**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

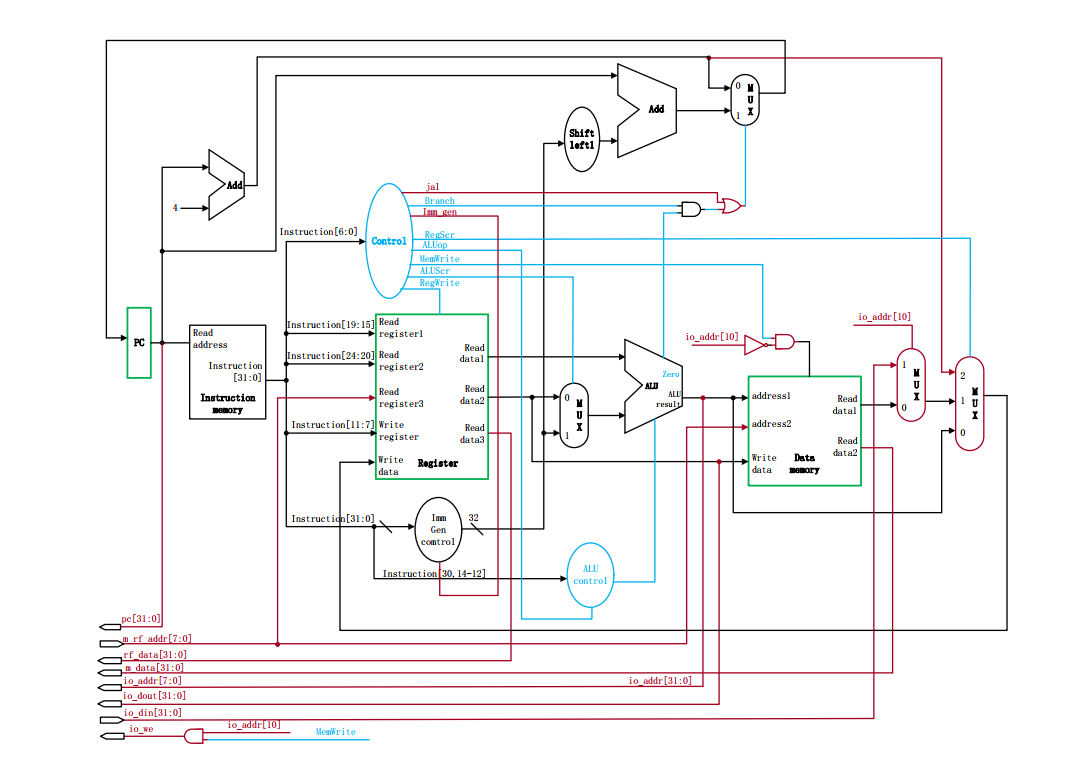
【实验环境】

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

1. module PC(
2. input clk,rst,
3. input [31:0]PCN,
4. output [31:0]PC
5. );
6. reg [31:0]PC\_reg;
7. initial begin
8. PC\_reg = 0;
9. end
10. always @(posedge clk, posedge rst) begin
11. if(rst) PC\_reg <= 0;
12. else PC\_reg <= PCN;
13. end
14. assign PC = PC\_reg;
16. 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；

1. module Registers(
2. input wire clk,we,
3. input wire [4:0] wa, ra0, ra1, ra2,
4. input wire [31:0] wd,
5. output wire [31:0] rd0, rd1, rd2
6. );
7. reg [31:0] regfile[0:31];
8. initial begin
9. regfile[0]=0;
10. end
12. assign  rd0 = regfile[ra0],
13. rd1 = regfile[ra1],
14. rd2 = regfile[ra2];
15. always @ (posedge clk)begin
16. if(we && wa != 0)regfile[wa] <= wd;
17. end
18. endmodule

控制单元根据不同的指令发出不同的控制信号；

1. module Control (
2. input [6:0] opcode,
3. output reg  Branch,
4. MemWrite,
5. ALUSrc,
6. Jal,
7. RegWrite,
8. Imm\_gen,
9. output reg [1:0] ALUop, RegScr
10. );
11. always @(\*) begin
12. case(opcode)
13. 7'b0110011: begin   //add
14. ALUop    = 2'b10;
15. RegScr   = 2'b00;
16. Imm\_gen  = 0;
17. Branch   = 0;
18. MemWrite = 0;
19. RegWrite = 1;
20. ALUSrc   = 0;
21. Jal      = 0;
22. end
23. 7'b0010011: begin   //addi
24. ALUop    = 2'b10;
25. RegScr   = 2'b00;
26. Imm\_gen  = 1;
27. Branch   = 0;
28. MemWrite = 0;
29. RegWrite = 1;
30. ALUSrc   = 1;
31. Jal      = 0;
32. end
33. 7'b0000011: begin   //lw
34. ALUop    = 2'b00;
35. RegScr   = 2'b01;
36. Branch   = 0;
37. Imm\_gen  = 1;
38. MemWrite = 0;
39. RegWrite = 1;
40. ALUSrc   = 1;
41. Jal      = 0;
42. end
43. 7'b0100011: begin   //sw
44. ALUop    = 2'b00;
45. RegScr   = 2'b00;
46. Branch   = 0;
47. Imm\_gen  = 1;
48. MemWrite = 1;
49. RegWrite = 0;
50. ALUSrc   = 1;
51. Jal      = 0;
52. end
53. 7'b1100011: begin   //beq
54. ALUop    = 2'b01;
55. RegScr   = 2'b00;
56. Branch   = 1;
57. Imm\_gen  = 1;
58. MemWrite = 0;
59. RegWrite = 0;
60. ALUSrc   = 0;
61. Jal      = 0;
62. end
63. 7'b1101111: begin   //jal
64. ALUop    = 2'b00;
65. RegScr   = 2'b10;
66. Branch   = 1;
67. Imm\_gen  = 1;
68. MemWrite = 0;
69. RegWrite = 1;
70. ALUSrc   = 1;
71. Jal      = 1;
72. end
73. default: begin
74. ALUop    = 2'b00;
75. RegScr   = 2'b00;
76. Branch   = 0;
77. Imm\_gen  = 0;
78. MemWrite = 0;
79. RegWrite = 0;
80. ALUSrc   = 0;
81. Jal      = 0;
82. end
83. endcase
84. end
85. endmodule

立即数生成单元负责符号位扩展，立即数的生成；

1. module ImmGen (
2. input [31:0]instruction,
3. input Imm\_gen,
4. output reg [31:0]Imm
5. );
6. always @(\*) begin
7. if(Imm\_gen == 1)begin
8. case(instruction[6:0])
9. 7'b0010011: begin   //addi
10. Imm[11:0] = instruction[31:20];
11. if(instruction[31]==1)
12. Imm[31:12] = 20'hfffff;
13. else Imm[31:12] = 20'h00000;
14. end
15. 7'b0000011: begin   //lw
16. Imm[11:0] = instruction[31:20];
17. if(instruction[31]==1)
18. Imm[31:12] = 20'hfffff;
19. else Imm[31:12] = 20'h00000;
20. end
21. 7'b0100011: begin   //sw
22. Imm[11:5] = instruction[31:25];
23. Imm[4:0]  = instruction[11:7];
24. if(instruction[31]==1)
25. Imm[31:12] = 20'hfffff;
26. else Imm[31:12] = 20'h00000;
27. end
28. 7'b1100011: begin   //beq
29. Imm[11]   = instruction[31];
30. Imm[9:4] = instruction[30:25];
31. Imm[3:0]  = instruction[11:8];
32. Imm[10]   = instruction[7];
33. if(instruction[31]==1)
34. Imm[31:12] = 19'h7ffff;
35. else Imm[31:12] = 19'h00000;
36. end
37. 7'b1101111: begin   //jal
38. Imm[19]   = instruction[31];
39. Imm[9:0] = instruction[30:21];
40. Imm[10]   = instruction[20];
41. Imm[18:11]  = instruction[19:12];
42. if(instruction[31]==1)
43. Imm[31:20] = 11'h7ff;
44. else Imm[31:20] = 11'h000;
45. end
46. default: begin
47. Imm[31:0] = 32'd0;
48. end
49. endcase
50. end
51. else  Imm[31:0] = 32'd0;
52. end
53. endmodule

ALU为实现多组算术运算和逻辑运算的组合逻辑电路；

1. module  ALU #(parameter WIDTH = 32)(
2. input [WIDTH -1: 0]  a, b,
3. input [2:0] s,
4. output reg[WIDTH -1: 0] y,
5. output zf
6. );
7. reg[WIDTH:0] temp;
8. localparam ADD=3'd0,SUB=3'd1,AND=3'd2,OR=3'd3,XOR=3'd4;
9. always @(\*)begin
10. case(s)
11. ADD:temp={0,a} + {0,b};
12. SUB:temp={0,a} - {0,b};
13. AND:temp={0,a} & {0,b};
14. OR: temp={0,a} | {0,b};
15. XOR:temp={0,a} ^ {0,b};
16. default:temp=0;
17. endcase
18. y[WIDTH -1:0]=temp[WIDTH -1:0];
19. end
20. assign zf = ~|y;
21. 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；

1. module Add
2. #(parameter Width = 32)(
3. input [Width-1:0] a,b,
4. output [Width-1:0] sum
5. );
6. assign sum = a + b;
7. endmodule

左移运算模块用于左移运算

1. module Add
2. #(parameter Width = 32)(
3. input [Width-1:0] a,b,
4. output [Width-1:0] sum
5. );
6. assign sum = a + b;
7. endmodule

还有多路选择器模块；

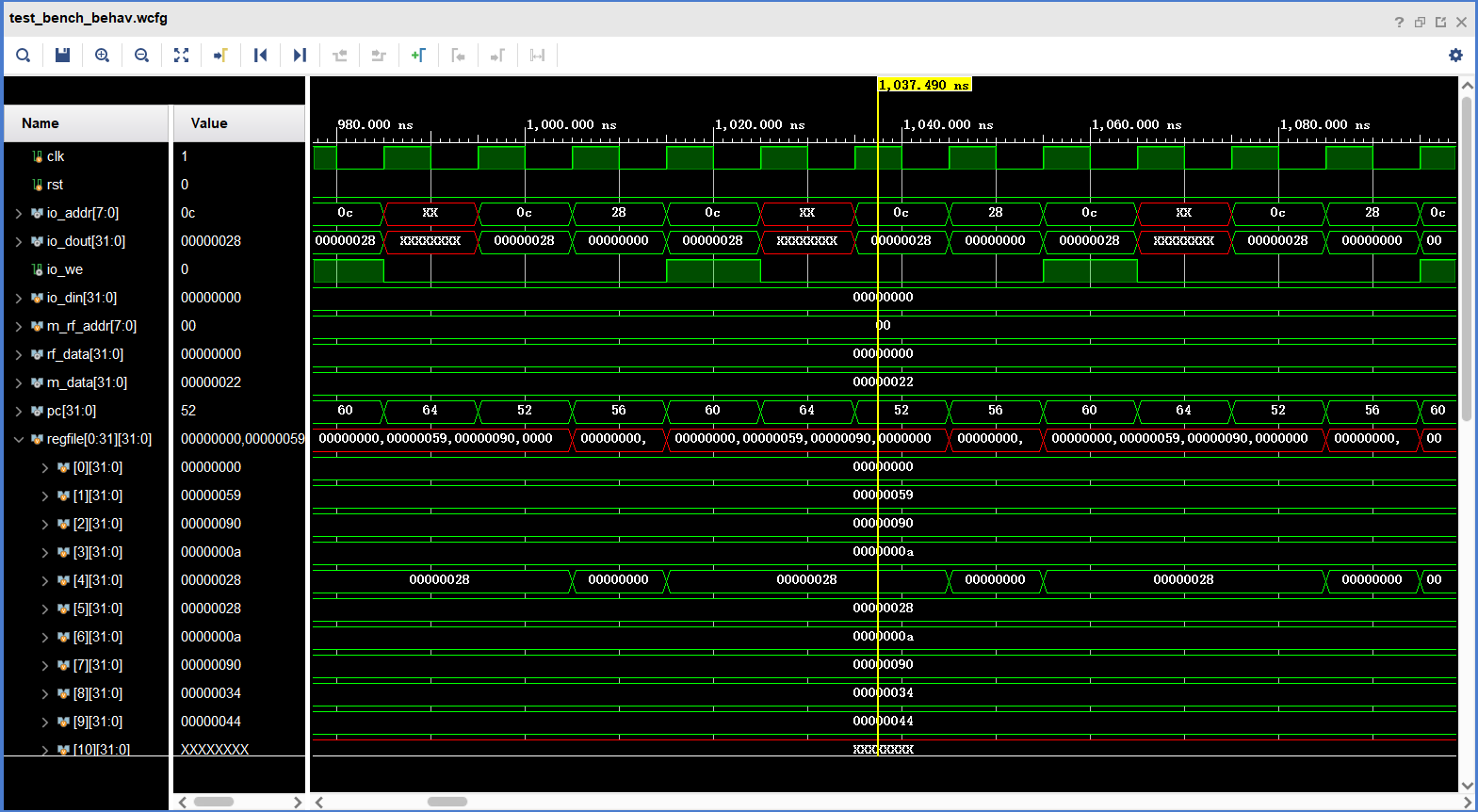
1. module MUX2\_1
2. #(parameter Width = 32)(
3. input [Width-1:0] i0,i1,
4. input sel,
5. output reg[Width-1:0] out
6. );
7. always @(\*)begin
8. case(sel)
9. 1'b0:out=i0;
10. default:out=i1;
11. endcase
12. end
13. endmodule

将这些模块根据数据通路连线，即可得到单周期CPU

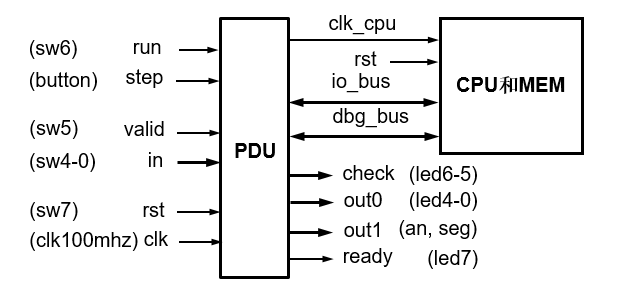
1. module CPU(
2. input clk,rst,
4. //IO\_BUS
5. output [7:0] io\_addr,
6. output [31:0] io\_dout,
7. output io\_we,
8. input [31:0] io\_din,
9. //Debug\_BUS
10. input [7:0] m\_rf\_addr,
11. output [31:0] rf\_data,
12. output [31:0] m\_data,
13. output [31:0] pc
14. );
15. wire [31:0]PCN;
16. wire [31:0]PC;
17. PC PC0(
18. .clk(clk),
19. .rst(rst),
20. .PCN(PCN),
21. .PC(PC)
22. );
23. assign pc = PC;
24. wire [31:0]Instruction;
25. Instruction\_Memory Instruction\_Memory(
26. .clk(clk),
27. .a(PC[9:2]),
28. .d(0),
29. .dpra(0),
30. .we(0),
31. .spo(Instruction),
32. .dpo()
33. );
34. wire Jal, Branch, Imm\_gen, MemWrite, ALUScr, RegWrite;
35. wire [1:0]ALUop, RegScr;
36. wire [31:0]Read\_data0, Read\_data1;
37. wire [31:0]Write\_data;
38. Registers Registers(
39. .clk(clk),
40. .we(RegWrite),
41. .wd(Write\_data),
42. .ra0(Instruction[19:15]),
43. .ra1(Instruction[24:20]),
44. .ra2(m\_rf\_addr[4:0]),
45. .wa(Instruction[11:7]),
46. .rd0(Read\_data0),
47. .rd1(Read\_data1),
48. .rd2(rf\_data)
49. );
51. Control Control(
52. .opcode(Instruction[6:0]),
53. .Branch(Branch),
54. .MemtoReg(MemtoReg),
55. .MemWrite(MemWrite),
56. .ALUSrc(ALUSrc),
57. .Imm\_gen(Imm\_gen),
58. .RegWrite(RegWrite),
59. .RegScr(RegScr),
60. .ALUop(ALUop),
61. .Jal(Jal)
62. );
64. wire [31:0]Imm;
65. ImmGen ImmGen(
66. .instruction(Instruction),
67. .Imm\_gen(Imm\_gen),
68. .Imm(Imm)
69. );
71. wire [31:0]ALUin2;
72. MUX2\_1 #(32)Mux1(
73. .i0(Read\_data0),
74. .i1(Imm),
75. .sel(ALUSrc),
76. .out(ALUin2)
77. );
79. wire [2:0] ALUcontrol;
80. ALUcontrol ALUcontroler(
81. .ALUop(ALUop),
82. .ALU\_Final(ALUcontrol)
83. );
85. wire [31:0] ALUresult;
86. wire zero;
87. ALU #(32)ALU(
88. .a(Read\_data0),
89. .b(ALUin2),
90. .s(ALUcontrol),
91. .y(ALUresult),
92. .zf(zero)
93. );
95. wire [31:0]io\_addr\_temp;
96. wire [7:0] DM\_a\_in;
97. wire DM\_E;
98. wire [31:0]ReadData;
99. assign io\_addr\_temp = ALUresult;
100. assign DM\_E = ~io\_addr\_temp[10] & MemWrite;
101. assign DM\_a\_in = ALUresult[9:0] >> 2;
102. Data\_Memory Data\_Memory(
103. .a(DM\_a\_in),
104. .d(Read\_data1),
105. .dpra(m\_rf\_addr),
106. .clk(clk),
107. .we(DM\_E),
108. .spo(ReadData),
109. .dpo(m\_data)
110. );
112. wire [31:0]MUX3out;
113. MUX2\_1 #(32)MUX3(
114. .i0(ReadData),
115. .i1(io\_din),
116. .sel(io\_addr\_temp[10]),
117. .out(MUX3out)
118. );
120. wire [31:0] PCadd4;
121. MUX3\_1 #(32)MUX4(
122. .i0(ALUresult),
123. .i1(MUX3out),
124. .i2(PCadd4),
125. .sel(RegScr),
126. .out(Write\_data)
127. );
129. Add #(32)Add1(PC, 32'd4, PCadd4);
131. wire [31:0] Br\_Add2;
132. Shiftleft Shfitleft1(Imm, Br\_Add2);
134. wire [31:0] sum\_b;
135. Add #(32)Add2(PC, Br\_Add2, sum\_b);
137. wire Branch1, Branch2;
138. assign Branch1 = Branch & zero;
139. assign Branch2 = Branch1 | Jal;
140. MUX2\_1 #(32)Mux2(
141. .i0(PCadd4),
142. .i1(sum\_b),
143. .sel(Branch2),
144. .out(PCN)
145. );
146. assign io\_we = io\_addr\_temp[10] & MemWrite;
147. assign io\_addr = io\_addr\_temp[7:0];
148. assign io\_dout = Read\_data1;
149. 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）

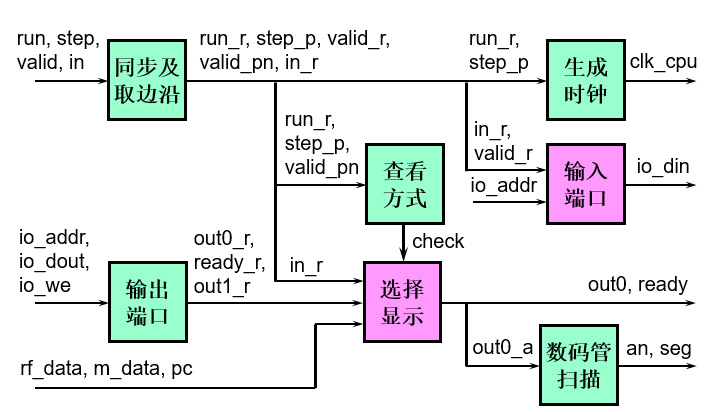
*每按动step一次，PDU产生一个周期的clk\_cpu*

*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）**

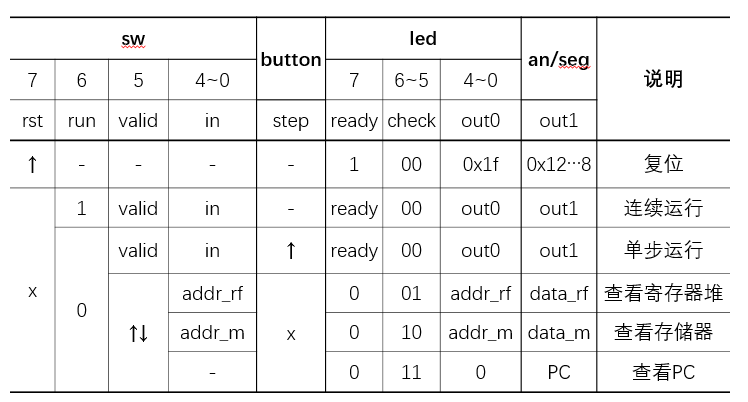
1. module  pdu\_1cycle(
2. input clk,
3. input rst,
4. input run,
5. input step,
6. output clk\_cpu,
7. input valid,
8. input [4:0] in,
9. output [1:0] check,
10. output [4:0] out0,
11. output [2:0] an,
12. output [3:0] seg,
13. output ready,
14. //IO\_BUS
15. input [7:0] io\_addr,
16. input [31:0] io\_dout,
17. input io\_we,
18. output [31:0] io\_din,
19. //Debug\_BUS
20. output [7:0] m\_rf\_addr,
21. input [31:0] rf\_data,
22. input [31:0] m\_data,
23. input [31:0] pc
24. );
25. reg [4:0] in\_r;
26. reg run\_r, step\_r, step\_2r, valid\_r, valid\_2r;
27. wire step\_p, valid\_pn;
28. reg clk\_cpu\_r;
29. reg [4:0] out0\_r;
30. reg [31:0] out1\_r;
31. reg ready\_r;
32. reg [19:0] cnt;
33. reg [1:0] check\_r;
34. reg [31:0] io\_din\_a;  reg ready\_a;
35. reg [4:0] out0\_a;
36. reg [31:0] out1\_a;
37. reg [3:0] seg\_a;
38. assign clk\_cpu = clk\_cpu\_r;
39. assign io\_din = io\_din\_a;
40. assign check = check\_r;
41. assign out0 = out0\_a;
42. assign ready = ready\_a;
43. assign seg = seg\_a;
44. assign an = cnt[19:17];
45. assign step\_p = step\_r & ~step\_2r;
46. assign valid\_pn = valid\_r ^ valid\_2r;
47. assign m\_rf\_addr = {{3{1'b0}}, in\_r};
48. always @(posedge clk) begin
49. run\_r <= run;
50. step\_r <= step;
51. step\_2r <= step\_r;
52. valid\_r <= valid;
53. valid\_2r <= valid\_r;
54. in\_r <= in;
55. end
56. always @(posedge clk, posedge rst) begin
57. if(rst)
58. clk\_cpu\_r <= 0;
59. else if (run\_r)
60. clk\_cpu\_r <= ~clk\_cpu\_r;
61. else
62. clk\_cpu\_r <= step\_p;
63. end
64. always @\* begin
65. case (io\_addr)
66. 8'h0c: io\_din\_a = {{27{1'b0}}, in\_r};
67. 8'h10: io\_din\_a = {{31{1'b0}}, valid\_r};
68. default: io\_din\_a = 32'h0000\_0000;
69. endcase
70. end
71. always @(posedge clk, posedge rst) begin
72. if (rst) begin
73. out0\_r <= 5'h1f;
74. out1\_r <= 32'h1234\_5678;
75. ready\_r <= 1'b1;
76. end
77. else if (io\_we)
78. case (io\_addr)
79. 8'h00: out0\_r <= io\_dout[4:0];
80. 8'h04: ready\_r <= io\_dout[0];
81. 8'h08: out1\_r <= io\_dout;
82. default: ;
83. endcase
84. end
85. always @(posedge clk, posedge rst) begin
86. if(rst)
87. check\_r <= 2'b00;
88. else if(run\_r)
89. check\_r <= 2'b00;
90. else if (step\_p)
91. check\_r <= 2'b00;
92. else if (valid\_pn)
93. check\_r <= check - 2'b01;
94. end
95. always @\* begin
96. ready\_a = 1'b0;
97. case (check\_r)
98. 2'b00: begin
99. out0\_a = out0\_r;
100. out1\_a = out1\_r;
101. ready\_a = ready\_r;
102. end
103. 2'b01: begin
104. out0\_a = in\_r;
105. out1\_a = rf\_data;
106. end
107. 2'b10: begin
108. out0\_a = in\_r;
109. out1\_a = m\_data;
110. end
111. 2'b11: begin
112. out0\_a = 5'b00000;
113. out1\_a = pc;
114. end
115. endcase
116. end
117. always @(posedge clk, posedge rst) begin
118. if (rst) cnt <= 20'h0\_0000;
119. else cnt <= cnt + 20'h0\_0001;
120. end
121. always @\* begin
122. case (an)
123. 3'd0: seg\_a = out1\_a[3:0];
124. 3'd1: seg\_a = out1\_a[7:4];
125. 3'd2: seg\_a = out1\_a[11:8];
126. 3'd3: seg\_a = out1\_a[15:12];
127. 3'd4: seg\_a = out1\_a[19:16];
128. 3'd5: seg\_a = out1\_a[23:20];
129. 3'd6: seg\_a = out1\_a[27:24];
130. 3'd7: seg\_a = out1\_a[31:28];
131. default: ;
132. endcase
133. end
134. endmodule

**3 CPU + PDU**

**3.1 模块链接实现**

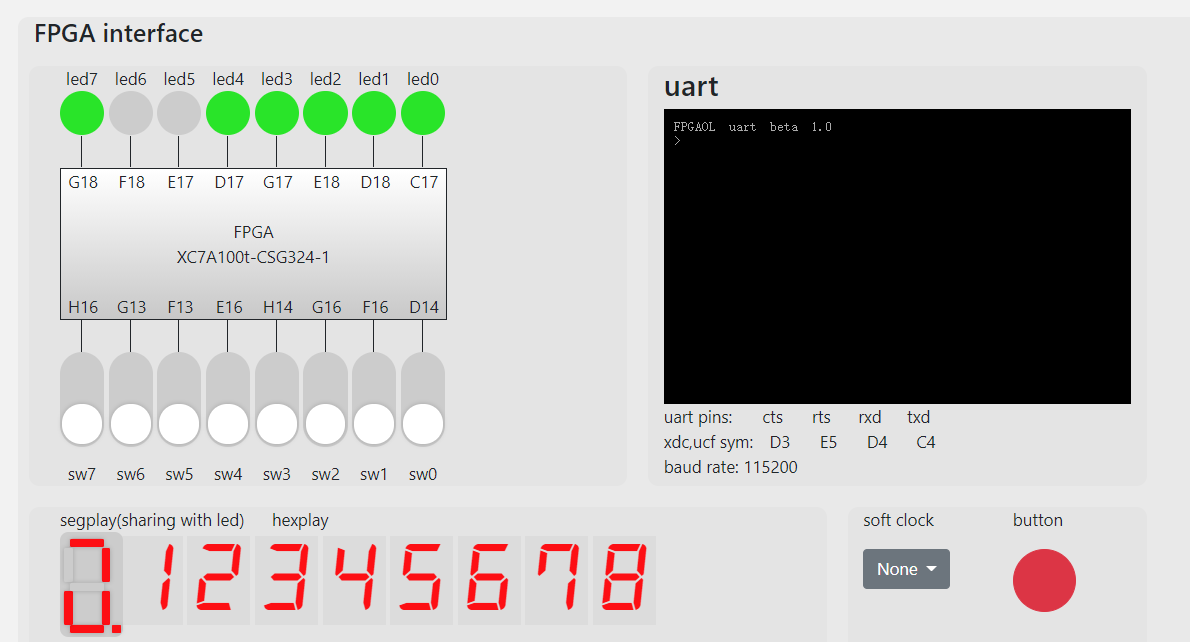
将两个模块连线连接起来

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. //IO\_BUS
13. wire [7:0] io\_addr;
14. wire [31:0] io\_dout;
15. wire io\_we;
16. wire [31:0] io\_din;
17. //Debug\_BUS
18. wire [7:0] m\_rf\_addr;
19. wire [31:0] rf\_data;
20. wire [31:0] m\_data;
21. wire [31:0] pc;
22. .clk(clk),
23. .rst(rst),
24. .io\_addr(io\_addr),
25. .io\_dout(io\_dout),
26. .io\_we(io\_we),
27. .io\_din(io\_din),
28. .m\_rf\_addr(m\_rf\_addr),
29. .rf\_data(rf\_data),
30. .m\_data(m\_data),
31. .pc(pc)
32. );
33. pdu\_1cycle debug(
34. .clk(clk),
35. .rst(rst),
36. .run(run),
37. .step(step),
38. .clk\_cpu(clk\_cpu),
39. .valid(valid),
40. .in(in),
41. .check(check),
42. .out0(out0),
43. .an(an),
44. .seg(seg),
45. .ready(ready),
46. .io\_addr(io\_addr),
47. .io\_dout(io\_dout),
48. .io\_we(io\_we),
49. .io\_din(io\_din),
50. .m\_rf\_addr(m\_rf\_addr),
51. .rf\_data(rf\_data),
52. .m\_data(m\_data),
53. .pc(pc)
54. );
55. endmodule

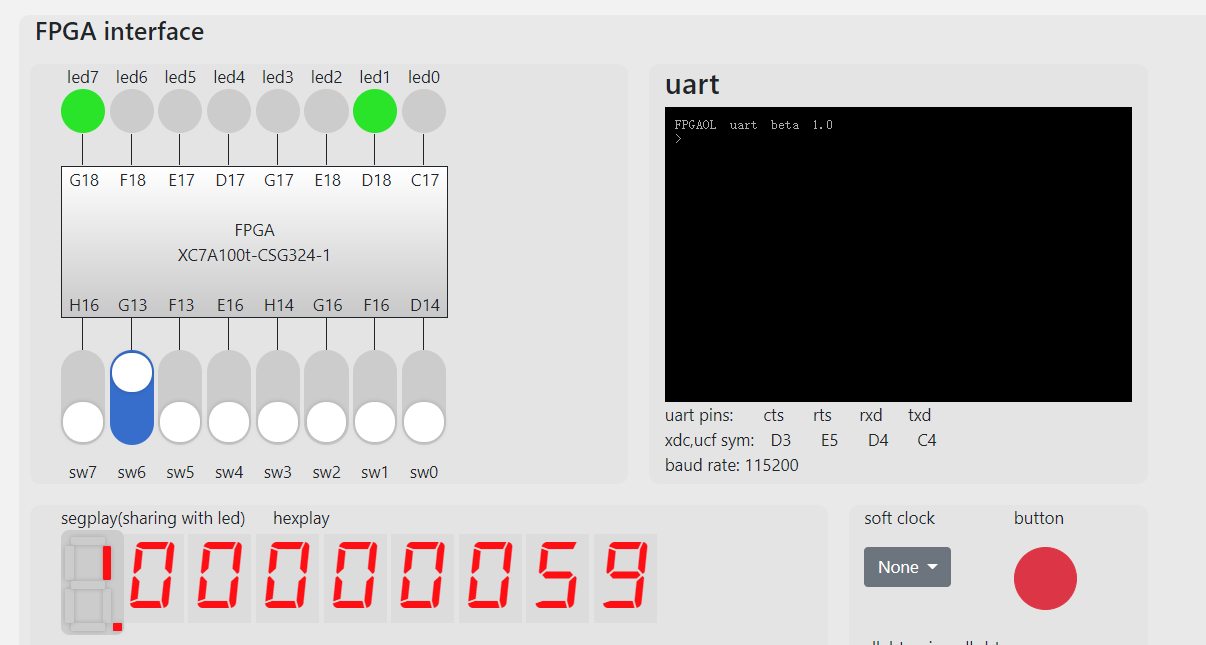


**3.2 平台测试**

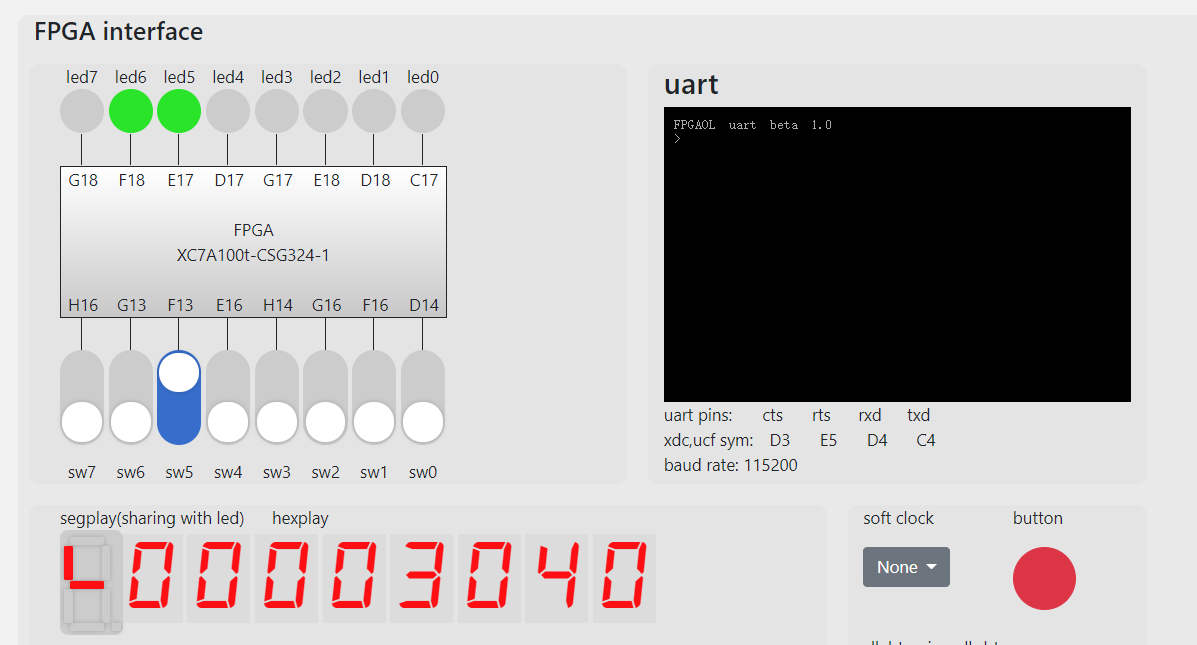
初始状态

****

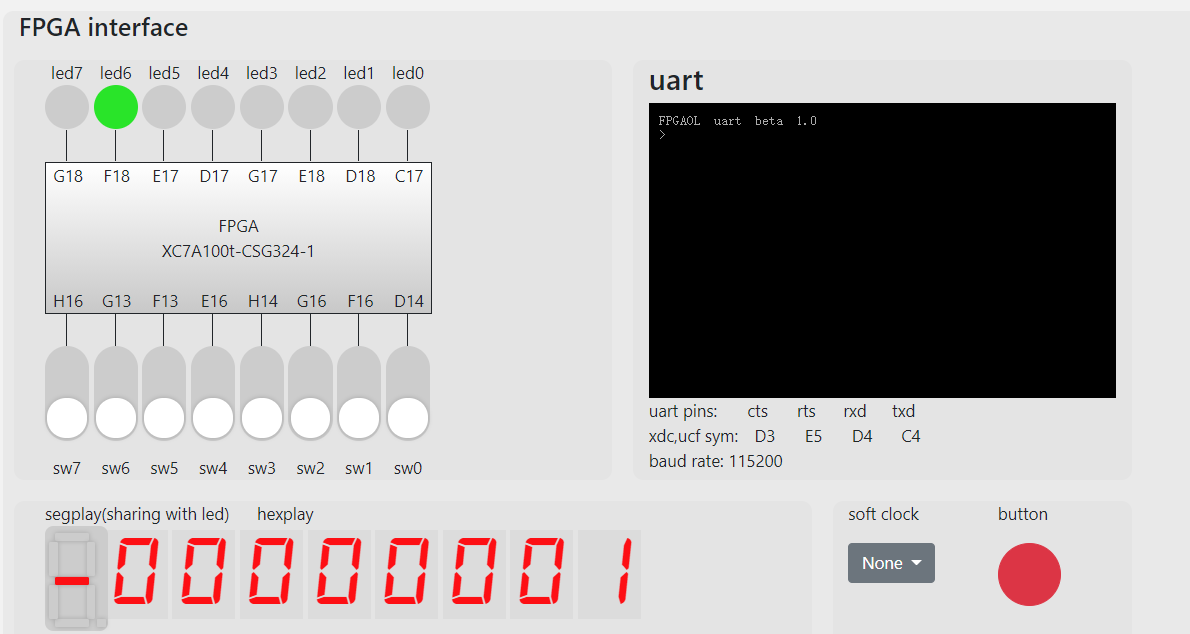
连续运行

****

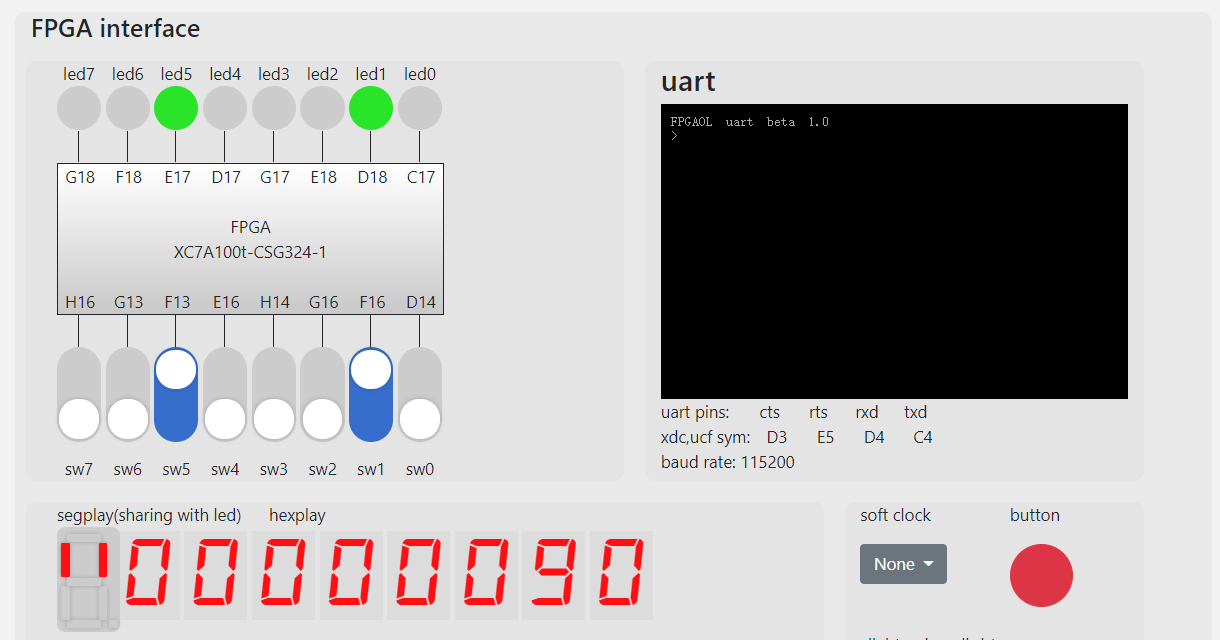
查看PC

****

查看存储器

****

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。