# 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

## 1. Abstract

The main objective of this project is to design a single cycle 32-bit MIPS RISC processor using VHDL, and implementing it on an FPGA with on-board input/output interface to input data and get results on-the-fly. To show whether it works properly for these instructions, we wrote a RC5 assembly code using only the supported instructions supported, converted the assembly code into machine code and ran it on FPGA.

The highest frequency of our processor is **76.8 MHz** according to timing summary. However, on FPGA, our processor can run at **135 MHz** stably. Details about timing analysis and how we multiple the sys_clk will be discussed in section 2.3.5.1 and 4.1. The FPGA resource utilization is **3.28%**, which is discussed in section 2.3.5.2.

Our RC5 assembly code is optimized to be efficient. It can run encryption/decryption in **513 cycles**, round key generation in **2574 cycles**. Details about how we achieved it will be discussed in section 3.1.2, 3.1.3 and 3.1.4. And in section 3.2, 3.3 and 3.4, we ran **4000 cases** in functional simulation and **35 cases** in timing simulation, to verify the processor as well as the RC5 code.

## 2. Processor Design and Test

### 2.1 Summary

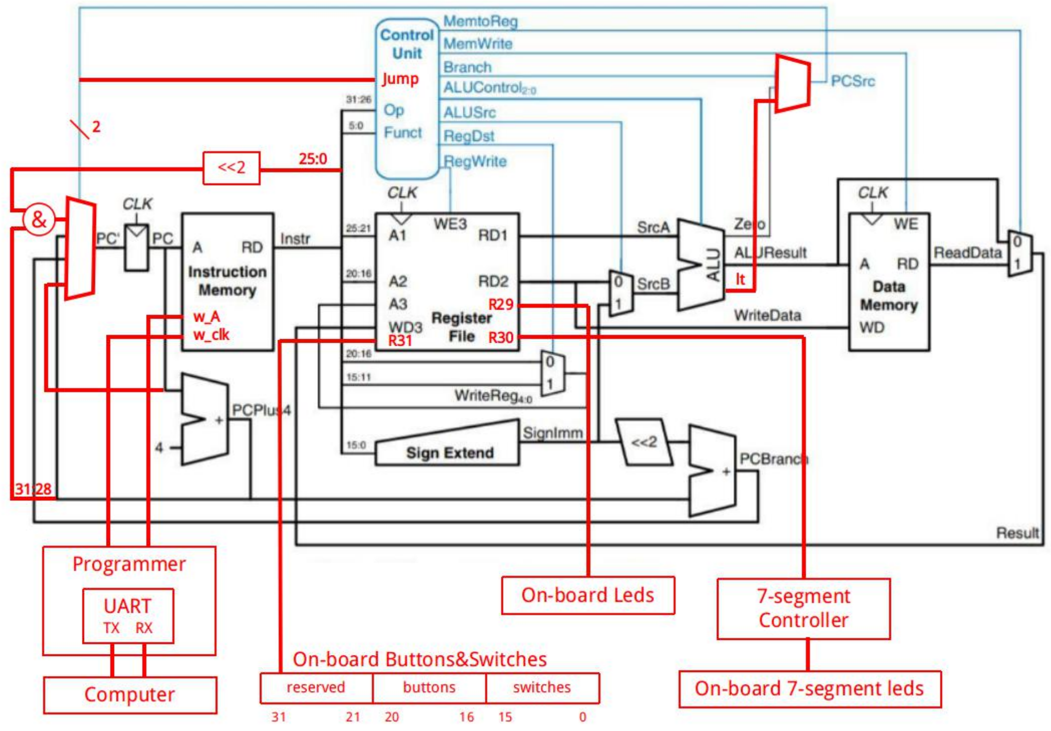


Figure 2.1 Processor block diagram

Above is the block diagram of our processor. The red parts are what we added to the original design, to fix the mistakes or add some new features. Minor changes (like renaming the ports) are not shown in this diagram.

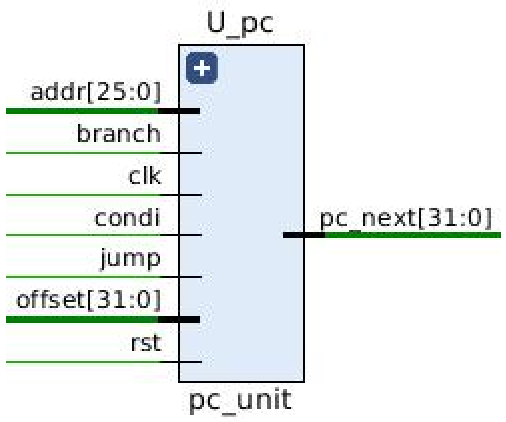
We added 3 new port to the register file, in order to provide an input/output to the processor, where r31 is always connected to the on-board buttons and switches; r29 is always connected to on-board leds; r30 is always connected to a 7-segment controller. In this way, the processor could get the input from buttons and switches by reading the REG[31] and show output by writing to REG[29] or REG[30].

In the next section we will introduce each modules in detail and describe how we design and test them.

## 2.2 Component Design and Test

### 2.2.1 Program Counter (PC) Register

#### 2.2.1.1 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 |

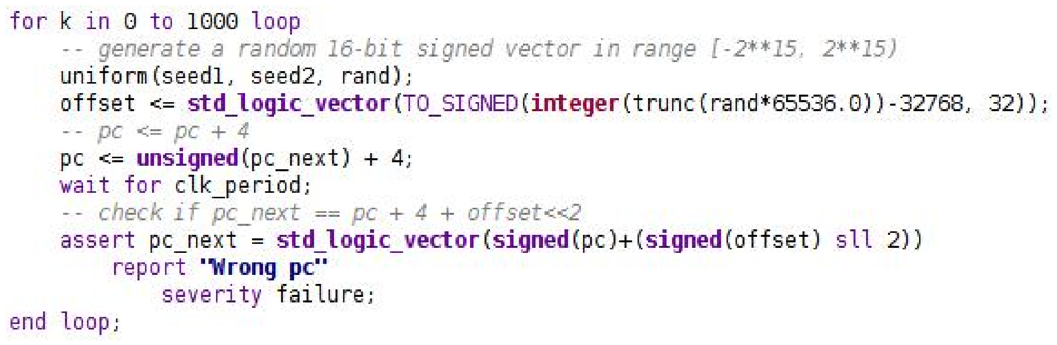
This is a 32-bit register that contain the address of the next instruction to be executed. pc_next will be updated synchronously under 4 different conditions, as show below:

| 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))*

#### 2.2.1.2 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’.



```
for k in 0 to 1000 loop
    -- generate a random 16-bit signed vector in range [-2**15, 2**15)
    uniform(seed1, seed2, rand);
    offset <= std_logic_vector(TO_SIGNED(integer(trunc(rand*65536.0))-32768, 32));
    -- pc <= pc + 4
    pc <= unsigned(pc_next) + 4;
    wait for clk_period;
    -- check if pc_next == pc + 4 + offset<<2
    assert pc_next = std_logic_vector(signed(pc)+(signed(offset) sll 2))
        report "Wrong pc"
            severity failure;
end loop;
```

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

#### 2.2.1.3 Functional simulation

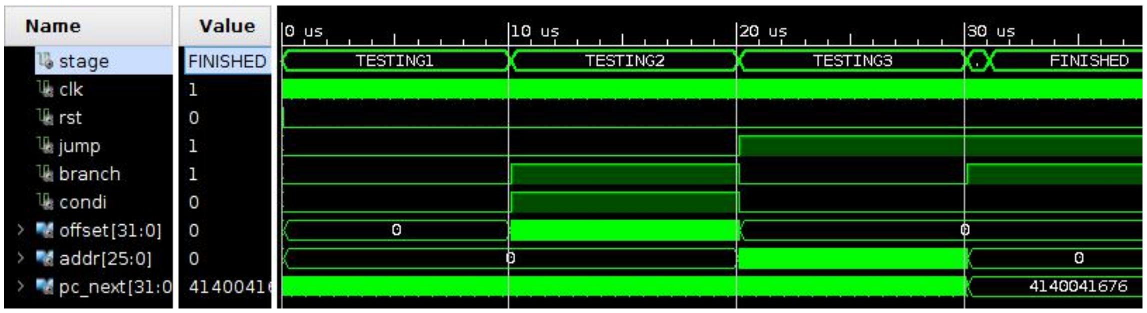


Figure 2.3 An overview. All cases passed.

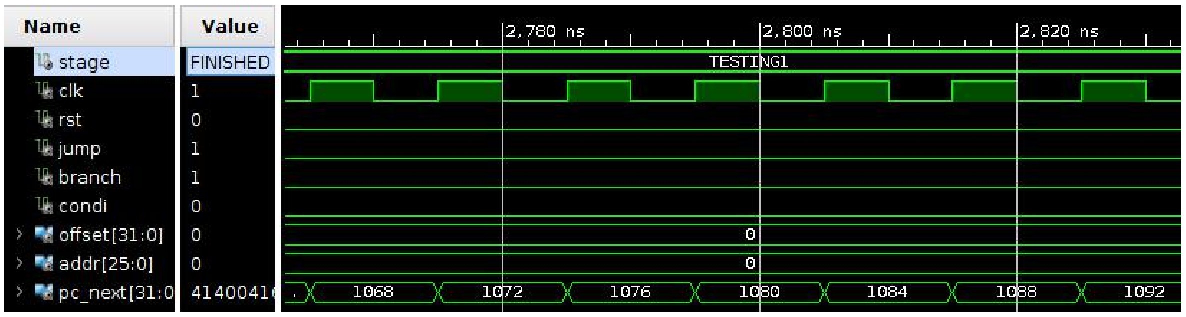


Figure 2.4 Testing continuously increment condition.

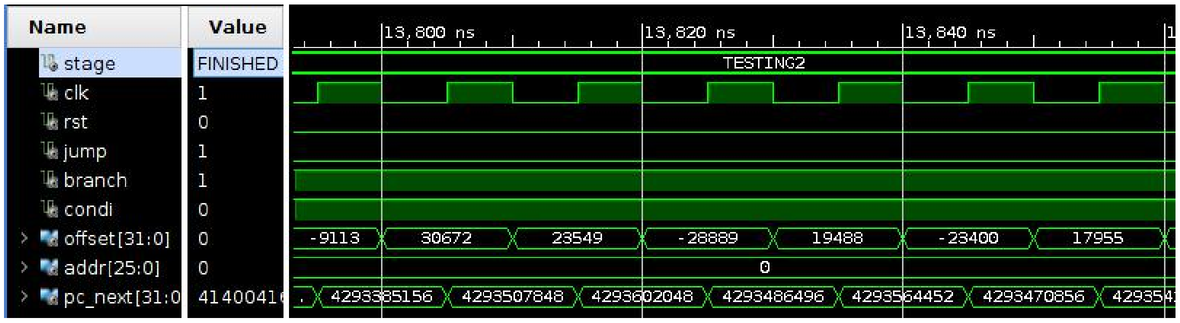


Figure 2.5 Testing branch condition.

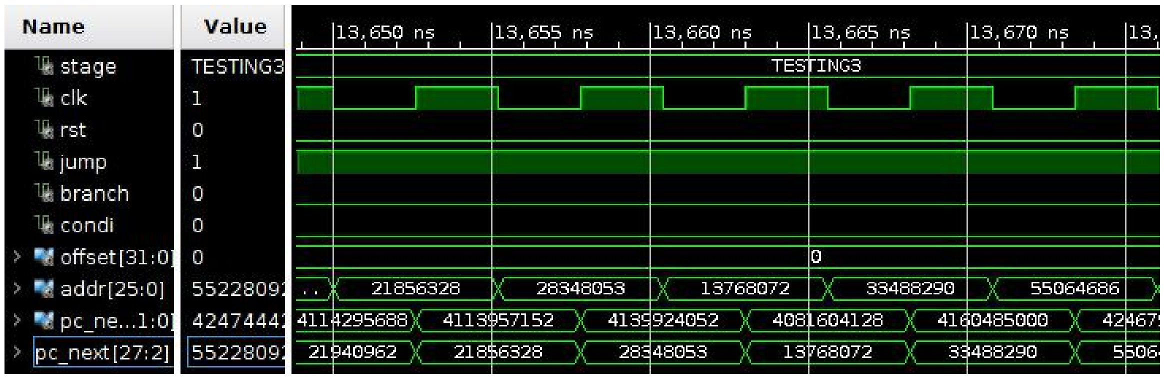


Figure 2.6 Testing jump condition

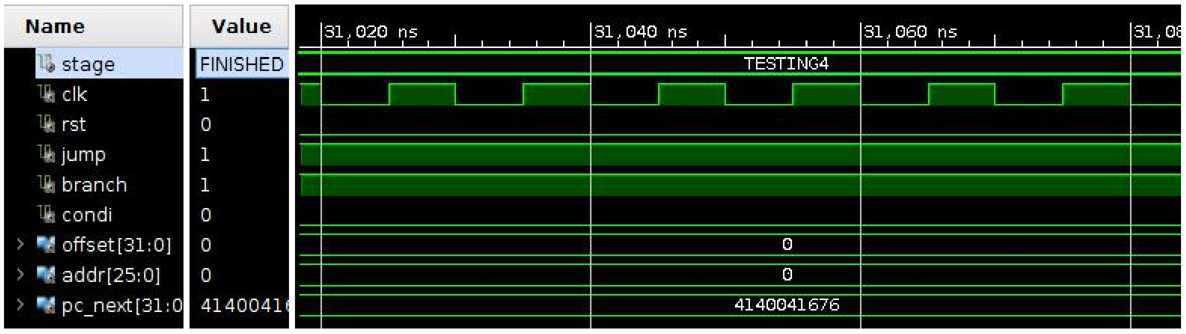


Figure 2.7 Testing halt condition

#### 2.2.1.4 Timing simulation

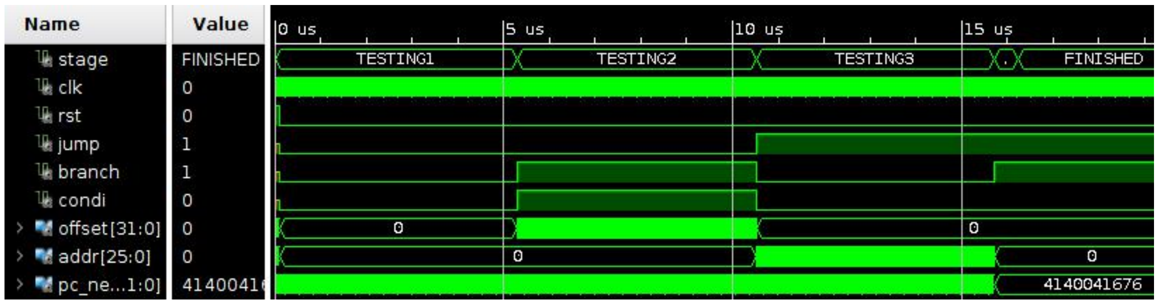


Figure 2.8 An overview. All cases passed.

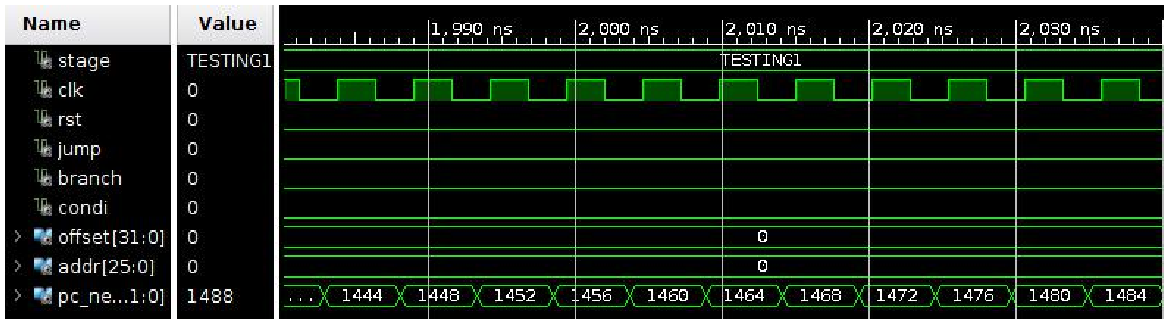


Figure 2.9 Testing continuously increment condition.

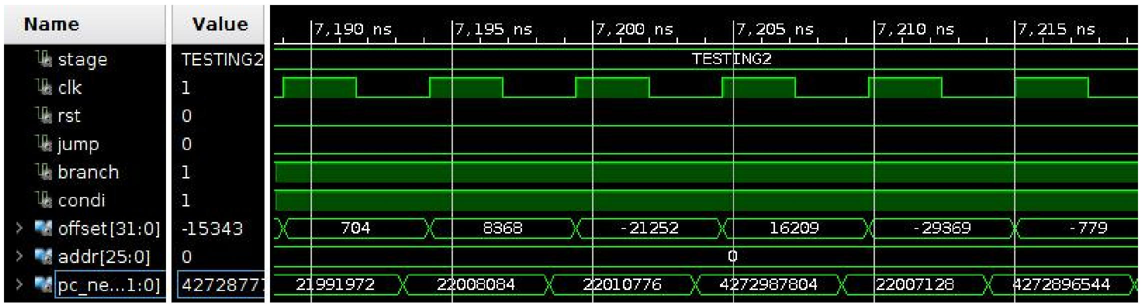


Figure 2.10 Testing branch condition

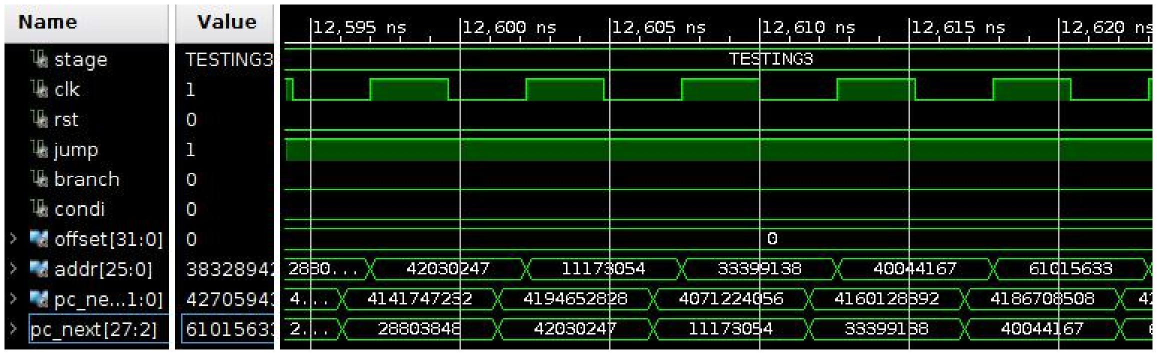


Figure 2.11 Testing jump condition

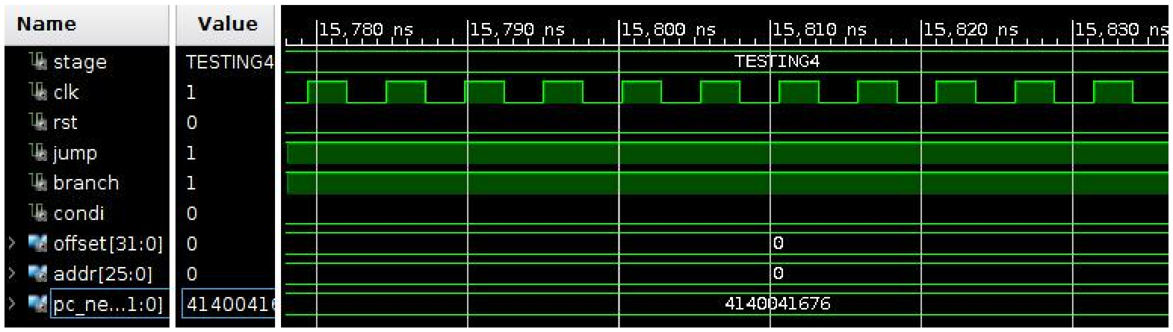
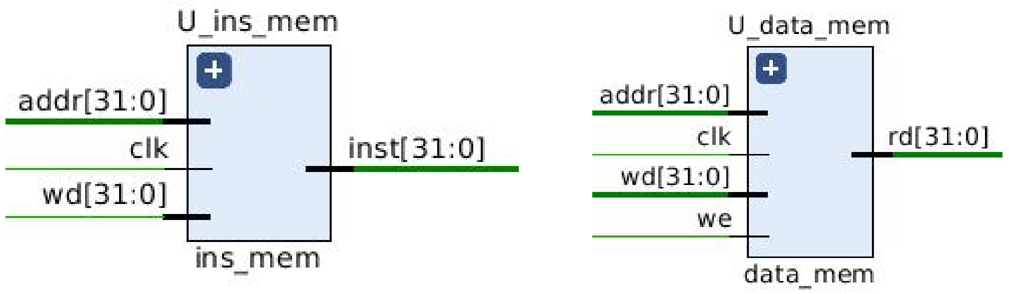


Figure 2.12 Testing halt condition

### 2.2.2 Instruction Memory & Data Memory

#### 2.2.2.1 Implementation of INS_MEM & DATA_MEM



Signal list:

| | | |
|---|---|---|
| in | addr: | 32-bit address |
| in | clk: | write clock signal |
| in | wd: | 32-bits value we want to write in the instruction memory |
| in | we: | write-enable signal; active-high; only in data_mem |
| out | inst: | the result 32-bit instruction. |

Instruction Memory and Data Memory are almost identical, except that data memory has a active-high write enable signal. So we introduce them together in this section

Each instruction/data has its own address, marked by a 32-bit address. The following considerations have to be taken into account during designing:

- Each instruction/data is one word long (32 bits).
- Each instruction/data is addressed by a multiple of 4 bytes (1 word).
- Each instruction/data lies at a multiple of 4 bytes. By contrast, the address in terms of bytes. Lowest 2 bits of the address is ignored.
- Read is asynchronous.

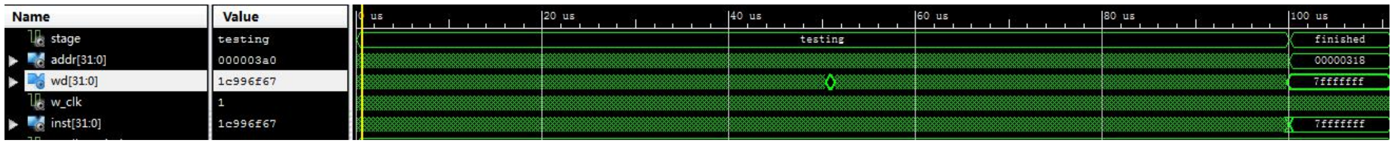
- Writing is synchronous.

#### 2.2.2.2 Testbench

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

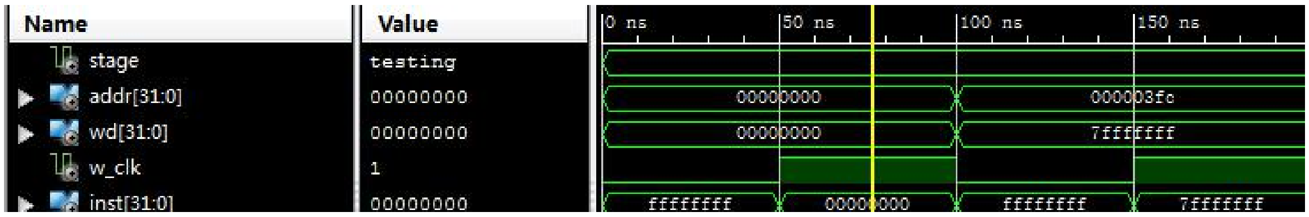
- Generate random vector to change addr and random variable to change wd
- When stage equals to finished, all the test case passed.

#### 2.2.2.3 Functional Simulation



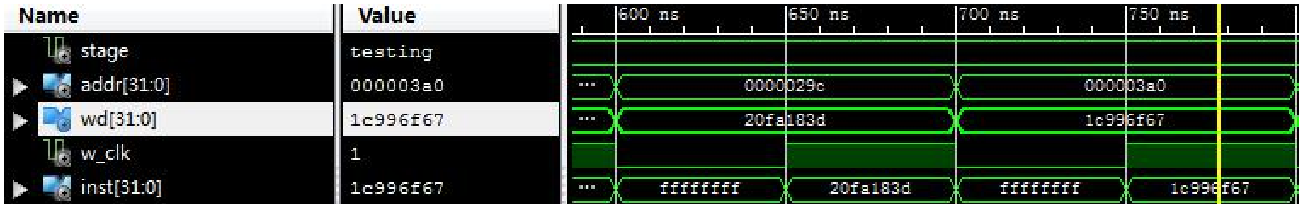
| Name | Value |
|---|---|
| stage | testing |
| addr[31:0] | 000003a0 |
| wd[31:0] | 1c996f67 |
| w_clk | 1 |
| inst[31:0] | 1c996f67 |

Figure 2.13 The stage is finished, which means all test cases passed.



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

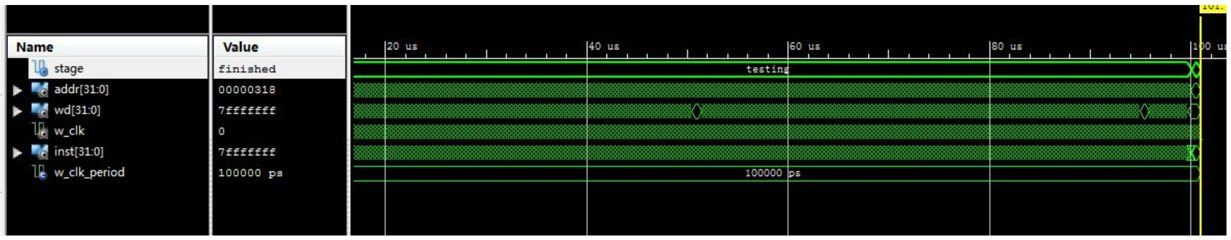
Figure 2.14 Example1: the output instruction is the same as the wd value we wrote in.



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

Figure 2.15 example2: the output instruction is the same as the wd value we wrote in.

#### 2.2.2.4 Timing Simulation



| Name | Value |
|---|---|
| stage | finished |
| addr[31:0] | 00000318 |
| wd[31:0] | 7fffffff |
| w_clk | 0 |
| inst[31:0] | 7fffffff |
| w_clk_period | 100000 ps |

Figure 2.16 The stage is finished, which means all test cases passed.

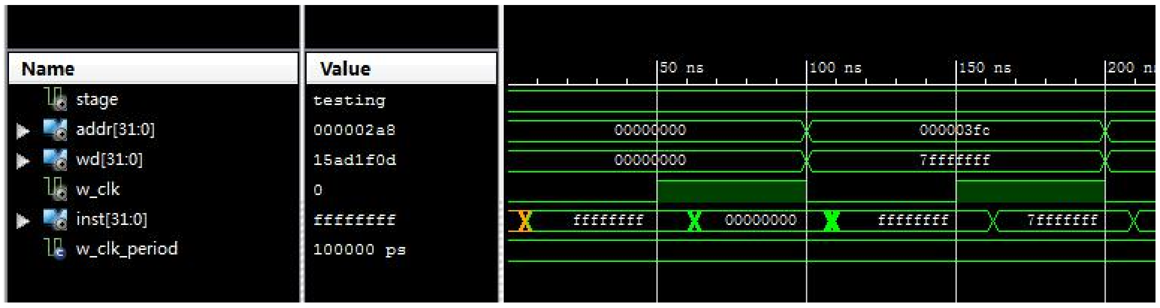


Figure 2.17 timing simulation for first example.

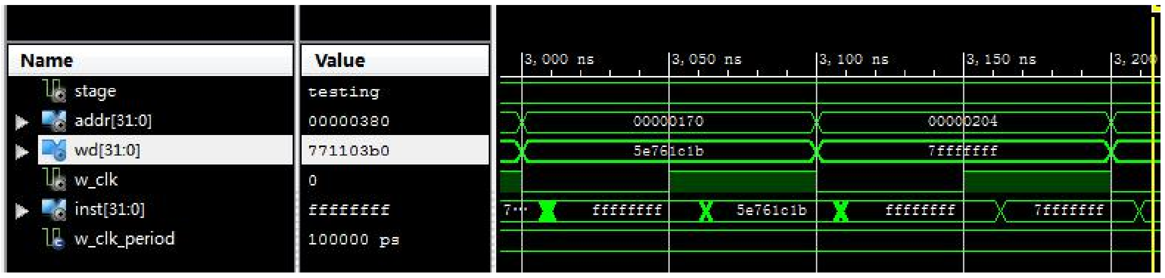
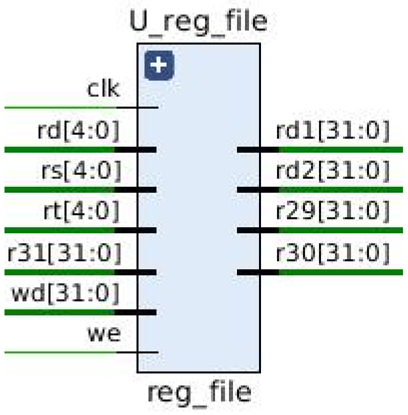


Figure 2.18 timing simulation for second example.

### 2.2.3 Register file

#### 2.2.3.1 Implementation



Signal list:

| | | |
|---|---|---|
| in | clk: | clock 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

### 2.2.3.2 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".

### 2.2.3.3 Functional Simulation

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

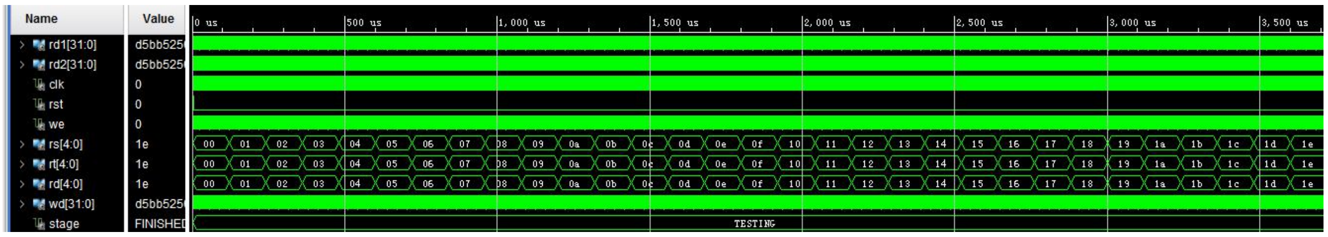


Figure 2.19 The whole simulation.

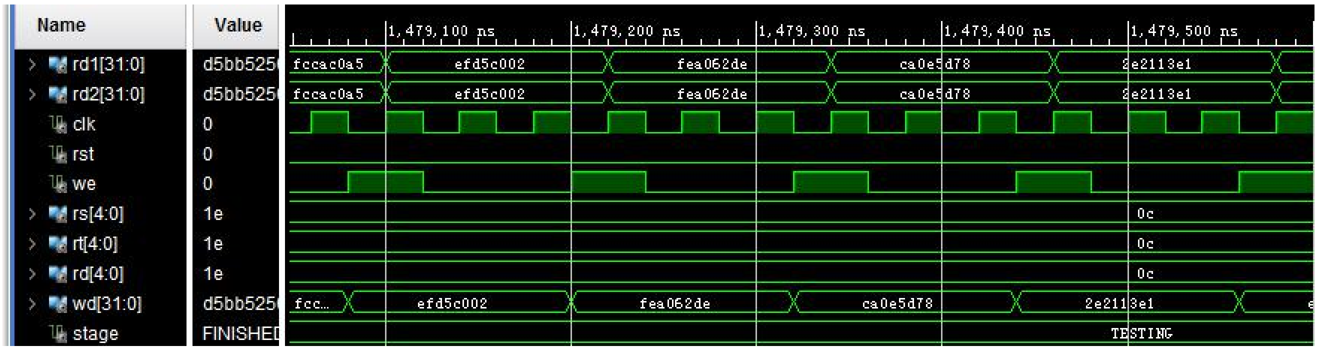


Figure 2.20 Some cases in detail. We can see that rd = wd @ risingede(clk)&we=1

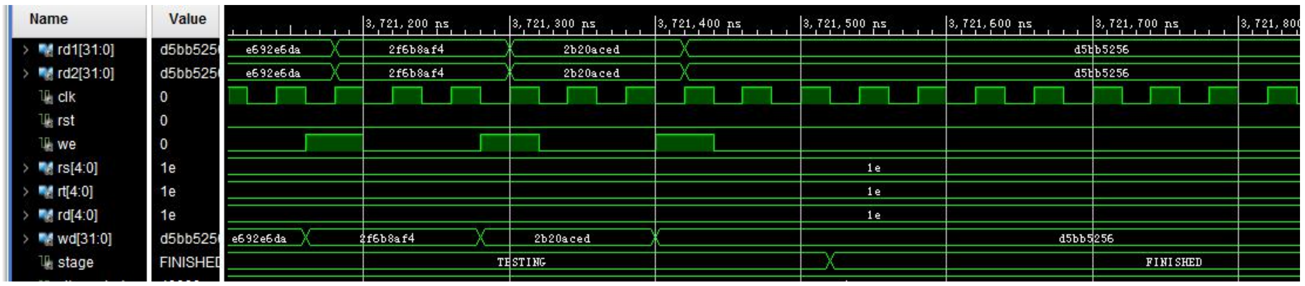


Figure 2.21 The last few cases. Stage signal changes to FINISHED when all cases have passed.

### 2.2.3.4 Timing Simulation

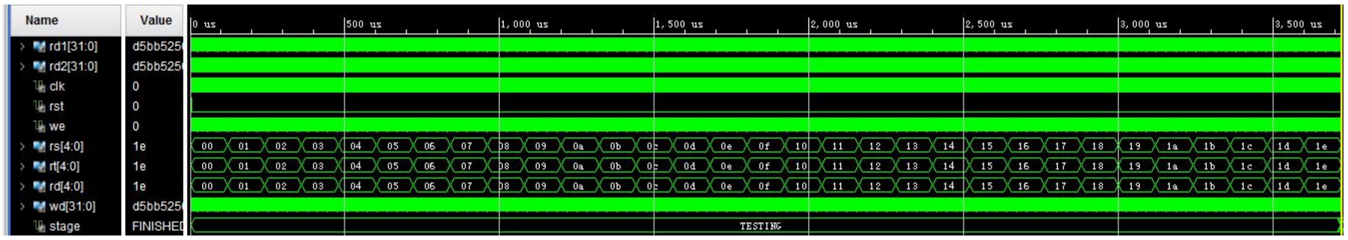


Figure 2.22 The whole simulation

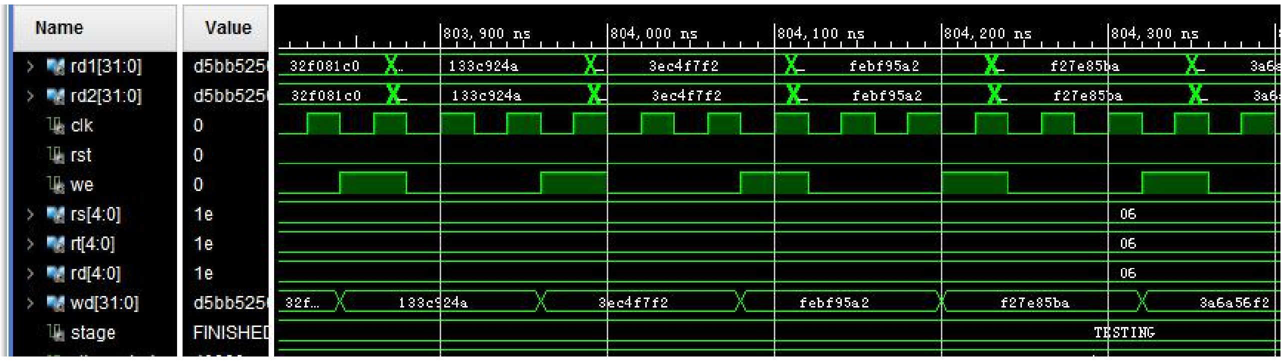


Figure 2.23 A few cases during the simulation

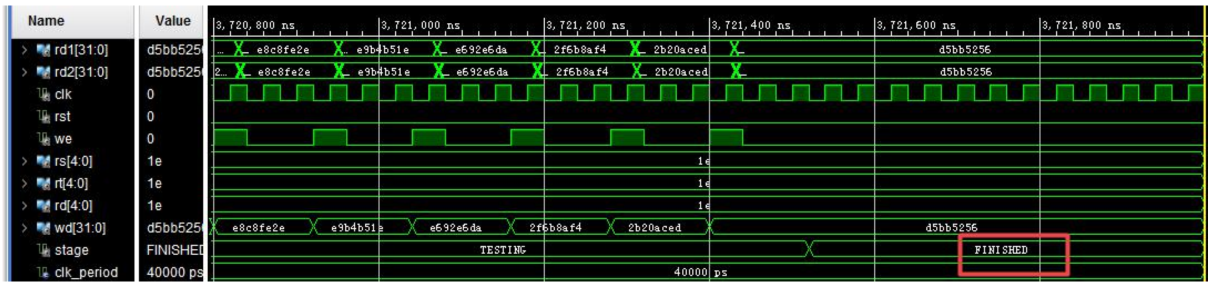
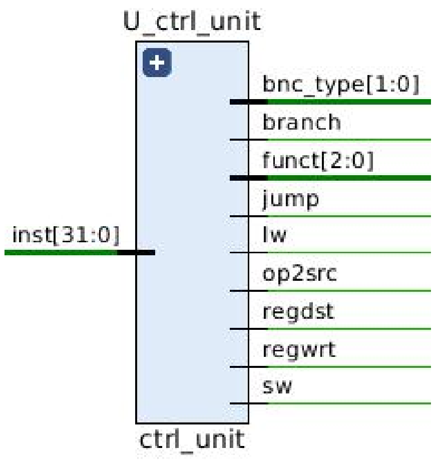


Figure 2.24 Last few cases. After all the cases finish, the state becomes “FINISHED”.

### 2.2.4 Control unit

#### 2.2.4.1 Implementation



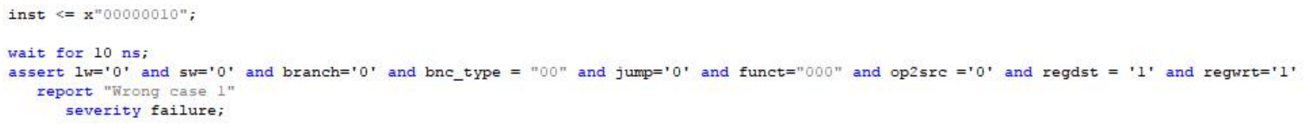
Signal list:

| | | |
|---|---|---|
| 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 |

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.

#### 2.2.4.2 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’.



```
inst <= x"00000010";

wait for 10 ns;
assert lw='0' and sw='0' and branch='0' and bnc_type = "00" and jump='0' and funct="000" and op2src ='0' and regdst = '1' and regwrt='1'
   report "Wrong case 1"
      severity failure;
```

### 2.2.4.3 Functional Simulation

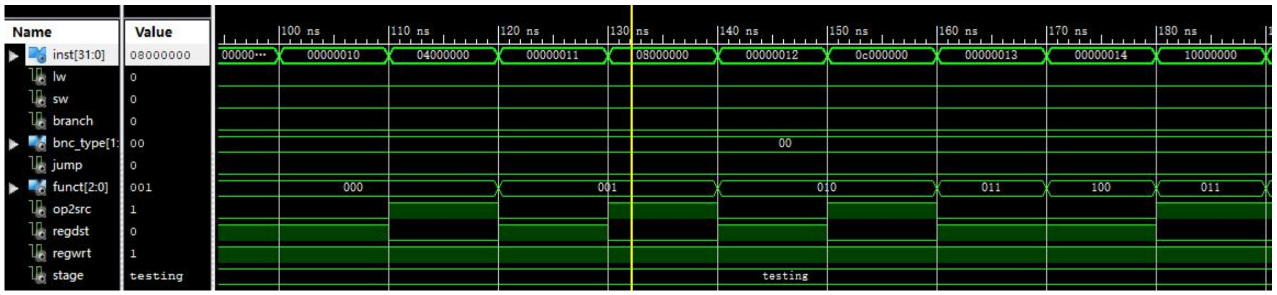


Figure 2.25 Testing first 9 conditions

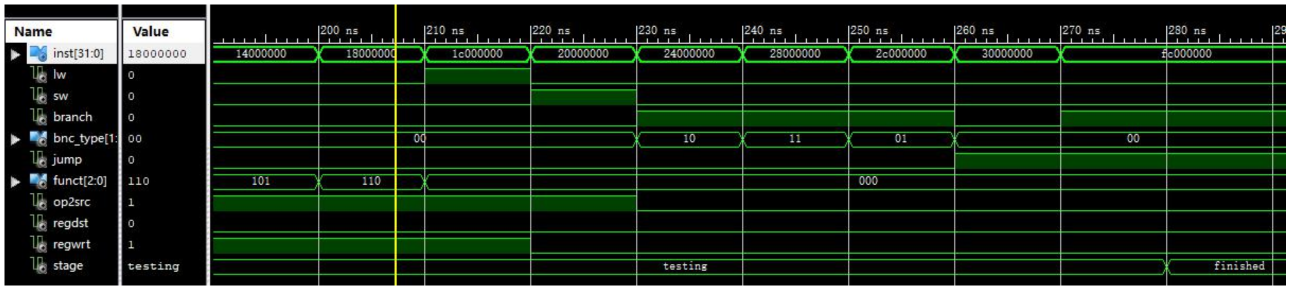


Figure 2.26 Testing latter 9 conditions

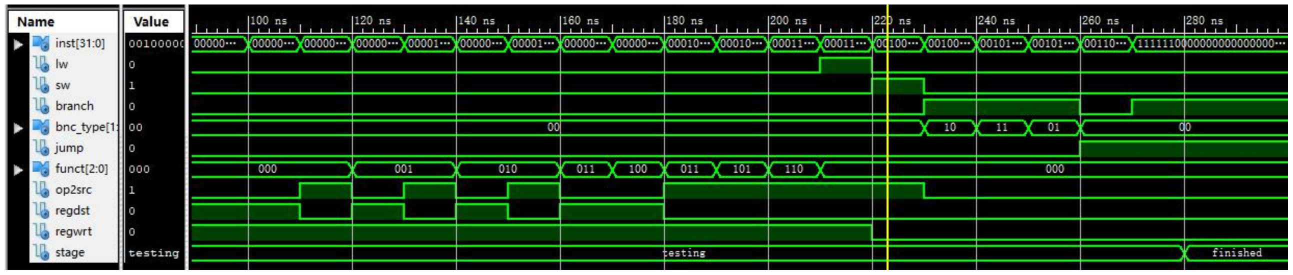


Figure 2.27 An overview with all cases passed.

### 2.2.4.4 Timing Simulation

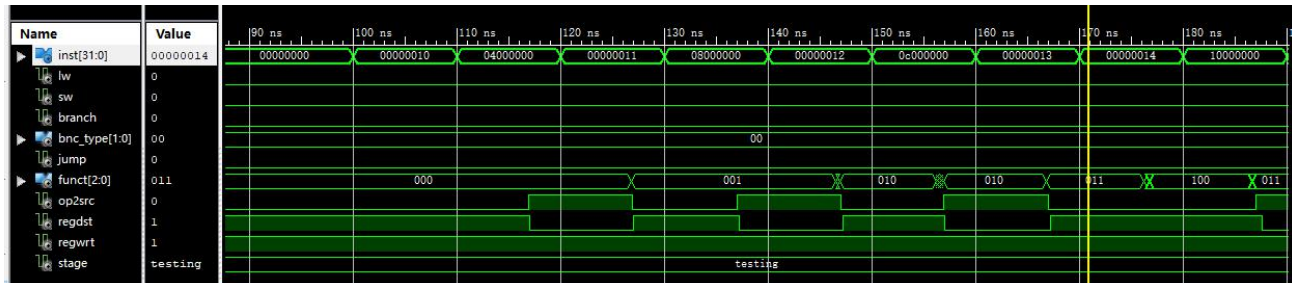


Figure 2.28 Testing first 9 conditions

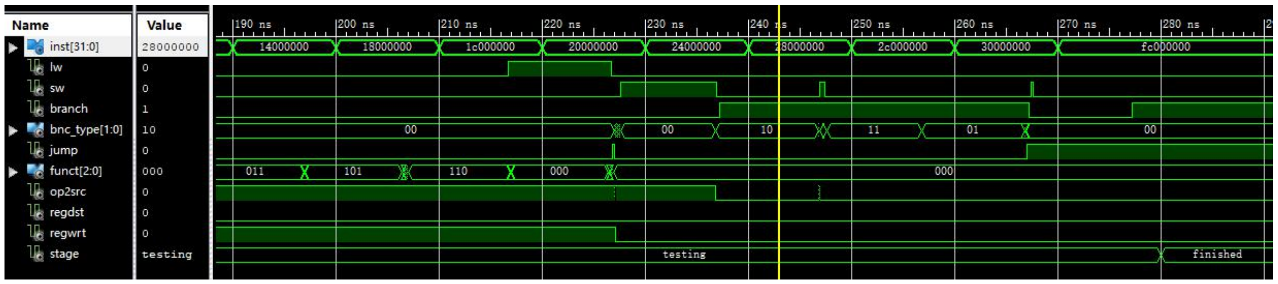


Figure 2.29 Testing latter 9 conditions

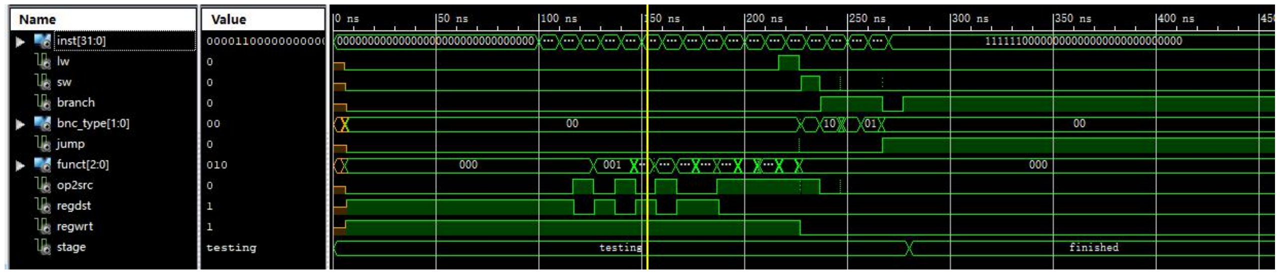
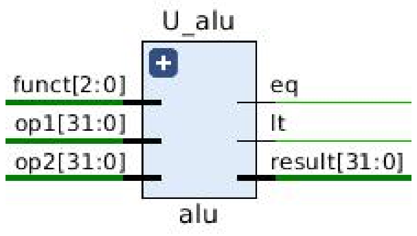


Figure 2.30 An overview with all cases passed.

### 2.2.5 ALU

#### 2.2.5.1 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 |

ALU operations based on funct signal:

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
```

#### 2.2.5.2 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.

#### 2.2.5.3 Functional Simulation

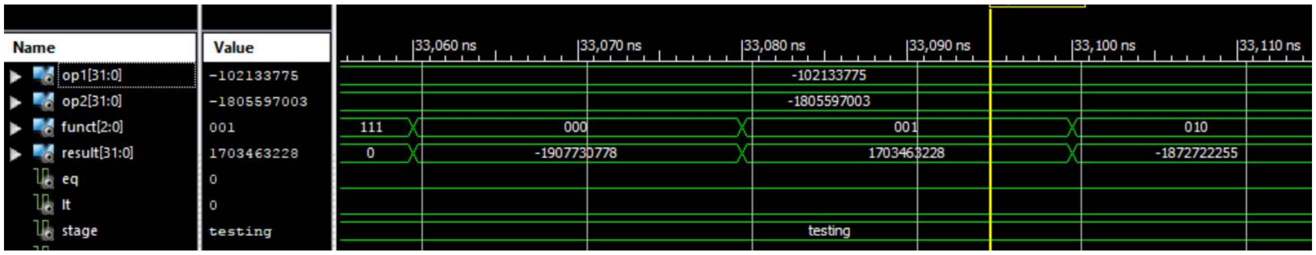


Figure 2.31 case "+" and "-" showed on decimal

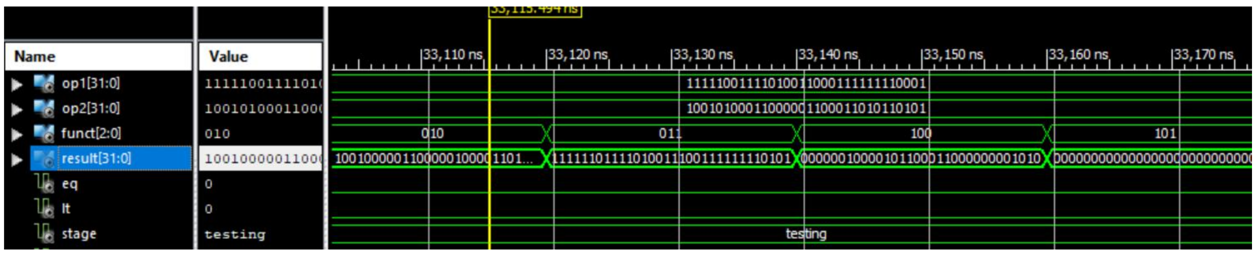


Figure 2.32 bit operation case showed on bits

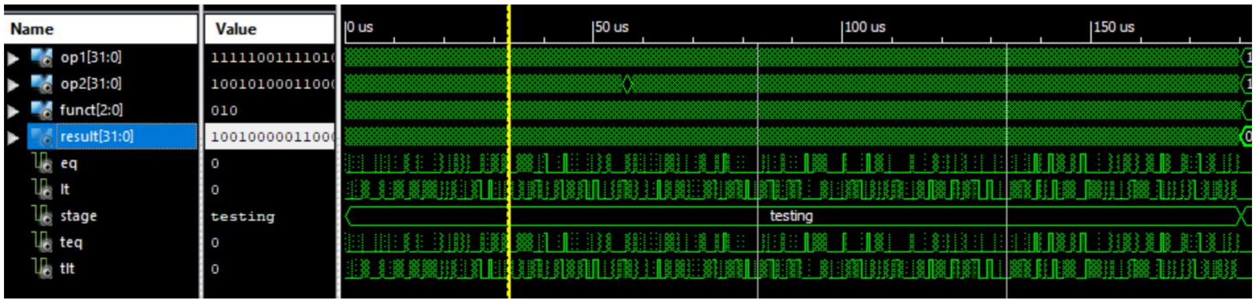


Figure 2.33 An overview. All cases passed

#### 2.2.5.4 Timing Simulation

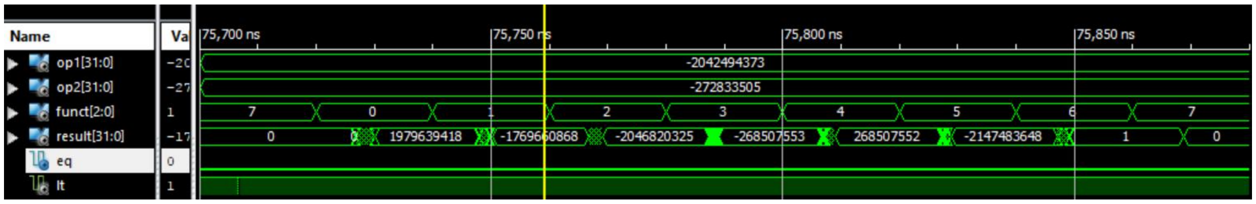


Figure 2.34 case “+” and “-” showed on decimal

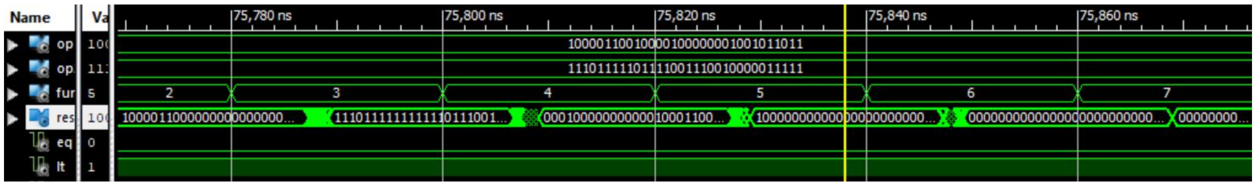


Figure 2.35 bit operation case showed on bits

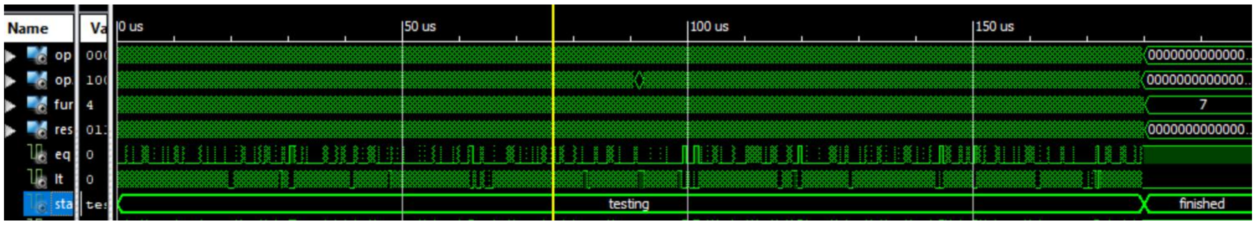
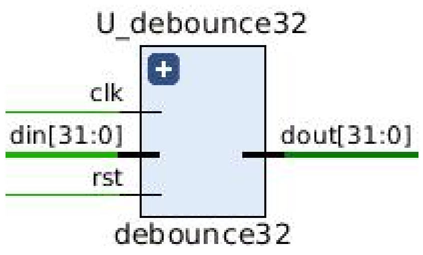


Figure 2.36 Overview. All cases passed

### 2.2.6 Debouncer

#### 2.2.6.1 Implementation



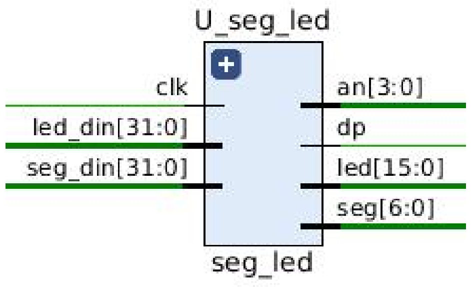
Signal list:

| | | |
|---|---|---|
| in | clk: | clock signal |
| in | rst: | reset signal, active-high |
| in | din: | input data |
| out | dout: | output data |

This is a unit for buttons and switches 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.

### 2.2.7 Seg_led

#### 2.2.7.1 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).

## 2.3 Processor Desgin and test

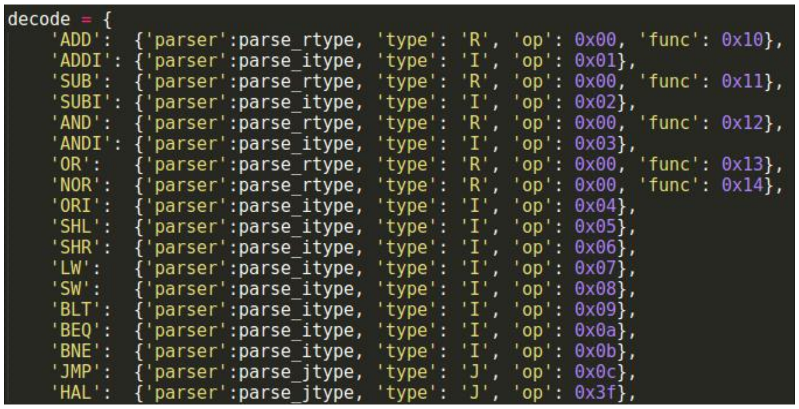
### 2.3.1 Assmebly comiler

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

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

During the trimming, all comment, blank line, extra white space or any other useless parts will be removed, to make the compiling easier. In addition, in our assembly labels are used to help mark the lines for branch and jump instructions. So in this stages, a label-to-address table will be created for compiling stages.

In the compiling stage, we use a decoding table and three separate parser for each of I/R/J type instruction, to convert assembly code to machine code.



```
decode = {
    'ADD':  {'parser':parse_rtype, 'type': 'R', 'op': 0x00, 'func': 0x10},
    'ADDI': {'parser':parse_itype, 'type': 'I', 'op': 0x01},
    'SUB':  {'parser':parse_rtype, 'type': 'R', 'op': 0x00, 'func': 0x11},
    'SUBI': {'parser':parse_itype, 'type': 'I', 'op': 0x02},
    'AND':  {'parser':parse_rtype, 'type': 'R', 'op': 0x00, 'func': 0x12},
    'ANDI': {'parser':parse_itype, 'type': 'I', 'op': 0x03},
    'OR':   {'parser':parse_rtype, 'type': 'R', 'op': 0x00, 'func': 0x13},
    'NOR':  {'parser':parse_rtype, 'type': 'R', 'op': 0x00, 'func': 0x14},
    'ORI':  {'parser':parse_itype, 'type': 'I', 'op': 0x04},
    'SHL':  {'parser':parse_itype, 'type': 'I', 'op': 0x05},
    'SHR':  {'parser':parse_itype, 'type': 'I', 'op': 0x06},
    'LW':   {'parser':parse_itype, 'type': 'I', 'op': 0x07},
    'SW':   {'parser':parse_itype, 'type': 'I', 'op': 0x08},
    'BLT':  {'parser':parse_itype, 'type': 'I', 'op': 0x09},
    'BEQ':  {'parser':parse_itype, 'type': 'I', 'op': 0x0a},
    'BNE':  {'parser':parse_itype, 'type': 'I', 'op': 0x0b},
    'JMP':  {'parser':parse_jtype, 'type': 'J', 'op': 0x0c},
    'HAL':  {'parser':parse_jtype, 'type': 'J', 'op': 0x3f},
```

Finally in the writing-back stage, the machine code will be written back to instruction memory (ins_mem.vhd). Please see our code (load_instructions.py) for more detail. It's well commented for reviewing.

### 2.3.2 Correctness of the Processor Design

The processor is tested with the sample code provided in the project description. We first hand calculate the expected results:

**Sample code 1:**

```
ADDI R1, R0, 7 // R1 = 7
ADDI R2, R0, 8 // R2 = 8
ADD R3, R1, R2 // R3 = R1 + R2 =15
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 should be 15. The following table show the state of all related registers after each line of code:

| register | Line 1 | Line 2 | Line 3 | Line 4 |
|---|---|---|---|---|
| R1 | 0 | 0 | 0 | 0 |
| R2 | 7 | 7 | 7 | 7 |
| R3 | 0 | 8 | 8 | 8 |

| | | | | |
|---|---|---|---|---|
| R4 | 0 | 0 | 15 | 15 |

**Sample code 2:**

*Too long to be show here*

The code is too long, here we just show the excepted results after each line of code:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| R2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 |
| R4 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| R4 | 0 | 0 | 14 | 14 | 14 | 14 | 4 | -1 | 0 | 10 | 10 | 10 | 10 | -11 | * | 0 |
| R5 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| MEM[12] | 0 | 0 | 0 | 0 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 |
| MEM[11] | 0 | 0 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |

* 10240

17 BEQ R5, R0, -2: R5=2 is not equal to R4=0. Go next line.

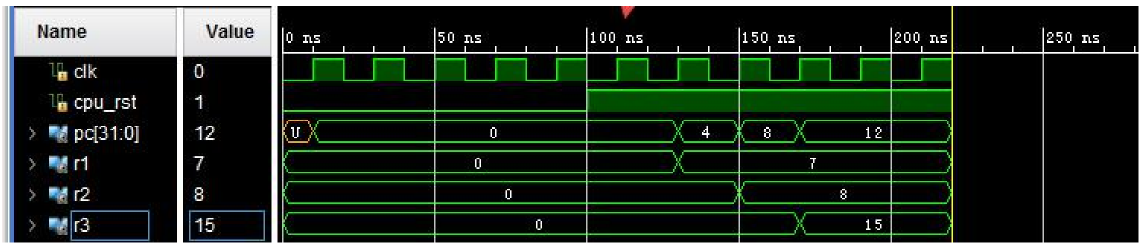
18 BLT R5, R4, 0: R5=2 is not less than R4=0. Go next line.

19 BNE R5, R4, 0: R5=2 is not equal to R4=0. Jump to pc+4+0 which is the next line.

20 JMP 20: It will jump to itself, which leads to an infinite loop.

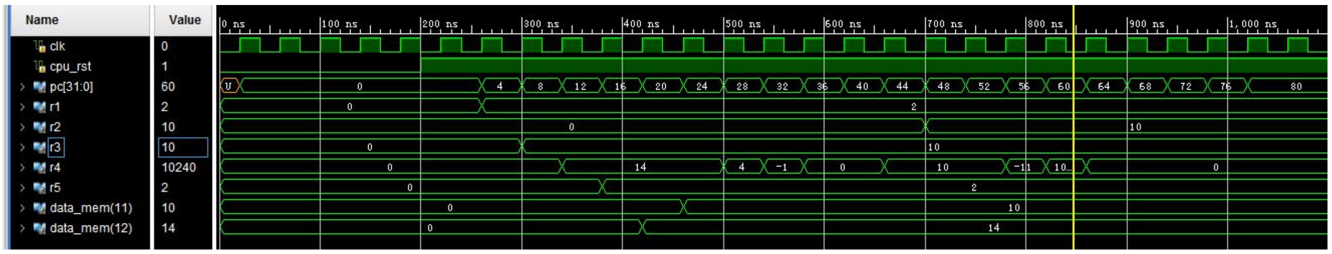
### 2.3.3 Functional simulation

**Sample code 1:** The operations of this code are described in 2.3.2. Now we run a functional simulation to verify it.



We can see that after the cpu_rst becomes high, the instructions in the inst_mem are executed one by one. Then 7 is written to r1, 8 is written to r2, and their sum 15 is written to r3, which is the same as expected.

**Sample code 2:** The operations of this code are described in 2.3.2, too. Now we run a functional simulation to verify it.

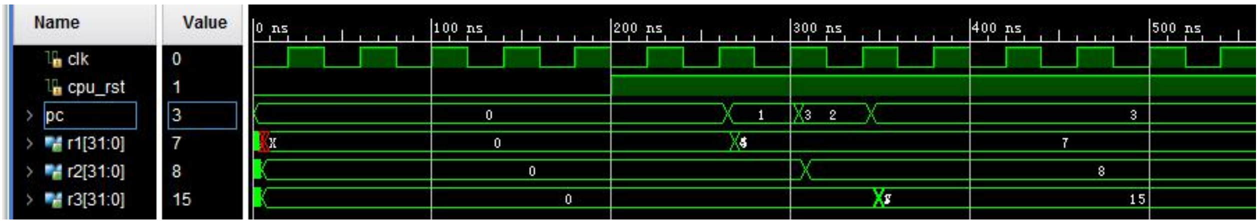


Compared to the description in 2.3.2, we find that the simulation results are correct.

### 2.3.4 Timing simulation

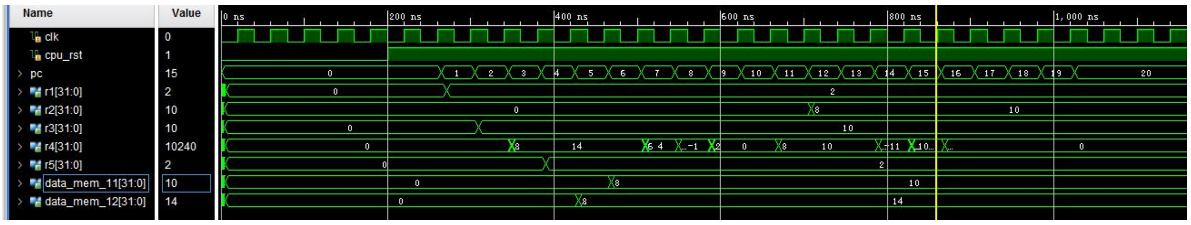
**Sample code 1:** Now we should check the timing simulation of this code. The sequence of the signals is the same as 2.3.3.

Notice that here we doesn't show the two least significant bits of pc (program counter), so the values it gives represent the sequence number of the instructions being executed instead of the addresses (0, 1, 2, 3… instead of 0, 4, 8, 12…)



We get the same results as the functional simulation. Notice that there are glitches at the time when the signals change, because of the race condition

**Sample code 2:** The configurations are the same with 2.3.3



Similarly to 2.3.4 (1), this timing simulation is in correspondence with its functional simulation. And we can also see delays and glitches in the wave form.

### 2.3.5 Timing Analysis and Optimization

#### 2.3.5.1 Critical path

The Figure 2.37 below shows the critical path of our original design. We can see that the longest delay happens when we do the LW operation. The data memory have to wait for the ALU calculating the address. Actually the address can be calculated directly without passing through the ALU, which can eliminate the extra delay occurs on those two multiplexers. So we change our original design into a new one as shown in the next picture, where the ALU is bypassed when doing LW/SW operation.

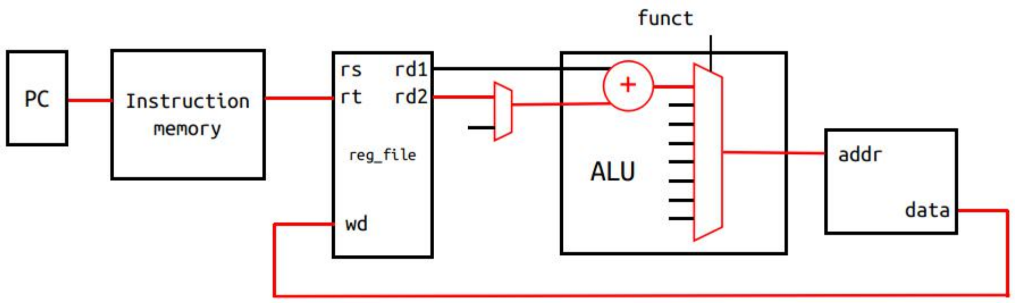


Figure 2.37 The critical path of the orignal design. Extra delay occurs in ALU.

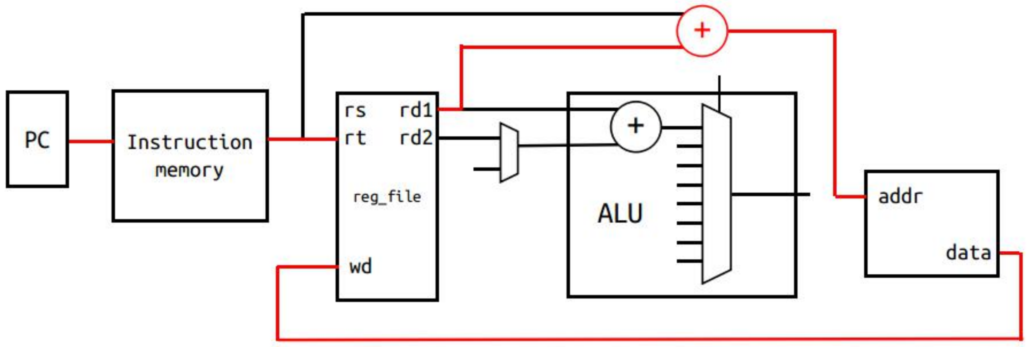
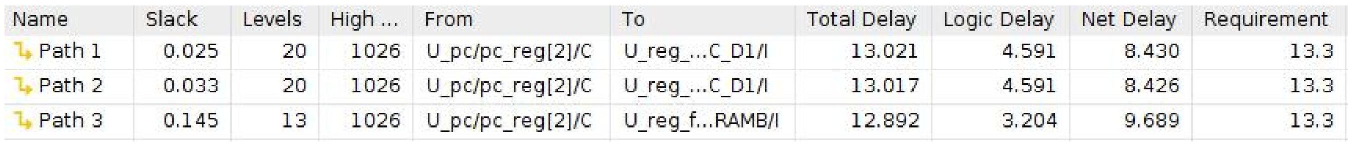


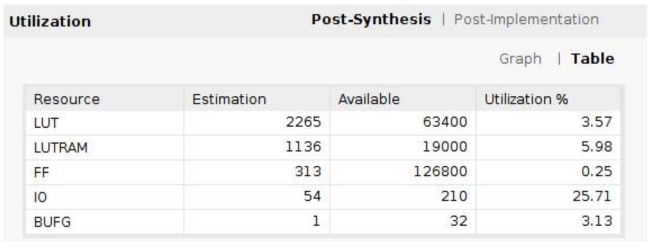
Figure 2.38 Optimized design where ALU is bypassed.

This change saves around 2 ns of delay, which reduce our total delay to **13.021ns**. Thus the highest frequency now can be **76.8MHz**.

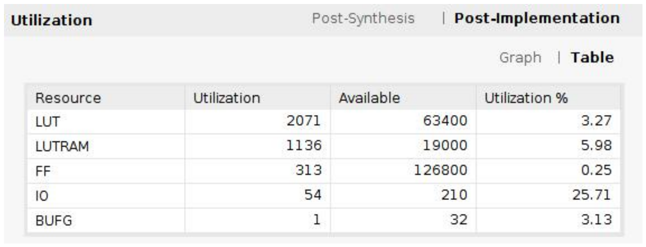


| Name | Slack | Levels | High ... | From | To | Total Delay | Logic Delay | Net Delay | Requirement |
|---|---|---|---|---|---|---|---|---|---|
| Path 1 | 0.025 | 20 | 1026 | U_pc/pc_reg[2]/C | U_reg_...C_D1/I | 13.021 | 4.591 | 8.430 | 13.3 |
| Path 2 | 0.033 | 20 | 1026 | U_pc/pc_reg[2]/C | U_reg_...C_D1/I | 13.017 | 4.591 | 8.426 | 13.3 |
| Path 3 | 0.145 | 13 | 1026 | U_pc/pc_reg[2]/C | U_reg_f...RAMB/I | 12.892 | 3.204 | 9.689 | 13.3 |

### 2.3.5.2 Resource Utilization


Utilization — **Post-Synthesis** | Post-Implementation — Graph | **Table**

| Resource | Estimation | Available | Utilization % |
|---|---|---|---|
| LUT | 2265 | 63400 | 3.57 |
| LUTRAM | 1136 | 19000 | 5.98 |
| FF | 313 | 126800 | 0.25 |
| IO | 54 | 210 | 25.71 |
| BUFG | 1 | 32 | 3.13 |


Utilization — Post-Synthesis | **Post-Implementation** — Graph | **Table**

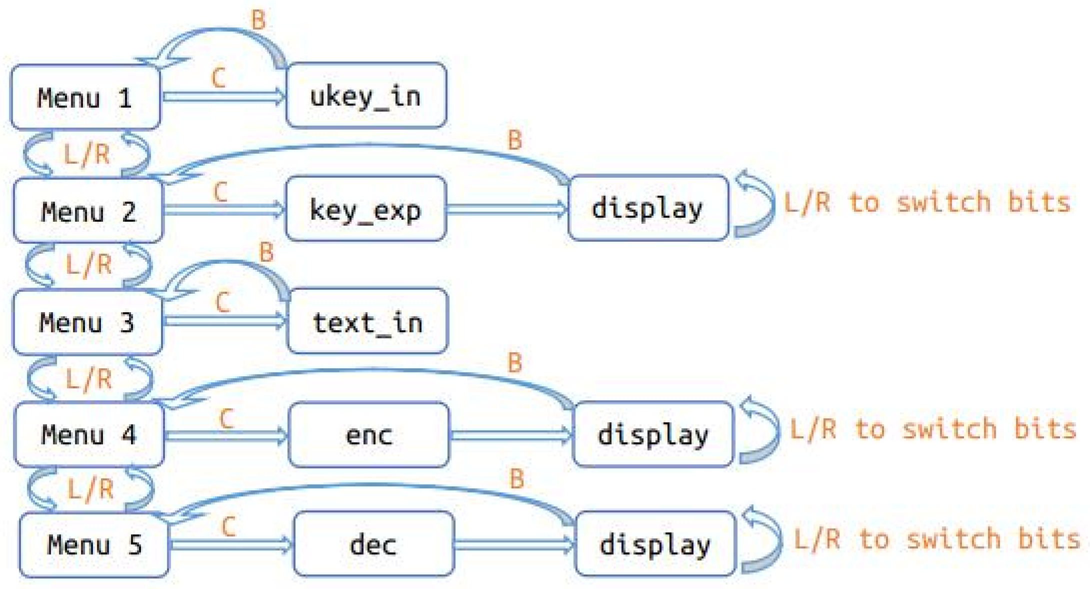
| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 2071 | 63400 | 3.27 |
| LUTRAM | 1136 | 19000 | 5.98 |
| FF | 313 | 126800 | 0.25 |
| IO | 54 | 210 | 25.71 |
| BUFG | 1 | 32 | 3.13 |

Our implementation takes **3.27%** of the LUTs. We can reduce it by setting the optimization level to HIGH when synthesize and implement the design. However, it will increase the critical path delay. So in our case, we choose higher clock frequency rather than low resource utilization.

## 3. RC5 Implementation

### 3.1 Assmebly code

#### 3.1.1 FSM diagram



This picture shows the diagram of our RC5 code. Thanks to the input/output interface of the processor, our program supports changing ‘ukey’ and ‘din’ on-the-fly. We wrote an FSM in our RC5 code, to let the user choose which subprogram to run independently. There are 4 buttons: LEFT(L), RIGHT(R), CONFIRM(C) and BACK(B). After power-on, the program will be in Menu 1. User can use L/R to switch between different menus and C to execute the subprogram.

**UKEY_IN**: This is the subprogram for sending 128-bit ukey. We can use switches and btnC to input 16-bit at a time and use L/R btn to switch between different bits. The ukey will be stored into data memory [50 to 53] in little-endian.

**KEY_EXP:** This is the key expansion function. Round-key will be generated based on the ukey in the data memory. The expanded round-key will be stored into data memory [0 to 25].

**TEXT_IN:** This is the subprogram for sending 64-bit input-text. We can use switches and btnC to input 16-bit at a time and use L/R btn to switch between different bits. The input-text will be stored into data memory [54 to 55] in little-endian.

**ENCRYPTION:** This is the encryption function. 64-bit Cypher-text encrypted from input-text will be stored into data memory [30 to 31] in little-endian.

**DECRYPTION:**This is the decryption function. 64-bit Plain-text decrypted from input-text will be stored into data memory [32 to 33] in little-endian.

**DISPLAY:** This subprogram will show the output-text on the 7-segment display. User can use L/R to switch from different bits.

Please check the source 'rc5_optimized.asm' for more details about how we implement the RC5 algorithm in assembly. Our source code is well commented for reviewing. Also in the next few sections, we will talk about the assembly implementation of rotation and XOR.

### 3.1.2 Time complexity analysis

The critical part in the RC5 implementation is how fast we can run the key expansion, encryption and decryption in terms of cycles. So in this section we will briefly analyze the time complexity of rc5.

The running cycles of **key expansion** is **$O(T_L + T_S + (T_{rot} + C_{exp})*78 + C_0)$**, where $T_L$ is for L-Initialization, $T_S$ is for S-Initialization, $T_{rot} + C_{exp}$ is for one round of skey-Expansion, $C_0$ is for other constant time operations.

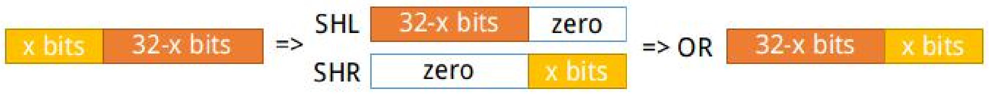
The running cycles of **encryption/decryption** is **$O((T_{rot} + T_{xor} + C_1)*2*12 + C_0)$**, where $(T_{rot} + T_{xor} + C_1)*2$ is for one round of encryption/decryption, $C_0$ is for other constant time operations.

Notice that the constant factors 78 and 24 are relatively large, so reducing the cycles spend on rotation($T_{rot}$) and XOR($T_{xor}$) operation will be very helpful to reduce the total running cycles.

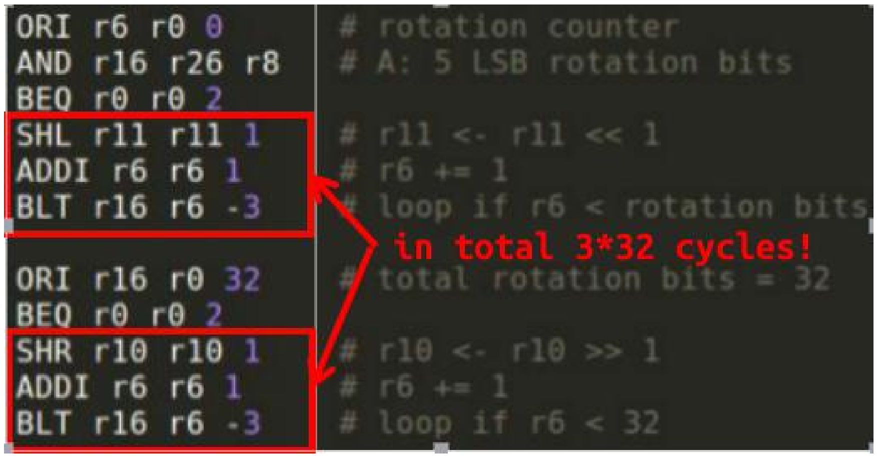
### 3.1.3 ROTATE optimization

In this project, we only have I-type shift instructions, which is a huge limitation.

A trivial solution to implement left rotation using I-type shift is:

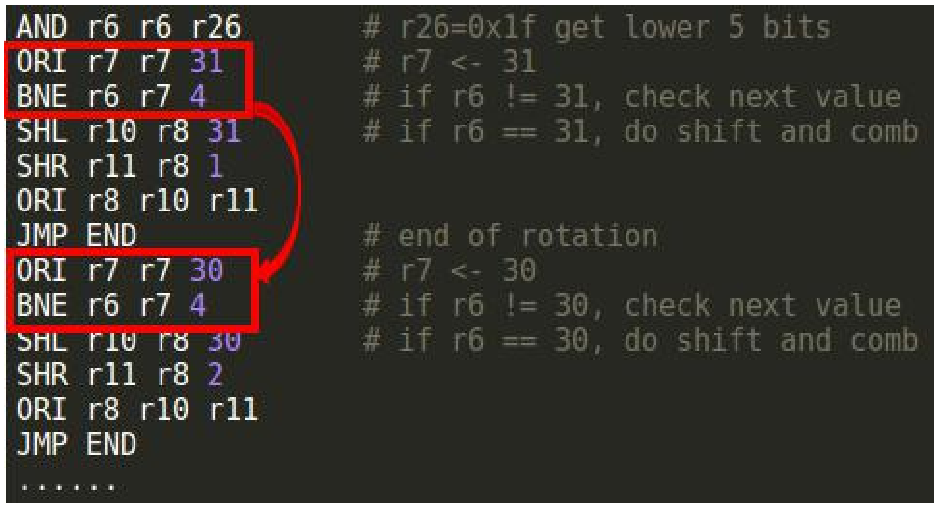


Suppose the rotation amount[1] is x, we can recursively shift left by 1 for x times. Then right shift by 1 for 32-x times. Finally combine them together using OR operation. This method will take at least 32*3=96 cycles per rotation. Below is an example code of rotl.



```
ORI  r6 r0 0          # rotation counter
AND  r16 r26 r8       # A: 5 LSB rotation bits
BEQ  r0 r0 2
SHL  r11 r11 1        # r11 <- r11 << 1
ADDI r6 r6 1          # r6 += 1
BLT  r16 r6 -3        # loop if r6 < rotation bits
                        in total 3*32 cycles!
ORI  r16 r0 32        # total rotation bits = 32
BEQ  r0 r0 2
SHR  r10 r10 1        # r10 <- r10 >> 1
ADDI r6 r6 1          # r6 += 1
BLT  r16 r6 -3        # loop if r6 < 32
```

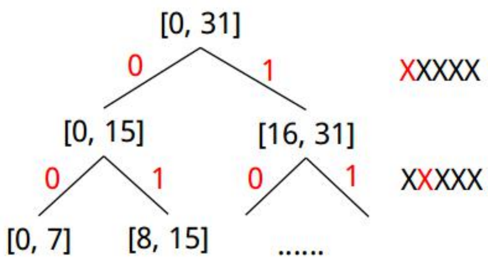
A smarter way is sequentially comparing the rotation amount with 0 to 31 and shifting the register by required amount directly. Because we don't have instructions to compare a register with an immediate number, so each comparison will take 2 cycle. Thus, the running time per rotation in average is 2*16+4=36 cycles.



```
AND r6 r6 r26       # r26=0x1f get lower 5 bits
ORI r7 r7 31        # r7 <- 31
BNE r6 r7 4         # if r6 != 31, check next value
SHL r10 r8 31       # if r6 == 31, do shift and comb
SHR r11 r8 1
ORI r8 r10 r11
JMP END             # end of rotation
ORI r7 r7 30        # r7 <- 30
BNE r6 r7 4         # if r6 != 30, check next value
SHL r10 r8 30       # if r6 == 30, do shift and comb
SHR r11 r8 2
ORI r8 r10 r11
JMP END
......
```

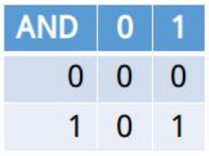
[1] The rotation amount is the lower 5 bits of the operand, so it's in range from 0 to 31.

An optimal way to do the rotation is using binary search, which takes only 13 cycles per rotation! We know that the rotation amount is a 5-bit number, so we can determine whether it's in [0,15] or [16,31] by checking its 5th bit. Then use 4th bit to reduce the range again, so on and so forth. Picking 1 bit (AND) and branch if it's 1 (BEQ) takes only 2 cycles. So the total running time is 2*log(32)+4=14 cycles. The assembly code to implement a binary search rotation is similar to the sequential comparison, but much longer (150+ lines). Please check our source code as it's too long to be shown here.

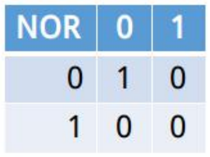


### 3.1.4 XOR optimization

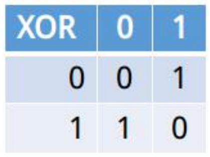
We don't have XOR instruction, so we have to implement it using AND, OR, NOR. If we looking at the truth table of them, we can easily find an efficient solution: **A XOR B = (A NOR B) NOR (A AND B)**, which takes only 3 cycles.



| AND | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |



| NOR | 0 | 1 |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 0 | 0 |



| XOR | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

## 3.2 Testbench

To meet the requirement, we tested 1000 random cases for each of key_expansion, encryption and decryption. We also tested another 10 random ukey, each with 100 cases of encryption and decrpytion. So there were in total 4000 cases tested in our testbench.

The random cases along with their expected result were generated from a python script (rc5.py), and stored in a text file for testbench to read. Assertion statement was used in the testbench for automatically checking.

The testbench for the timing simulation is similar, except that the number of cases is reduced. We tested 5 random cases for each of key_expansion, encryption and decryption. Then tested another 2 random ukey, each with 5 cases of encryption and decrpytion.

```
-- read result from 7-segment display
encryption(63 downto 32) <= display;
btn(3) <= '1'; wait for btn_delay; btn(3) <= '0'; wait for btn_delay;
encryption(31 downto 0) <= display;
wait for clk_period;

readline(file_VECTORS, v_ILINE);
read(v_ILINE, expected);
assert encryption = expected
    report "wrong"
        severity failure;
```

## 3.3 Functional simulation

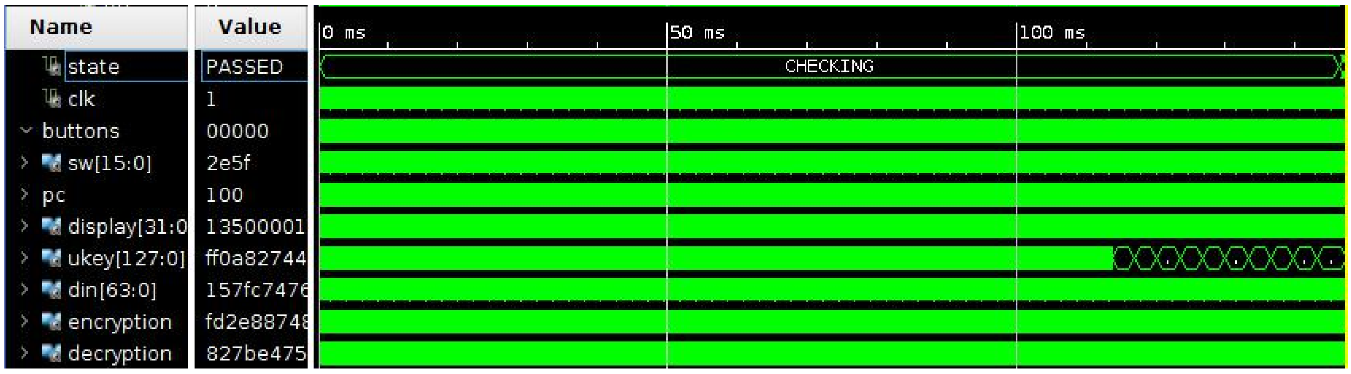


Figure 3.1 All 4000 cases. 0 to 110ms is 3*1000 cases for each of key expansion, encryption and decryption. After 110ms is the wave of 10 different ukeys each with 100 dins.

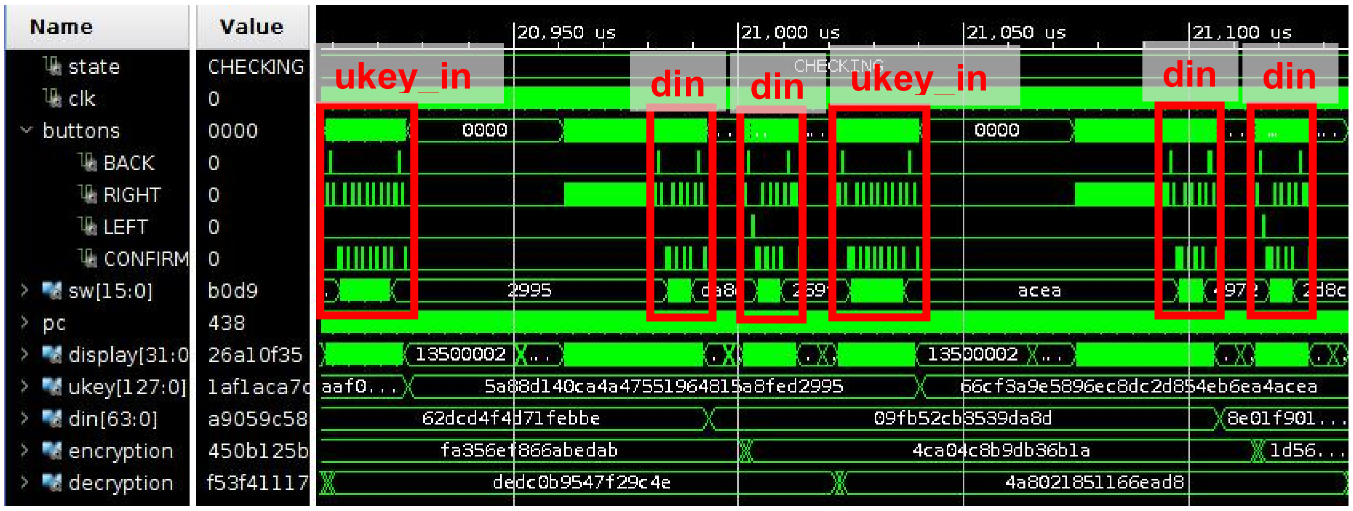


Figure 3.2 This screenshot shows 2 random ukey each with 1 din for encryption and 1 din for decryption. The red box shows how the testbench input data using buttons and switches.

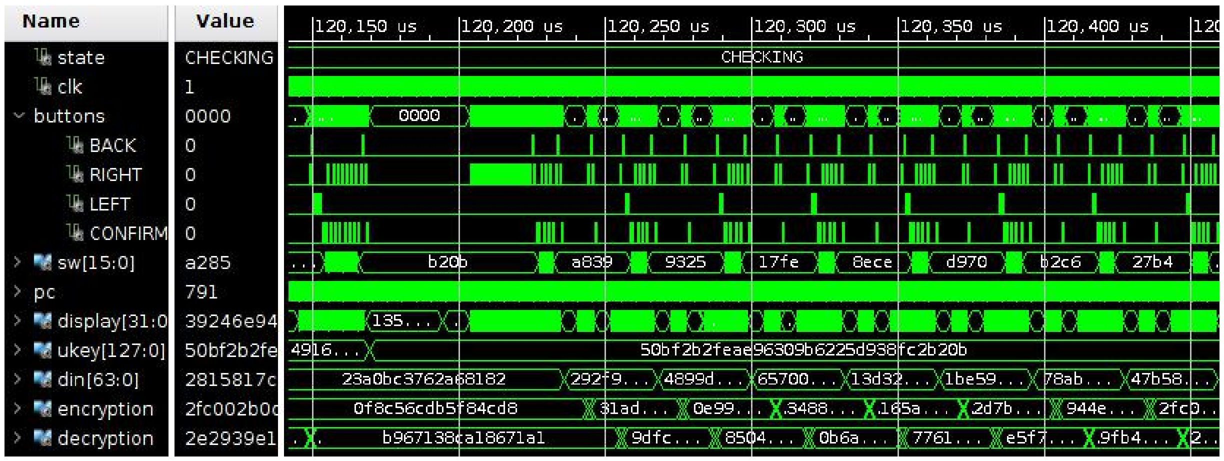


Figure 3.3 Screenshot of 1 ukey followed by 100 din for each encryption and decryption.

Now let's figure out how much cycles does it take to do the key_expansion, encryption and decryption. Note that the clock period is 10 ns.

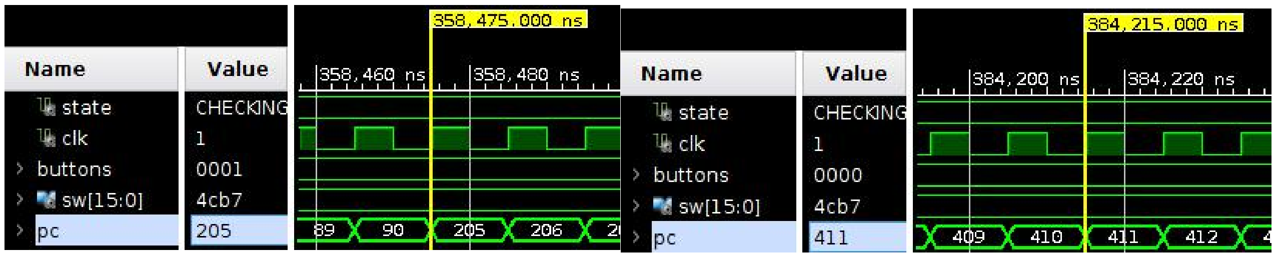


Figure 3.4 Key_expansion starts at line 205 and ends before line 411.

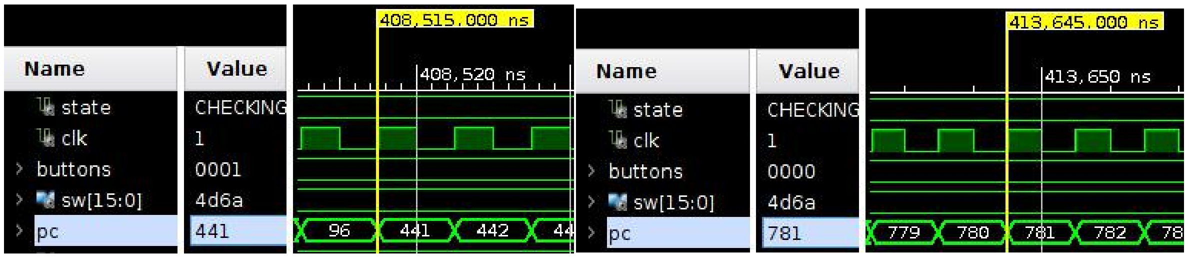


Figure 3.5 Encryption starts at line 441 and ends before line 781.

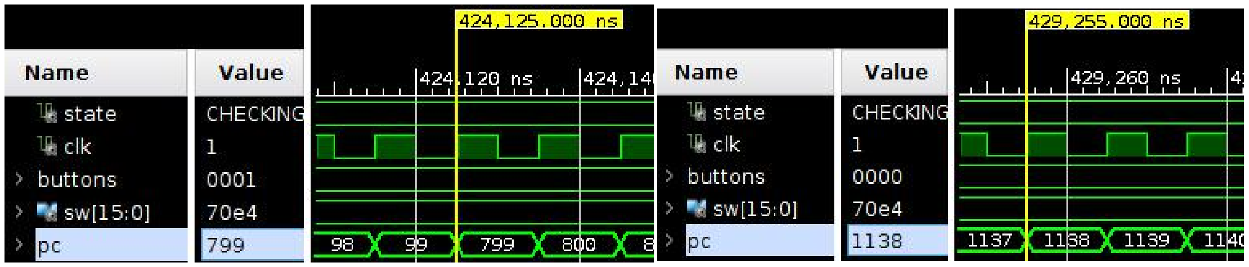


Figure 3.6 Decryption starts at line 799 and ends before line 1138.

Then we can get the following table:

| function | Start at | End at | Cycles |
|---|---|---|---|
| key_expansion | 358475 | 384215 | 2574 |
| encryption | 408515 | 413645 | 513 |
| decryption | 424125 | 429255 | 513 |

Thanks to the optimization of rotation operation, our RC5 implementation is very efficient.

## 3.4 Timing simulation

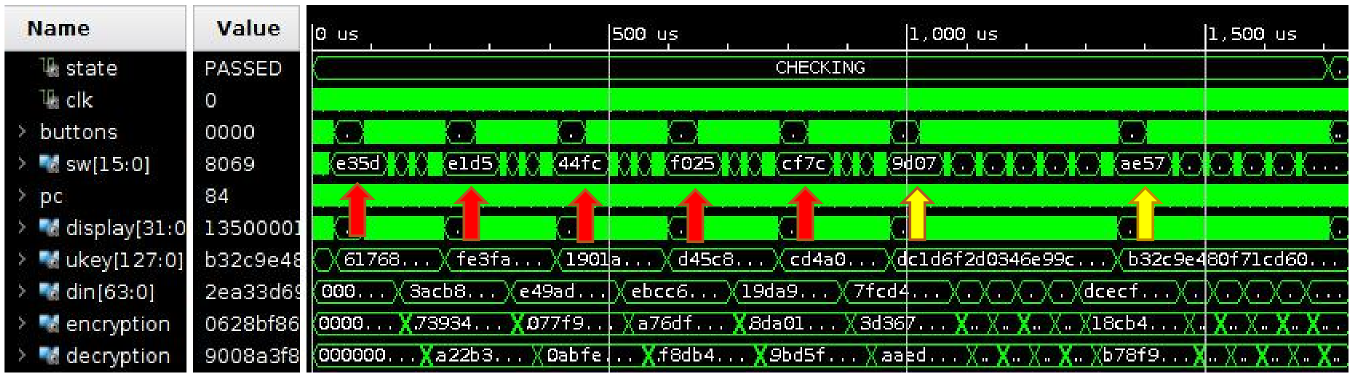


Figure 3.7 All 25 cases. The red arrow shows 5 ukey each followed by 1 encryption and 1 decryption. The yellow arrow shows 2 ukey each followed by 5 encryption and decryption.

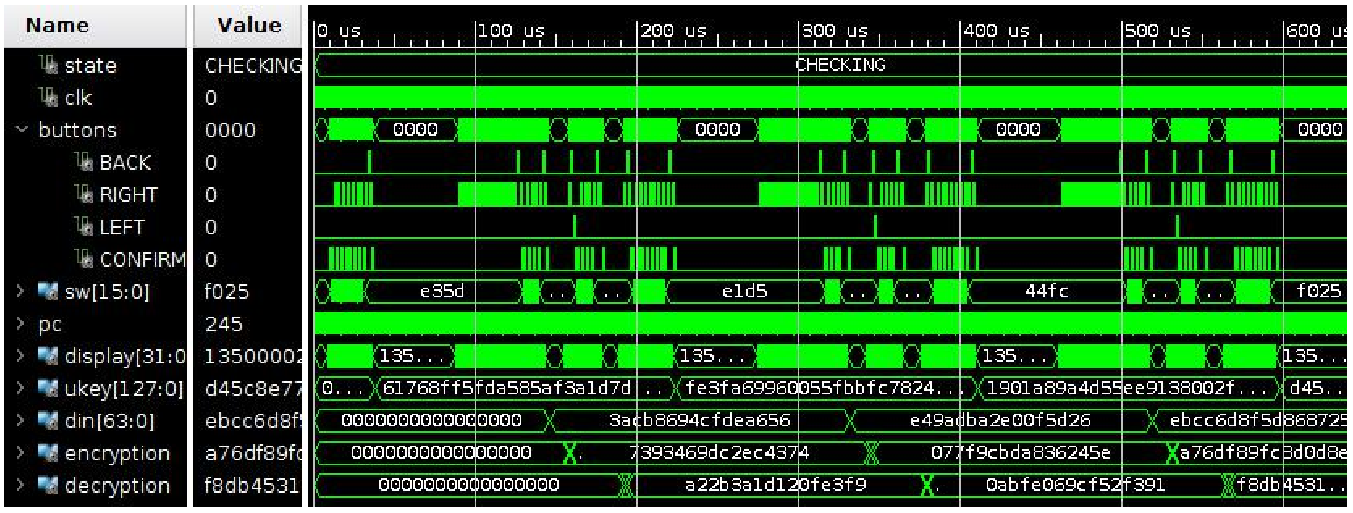


Figure 3.8 1 One ukey followed by 1 encryption and 1 different decryption.

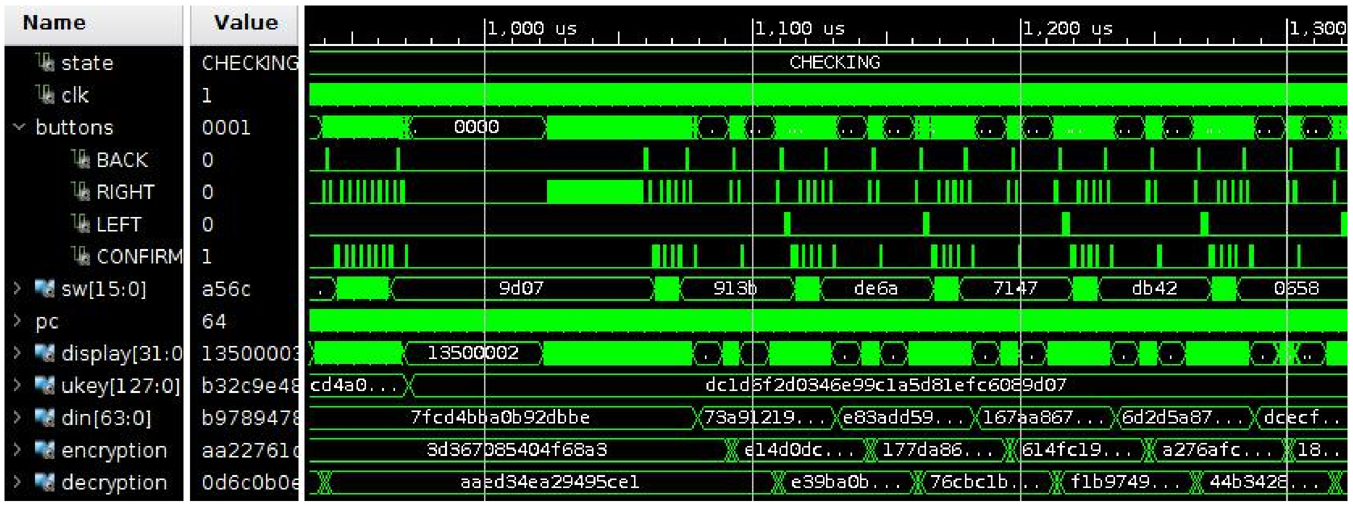


Figure 3.9 One ukey followed by 5 encryption and decryption.

## 4. FPGA Implementation

### 4.1 Clock Configuration

### 4.2 Board Test

## 5. Summary

Put your code in GitHub repository and share it with the TAs. (mos283,cm4233)

2. Put your processor source code and assembly code in a zipped folder.

3. Your report in pdf format including your

4. design block diagram,

5. simulation screen shots

6. performance and area analysis of design

7. description of RC5 implementation in assembly
8. description of processor interfaces (how do you provide inputs and display results)
9. details about how you verified your overall design
10. demo video of the processor executing RC5 algorithms on FPGA.(5 to 10 minutes)