PROJECT

REPORT

NEW YORK UNIVERSITY

TANDON SCHOOL OF

ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Submitted By:

Tianyu Gu tg1553

Fengyang Jiang fj483

Yingqi Huang yh1990

Junlun Xiao jx755

Yiren Dai yd1257

Lin Lu ll3374

1. Introduction

The main objective of this project is to design a single cycle 32-bit MIPS (Microprocessor without Interlocked Pipeline Stages) RISC (Reduced Instruction Set Computer) processor using VHDL (Very high speed integrated circuit Hardware Description Language), implementing it on FPGA (Field Programmable Gate Array). This 32-bit processor supports 3 types of instructions, R-Type for arithmetic instructions, I-Type for immediate value operations and load and store instructions, J-Type for jump instructions. To show whether it works properly for these instructions, we wrote a RC5 assembly code using the instructions it supports, and converted the assembly code into machine code (Byte Code) and ran it on FPGA.

2. NYU-6463 Processor

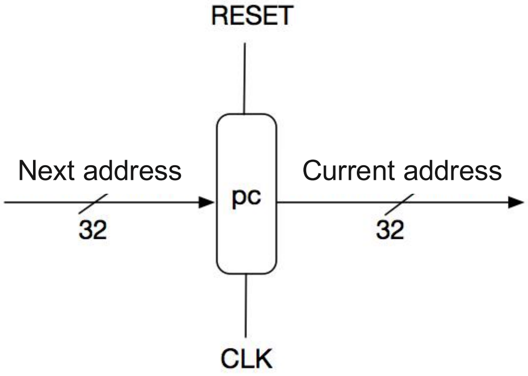
2.1 Processor Components

The NYU-6463 Processor performs the tasks of instruction fetch, instruction decode, execution and memory access all in one clock cycle. First, the PC value is used as an address to index the instruction memory which supplies a 32-bit value of the next instruction to be executed. This instruction is then split into the different fields as shown in Table above. The instructions' opcode field bits [31-26] are sent to the control unit to determine the type of instruction to execute. The type of instruction then determines which control signals are to be asserted and what function the ALU is to perform, therefore, decoding the instruction. The instruction register address fields Rs bits [25 - 21], Rt bits [20 - 16], and Rd bits [15-11] are used to address the register file. The register file reads in the requested addresses and outputs the data values contained in these registers. These data values can then be operated on by the ALU whose operation is determined by the control unit to either compute a memory address (e.g. load or store), compute an arithmetic result (e.g. add, and or sub), or perform a compare (e.g. branch operations). If the instruction decoded is arithmetic, the ALU result is0 written to a register. If the instruction decoded is a load or a store, the ALU result is then used to address the data memory. The final step writes the ALU result or memory value back to the register file.

2.1.1 Program counter (PC) register

2.1.1.1 Brief Introduction

This is a 32-bit register with a clock and a synchronous reset acting on it. The output of PC directly connects to the Instruction Memory, and it also connects to an adder to implement the instructions of Branch and Jump with other signals. The PC Register will finally obtain the address of an instruction for the instruction memory using the current value of PC and increment its value for the next instruction.

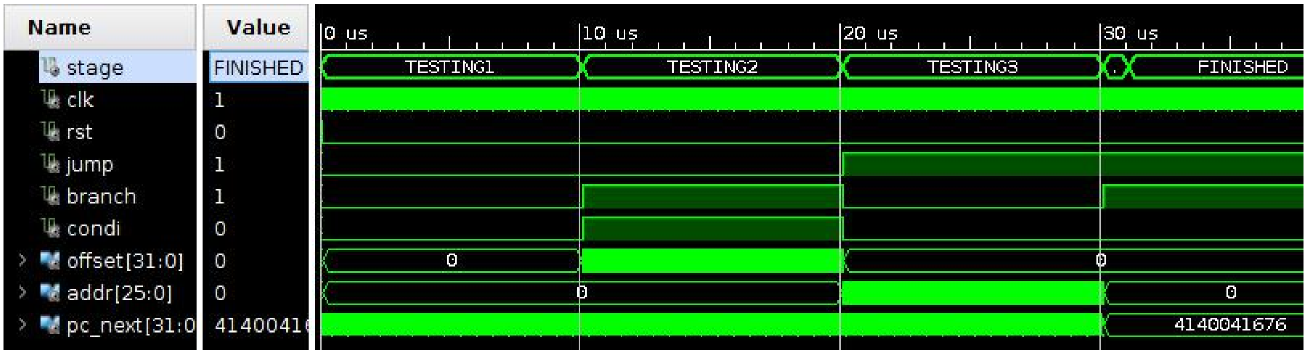


PC has 4 different sources:

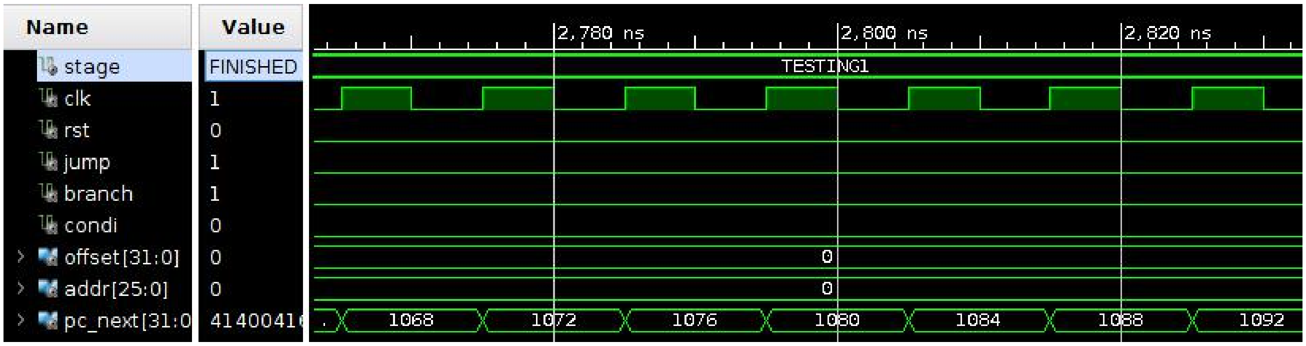
1. Continuously increasing: nextPC = PC + 4
2. Branch instruction: nextPC = PC + 4 + offset*4
3. Jump instruction: nextPC = (PC+4)[31:28] & addr & “00”
4. Halt instruction: nextPC = PC

For each situation, we tested the PC unit on 1000 random cases and used ‘assert’ statement to check the output automatically (see tb_pc.vhd for details). The stage signal would change to FINISH only when all test cases passed, otherwise, the simulation would stop with severity level ‘failure’.

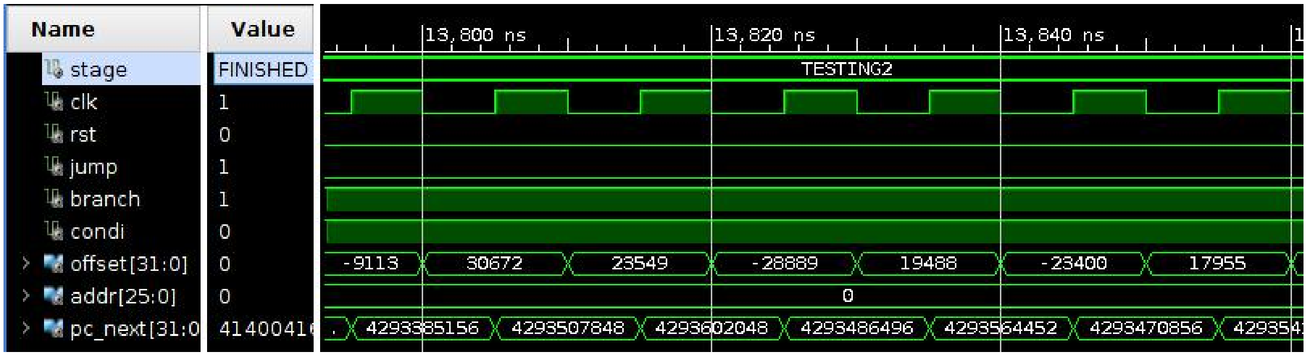
2.1.1.2 Functional simulation



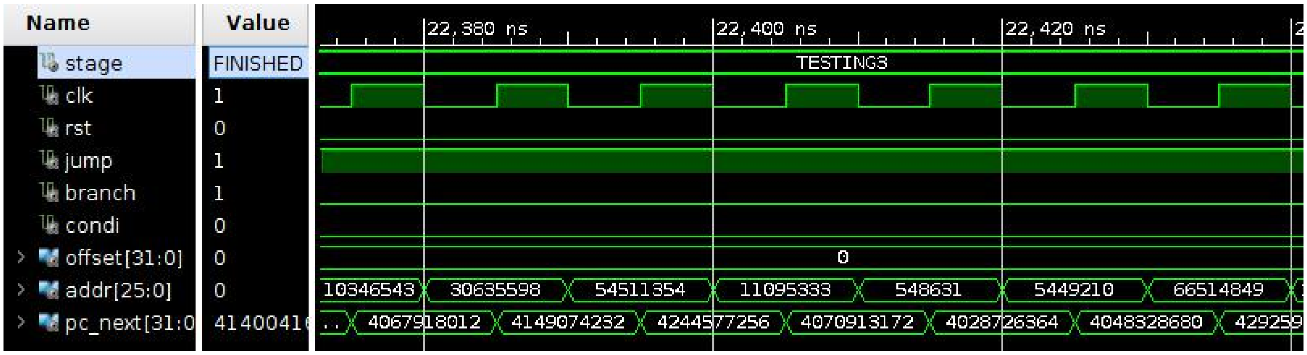
▲ Signal wave overview. All cases passed.



▲ Testing continuously increasing

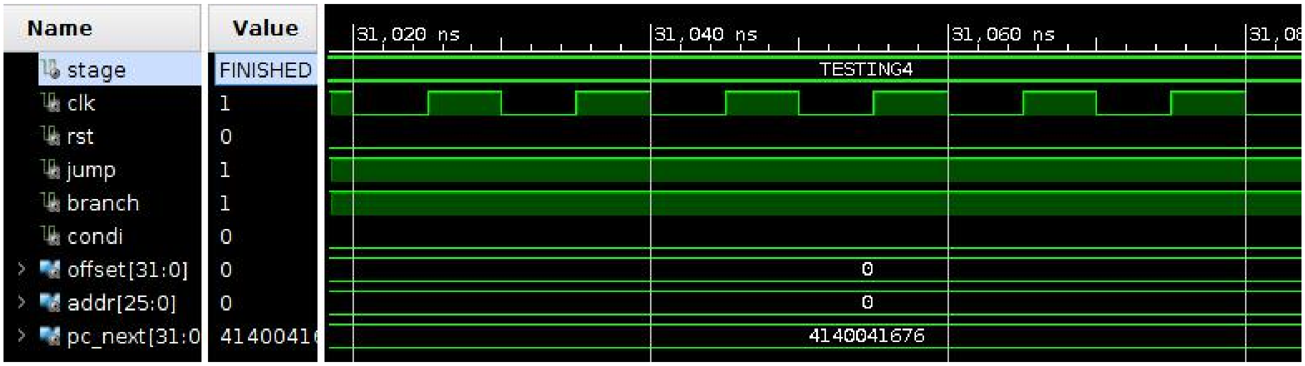


▲ Testing branch instruction



Name	Value
stage	FINISHED
clk	1
rst	0
jump	1
branch	1
condi	0
offset[31:0]	0
addr[25:0]	0
pc_next[31:0	41400416

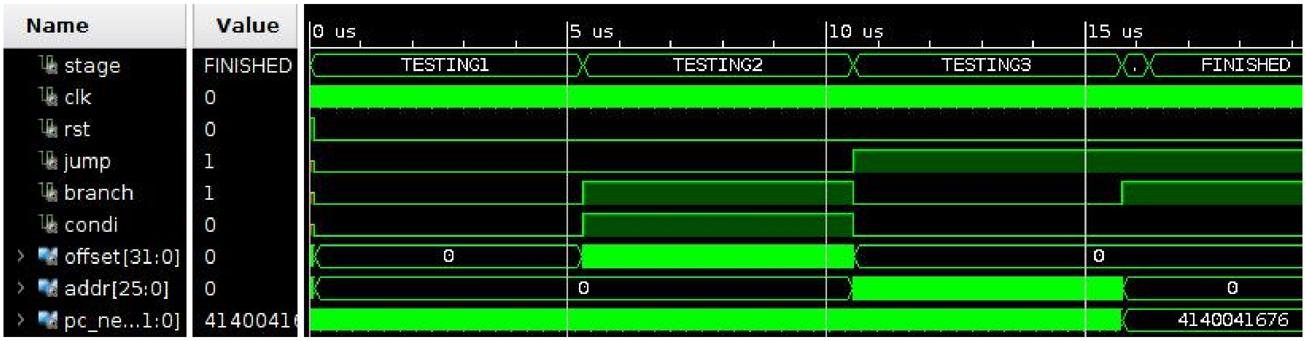
▲ Testing jump instruction



Name	Value
stage	FINISHED
clk	1
rst	0
jump	1
branch	1
condi	0
offset[31:0]	0
addr[25:0]	0
pc_next[31:0	41400416

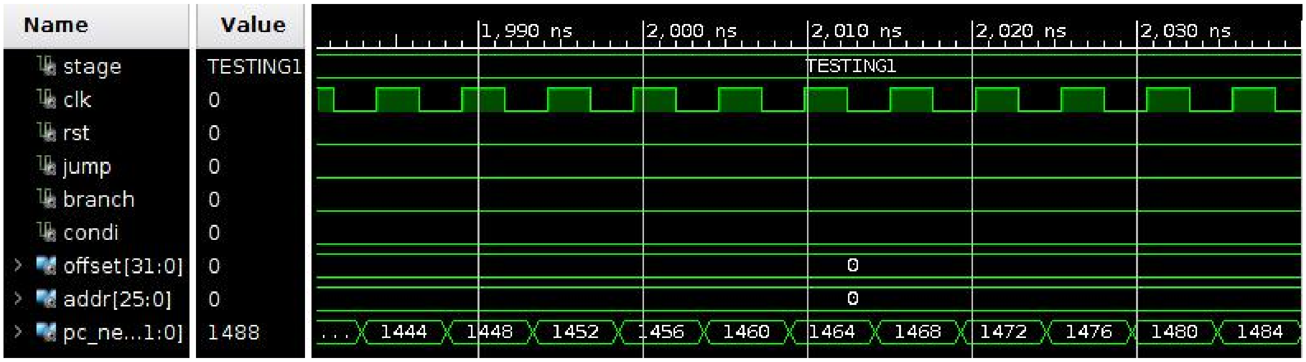
▲ Testing halt instruction

2.1.1.3 Timing simulation

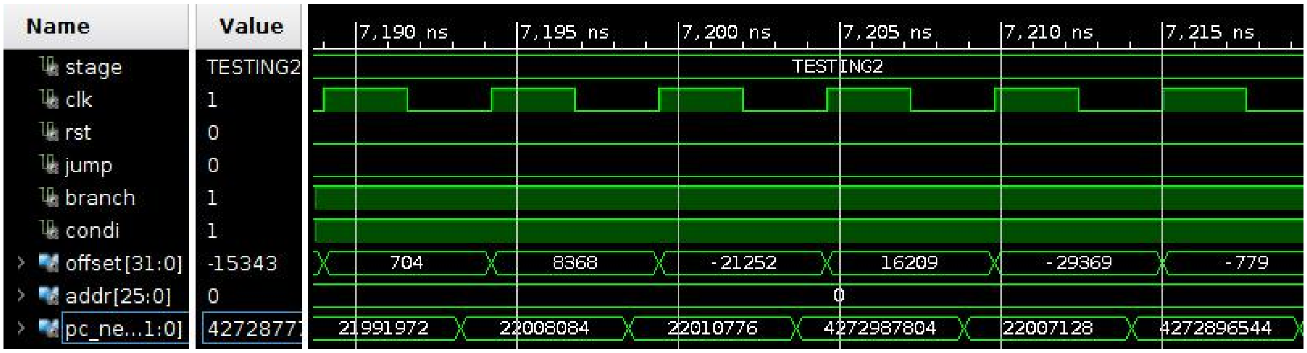


Name	Value
stage	FINISHED
clk	0
rst	0
jump	1
branch	1
condi	0
offset[31:0]	0
addr[25:0]	0
pc_ne...1:0]	41400416

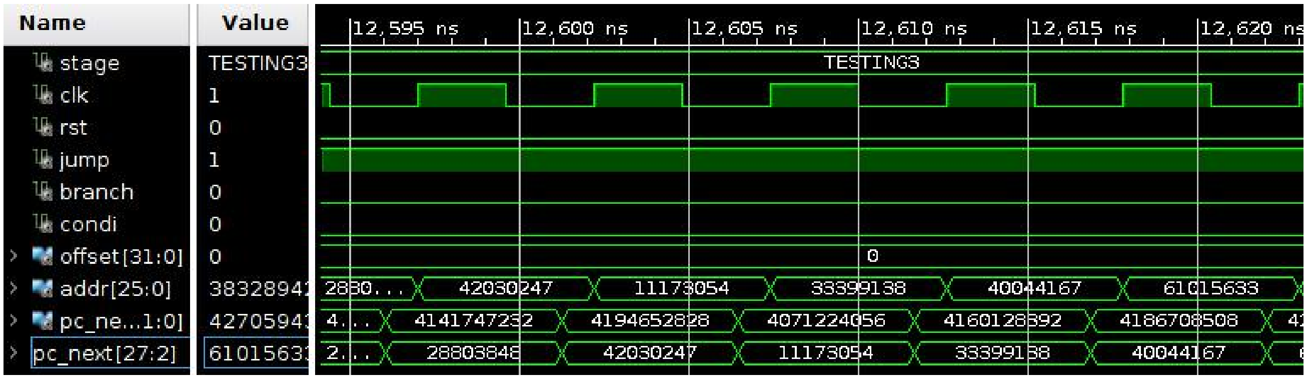
▲ Signal wave overview. All cases passed.



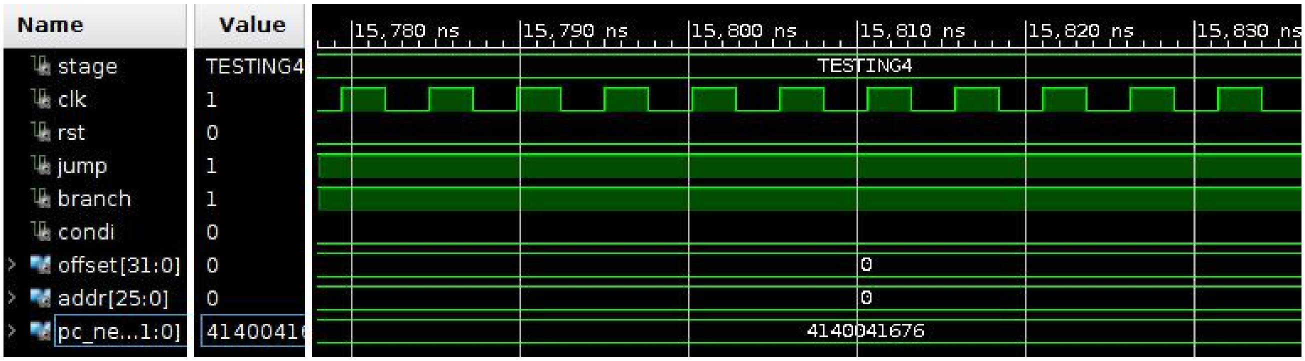
▲ Testing continuously increasing



▲ Testing branch instruction



▲ Testing jump instruction

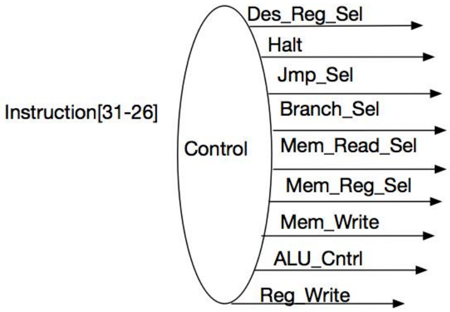


▲ Testing halt instruction

2.1.2 Control Unit

2.1.2.1 Brief Introduction

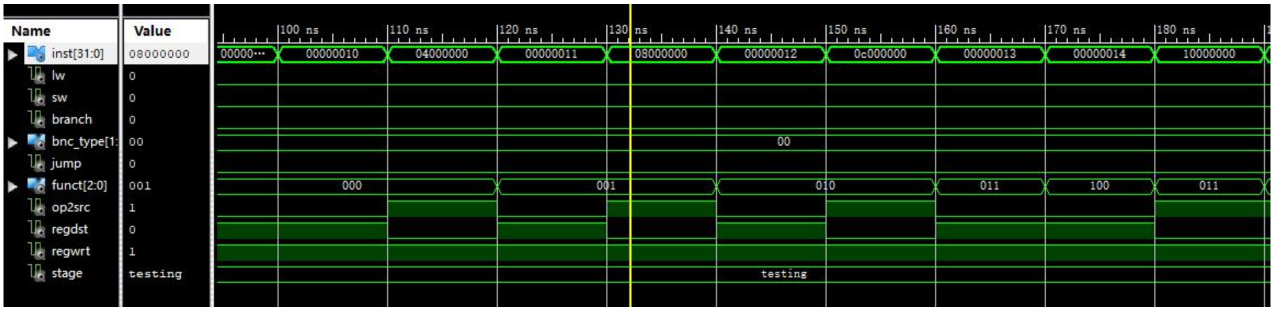
The control unit of the block diagram examines the instruction opcode bits [31 - 26] and decodes the instruction to generate control signals to be used in the additional modules. The Dst_Reg_Del determines which register will be selected to be written into the register file. The Jmp_Sel control signal selects the jump address to be sent to the PC. The Branch_Sel is used to select the branch address to be sent to the PC. The Mem_Read_sel control signal is asserted during a load instruction when the data memory is read to load a register with its memory contents. The Mem_Reg_select control signal determines if the ALU result or the data memory output is written to the register file. The MEM_Write control signal is asserted when during a store instruction when a registers value is stored in the data memory. The ALU_Cntrl control signal determines if the ALU second operand comes from the register file or the sign extend. The Reg_Write control signal is asserted when the register file needs to be written.



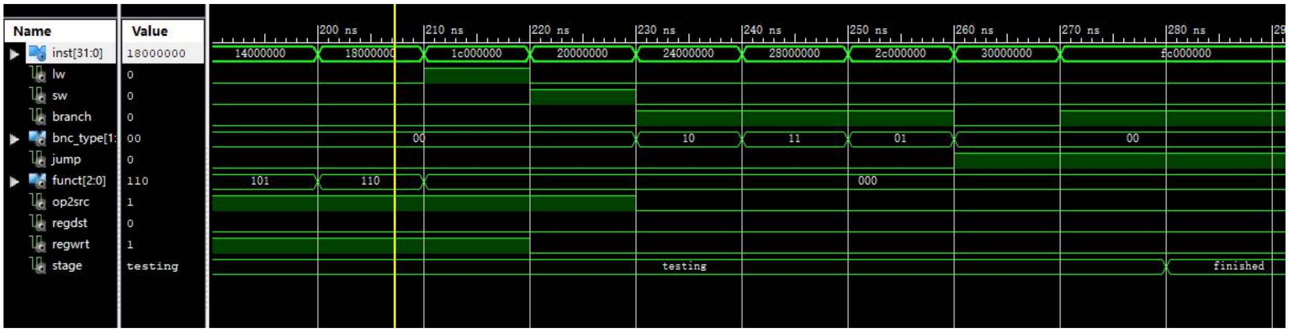
We tested all 18 instrutions and checked if the output controll signals are correct.

2.1.2.2 Functional Simulation

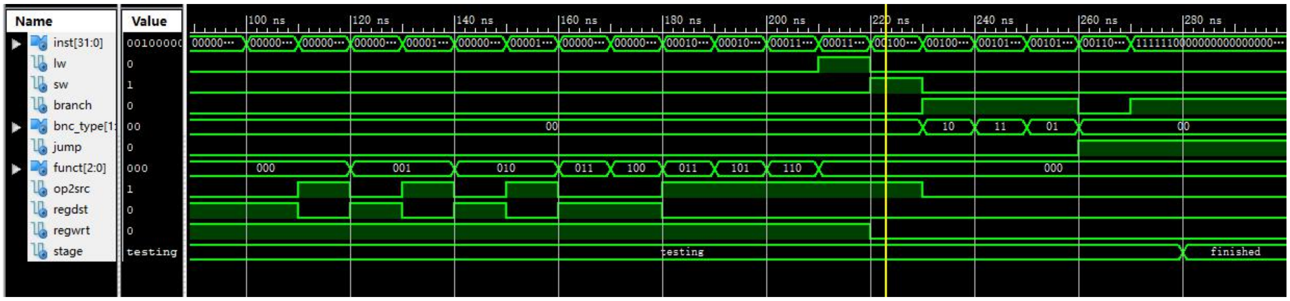
First 9 insts:



Last 9 insts:

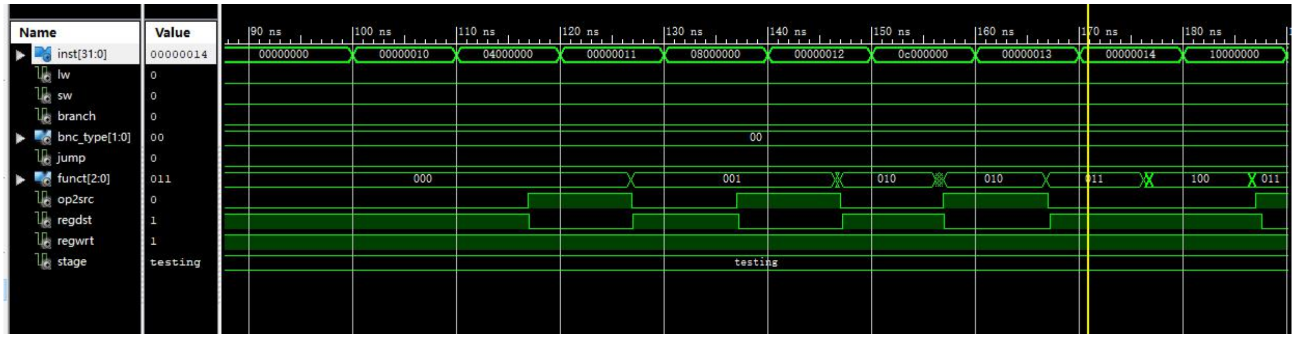


In all :

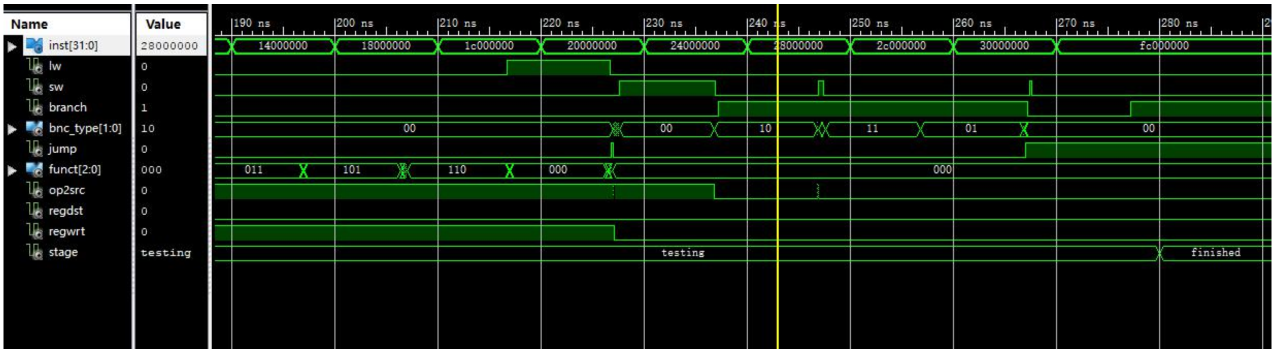


2.1.2.3 Timing Simulation

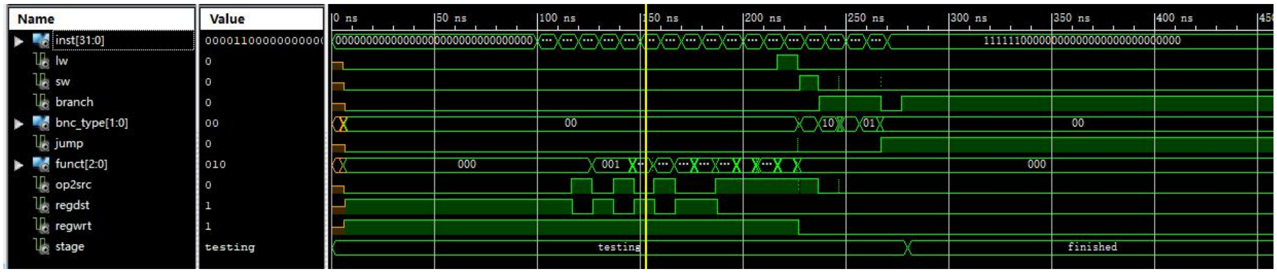
First 9:



Last 9:



In all:

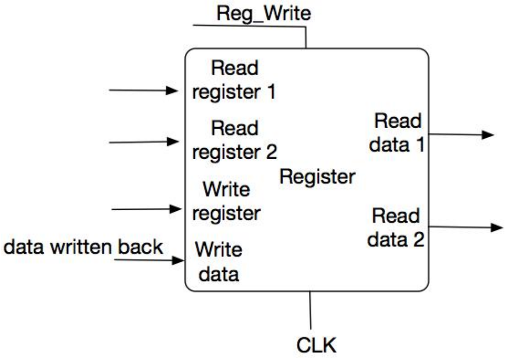


Name	Value
inst[31:0]	0000110000000000(
lw	0
sw	0
branch	0
bnc_type[1:0]	00
jump	0
funct[2:0]	010
op2src	0
regdst	1
regwrt	1
stage	testing

2.1.3 Register File

2.1.3.1 Brief Introduction

This block contains 32, 32-bit registers and is synchronous with clock. The Register File gets the register address from Instruction Memory, and output 2 read data as inputs for ALU. There is also an independent write register in this block, which is used to store the data written back from Data Memory.



The checking operations are described below:

1. Let rs, rt, rd <= “00000” -- point to reg(0)
2. Generate a random number and let wd <= number -- a random value to be written
3. Set we <= ‘1’ to perform write function -- write the random value into rd
4. Check if rd1 and rd2 are equal to wd -- check the values read from rs and rt

Then repeat 2-4 to generate 1000 random numbers in total to check if the read values are equal to the written values in all the 1000 cases.
After these 1000 case, let rs, rt and rd point to reg(1) (“00001”) and do another 1000 random cases, then reg(2), reg(3), …… until reg(30). (reg(31) is not used for write operation)

So there are totally 31*1000 = 31000 cases checked in this test bench. If anything goes wrong during the simulation, it will stop and report “Wrong”. If all cases pass, it will show that “1000*31 cases passed”.

2.1.3.2 Functional Simulation:

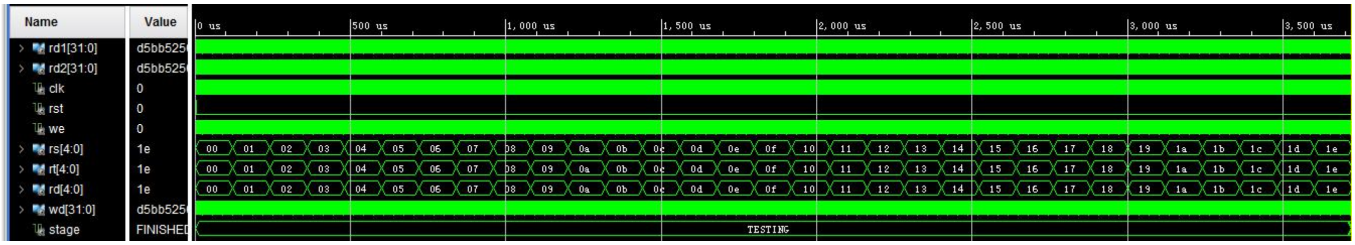


Figure 1. The whole simulation.

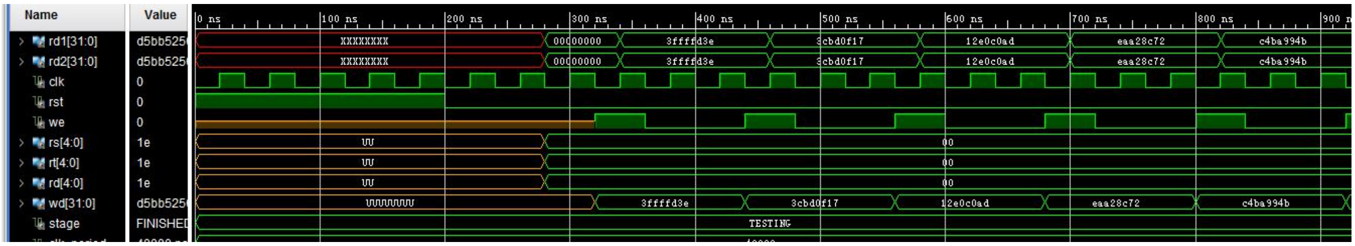


Figure 2. The first few cases

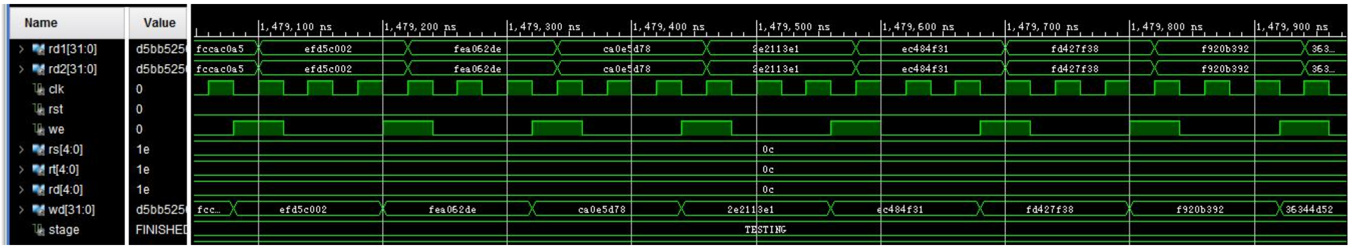


Figure 3. A few cases during the simulation

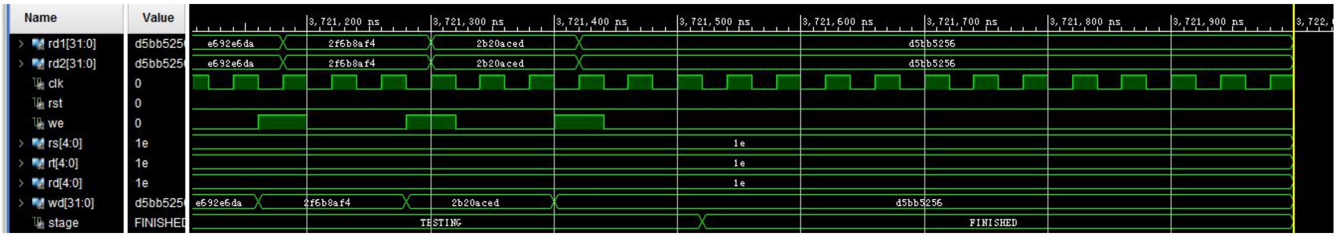


Figure 4. The last few cases.

Notice that after all the cases finished, the state will change from TESTING to FINISHED.

2.1.3.3 Timing simulation

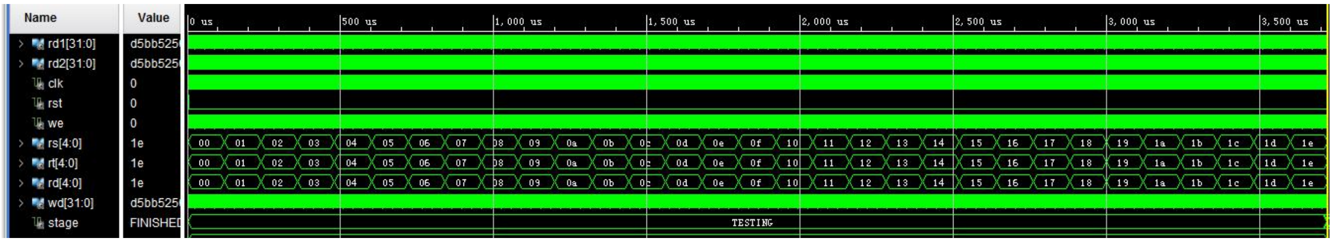


Figure 1. The whole simulation

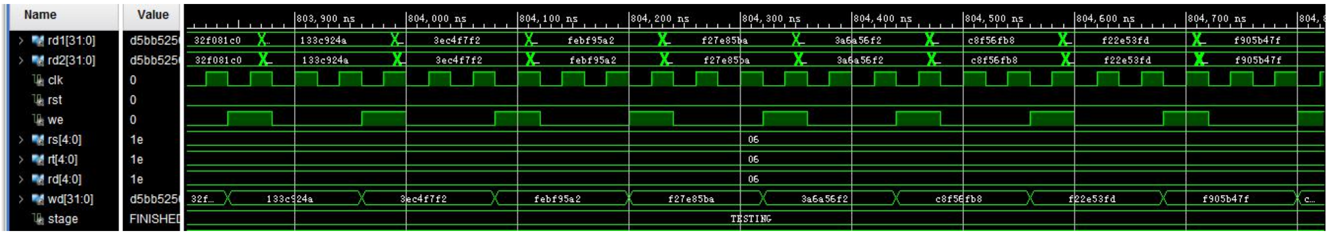


Figure 2. A few cases during the simulation

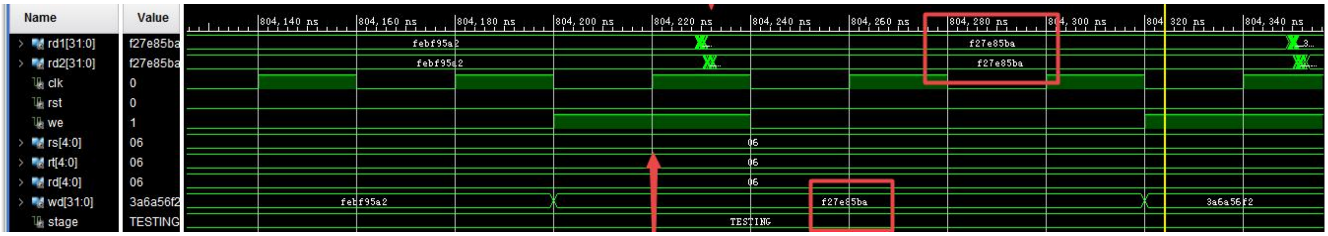


Figure 3. One case in the simulation

We can see there's a delay (about 10ns) between the clock rising edge and the data being read, because it will take some time to write a data into RF and then read it.

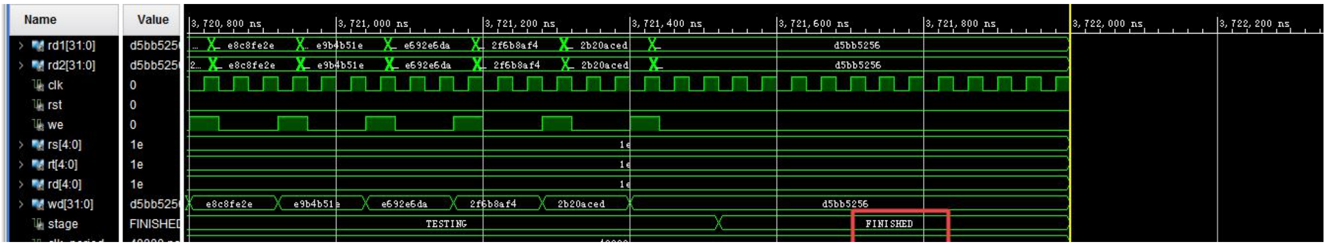


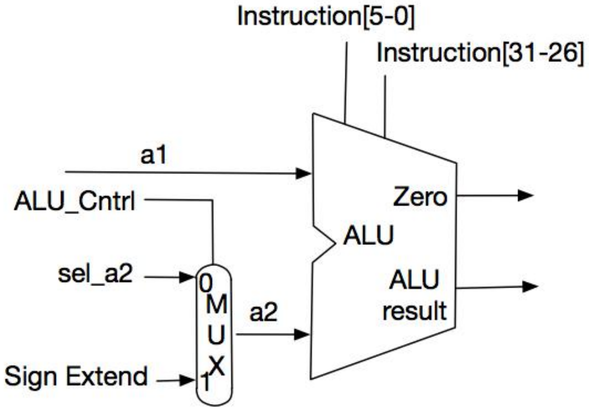
Figure 4. Last few cases

After all the cases finish, the state becomes "FINISHED".

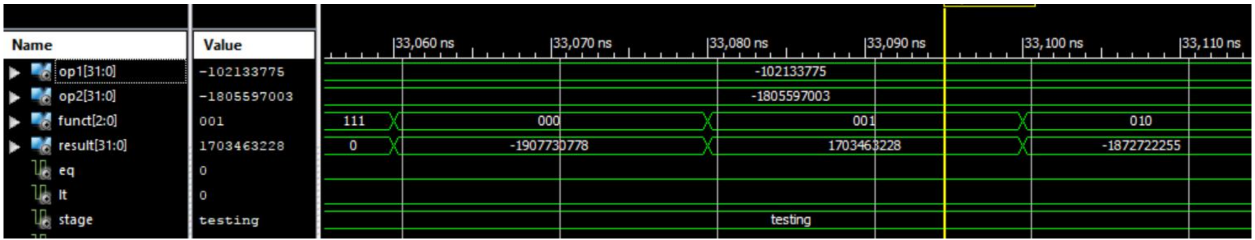
2.1.4 ALU

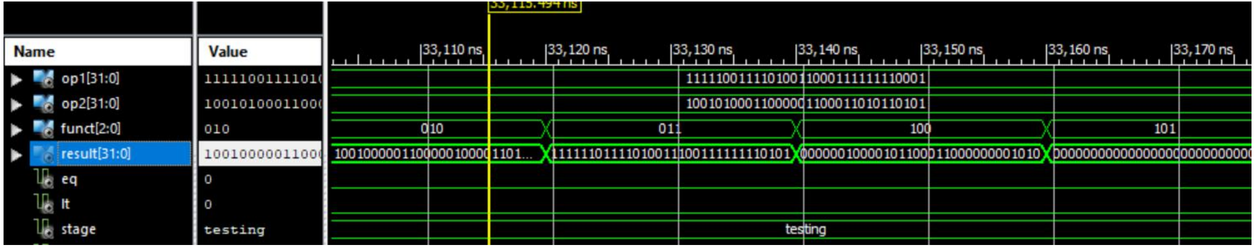
2.1.4.1 Brief Introduction:

This block performs operations such as addition, subtraction, comparison, etc., and it uses Opcode and Function as control signals to determine which kind of arithmetic or logic operation it will perform. There are 2 inputs which are a1 and a2, and by computing the 2 inputs it will generate an output which is zero or ALU result. The output zero is used for branch instructions, and the ALU result is used for others.

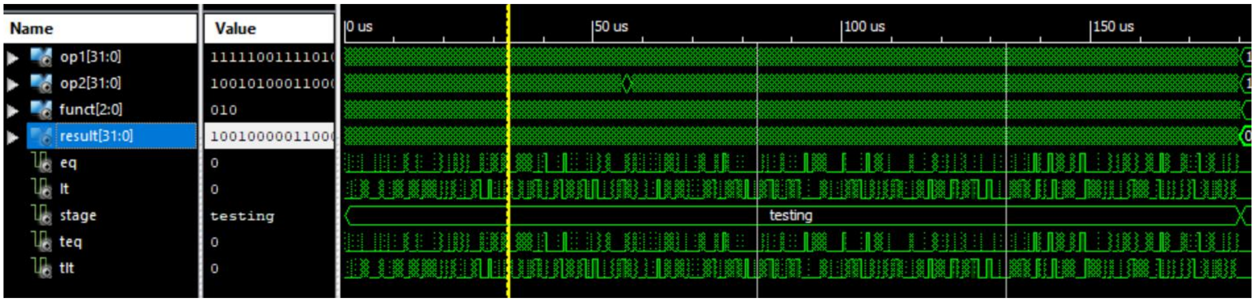


2.1.4.2 Functional Simulation

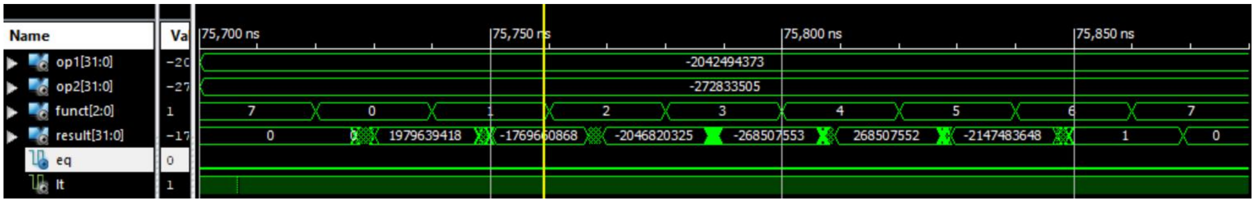


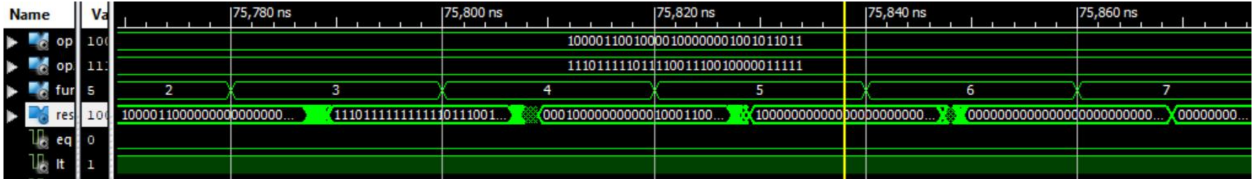


All 1000 random cases:

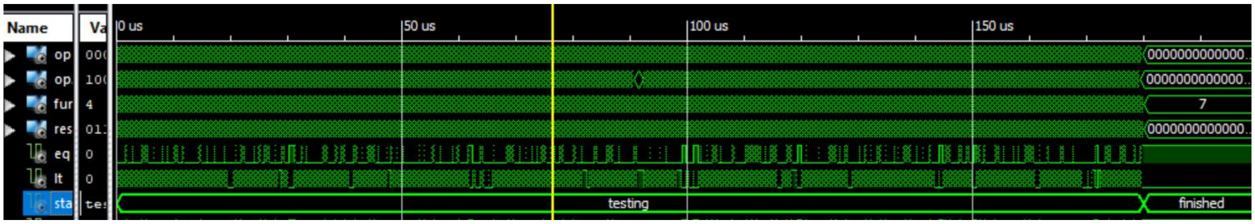


2.1.4.3 Timing simulation





All 1000 random cases:



2.1.5 Instruction Memory

2.1.5.1 Brief Introduction

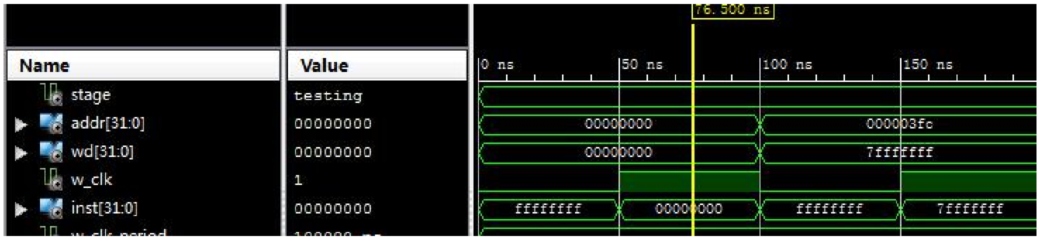
Each instruction has its own address, marked by the 32-bit vector of the program counter. When a given instruction is needed, the Instruction Memory delivers a 32-bit wide instruction. The following considerations have to be taken into account for the Instruction Memory:

- Each instruction is one word long (32 bits).
- Each instruction is addressed by a multiple of 4 bytes (1 word).
- Each instruction lies at a multiple of 4 bytes. By contrast, the program counter counts in terms of bytes. To avoid systematic "jumps" by four instructions when the new program counter is proposed, the program counter is divided by 4.
- At the rising edge of w_clk, we write the value of the wd into instruction memory. In this way, it can support changing the program while our processor is running on the FPGA.

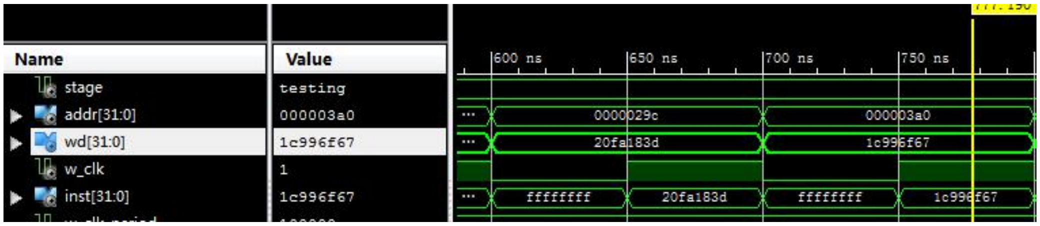
Random generate 1000 test cases.

1. Generate random variable to change addr and random variable to change instruction
2. When stage equals to finished, all the test case passed.

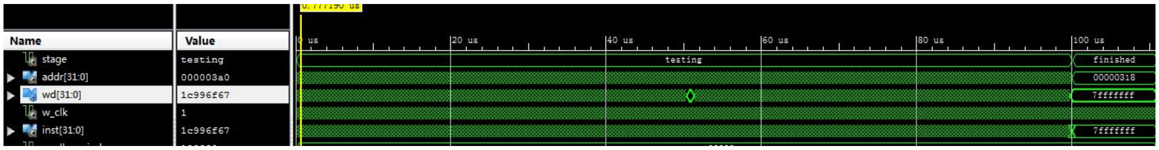
2.1.5.2 Functional Simulation



Name	Value
stage	testing
addr[31:0]	00000000
wd[31:0]	00000000
w_clk	1
inst[31:0]	00000000

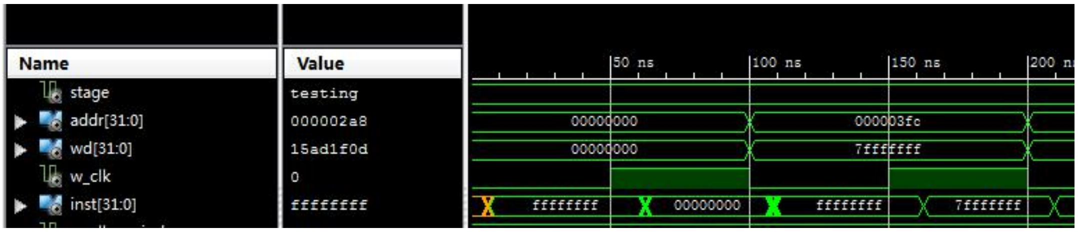


Name	Value
stage	testing
addr[31:0]	000003a0
wd[31:0]	1c996f67
w_clk	1
inst[31:0]	1c996f67

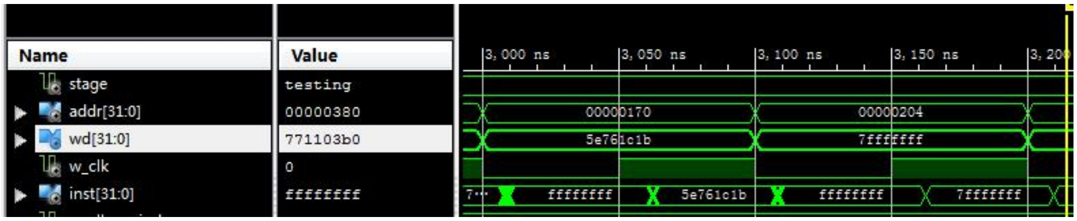


Name	Value
stage	testing
addr[31:0]	000003a0
wd[31:0]	1c996f67
w_clk	1
inst[31:0]	1c996f67

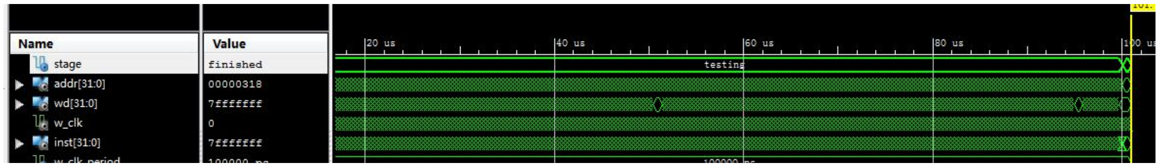
2.1.5.3 Timing Simulation



Name	Value
stage	testing
addr[31:0]	000002a8
wd[31:0]	15ad1f0d
w_clk	0
inst[31:0]	ffffffff



Name	Value
stage	testing
addr[31:0]	00000380
wd[31:0]	771103b0
w_clk	0
inst[31:0]	ffffffff

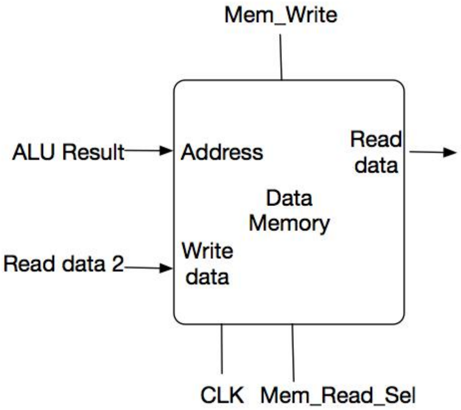


Name	Value
stage	finished
addr[31:0]	00000318
wd[31:0]	7fffffff
w_clk	0
inst[31:0]	7fffffff

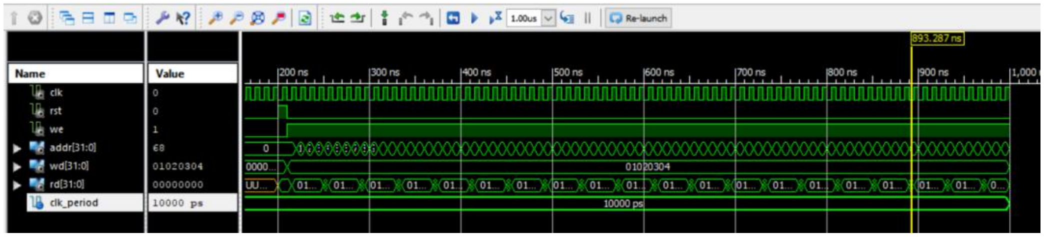
2.1.6 Data Memory

2.1.6.1 Brief Introduction

The data memory unit is only accessed by the load and store instructions. The load instruction is controlled by Mem_Read signal and it uses the ALU Result value as an address to index the data memory. The output of Data Memory for load instruction will be written back to Register File. A store instruction is controlled by Mem_Write signal and it writes the data which is read from the register into the computed address of Data Memory.



2.1.6.2 Functional Simulation

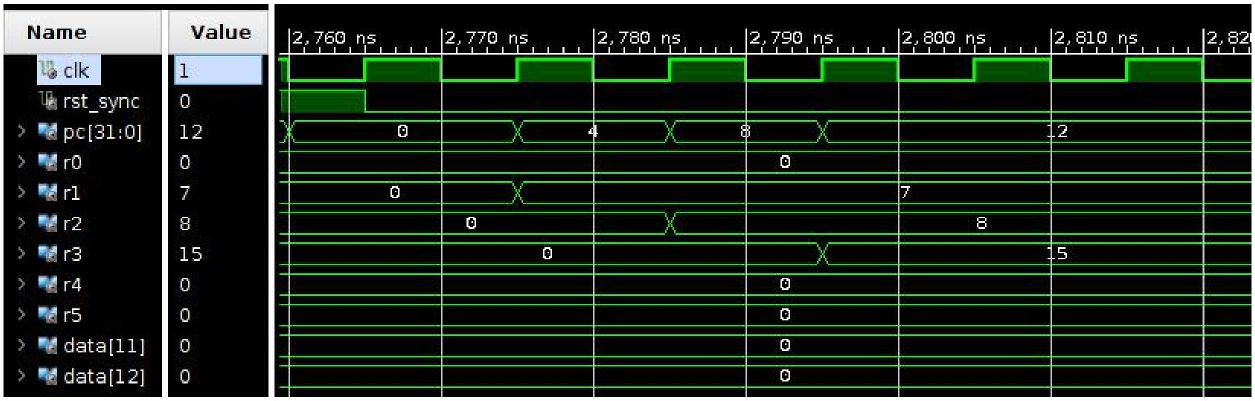


Name	Value
clk	0
rst	0
we	1
addr[31:0]	68
wd[31:0]	01020304
rd[31:0]	00000000
clk_period	10000 ps

2.2 Complete Integrated processor sample code Test

2.2.1 Code 1 Functional Simulation

```
Sample code 1:
00000100000000010000000000000111    --ADDI R1, R0, 7 // R1 = 7
00000100000000100000000000001000    --ADDI R2, R0, 8 // R2 = 8
00000000001000001000110000010000    --ADD R3, R1, R2 // R3 = R1 + R2 =15
11111100000000000000000000000000    --HAL // HALT
```



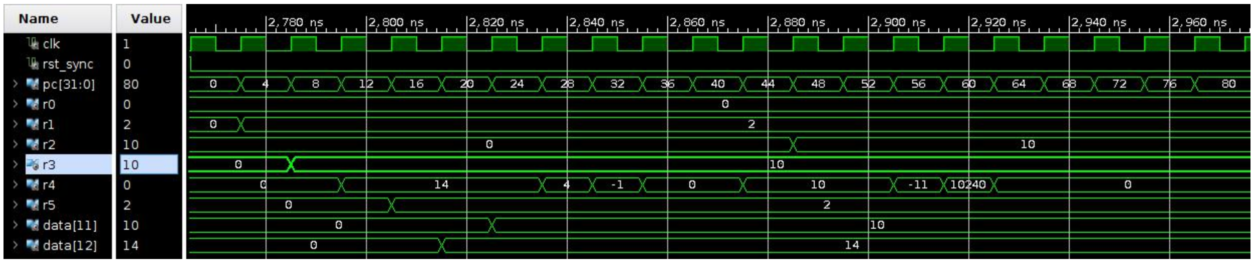
Name	Value
clk	1
rst_sync	0
pc[31:0]	12
r0	0
r1	7
r2	8
r3	15
r4	0
r5	0
data[11]	0
data[12]	0

As we can see from the result, the final result in R3 is 15

2.2.2 Code 2 Functional Simulation

```
Sample code 2:
000001 00000 00001 0000000000000010        --ADDI R1, R0, 2 //R1=R0+2(decimal)
000001 00000 00011 0000000000001010        --ADDI R3, R0, 10 //R3=R0+10(decimal)
000001 00000 00100 0000000000001110        --ADDI R4, R0, 14 //R4=R0+14(decimal)
000001 00000 00101 0000000000000010        --ADDI R5, R0, 2 //R5=R0+2
001000 00011 00100 0000000000000010        --SW R4, 2(R3) //Mem[R3+2]=R4
001000 00011 00011 0000000000000001        --SW R3, 1(R3) //Mem[R3+1]=R3
000000 00100 00011 00100 00000 010001      --SUB R4, R4, R3 //R4=R4-R3
000010 00000 00100 0000000000000001        --SUBI R4, R0, 1 //R4=R0-1(decimal)
000000 00011 00010 00100 00000 010010      --AND R4, R2, R3 //R4=R2 and R3
000011 00010 00100 0000000000001010        --ANDI R4, R2, 10 //R4=R2 and 10(decimal)
000000 00011 00010 00100 00000 010011      --OR R4, R2, R3 //R4= R2 or R3
000111 00011 00010 0000000000000001        --LW R2, 1(R3) //R2=Mem[1+R3]
000100 00010 00100 0000000000001010        --ORI R4, R2, 10 //R4=R2 or 10(decimal)
000000 00011 00010 00100 00000 010100      --NOR R4, R2, R3 //R4= R2 nor R3
```

```
000101 00010 00100 0000000000001010       --SHL R4, R2, 10 //R4= R2 << 10(decimal)
000110 00010 00100 0000000000001010       --SHR R4, R2, 10 //R4=R2 >> 10(decimal)
001010 00000 00101 1111111111111110       --BEQ R5, R0, -2
001001 00100 00101 0000000000000000       --BLT R5, R4, 0
001011 00100 00101 0000000000000000       --BNE R5, R4, 0
001100 00000000000000000000010100         --JMP 20
111111 00000000000000000000000000         --HAL
```



2.3 RC5 Implementation

2.3.1 Brief Introduction

We wrote the RC5 code in assembly (rc5.asm & rc5_optimized.asm). We also wrote a simple compiler in python (load_tb_instructions.py) to help convert assembly to machine code (rc5.binary & rc5_optimized.binary).

2.3.2 Simple RC5 test code

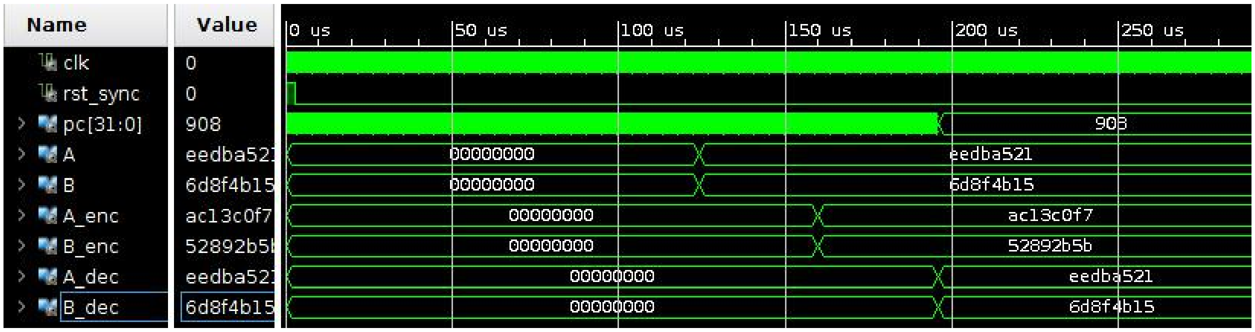
In our simple test code (rc5.asm), we fixedly set:

ukey = 0x91cea91001a5556351b241be19465f91
A_in = 0xeedba521
B_in = 0x6d8f4b15.

Then do one round of encryption and decryption on A_in and B_in.
The expected encryption result is: A_enc = 0xac13c0f7, B_enc = 0x52892b5b

2.3.2.1 Functional simulation

A, B, A_enc, B_enc, A_dec, B_dec are picked from data_mem and renamed, where A, B are raw input at data_mem[40,41]; A_enc, B_enc are encrypted data at data_mem[42,43]; A_dec, B_dec are decrypted data at data_mem[44,45].



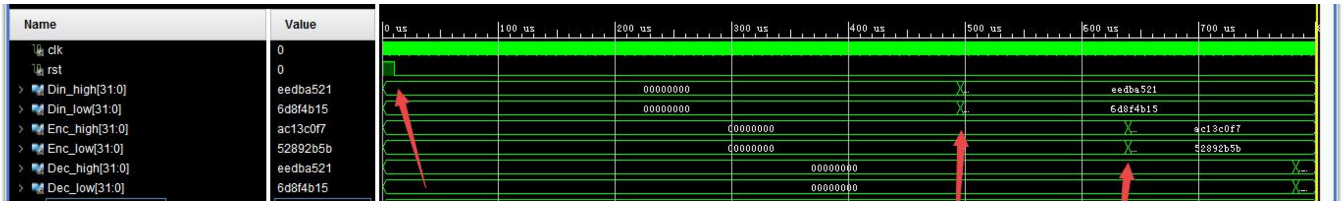
Name	Value
clk	0
rst_sync	0
pc[31:0]	908
A	eedba521
B	6d8f4b15
A_enc	ac13c0f7
B_enc	52892b5b
A_dec	eedba521
B_dec	6d8f4b15

2.3.2.2 Timing simulation

In this simulation, we will just show the values in data_mem(40 to 45). We renamed those values so that they will be clearer in the waveform. They are labeled with Din(initial), Enc(encrypted), Dec(decrypted). Besides, a label of high/low (or say A/B) indicates whether the 32-bit signal is the high 32 bits or the low 32 bits of the 64-bit value.

The sequence of the signals is:

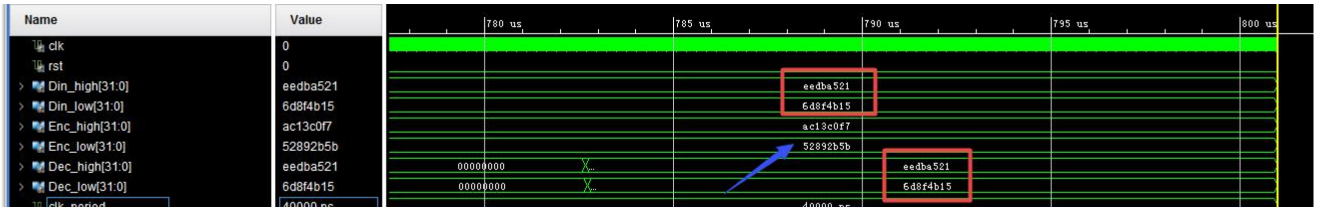
clk, rst, Din_high, Din_low, Enc_high, Enc_low, Dec_high, Dec_low



Name	Value
clk	0
rst	0
Din_high[31:0]	eedba521
Din_low[31:0]	6d8f4b15
Enc_high[31:0]	ac13c0f7
Enc_low[31:0]	52892b5b
Dec_high[31:0]	eedba521
Dec_low[31:0]	6d8f4b15

This is the whole waveform of the timing simulation. We add some arrows to mark the events in the process.

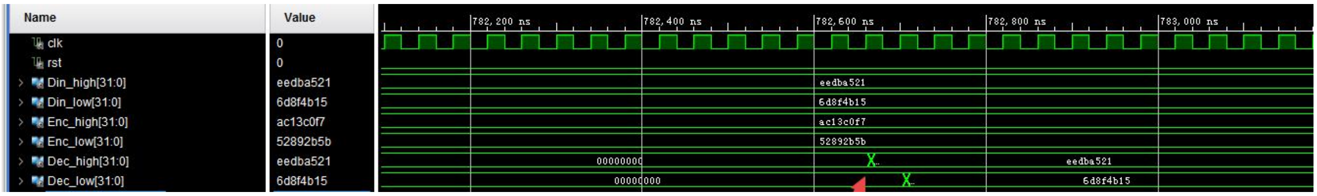
- At the 1st arrow, we let rst become low so that we can begin the Key-Generation.
- At the 2nd arrow, the key-generation is completed, and the value of Din is entered so that we will begin the Encryption.
- At the 3rd arrow, the encrypted value is obtained and saved, so we begin Decryption on this value.
- At the 4th arrow, the decrypted value is obtained and saved.



Name	Value
clk	0
rst	0
Din_high[31:0]	eedba521
Din_low[31:0]	6d8f4b15
Enc_high[31:0]	ac13c0f7
Enc_low[31:0]	52892b5b
Dec_high[31:0]	eedba521
Dec_low[31:0]	6d8f4b15
clk_period	40000 ns

This show the final results. We can see that the encrypted-decrypted value is the same as the initial Din (the two red boxes). Besides, the encrypted value (pointed by the blue arrow) is also a correct encryption result:

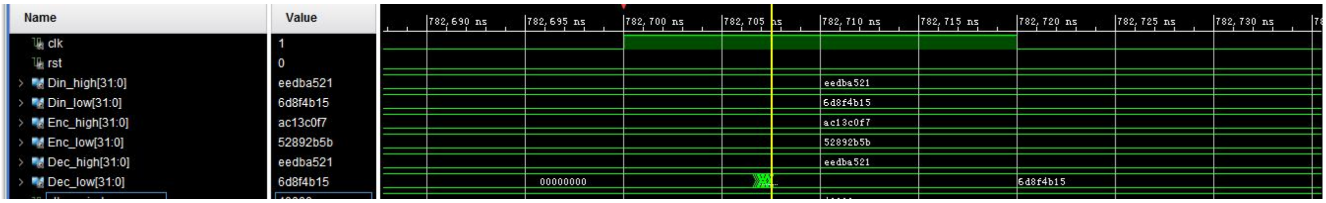
- Enc_high & Enc_low = AC13C0F752892B5B



Name	Value
clk	0
rst	0
Din_high[31:0]	eedba521
Din_low[31:0]	6d8f4b15
Enc_high[31:0]	ac13c0f7
Enc_low[31:0]	52892b5b
Dec_high[31:0]	eedba521
Dec_low[31:0]	6d8f4b15

Here we go into some details about the time we get the final decrypted value. Notice that the low 32 bits are obtained one cycle later than the high 32 bits.

It’s easy to understand that: the CPU will perform one instruction in one cycle. So we have to first store the high 32 bits of the 64-bit output, then store the low 32 bits.

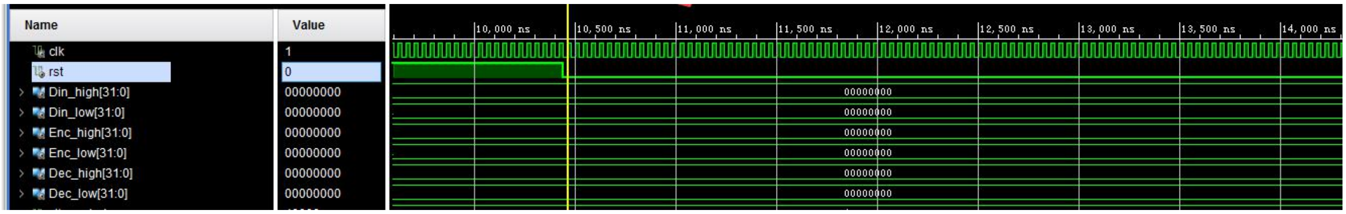


Name	Value
clk	1
rst	0
Din_high[31:0]	eedba521
Din_low[31:0]	6d8f4b15
Enc_high[31:0]	ac13c0f7
Enc_low[31:0]	52892b5b
Dec_high[31:0]	eedba521
Dec_low[31:0]	6d8f4b15

This figure is the more detailed capture of the time we get the final value (low 32 bits). As this is a timing simulation, there is a delay between the clock rising edge and the completion of the store operation (showed by the two arrows).

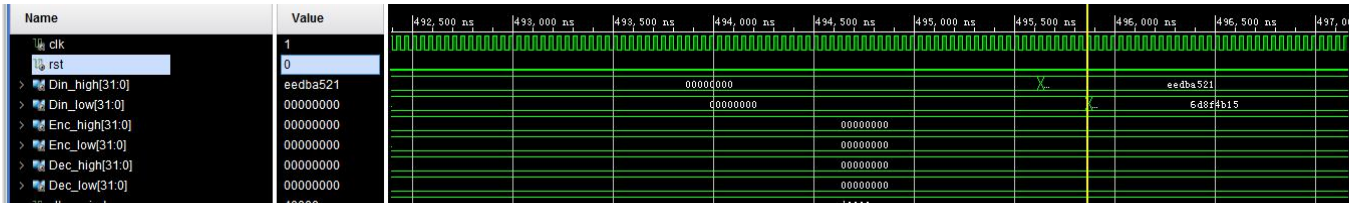
Clock frequency and Latency: The critical path delay is 38 ns, so the highest clock frequency is approximately 25 MHz. We also estimated the clock cycles that the CPU uses to run Key-Generation, Encryption, and Decryption. In the following screenshot:

- Time 1: Key-Generation started
- Time 2: Key-Generation ended & Encryption started
- Time 3: Encryption ended & Decryption started
- Time 4: Decryption ended



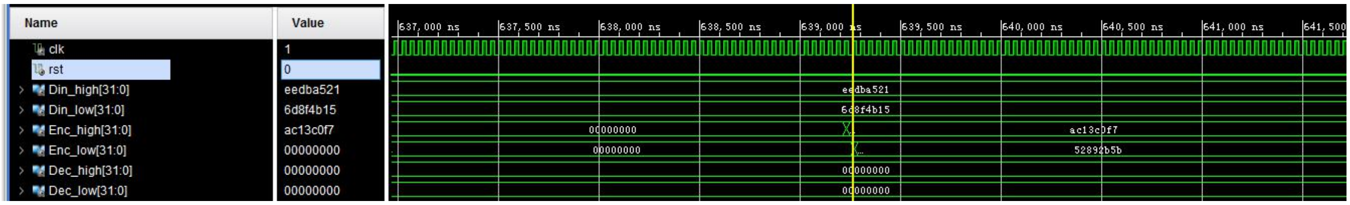
Name	Value
clk	1
rst	0
Din_high[31:0]	00000000
Din_low[31:0]	00000000
Enc_high[31:0]	00000000
Enc_low[31:0]	00000000
Dec_high[31:0]	00000000
Dec_low[31:0]	00000000

Time 1 = 10460ns



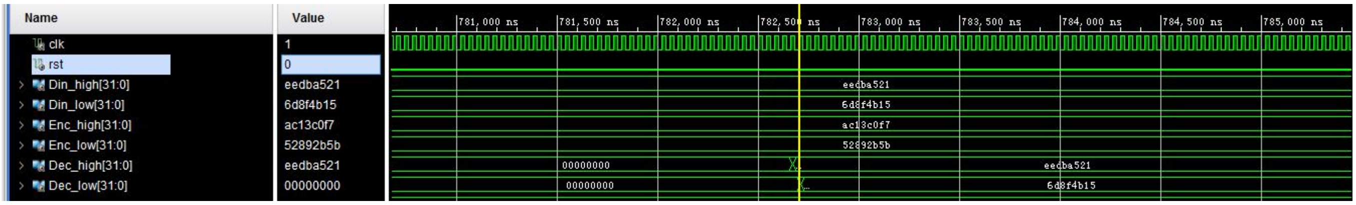
Name	Value
clk	1
rst	0
Din_high[31:0]	eedba521
Din_low[31:0]	00000000
Enc_high[31:0]	00000000
Enc_low[31:0]	00000000
Dec_high[31:0]	00000000
Dec_low[31:0]	00000000

Time 2 = 495860ns



Name	Value
clk	1
rst	0
Din_high[31:0]	eedba521
Din_low[31:0]	6d8f4b15
Enc_high[31:0]	ac13c0f7
Enc_low[31:0]	00000000
Dec_high[31:0]	00000000
Dec_low[31:0]	00000000

Time 3 = 639260ns



Name	Value
clk	1
rst	0
Din_high[31:0]	eedba521
Din_low[31:0]	6d8f4b15
Enc_high[31:0]	ac13c0f7
Enc_low[31:0]	52892b5b
Dec_high[31:0]	eedba521
Dec_low[31:0]	00000000

Time 4 = 782700ns

We can use this formula to calculate the clock cycles:

	Start	End	Time-Diff	Latency	clk period
Key-Gen	10460	495860	485400	12135	40
Encryption	495860	639260	143400	3585	
Decryption	639260	782700	143440	3586	

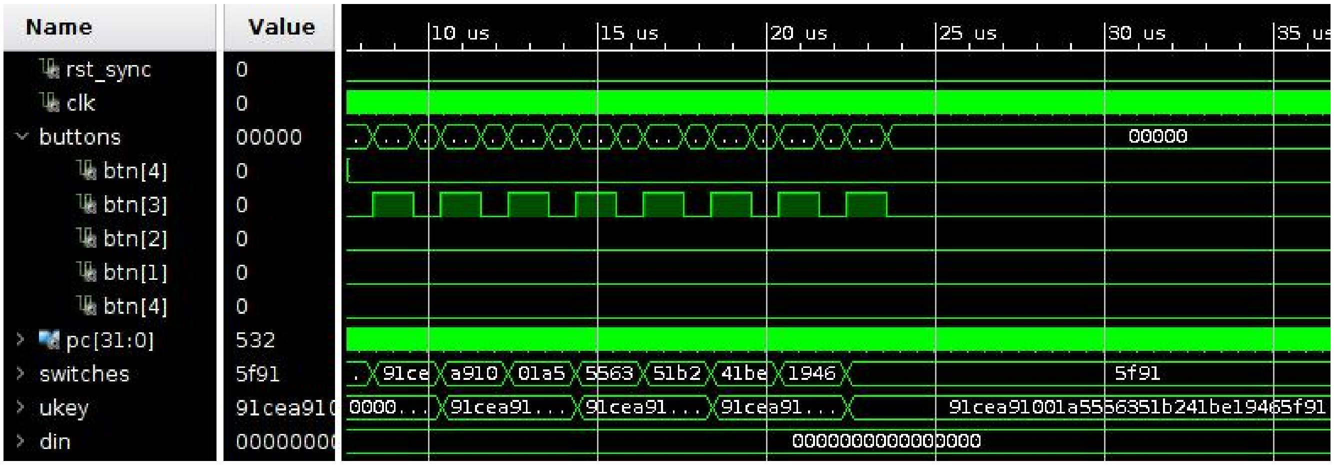
* *The Latency is presented in number of cycles.*

Notice that there are a few read & write operations before or after each of these three functions, but the cycles they take can be ignored compared to the cycles taken by Key-Gen, Encryption and Decryption.

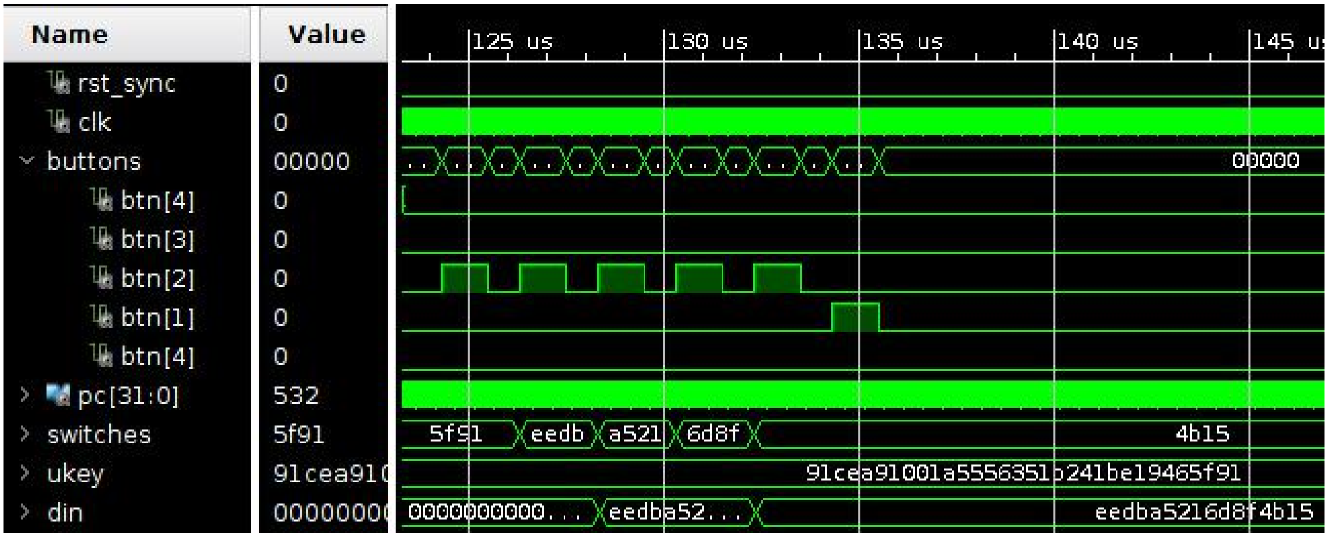
2.3.3 Additional optimized RC5 code

In our optimized code (rc5_optimized.asm), we mainly tested using buttons and switches to input data, which would be very helpful to our fpga implementation. For now we just simply show the simulation results. We will give a detailed introduction of our special design on buttons&switches in presentation 2 and will add it into next report.

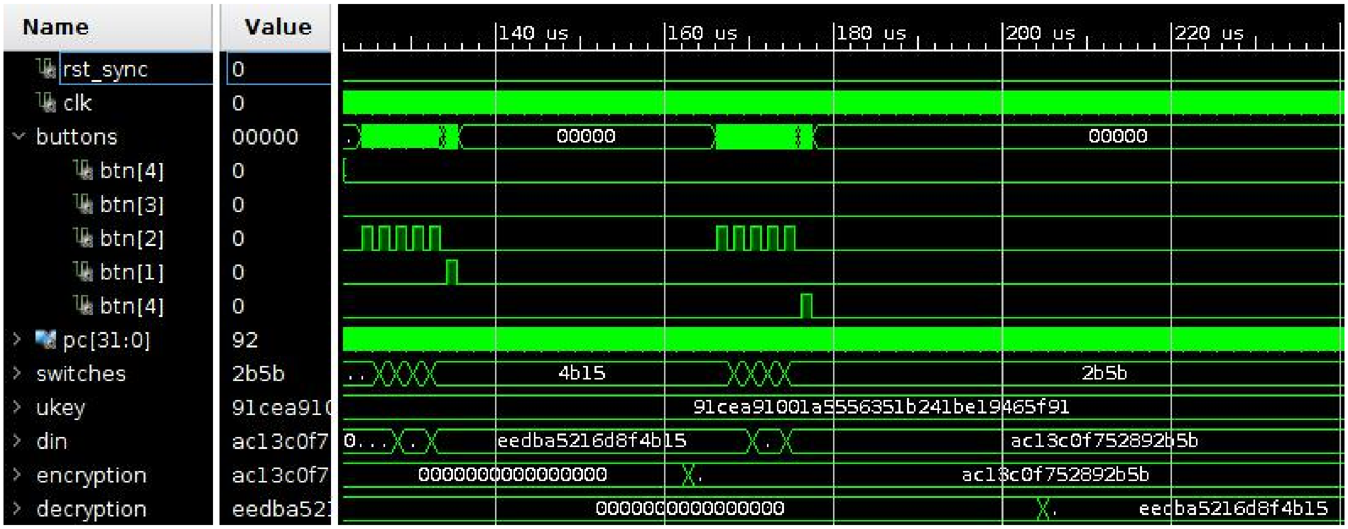
Using button3 & switches to input ukey:



Using button2 & switches to input din, then starting encryption by button1



Both encryption and decryption:



Name	Value
rst_sync	0
clk	0
buttons	00000
btn[4]	0
btn[3]	0
btn[2]	0
btn[1]	0
btn[4]	0
pc[31:0]	92
switches	2b5b
ukey	91cea91(
din	ac13c0f7
encryption	ac13c0f7
decryption	eedba52