VE370 Introduction to Computer Organization
Lab 3 - Single Cycle Processor
Report

Yiwen Yang 519370910053

1 Introduction

A simple RISC-V 32-bit single cycle processor is designed utilizing Verilog in this lab. Current supported instruction set includes:

- The arithmetic-logical instructions **add**, **addi**, **sub**, **and**, and **or**
- The memory-reference instructions **lw** and **sw**
- The branch instructions **beq** and **bne**

2 Modules design

2.1 Top module RTL design

The **top** module works as a 32-bit RISC-V single cycle processor with a **clk** input. Following is the general RTL architecture of **top**.

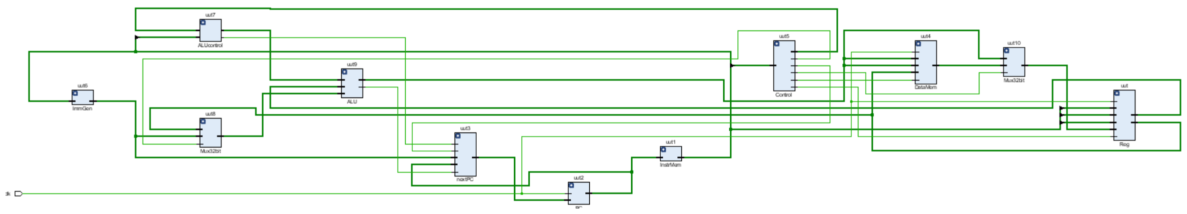


Figure 1: RTL diagram of **top**

2.2 Register File

The **Reg** module is implemented to function as the register file. **Reg** is designed to have 32 32-bit registers, with two read data ports and one write data port. Figure 2 shows the rtl design of the register file.

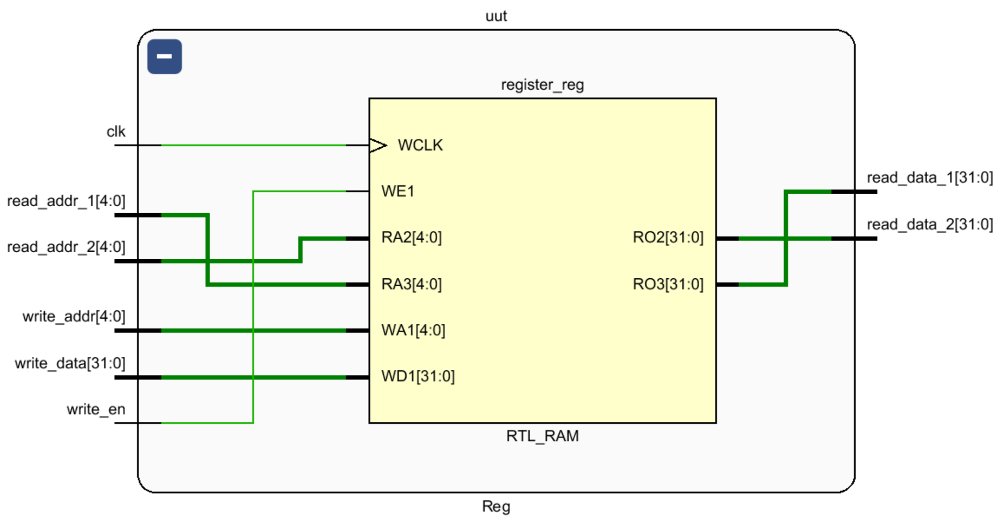


Figure 2: RTL diagram of **Reg**

2.3 Instruction Memory

The **InstrMem** module is implemented to function as the instruction memory. **InstrMem** takes in the index of current instruction from **PC** and outputs the 32-bit instruction from the memory. The size of the memory depends on the count of instructions to be executed. The instructions should be embedded in *InstrMem.v* before running, or stored into */sim/instructions.prog* before the simulation. The index of the instruction is stepped by 1 instead of 4 bytes for the address. Figure 3 shows the rtl design of the instruction memory.

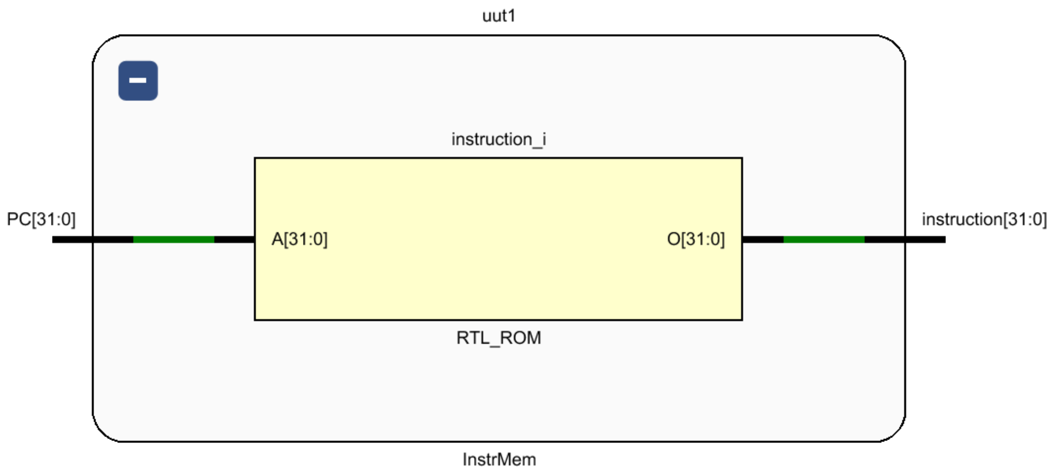


Figure 3: RTL diagram of **InstrMem**

2.4 Program Counter

The **PC** module is implemented to function as the program counter. **PC** initializes the first instruction address to be at 0x0, then takes in next pc address from **nextPC** and outputs it to the instruction memory at next clock cycle to ensure that each instruction is completed in exactly a single clock cycle. Figure 4 shows the rtl design of the program counter.

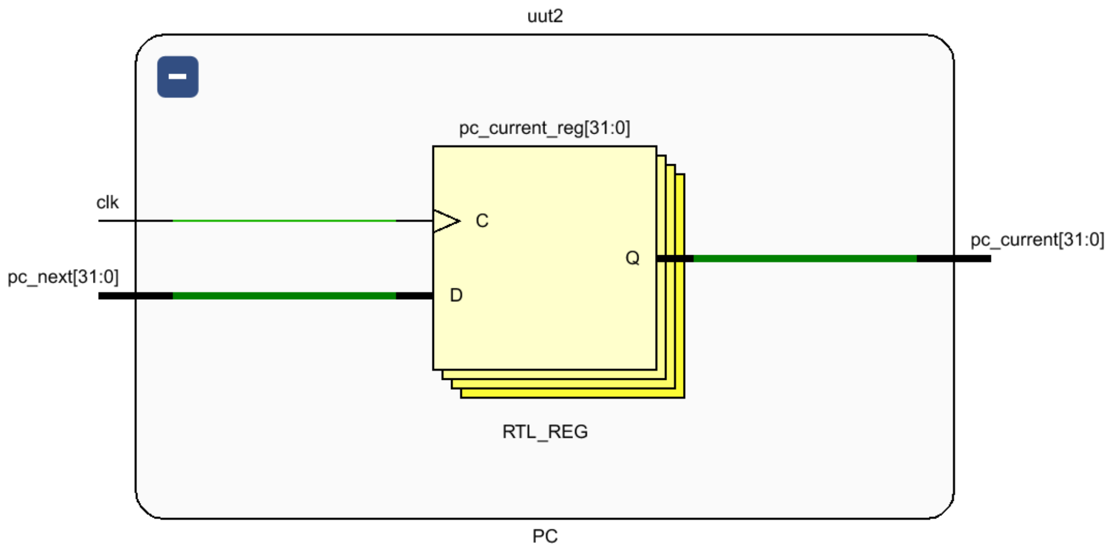


Figure 4: RTL diagram of **PC**

The **nextPC** module is a combinational circuit that controls the next instruction to be executed. When no B-type instruction is executed, or the branch condition is not satisfied in a B-type instruction, the 32-bit Mux will select signal of **pc_current** + 1 as **pc_next**; If the branch condition is satisfied, the 32-bit Mux will select **pc_current** + **immediate** >> **1**. A single bit Mux is added to tell from **beq** and **bne**. Figure 5 shows the rtl design of nextPC.

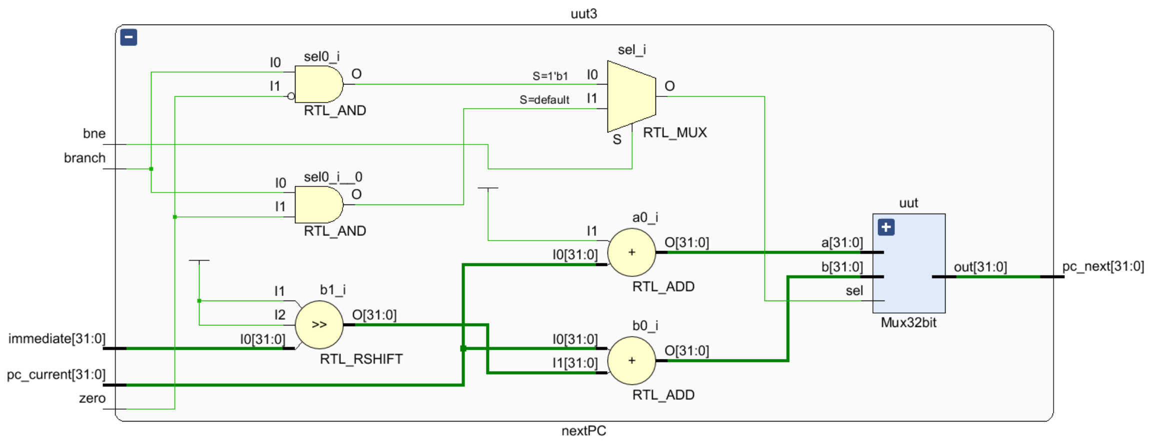


Figure 5: RTL diagram of **nextPC**

2.5 Data Memory

The **DataMem** module is implemented to function as the data memory. **DataMem** provides a 32×32 memory space and one read data port as well as one write data port. The index of the memory is the address divided by four, so a right shifter is utilized. Figure 6 shows the rtl design of the data memory.

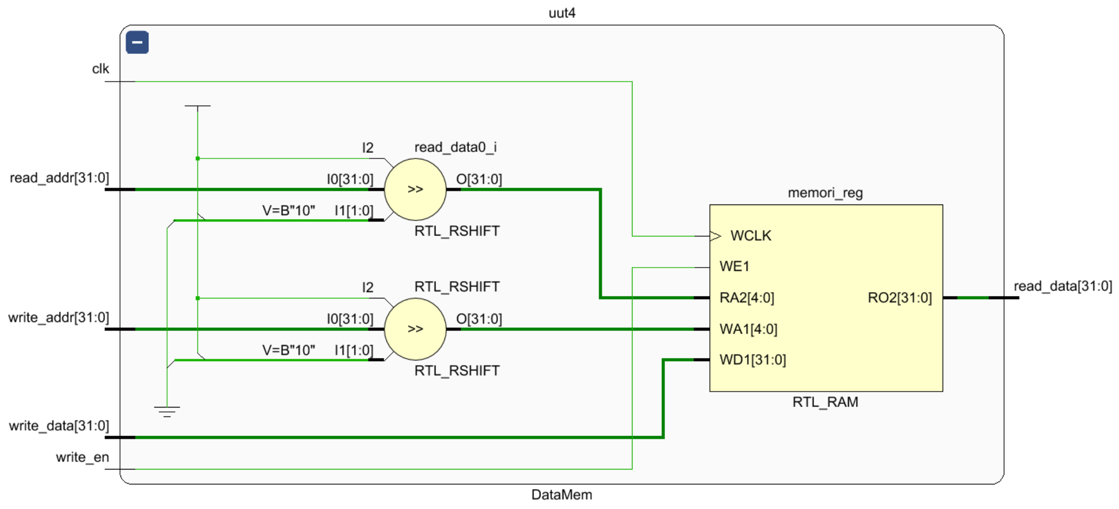


Figure 6: RTL diagram of **DataMem**

2.6 Control Unit

The **Control** module is implemented to function as the control unit. **Control** takes in the opcode of the instruction so as to output various control signals. **ALUop** decides the operation of the ALU unit; **ALUsrc** decides the source of the ALU input; **branch** decides whether PC should branch to other instructions; **memRead** decides read enable of data memory; **memToReg** decides whether to write the memory output to the register input; **memWrite** decides write enable of data memory; **regWrite** decides write enbale of register. Figure 7 shows the rtl design of the control unit.

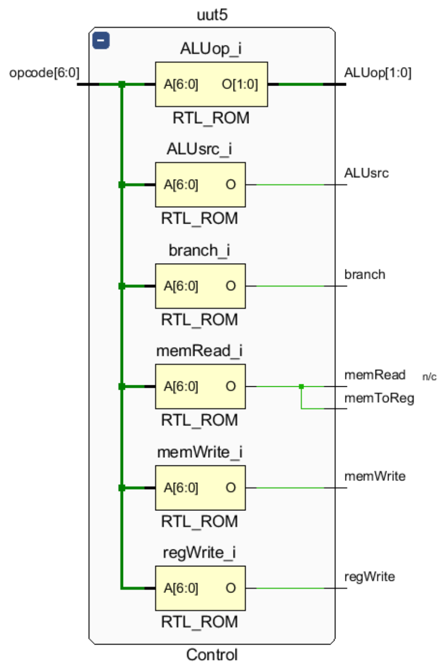


Figure 7: RTL diagram of **Control**

2.7 ALU Control

The **ALUcontrol** module is implemented to function as the ALU control. **ALUcontrol** outputs a 4-bit number to control the working mode of the ALU unit based on the **ALUop** signal and funct3 of the instruction. Figure 8 shows the rtl design of the ALU control.

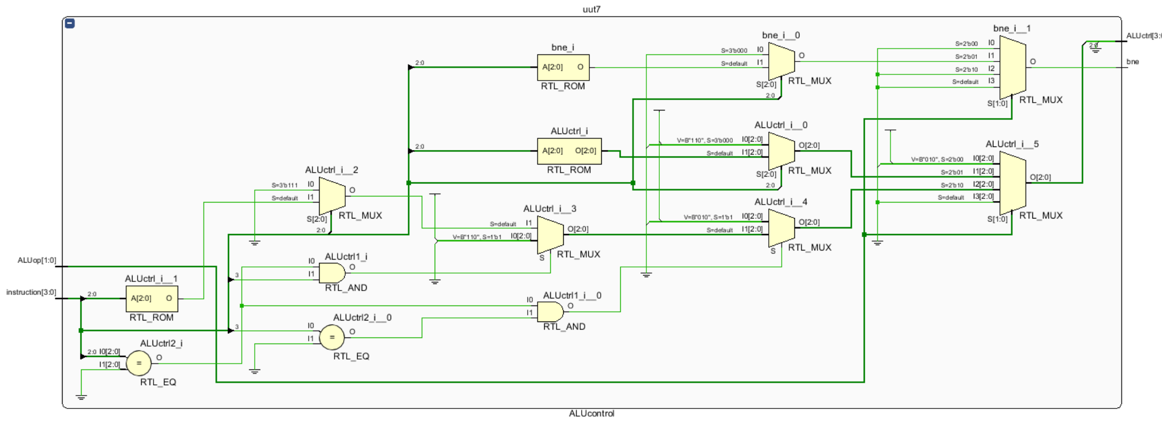


Figure 8: RTL diagram of **ALUcontrol**

2.8 ALU

The **ALU** module is implemented to function as the ALU. **ALU** depends on the ALU control signal to decide the operation type: **0000** for **AND**, **0001** for **OR**, **0010** for + and **0110** for **-**. Besides the result of the operation on **a** and **b**, a **zero** signal is also provided to mark whether the result is zero. Figure 9 shows the rtl design of the ALU.

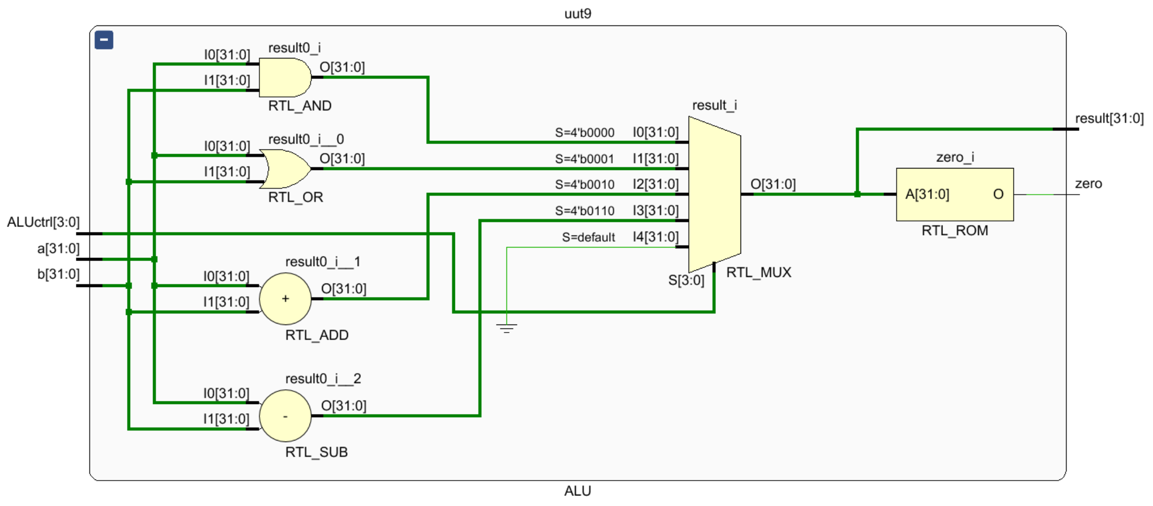


Figure 9: RTL diagram of **alu**

2.9 Immediate Generator

The **ImmGen** module is implemented to function as the immediate generator. **ImmGen** extracts the immediate number from the instruction and extend it to 32 bits. Figure 10 shows the rtl design of the immediate generator.

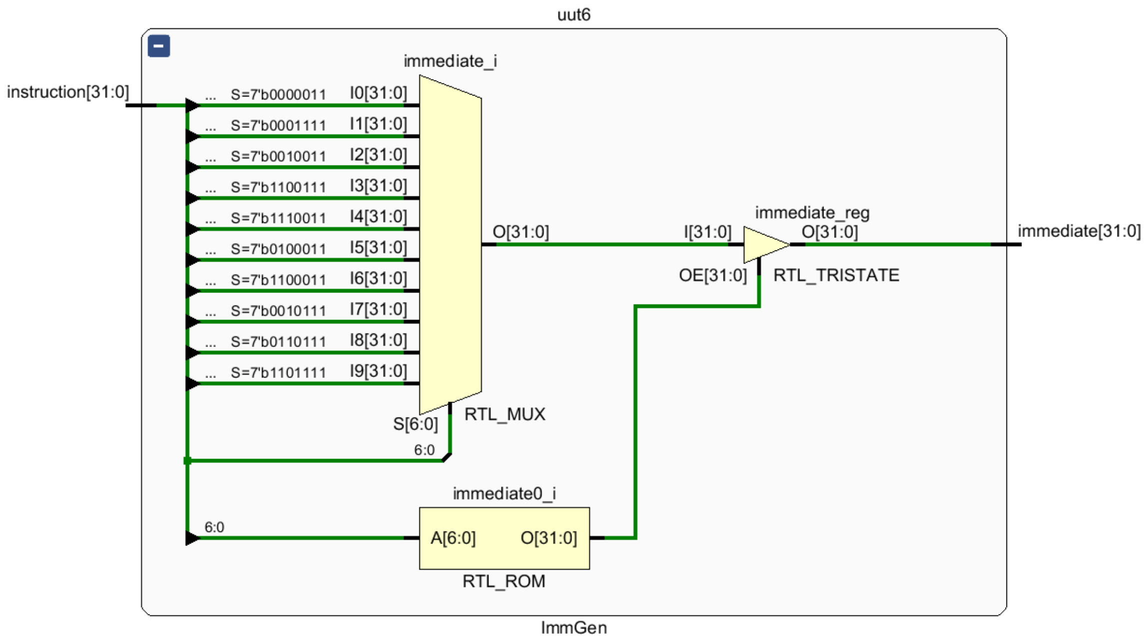


Figure 10: RTL diagram of **ImmGen**

2.10 32-bit Mux

The **Mux32bit** module is implemented for certain mux usage in a 32-bit single cycle processor. Figure 11 shows the rtl design of the 32-bit Mux.

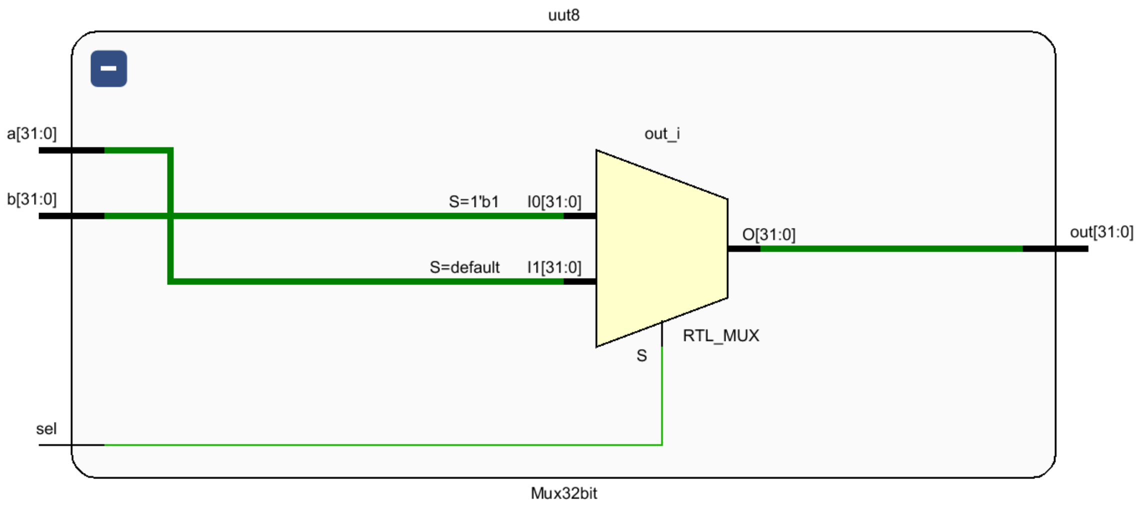


Figure 11: RTL diagram of **Mux32bit**

3 Simulation results

3.1 Modules

All simulation in this part is executed with *instructions.prog*, which includes the binaries disassembled from *lab3_testcase.s*.

3.1.1 Register File

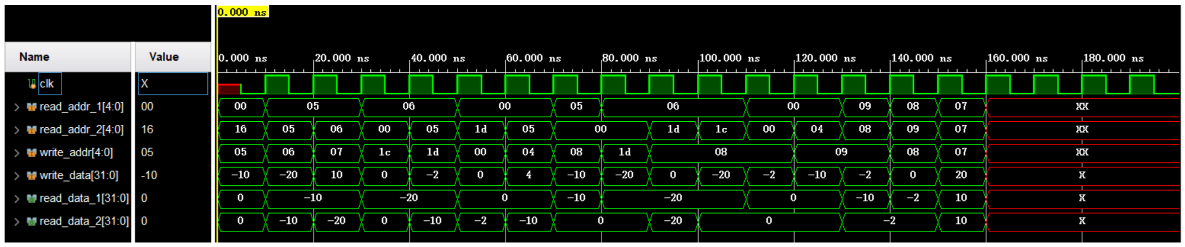


Figure 12: Simulation of **Reg**

3.1.2 Instruction Memory

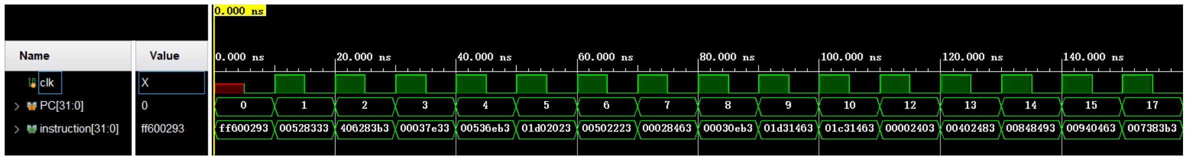


Figure 13: Simulation of **InstrMem**

3.1.3 Program Counter

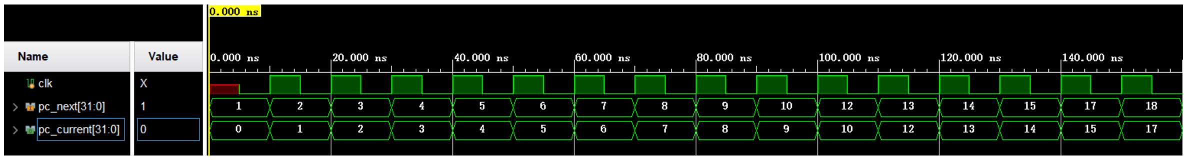


Figure 14: Simulation of **PC**

3.1.4 Next Program Counter

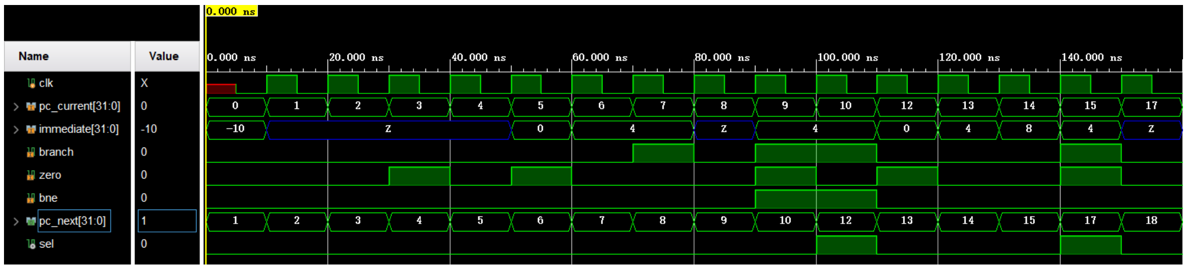


Figure 15: Simulation of **nextPC**

3.1.5 Data Memory

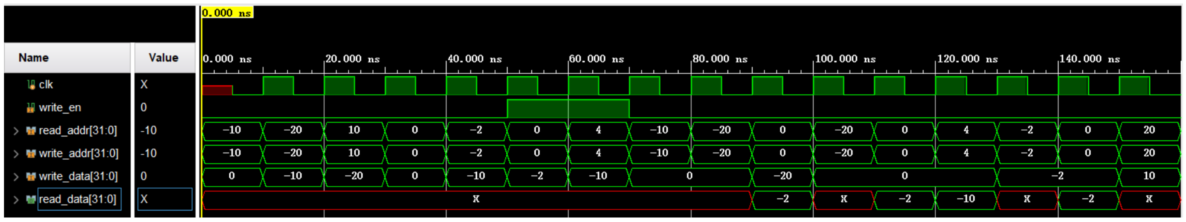


Figure 16: Simulation of **dataMem**

3.1.6 Control Unit

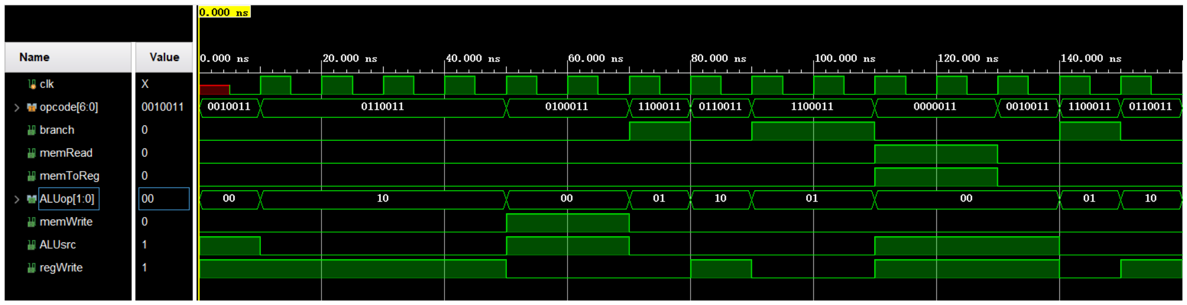


Figure 17: Simulation of **Control**

3.1.7 ALU Control

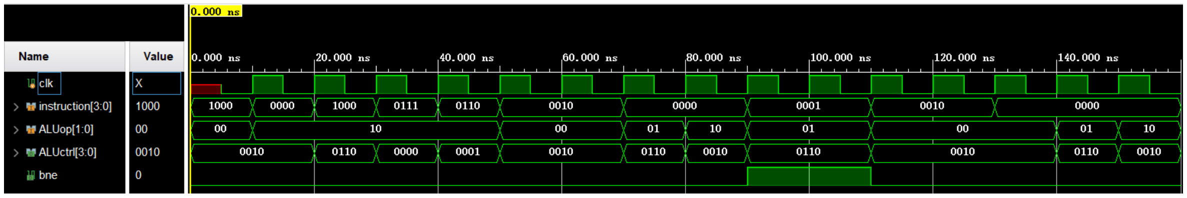


Figure 18: Simulation of **ALUcontrol**

3.1.8 ALU

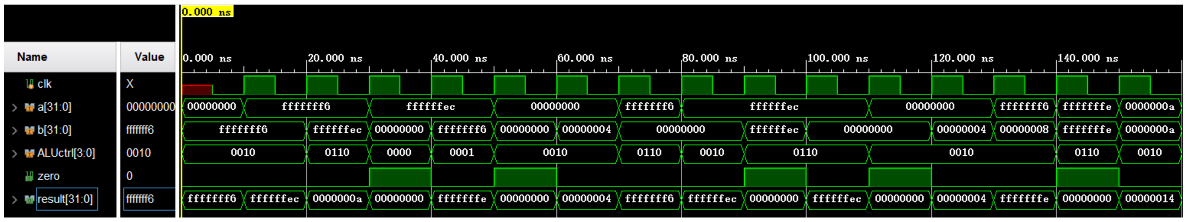


Figure 19: Simulation of **Reg**

3.1.9 Immediate Generator

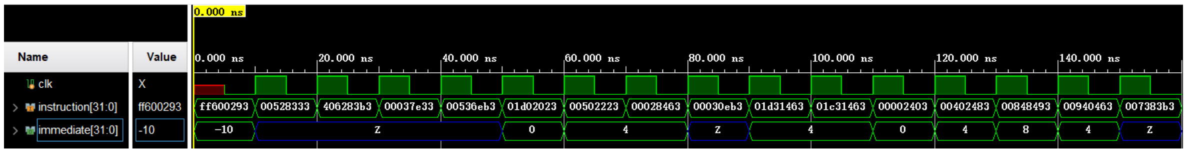


Figure 20: Simulation of **ImmGen**

3.1.10 32-bit Mux

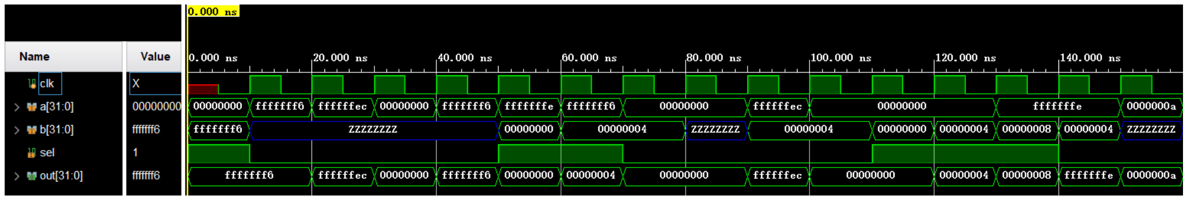


Figure 21: Simulation of **Mux32bit**

3.2 Instructions

3.2.1 addi, add

The simulation to test **add** and **addi** is based on following codes.

```
addi x5, x0, 10
addi x6, x0, 2
add x1, x5, x6
```

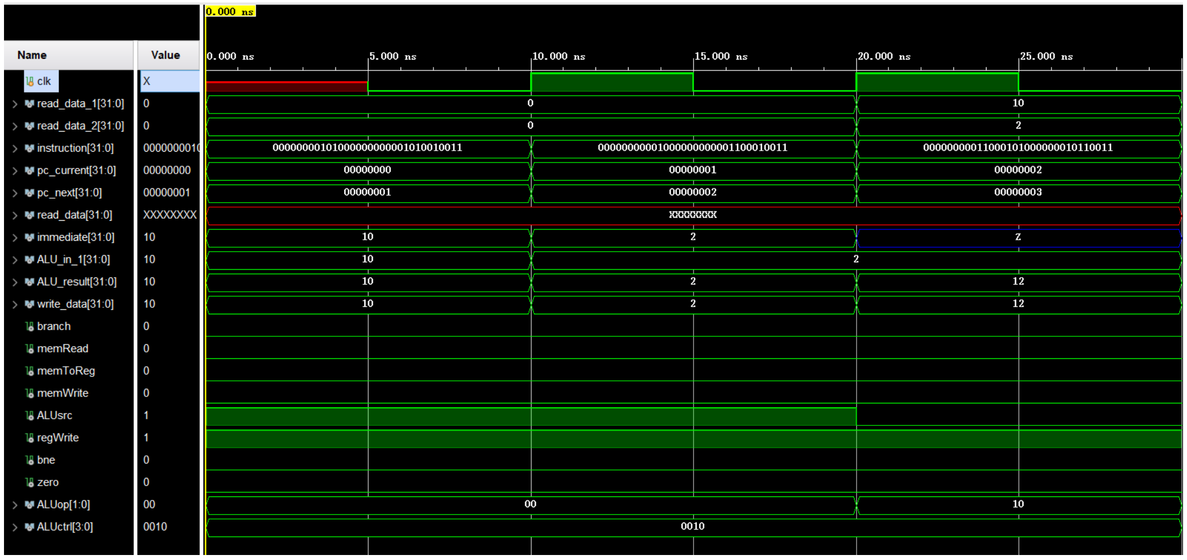


Figure 22: Simulation of **addi** and **add**

Figure 22 shows all ports signals. Take the first instruction as an example for **addi**, one register file read port and the write port are used, so **read_data_1** gives the value of **x0** , while **write_data** receives the result of the calculation as **x0 + 10** and **regWrite** is **1**. The immediate number is extracted from the instruction then extended to a 32-bit number that equals to **10**. The option code for **addi** is designed to be **00**, so that the ALU operation would be add and **ALUctrl** is **0010**. Sources of ALU input are **read_data_1** and immediate, so **ALUsrc** is **1**. No branches or memory access is required, so all other control signals are **0**.

The **add** instruction reads two value from the register file and add them with ALU, writing the result back to the register file, so **read_data_1** gives the value of **x5** and **read_data_2** gives the value of **x6**, **write_data** passes the result to **x1**. Since no immediate number is used, then the control unit sets **ALUsrc** to be **0** so that ALU takes in two inputs both from the register file. The option code for **add** is designed to be **10**; The ALU operation would be add according to **funct3** so that **ALUctrl** is **0010**. No branches or memory access is required, so all other control signals are **0**.

3.2.2 lw, sw

The simulation to test **lw** and **sw** is based on following codes.

```
    addi x5, x0, 7
    sw x5, 0, x0
    lw x6, 0, x0
```

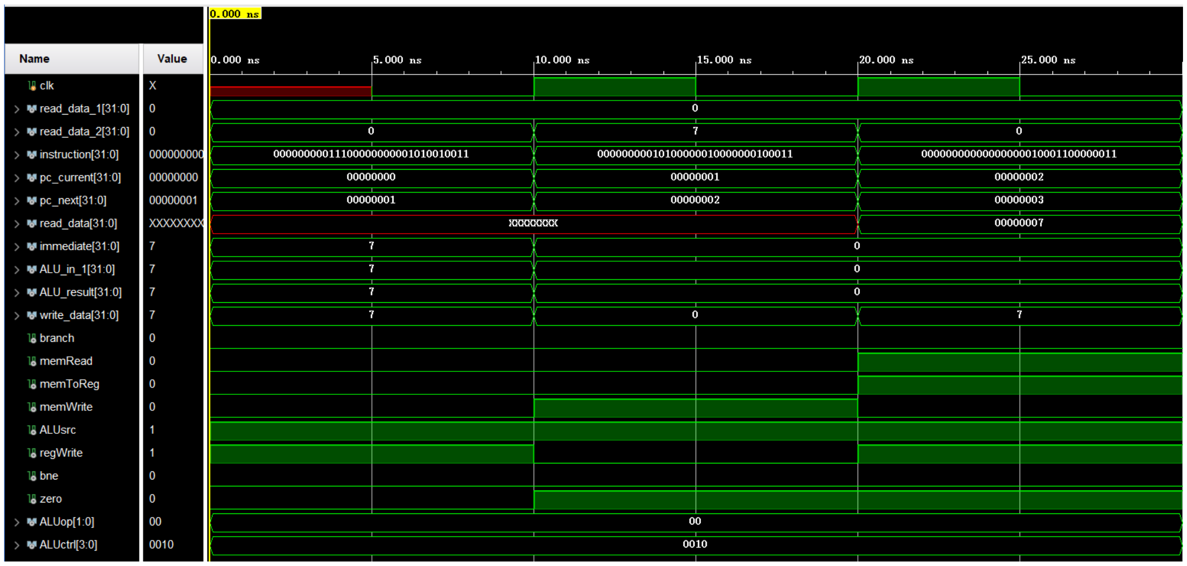
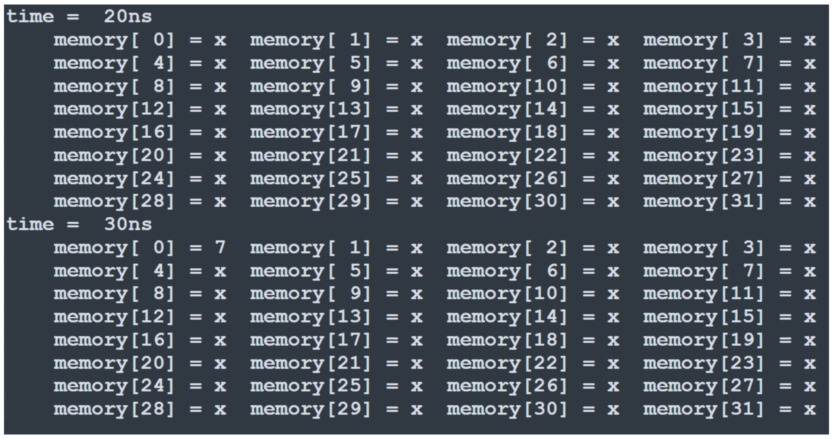


Figure 23: Simulation of **lw** and **sw**



```
time =   20ns
    memory[ 0] = x  memory[ 1] = x  memory[ 2] = x  memory[ 3] = x
    memory[ 4] = x  memory[ 5] = x  memory[ 6] = x  memory[ 7] = x
    memory[ 8] = x  memory[ 9] = x  memory[10] = x  memory[11] = x
    memory[12] = x  memory[13] = x  memory[14] = x  memory[15] = x
    memory[16] = x  memory[17] = x  memory[18] = x  memory[19] = x
    memory[20] = x  memory[21] = x  memory[22] = x  memory[23] = x
    memory[24] = x  memory[25] = x  memory[26] = x  memory[27] = x
    memory[28] = x  memory[29] = x  memory[30] = x  memory[31] = x
time =   30ns
    memory[ 0] = 7  memory[ 1] = x  memory[ 2] = x  memory[ 3] = x
    memory[ 4] = x  memory[ 5] = x  memory[ 6] = x  memory[ 7] = x
    memory[ 8] = x  memory[ 9] = x  memory[10] = x  memory[11] = x
    memory[12] = x  memory[13] = x  memory[14] = x  memory[15] = x
    memory[16] = x  memory[17] = x  memory[18] = x  memory[19] = x
    memory[20] = x  memory[21] = x  memory[22] = x  memory[23] = x
    memory[24] = x  memory[25] = x  memory[26] = x  memory[27] = x
    memory[28] = x  memory[29] = x  memory[30] = x  memory[31] = x
```

Figure 24: Memory records

Figure 23 shows all ports signals. The **sw** instruction reads **x5** into **read_data_2** which represents the value to be saved into the memory and **x0** into **read_data_1** which represents the head address in the memory; There is no register write back required, so **write_data** is not used and **regWrite** is **0**. The immediate number is the offset which is **0** in this example, and the **ALUsrc** is chosen to be **1** to add the offset with the address. The option code of **sw** is designed as **00**, and **ALUctrl** signal is **0010** so that ALU operation is add. Memory writing is required, so **memWrite** is **1**. No branches or memory reading is required, so all other control signals are **0**. Figure 24 shows that **memory**[**0**] changed from **x** to **7**, which proves that **sw** was operated correctly.

The **lw** instruction reads **x0** into **read_data_1** which represents the head address in the memory and writes the loading result back to the register file **x6**. The result **write_data** contains the read value from the memory which equals to **7** and **memToReg** and **regWrite** are **1**. The immediate number is the offset which is **0** in this example, and the **ALUsrc** is chosen to be **1** to add the offset with the address. The option code of **sw** is designed as **00**, and **ALUctrl** signal is **0010** so that ALU operation is add. Memory reading is required, so **memRead** is **1**. No branches or memory writing is required, so all other control signals are **0**.

3.2.3 beq

The simulation to test **beq** is based on following codes.

```
    addi x5, x0, 1
    addi x6, x0, 2
    beq x5, x6, ERR
    addi x5, x0, -1
ERR:
    addi x0, x0, 0
```

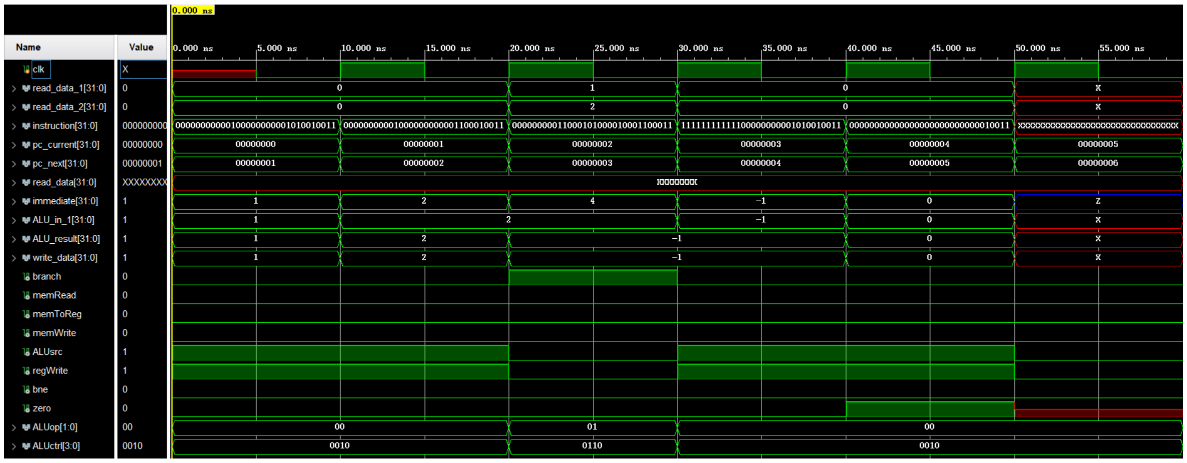


Figure 25: Simulation of **beq**

Figure 25 shows all ports signals. The **beq** instruction reads **x5** into **read_data_1** and **x6** into **read_data_2**; There is no register write back required, so **write_data** is not used and **regWrite** is **0**. The immediate number is the difference between the instruction address to jump to and the current instruction address divided by 2, which is **4** in this example. Then **ALUsrc** is chosen to be **0** and the opcode of **beq** is designed to be **01** so as to subtract the two read values from the register file, i.e., **ALUctrl** is **0110**. Since this is a branch operation, **branch** signal is controlled to be **1**; Since the subtraction result is not zero (**zero** is **0**), which means that the branch condition is not satisfied, the program counter chooses to continue to the next instruction instead of jumping to **ERR**. It can be seen from the wave panel that **x5** was set to be **-1** after 30ns which implies that **beq** worked properly.No memory access is required, so all other control signals are **0**.