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

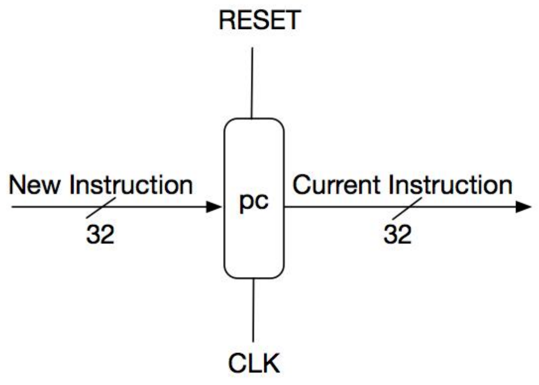
The NYU-6463 Processor performs the tasks of instruction fetch, instruction decode, execution and memory access all in one clock cycle. First, the PC value is used as an address to index the instruction memory which supplies a 32-bit value of the next instruction to be executed. This instruction is then 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

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.

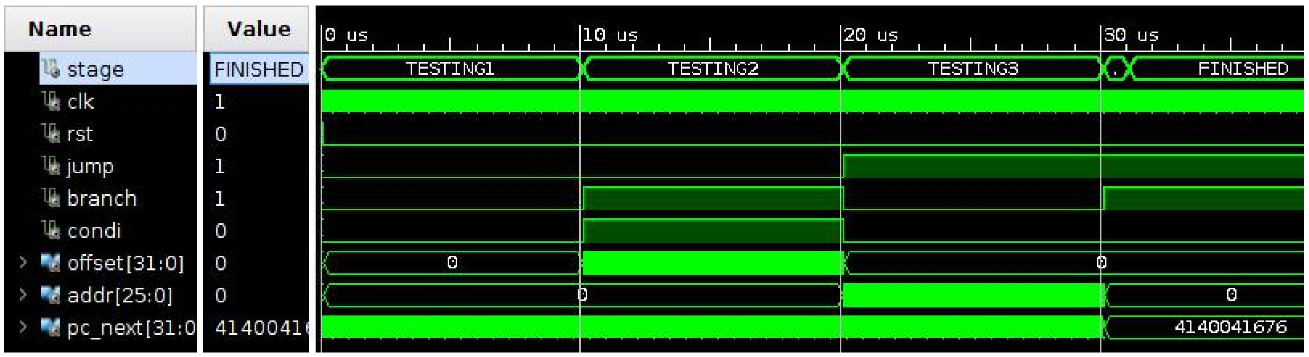


Functional Simulation:

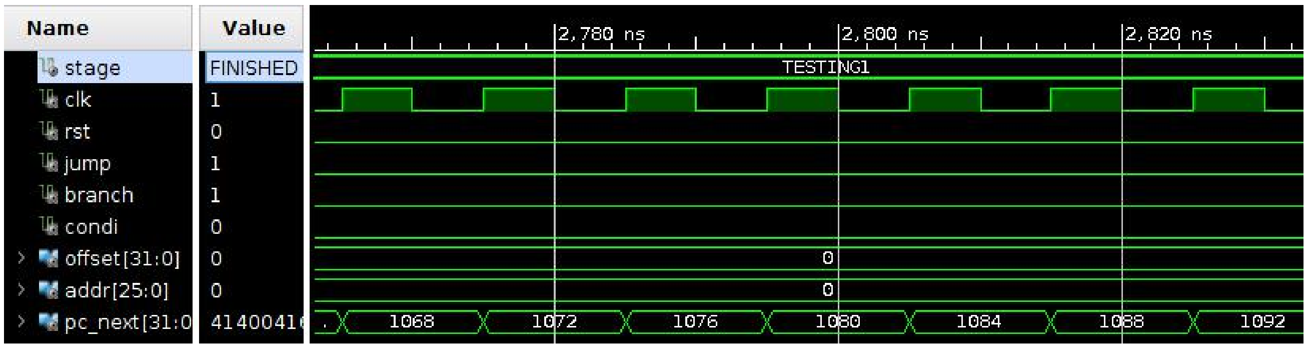
PC has 4 different source:

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

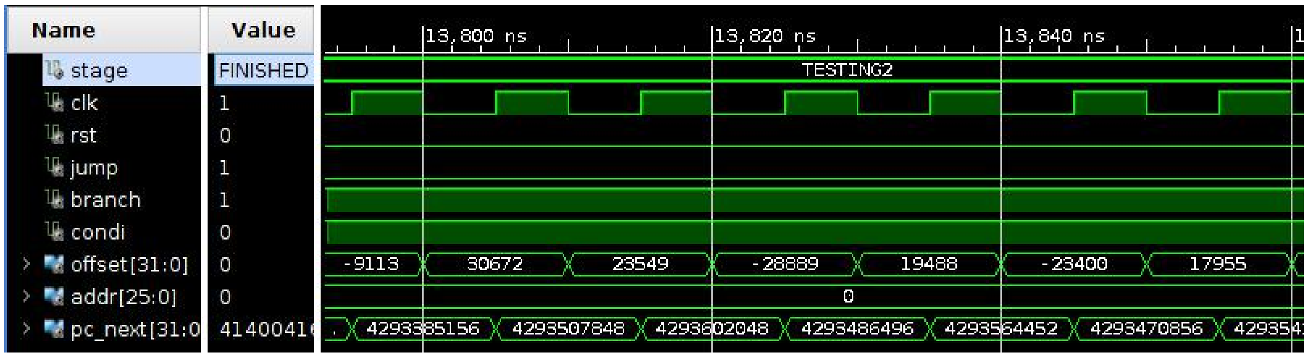
For each situation, we tested the PC unit on 1000 random cases and used ‘assert’ statement to check the output automatically (see tb_pc.vhd for details). The stage signal would change to FINISH only when all test cases passed, otherwise, the simulation would stop with severity level ‘failure’.



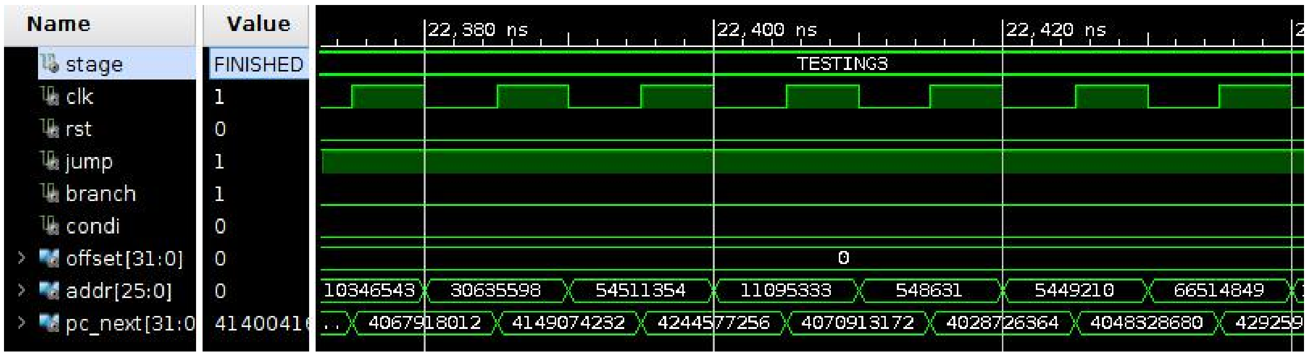
▲ Signal wave overview. All cases passed.



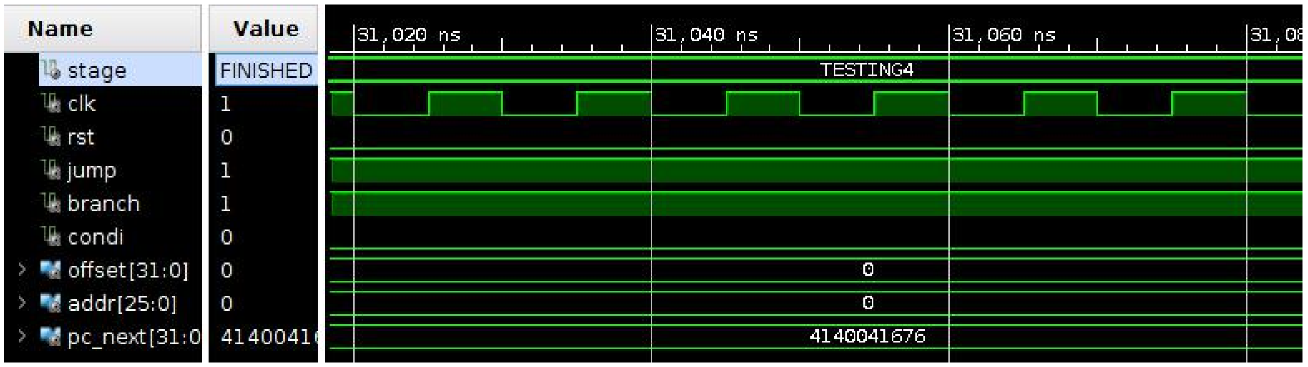
▲ Testing continuously increasing



▲ Testing branch instruction



▲ Testing jump instruction

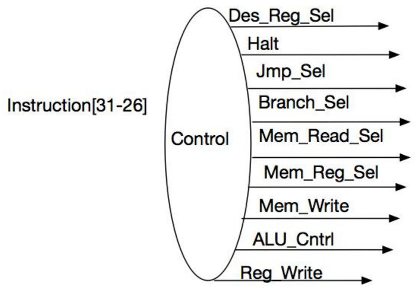


▲ Testing halt instruction

2.1.2 Control Unit:

Brief Introduction:

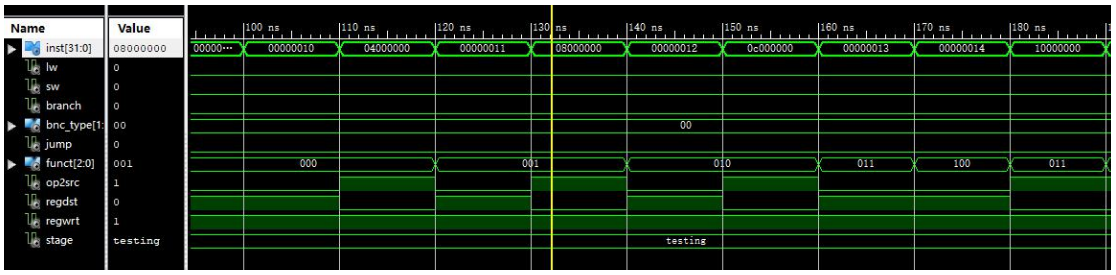
The control unit of the block diagram examines the instruction opcode bits [31 – 26] and decodes the instruction to generate control signals to be used in the additional modules. The Dst_Reg_Del determines which register will be selected to be written into the register file. The Jmp_Sel control signal 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.



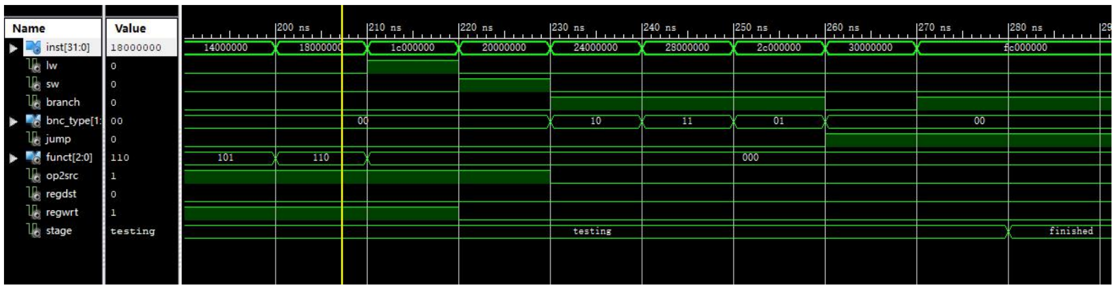
Functional Simulation:

The 18 pictures shows in the sequence according to that 18 instructions in the process pdf.

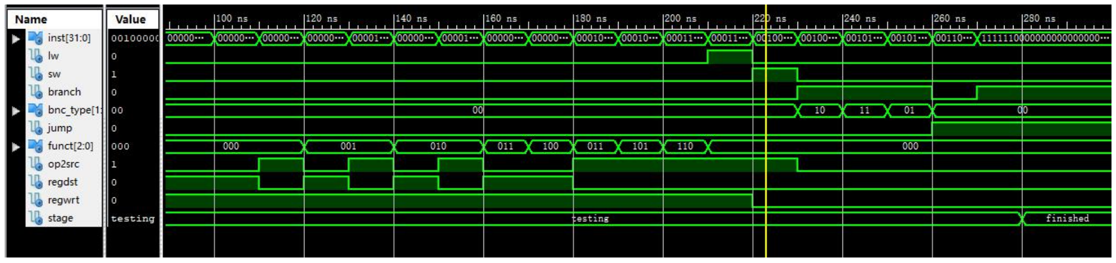
First 9 insts:



Last 9 insts:



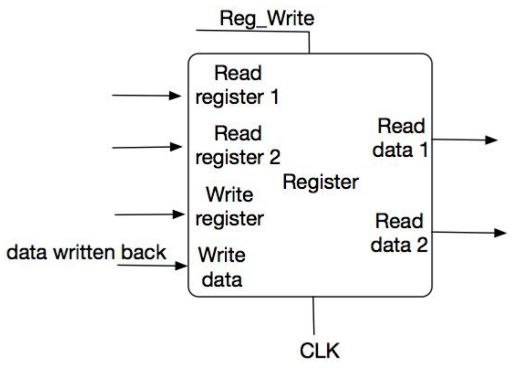
In all :



2.1.3 Register File

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.



Functional Simulation:

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”.

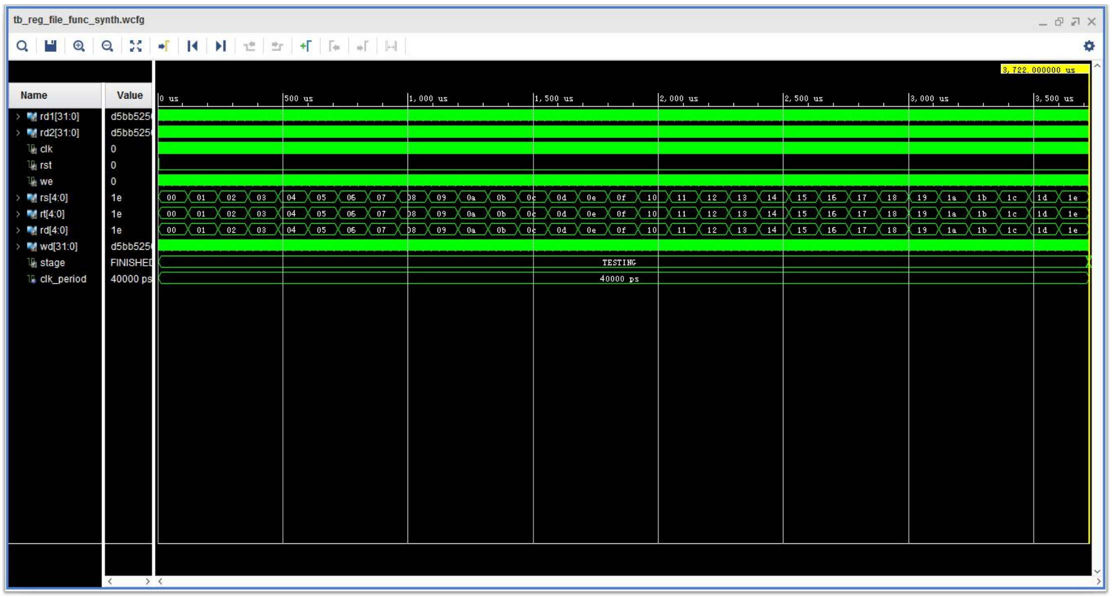


Figure 1. The whole simulation.

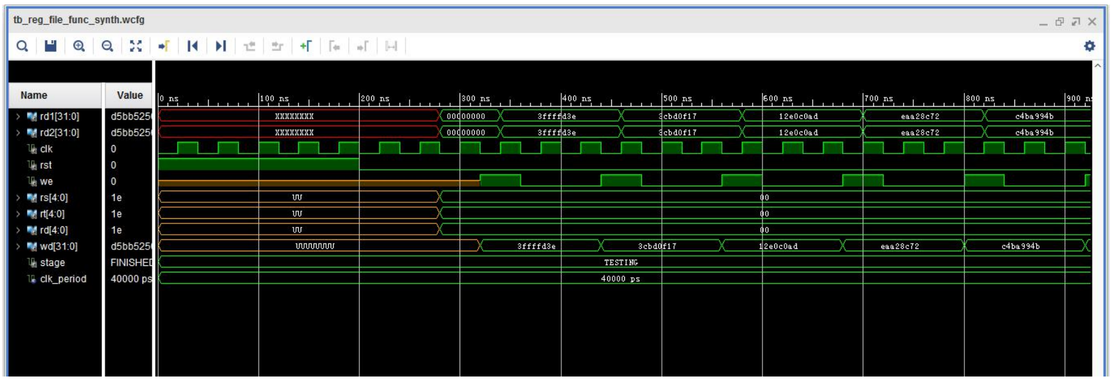


Figure 2. The first few cases

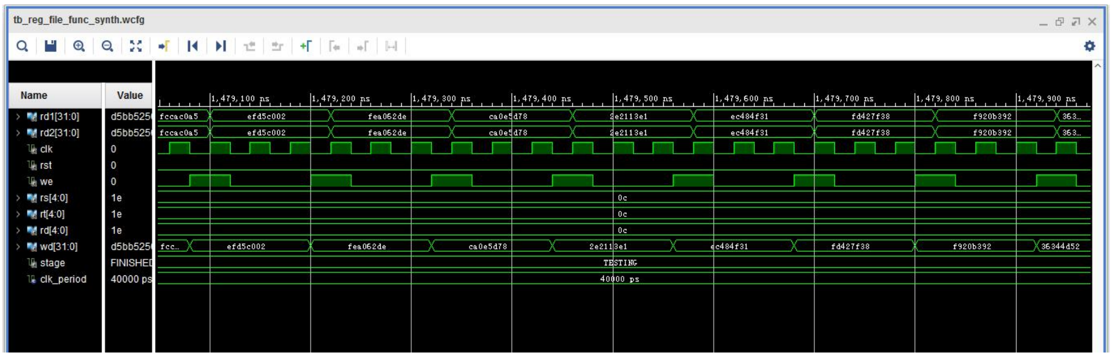


Figure 3. A few cases during the simulation

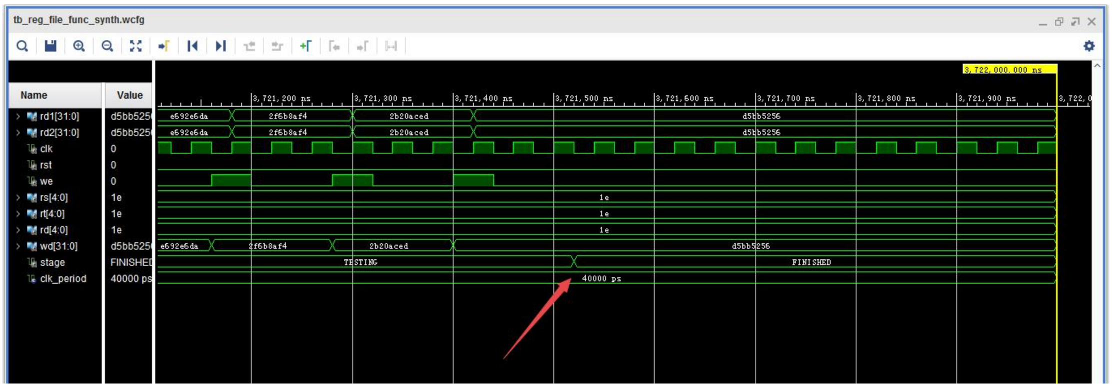
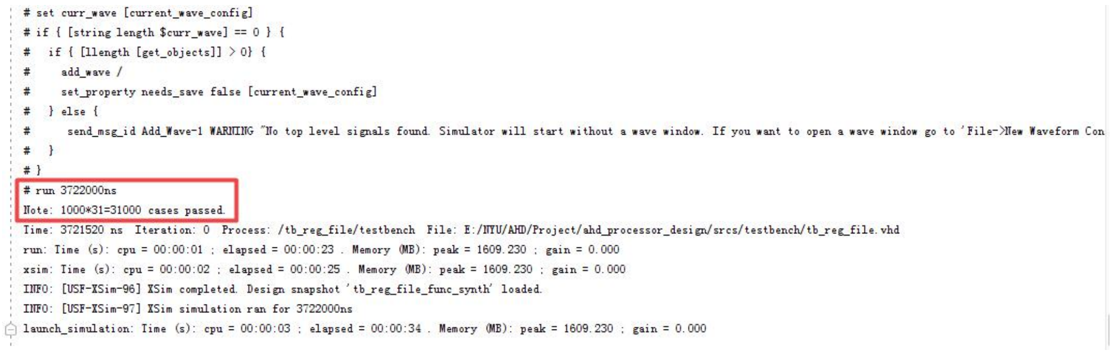


Figure 4. The last few cases.

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



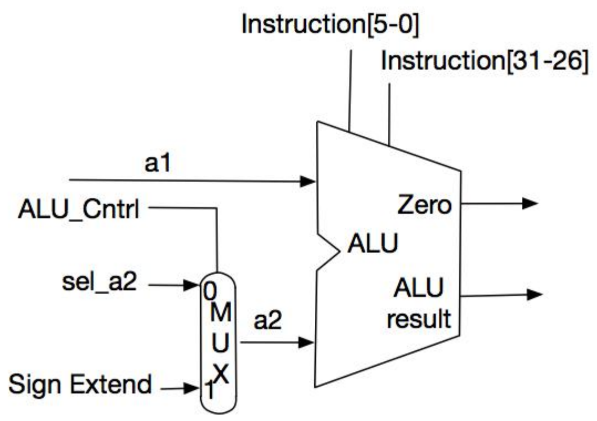
```
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want to open a wave window go to 'File->New Waveform Con
#   }
# }
# run 3722000ns
Note: 1000*31=31000 cases passed.
Time: 3721520 ns  Iteration: 0  Process: /tb_reg_file/testbench  File: E:/NYU/AHD/Project/ahd_processor_design/srcs/testbench/tb_reg_file.vhd
run: Time (s): cpu = 00:00:01 ; elapsed = 00:00:23 . Memory (MB): peak = 1609.230 ; gain = 0.000
xsim: Time (s): cpu = 00:00:02 ; elapsed = 00:00:25 . Memory (MB): peak = 1609.230 ; gain = 0.000
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_reg_file_func_synth' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 3722000ns
launch_simulation: Time (s): cpu = 00:00:03 ; elapsed = 00:00:34 . Memory (MB): peak = 1609.230 ; gain = 0.000
```

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

2.1.4 ALU

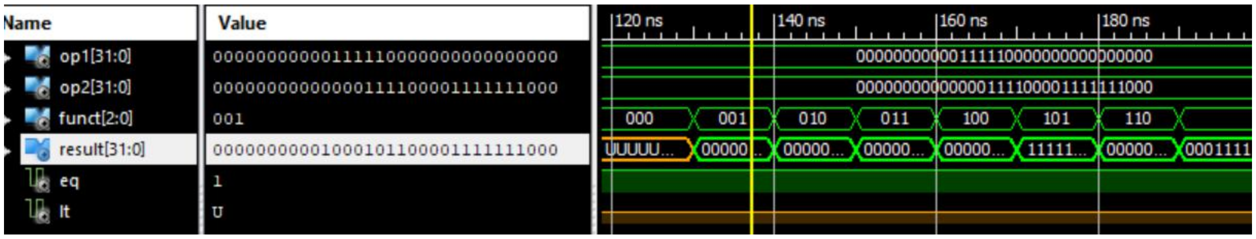
Brief Introduction:

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.



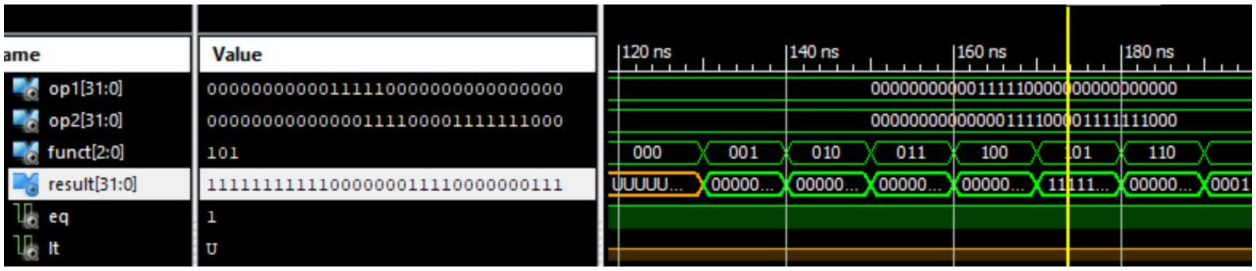
Functional Simulation:

Asynchronous
eq is HIGH when op1 == op2, otherwise LOW.
lt is HIGH when op1 < op2, otherwise LOW.
funct:
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
Function simulation:
op1+op2:



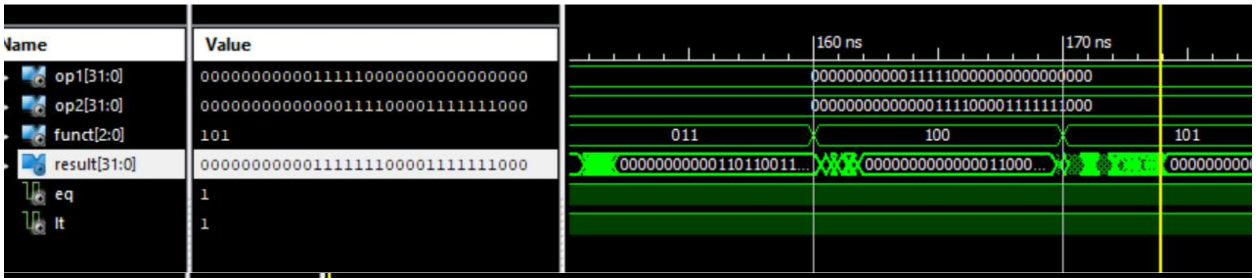
```
Name            Value                               |120 ns        |140 ns        |160 ns        |180 ns
op1[31:0]       00000000000111110000000000000000    00000000000111110000000000000000
op2[31:0]       00000000000000111100001111111000    00000000000000111100001111111000
funct[2:0]      001                                 000  001  010  011  100  101  110
result[31:0]    00000000001000101100001111111000    UUUUU...  00000  00000...  00000...  00000...  11111...  00000...  0001111
eq              1
lt              U
```

~op1|op2:

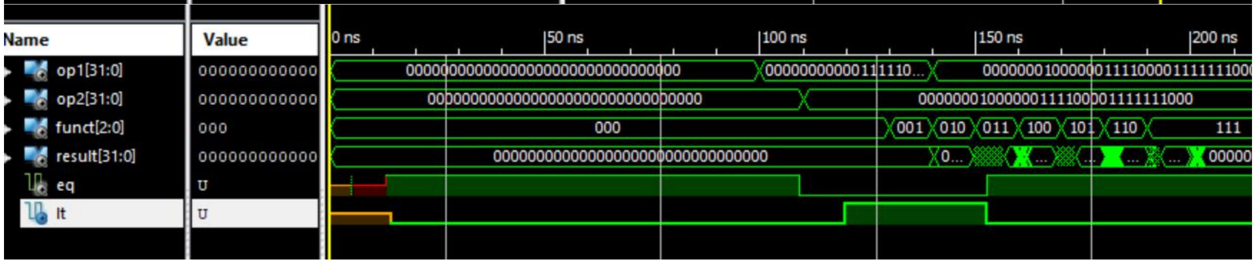


```
ame            Value
op1[31:0]      00000000000111110000000000000000
op2[31:0]      00000000000000111100001111111000
funct[2:0]     101
result[31:0]   11111111111000000011110000000111
eq             1
lt             U
```

Post-route simulation:



```
Name           Value
op1[31:0]      00000000000111110000000000000000
op2[31:0]      00000000000000111100001111111000
funct[2:0]     101
result[31:0]   00000000000111111100001111111000
eq             1
lt             1
```



```
Name           Value
op1[31:0]      000000000000
op2[31:0]      000000000000
funct[2:0]     000
result[31:0]   000000000000
eq             U
lt             U
```

The delay is about 13.903ns.

2.1.5 Instruction Memory

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.

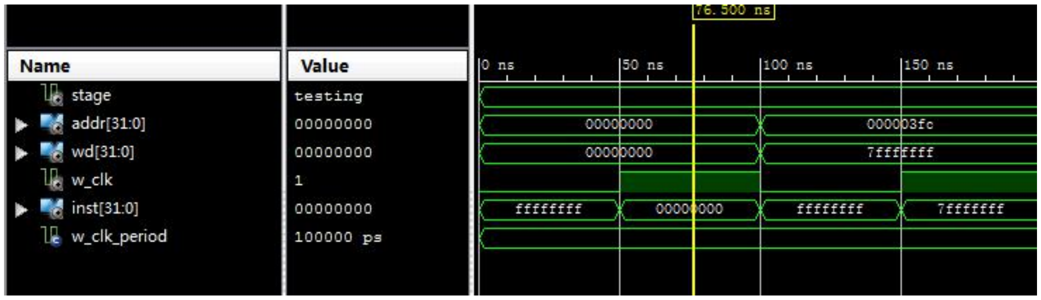
• At the rising edge of w_clk, we write the value of the wd into instruction memory. In this way, it can support changing the program while our processor is running on the FPGA.

Functional Simulation:

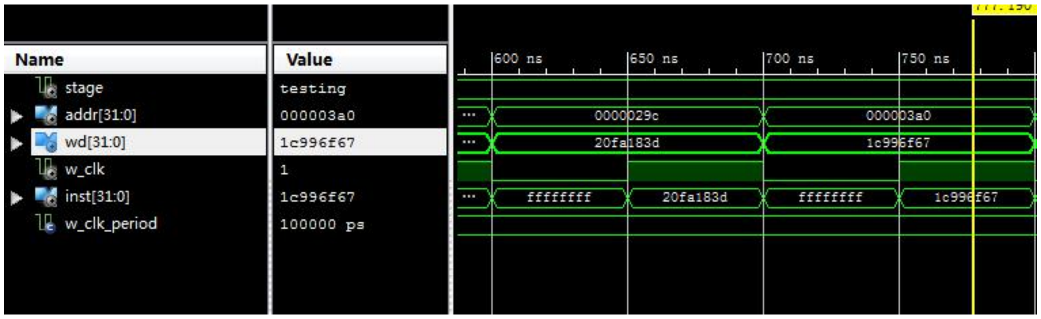
Random generate 1000 test cases.

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.

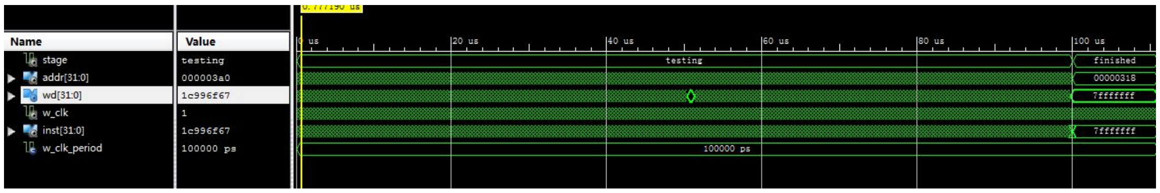
Function Simulation:



Name	Value
stage	testing
addr[31:0]	00000000
wd[31:0]	00000000
w_clk	1
inst[31:0]	00000000
w_clk_period	100000 ps



Name	Value
stage	testing
addr[31:0]	000003a0
wd[31:0]	1c996f67
w_clk	1
inst[31:0]	1c996f67
w_clk_period	100000 ps

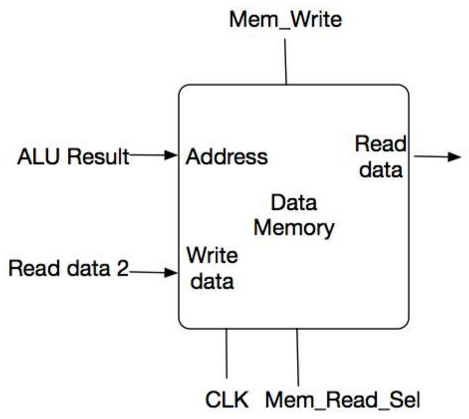


Name	Value
stage	testing
addr[31:0]	000003a0
wd[31:0]	1c996f67
w_clk	1
inst[31:0]	1c996f67
w_clk_period	100000 ps

2.1.6 Data Memory

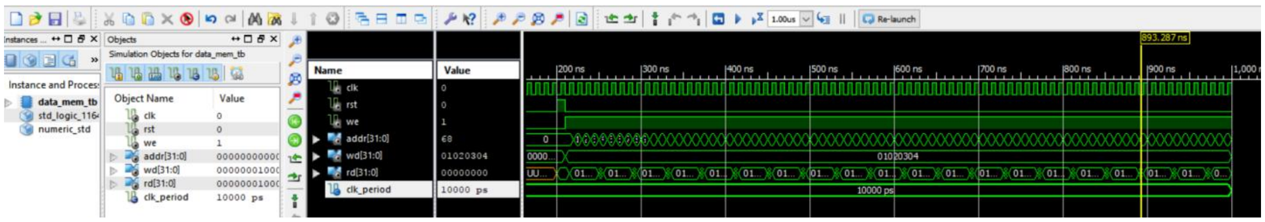
Brief Introduction:

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.



Functional Simulation:

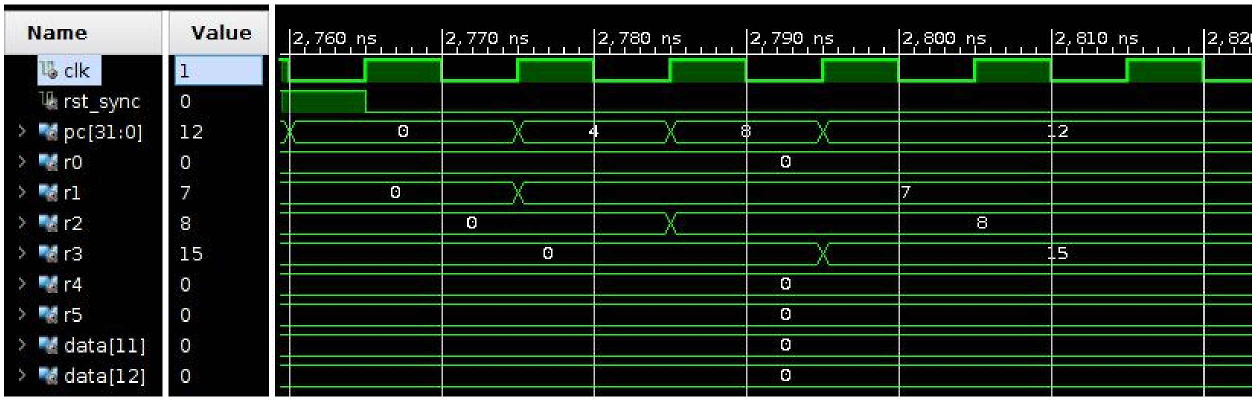
1. addr from 0 – 127 and wd = 00000001000000100000001100000100
2. addr from 0 – 127 and wd = 00000100000000110000001000000001
3. addr from 0 – 127 and wd = 11111111111111101111110111111100
4. addr from 0 – 127 and wd = 11111100111111011111111011111111
5. addr from 0 – 127 and wd = 00000001000000101111110111111100
6. addr from 0 – 127 and wd = 11111100111111010000001000000001
7. addr from 0 – 127 and wd = 00000100000000111111111011111111
8. addr from 0 – 127 and wd = 11111110111111110000001100000100



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
00000000010000010001100000010000 --ADD R3, R1, R2 // R3 = R1 + R2 =15
11111100000000000000000000000000 --HAL // HALT

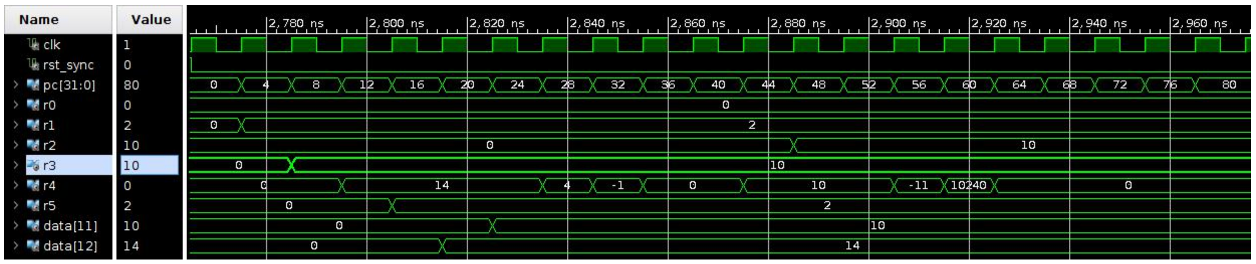


As we can see from the result, the final result in R3 is 15

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
```



2.2.3 RC5 Functional Simulation

We set

ukey = 0x91cea91001a5556351b241be19465f91

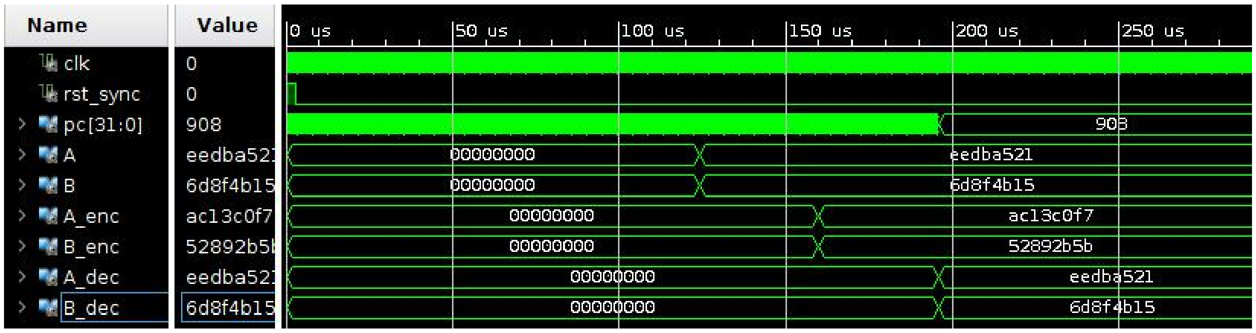
A_in = 0xeedba521

B_in = 0x6d8f4b15.

The expected encryption result is

A_enc = 0xac13c0f7, B_enc = 0x52892b5b

Our code did one round of encryption and decryption on A_in and B_in. We can see in the screenshot below that, our code can encrypt and decrypt the data correctly. (Code can be found in file ‘rc5.asm’)


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
0 us
50 us
100 us
150 us
200 us
250 us
908
00000000
eedba521
00000000
6d8f4b15
00000000
ac13c0f7
00000000
52892b5b
00000000
eedba521
00000000
6d8f4b15