合肥工業大學

系统硬件综合设计报告

设计题目	多周期/流水线 CPU 设计
学生姓名	孙　淼
学　　号	2018211958
专业班级	计算机科学与技术 18-2 班
指导教师	刘军 陈田 李建华 安鑫
完成日期	2020 年 12 月 25 日

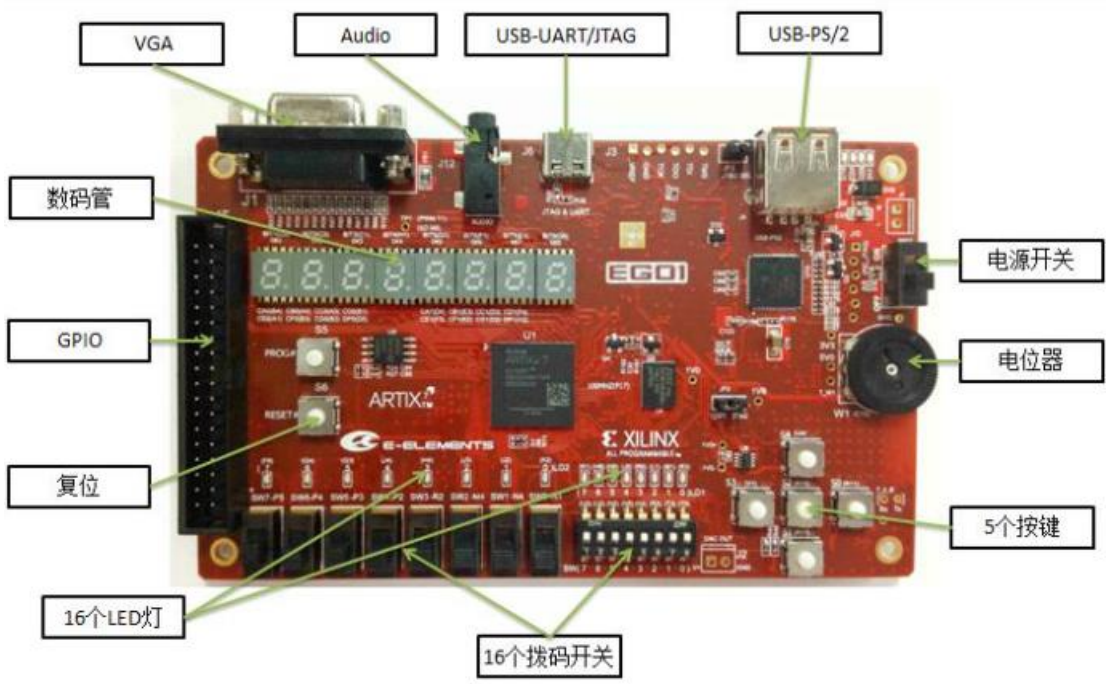
目录

01 写在前面

经过多门课程的理论和时间准备，我最终在大三上的第 17 周完成了多周期流水线 CPU 的设计和实现，该实验涉及的相关课程很多，主要的有陈田老师的计算机组成原理课程，李建华老师的计算机体系结构课程，徐娟老师的汇编语言课程，刘军老师的数字逻辑课程，需要自己学习的知识有 Verilog 硬件语言和 FPGA 实验板的相关知识。

1.1 课外准备工作

对 FPGA 开发板以及相关的知识如 Verilog 硬件语言进行学习，

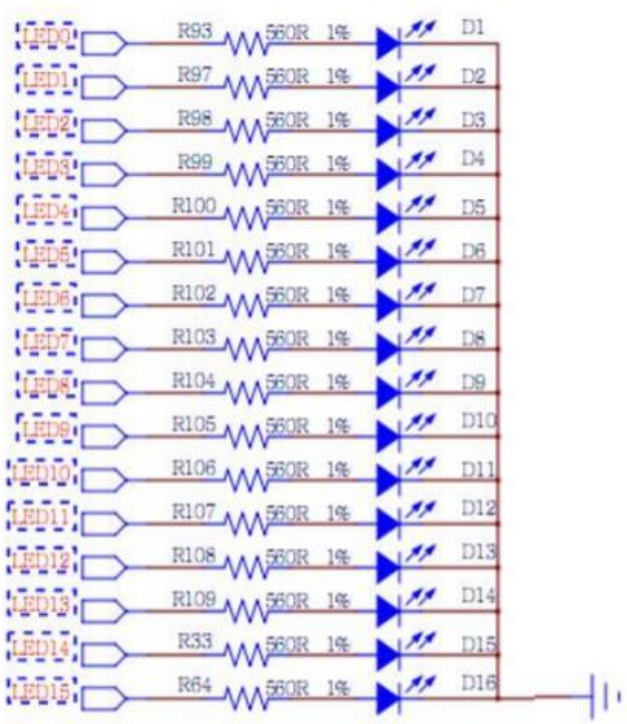


EGO1 是依元素科技基于 Xilinx Artix-7 FPGA 研发的便携式数模混合基础教学平台。EGO1 配备的 FPGA (XC7A35T-1CSG324C)具有大容量高性能等特点，能实现较复杂的数字逻辑设计;在 FPGA 内可以构建 MicroBlaze 处理器系统，可进行 SoC 设计。该平台拥有丰富的外设，

以及灵活的通用扩展接口。

由于我最终下板时是将指令在 LED 灯上进行显示，所以需要对板上 LED 灯的相关知识进行补充学习。

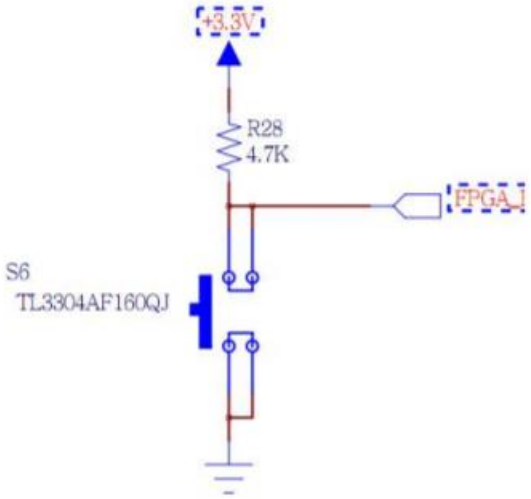
LED 在 FPGA 输出高电平时被点亮。



对应的管脚约束如下

名称	原理图标号	FPGA IO PIN	颜色
D1_0	LED1_0	K3	Green
D1_1	LED1_1	M1	Green
D1_2	LED1_2	L1	Green
D1_3	LED1_3	K6	Green
D1_4	LED1_4	J5	Green
D1_5	LED1_5	H5	Green
D1_6	LED1_6	H6	Green
D1_7	LED1_7	K1	Green
D2_0	LED2_0	K2	Green
D2_1	LED2_1	J2	Green
D2_2	LED2_2	J3	Green
D2_3	LED2_3	H4	Green
D2_4	LED2_4	J4	Green
D2_5	LED2_5	G3	Green
D2_6	LED2_6	G4	Green
D2_7	LED2_7	F6	Green

此外，为了便于我们观察分析指令，就需要我们实现按一次键，运行一个时钟周期，因此不能使用系统的时钟，所以还需要对专用按键逻辑复位 RST 的相关知识进行学习，以便用它来实现时钟信号的逻辑触发。



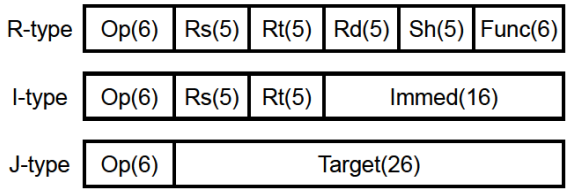
名称	原理图标号	FPGA IO PIN
复位引脚	FPGA_RESET	P15

这一部分的其他内容将在第六部分具体展开说明。

1.2 实验关键部分

本次实验实现的 CPU 基本特性如下：

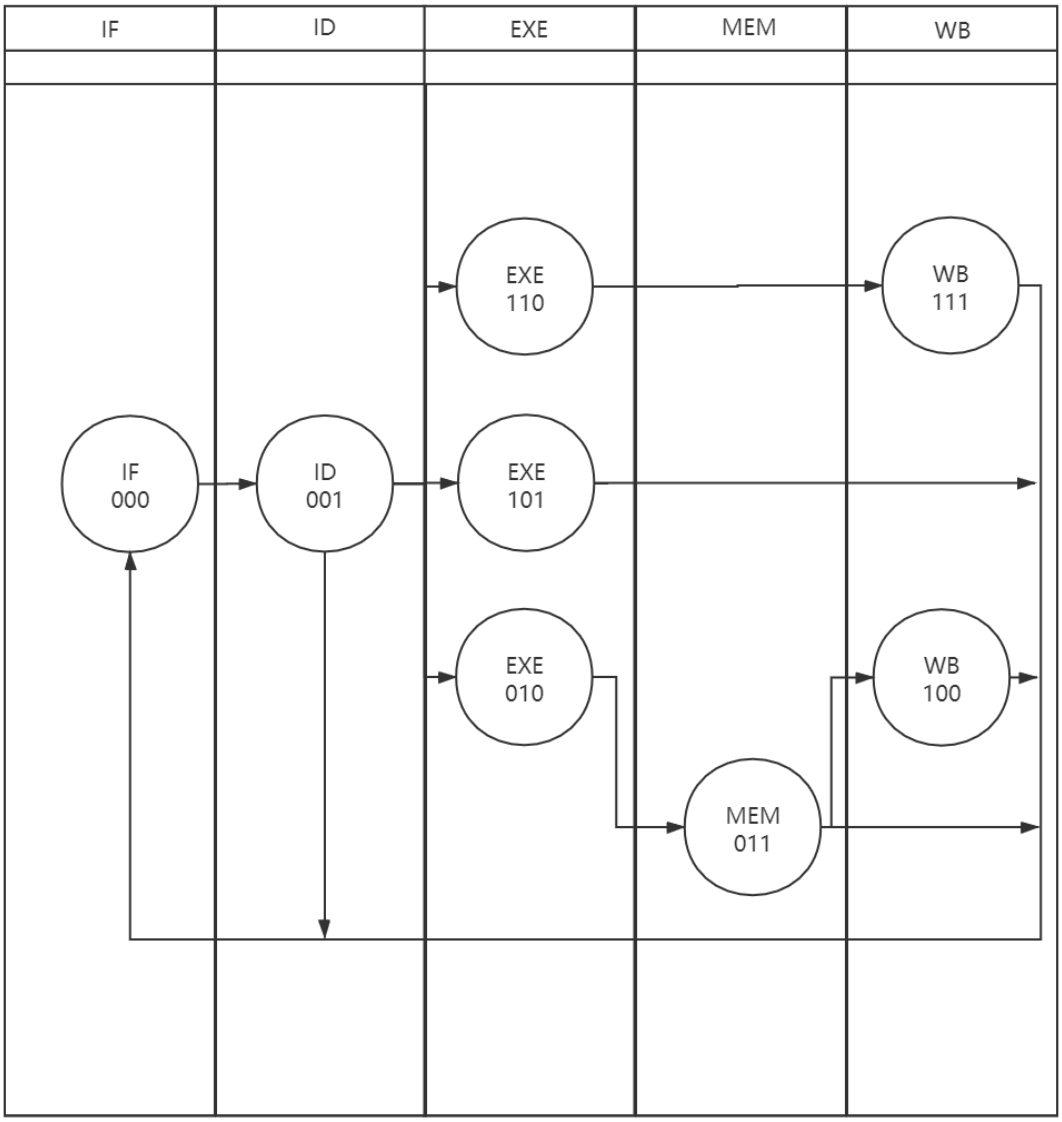
- 基于 MIPS32 指令集架构，支持 MIPS32 指令集中的部分指令，这将在第三部分展开介绍。采用 32 位定长指令格式，指令格式分别为 R 型、I 型、J 型。编码规则如下图所示。



- 将整个 CPU 的执行过程分成几个阶段，每个阶段用一个时钟去完

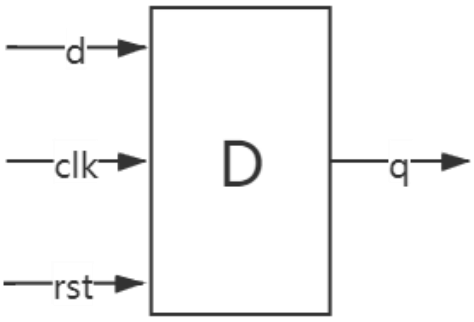
成，然后开始下一条指令的执行，而每种指令执行时所用的时钟数不尽相同，这就是所谓的多周期 CPU。CPU 在处理指令时，一般需要经过以下几个阶段：取指令(IF)、指令译码(ID)、指令执行(EXE)、存储器访问(MEM)、结果写回(WB)。

- 状态的转移有的是无条件的，有的是有条件的，以下图为例，IF

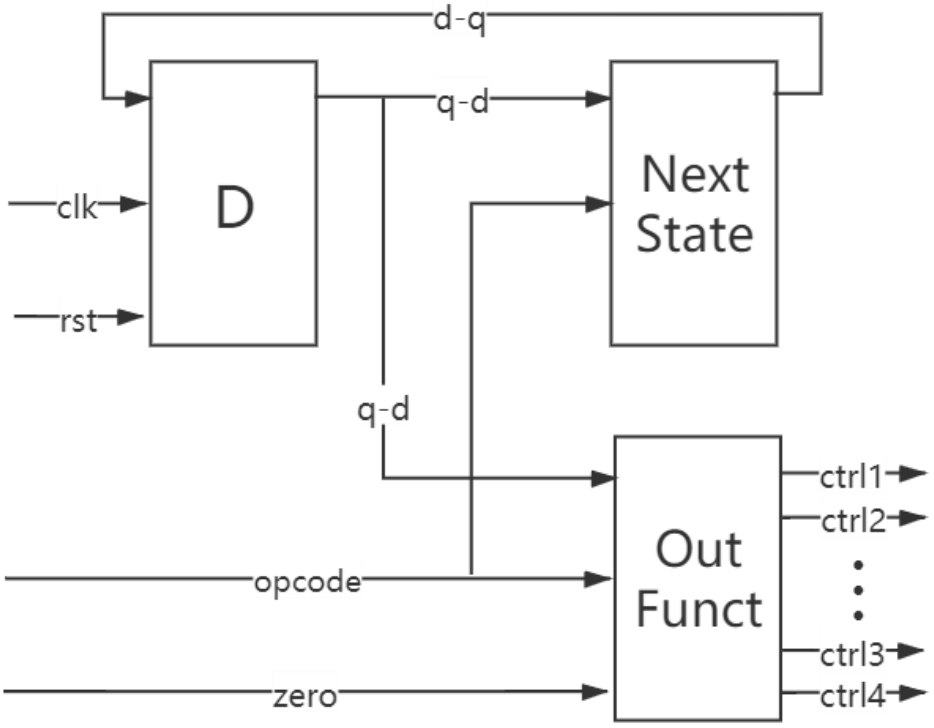


到 ID 的转移就是无条件的，ID 到 EXE 就是有条件的，具体的五个阶段之间的转移是借助识别指令操作码实现的，具体的实现见第三部分 NextState 模块。

- 存储采用哈佛结构，使用分开的指令、数据存储器。
- 多周期 CPU 控制部件的电路结构由 D 触发器实现

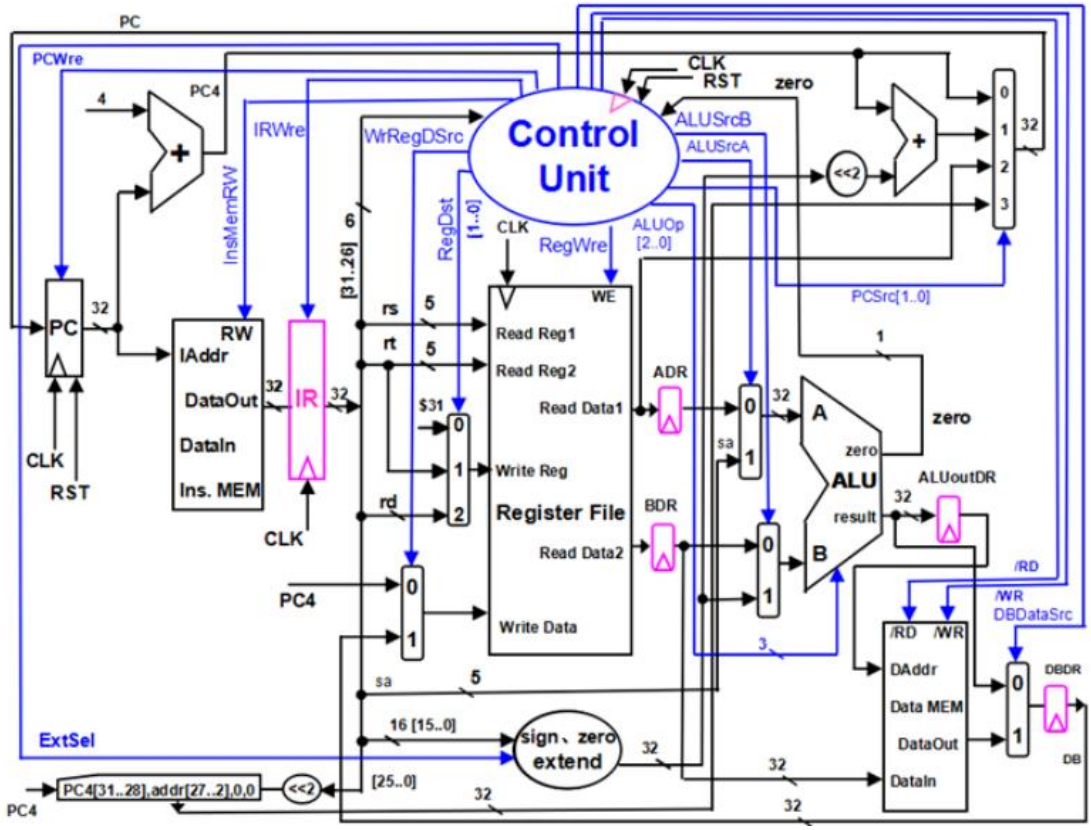


三个 D 触发器用于保存当前状态，是时序逻辑电路，RST 用于初始化状态“000”，另外两个部分都是组合逻辑电路，一个用于产生下一个阶段的状态，另-一个用于产生每个阶段的控制信号。



02 数据通路图

2.1 数据通路图与控制线路



上图是一个简单的基本上能够在多周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。下面对上图的一些细节略作解释，更具体的说明见第三部分

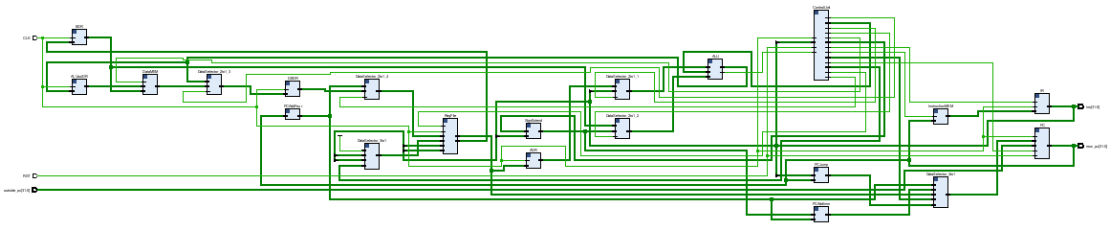
- 指令和数据各存储在不同存储器中,即有指令存储器和数据存储器。访问存储器时，先给出地址,然后由读或写信号控制操作。对于寄存器组，读操作时,给出寄存器地址(编号)，输出端就直接输出相应数据;而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发写入。
- 五个红紫色的寄存器分别是一个 IR 指令寄存器和四个降低大延

迟作用的寄存器 ADR、BDR、ALUoutDR 和 DBDR，其中 IR 指令寄存器的目的是使指令代码保持稳定，ADR、BDR、ALUoutDR、DBDR 四个寄存器不需要写使能信号，其作用是切分数据通路，将大组合逻辑切分为若干个小组合逻辑，大延迟变为多个分段小延迟。它们的具体实现和具体作用将在第三部分详细介绍。

- 14 个关键标蓝的控制信号是由 Control Unit 发出，它们具体的功能和控制对象以及实现方法将在第三部分的 Control Unit 模块详细介绍。

03 各模块详细设计与代码

接下来按照在 Vivado 上面进行的实验步骤进行介绍，完成各个模块的代码后，我们可以在 RTL ANALYSIS 里面进行 Schematic。



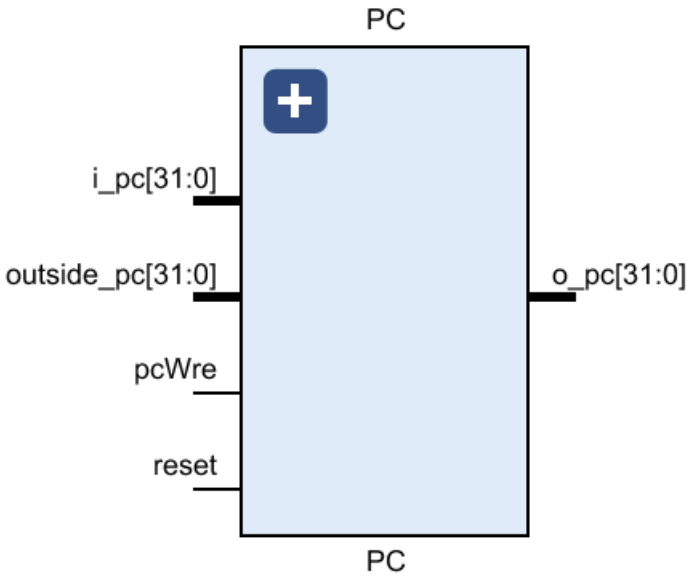
我们按照设计时候的思路顺序对这其中的各个模块进行分析。

3.1 PC 及相关模块

首先是程序计数器 PC 的 i_pc

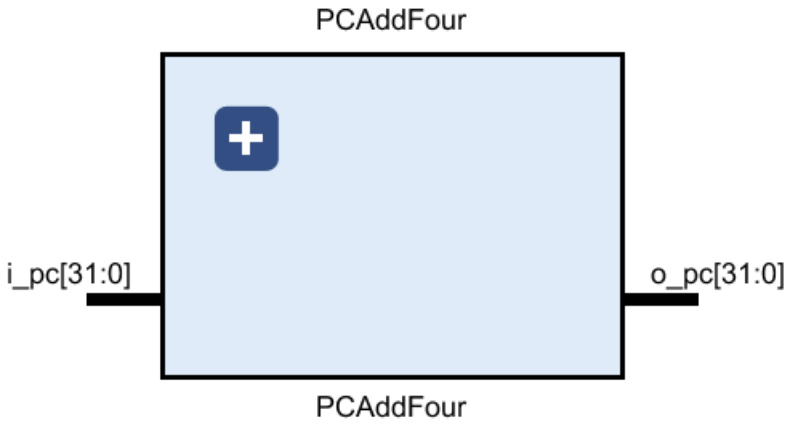
考虑到 PC 的下一条指令地址有几种情况：

- 顺序存储的下一条指令的地址
- 子程序的地址
- 执行跳转指令后的地址



针对这几种不同的情况，需要加入三个对应的单元

- PCAddFour

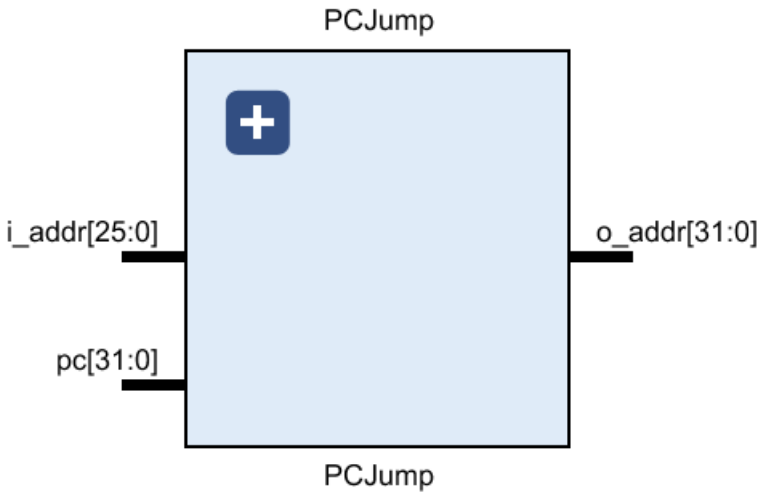


该单元的功能就是实现 PC<-PC+4，所以与该单元相关的指令有 add、addi、sub、or、ori、and、slt、slti、sll、sw、lw、beq。

```
1. `timescale 1ns / 1ps
2. // 实现 PC 递增
3. // @param i_pc 输入的 pc 值
4. // @param o_pc 输出的 pc 值
5. module PCAddFour(i_pc, o_pc);
```

```
6.    input wire [31:0] i_pc;
7.    output wire [31:0] o_pc;
8.    assign o_pc[31:0] = i_pc[31:0] + 4;
9.  endmodule
```

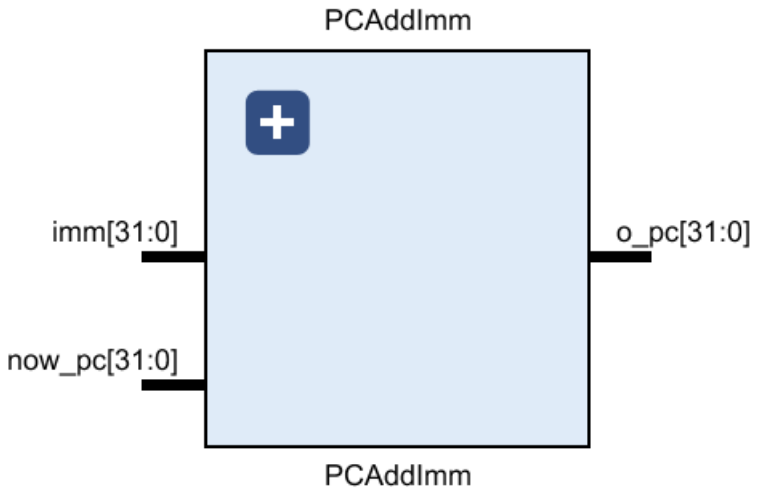
- PCJump



该单元的功能就是实现跳转，比如 PC<-rs，与之相关的指令有 jr；或者是 PC<-{PC[31..28],addr[37..2],0,0}，与之相关的指令有 j，jal。(这里对应两种不同的指令,所以我将其分为两类,于是 1+2+1=4)

```
1.  `timescale 1ns / 1ps
2.  // pc 跳转调用子程序 j jal
3.  // @param pc 执行该指令时 pc 的值
4.  // @param i_addr 输入的地址
5.  // @param o_addr 输出的地址
6.
7.  module PCJump(pc, i_addr, o_addr);
8.    input [31:0] pc;
9.    input [25:0] i_addr;
10.   output reg[31:0] o_addr;
11.   reg [27:0] mid; // 用于存放中间值
12.   // 输出地址的前四位来自 pc[31:28]，中间 26 位来自 i_addr[27:2]，后两位是 0
13.   always @(i_addr) begin
14.      mid = i_addr << 2;
15.      o_addr <= {pc[31:28], mid[27:0]};
16.   end
17. endmodule
```

- PCAddImm

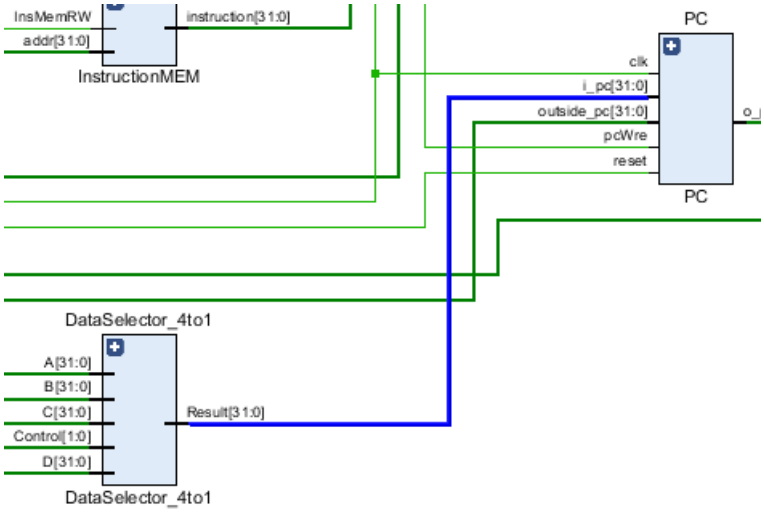


该单元的功能就是实现下一个地址与符号拓展的立即数加得到新地址，与之相关的指令有 beq。

```
1.  `timescale 1ns / 1ps
2.  // PC 加立即数
3.  // @param now_pc  当前 pc 值
4.  // @param o_pc 输出 pc 值
5.  // @param imm 立即数
6.  module PCAddImm(now_pc, imm, o_pc);
7.    input [31:0] now_pc, imm;
8.    output [31:0] o_pc;
9.    // 内存单元是以字节为单位的，32 位地址大小为 4 个字节，所以 pc=pc+imm*4
10.   assign o_pc = now_pc + (imm << 2);
11. endmodule
```

上面介绍到，此处我将 PC 的跳转情况分为四类，所以最后需要一个四选一的选择器。

在下图可以看到，PC 的输入 i_pc 是由四选一选择器选择出来的，选择器的四个选择端口就是来自上面分析的四种情况对应得到的下一条指令地址。

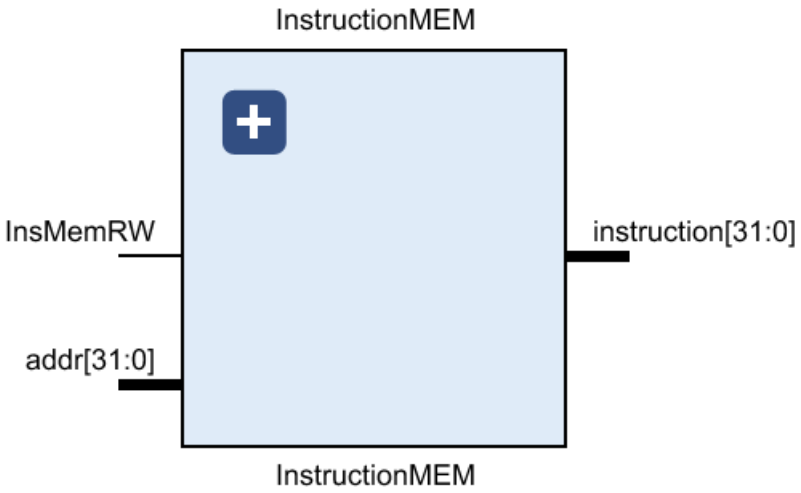


接下来我们解释一下程序计数器 PC 的 outside_pc，这是一个给定的值，即 32 位 0，因为当 RST 和 PCWre 信号变化时，需要判断 RST 是否为 1，若为 1，就要进行复位，也就是将 PC 地址重置为预设的 outside_pc，也就是 0，若 RST 为 0，那么改变的就是 PCWre，我们就将输入的 pc 值进行输出，否则不变。

```
1.  `timescale 1ns / 1ps
2.  // PC 模块的实现
3.  // @param clk 时钟信号
4.  // @param pcWre 信号
5.  // @param reset 信号
6.  // @param i_pc 输入的 pc 值
7.  // @param o_pc 输出的 pc 值
8.  // @param outside_pc ???
9.  module PC(clk, i_pc, pcWre, reset, outside_pc, o_pc);
10.   input wire clk, pcWre, reset;
11.   input wire [31:0] i_pc, outside_pc;
12.   output reg [31:0] o_pc;
13.   always @(pcWre or reset) begin // 这里和单周期不太一样，存在延迟的问题，只有当
    pcWre 改变的时候或者 reset 改变的时候再检测
14.    // reset
15.     if (reset) begin
16.       o_pc = outside_pc;
17.     end else if (pcWre) begin
18.       o_pc = i_pc;
19.     end else if (!pcWre) begin
20.         o_pc = o_pc;
```

```
    end
  end
endmodule
```

3.2 指令存储器与指令寄存器



接下来我们介绍 PC 后面的 InstructionMEM，作为（指令）存储器，其最关键的控制信号就是 InsMemRW，当该信号为 0 时，写指令存储器，当该信号为 1 时，读指令存储器。

内部的具体实现与单周期中的设计是类似的，定义了 8 位的寄存器数组，并将指令的 32 位二进制数分割成 4 个 8 位的小段并顺序存储在数组中。当指令的读信号来临时，根据 pc 值从 mem 中依次取出 mem[pc]。mem[pc+1]，mem[pe+2]。mem[pc+3]，并存入指令对应的 instruction[31:24]，instruction[23: 16]. instruction[15:8]，instruction[7:0]。

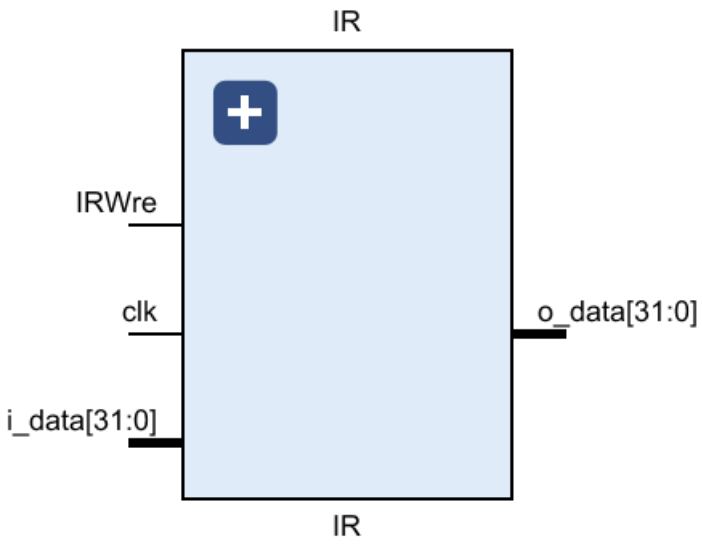
```
`timescale 1ns / 1ps
// 指令存储单元的实现
// @param InsMemRW 指令存储单元信号
// @param addr pc 上指令的地址
// @param outside_pc 获取初始化的 pc
// @param instruction 取得的指令
module InstructionMEM (addr, InsMemRW, instruction);
    input InsMemRW;
```

```
9.      input [31:0] addr;
10.     output reg [31:0] instruction;
11.     // 8位内存单元，每条指令的二进制代码占四个内存单元
12.     reg [7:0] mem [0:127];
13.      initial begin
14.          mem[0]=8'b11100000;
15.          mem[1]=8'b00000000;
16.          mem[2]=8'b00000000;
17.          mem[3]=8'b00000010;
18.
19.          mem[4]=8'b11100111;
20.          mem[5]=8'b11100000;
21.          mem[6]=8'b00000000;
22.          mem[7]=8'b00000000;
23.
24.          mem[8]=8'b00001000;
25.          mem[9]=8'b00000001;
26.          mem[10]=8'b00000000;
27.          mem[11]=8'b00000100;
28.
29.          mem[12]=8'b00001000;
30.          mem[13]=8'b00000010;
31.          mem[14]=8'b00000000;
32.          mem[15]=8'b00001000;
33.
34.          mem[16]=8'b11000000;
35.          mem[17]=8'b01000010;
36.          mem[18]=8'b00000000;
37.          mem[19]=8'b00000000;
38.
39.          mem[20]=8'b00000000;
40.          mem[21]=8'b01000001;
41.          mem[22]=8'b00011000;
42.          mem[23]=8'b00000000;
43.
44.          mem[24]=8'b00000100;
45.          mem[25]=8'b01100001;
46.          mem[26]=8'b00011000;
47.          mem[27]=8'b00000000;
48.
49.          mem[28]=8'b11010000;
50.          mem[29]=8'b01000011;
51.          mem[30]=8'b11111111;
52.          mem[31]=8'b11111110;
```

```
53.
54.         mem[32]=8'b01001000;
55.         mem[33]=8'b00100001;
56.         mem[34]=8'b00000000;
57.         mem[35]=8'b00000001;
58.
59.         mem[36]=8'b01000000;
60.         mem[37]=8'b01000001;
61.         mem[38]=8'b00011000;
62.         mem[39]=8'b00000000;
63.
64.         mem[40]=8'b00000000;
65.         mem[41]=8'b01000000;
66.         mem[42]=8'b00011000;
67.         mem[43]=8'b00000000;
68.
69.         mem[44]=8'b01000100;
70.         mem[45]=8'b01100010;
71.         mem[46]=8'b00001000;
72.         mem[47]=8'b00000000;
73.
74.         mem[48]=8'b01100000;
75.         mem[49]=8'b00000010;
76.         mem[50]=8'b00001000;
77.         mem[51]=8'b10000000;
78.
79.         mem[52]=8'b10011000;
80.         mem[53]=8'b00100010;
81.         mem[54]=8'b00110000;
82.         mem[55]=8'b00000000;
83.
84.         mem[56]=8'b10011000;
85.         mem[57]=8'b01000001;
86.         mem[58]=8'b00111000;
87.         mem[59]=8'b00000000;
88.
89.         mem[60]=8'b10011100;
90.         mem[61]=8'b00100110;
91.         mem[62]=8'b00000000;
92.         mem[63]=8'b00000001;
93.
94.         mem[64]=8'b10011100;
95.         mem[65]=8'b11000111;
96.         mem[66]=8'b00000000;
```

```
97.          mem[67]=8'b00000001;
98.
99.          mem[68]=8'b11101000;
100.          mem[69]=8'b00000000;
101.          mem[70]=8'b00000000;
102.          mem[71]=8'b00000001;
103.
104.          mem[72]=8'b11000100;
105.          mem[73]=8'b01000100;
106.          mem[74]=8'b00000000;
107.          mem[75]=8'b00000000;
108.
109.          mem[76]=8'b11111100;
110.          mem[77]=8'b00000000;
111.          mem[78]=8'b00000000;
112.          mem[79]=8'b00000000;
113.         instruction = 0;
114.      end
115.     always @(addr or InsMemRW)
116.         if (InsMemRW) begin
117.           instruction[31:24] = mem[addr];
118.           instruction[23:16] = mem[addr+1];
119.           instruction[15:8] = mem[addr+2];
120.           instruction[7:0] = mem[addr+3];
121.         end
122. endmodule
```

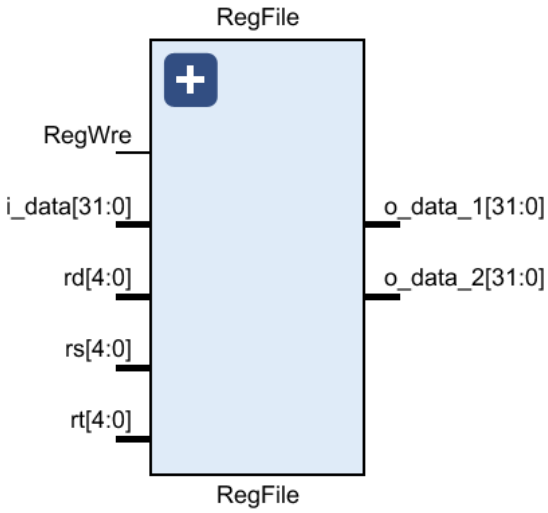
读出的指令并不会直接分析，而是会暂存在一个指令寄存器 IR 里面，目的是使运行稳定，这与多周期 CPU 的运行原理有关。



IR 情况如上，其控制信号 IRWre 是 IR 的写使能信号，当 IRWre 为 0 时，指令寄存器 IR 不更改，当 IRWre 为 1 时，指令寄存器 IR 写使能，向指令存储器发出读指令的代码后，这个信号就随之发出，并且在时钟上升沿的时候 IR 接收从指令存储器 InstructionMEM 发来的指令代码，所以该单元与每一条指令都相关。

```
1.  `timescale 1ns / 1ps
2.  // 用于临时存储指令的二进制形式
3.  // @param i_data 输入的数据
4.  // @param clk 时钟信号
5.  // @param IRWre 输入 IR 的控制信号
6.  // @param o_data 输出的数据
7.  module IR(i_data, clk, IRWre, o_data);
8.    input clk, IRWre;
9.    input [31:0] i_data;
10.   output reg[31:0] o_data;
11.   always @(negedge clk) begin // 存在延迟的问题，所以用下降沿触发，对数据传输没有什么影响
12.     if (IRWre) begin
13.         o_data = i_data;
14.      end
15.   end
16. endmodule
```

3.3 寄存器堆



送出的指令经过三选一选择器选择后送入 RegFile，其原理就是根

据指令中的 rs，rt 到对应的寄存器中获取数据，然后作为 o_data_1 和 o_data_2 输出，又或者是根据指令中的 rd 将输入的数据存入寄存器中，需要注意的是，在 MIPS 指令集中，一号寄存器始终为 0，（李建华老师提到过，这么做的目的是提高一些与 0 进行比较的指令的执行速度，因为很多比较指令都会化为与 0 比较，比如 A>B?会被化为 A-B>0?）所以我们在设计 RegFile 时候也需要考虑到这一点。

其中 RegWre 控制信号是对指令对寄存器的写与否进行判断，如果 RegWre 为 0，就不写寄存器，比如指令 beq、j、sw、jr、halt，如果 RegWre 为 1，寄存器组写使能，比如指令 add、sub、addi、or、and、ori、slt、slti、sll、lw、jal。

```
1.  `timescale 1ns / 1ps
2.  // 寄存器组的实现
3.  // @param rs 输入数据源 1 所在的寄存器号
4.  // @param rt 输入数据源 2 所在的寄存器号
5.  // @param rd 结果存储的寄存器号
6.  // @param i_data 输入的数据
7.  // @param RegWre 输入寄存器组的控制信号
8.  // @param clk 时钟信号
9.  // @param o_data_1 输出数据 1
10. // @param o_data_2 输出数据 2
11. module RegFile (rs, rt, rd, i_data, RegWre, clk, o_data_1, o_data_2);
12.   input [4:0] rs, rt, rd;
13.   input [31:0] i_data;
14.   input RegWre, clk;
15.   output [31:0] o_data_1, o_data_2;
16.   reg [31:0] register [0:31];
17.   initial begin
18.    // 只需要确定零号寄存器的值就好，$0 恒等于 0 mips 的规定
19.     register[0] = 0;
20.   end
21.   assign o_data_1 = register[rs];
22.   assign o_data_2 = register[rt];
23.   always @(i_data or rd) begin
```

```
    // rd != 0 是确保零号寄存器不会改变的作用(MiPS)
     if ((rd != 0) && (RegWre == 1)) begin
       register[rd] = i_data;
     end
   end
endmodule
```

如果有读寄存器的操作，那么读出来的两个数据 o_data_1 和 o_data_2 分别送入 ADR 和 BDR，这些寄存器的作用在数字电路中提到过，目的是切分数据通路，将大组合逻辑切分为若干个小组合逻辑，从而可以化大延迟为小延迟，在最后会对这些寄存器进行介绍，现在只介绍主要的模块单元。

3.4 加法器及相关模块

读出的 o_data_1 和 o_data_2 最终将送入加法器 ALU，但是送入之前，需要进行一次二选一选择器的选择，这一部分涉及好几个模块，我们按照数据通路图由上而下来介绍：

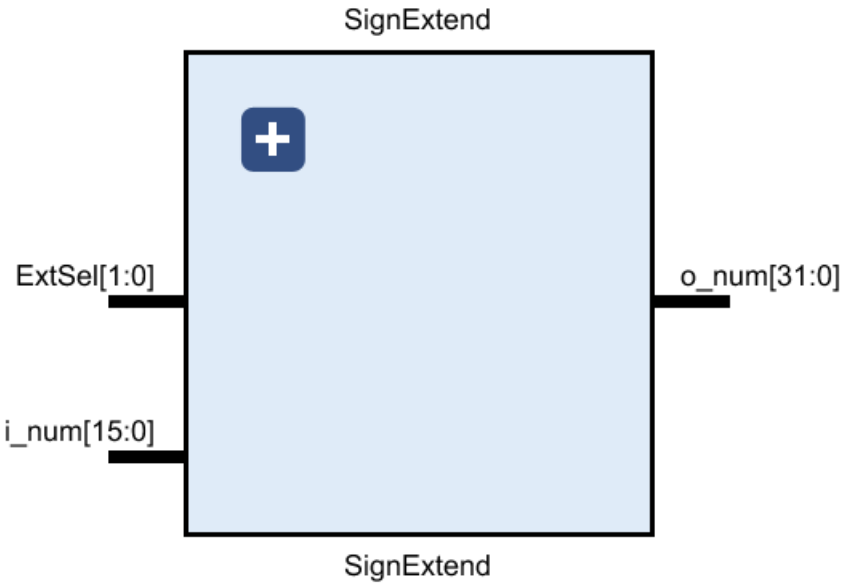
首先是对应 o_data_1 和 o_data_2 的两个控制信号，ALUSrcA 和 ALUSrcB：

当 ALUSrcA 为 0，对应的二选一的选择器的输出值则来自寄存器堆 data1 输出，相关指令：add、 sub、 addi、 or、 and、ori、beg、slt、 slti、 sw、lw；当 ALUSrcA 为 1，则来自移位数 sa，同时，进行(zero-extend)sa， 即{27(0),sa}，相关指令：sll

当 ALUSrcB 为 0，对应的二选一的选择器的输出值则来自寄存器堆 data2 输出，相关指令：add、sub、addi、or、and、ori、beg、slt、 slti、sll；

当 ALUSrcB 为 1，来自 sign 或 zero 扩展的立即数,相关指令:addi、 slti、 ori、 lw、sw。下面介绍其引出的符号拓展单元。

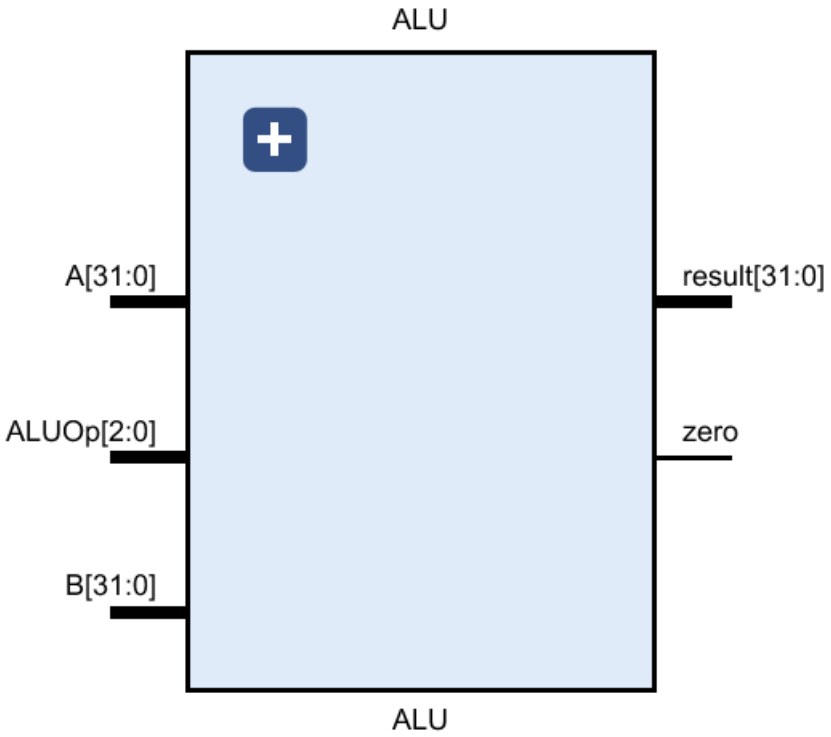
该单元就是拓展单元，其具体拓展方式取决于控制信号 ExtSel，此处有三种拓展方式,所以 ExtSel 是两位数,对应四种二进制组合，当 ExtSel 为 00，是 sa 的拓展，(zero-extend)immediate,与之相关



指令有 ori；当 ExtSel 为 01 时，是无符号拓展，当 ExtSel 为 10 时，是有符号拓展，(sign-extend)immediate，与之相关的指令有 addi、slti、lw、sw、beq。

```
1.  `timescale 1ns / 1ps
2.  // 符号扩展单元的实现
3.  // @param i_num 输入的数据
4.  // @param ExtSel 控制符号扩展单元的信号
5.  // @param o_num 输出的数据
6.  module SignExtend(i_num, ExtSel, o_num);
7.    input [15:0] i_num;
8.    input [1:0] ExtSel;
9.    output reg[31:0] o_num;
10.   initial begin
11.     o_num = 0;
12.   end
13.   always @(i_num or ExtSel) begin
```

```
    case(ExtSel)
        // ExtSel 为00时，sa位扩展
        2'b00: o_num <= {{27{0}}, i_num[10:6]};
        // ExtSel 为01时，无符号立即数扩展
        2'b01: o_num <= {{16{0}}, i_num[15:0]};
        // ExtSel 为10时，有符号立即数扩展,(符号扩展：将扩展后的数据的高(32-n)位置为立即数的最高)
        2'b10: o_num <= {{16{i_num[15]}}, i_num[15:0]};
        // 其它情况默认    有符号立即数扩展
        default: o_num <= {{16{i_num[15]}}, i_num[15:0]}; // 默认符号扩展
    endcase
  end
endmodule
```



上面就是加法器 ALU，这里的 ALU 与上学期所作做的单周期 CPU 里面的设计是一样的，因为 ALU 的运算种类较多，所以当其引脚 A，B，ALUOp 发生变化时，我们选择使用 case 语句来根据 ALUOp 的值来执行对应的操作，具体可以参照下表：

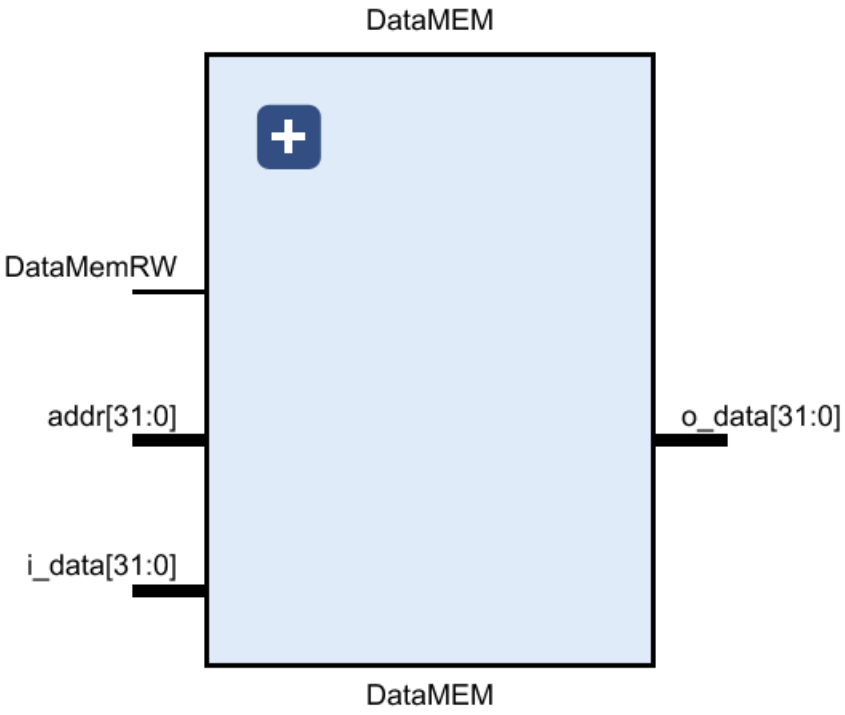
ALUOp[2..0]	执行的运算	对应的功能
000	Y=A+B	加
001	Y=A-B	减
010	if (A < B &&((A[31] == 0 && B[31]==0) || (A[31] == 1 && B[31]==1))) result = 1; else if (A[31] == 0 && B[31]==1) result = 0; else if (A[31] == 1 && B[31]==0) result = 1; else result = 0;	带符号比较 AB
011	Y=(A<B)?1:0	不带符号比较 AB
100	Y=B<<A	B 左移 A 位
101	Y=A∪B	或
110	Y=A∩B	与
111	Y=A⊕B	异或

具体的实现见下面代码

```
1.  `timescale 1ns / 1ps
2.  module ALU(A, B, ALUOp, zero, result);
3.    input [31:0] A, B;
4.    input [2:0] ALUOp;
5.    output zero;
6.    output reg [31:0] result;
7.    initial begin
8.          result = 0;
9.    end
10.   assign zero = (result? 0 : 1);
11.   always @(A or B or ALUOp) begin
12.     case(ALUOp)
13.       3'b000: result = A + B;
14.       3'b001: result = A - B;
15.       3'b010: begin
```

```
16.          if (A < B &&(( A[31] == 0 && B[31]==0)  || (A[31] == 1 && B[31]==1
    ))  result = 1;
17.          else if (A[31] == 0 && B[31]==1)  result = 0;
18.          else if (A[31] == 1 && B[31]==0)  result = 1;
19.          else result = 0;
20.       end
21.       3'b011: result = (A < B ? 1 : 0);
22.       3'b100: result = B << A;
23.       3'b101: result = A | B;
24.       3'b110: result = A & B;
25.       3'b111: result = (~A & B) | (A & ~B);
26.       default: result = 0;
27.     endcase
28.   end
29. endmodule
```

3.5 数据存储器



数据存储器的实现与单周期是类似的。定义 8 位的寄存器数组，将 32 位的数据分割成 4 个 8 位的小段进行存取。不过这里我将 D，R 两个信号合并为 DataMEMRW，当 DataMEMRW 为 1（/WR）时进行写操作，当 DataMEMRW 为 0(/RD)时进行读操作。写与读操作的单位是 8 位二进制

数，一共操作四次。与之相关的指令有 lw。

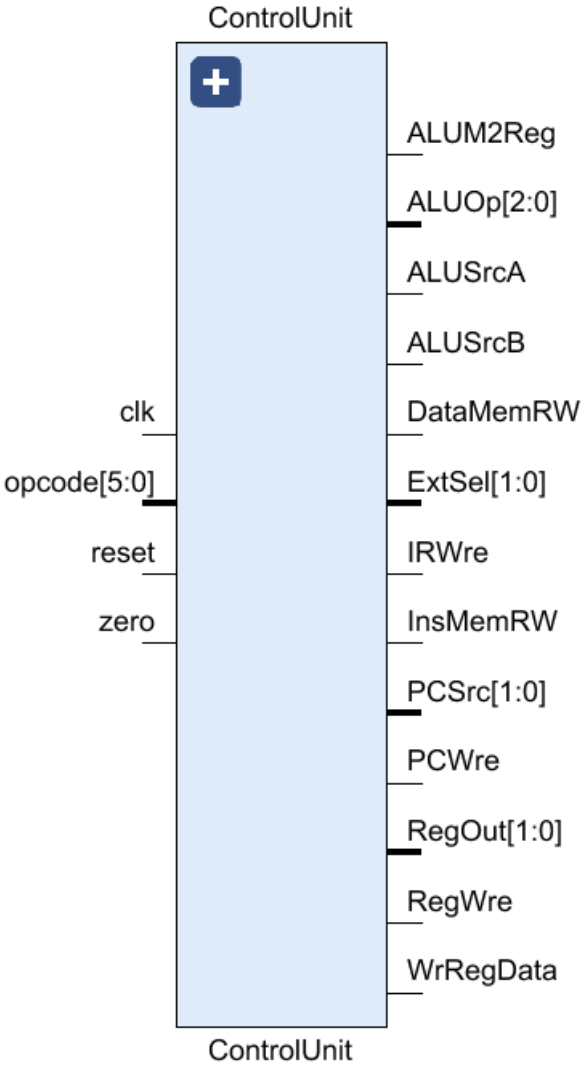
输出的数还需与 ALU 的运算结果输出进行选择，与之相关的指令有 add、sub、addi、or、and、ori、slt、slti、sll。

该值经过 DBDR 后再与来自 PC+4 的数据进行二选一选择器的选择，然后写回入寄存器组寄存器。

```
1.  `timescale 1ns / 1ps
2.  // 数据存储器的实现
3.  // @param i_data 输入的数据
4.  // @param addr 输入的地址
5.  // @param DataMemRW 输入数据存储器的信号，用 1 代表/WR 信号，用 0 代表/RD 信号，我将这两个信号合为一个
6.  // @param o_data 读取的数据
7.  module DataMEM (i_data, addr, DataMemRW, o_data);
8.      input [31:0] i_data;
9.      input [31:0] addr;
10.     input DataMemRW;
11.     output reg [31:0] o_data;
12.     reg [7:0] memory [0:63];
13.      initial begin
14.         o_data = 0;
15.      end
16.       // 使用大端方式储存，这里有更改（不需要乘 4）
17.     always @(addr or i_data or DataMemRW) begin
18.       if (DataMemRW) begin // 1 为 /WR
19.           memory[addr] = i_data[31:24];
20.           memory[addr+1] = i_data[23:16];
21.           memory[addr+2] = i_data[15:8];
22.           memory[addr+3] = i_data[7:0];
23.       end else begin // 0 为 /RD
24.           o_data[31:24] = memory[addr];
25.           o_data[23:16] = memory[addr+1];
26.           o_data[15:8] = memory[addr+2];
27.           o_data[7:0] = memory[addr+3];
28.       end
29.     end
30. endmodule
```

3.6 控制单元及其三个子模块

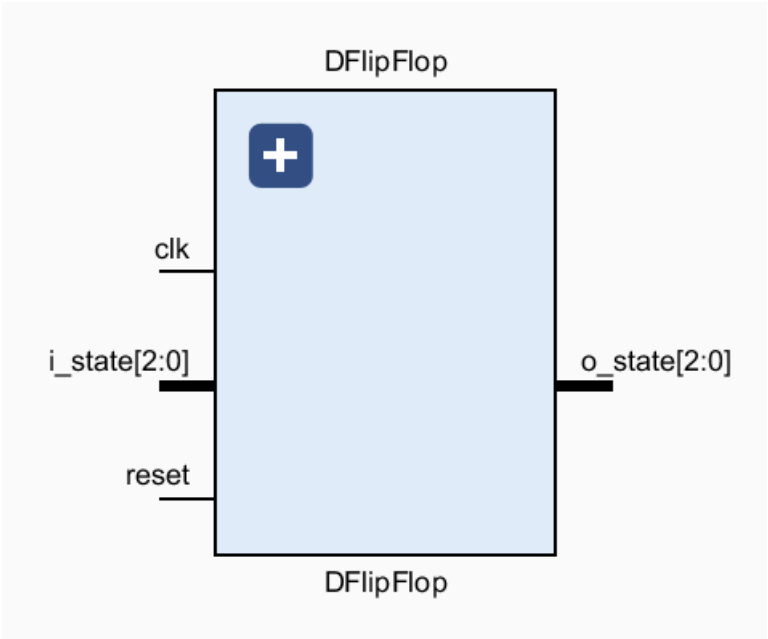
上面解释各个模块的时候已经对各个模块对应的控制信号进行了介绍，所以此处对控制单元以及其总体的架构进行介绍。



控制单元的实现就与单周期有很大的差别了，也是本次 cpu 设计的难点对于多周期，指令是使用流水线执行的。而流水线有五个状态：IP、ID、EXE、MEM、WB。不同的指令所经过的阶段数目是不同的，这些从多周期 CPU 状态转移阁就可以看出。现如今，ControlUnit 可以分为三个部分： D 触发器、状态转移、信号输出。ControlUnit 作为这三

个部分的顶层模块，将这三部分连接成一个统一的整体。所以 ControlUnit 的实现类似于项层模块，只需要调用这三个模块。现在来谈谈 ControlUnit 三个部分 Dflipflop、NextState、OutputFunc 的实现:

- Dflipflop

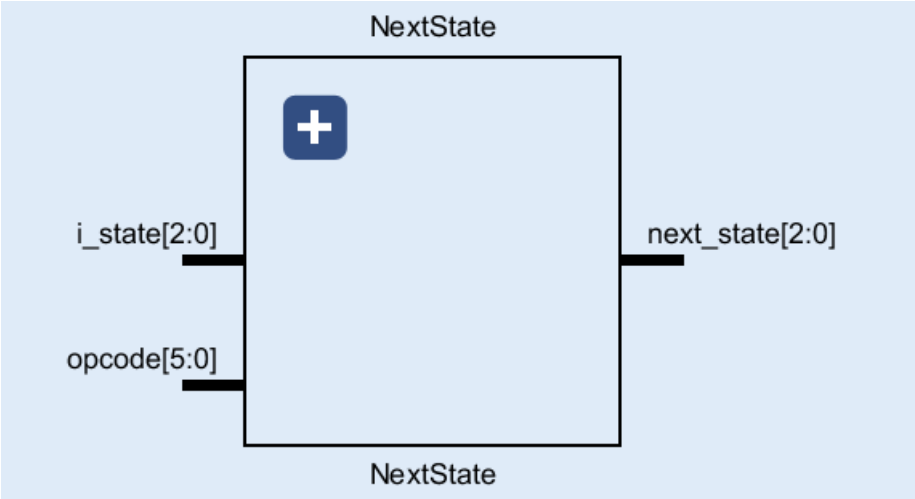


触发器的作用在于触发状态的改变。它的输入来自于 NextState 的输出。因为一个阶段的时间是一个时钟周期，当时钟上升沿到来时进行状态切换,即将触发器的输入作为输出。如果 RST 信号为 1,则将输出状态重置为 IP 阶段。

```
1.  `timescale 1ns / 1ps
2.  // D 触发器的实现
3.  // @param i_state 输入的状态，也就是下一个状态
4.  // @param reset 重置信号
5.  // @param clk 时钟信号
6.  // @param o_state 输出的状态
7.  // D 触发器，上升沿时，如果 RST 信号为 1，则输出状态重置为 000，对应 IF
8.  module DFlipFlop(i_state, reset, clk, o_state);
9.      input [2:0]i_state;
10.     input reset, clk;
```

```
    output reg[2:0]o_state;
    always @(posedge clk) begin
        if (reset) o_state = 3'b000;
        else o_state = i_state;
    end
endmodule
```

- NextState



首先为每个状态定义一个参数，参数的值为对应的二进制标识。

而且，对于 EXE 状态，由于有三条支路，故定义了 aEXE, bEXE, cEXE;

对于 WB 状态，由于有两条支路，故定义了 aWB，cWB。

当输入 NextState 的状态码与操作码发生改变时，使用 case 产生下一状态，对 case 的设计如下：

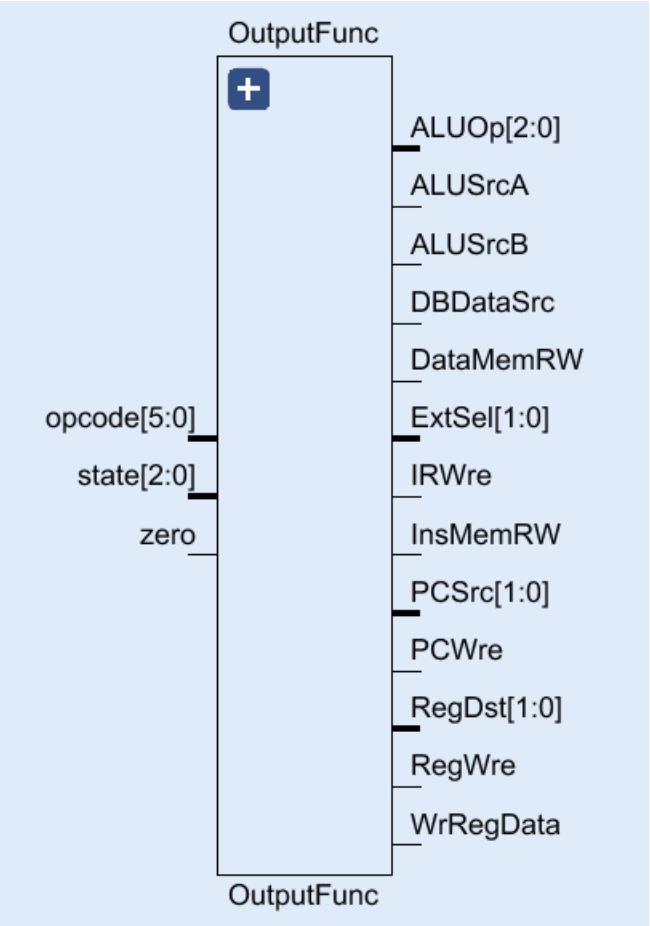
1. 由于所有指令都需要经过 IF-ID 阶段，所以 IF 的 NextState 显然就是 ID;
2. 对于 ID 阶段，有四条分叉的支路，对于 j, jal, jr, halt 指令，则 NextState 显然为 IF;其它的 NextState 是 EXE。现在分析 NextState 为 EXE 的指令：当 sw、lw 指令时，NextState 为 cEXE;当 beqg 指令时，NextState 为 bEXE;剩余的为 aEXE;

3. 对于 aEXE， NextState 自然为 aTB；对于 bEXE， NextState 自然为 IP;对于 cEXE，NextState 自然为 MEM(状态转移图)；
4. 对于 MEM，如果是 lw 指令，那么 NextState 为 cTB；如果是 aw 指令，那么 NextState 为 IF 指令；
5. aWB 与 cWB 的 NextState 都为 IP。

```
`timescale 1ns / 1ps
// NextState 模块的实现
// @param i_state 输入的状态
// @param opcode 输入的操作码
// @param next_state 下一状态
module NextState(i_state, opcode, next_state);
    input [2:0]i_state;
    input [5:0]opcode;
    output reg[2:0]next_state;
    parameter [2:0] IF = 3'b000, // IF 状态
                    ID = 3'b001, // ID 状态
                    aEXE = 3'b110, // 第一条分支的 EXE 状态
                    bEXE = 3'b101, // 第二条分支的 EXE 状态
                    cEXE = 3'b010, // 第三条分支的 EXE 状态
                    MEM = 3'b011, // MEM 状态
                    aWB = 3'b111, // 第一个分支的 WB 状态
                    cWB = 3'b100; // 第三个分支的 WB 状态

    always @(i_state or opcode) begin
        case (i_state)
        //所有指令都经过 IF-ID
            IF: next_state = ID;
        //四条分叉，其中 EXE 有三种
            ID: begin
                case (opcode[5:3])
                    3'b110: begin
                        if (opcode == 6'b110100) next_state = bEXE; // beq 指令
                        else next_state = cEXE; // sw, lw 指令
                    end
                    3'b111: next_state = IF; // j, jal, jr, halt 指令

                    default: next_state = aEXE; // add, sub 等指令
                endcase
```

```
            end
        //只有一条路
            aEXE: next_state = aWB;
            bEXE: next_state = IF;
            cEXE: next_state = MEM;
         //
            MEM: begin
                if (opcode == 6'b110001) next_state = cWB; // lw 指令
                else next_state = IF; // sw 指令
            end
         //两种写回都是 IF 结尾
            aWB: next_state = IF;
            cWB: next_state = IF;
            default: next_state = IF;
        endcase
    end
endmodule
```

- OutputFunc



输出模块的大致思路类似于单周期 ControlUnit 的实现,只不过在多

周期中是通过状态来定义控制信号的值。我定义了两个参数列表，一个用于定义各个状态，一个用于定义各种指令。这样的话在实现的时候就没有必要去查 ALU 操作的 opcode 了。根据控制信号作用表来定义各个控制信号。在最后，需要在 IF 阶段将 RegWre 与 DataMEMRW 置为 0，防止在 IF 阶段写数据。

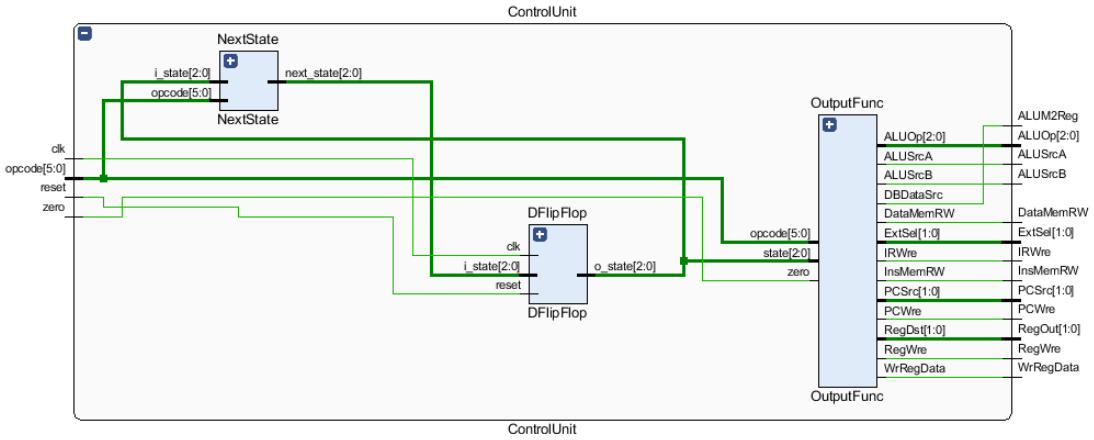
```
1.  `timescale 1ns / 1ps
2.  // 输出函数模块的实现
3.  // @param state 当前状态
4.  // @param opcode 操作码
5.  // @param PCWre PC 的控制信号
6.  // @param InsMemRW 指令存储器的控制信号
7.  // @param IRWre IR 的控制信号
8.  // @param WrRegData 控制寄存器组写数据端口的数据选择器
9.  // @param RegWre 寄存器组的控制信号
10. // @param ALUSrcA 控制 ALU 的 A 输入端口的数据选择器
11. // @param ALUSrcB 控制 ALU 的 B 输入端口的数据选择器
12. // @param DataMemRW 数据存储器的控制信号
13. // @param DBDataSrc 控制数据存储器输出端口的数据选择器
14. // @param ExtSel 符号扩展单元的控制信号
15. // @param RegDst 控制寄存器组写寄存器端口的数据选择器
16. // @param PCSrc 四选一选择器的控制信号
17. // @param ALUOp ALU 的控制信号
18. module OutputFunc(state, opcode, zero, PCWre, InsMemRW, IRWre, WrRegData, Re
    gWre, ALUSrcA, ALUSrcB, DataMemRW, DBDataSrc, ExtSel, RegDst, PCSrc, ALUOp);

19.     input [2:0]state;
20.     input [5:0]opcode;
21.     input zero;
22.     output reg PCWre, InsMemRW, IRWre, WrRegData, RegWre, ALUSrcA, ALUSrcB,
    DataMemRW, DBDataSrc;
23.     output reg[1:0]ExtSel, RegDst, PCSrc;
24.     output reg[2:0]ALUOp;
25.     parameter [2:0] IF = 3'b000, // IF 状态
26.                     ID = 3'b001, // ID 状态
27.                     aEXE = 3'b110, // 第一支路的 EXE 状态
28.                     bEXE = 3'b101, // 第二支路的 EXE 状态
29.                     cEXE = 3'b010, // 第三支路的 EXE 状态
30.                     MEM = 3'b011, // MEM 状态
```

```
31.                     aWB = 3'b111, // 第一支路的 WB 状态
32.                     cWB = 3'b100; // 第三支路的 WB 状态
33.     parameter [5:0] addi = 6'b000010,
34.                     ori = 6'b010010,
35.                     sll = 6'b011000,
36.                     add = 6'b000000,
37.                     sub = 6'b000001,
38.                     slt = 6'b100110,
39.                     slti = 6'b100111,
40.                     sw = 6'b110000,
41.                     lw = 6'b110001,
42.                     beq = 6'b110100,
43.                     j = 6'b111000,
44.                     jr = 6'b111001,
45.                     Or = 6'b010000,
46.                     And = 6'b010001,
47.                     jal = 6'b111010,
48.                     halt = 6'b111111;
49.
50.     always @(state) begin
51.         // 对 PCWre 定值
52.         if (state == IF && opcode != halt) PCWre = 1;
53.         else PCWre = 0;
54.         // 对 InsMemRW 定值
55.         InsMemRW = 1;
56.         // 对 IRWre 定值
57.         if (state == IF) IRWre = 1;
58.         else IRWre = 0;
59.         // 对 WrRegData 定值
60.         if (state == aWB || state == cWB) WrRegData = 1;
61.         else WrRegData = 0;
62.         // 对 RegWre 定值
63.         if (state == aWB || state == cWB || opcode == jal) RegWre = 1;
64.         else RegWre = 0;
65.         // 对 ALUSrcA 定值
66.         if (opcode == sll) ALUSrcA = 1;
67.         else ALUSrcA = 0;
68.         // 对 ALUSrcB 定值
69.         if (opcode == addi || opcode == ori || opcode == slti|| opcode == sw
    || opcode == lw) ALUSrcB = 1;
70.         else ALUSrcB = 0;
71.         // 对 DataMemRW 定值
72.         if (state == MEM && opcode == sw) DataMemRW = 1;
73.         else DataMemRW = 0;
```

```
74.         // 对 DBDataSrc 定值
75.         if (state == cWB) DBDataSrc = 1;
76.         else DBDataSrc = 0;
77.         // 对 ExtSel 定值
78.         if (opcode == ori) ExtSel = 2'b01;
79.         else if (opcode == sll) ExtSel = 2'b00;
80.         else ExtSel = 2'b10;
81.         // 对 RegDst 定值
82.         if (opcode == jal) RegDst = 2'b00;
83.         else if (opcode == addi || opcode == ori || opcode == lw) RegDst = 2
    'b01;
84.         else RegDst = 2'b10;
85.         // 对 PCSrc 定值
86.         case(opcode)
87.             j: PCSrc = 2'b11;
88.             jal: PCSrc = 2'b11;
89.             jr: PCSrc = 2'b10;
90.             beq: begin
91.                 if (zero) PCSrc = 2'b01;
92.                 else PCSrc = 2'b00;
93.             end
94.             default: PCSrc = 2'b00;
95.         endcase
96.
97.         // 对 ALUOp 定值
98.         case(opcode)
99.             sub: ALUOp = 3'b001;
100.             Or: ALUOp = 3'b101;
101.             And: ALUOp = 3'b110;
102.             ori: ALUOp = 3'b101;
103.             slt: ALUOp = 3'b010;
104.             slti: ALUOp = 3'b010;
105.             sll: ALUOp = 3'b100;
106.             beq: ALUOp = 3'b001;
107.             default: ALUOp = 3'b000;
108.         endcase
109.
110.         // 防止在 IF 阶段写数据
111.         if (state == IF) begin
112.             RegWre = 0;
113.             DataMemRW = 0;
114.         end
115.     end
116.
```

```
117. endmodule
```

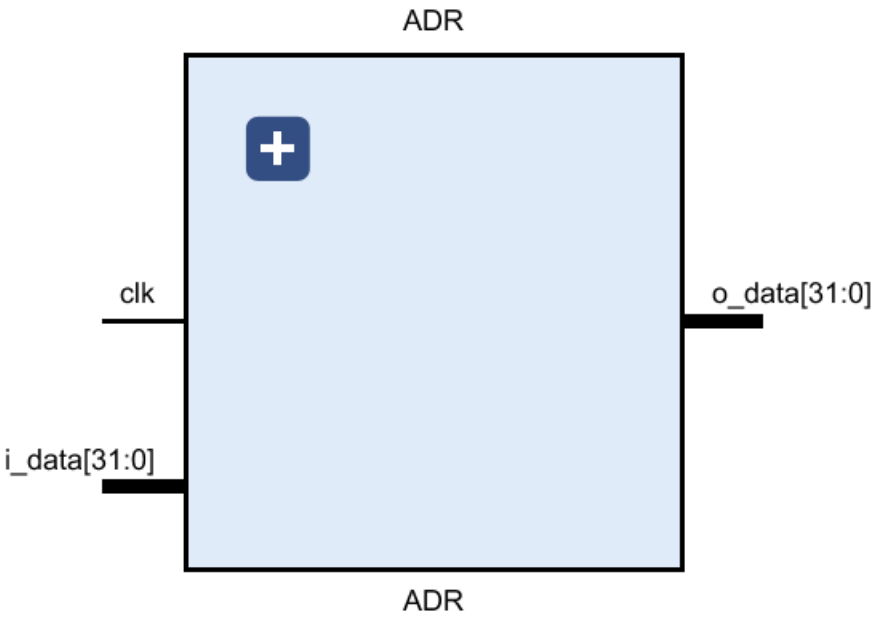
然后通过 ControlUnit 将它们连结起来：




```
1. `timescale 1ns / 1ps
2. // 控制单元 CU 的实现
3. // @param opcode 操作码
4. // @param zero 输入的 zero 信号
5. // @param clk 时钟信号
6. // @param reset 重置信号
7. // @param PCWre, InsMemRW, IRWre, WrRegData, RegWre, ALUSrcB, DataMemRW, ALUM2Reg，ExtSel, RegOut, PCSrc，ALUOp 控制信号
8. module ControlUnit(opcode, clk, reset, zero, PCWre, InsMemRW, IRWre, WrRegData, RegWre, ALUSrcA, ALUSrcB, DataMemRW, ALUM2Reg, ExtSel, RegOut, PCSrc, ALUOp);
9.     input [5:0]opcode;
10.    input zero, clk, reset;
11.    output PCWre, InsMemRW, IRWre, WrRegData, RegWre,ALUSrcA, ALUSrcB, DataMemRW, ALUM2Reg;
12.    output [1:0]ExtSel, RegOut, PCSrc;
13.    output [2:0]ALUOp;
14.
15.    wire [2:0]i_state, o_state;
16.
17.    DFlipFlop DFlipFlop(i_state, reset, clk, o_state);
18.    NextState NextState(o_state, opcode, i_state);
19.    OutputFunc OutputFunc(o_state, opcode, zero, PCWre, InsMemRW, IRWre, WrRegData, RegWre, ALUSrcA, ALUSrcB, DataMemRW, ALUM2Reg, ExtSel, RegOut, PCSrc, ALUOp);
20.
21. endmodule
```

3.7 四个分段寄存器

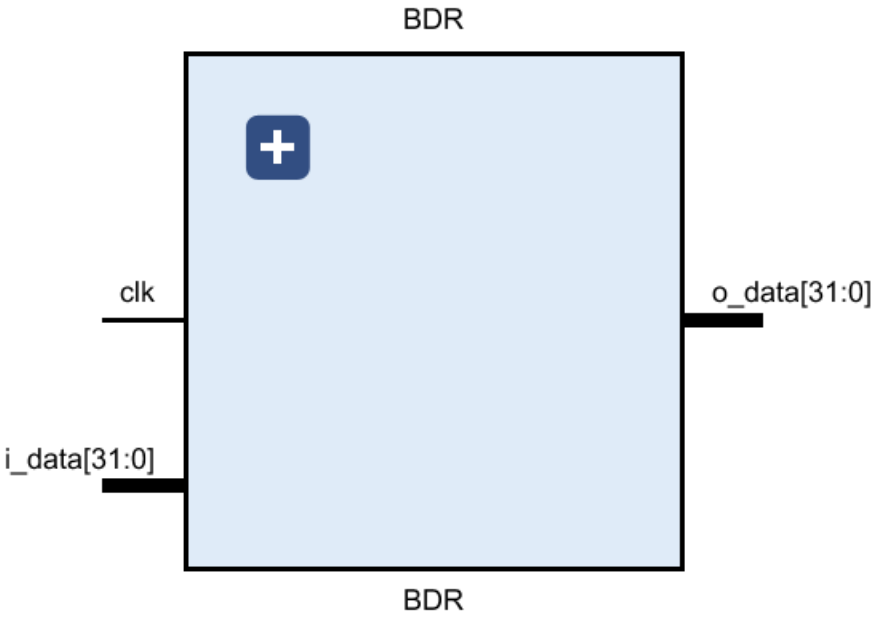
- ADR



临时存储的寄存器 ADR，目的是将输入寄存器的值在一定条件下进行输出（在这里就是在时钟信号的触发下进行输出），具体来说，ADR 是将从寄存器堆输出端口 1 读取的数据在一定条件下输出到加法器输入端口 A 的寄存器。

```
`timescale 1ns / 1ps
// 切割数据通路
// @param i_data 输入的数据
// @param o_data 输出的数据
// @param clk 时钟信号
module ADR(i_data, clk, o_data);
  input clk;
  input [31:0] i_data;
  output reg[31:0] o_data;
  always @(posedge clk) begin
    o_data = i_data;
  end
endmodule
```

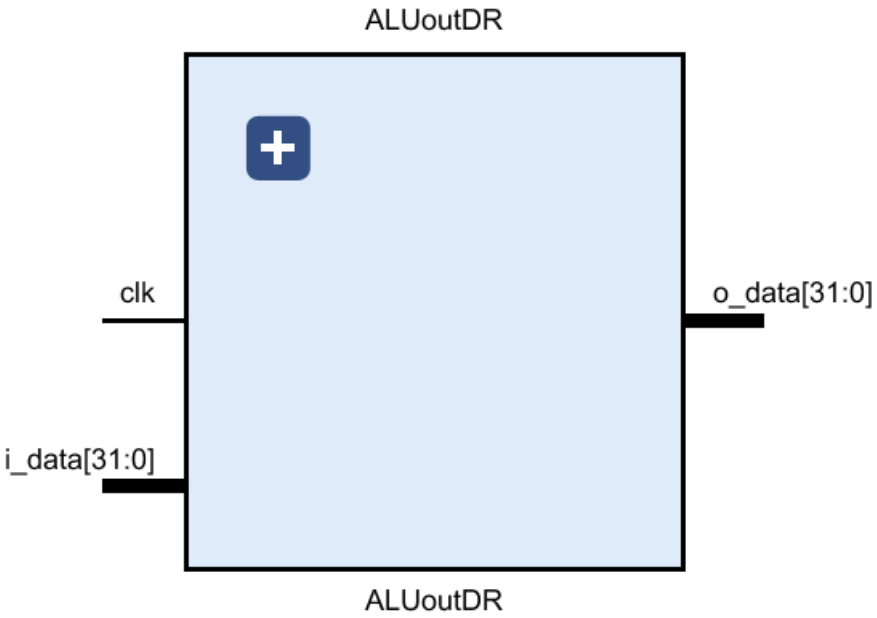
- BDR



临时存储的寄存器 BDR，目的是将输入寄存器的值在一定条件下进行输出（在这里就是在时钟信号的触发下进行输出），具体来说，BDR 是将从寄存器堆输出端口 2 读取的数据在一定条件下输出到加法器输入端口 B 的寄存器。

```
1.  `timescale 1ns / 1ps
2.  // 切割数据通路
3.  // @param i_data 输入的数据
4.  // @param o_data 输出的数据
5.  // @param clk 时钟信号
6.  module BDR(i_data, clk, o_data);
7.    input clk;
8.    input [31:0] i_data;
9.    output reg[31:0] o_data;
10.   always @(posedge clk) begin
11.     o_data = i_data;
12.   end
13. endmodule
```

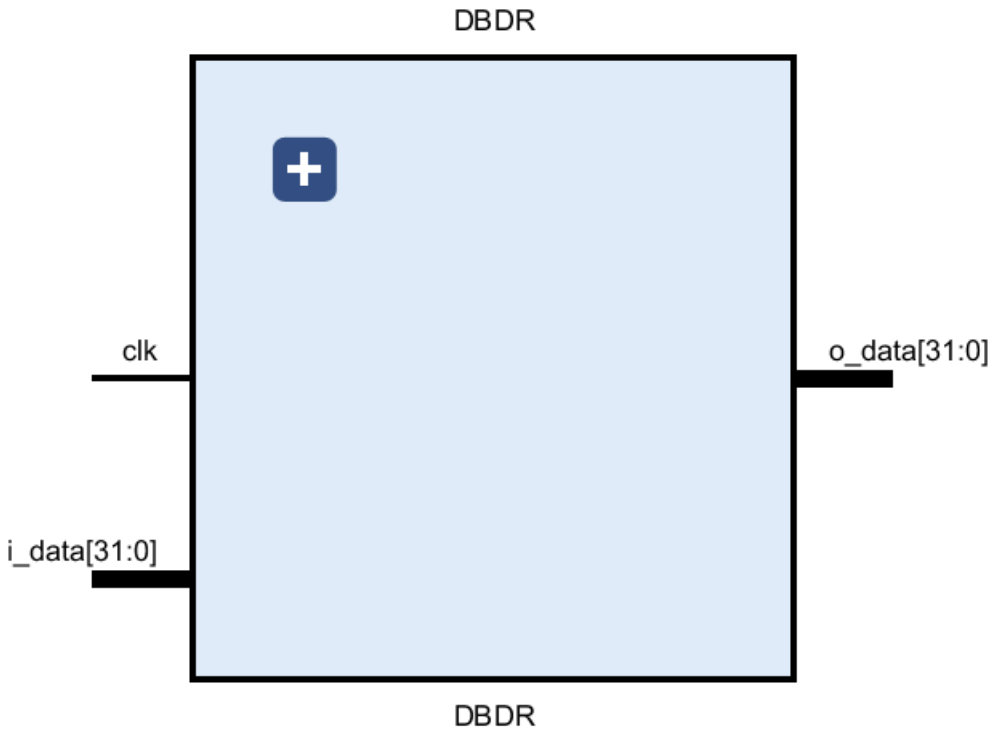
- ALUoutDR



临时存储的寄存器 ALUoutDR，目的是将输入寄存器的值在一定条件下进行输出（在这里就是在时钟信号的触发下进行输出），具体来说，ALUoutDR 是将从加法器结果输出端口读取的地址在一定条件下输出到数据存储器写地址端口的寄存器。

```
1.  `timescale 1ns / 1ps
2.  // 切割数据通路
3.  // @param i_data 输入的数据
4.  // @param o_data 输出的数据
5.  // @param clk 时钟信号
6.  module ALUoutDR(i_data, clk, o_data);
7.    input clk;
8.    input [31:0] i_data;
9.    output reg[31:0] o_data;
10.   //上升沿触发
11.   always @(posedge clk) begin
12.     o_data = i_data;
13.   end
14. endmodule
```

- DBDR

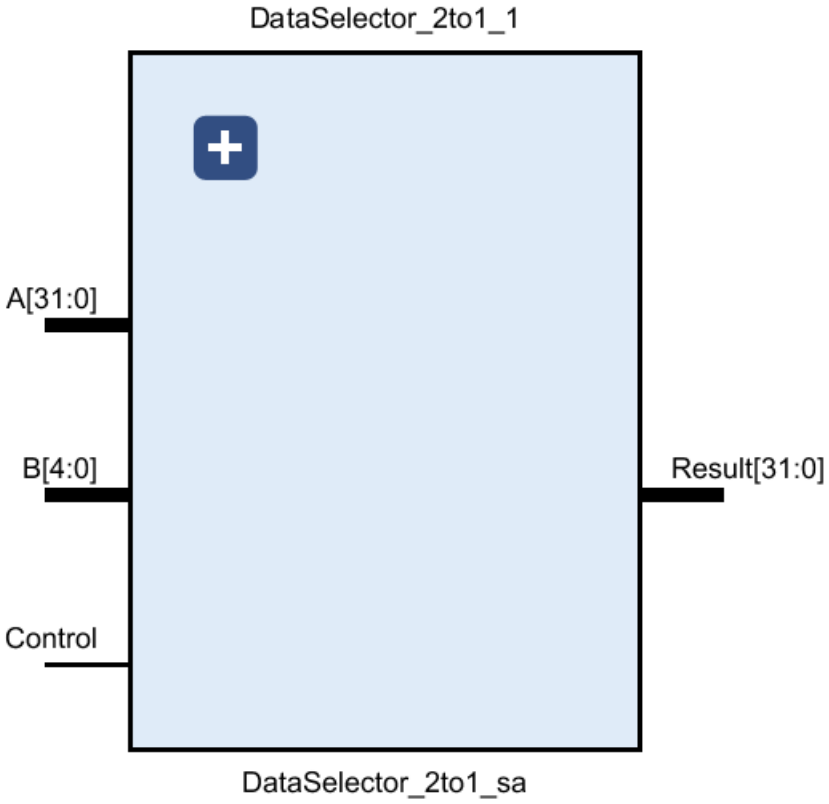


临时存储的寄存器 DBDR，目的是将输入寄存器的值在一定条件下进行输出（在这里就是在时钟信号的触发下进行输出），具体来说，DBDR 是将从数据存储器输出端口读取的数据在一定条件下输出到寄存器堆写数据端口前的二选一选择器的 1 端口的寄存器。

```
1.  `timescale 1ns / 1ps
2.  // 切割数据通路
3.  // @param i_data 输入的数据
4.  // @param o_data 输出的数据
5.  // @param clk 时钟信号
6.  module DBDR(i_data, clk, o_data);
7.    input clk;
8.    input [31:0] i_data;
9.    output reg[31:0] o_data;
10.   always @(posedge clk) begin
11.     o_data = i_data;
12.   end
13. endmodule
```

3.8 三种多路选择器

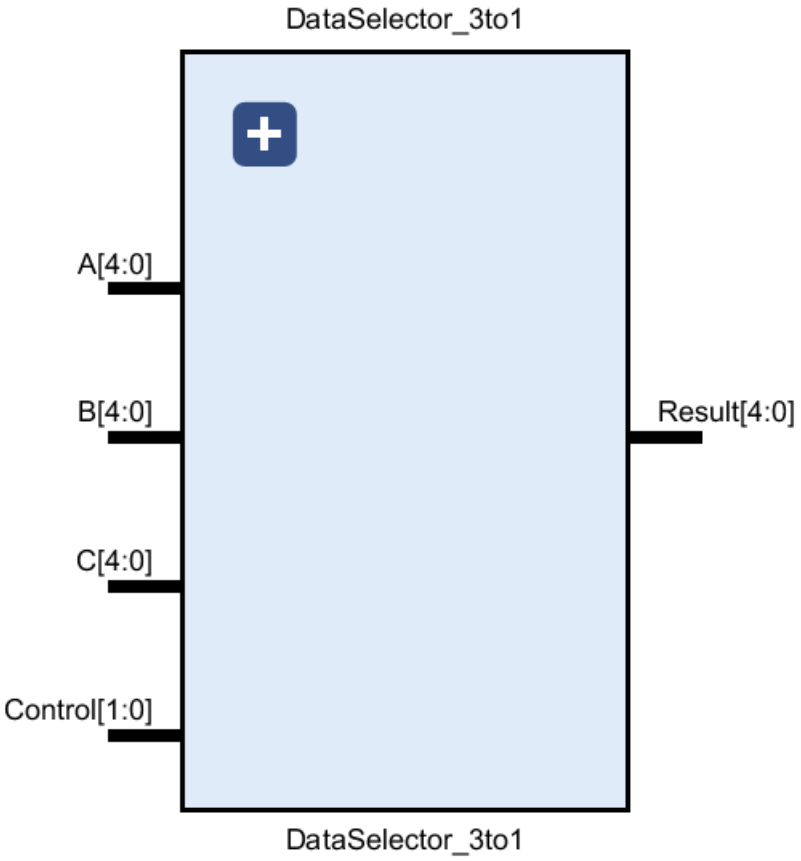
- 二选一多路选择器



选择器的实现其实很简单，就是根据控制信号选择某个输入作为输出。对于二选一选择器，则定义其控制信号为一位，即 0，1。在每次输入、时钟信号或者是控制信号变化时，使用 case 语句选择某个输入输出。

```
1.  `timescale 1ns / 1ps
2.  // 二选一数据选择器实现
3.  // @param A 输入 1
4.  // @param B 输入 2
5.  // @param Control 选择器的控制信号
6.  // @param Result 结果
7.  module DataSelector_2to1(A, B, Control, Result);
8.    input [31:0] A, B;
9.    input Control;
10.   output [31:0] Result;
11.   assign Result = (Control == 1'b0 ? A : B);
12. endmodule
```

- 三选一多路选择器

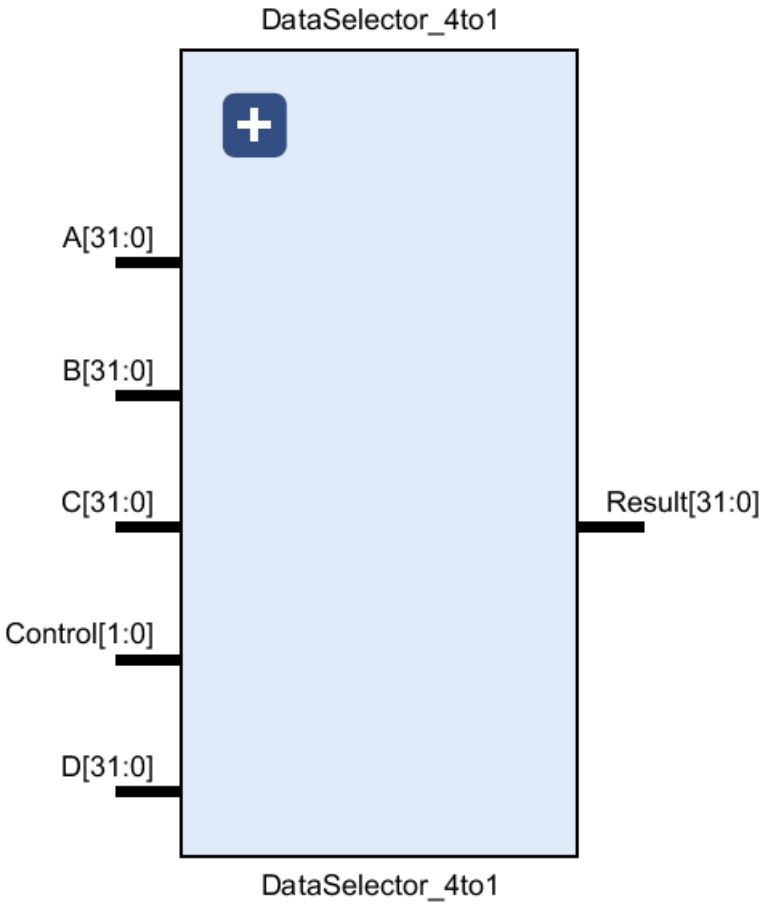


对于三选一选择器，则定义其控制信号为两位，即 00，01，10,对于没有使用的 11,则将其输出定义为 0。这样做的原因在于:在数据存储器种，我限制了对 0 号寄存器的写操作，所以这样做并不会在寄存器上写值。在每次输入、时钟信号、控制信号变化时，使用 case 语句选择某个输入输出。。

```
1.  `timescale 1ns / 1ps
2.  // 三选一数据选择器的实现
3.  // @param A 输入 1
4.  // @param B 输入 2
5.  // @param C 输入 3
6.  // @param Control 选择器的控制信号
7.  // @param Result 选择的结果
8.  module DataSelector_3to1(A, B, C, Control, Result);
9.    input [4:0] A, B, C;
10.   input [1:0] Control;
11.   output reg[4:0] Result;
```

```
12.   always @(Control or A or B or C) begin
13.     case(Control)
14.         2'b00:Result = A;
15.         2'b01:Result = B;
16.         2'b10:Result = C;
17.         default: Result = 0;
18.      endcase
19.   end
20. endmodule
```

- 四选一多路选择器



对于四选一选择器，则定义其控制信号为两位，即 00，01，10，11，在每次输入、时钟信号、控制信号变化时，使用 case 语句选择某个输入输出。

```
1.  `timescale 1ns / 1ps
2.  // 四选一数据选择器的实现
3.  // @param A 输入 1
4.  // @param B 输入 2
```

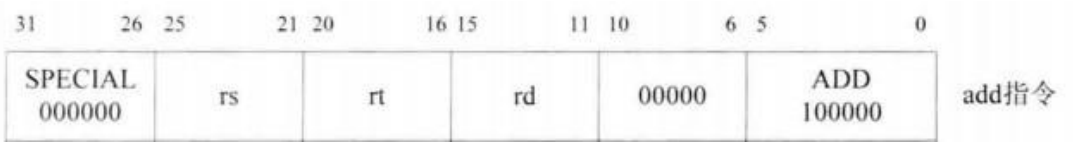
```
// @param C 输入 3
// @param D 输入 4
// @param Control 数据选择器的控制信号
// @param Result 选择的结果
module DataSelector_4to1(A, B, C, D, Control, Result);
  input [31:0] A, B, C, D;
  input [1:0]Control;
  output reg[31:0] Result;
  always @(Control or A or B or C or D) begin
    case(Control)
        2'b00: Result = A;
        2'b01: Result = B;
        2'b10: Result = C;
        2'b11: Result = D;
        default: Result = 0;
     endcase
  end
endmodule
```

04 支持的 16 条指令

4.1 支持的 16 条指令

- 算数运算指令

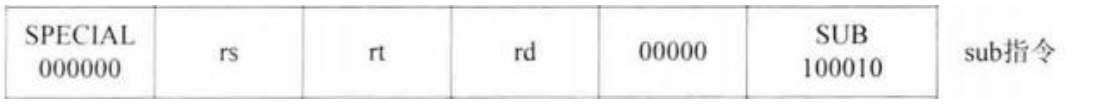
1. 当功能码是 6b100000 时，表示 add 指令，加法运算。



指令用法为：add rd，rs，rt。

指令作用为：rd ← rs+rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行加法运算，结果保存到地址为 rd 的通用寄存器中。但是有一种特殊情况：如果加法运算溢出，那么会产生溢出异常，同时不保存结果。

2. 当功能码是 6b100010 时，表示 sub 指令，减法运算。



指令用法为：sub rd，rs，rt。

指令作用为：rd ← rs-rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行减法运算，结果保存到地址为 rd 的通用寄存器中。但是有一种特殊情况：如果减法运算溢出，那么产生溢出异常，同时不保存结果。

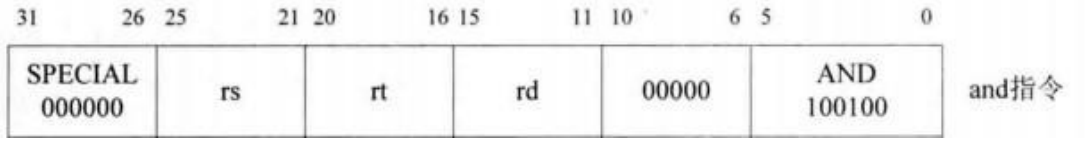
3. 当指令码是 6b001000 时，表示 addi 指令，加法运算。

指令用法为：addi rt，rs，immediate。

指令作用为：rt ← rs+（sign_extended）immediate，将指令中的 16 位立即数进行符号扩展，与地址为 rs 的通用寄存器的值进行加法运算，结果保存到地址为 rt 的通用寄存器中。但是有一个特殊情况：如果加法运算溢出，那么产生溢出异常，同时不保存结果。

- 逻辑运算指令

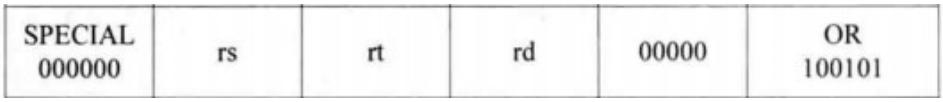
4. 当功能码是 6b100100 时，表示是 and 指令，逻辑“与”运算。



指令用法为：and　rd，rs，rt。

指令作用为：rd ← rs AND rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行逻辑“与”运算，运算结果保存到地址为 rd 的通用寄存器中。

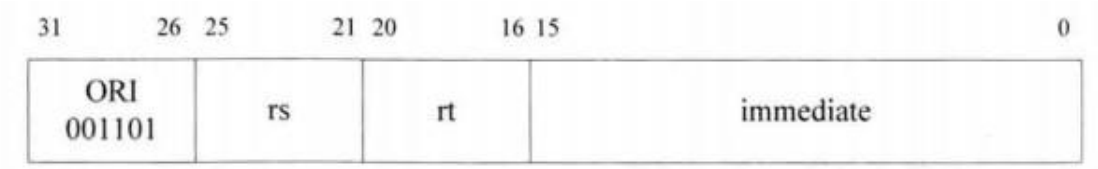
5. 当功能码是 6b100101 时，表示是 or 指令，逻辑“或”运算。



指令用法为：or　rd，rs，rt。

指令作用为：rd ← rs OR rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行逻辑“或”运算，运算结果保存到地址为 rd 的通用寄存器中。

6. 当指令码是 6b001101，表示是 ori 指令，或运算。

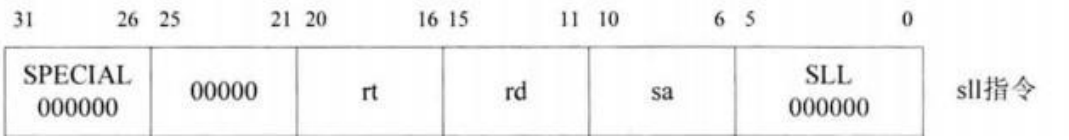


指令用法为：ori rs，rt，immediate。

作用是将指令中的 16 位立即数进行无符号扩展至 32 位，然后与索引为 rs 的通用寄存器的值进行逻辑“或”运算，运算结果保存到索引为 rt 的通用寄存器中。

- 移位指令

7. 当功能码是 6b000000，表示是 sll 指令，逻辑左移。

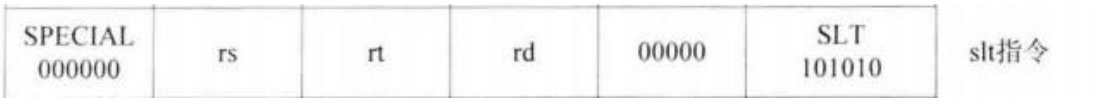


指令用法为：sll rd，rt，sa。

指令作用为：rd ← rt << sa（logic），将地址为 rt 的通用寄存器的值向左移 sa 位，空出来的位置使用 0 填充，结果保存到地址为 rd 的通用寄存器中。

- 比较指令

8. 当功能码是 6b101010 时，表示 slt 指令，比较运算。



指令用法为：slt rd，rs，rt。

指令作用为：rd ←（rs<rt），将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值按照有符号数进行比较，如果前者小于后者，那么将 1 保存到地址为 rd 的通用寄存器中；反之，将 0 保存到地址为 rd 的通用寄存器中。

9. 当指令码是 6b001010 时，表示 slti 指令，比较运算。

指令用法为：slti rt，rs，immediate。

指令作用为：rt ←（rs <（sign_extended）immediate），将指令中的 16 位立即数进行符号扩展，与地址为 rs 的通用寄存器的值按照有符号数进行比较，如果前者大于后者，那么将 1 保存到地址为 rt 的通用寄存器中；反之，将 0 保存到地址为 rt 的通用寄存器中。

- 存储器读写指令

10. 当指令中的指令码为 6b101011 时，是 sw 指令，字存储指令。

指令用法为：sw rt，offset（base）。

指令作用为：将地址为 rt 的通用寄存器的值存储到内存中的指定地址。该指令有地址对齐要求，要求计算出来的存储地址的最低两位为 00。

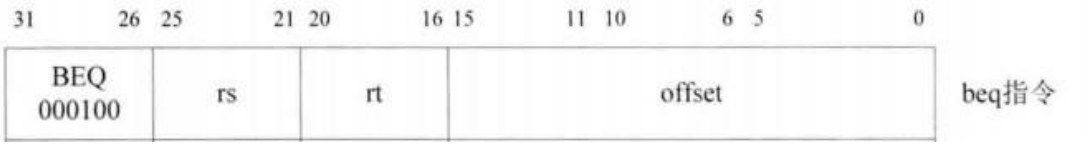
11. 当指令中的指令码为 6b100011 时，是 lw 指令，字加载指令。

指令用法为：lw rt，offset（base）。

指令作用为：从内存中指定的加载地址处，读取一个字，保存到地址为 rt 的通用寄存器中。该指令有地址对齐要求，要求加载地址的最低两位为 00。

- 分支指令

12. 当指令中的指令码为 6b000100 时，表示 beq 指令。

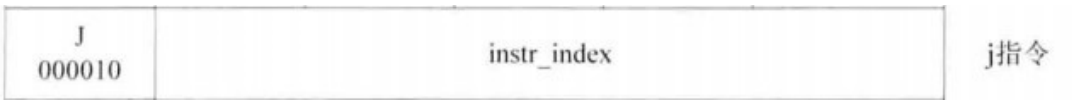


指令用法为：beq rs，rt，offset。

指令作用为：if rs=rt then branch，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行比较，如果相等，那么发生转移。

- 跳转指令

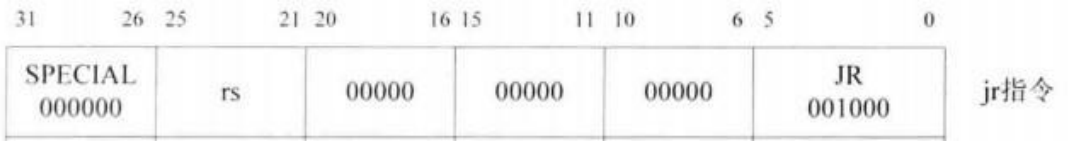
13. 当指令中的指令码为 6b000010 时，表示 j 指令。



指令用法为：j target。

指令作用为：pc ← （pc+4）[31，28]|| target || ‘00’，转移到新的指令地址，其中新指令地址的低 28 位是指令中的 target（也就是上图中的 instr_index）左移两位的值，新指令地址的高 4 位是跳转指令后面延迟槽指令的地址高 4 位。

14. **当指令中的指令码为 SPECIAL，功能码为 6b001000 时，表示 jr 指令。**

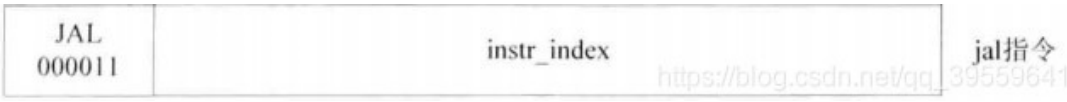


指令用法为：jr rs。

指令作用为：pc ← rs，将地址为 rs 的通用寄存器的值赋给寄存器 PC，作为新的指令地址。

- 调用子程序指令

15. **当指令中的指令码为 6b000011 时，表示 jal 指令。**



指令用法为：jal target。

指令作用为：pc ← （pc+4）[31，28]||target||‘00’，转移到新的指令地址，新指令地址与指令 j 相同，不再解释。但是，指令 jal 还要将跳转指令后面第 2 条指令的地址作为返回地址保存到寄存器$31。

- 停机指令

halt

111111	00000000000000000000000000(26 位)

4.2 指令存储器内的指令设计

为了在总指令条数尽可能少的情况下完成 16 条指令的运行，我对数据存储器内的指令做了如下设计:

指令情况如下：

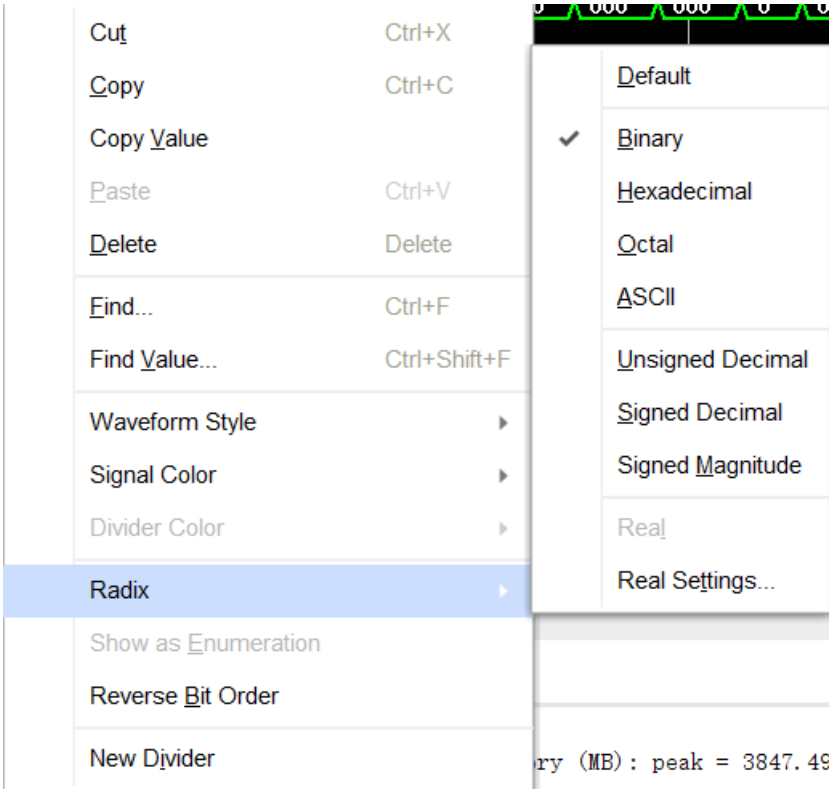
```
1.  j 2
2.  jr $31
3.  addi $1,$2,4
4.  addi $2,$0,8
5.  sw $2,0($2)
6.  add $3,$2,$1
7.  sub $3,$3,$1
8.  beq $2,$3,-2
9.  ori $1,$1,1
10. or $3,$2,$1
11. add $3,$2,$0
12. and $1,$3,$2
13. sll $1,$2,2
14. slt $6,$1,$2
15. slt $7,$2,$1
16. slti $6,$1,1
17. slti $7,$6,1
18. jal 1
19. lw $4,0($2)
20. halt
```

因为存在跳转，所以部分指令被执行多次，最终执行了 22 条指令

05 仿真波形与分析

为了方便对仿真结果进行分析，我将对 ins，memory 等值在适当的情况下进行二进制，十进制数的表示转换，这样能便于进行对比

运行，易于分析：



5.1 仿真代码

为了方便对实际仿真的情况更好的进行分析，此处附上仿真代码，很简单，各个部分的功能都注释了。

```
1.  `timescale 1ns / 1ps
2.
3.  module cpu_sim;
4.
5.  // Inputs
6.      reg CLK;
7.      reg RST;
8.      reg [31:0] outside_pc;
9.      // Outputs
10.     wire [31:0] ins, now_pc;
11.
12.     // Instantiate the Unit Under Test (UUT)
13.     Main uut (
14.         .CLK(CLK),
15.         .RST(RST),
```

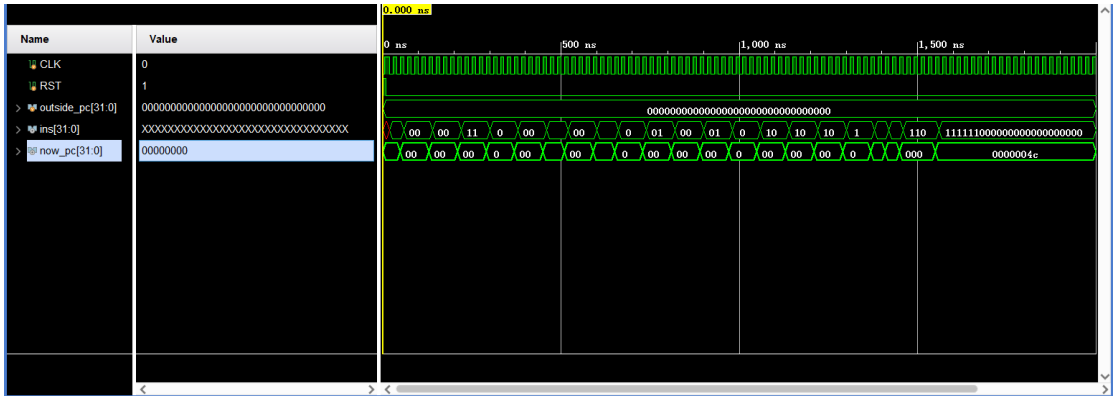
```
        .outside_pc(outside_pc),
        .ins(ins),
        .now_pc(now_pc)
    );

    initial
    begin
        // Initialize Inputs
      CLK = 0;
      RST = 1;
      outside_pc = 0; // 这里设置外部 pc
      #5; // 刚开始设置 pc 为 0
          CLK = !CLK;
      #5;
          RST = 0;

      forever #10 begin // 产生时钟信号
          CLK = !CLK;
      end


    end

    endmodule
```

由于设计仿真代码的时候是时钟信号是#10 一次,为了方便观察总体波形，我们再运行 1000ns 即可。

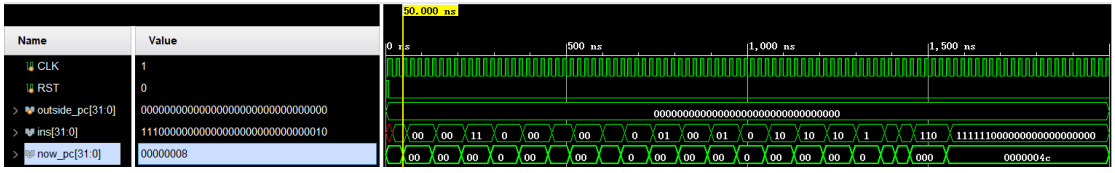
后面的部分涉及到 Vivado 的使用，由于上学期设计单周期 CPU 的时候已经使用过，现在使用起来就很得心应手了。

5.2 仿真波形分析

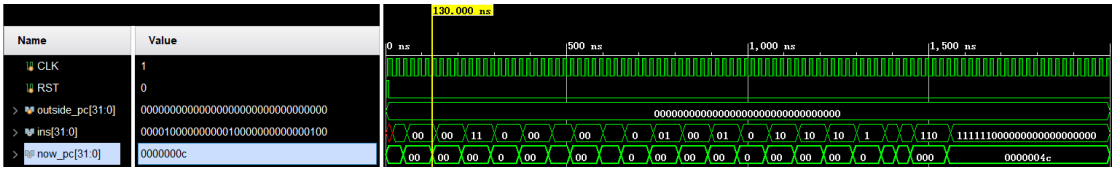
仿真情况与对应分析如下：



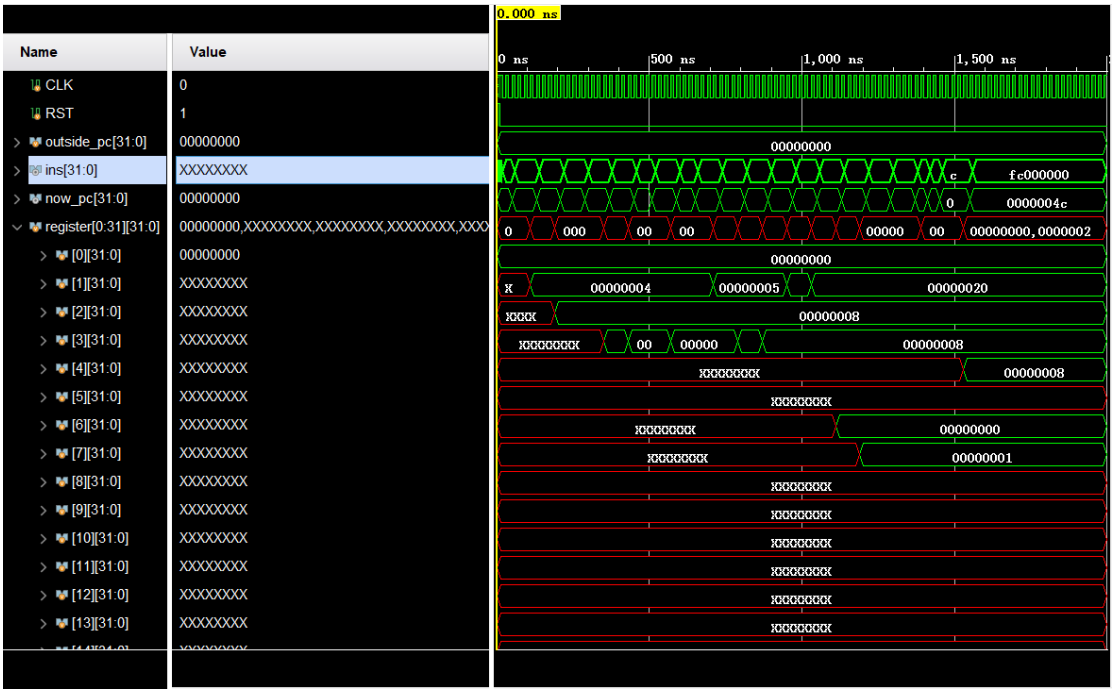
第一条指令被运行，此时 ins 为 11100000000000000000000000000010，对应第一条指令 j 2，



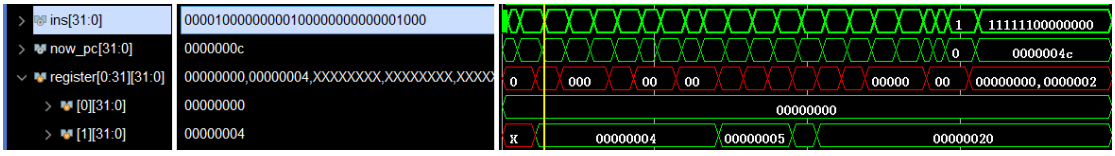
由于第一条指令是跳转指令，跳转到第三条指令（因为第一条是 0，0+2=2，对应第三条指令），所以第三条指令被运行，此时 ins 为 00001000000000001000000000000100，对应第三条指令 addi $1,$0,4



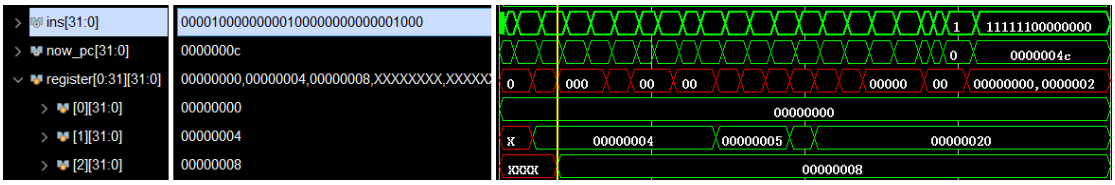
也就是将立即数 4 作位拓展之后与 rs$0 相加放入$1，为了方便我们观察寄存器值的变化，以此来检验指令的执行情况，我在 scope 里面将 regster 放入并刷新，得到了寄存器的变化波形



第二条指令，执行完毕后，一号寄存器的值变为了 4

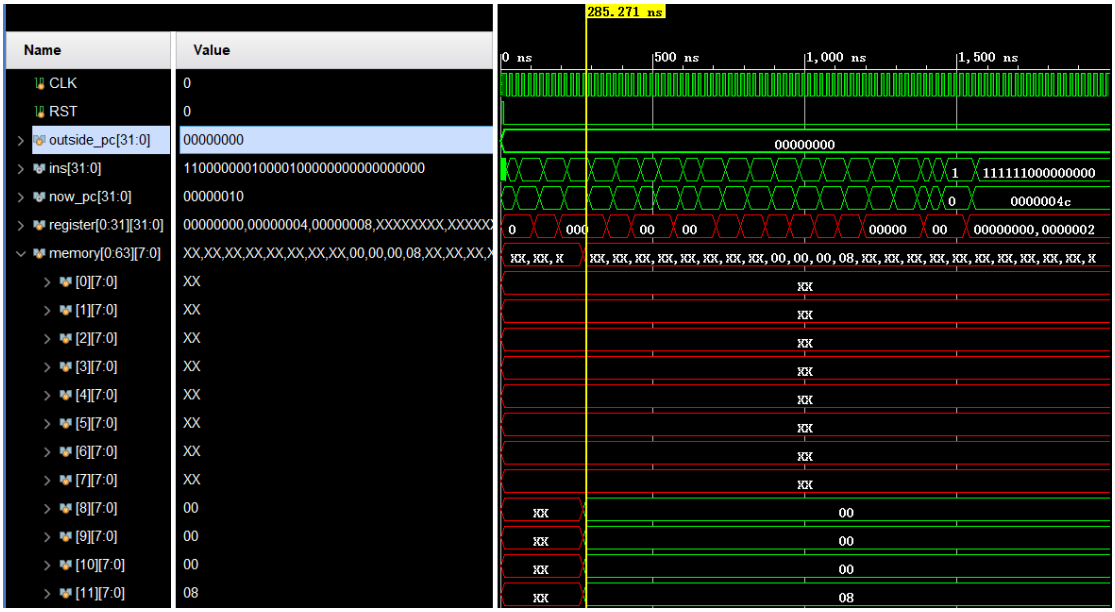


第三条指令 addi $2,$0,8,对应的 32 位二进制位 ins 为 00001000000000010000000000001000，可以看到，二号寄存器的值变为 8，



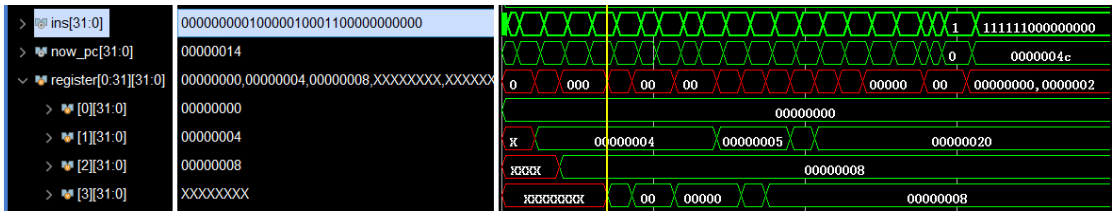
第四条指令 sw$2,0($2)，对应 32 位二进制数是 11000000010000100000000000000000 也就是将$2 中的内容和立即数 0 作符号拓展相加，并作为内存地址单元地址，读取该地址中的数，存入$2。

此处涉及到 mem，为了方便我们观察和比较，我将 memory 也加进来，刷新，观察其 value，果然，其内容变为 8($2)

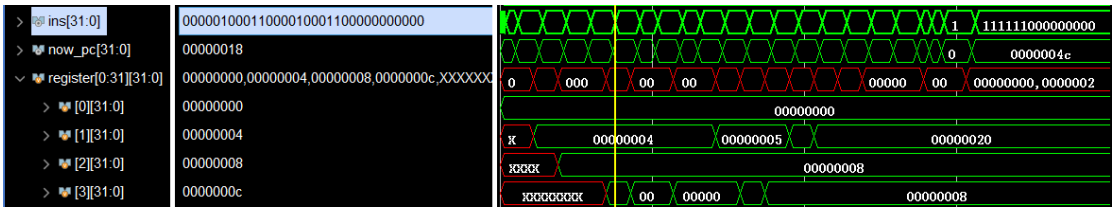


第五条指令，add $3,$2,$1，对应的二进制指令为

00000000010000010001100000000000

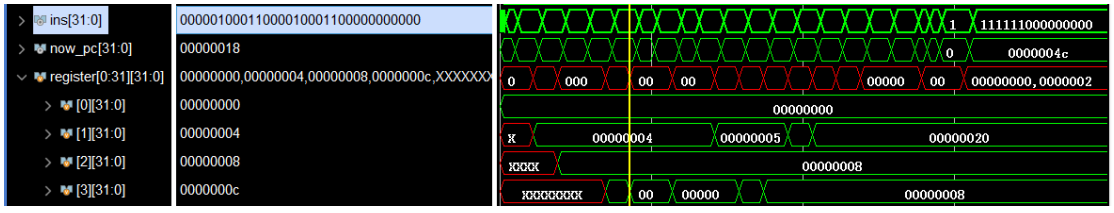


执行完成后，三号寄存器的值变为 12



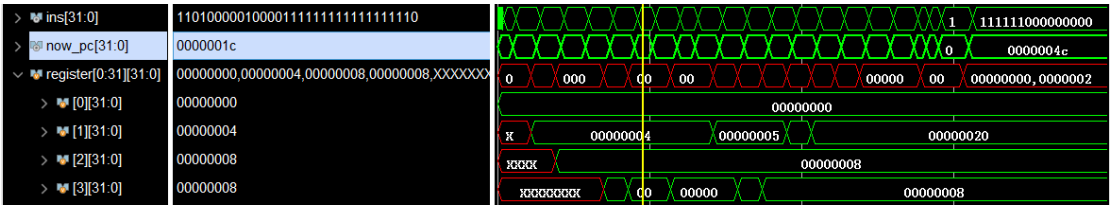
第六条指令，sub $3,$3,$1，对应的二进制指令为

00000100011000010001100000000000



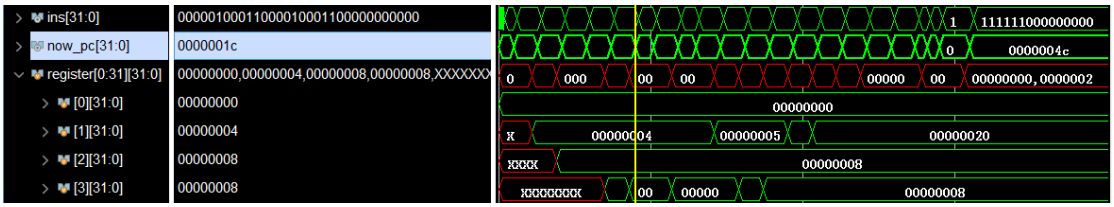
执行完成后，三号寄存器的值变为 8

第七条指令，beq $2,$3,-2，对应的二进制指令为11010000010000111111111111111110

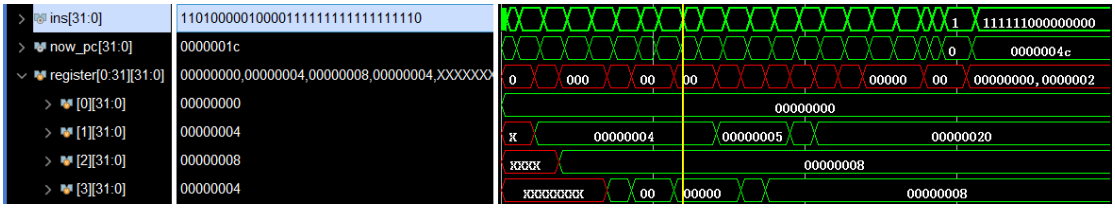


因为$2,$3 值相等，都为 8，所以跳转到指令 sub $3,$3,$1

第八条指令 sub $3,$3,$1，对应二进制为00000100011000010001100000000000

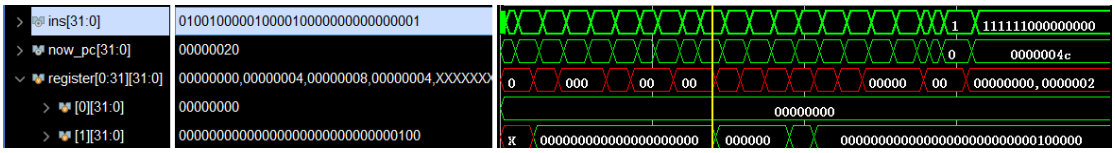


执行完毕后，三号寄存器的值变为 4

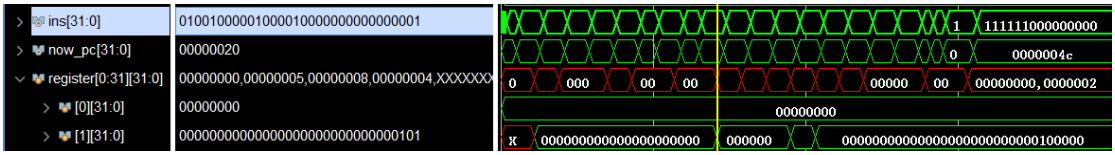


指令九，再次执行 beq $2,$3,-2，对应二进制为11010000010000111111111111111110，此时二号寄存器与三号寄存器的值不相等，所以不跳转，继续执行下一条指令 ori $1,$1,1。

指令十，ori $1,$1,1，对应的二进制是01001000001000010000000000000001，

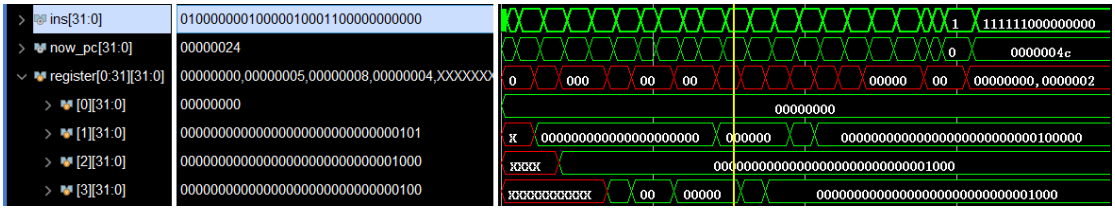


与 1 做完或运算后，得到 0100|0101->0101

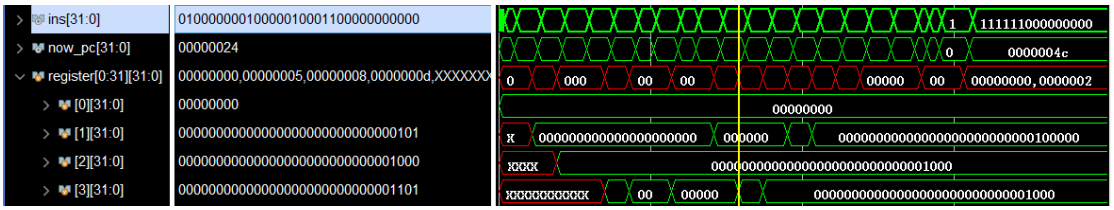


指令十一，是 or$3,$2,$1，对应的二进制是

01000000010000010001100000000000

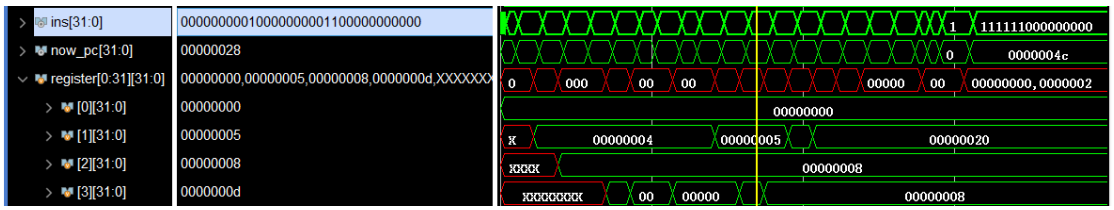


执行完毕后，三号寄存器值变为一号和二号寄存器值的或运算结果。

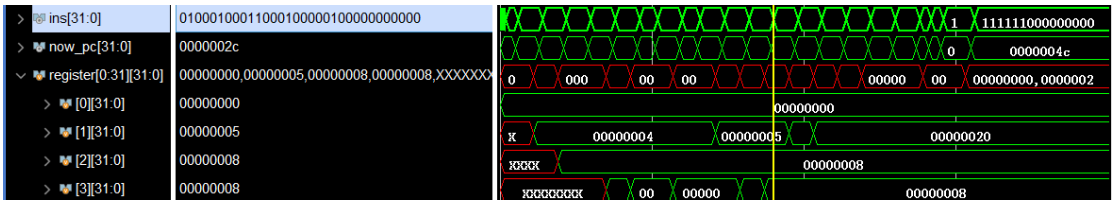


指令十二，add $3,$2,$0，对应二进制为

00000000010000000001100000000000，

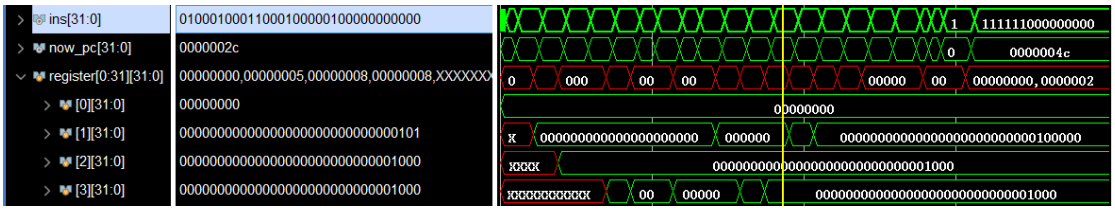


执行完毕后，

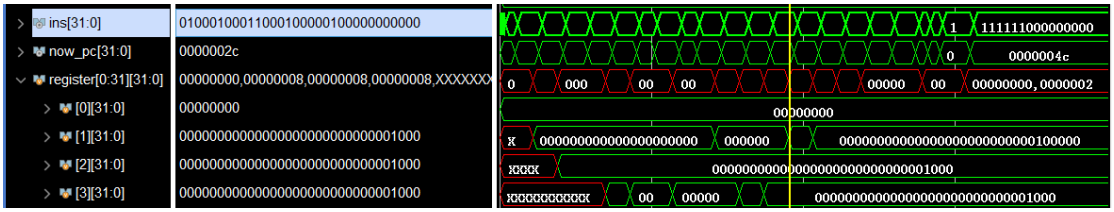


指令十三，and $1,$3,$2，对应的二进制为

01000100011000100000100000000000

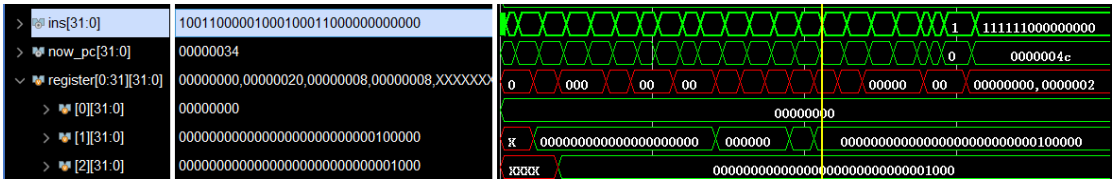


执行完毕后，

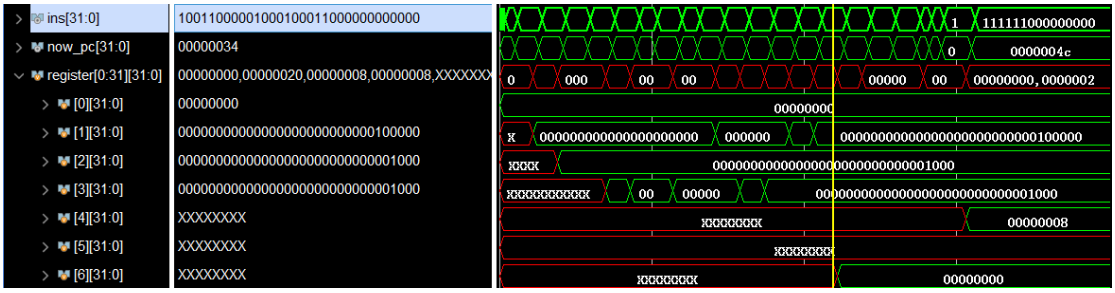


指令十四，sll $1,$2,2，对应二进制为01100000000000100000100010000000，

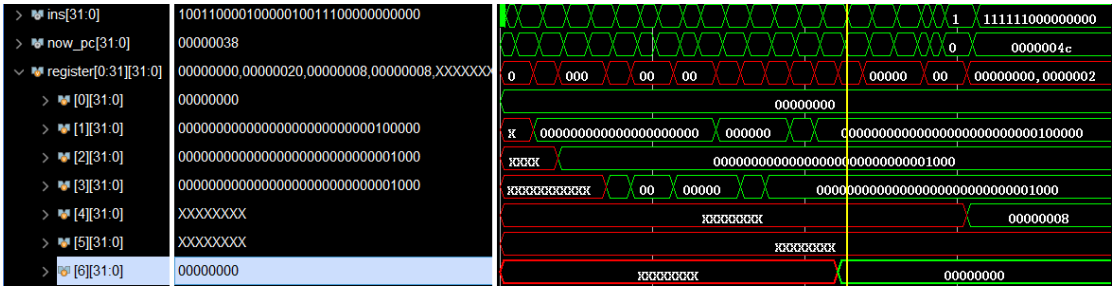
将二号寄存器值左移 2 位，值给一号寄存器，执行完毕后，得到结果：



指令十五，slt $6,$1,$2，对应的二进制数是10011000001000100011000000000000，因为一号寄存器的值大于二号寄存器的值，

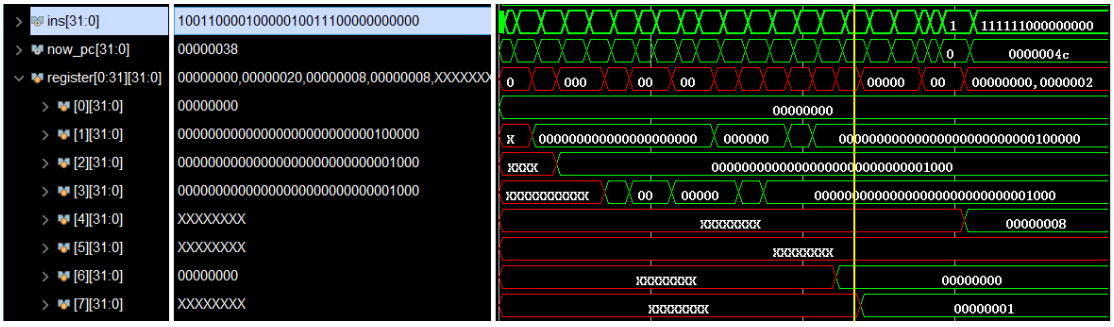


于是给六号寄存器 0 值；

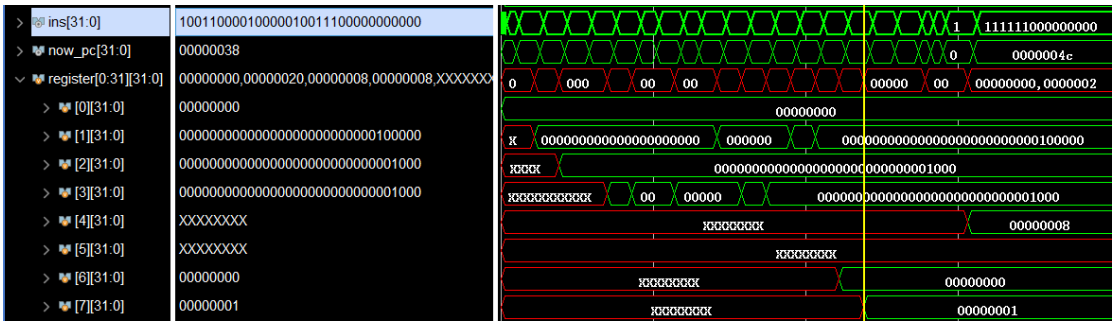


指令十六，slt $7,$2,$1，对应二进制为10011000010000010011100000000000，二号寄存器的值小于一号寄存

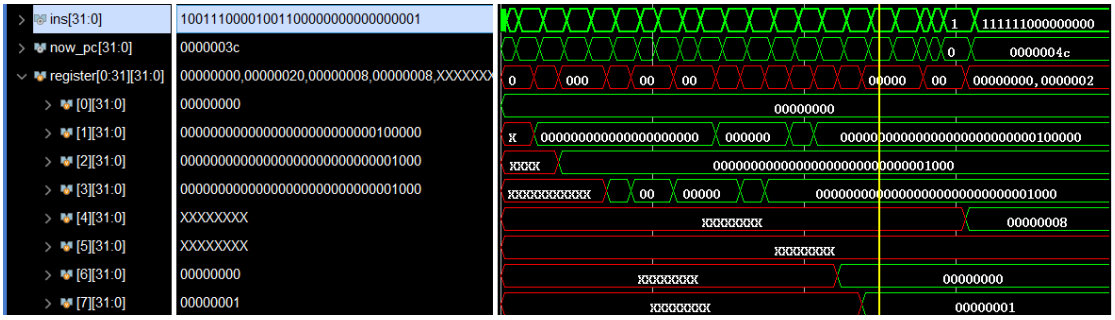
器的值，



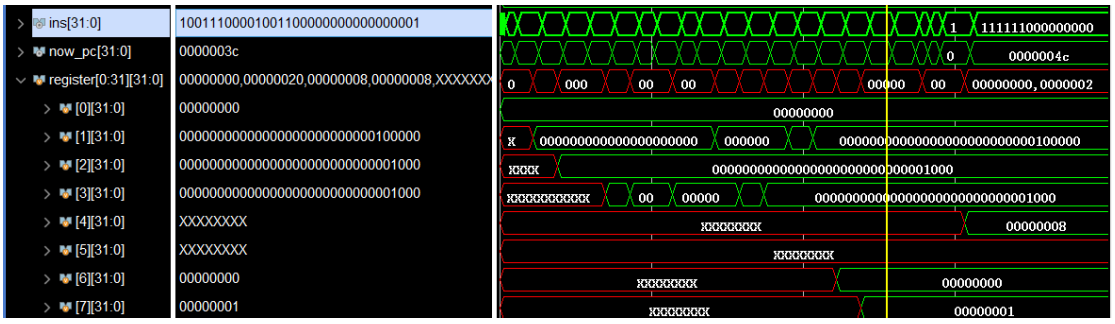
所以七号寄存器的值为 1。



指令十八，slti $7,$6,1,因为六号寄存器的值小于 1，



所以$7 的值仍为 1，



指令十九，jal 1，对应二进制为 10011100110001110000000000000001

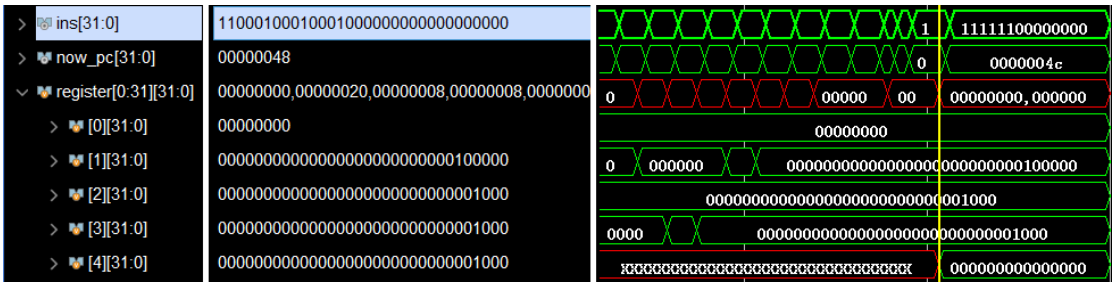
调用子程序，对应的子程序是地址为 1 的指令，也就是第二条指令

于是执行指令二十，jr $31，对应的二进制位11101000000000000000000000000001

执行完毕后，回到断点。

执行指令二十一，lw $4,0($2)，对应的二进制为11000100010001000000000000000000，也就是将$2 中的内容存入$4中，

故执行完毕后，



最后执行指令二十二，停机指令，halt，对应二进制为11111100000000000000000000000000

06 下载到 FPGA 与分析

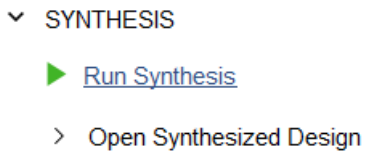
这一部分的难度我感觉应该是整个实验中最大的，因为没有任何的教程，所有需要的相关知识都需要我们自己去检索，所以光是研究怎么去下板就花费了我一两天的时间，至于后续成功下板过程中遇到的问题更是数不胜数，这里先选有一定价值的部分做介绍，最后再附录一

些有趣的小问题的介绍。

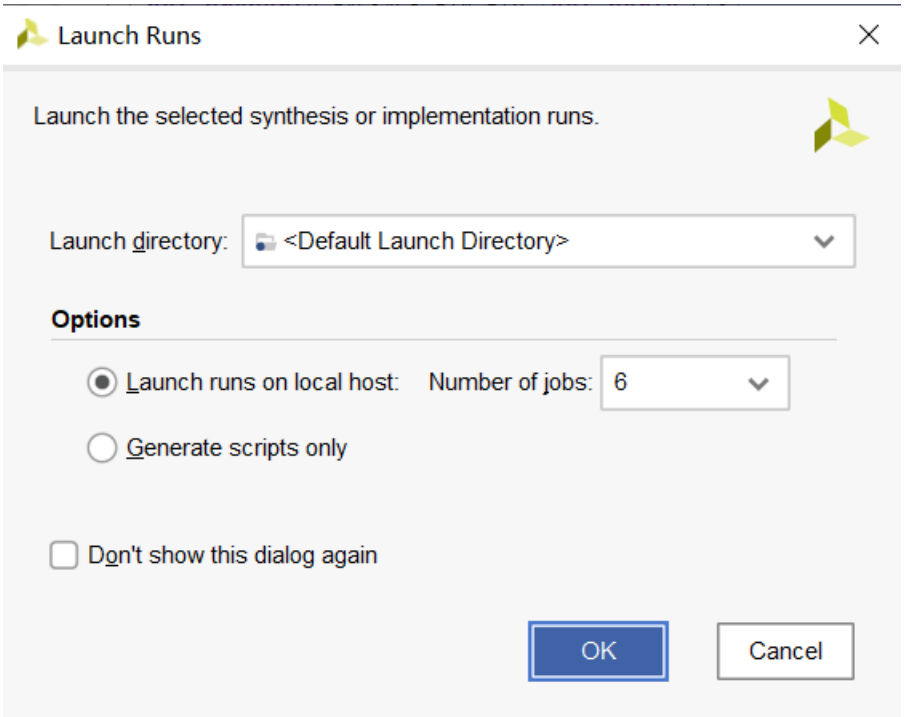
6.1 下板步骤

需要解释的是，我没有选择编写约束代码，而是通过可视交互的引脚赋值来是实现的，具体步骤如下：

- 运行 SYSTHESIS 下的 Run Synthesis

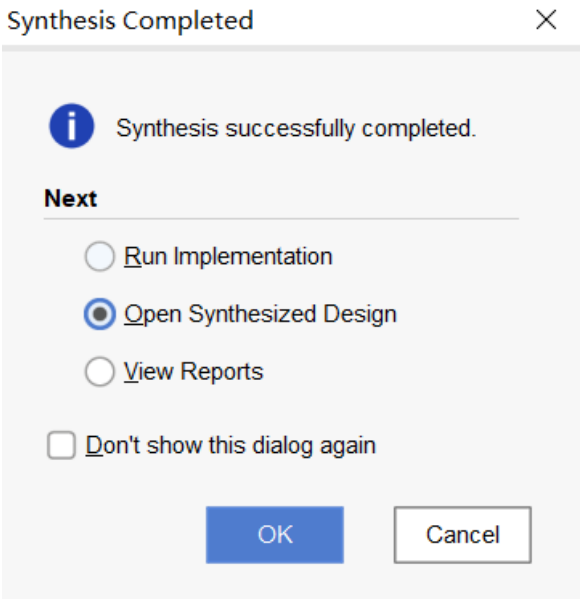


- 弹窗都是默认值，但是这一部分我有略作了解，这一节最后有趣的小问题部分我将介绍

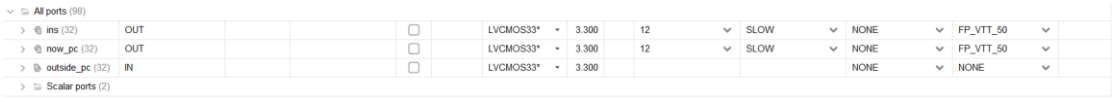


随后耐心等待即可

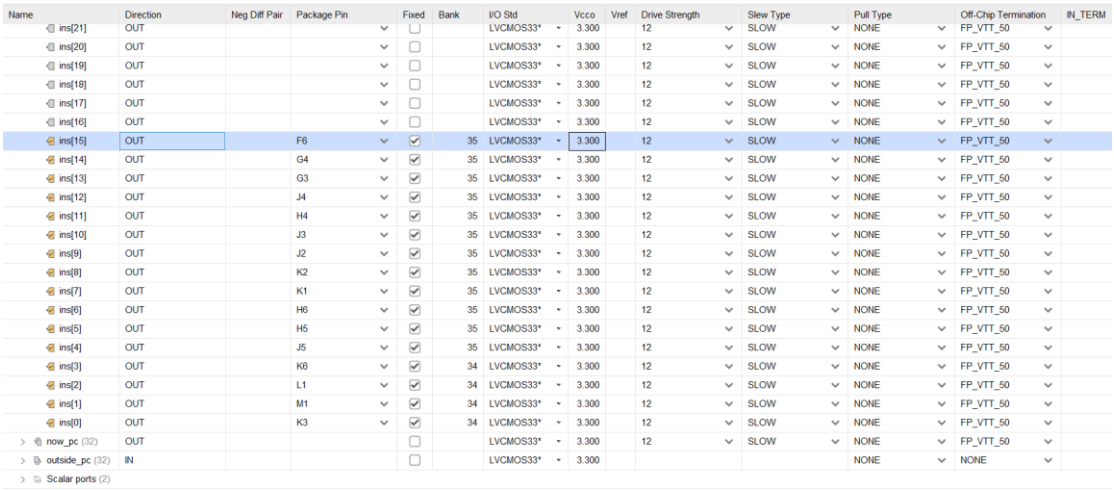
- 运行完后，我们选择 Open Synthesis Design 进行引脚分配



- 在随后的 I/O ports 界面



为了能在 FPGA 上动态的显示，我们可以选择把 LED 灯的引脚分配给 ins 或者是 now_pc,为了与我们之前仿真波形和指令更好的进行比较和对应，我们选择 ins，由于指令都是 32 位，但是 FPGA 板上只有 16 个 LED 灯，所以我们干脆就将 ins 的低 16 位对应显示在 16 个 LED 灯上，并且调整 I/O Std 到 3.3v，对应第一部分对 FPGA 引脚的介绍，最终的分配情况如下：

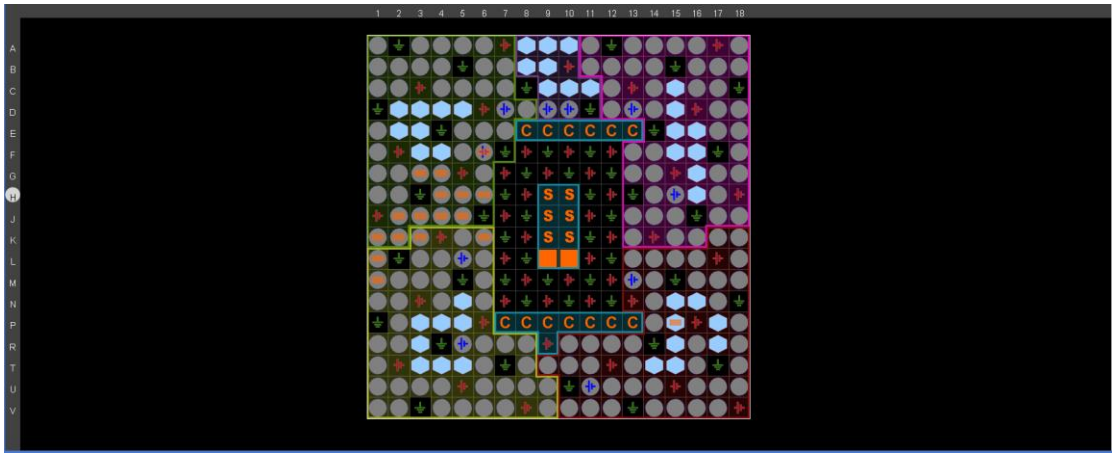


- 在 Scalar ports 的引脚分配时，我研究发现有两种分配方式，

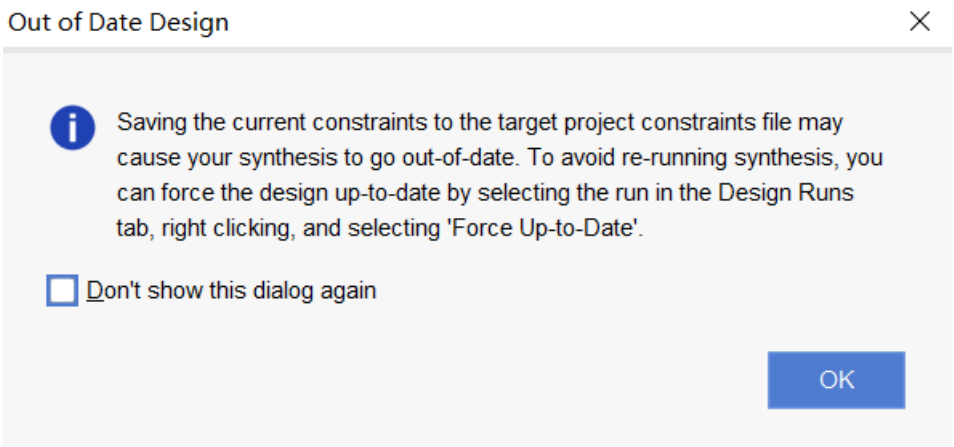
一种是使用系统默认的时钟，给 CLK P17,RST P15 ，这种方式下 FPGA 的时钟信号有板内置的时钟发生，运行速度很快，不方便我们观察和验收。

于是我研究使用了第二种方式，将 RST 按钮复用为时钟信号，也就是给 CLK 以 RST 的默认引脚 P15，这样就能实现按一次 FPGA 板上的 RST 按钮，就运行一个时钟周期，大大方便我们做实验。

完成所有的分配后，我们可以看到 I/O Planning 如下：

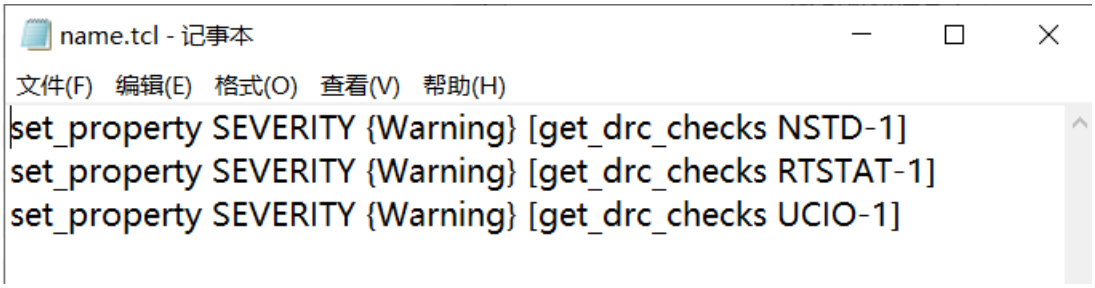


然后点击保存，Vivado 就会帮我们自动生成对应的约束文件：

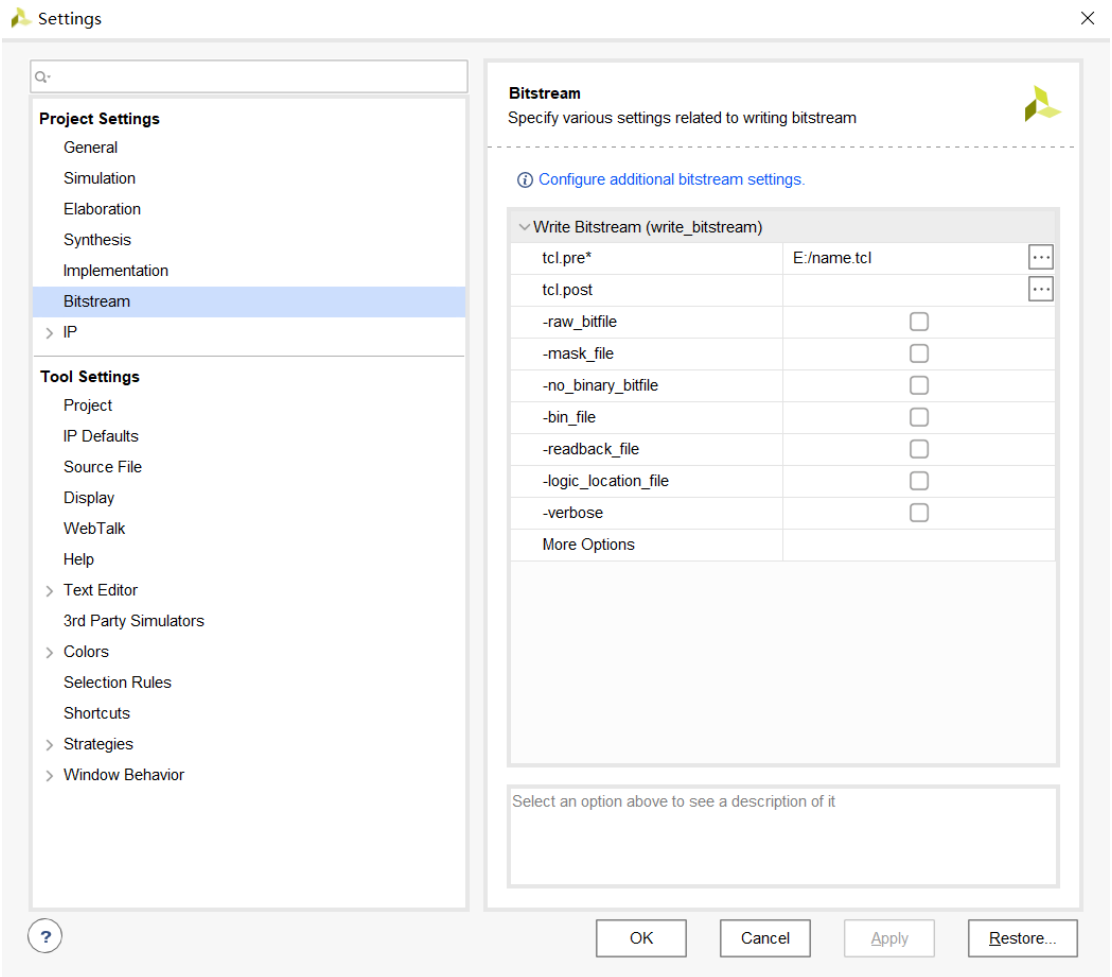


- 但是这时候的约束文件直接运行 Generate Bitstream 会报错，我们可以在 Messages 窗口中找到 log 和修改意见，按照修改意

见，我们新建记事本，添加以下三句

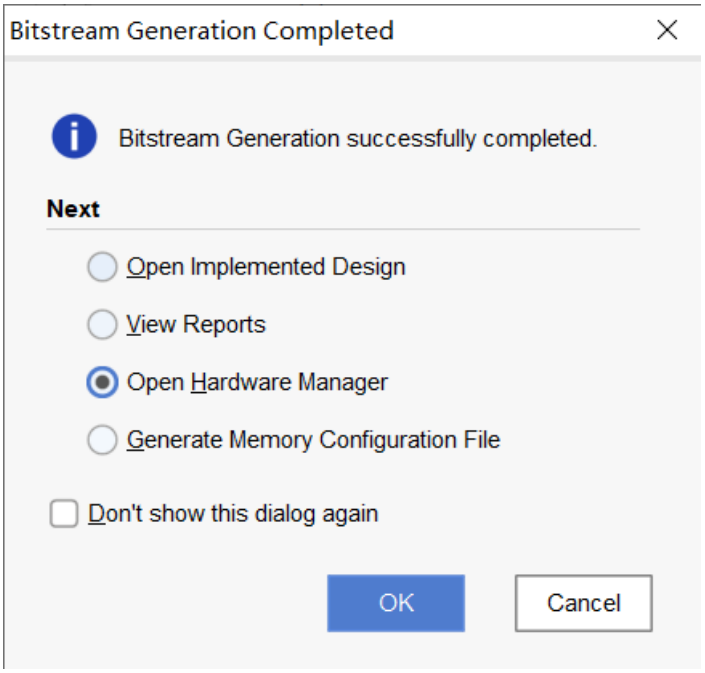


然后右键 Generate Bitstream，选择 Bitstream Settings，将

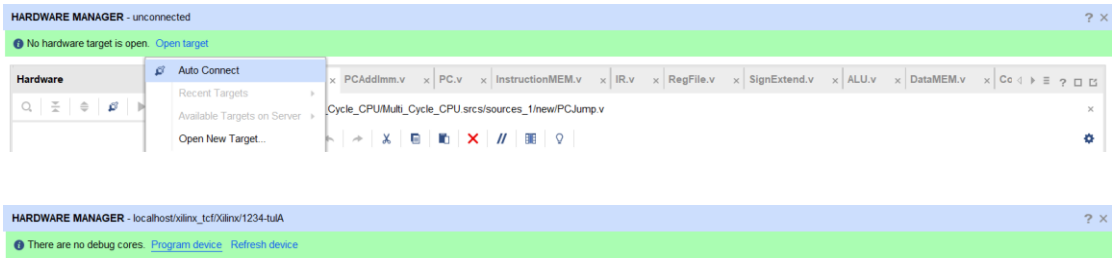


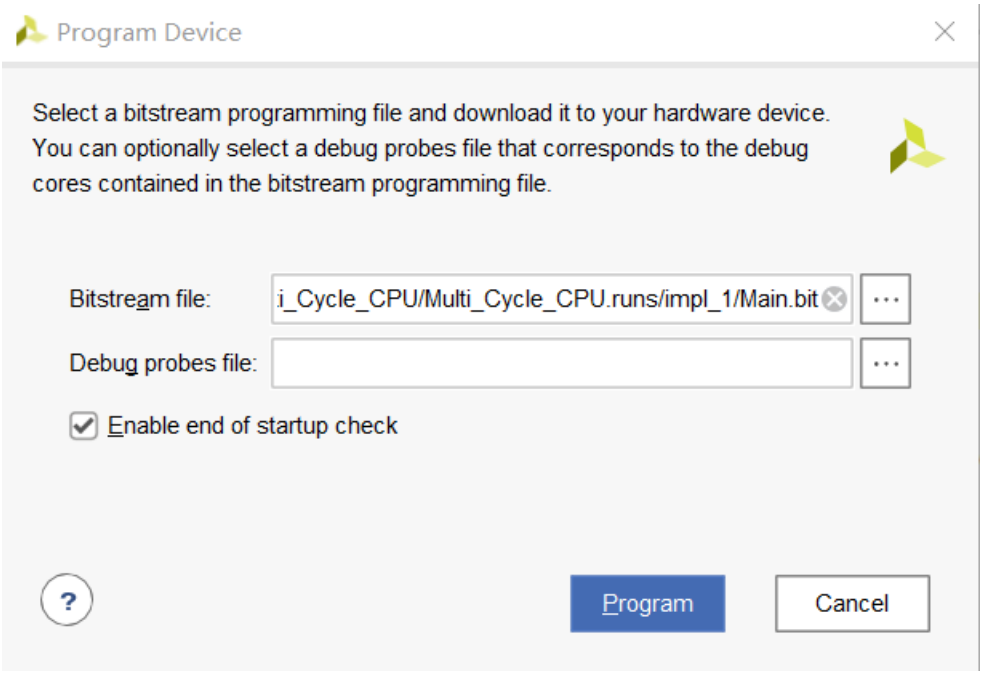
其添加到 tcl.pre*中。

- 最后点击 Generate Bitstream，进行下板，然后选择 Open Hardware Manager。



再选择 Auto Connect，就已经成功下板了。





我们点击 Program Device 就能看到运行了。

约束代码 cpu_test1.xdc 如下：

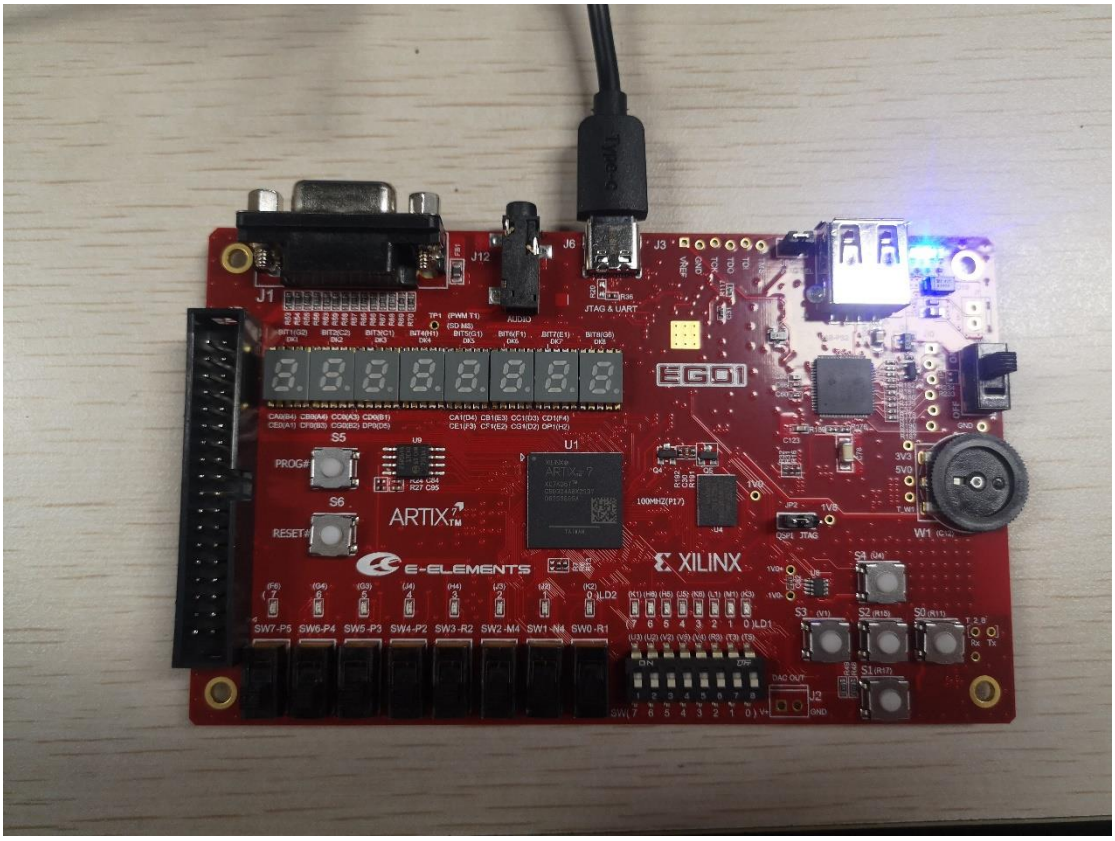
```
set_property PACKAGE_PIN P15 [get_ports CLK]
set_property IOSTANDARD LVCMOS33 [get_ports CLK]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[31]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[30]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[29]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[28]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[27]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[26]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[25]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[24]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[23]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[22]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[21]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[20]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[19]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[18]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[17]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[16]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {now_pc[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[31]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[30]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[29]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[28]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[27]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[26]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[25]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[24]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[23]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[22]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[21]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[20]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[19]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[18]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[17]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[16]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {outside_pc[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[31]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[30]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[29]}]
```

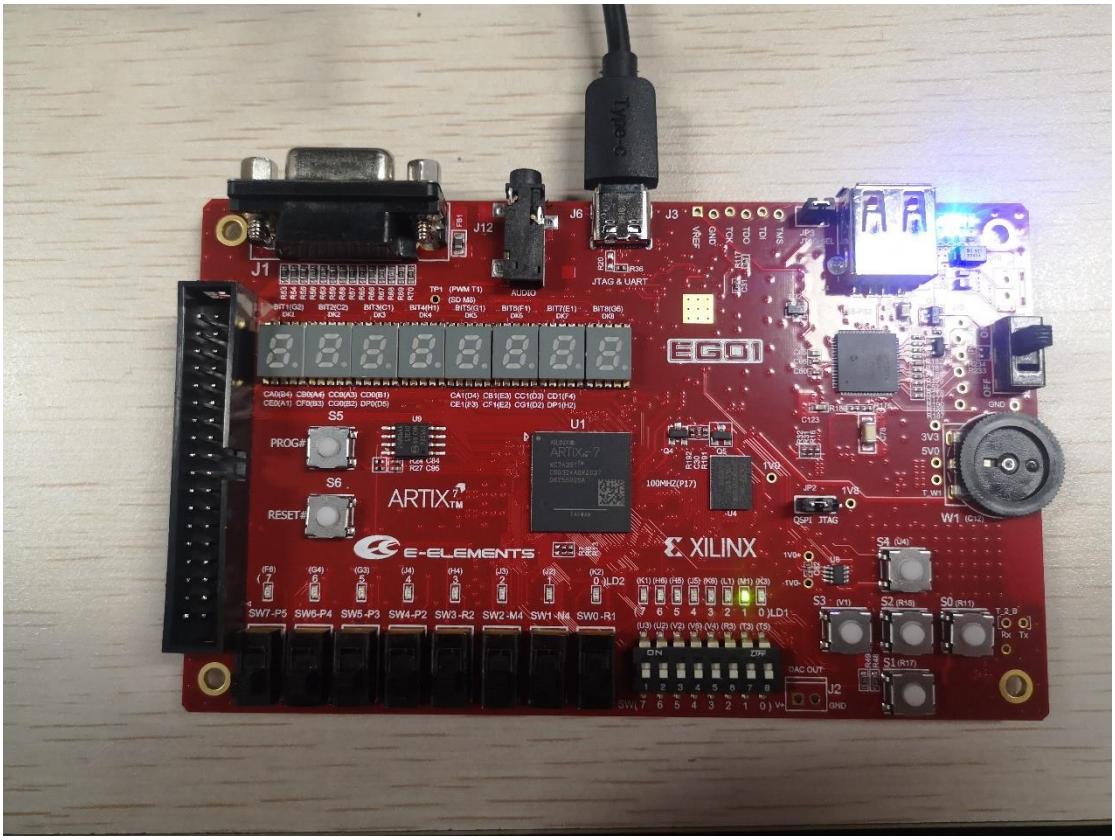
```
set_property IOSTANDARD LVCMOS33 [get_ports {ins[28]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[27]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[26]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[25]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[24]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[23]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[22]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[21]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[20]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[19]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[18]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[17]}]
set_property IOSTANDARD LVCMOS33 [get_ports {ins[16]}]

set_property IOSTANDARD LVCMOS33 [get_ports RST]

set_property PACKAGE_PIN K3 [get_ports {ins[0]}]
set_property PACKAGE_PIN M1 [get_ports {ins[1]}]
set_property PACKAGE_PIN L1 [get_ports {ins[2]}]
set_property PACKAGE_PIN K6 [get_ports {ins[3]}]
set_property PACKAGE_PIN J5 [get_ports {ins[4]}]
set_property PACKAGE_PIN H5 [get_ports {ins[5]}]
set_property PACKAGE_PIN H6 [get_ports {ins[6]}]
set_property PACKAGE_PIN K1 [get_ports {ins[7]}]
set_property PACKAGE_PIN K2 [get_ports {ins[8]}]
set_property PACKAGE_PIN J2 [get_ports {ins[9]}]
set_property PACKAGE_PIN J3 [get_ports {ins[10]}]
set_property PACKAGE_PIN H4 [get_ports {ins[11]}]
set_property PACKAGE_PIN J4 [get_ports {ins[12]}]
set_property PACKAGE_PIN G3 [get_ports {ins[13]}]
set_property PACKAGE_PIN G4 [get_ports {ins[14]}]
set_property PACKAGE_PIN F6 [get_ports {ins[15]}]
```

按按钮 RESET，就执行一个时钟周期，读指令具体情况如下：

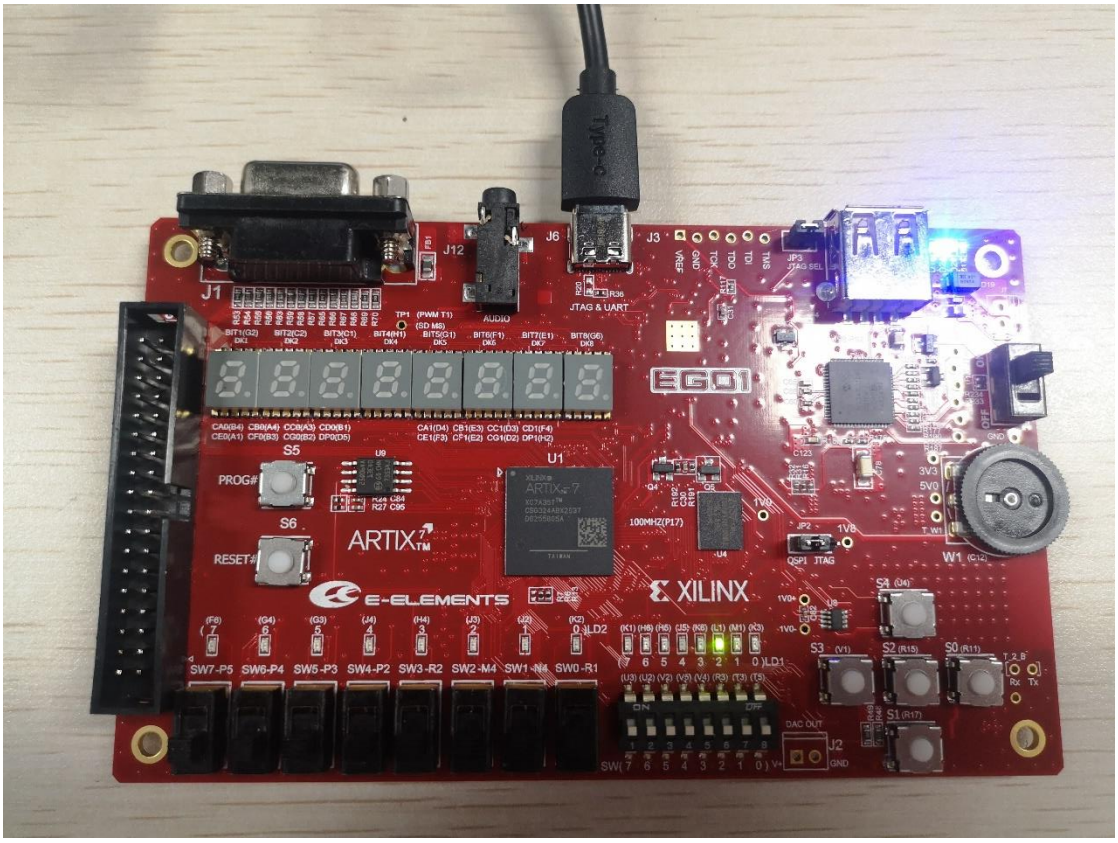
第一个时钟周期未读，



第二个时钟周期读入第一条指令，其低 16 位二进制数如下 0000000000000010，所以 FPGA 板情况如下：



第三个时钟周期读入第二条指令（因为第一条指令是跳转，所以第二条对应指令表的第三条）0000000000000100



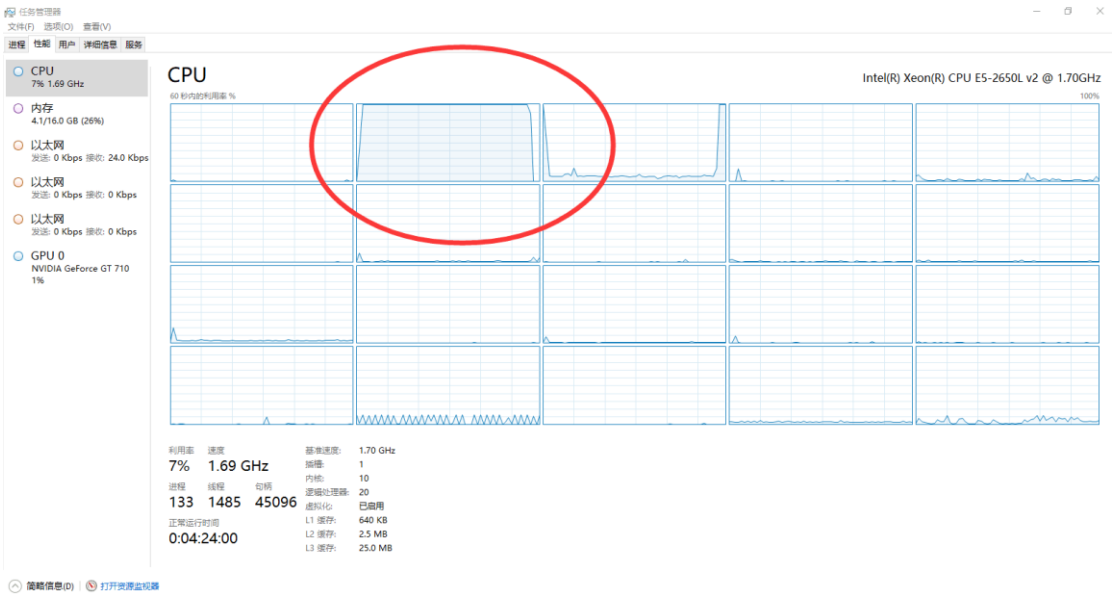
后续指令情况就不再赘述，情况与仿真的指令运行情况一致，验收时也已经给老师成功展示。

6.2 有趣的小问题

每次综合/实现时弹出的对话框中的 Number of Jobs 就是多线程控制，拉到 16 应该就会快了吧，但是实际用时 1 分 48 秒，查到 Windows 系统下 vivado 默认是使用 2 个线程编译工程，使用 get_param general.maxThreads 查询

```
get_param general.maxThreads
2
```

按照查到的博客上的方法，使用 set_param general.maxThreads 设置最大线程数，我的电脑是 10 核 20 线程，我就先设成 10 试试 set_param general.maxThreads 10，依旧是 1 核有难，19 核围观，惨不忍睹



结果是 1 分钟 51 秒左右，换成 20 线程试试，还是 1 分 51 秒。看来设置 general.maxThreads 是无效的，根本起不到加速的作用在测试的时候我专门注意了一下各个 CPU 核心的占用率，总结得出：双核高频最好，就在此时，想起了之前一个视频教程，用的是 Ubuntu，好像要快一点，随即将系统换成 Ubuntu 18.04，同配置，实现起来确实要比 Windows 快那么一点点，但是也不怎么明显。快了 5~10 秒那个样子。先在我的笔记本上做测试，在 Ubuntu 18.04 上，结果快了一倍，证实了 Vivado 需要的是高主频 CPU，堆核心的方法无用。

07 心得与体会

在本次系统硬件综合设计中，我充分将所学到的理论知识与实践相结

合起来。并基于 Verilog 硬件语言亲手制作了一个简单的 CPU，真正的掌握了一个 CPU 设计的全部模块与步骤，对之前学过的很多知识有了更具象，更深刻的理解了，更具体的感悟都穿插在实验报告的每个部分里面，与实验内容是相结合的。

在完成本次课程设计的过程中，我主要参考的是在网上检索的各种信息。

这一过程使我学到了很多东西，并且复习了之前的专业课知识。最终实现的 CPU 共能处理 16 种 MIPS32 指令，并下板调试成功。也算是

为大学以来的所有计算机硬件相关学习画上了一个还算完美的句号，这是人生一笔宝贵的财富，让我受益终身。