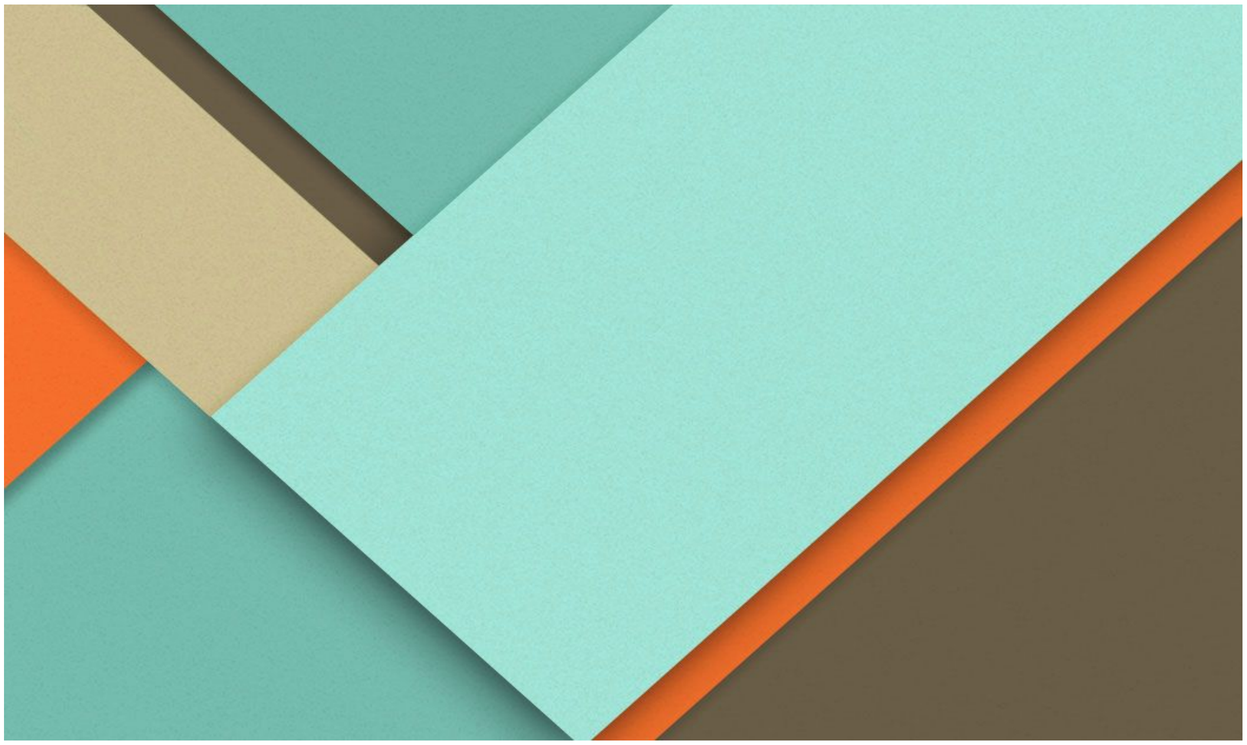


NYU-6463 Processor Design

12.13.2016

Team members:

Wei Lin, Enbo liu, Yang Shi, Yao-An Tsai, Xun(Godwin) Wang, Yifei Wang

Project Github Page: https://github.com/godwinxunwang/LongLiveFPGA

Overview

We are "LongLiveFPGA" group. We developed an FPGA (Field Programmable Gate Array) implementation of a single-cycle processor using VHDL. We used Xilinx ISE Design Suite to synthesize and implement our design. Then, we performed functional and timing simulations using ISim/ModelSim. We used Digilent Adept Software to load the bit file into the FPGA. The FPGA that we used in this design was Nexys4 DDR Artix-7. After implementing and testing this design, we adapted the RC5 encryption, decryption, and round-key generation algorithms into MIPS Assembly language. We ran the RC5 MIPS program on our FPGA, and examined the output.

Project Goals

- Implement a 32-bit processor in VHDL.
- The processor supports MIPS J, R, I type instructions.
- The single-cycle implementation of the MIPS processor supports PC update, ALU Operation, Memory Access, and Register Access.

Workflow

1. Implement the NYU-6463 Processor with the specification described.
2. Do a performance (max speed of your processor) and area (number of gates you used from each type) analysis.
3. Write an assembly program to implement the RC5 block cipher (encryption and decryption) as well as round key generation using the instructions in Table 1.
4. Convert the assembly codes into machine code and run them on your designed processor and show that it works properly.
5. Implement your design on FPGA.
6. Describe how many cycles are required to complete RC5 encryption and decryption on NYU-6463 Processor.

7. Support updating the instruction memory (changing the program being executed) while your processor is running on the FPGA (optional).

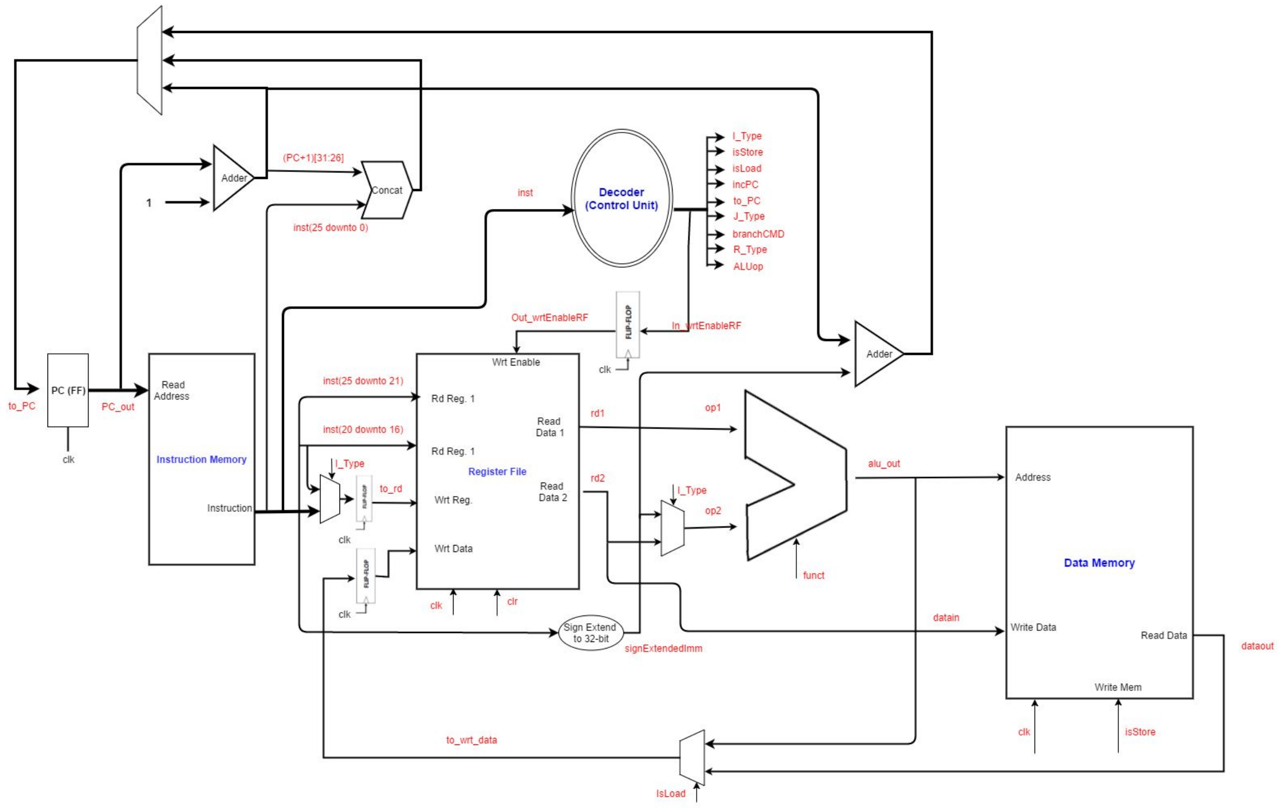
DESIGN & IMPLEMENTATION

I. Design block diagram (single-cycle datapath)

The block diagram for our design is shown in the below figure. Because our professor needs to support some simple MIPS-like instructions, we adapted the basic architecture of MIPS[1] professor into our design.

In each clock cycle, the processor task consists of instruction fetch, instruction decode, execution, memory access and write-back. At first, the PC passes an address to the instruction memory to indicate which instruction to be executed. Then, the instruction memory finds the corresponding line of instruction and passes it to the register file and also the decoder. The register file supports two register reads and one register write. For R-type instructions, it will do the three register operations, and the result from the ALU will be written back to the register file. For I-type instructions, it will passes one register value to the ALU, and also get results from the ALU. The data value from the RF are operated in the ALU, whose operation is controlled by the decoder. The decoder determines a lot of things, including branching steps and the controls of every multiplexers in this design. Based on the fetched instruction from the instruction memory, the decoder can output certain values to multiplexers and ALU. Finally, the ALU calculates and passes the result back to the register file or to the data memory. In order to avoid the hazard when reading and writing the register file, we added flip-flops before write-in signals to force this processor to execute write after read for each instruction.

[1] Kane, Gerry, and Joe Heinrich. *MIPS RISC architectures*. Prentice-Hall, Inc., 1992.



II. High level description of how you implement and run 3 components of RC5 (encryption, decryption and key expansion) on your designed processor

For the RC5 encryption, decryption, and round-key expansion functions, we followed Ronald L. Rivest's paper *RC5 Encryption Algorithm* [2] to implement the 3 components using the assembly supported by our processor. You may find the implemented assembly code files (with extension .asm) in this project's GitHub page, under the directory "Assembler". The assembly code file *rc5KeyGeneration.asm* is for the RC5 encryption round-key expansion function, the assembly code file *rc5Encryption.asm is for the* RC5 encryption function, and *rc5Decryption.asm is for the* decryption function.

We have designated the some specific locations in the data memory to store the input and output values for these three RC5 functions. Below table shows which location in the data memory are reserved for specific purposes.

[2] Rivest, Ronald L. "The RC5 encryption algorithm." *International Workshop on Fast Software Encryption*. Springer Berlin Heidelberg, 1994.

Index in Data Memory	What is this in RC5	Purpose
DMem[100] to DMem[125]	Skey[0] to Skey[25]	Used for Encryption and Decryption
DMem[200] to DMem[203]	L_array[0] to L_array[3]	Used for key generation
DMem[300] to DMem[301]	A and B	Used for Encryption and Decryption
DMem[400] to DMem[403]	User-key for Key generation	User-key for Key generation
DMem[512]	select RC5 function	Determines whether to do Encryption(1) or Decryption(0)
DMem[513]	Program stating point	Determines whether to start the entire program
DMem[514]	select RC5 function	Determines whether to skip the round-key expansion part
DMem[600]	Result of High Encrypted value	Store the result value
DMem[601]	Result of Low Encrypted value	Store the result value
DMem[602]	Result of High Decrypted value	Store the result value
DMem[603]	Result of Low Decrypted value	Store the result value

For RC5 encryption, our assembly program assumes the input values are stored in the data memory, and it will firstly fetch the input values from the data memory. The input values for RC5 encryption function should be two 32-bit binary numbers. We also assume the assume that RC5 round-key expansion function has been performed on this processor, so that the array of Skey[0], Skey[1], Skey[2] , ..., Skey[26] has already been computed and stored in the data memory. After fetching the values to the register file, it will start the computation following the RC5 encryption algorithm. In our assembly program, we use $8 and $9 to store the values for register A and register B. We also use

$1 as the iterator for the “for loop” and $2 as the double of the iterator. The data-dependent rotation is performed by using *shl* and *shr* commands. Because *shl* and *shr* commands are I-type instructions, we cannot directly use them to perform the data-dependent rotation due to the limitations in inputting immediate value. So, we use the *shl* to rotate to the left by 1 for A or B times to achieve the desired command. When the “for loop” is running, it should constantly check the value of the iterator *i.* When i does not reach the maximum value, the branch command “*bne $0, $31, forEn*” at the end of this program will let the program counter jump back to the line where the label “forEn” is. When i reaches the maximum value, it will skip that branch instruction, and proceed to “halt” to end this program.

For RC5 decryption, similar to the encryption function, our decryption assembly program also assumes the user-specified values are stored at some specific locations in the data memory. It will fetch the user-specified values (two 32-bit binary number) from the data memory, and then perform the decryption. It also uses $1 as the iterator for the “for loop” and $2 as the double of the iterator. The iterator *i* should counter from 12 downto 1. For the data-dependent rotation command, we use the *shr* to rotate to the right by 1 for A or B times to achieve the desired command. Also, the branching command “bne $0, $1, forDe” at the end of the program will constantly check the value of *i*. Once *i* reaches 1, it will proceed to the rest three commands.

For RC5 round-key generation, our assembly program stores the user-specified in $6, $7, $8, $9 from MSB to LSB, which are actually the same values of the L_array from 0 to 3. We followed the round-key generation pseudo code and implemented this function. Those jump and branching commands at the end of the program ensures the iterations happened in the *for loop*. Finally, this assembly program will stored the generated keys back to the data memory.

However, in real world, it would be impossible to load the three programs separately into the instruction memory, so we integrated these three different functions into one assembly program (file *rc5.asm* under Assembler folder). The integrated RC5 program assumes a few points. DMem[512] determines whether to do Encryption(1) or Decryption(0). DMem[513] determines whether to start the entire program, and DMem[514] determine whether to skip the round-key expansion part. At the starting point, this program constantly loads the value from DMem[513] to $15 and checks the loaded value. Once the value turns to 1, it will proceed to the consequent instructions. Then, it will load the value from DMem[513] to $ 15 and check the value. If the value is 1, it will skip the round-key expansion part, and proceed to the location where label "skipKeyExpansion" is at. Before proceeding to the encryption algorithm, it will check the load and check the value in DMem[512], if the value is 1, it will do encryption; if the value is 0, it will skip the encryption function and just do decryption.

III. Description of processor interfaces (how you control the inputs and observe the outputs)

Any program that needed to be executed by this processor has to be firstly hardwired into the Instruction memory. For example, to perform the RC5 encryption, decryption or the key expansion with this processor, the user has to firstly write the hexadecimal instructions that achieves the functions into the Instruction memory. When the board was turned on, within each clock cycle, the machine will fetch one new instruction from the instruction memory, decode it, execute it and then refresh the data memory.

For the encryption, decryption and key expansion functions, in this project, the process is designed as follows:

Firstly, the user clicks on the central button to start the input process. And then, immediately the user will see a number '1' is displayed on the 7 segment LED display. At this stage, the user is supposed to input the 128-bits user-key for round key generation. For the input, the user needs to switch the 16 switches to input the lowest 16 bits of the user-key, and then, click on the central button again to input the next 16 bits segment of the user-key. When the central button was clicked for the second time, the number displayed on the LED will increase to '2'. Each time the central button was clicked the number will increase by one. So, click the central button 8 times, when the number displayed on LED has increased to 8, the user-key input is done. You will see a 'choose 1' statement displayed on the LED. This means that right now the you are supposed to indicate whether you want to do an encryption or decryption using this machine.

The user has the choice to do encryption or to do decryption by clicking on left button or the right button. When the user has chosen, number '1' will again displayed on the LED which means that you can input the data that needs to be encrypted/decrypted for now. The number '1 ' indicates that you can input the lowest 16-bits value of the 64-bits user input. And very similar to the user-key input process, you click on the central button to input the next 16-bits segment of the user-input, and you will know that you finished when you have input the 4th segment of the input. When you click on the central button at after the 4th click, you will see a statement ‘choose 2’ displayed on the LED. Here you are ready to view the output results, and at the same time you can also go back to ‘choose 1’ stage to input another 64 bits user data for encryption/decryption.

At ‘choose 2’ stage, the processor is then ready to output the execution results. In our design, the processor supports two ways of presenting the outputs: One is displaying the outputs cycle by cycles and the other is displaying the outputs only at when the program finished execution. The user can click right button to view the outputs instruction by instruction, and to view the outputs, switch the leftmost switch(switch 15) to high. Then the current instruction being executed will be displayed on the LED and the centrol button is the controler of the output flow, the user can click the central button to let the processor fetch the next instruction and view the instruction on LED.

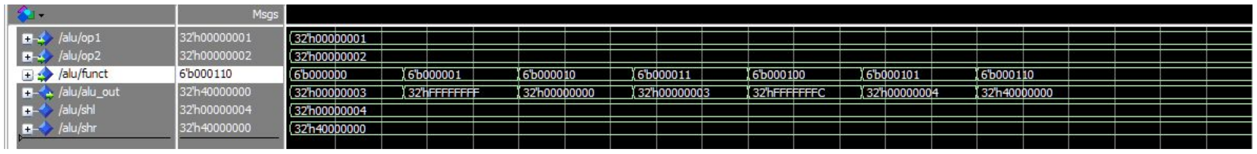
The user can also click the left button at choose 2 stage to let the processor finish the whole execution and then see the final results. To go back to ‘choose 1’ stage, the user should click the lower button at choose 2 stage, in this way, the user can input another sequence that needs to be encoded or decoded and store it into the datamemory.

For viewing the output(outputs are the data stored in data memory and register file), first switch the leftmost switch to low, and then switch the rightmost switch to choose between whether to see the data memory or register file’s content, ‘1’ is for data memory, ‘0’ is for register file. Since the data is stored by 32 bits a piece on the datamemory and register file. The size of data memory is 2^10 pieces of 32 bits data, and the size of register file is 32 pieces of 32 bits data. To view the stored data, use the switches at positions 1 to 10 to specify a position in the data memory or register file. When you switch, the 32 bits data will be displayed as 8 hexadecimal value on the LED display. And obviously, since this design supports cycle by cycle execution, when the leftmost switch is low, the user is also able to view the data in the register file or data memory after each instruction executed.

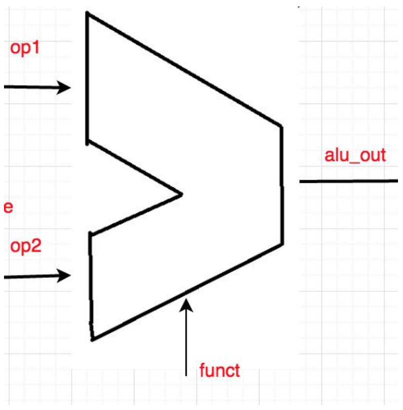
SIMULATION

Simulation screenshots for each of the checkpoints in next page.

1. Complete designing ALU.



```
                 Msgs
/alu/op1        32'h00000001   32'h00000001
/alu/op2        32'h00000002   32'h00000002
/alu/funct      6'b000110      6'b000000    6'b000001    6'b000010    6'b000011    6'b000100    6'b000101    6'b000110
/alu/alu_out    32'h40000000   32'h00000003 32'hFFFFFFFF 32'h00000000 32'h00000003 32'hFFFFFFFC 32'h00000004 32'h40000000
/alu/shl        32'h00000004   32'h00000004
/alu/shr        32'h40000000   32'h40000000
```

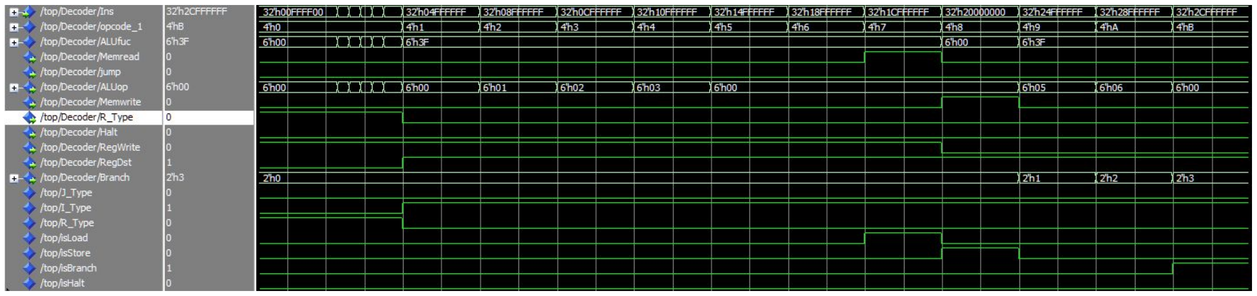


Input: op1, op2, funct

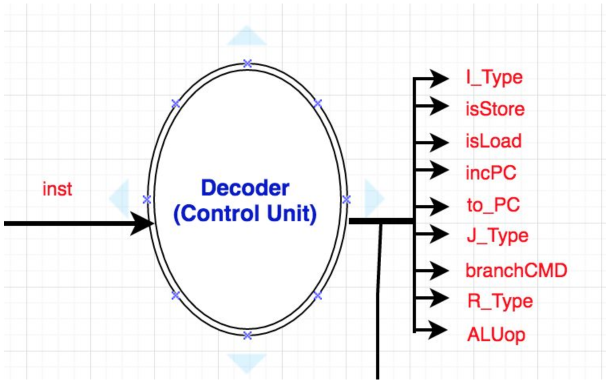
Output: alu_out

The screenshot shows the result (alu_out) of ALU calculation, also which of the ALU's function is applied is decided by 6-bits funct.

2. Complete designing Decoder Unit.



```
/top/Decoder/Ins        32'h2CFFFFFF   32'h00FFFF00 ... 32'h04FFFFFF 32'h08FFFFFF 32'h0CFFFFFF 32'h10FFFFFF 32'h14FFFFFF 32'h18FFFFFF 32'h1CFFFFFF 32'h20000000 32'h24FFFFFF 32'h28FFFFFF 32'h2CFFFFFF
/top/Decoder/opcode_1   4'hB           4'h0         4'h1         4'h2         4'h3         4'h4         4'h5         4'h6         4'h7         4'h8         4'h9         4'hA         4'hB
/top/Decoder/ALUfuc     6'h3F          6'h00 ...    6'h3F                                                                                    6'h00        6'h3F
/top/Decoder/Memread    0
/top/Decoder/jump       0
/top/Decoder/ALUop      6'h00          6'h00 ...    6'h00        6'h01        6'h02        6'h03        6'h00                                                          6'h05        6'h06        6'h00
/top/Decoder/Memwrite   0
/top/Decoder/R_Type     0
/top/Decoder/Halt       0
/top/Decoder/RegWrite   0
/top/Decoder/RegDst     1
/top/Decoder/Branch     2'h3           2'h0                                                                                                                  2'h1         2'h2         2'h3
/top/J_Type             0
/top/I_Type             1
/top/R_Type             0
/top/isLoad             0
/top/isStore            0
/top/isBranch           1
/top/isHalt             0
```

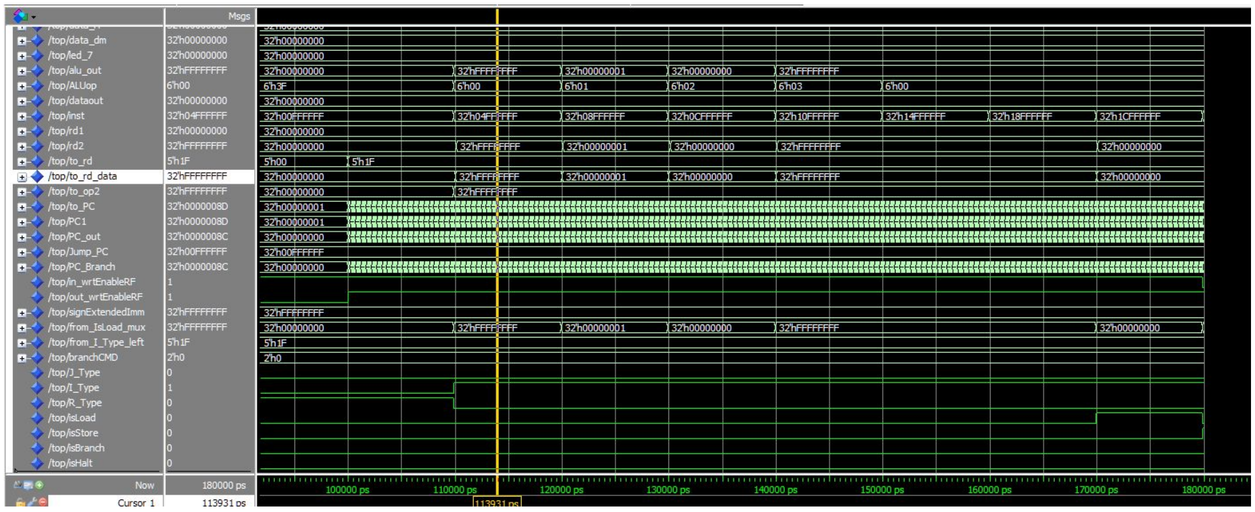


Input: inst

Output: I_type, isStore, isLoad, incPC, to_PC, J_Type, branchCMD, R_Type, ALUop

The screenshot shows the result of Decoder. The output is translated from inst.

3. Complete the processor design.



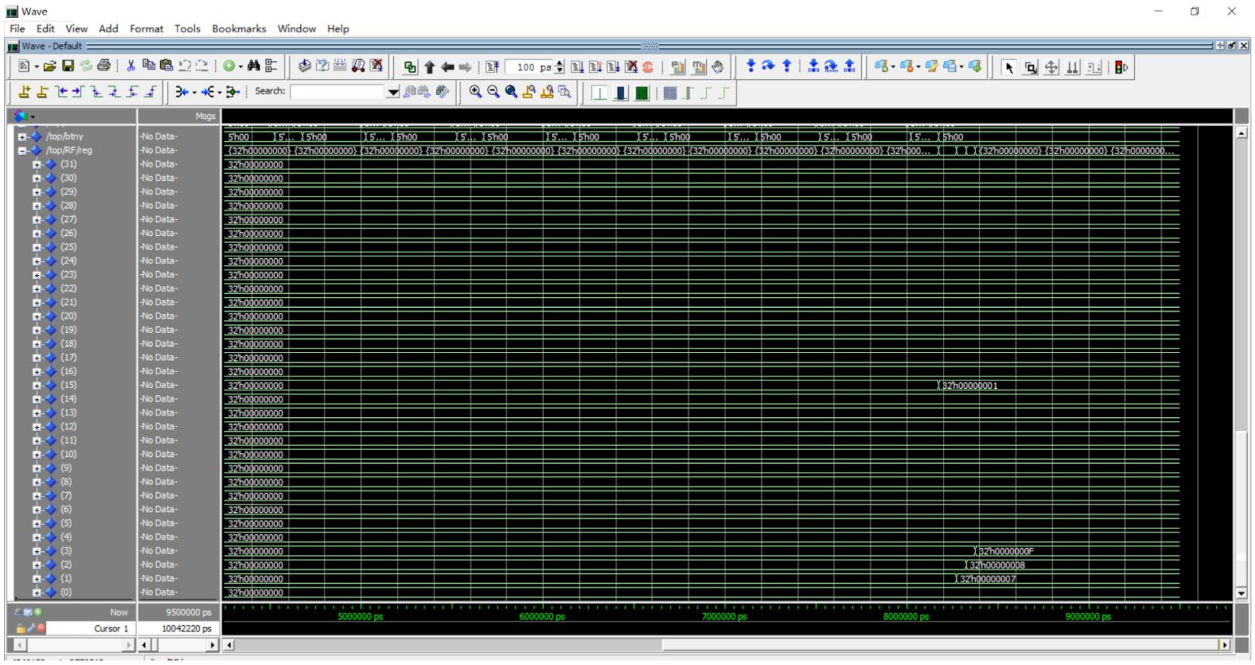
	Msgs
/top/data_dm	32'h00000000
/top/led_7	32'h00000000
/top/alu_out	32'hFFFFFFFF
/top/ALUop	6'h00
/top/dataout	32'h00000000
/top/inst	32'h04FFFFFF
/top/rd1	32'h00000000
/top/rd2	32'hFFFFFFFF
/top/to_rd	5'h1F
/top/to_rd_data	32'hFFFFFFFF
/top/to_op2	32'hFFFFFFFF
/top/to_PC	32'h0000008D
/top/PC1	32'h0000008D
/top/PC_out	32'h0000008C
/top/Jump_PC	32'h00FFFFFF
/top/PC_Branch	32'h0000008C
/top/in_wrtEnableRF	1
/top/out_wrtEnableRF	1
/top/signExtendedImm	32'hFFFFFFFF
/top/from_IsLoad_mux	32'hFFFFFFFF
/top/from_I_Type_left	5'h1F
/top/branchCMD	2'h0
/top/J_Type	0
/top/I_Type	1
/top/R_Type	0
/top/isLoad	0
/top/isStore	0
/top/isBranch	0
/top/isHalt	0
Now	180000 ps
Cursor 1	113931 ps

The screenshot shows all the operations of the NYU-6463 Processor. In the button of the image indicates R,I,J-Types results. Also, in the center of the image shows the Load/write of Data-Memory.

4. Run the sample programs given below and verify your CPU.

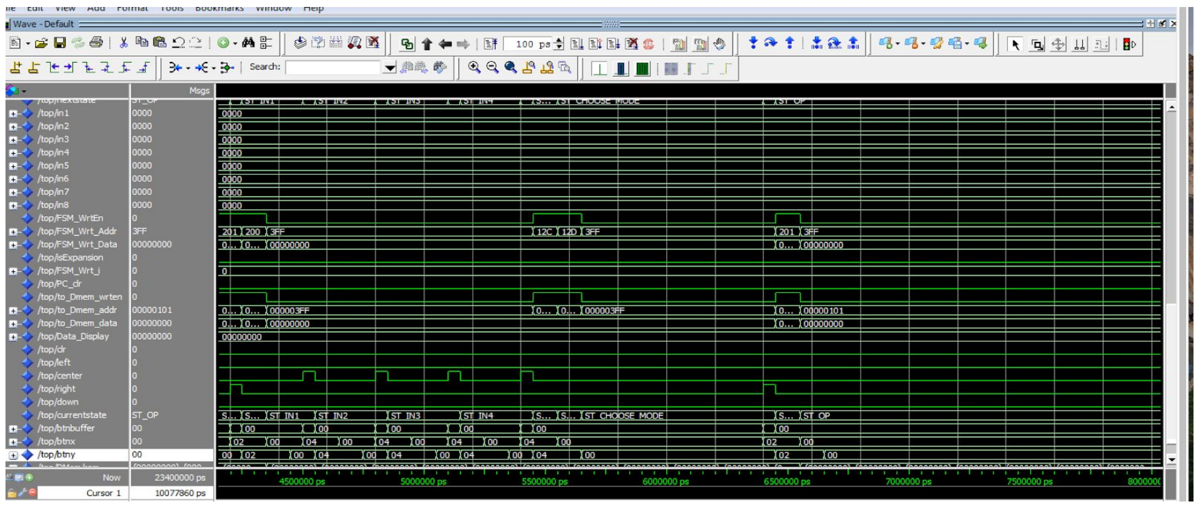
When running the sample program, the state machine should not be used, the machine simply fetch the instructions from the Instruction memory, decode it, execute and update the memory components' content. So in order to show the program running correctly, we skip the state machine for input and view the final result in Register-File and Data-memory.

1. Sample program I: The result data in the RF is shown as below:

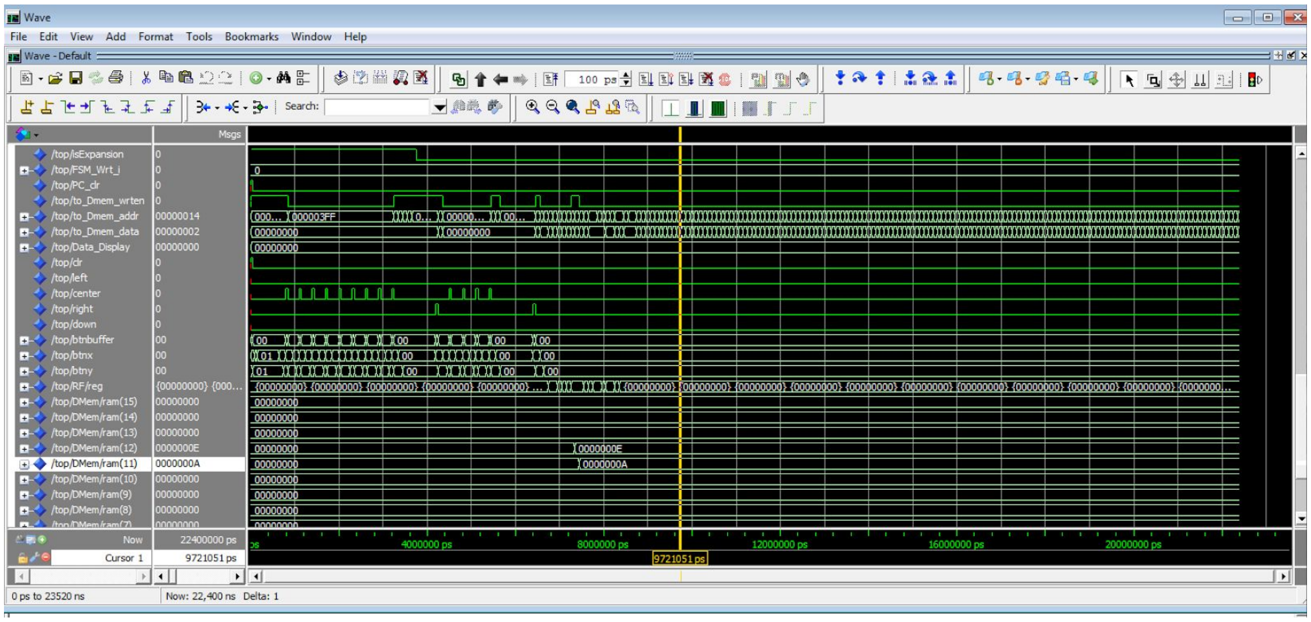


As can be seen in register file the location R1 = 7; R2 = 8 and R3 = 15.

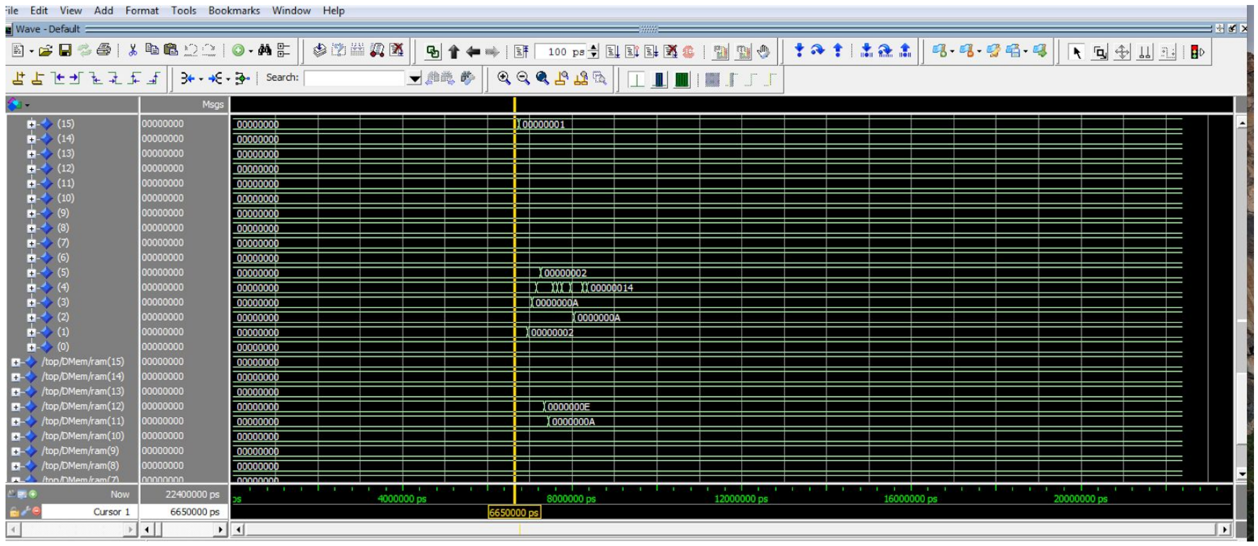
2. Sample Program II:



```
/top/nextstate      ST_OP
/top/in1            0000
/top/in2            0000
/top/in3            0000
/top/in4            0000
/top/in5            0000
/top/in6            0000
/top/in7            0000
/top/in8            0000
/top/FSM_WrtEn      0
/top/FSM_Wrt_Addr   3FF
/top/FSM_Wrt_Data   00000000
/top/isExpansion    0
/top/FSM_Wrt_j      0
/top/PC_clr         0
/top/to_Dmem_wrten  0
/top/to_Dmem_addr   00000101
/top/to_Dmem_data   00000000
/top/Data_Display   00000000
/top/clr            0
/top/left           0
/top/center         0
/top/right          0
/top/down           0
/top/currentstate   ST_OP
/top/btnbuffer      00
/top/btnx           00
/top/btny           00
Now      23400000 ps
Cursor 1 10077860 ps
```



```
/top/isExpansion     0
/top/FSM_Wrt_j       0
/top/PC_clr          0
/top/to_Dmem_wrten   0
/top/to_Dmem_addr    00000014
/top/to_Dmem_data    00000002
/top/Data_Display    00000000
/top/clr             0
/top/left            0
/top/center          0
/top/right           0
/top/down            0
/top/btnbuffer       00
/top/btnx            00
/top/btny            00
/top/RF/reg          {00000000} {000...
/top/DMem/ram(15)    00000000
/top/DMem/ram(14)    00000000
/top/DMem/ram(13)    00000000
/top/DMem/ram(12)    0000000E
/top/DMem/ram(11)    0000000A
/top/DMem/ram(10)    00000000
/top/DMem/ram(9)     00000000
/top/DMem/ram(8)     00000000
/top/DMem/ram(7)     00000000
Now      22400000 ps
Cursor 1 9721051 ps
0 ps to 23520 ns    Now: 22,400 ns  Delta: 1
```



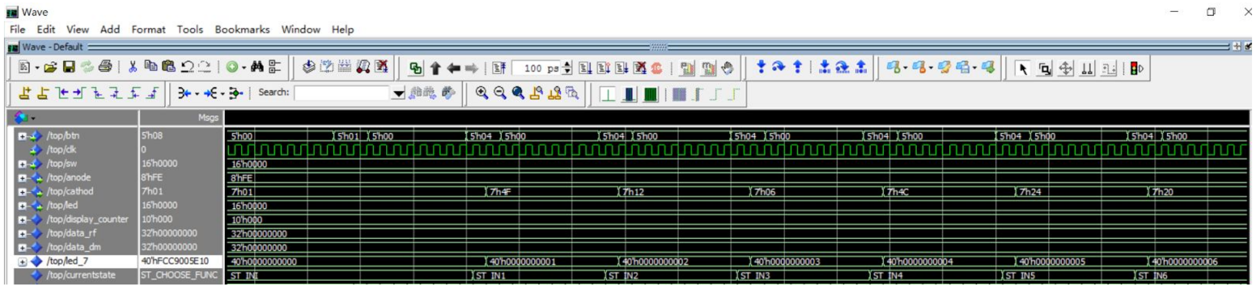
```
(15)   00000000   00000001
(14)   00000000
(13)   00000000
(12)   00000000
(11)   00000000
(10)   00000000
(9)    00000000
(8)    00000000
(7)    00000000
(6)    00000000
(5)    00000000   00000002
(4)    00000000   00000014
(3)    00000000   0000000A
(2)    00000000   0000000A
(1)    00000000   00000002
(0)    00000000
/top/DMem/ram(15)  00000000
/top/DMem/ram(14)  00000000
/top/DMem/ram(13)  00000000
/top/DMem/ram(12)  00000000   0000000E
/top/DMem/ram(11)  00000000   0000000A
/top/DMem/ram(10)  00000000
/top/DMem/ram(9)   00000000
/top/DMem/ram(8)   00000000
/top/DMem/ram(7)   00000000
Now      22400000 ps
Cursor 1 6650000 ps
```

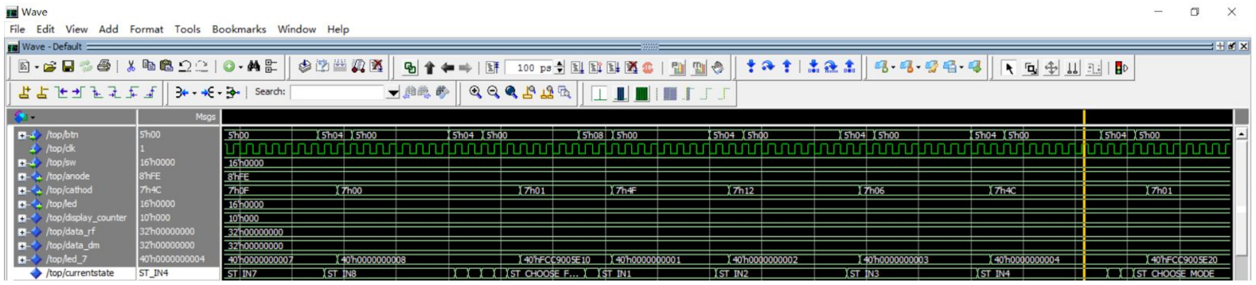
5. Complete RC5 Assembly code for Encryption and Decryption.

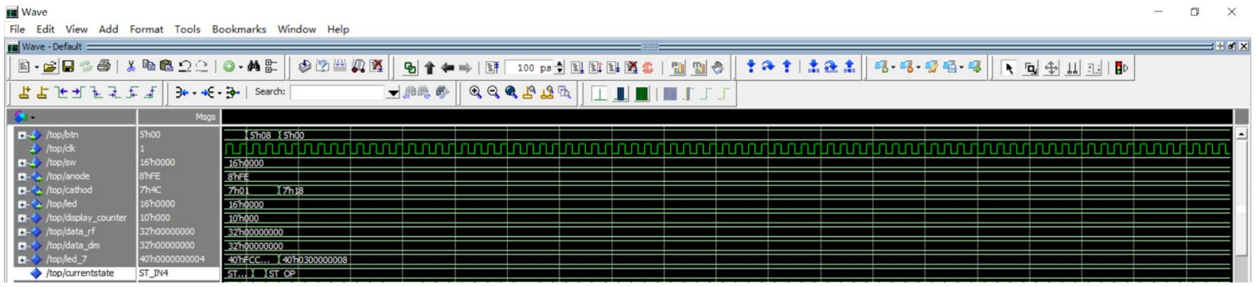
Please refer to the assembly code files (.asm) saved under Assembler folder. The assembly code file *rc5Encryption.asm is for the* RC5 encryption function, and *rc5decryption.asm is for the* decryption function.

6. Complete the processor design with additional interfaces and single stepping (7-segment LEDs, Switches for input control).

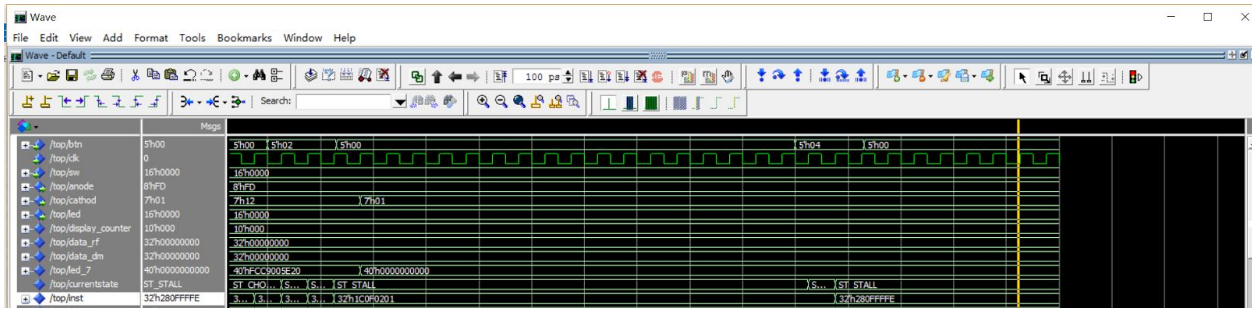
The following 3 figures show the simulation result of input interfaces(buttons and switches) and output interfaces(7-segment LEDs).







We apply “stall”(Keep PC point to the same address of Instruction memory and set WrtEn signals of Register File and Data Memory to ‘0’.) to implement single stepping. The figure shows the simulation result of single stepping.



```
/top/btn               5'h00
/top/clk               0
/top/sw                16'h0000
/top/anode             8'hFD
/top/cathod            7'h01
/top/led               16'h0000
/top/display_counter   10'h000
/top/data_rf           32'h00000000
/top/data_dm           32'h00000000
/top/led_7             40'h0000000000
/top/currentstate      ST_STALL
/top/inst              32'h280FFFFE
```

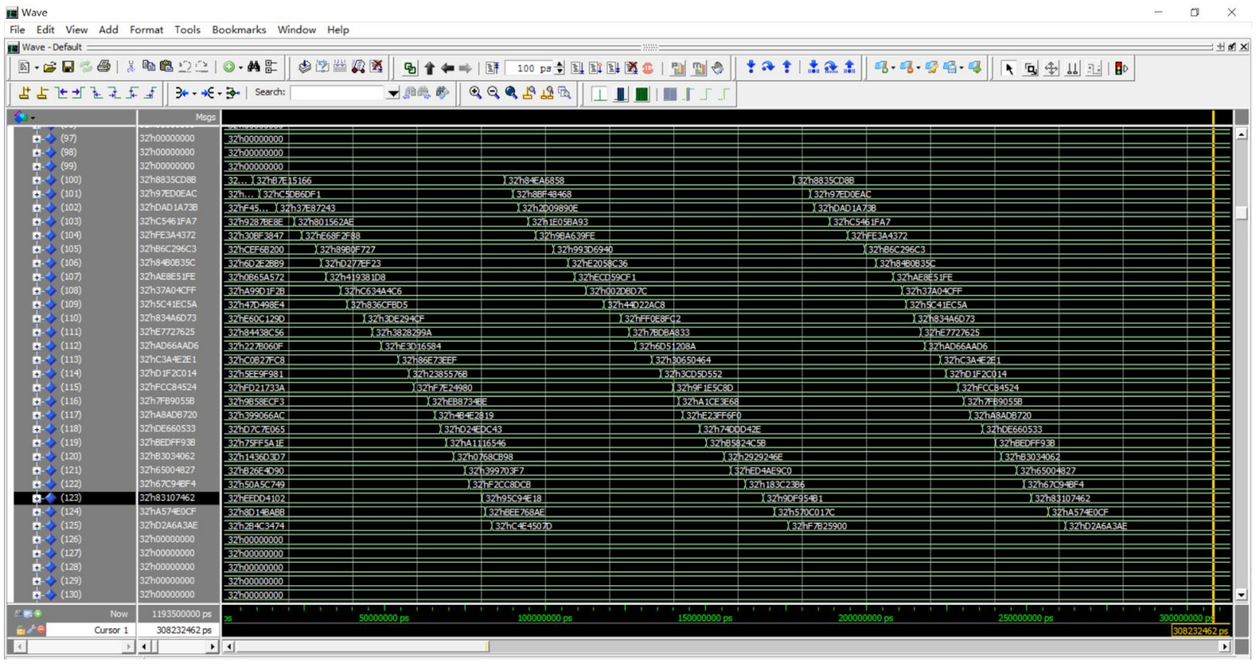
Press center button to execute one cycle each time.

7. Complete RC5 Assembly code (Key Expansion, Encryption and Decryption).

The assembly code file *rc5KeyGeneration.asm* is for the RC5 encryption round-key expansion function. The final integrated assembly program file is named as *rc5.asm*. Please refer to the assembly code files (.asm) saved under Assembler folder. For the description of the final integrated assembly program, please see the last part of Section II in *Design & Implementation* Chapter.

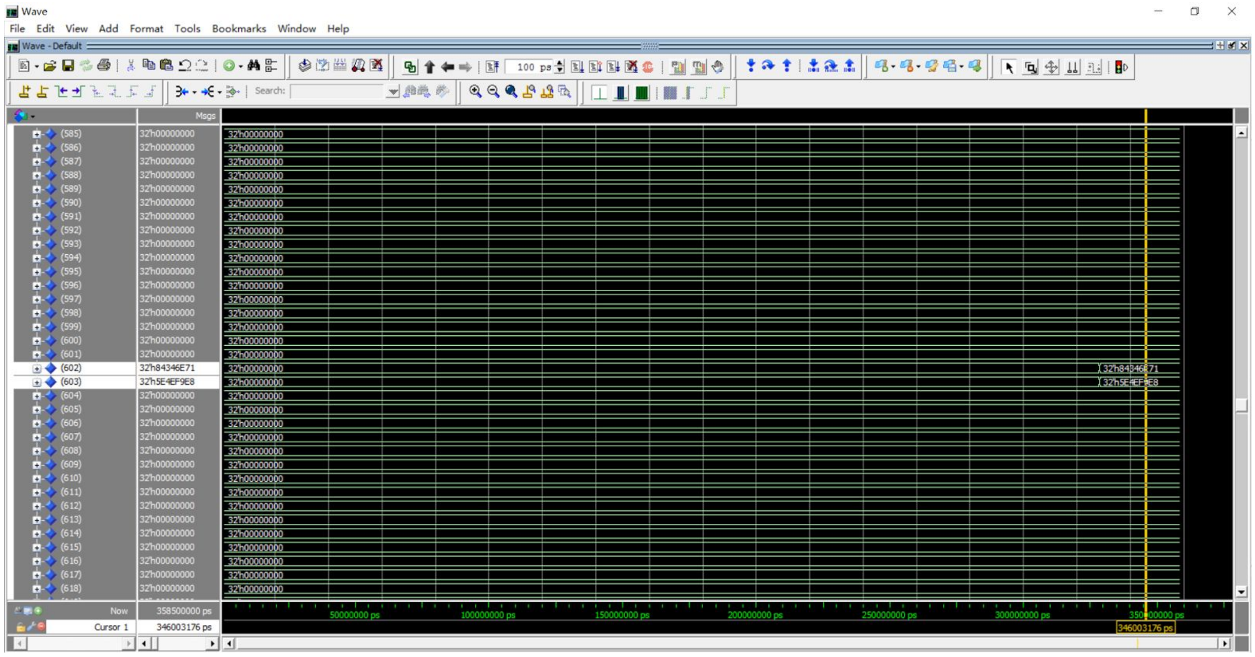
Simulation:

Key Expansion:



```
(97)    32'h00000000
(98)    32'h00000000
(99)    32'h00000000
(100)   32'h8835CD8B
(101)   32'h97ED0EAC
(102)   32'hDAD1A73B
(103)   32'hC5461FA7
(104)   32'hFE3A4372
(105)   32'hB6C296C3
(106)   32'h84B0B35C
(107)   32'hAE8E51FE
(108)   32'h37A04CFF
(109)   32'h5C41EC5A
(110)   32'h834A6D73
(111)   32'hE7727625
(112)   32'hAD66AAD6
(113)   32'hC3A4E2E1
(114)   32'hD1F2C014
(115)   32'hFCC84524
(116)   32'h7FB9055B
(117)   32'hA8ADB720
(118)   32'hDE660533
(119)   32'hBEDFF93B
(120)   32'hB3034062
(121)   32'h65004827
(122)   32'h67C94BF4
(123)   32'h83107462
(124)   32'hA574E0CF
(125)   32'hD2A6A3AE
(126)   32'h00000000
(127)   32'h00000000
(128)   32'h00000000
(129)   32'h00000000
(130)   32'h00000000
Now       1193500000 ps
Cursor 1  308232462 ps
```

Encryption and Decryption:



```
(585)   32'h00000000
(586)   32'h00000000
(587)   32'h00000000
(588)   32'h00000000
(589)   32'h00000000
(590)   32'h00000000
(591)   32'h00000000
(592)   32'h00000000
(593)   32'h00000000
(594)   32'h00000000
(595)   32'h00000000
(596)   32'h00000000
(597)   32'h00000000
(598)   32'h00000000
(599)   32'h00000000
(600)   32'h00000000
(601)   32'h00000000
(602)   32'h84346E71
(603)   32'h5E4EF9E8
(604)   32'h00000000
(605)   32'h00000000
(606)   32'h00000000
(607)   32'h00000000
(608)   32'h00000000
(609)   32'h00000000
(610)   32'h00000000
(611)   32'h00000000
(612)   32'h00000000
(613)   32'h00000000
(614)   32'h00000000
(615)   32'h00000000
(616)   32'h00000000
(617)   32'h00000000
(618)   32'h00000000
Now       358500000 ps
Cursor 1  346003176 ps
```

8. Optional: Complete the processor design with mechanism to update Instruction memory (I-Mem). (You can use just switches & push buttons to update I-Mem, or use other interfaces such as UART)

PERFORMANCE ANALYSIS

I. Performance and area analysis

Max speed of the design: 50.123MHz

Actually 50.123MHz is the max speed of the design including FSM which is used to make a bridge between users (or I/O) and the processor we design using FPGA board. The max speed of processor we design is much faster than 50.123MHz.

Resource Utilization:

Resources	Number
Number of Slice Registers	1188
Number of Slice LUTs	2960

Number of IOBs	53

Number of Gates:

Gates	Number
RAMs	5
Adders/Subtractors	4
Registers	46
Comparators	2
Multiplexers	56
Xors	1

II. Details about how you verified your overall design

As shown in Simulation section, ALU and Decoder will be tested at first.

ALU:

OP1	**OP2**	**ALU_out**	**Function**	**Operation**
0x00000001	0x00000002	0x00000003	0x00	ADD
0x00000001	0x00000002	0xFFFFFFFF	0x01	ADDI
0x00000001	0x00000002	0x00000000	0x02	SUBI
0x00000001	0x00000002	0x00000003	0x03	ANDI
0x00000001	0x00000002	0xFFFFFFFC	0x04	ORI
0x00000001	0x00000002	0x00000004	0x05	SHL

0x00000001	0x00000002	0x40000000	0x06	SHR

Decoder:

MNEMONIC	Instruction	I-Type	R-Type	J-Type	isStore	isLoad	ALUop	BranchCMD	IsHalt
ADD	0x00FFFF10	0	1	0	0	0	0x10	00	0
SUB	0x00FFFF11	0	1	0	0	0	0x11	00	0
AND	0x00FFFF12	0	1	0	0	0	0x12	00	0
OR	0x00FFFF13	0	1	0	0	0	0x13	00	0
NOR	0x00FFFF14	0	1	0	0	0	0x14	00	0
ADDI	0x04FFFFFF	1	0	0	0	0	0x00	00	0
SUBI	0x08FFFFFF	1	0	0	0	0	0x01	00	0
ANDI	0x0CFFFFFF	1	0	0	0	0	0x02	00	0
ORI	0x10FFFFFF	1	0	0	0	0	0x03	00	0
SHL	0x14FFFFFF	1	0	0	0	0	0x00	00	0
SHR	0x18FFFFFF	1	0	0	0	0	0x00	00	0
LW	0x1CFFFFFF	1	0	0	0	1	0x00	00	0
SW	0x20FFFFFF	1	0	0	1	0	0x00	00	0
BLT	0x24FFFFFF	1	0	0	0	0	0x05	01	0
BEQ	0X28FFFFFF	1	0	0	0	0	0x06	10	0
BNE	0X2CFFFFFF	1	0	0	0	0	0x00	11	0
JMP	0X30FFFFFF	0	0	1	0	0	0x00	00	0
HAL	0XFCFFFFFF	0	0	0	0	0	0x00	00	1

After that, each instructions will be used to verify the correctness of the design.

Instruction	Related RF Register1	Related RF Register2	Related RF Register3 or Imm	Related DMEM Register
0x00FFFF10	7	31	31	~
0x00FFFF11	7	31	31	~
0x00FFFF12	7	31	31	~
0x00FFFF13	7	31	31	~
0x00FFFF14	7	31	31	~
0x04FFFFFF	7	31	FFFF	~
0x08FFFFFF	7	31	FFFF	~
0x0CFFFFFF	7	31	FFFF	~
0x10FFFFFF	7	31	FFFF	~
0x14FFFFFF	7	31	FFFF	~
0x18FFFFFF	7	31	FFFF	~
0x1CFFFFFF	7	31	FFFF	FFFFFFFF
0x20FFFFFF	7	31	FFFF	FFFFFFFF
0x24FFFFFF	7	31	FFFF	~
0X28FFFFFF	7	31	FFFF	~
0X2CFFFFFF	7	31	FFFF	~
0X30FFFFFF	7	~	3FFFFFF	~
0XFCFFFFFF	~	~	~	~

Sample program will be executed next to verify the design.

	rs	rt	imm	rd
$1 ADDI R1, R0, 7	0	0	7	7
$2 ADDI R2, R0, 8	0	0	8	8
$3 ADD R3, R1, R2	7	8	~	15
HAL	~	~	~	~

RC5 is the last program to be used as verifying program.