NYU - 6463 Processor Design Report

Arora Siddhartha (sa4186), Bou Khalil Carl (cbk289), Kwan Kevin (kk3439), Moyseyev Dmytro (dm3511), Plaudis Roberts (rp1730), Sharma Anusha (as10553)

EL - 6463 Advanced Computer Hardware Design
Department of Electrical & Computer Engineering
Prof. Ramesh Karri

NYU | TANDON SCHOOL OF ENGINEERING

Table of Contents

I. Processor Description and Implementation

The processor implemented for the final project is a single cycle 32-bit processor. This means that it fully executes an instruction in one clock cycle only. A 32-bit processor is a processor having 32-bits long instructions. This instruction can be divided depending on the type of the instruction (R, I or J), but one constant is the opcode which is defined by the first 6 bits of the instructions starting at the Most Significant Bit (MSB). Below is a figure showing the different fields of an instruction.

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)

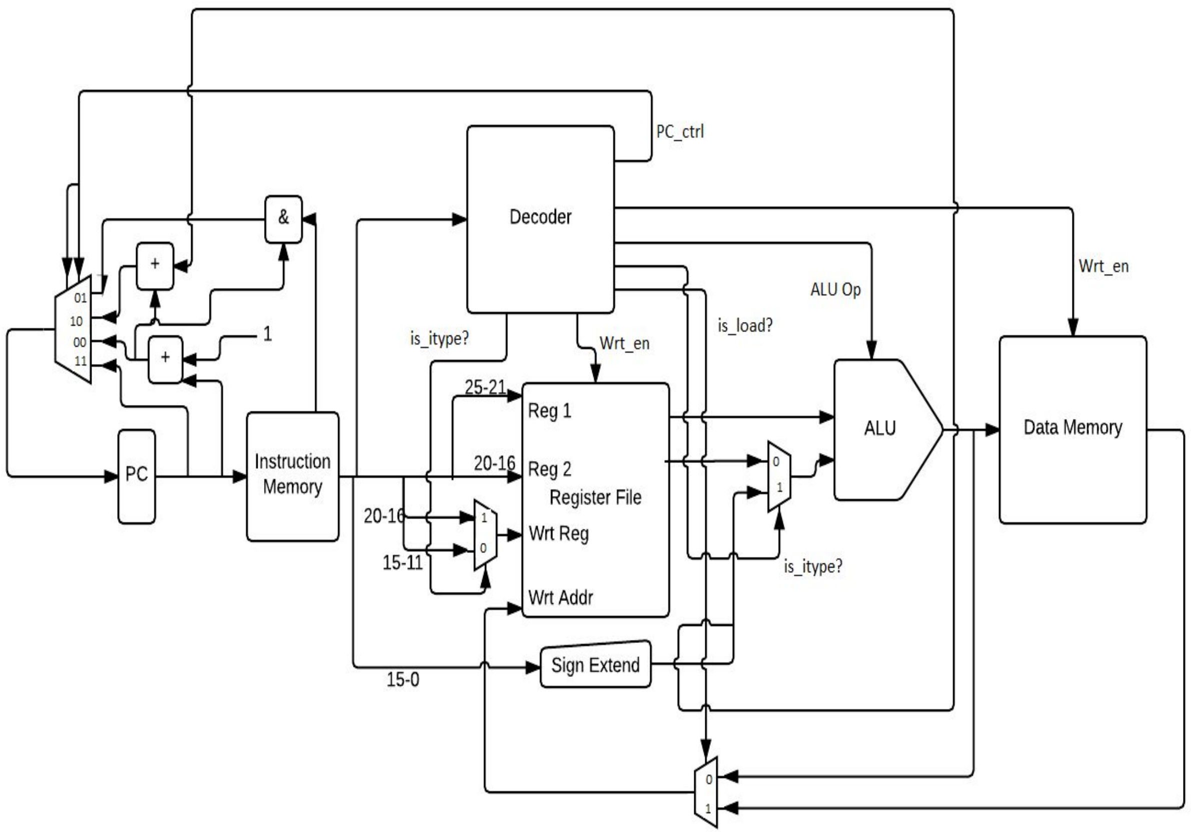
Opcode (6-bits)	Address (26-bits)

Instruction content for R, I and J type instructions, respectively

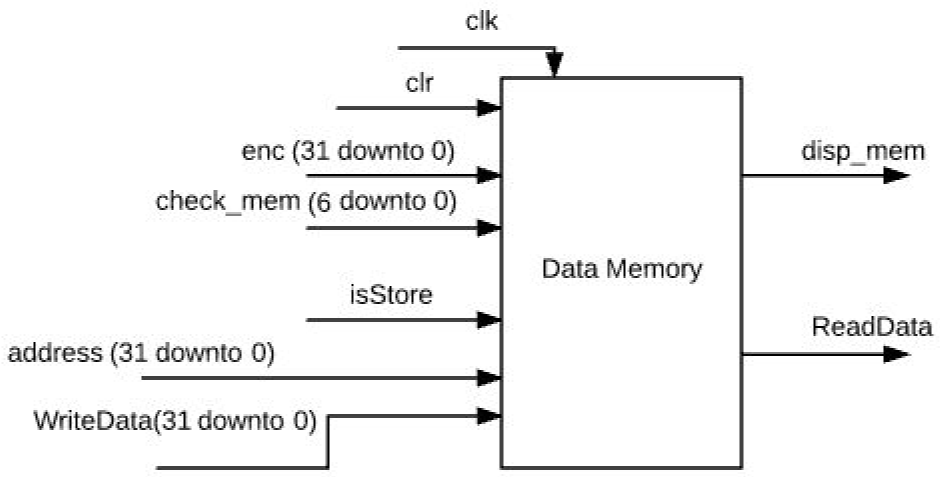
A processor is an ensemble of components that perform different crucial tasks allowing it to work properly. A big number of instructions can be implemented on a processor but for this project only a few were (Refer to table 1 of the project description PDF). The components that constitute our processor are the following: the Instruction Memory (IM) in which the code to execute will be placed, the 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, the Arithmetic Logic Unit (ALU) which is responsible for the arithmetic operations such as addition and subtraction, the decoder which breaks down the instruction to identify the value of the control signals as the instruction type for instance and finally, the Data Memory (DM) in which values can be stored. The DM is a crucial part for the RC5 execution on this processor and it will be detailed in the designated paragraph (VI). Below, in paragraph II, is the block diagram of the processor implemented. It was implemented using components for each one of the blocks mentioned above and they were all integrated in a top level module. The block diagram below shows only the top level module and treats the components as black boxes having inputs and outputs. The processor was implemented using this methodology to facilitate debugging and testing.
The processor was exported and programmed on the Nexys 4 DDR board and thus its inputs and outputs had to be mapped to some signals in the code. In the following will be described the functionality of every I/O port used.
The first user input that is indispensable to every design is the clear. The center button out of the five is the one fulfilling this task. When pushed, the system will reset the whole processor. This

means that the Program Counter (PC) is pointing to the first instruction in the IM, the registers are all set to 0 again and so is the DM, except in some cases as it will be discussed below (paragraph VI).
The other user input added was to be able to run the program using single stepping, that is to execute an instruction only when the user wants to, opposed to a normal execution where the processor will run the program automatically and move from an instruction to another every clock cycle. In other words, the clock had to be controlled by the user. To this end, the two first switches, 15 and 14, were used. When the switch 14 is set, the processor will execute the code normally. However, when is it not ('0'), the clock is now emulated by the switch 15. So every time the switch moves from 0 to 1, it simulates a rising edge of the clock and so the processor will move to the next instruction. Similarly, when it goes from 0 to 1, it is a falling edge of the clock. When using single stepping, the clock is the 15th switch.
Thanks to the 7-segment display, various information can be shown to the user. With this processor, the user can see the instruction itself, the PC, the values inside the registers and the values inside the DM. To choose between these option, the user has to change the values of the switches 7 and 8. When both are to 0, the instruction will be displayed. When both are set 1, the value of the PC will be shown on the display. When the 8th bit is set to 0 and the 7th to 1, the content of the RF can be seen. When in this mode, the switches from 4 down to 0 can be used to choose what register the user wants to look at. Since there are 32 registers, 5 bits are enough to access all of them. Finally, when the 8th switch is set and the 7th is to 0, the content of the DM can be seen. The switches from 6 down to 0 are used to choose what memory location one wants to look at. With 7 bits to access the locations, the memory can be as big as 128 locations. For this project, this memory is more than big enough.
Finally, the switch 9 is also used by the user but its functionality is specific to the RC5. When it is set to 0, the RC5 will decrypt the plaintext, while if it's set to 1, the RC5 will encrypt the plaintext. Details about that can be found in paragraph VI.

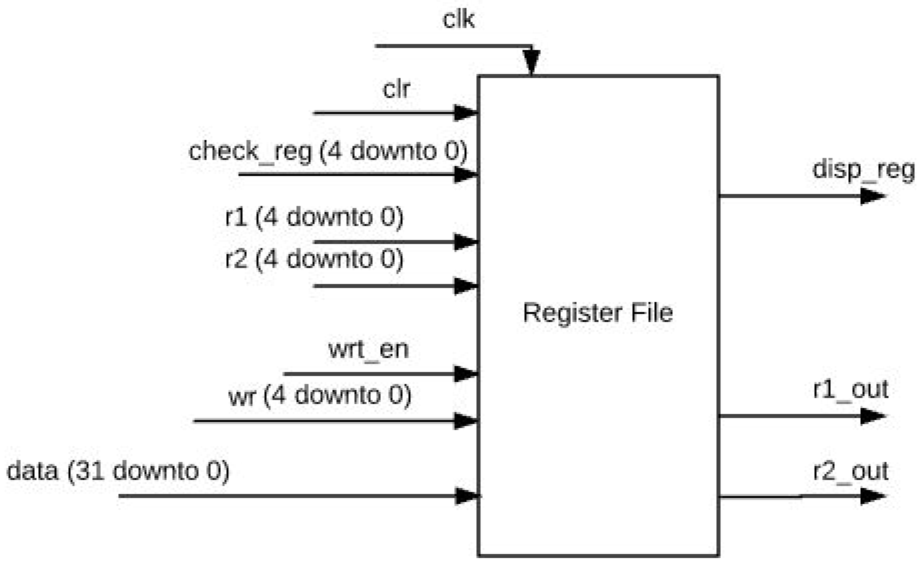
II. Block Diagrams and finite state machine



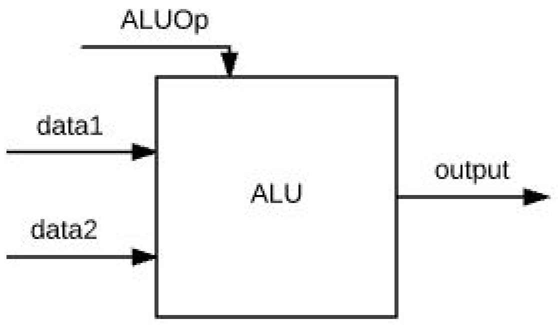
Data Memory:

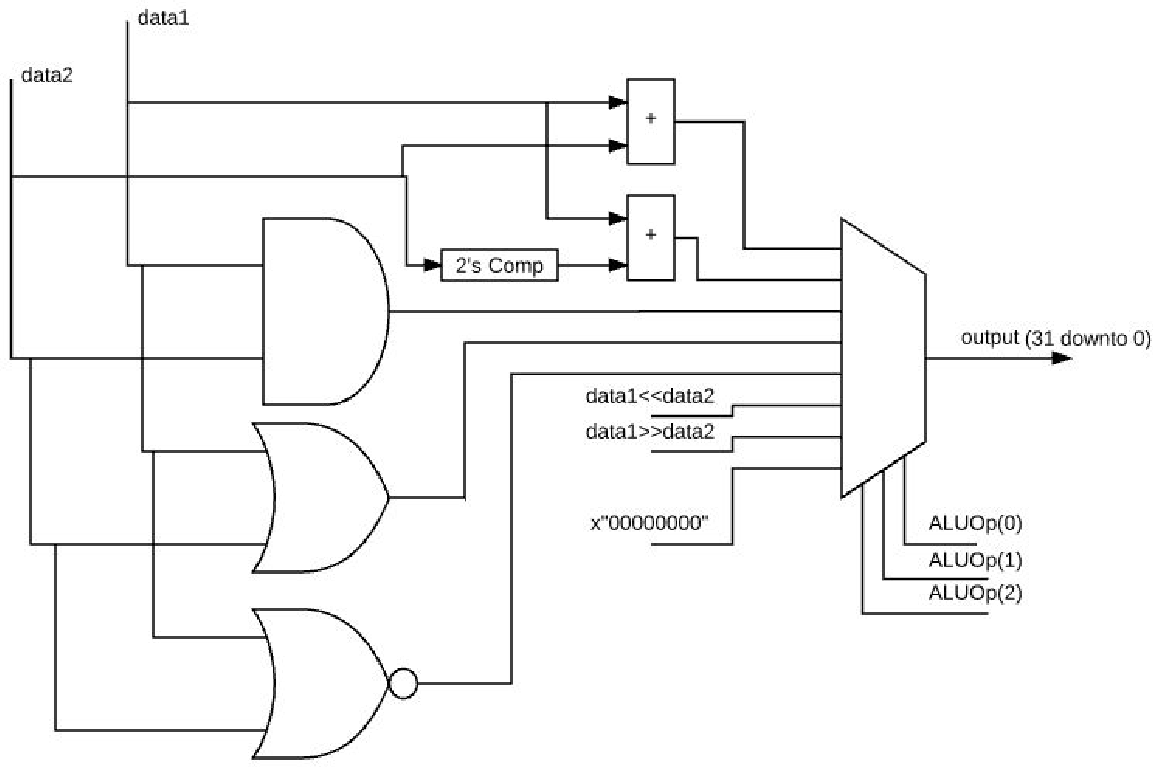


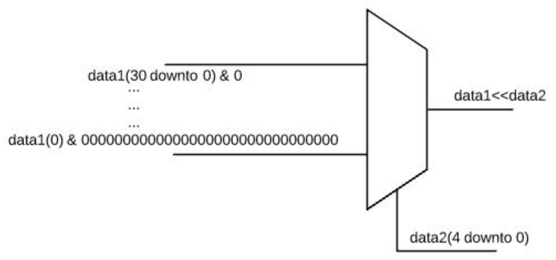
Registers:

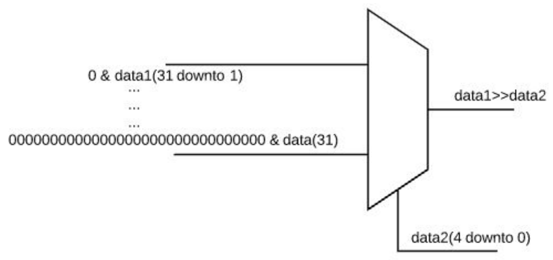


ALU:

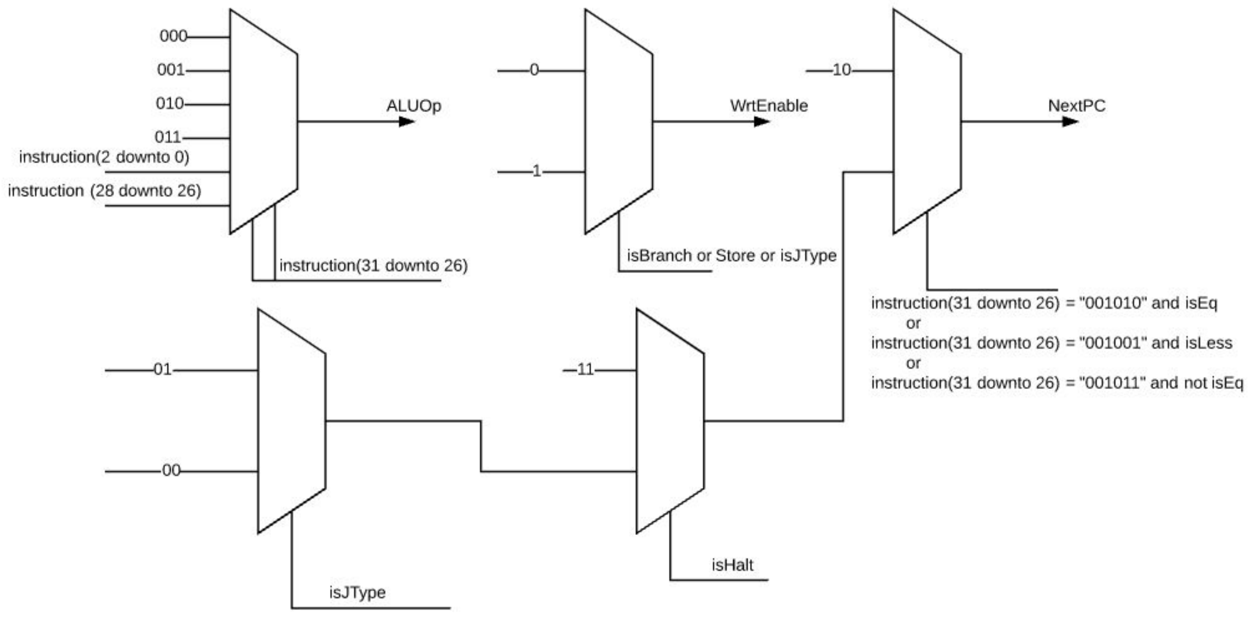

ALUOp
data1
data2
ALU
output


data1
data2
+
+
2's Comp
output (31 downto 0)
data1<<data2
data1>>data2
x"00000000"
ALUOp(0)
ALUOp(1)
ALUOp(2)

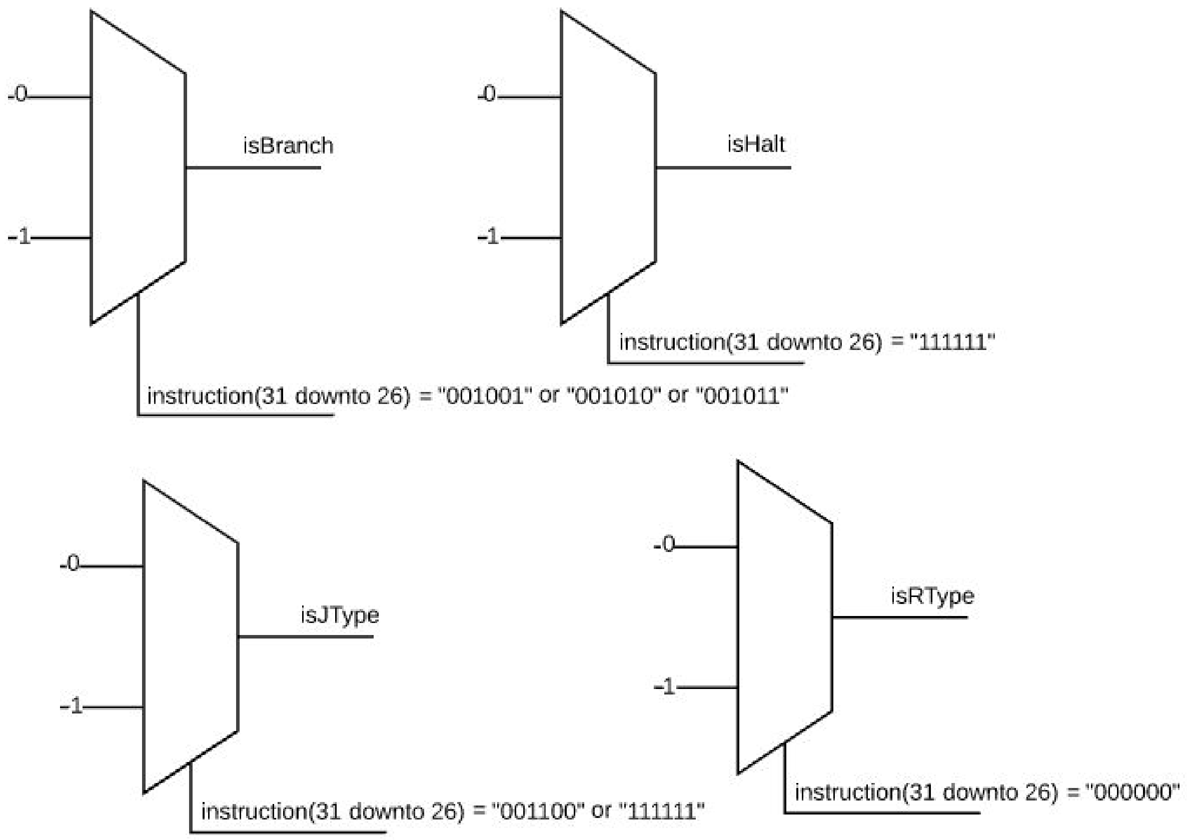

data1(30 downto 0) & 0
...
...
...
data1(0) & 0000000000000000000000000000000
data1<<data2
data2(4 downto 0)


0 & data1(31 downto 1)
...
...
...
0000000000000000000000000000000 & data(31)
data1>>data2
data2(4 downto 0)

Decoder:



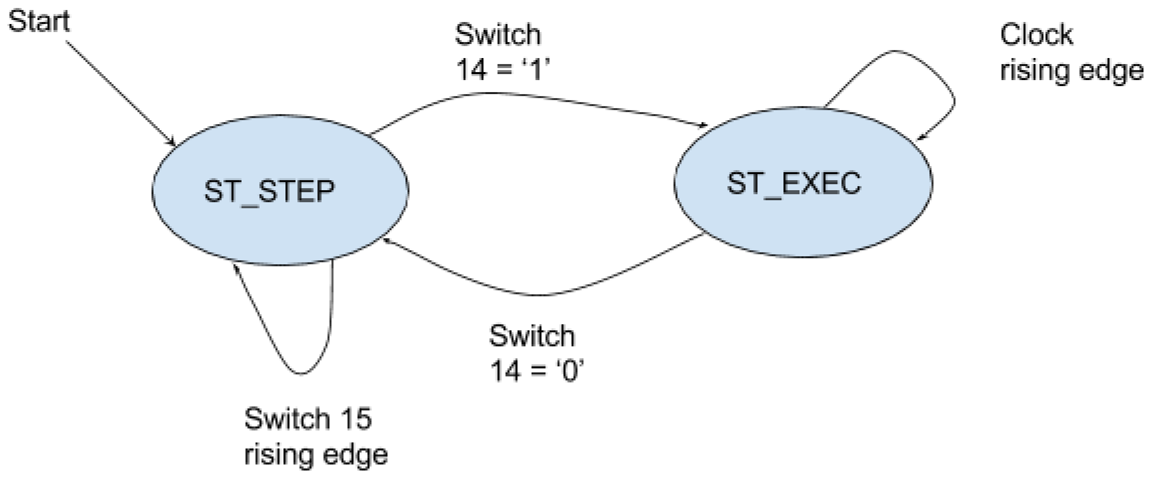
000
001
010
011
instruction(2 downto 0)
instruction (28 downto 26)
instruction(31 downto 26)
ALUOp
0
1
isBranch or Store or isJType
WrtEnable
10
NextPC
instruction(31 downto 26) = "001010" and isEq
or
instruction(31 downto 26) = "001001" and isLess
or
instruction(31 downto 26) = "001011" and not isEq
01
00
isJType
11
isHalt



0
1
isBranch
instruction(31 downto 26) = "001001" or "001010" or "001011"
0
1
isHalt
instruction(31 downto 26) = "111111"
0
1
isJType
instruction(31 downto 26) = "001100" or "111111"
0
1
isRType
instruction(31 downto 26) = "000000"

Finite State Machine

A finite state machine is used to choose between a single stepping execution or a normal execution.



III. Components of the Processor

Arithmetic Logic Unit (ALU)

ALU is one of the fundamental blocks of our processor. It is responsible for arithmetic, logic, and immediate operations such as ADD, SUB, AND, OR, NOR, shift left, and shift right. The unit receives two data signals data1 and data2: one from the Register File, and the second one either from the register file or from the Instruction Memory directly. Based on the control signal ALUOp that comes from the Decoder the ALU performs a particular operation. The output is written as an address to the Data Memory Unit.

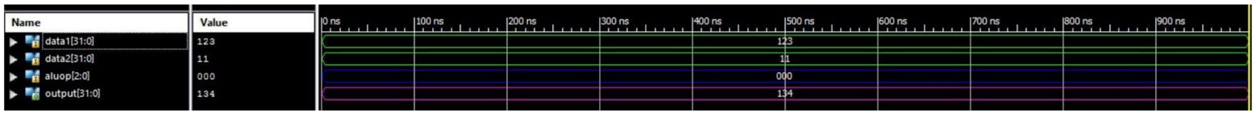


Fig. 1 ADD Functional Simulation

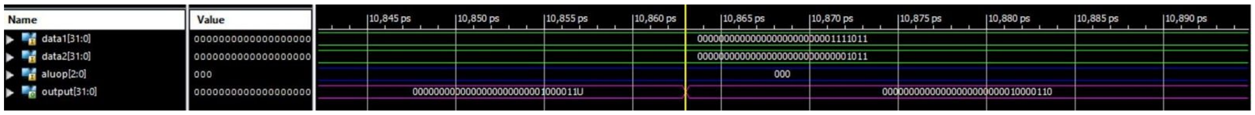

Name
Value
data1[31:0] 00000000000000000000000011111011
data2[31:0] 00000000000000000000000000001011
aluop[2:0] 000
output[31:0] 0000000000000000000000001000011U
00000000000000000000000010000110

Fig. 2 ADD Timing Simulation

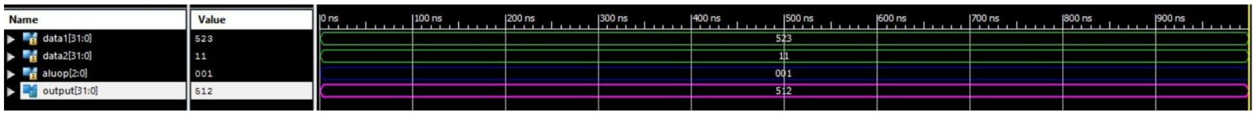


Fig. 3 SUB Functional Simulation

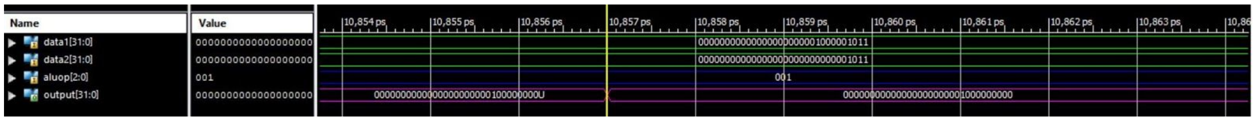

Name
Value
data1[31:0] 00000000000000000000001000001011
data2[31:0] 00000000000000000000000000001011
aluop[2:0] 001
output[31:0] 00000000000000000000001000000000U
00000000000000000000001000000000

Fig. 4 SUB Timing Simulation

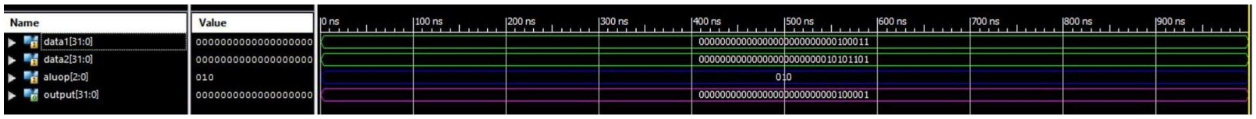

Name
Value
data1[31:0] 00000000000000000000000000100011
data2[31:0] 00000000000000000000000010101101
aluop[2:0] 010
output[31:0] 00000000000000000000000000100001

Fig. 5 AND Functional Simulation

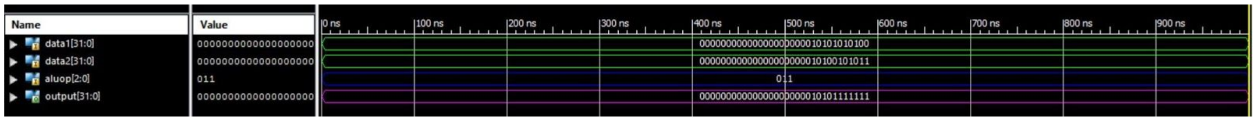

Name
Value
data1[31:0] 00000000000000000000010101010100
data2[31:0] 00000000000000000000010100101011
aluop[2:0] 011
output[31:0] 00000000000000000000010101111111

Fig. 6 AND Timing Simulation

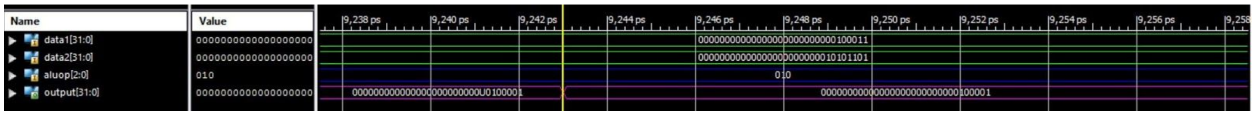


Fig. 7 OR Functional Simulation

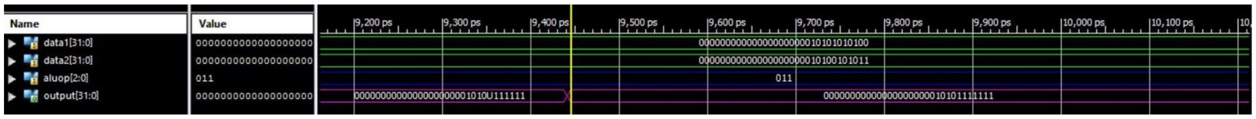


Fig. 8 OR Timing Simulation

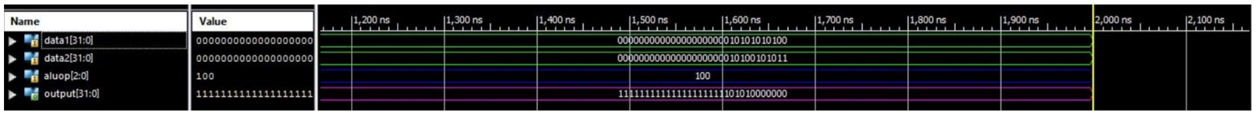


Fig. 9 NOR Functional Simulation

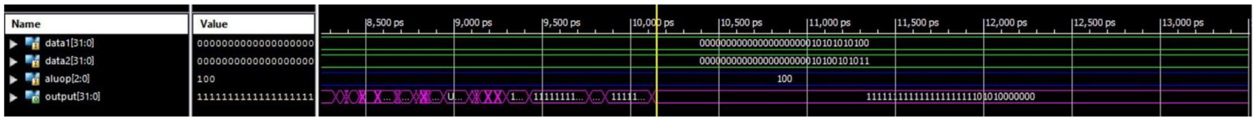


Fig. 10 NOR Timing Simulation

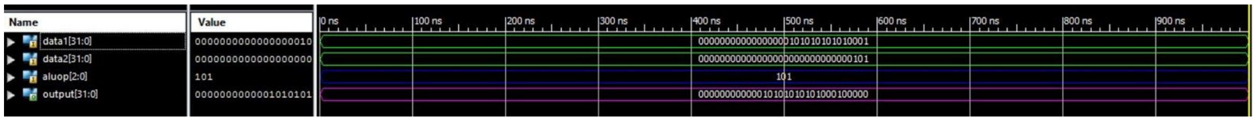


Fig. 11 Shift Left Functional Simulation

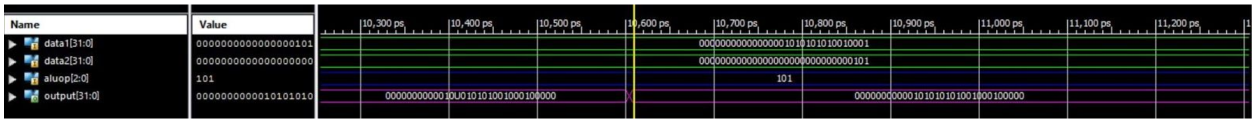


Fig. 12 Shift Left Timing Simulation

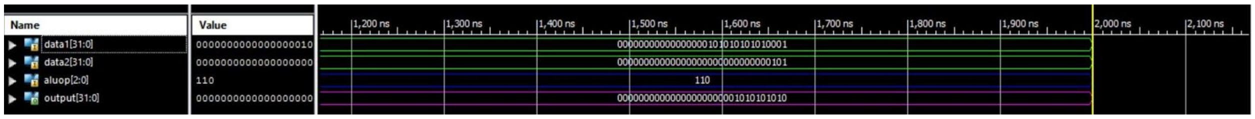


Fig. 13 Shift Right Functional Simulation

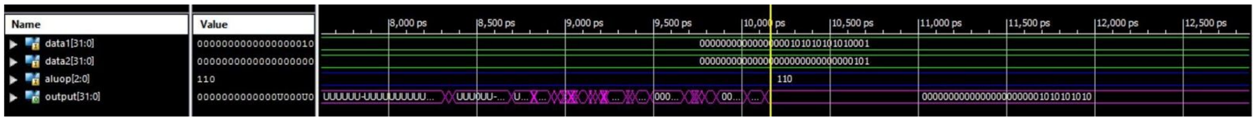
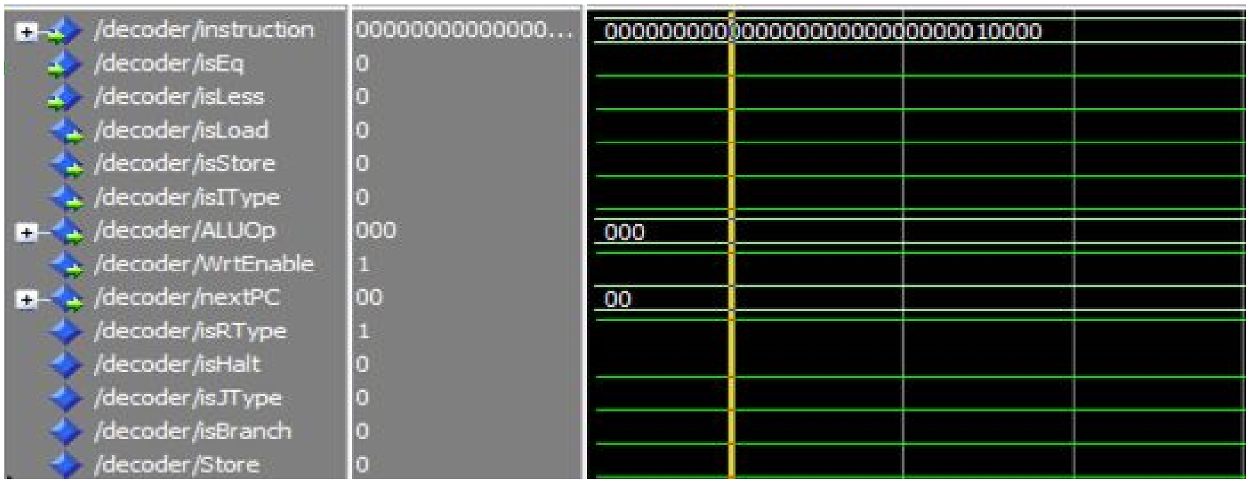


Fig. 14 Shift Right Timing Simulation

Decoder Unit

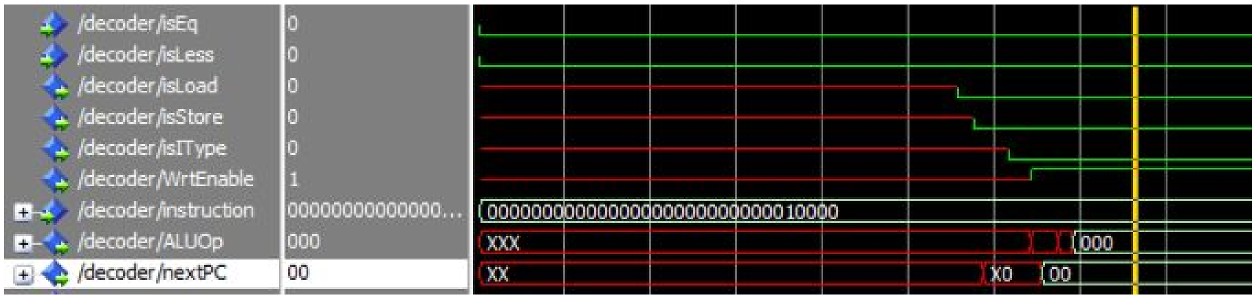
The Decoder Unit is responsible for generating the proper control signals for the rest of the blocks in the design hierarchy. It makes use of the first 6-bits and the last 6-bits of the 32-bit input instruction to generate the appropriate control signals for the microprocessor.

1. Functional and Timing Simulations for R-TYPE instructions



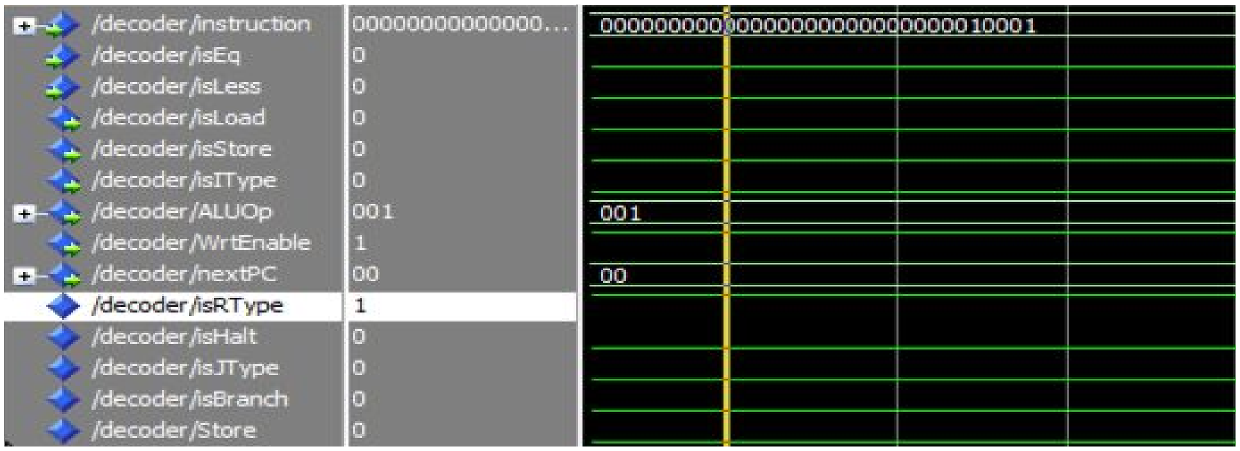
```
/decoder/instruction   00000000000000...   00000000000000000000000000010000
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       0
/decoder/ALUOp         000                 000
/decoder/WrtEnable     1
/decoder/nextPC        00                  00
/decoder/isRType       1
/decoder/isHalt        0
/decoder/isJType       0
/decoder/isBranch      0
/decoder/Store         0
```

Fig. 15 ADD Functional Simulation



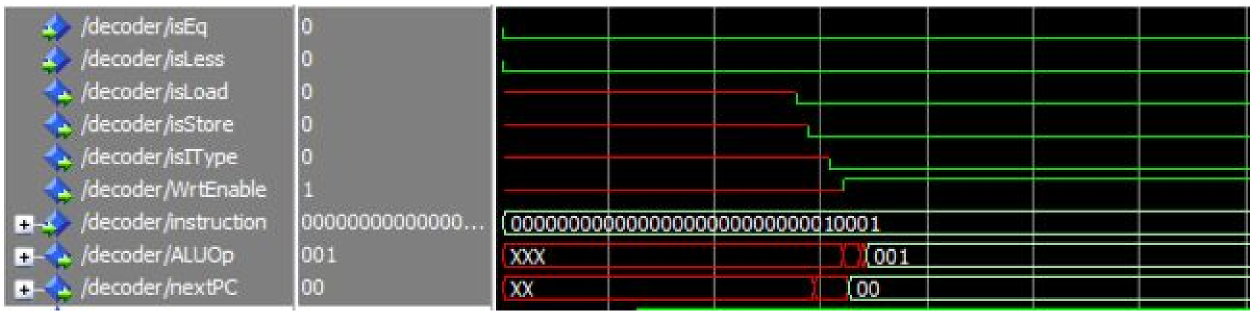
```
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       0
/decoder/WrtEnable     1
/decoder/instruction   00000000000000...   00000000000000000000000000010000
/decoder/ALUOp         000                 XXX  000
/decoder/nextPC        00                  XX  X0  00
```

Fig. 16 ADD Timing Simulation



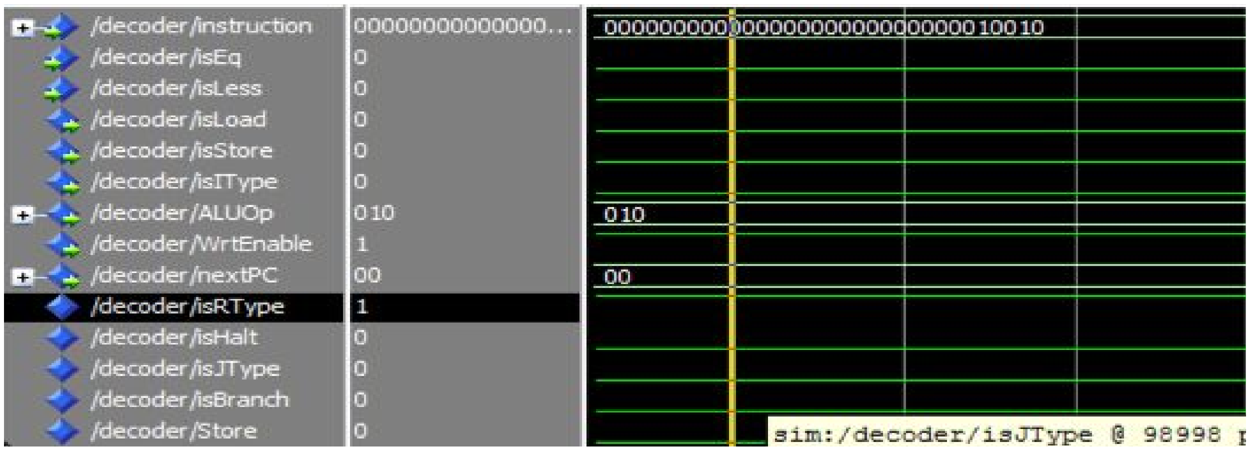
```
/decoder/instruction   00000000000000...
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       0
/decoder/ALUOp         001
/decoder/WrtEnable     1
/decoder/nextPC        00
/decoder/isRType       1
/decoder/isHalt        0
/decoder/isJType       0
/decoder/isBranch      0
/decoder/Store         0
```

Fig. 17 SUB Functional Simulation



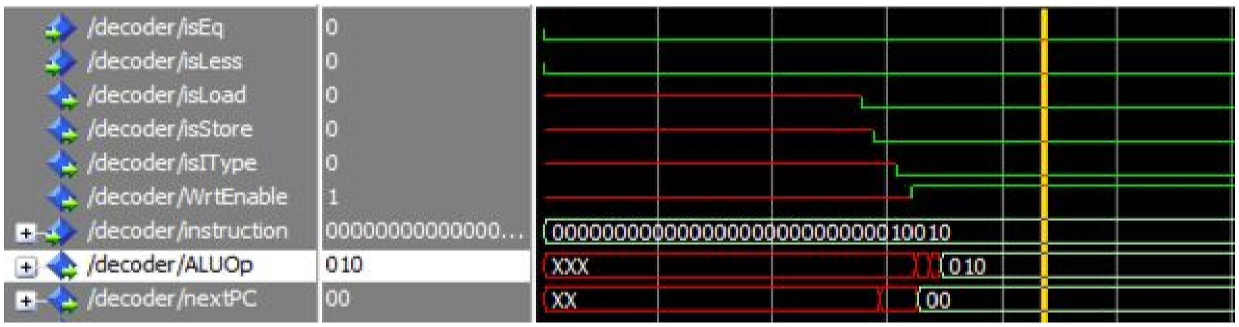
```
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       0
/decoder/WrtEnable     1
/decoder/instruction   00000000000000...
/decoder/ALUOp         001
/decoder/nextPC        00
```

Fig. 18 SUB Timing Simulation



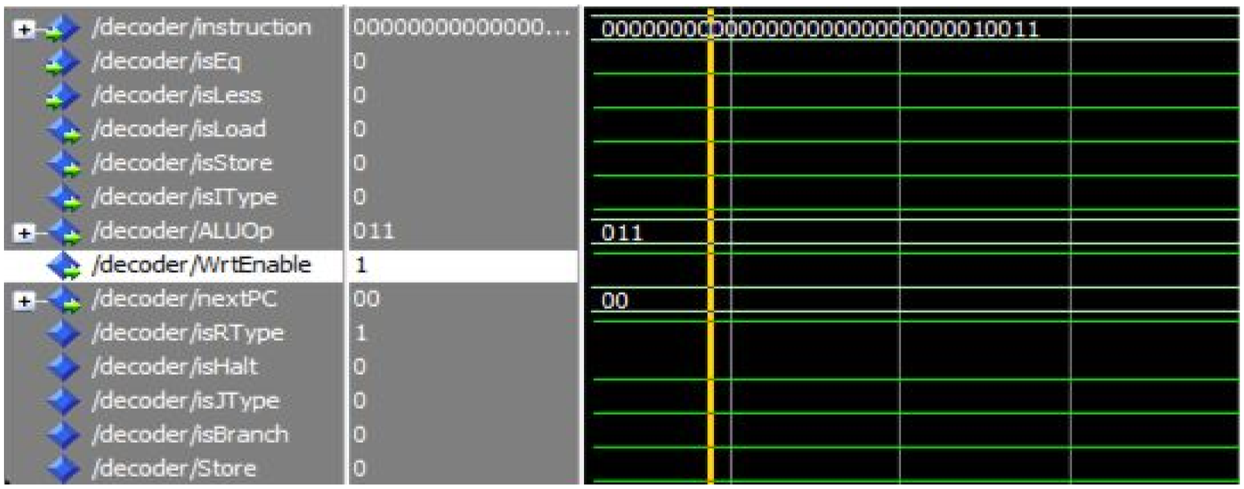
```
/decoder/instruction   00000000000000...
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       0
/decoder/ALUOp         010
/decoder/WrtEnable     1
/decoder/nextPC        00
/decoder/isRType       1
/decoder/isHalt        0
/decoder/isJType       0
/decoder/isBranch      0
/decoder/Store         0
```

Fig 19. AND Functional Simulation



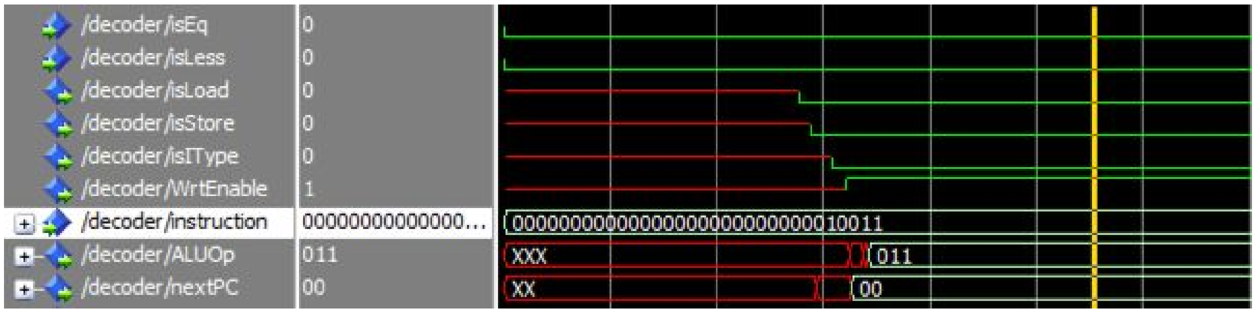
/decoder/isEq	0
/decoder/isLess	0
/decoder/isLoad	0
/decoder/isStore	0
/decoder/isIType	0
/decoder/WrtEnable	1
/decoder/instruction	0000000000000...
/decoder/ALUOp	010
/decoder/nextPC	00

Fig. 20 AND Timing Simulation



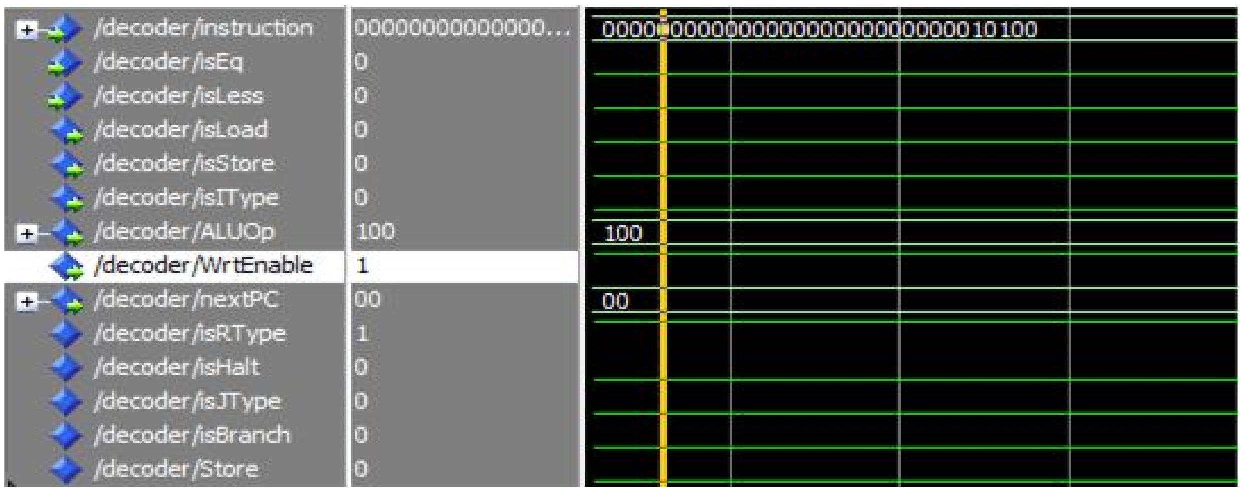
/decoder/instruction	0000000000000...
/decoder/isEq	0
/decoder/isLess	0
/decoder/isLoad	0
/decoder/isStore	0
/decoder/isIType	0
/decoder/ALUOp	011
/decoder/WrtEnable	1
/decoder/nextPC	00
/decoder/isRType	1
/decoder/isHalt	0
/decoder/isJType	0
/decoder/isBranch	0
/decoder/Store	0

Fig. 21 OR Functional simulation



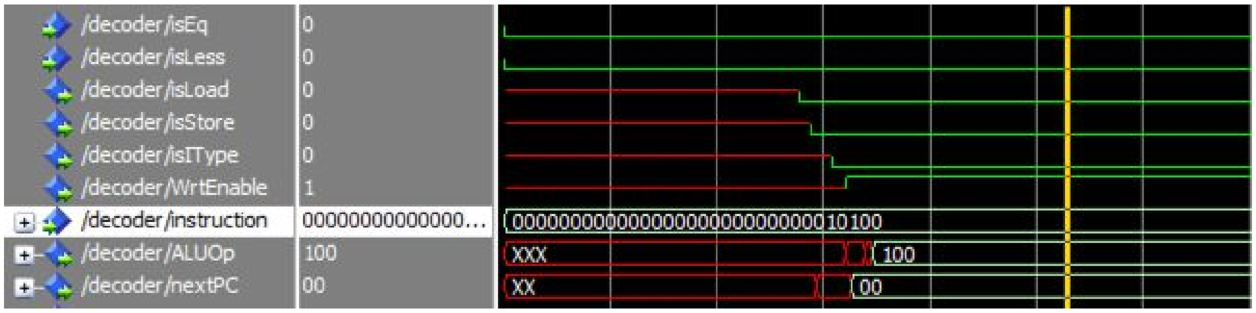
/decoder/isEq	0
/decoder/isLess	0
/decoder/isLoad	0
/decoder/isStore	0
/decoder/isIType	0
/decoder/WrtEnable	1
/decoder/instruction	0000000000000...
/decoder/ALUOp	011
/decoder/nextPC	00

Fig. 22 OR Timing Simulation



/decoder/instruction	0000000000000...
/decoder/isEq	0
/decoder/isLess	0
/decoder/isLoad	0
/decoder/isStore	0
/decoder/isIType	0
/decoder/ALUOp	100
/decoder/WrtEnable	1
/decoder/nextPC	00
/decoder/isRType	1
/decoder/isHalt	0
/decoder/isJType	0
/decoder/isBranch	0
/decoder/Store	0

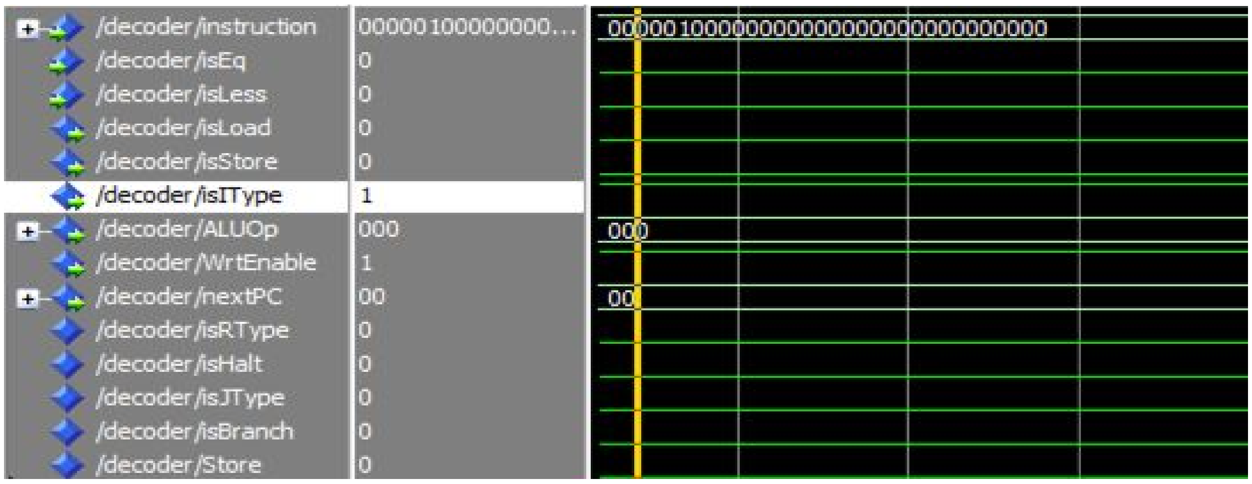
Fig. 23 NOR Functional simulation



/decoder/isEq	0
/decoder/isLess	0
/decoder/isLoad	0
/decoder/isStore	0
/decoder/isIType	0
/decoder/WrtEnable	1
/decoder/instruction	00000000000000...
/decoder/ALUOp	100
/decoder/nextPC	00

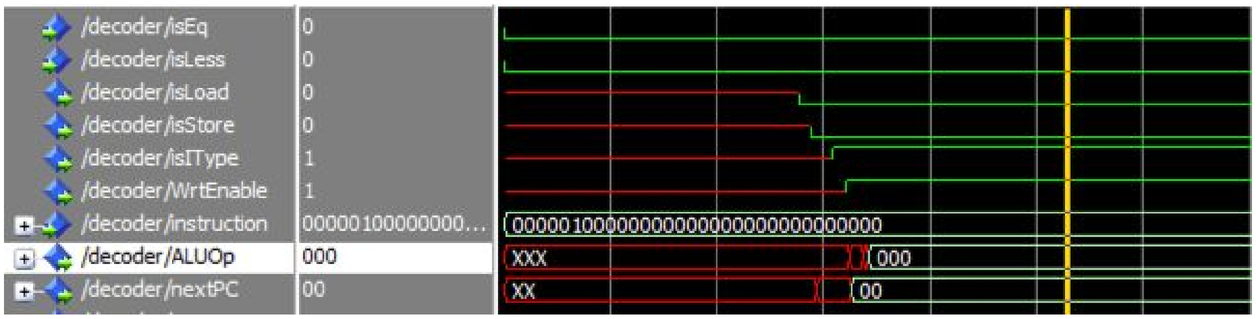
Fig 24. NOR Timing Simulation

2. Functional and Timing Simulation for I-TYPE instructions



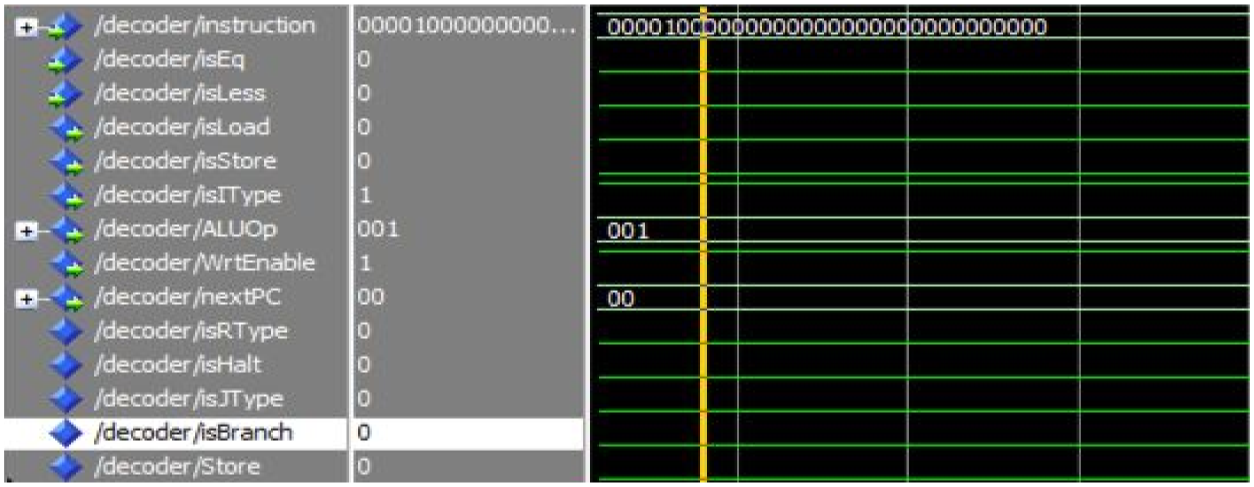
/decoder/instruction	00000100000000...
/decoder/isEq	0
/decoder/isLess	0
/decoder/isLoad	0
/decoder/isStore	0
/decoder/isIType	1
/decoder/ALUOp	000
/decoder/WrtEnable	1
/decoder/nextPC	00
/decoder/isRType	0
/decoder/isHalt	0
/decoder/isJType	0
/decoder/isBranch	0
/decoder/Store	0

Fig 25. ADDI Functional Simulation



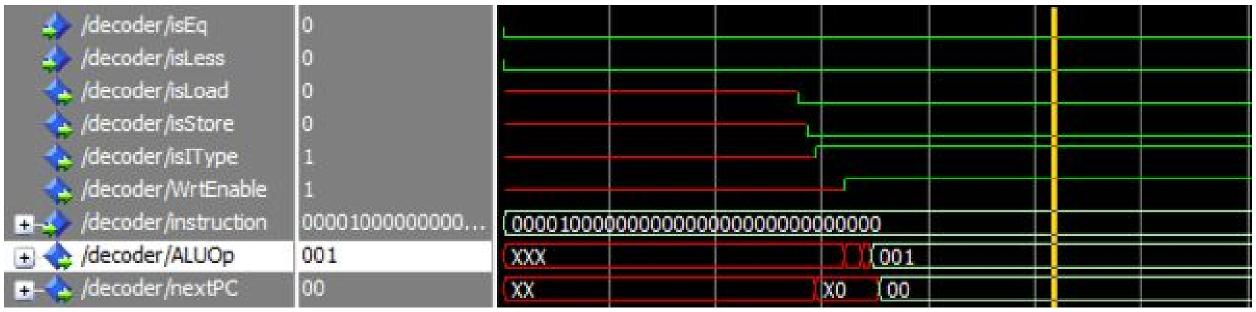
```
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       1
/decoder/WrtEnable     1
/decoder/instruction   00000100000000...   00000100000000000000000000000000
/decoder/ALUOp         000                 XXX  000
/decoder/nextPC        00                  XX   00
```

Fig. 26 ADDI Timing Simulation



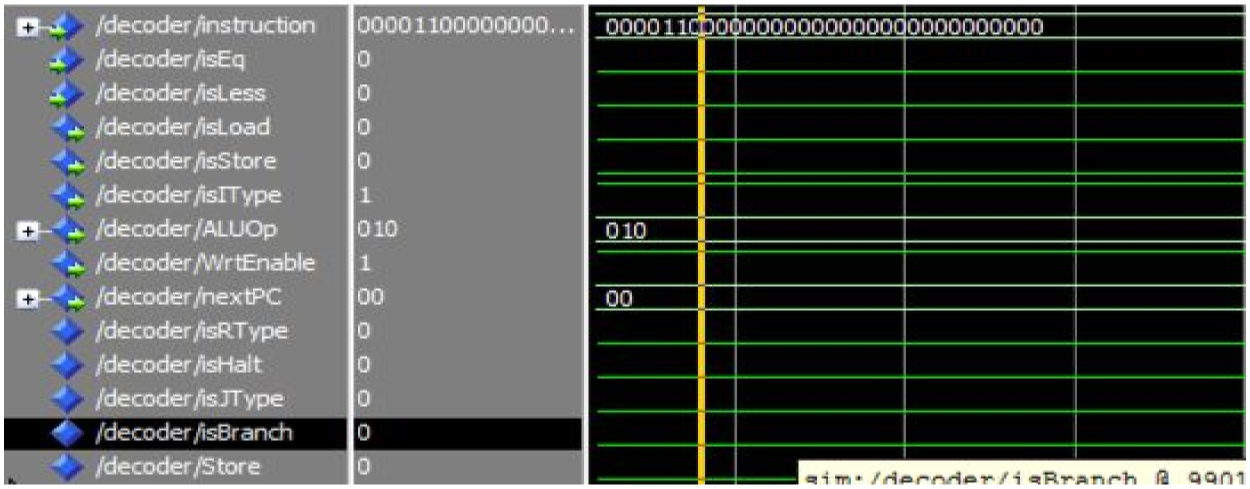
```
/decoder/instruction   00001000000000...   00001000000000000000000000000000
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       1
/decoder/ALUOp         001                 001
/decoder/WrtEnable     1
/decoder/nextPC        00                  00
/decoder/isRType       0
/decoder/isHalt        0
/decoder/isJType       0
/decoder/isBranch      0
/decoder/Store         0
```

Fig. 27 SUBI Functional Simulation



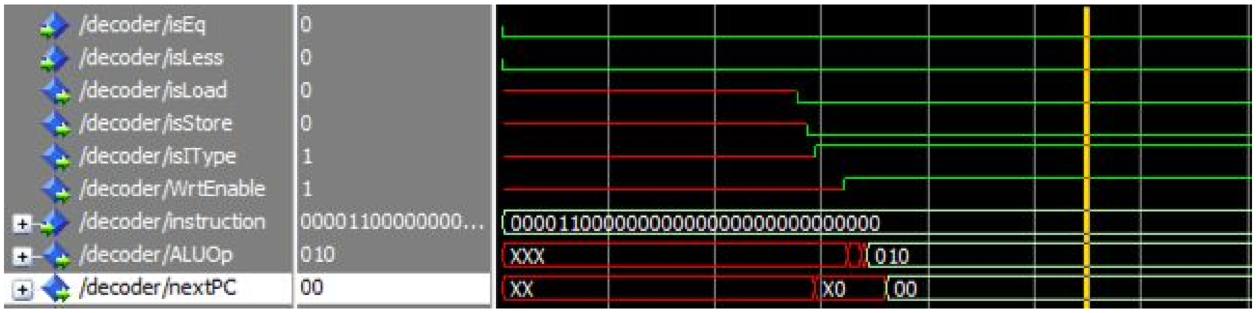
```
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       1
/decoder/WrtEnable     1
/decoder/instruction   00001000000000...   00001000000000000000000000000000
/decoder/ALUOp         001                 XXX  001
/decoder/nextPC        00                  XX  X0  00
```

Fig. 28 SUBI Timing Simulation



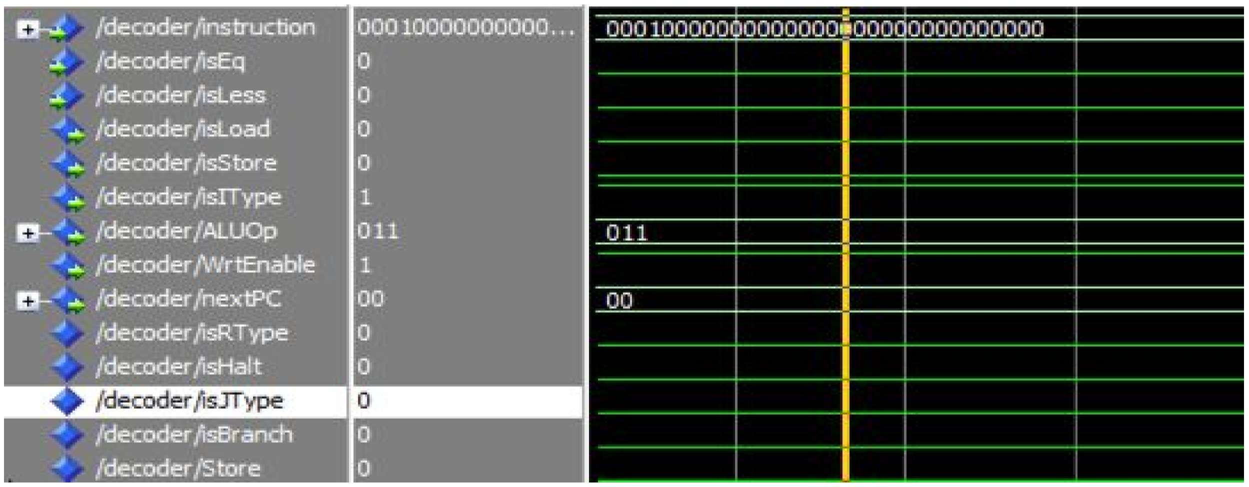
/decoder/instruction	00001100000000...
/decoder/isEq	0
/decoder/isLess	0
/decoder/isLoad	0
/decoder/isStore	0
/decoder/isIType	1
/decoder/ALUOp	010
/decoder/WrtEnable	1
/decoder/nextPC	00
/decoder/isRType	0
/decoder/isHalt	0
/decoder/isJType	0
/decoder/isBranch	0
/decoder/Store	0

Fig.29 ANDI Functional Simulation



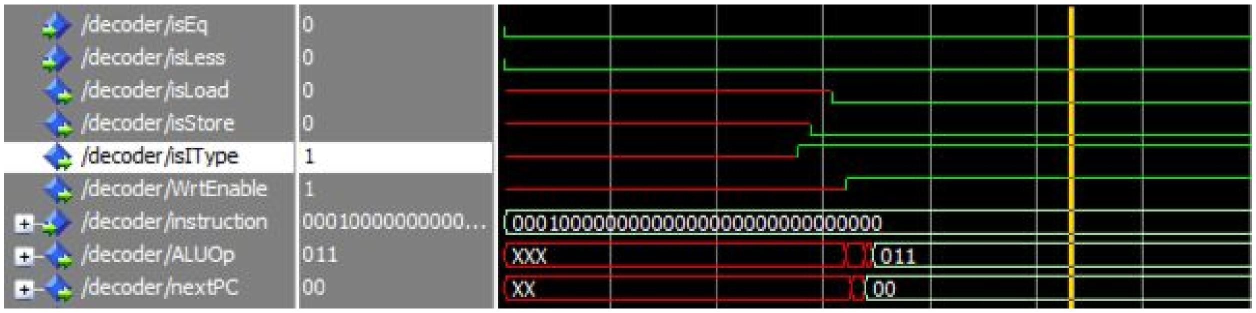
/decoder/isEq	0
/decoder/isLess	0
/decoder/isLoad	0
/decoder/isStore	0
/decoder/isIType	1
/decoder/WrtEnable	1
/decoder/instruction	00001100000000...
/decoder/ALUOp	010
/decoder/nextPC	00

Fig. 30 ANDI Timing Simulation



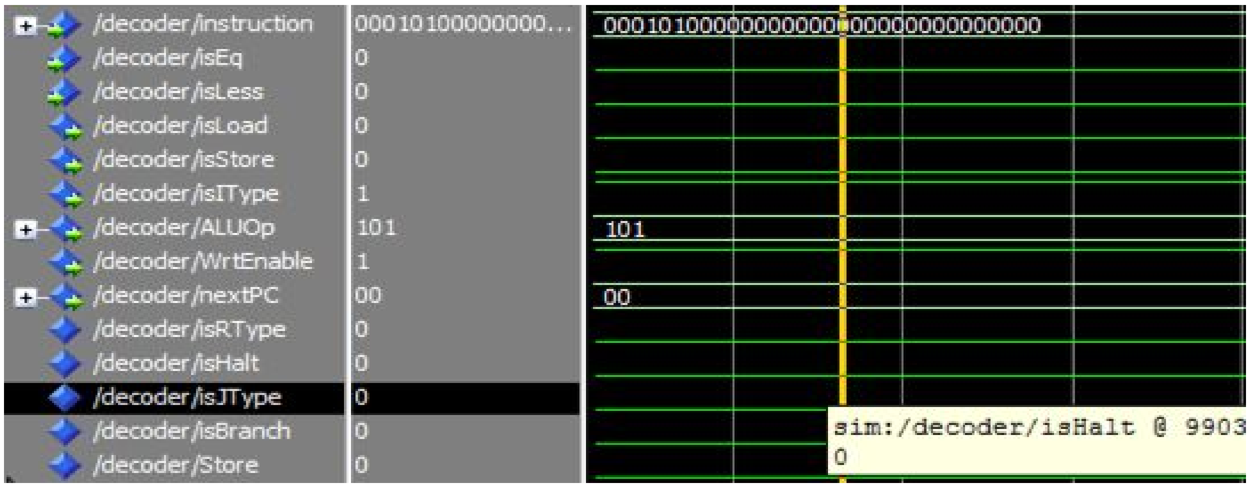
/decoder/instruction	00010000000000...
/decoder/isEq	0
/decoder/isLess	0
/decoder/isLoad	0
/decoder/isStore	0
/decoder/isIType	1
/decoder/ALUOp	011
/decoder/WrtEnable	1
/decoder/nextPC	00
/decoder/isRType	0
/decoder/isHalt	0
/decoder/isJType	0
/decoder/isBranch	0
/decoder/Store	0

Fig. 31 ORI Functional Simulation



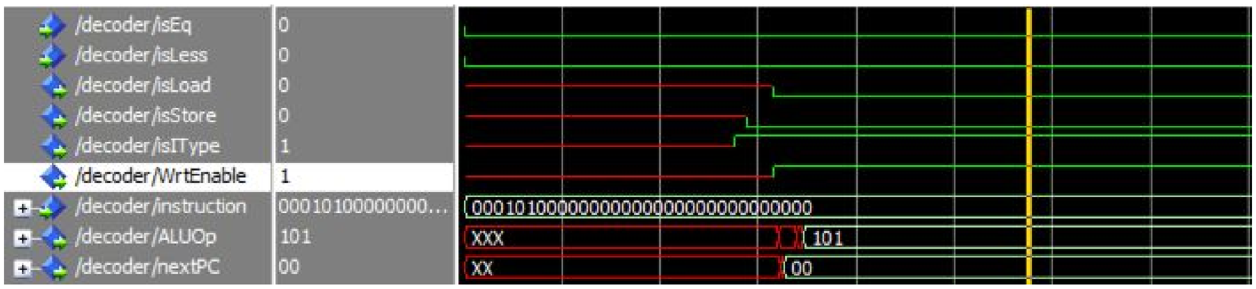
```
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       1
/decoder/WrtEnable     1
/decoder/instruction   00010000000000...
/decoder/ALUOp         011
/decoder/nextPC        00
```

Fig. 32 ORI Timing Simulation



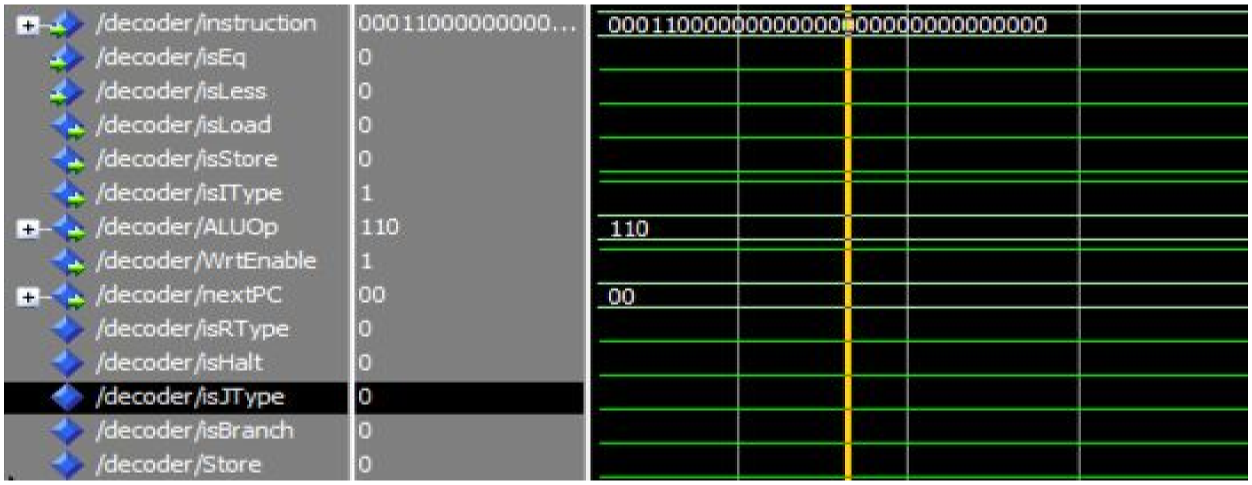
```
/decoder/instruction   00010100000000...
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       1
/decoder/ALUOp         101
/decoder/WrtEnable     1
/decoder/nextPC        00
/decoder/isRType       0
/decoder/isHalt        0
/decoder/isJType       0
/decoder/isBranch      0
/decoder/Store         0
```

Fig. 33 SHL Functional Simulation



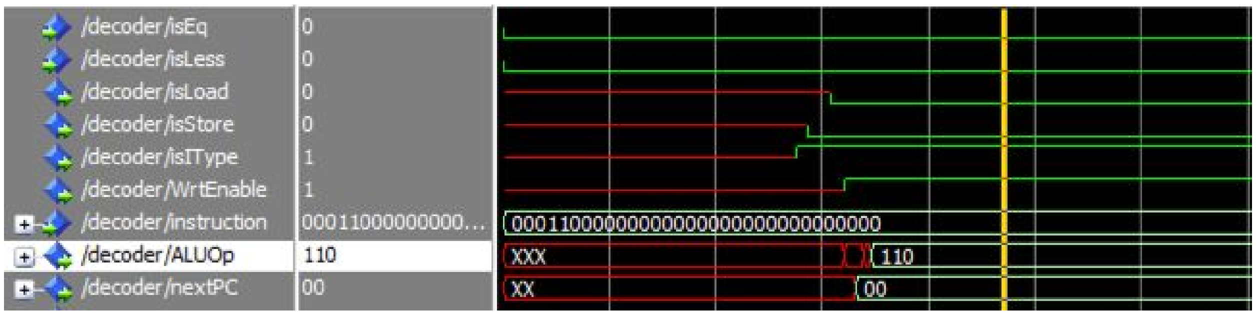
```
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       1
/decoder/WrtEnable     1
/decoder/instruction   00010100000000...
/decoder/ALUOp         101
/decoder/nextPC        00
```

Fig. 34 SHL Timing Simulation



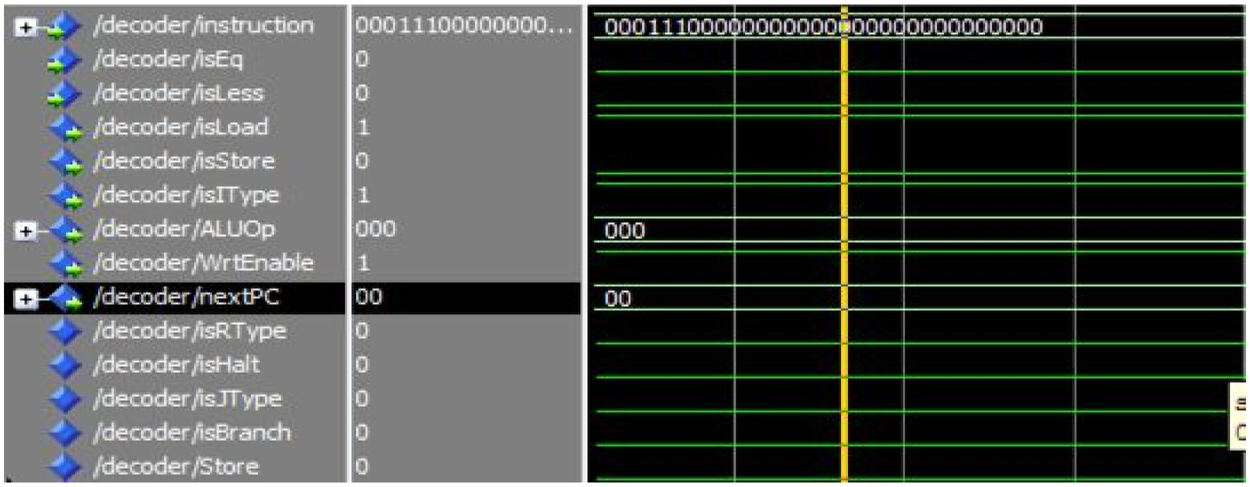
```
/decoder/instruction   00011000000000...   000110000000000000000000000000
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       1
/decoder/ALUOp         110                 110
/decoder/WrtEnable     1
/decoder/nextPC        00                  00
/decoder/isRType       0
/decoder/isHalt        0
/decoder/isJType       0
/decoder/isBranch      0
/decoder/Store         0
```

Fig. 35 SHR Functional Simulation



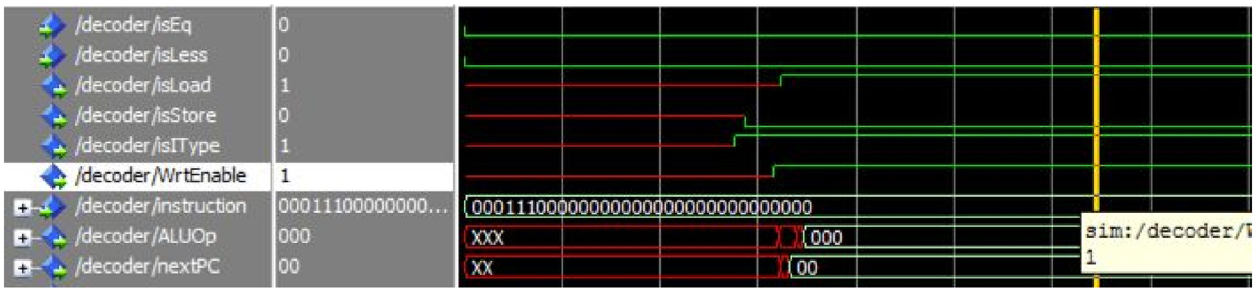
```
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       1
/decoder/WrtEnable     1
/decoder/instruction   00011000000000...   00011000000000000000000000000000
/decoder/ALUOp         110                 XXX   110
/decoder/nextPC        00                  XX    00
```

Fig. 36 SHR Timing Simulation



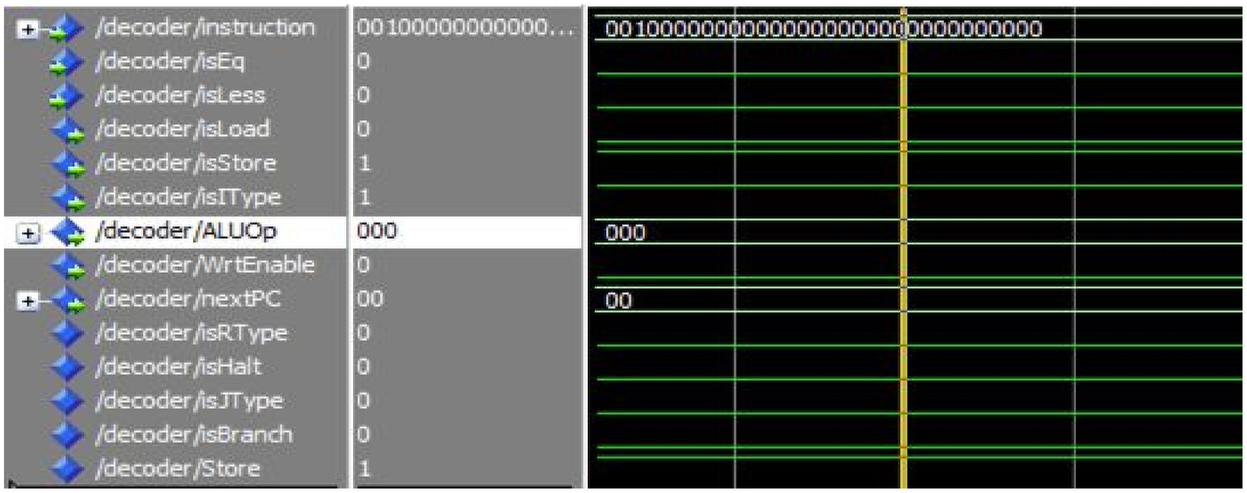
```
/decoder/instruction   00011100000000...   000111000000000000000000000000
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        1
/decoder/isStore       0
/decoder/isIType       1
/decoder/ALUOp         000                 000
/decoder/WrtEnable     1
/decoder/nextPC        00                  00
/decoder/isRType       0
/decoder/isHalt        0
/decoder/isJType       0
/decoder/isBranch      0
/decoder/Store         0
```

Fig. 37 LW Functional Simulation



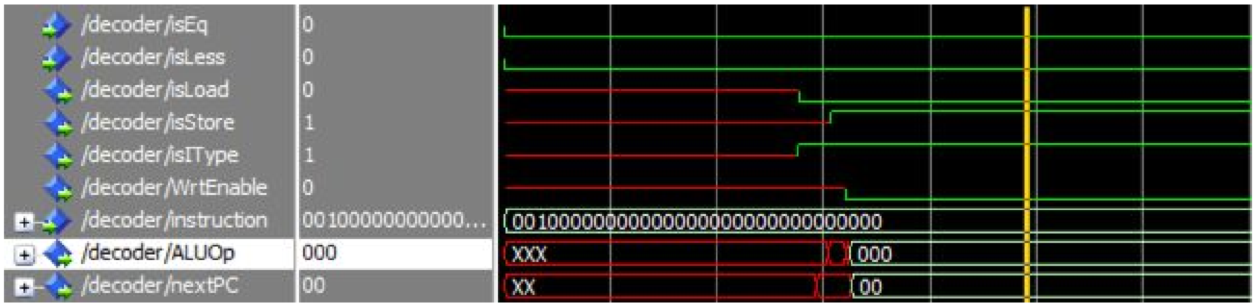
```
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        1
/decoder/isStore       0
/decoder/isIType       1
/decoder/WrtEnable     1
/decoder/instruction   00011100000000...   00011100000000000000000000000000
/decoder/ALUOp         000                 XXX  000
/decoder/nextPC        00                  XX   00
```

Fig. 38 LW Timing Simulation



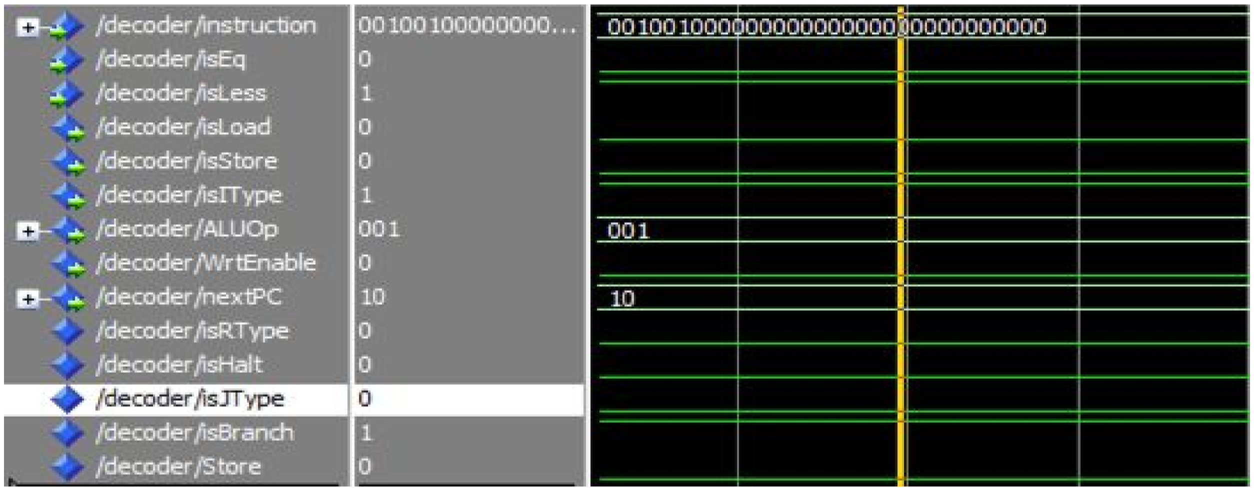
```
/decoder/instruction   00100000000000...   00100000000000000000000000000000
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       1
/decoder/isIType       1
/decoder/ALUOp         000                 000
/decoder/WrtEnable     0
/decoder/nextPC        00                  00
/decoder/isRType       0
/decoder/isHalt        0
/decoder/isJType       0
/decoder/isBranch      0
/decoder/Store         1
```

Fig. 39 SW Functional Simulation



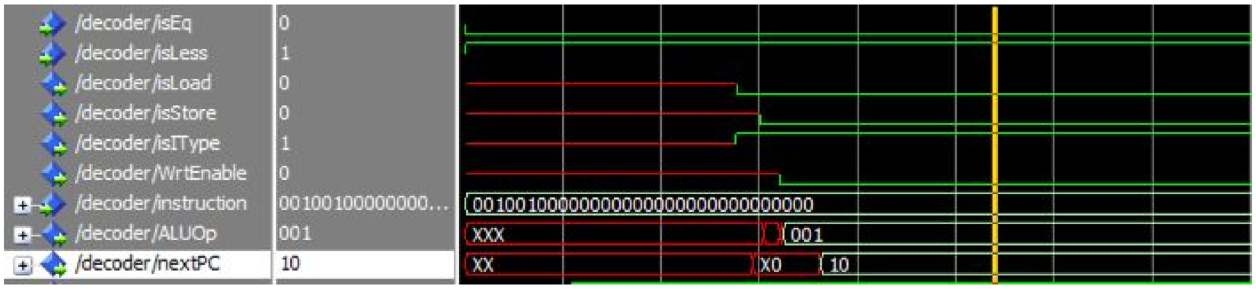
```
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       1
/decoder/isIType       1
/decoder/WrtEnable     0
/decoder/instruction   00100000000000...   00100000000000000000000000000000
/decoder/ALUOp         000                 XXX  000
/decoder/nextPC        00                  XX   00
```

Fig. 40 SW Timing Simulation



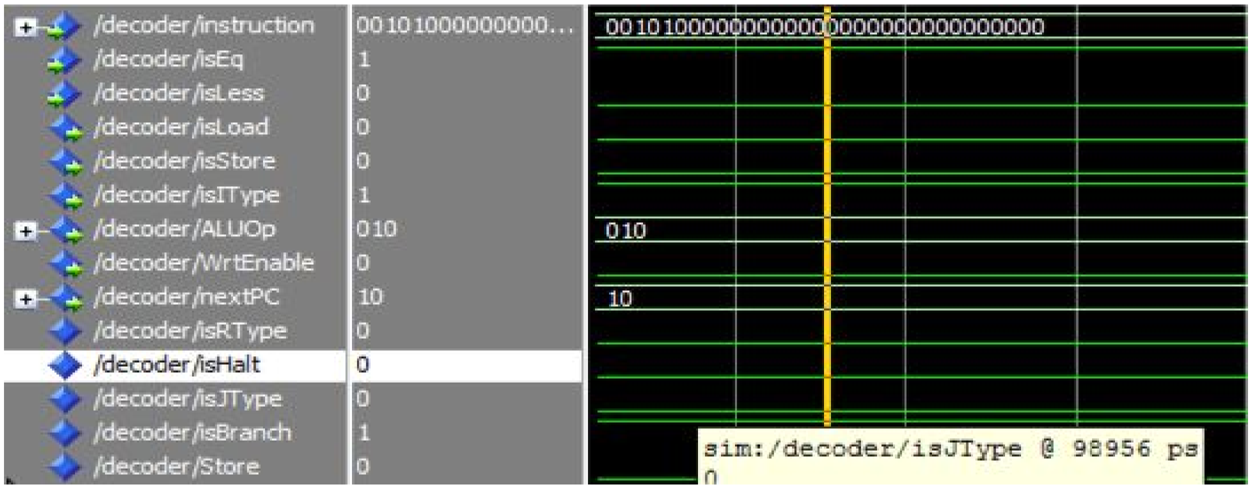
```
/decoder/instruction   00100100000000...   00100100000000000000000000000000
/decoder/isEq          0
/decoder/isLess        1
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       1
/decoder/ALUOp         001                 001
/decoder/WrtEnable     0
/decoder/nextPC        10                  10
/decoder/isRType       0
/decoder/isHalt        0
/decoder/isJType       0
/decoder/isBranch      1
/decoder/Store         0
```

Fig. 41 BLT Functional Simulation



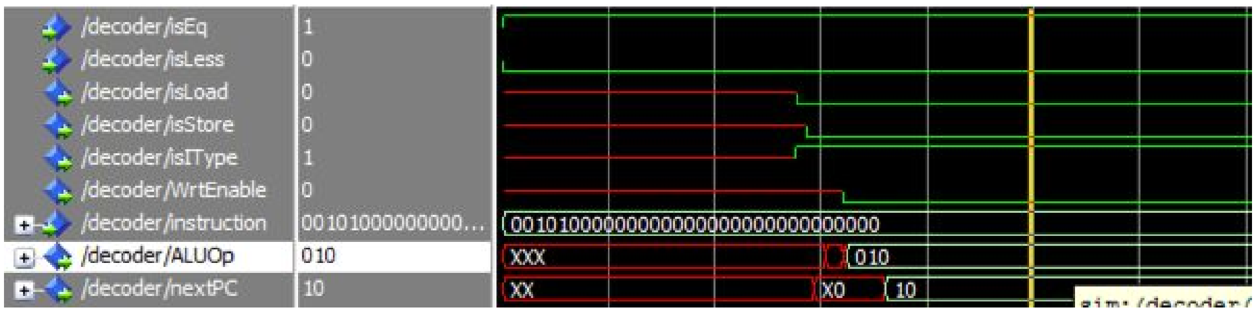
```
/decoder/isEq          0
/decoder/isLess        1
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       1
/decoder/WrtEnable     0
/decoder/instruction   00100100000000...   00100100000000000000000000000000
/decoder/ALUOp         001                 XXX  001
/decoder/nextPC        10                  XX  X0  10
```

Fig. 41 BLT Timing Simulation



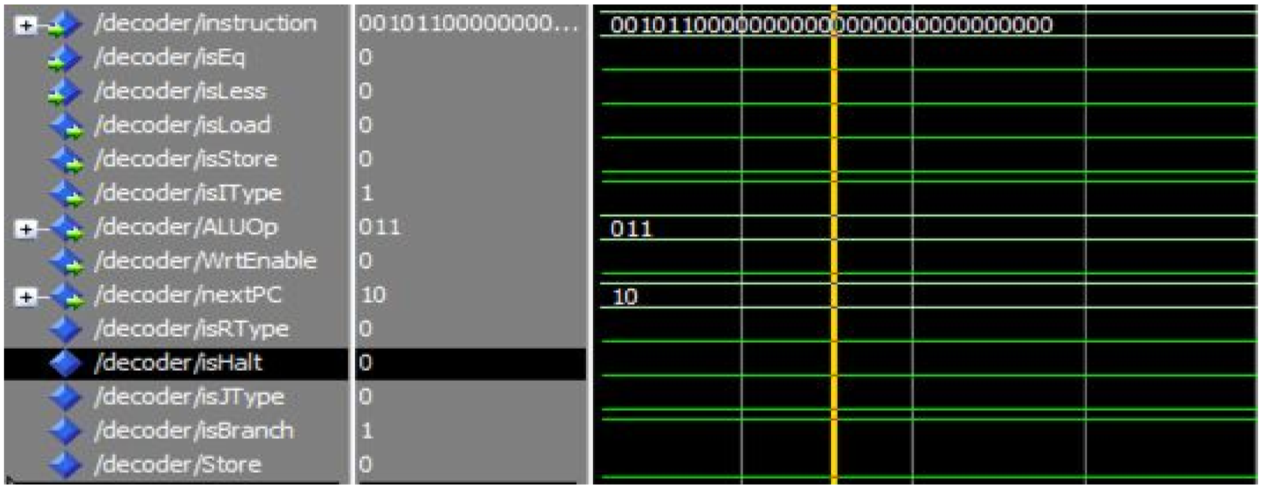
```
/decoder/instruction   00101000000000...   00101000000000000000000000000000
/decoder/isEq          1
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       1
/decoder/ALUOp         010                 010
/decoder/WrtEnable     0
/decoder/nextPC        10                  10
/decoder/isRType       0
/decoder/isHalt        0
/decoder/isJType       0
/decoder/isBranch      1
/decoder/Store         0
sim:/decoder/isJType @ 98956 ps
0
```

Fig. 42 BEQ Functional Simulation



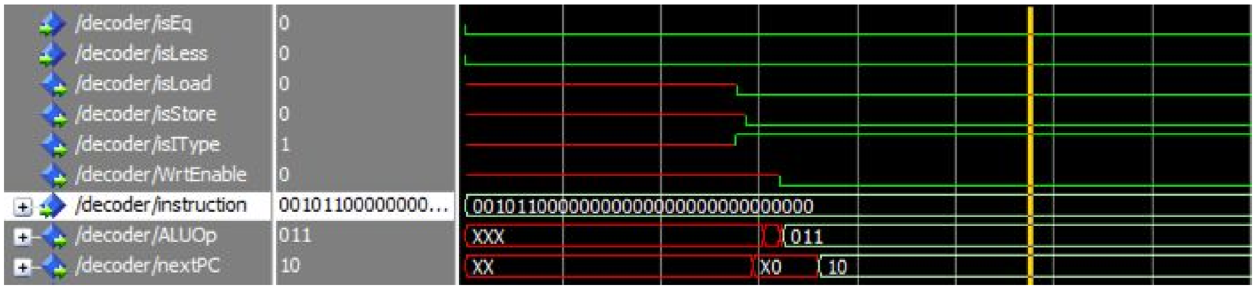
```
/decoder/isEq         1
/decoder/isLess       0
/decoder/isLoad       0
/decoder/isStore      0
/decoder/isIType      1
/decoder/WrtEnable    0
/decoder/instruction  00101000000000...   00101000000000000000000000000000
/decoder/ALUOp        010                 XXX   010
/decoder/nextPC       10                  XX   X0   10
```

Fig. 43 BEQ Timing Simulation



```
/decoder/instruction  00101100000000...   00101100000000000000000000000000
/decoder/isEq         0
/decoder/isLess       0
/decoder/isLoad       0
/decoder/isStore      0
/decoder/isIType      1
/decoder/ALUOp        011                 011
/decoder/WrtEnable    0
/decoder/nextPC       10                  10
/decoder/isRType      0
/decoder/isHalt       0
/decoder/isJType      0
/decoder/isBranch     1
/decoder/Store        0
```

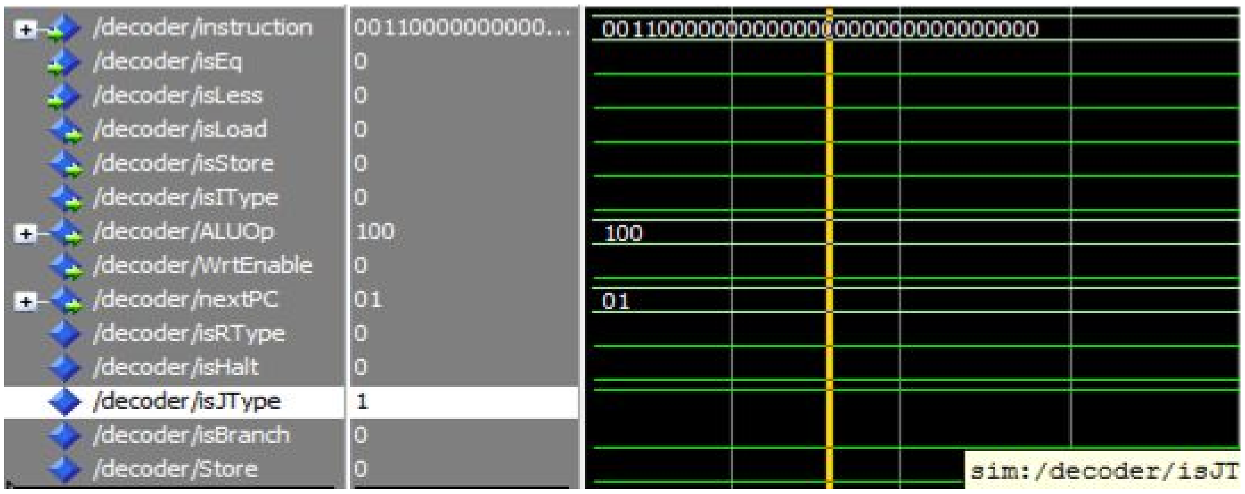
Fig. 44 BNE Functional Simulation



```
/decoder/isEq         0
/decoder/isLess       0
/decoder/isLoad       0
/decoder/isStore      0
/decoder/isIType      1
/decoder/WrtEnable    0
/decoder/instruction  00101100000000...   00101100000000000000000000000000
/decoder/ALUOp        011                 XXX   011
/decoder/nextPC       10                  XX   X0   10
```

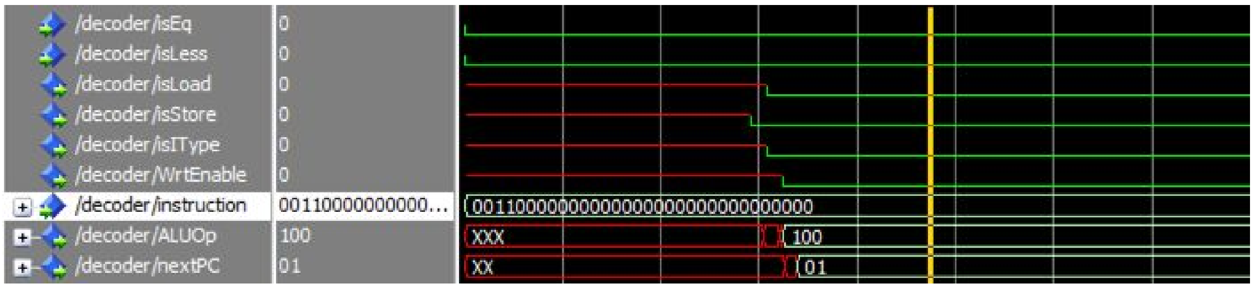
Fig. 45 BNE Timing Simulation

3. Functional and Timing Simulations for J-TYPE instructions



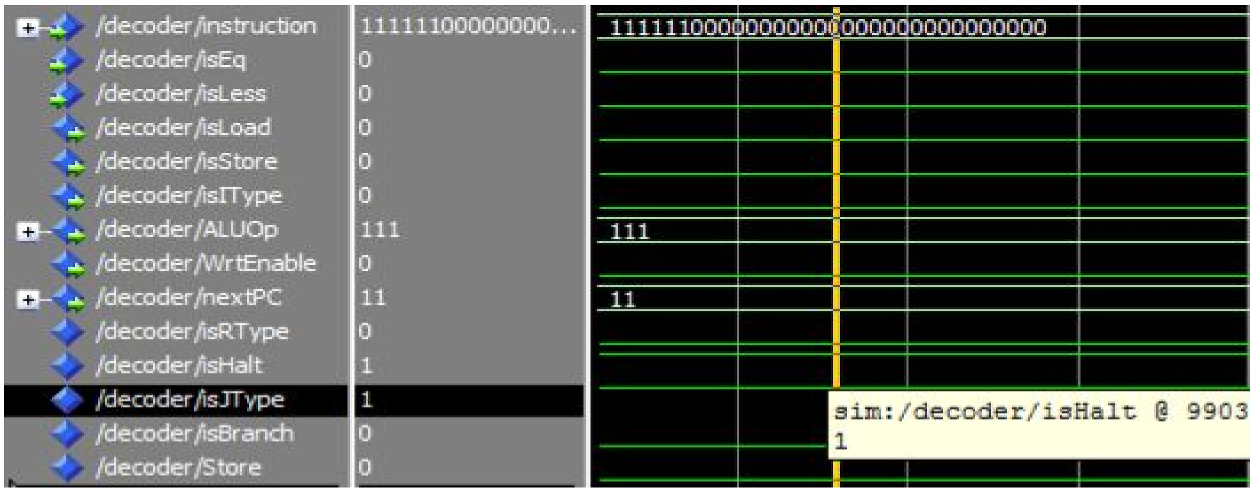
```
/decoder/instruction   00110000000000...   00110000000000000000000000000000
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       0
/decoder/ALUOp         100                 100
/decoder/WrtEnable     0
/decoder/nextPC        01                  01
/decoder/isRType       0
/decoder/isHalt        0
/decoder/isJType       1
/decoder/isBranch      0
/decoder/Store         0
sim:/decoder/isJT
```

Fig. 46 JMP Functional Simulation



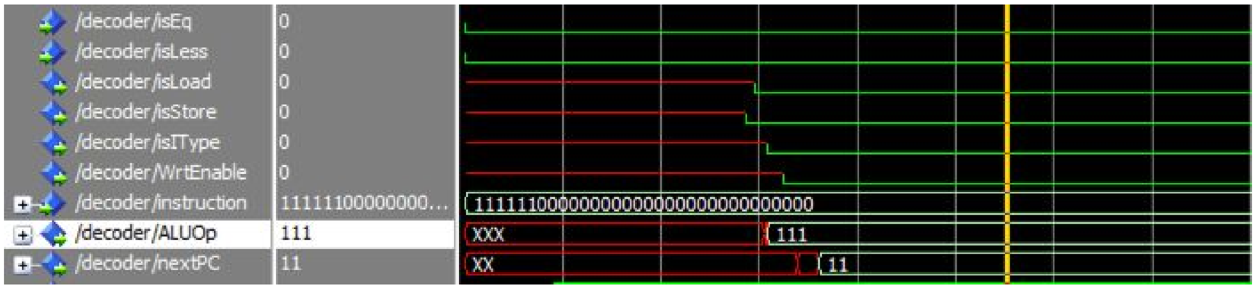
```
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       0
/decoder/WrtEnable     0
/decoder/instruction   00110000000000...   00110000000000000000000000000000
/decoder/ALUOp         100                 XXX   100
/decoder/nextPC        01                  XX    01
```

Fig. 47 JMP Timing Simulation



```
/decoder/instruction   11111100000000...   11111100000000000000000000000000
/decoder/isEq          0
/decoder/isLess        0
/decoder/isLoad        0
/decoder/isStore       0
/decoder/isIType       0
/decoder/ALUOp         111                 111
/decoder/WrtEnable     0
/decoder/nextPC        11                  11
/decoder/isRType       0
/decoder/isHalt        1
/decoder/isJType       1
/decoder/isBranch      0
/decoder/Store         0
sim:/decoder/isHalt @ 9903
1
```

Fig 48. HLT Functional Simulation



```
/decoder/isEq         0
/decoder/isLess       0
/decoder/isLoad       0
/decoder/isStore      0
/decoder/isIType      0
/decoder/WrtEnable    0
/decoder/instruction  11111100000000...   11111100000000000000000000000000
/decoder/ALUOp        111                 XXX  111
/decoder/nextPC       11                  XX  11
```

Fig. 49 HLT Timing Simulation

IV. Design Verification

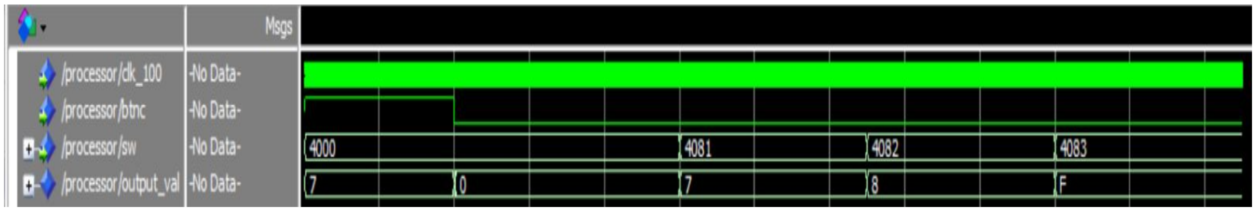
The first part of the verification was to make sure that every component was doing what it was supposed to do. To this end, every component was tested individually. Above are the screenshots for the ALU and the decoder. After that, the complete processor was put together and the following programs were tested to check that every instruction is working properly.

Sample program 1

```
 ADDI R1, R0, 7
 ADDI R2, R0, 8
ADD R3, R1, R2
 HAL
```

The above assembly code should result in the following output.
R1 = 7;
R2 = 8;
R3 = 15;

When executing the above code in our processor we get the following output



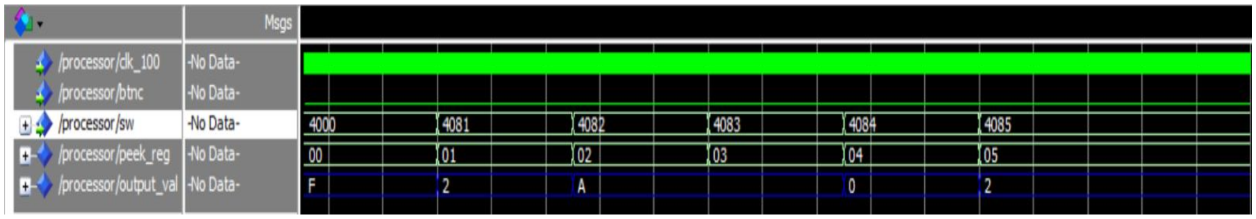
As we can see from the screenshot.
R1 = 7;
R2 = 8;
R3 = 15;
Hence output is verified.

Sample Program 2.

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 22
HAL

The above assembly code results in the following output
R1 = 2
R2 = A (10 in Decimal)
R3 = A (10 in Decimal)
R4 = 0
R5 = 2

When executing the above code in our processor we get the following output


```
                        Msgs
/processor/clk_100      -No Data-
/processor/btnc         -No Data-
/processor/sw           -No Data-   4000   4081   4082   4083   4084   4085
/processor/peek_reg     -No Data-   00     01     02     03     04     05
/processor/output_val   -No Data-   F      2      A             0      2
```

As we can see from the above screenshot , the following are the outputs of the registers-

R1 = 2
R2 = A (10 in Decimal)
R3 = A (10 in Decimal)
R4 = 0
R5 = 2

Hence output is verified.

V. Performance and Area Analysis

According to the synthesis report generating post synthesis, the device utilization of the process was

Post Synthesis:

Slice Logic Utilization:		
Number of Slice Registers:	5191 out of 126800	4%
Number of Slice LUTs:	4489 out of 63400	7%
Slice Logic Distribution:		
Number of LUT Flip Flop pairs used:	8046	
Number with an unused Flip Flop:	2855 out of 8046	35%
Number with an unused LUT:	3557 out of 8046	44%
Number of fully used LUT-FF pairs:	1634 out of 8046	20%
Number of unique control sets:	166	
IO Utilization:		
Number of IOs:	50	
Number of bonded IOBs:	42 out of 210	20%

Post Place and Route:

Slice Logic Utilization:		
Number of Slice Registers:	5,191 out of 126,800	4%
Number of Slice LUTs:	4,446 out of 63,400	7%
Slice Logic Distribution:		
Number of occupied Slices:	2,576 out of 15,850	16%
Number of LUT Flip Flop pairs used:	7,488	
Number with an unused Flip Flop:	2,297 out of 7,488	30%
Number with an unused LUT:	3,042 out of 7,488	40%
Number of fully used LUT-FF pairs:	2,149 out of 7,488	28%
Number of slice register sites lost to control set restrictions:	0 out of 126,800	0%
IO Utilization:		
Number of bonded IOBs:	42 out of 210	20%

An analysis of the above utilization shows that the total number of Slice Registers and IOs used in the design remain the same but the total number of Slice LUTs and Slice LUT Flip Flop pairs utilized has

decreased post Place and Route. The reason for this decrease is the logic optimization for the design that takes place during Place and Route. We can also observe from the PAR report that our design is using 16% of the total available slices on the FPGA Board.
We have 42 IOBs: 1 for the clock, 1 for the center button, 12 for the switches, 12 for the leds and 16 for the 7-segment display.

From the Timing summary generated in the Post Synthesis report we have
Minimum Period : 10.961ns
Maximum Frequency : 91.231MHz
Maximum Combinational Path Delay : 6.788ns

From the Timing summary generated in the Post PAR report we have:
Minimum Period: 17.459ns
Maximum Frequency : 57.277MHz

From these results, the maximum speed of operation for our design came out to be 57.277MHz.

There are two latencies to the circuit depending on what code is in the IM.
A 20 ns period clock was used in the following because it is the actual clock running on our processor on the FPGA.

For the code having the RC5 key generation and decryption we have:
Latency of the circuit: 5880 cycles
Total or Propagational Delay: 117.6 ms (117600 ns)

For the code having the RC5 key generation and encryption we have:
Latency of the circuit: 6055 cycles
Total or Propagational Delay: 121.1 ms (121100 ns)

VI. RC5 implemented on the processor

RC5 or “Rivest Cipher” is a symmetric key block cipher notable for its simplicity. It was designed by Ronald Rivest in 1994. The RC5 cipher designed by us has a block size of 64 bits, key size of 128 bits and operates for 78 rounds. The key feature of RC5 is the use of data dependent rotation.

The following is the algorithm for the key expansion.

```
“do 3*max(t, c); t=26, c=4
  A = S[i] = (S[i] + A + B) <<< 3;
  B = L[j] = (L[j] + A + B) <<< (A + B);
  i = (i + 1) mod (t);
  j = (j + 1) mod (c);”
```

To initialize S Array, we have

```
“S[0] = 0xB7E15163 (Pw)
for i=1 to 25 do  S[i] = S[i-1]+ 0x9E3779B9 (Qw)”
```

To initialize L Array , we have

```
“for i = b - 1 downto 0 do
    L[i/u] = (L[i/u] <<< 8) + K[i];”
```

The following is the algorithm for encryption.

```
“A = A + S[0];
B = B + S[1];
for i = 1 to 12 do
        A = ((A xor B) <<< B) + S[2×i];
        B = ((B xor A) <<< A) + S[2×i + 1];”
```

Where A and B are 32 bit halves of a 64 bit input.

Now we have the Algorithm for decryption

```
“for i = 12 downto 1 do
        B = ((B - S[2×i +1]) >>> A) xor A;
        A = ((A - S[2×i]) >>> B) xor B;
B = B - S[1];
A = A - S[0];”
```

Again , A and B are 32 bit halves of a 64 bit input.

To implement the complete RC5 on the processor multiple steps were taken. First of all, the algorithms above had to be translated to assembly to then be translated to machine code, that being binary and hexadecimal. Delivered with this report can be found the assembly codes as well as the machine codes for the RC5 (more details on the code in the last paragraph VIII). Before joining the three algorithms together they had to be tested each individually. The screenshots below show that process for the key expansion but the screenshots for the encryption and decryption are taken from the final complete code, but they show that these two functions work on the processor. After testing and making sure that the codes ran properly, they were all joined together in a single file. Of course some changes had to be made to ensure a proper functionality for the code. Instead of the halt instruction in the key expansion algorithm, a branch instruction, which could’ve also been a jump, was introduced to make the processor jump to a part of the code where some comparison were made to check if it needs to do the encryption of the plaintext or it’s decryption. A value is compared to 0 and if it is equal to 0 then the PC jumps to the part of the code where the decryption algorithm is, otherwise the PC increments and the processor begin the encryption. This value being compared is loaded from the memory, and the value inside that memory location is set by the user using the switch 9 as described above. If the switch is to 1 then the encryption will be done, otherwise the decryption is ran.
As seen above, the RC5, being the key generation, the encryption or decryption, needs a user input. For this project the user inputs were pre-initialized in the memory, so every time a value was needed by the algorithm a load instruction was used and every time a value had to be saved a store instruction was executed. All of the inputs are pre-initialized except for the one deciding on running the encryption or decryption. The values needed for the RC5 are the user key and the plaintext (A and B in the encryption and decryption algorithm) as user defined inputs, but other values have to be stored as well such as the l array and the Pw and Qw magic constants. The result is also stored in the DM, so the 26 values of the skey as well as the encrypted or decrypted plaintext. This being said, only 41 memory location were used for the RC5 on the processor and below is a table showing how that information is stored in the RAM, meaning what value occupies what location in the memory.

ukey(3)	ukey(2)	ukey(1)	ukey(0)	L[0]	L[1]	L[2]	L[3]	Pw	Qw
S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]	S[9]
S[10]	S[11]	S[12]	S[13]	S[14]	S[15]	S[16]	S[17]	S[18]	S[19]
S[20]	S[21]	S[22]	S[23]	S[24]	S[25]	A	B	A’	B’

Enc

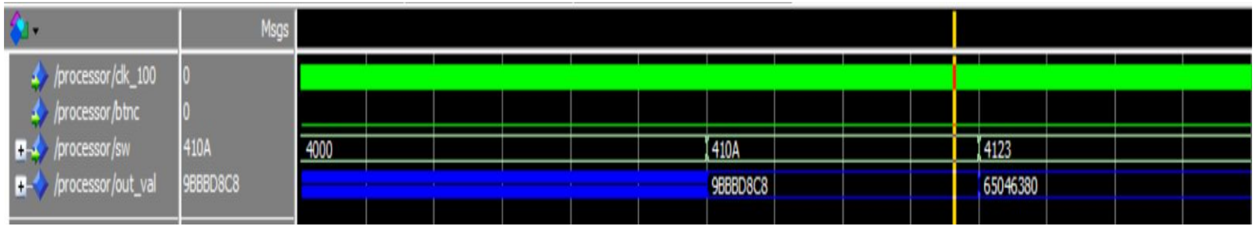
The locations are 0 in the top left corner and then increment by 1 as one goes to the right and down. This means that Enc is in memory location 40 and that is the value deciding on whether to encrypt or decrypt. And Qw is in location 9 and S[0] in location 10. Ukey is the user key usually inputted by the user, l array is the array used for the key expansion, S is the skey used in the encryption and decryption, A and B are the input plaintext and finally A’ and B’ are the output plaintext (encrypted or decrypted). So, to choose between encryption and decryption, the switch 9 is directly mapped to the location 40 of the memory, the DM has an actual input just for that value, and this value is overwritten at every rising edge of the clock. The user should know that he can change the value of that signal until it is loaded by the program, meaning right after the end of the key expansion. After that, the value will be overwritten but it will not matter because the value has already been read.

RC5 Key Generation testing

RC5 block cipher works on the principle of data dependent rotation. Our processor implements a RC5 block cipher which has a block size of 64 bits and key size of 128 bits.
A 26 by 32 bit SKEY RAM is generated using the RC5 key expansion algorithm.
Upon hand calculation we get the following values of the SKEY RAM .

SKEY = [X"9BBBD8C8", X"1A37F7FB", X"46F8E8C5", X"460C6085", X"70F83B8A", X"284B8303", X"513E1454", X"F621ED22", X"3125065D", X"11A83A5D", X"D427686B", X"713AD82D", X"4B792F99", X"2799A4DD", X"A7901C49", X"DEDE871A", X"36C03196", X"A7EFC249", X"61A78BB8", X"3B0A1D2B", X"4DBFCA76", X"AE162167", X"30D76B0A", X"43192304", X"F6CC1431", X"65046380"]

Implementation of the above RC5 key expansion algorithm on our processor resulted in the following output depicted in the screenshot below

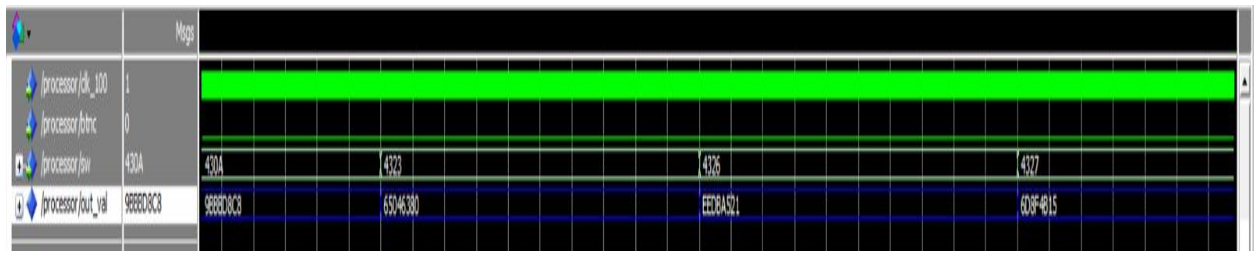


As we can see from the above screenshot that the value of the first key in SKEY ROM is “9BBBD8C8” and the last value is “65046380”.
Both these values match with the hand calculated values of the SKEY RAM.
Hence our processor supports RC5 Key Generation.

RC5 Encryption testing

Using the key generated from the RC5 key generation algorithm, We can encrypt a user given 64 input and output the encrypted value.
Assuming the inputs given by user are (in hexadecimal)
A - X“00000000”
B - X“00000000”
Using the above algorithm, the result we obtain through hand calculation is as follows
A = X“EEDBA521”
B = X“6D8F4B15”

Now,
Executing the above algorithm with Key Expansion in our processor with the inputs
A= X“00000000”
B= X“00000000”
we get the following output

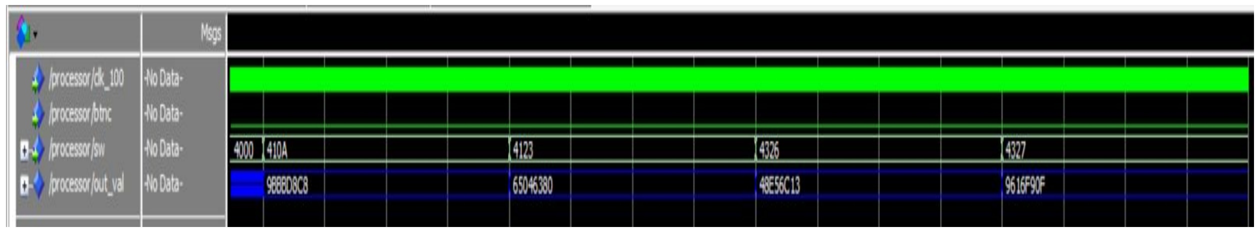


As we can see from the above screenshot, the processor first runs the key expansion algorithm and after it has created the SKEY RAM, it executes the encryption process.
The following is the encrypted output for the given inputs.
A = X“EEDBA521”
B = X“6D8F4B15”
Therefore , we have successfully executed key expansion and encryption in our processor.

RC5 Decryption testing

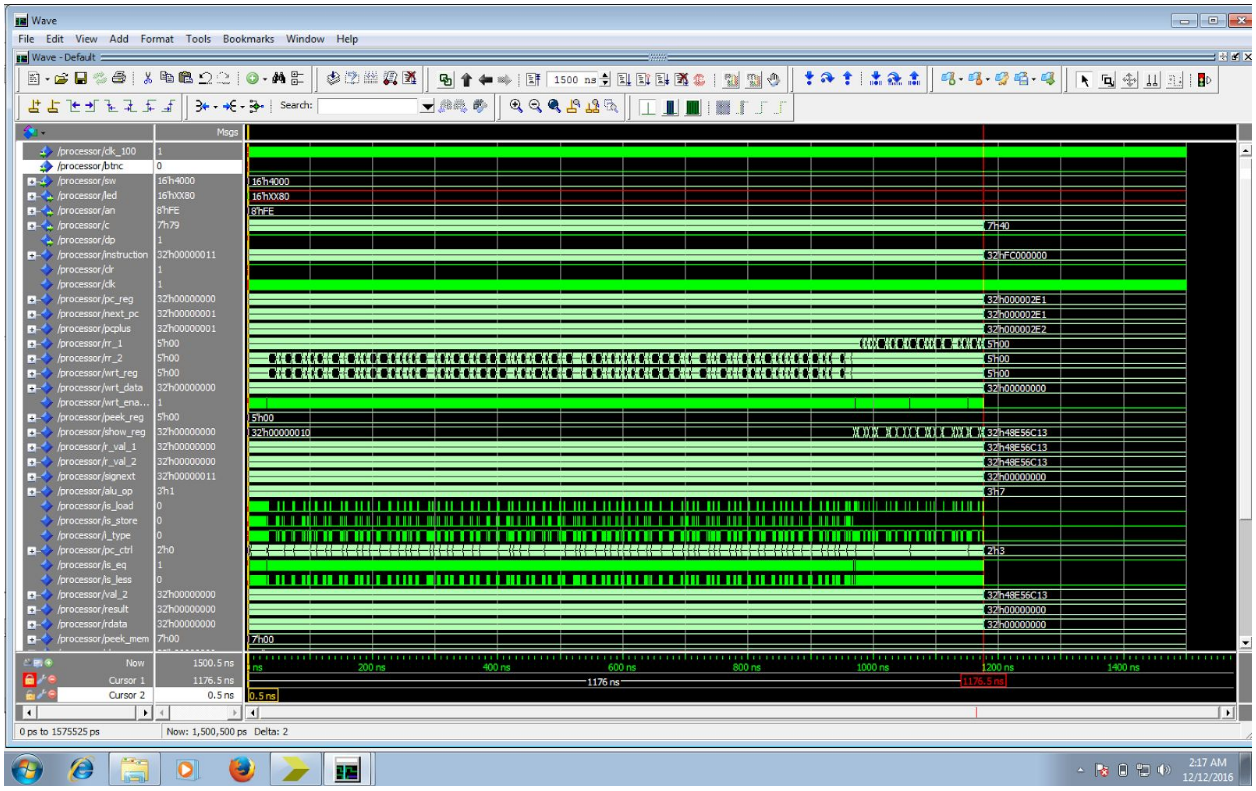
Using the key generated from the RC5 key generation algorithm, We can decrypt a user given 64 input and output the decrypted value.
Assuming the inputs given by user are
A - X“00000000”
B - X“00000000”
Using the above algorithm , the result we obtain through hand calculation is as follows
A = X“48E56C13”
B = X“9616F90F”

Now,
Executing the above algorithm with Key Expansion in our processor with the inputs
A= X“00000000”
B= X“00000000”
we get the following output


Msgs
/processor/clk_100 -No Data-
/processor/btnc -No Data-
/processor/sw -No Data-
/processor/out_val -No Data-
4000
410A
4123
4326
4327
9BBBD8C8
65046380
48E56C13
9616F90F

As we can see from the above screenshot, the processor first runs the key expansion algorithm and after it has created the SKEY RAM, it executes the decryption process.
The following is the encrypted output for the given inputs.
A = X“48E56C13”
B = X“9616F90F”
Therefore , we have successfully executed key expansion and decryption in our processor.

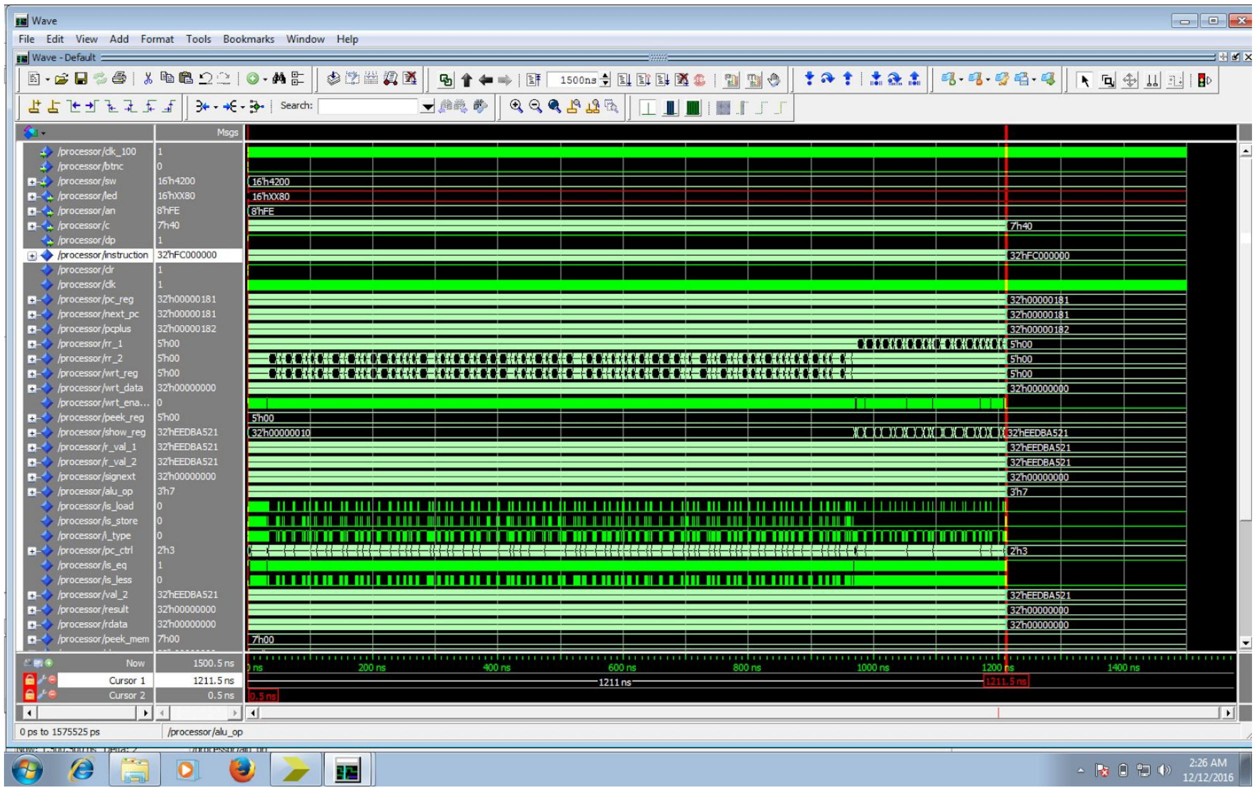
The processor needs 5880 cycles to run the key generation and the decryption. As seen in the following screenshot it takes 1176ns to complete the RC5 with a 200ps clock.



	Msgs
/processor/clk_100	1
/processor/btnc	0
/processor/sw	16'h4000
/processor/led	16'hXX80
/processor/an	8'hFE
/processor/c	7'h79
/processor/dp	1
/processor/instruction	32'h00000011
/processor/clr	1
/processor/clk	1
/processor/pc_reg	32'h00000000
/processor/next_pc	32'h00000001
/processor/pcplus	32'h00000001
/processor/rr_1	5'h00
/processor/rr_2	5'h00
/processor/wrt_reg	5'h00
/processor/wrt_data	32'h00000000
/processor/wrt_ena...	1
/processor/peek_reg	5'h00
/processor/show_reg	32'h00000000
/processor/r_val_1	32'h00000000
/processor/r_val_2	32'h00000000
/processor/signext	32'h00000011
/processor/alu_op	3'h1
/processor/is_load	0
/processor/is_store	0
/processor/i_type	0
/processor/pc_ctrl	2'h0
/processor/is_eq	1
/processor/is_less	0
/processor/val_2	32'h00000000
/processor/result	32'h00000000
/processor/rdata	32'h00000000
/processor/peek_mem	7'h00
Now	1500.5 ns
Cursor 1	1176.5 ns
Cursor 2	0.5 ns

0 ps to 1575525 ps   Now: 1,500,500 ps Delta: 2

Similarly, it take 6055 cycles to run the key generation and the encryption. As seen in the following screenshot it takes 1211ns to complete the RC5 with a 200ps clock.



	Msgs
/processor/clk_100	1
/processor/btnc	0
/processor/sw	16'h4200
/processor/led	16'hXX80
/processor/an	8'hFE
/processor/c	7'h40
/processor/dp	1
/processor/instruction	32'hFC000000
/processor/clr	1
/processor/clk	1
/processor/pc_reg	32'h00000181
/processor/next_pc	32'h00000181
/processor/pcplus	32'h00000182
/processor/rr_1	5'h00
/processor/rr_2	5'h00
/processor/wrt_reg	5'h00
/processor/wrt_data	32'h00000000
/processor/wrt_ena...	0
/processor/peek_reg	5'h00
/processor/show_reg	32'hEEDBA521
/processor/r_val_1	32'hEEDBA521
/processor/r_val_2	32'hEEDBA521
/processor/signext	32'h00000000
/processor/alu_op	3'h7
/processor/is_load	0
/processor/is_store	0
/processor/i_type	0
/processor/pc_ctrl	2'h3
/processor/is_eq	1
/processor/is_less	0
/processor/val_2	32'hEEDBA521
/processor/result	32'h00000000
/processor/rdata	32'h00000000
/processor/peek_mem	7'h00
Now	1500.5 ns
Cursor 1	1211.5 ns
Cursor 2	0.5 ns

0 ps to 1575525 ps   /processor/alu_op

VII. Test benches

To verify the processor even more, test benches were run. The test benches were testing the RC5 function. This will test the processor itself as well as the assembly code written. Along this report is the modified version of the VHDL code that supports test benches. To this end, the top level had to be modified to take in inputs of 128 bits for the user key and 64 bits for the plaintext as well as outputting a vector of 64 bits for the modified plaintext. Also, since the A, B and user key are stored in the memory and the result is given by it, this component had to be modified to be able to store values coming from the input file of the test bench (test.txt) and output the result to the output file (output.txt).
The input file has on a single line the plaintext first, followed by the user key, followed by the encryption bit and finally followed by the expected result.
The output file has both the result from the processor (dout) and the expected result (output) one above the other starting with the processor's result.
Alongside the code and this report is the test.txt file tested and its result file output.txt.
The expected results were calculated using a golden model, the C code for the RC5 given by the Professor. A text file is also attached to this report showing the result of that golden model. However, the key in these files is written backwards, meaning the first value is the LSB and not the MSB and the last value is the MSB not the LSB.

VIII. Links

Here is the link to the video showing and explaining the processor running the RC5 code on FPGA: https://www.youtube.com/watch?v=MQTYQ6v2TCM&feature=youtu.be

Here is the link to the github repository used by the group: https://github.com/plaudis/NYU6463_Processor.git

IX. Codes

The assembly and machine codes for the RC5 have been delivered alongside this report. There is a folder for each part of the RC5 as well as the complete RC5. In each folder, except the complete RC5 one, there are 4 files. One called "XXXASS", which contains the assembly code with comments. Unfortunately, these codes can be wrong and so are not very accurate. However, the file named "XXXASS2" has the assembly code without any comments and in the format needed for the C code we created that converts assembly code to machine code and that code is right. In fact it is the one used in all the screenshots above and in the video. The file "XXXBIN" is the binary of the code and the "XXXHEX" file is the code in hexadecimal and it's the one that was saved in the instruction memory in VHDL. The only file missing the complete RC5 folder is the "XXXASS".
If one wants to use the C code to convert assembly code into machine code, one should follow the exact syntax as in the delivered files and one should insert a space after the halt instruction before executing the code. Otherwise, errors will occur.