**NYU-6463 Processor Design**



**EL-6463 ADVANCED HARDWARE DESIGN**

|  |
| --- |
| **By,** |
| **Dan Wu (dw1987)** |
| **Kush Agarwal (ka1809)** |
| **Madhumita Sawant (mms916)** |
| **Monica Reddy Tirupari (mrt374)** |
| **Shubham Mahalank (sm7675)** |
| **Xinmei Jin(xj573)** |
| **Zhaoxi Chen(zc1134)** |

**Table of Contents**

1. **Introduction**
2. **Processor Description**
   1. Instruction Types
   2. Instruction Set
   3. Working of the Processor
3. **Block Diagram and Modules**
   1. Block Diagram
   2. Modules
4. **Simulations of Each Module** 
   1. Instruction Memory
   2. Decoder
   3. Register File
   4. ALU
   5. Data Memory
5. **Simulation of NYU 6463 Processor**
   1. Function Simulation
   2. Timing Simulation
6. **Implementation of RC5 on the Processor**
   1. Key Expansion
   2. Encryption
   3. Decryption
7. **Performance Analysis**
   1. Resource Utilization
   2. Timing Summary/Analysis
8. **Sample Codes**
   1. Sample Code 1
   2. Sample Code 2
9. **Implementation on FPGA**
10. **Links**
11. **Introduction**

We have implemented a 32-bit single cycle processor in VHDL, named NYU-6463 Processor, which can execute various programs. The processor supports the instruction set specified in the next section.

It fully executes an instruction in one clock cycle only. It’s a processor having 32-bit long instructions. These instructions can be divided depending on the type of the instruction, but one constant is the opcode which is defined by the first 6 bits of the instructions starting at the Most Significant Bit (MSB).

A processor is an ensemble of components that perform different crucial tasks allowing it to work properly. A large number of instructions can be implemented on a processor but for this project only a few were used.

The components that constitute this processor are as follows:

* 1. Program Counter:
  2. Instruction Memory (IM) in which the code to execute will be placed.
  3. Register File (RF) which will provide the values of the registers needed for an instruction and specify the register it has to write to after the execution of the instruction.
  4. Arithmetic Logic Unit (ALU) which is responsible for the arithmetic operations such as addition and subtraction.
  5. Decoder which breaks down the instruction to identify the value of the control signals as the instruction type for instance.
  6. Data Memory (DM) in which all the values can be stored.

This report will explore all the processor components in detail. The processor simulations and individual module simulations have been specified and

1. **Processor Description**
   1. **Instruction Types**

NYU-6463 Processor has three instruction types:

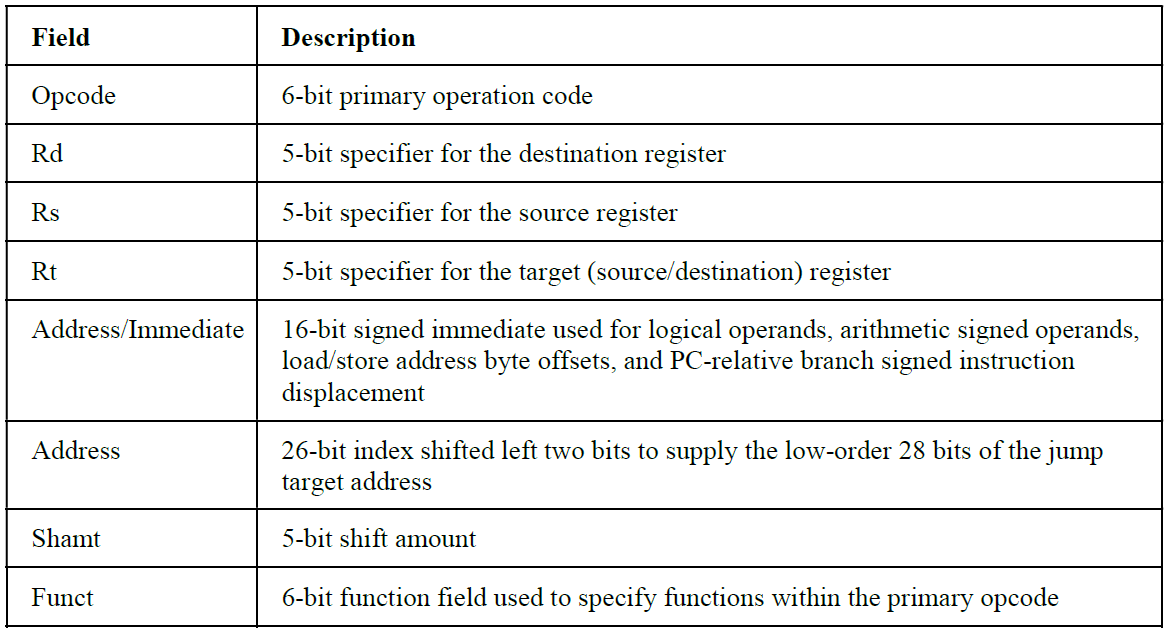
* + R-Type, for arithmetic instructions
  + I-Type, for immediate value operations, load and store instructions
  + J-Type, for Jump instructions

The three instruction types are shown in the table below:



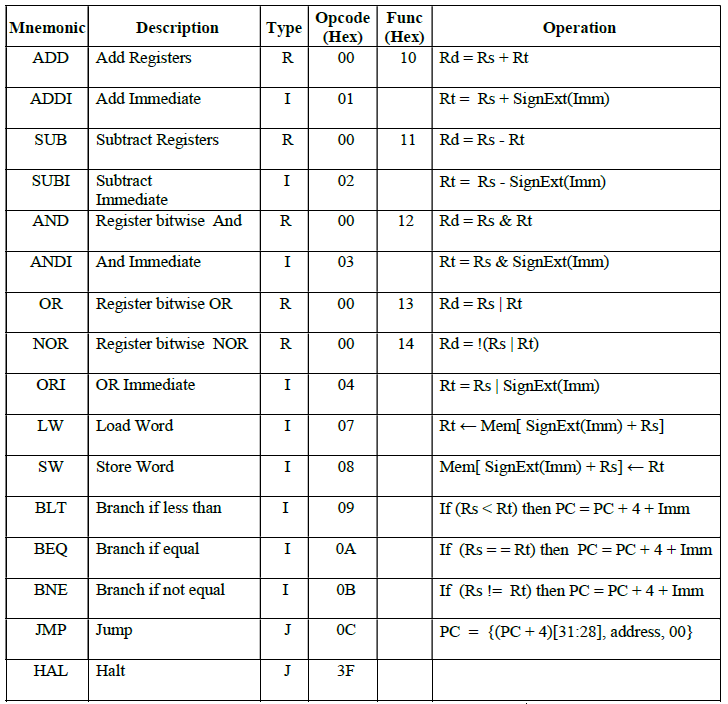
Each of three instructions are 32-bit instructions and they define the type of instructions as well as other information like operand values and destination where the results are to be stored. Each of the instruction fields has its own importance and the tasks to be performed by the processor depends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

The description of various instruction fields is given below:



* 1. **Instruction Set**

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



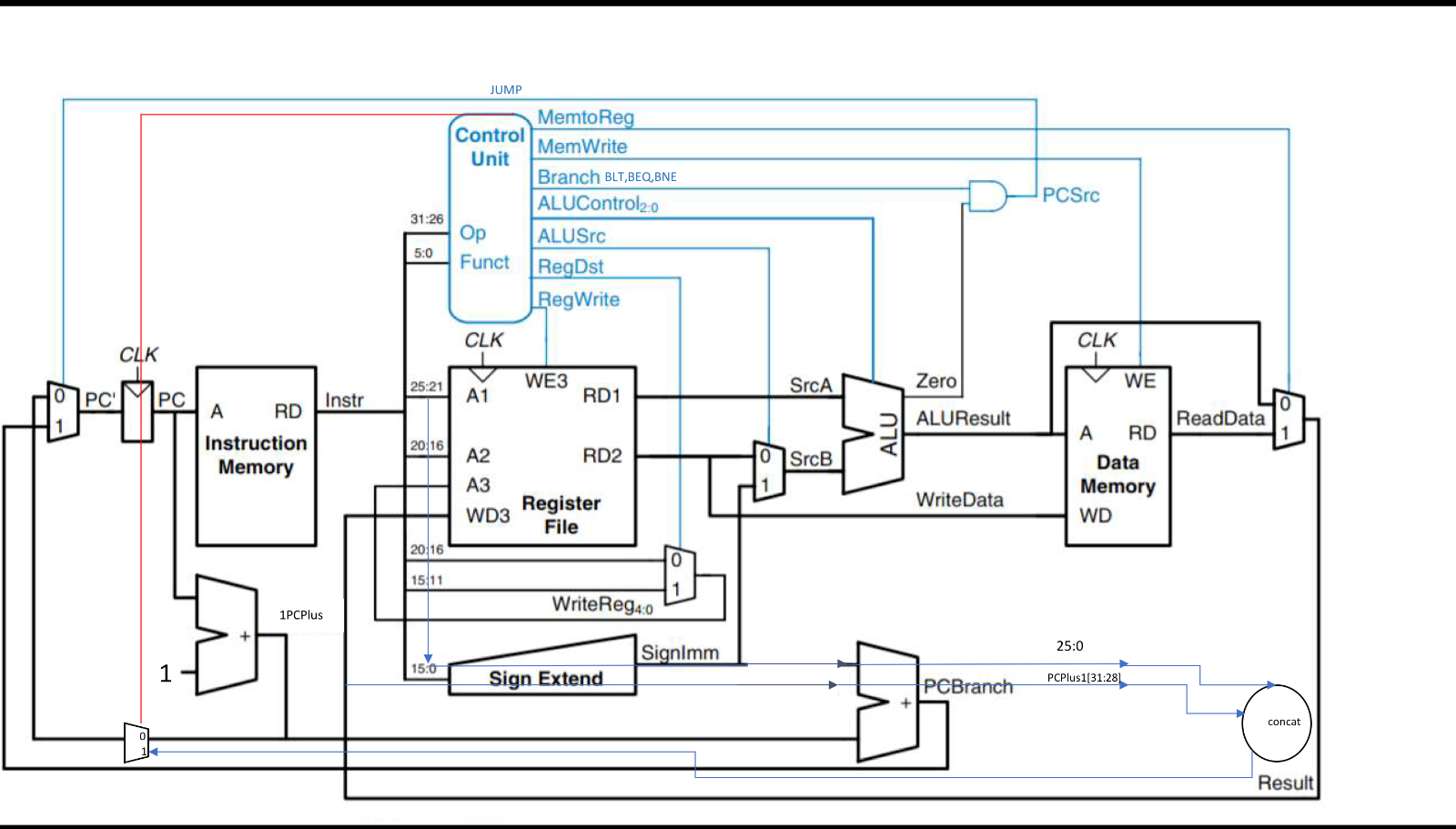
* 1. **Working of the Processor**

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.

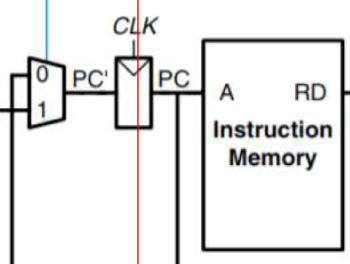
Depending on the data in the instruction fields, the Control Unit decides whether to compute the memory address or to perform the arithmetic or comparison operation. Depending on the Instruction decoded, the results from the ALU are directed appropriately. If the task performed is arithmetic, the ALU 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.

1. **Block Diagram and Modules**
   1. **Block Diagram**



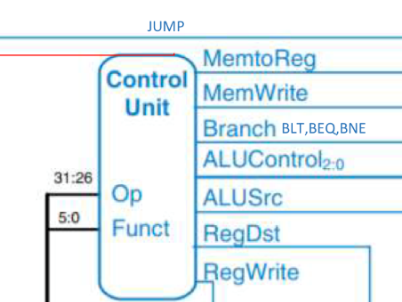
* 1. **Modules**
     1. Program Counter:

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



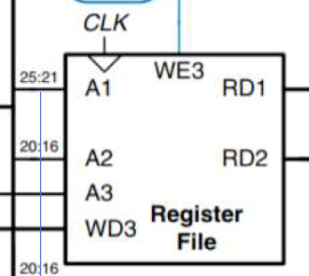
* + 1. Decoder

The Decoder Unit is responsible for generating the proper control signals for the rest of the blocks in the design hierarchy. It makes use of the first 6-bits and the last 6-bits of the 32-bit input instruction to generate the appropriate control signals for the microprocessor.



* + 1. 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.

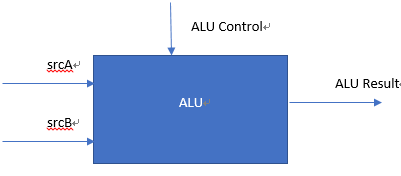


* + 1. ALU

ALU is one of the necessary component in a processor. It is responsible for arithmetic, logic and immediate operations like Addition, Subtraction, AND, OR, NOR, Left Shift and Right Shift. The unit receives two inputs srcA and srcB each of 32-bits. One input is from the register file and the other input is from the register file or instruction memory directly.

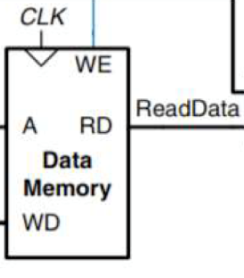
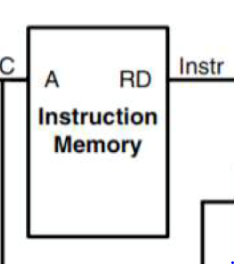
ALUControl is a control signal based on which the corresponding operation to be performed by ALU is selected. The output is sent as an address to Data Memory Unit.

Below is the block diagram for the ALU module.



* + 1. Instruction Memory (IM) and Data Memory

PC (Program Counter) is a 32 bit register that holds the address of next instruction to be executed. The Data Memory, RAM, is an array of 64 memory locations. Each having a 32-bit data.

1. **Simulations for Each Module**

Simulation is carried out using ModelSim and Vivado. The screenshots of the functional and the timing simulation have been included in the report. The screenshots were captured at three stages mainly. 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 screenshots.

The complete design of the processor was then simulated and the screen shots were captured at different stages of the operation. The sections 4 and 5 show the simulation screenshots, of each of the components of the Processor and finally, the complete NYU-6463 Processor.

* 1. **Instruction Memory (IM):**

As described in the project requirement, the IM module is used to restore all the instruction needed to be executed in correct order. In this simulation, the instruction is set to the part 1 sample code.

00000100000000010000000000000111 --ADDI R1, R0, 7 // R1 = 7

00000100000000100000000000001000 --ADDI R2, R0, 8 // R2 = 8

00000000010000010001100000010000 --ADD R3, R1, R2 // R3 = R1 + R2 =15

11111100000000000000000000000000 --HAL // HALT



* 1. **Decoder**

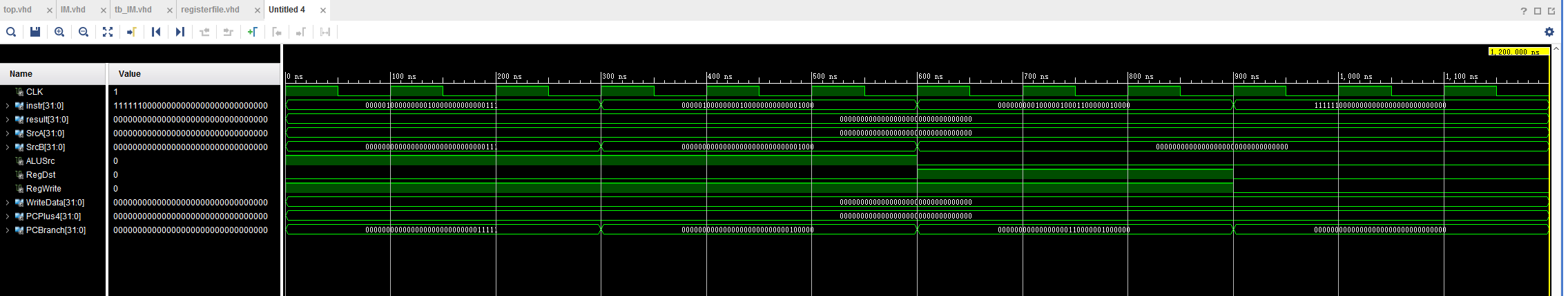
4.2.1. Functional Simulation

Decoder unit for the circuit (which is the control unit in the image). The input of instruction is just the test code 1 from part 1 requirement



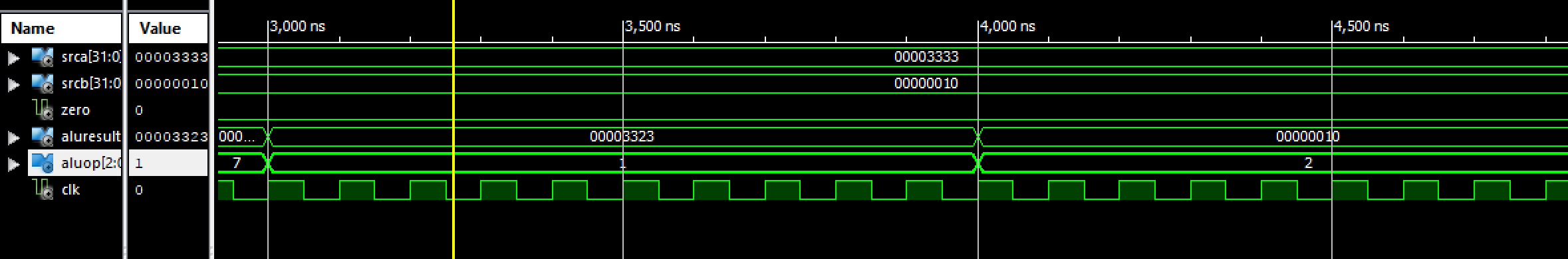
* 1. **Register File**

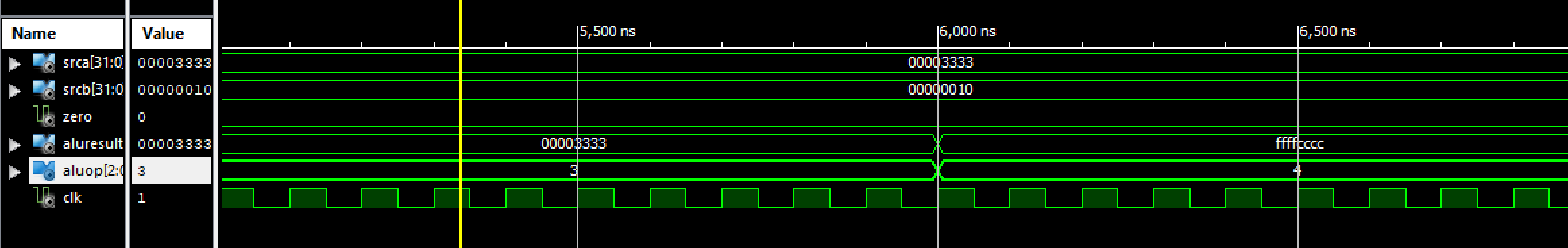
The input instruction also comes from the Sample Code 1, which has 4 instructions; the input of ALUSrc, RegDst, RegWrite comes from the same time output of the decoder simulation up above.

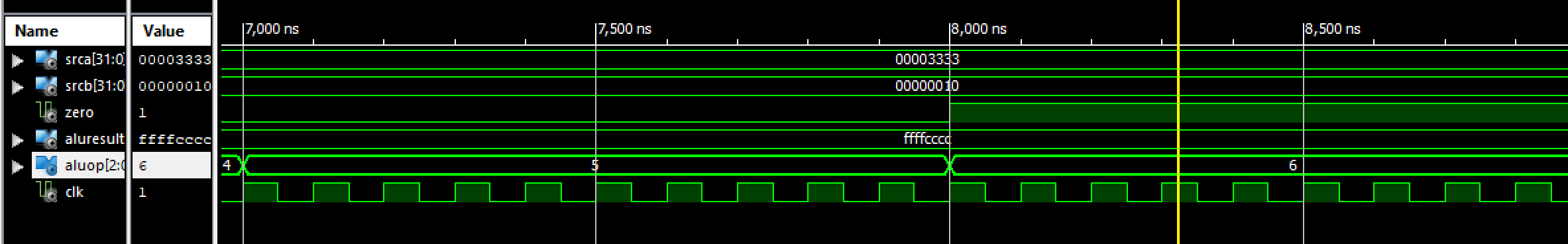


* 1. **ALU**

4.4.1. Functional Simulation



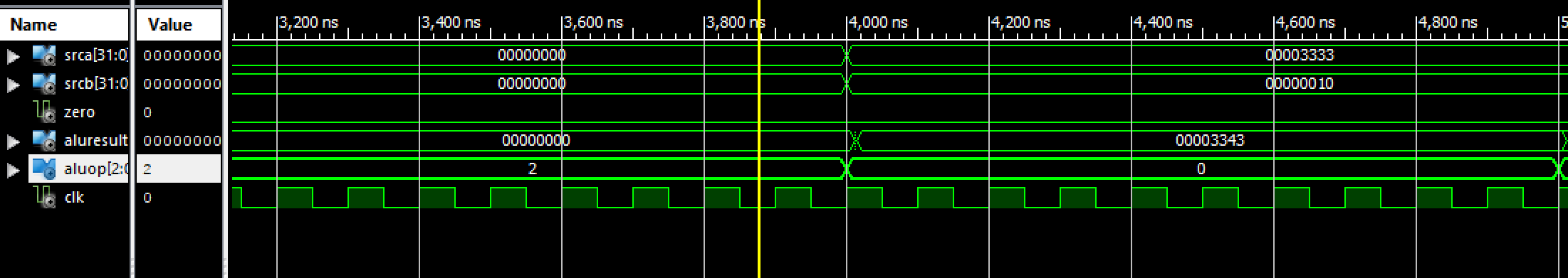


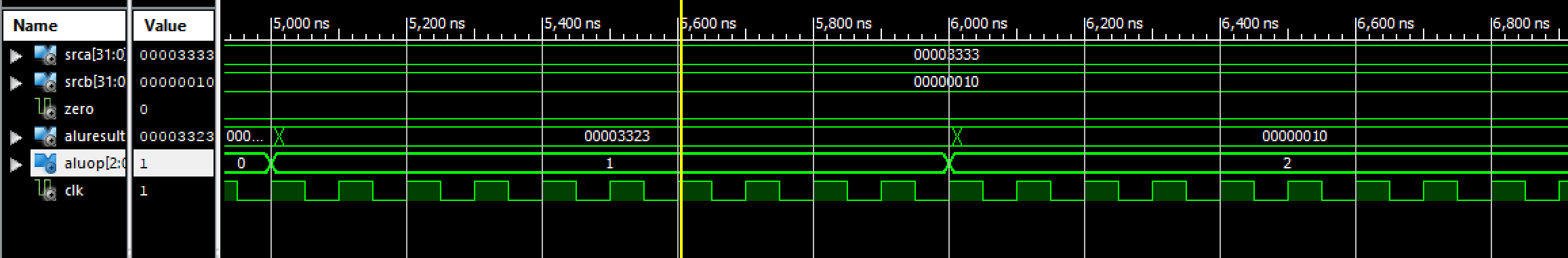


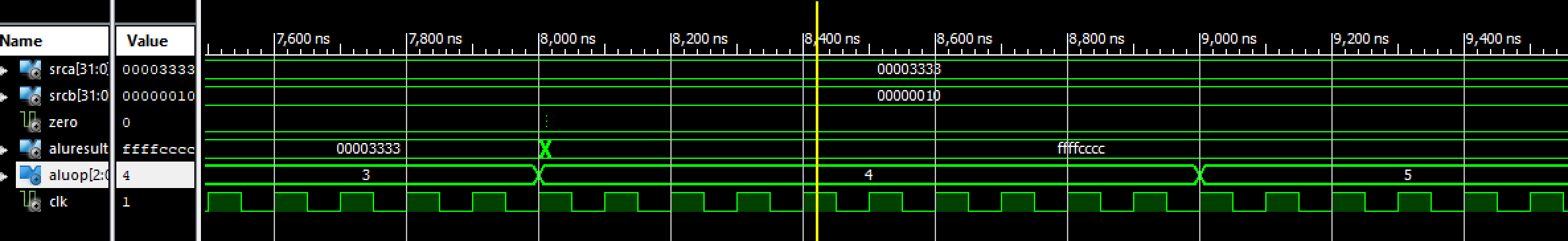
Operation Table:

|  |  |  |  |
| --- | --- | --- | --- |
| Operand 1 (srcA) | Operand 2 (srcB) | ALU Operation  (ALUControl) | ALUResult |
| 000000FF | 00000003 | 000 (Add) | 00000102 |
| 000000FF | 00000003 | 001 (Subtract) | 000000FC |
| 000000FF | 00000003 | 010 (AND) | 00000003 |
| 000000FF | 00000003 | 011 (OR) | 000000FF |
| 000000FF | 00000003 | 100 (NOR) | FFFFFF00 |
| 000000FF | 00000003 | 101 (Left Shift) | 000007F8 |
| 000000FF | 00000003 | 110 (Right Shift) | E000001F |

4.4.2. Timing Simulation



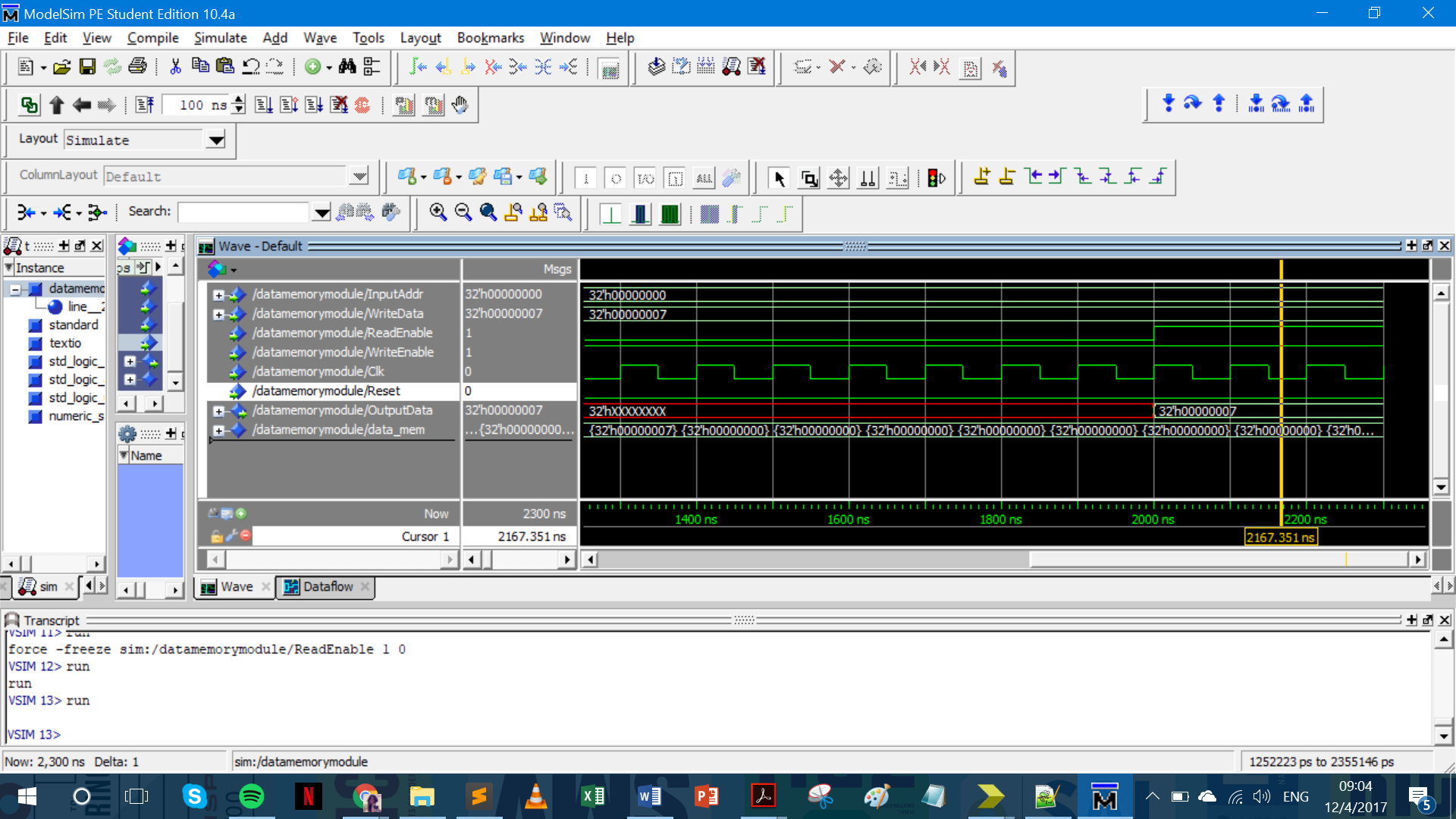




* 1. **Data Memory (DM):**

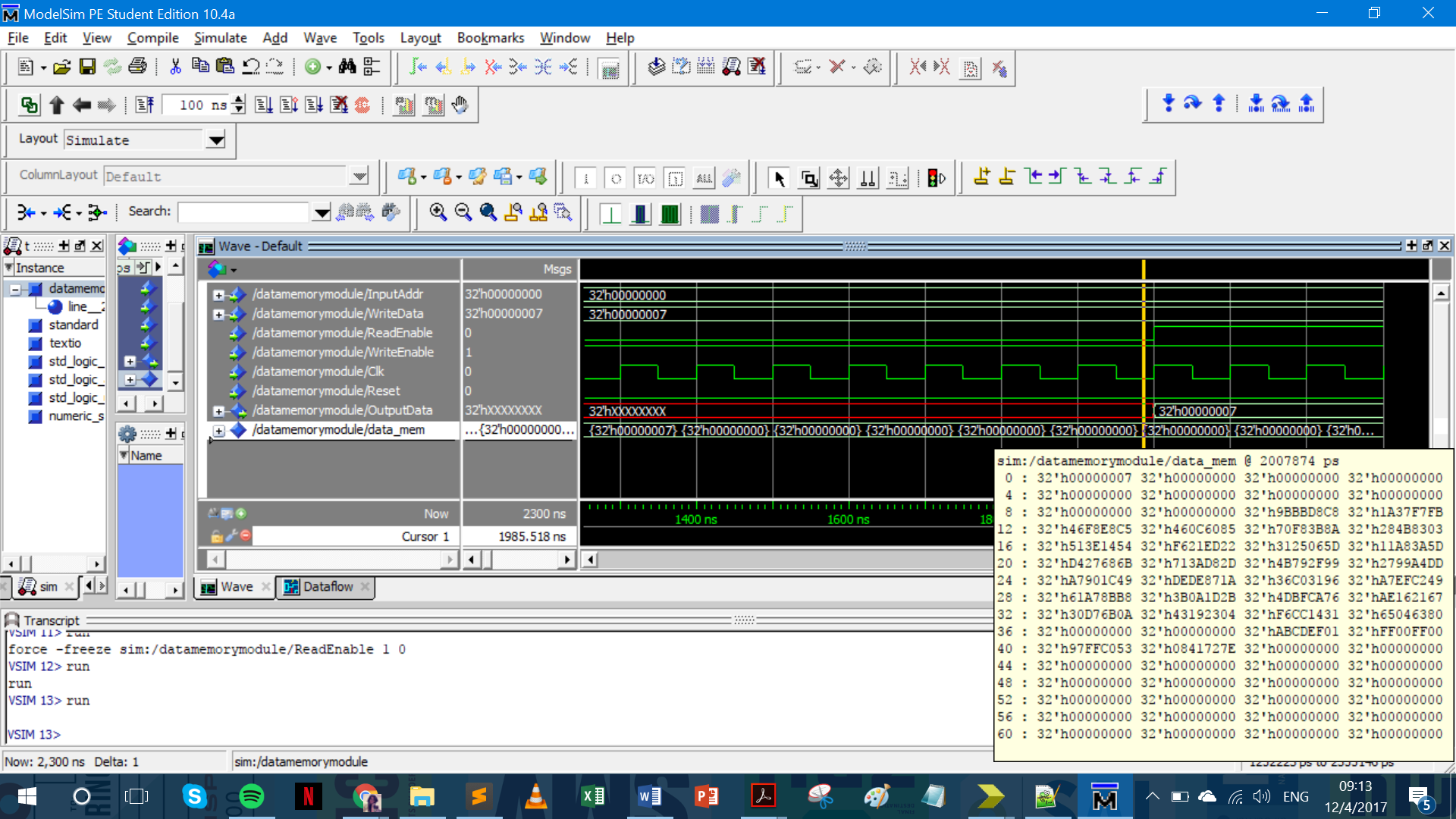
Some of the test case simulations are done using a testbench and some are done by simply forcing the values.

* To write data and read output.



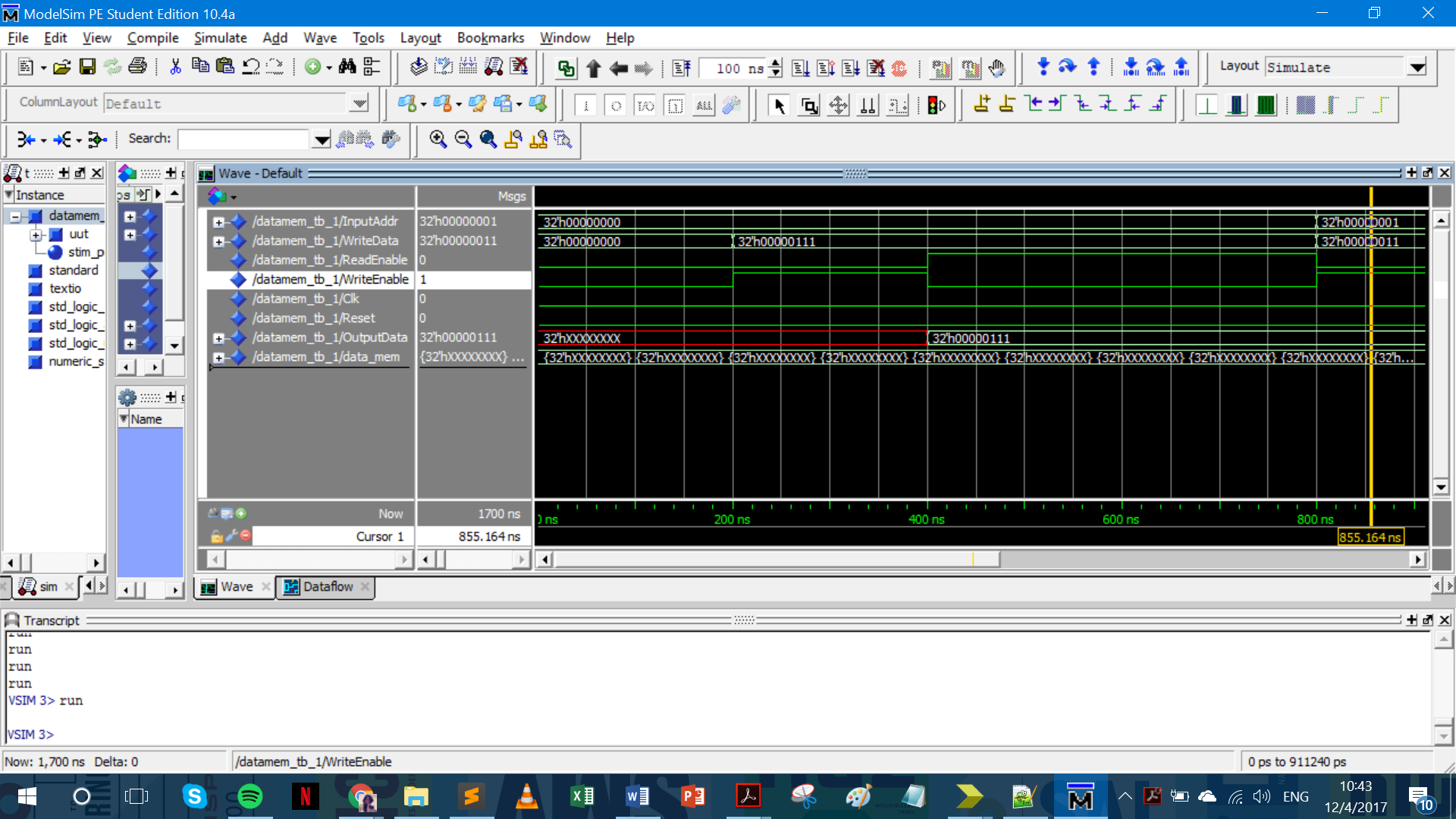
Here I want to write data to Data Memory location 32h’0, i.e. the first location. The data to be written is 32h’07.

So, first I made the read enable ‘0’ (disabled reading) and wrote the data in the memory location that we want. Then once I ran a couple of those cycles, I made read enable ‘1’ to read the output data. The output data is the data shown of the memory location where changes are made.



In the image above, you can see that the 0th memory location has the new data, 32h’00000007 written in it (encircled in red).

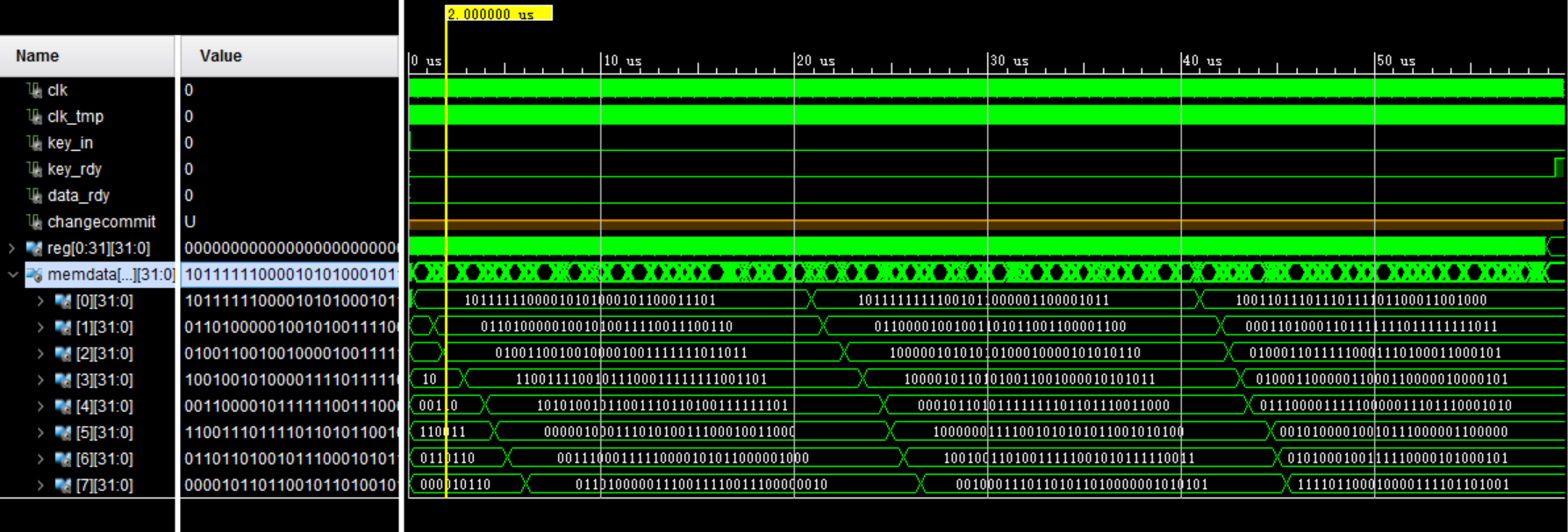
* To read data



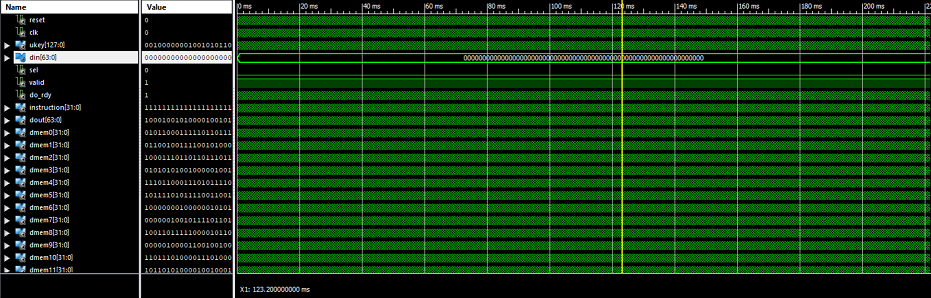
1. **Simulation of NYU 6463 Processor**

We have tried the Functional and the Timing Simulation for the NYU-6463 Processor. The screenshots of the same have been captured and shown below. D output is showing in Reg[1] and Reg[2].

**5.1.Functional Simulation:**

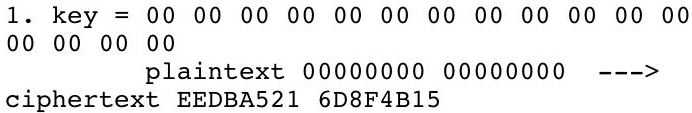


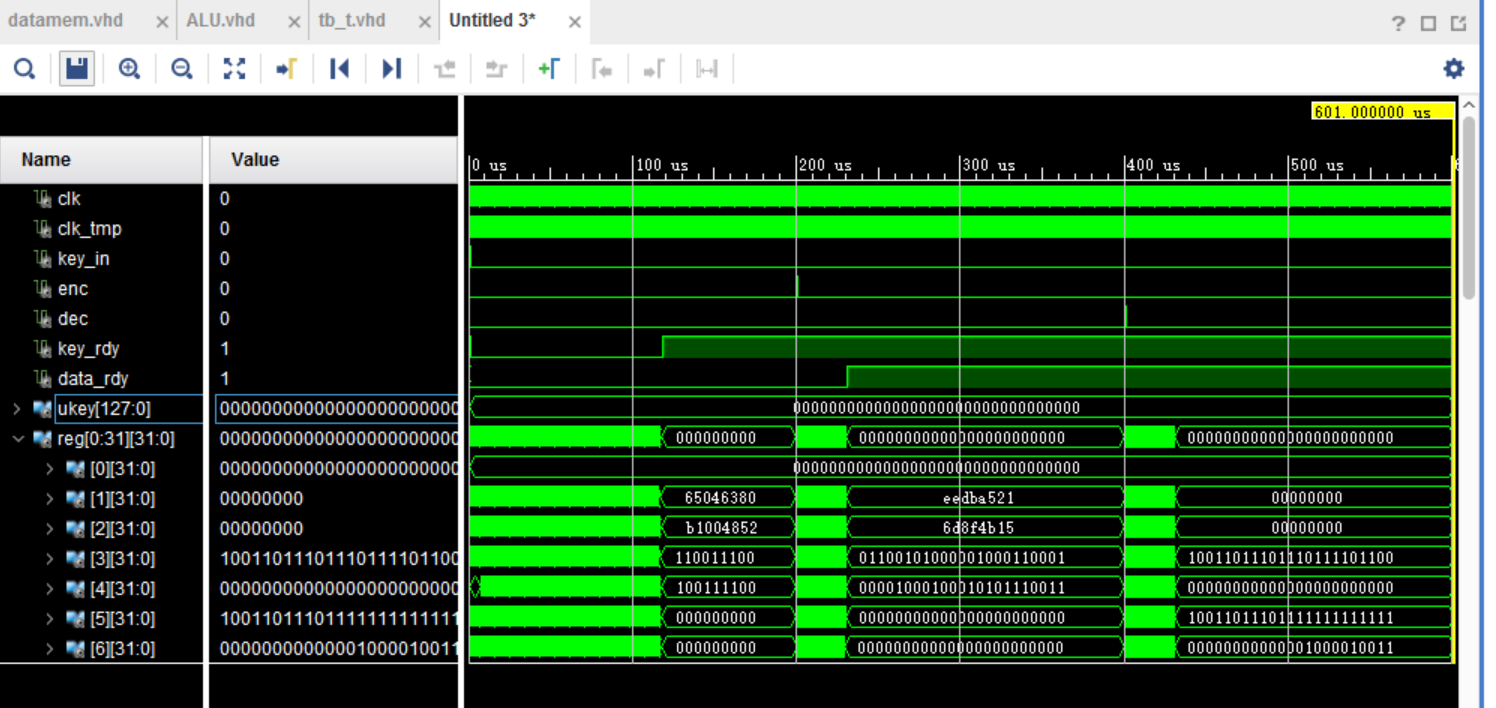
We also do 1000 test vectors for function simulation. The result is as below:

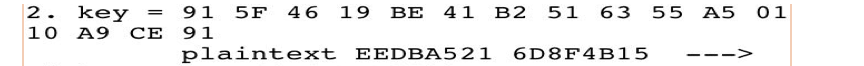


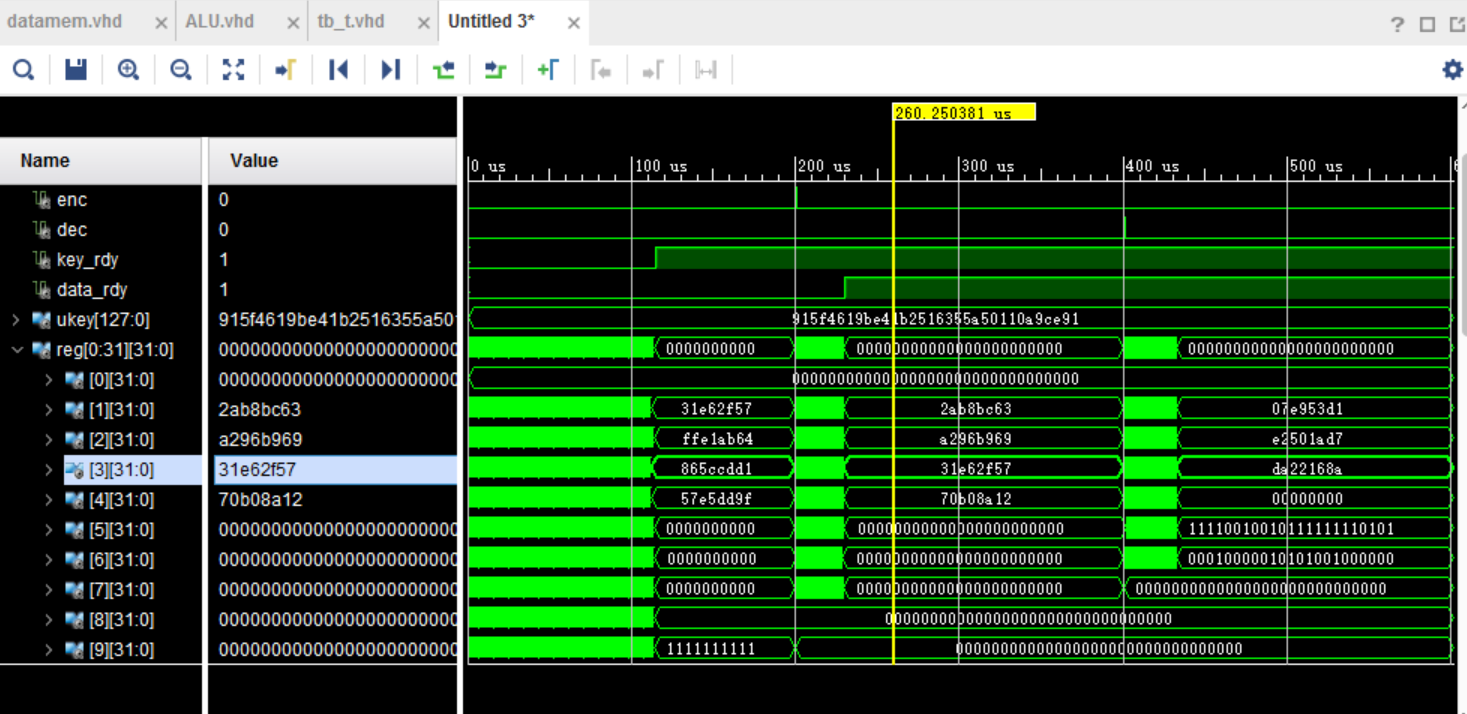
**5.2.Timing Simulation:**

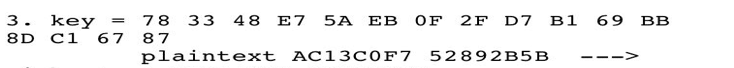
**Test Vectors:**

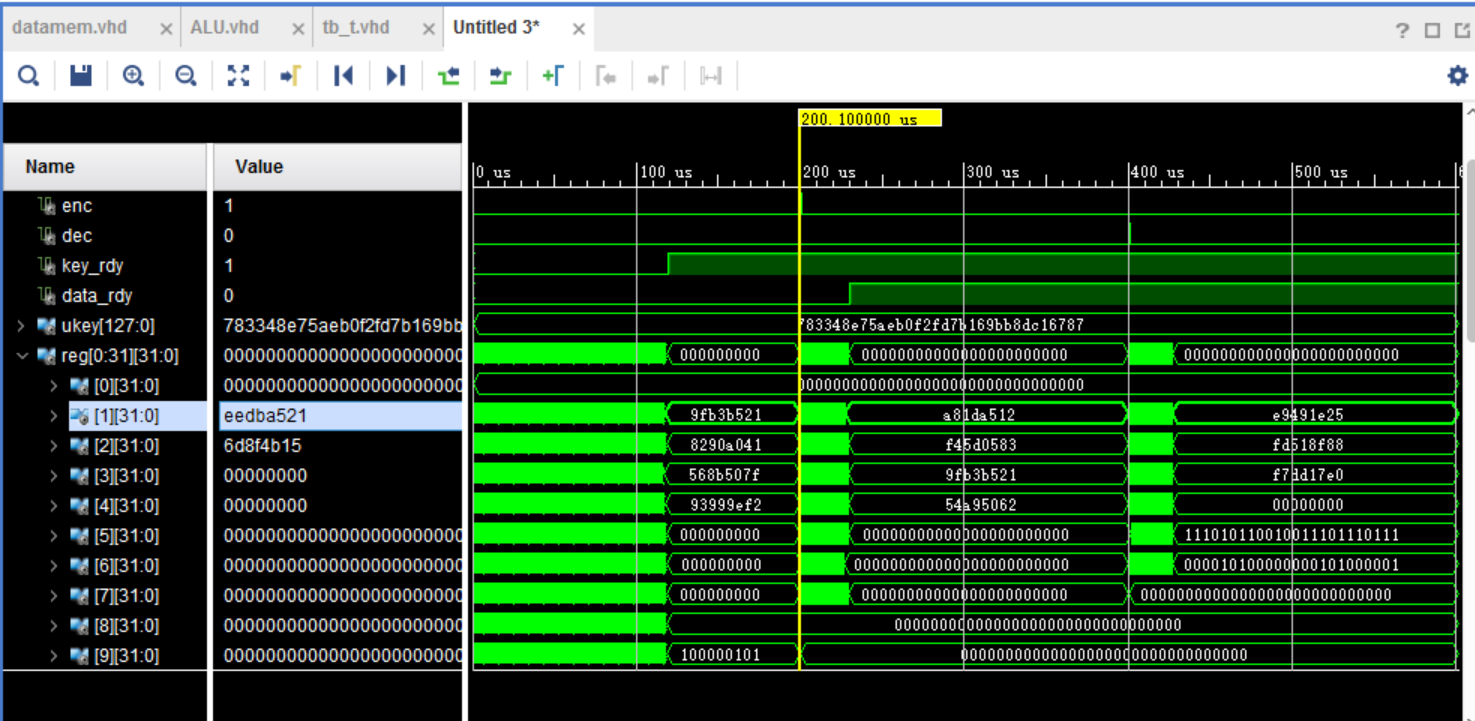
****

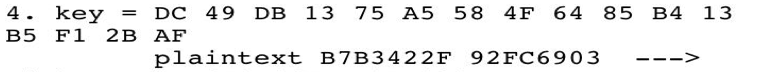
****

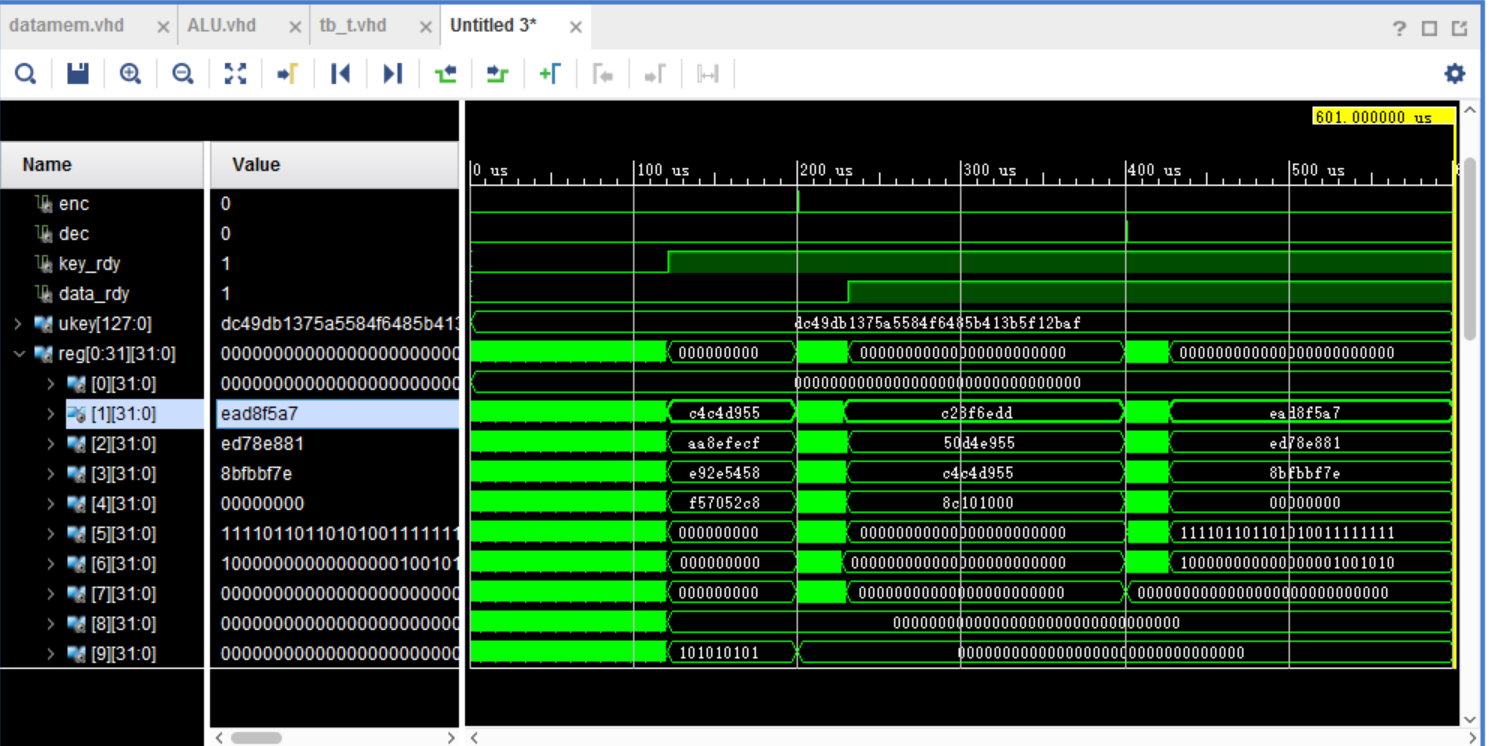
****

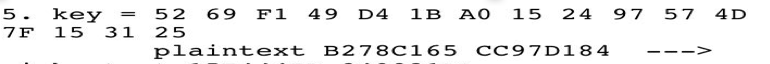
****

****

****







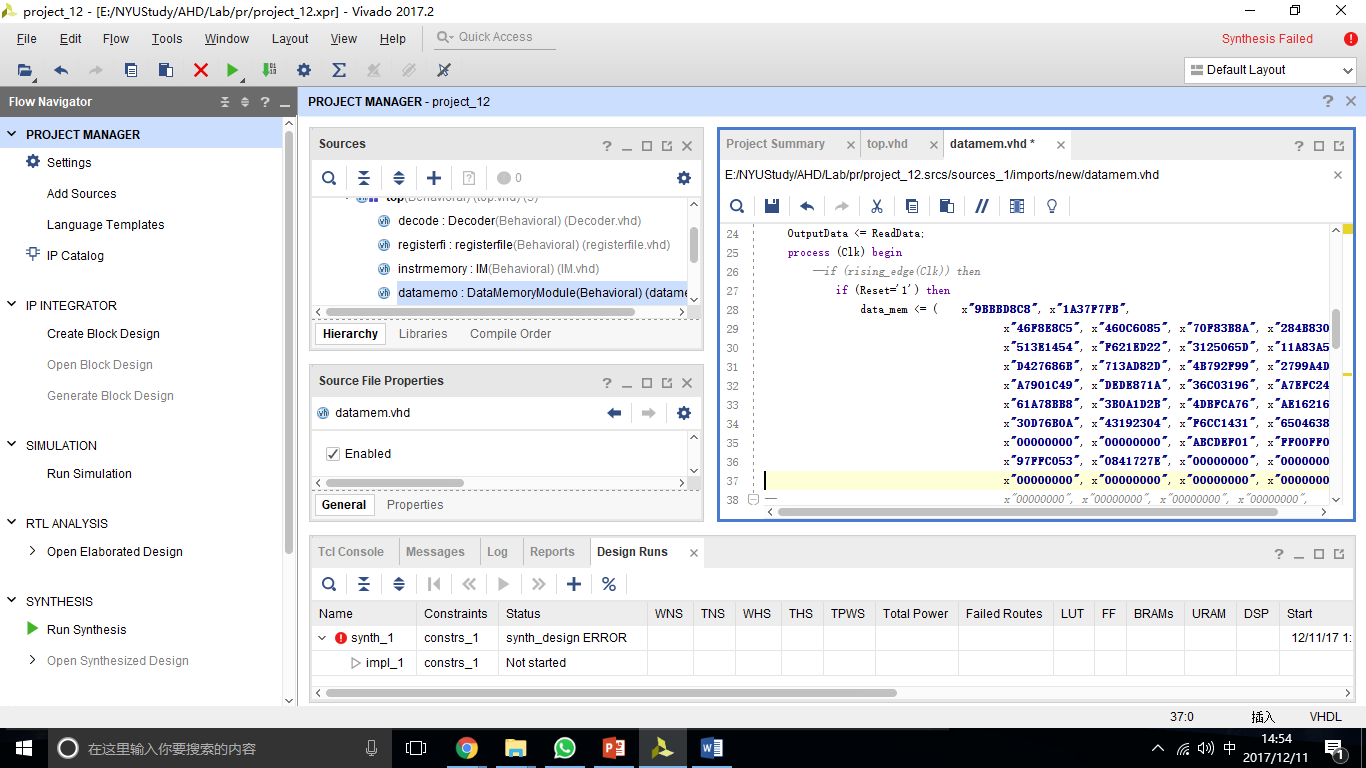


Note: Ours key array initialization is different from RC5 paper.We have check out output From LAB6 files.

1. **Implementation of RC5 on the Processor**

Initially created the RC5 Assembly Code which is submitted with all the other Project Codes. The implementation of RC5 has been carried out successfully on the designed processor. All the three cycles of RC5 were implemented successfully.

The initial S-array, as declared in the program, is shown in the screenshot below.



The RC5 Assembly Code stores in instruction memory, ready to be called. Explanation of it as follow:

1. **Encryption:**

In RF:  
R[1] represents A  
R[2] represents B  
R[3] stores S[2\*i] or S[2\*i+1]  
R[4] stores tmp ((A XOR B) <<< B) or ((B XOR A) <<< A)   
R[5] is left shift counter  
R[6] is the main loop counter: i  
R[12] stores 12 -- the main for loop cycles  
In DataMem:  
Mem[0]to Mem[26] store S[0] to S[26]  
========================================================================  
lw $0, $3, 0       //R[3] <= S[0]  
add $1, $3, $1     //A <= A + S[0]  
lw $0, $4, 1       //R[4] <= S[1]  
add $2, $4, $2     //B <= B + S[1]   
addi $0, $12, 12   //R[12] =12  
////for i=1 to 12 do ///  
or $1, $2, $4          
and $1, $2, $5      
sub $4, $5, $4     //tmp <= A xor B

andi $2, $5, 31    // last 5 bits of B  
addi $6, $6, 1     // counter<=1  
////((A XOR B) <<< B)///  
beq $0, $5, 3      //if R[5]==0: PC=PC+1+3           
SHL $4, $4, 1      //R[4] = R[4]<<1  
SUBI $5, $5, 1     //R[5] = R[5]-1  
BNE $0, $5, -3     //if R[0]!=R[5]: PC=PC+1-3         //tmp <= tmp<<B  
SHL $6, $7, 1      //R[7] <= 2\*i  
LW $7, $3, 0       //R[3] <= S[2\*i]  
hal  
ADD $4, $3, $1     //A<= tmp + S[2\*i]  
  
OR $1, $2, $4     
and $1, $2, $5     
sub $4, $5, $4     //tmp <= A xor B  
andi $1, $5, 31    //last 5 bits of A  
////((B XOR A) <<< A) ///  
beq $0, $5, 3      //if R[5]==0: PC=PC+1+3  
SHL $4, $4, 1      //R[4] = R[4]<<1  
SUBI $5, $5, 1     //R[5] = R[5]-1  
BNE $0, $5, -3     //if R[0]!=R[5]: PC=PC+1-3         //tmp <= tmp<<A  
addi $7, $7, 1     //R[7] <= 2\*i+1  
LW $7, $3, 0       //R[3] <= S[2\*i+1]  
hal  
add $4, $3, $2     //B<= tmp + S[2\*i+1]  
bne $12, $6, -26   //if R[6]! = 12: PC=PC+1-26  
hal

1. **Decryption:**

In RF:  
R[1] is A  
R[2] is B  
R[3] stores S[2\*i] or S[2\*i+1]  
R[4] is right shift counter  
R[12] is the main loop counter: i  
DataMem:  
Mem[0]to Mem[26] store S[0] to S[26]  
======================================================================  
addi $0, $12, 12 //R[12]=R[0] + 12        //R[12]=12  
shl $12, $13, 1      //R[13]=R[12]<<1        //R[13]=2\*i  
addi $13, $14, 1    //R[14]=R[13] + 1          //R[14]=2\*i+1  
lw $14, $3, 0        //R[3]=Mem[R14] //R[3]=S[2\*i+1]  
hal                  //HALT  
sub $2, $3, $2       //R[2]=R[2]-R[3]           //B<=B-S[2\*i+1]  
andi $1, $4, 31      //R[4]=R[1] and 0(\*27)11111          //R[4]=last 5 bits of A        
   
////((B-S[2\*i+1]) >>> A) ////////////////////////////////////  
beq $0, $4, 3        //If (R[0]==R[4]) then PC=PC+1+3       
shr $2, $2, 1        //R[2]=R[2]>>1  
subi $4, $4, 1       //R[4]=R[4] - 1                      //i=i-1  
bne $0, $4, -3       //If (R[0]!==R[4]) then PC=PC+1-3    //B<=B>>>A  
  
or $1, $2, $5        //R[5]=R[1] or R[2]  
and $1, $2, $6       //R[6]=R[1] and R[2]  
sub $5, $6, $2       //R[2]=R[5]-R[6]                     //B=B XOR A  
  
sub $3, $3, $3       //R[3]=R[3]-R[3]                     //R[3]=0  
lw $13, $3, 0        //R[3]=Mem[R13]                      //R[3]=S[2\*i]  
hal                  //HALT  
sub $1, $3, $1       //R[1]=R[1]-R[3]                     //A<=A-S[2\*i]  
andi $2, $4, 31      //R[4]=R[2] and 0(\*27)11111          //R[4]=last 5 bits of B  
  
////((A-S[2\*i]) >>> B) ////////////////////////////////////  
beq $0, $4, 3        //If (R[0]==R[4]) then PC=PC+1+3  
shr $1, $1, 1        //R[1]=R[1]>>1  
subi $4, $4, 1       //R[4]=R[4] - 1                      //i=i-1  
bne $0, $4, -3       //If (R[0]!==R[4]) then PC=PC+1-3    //A<=A>>>B  
  
or $1, $2, $5        //R[5]=R[1] or R[2]  
and $1, $2, $6       //R[6]=R[1] and R[2]  
sub $5, $6, $1       //R[1]=R[5]-R[6]                     //A=A XOR B  
  
subi $12, $12, 1     //R[12]=R[12] - 1  
bne $12, $0, -27    //If (R[12]!==R[0]) then PC=PC+1-27  
  
sub $3, $3, $3       //R[3]=R[3]-R[3]                     //R[3]=0  
lw $0, $3, 1         //R[3]=Mem[1]                        //R[3]<=S[1]  
hal                  //HALT  
sub $2, $3, $2       //R[2]=R[2]-R[3]                     //B<=B-S[1]  
  
sub $3, $3, $3       //R[3]=R[3]-R[3]                     //R[3]=0  
lw $0, $3, 0         //R[3]=Mem[0]                        //R[3]<=S[0]  
hal                  //HALT  
sub $1, $3, $1       //R[1]=R[1]-R[3]                     //A<=A-S[0]  
hal                  //HALT

1. **Key\_expansion:**

In Register:  
R[1] is A  
R[2] is B  
R[3] stores S[i]  
R[4] stores L[j]  
R[5] left shift counter   
In DataMem:  
Mem[0] to Mem[26] store S[0] to S[26]  
Mem[27] to Mem[30] store L[0] to L[3]  
---------------------------------------------------------------------------------------------------------------------------------  
addi $0, $10, 26   //R[10] = R[0] + 26                         
addi $0, $11, 4    //R[11] = R[0] + 4  
addi $0, $12, 78   //R[12] = R[0] +78      //DO 78 TIMES  
lw $6, $3, 0       //R[3] = M[0 + R[6]]    // Load S[i]  
hal                 
add $3, $1, $9     //R[9] = R[3] + R[1]                         
add $9, $2, $9     //R[9] = R[9] + R[2]   //S[i] = S[i] + A + B   
shl $9, $9, 3      //R[9] = R[9] << 3     // left shift S[i] + A + B by 3     
add $0, $9, $1     //R[1] = R[0] + R[9] //A = S[i] = (S[i] + A + B) <<< 3   
sw $6, $1, 0       //M[0 + R[6]] = R[1]   // Store A to S[i]  
hal  
add $1, $2, $8     //R[8] = R[1] + R[2] //A+B  
andi $8, $8, 31    //R[8] = R[8] AND 0(\*27) 11111     // last 5 bits of A+B  
lw $7, $4, 26      //R[4] = M[26 + R[7]] //load L[j]   
hal  
add $4, $1, $9     //R[9] = R[1] + R[4] // A + L[j]  
add $9, $2, $9     //R[9] = R[9] + R[2] // A + B + L[j]  
beq $0, $8, 3      //IF (R[0] == R[8]) THEN PC = PC + 1 + 3

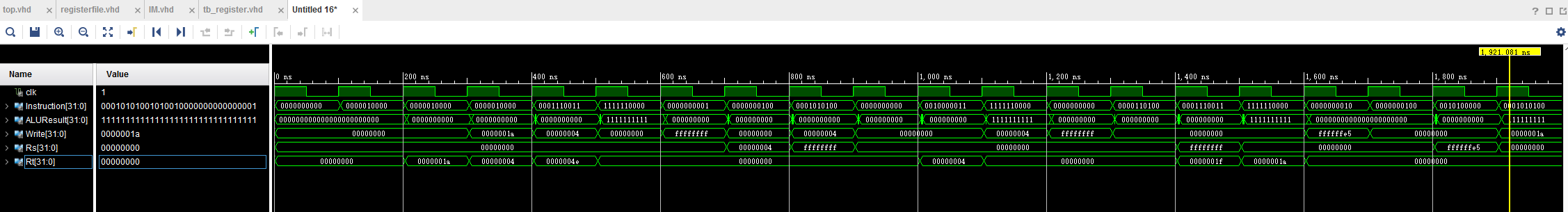
//loop to left rotate  
shl $9, $9, 1      //R[9] = R[9] << 1  
subi $8, $8, 1     //R[8] = R[8] - 1 //Decrement  
bne $0, $8, -3     //IF (R[0] != R[8]) THEN PC = PC + 1 - 3    //compare                        
add $0, $9, $2     //R[2] = R[0] + R[9] // R[2] = B  
sw $7, $2, 26      //M[26 + R[7]] = R[2] //store B to L[j]  
hal  
addi $6, $6, 1     //R[6] = R[6] + 1                   
addi $7, $7, 1     //R[7] = R[7] + 1                
bne $6, $10, 1     //IF (R[6] != R[10]) THEN PC = PC + 1 + 1

//i = (i + 1) mod (26)  
sub $6, $6, $6     //R[6] = R[6] - R[6]  
bne $7, $11, 1     //IF (R[7] != R[11]) THEN PC = PC + 1 + 1  //j = (j + 1) mod (4);  
sub $7, $7, $7     //R[7] = R[7] - R[7]  
addi $5, $5, 1     //R[5] = R[5] + 1  
bne $5, $12, -29   //IF (R[5] != R[12]) THEN PC = PC + 1 - 29  // check 78 loop

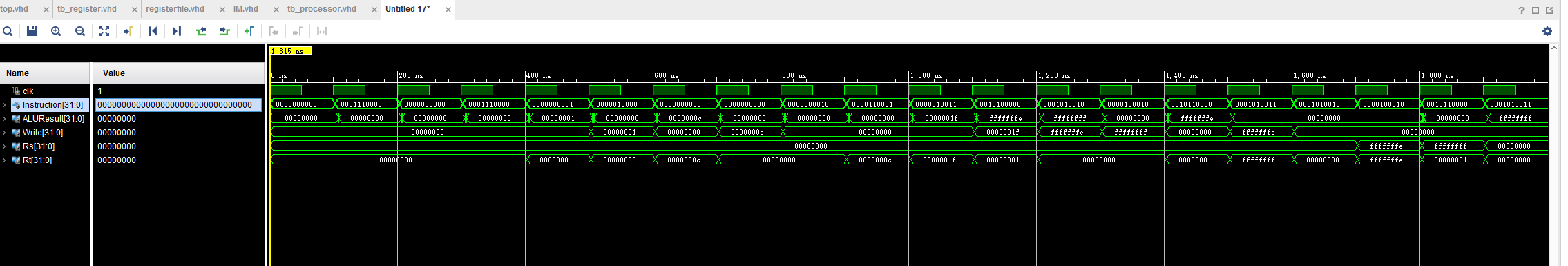
The RC5 machine code is placed in instruction memory, we simply run the processor, then we get this three-part timing simulation.

For ukey= 0 & din=0:

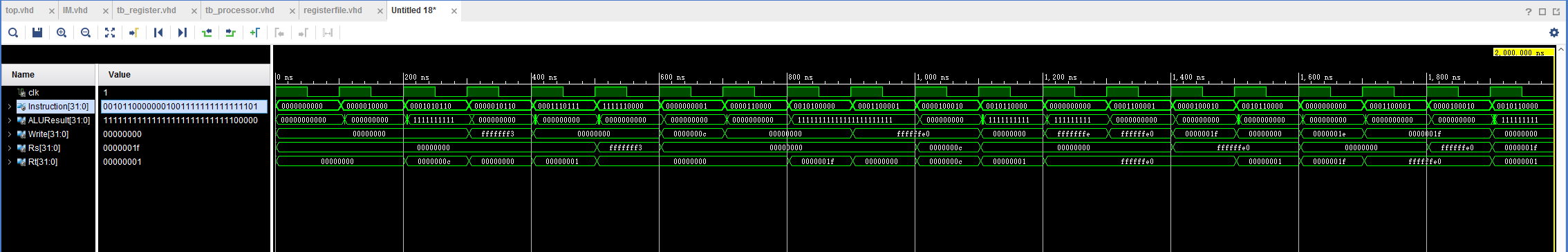
**Key Expansion Timing simulation:**

****

**RC5 Encryption Timing simulation:**

****

**RC5 Decryption Timing simulation:**

****

**Clock Cycles Required (Summary):**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Encryption | Decryption | Key Expansion | Total Operation |
| Clock Cycles | 1248 | 1272 | 5442 | 7962 |

1. **Performance Analysis**
   1. **Resource Utilization:**

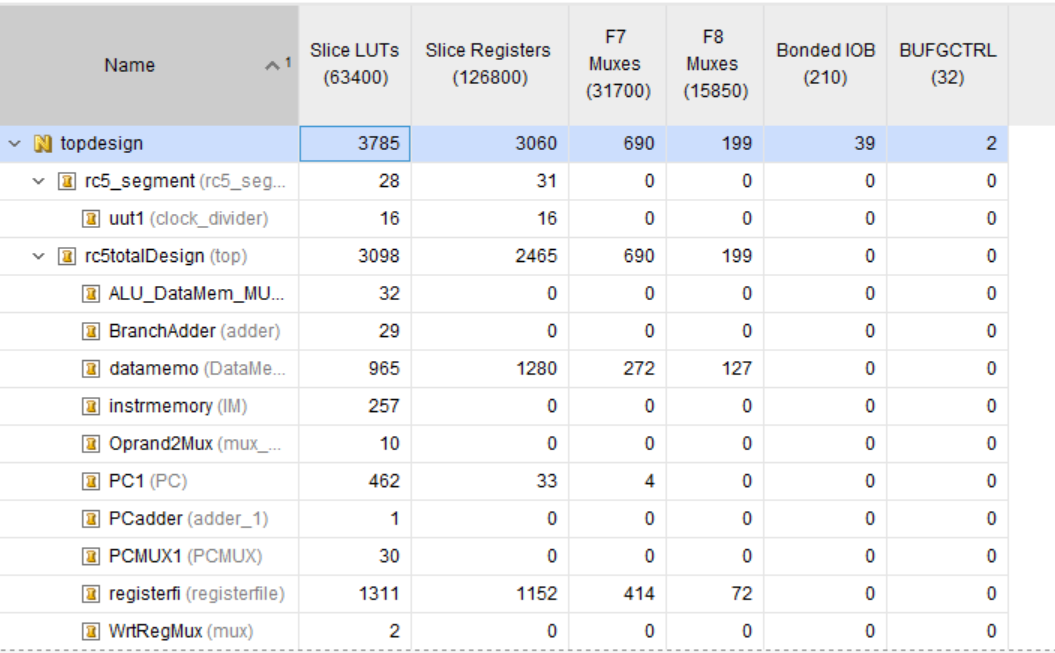
Timing numbers generated by Synthesizing the project gives only a synthesis estimate.

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. Since the circuit is pretty small, due to optimization the PAR Area is smaller than synthesis.

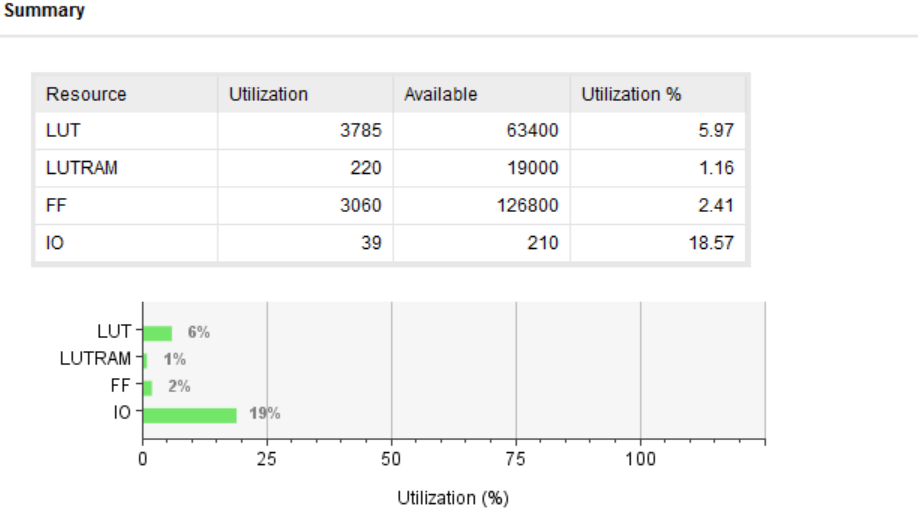
**NYU-6463 Processor Details**

Utility Report:



LUT Summary:

The total LUT utilization in the Processor is **approximately 6%**.



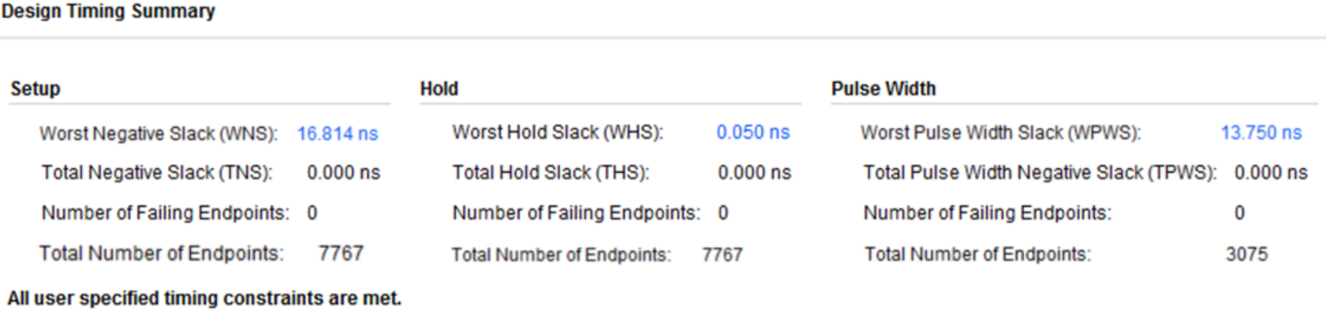
ALU

|  |  |  |
| --- | --- | --- |
|  | Synthesis Stage | Place and Route Stage |
| LUT FF Pairs Used | 322 | 306 |
| Fully Used LUT FF Pairs | 0 | 0 |
| Number of bonded IOBs | 99 out of 210 | 99 out of 210 |

Data Memory

|  |  |  |
| --- | --- | --- |
|  | Synthesis Stage | Place and Route Stage |
| Slice LUTs | 780 | 746 |
| LUT FF pairs used | 2794 | 2215 |
| LUT FF pairs with unused FFs | 714 | 135 |
| LUT FF pairs with unused LUTs | 2014 | 1469 |
| Fully used LUT-FF pairs | 66 | 611 |
| Number of bonded IOBs | 74 | 74 |

* 1. **Time Report:**



Critical path delay: 16.814 ns.

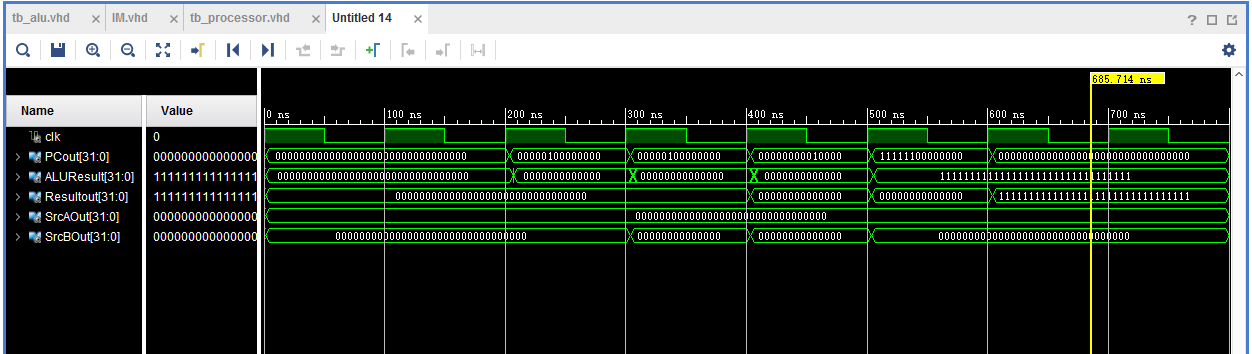
Max frequency of operation: 159.083MHz.

1. **Sample Codes**
   1. **Sample Code 1**

The Sample Code is as follows:

00000100000000010000000000000111 --ADDI R1, R0, 7 // R1 = 7 00000100000000100000000000001000 --ADDI R2, R0, 8 // R2 = 8 00000000010000010001100000010000 --ADD R3, R1, R2 // R3 = R1 + R2 =15 11111100000000000000000000000000 --HAL // HALT

The Sample Code 1 (above) provided by Professor has been tested and the following screenshot shows the simulation for the same.



* 1. **Sample Code 2**

The Sample Code is as follows:

ADDI R1, R0, 2

ADDI R3, R0, 10

ADDI R4, R0, 14

ADDI R5, R0, 2

SW R4, 2(R3)

SW R3, 1(R3)

SUB R4, R4, R3

SUBI R4, R0, 1

AND R4, R2, R3

ANDI R4, R2, 10

OR R4, R2, R3

LW R2, 1(R3)

ORI R4, R2, 10

NOR R4, R2, R3

SHL R4, R2, 10

SHR R4, R2, 10

BEQ R5, R0, -2

BLT R5, R4, -2

BNE R5, R4, 0

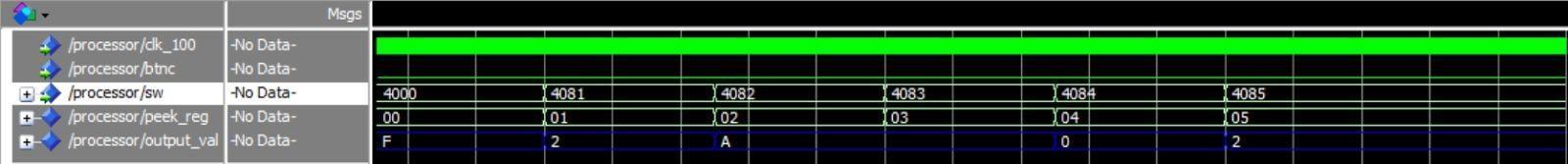
JMP 20

HAL

The above assembly code results,

R1 = 2, R2 = A, R3 = A, R4 = 0, R5 = 2

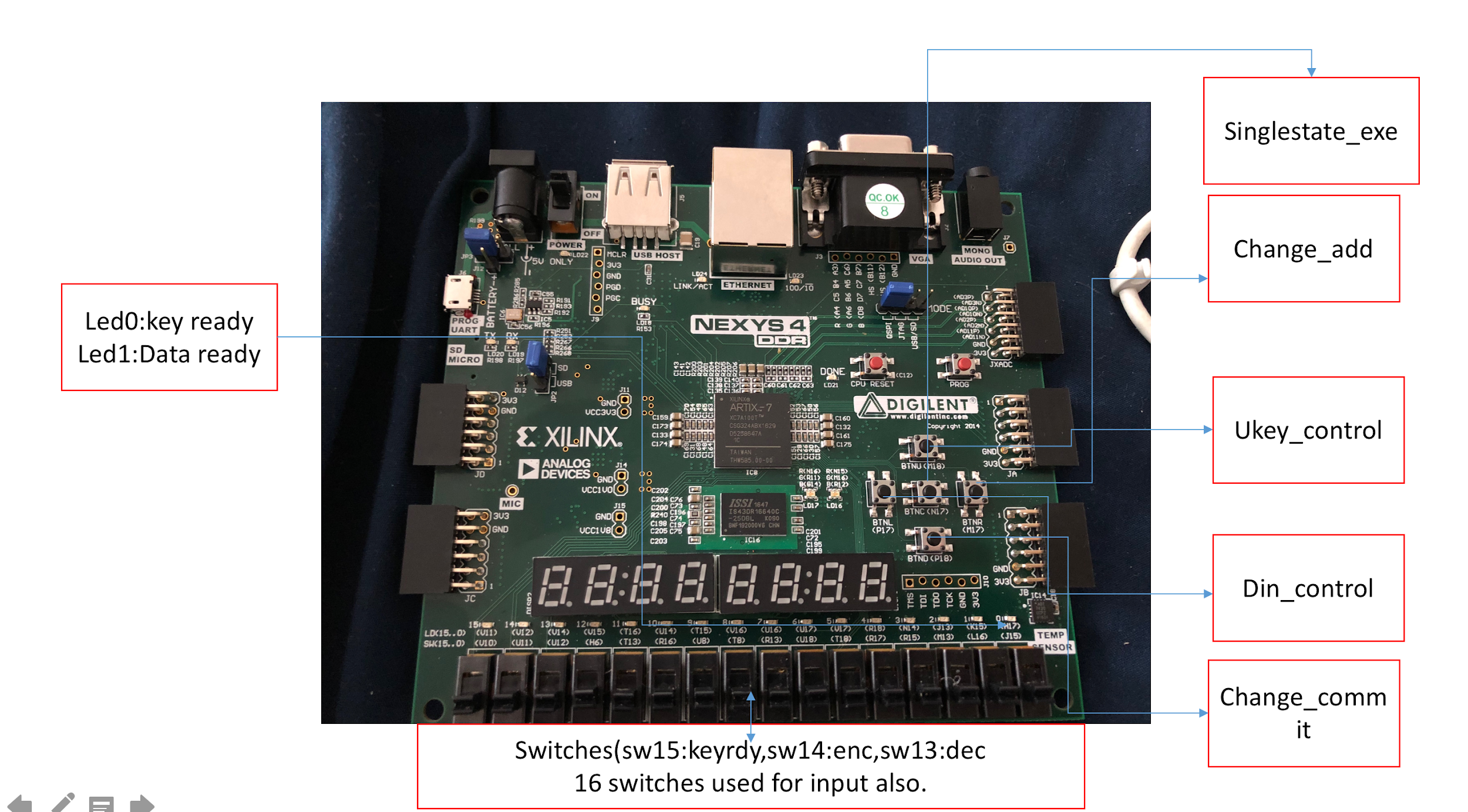
The Sample Code 2(above) provided by Professor has been tested and the following screenshot shows the simulation for the same.

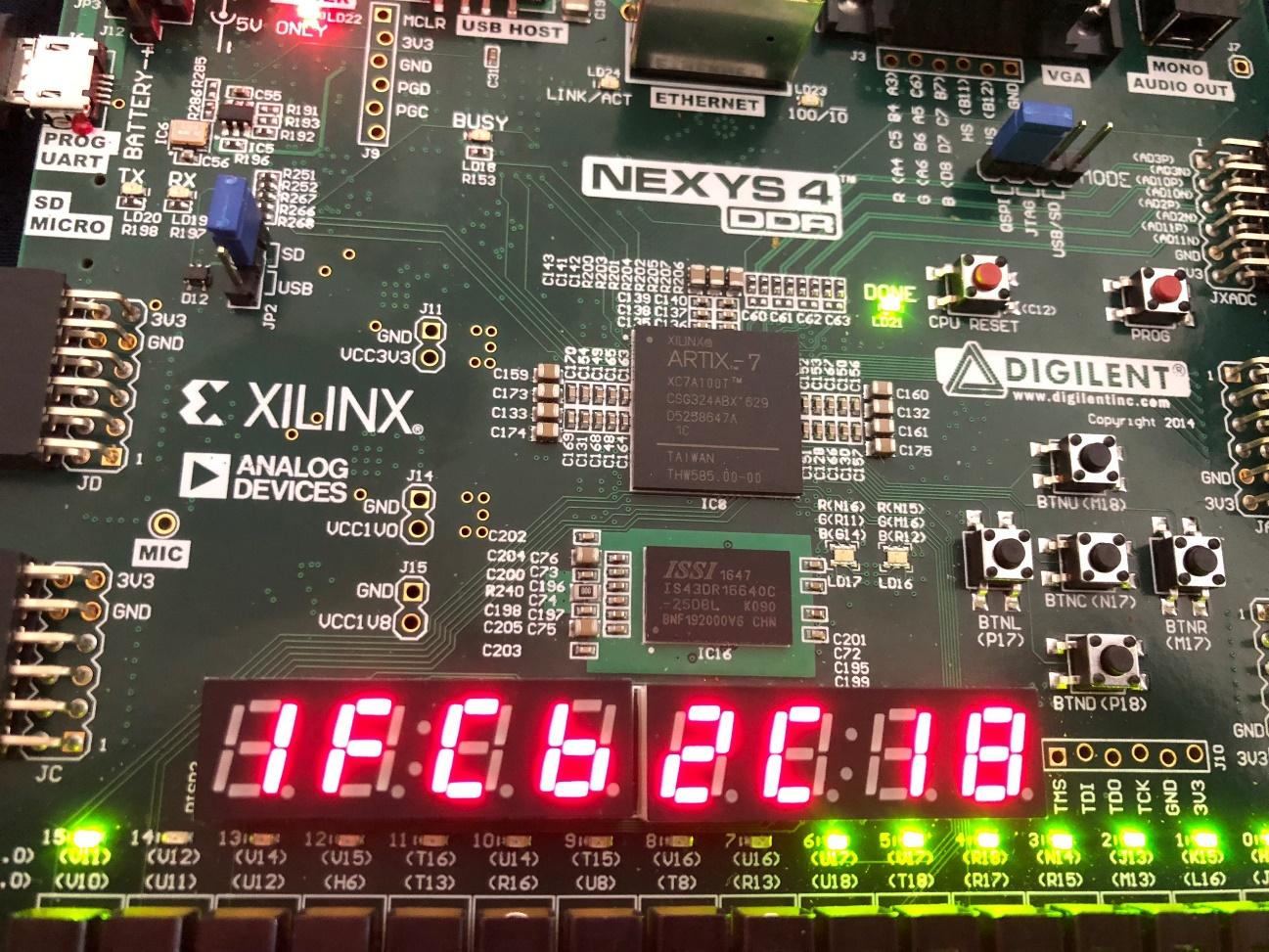


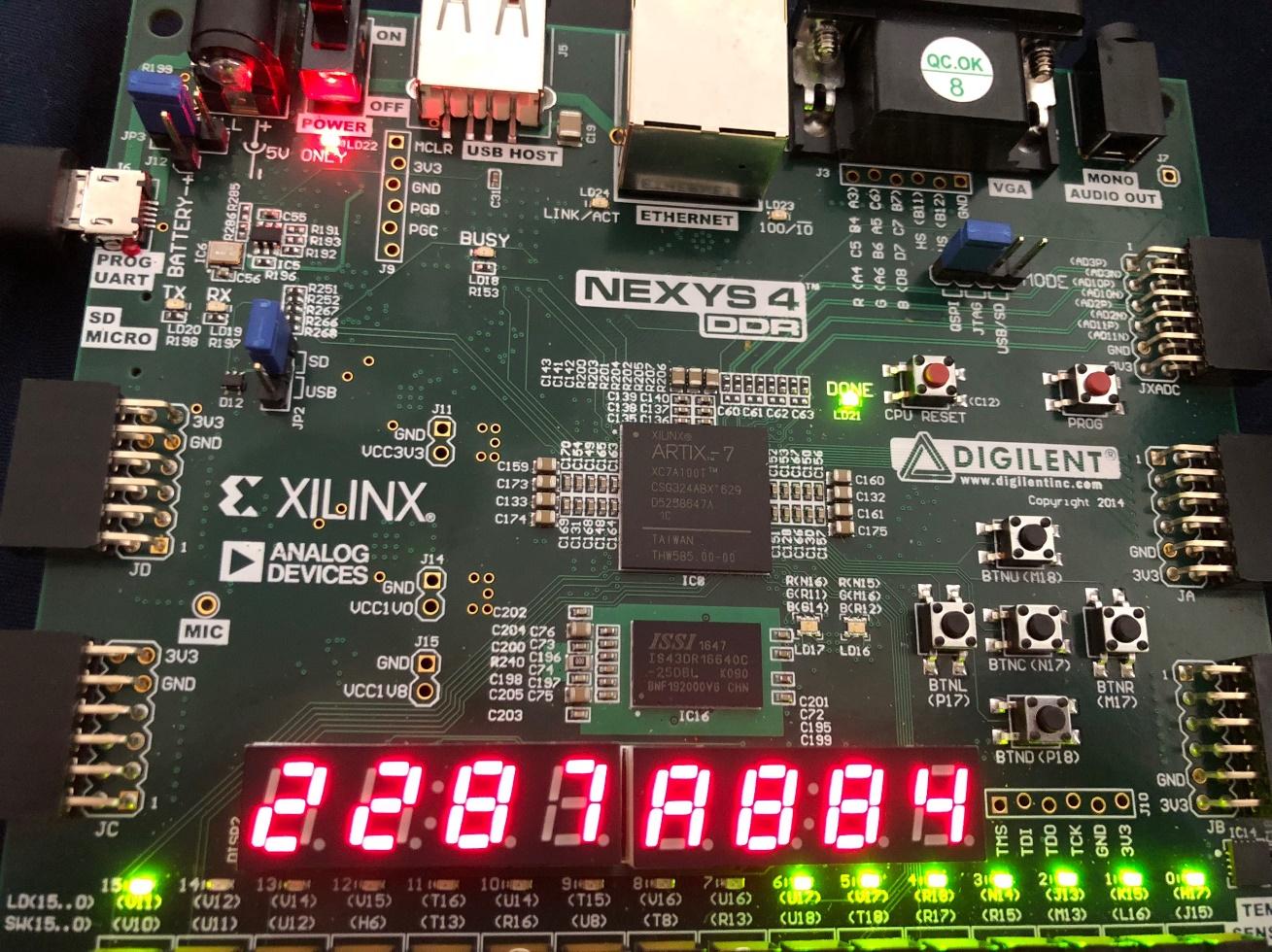
Test:

For our given TEST VECTOR:

Ukey:x”0000ee8812000043000550010022ee88” and Din:x”1234ffffffff5678”

Dout:x”1fcb2c182287a884”





1. **Links**

GitHub Link:<https://github.com/Nie13/6463processor>

Demo Video: <https://youtu.be/iMzTdjy12Sk>