Hardware Design and Implementation of LC2K Multiple Cycle Datapath and Controller

Rijun Cai
12348003

December 24, 2013

1 Objective

The objective of this laboratory is to design a LC-2K processor with multiple cycle datapath using VHDL hardware description language and to simulate the execution of the instructions in machine language in ModelSim.

2 Laboratory Tasks

- Design a multiple cycle datapath for LC2K ISA
- Design a FSM controller for this datapath
- Design the components of the datapath and the controller in VHDL language
- Connect the components to form a whole VHDL model of the LC2K processor
- Simulate the execution of each instruction of LC2K ISA in ModelSim

3 Designing Datapath

We use the LC2K multiple cycle datapath in Figure 1 from our lecture. All element parts in this datapath are triggered by high voltage level of control signals except that the ALU result register, the register file and the memory are triggered by rising edge of the clock. At very first, all elements may he initialized and disabled.

4 Designing Controller

In the following, I will discuss the execution of the most important instructions within the datapath in detail. The execution of each instruction within the datapath consists of several cycles. The lines highlighted blue indicates the activated path in that cycle. All LC2K instructions have the format such as follows:

```
opcode RegA RegB desReg
opcode RegA RegB offset
opcode RegA RegB
opcode
```

4.1 FET0

Get instruction from the memory into the IR register. All instruction execution in this cycle is the same. See Figure 2.

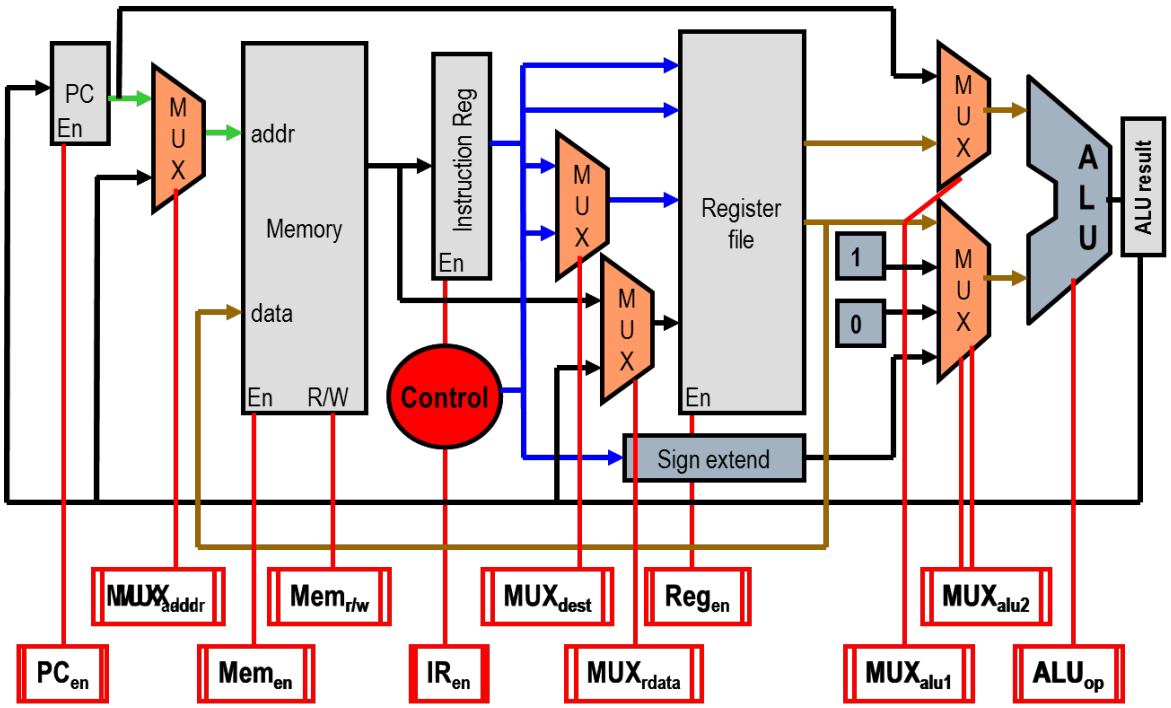

Figure 1: Datapath

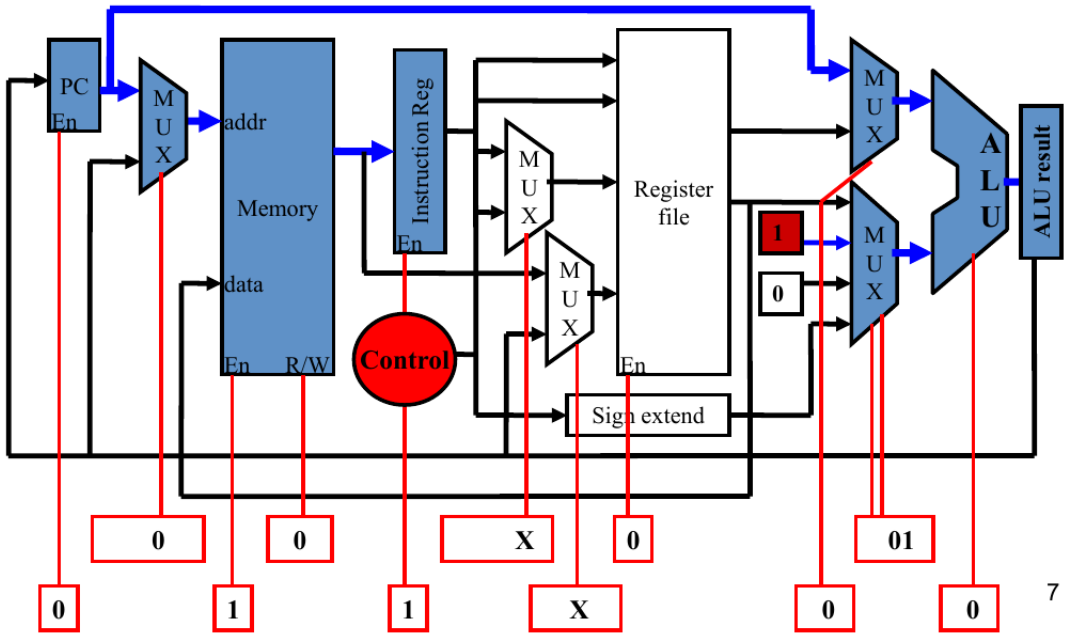

Figure 2: FET0

4.2 DEC1

When the rising edge of the CLK arrives, the instruction is stored in the IR and the PC + 1 is stored in the ALU Result Register. In this state, the "`next state`" output of the FSM is `UNKNOWN`, which indicates that the next state of the FSM is determined by an extra logic circuit, which calculates the next state of the FSM according to the instruction. See Figure 3.

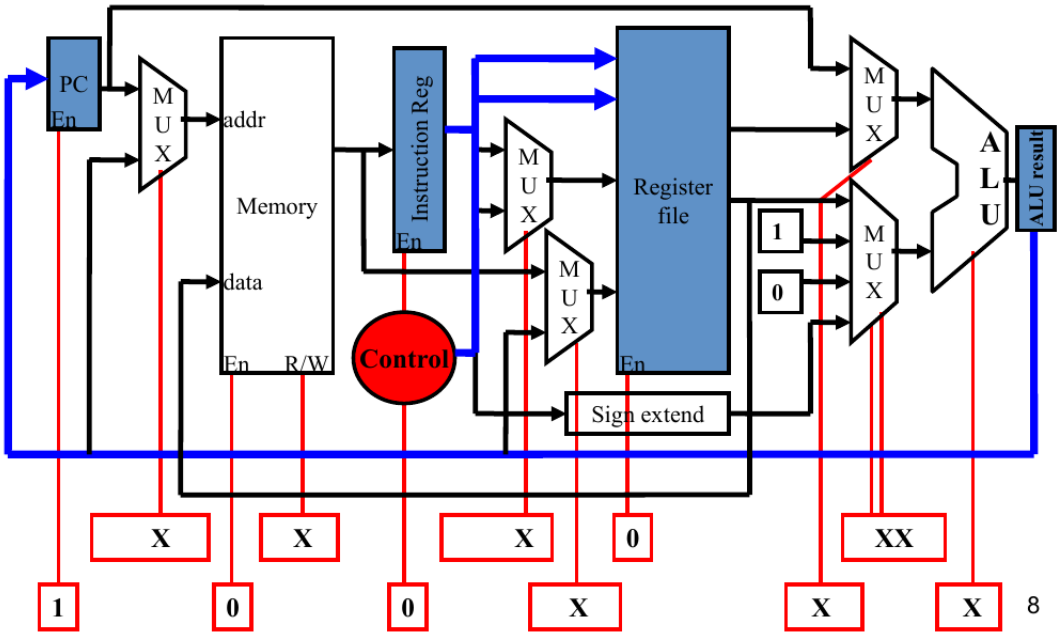

Figure 3: DEC1

4.3 ADD2

In this cycle, `RegA + RegB` is calculated by the ALU and the result is stored in the ALU Result Register at the end of this cycle. See Figure 4.

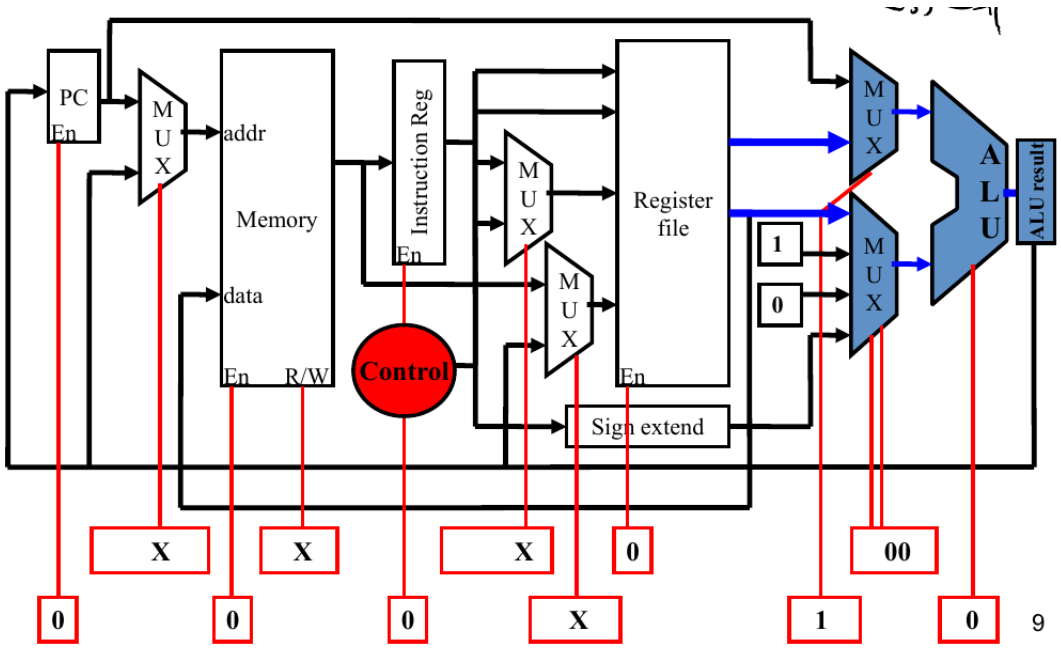

Figure 4: ADD2

4.4 ADD3

In this cycle, the result of the ALU is stored in `desReg`. The FSM returns to FET0 after this cycle. See Figure 5.

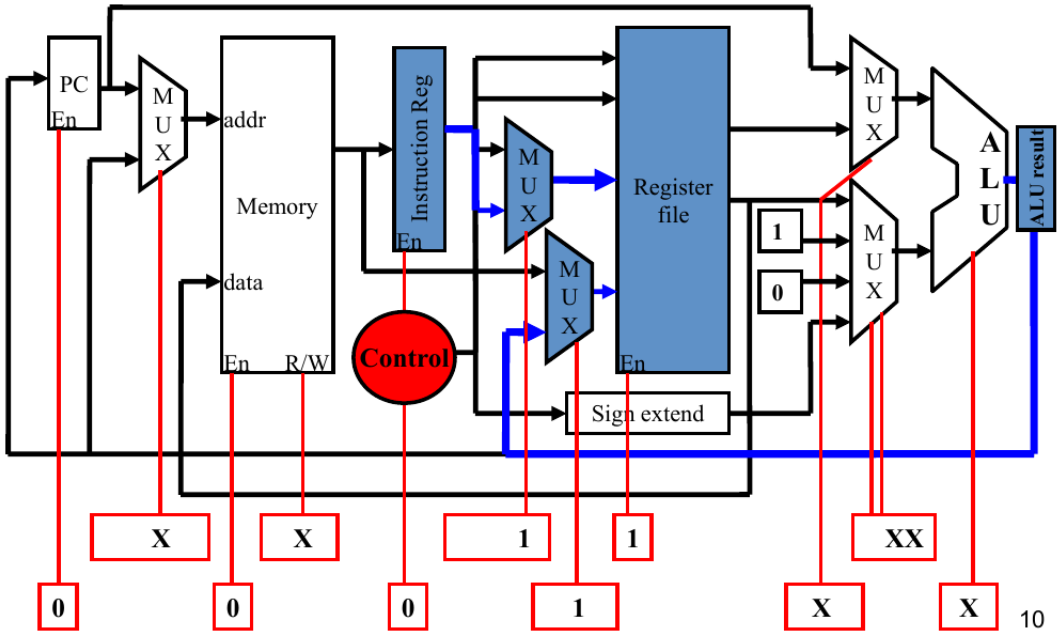

Figure 5: ADD3

4.5 NAND4

This cycle is almost the same as ADD2 except that `ALUop` is `1` so that the ALU will do the NAND calculation instead of the ADD calculation.

4.6 NAND5

This cycle is exactly the same as ADD3.

4.7 LW6

The address of the data to be accessed is calculated in this cycle. The result is stored in the ALU Result Register. See Figure 6.

4.8 LW7

The address calculated in the previous cycle is sent to the memory and then the memory output the desired data. See Figure 7.

4.9 LW8

The data from the memory are eventually stored in `desReg` in this cycle. The FSM returns to FET0 after this cycle. See Figure 8.

4.10 SW9

In this cycle, the destination address is calculated by the ALU.

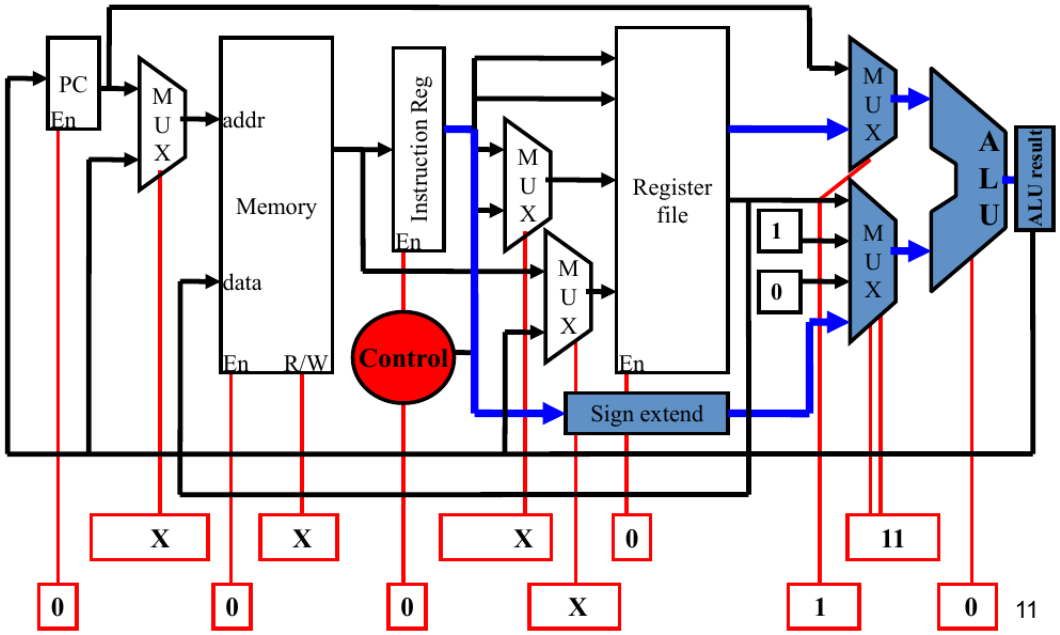

Figure 6: LW6

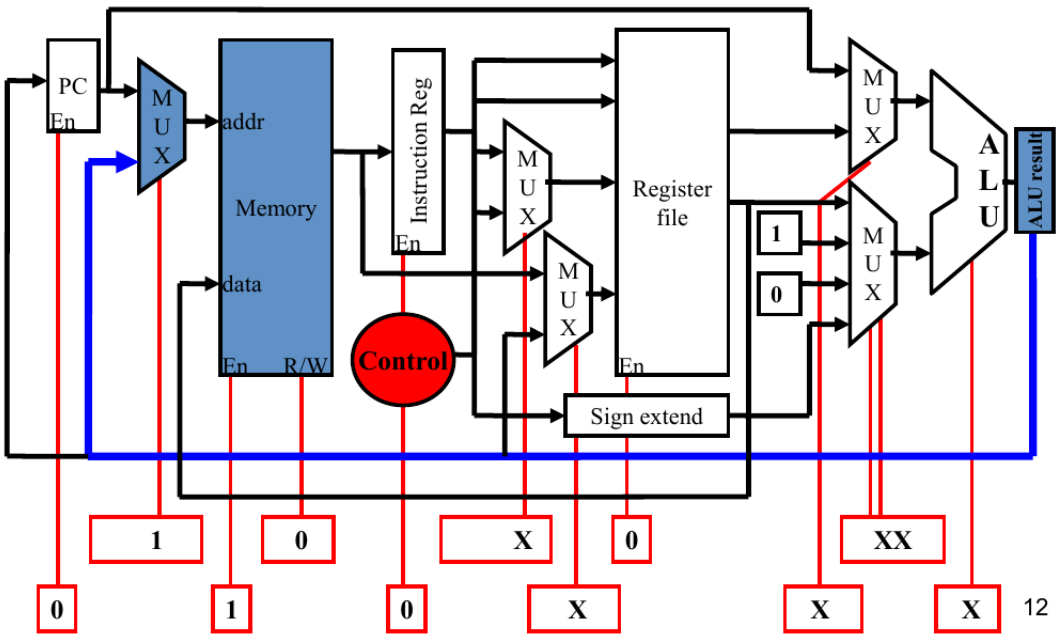

Figure 7: LW7

4.11 SW10

In this cycle, the data of `RegB` are sent to the memory data input port, and are stored in the desired location at the end of this cycle then the FSM returns to FET0.

4.12 BEQ11

The address to which the CPU is going to branch is calculated in this cycle. See Figure 9.

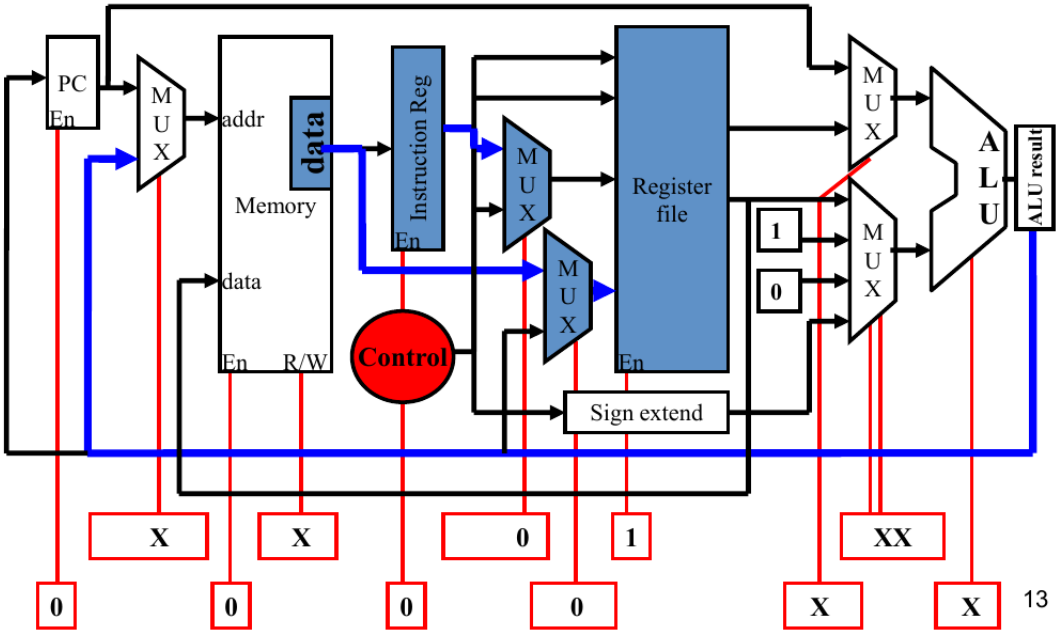

Figure 8: LW8

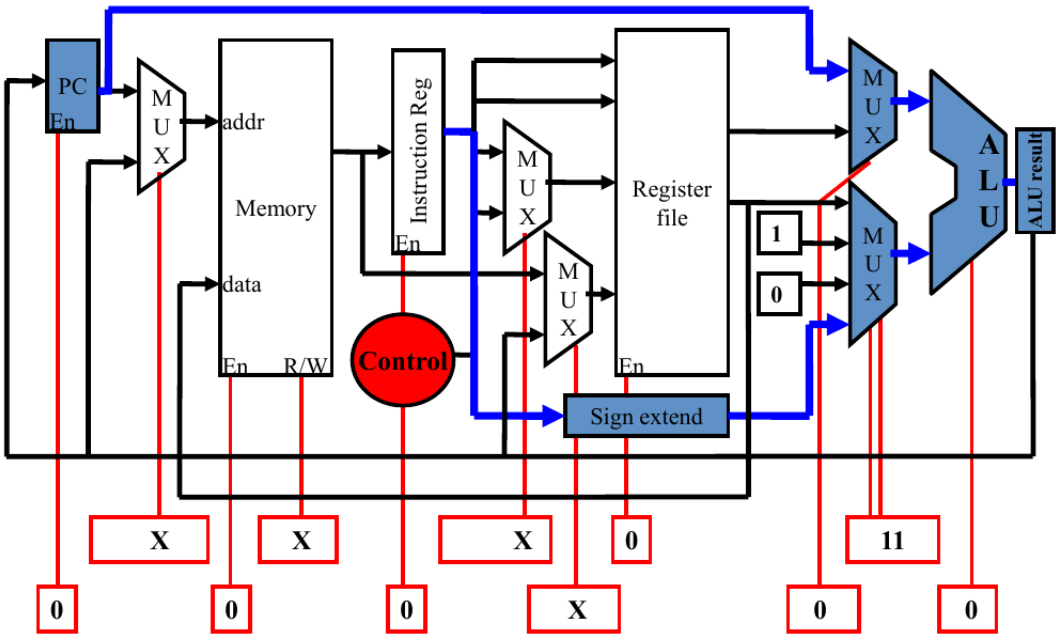

Figure 9: BEQ11

4.13 BEQ12

In this cycle, the data of `RegA` and `RegB` are sent to the ALU and the equality of them are tested. `PCen` is controlled by an extra logic circuit that takes `eq?` as an input. So when `eq?` is `1`, the PC Register is enable then the CPU will branch to the desired address. See Figure 10.

4.14 JALR13

In this cycle, `PC + 1`, which is calculated in DEC1, is stored in the ALU Result Register.

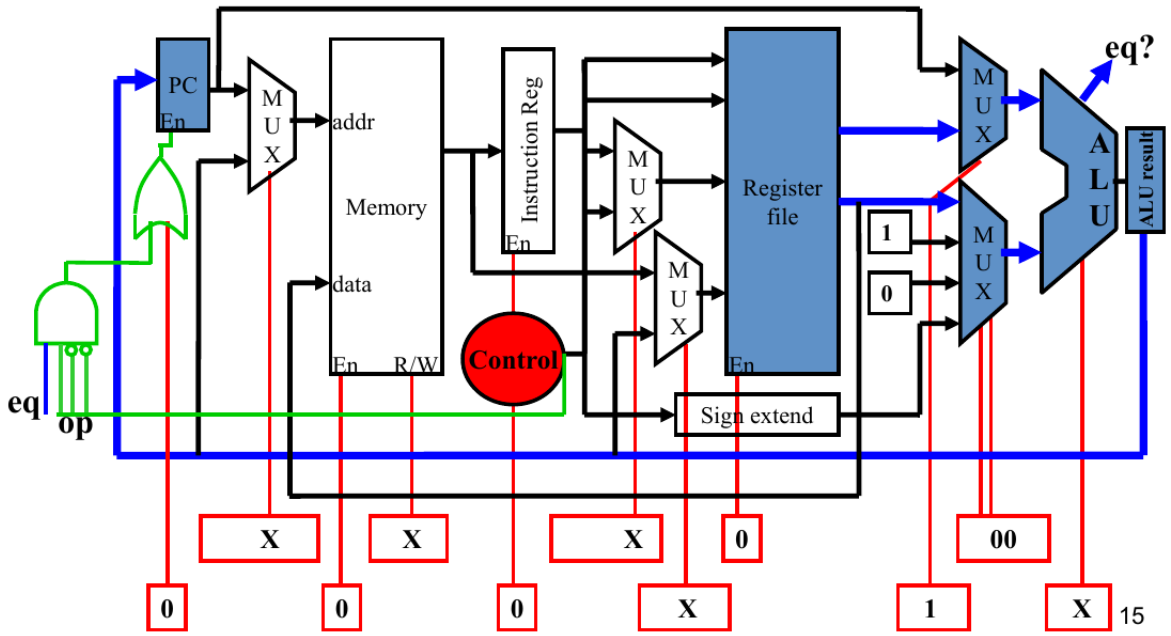

Figure 10: BEQ12

4.15 JALR14

In this cycle, `PC + 1` is stored in `RegB`, then the data of `RegA` is sent to the ALU and added with `0`.

4.16 JALR15

In this cycle, the destination address, which is stored in the ALU Result Register in the previous cycle, is sent to the PC register.

5 Component Implementation in VHDL

Please refer to the source code.

6 Verifying Component in ModelSim

6.1 ALU

Add See Figure 11.

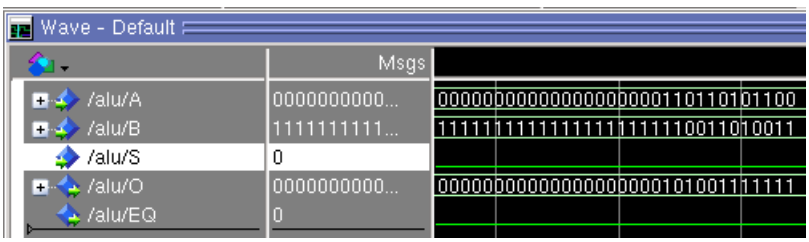

Figure 11: ALU Add

Nand See Figure 12.

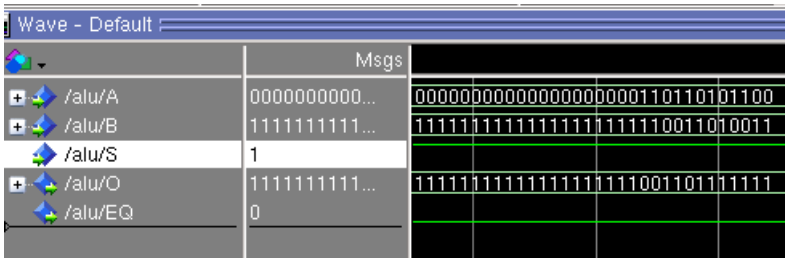

Figure 12: ALU Nand

6.2 Multiplexer

2-to-1 Multiplexer See Figure 13.

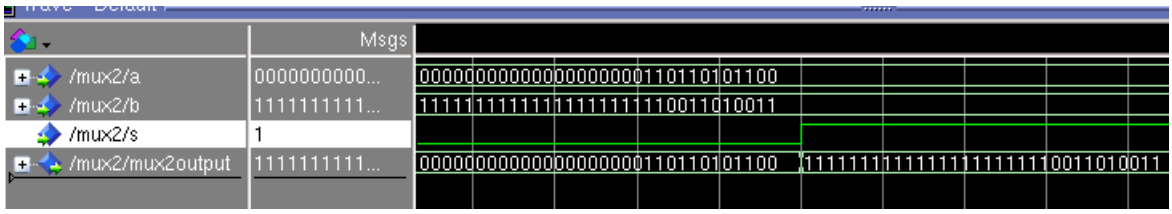

Figure 13: 2-to-1 Multiplexer

4-to-1 Multiplexer See Figure 14.

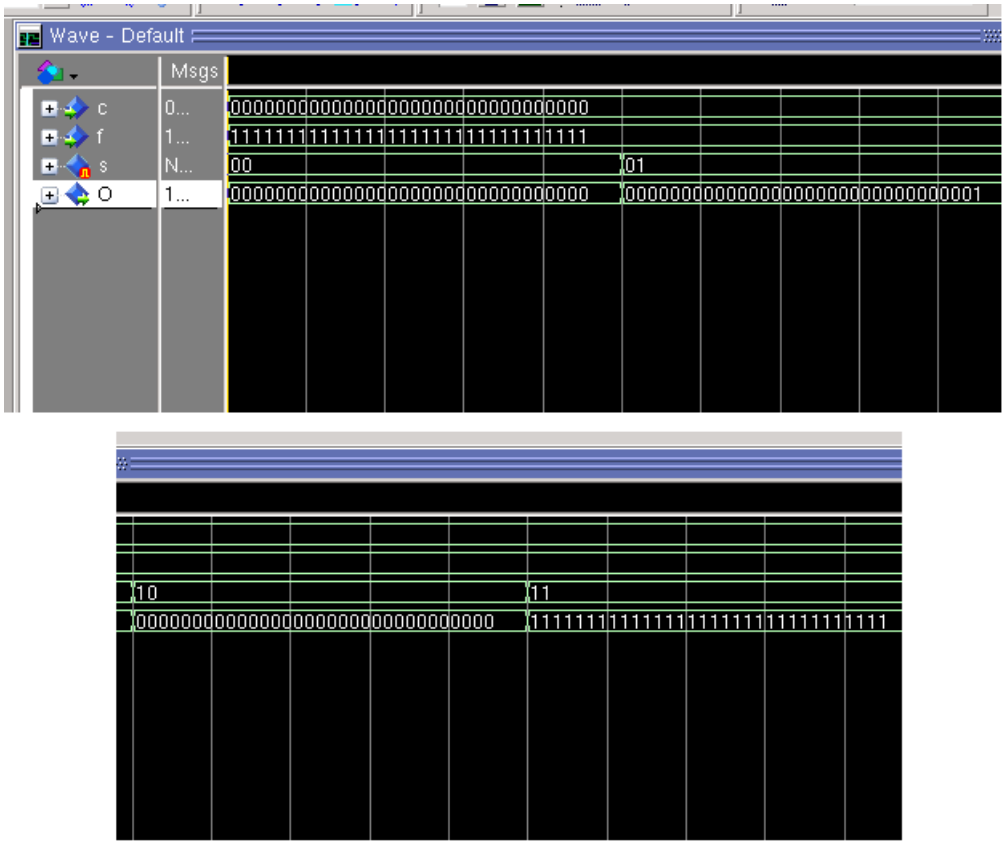

Figure 14: 4-to-1 Multiplexer

7 Describing Top Level Design in VHDL

Please refer to the source code.

8 Simulating the Whole System

For convenience, I wrote a testing framework so that I didn't have to set the `CLK` and `RST` manually every time.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity TESTBOARD is
    port(
            PC: out STD_LOGIC_VECTOR (31 downto 0);
            IR: out STD_LOGIC_VECTOR (31 downto 0)
        );
end TESTBOARD;

architecture STR of TESTBOARD is

    component LC2K
        port(
                rst: in  STD_LOGIC;
                clk: in  STD_LOGIC;
                outPC: out std_logic_vector (31 downto 0);
                outMem: out std_logic_vector (31 downto 0);
                outIR: out std_logic_vector (31 downto 0);
                outRF1: out std_logic_vector (31 downto 0);
                outRF2: out std_logic_vector (31 downto 0);
                inALU1: out std_logic_vector (31 downto 0);
                inALU2: out std_logic_vector (31 downto 0);
                outALU: out std_logic_vector (31 downto 0);
                outp: out std_logic_vector (31 downto 0)
            );
    end component;

    constant period : time := 25 ns;

    signal rst : STD_LOGIC;
    signal clk : STD_LOGIC;
    signal outMem: std_logic_vector (31 downto 0);
    signal outRF1: std_logic_vector (31 downto 0);
    signal outRF2: std_logic_vector (31 downto 0);
    signal inALU1: std_logic_vector (31 downto 0);
    signal inALU2: std_logic_vector (31 downto 0);
    signal outALU: std_logic_vector (31 downto 0);
    signal outp: std_logic_vector (31 downto 0);

begin
    RSTprocess : process
    begin
        rst <= '1';
        wait for period * 2;
        rst <= '0';
        wait;
```

```
48      end process RSTprocess;
49
50      CLKprocess : process
51      begin
52          clk <= '0';
53          wait for period;
54          clk <= '1';
55          wait for period;
56      end process CLKprocess;
57
58      U_CPU : LC2K port map (rst, clk, PC, outMem, IR, outRF1, outRF2,
59          inALU1, inALU2, outALU, outp);
60
61  end STR;
```

Besides, I also wrote a Python script to automatically convert machine codes (generated by `assemble` in Lab 1) to memory presentations in VHDL so that I didn't have to convert the machine codes to binary presentations and fill in the memory's VHDL code manually every time.

```
1   #!/usr/bin/env python2
2
3   buf = []
4   try:
5       while True:
6           x = input()
7           if not (-2 ** 31 <= x < 2 ** 31):
8               break
9           buf.append(x)
10  except (EOFError, TypeError):
11      pass
12
13  for addr, ins in enumerate(buf):
14      if ins >= 0:
15          print 'regs(%d) <= "%s";' % (addr,
16                  bin(ins)[2:].rjust(32, '0'))
17      else:
18          print 'regs(%d) <= "%s";' % (addr,
19                  bin((1 << 32) + ins)[2:])
```

8.1 Test Case 1: Basic Test

Assembly code:

```
1           lw      0       1       five    load reg1 with 5 (symbolic address)
2           lw      1       2       3       load reg2 with -1 (numeric address)
3   start   add     1       2       1       decrement reg1
4           beq     0       1       2       goto end of program when reg1 == 0
5           beq     0       0       start   go back to the beginning of the loop
6           noop
7   done    halt                            end of program
8   five    .fill   5
9   neg1    .fill   -1
10  stAddr  .fill   start                   will contain the address of start (2)
```

Assemble and convert the code.

```
$ ./assemble basic_test.asm basic_test.mc
$ ./convert_to_vhdl_mem.py < basic_test.mc > basic_test.mem
```

Then copy and paste the content of `basic_test.mem` to `TestMem.vhdl`. Now test it in ModelSim. The wave graph is shown in Figure 15.

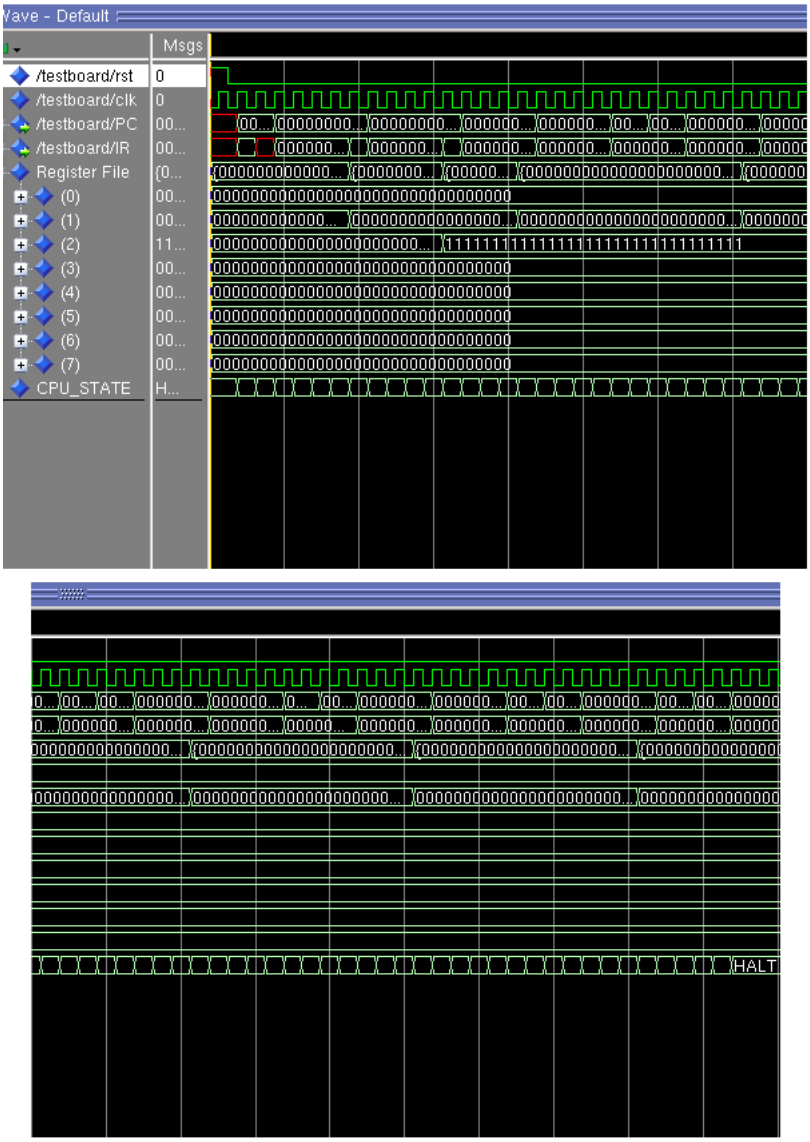

Figure 15: Wave Graph of Test 1

And the final state of the register file is shown in Figure 16.

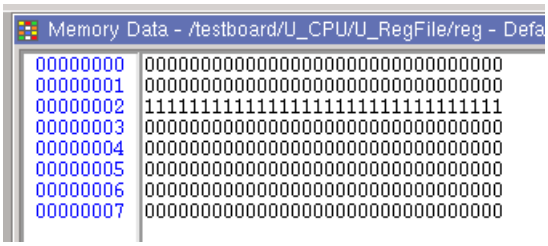

Figure 16: Final State of the Register File in Test 1

8.2 Test Case 2: Factorial Function

Assembly code:

```
init        lw      0   1   n
            lw      0   3   const+1
            lw      0   7   sp
            lw      0   4   const-2
            add     7   4   7
loop        beq     1   2   after-loop
            lw      0   4   const+1
            add     2   4   2
            sw      7   1   1
            sw      7   2   2
            add     0   3   1
            lw      0   4   mul-addr
            jalr    4   6
            lw      7   2   2
            lw      7   1   1
            beq     0   0   loop
after-loop  halt
mul         lw      0   4   const-2
            add     7   4   7
            sw      7   6   2
            sw      7   5   1
            add     0   0   3
            lw      0   5   mul-mask
            lw      0   6   mul-checker
mul-iter    nand    2   5   4
            beq     4   6   mul-skip
            add     3   1   3
mul-skip    add     5   5   5
            beq     5   0   mul-finish
            add     1   1   1
            beq     0   0   mul-iter
mul-finish  lw      7   5   1
            lw      7   6   2
            lw      0   4   const+2
            add     7   4   7
            jalr    6   4
const-2     .fill   -2
const+1     .fill   1
const+2     .fill   2
sp          .fill   255
mul-addr    .fill   mul
mul-mask    .fill   1
mul-checker .fill   -1
n           .fill   10
```

The final state of the register file is shown in Figure 17.

8.3 Test Case 3: k-Combinations of N

Assembly code:

```
init        lw      0   1   n
            lw      0   2   r
```

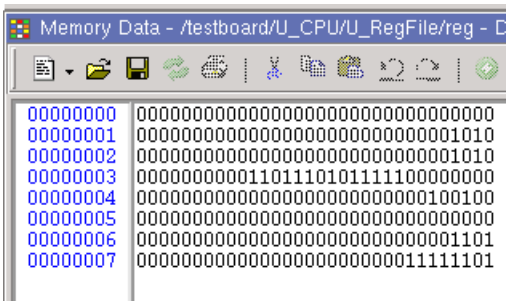

Figure 17: Final State of the Register File in Test 2

```
3                lw      0   7   init-sp
4                lw      0   4   comb-addr
5                jalr    4   6
6                halt
7   comb         beq     2   0   basic-sit
8                beq     1   2   basic-sit
9                lw      0   4   const-4
10               add     7   4   7
11               sw      7   6   4           save the return address
12               sw      7   2   3           save R2(r) (as a local variable)
13               sw      7   1   2           save R1(n) (as a local variable)
14               lw      0   4   const-1
15               add     1   4   1
16               lw      0   4   comb-addr
17               jalr    4   6
18               sw      7   3   1           save C(n - 1, r) to stack
19               lw      0   4   const-1
20               lw      7   1   2           get local variable n
21               lw      7   2   3           get local variable r
22               add     1   4   1
23               add     2   4   2
24               lw      0   4   comb-addr
25               jalr    4   6
26               lw      7   4   1
27               add     3   4   3           calulate C(n - 1, r - 1) + C(n - 1, r)
28               lw      7   6   4           restore R6
29               lw      0   4   const+4
30               add     7   4   7
31               jalr    6   4
32  basic-sit    lw      0   3   const+1
33               jalr    6   4
34  init-sp      .fill   255
35  const+1      .fill   1
36  const+4      .fill   4
37  const-1      .fill   -1
38  const-4      .fill   -4
39  comb-addr    .fill   comb
40  n            .fill   7
41  r            .fill   3
```

The final state of the register file is shown in Figure 18.
For more test cases, please refer to the source code.

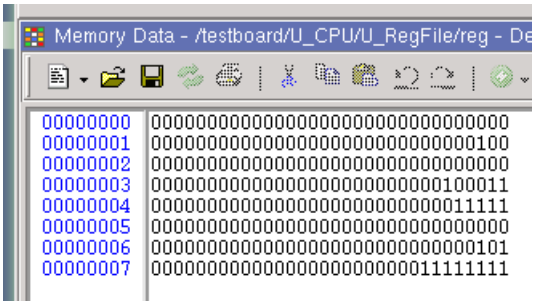

Figure 18: Final State of the Register File in Test 3

9 Conclusion and Discussion

In the simulation, the final states of the register file are identical to those of `simulate` in Lab 2 for all test cases. So it can be inferred that the CPU works as expected.

During the designing process, I noticed that the register file is not triggered by clock edge and the memory is asynchronous, which may require more states in the FSM to ensure the register file and the memory work stably. But I don't want to add more states to the FSM, so I change the designs of the register file and the memory. With the synchronous memory and register file that triggered by clock edge, the CPU do not need extra states to perform the write operations. But I haven't made good use of this feature by now, so the CPU can be optimized further.

10 Sum-up

In this experiment, I implemented a LC2K CPU with multiple cycle datapath. Through this, I came to know more about the underlying details of how the CPU work, which is indispensable in designing high-performance programs.