中国科学技术大学计算机学院
《计算机组成原理实验》报告

实验题目： 单周期 CPU 设计

学生姓名： 吴毅龙

学生学号： PB19111749

完成日期： 2020/5/6

计算机实验教学中心制

2020 年 09 月

【实验题目】

单周期 CPU 设计

【实验目的】

- 理解 CPU 的结构和工作原理
- 掌握单周期 CPU 的设计和调试方法
- 熟练掌握数据通路和控制器的设计和描述方法

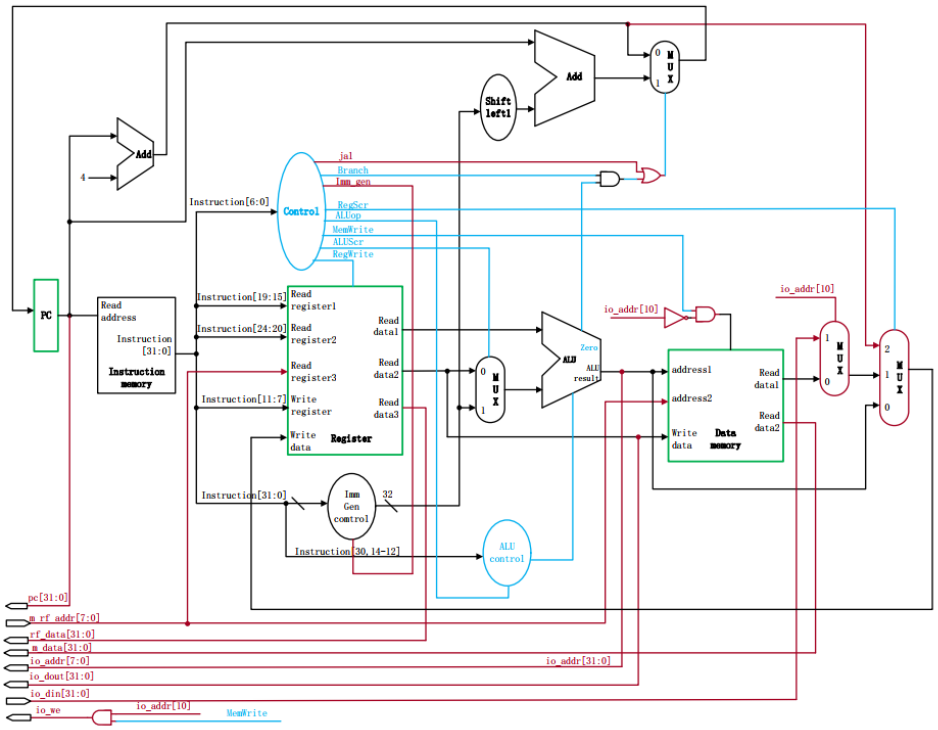
【实验环境】

Vivado 2019.2

【实验练习】

1. 设计实现单周期 RISC-V CPU

1.1 单周期 CPU 数据通路



根据文档提供的单周期 CPU 数据通路，需要实现的模块有 PC、指令存储器、寄存器、控制单元、立即数生成单元、左移运算单元、ALU、

ALU 控制单元、数据存储器、多路选择器、加法器。由于采用的是数据存储与指令存储分离的存储结构，因此此次实现的 CPU 属于哈佛结构。每一条指令执行的具体过程反映在数据通路上的过程是，pc 由程序计数器传输给指令存储器取指令，然后将指令的不同部分分别送至控制单元、寄存器、立即数生成单元。控制单元根据输入指令的类型输出不同的控制信号，控制后续单元的行为。

1.2 数据通路解释

若是 add 指令，则将寄存器取出的两个操作数通过多路选择器传输到 ALU 中，ALU 将计算结果传输至 MUX4 多路选择器，之后写入相应的寄存器中。

若是 addi 指令，则将目标寄存器和源寄存器的地址传入寄存器堆中，将立即数段传入立即数生成单元，ALU 将生成的立即数和从寄存器堆中取得的操作数加和，之后的操作同 add 指令。

若是 lw 指令，则将目标寄存器和源寄存器的地址传入寄存器堆中，将立即数段传入立即数生成单元，ALU 将生成的立即数和从寄存器堆中取得的操作数加和作为地址传入数据存储器，取数之后，将取出的数存入寄存器堆的目标寄存器中。

若是 sw 指令，则将两个源寄存器的地址传入寄存器堆中，将立即数段传入立即数生成单元，ALU 将生成的立即数和从寄存器堆中取得的操作数加和作为地址传入数据存储器，从 rs2 中取出的数作为写入数据传输到数据存储器中存入。

若是 beq 指令，则将两个源寄存器的地址传入寄存器堆中，ALU

将两个从寄存器堆中取出的数做减法，将示零信号传输给相应的逻辑判断模块。立即数段传入立即数生成单元，经过左移一位操作后与 pc 值相加，然后在逻辑判断模块结果的选择下传输给 PC。

若是 jal 指令，则将 PC+4 的值存入目标寄存器中，将立即数段传入立即数生成单元，经过左移一位操作后与 pc 值相加，然后在逻辑判断模块结果的选择下传输给 PC。

1.3 各个模块实现

具体到各个模块，PC 程序计数器负责更改 pc 的值

```
module PC(
    input clk,rst,
    input [31:0]PCN,
    output [31:0]PC
    );

    reg [31:0]PC_reg;
    initial begin
        PC_reg = 0;
    end
    always @(posedge clk, posedge rst) begin
        if(rst) PC_reg <= 0;
        else PC_reg <= PCN;
    end

    assign PC = PC_reg;

endmodule

```

指令存储器负责存储指令并输出指令，利用分布式存储器实现；

寄存器堆主要负责各个寄存器的读写操作；

```
module Registers(
    input wire clk,we,
    input wire [4:0] wa, ra0, ra1, ra2,
```

```
    input wire [31:0] wd,
    output wire [31:0] rd0, rd1, rd2
    );
    reg [31:0] regfile[0:31];

    initial begin
        regfile[0]=0;
    end

    assign  rd0 = regfile[ra0],
            rd1 = regfile[ra1],
            rd2 = regfile[ra2];

    always @ (posedge clk)begin
        if(we && wa != 0)regfile[wa] <= wd;
    end
endmodule

```

控制单元根据不同的指令发出不同的控制信号；

```
module Control (
    input [6:0] opcode,
    output reg  Branch,
                MemWrite,
                ALUSrc,
                Jal,
                RegWrite,
                Imm_gen,
    output reg [1:0] ALUop, RegScr
);

    always @(*) begin
        case(opcode)
            7'b0110011: begin   //add
                ALUop    = 2'b10;
                RegScr   = 2'b00;
                Imm_gen  = 0;
                Branch   = 0;
                MemWrite = 0;
                RegWrite = 1;
                ALUSrc   = 0;
                Jal      = 0;
            end
```

```
24.             7'b0010011: begin   //addi
25.                 ALUop    = 2'b10;
26.                 RegScr   = 2'b00;
27.                 Imm_gen  = 1;
28.                 Branch   = 0;
29.                 MemWrite = 0;
30.                 RegWrite = 1;
31.                 ALUSrc   = 1;
32.                 Jal      = 0;
33.             end
34.             7'b0000011: begin   //lw
35.                 ALUop    = 2'b00;
36.                 RegScr   = 2'b01;
37.                 Branch   = 0;
38.                 Imm_gen  = 1;
39.                 MemWrite = 0;
40.                 RegWrite = 1;
41.                 ALUSrc   = 1;
42.                 Jal      = 0;
43.             end
44.             7'b0100011: begin   //sw
45.                 ALUop    = 2'b00;
46.                 RegScr   = 2'b00;
47.                 Branch   = 0;
48.                 Imm_gen  = 1;
49.                 MemWrite = 1;
50.                 RegWrite = 0;
51.                 ALUSrc   = 1;
52.                 Jal      = 0;
53.             end
54.             7'b1100011: begin   //beq
55.                 ALUop    = 2'b01;
56.                 RegScr   = 2'b00;
57.                 Branch   = 1;
58.                 Imm_gen  = 1;
59.                 MemWrite = 0;
60.                 RegWrite = 0;
61.                 ALUSrc   = 0;
62.                 Jal      = 0;
63.             end
64.             7'b1101111: begin   //jal
65.                 ALUop    = 2'b00;
66.                 RegScr   = 2'b10;
67.                 Branch   = 1;
```

```
                Imm_gen  = 1;
                MemWrite = 0;
                RegWrite = 1;
                ALUSrc   = 1;
                Jal      = 1;
            end
            default: begin
                ALUop    = 2'b00;
                RegScr   = 2'b00;
                Branch   = 0;
                Imm_gen  = 0;
                MemWrite = 0;
                RegWrite = 0;
                ALUSrc   = 0;
                Jal      = 0;
            end
        endcase
    end
endmodule
```

立即数生成单元负责符号位扩展，立即数的生成；

```
module ImmGen (
    input [31:0]instruction,
    input Imm_gen,
    output reg [31:0]Imm
);
    always @(*) begin
    if(Imm_gen == 1)begin
        case(instruction[6:0])
            7'b0010011: begin   //addi
                Imm[11:0] = instruction[31:20];
                if(instruction[31]==1)
                     Imm[31:12] = 20'hfffff;
                else Imm[31:12] = 20'h00000;
            end
            7'b0000011: begin   //lw
                Imm[11:0] = instruction[31:20];
                if(instruction[31]==1)
                     Imm[31:12] = 20'hfffff;
                else Imm[31:12] = 20'h00000;
            end
            7'b0100011: begin   //sw
                Imm[11:5] = instruction[31:25];
```

```
                Imm[4:0]  = instruction[11:7];
                if(instruction[31]==1)
                    Imm[31:12] = 20'hfffff;
                else Imm[31:12] = 20'h00000;
            end
            7'b1100011: begin   //beq
                Imm[11]   = instruction[31];
                Imm[9:4] = instruction[30:25];
                Imm[3:0]  = instruction[11:8];
                Imm[10]   = instruction[7];
                if(instruction[31]==1)
                    Imm[31:12] = 19'h7ffff;
                else Imm[31:12] = 19'h00000;
            end
            7'b1101111: begin   //jal
                Imm[19]   = instruction[31];
                Imm[9:0] = instruction[30:21];
                Imm[10]   = instruction[20];
                Imm[18:11]  = instruction[19:12];
                if(instruction[31]==1)
                    Imm[31:20] = 11'h7ff;
                else Imm[31:20] = 11'h000;
            end
            default: begin
                Imm[31:0] = 32'd0;
            end
        endcase
        end
        else  Imm[31:0] = 32'd0;
    end
endmodule
```

ALU 为实现多组算术运算和逻辑运算的组合逻辑电路；

```
module  ALU  #(parameter WIDTH = 32)(
    input [WIDTH -1: 0]  a, b,
    input [2:0] s,
    output reg[WIDTH -1: 0] y,
    output zf
    );
    reg[WIDTH:0] temp;
    localparam ADD=3'd0,SUB=3'd1,AND=3'd2,OR=3'd3,XOR=3'd4;
    always @(*)begin
        case(s)
```

```
            ADD:temp={0,a} + {0,b};
            SUB:temp={0,a} - {0,b};
            AND:temp={0,a} & {0,b};
            OR: temp={0,a} | {0,b};
            XOR:temp={0,a} ^ {0,b};
            default:temp=0;
        endcase
        y[WIDTH -1:0]=temp[WIDTH -1:0];
    end

    assign zf = ~|y;
endmodule

```

数据存储器负责数据的存储输出，，利用分布式存储器实现；

加法器用于加法运算；

```
module Add
    #(parameter Width = 32)(
    input [Width-1:0] a,b,
    output [Width-1:0] sum
    );
    assign sum = a + b;
endmodule
```

左移运算模块用于左移运算

```
module Add
    #(parameter Width = 32)(
    input [Width-1:0] a,b,
    output [Width-1:0] sum
    );
    assign sum = a + b;
endmodule
```

还有多路选择器模块；

```
module MUX2_1
    #(parameter Width = 32)(
    input [Width-1:0] i0,i1,
    input sel,
```

```
    output reg[Width-1:0] out
    );
    always @(*)begin
        case(sel)
            1'b0:out=i0;
            default:out=i1;
        endcase
    end
endmodule
```

将这些模块根据数据通路连线，即可得到单周期 CPU

```
module CPU(
    input clk,rst,

    //IO_BUS
    output [7:0] io_addr,
    output [31:0] io_dout,
    output io_we,
    input [31:0] io_din,

    //Debug_BUS
    input [7:0] m_rf_addr,
    output [31:0] rf_data,
    output [31:0] m_data,
    output [31:0] pc
    );

    wire [31:0]PCN;
    wire [31:0]PC;
    PC PC0(
        .clk(clk),
        .rst(rst),
        .PCN(PCN),
        .PC(PC)
    );
    assign pc = PC;

    wire [31:0]Instruction;
    Instruction_Memory Instruction_Memory(
        .clk(clk),
        .a(PC[9:2]),
        .d(0),
        .dpra(0),
```

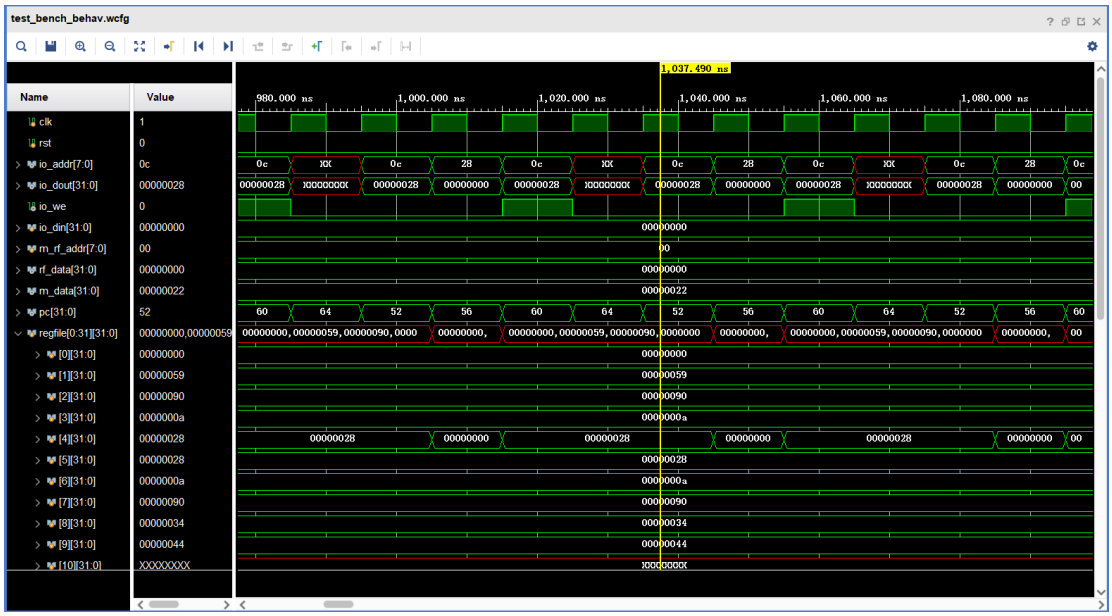
```
33.         .we(0),
34.         .spo(Instruction),
35.         .dpo()
36.     );
37.     wire Jal, Branch, Imm_gen, MemWrite, ALUScr, RegWrite;
38.     wire [1:0]ALUop, RegScr;
39.     wire [31:0]Read_data0, Read_data1;
40.     wire [31:0]Write_data;
41.     Registers Registers(
42.         .clk(clk),
43.         .we(RegWrite),
44.         .wd(Write_data),
45.         .ra0(Instruction[19:15]),
46.         .ra1(Instruction[24:20]),
47.         .ra2(m_rf_addr[4:0]),
48.         .wa(Instruction[11:7]),
49.         .rd0(Read_data0),
50.         .rd1(Read_data1),
51.         .rd2(rf_data)
52.     );
53.
54.     Control Control(
55.         .opcode(Instruction[6:0]),
56.         .Branch(Branch),
57.         .MemtoReg(MemtoReg),
58.         .MemWrite(MemWrite),
59.         .ALUSrc(ALUSrc),
60.         .Imm_gen(Imm_gen),
61.         .RegWrite(RegWrite),
62.         .RegScr(RegScr),
63.         .ALUop(ALUop),
64.         .Jal(Jal)
65.     );
66.
67.     wire [31:0]Imm;
68.     ImmGen ImmGen(
69.         .instruction(Instruction),
70.         .Imm_gen(Imm_gen),
71.         .Imm(Imm)
72.     );
73.
74.     wire [31:0]ALUin2;
75.     MUX2_1 #(32)Mux1(
76.         .i0(Read_data0),
```

```
        .i1(Imm),
        .sel(ALUSrc),
        .out(ALUin2)
    );

    wire [2:0] ALUcontrol;
    ALUcontrol ALUcontroler(
        .ALUop(ALUop),
        .ALU_Final(ALUcontrol)
    );

    wire [31:0] ALUresult;
    wire zero;
    ALU #(32)ALU(
        .a(Read_data0),
        .b(ALUin2),
        .s(ALUcontrol),
        .y(ALUresult),
        .zf(zero)
    );

    wire [31:0]io_addr_temp;
    wire [7:0] DM_a_in;
     wire DM_E;
     wire [31:0]ReadData;
     assign io_addr_temp = ALUresult;
     assign DM_E = ~io_addr_temp[10] & MemWrite;
     assign DM_a_in = ALUresult[9:0] >> 2;
     Data_Memory Data_Memory(
         .a(DM_a_in),
         .d(Read_data1),
         .dpra(m_rf_addr),
         .clk(clk),
         .we(DM_E),
         .spo(ReadData),
         .dpo(m_data)
     );

     wire [31:0]MUX3out;
     MUX2_1 #(32)MUX3(
         .i0(ReadData),
         .i1(io_din),
         .sel(io_addr_temp[10]),
         .out(MUX3out)
```

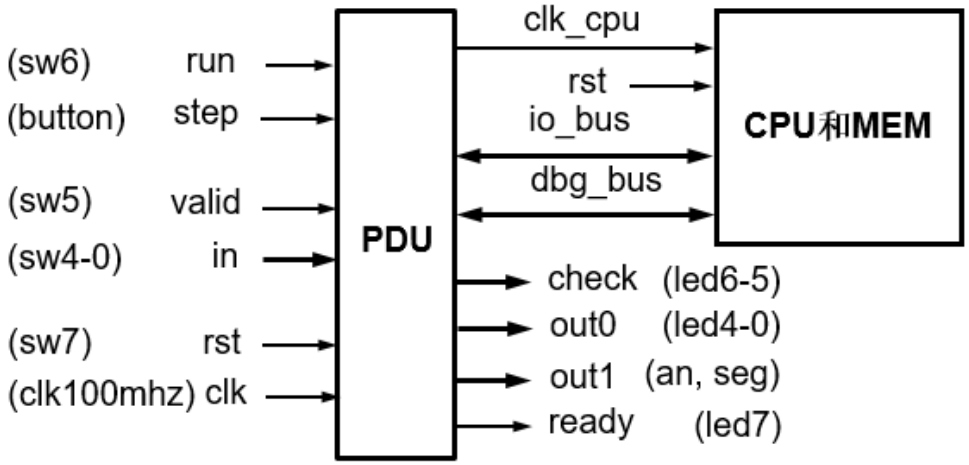
```
    );

    wire [31:0] PCadd4;
    MUX3_1 #(32)MUX4(
        .i0(ALUresult),
        .i1(MUX3out),
        .i2(PCadd4),
        .sel(RegScr),
        .out(Write_data)
    );

    Add #(32)Add1(PC, 32'd4, PCadd4);

    wire [31:0] Br_Add2;
    Shiftleft Shfitleft1(Imm, Br_Add2);

    wire [31:0] sum_b;
    Add #(32)Add2(PC, Br_Add2, sum_b);

    wire Branch1, Branch2;
    assign Branch1 = Branch & zero;
    assign Branch2 = Branch1 | Jal;
    MUX2_1 #(32)Mux2(
        .i0(PCadd4),
        .i1(sum_b),
        .sel(Branch2),
        .out(PCN)
    );
    assign io_we = io_addr_temp[10] & MemWrite;
    assign io_addr = io_addr_temp[7:0];
    assign io_dout = Read_data1;
endmodule
```

1.4 仿真结果

导入 coe 文件对指令存储器进行初始化，得到仿真结果



2. 处理器调试单元



控制 CPU 的运行方式：run = 1 连续运行，0 单步运行

管理外设（开关 sw、指示灯 led、数码管 an & seg），显示运行结果和数据通路状态

2.1 CPU 运行方式

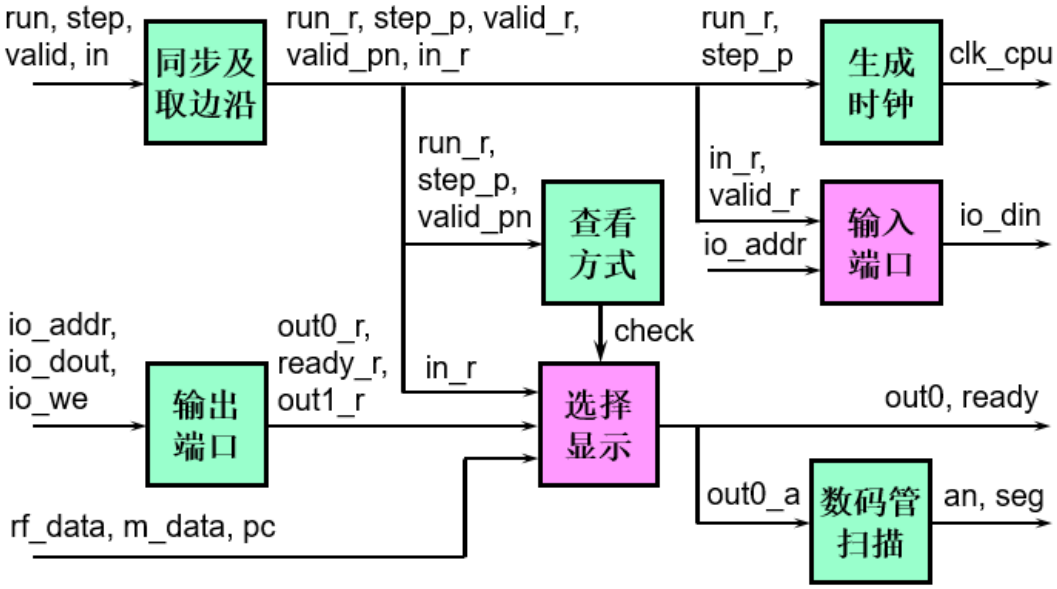
run = 1：连续运行

PDU 向 CPU 输出连续时钟信号 clk_cpu
CPU 通过 I/O_BUS 访问外设
输入端口：in，valid
输出端口：out0, out1, ready

run = 0：单步运行（每次执行一条指令）

每按动 step 一次，PDU 产生一个周期的 clk_cpu
执行外设输入指令前，应先设置好 valid 或 in 后，再按动 step
执行任何指令后，led 和数码管(an, seg)显示当前程序运行结果
随后可以通过改变 valid 和 in 查看寄存器堆、存储器和 PC 的内容

2.2 PDU 逻辑结构图



2.3 PDU 模块（文档提供）

```
1. module  pdu_1cycle(
2.    input clk,
3.    input rst,
4.
5.    input run,
6.    input step,
7.    output clk_cpu,
8.    input valid,
9.    input [4:0] in,
10.
11.   output [1:0] check,
12.   output [4:0] out0,
13.   output [2:0] an,
14.   output [3:0] seg,
15.   output ready,
16.
17.   //IO_BUS
18.   input [7:0] io_addr,
19.   input [31:0] io_dout,
```

```
  input io_we,
  output [31:0] io_din,

  //Debug_BUS
  output [7:0] m_rf_addr,
  input [31:0] rf_data,
  input [31:0] m_data,
  input [31:0] pc
);

reg [4:0] in_r;
reg run_r, step_r, step_2r, valid_r, valid_2r;
wire step_p, valid_pn;

reg clk_cpu_r;
reg [4:0] out0_r;
reg [31:0] out1_r;
reg ready_r;
reg [19:0] cnt;
reg [1:0] check_r;
reg [31:0] io_din_a;  reg ready_a;
reg [4:0] out0_a;
reg [31:0] out1_a;
reg [3:0] seg_a;

assign clk_cpu = clk_cpu_r;
assign io_din = io_din_a;
assign check = check_r;
assign out0 = out0_a;
assign ready = ready_a;
assign seg = seg_a;
assign an = cnt[19:17];
assign step_p = step_r & ~step_2r;
assign valid_pn = valid_r ^ valid_2r;
 assign m_rf_addr = {{3{1'b0}}, in_r};

always @(posedge clk) begin
  run_r <= run;
  step_r <= step;
  step_2r <= step_r;
  valid_r <= valid;
  valid_2r <= valid_r;
  in_r <= in;
end
```

```
64.
65.always @(posedge clk, posedge rst) begin
66.  if(rst)
67.    clk_cpu_r <= 0;
68.  else if (run_r)
69.    clk_cpu_r <= ~clk_cpu_r;
70.  else
71.    clk_cpu_r <= step_p;
72.end
73.
74.always @* begin
75.  case (io_addr)
76.    8'h0c: io_din_a = {{27{1'b0}}, in_r};
77.    8'h10: io_din_a = {{31{1'b0}}, valid_r};
78.    default: io_din_a = 32'h0000_0000;
79.  endcase
80.end
81.
82.always @(posedge clk, posedge rst) begin
83.if (rst) begin
84.  out0_r <= 5'h1f;
85.  out1_r <= 32'h1234_5678;
86.  ready_r <= 1'b1;
87.end
88.else if (io_we)
89.  case (io_addr)
90.    8'h00: out0_r <= io_dout[4:0];
91.    8'h04: ready_r <= io_dout[0];
92.    8'h08: out1_r <= io_dout;
93.    default: ;
94.  endcase
95.end
96.
97.always @(posedge clk, posedge rst) begin
98.if(rst)
99.    check_r <= 2'b00;
100.  else if(run_r)
101.    check_r <= 2'b00;
102.  else if (step_p)
103.    check_r <= 2'b00;
104.  else if (valid_pn)
105.    check_r <= check - 2'b01;
106.end
107.
```

```
always @* begin
  ready_a = 1'b0;
  case (check_r)
    2'b00: begin
      out0_a = out0_r;
      out1_a = out1_r;
      ready_a = ready_r;
    end
    2'b01: begin
      out0_a = in_r;
      out1_a = rf_data;
    end
    2'b10: begin
      out0_a = in_r;
      out1_a = m_data;
    end
    2'b11: begin
      out0_a = 5'b00000;
      out1_a = pc;
    end
  endcase
end

always @(posedge clk, posedge rst) begin
  if (rst) cnt <= 20'h0_0000;
  else cnt <= cnt + 20'h0_0001;
end

always @* begin
  case (an)
    3'd0: seg_a = out1_a[3:0];
    3'd1: seg_a = out1_a[7:4];
    3'd2: seg_a = out1_a[11:8];
    3'd3: seg_a = out1_a[15:12];
    3'd4: seg_a = out1_a[19:16];
    3'd5: seg_a = out1_a[23:20];
    3'd6: seg_a = out1_a[27:24];
    3'd7: seg_a = out1_a[31:28];
    default: ;
  endcase
end

endmodule
```

3 CPU + PDU

3.1 模块链接实现

将两个模块连线连接起来

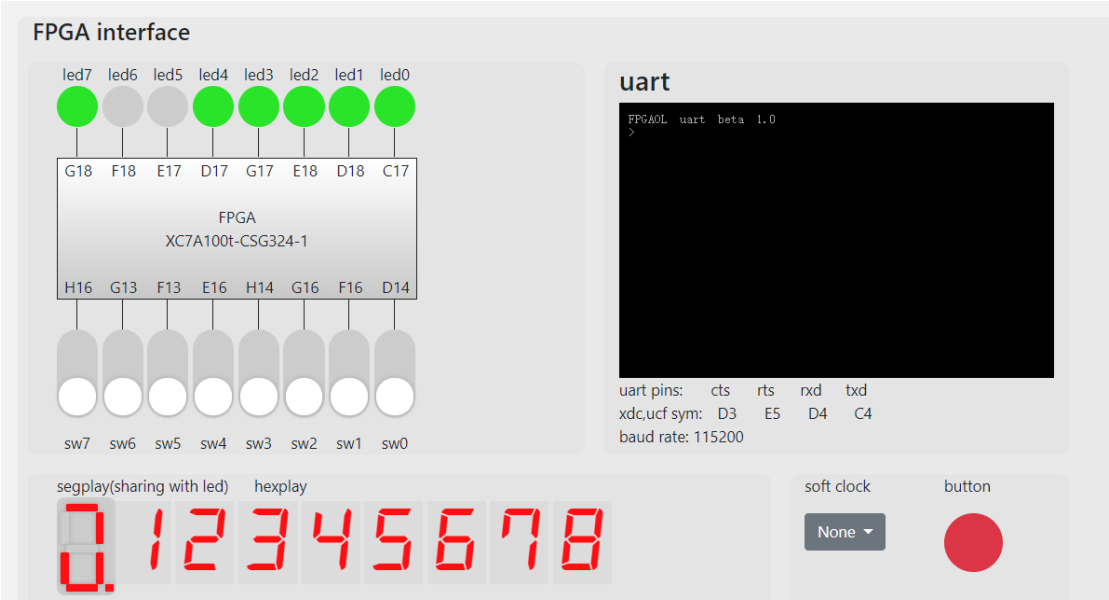
```
1. module Lab4(
2.     input clk,rst,
3.     input run,step,
4.     input valid,
5.     input  [4:0] in,
6.     output [1:0] check,
7.     output [4:0] out0,
8.     output [2:0] an,
9.     output [3:0] seg,
10.    output ready
11.    );
12.
13.    //IO_BUS
14.    wire [7:0] io_addr;
15.    wire [31:0] io_dout;
16.    wire io_we;
17.    wire [31:0] io_din;
18.
19.    //Debug_BUS
20.    wire [7:0] m_rf_addr;
21.    wire [31:0] rf_data;
22.    wire [31:0] m_data;
23.    wire [31:0] pc;
24.        .clk(clk),
25.        .rst(rst),
26.
27.        .io_addr(io_addr),
28.        .io_dout(io_dout),
29.        .io_we(io_we),
30.        .io_din(io_din),
31.
32.        .m_rf_addr(m_rf_addr),
33.        .rf_data(rf_data),
34.        .m_data(m_data),
35.        .pc(pc)
36.    );
37.
```

```
    pdu_1cycle debug(
        .clk(clk),
        .rst(rst),

        .run(run),
        .step(step),
        .clk_cpu(clk_cpu),

        .valid(valid),
        .in(in),

        .check(check),
        .out0(out0),
        .an(an),
        .seg(seg),
        .ready(ready),

        .io_addr(io_addr),
        .io_dout(io_dout),
        .io_we(io_we),
        .io_din(io_din),

        .m_rf_addr(m_rf_addr),
        .rf_data(rf_data),
        .m_data(m_data),
        .pc(pc)
    );

endmodule
```

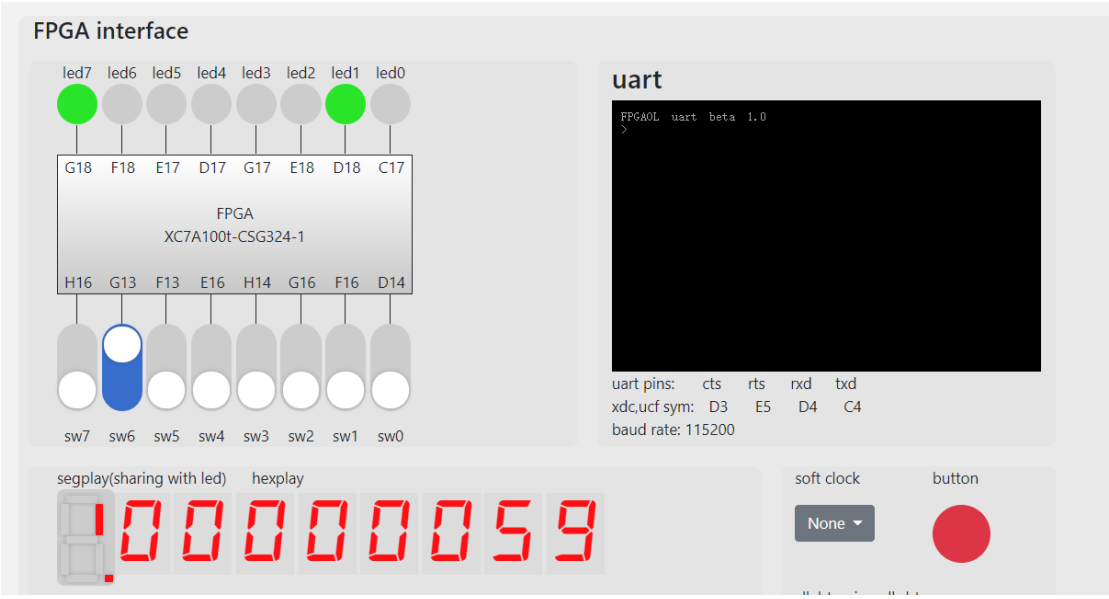
sw				button	led			an/seg	说明
7	6	5	4~0		7	6~5	4~0		
rst	run	valid	in	step	ready	check	out0	out1	
↑	-	-	-	-	1	00	0x1f	0x12…8	复位
x	1	valid	in	-	ready	00	out0	out1	连续运行
	0	valid	in	↑	ready	00	out0	out1	单步运行
		↑↓	addr_rf	x	0	01	addr_rf	data_rf	查看寄存器堆
			addr_m		0	10	addr_m	data_m	查看存储器
			-		0	11	0	PC	查看PC

3.2 平台测试

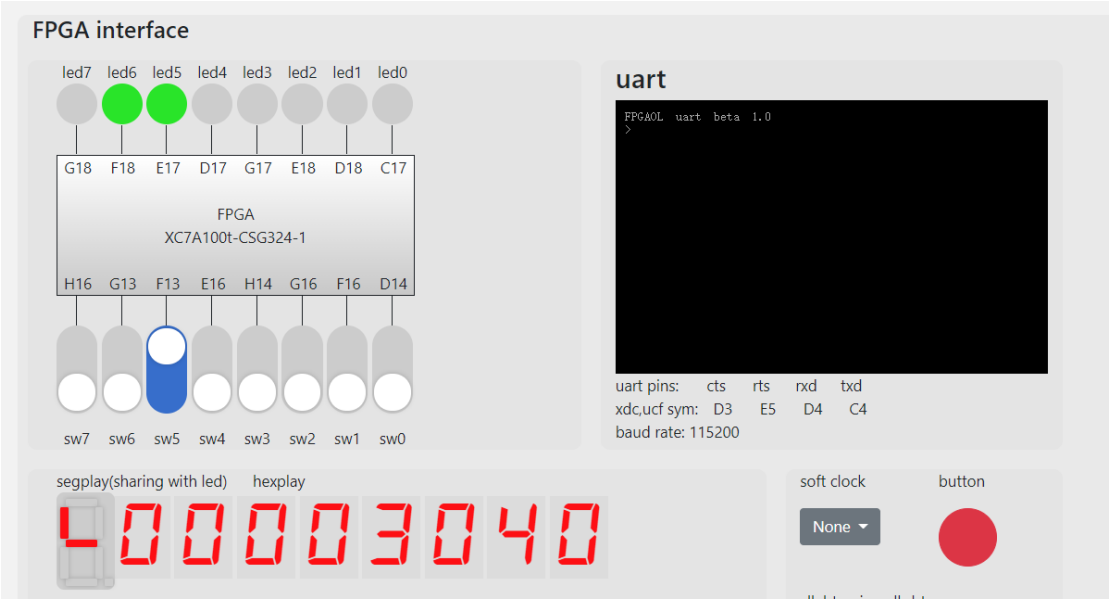
初始状态



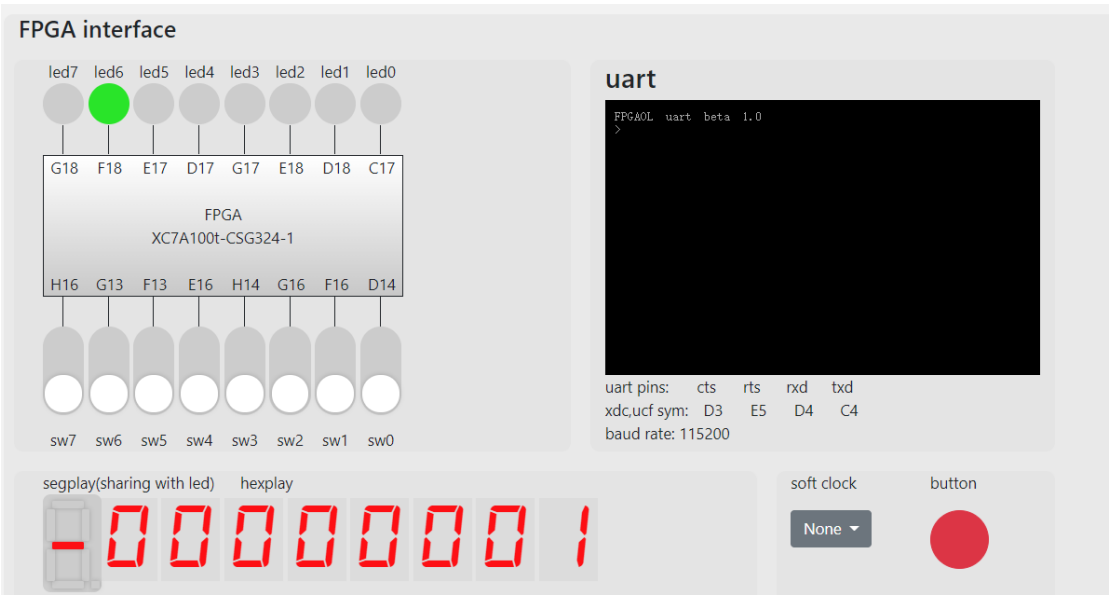
连续运行



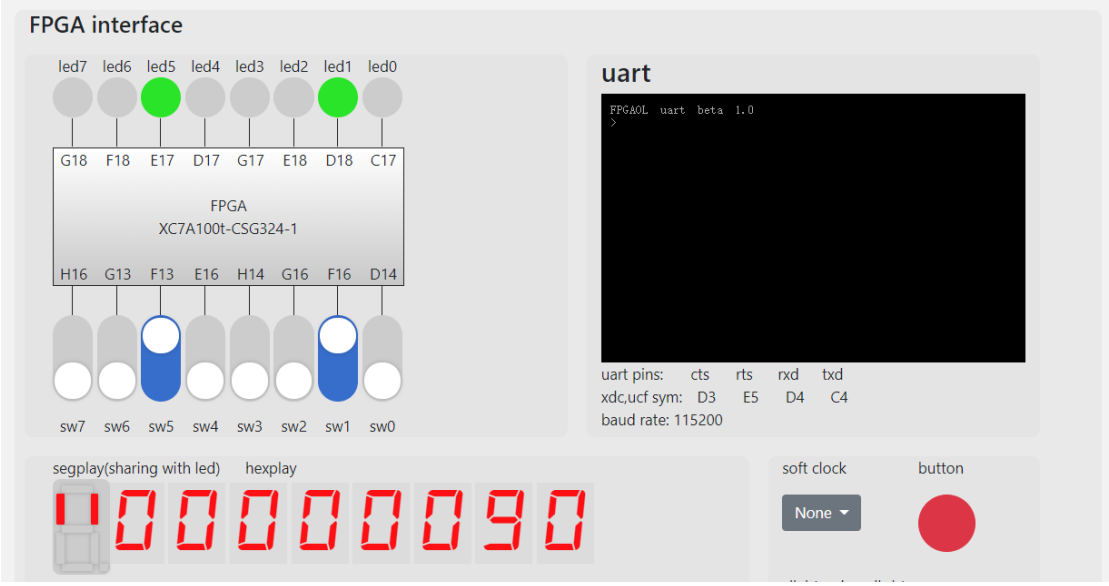
查看 PC


FPGA interface
led7 led6 led5 led4 led3 led2 led1 led0
G18 F18 E17 D17 G17 E18 D18 C17
FPGA
XC7A100t-CSG324-1
H16 G13 F13 E16 H14 G16 F16 D14
sw7 sw6 sw5 sw4 sw3 sw2 sw1 sw0
uart
FPGAOL uart beta 1.0
>
uart pins: cts rts rxd txd
xdc,ucf sym: D3 E5 D4 C4
baud rate: 115200
segplay(sharing with led)
hexplay
00003040
soft clock
None
button

查看存储器


FPGA interface
led7 led6 led5 led4 led3 led2 led1 led0
G18 F18 E17 D17 G17 E18 D18 C17
FPGA
XC7A100t-CSG324-1
H16 G13 F13 E16 H14 G16 F16 D14
sw7 sw6 sw5 sw4 sw3 sw2 sw1 sw0
uart
FPGAOL uart beta 1.0
>
uart pins: cts rts rxd txd
xdc,ucf sym: D3 E5 D4 C4
baud rate: 115200
segplay(sharing with led)
hexplay
00000001
soft clock
None
button

查看寄存器



【总结与思考】

通过本次实验，我进行了单周期 CPU 的设计，对于 RSIC-V 指令集以及计算机组成原理有了更深一层的理解，对于不同类型指令的执行过程有了建立在数据通路层面的认识。实验题目的难度层层递进，有基础操作的考核，也有所学知识的综合，难易结合，既有复习又有思考，让所学在实践中得以运用，加深了我对计算机组成原理知识的理解。希望今后实验可以保持本次实验中详细实验指导描述的优点，辅助完成每项试验内容。