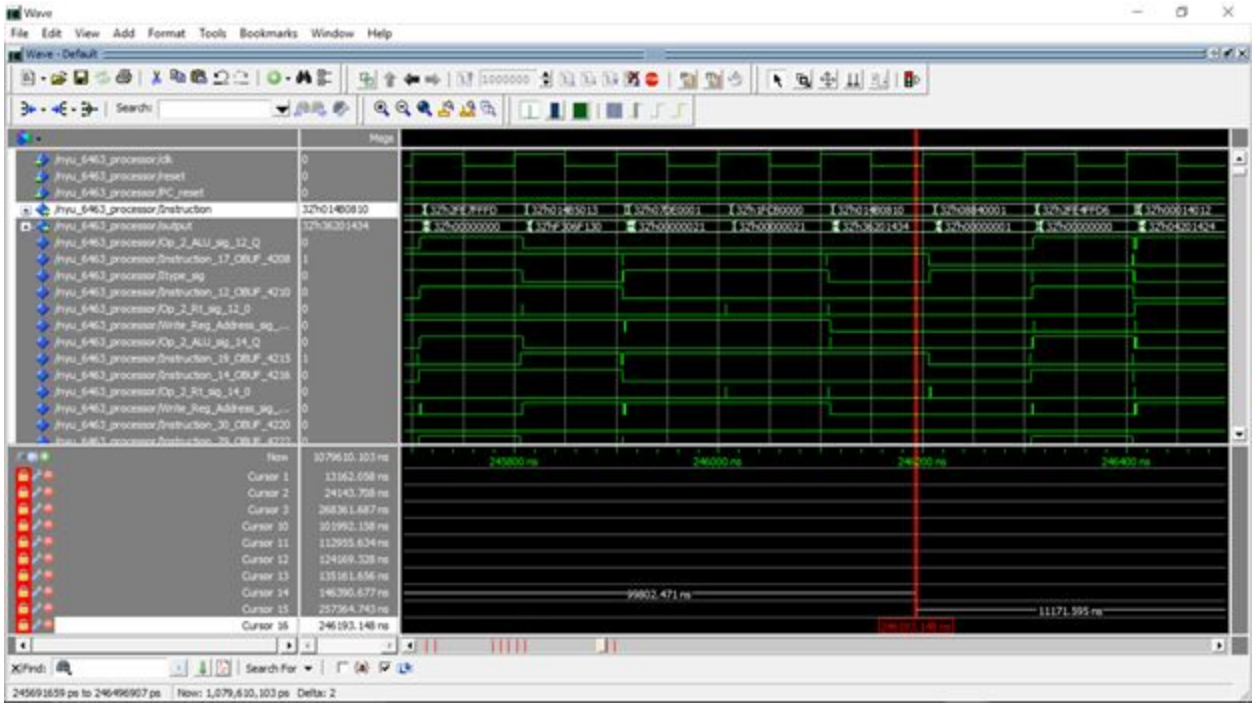
B2


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
/nyu_6463_processor/clk
/nyu_6463_processor/reset
/nyu_6463_processor/PC_reset
/nyu_6463_processor/Instruction
/nyu_6463_processor/output
Now
Cursor 1
Cursor 2
Cursor 3
Cursor 4
10992.122 ns
/nyu_6463_processor/output

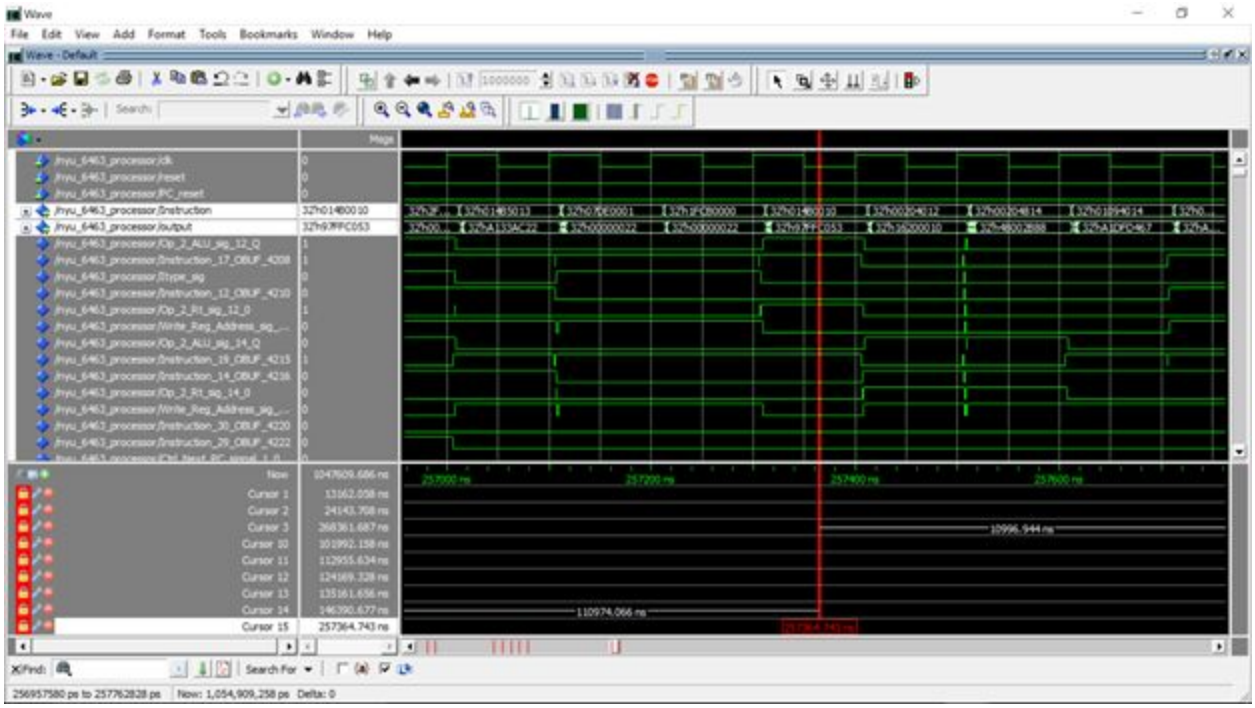
A11



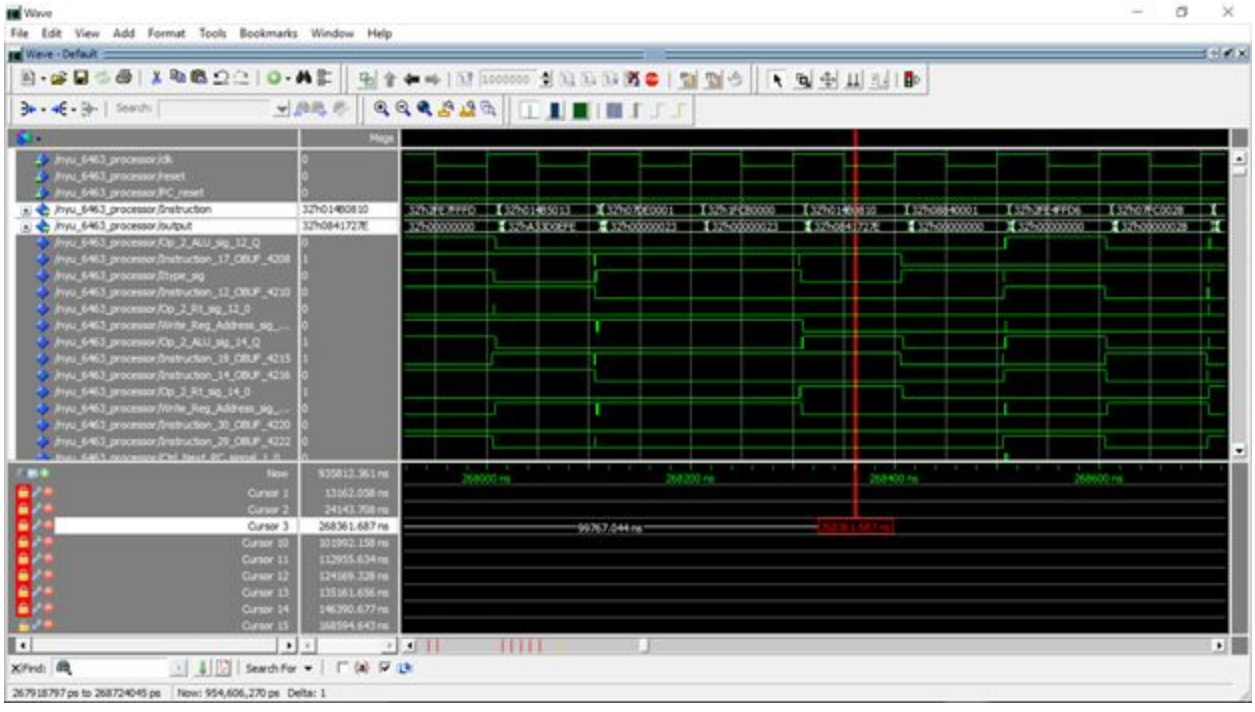
B11



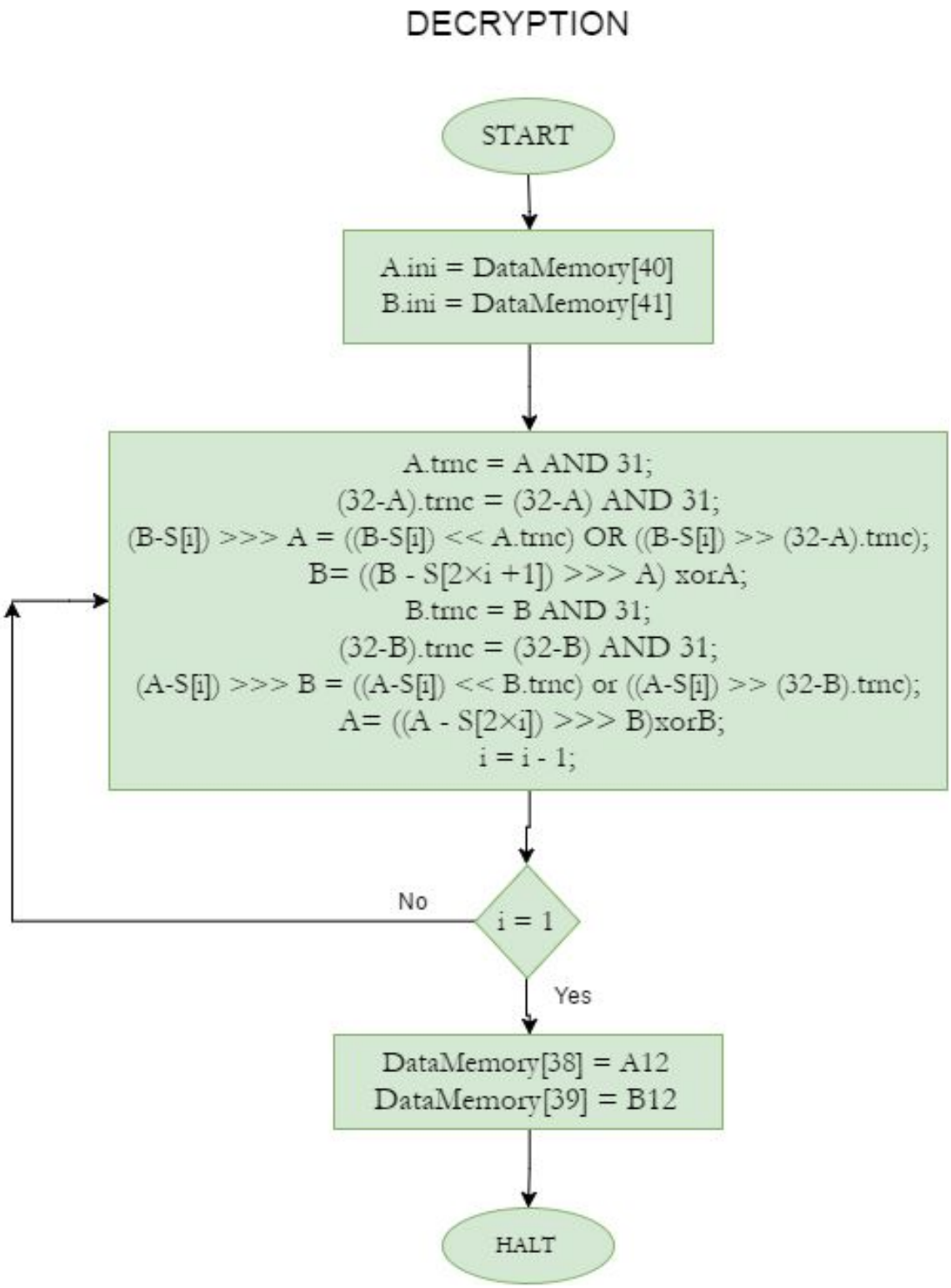
A12



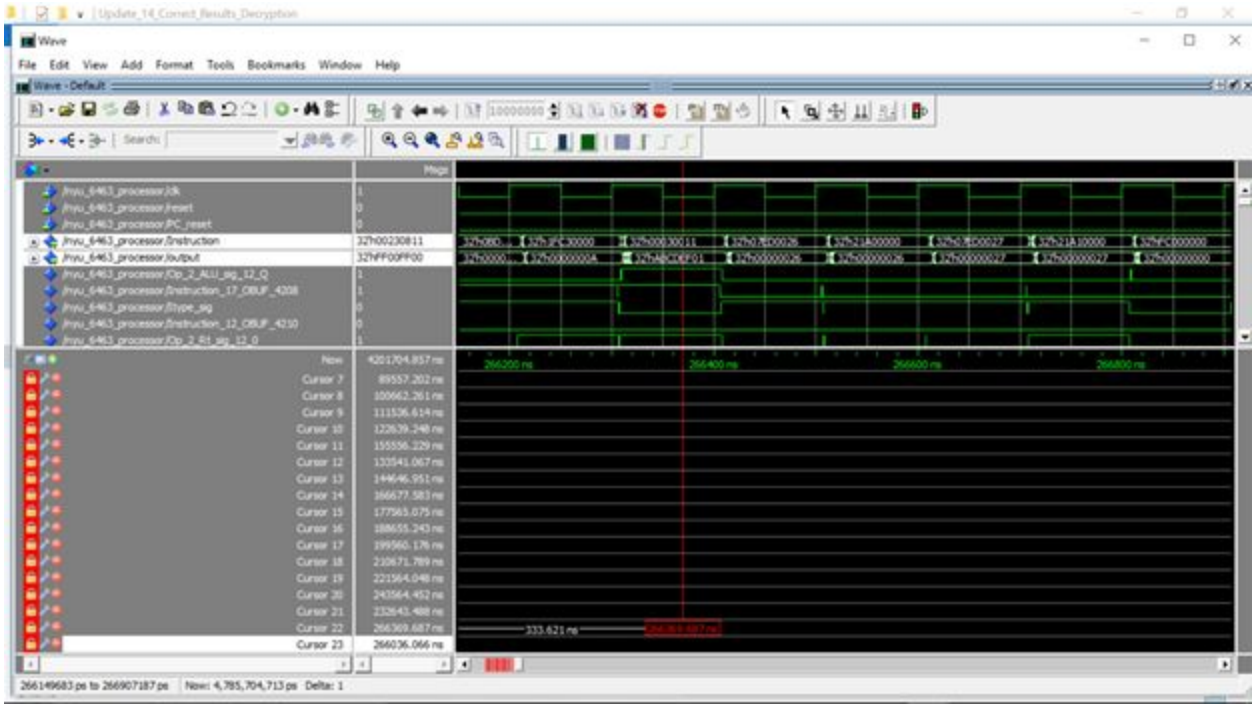
B12



4.3 Decryption

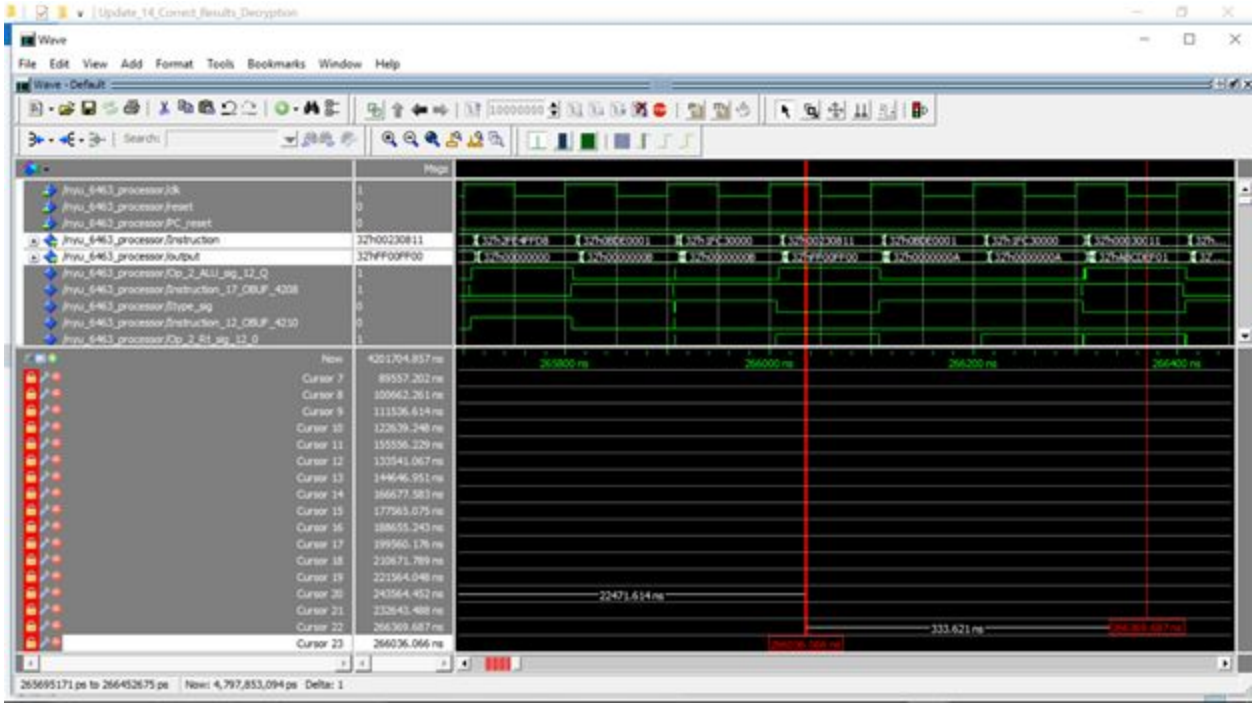


B12

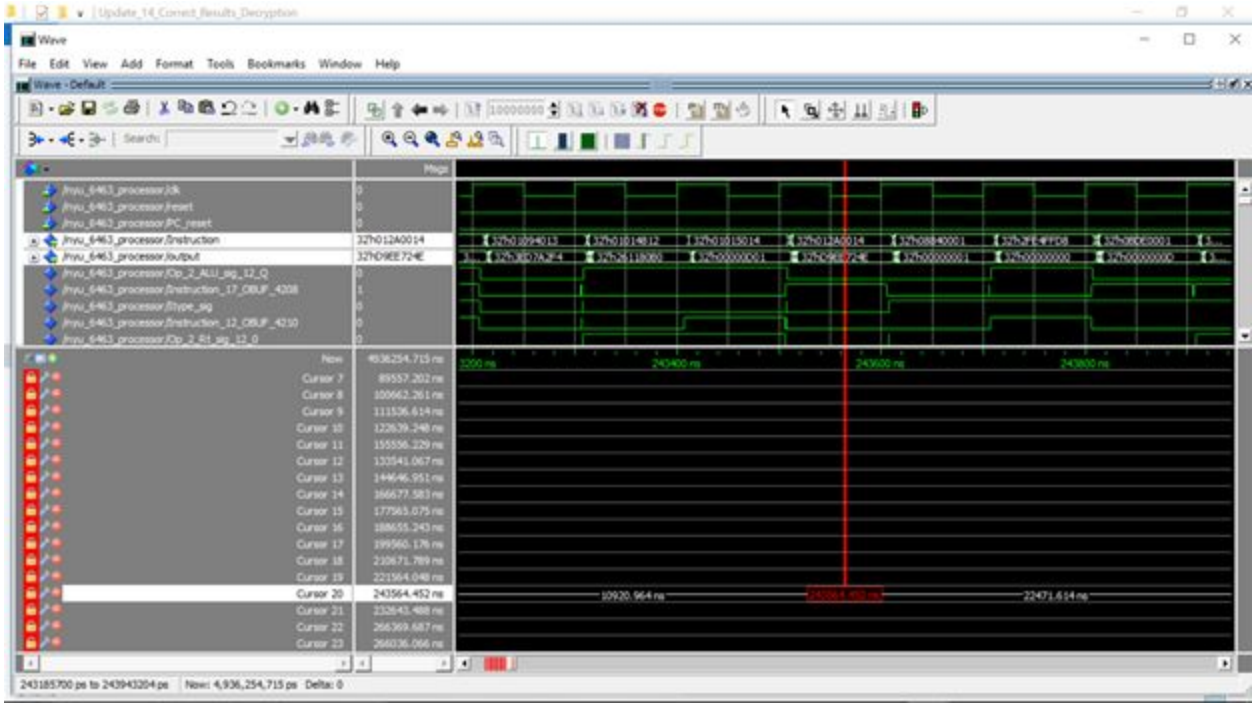

Wave
File Edit View Add Format Tools Bookmarks Window Help
/nyu_6463_processor/clk
/nyu_6463_processor/reset
/nyu_6463_processor/PC_reset
/nyu_6463_processor/Instruction
/nyu_6463_processor/output
32'h00230811
32'hFF00FF00
Cursor 23
266036.066 ns
333.621 ns

0

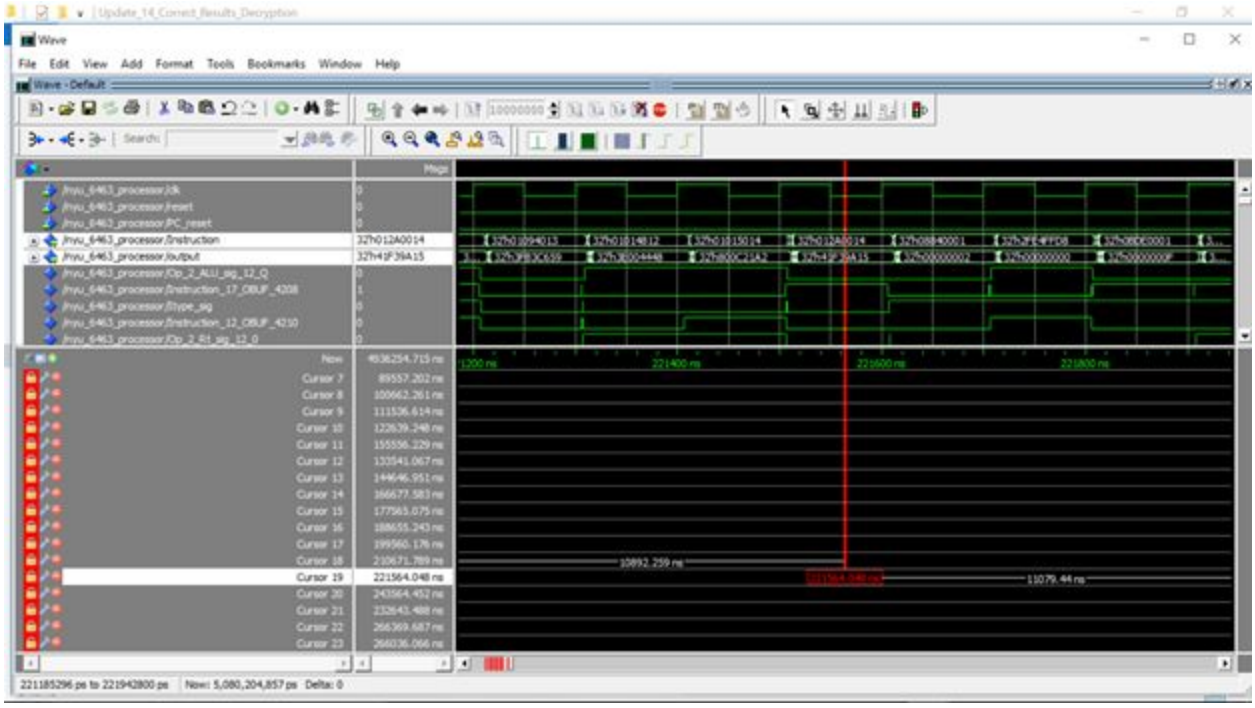
A12


Wave
File Edit View Add Format Tools Bookmarks Window Help
/nyu_6463_processor/clk
/nyu_6463_processor/reset
/nyu_6463_processor/PC_reset
/nyu_6463_processor/Instruction
/nyu_6463_processor/output
32'h00230811
32'hFF00FF00
Cursor 23
266036.066 ns
22471.614 ns
333.621 ns

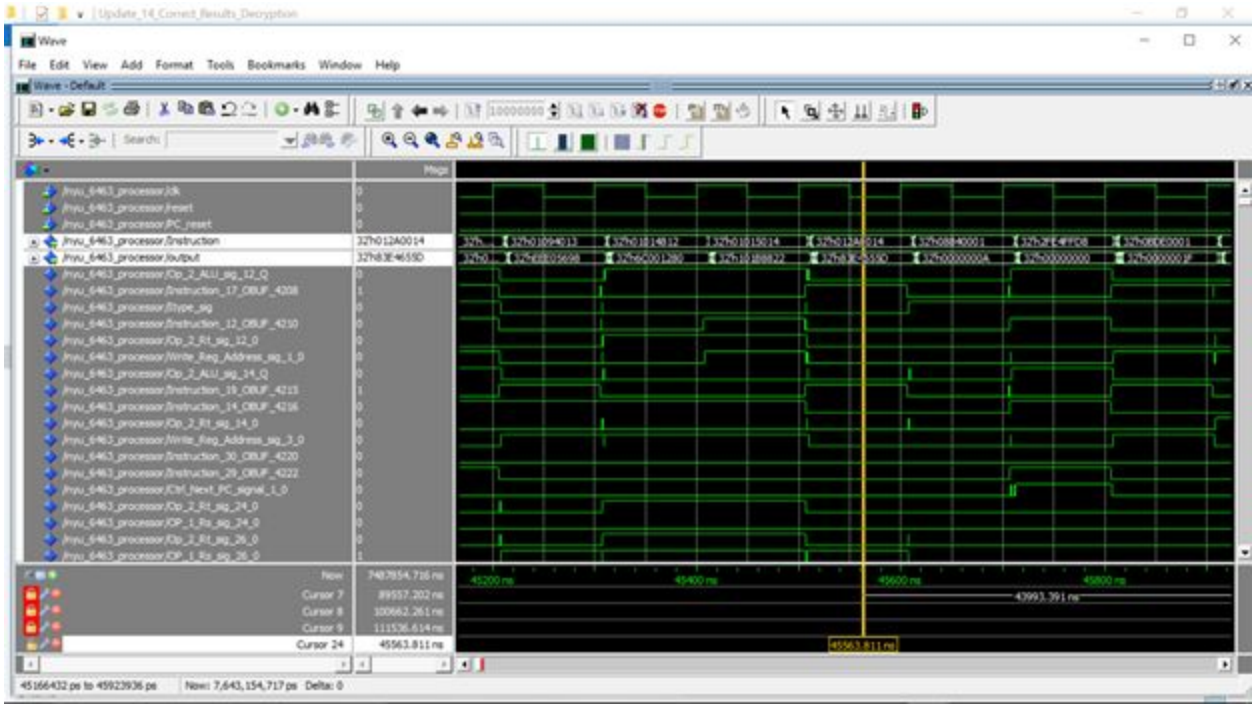
B11



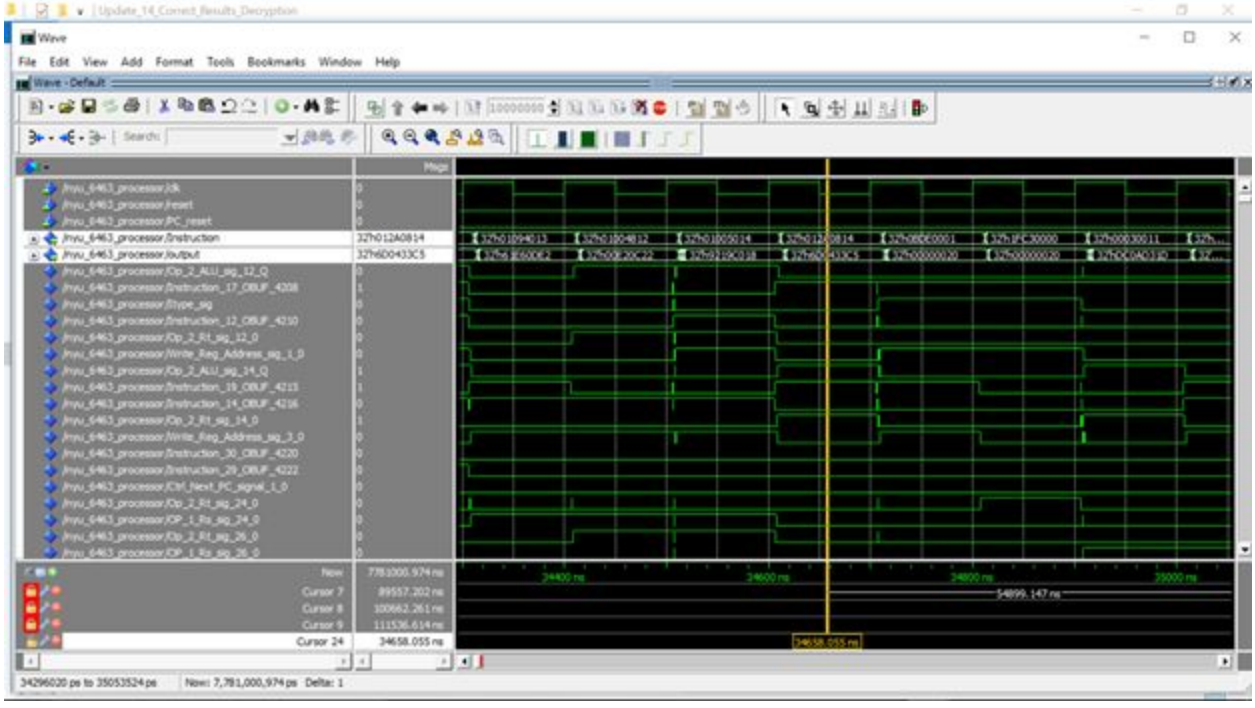
A11



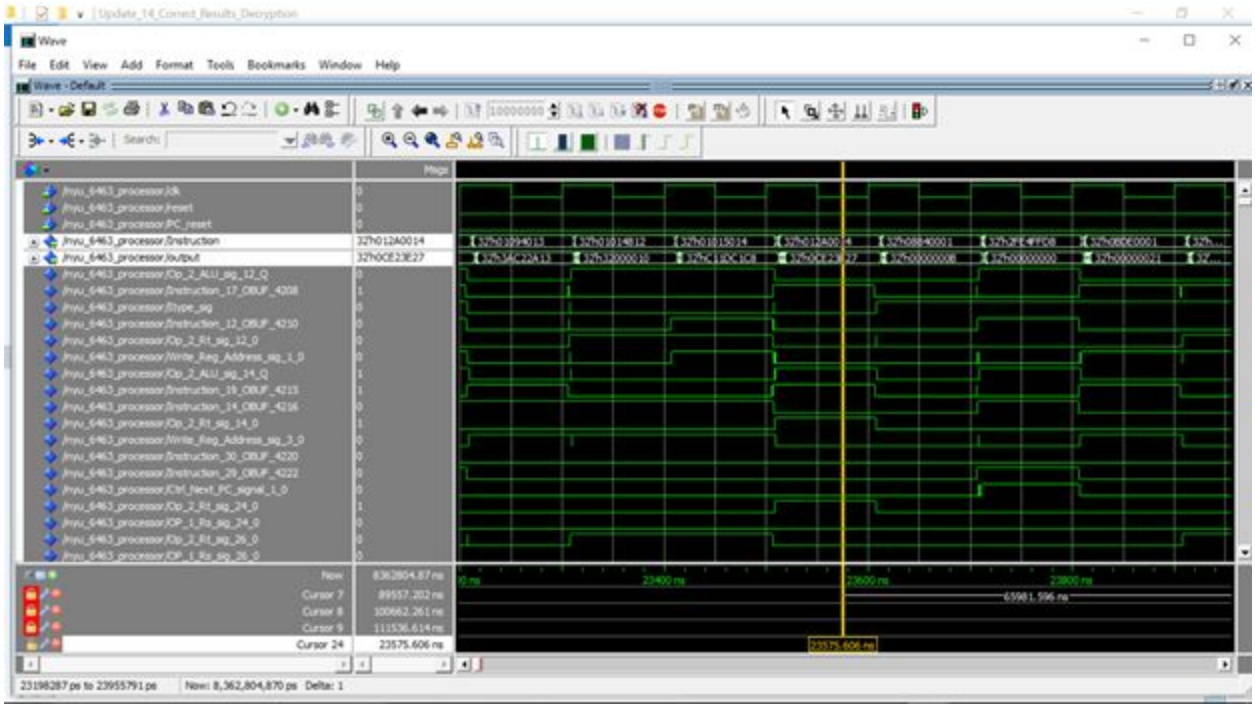
B2



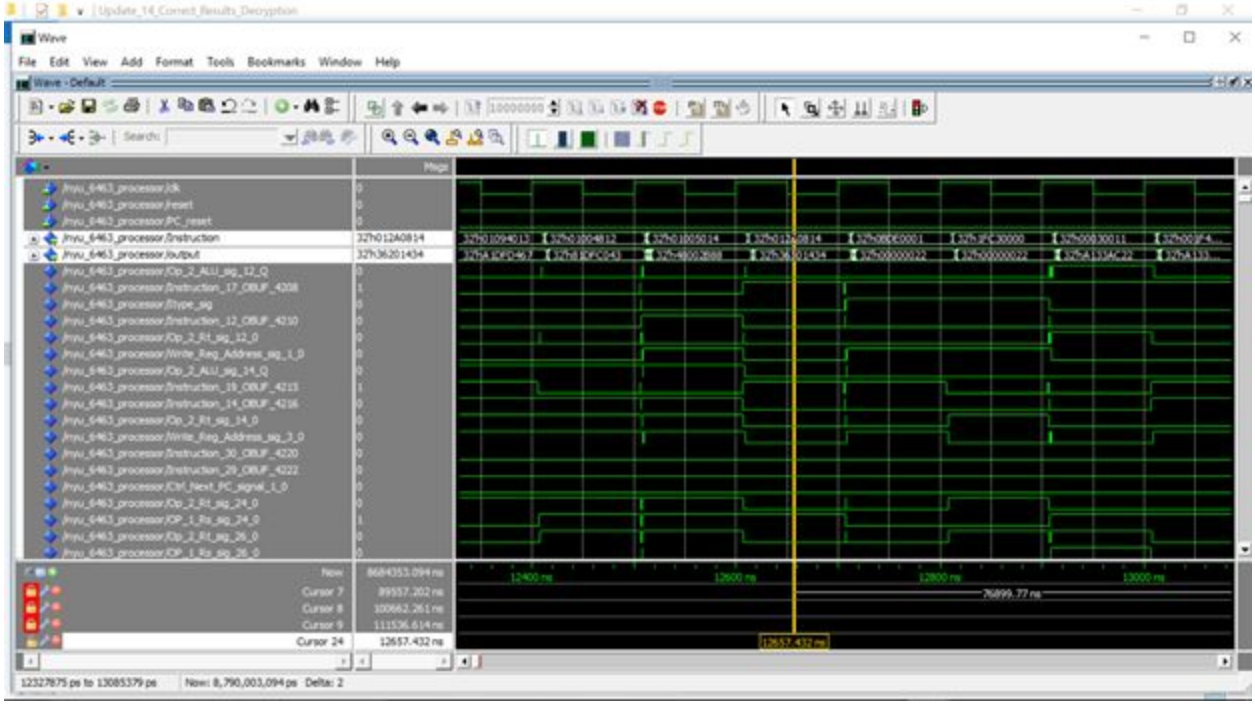
A2



B1



A1



5 Processor Interface

5.1 UART Interface to Write Instruction Memory:

Here the address of the file having hex codes (included in this file, name : "UART_Send") is entered. The hex codes are read from the file and sent over the UART to FPGA.

The FPGA receives the 6 bytes of Data (1 byte : x"04" to indicate write process, 1 byte for the address on which the instruction will be written on the data memory,
4 bytes of data which is the actual instruction.)

Presently, we are able to receive the correct instructions, and address on which instruction is to be written. We have verified that the instruction and address received at the Instruction memory is correct. However, due to some reason, probably due to synchronization, the instruction being actually written is getting corrupted in some cases.

It can been implemented if given a little more time.

5.2 7-segment display to view output results from Register File and ALU:

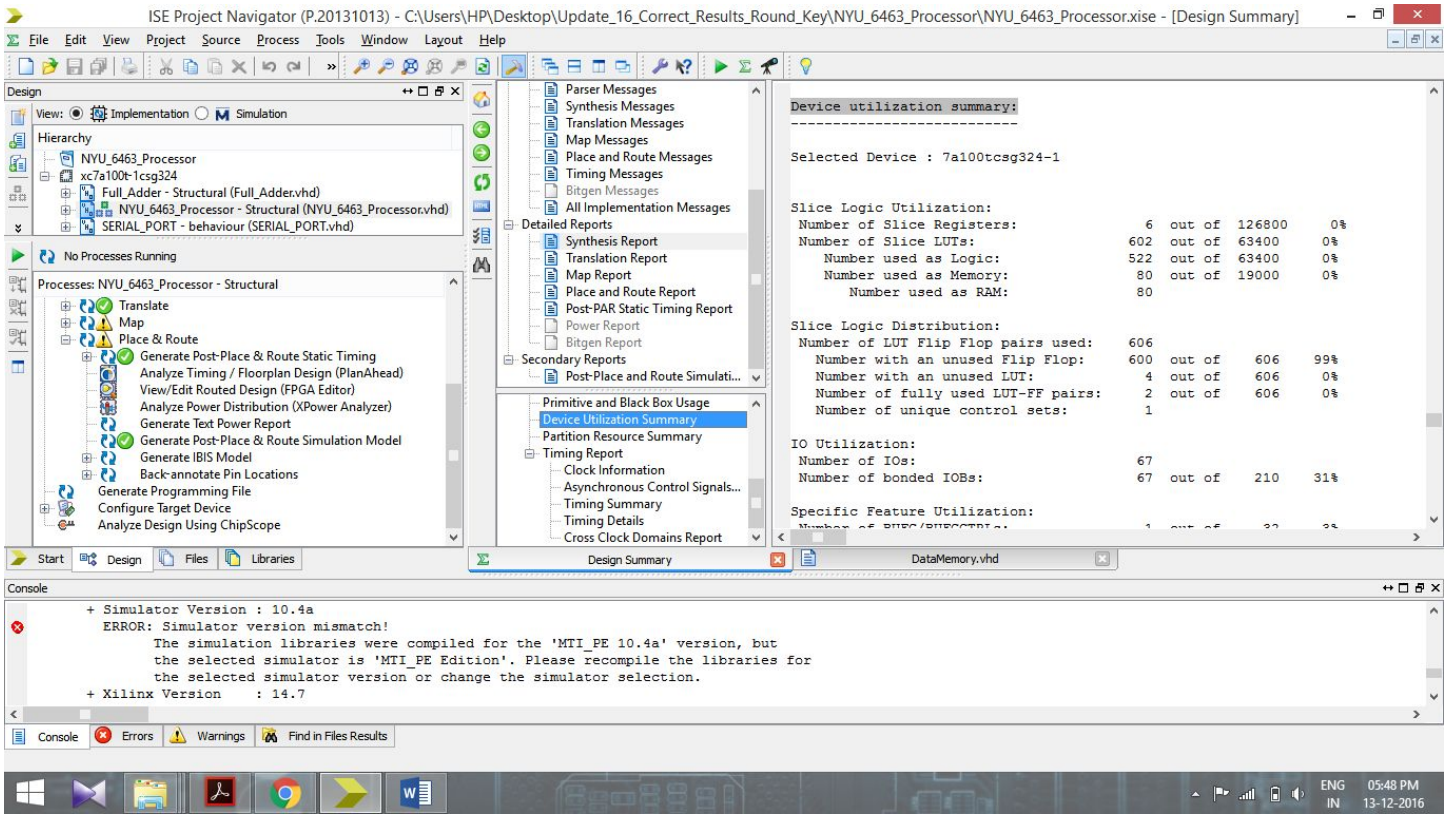
The Seven Segment display on the FPGA Board shows the output from Register File and ALU. The selection to see the output is done by selecting two push buttons, ‘Toggle’ and ‘Toggle1’. The button ‘Toggle’ gives the LSB of the selection and the button ‘Toggle1’ gives MSB of the selection. The respective output selected to be shown on the seven-segment display is shown in the table below.

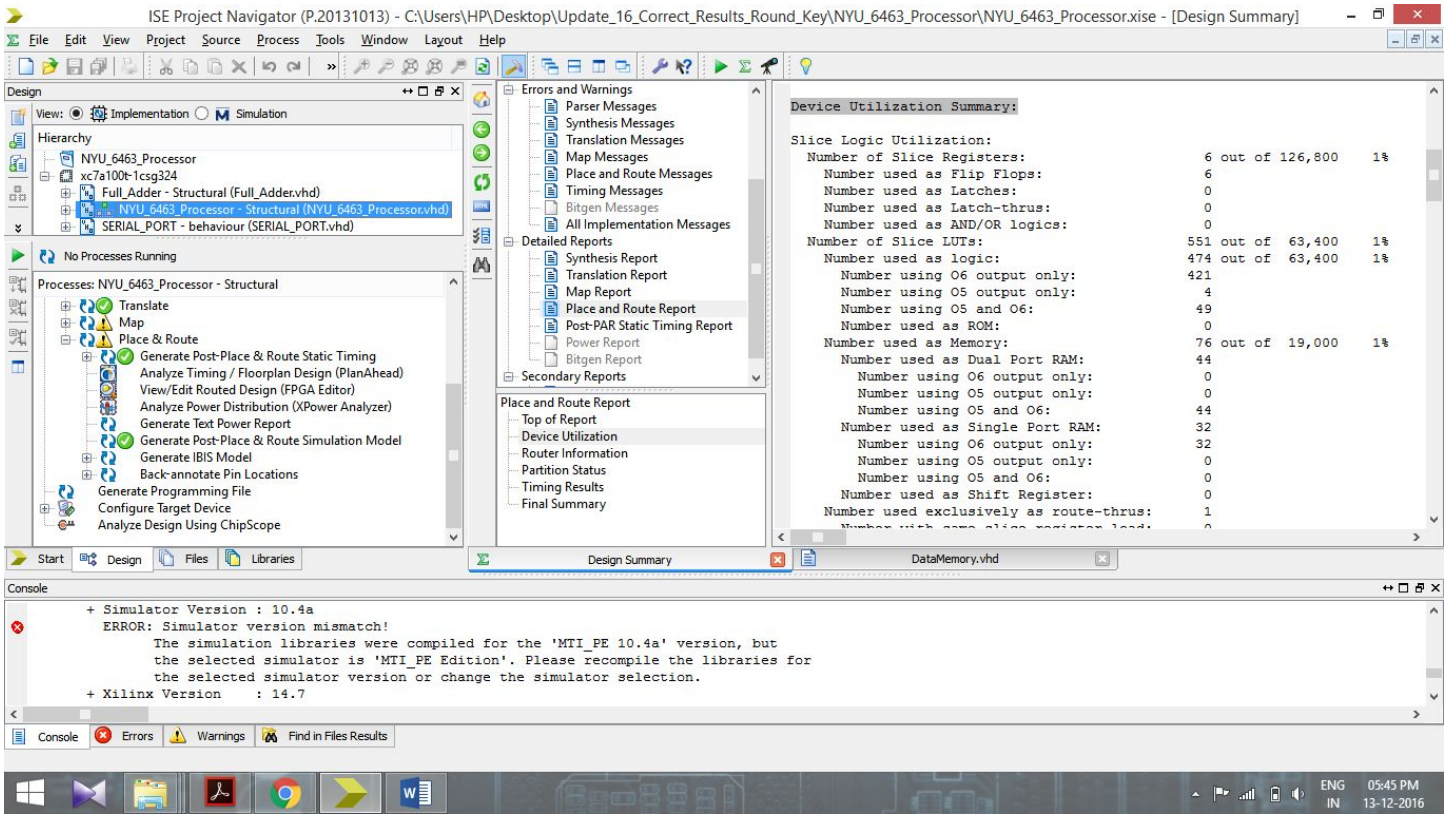
Toggle1 (MSB)	Toggle (LSB)	Output selected
0	0	ALU
0	1	Rs
1	1	Rt
1	1	Rd

6 Performance and Area Analysis

6.1 Resource Utilization

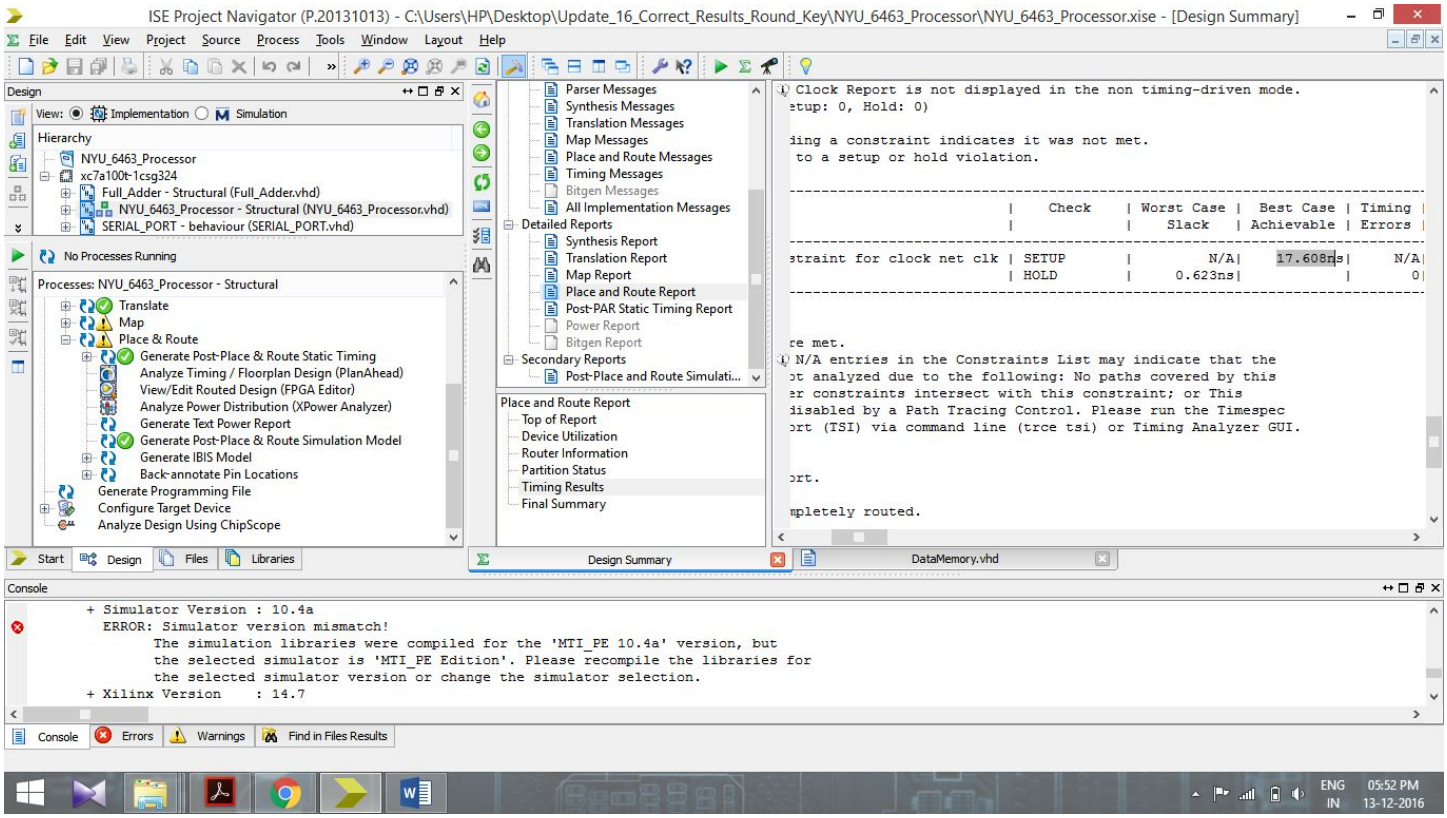
Timing numbers generated by Synthesizing the project gives only a synthesis estimate. However, after Post Place - and - route trace report, we can look for accurate timing information from the Trace Report. Number of Slice LUT, number of used LUTS, Sliced Registers and LUT Flip Flop pairs decreases in Post Place-and-Route(PAR) unlike synthesis(XST). In all a difference of 1% when generating the Post Place and route model than the synthesized model. IO utilization seems to be unchanged from the table below. Holistically, since the circuit is pretty small, due to optimization the place and route area is smaller than synthesis.

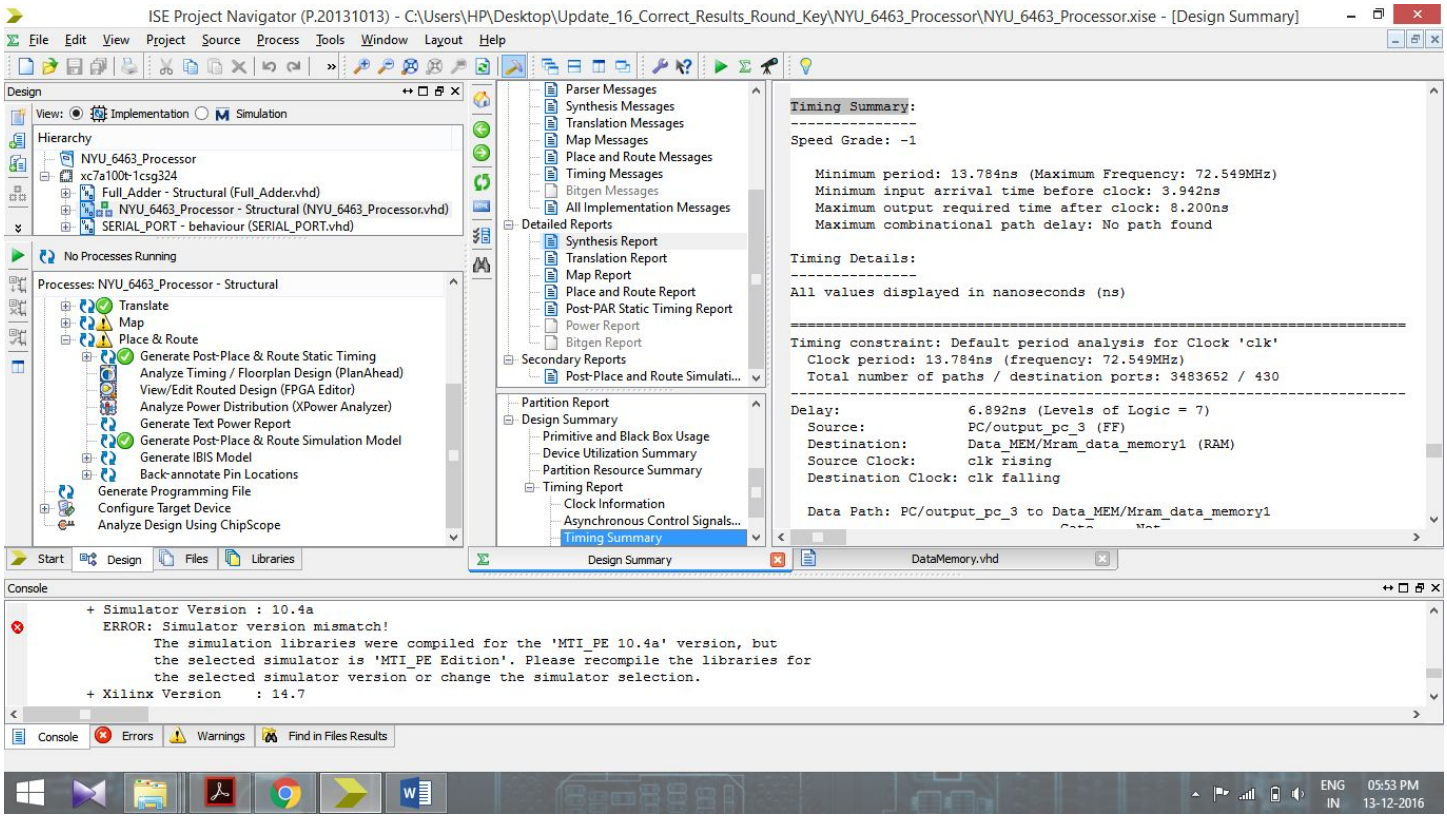




Resources	Post Synthesis	Post PAR
Slice Registers (FFs)	6 out of 126800 0%	6 out of 126,800 1%
Slice LUTS	602 out of 63400 0%	551 out of 63,400 1%
IOBs	67 out of 210 31%	67 out of 210 31%
Min Period	13.784ns	10.760 ns
Max Freq	72.549MHz	92.937 MHz

Timing simulation for MIPS





Parameter	Post Synthesis	Post PAR
Min Period(Tp)[Critical Path delay]	**13.784ns**	**17.608ns**
Max Freq(f)	**72.549MHz**	**56.79MHz**
Latency(L)	**268**	**268**
Propagation delay of Design-L*Tp	**3694.112ns**	**4718.944ns**

Latency: The number of cycles it takes to produce valid output for an input
Critical path delay of the design is given as minimum period

Critical path delay is found to be **17.608ns** from PAR Report. So the maximum frequency with which the circuit can be run is clk= 1/**17.608ns** = **56.79MHz**.

Total number of cycles required as per timing simulations and cross verified from the assembly code by taking time complexity into consideration				
CODE	**Round Key**	**Encryption**	**Decryption**	**Sample 2**
Number of cycles	**13299**	**2683**	**2663**	**32**

7 Verification of overall design

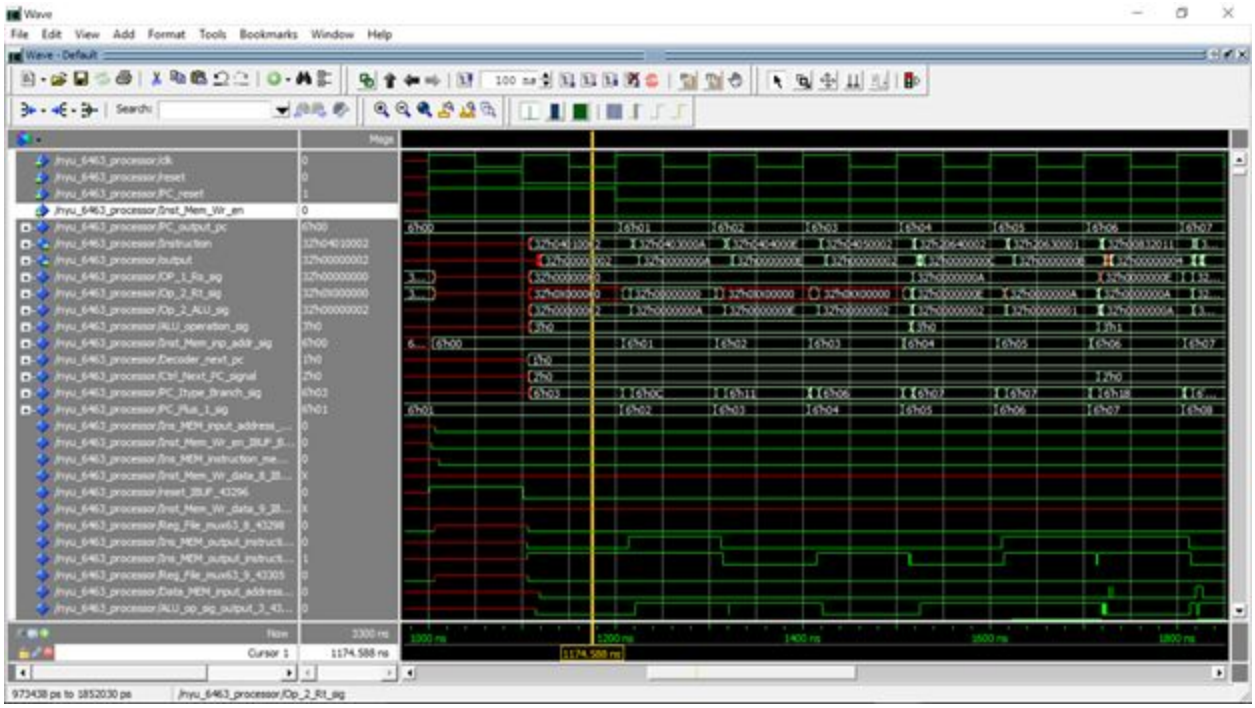
We verified the processor design by running the assembly code for the following on **Functional, Timing simulation and FPGA. We cross checked the intermediate step values using C++ code** :

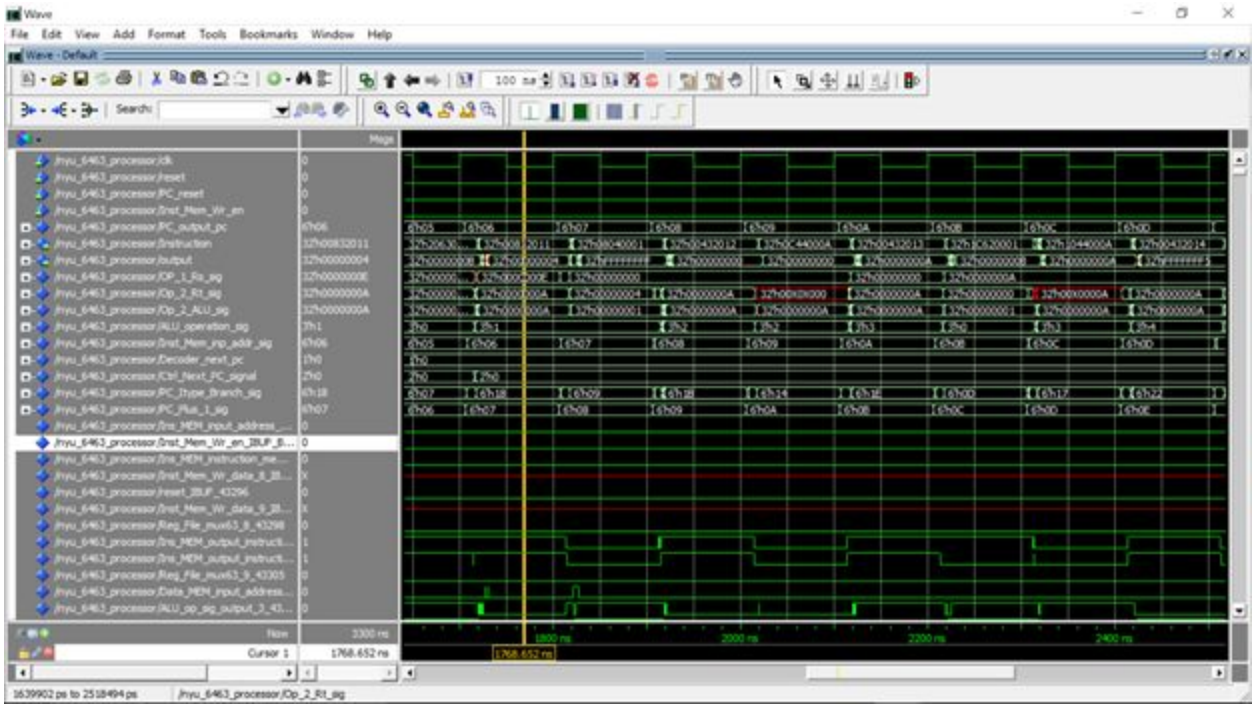
- Sample code 1
- Sample code 2
- RC5 Key Expansion
- RC5 Encryption
- RC5 Decryption

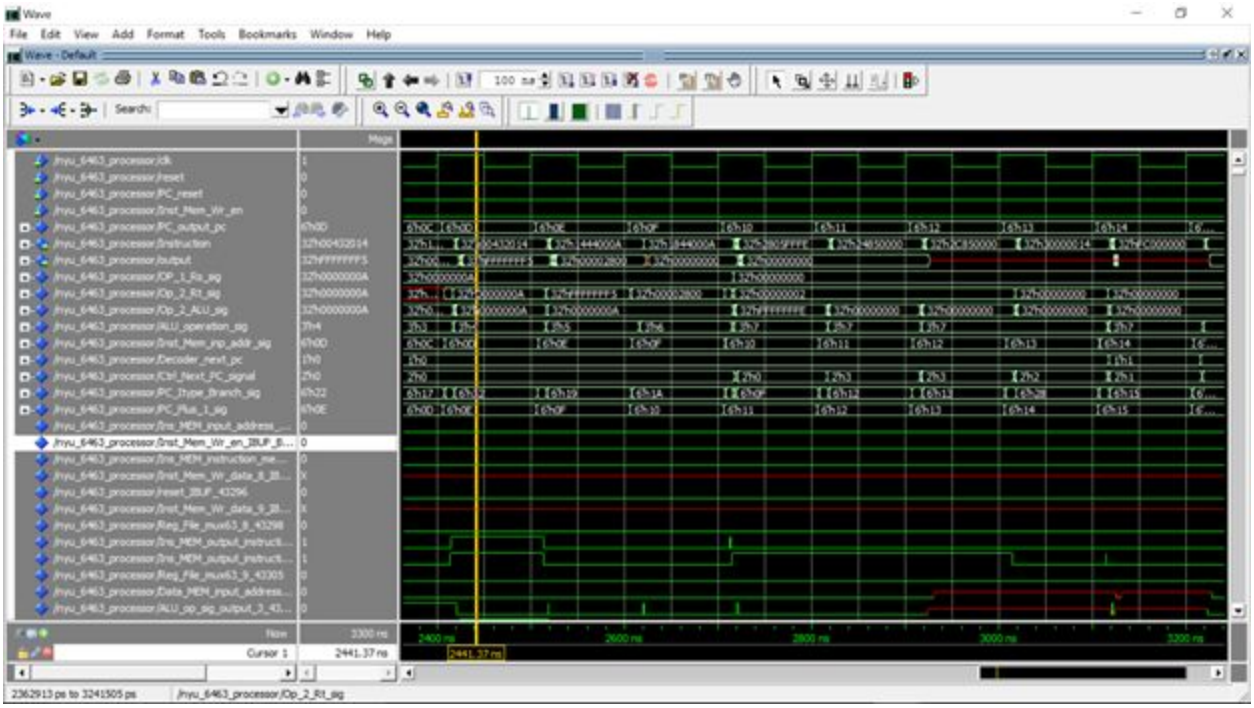
8 Sample 2 code

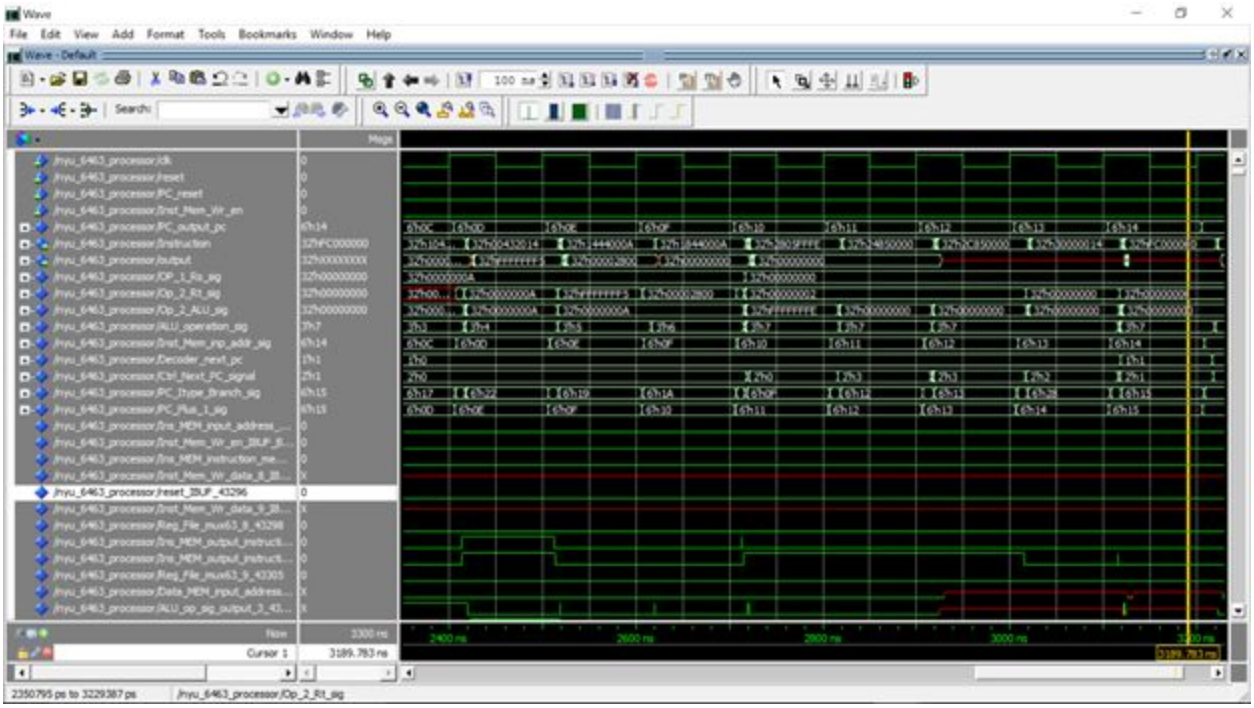
The UART module and Software is complete and working. We have verified the sample 2 code using UART. The software has the feature to read the inputs such as "memory location" and "instruction" from a text file, and then send the 6 bytes of data over the UART (1 byte fixed "04" to indicate write operation, 1 byte Instruction memory address, 4 bytes of Instruction data).
After reception of the complete packet, the UART segregates and sends the address and the instruction, to be written on the instruction memory.
The manual clock is provided to processor using a "button" to perform single stepping execution.

Timing Simulations of Sample 2 code integrated with UART module:









Final Register File:

Register	Values (decimal)

R0	0
R1	2
R2	10
R3	10
R4	0
R5	2
R6	0
R7	0
R8	0
R9	0
R10	0
R11	0
R12 to R31	0

Final Data Memory Values:
0 to 10th location <= 0
11th Location <= 10 (decimal)
12th location <= 14 (decimal)
13th to 63rd Location <= 0

Final Program Counter Value: 21 (decimal).
NOTE: The instruction memory is being written from location 1 and NOT 0, so total instructions are 21. The 21st instruction is halt (0xFC000000).
