PROJECT REPORT

**Submitted By:**

Tianyu Gu tg1553

Fengyang Jiang fj483

Yingqi Huang yh1990

Junlun Xiao jx755

Yiren Dai yd1257

Lin Lu ll3374

NEW YORK UNIVERSITY

TANDON SCHOOL OF

ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

# 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

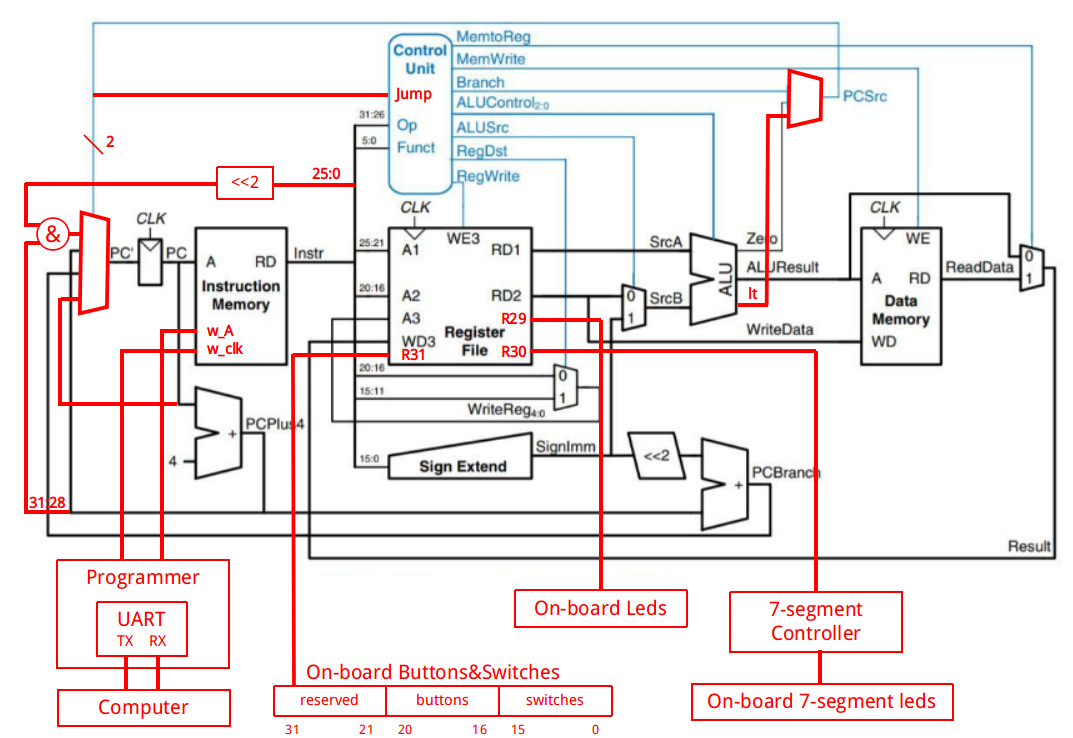


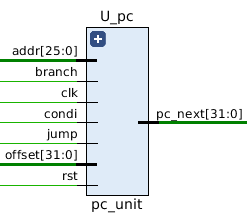
Figure 2.1

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 of PC



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’.

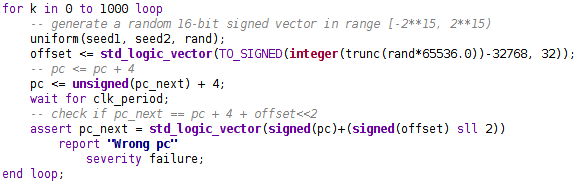


Figure 2.2 This is an example code of checking the branch condition

#### Functional simulation

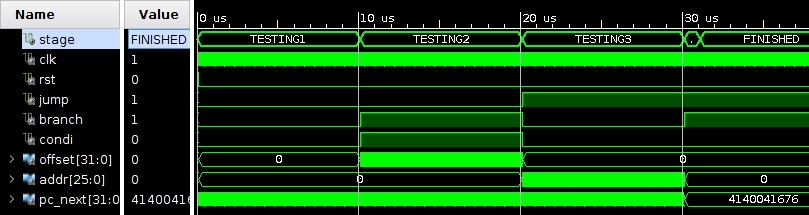


Figure 2.3 An overview. All cases passed.

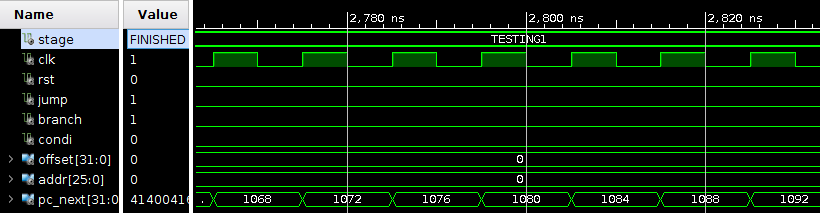


Figure 2.4 Testing continuously increment condition.

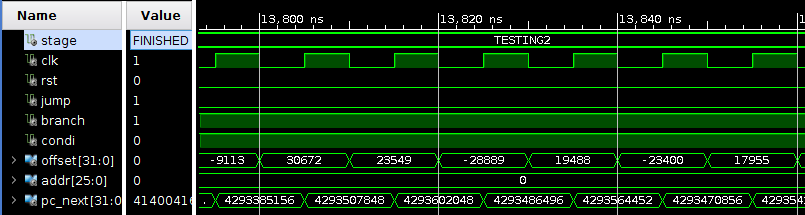


Figure 2.5 Testing branch condition.

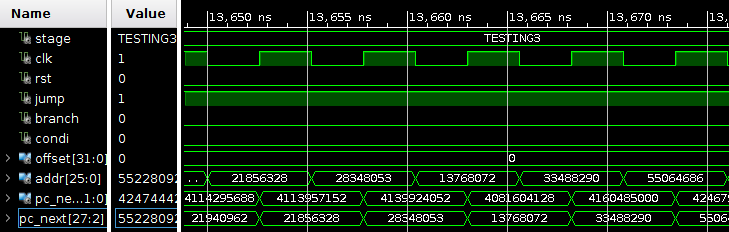


Figure 1.3.4 Testing jump condition

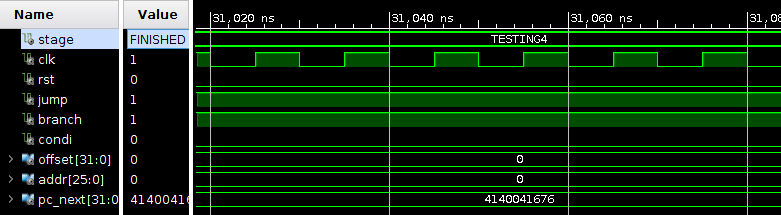


Figure 1.3.5 Testing halt condition

#### Timing simulation

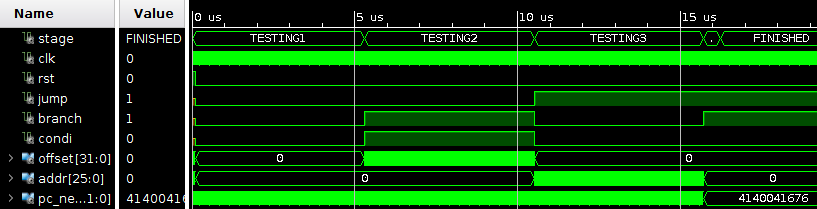


Figure 1.3.1 An overview. All cases passed.

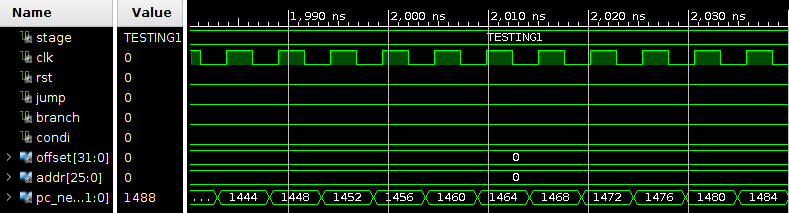


Figure 1.3.2 Testing continuously increment condition.

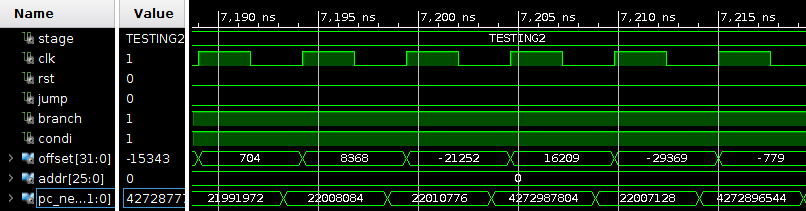


Figure 1.3.3 Testing branch condition

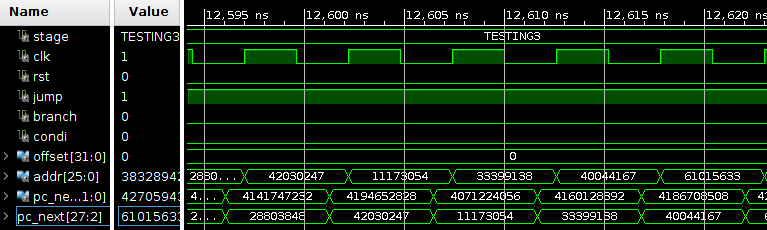


Figure 1.3.4 Testing jump condition

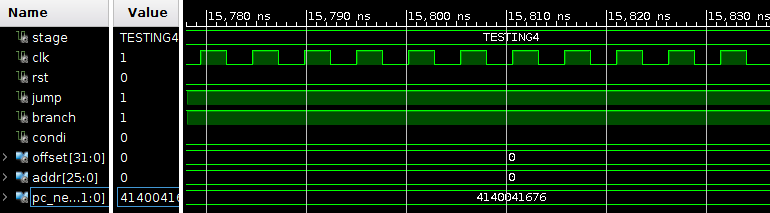
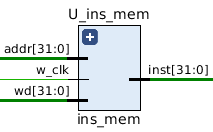


Figure 1.3.5 Testing halt condition

#### Instruction memory

###### Implementation of INS\_MEM



Signal list:

in addr: 32-bit address

in w\_clk: write clock signal

in wd: 32-bits value we want to write in the instruction memory

out inst: the result 32-bit instruction.

Each instruction has its own address, marked by a 32-bit program counter. When a given instruction is needed, the Instruction Memory delivers a 32-bit 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.

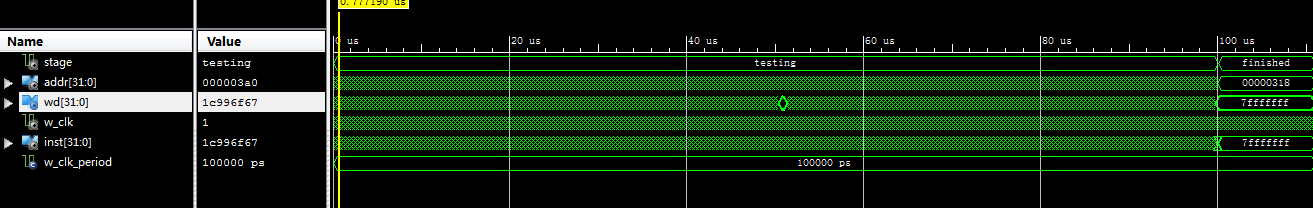
• At the rising edge of w\_clk, we write the value of the wd into instruction memory. In this way, it can support changing the program while our processor is running on the FPGA.

###### Testbench

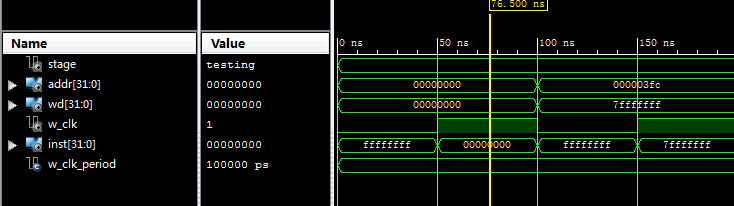
Randomly generate 1000 test cases using the same random vector generator in 2.2.1.2.

* Generate random vector to change addr and random variable to change wd
* When stage equals to finished, all the test case passed.

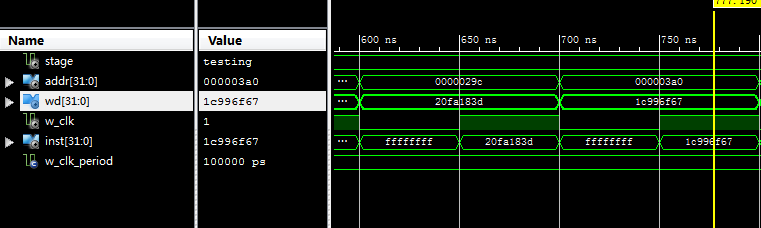
###### Functional Simulation



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

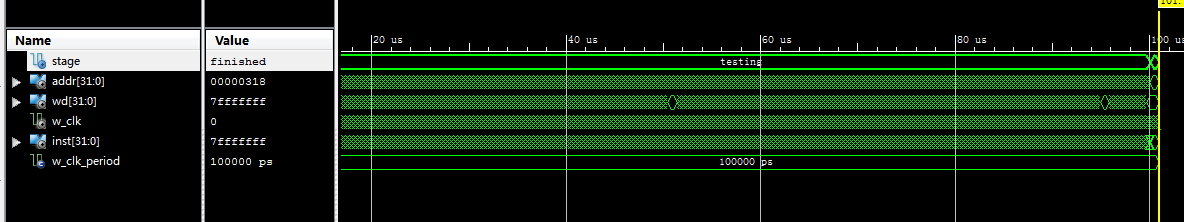


example1: the output instruction is the same as the wd value we wrote in.

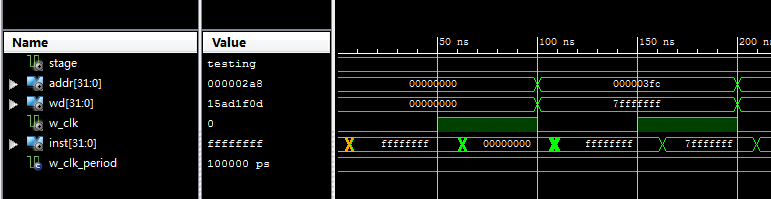


example2: the output instruction is the same as the wd value we wrote in.

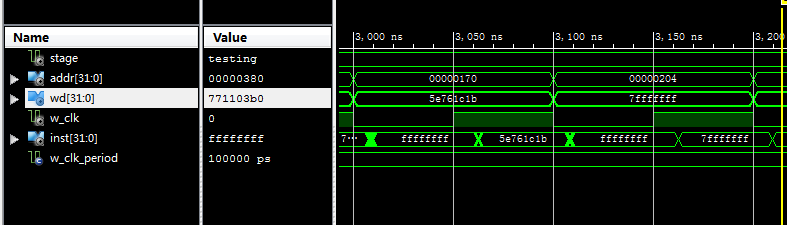
###### Timing Simulation



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



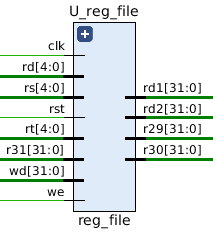
timing simulation for first example.



timing simulation for second example.

#### Register file

###### Implementation



Signal list:

in clk: clock signal

in rst: reset signal

in rs, rt, rd: 5-bit address

in wd: 32-bit data that should be written into reg(rd)

in we: write-enable signal, active-high

in r31: input interface for BTN & SW; read-only

out r29: output interface for LED Display

out r30: output interface for 7-Segment Display

out rd1, rd2: 32-bit 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 = REG[rs]; rd2 = REG[rt]; r29 = REG[29]; r30 = REG[30]
* Synchronous write: REG[rd] = wd @ rising\_edge(clk)
* Input/Output interface: r31, r29 and r30

###### Testbench

For each register, we generated 1000 random vectors to do write operation, and then check if the read values are equal to the written values in all the 1000 cases. So there are totally 31\*1000 = 31000 cases (r31 is read-only) being checked in this testbench. Assertion statement was used to check the output automatically. If anything goes wrong, the simulation will stop and raise a ‘failure’ exception. If all cases pass, stage signal will be set to “Finished”.

###### 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

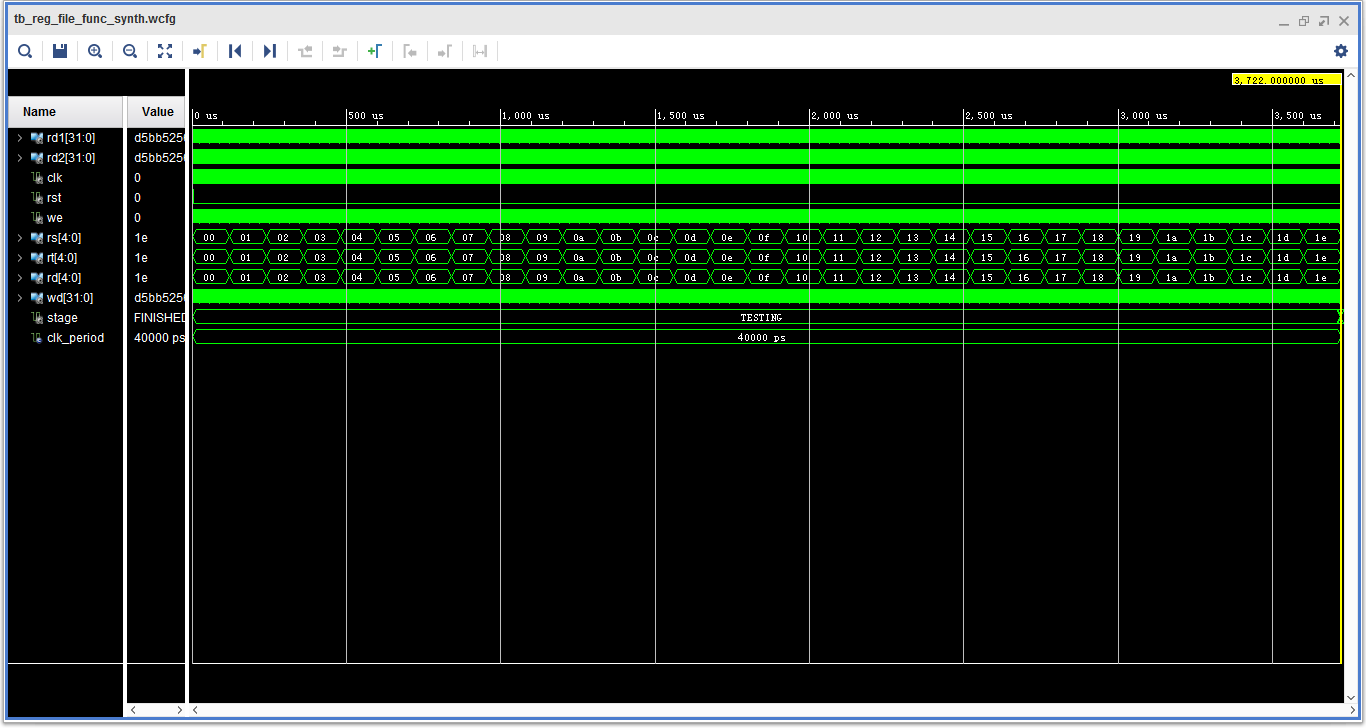


Figure 1. The whole simulation.

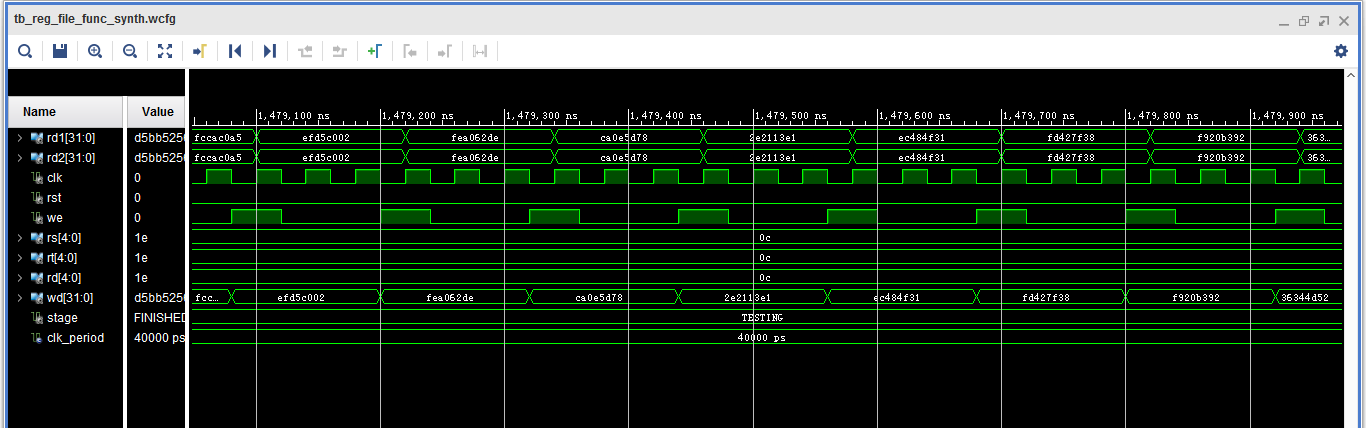


Figure 3. Some cases in detail. We can see that rd = wd @ risingede(clk)&we=1

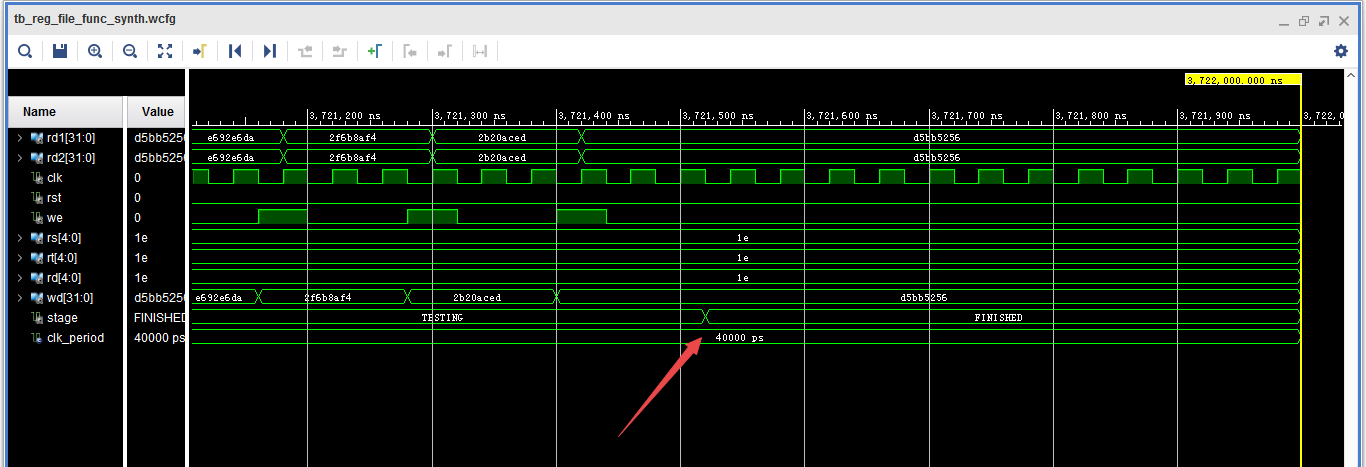


Figure 4. The last few cases. Stage signal changes to FINISHED when all cases have passed.

###### Timing Simulation

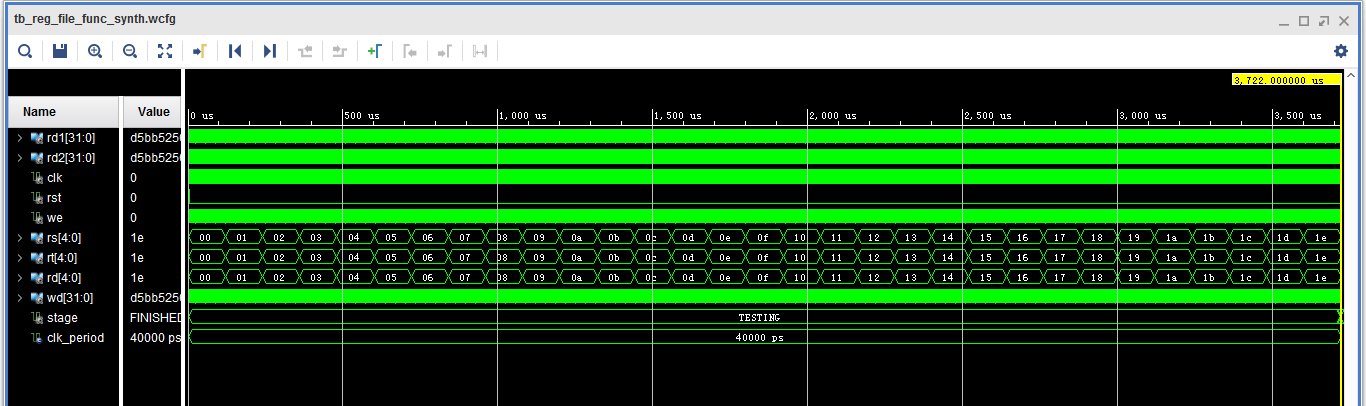


Figure 1. The whole simulation

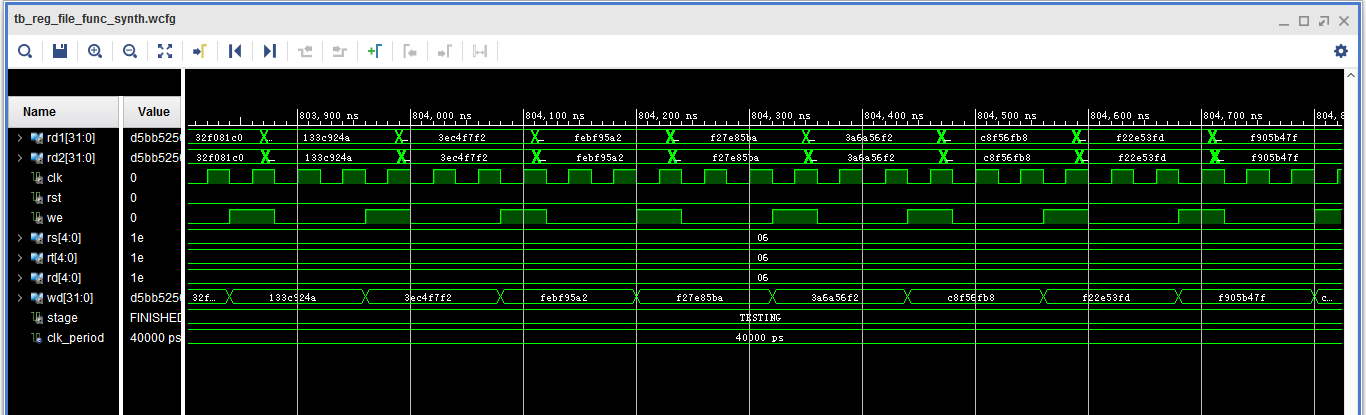


Figure 2. A few cases during the simulation

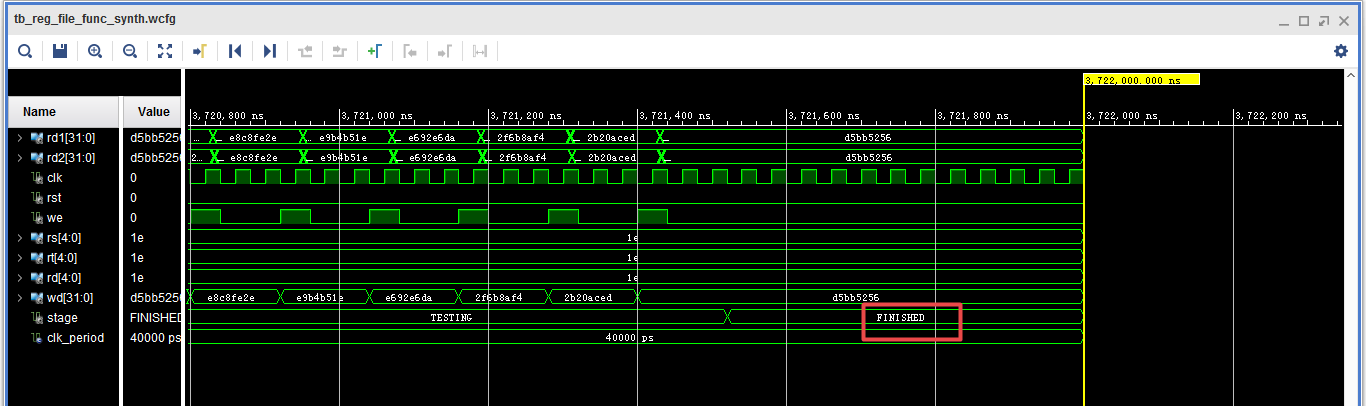


Figure 4. Last few cases

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

#### 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

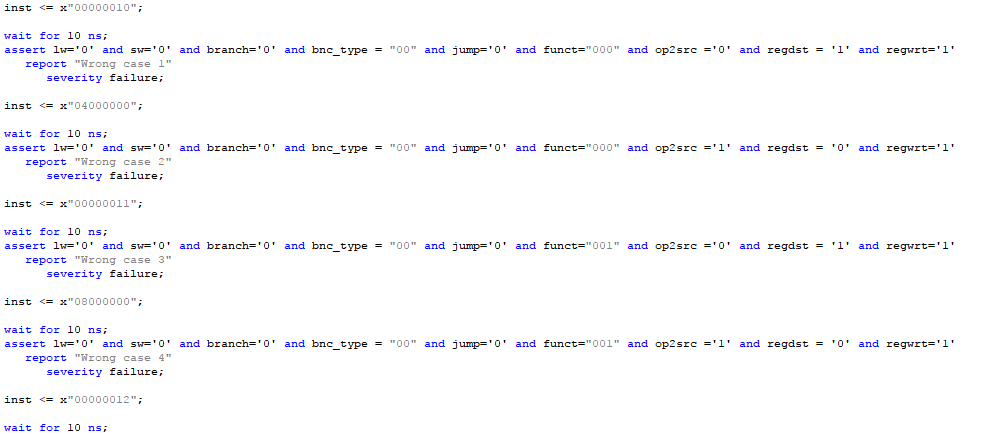
out op2src: op2 source selector signal for ALU

out regdst: register destination selector signal for reg\_file

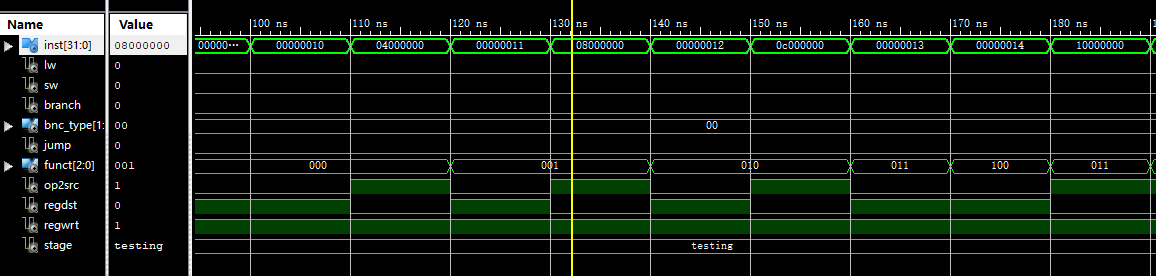
out regwrt: write-enable signal for file

###### Testbench

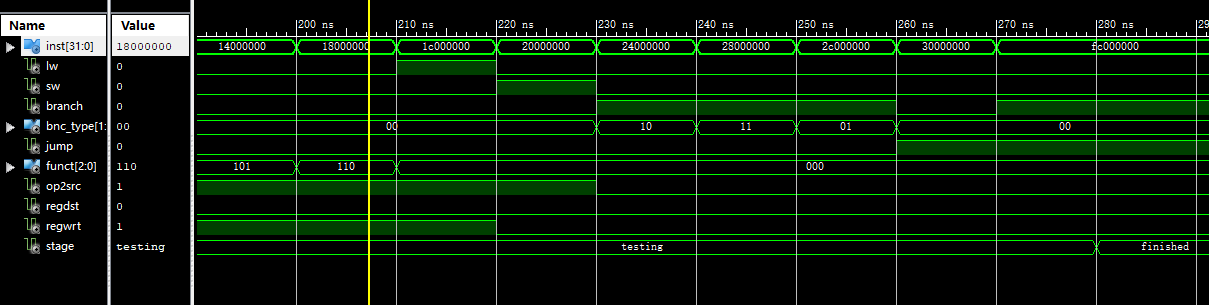
For each condition, I created one instruction to test correctness(bits unused is set to be zero) and used ‘assert’ statement to check the output automatically (see tb\_ctrl\_unit.vhd for details). The stage signal would change to FINISH only when all test cases passed, otherwise, the simulation would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



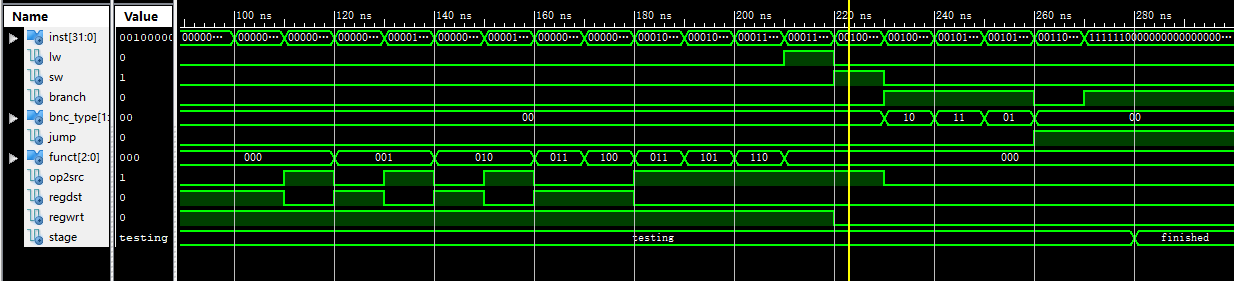
###### Functional Simulation



Testing first 9 conditions

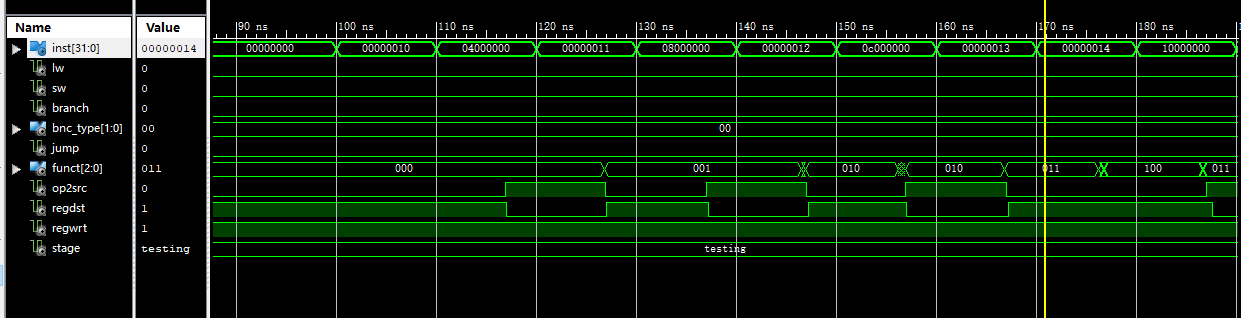


Testing latter 9 conditions

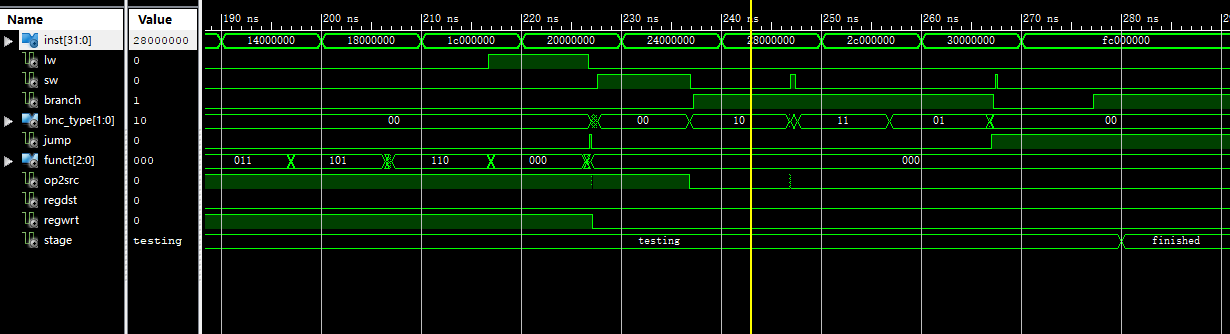


An overview with all cases passed.

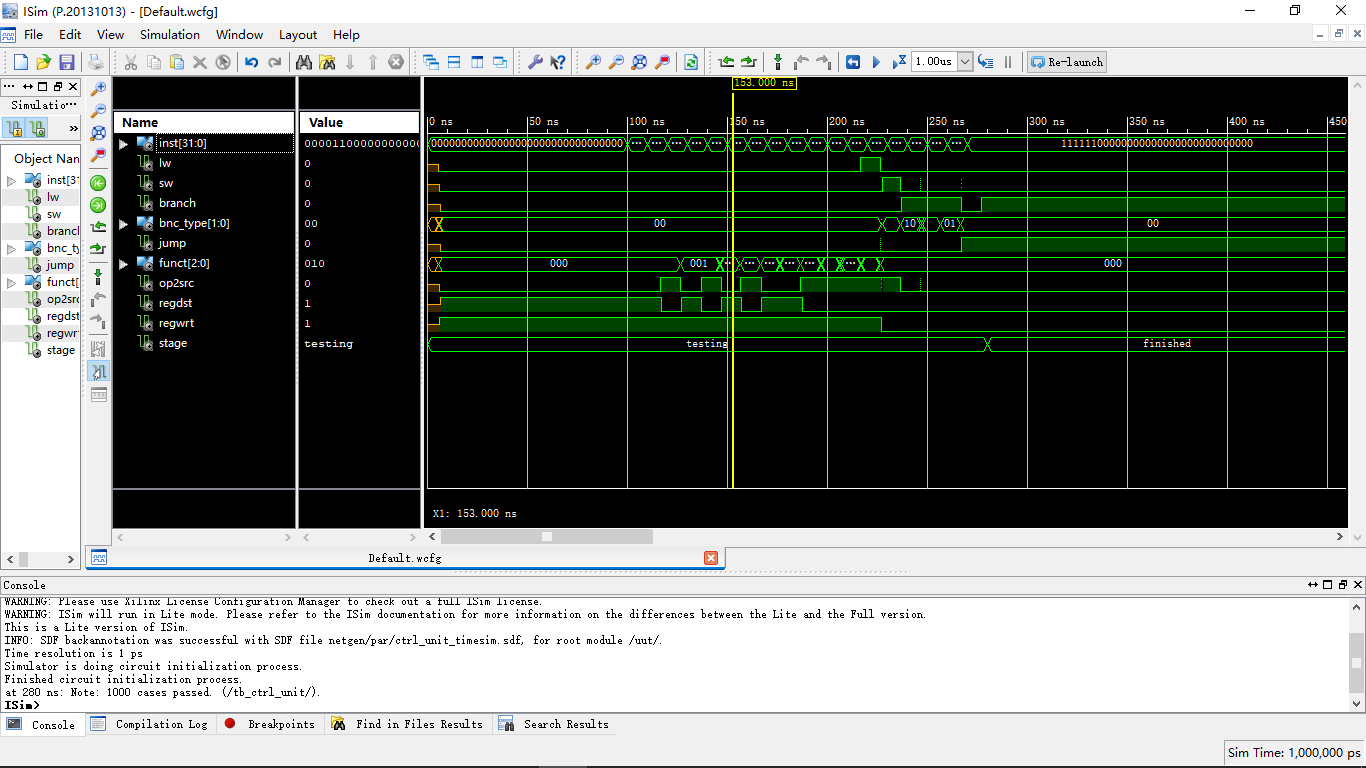
###### Timing Simulation



Testing first 9 conditions



Testing latter 9 conditions



An overview with all cases passed.

#### Data memory

###### Implementation

###### Testbench

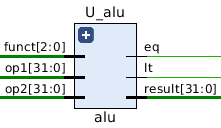
###### Functional Simulation

###### 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)

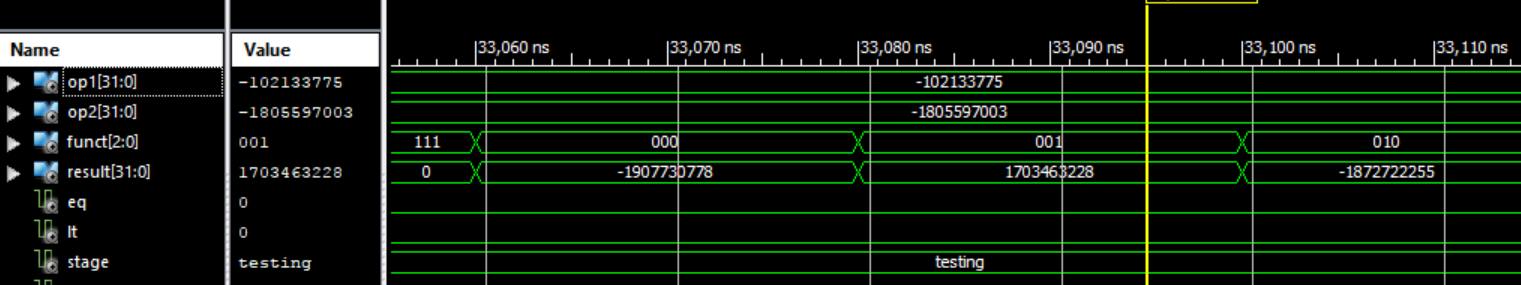
5: do result = op1 << op2

6: do result = op1 >> op2

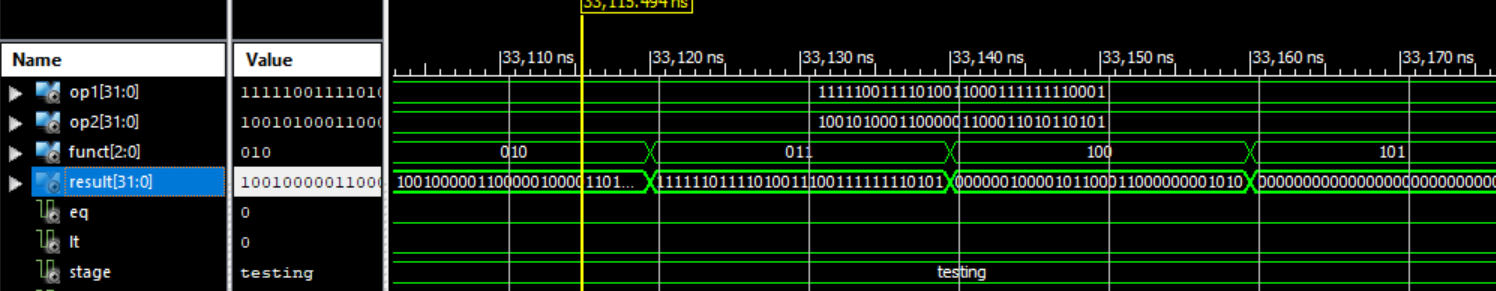
###### Testbench

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.

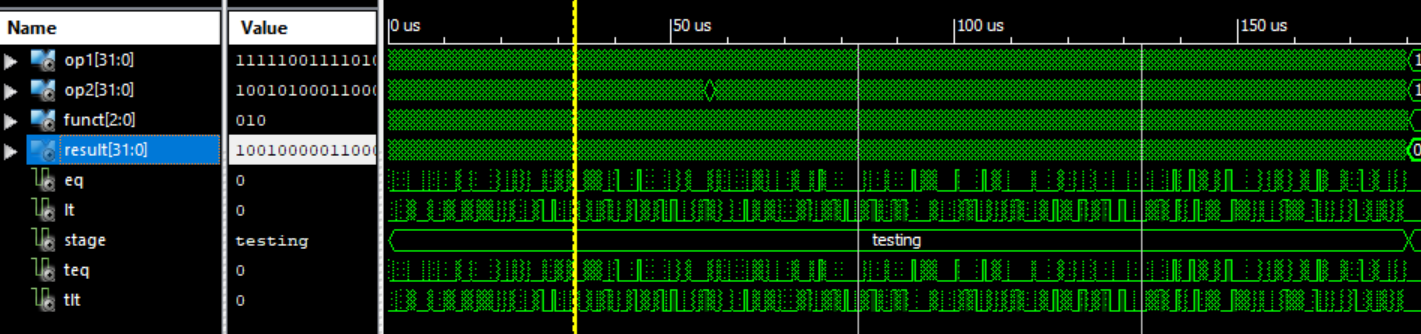
###### Functional Simulation



case “+” and “-” showed on decimal

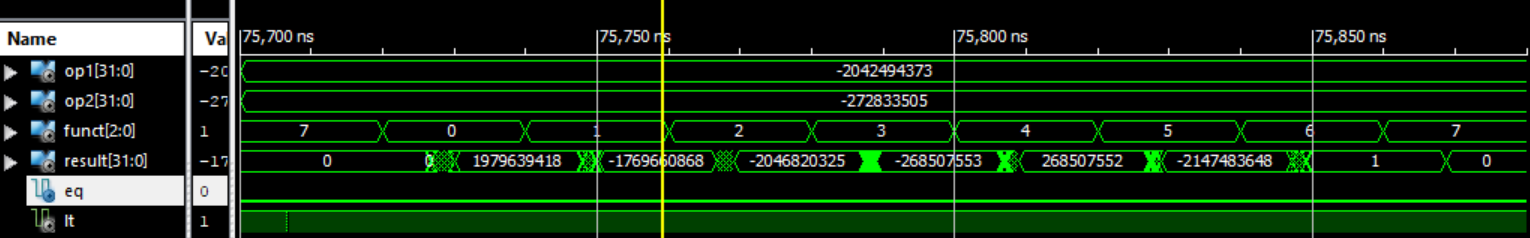


bit operation case showed on bits

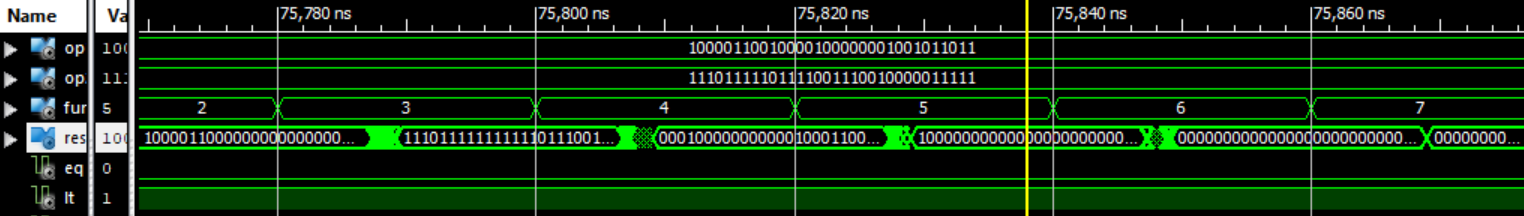


An overview. All cases passed

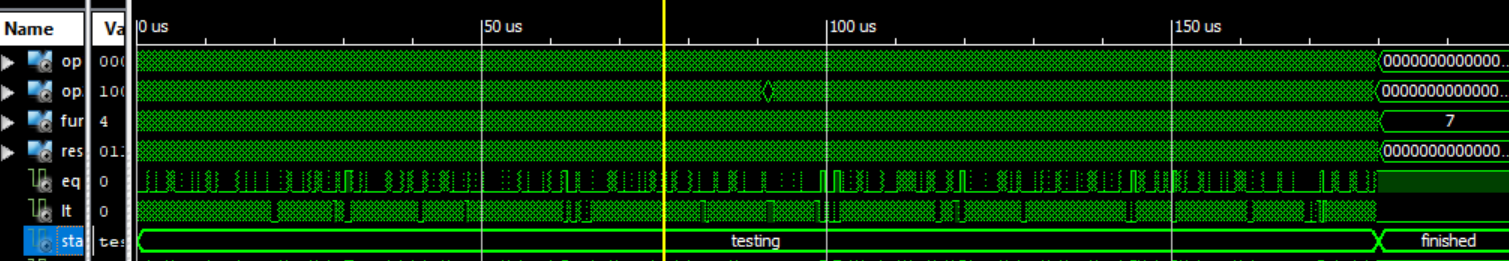
###### Timing Simulation



case “+” and “-” showed on decimal



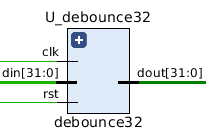
bit operation case showed on bits



Overview. All cases passed

#### Debouncer

###### Implementation



Signal list:

in clk: clock signal

in rst: reset signal, active-high

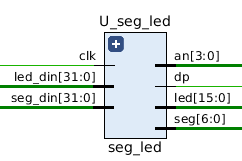
in din: input data

out dout: output data

This is a unit for input debouncing. It uses a 3-state FSM to achieve debounce. It gets the data only when the state is idle. When pressing and releasing are not stable, the machine will keep in the “pressing” or “releaseing” state and will not send the signal to output.

#### Seg\_led

###### Implementation



Signal list:

in clk: clock signal

in led\_din: 32-bit input for led

in seg\_din: 32-bit input for 7-segment display

out an: 7-segment display digit selector signal

out dp: 7-segment display dp control signal

out seg: 7-segment display seg control signal

out led: LED output

This is a controller for on-board led and 7-segment display, used to show the value of two special register. The led will show the lower 16 bits of led\_din signal, which is connected to r29. And 7-segment will show the value of seg\_din in hexadecimal format, which is connected to r30.

With this design, we can output the internal value of the CPU on FPGA, simply by writing it into r29 (binary display) or r30 (hex display).

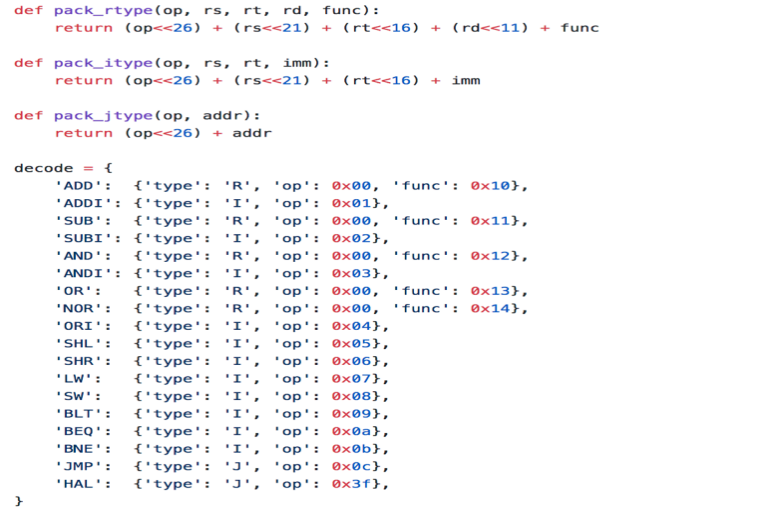
## Processor Desgin and test

#### Assmebly comiler

We wrote out instructions in assembly and used a simple compiler written in python (load\_tb\_instructions.py) to help convert assembly to machine code (\*.binary), which can be conducted directly on the FPGA board.

Our compiler contains three main parts: trimming, compiling and writing-back.

During trimming, all comment, extra white space,



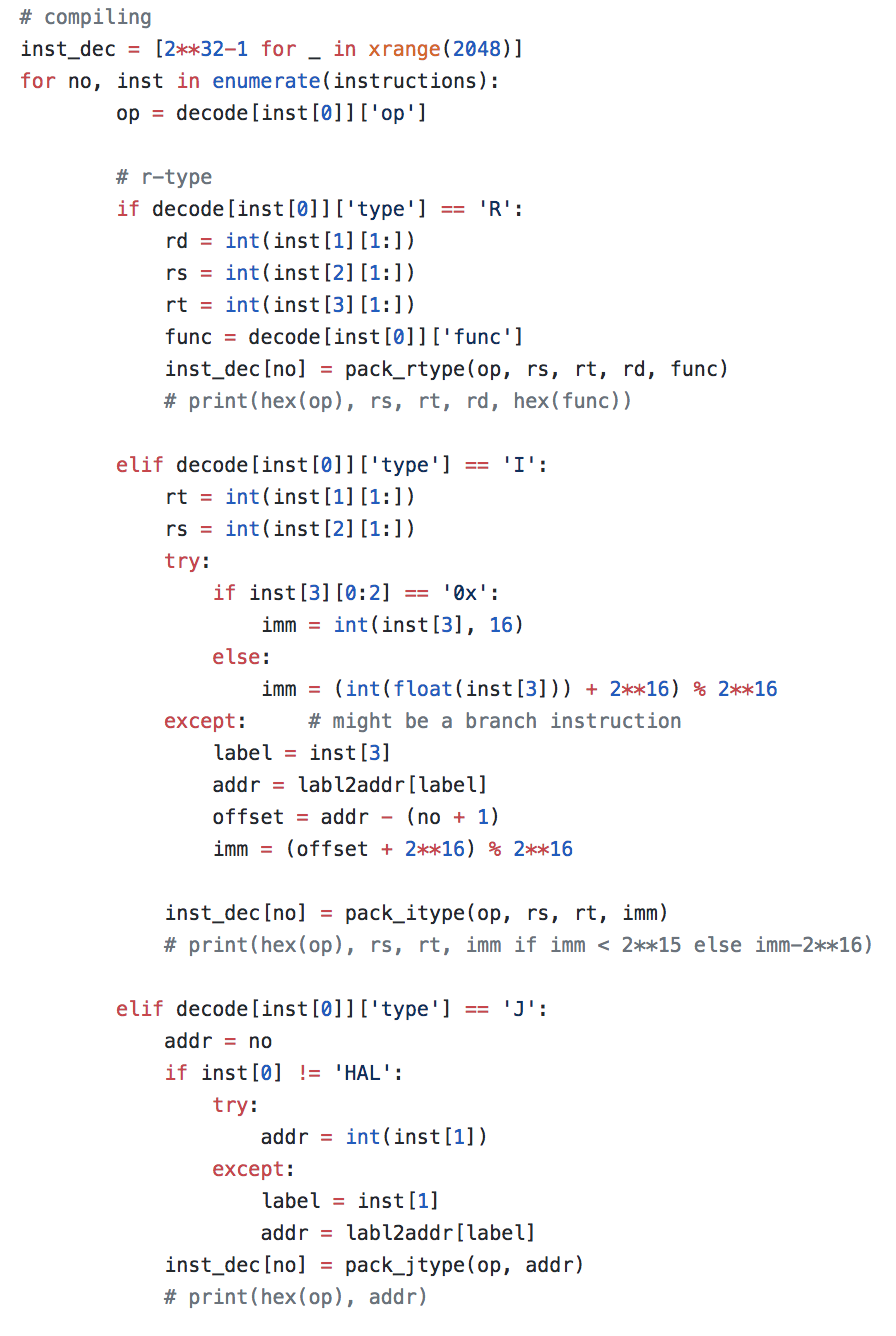
divide instruction types

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



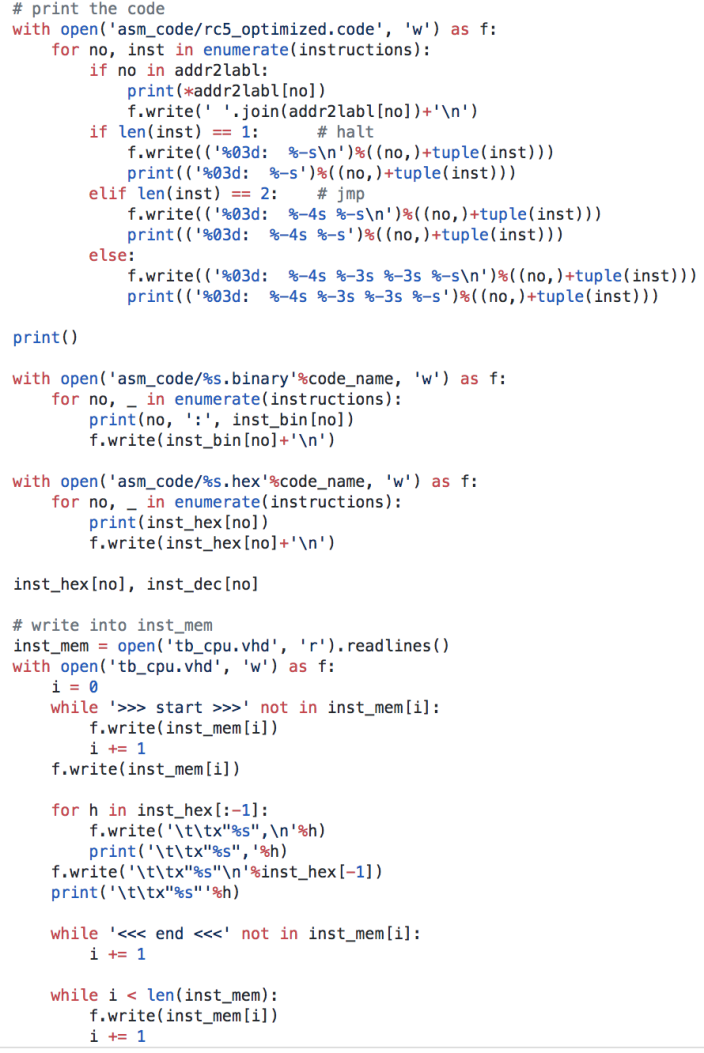
read input stream

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 in 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