NYU TANDON SCHOOL OF ENGINEERING

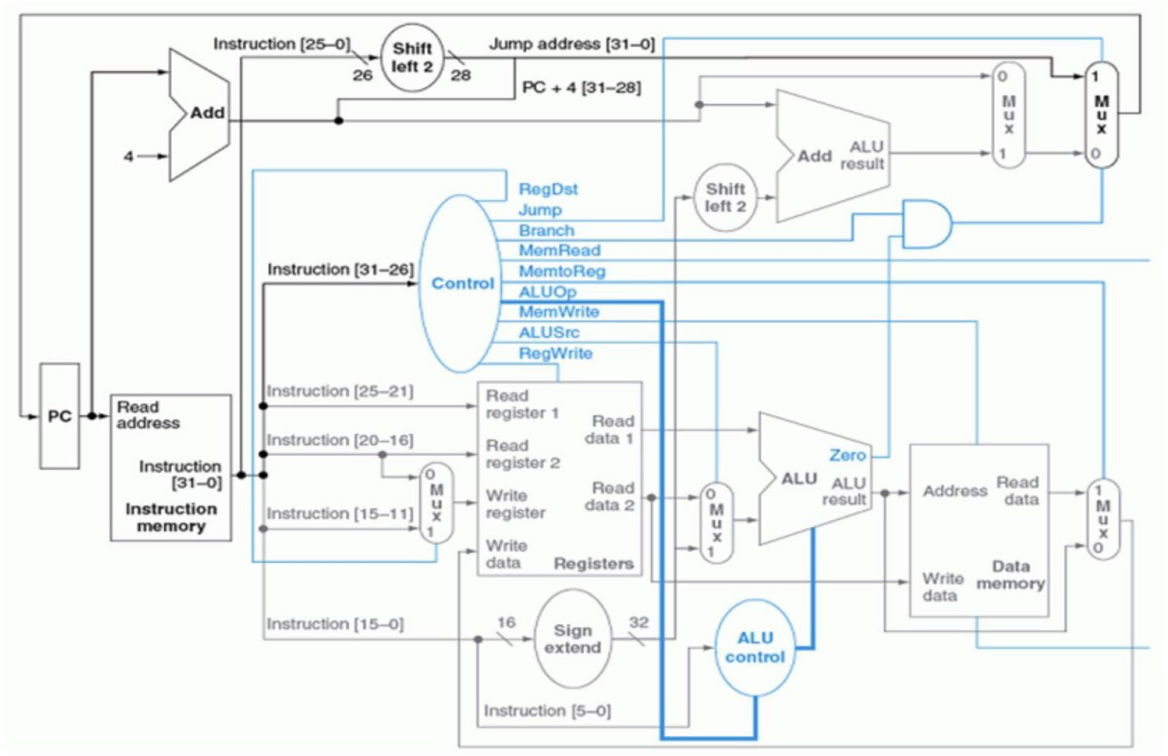
EL-GY 6463 Final Project Report

NYU-6463 Processor Design and RC5 Implementation

Mengyuan Zhu (mz1893)
Shuai Zhang (sz1950)
Yezhong Xu (yx1209)
Can Xu (cx461)
Johnny Chu (jc6875)
QiaoXin Xing (qx389)

Part I Introduction

For the advance computer hardware final project we are tasked with implementing a 32 bit processor in VHDL, creating all the individual components that make up the processor, and the following test benches for testing the processor and each of its individual components. The processor that we chose to simulate for this project is the following shown below:



The following architecture was taken off the slides in class. The core components that will be discussed in this report are the CPU main code, the program counter, the control unit, the data memory, the instruction memory, the ALU, and the register.

The CPU code
This is our top level code

Program counter
The program counter also known as the PC indicates where a computer is in its program sequence. The PC is incremented after fetching an instruction, and holds the memory address of the next instruction that would be executed. In the following code snippet shown below:

```
23
24  entity pc is
25      port(
26          clk: in std_logic;
27          address_to_load: in std_logic_vector(31 downto 0);
28          current_address: out std_logic_vector(31 downto 0)
29      );
30  end pc;
31
32  architecture Behavioral of pc is
33
34      signal address: std_logic_vector(31 downto 0):= "00000000000000000000000000000000";
35
36  begin
37
38      process(clk)
39          begin
40          current_address <= address;
41          if clk='1' and clk'event then
42              address <= address_to_load;
43          end if;
44      end process;
45
46  end Behavioral;
47
```

Instruction Memory

The instruction memory is the part of a CPU's control unit that holds the instruction currently being executed or decoded. The instruction to be executed is loaded into the instruction register which holds it while it is being decoded, prepared, and executed.

The control unit

The control unit is the component that fetching instructions from the memory and begins the execution of those instructions by directing the coordinated operations of the ALU, registers and other components.

The control unit sets up the behavior of all the MUXes used in the processor. It takes in the opcode from the top level CPU code main and configures MUX1 with reg_dest which tells the register to read or write. In MUX2 with alu_src it modifies the data headed into the ALU.
In mux3 it decides if the alu_result and read_data from data memory is going into the Register's Write Data. In MUX4 it is one of the factors in deciding whether or not to increment by 2 or 4. In MUX5 with jump it chooses between the result of mux4 and the jump address.

```
200
201     CONTROL1: control port map (
202         opcode => opcode,
203         reg_dest => reg_dest,
204         jump => jump,
205         branch => branch,
206         mem_read => mem_read,
207         mem_to_reg => mem_to_reg,
208         mem_write => mem_write,
209         alu_src => alu_src,
210         reg_write => reg_write,
211         alu_op => alu_op
212     );
213
```

Other components that it influences is the data memory, the register, and the ALU control. It tells the data memory whether or not it is in read or write mode with mem_read and

mem_write, tells the register when it is allowed to write with reg_write, and informs the ALU control of what operations to do.

Data Memory

The data memory is a register that stores the data that is being transferred to and from the immediate access storage. In our processor data memory takes in mem_write, mem_read, di_vld, din, and ukey from the Top level CPU code, alu_result from the ALU, and read_data_2 from the register. While sending out read_data, enc_out, and dec_out back to the information back to the top level CPU code main. It takes in

```
299
300     MEM: memory port map (
301         clk => clk,
302         clr => clr,
303         address => alu_result,
304         write_data => read_data_2,
305         MemWrite => mem_write,
306         MemRead => mem_read,
307         di_vld => di_vld,
308         din => din,
309         ukey => ukey,
310         read_data => mem_read_data,
311         enc_out => enc_out,
312         dec_out => dec_out
313     );
```

The ALU

The ALU or arithmetic logic unit is a combinational digital electronic circuit that performs arithmetic and bitwise operations on integer binary numbers. In this simulated processor the ALU is split into 2 parts, the ALU and the ALU control.

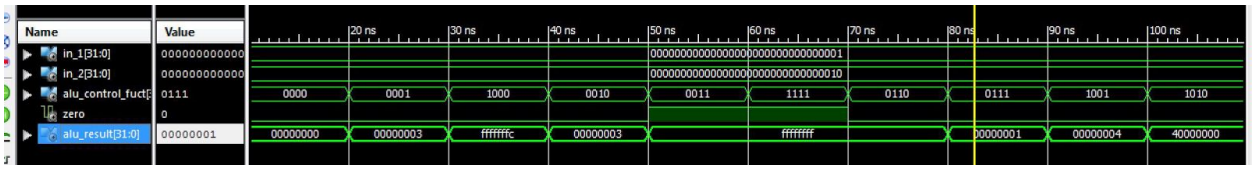
First is the ALU control, which takes in funct from the top level CPU main code and the alu_op from the control unit and tells the ALU what operation it should perform next.

```
233
234     ALU_CONTRL: alu_control port map (funct, alu_op, alu_control_fuct);
235
```

It is in the ALU where the actual operations are performed. It takes in alu_in_2 from the SGN_EXT mux that has modified data from the register's read_data_2 and read_data_1 from the register and performs the instructed operation sent over in alu_control_funct from the ALU control.

```
245
246     ALU1: alu port map (read_data_1, alu_in_2, alu_control_fuct, alu_zero, alu_result);
247
```

```
-- Stimulus process
stim_proc: process
begin
   -- hold reset state for 100 ns.
   --wait for 100 ns;
   in_1 <= x"00000001";
   in_2 <= x"00000002";

   -- insert stimulus here
   wait for 10 ns;
   alu_control_fuct <= "0000";
   wait for 10 ns;
   alu_control_fuct <=  "0001";
   wait for 10 ns;
   alu_control_fuct <= "1000";
   wait for 10 ns;
   alu_control_fuct <= "0010";
   wait for 10 ns;
   alu_control_fuct <= "0011";
   wait for 10 ns;
   alu_control_fuct <= "1111";
   wait for 10 ns;
   alu_control_fuct <= "0110";
   wait for 10 ns;
   alu_control_fuct <= "0111";
   wait for 10 ns;
   alu_control_fuct <=  "1001";
   wait for 10 ns;
   alu_control_fuct <=  "1010";
   wait for 10 ns;

   wait;
end process;
```

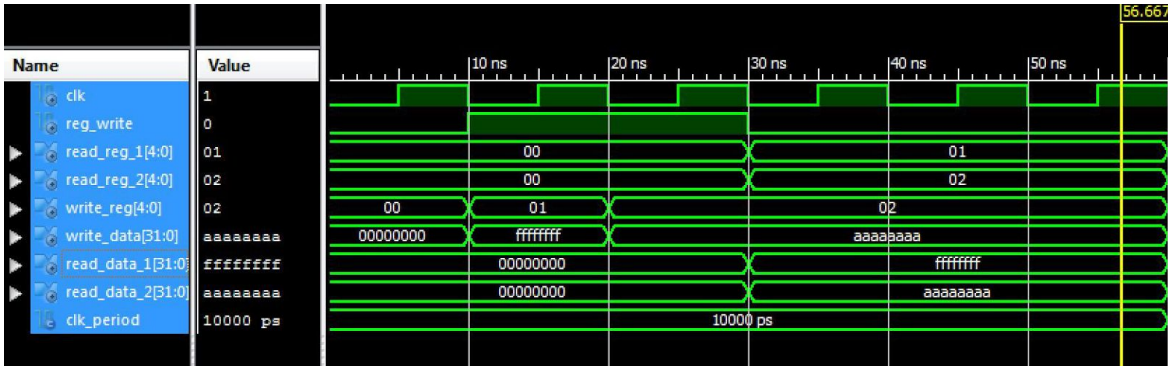


Register

A register is a small amount of storage that the CPU can quickly access to read or write whatever information it wants to store. It takes in data from the top level CPU main code in rs and rt and stores it in read_reg_1 and read_reg_2. Where it waits to be informed by the control unit on when to write with reg_write, MUX1 which chooses between rt and rs with write_reg, and MUX3 which chooses betwee.

```
222     REG: registers port map (
223         clk => clk,
224         clr => clr,
225         reg_write => reg_write,
226         read_reg_1 => rs,
227         read_reg_2 => rt,
228         write_reg => write_reg,
229         write_data => write_data,
230         read_data_1 => read_data_1,
231         read_data_2 => read_data_2
232     );
233
```

```
97      begin
98         -- hold reset state for 100 ns.
99         --wait for 100 ns;
100
101        wait for clk_period*1;
102
103        -- insert stimulus here
104        reg_write <= '1';
105        write_reg <= "00001";
106        write_data <= x"FFFFFFFF";
107        wait for 10 ns;
108        write_reg <= "00010";
109        write_data <= x"AAAAAAAA";
110        wait for 10 ns;
111        reg_write <= '0';
112        read_reg_1 <= "00001";
113        read_reg_2 <= "00010";
114
115
116        wait;
117     end process;
118
119  END;
```

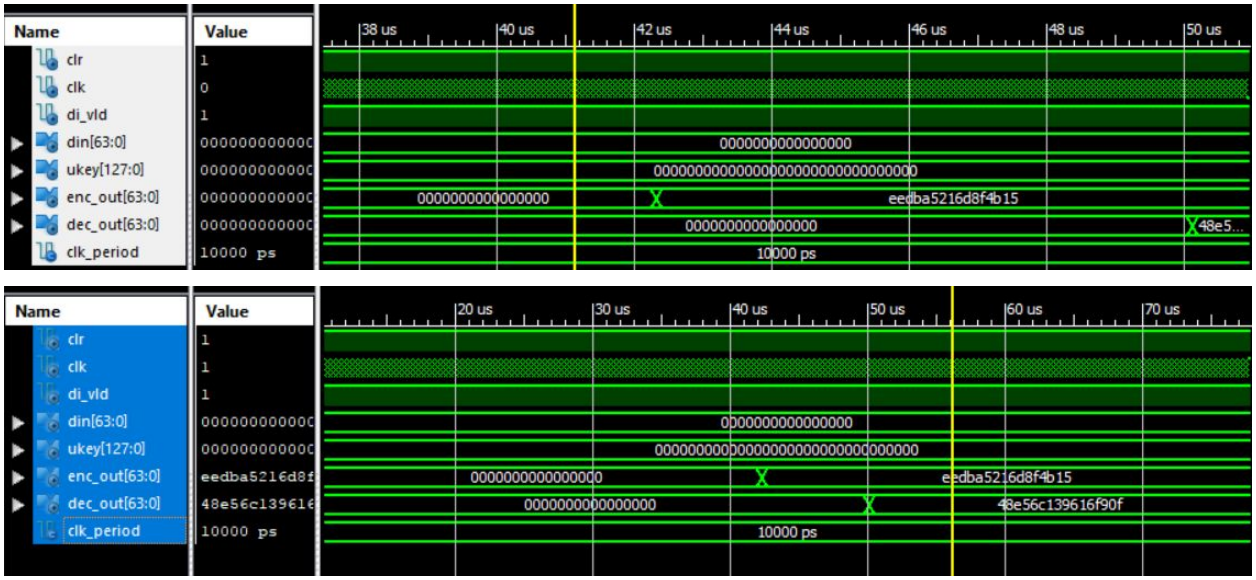


Part II Simulation screenshots

Function simulation(1000 test vectors)

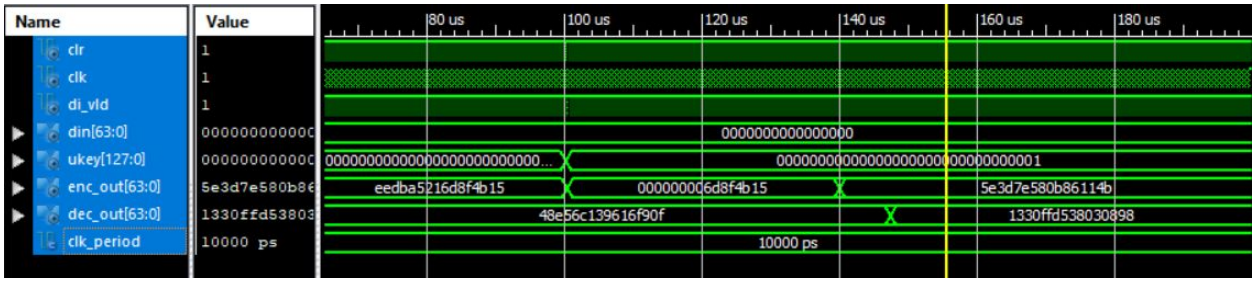
1. Ukey = X”00000000 00000000 00000000 00000000’’
   Din = X“00000000 00000000”

```
1.key=00000000000000000000000000000000
 plaintext 00 00 ----- encryption EEDBA521 6D8F4B15 ------decryption 48E56C13 9616F90F
```



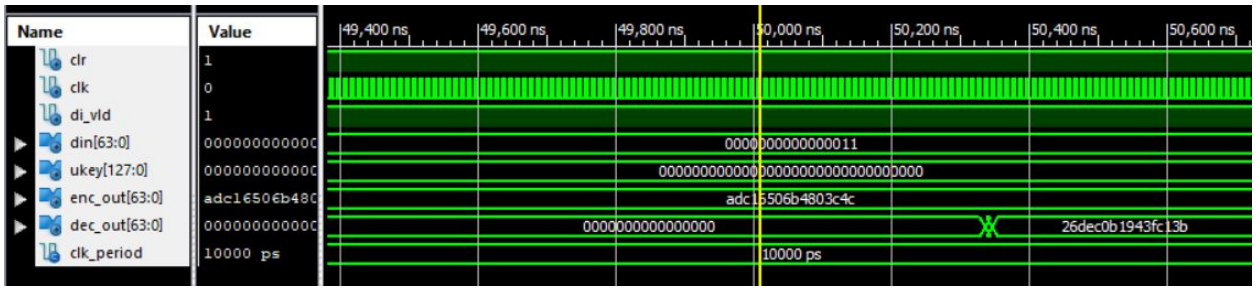
2.

   Ukey = X”00000000 00000000 00000000 00000001’’
   Din = X“00000000 00000000”



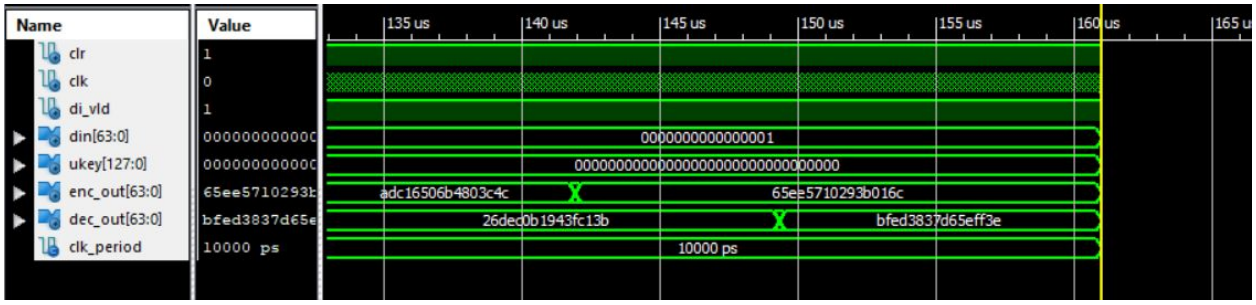
3.

```
1.key=00000000000000000000000000000000
 plaintext 00 11 ----- encryption ADC16506 B4803C4C ------decryption 26DEC0B1 943FC13B
```



4.

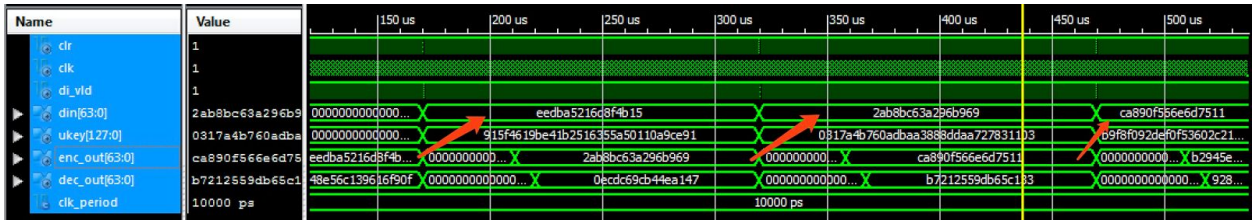
```
1.key=00000000000000000000000000000000
 plaintext 00 01 ----- encryption 65EE5710 293B016C ------decryption BFED3837 D65EFF3E
```

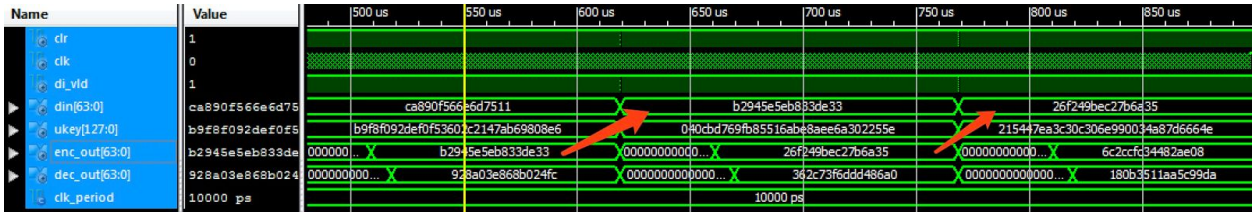


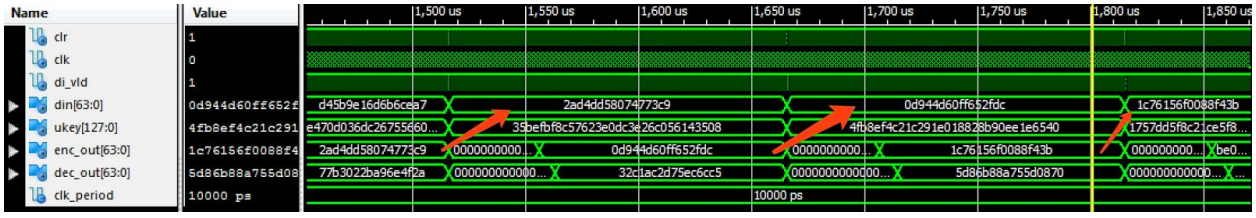
1000 Test Vectors:

```
10000 00000000000000000000000000000000 00000000 00000000
20000 915F4619BE41B2516355A50110A9CE91 EEDBA521 6D8F4B15
30000 783348E75AEB0F2FD7B169BB8DC16787 AC13C0F7 52892B5B
40000 DC49DB1375A5584F6485B413B5F12BAF B7B3422F 92FC6903
50000 5269F149D41BA0152497574D7F153125 B278C165 CC97D184
60000 2AB516CF7389A2532CBDAC8F09A5261B 15E444EB 249831DA
70000 D35E287696D8972EF364D8D2C73EA8A6 345FF946 1B419FF9
80000 E659A39507CDD7156DE3875D7A092305 4A27EF85 48ABCB1D
90000 9B9323BFEE4147A30D2FCDC3E3235C9B 656109CB CC75FD65
100000 3BD2F73E45C2866A606EAE5AAA10DEC2 BB507CB2 FAF7F22D
110000 A92344C77827ACF708C75ED3643991C7 7027EA27 2380640
120000 1ACD8F110C4795D5F2E3AB5908D111DD 6E6578CD 458C0633
130000 903E1166A74AC3A275D26DE615A07872 61413BB2 158AED3B
140000 39A30987519DB7F3BA7BB1EFD81F6993 806D97B3 8C77BA57
150000 513B9B4B2F3D7B07019F3D2BD2C799E7 9D7B2117 68E93871
160000 C2564774D06853C004686D988F10D3E0 C7EB4EC0 216B5712
170000 462055825896l9BE92A0192E3AA2EEBE BD2C1E3E 68707554
180000 C83B2E052F2D89E1E34DD771EF65C341 8AF0CB81 B649A5CF
190000 66A88E48123409EC64DE02549078CF84 82B48AA4 FD7A3BAE
200000 ABB8509CDF74C050B2A2E2600CEA4E60 D4369010 DBEDD56
210000 2EF735DF0ED36D1721EB06D72277AA97 A3D1D0E7 4943EEB1
220000 95240D7AAAEA9E36322CC7CE1770E086 AB198A46 7D25FEB4
230000 458CA4EAC0869C6A53BA18BE36D01F72 F0CB772 FA4E7C03
240000 164A14707034E5D048A6702C70F083E8 A3E30688 AD11287
250000 0E1E1732BC22D16A8AF2862EE6E870C2 D6591BC2 F9C1EC54
260000 7AC573A172077AF19683D9C56BE16611 511F271 755E943A
270000 D6469534A17C7350EC5281AC22325D40 DEA264F0 44C4568C
280000 A3C15DD38EF37C97C48B710BDC9FE367 38665DA7 8430D878
290000 9F195F81A2F9D8A1CAE7A4B90C751809 CF7560F9 D482A647
300000 71C8EB4A5380534E00ECC11A89CA969E 3A81377E 138A70C3
310000 F8DDC0659321DDB58C5303911D9F4B35 48461555 CE45332E
320000 0B1903BF56C935A75053E24FC8B36847 50FFD2E7 38946E28
330000 2B9D6913232BBE831B2B678B5B9519A3 3E312893 63223EBB
340000 ED8121DBD84D78B36C1B4897C08FD593 A70AD863 61CE619B
350000 F50C0C440FC8E3D42044D788EFD2E914 5EB35F84 6F2CE7C6
360000 AC6D432B3A9DA0A327B51607736F21BB F5AFEB1B 16F51BD3
370000 21A568DB8A0D0C4792D51753B78918B7 B6C42E77 907BF638
```

```
9590000 E1859275A6AF2AB10AE128A5635BA169 8C5CAF49 E9C1D18D
9600000 C59FC471F5C5687DC4237F01C235AAC5 FF8F2115 B828C0CB
9610000 104AED765F2E6AA67D886FD67F1C632E 3C45BFCE BF9E9C34
9620000 026527E344E37497035957AB1D531C2F 1992F65F 2DEEBEBC
9630000 257B43C7647567ABC33D9D8FB5C789BB 5E7852FB 25CA7086
9640000 3F1B5A515CDF542DE92D7B8500911D5D 5F6BF67D C7028969
9650000 692AD0DEE9A0C336DED84B6A964C1696 E193CD26 240C94C1
9660000 6DDF805B946D3D739F6D9A13BECD7813 F82D1433 B1E097E0
9670000 B245EF55283561A92B0D94355EADDCC1 E68E8BB1 C8C40676
9680000 55A0C45E1EC8498EAFD6506EAFAE22BE E5F21E5E 8A6C8502
9690000 710C908C954E5F80750E1C84444EAB80 68785040 6B750B32
9700000 B34C2AAA3F5E9276ACEC09BA08C41D0E 8A56C40E C8ED82EF
9710000 48D663541820E1E4ACC0B62CAE00AB84 399F1B54 D4C99029
9720000 01A3615BB28D10578CD35B3FA3B9A897 56C3857 FEF84807
9730000 B2DC1B70F7E47C18252ECBC0D3EC1A58 CE201BA8 49C6E6B1
9740000 3959B6B7D6A7CF17495D5C67B7ABAF57 DD58FB07 48B5746
9750000 08BBA47D05F31D391983535562571571 1DE434D1 8DAF2997
9760000 EE121AA057F8647838AC0D806A304B58 BCF21728 A5261482
9770000 413D24DD62A5AD8540EFC8ED9E19957D 4E960F4D 86B4CBC
9780000 20449F5CA8C0E66C9EE0A2546E7E1ED4 58803314 8F0558A
9790000 A7AE6544D7CAD7D4AF6E2F28A72E325C E07E5BEC 21D35B8D
9800000 165167C7E5393147F0B511736D7D3007 11FFBD47 22633755
9810000 C807158F1F69260BD153509F2CC57423 42536FC3 7E435860
9820000 B53D064F649749D7BD737963C453AA97 5C6CB537 9740D560
9830000 DEF646B436B636A40E66BB2499463684 25B4F014 BC8A2DE9
9840000 837AE3D47606CE48E144100C17284EB0 D0436F00 59BAE488
9850000 CF018CF9E81136412F152095DE811139 E2C69C89 9931985E
9860000 AB7A34BA0C2E458643D2BA86D5763606 C8AD3EF6 47246EF4
9870000 B3EE80B6E772B012F1D4317638EE4BC2 35E21982 4C525FF1
9880000 C2DA3F507622BFD0D250C7D82334B568 38CB2698 3470F67C
9890000 6A989E0A707EE87E1282CE0A378844A6 9174CD96 C6AD7E3D
9900000 DC0B26437F05F79F37075FA3439F48AF 556D8A8F 7483B7A7
9910000 6C36AEA2F29EB466F0C04E2A09AE214E EDAE319E 80FE7F89
9920000 B93AB35652DC3BA264C21EB6055014AA 3056290A 9157B8FB
9930000 F8D7E317731726D7837D3A370E5FE98F A2B6405F 9989810A
9940000 3D7BF67FF04782C3444324576D0FE0D3 C4EE0683 A95EEC77
9950000 F8B827E6B24E894E8A4A99A6A7E2BC9E 95C54F4E D96B4538
```

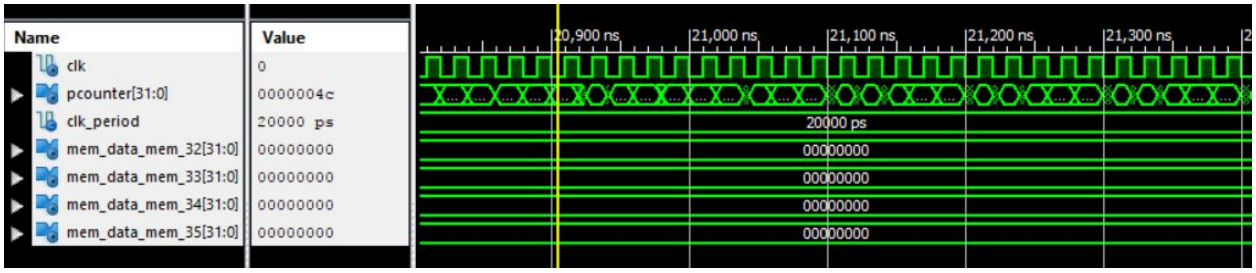


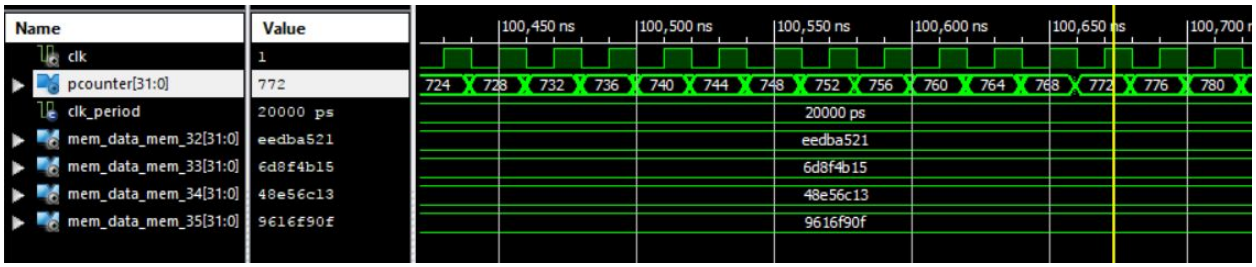




Timing simulation (5 test vectors):

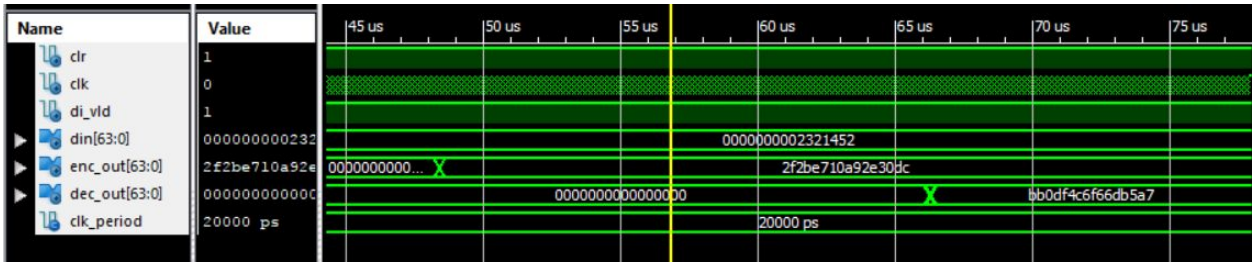
1. Ukey = X”00000000 00000000 00000000 00000000’’
   Din = X“00000000 00000000”



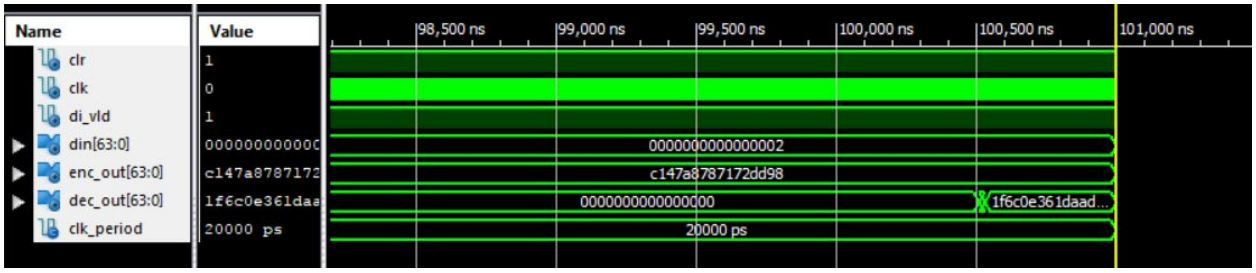



```
1.key=00000000000000000000000000000000
 plaintext 00 00 ----- encryption EEDBA521 6D8F4B15 ------decryption 48E56C13 9616F90F
```

2.



3.



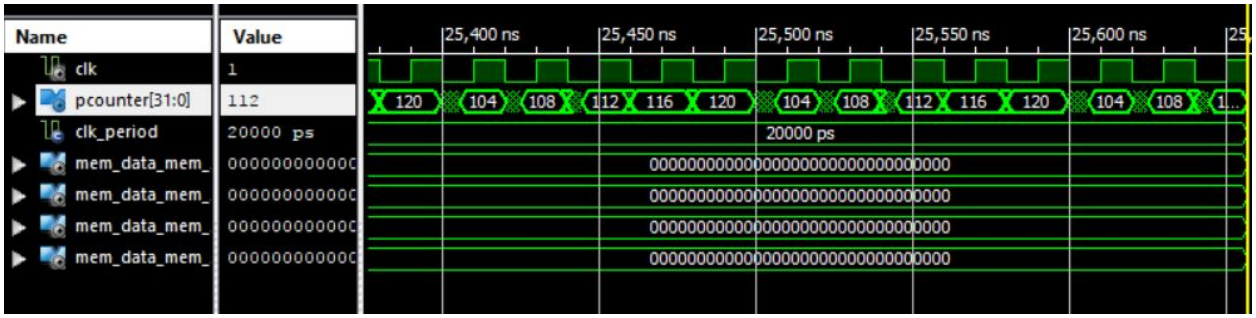

```
1.key=00000000000000000000000000000000
 plaintext 00 02 ----- encryption C147A878 7172DD98 ------decryption 1F6C0E36 1DAADBC9
```

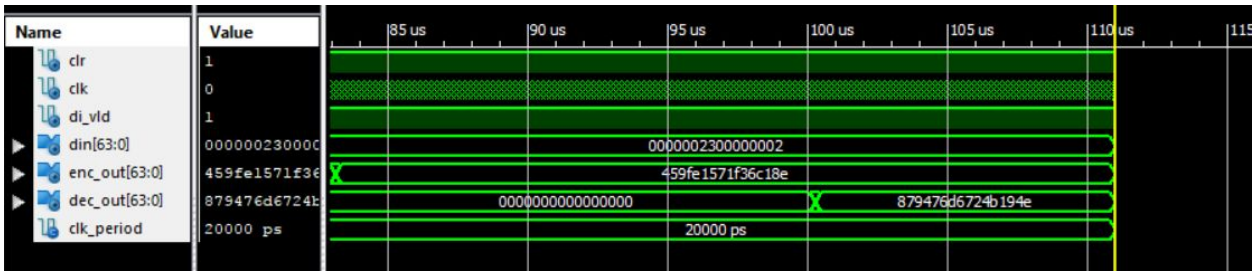
4.

```
1.key=00000000000000000000000000000000
 plaintext 01 01 ----- encryption 8731223 2C9311D6 ------decryption AFFA8B9C 14FBBCC4
```

5.

```
1.key=00000000000000000000000000000000
 plaintext 23 02 ----- encryption 459FE157 1F36C18E ------decryption 879476D6 724B194E
```


Name
Value
clk
1
pcounter[31:0]
112
clk_period
20000 ps
mem_data_mem_
mem_data_mem_
mem_data_mem_
mem_data_mem_
25,400 ns
25,450 ns
25,500 ns
25,550 ns
25,600 ns
120
104
108
112
116
120
104
108
112
116
120
104
108
20000 ps


Name
Value
clr
1
clk
0
di_vld
1
din[63:0]
enc_out[63:0]
459fe1571f3
dec_out[63:0]
879476d6724b
clk_period
20000 ps
85 us
90 us
95 us
100 us
105 us
110 us
0000002300000002
459fe1571f36c18e
0000000000000000
879476d6724b194e
20000 ps

Part III Performance and area analysis:

Number of clock cycles to perform Encryption/Decryption:
Encryption: 914
Decryption: 932

Number of clock cycles to perform Round Key Generation:
3897

Clock frequency of the processor:

Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	T
Autotimespec constraint for clock net clk_slow_BUFG	SETUP	N/A	12.347ns	N/A	
	HOLD	0.016ns		0	

Best case achievable is 12.347 ns. (propagation delay)
The fastest speed it can run is 1/ 12.347ns= 8.099*10^7Hz.

FPGA utilization (LUTs):

	Synthesis stage	Place and Route stage
LUT and FF pairs usage	6332(2%)	5129(8%)
IOB usage	44	44
RAM/DSP blocks used (if any)	0	0

PART IV RC5 implementation

1. Key Expansion:

Before the key expansion starts, the Instruction Memory and Data Memory have been initialized by 155 and 64 components. The Ins Mem is filled with the machine code of translated assembly program to implement key expansion algorithm. The ukey is fed through top-level port map, and the first 4 addresses in the Data Mem are used to store the 128 bit ukey.

2. Encryption:

After the key expansion process, the Data Mem has been loaded with 26 keys starting from address 0 to 25. The encryption process will be proceeded. Similarly, the machine code of translated assembly program to implement RC5 encryption algorithm has been loaded into Ins Mem at the very beginning. Then encrypted text will be stored in the address 32 and 33 of the Data Mem.

3. Decryption:

After the encryption process, the decryption will be proceeded. The instructions have been loaded at the very beginning and the deciphered text will be stored in the address 34 and 35 of the Data Mem.

Description of RC5 implementation in assembly:

How to realize shift faster: if we need to shift Rt for Rs bits

1 if (Rs == 0) jump to 12

2 if (Rs == 3) jump to 9

3 if (Rs == 2) jump to 10

4 if (Rs == 1) jump to 11

5 Shift Rt for 4 bits

6 Rs = Rs - 4

7 if (Rs != 0) jump to 1

8 if (Rs == 0) jump to 12

9 Shift Rt for 1 bits

10 Shift Rt for 1 bits

11 Shift Rt for 1 bits

12 continue the code

Key generation:

```
25  LW(1, 0, 0)
26  ADD(2, 0, 0)
27  ADD(3, 0, 0)
28  SHL(0, 2, 3)
29  ADD(3, 2, 0)
30  LW(5, 4, 0)
31  ADD(0, 4, 3)
32  ANDI(0, 0, 25)
33  SHL(3, 3, 1)
34  SUBI(0, 0, 1)
35  BNE(0, 0, -3)
36  BNE(1, 25, 1)
37  SUBI(1, 1, 25)
38  ADDI(1, 1, 1)
39  BNE(5, 3, 1)
40  SUBI(5, 5, 3)
41  ADDI(5, 5, 1)
42  BNE(6, 77, 1)
43  HAL()
44  BEQ(1, 1, -19)
```

Encryption:

```
LW(0,1,0)              #LW R1, 30(R0)          // R1=A(0)
LW(0,2,1)               #LW R2, 31(R0)         // R2=B(0)
LW(0,9,6)               #LW R9, 0(R0)          //R9=S[0]
ADDI(0,11,1)
ADDI(0,12,2)
ADDI(0,12,3)
ADD(1,9,7)              #ADD R1, R1, R9        //A=A+S[0]
LW(0,9,1)               #LW R9, 1(R0)          //R9=S[1]
ADD(2,9,2)              #ADD R2, R2, R9        //B=B+S[1]
ADDI(0,3,8)             #ADDI R3, R0, 2        // I=2
ADDI(0,8,32)            #ADDI R8, R0,25        // R8=25
NOR(1,1,4)              #NOR R4, R1, R1        // NOR(A)
NOR(2,2,5)              #NOR R5, R2, R2        // NOR(B)
AND(1,5,6)              #AND R6, R1, R5        // A NOR(B)
AND(2,4,7)              #AND R7, R2, R4        // NOR(A) B
OR(6,7,1)               #OR R1, R6, R7         // A XOR B
ANDI(2,10,31)           #AND R10, R2, 31       // LEAST 5-BITS OF B   -----andi

BEQ(10,0,9)                                    //if r10=0   jump
BEQ(10,11,7)
BEQ(10,12,5)
BEQ(10,13,3)
SHL(1,1,4)              #SHL R1, R1, 4         // SHL 4-BIT
SUBI(10,10,4)           #SUBI R10, R10, 1       // R10=R10-1
BNE(10,0,-6)            #BNE R10, R0, -3       // SHL R10-BITS
SHL(1,1,1)
SHL(1,1,1)
SHL(1,1,1)

LW(3,9,0)               #LW R9, 0(R3)          //R9=S[I]
ADD(1,9,1)              #ADD R1, R1, R9        //A=A+S[I]
ADDI(3,3,1)             #ADDI R3, R3, 1        //I=I+1
NOR(1,1,4)              #NOR R4, R1, R1        // NOR(A)
NOR(2,2,5)              #NOR R5, R2, R2        // NOR(B)
AND(1,5,6)              #AND R6, R1, R5        // A NOR(B)
AND(2,4,7)              #AND R7, R2, R4        // NOR(A) B
OR(6,7,2)               #OR R2, R6, R7         // A XOR B   111
ANDI(1,10,31)           #AND R10, R1, 31       // LEAST 5-BITS OF A     -----andi


BEQ(10,0,9)      #if r10=0   jump
BEQ(10,11,7)
BEQ(10,12,5)
BEQ(10,13,3)
SHL(2,2,4)              #SHL R2, R2, 1         // SHL 4-BIT
SUBI(10,10,4)           #SUBI R10, R10, 1      // R10=R10-1
BNE(10,0,-6)            #BNE R10, R0, -3       // SHL R10-BITS
SHL(1,1,1)
SHL(1,1,1)
SHL(1,1,1)


LW(3,9,0)               #LW R9, 0(R3)          //R9=S[I]
ADD(2,9,2)              #ADD R2, R2, R9        //B=B+S[I]
ADDI(3,3,1)             #ADDI R3, R3, 1        // I=I+1
BNE(3,8,-27)            #BNE R3, R8, -25       //I!=25 LOOP
SW(0,1,32)              #SW R1, 32(R0)         // A(OUT)=R1
SW(0,2,33)              #SW R2, 33(R0)         // B(OUT)=R2
```

Decryption:

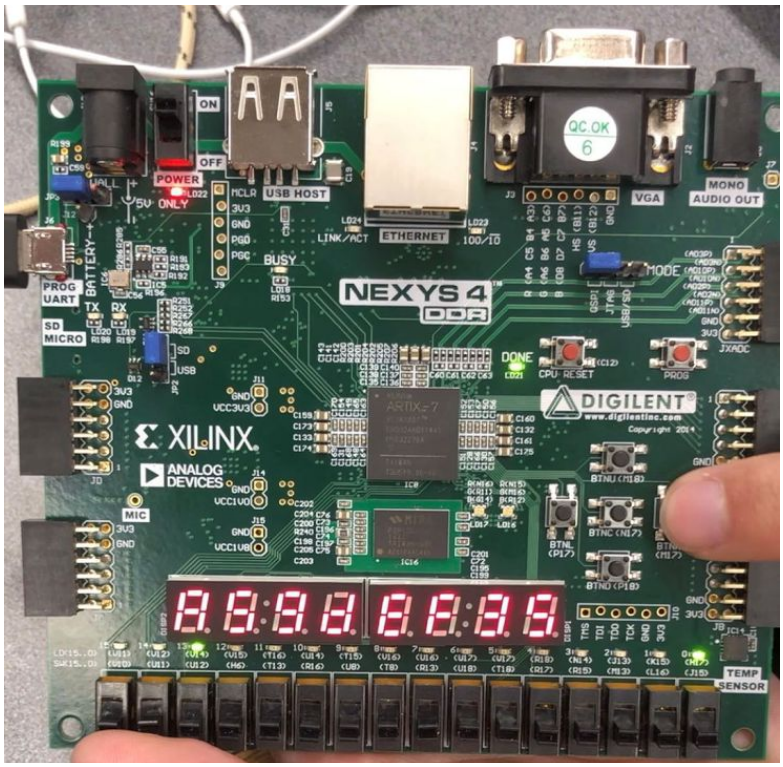
```
LW 0 1 30           R1 = M[30]
LW 0 2 31           R2 = M[31]
ADDI 0 3 25         R3 = i = 25
ADDI 0 10 1         R10 = 1 (break loop)
ADDI 0 11 4         R11 = 4
ADDI 0 12 3         R12 = 3
ADDI 0 13 2         R13 = 2
ADDI 0 14 1         R14 = 1
LW 3 8 0            R8 = M[R3] = M[i]
SUB 2 8 2           R2 = R2 - R8
ANDI 1 9 31         R9 = R1 last 5 bits
BEQ 9 0 9
-------------------------------------------------------------
BEQ 12 9 5          R2 shift R9 bits
BEQ 13 9 5
BEQ 14 9 5
SHR 2 2 4
SUBI 9 9 4
BEQ 0 9 -7
SHR 2 2 1
SHR 2 2 1
SHR 2 2 1

-------------------------------------------------------------
NOR 1 1 4           R2 do XOR with R1
NOR 2 2 5
AND 1 5 6
AND 2 4 7
OR 6 7 2
SUBI 3 3 1
LW 3 8 0
SUB 1 8 1
ANDI 2 9 31
BEQ 9 0 9
```

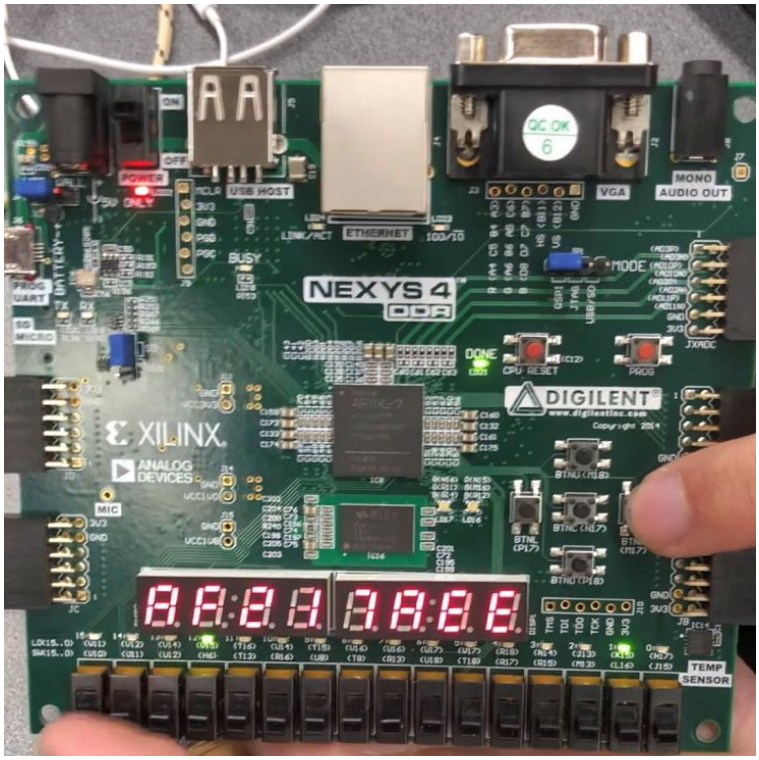
```
-------------------------------------------------------------
BEQ 12 9 6          R1 shift R9 bits
BEQ 13 9 6
BEQ 14 9 6
SHR 1 1 1
SUBI 9 9 4
BEQ 0 9 -6
BEQ 0 9 3
SHR 1 1 1
SHR 1 1 1
SHR 1 1 1
-------------------------------------------------------------
NOR 1 1 4           R1 Xor R2
NOR 2 2 5
AND 1 5 6
AND 2 4 7
OR 6 7 2
SUBI 3 3 1
BNE 3 10 -41        loop

LW 3 8 0
SUB 2 8 2
SUBI 3 3 1
LW 3 8 0
SUB 1 8 1
SW 0 1 34           M[34] = R1
SW 0 2 35           M[35] = R2
```

Part V Processor interfaces (how do you provide inputs and display results)

```
U0: final_inout      PORT MAP(clr=>clr, clk_slow=>clk_slow, in_switch=>in_switch, led_an=>led_an, led_segment=>led_segment,
                              key_vld_botton=>key_vld_botton, data_vld_botton=>data_vld_botton, work_botton=>work_botton,
                              work=>work,
                             dout_botton=>dout_botton, enc=>enc, enc_botton=>enc_botton, enc_indicate=>enc_indicate,
                             led_indicate=>led_indicate, di_vld=>di_vld, ukey=>ukey, din=>din, dout=>dout);
U1: Processor        PORT MAP(clr=>clr, clk_slow=>clk_slow, di_vld=>di_vld,
                              ukey=>ukey, din=>din,dout_enc=>dout_enc,dout_dec=>dout_dec);
```

We use the 16 switches to input for din and ukeys, when we press P17 and switch we will put the ukeys. The N17 and switches will be responsible for din, the M17 will start RC5 and the P18 will change the result for encryption and decryption.





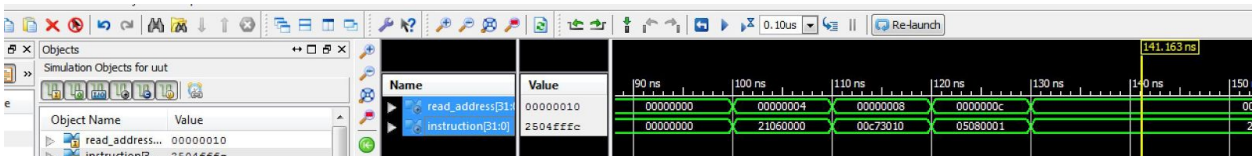
You can check the youtube video for details and the whole process.

Part VI Details about how you verified your overall design

The following below is the test bench designed for testing the instruction memory. In the test bench 4 values was inserted into the input for the instruction memory.

```
BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: instruction_memory PORT MAP (
          read_address => read_address,
          instruction => instruction
        );



   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;

      --wait for <clock>_period*10;

      -- insert stimulus here
      read_address <= x"00000004";
      wait for 10 ns;
      read_address <= x"00000008";
      wait for 10 ns;
      read_address <= x"0000000C";
      wait for 10 ns;
      read_address <= x"00000010";

      wait;
   end process;

END;
```

The following behavioral analysis showed that the inputted addressed did lead to the corresponding output that we expected.



Part VII Conclusion

Github repo:
https://github.com/yemazon/EL6463Project_Processor_RC5

Demo Video:
https://youtu.be/skBHDx-i6Pg