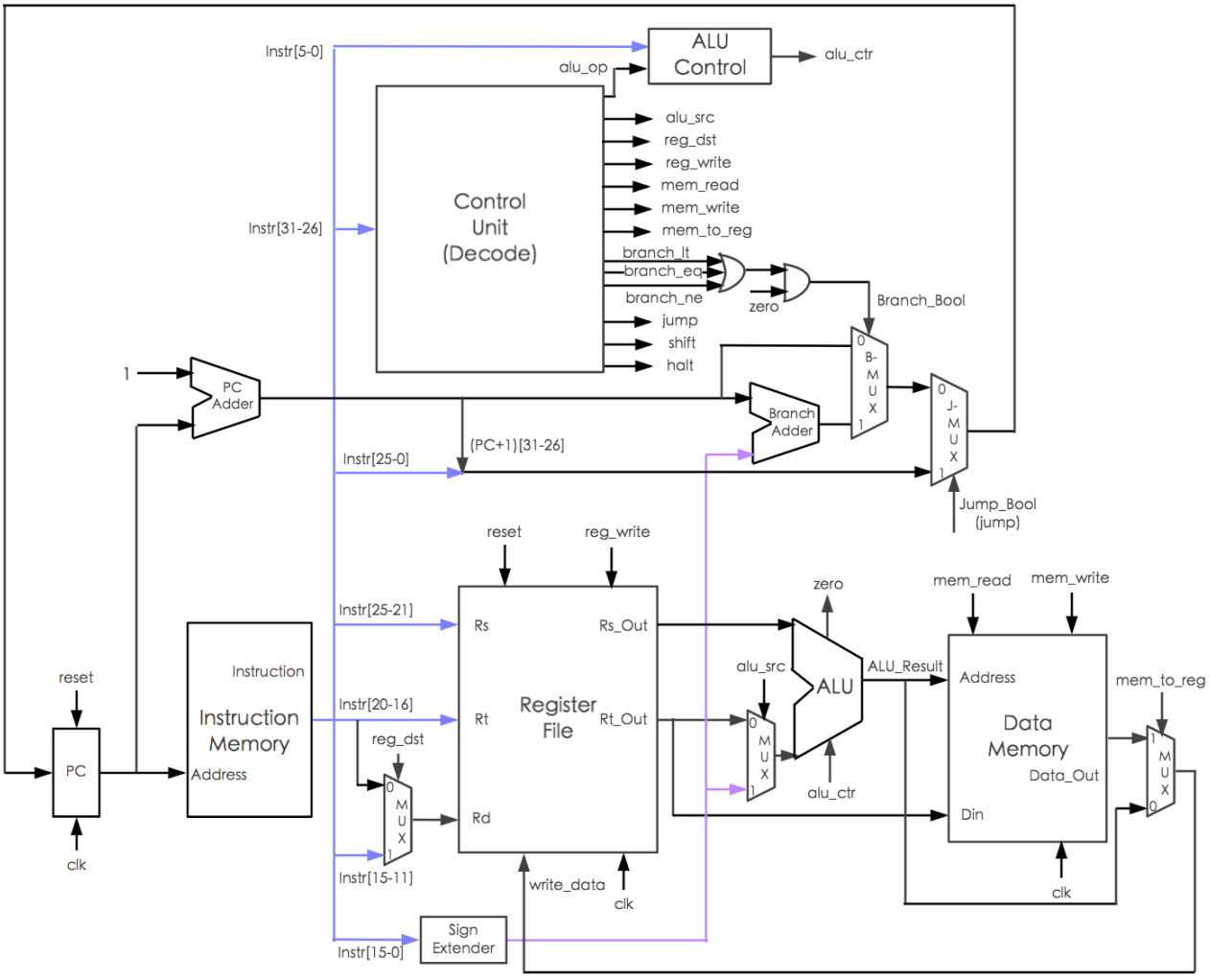
ADVANCED COMPUTER HARDWARE DESIGN

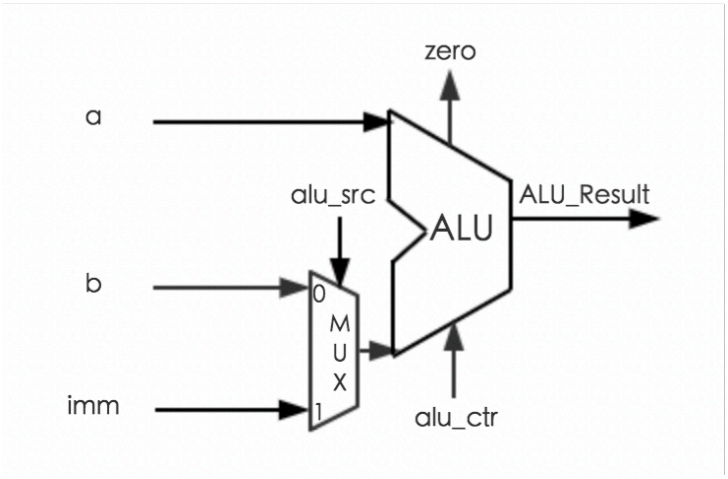
Processor Design
&
RC5 Implementation

Chi Zhang
Hao Dong
Jian Song
Wen Dai
Zijun Ma

Design block diagram



ALU Design

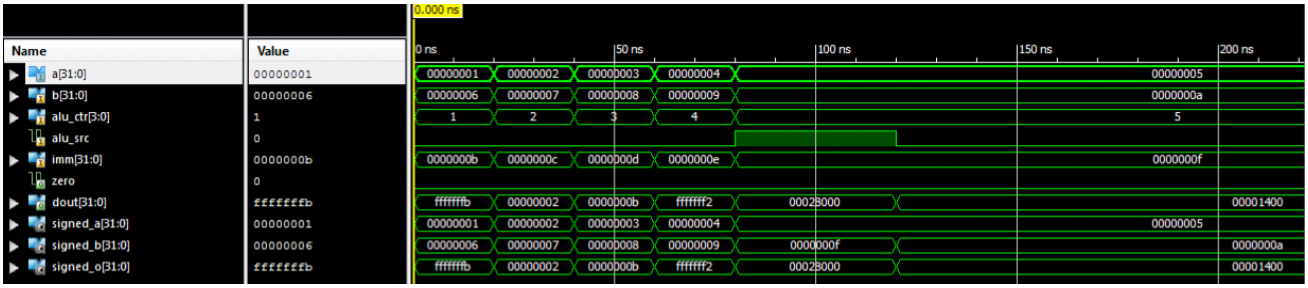


For ALU section, our design is aim to include all the operations except JMP command (because it involves PC calculation) in the ALU. Those operations have Opcode respectively 00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0A, 0B, 0C, 3F.

In our design, the input of ALU is a(Rs), b(Rt), Imm, alu_ctr, alu_src. For these inputs, a(Rs), b(Rt) are the outputs of Register file section, Imm is the last 16 bits of the instruction, alu_ctr is the output of ALU control unit, which will select calculation operation according to the opcode, alu_src is the output of control unit and used to select the actual input signed_b for operation, if alu_src = 1, which means the operation is in I type or J type, sign-extended Imm will be given to the signal signed_b, otherwise the signed value of b(Rt) will be given to signed_b. For the basic calculation (+, -, &, or, nor), we just need Rs and Rt/Imm for calculation. For the advanced calculation (LW, SW, BLT, BEQ, BNE), we need to use Rs, Rt for condition and Imm for address calculation.

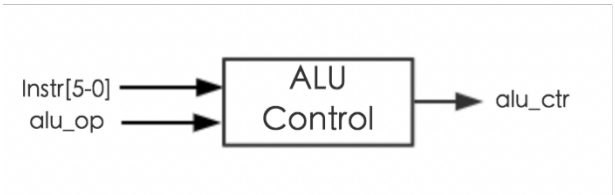
After calculation, ALU calculation result is given to dout, and another output zero will go to '1' if dout is X"0000000000000000". It will be used for branch operation.

Simulation:



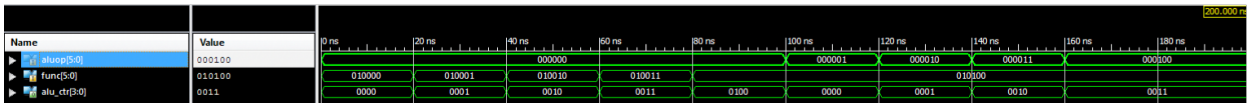
Control Unit Design

ALU Control

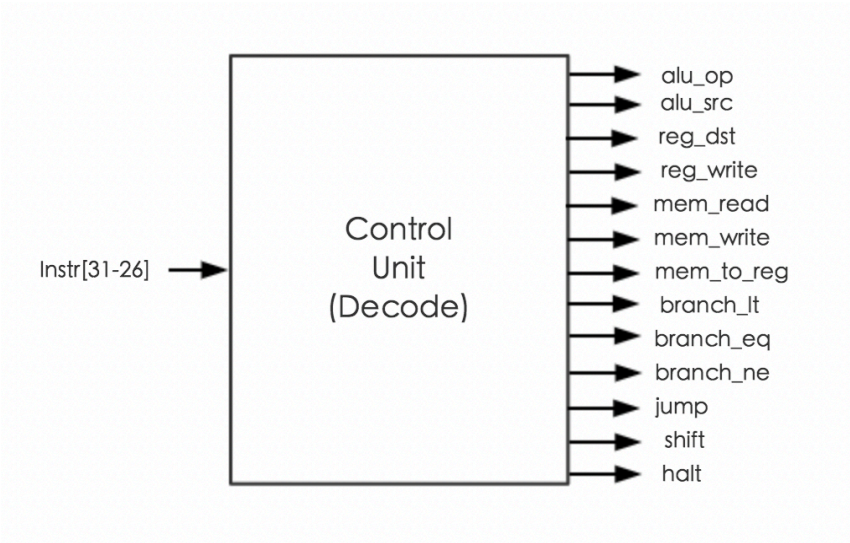


ALU control component was used to decide which kind of computation will be executed in the ALU. The input signal is alu_op derived from the control unit. And for some operations sharing the same alu_op, functional code (Instr[5-0]) is used for further specification. The out signal is basically a combination of them, which represents each of computation, for example, add, sub, beq, etc.

Simulation:

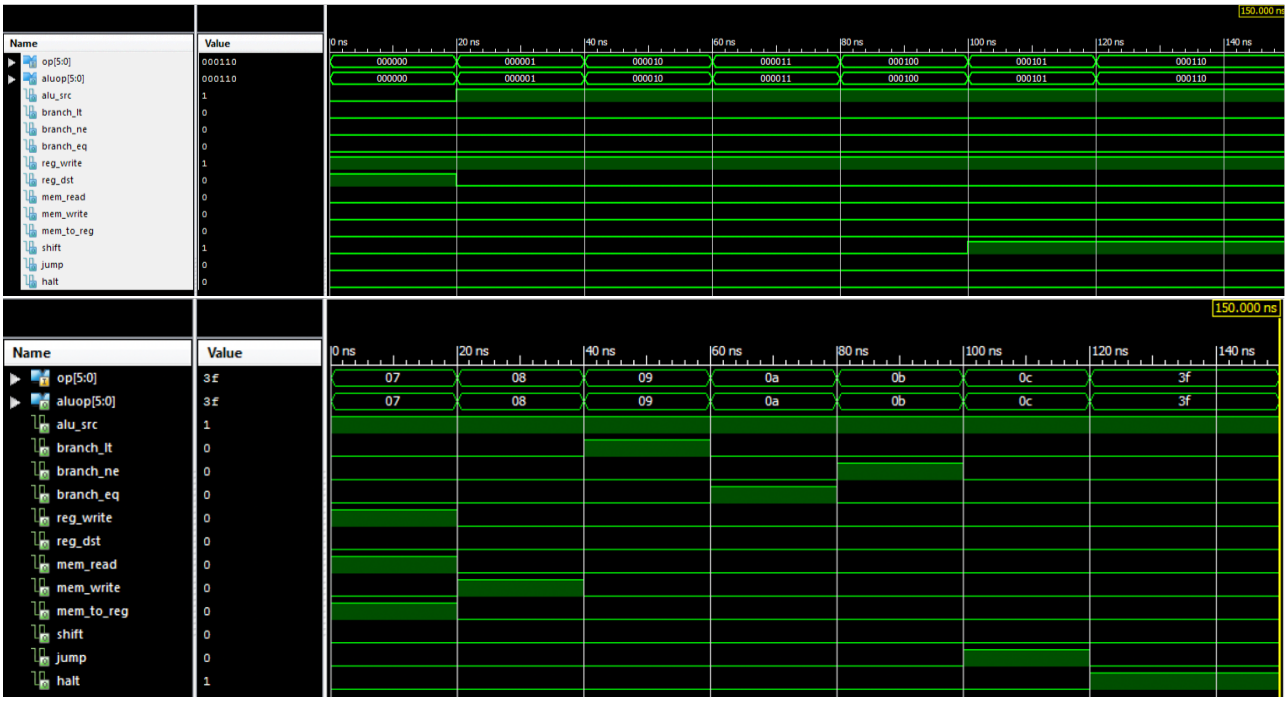


Instruction Decode



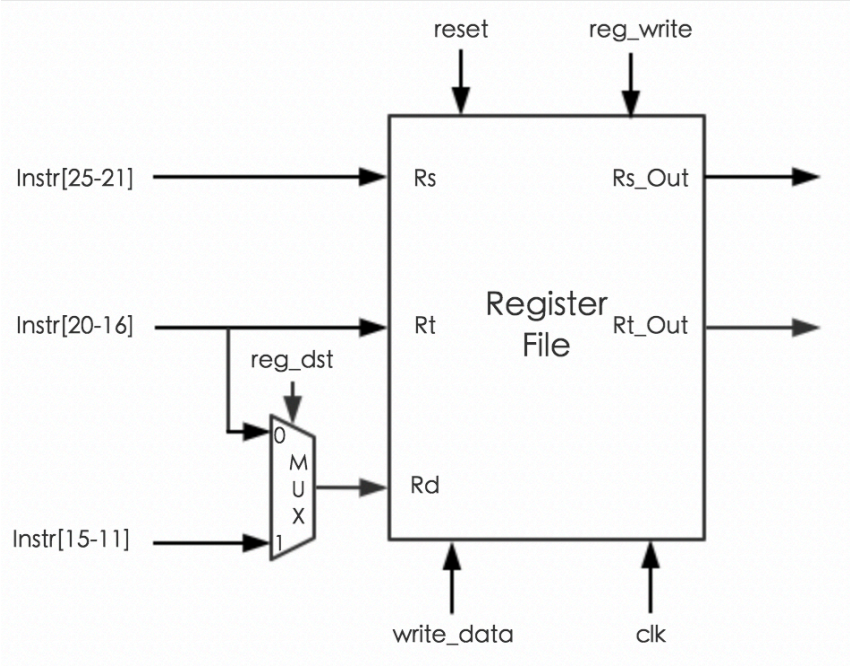
This component was used for parsing the operation code and generating some control signals, the input signal is apparently the opcode (Instr[31-26]). Then according to the table 2 from project instructions, we can decide the value for all the control signals. For example, for the operation add and addi, a write operation should be executed, so a reg_write signal will be set as “1”, in order to enable this operation. The same goes for other control signals such as branch and jump.

Simulation:



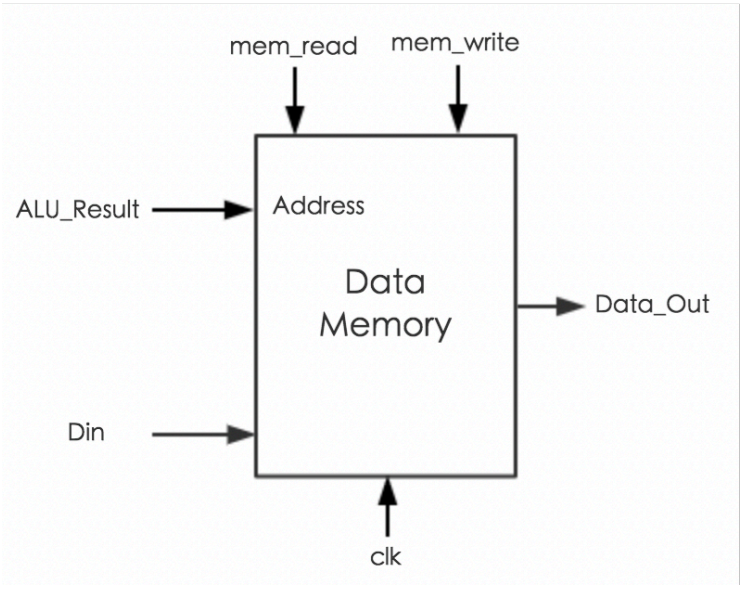
Complete Processor Design

Register File



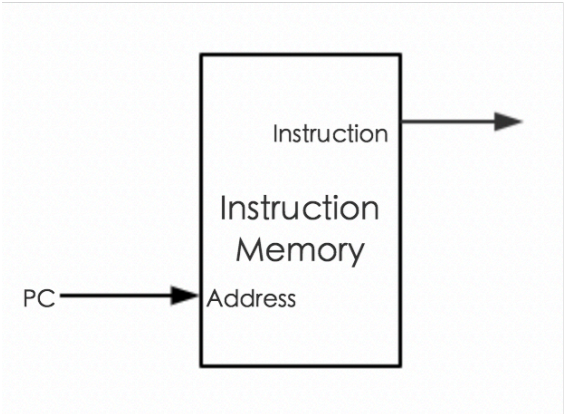
This component was used as registers, which will be used for further storage and computation. Here we defined 32 32-bit registers, each with an initial value X“00000000”. For one clock, two read operations and one write operation may be executed. For read operation, it’s controlled by the address signal and can be read at any time, while for write operation, it should be executed with the rising clock signal and write_enable signal.

Data Memory



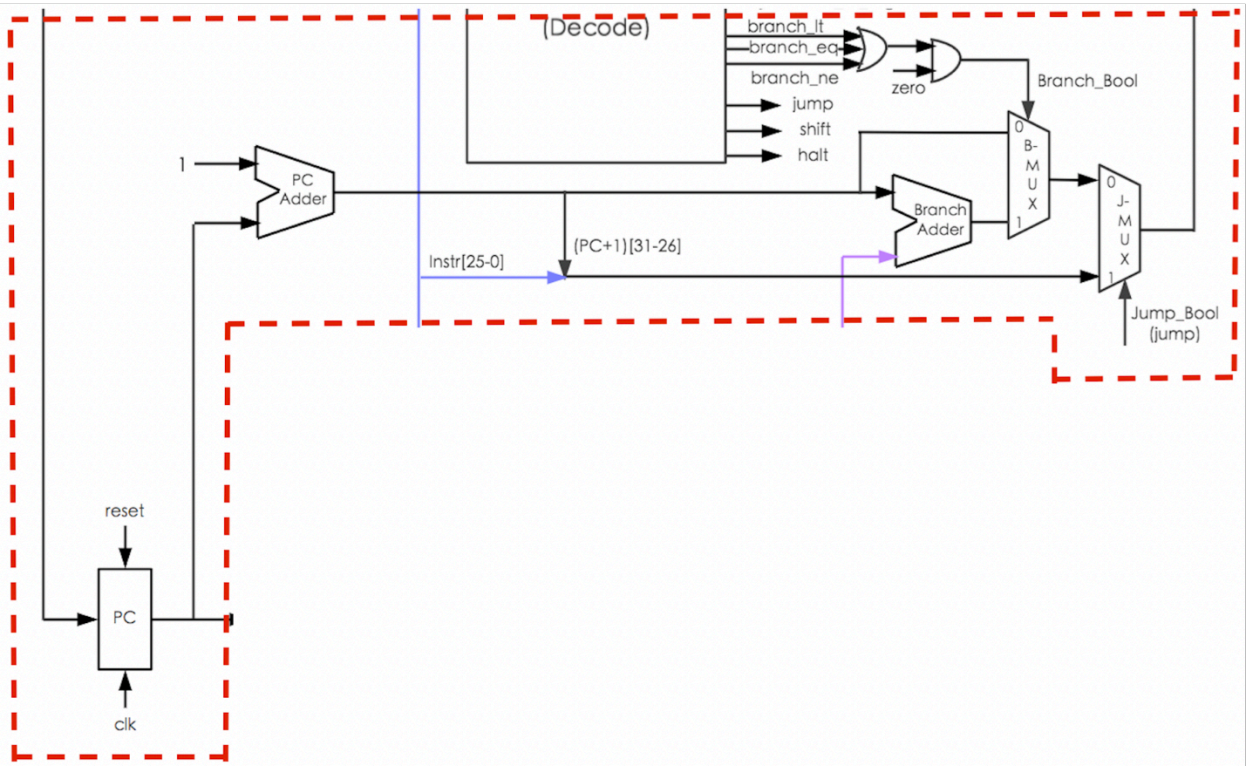
This component was used to store the output, which will be shown or used in further operation. For example, in the key expansion, the 26 skeys will be stored at the address 2 to address 27, also the encryption and decryption output were stored in the data memory. When writing into the data memory, the write_enable signal should be monitored as high level, while loading data from the memory, the read_enable signal should be monitored as high level.

Instruction Memory



This component was used to store all the instruction derived from RC5 assembly code, including key expansion, encryption and decryption. PC will point to them one by one at the beginning of operation state. Also, this component will pass the instruction code as a signal to the main file that will show which instruction is running in each clock.

Instruction Fetch



This component was used to control PC. There are several ways that'll lead to different change of program counter. There are three situations.

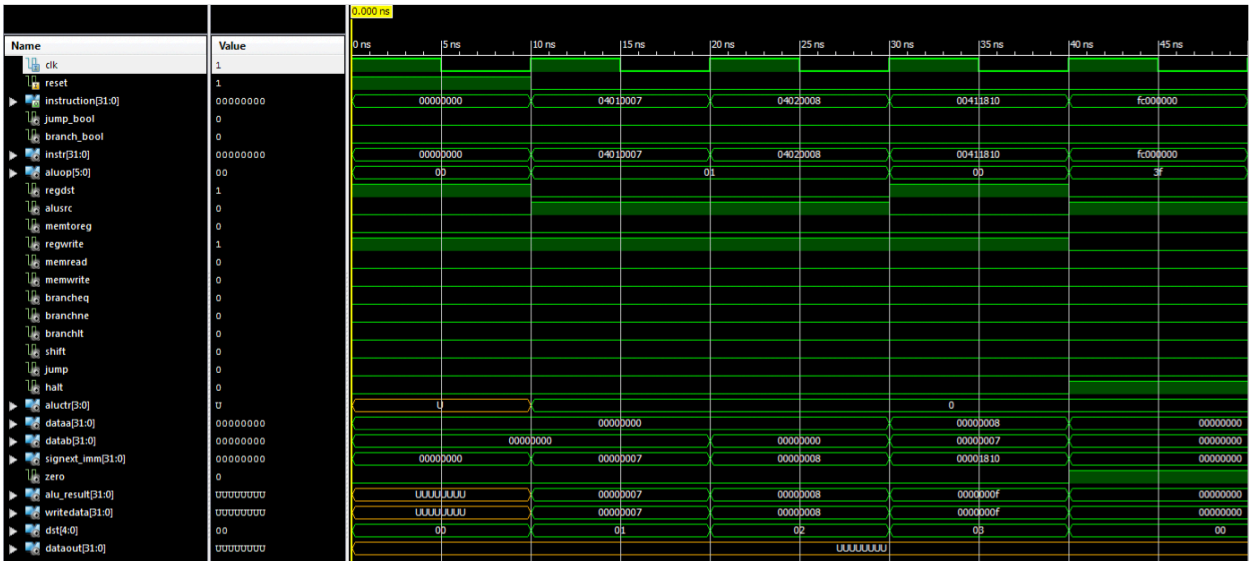
In the 32-bit processor, pc will add 1 automatically after each normal operation. For some special operations, such as jump and branch, it'll change in different way accordingly, which is decided by the control signals via MUXs and adders. For example, after PC added 1, it will be also added with an immediate number, which would be the branch address. With the branch control signal branch_bool,

Branch MUX can select the next PC to be normal PC or a branch PC. The output of Branch MUX will be used by the Jump MUX, which can select the next PC to be Branch MUX output or a jump PC, with a select signal jump_bool, which is the output of control unit, jump.

CPU Verification with Sample Programs

Sample Program 1:

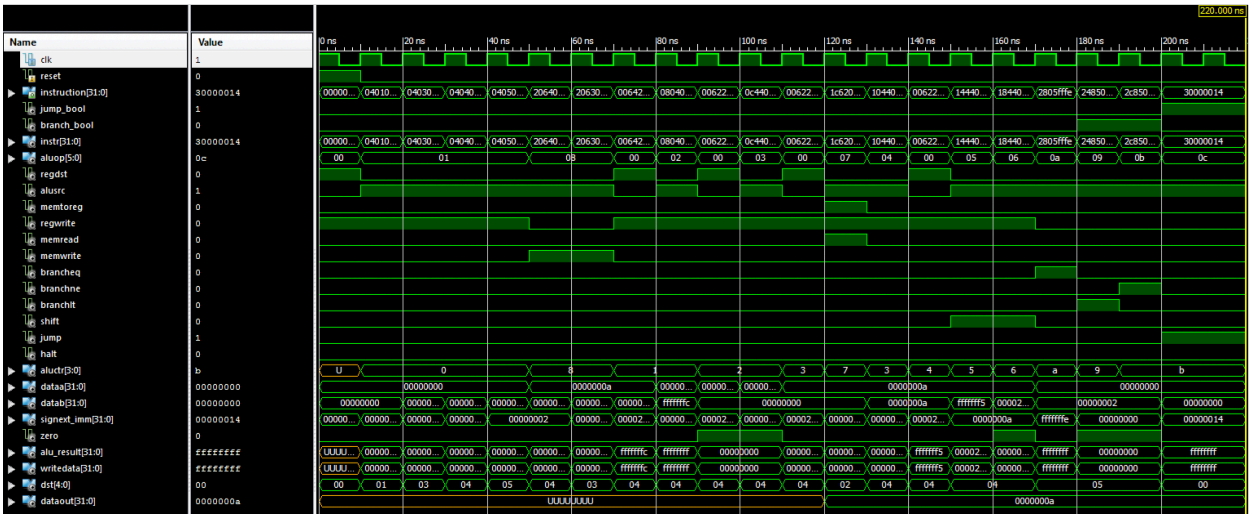
```
ADDI R1, R0, 7          // R1 = 7
ADDI R2, R0, 8          // R2 = 8
ADD R3, R1, R2          // R3 = R1 + R2 =15
HAL                     // HALT
```



Sample Program 2:

This program checks complete instruction set of 18 Instructions.

```
ADDI R1, R0, 2          // R0=0, R1=2
ADDI R3, R0, 10         // R3=10D (D=Decimal)
ADDI R4, R0, 14         // R4=14D
ADDI R5, R0, 2          // R5=2
SW R4, 2(R3)            // 14D is stored in Data memory location 12D
SW R3, 1(R3)            // 10D is stored in Data memory location 11D
SUB R4, R4, R3          // R4=2
SUBI R4, R0, 1          // R4=-1D
AND R4, R2, R3          // R4=0
ANDI R4, R2, 10         // R4=0
OR R4, R2, R3           // R4=10D
LW R2, 1(R3)            // R2=10D (Loaded back from memory)
ORI R4, R2, 10          // R4=10D
NOR R4, R2, R3          // R4=X“fffffff5”
SHL R4, R2, 10          // R4=X“00002800”
SHR R4, R2, 10          // R4=X“01400000”
BEQ R5, R0, -2          // No Branch
BLT R5, R4, 0           // Branch to the next instruction
BNE R5, R4, 0           // Branch to the next instruction
JMP 20                  // Jump
HAL                     // Halt
```



RC5 Assembly code for Key Expansion, Encryption and Decryption

All our assembly codes are hand written accords to the given functions. Any input, output, and rotation key is stored in data memory, and any intermediate variable is temporarily stored in general register. To use inputs, we load data from memory to register via “LW” instruction. Similarly, to store outputs, we store results from register to memory via “SW” instruction. The operation parts are accord with the RC5 functions, using “ADDI”, “SUBI”, “ADD”, “OR”, “SHL”, “SHR”, etc. They all have loops, we use “BLT”, “BEQ”, “BNE” to take the place of loop, take “BLT” for example, when we execute “BLT Rs, Rt, -10”, if “Rs < Rt” then PC will back for 10 units, else next command will be excuted, for the rotation part, for example, when we implement “ROTL(A^B,B)” function, we used a register “Rs” to store the result of “A ^ B”, then we used “Rt” to store the result of “0x0000001F & Rb” that is the final shift number of our rotate, then we used “Rk” to store the subtract of “32 and “Rt”, then we copy “Rs” to “R1” and “R2”, then we use “SHL R1, 1” and a loop to shift “R1” left for “Rt” times and then store it in “R1”, and we use “SHR R2, 1” and a loop to shift “Rk” times and then store it in “R2”, and execute “AND Rs, R1, R2” to use “Rs” to store the result of “R1 & R2”. That is the major difficult we faced when we wrote the assembly code.

Assembly Code to Machine Code

We transferred the assembly code to machine code through a java program, the following is a glimpse of our program

```
public String covertToBit(String s){
    StringBuilder sb = new StringBuilder();
    String[] ss;
    if(((s.startsWith("ADD") || s.startsWith("SUB") || s.startsWith("AND") || s.startsWith("OR"))
        && s.charAt(3) != 'I') || s.startsWith("NOR")){
        sb.append("000000");
        ss = s.split(",");
        sb = constructSb(ss, sb);
        sb.append("00000");
        if(s.startsWith("ADD")) sb.append("010000");
        else if(s.startsWith("SUB")) sb.append("010001");
        else if(s.startsWith("AND")) sb.append("010010");
        else if(s.startsWith("OR")) sb.append("010011");
        else if(s.startsWith("NOR")) sb.append("010100");
    }else if(s.startsWith("JMP")) sb.append("001100");
    else if(s.startsWith("HAL"))  sb.append("11111100000000000000000000000000");
    else {
        ss = s.split(",");
        if(s.startsWith("ADDI")) sb.append("000001");
        else if(s.startsWith("SUBI")) sb.append("000010");
        else if(s.startsWith("ANDI")) sb.append("000011");
        else if(s.startsWith("ORI"))  sb.append("000100");
        else if(s.startsWith("SHL"))  sb.append("000101");
        else if(s.startsWith("SHR"))  sb.append("000110");
        else if(s.startsWith("LW"))   sb.append("000111");
        else if(s.startsWith("SW"))   sb.append("001000");
        else if(s.startsWith("BLT"))  sb.append("001001");
        else if(s.startsWith("BEQ"))  sb.append("001010");
        else if(s.startsWith("BNE"))  sb.append("001011");
        sb = constructSb1(ss, sb);
    }
    return sb.toString();
}

public StringBuilder constructSb(String[] ss, StringBuilder sb){
    int index, a;
```

Here is a glimpse of the execution process of our program, read the assembly code line by line, and translate them line by line, then wrote the machine code to another file.

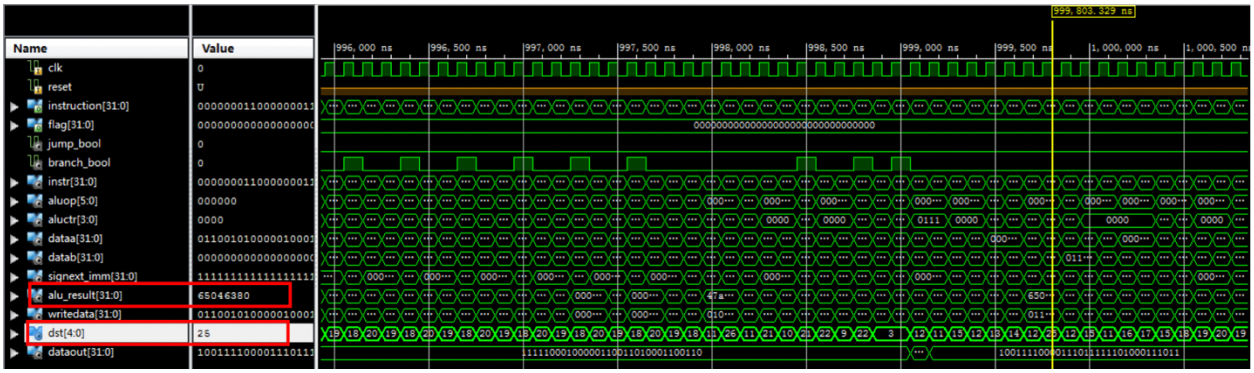
Here is a glimpse of our machine code file

```
AppledeMacBook-Pro-4:Desktop david$ java TransferBit
read by line
line1: ADDI R2,R0,2      //POINTS TO SKEY[0]
line2: ADDI R3,R0,1      //COUNTER
line3: ADDI R4,R0,26
line4: OR R7,R0,R30      //R7 = S[i-1]
line5: SW R7, (R2)
line6: ADDI R2,R2,1
line7: ADDI R8,R0,0      //R8 = S[i]
line8: ADD R8,R7,R29     //R8 = S[i-1] + Q
line9: SW        R8,(R2)          //STORE S[i] TO MEM
line10: ADDI R2,R2,1     //MOVE EMPTY MEM POINTER 1 DOWN
line11: ADDI R7,R8,0     //RENEW R7(S[i-1])
line12: ADDI R3,R3,1
line13: BLT R4,R3,-6
line14: ADDI R21,R0,0    //COUNTER FOR i
line15: ADDI R22,R0,0    //COUNTER FOR j
line16: ADDI R3,R0,0     //COUNTER FOR k
line17: ADDI R24,R0,26
line18: ADDI R23,R0,4    //c=4
line19: ADDI R4,R0,78
line20: ADDI R9,R0,53    //POINTS TO L[0]
```

```
"00000011110000000011100000010011",
"00100000010001110000000000000000",
"00000100010000100000000000000001",
"00000100000010000000000000000000",
"00000011101001110100000000010000",
"00100000010010000000000000000000",
"00000100010000100000000000000001",
"00000101000001110000000000000000",
"00000100011000110000000000000001",
```

RC5 Simulation

Key expansion:

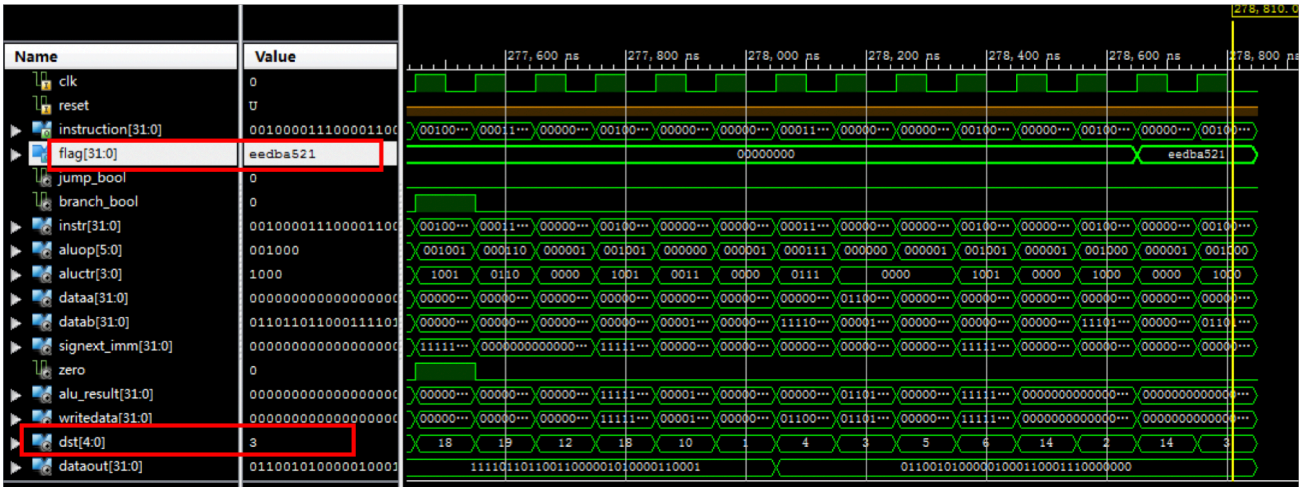


Above is the function simulation result of key expansion. We put key expansion codes into instruction memory file, and hard-coded Ukey(0x00000000000000000000000000000000) in the data memory file, to see if a correct Skey could be generated.

On the screenshot, dst[25] means R25 which stores Skey[25], and the value inside the alu_result is the actual result, 0x65046380. This is corresponded with the reference result from previous labs.

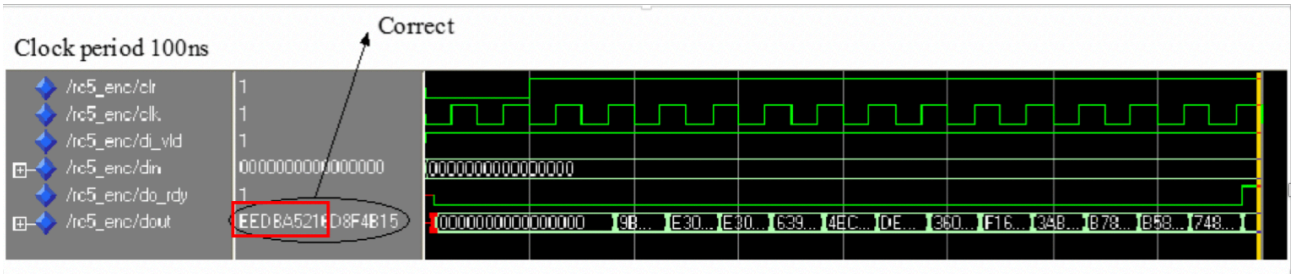
Skey = [X"9BBBD8C8", X"1A37F7FB", X"46F8E8C5", X“460C6085”, X“70F83B8A”, X“284B8303”, X“513E1454”, X“F621ED22”, X"3125065D", X"11A83A5D", X"D427686B", X"713AD82D", X"4B792F99", X"2799A4DD", X"A7901C49", X"DEDE871A", X"36C03196", X"A7EFC249", X"61A78BB8", X"3B0A1D2B", X"4DBFCA76", X"AE162167", X"30D76B0A", X"43192304", X"F6CC1431", X"65046380“]

RC5 encryption:

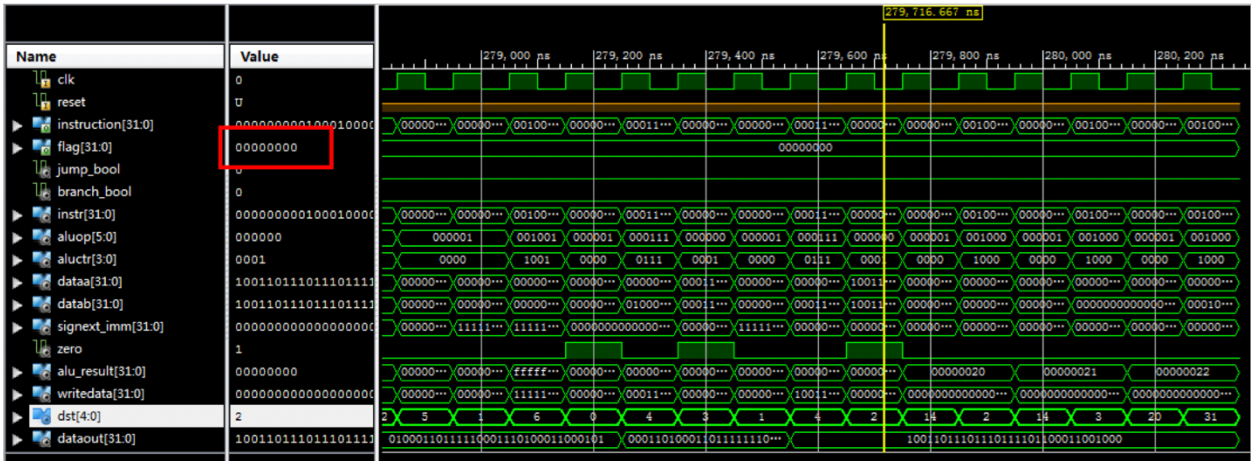


This screenshot shows the result of RC5 encryption with Din(0x0000000000000000). We put codes into instruction memory file and hard-coded the generated Skeys into data memory file. From the screenshot, “flag”

points to the memory block that stores the higher 32 bits of the result. It is corresponded with the reference result in the lecture provided by professor.



RC5 Decryption:



This screenshot shows the result of RC5 Decryption using the output from Encryption as input. So the result should be the same as the input of RC5 Encryption, which is all 0s'. We put codes into instruction memory file and hard-coded the generated Skeys into data memory file. From the screenshot, "flag" points to the memory block that stores the higher 32 bits of the result. It is exactly the input we used for Encryption.

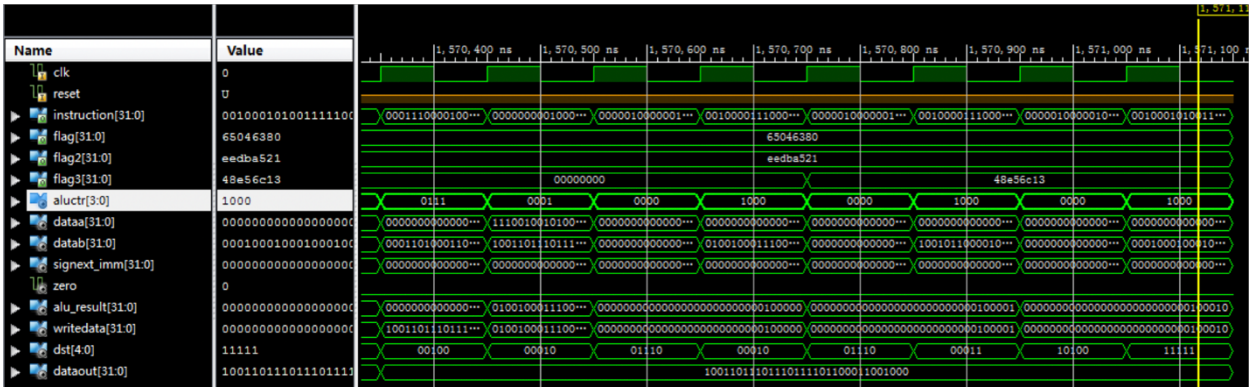
RC5 Key expansion, encryption, and decryption:

After separate module tests, we put all codes together to see if it works. "Flag", "Flag2", "Flag3" on the above screenshots stand for the results of key expansion, encryption, and decryption respectively. Clearly, "flag" and "flag2" show the

identical results as separate ones. And since we used 0x0000000000000000 for both encryption and decryption, “flag2” are different from the previous one, but we proved that it is also the correct output.

Processor Interfaces

To implement our processor and RC5 design on the FPGA board, we need design our processor interfaces. We use Main file to implement the interfaces. In this file, we use our FPGA switches along with five buttons as the input control, and use the seven-segment leds to display the dout signal, which can be the dout of key expansion, encryption and decryption.



Implementation Description

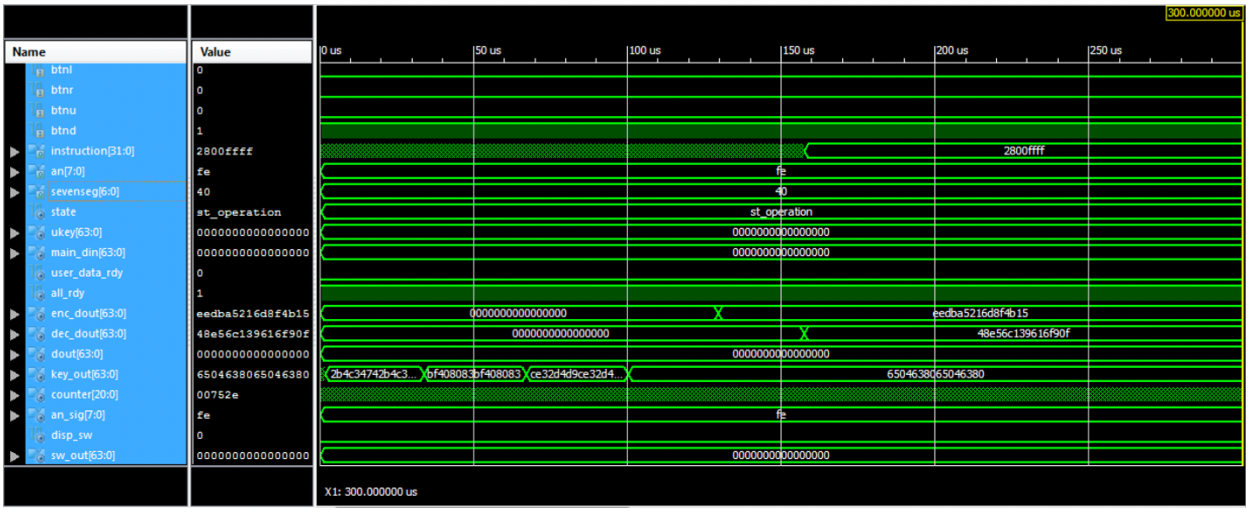
In this Main.vhdl file, we also defined our state machine, which includes four states: At the beginning or after reset, it will be the ST_IDLE state. In this state, 4 buttons control switch inputs to give the value to the signal ukey, it is used for the input of key_expansion. When the btnl button is enabled, it will step into ST_UKEY_RDY state. In this state, input ukey will display on seven segnment, at the same time, value of switches will give to signal main_din, which is the input of encryption and decryption. When button btnr is enabled, it steps into the next state, ST_DATA_SET. In this state, main_din will display on seven segment, at the same time, those inputs will be written into data memory. When btnd is enabled,

the whole process starts running, it will start running the machine code in instruction memory. After this operation, we use switch to select memory address, and use seven segment to see the data stored in that memory location.

Performance and area analysis

Function Simulation Case 1: ukey = X“00000000000000000000000000000000”

main_din = X“0000000000000000”

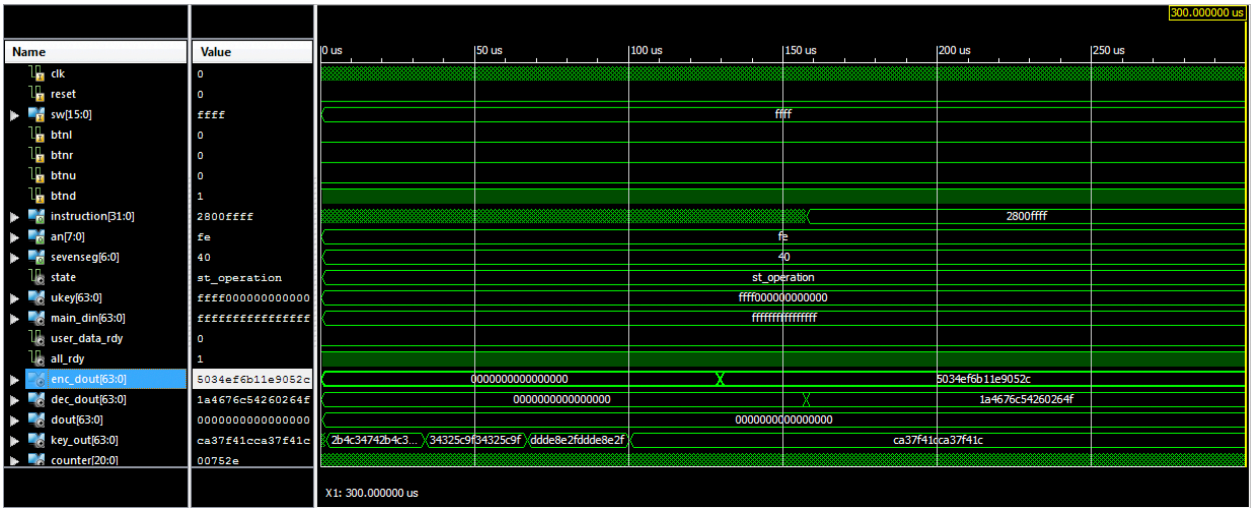


Case 2: ukey = X“0000000000000000ffff000000000000” main_din = X“ffffffffffffffff”

Timing Analysis
The latency is about **40000** clock periods. According to the synthesis report, the minimum period is 3.939ns (maximum frequency is 253.87MHZ). **Propagation delay (total delay) = latency * critical path delay**
= 40000 (clock periods) * 3.939ns
= **157.56us** According to the PAR report, the minimum period is 5.437ns(maximum
frequency is 183.92MHZ).
Propagation delay (total delay) = latency * critical path delay
= 40000 (clock periods) * 5.437ns = **217.48us**

Resources Utilization

For **synthesis**, 3371 out of 126800 slice registers are used. 4740 out of 63400 Slice LUTs are used as Logic in Slice Logic Utilization. 5738 LUT Flip Flop pairs are used in Slice Logic Distribution. 69/210 boned IOBs are used in IO Utilization. Also 3 BUFGs are used.



For **post-route phase**, 3371 out of 126800 slice registers are used. 4582 out of 63400 Slice LUTs are used as Logic in Slice Logic Utilization. 2075/15850 slices are occupied. 5890 LUT Flip Flop pairs are used in Slice Logic Distribution. 69/210 boned IOBs are used in IO Utilization. Also 3 BUFGs are used.

The main differences are the utilization of slices. The IO utilizations are the same. The cause for these differences is that Synthesis estimates how the design will be packed and placed into target architecture, in other words, the resource utilization is an estimation, however the resource utilization after post-route phase is the actual utilization in the simulation.

Overall Design Verification Details

When an instruction was extracted from the instruction memory, firstly, the instruction will be parsed into several parts, in which the operation code will be further parsed via control unit, it will generate several control signals. Two destinations (rs, rt) and immediate number will also be parsed, the former two will be passed to register file and read matched data, together with the immediate number, these three numbers will enter the ALU component, with the control signal from ALU control, the ALU will execute specified computation, if necessary, the result will be passed into the data memory as address and data need to be stored. PC will automatically change under the control signal like jump and branch.

To verify our design, we have two main parts to do the verification. The first part is the CPU verification. As mentioned in previous report, our CPU can run the given operations and sample programs properly. We tested every component of the CPU with specified inputs, and all of them have right output. When testing the sample programs, at every clock, each instruction was extracted properly, which means PC works perfectly. We checked the Rs, Rt, and immediate signal, and finally, the

ALU result, it's the same as expected. And in the later usage of this CPU, it performs well with right results. The second part is the RC5 assembly code. After transfering the assembly code into machine code, we verified the three functions of RC5 independently first. By hard coding the corresponding machine codes in to the instruction memory, we can get the right skey array as given in the class. And by using this skey array, we verified the encryption and decryption functions, which turned out to be all correct. At last, after finishing the processor interfaces design, we ran the whole RC5 (key expansion, encryption, decryption) on our processer. According to the simulation, all the results are correct.