Lab Report

X

December 29, 2019

1 Introduction

This experiment uses verilog to design and implement a processor based on the RISC-V instruction set, and implement some instructions in the RV32I instruction set.

The Basic tasks

1. Design a single-cycle 32-bit RISC-V processor
2. Improved on the basis of single-cycle RISC-V processor to realize multi-cycle RISC-V processor
   (a) Single launch, five-stage pipeline
   (b) Implement data forwarding and pipeline blocking
   (c) branch prediction
3. The instructions that must be implemented include: add, addi, sub, and, or, xor, blt, beq, jal, sll, srl, lw, sw
4. Use modelsim for simulation verification and view the corresponding signal waveform
5. Use vivado to synthesize code to optimize timing as much as possible
6. Run the test instruction and get the correct output

The additional tasks completed

1. The rest instructions of RISC-V: sra, xor, sltu, slt, sub, add, srai, srli, slli, andi, ori, xori, sltiu, slti, sh, sb, lhu, lbu, lw, lh, lb, bgeu, bltu, bge, blt, bne, beq, jalr, lui, auipc
2. The simple prediction implement
3. The preliminary synthesis in Vivado

2 Lab procedures

2.1 The Single-cycle CPU

When the CPU processes instructions, it generally needs to go through five steps: instruction fetching, instruction decoding, instruction execution, memory access, and write back. A single-cycle CPU completes these five stages of processing in one clock cycle. In order to reduce the workload of designing Multi-cycle CPU, I implement the different states respectively.

2.1.1 instruction fetch (IF)

According to the instruction address in the program counter PC, an instruction is fetched from the instruction memory and sent to the decoding module. At the same time, the PC generates the instruction address required for the next instruction according to the auto-increment, but when a branch instruction is encountered, if the branch is established, the controller needs to send the jump address to the PC.

The verilog code:

```
1  module risc_v_32_if(clk,clrn,inst_i,MUX_1,PCSrc,pc,inst_o);
2      input          clk, clrn;                          // clock and reset
3      input  [31:0]  inst_i;                             // instruction
4      input  [31:0]  MUX_1;                              // addr for jump
5      input          PCSrc;                              // MUX choose signal
6      output [31:0] pc;                                  // program counter
7      output [31:0] inst_o;                              // instruction
8
9
10     reg    [31:0]   next_pc;                           // next pc
11     wire   [31:0]   pc_plus_4 = pc + 4;                // pc + 4
12
13     always @ (*) begin
14         if (PCSrc) next_pc = MUX_1;
15         else       next_pc = pc_plus_4;
16     end
17
18     // pc
19     reg  [31:0]      pc;
20     always @ (posedge clk or negedge clrn) begin
21         if (!clrn) pc <= 0;
22         else       pc <= next_pc;
23     end
24
25     assign inst_o = inst_i;
26
27 endmodule
```

2.1.2 Instruction Decoding (ID)

The instruction obtained in the instruction fetch operation is analyzed and decoded to determine the operation that this instruction needs to complete, so as to generate a corresponding operation control signal for driving various operations in the execution state.

The Verilog code:

```
1  module risc_v_32_id(inst,pc_i,pc_o,inst_decode,imm_out,rd1,rd2,wr);
2      input  [31:0]  inst;                               // instruction
3      input  [31:0]  pc_i;                               // program counter
4      output [31:0] pc_o;                                // program counter
5      output [36:0] inst_decode;                         // instruction decode, if inst_decode ==
           1, means ex instruction is the corresponding inst
6                                                         // see line 28-64 //attention for the
                                                              order
```

```
7   output reg [31:0] imm_out;                    // the extended immediate // already shift
8   output [4:0]  rd1;                            // = rs1
9   output [4:0]  rd2;                            // = rs2
10  output [4:0]  wr;                             // reg to write
11
12  // instruction format
13  wire [6:0]    opcode = inst[6:0];   //
14  wire [2:0]    func3  = inst[14:12]; //
15  wire [6:0]    func7  = inst[31:25]; //
16  wire [4:0]    rd     = inst[11:7];  //
17  wire [4:0]    rs     = inst[19:15]; // = rs1
18  wire [4:0]    rt     = inst[24:20]; // = rs2
19  wire [4:0]    shamt  = inst[24:20]; // == rs2;
20  wire          sign   = inst[31];
21
22  assign rd1  = rs;
23  assign rd2  = rt;
24  assign wr   = rd;
25  assign pc_o = pc_i;
26
27
28  // instruction decode
29  wire          i_auipc = (opcode == 7'b0010111); // auipc
30  wire          i_lui   = (opcode == 7'b0110111); // lui
31  wire          i_jal   = (opcode == 7'b1101111); // jal
32  wire          i_jalr  = (opcode == 7'b1100111) & (func3 == 3'b000); // jalr
33  wire          i_beq   = (opcode == 7'b1100011) & (func3 == 3'b000); // beq
34  wire          i_bne   = (opcode == 7'b1100011) & (func3 == 3'b001); // bne
35  wire          i_blt   = (opcode == 7'b1100011) & (func3 == 3'b100); // blt
36  wire          i_bge   = (opcode == 7'b1100011) & (func3 == 3'b101); // bge
37  wire          i_bltu  = (opcode == 7'b1100011) & (func3 == 3'b110); // bltu
38  wire          i_bgeu  = (opcode == 7'b1100011) & (func3 == 3'b111); // bgeu
39  wire          i_lb    = (opcode == 7'b0000011) & (func3 == 3'b000); // lb
40  wire          i_lh    = (opcode == 7'b0000011) & (func3 == 3'b001); // lh
41  wire          i_lw    = (opcode == 7'b0000011) & (func3 == 3'b010); // lw
42  wire          i_lbu   = (opcode == 7'b0000011) & (func3 == 3'b100); // lbu
43  wire          i_lhu   = (opcode == 7'b0000011) & (func3 == 3'b101); // lhu
44  wire          i_sb    = (opcode == 7'b0100011) & (func3 == 3'b000); // sb
45  wire          i_sh    = (opcode == 7'b0100011) & (func3 == 3'b001); // sh
46  wire          i_sw    = (opcode == 7'b0100011) & (func3 == 3'b010); // sw
47  wire          i_addi  = (opcode == 7'b0010011) & (func3 == 3'b000); // addi
48  wire          i_slti  = (opcode == 7'b0010011) & (func3 == 3'b010); // slti
49  wire          i_sltiu = (opcode == 7'b0010011) & (func3 == 3'b011); // sltiu
50  wire          i_xori  = (opcode == 7'b0010011) & (func3 == 3'b100); // xori
51  wire          i_ori   = (opcode == 7'b0010011) & (func3 == 3'b110); // ori
52  wire          i_andi  = (opcode == 7'b0010011) & (func3 == 3'b111); // andi
53  wire          i_slli  = (opcode == 7'b0010011) & (func3 == 3'b001) & (func7 == 7'b0000000
        ); // slli
54  wire          i_srli  = (opcode == 7'b0010011) & (func3 == 3'b101) & (func7 == 7'b0000000
        ); // srli
55  wire          i_srai  = (opcode == 7'b0010011) & (func3 == 3'b101) & (func7 == 7'b0100000
        ); // srai
```

```
56      wire          i_add  = (opcode == 7'b0110011) & (func3 == 3'b000) & (func7 == 7'b0000000
            ); // add
57      wire          i_sub  = (opcode == 7'b0110011) & (func3 == 3'b000) & (func7 == 7'b0100000
            ); // sub
58      wire          i_sll  = (opcode == 7'b0110011) & (func3 == 3'b001) & (func7 == 7'b0000000
            ); // sll
59      wire          i_slt  = (opcode == 7'b0110011) & (func3 == 3'b010) & (func7 == 7'b0000000
            ); // slt
60      wire          i_sltu = (opcode == 7'b0110011) & (func3 == 3'b011) & (func7 == 7'b0000000
            ); // sltu
61      wire          i_xor  = (opcode == 7'b0110011) & (func3 == 3'b100) & (func7 == 7'b0000000
            ); // xor
62      wire          i_srl  = (opcode == 7'b0110011) & (func3 == 3'b101) & (func7 == 7'b0000000
            ); // srl
63      wire          i_sra  = (opcode == 7'b0110011) & (func3 == 3'b101) & (func7 == 7'b0100000
            ); // sra
64      wire          i_or   = (opcode == 7'b0110011) & (func3 == 3'b110) & (func7 == 7'b0000000
            ); // or
65      wire          i_and  = (opcode == 7'b0110011) & (func3 == 3'b111) & (func7 == 7'b0000000
            ); // and
66
67      assign inst_decode = {i_and,i_or,i_sra,i_srl,i_xor,i_sltu,i_slt,i_sll,i_sub,i_add,i_srai,i_srli
            ,i_slli,i_andi,i_ori,i_xori,i_sltiu,i_slti,i_addi,i_sw,i_sh,i_sb,i_lhu,i_lbu,i_lw,i_lh,
            i_lb,i_bgeu,i_bltu,i_bge,i_blt,i_bne,i_beq,i_jalr,i_jal,i_lui,i_auipc};
68
69      // branch offset          31:13          12      11       10:5          4:1      0
70      wire  [31:0]  broffset  = {{19{sign}},inst[31], inst[7], inst[30:25], inst[11:8],1'b0};   // beq,
            bne, blt, bge, bltu, bgeu
71      wire  [31:0]  simm      = {{20{sign}},inst[31:20]};                                       // lw,
            addi, slti, sltiu, xori, ori, andi
72      wire  [31:0]  jroffset  = {{20{sign}},inst[31:21],1'b0};                                  // jalr
73      wire  [31:0]  stimm     = {{20{sign}},inst[31:25], inst[11:7]};                           // sw
74      wire  [31:0]  uimm      = {inst[31:12],12'h0};                                            // lui,
            auipc
75      wire  [31:0]  jaloffset = {{11{sign}},inst[31], inst[19:12], inst[20], inst[30:21],1'b0}; // jal
76      // jal target          31:21          20      19:12       11       10:1      0
77
78
79      //determaine which extended immediate number to putout
80      //shift already been done
81      wire   [5:0]  outselect = {i_beq|i_bne|i_blt|i_bge|i_bltu|i_bgeu, i_lw|i_addi|i_slti|i_sltiu|
            i_xori|i_ori|i_andi, i_jalr, i_sw, i_lui|i_auipc, i_jal};
82
83      always @(*)
84      case (outselect)
85          6'b100000: imm_out = broffset;
86          6'b010000: imm_out = simm;
87          6'b001000: imm_out = jroffset;
88          6'b000100: imm_out = stimm;
89          6'b000010: imm_out = uimm;
90          6'b000001: imm_out = jaloffset;
91          default: ;
```

```
92      endcase
93
94  endmodule
```

2.1.3 Execute (EX)

According to the ALU control signal obtained by the instruction decoding, the instruction operation is executed specifically.

The implementation of each instruction will be shown in Verilog next.

```
1       always @(*) begin                       // 30 instructions
2           alu_out = 0;                        // alu output
3   //      mem_out = 0;                        // mem output
4           m_addr = 0;                         // memory address
5           wreg    = 0;                        // write regfile
6           wmem    = 0;                        // write memory (sw)
7           rmem    = 0;                        // read memory (lw)
8           PCSrc   = 0;                        // pc MUX control
9           case (1'b1)
10              i_add: begin                    // add
11                  alu_out = a + b;
12                  wreg    = 1; end
13              i_sub: begin                    // sub
14                  alu_out = a - b;
15                  wreg    = 1; end
16              i_and: begin                    // and
17                  alu_out = a & b;
18                  wreg    = 1; end
19              i_or: begin                     // or
20                  alu_out = a | b;
21                  wreg    = 1; end
22              i_xor: begin                    // xor
23                  alu_out = a ^ b;
24                  wreg    = 1; end
25              i_sll : begin                   // sll
26                  alu_out = a << b[4:0];
27                  wreg    = 1; end
28              i_srl : begin                   // srl
29                  alu_out = a >> b[4:0];
30                  wreg    = 1; end
31              i_sra: begin                    // sra
32                  alu_out = $signed(a) >>> b[4:0];
33                  wreg    = 1; end
34              i_slli : begin                  // slli
35                  alu_out = a << shamt;
36                  wreg    = 1; end
37              i_srli : begin                  // srli
38                  alu_out = a >> shamt;
39                  wreg    = 1; end
40              i_srai : begin                  // srai
41                  alu_out = $signed(a) >>> shamt;
```

```
42                  wreg    = 1; end
43              i_slt : begin                                   // slt
44                  if ($signed(a) < $signed(b))
45                    alu_out = 1; end
46              i_sltu: begin                                   // sltu
47                  if ({1'b0,a} < {1'b0,b})
48                    alu_out = 1; end
49              i_addi: begin                                   // addi
50                  alu_out = a + imm_in;
51                  wreg    = 1; end
52              i_andi: begin                                   // andi
53                  alu_out = a & imm_in;
54                  wreg    = 1; end
55              i_ori: begin                                    // ori
56                  alu_out = a | imm_in;
57                  wreg    = 1; end
58              i_xori: begin                                   // xori
59                  alu_out = a ^ imm_in;
60                  wreg    = 1; end
61              i_slti : begin                                  // slti
62                  if ($signed(a) < $signed(imm_in))
63                    alu_out = 1; end
64              i_sltiu : begin                                 // sltiu
65                  if ({1'b0,a} < {1'b0,imm_in})
66                    alu_out = 1; end
67              i_lw: begin                                     // lw
68                  alu_out = a + imm_in;
69                  m_addr = {alu_out[31:2],2'b00};             // alu_out[1:0] != 0, exception
70                  rmem    = 1;
71                //mem_out = d_f_mem;
72                  wreg    = 1; end
73              i_lbu: begin                                    // lbu
74                  alu_out = a + imm_in;
75                  m_addr = alu_out;
76                  rmem    = 1;
77                 /* case(m_addr[1:0])
78                    2'b00: mem_out = {24'h0,d_f_mem[ 7: 0]};
79                    2'b01: mem_out = {24'h0,d_f_mem[15: 8]};
80                    2'b10: mem_out = {24'h0,d_f_mem[23:16]};
81                    2'b11: mem_out = {24'h0,d_f_mem[31:24]};
82                  endcase*/
83                  wreg    = 1; end
84              i_lb: begin                                     // lb
85                  alu_out = a + imm_in;
86                  m_addr = alu_out;
87                  rmem    = 1;
88                  /*case(m_addr[1:0])
89                    2'b00: mem_out = {{24{d_f_mem[ 7]}},d_f_mem[ 7: 0]};
90                    2'b01: mem_out = {{24{d_f_mem[15]}},d_f_mem[15: 8]};
91                    2'b10: mem_out = {{24{d_f_mem[23]}},d_f_mem[23:16]};
92                    2'b11: mem_out = {{24{d_f_mem[31]}},d_f_mem[31:24]};
93                  endcase*/
```

```
94              wreg    = 1; end
95          i_lhu: begin                                    // lhu
96              alu_out = a + imm_in;
97              m_addr = {alu_out[31:1],1'b0};              // alu_out[0] != 0, exception
98              rmem    = 1;
99              /*case(m_addr[1])
100               1'b0: mem_out = {16'h0,d_f_mem[15: 0]};
101               1'b1: mem_out = {16'h0,d_f_mem[31:16]};
102             endcase*/
103             wreg    = 1; end
104         i_lh: begin                                     // lh
105             alu_out = a + imm_in;
106             m_addr = {alu_out[31:1],1'b0};              // alu_out[0] != 0, exception
107             rmem    = 1;
108             /*case(m_addr[1])
109               1'b0: mem_out = {{16{d_f_mem[15]}},d_f_mem[15: 0]};
110               1'b1: mem_out = {{16{d_f_mem[31]}},d_f_mem[31:16]};
111             endcase*/
112             wreg    = 1; end
113         i_sb: begin                                     // sb
114             alu_out = a + imm_in;
115             m_addr = alu_out;
116             wmem    = 1; end
117         i_sh: begin                                     // sh
118             alu_out = a + imm_in;
119             m_addr = {alu_out[31:1],1'b0};              // alu_out[0] != 0, exception
120             wmem    = 1; end
121         i_sw: begin                                     // sw
122             alu_out = a + imm_in;
123             m_addr = {alu_out[31:2],2'b00};             // alu_out[1:0] != 0, exception
124             wmem    = 1; end
125         i_beq: begin                                    // beq
126             if (a == b) begin
127               next_pc = pc + imm_in;
128               PCSrc   = 1; end end
129         i_bne: begin                                    // bne
130             if (a != b) begin
131               next_pc = pc + imm_in;
132               PCSrc   = 1; end end
133         i_blt: begin                                    // blt
134             if ($signed(a) < $signed(b)) begin
135               next_pc = pc + imm_in;
136               PCSrc   = 1; end end
137         i_bge: begin                                    // bge
138             if ($signed(a) >= $signed(b)) begin
139               next_pc = pc + imm_in;
140               PCSrc   = 1; end end
141         i_bltu: begin                                   // bltu
142             if ({1'b0,a} < {1'b0,b}) begin
143               next_pc = pc + imm_in;
144               PCSrc   = 1; end end
145         i_bgeu: begin                                   // bgeu
```

```
                if ({1'b0,a} >= {1'b0,b}) begin
                  next_pc = pc + imm_in;
                  PCSrc  = 1; end end
            i_auipc: begin                                  // auipc
                alu_out = pc + imm_in;
                wreg    = 1; end
            i_lui: begin                                    // lui
                alu_out = imm_in;
                wreg    = 1; end
            i_jal: begin                                    // jal
                alu_out = pc_plus_4;
                wreg    = 1;
                next_pc = pc + imm_in;
                PCSrc  = 1; end
            i_jalr: begin                                   // jalr
                alu_out = pc_plus_4;
                wreg    = 1;
                next_pc = a + imm_in;
                PCSrc  = 1; end
            default: ;
        endcase
    end
```

2.1.4 Memory access (MEM)

All operations that need to access the memory will be performed in this step. This step writes data to the storage unit (store word) specified by the data address in the memory or obtains the data in the data address unit (load word) from the memory.

The verilog code:

```
module risc_v_32_mem(m_addr,d_f_mem,inst_decode,mem_out);

    input  [31:0]  m_addr;                                  // mem or i/o addr
    input  [31:0]  d_f_mem;                                 // load data
    input  [36:0]  inst_decode;                             // instruction decode, if inst_decode
        == 1, means ex instruction is the corresponding inst

    output reg [31:0] mem_out;                                  // mem output

    // instruction
    wire          i_lb    = inst_decode[10]; // lb
    wire          i_lh    = inst_decode[11]; // lh
    wire          i_lw    = inst_decode[12]; // lw
    wire          i_lbu   = inst_decode[13]; // lbu
    wire          i_lhu   = inst_decode[14]; // lhu



    always @(*) begin                                       // load instructions
        mem_out = 0;                                        // mem output
        case (1'b1)
            i_lw: begin                                     // lw
```

```
22                  mem_out = d_f_mem;end
23              i_lbu: begin                                    // lbu
24                  case(m_addr[1:0])
25                    2'b00: mem_out = {24'h0,d_f_mem[ 7: 0]};
26                    2'b01: mem_out = {24'h0,d_f_mem[15: 8]};
27                    2'b10: mem_out = {24'h0,d_f_mem[23:16]};
28                    2'b11: mem_out = {24'h0,d_f_mem[31:24]};
29                  endcase end
30              i_lb: begin                                     // lb
31                  case(m_addr[1:0])
32                    2'b00: mem_out = {{24{d_f_mem[ 7]}},d_f_mem[ 7: 0]};
33                    2'b01: mem_out = {{24{d_f_mem[15]}},d_f_mem[15: 8]};
34                    2'b10: mem_out = {{24{d_f_mem[23]}},d_f_mem[23:16]};
35                    2'b11: mem_out = {{24{d_f_mem[31]}},d_f_mem[31:24]};
36                  endcase end
37              i_lhu: begin                                    // lhu
38                  case(m_addr[1])
39                    1'b0: mem_out = {16'h0,d_f_mem[15: 0]};
40                    1'b1: mem_out = {16'h0,d_f_mem[31:16]};
41                  endcase end
42              i_lh: begin                                     // lh
43                  case(m_addr[1])
44                    1'b0: mem_out = {{16{d_f_mem[15]}},d_f_mem[15: 0]};
45                    1'b1: mem_out = {{16{d_f_mem[31]}},d_f_mem[31:16]};
46                  endcase end
47              default: ;
48          endcase
49      end
50
51
52  endmodule
```

2.1.5 Write Back (WB)

The result of the instruction execution or the data obtained by accessing the memory is written back to the corresponding destination register.

The verilog code:

```
1  module risc_v_32_wb(mem_out,alu_out,MemtoReg,data_2_rf);
2      input      [31:0]  mem_out;                    // Read data from data memory
3      input      [31:0]  alu_out;                    // ALU output
4      input              MemtoReg;                   // MUX choose signal
5      output     [31:0]  data_2_rf;                  // data write to  register  file
6
7      wire   [31:0]  data_2_rf = MemtoReg ? mem_out : alu_out;
8
9  endmodule
```

2.1.6 Others

In addition, it is necessary to write a register file module, read the register and obtain the operand according to the source register number obtained by the ID module, and write the destination register

number and data given by the WB module back to the corresponding register.

The verilog code:

```
1  module risc_v_32_regfile(clk,clrn,rd1,rd2,wr,wd,wreg,read_data1,read_data2);
2      input         clk, clrn;                                    // clock and reset
3      input  [4:0]  rd1;                                          // read register1
4      input  [4:0]  rd2;                                          // read register2
5      input  [4:0]  wr;                                           // write register
6      input  [31:0] wd;                                           // write data
7      input         wreg;                                         // if == 1, write register
8      output [31:0] read_data1;
9      output [31:0] read_data2;
10
11     reg  [31:0]  regfile  [1:31];
12
13     wire [31:0]   read_data1 = (rd1==0) ? 0 : regfile[rd1];     // read port
14     wire [31:0]   read_data2 = (rd2==0) ? 0 : regfile[rd2];     // read port
15
16     always @ (posedge clk) begin
17         if (wreg && (wr != 0)) begin
18             regfile [wr] <= wd;                    // write port
19         end
20     end
21
22 endmodule
```

2.2 Multi-cycle CPU

The overall processing process in multiple cycles is divided into five levels of IF, ID, EX, MEM, and WB, corresponding to the five processing stages of multiple cycles. The execution of an instruction requires 5 clock cycles. When the rising edge of each clock cycle comes, a series of data and control information represented by this instruction will be transferred to the next level of processing. Therefore, each of them needs to be added on a single cycle basis. Registers between levels.

In addition, the pipeline will face the situation that the next instruction cannot be executed in one clock cycle, that is, the pipeline is blocked, so the forwarding and hazard control modules must be added to the pipeline design to avoid data hazard or control hazard. When blocking occurs, keep the PC and the current fetch instruction unchanged, and clear the control signals at the IF/ID level.

Therefore, the five-stage pipeline CPU design includes if, if/id reg, id, id/ex reg, ex, ex/mem reg, mem, mem/wb reg, wb, hazard detect, and forwarding modules under the top riscv module.

To avoid redundancy, the following only talk about the forwarding, hazard detect, registers between different stages and control hazard handle by prediction. The correspond part modified in other modules.

2.2.1 Forwarding

When a data hazard occurs, the data needs to be pushed forward before being written back to the register, otherwise the data read from the register file may be data that has not yet been updated. When the destination register number of the previous instruction is the same as the source register number of the current instruction, forwarding is required. Forwarding data may come from EX-stage ALUresult, MEM-stage ALUresult, or load-use hazard from the result of data storage access. The specific analysis can be got in the book Computer Organization and Design, section 4.7, Chapter 4.

The verilog code:

```
1  module risc_v_32_forward(a1,a2,a3,b1,b2,b3,idex_rs1,idex_rs2,exmem_wreg,exmem_rd,
       memwb_wreg,memwb_rd,a,b);
2
3      input       [31:0]  a1,a2,a3;                       //MUX1 inputs
4      input       [31:0]  b1,b2,b3;                       //MUX2 inputs
5      input        [4:0]  idex_rs1,idex_rs2;              //rs1,rs2
6      input               exmem_wreg,memwb_wreg;          //write register
7      input        [4:0]  exmem_rd,memwb_rd;              //rd
8
9      output reg [31:0] a,b;                              //output to alu/ex stage
10
11     reg [1:0] forward_a;
12     reg [1:0] forward_b;
13
14     always @ (*) begin
15         forward_a = 0;
16         forward_b = 0;
17         if (exmem_wreg && (exmem_rd != 0) && (exmem_rd == idex_rs1))
                                                            forward_a = 2'b10;
18         if (exmem_wreg && (exmem_rd != 0) && (exmem_rd == idex_rs2))
                                                            forward_b = 2'b10;
19         if (memwb_wreg && (memwb_rd != 0) && !(exmem_wreg && (exmem_rd != 0) && (
               exmem_rd == idex_rs1)) && (memwb_rd == idex_rs1)) forward_a = 2'b01;
20         if (memwb_wreg && (memwb_rd != 0) && !(exmem_wreg && (exmem_rd != 0) && (
               exmem_rd == idex_rs2)) && (memwb_rd == idex_rs2)) forward_b = 2'b01;
21     end
22
23     //MUX1
24     always @ (*) begin
25         a = a1;
26         case (forward_a)
27             2'b00:  a = a1;
28             2'b01:  a = a2;
29             2'b10:  a = a3;
30         endcase
31     end
32
33     //MUX2
34     always @ (*) begin
35         b = b1;
36         case (forward_b)
37             2'b00:  b = b1;
38             2'b01:  b = b2;
39             2'b10:  b = b3;
40         endcase
41     end
42 endmodule
```

2.2.2 hazard detection

Load-use data hazard not only needs to be forwarded, but also PC and IF/ID register stall, so the PCwre signal is given to control the blocking of the pipeline. rs1 and rs2 are to ensure that the source register number divided from the instruction is meaningful. MemRead = 1 indicates that the previous

instruction is an load instruction.

The verilog code:

```
1  module risc_v_32_hazard(memread,inst_decode,ifid_rs1,ifid_rs2,idex_rd,pc_write,ifid_write,
       MUX_out);
2
3      input          memread;                          // memory read
4      input   [36:0] inst_decode;                      // instruction decode, if inst_decode == 1,
           means ex instruction is the corresponding inst
5      input    [4:0] ifid_rs1;                         // rs1
6      input    [4:0] ifid_rs2;                         // rs2
7      input    [4:0] idex_rd;                          // rd
8
9      output         pc_write;                         // update pc
10     output         ifid_write;                       // update ifid  regiest
11     output  [36:0] MUX_out;                          // MUX output
12
13     wire           stall;
14     reg     [36:0] MUX_out;
15
16 /* always @ (*) begin
17         stall = 0;
18         if (memread && ((idex_rd == ifid_rs1) || (idex_rd == ifid_rs2))) stall = 1;
19     end
20 */
21     assign stall = (memread && ((idex_rd == ifid_rs1) || (idex_rd == ifid_rs2)));
22
23     assign   pc_write = ~stall;
24     assign ifid_write = ~stall;
25
26     //MUX
27     always @ (stall, inst_decode) begin
28         MUX_out = 0;
29         case (~stall)
30             1'b0:   MUX_out = 0;
31             1'b1:   MUX_out = inst_decode;
32         endcase
33     end
34
35 endmodule
```

2.2.3 Registers between different stages

Registers are used to atore and pass data. The ID/EX register as an example is shown below.

The verilog code:

```
1  module idexreg(clk,clrn,PCsrc,inst_i,pc_i,inst_decode_i,imm_in,read_data1_i,read_data2_i,wr_i,
       rs1_i,rs2_i,inst_o,pc_o,inst_decode_o,imm_out,read_data1_o,read_data2_o,wr_o,rs1_o,
       rs2_o);
2      input           clk, clrn;                       // clock and reset
3      input           PCsrc;                           // pc MUX control signal
```

```
4       input       [31:0]  pc_i;                                    // program counter
5       input       [31:0]  inst_i;                                  // instruction
6       input       [36:0]  inst_decode_i;                           // instruction decode, if inst_decode
            == 1, means ex instruction is the corresponding inst
7       input       [31:0]  imm_in;                                  // the extended immediate // already
            shift
8       input       [31:0]  read_data1_i;                            // read_data1 from regfile
9       input       [31:0]  read_data2_i;                            // read_data2 from regfile
10      input        [4:0]  wr_i;                                    // reg to write
11      input        [4:0]  rs1_i;                                   // reg1 to read
12      input        [4:0]  rs2_i;                                   // reg2 to read
13
14      output reg [31:0] pc_o;                                      // program counter
15      output reg [31:0] inst_o;                                    // instruction
16      output reg [36:0] inst_decode_o;                             // instruction decode, if inst_decode
            == 1, means ex instruction is the corresponding inst
17      output reg [31:0] imm_out;                                   // the extended immediate // already
            shift
18      output reg [31:0] read_data1_o;                              // read_data1 from regfile
19      output reg [31:0] read_data2_o;                              // read_data2 from regfile
20      output reg [4:0]  wr_o;                                      // reg to write
21      output reg [4:0]  rs1_o;                                     // reg1 to read
22      output reg [4:0]  rs2_o;                                     // reg2 to read
23
24      always @ (posedge clk) begin
25          if (clrn && (!PCsrc)) begin
26              pc_o          <= pc_i;
27              inst_o        <= inst_i;
28              inst_decode_o <= inst_decode_i;
29              imm_out       <= imm_in;
30              read_data1_o <= read_data1_i;
31              read_data2_o <= read_data2_i;
32              wr_o          <= wr_i;
33              rs1_o         <= rs1_i;
34              rs2_o         <= rs2_i;
35          end
36          else begin
37              pc_o          <= 0;
38              inst_o        <= 0;
39              inst_decode_o <= 0;
40              imm_out       <= 0;
41              read_data1_o <= 0;
42              read_data2_o <= 0;
43              wr_o          <= 0;
44              rs1_o         <= 0;
45              rs2_o         <= 0;
46          end
47      end
48
49  endmodule
```

2.2.4 Prediction

Stalling until the branch is complete is too slow. One improvement over branch stalling is to predict that the conditional branch will not be taken and thus continue execution down the sequential instruction stream. If the conditional branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target. If conditional branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.

The PCSrc will be modified if branch is taken. The PCSrc assert produce in ex state shown following, there are also other modify in other modules.

The verilog code:

```
1        i_beq: begin                                    // beq
2            if (a == b) begin
3              next_pc = pc + imm_in;
4              PCSrc  = 1; end end
5        i_bne: begin                                    // bne
6            if (a != b) begin
7              next_pc = pc + imm_in;
8              PCSrc  = 1; end end
9        i_blt: begin                                    // blt
10           if ($signed(a) < $signed(b)) begin
11             next_pc = pc + imm_in;
12             PCSrc  = 1; end end
13       i_bge: begin                                    // bge
14           if ($signed(a) >= $signed(b)) begin
15             next_pc = pc + imm_in;
16             PCSrc  = 1; end end
17       i_bltu: begin                                   // bltu
18           if ({1'b0,a} < {1'b0,b}) begin
19             next_pc = pc + imm_in;
20             PCSrc  = 1; end end
21       i_bgeu: begin                                   // bgeu
22           if ({1'b0,a} >= {1'b0,b}) begin
23             next_pc = pc + imm_in;
24             PCSrc  = 1; end end
25       i_jal: begin                                    // jal
26           alu_out = pc_plus_4;
27           wreg    = 1;
28           next_pc = pc + imm_in;
29           PCSrc  = 1; end
30       i_jalr: begin                                   // jalr
31           alu_out = pc_plus_4;
32           wreg    = 1;
33           next_pc = a + imm_in;
34           PCSrc  = 1; end
```

3 Lab results

3.1 Using fault test code

The test assembly code is as follows:

```
1       addi x1,x0,10;  iterations
2       addi x2,x0,0
3       addi x3,x0,0
4       addi x6,x0,2
5       loop1:
6        sll   x4,x3,x6
7       lw    x5,0(x4)
8       add  x2,x2,x5;   sum
9       addi x3,x3,1
10      bge   x3,x1,loop2
11       jal   x0,loop1
12      loop2:
13      sw    x2,0(x0)
```

3.1.1 Single-cycle CPU

The result is shown in figure 1.

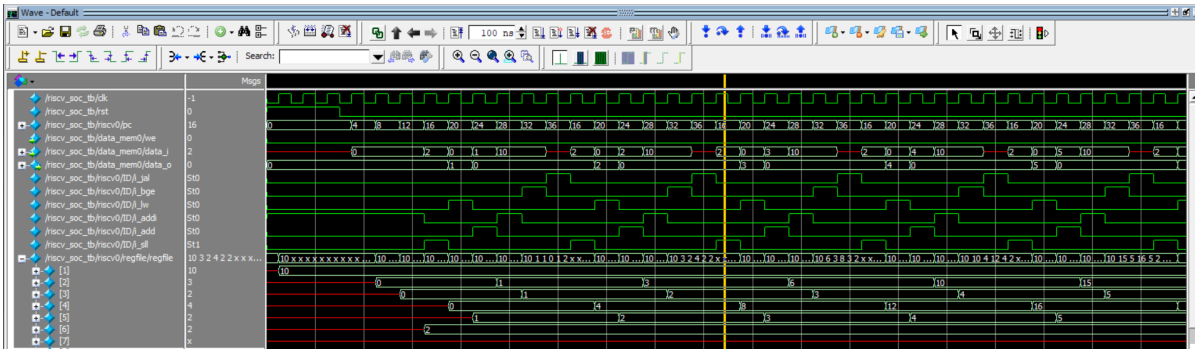

Figure 1: Single-cycle CPU using fault test code

As can be seen from the figure, the cpu executes an instruction every clock cycle, and the pc increases by 4. When the PC is 0 to 12, the executed instruction is addi, which indicates that the addition operation is performed. At this time, i_addi is at a high potential, and ALUresult is written into the corresponding destination register. When the PC is 16, the executed instruction is sll, which indicates that the left shift operation is performed. At this time, i_sll is high, and ALUresult is written to x4. When the PC is 20, the executed instruction is lw, which indicates that the addition operation is performed. At this time, i_lw is high and MemRead is also high. Access the data memory and write the result to x5. When the PC is 36, the executed instruction is jal, the PCsrc is high, the imme is -20, and the PC is updated to 16 accordingly to implement the instruction jump.

3.1.2 Multi-cycle CPU

The result is shown in figure 2.

It can be seen from the figure that the execution of each instruction requires 5 clock cycles, and the data is written back to the register file in the last clock cycle. It is observed that when pc is 24, the executed instruction is add x2, x2, x5 and there is data correlation with the previous instruction lw x5,0 (x4), so the pipeline is blocked for one clock cycle, pc remains at 24, while forwarding x5 Data and write ALUresult back to x2. When pc is 36, the prediction branch is not executed, and PCSrc is 1, thereby clearing the IF / ID register. It is observed that the register value is the same as the single-cycle CPU simulation result, and the test result is correct.

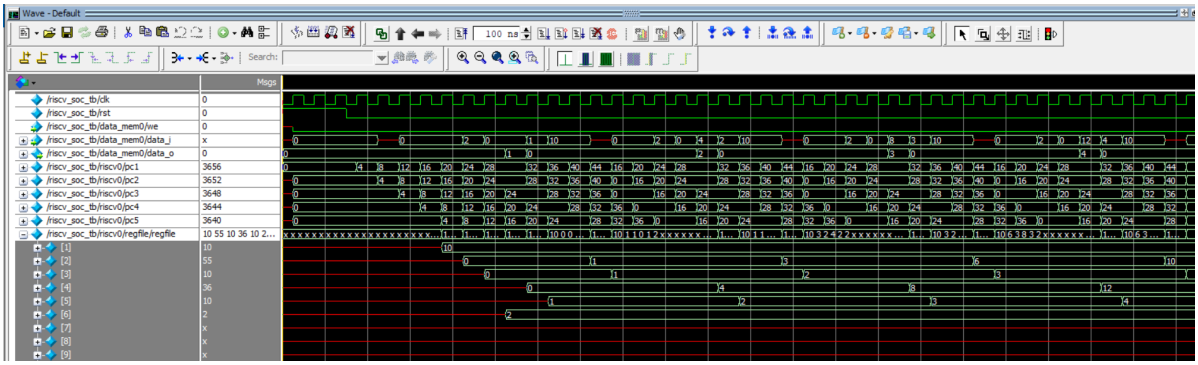

Figure 2: Multi-cycle CPU using fault test code

3.2 Using custom test code

The test assembly code is as follows:

```
1     addi x1,x0,1;  iterations
2     addi x2,x1,5
3     addi x10,x0,1
4     addi x11,x0,1
5     addi x12,x0,0
6     addi x13,x0,3
7     add x4,x1,x2
8     loop1:
9     xor x11,x10,x12
10    or  x2,x11,x13
11    sub x10,x11,x2
12     sll  x12,x11,x1
13    addi x2,x2,1
14    addi x3,x2,4
15    lw x5,5(x3)
16    add x9,x5,x2
17    addi x6,x3,2
18    sub x7,x3,x6
19     srl  x8,x2,x1
20    blt  x3,x6,loop1
21     srl  x8,x2,x1
22     jal  x0,loop1
```

This test code made a data address not a multiple of 4, which is an exception. From the result, we can see the CPU handle it properly.

3.2.1 Single-cycle CPU

The result is shown in figure 3.

It is observed that the instruction is xor and XOR operation is performed, the source operands are 1 and 0, the ALU calculation result is 1, the result is correct; when the instruction is or, the source operand is 1 and 3, the ALU calculation result is 3, the result is correct; When the instruction is sub, the subtraction operation is performed. The source operands are 1 and 3. The ALU calculation result is -2, and the result is correct. When the instruction is srl, i_srl is high level at this time to perform a right shift operation, and the result is correct. blt, at this time i_blt is high, the condition is met, and the branch jumps.

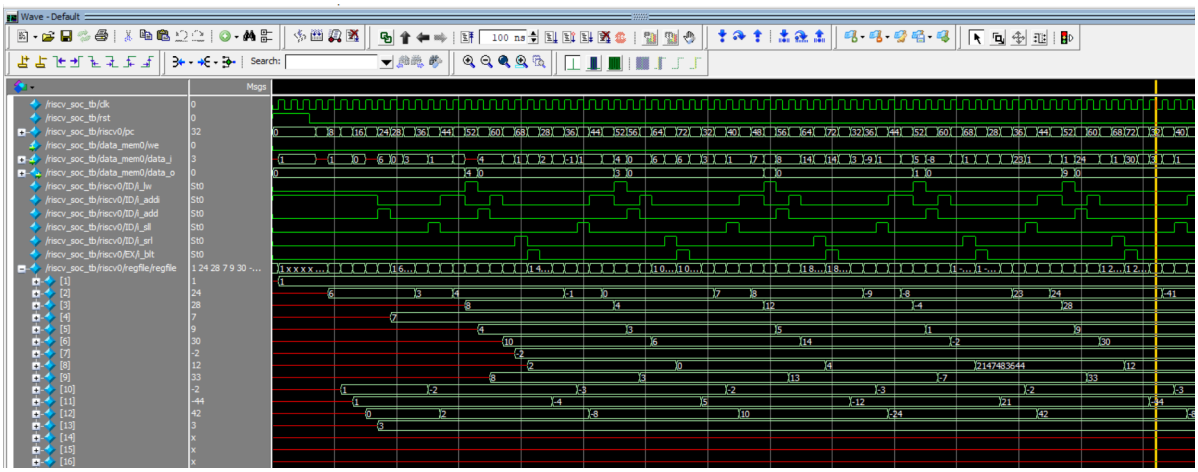

Figure 3: Single-cycle CPU using custom test code

3.2.2 Multi-cycle CPU

The result is shown in figure 4.

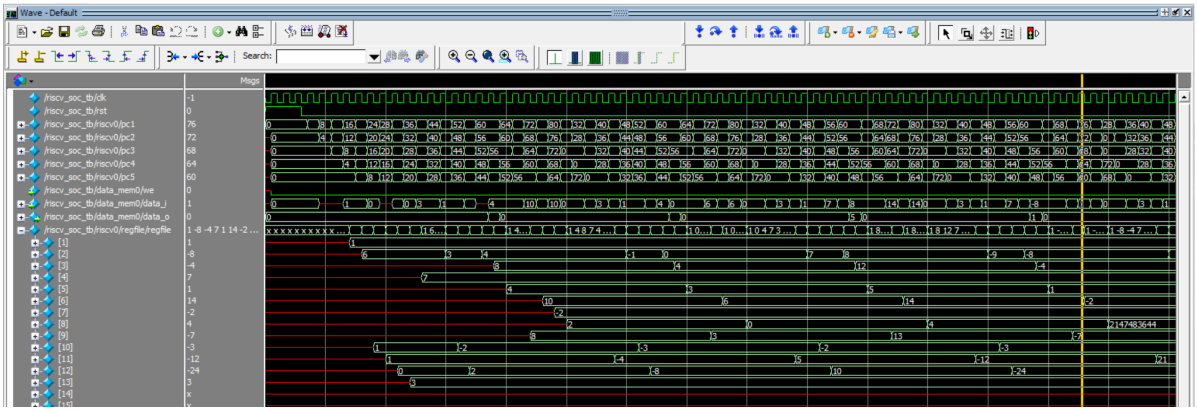

Figure 4: Multi-cycle CPU using custom test code

When the instruction is lw, and the source register x5 in the next instruction is the same as the destination register in lw, a load-use hazard occurs. It is observed that the pipeline is blocked for one cycle, and at the same time, the signals of the ex stage flush are 0. When the instruction is blt and the branch is established, the branch prediction fails. It is observed that pc flush is 0 in the next clock cycle, and the jump is achieved in the next clock cycle. The remaining register values are the same as the single-cycle simulation results.

3.3 The preliminary synthesis in Vivado

3.3.1 Single-cycle CPU

The Single-cycle CPU Schematic is shown in figure 5.
The Single-cycle CPU Utilization is shown in figure 6.

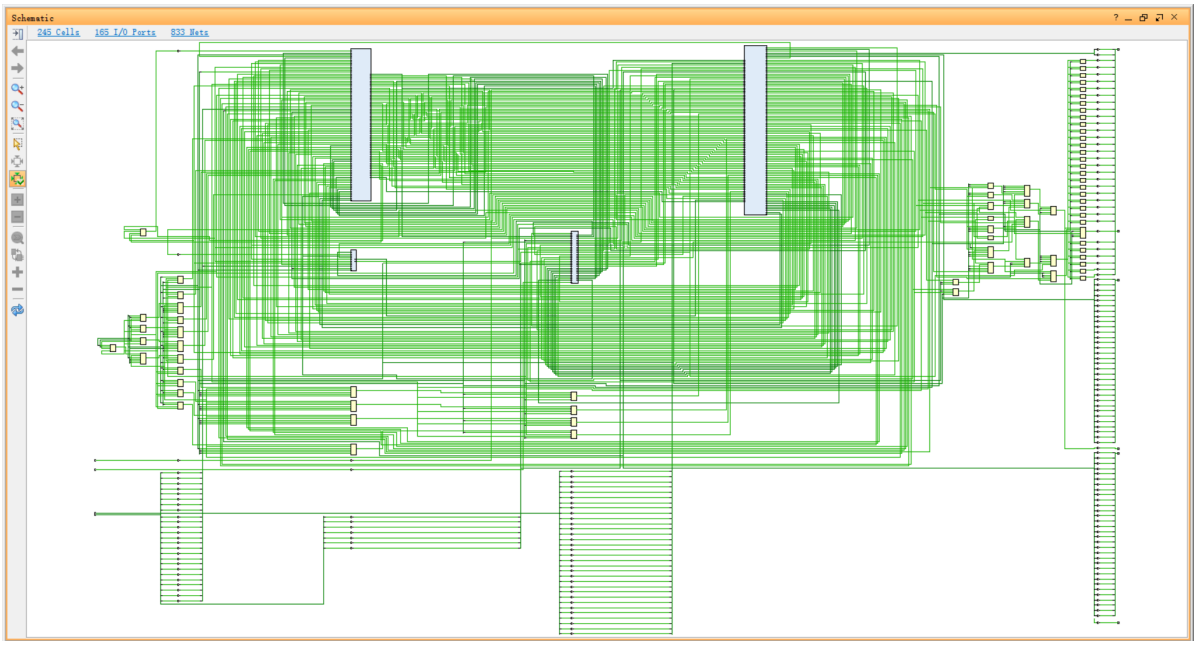

Figure 5: Single-cycle CPU Schematic

3.3.2 Multi-cycle CPU

The Multi-cycle CPU Schematic is shown in figure 7.
The Multi-cycle CPU Utilization is shown in figure 8.

4 Experimental impressions

This big assignment requires us to use Verilog to design and implement a single-cycle, five-stage pipeline CPU based on the Datapath we learned in the class. I learned a lot from this big assignment. First I reviewed the content in Chapter 4 of the class, and then wrote the control logic according to the datapath diagram on the ppt, which deepened my understanding of each control signal. At the same time, the design of a multi-stage pipeline requires that we have an overall grasp of the operations performed by each stage, especially what level of control signal and which level of data are needed when the regfile is written back. The hazard control and forwarding unit requires that we have a comprehensive consideration of each situation encountered during the execution of the instruction. Specifically, we need to determine whether the control signal is ex or mem, whether it is a MemRead or MemtoReg signal, etc. Constantly debugging, I also found a lot of understanding errors and inadequate knowledge in the beginning of the class. I am still grateful to the teacher and the assistant for this big assignment, which helped me deepen my understanding of the textbook knowledge and the theory. In connection with reality, I have a deeper understanding of the design of the CPU.

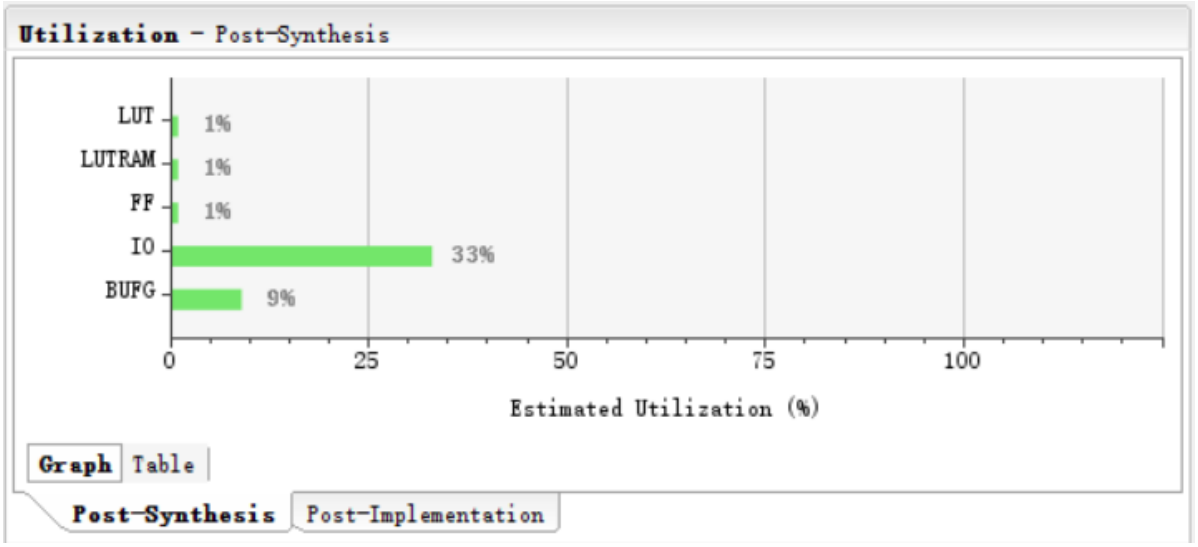

Figure 6: Single-cycle CPU Utilization

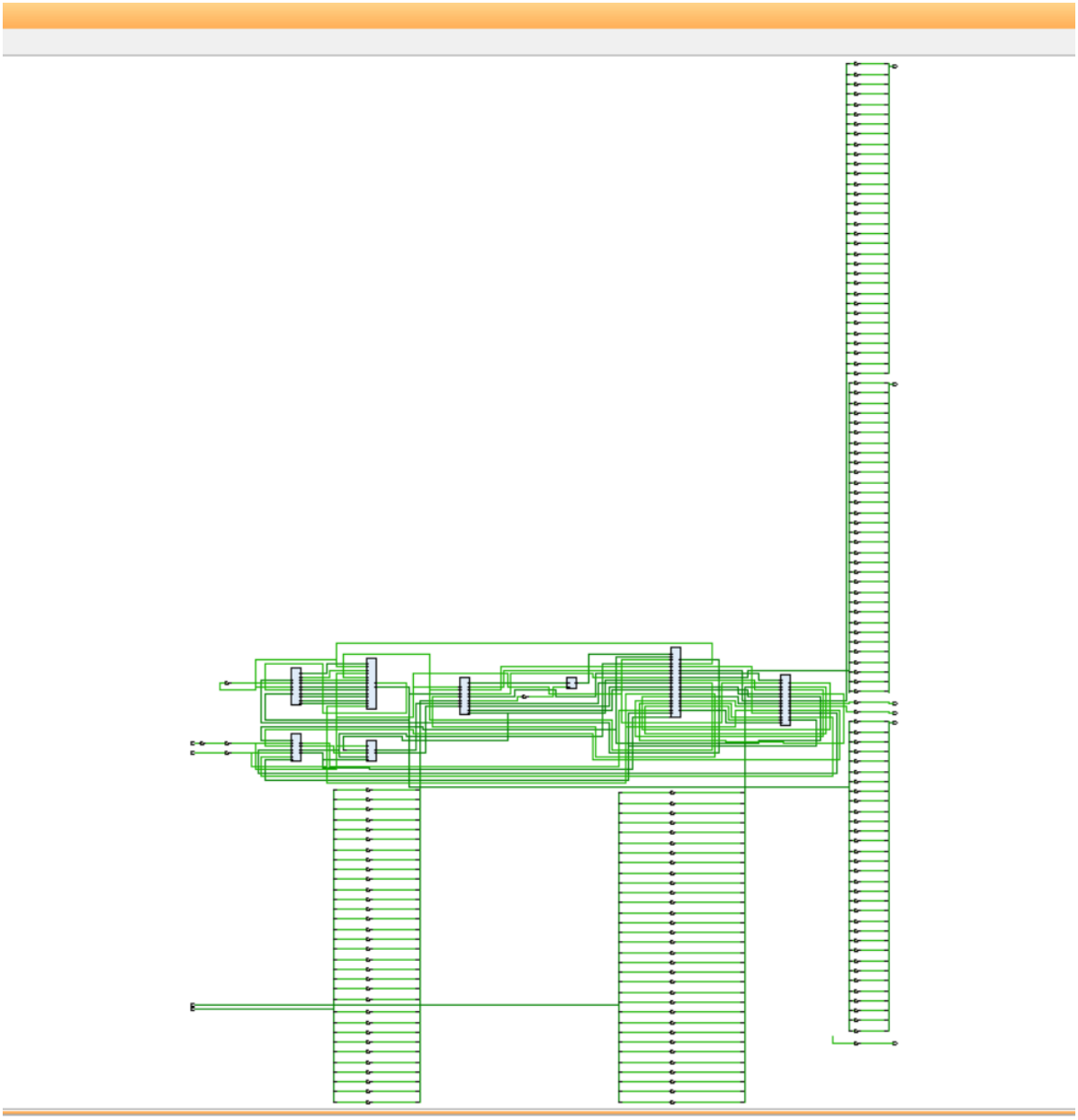

Figure 7: Multi-cycle CPU Schematic

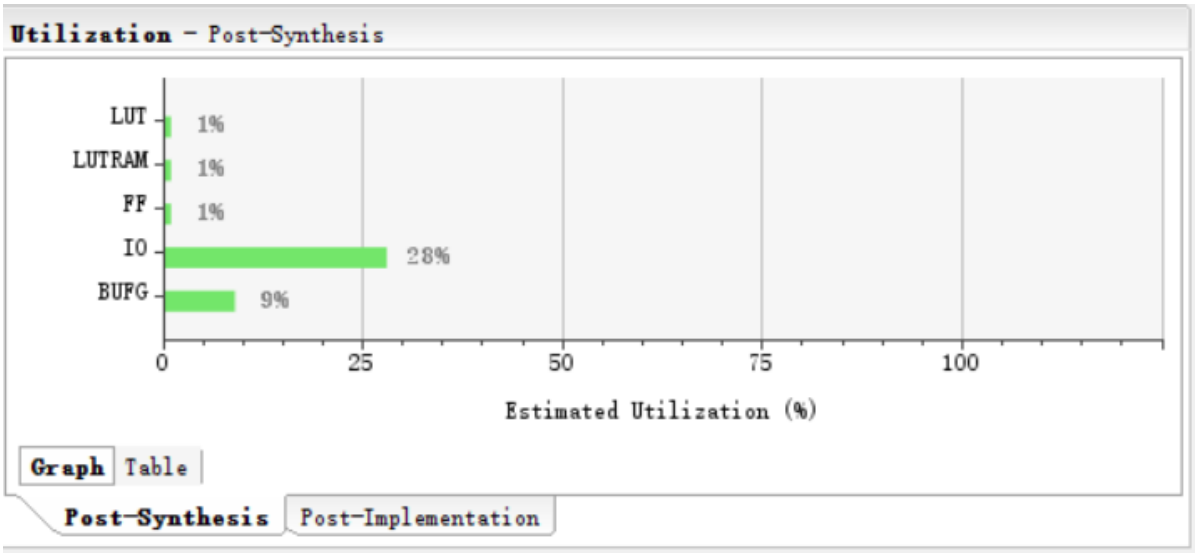

Figure 8: Multi-cycle CPU Utilization