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

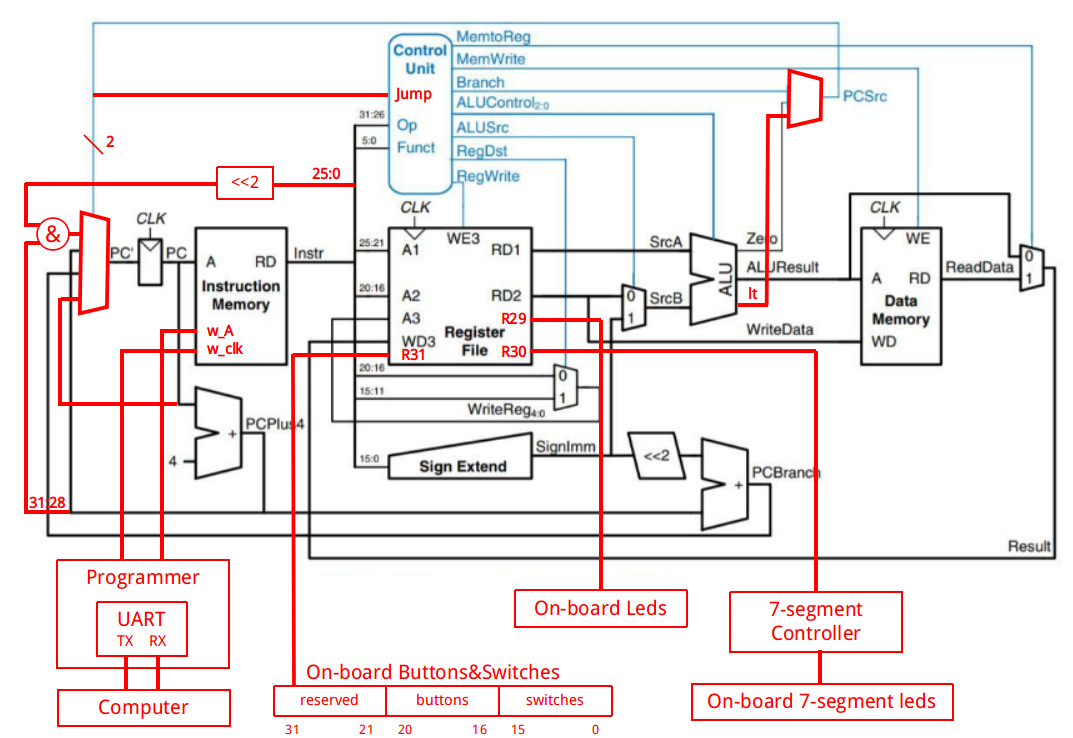
# Abstract

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.

# Processor Design and Test

## Summary

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.

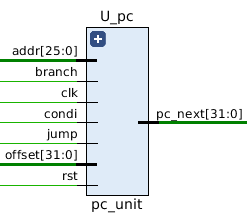


Above is the block diagram of our processor. The red parts are what we added to the original design. Minor changes (like renaming the ports) are not shown in this diagram. Some modules are designed for FPGA implementation and not introduced in this report, but will be in presentation 2. We will add them later in the final report after FPGA implementation.

## Component design and test

#### Program counter (PC) register

###### Implementation

Signal list:

in clk: clock signal

in rst: reset signal, active-high

in jump: pc\_src select signal for JMP, active-high

in addr: address to jump

in branch: pc\_src select signal for BXX, active-high

in condi: branch condition signal, active-high

in offset: offset to branch

out pc\_next: address of next instruction

This is a 32-bit register that contain the address of the next instruction to be executed. pc\_next will be updated synchronously under 4 different conditions, as show below:

|  |  |  |
| --- | --- | --- |
| Type | PC\_next | Condition |
| Continuously | PC + 4 | jump=0 & branch=0 \* |
| Branch | PC + 4 + offset\*4 | jump=0 & branch=1 & condi=1 |
| Jump | (PC+4)[31:28] & addr & “00” | jump=1 & branch=0 |
| Halt | PC | jump=1 & branch=1 |

*\* full logic should be jump=0 & (branch=0 | (branch=1 and condi=0))*

#### Testbench

For each condition, I 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’.

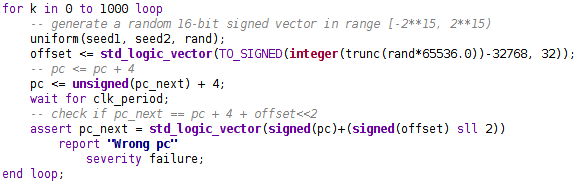


Figure 1.2.1 This is an example code of checking the branch condition

#### Functional simulation

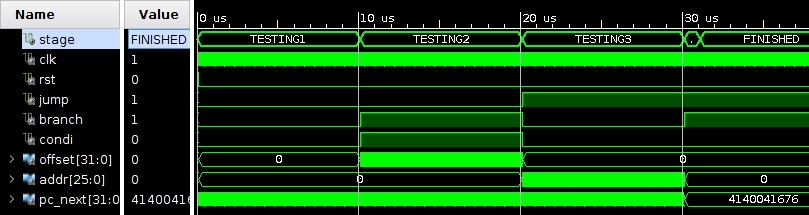
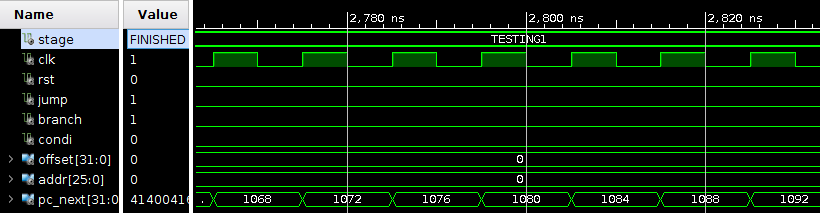
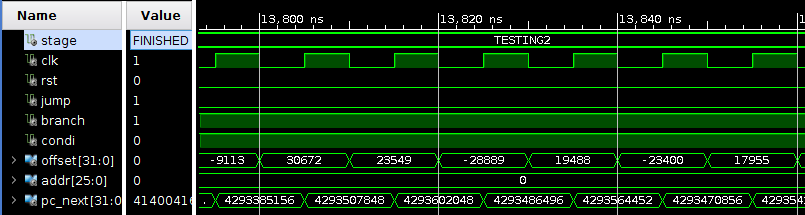


Figure 1.3.1 An overview. All cases passed.

 Figure 1.3.2 Testing continuously increment condition.

 Figure 1.3.3 Testing branch condition

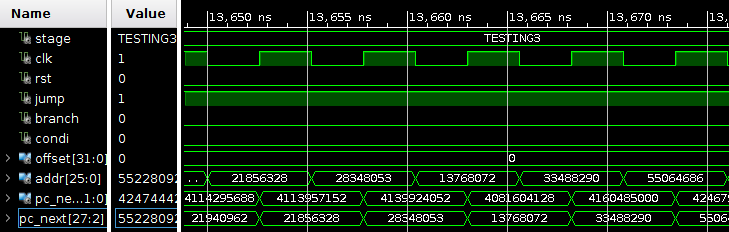


Figure 1.3.4 Testing jump condition

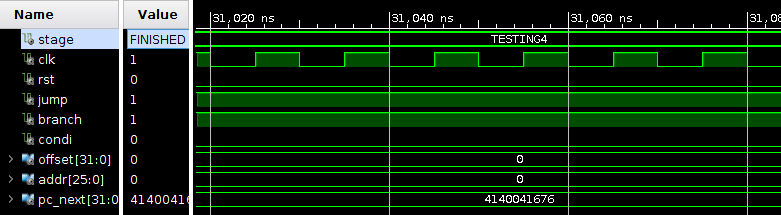


Figure 1.3.5 Testing halt condition

#### Timing simulation

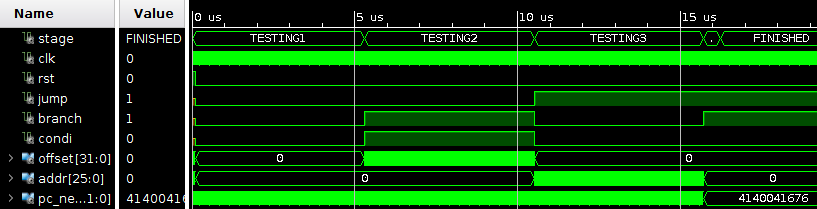


Figure 1.3.1 An overview. All cases passed.

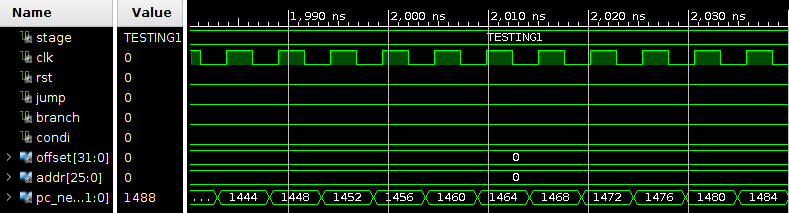


Figure 1.3.2 Testing continuously increment condition.

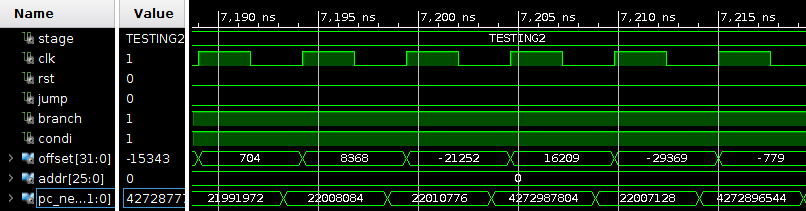


Figure 1.3.3 Testing branch condition

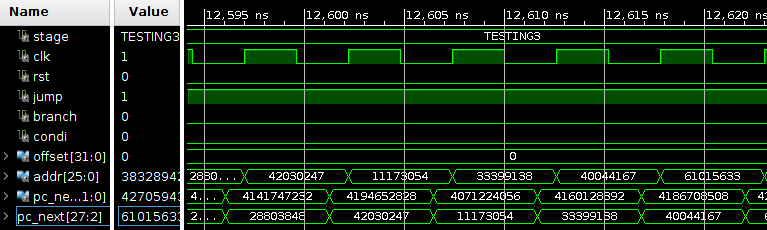


Figure 1.3.4 Testing jump condition

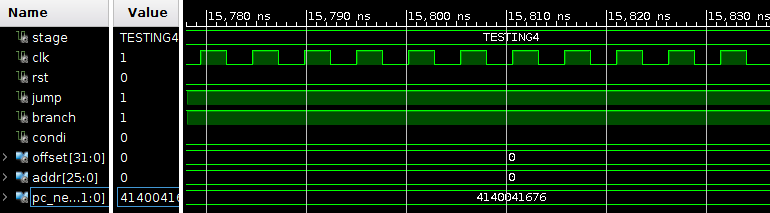


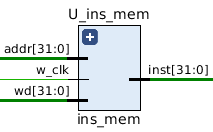
Figure 1.3.5 Testing halt condition

#### Timing analysis

|  |  |
| --- | --- |
| Critical path delay | 5.2 ns |
| Highest frequency | 192 MHz |

#### Instruction memory

###### Implementation



IN PORT:

Addr[31:0]: 32 bits address

W\_clk: clock signal

Wd[31:0]: 32 bits of the value we want to write in the instruction memory.

OUT PORT:

Inst[31:0]: the result 32 bits instructions.

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.

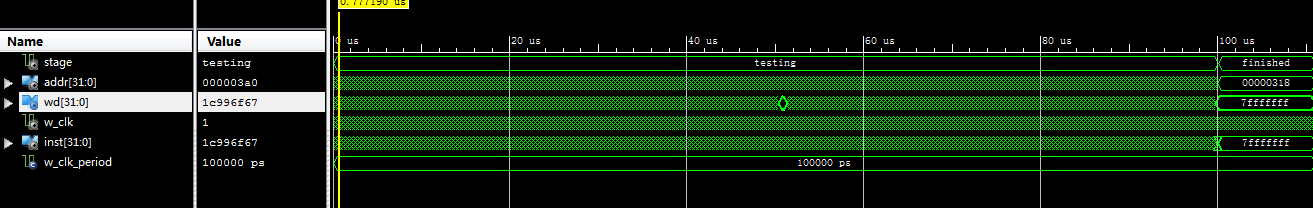
• At the rising edge of w\_clk, we write the value of the wd into instruction memory. In this way, it can support changing the program while our processor is running on the FPGA.

###### Testbench

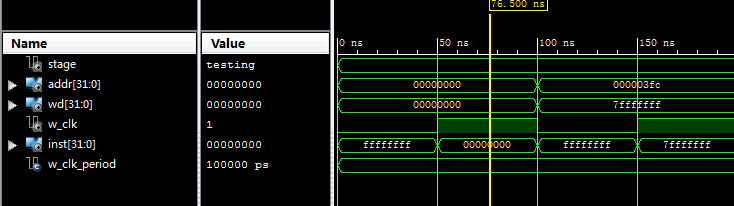
Random generate 1000 test cases.

1. Generate random variable to change addr(9 downto 2) and random variable to change instruction
2. When stage equals to finished, all the test case passed.

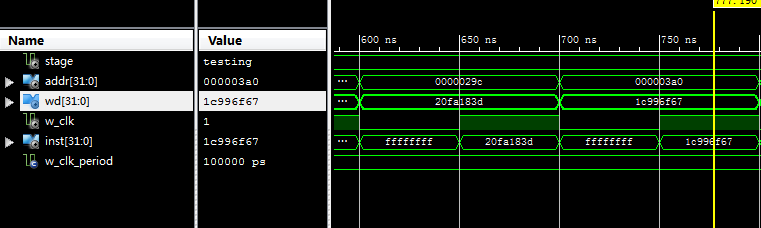
###### Functional Simulation



The stage is finished, which means all test cases passed.

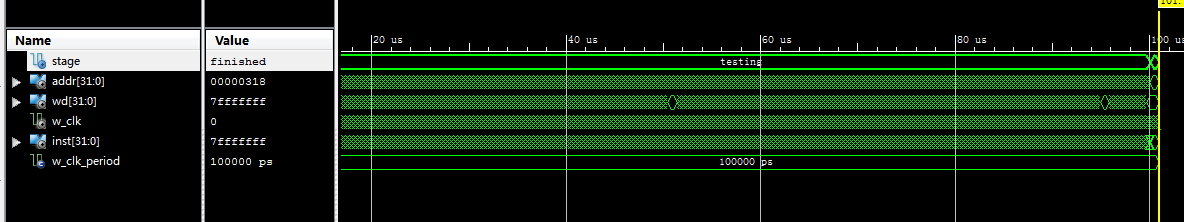


example1: the output instruction is the same as the wd value we wrote in.

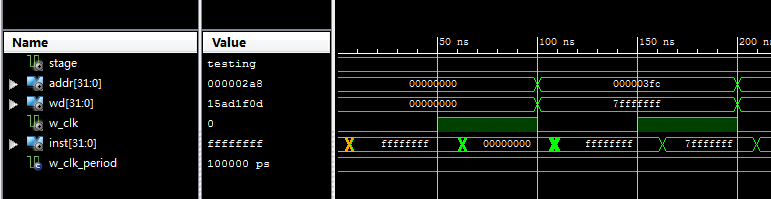


example2: the output instruction is the same as the wd value we wrote in.

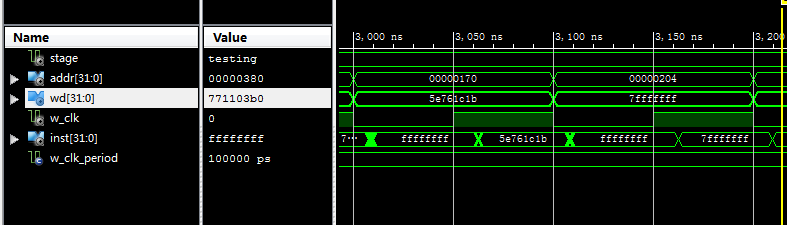
###### Timing Simulation



The stage is finished, which means all test cases passed.



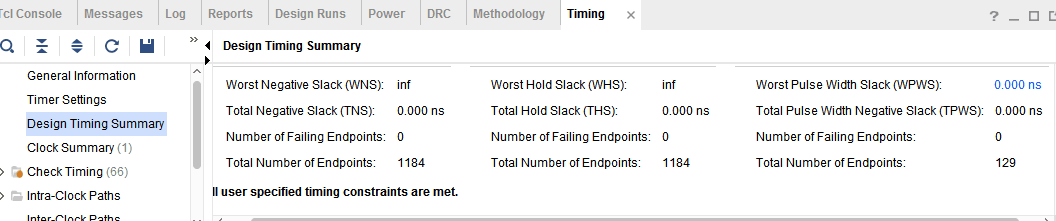
timing simulation for first example.



timing simulation for second example.

###### Timing Analysis

After several experiment, if we set the clock cycle as 1.82ns, we can get the timing summary as follows:

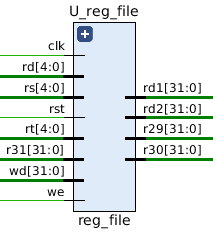




For the instruction memory part, the critical path delay is 1.82ns and the max frequency is 549.451MHZ. The latency is 1 clock cycle, so the propagation delay is 1.82ns.

#### Register file

###### Implementation



There are mainly these ports in Reg\_File:

In Ports:

* clk (clock signal) and rst (reset signal)
* rs, rt, rd: 5-bit inputs, which represent the address in Register File
* wd: 32-bit input, which contain the data that should be written into reg(rd)
* we: Write Enable signal
* r31: 32-bit input, for BTN & SW read-only

Out Ports:

* r29: 32-bit output, for LED Display
* r30: 32-bit output, for 7-Seg Display
* rd1, rd2: 32-bit outputs, which contain the data read from reg(rs) and reg(rt)

This component (Register File) is used to stage data between memory and the functional units in CPU.

The main functions of Reg\_File include:

* Asynchronous read: rd1 is always equal to the 32-bit data in the Register File, whose address is given by rs; and rd2 is always equal to the 32-bit data in RF with addres rt.
* Synchronous write: When clock rising edge, if the write enable signal “we” is high, the 32-bit data in wd will be written to RF with the address rd.
* The 32-bit output r29 is always equal to the value in reg(29), which will be used for LED Display. If we want to display something by LED, we should copy that value to reg(29).
* The 32-bit output r30 is always equal to the value in reg(30), used for 7-Seg Display. Similar to r29.
* The 32-bit input r31 contains the value obtained from SW and BTN buttons. reg(31) is always equal to r31, and cannot be modified by other operations (read-only).

###### Testbench

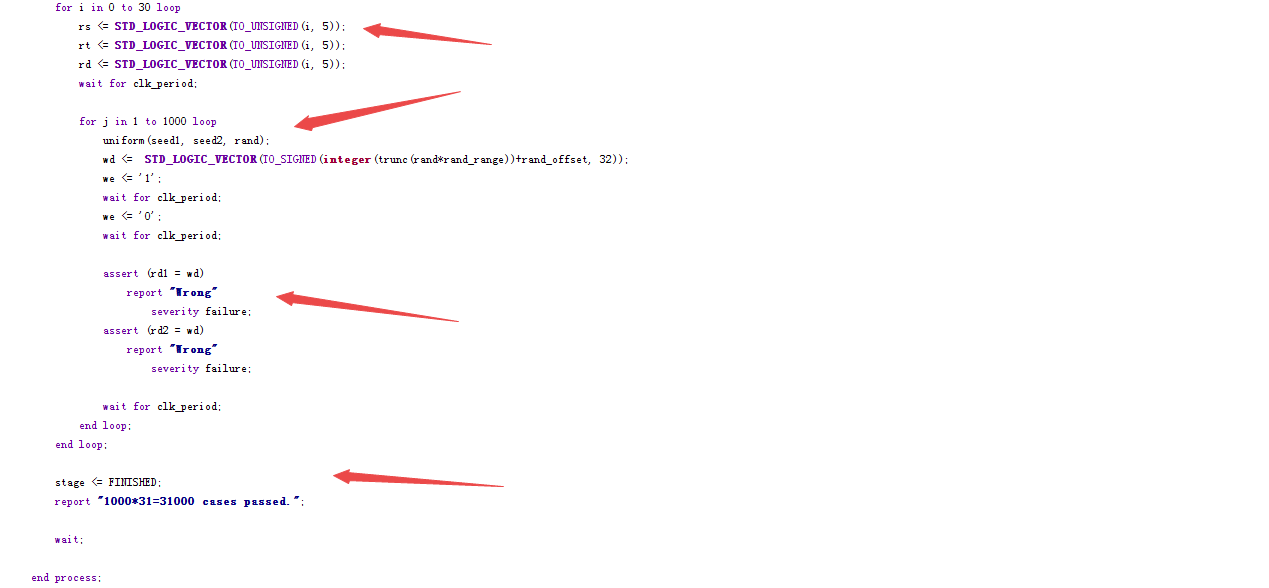
The checking operations in my test bench are described below:

* Let rs, rt, rd <= “00000” -- point to reg(0)
* Generate a random 32-bit number and let wd <= this number -- a random value to be written
* Set we <= ‘1’ to perform write function -- write the random value into reg(rd)
* 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”.



* Arrow 1: Outer loop, to check every register file from reg(0) to reg(30)
* Arrow 2: Inner loop, 1000 cases for each register. In each case, write a random number into it.
* Arrow 3: Check the data read from the register.
* Arrow 4: The report message after all cases pass.

###### Functional Simulation

Based on the test bench described above, we can run functional simulation now.

The sequence of the signals is:

rd1, rd2, clk, rst, we, rs, rt, rd, wd, state, clock\_period

(1)

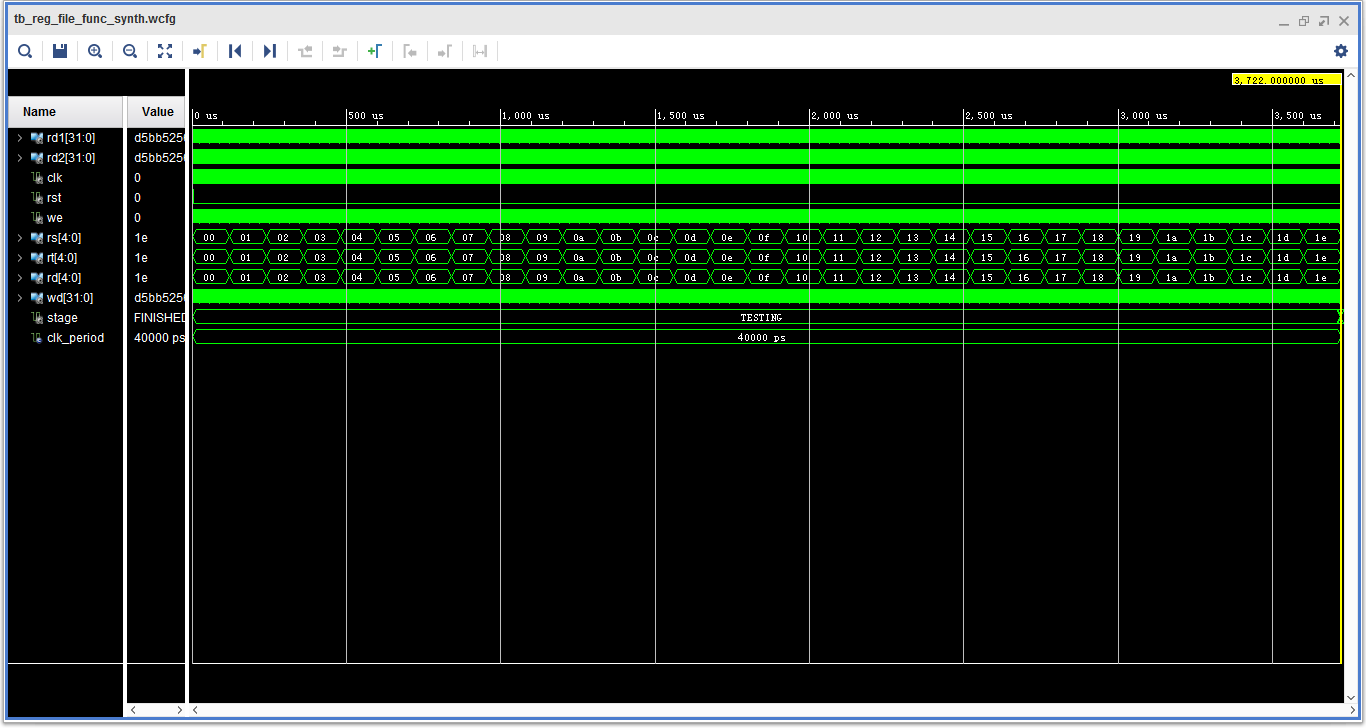


Figure 1. The whole simulation.

In this figure, we can just see the change of the addresses. In each address, there’s 1000 cases performed.

(2)

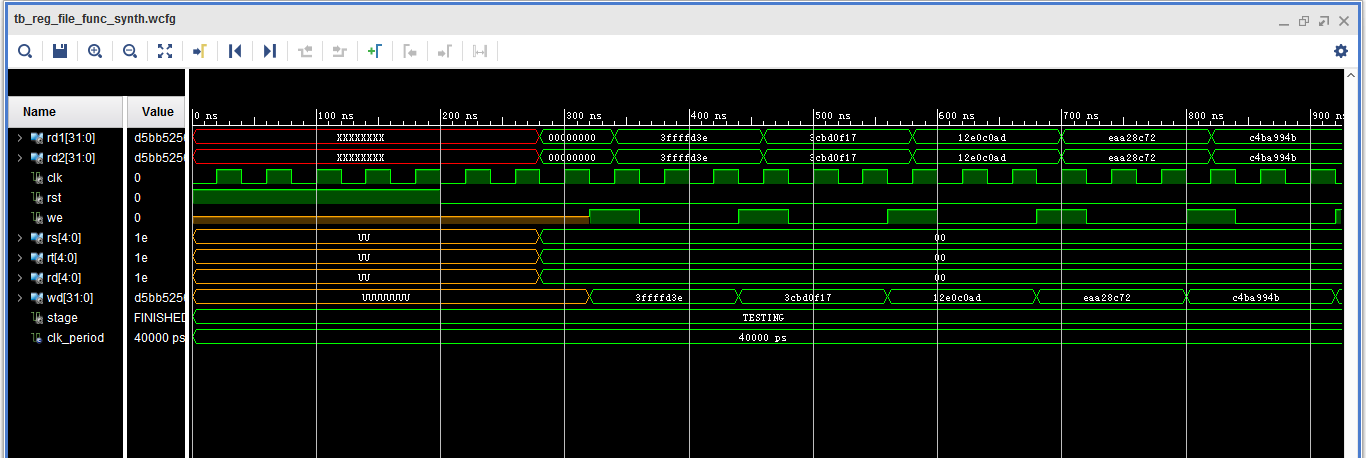


Figure 2. The first few cases

In each case, we give wd a random value, and set we signal to high. Then at the clock rising edge, rd1 and rd2 get the value which is equal to wd. (Notice that rs, rt and rd have the same value, so they point to the same address in RF)

(3)

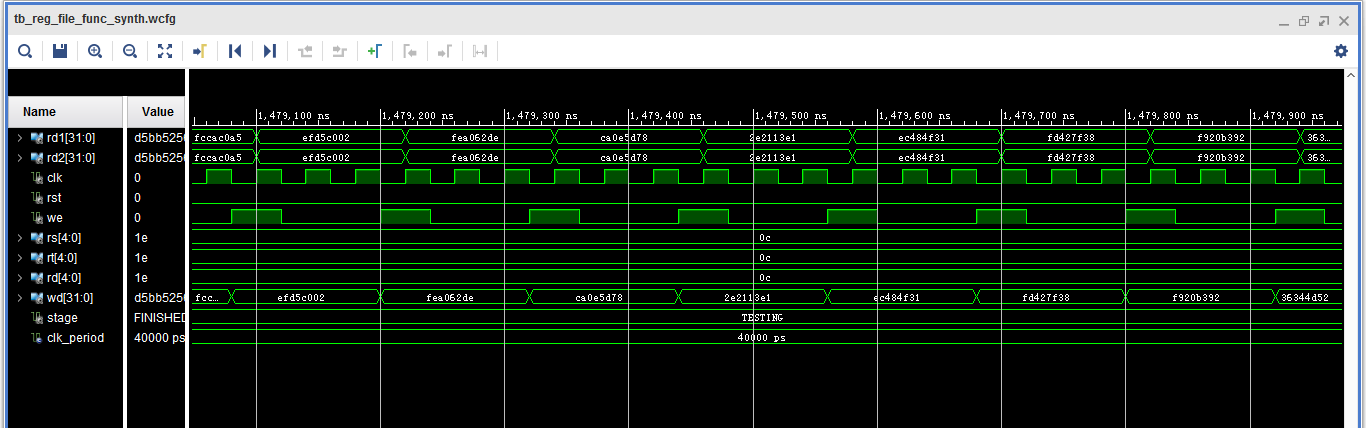


Figure 3. A few cases during the simulation

(4)

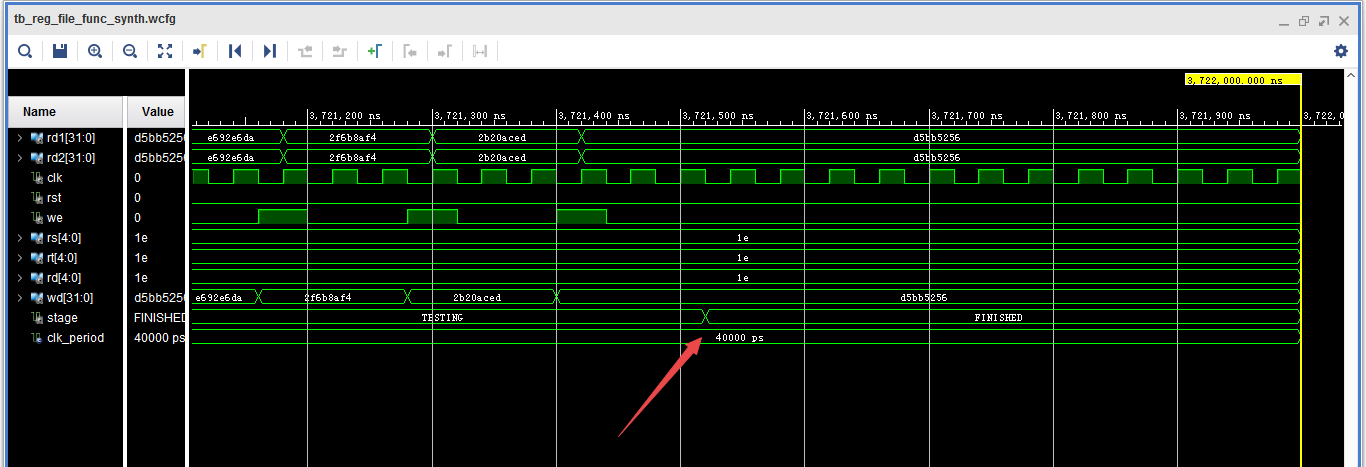


Figure 4. The last few cases.

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

(5)

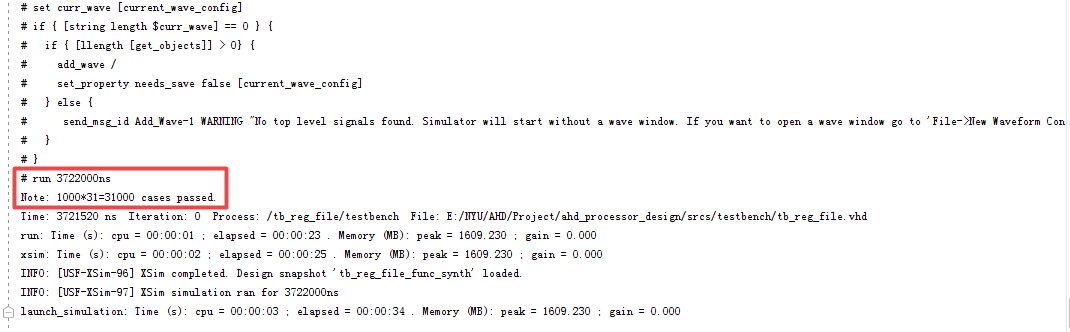


Figure 5. The report message after passing all the test cases.

At this time, if we check the report, we’ll find a “1000\*31=31000 cases passed” message. That means all our test cases succeed.

###### Timing Simulation

This time we will perform timing simulation on the same test bench.

(1)

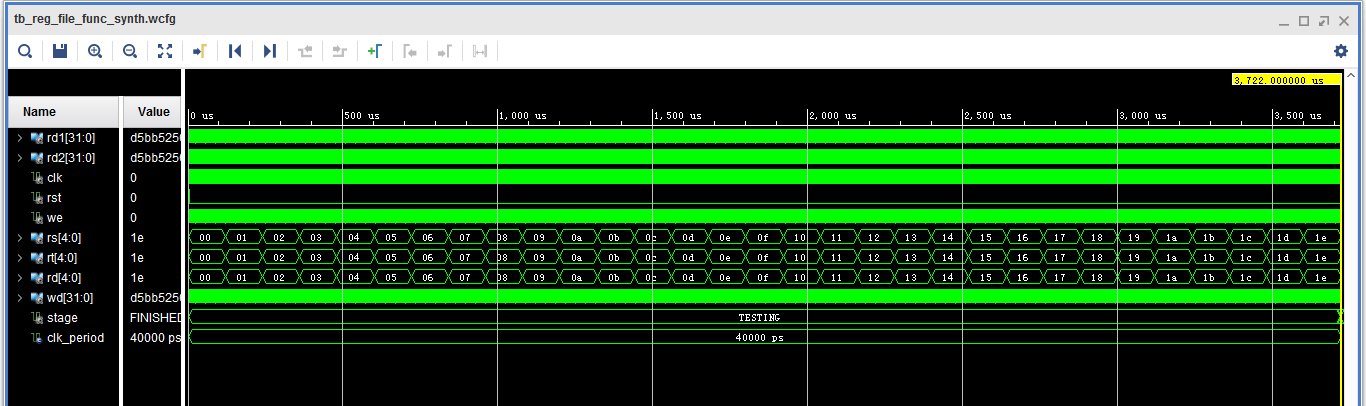


Figure 1. The whole simulation

These operations are the same as functional simulation.

(2)

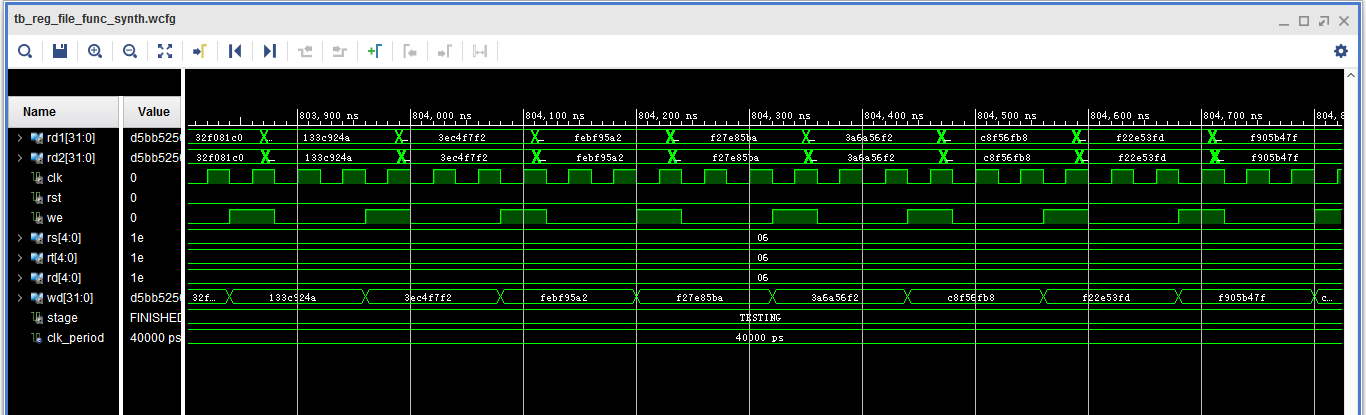


Figure 2. A few cases during the simulation

We can see that when we write different values to the register, it can be read properly.

(3)

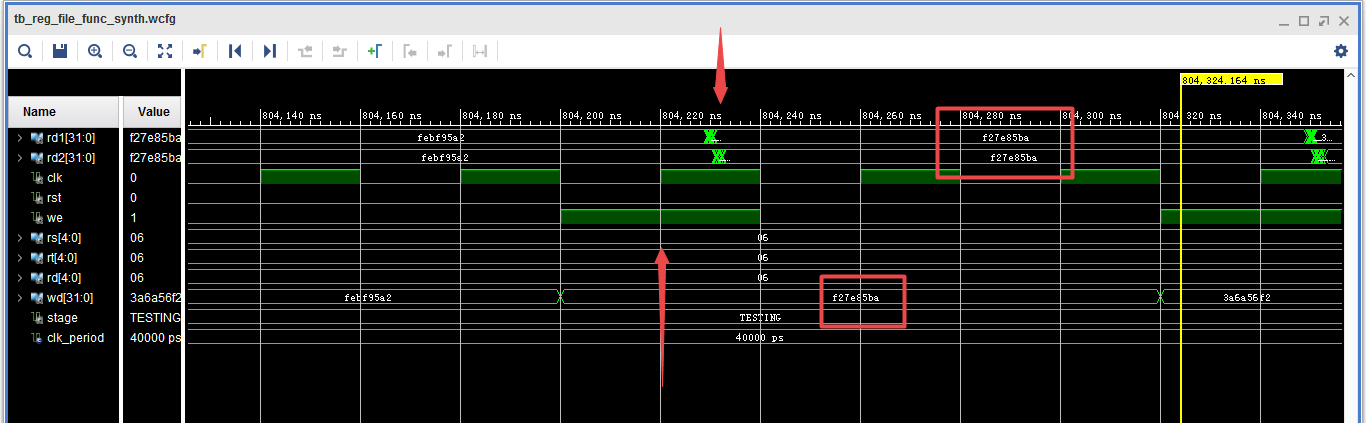


Figure 3. One case in the simulation

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.

(4)

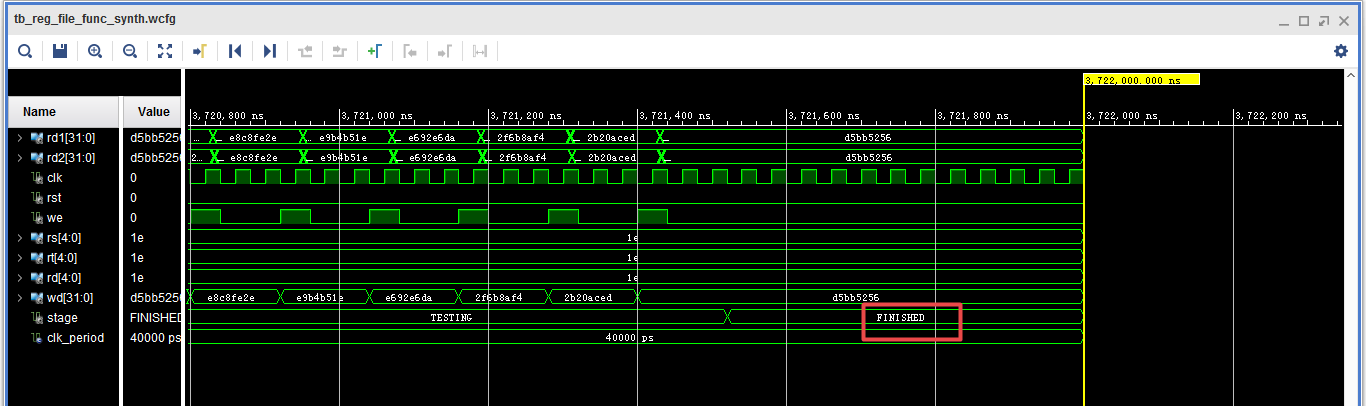


Figure 4. Last few cases

After all the cases finish, the state becomes “FINISHED”.

(5)

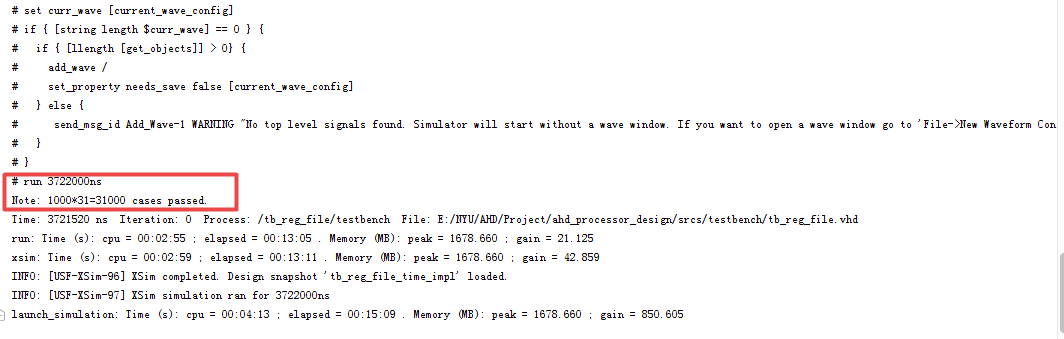
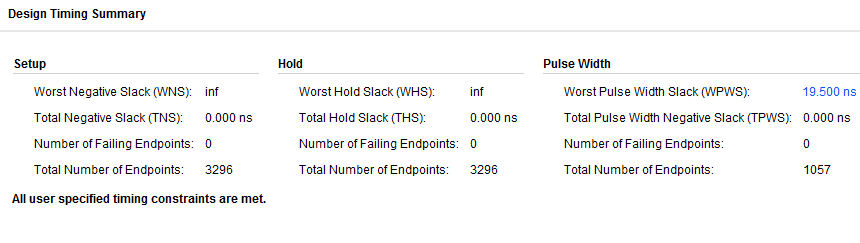


Figure 5. The report message

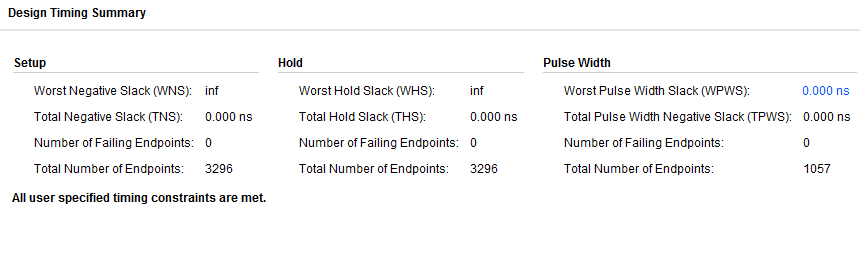
Still, when the simulation is done, a “1000\*31=31000 cases passed” message appears, which indicates the success of our test.

###### Timing Analysis

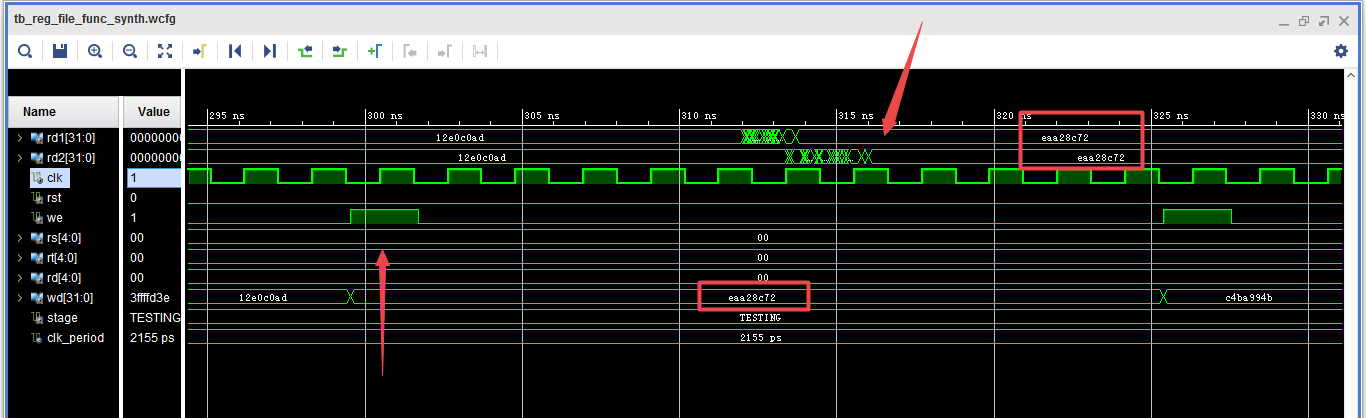
First we set clock period = 40ns and check the timing summary:



All timing constraints are met. The WPWS is positive, so we gradually decrease the clock period and check the timing summary, and finally when we set clock period = 2.155ns, we get this summary:



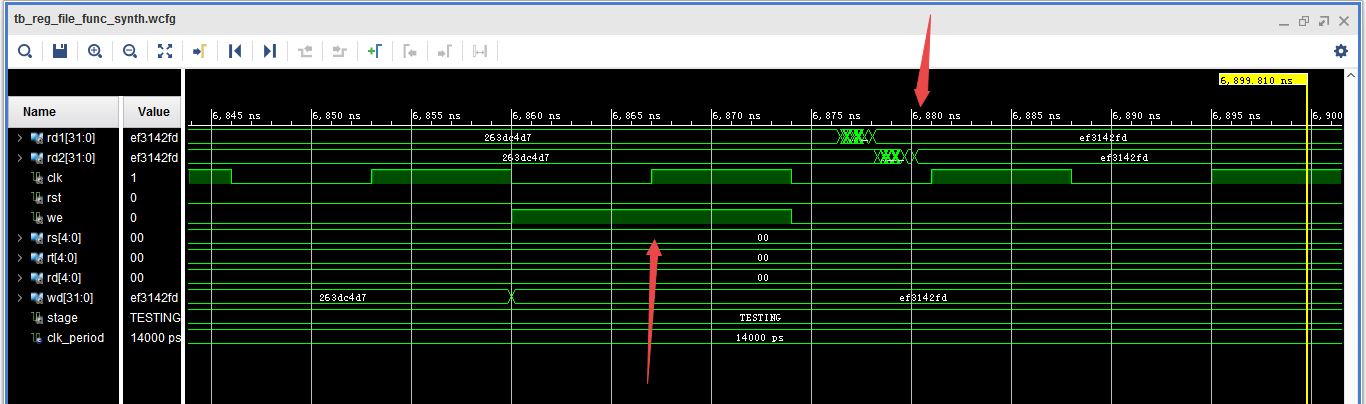
WPWS becomes 0ns. If we further decrease the clock period, it will fail.So we use 2.155ns as the clock period to run timing simulation, and we could still read the correct data which has been written into the EF.However, there’s a problem. Let us show the simulation waveform:



Although we could read the correct data, the delay between them is about 7.5 clock cycles (16ns). That means both write and read operation may take longer than one cycle to complete.

In our Top CPU, each instruction only takes one cycle to execute, thus if we write something into RF, it must be valid within one cycle, so that when the following instructions come, the RF could provide a correct value.

Since both write and read operations take time (due to the buffers), the write operation itself may take not as long as 16ns to complete. However, to ensure the function’s correctness, we choose to set clock period = write time + read time, which is around 16ns. Based on this value, we further tried several different clock period and run simulations. Finally, we choose clock period = 14ns. The result is showed below:



We could see this is just a safe value. So for the Reg\_File, we regard 14ns as the minimum clock period.

|  |  |
| --- | --- |
| Critical Path Delay | 14ns |
| Frequency | 71.43MHz |
| Latency | 1 cycle |
| Propagation Delay | 14ns |

This delay (14ns) is just for the single part Reg\_File. It indicates that the clock period in Top CPU can’t be shorter than 14ns. Actually, the clock period in top module usually takes a much longer value.

#### Control unit

###### Implementation

#### ctrl_unit.png

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.

in inst: 32-bit instruction

out lw: write back source selector signal for reg\_file

out sw: write-enable signal for data memory

out branch: branch indicator

out bnc\_type: branch type indicator

out jump: jump indicator

out funct: operation selector signal for ALU

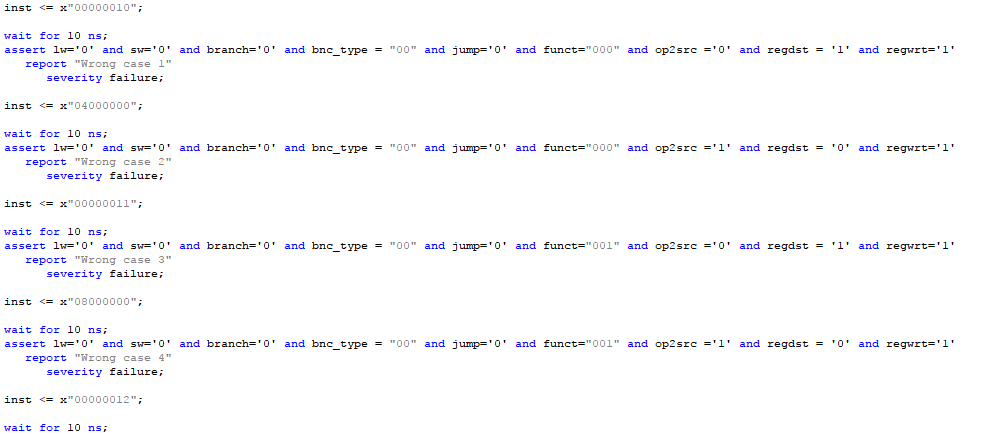
out op2src: op2 source selector signal for ALU

out regdst: register destination selector signal for reg\_file

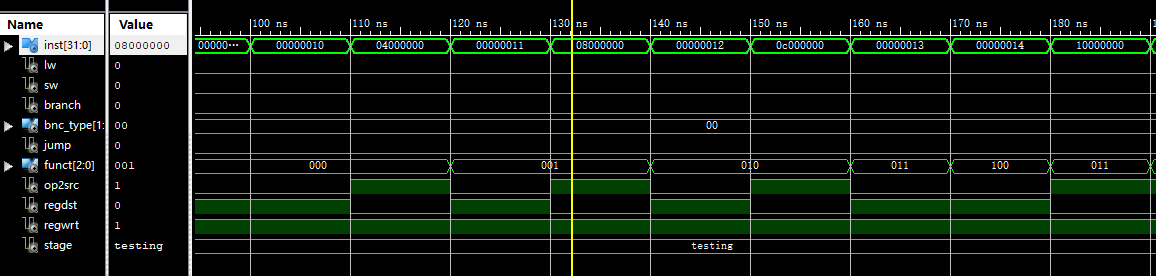
out regwrt: write-enable signal for file

###### Testbench

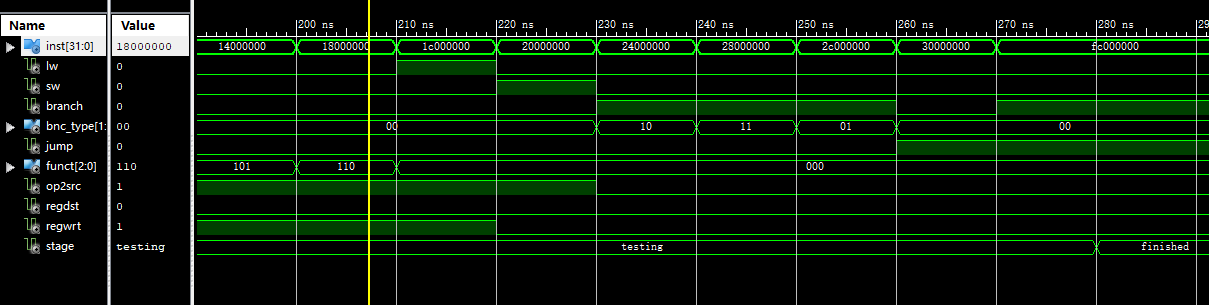
For each condition, I created one instruction to test correctness(bits unused is set to be zero) and used ‘assert’ statement to check the output automatically (see tb\_ctrl\_unit.vhd for details). The stage signal would change to FINISH only when all test cases passed, otherwise, the simulation would stop with severity level ‘failure’.



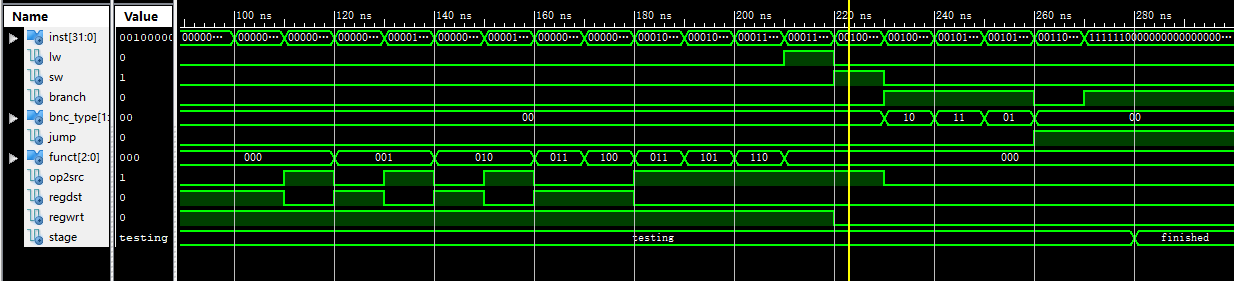
###### Functional Simulation



Testing first 9 conditions

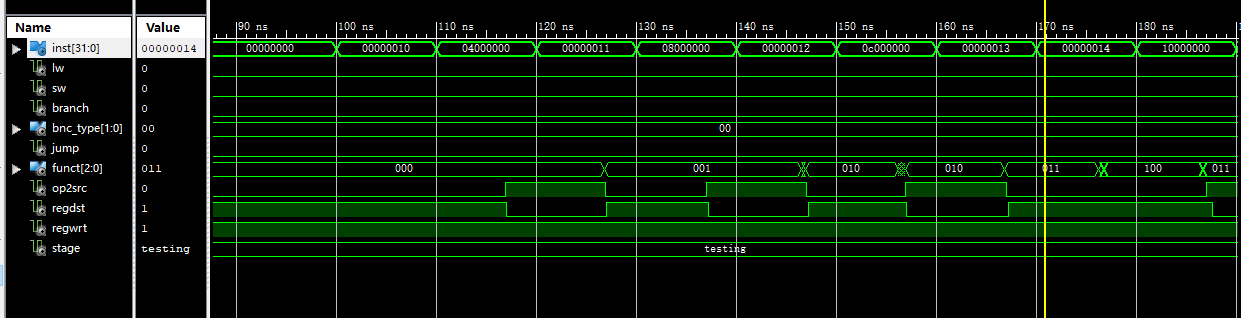


Testing latter 9 conditions

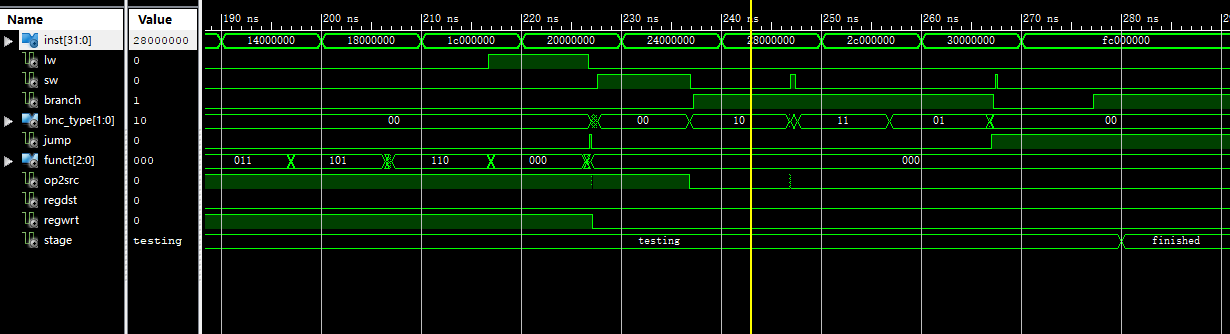


An overview with all cases passed.

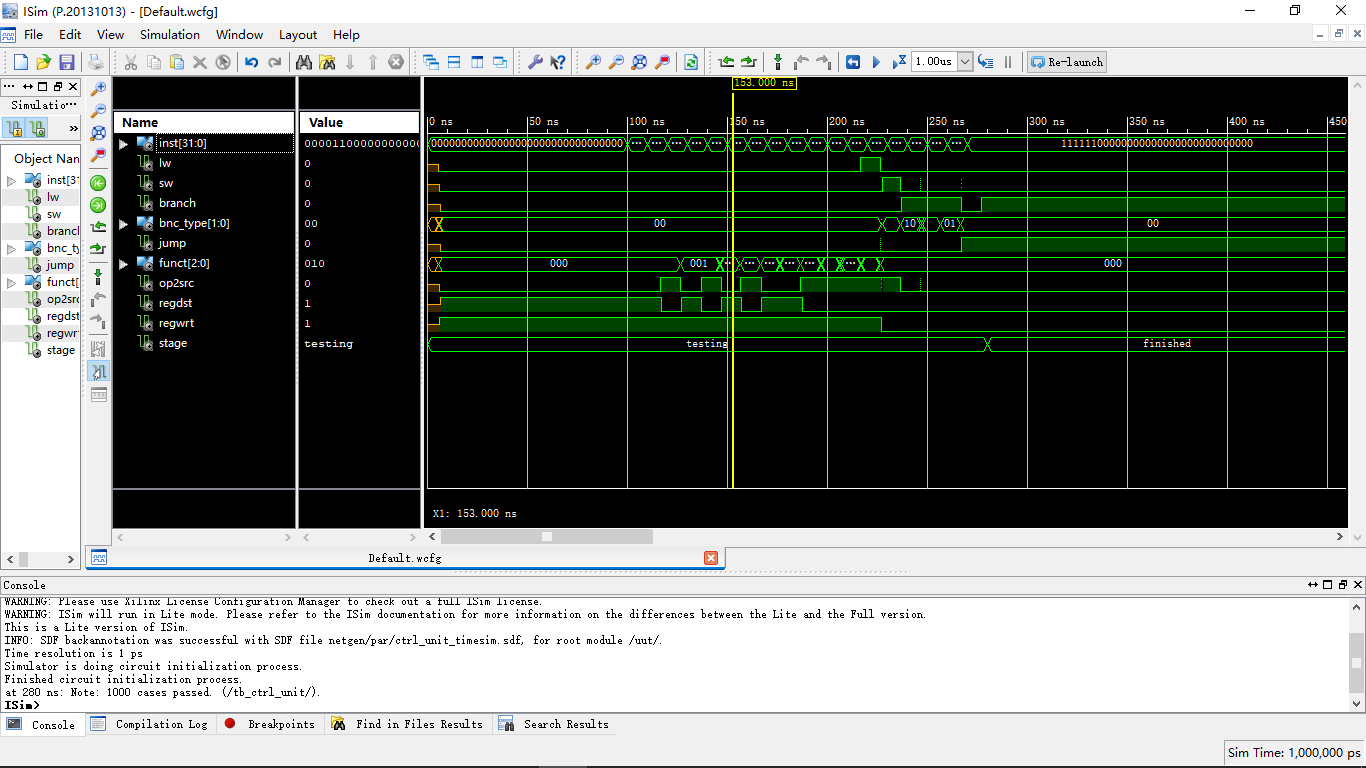
###### Timing Simulation



Testing first 9 conditions



Testing latter 9 conditions



An overview with all cases passed.

###### Timing Analysis

#### Data memory

###### Implementation

###### Testbench

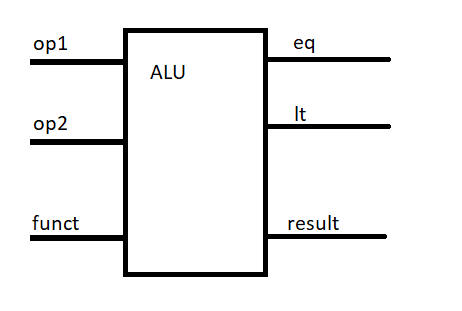
###### Functional Simulation

###### Timing Simulation

###### Timing Analysis

#### Alu

###### Implementation



Signal list:

in op1: operand1

in op2: operand2

in funct: function selection

out eq: HIGH when op1 == op2, otherwise LOW

out lt: HIGH when op1 < op2, otherwise LOW

out result: function result

Function:

0: do result = op1 + op2

1: do result = op1 - op2

2: do result = op1 & op2

3: do result = op1 | op2

4: do result = !(op1 | op2)

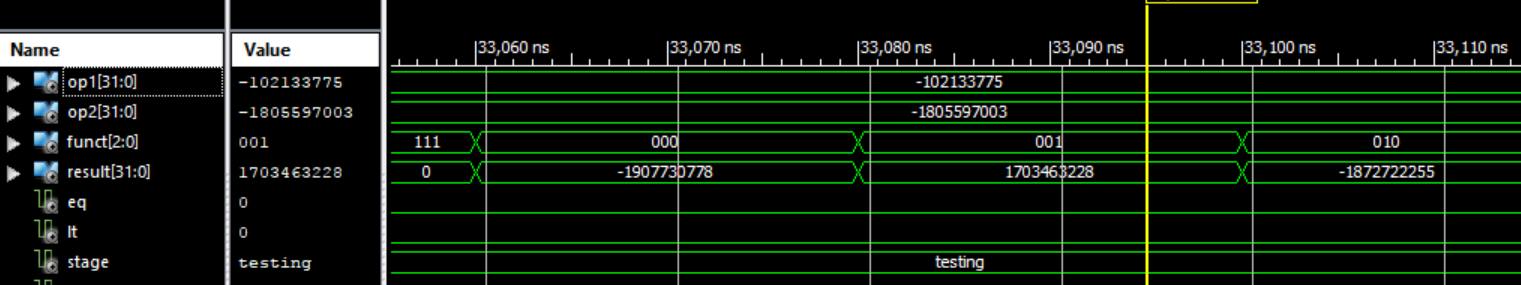
5: do result = op1 << op2

6: do result = op1 >> op2

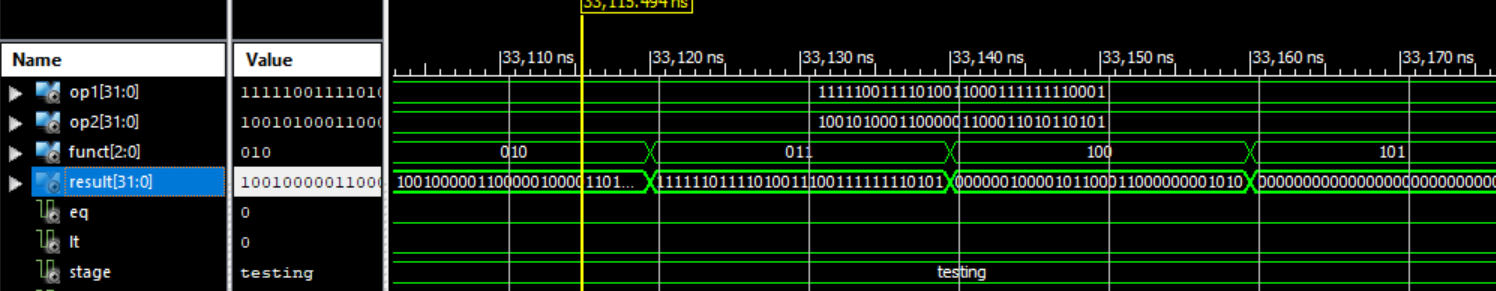
###### Testbench

For testbench, I tested the ALU unit on 1000 random op1 and op2 for all the functions and used ‘assert’ statement to check the status automatically. The testbench would finish and show “1000 cases passed” only if all cases passed.

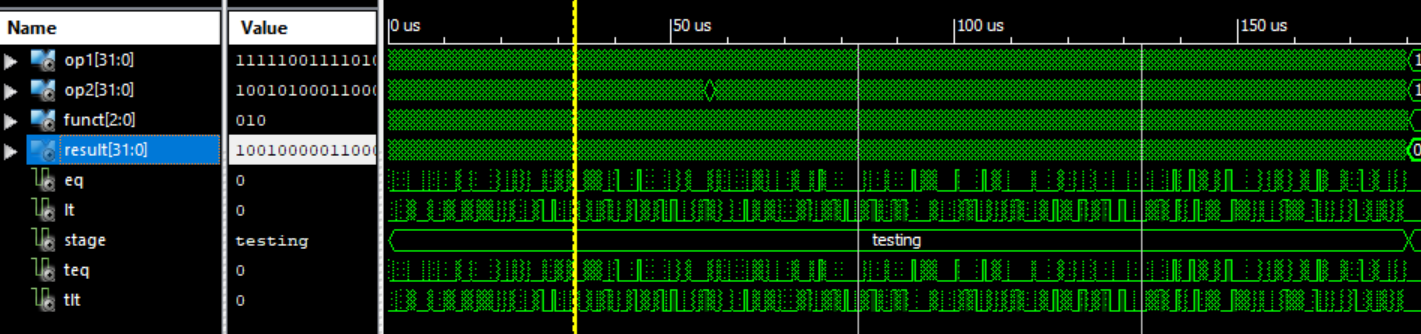
###### Functional Simulation



case “+” and “-” showed on decimal

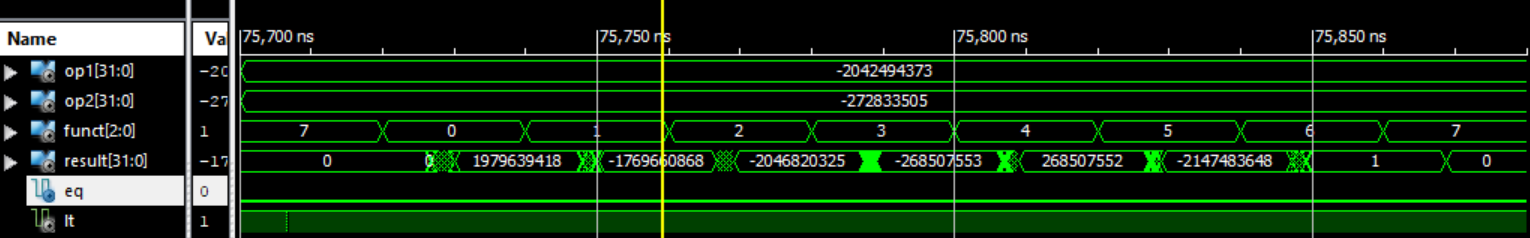


bit operation case showed on bits

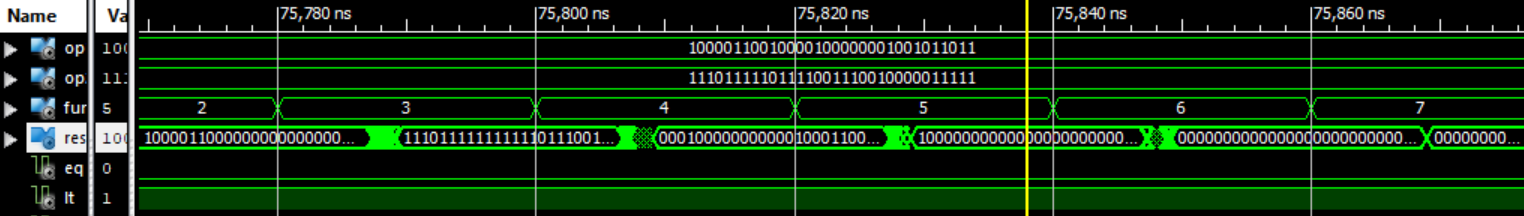


An overview. All cases passed

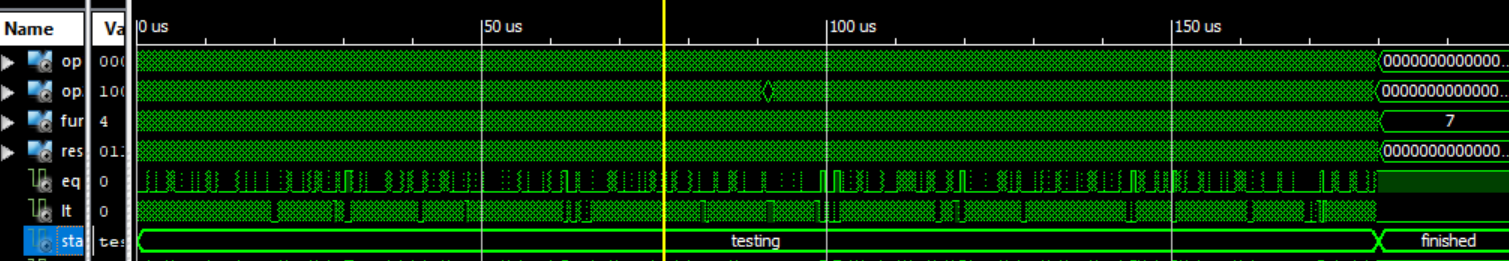
###### Timing Simulation



case “+” and “-” showed on decimal



bit operation case showed on bits



Overview. All cases passed

###### Timing Analysis

|  |  |
| --- | --- |
| Critical path delay | 10.4 ns |
| Highest frequency | 96.2 MHz |

#### Debouncer

###### Implementation

#### Seg\_led

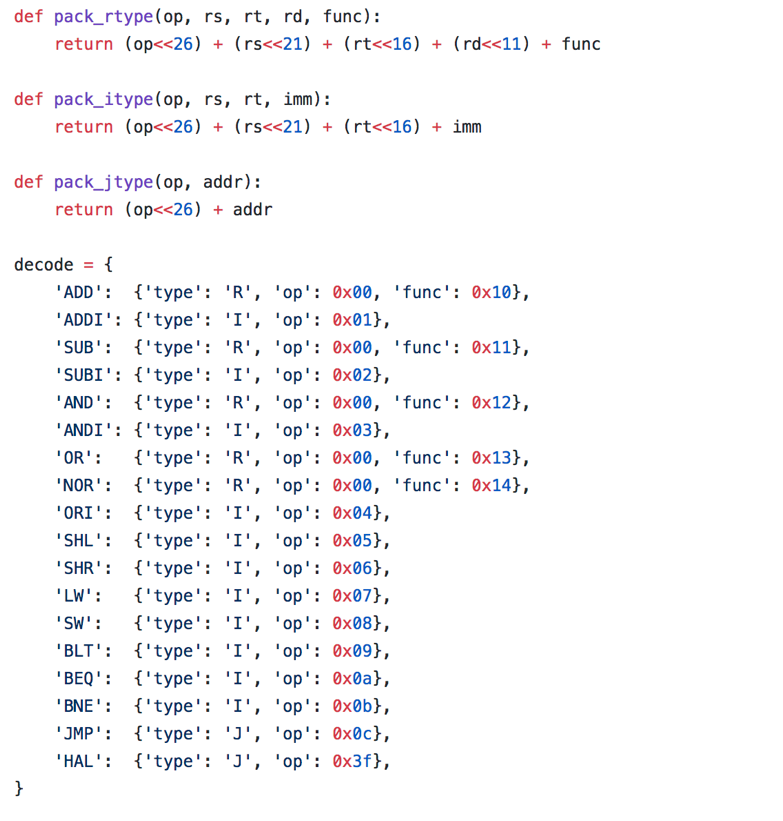
###### Implementation

## Processor Desgin and test

#### Assmebly comiler

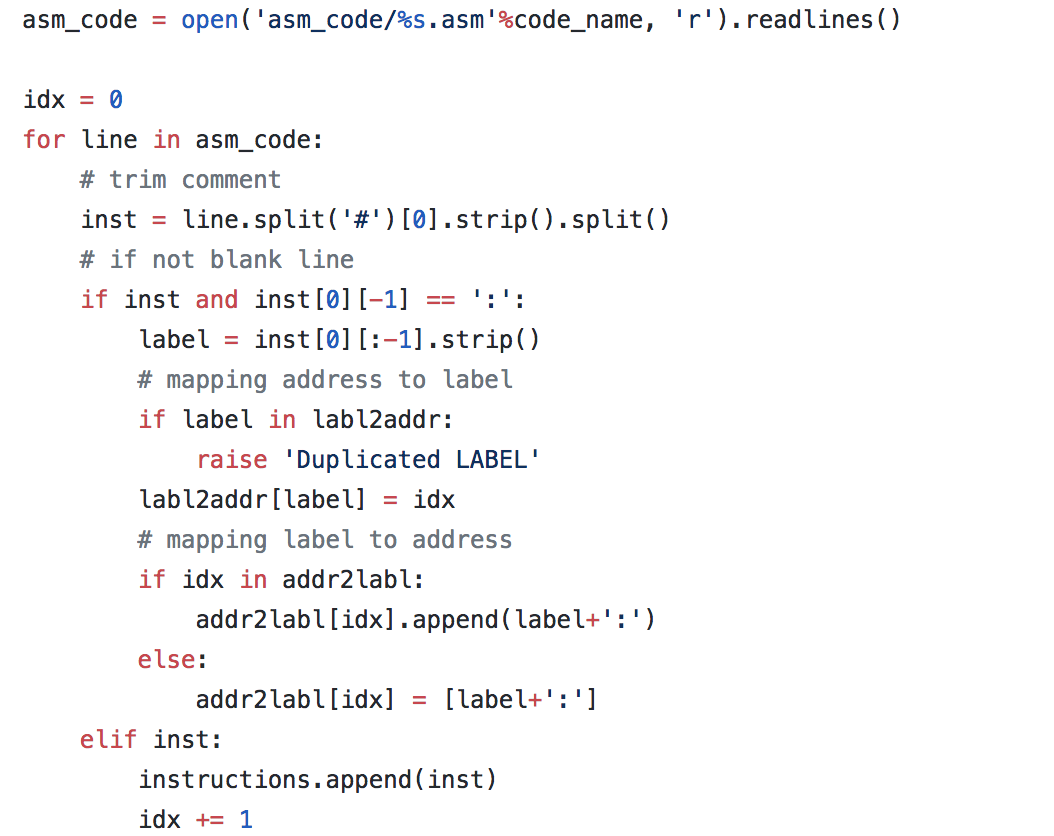
We wrote the RC5 code in assembly language (rc5.asm & rc5\_optimized.asm) to give instructions of how the MIPS workd. We also wrote a simple compiler in python (load\_tb\_instructions.py) to help convert assembly to machine code (rc5.binary & rc5\_optimized.binary) , which can be conducted directly on the FPGA board.

In our python, first, we define three functions seperately for I, R and J type to use bit shifting to combine the input such as op, rs, rt, rd, imm and addr to its machine code format. Then, in an Array, we define the details about type, opcode and funcode of each of the needed instructions as dictionary.



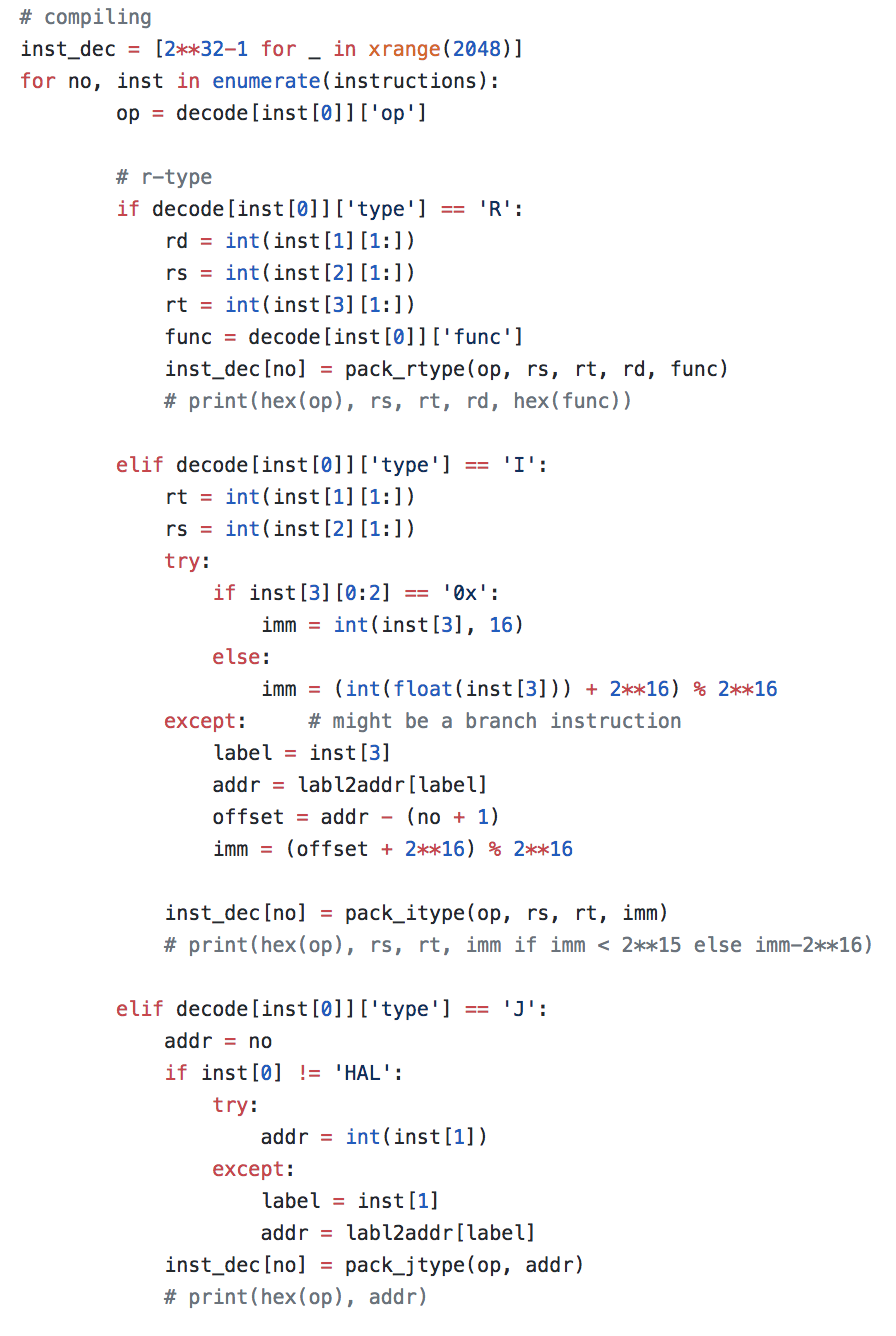
divide instruction types

Then, we use python File readline fuction to get the assembly instructions.



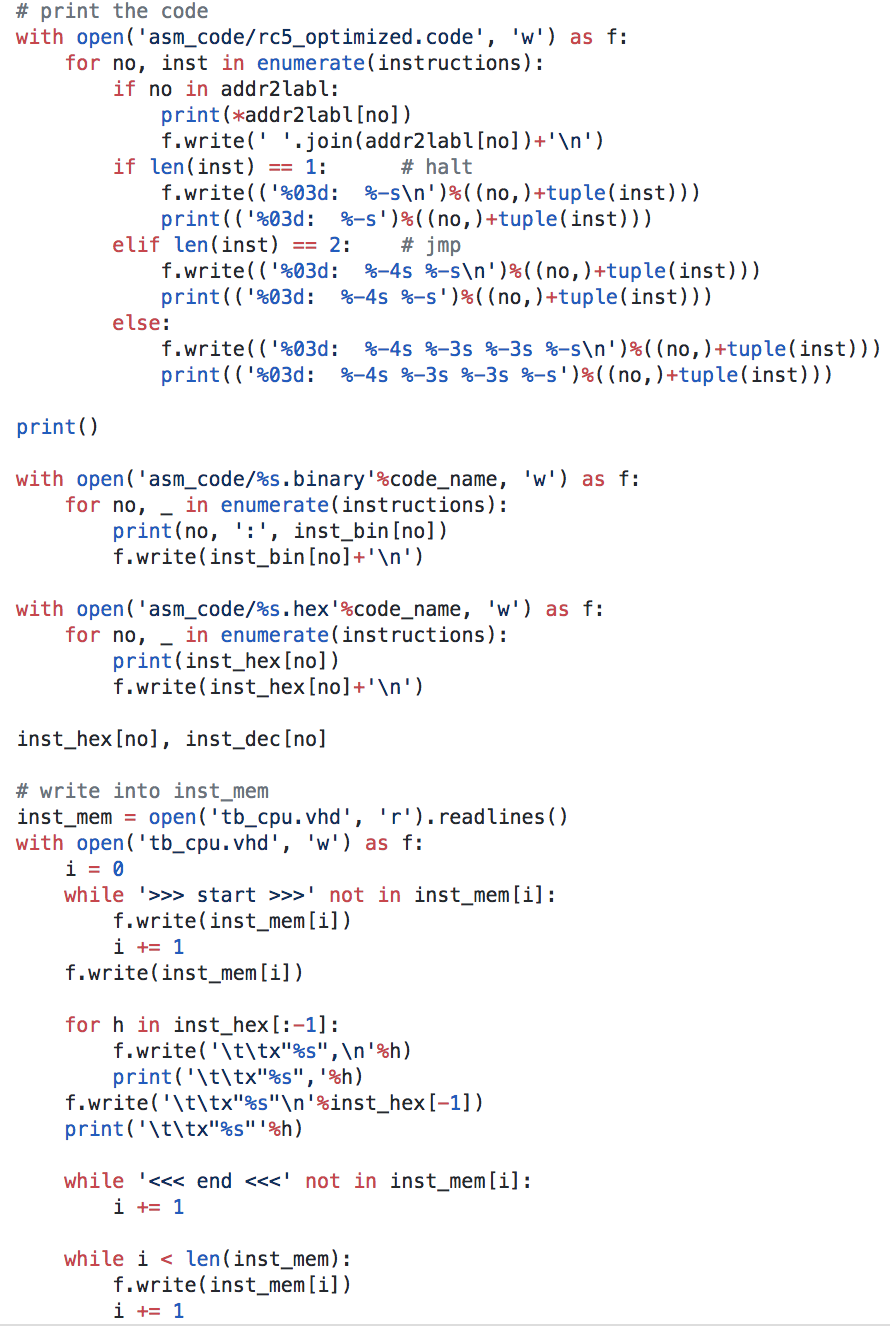
read input stream

Sequentially, for each current instruction, we split it by language pattern.



judge the instruction type

Then, find out what type it is and call the according function for the machine code. At last, just write these machine code into the inst\_mem part of the tb\_cpu.vhd.



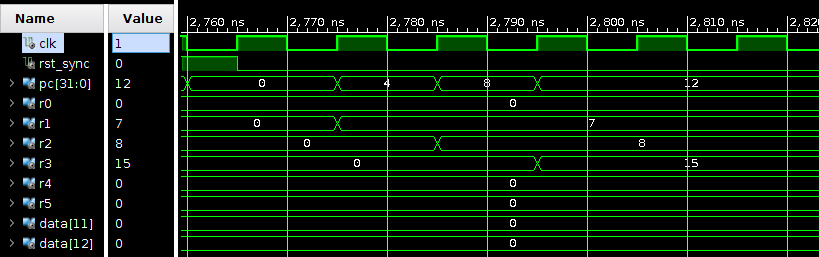
write output into the right place into inst\_mem of tb\_cpu.vhd

#### Correctness of the Processor Design

The processor is tested with the sample code provided on the project description. The results are as follows:

Sample code 1:

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



Here, the instruction is to add the immediate value 7 and R0 and store it in R1. As we can see from the above screenshot, R1 is 7. Similarly, R2 is 8. Then we add the value of R1 and R2 and store the result in R3. It’s easy to get that R3 is 15. The result we get is the same as the hand calculation.

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)

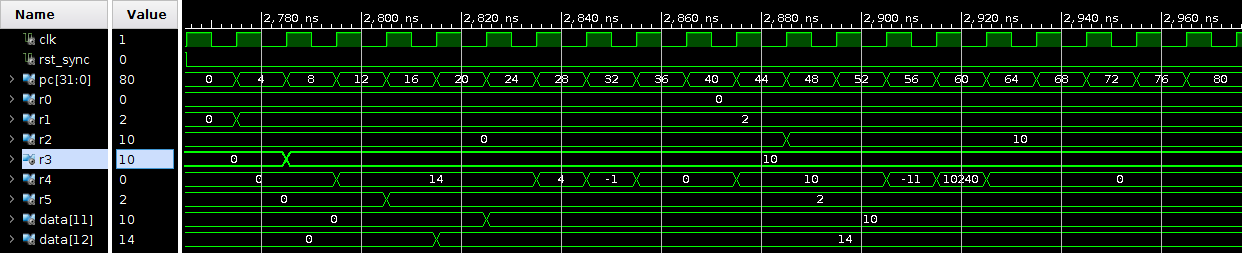
001010 00000 00101 1111111111111110 --BEQ R5, R0, -2

001001 00100 00101 0000000000000000 --BLT R5, R4, 0

001011 00100 00101 0000000000000000 --BNE R5, R4, 0

001100 00000000000000000000010100 --JMP 20

111111 00000000000000000000000000 --HAL



(1) ADDI R1, R0, 2 //R1=R0+2(decimal)

The instruction is to add the immediate value 2 and R0 and store it in R1. So as seen from the above screenshot, R1 is 2 after the instruction.

(2) ADDI R3, R0, 10 //R3=R0+10(decimal)

The instruction is to add the immediate value 10 and R0 and store it in R3. So as seen from the above screenshot, R3 is 10 after the first instruction.

(3) ADDI R4, R0, 14 //R4=R0+14(decimal)

The instruction is to add the immediate value 14 and R0 and store it in R4. So as seen from the above screenshot, R4 is 14 after the instruction.

(4) ADDI R5, R0, 2 //R5=R0+2

The instruction is to add the immediate value 2 and R0 and store it in R5. So as seen from the above screenshot, R5 is 2 after the instruction.

(5) SW R4, 2(R3) //Mem[R3+2]=R4

The instruction is to store the word from the memory location R3+2. So as seen from the above screen shot, R3 is 10, Mem[12] = 14, so R4 is 14 after the instruction.

(6) SW R3, 1(R3) //Mem[R3+1]=R3

This instruction is to store the word from the memory location R3+1. R3 is 10, Mem[11] is 10. So after the instruction, the value of R3 is still 10.

(7) SUB R4, R4, R3 //R4=R4-R3

The instruction is to subtract R3 from R4 and store in R4. R4 is 14, R3 is 10, so R4 will be 4 after this instruction, which matches the result in the screenshot.

(8) SUBI R4, R0, 1 //R4=R0-1(decimal)

The instruction is to subtract R0 from the immediate value 1 and store in R4. R0 is 0, so after the instruction, R4 will change to -1.

(9) AND R4, R2, R3 //R4=R2 and R3

The instruction is to AND R2 and R3 and store in R4. R2 is 0, R3 is 10, after this instruction, R4 is 0.

(10) ANDI R4, R2, 10 //R4=R2 and 10(decimal)

The instruction is to AND Immediate R2 and the immediate value 10 and store it in R4. As R2 is 0, the result in R4 is also 0.

(11) OR R4, R2, R3 //R4= R2 or R3

The instruction is to OR R2 and R3 and store the result in R4. R2 is 0, but R3 is 10, so the result is 10. As we can see from the screenshot, the result in R4 is 10.

(12) LW R2, 1(R3) //R2=Mem[1+R3]

The instruction is to Load the Data from the memory location [1+R3] and store it in the R2. R3 is 10, Mem[11] is 10, so R2 change to 10 after this instruction.

(13) ORI R4, R2, 10 //R4=R2 or 10(decimal)

The instruction is to OR R2 and the immediate value 10 and store in the register R4. R2 is 10, so R4 is still 10, the value doesn’t change.

(14) NOR R4, R2, R3 //R4= R2 nor R3

The instruction is to implement NOR on R2 and R3 and store the result in R4. R2 is 10 and R3 is 10, R4 should be -11. (01010 nor 01010 = 10101)

(15) SHL R4, R2, 10 //R4= R2 << 10(decimal)

The instruction performs a logical left shift, left shift R2 by 10 bits, R2 is 10, after the instruction, the result should be 10240.

(16) SHR R4, R2, 10 //R4=R2 >> 10(decimal)

The instruction performs a logical left shift, left shift R2 by 10 bits, R2 is 10, after the instruction, the result should be 0.

(17) BEQ R5, R0, -2

The instruction “BEQ” compares the branch for equality, if R5 equals to R0, it will go backward 2 instructions. In the given instruction, it checks if R0 is equal to R5, which is false.

(18) BLT R5, R4, 0

The instruction “BLT” checks if R5 is less than R4. R5 is 2 and R4 is 0. So it R5 is not less than R4.

(19) BNE R5, R4, 0

The instruction “BNE” compares the branch for equality. R5 is 2 and R4 is 0, which is not equal. But as the offset is 0, it will do the next instruction.

(20) JMP 20

Jump instruction is used to jump to the given address. In this case, it will jump to itself.

(21) HAL

HAL is used to stop the program.

As we can see from the screenshot and the hand calculation, our processor can correctly do the job. The results of the screenshot and the hand calculation matches.

#### Functional simulation

#### Timing simulation

#### Analysis

# RC5 Implementation

## Assmebly code

#### FSM diagram

#### Time complexity analysis

#### XOR optimization

#### ROTATE optimization

## Testbench

## Functional simulation

## Timing simulation

# Summary