

**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 4](#_Toc60312266)

[1.1 课外准备工作 4](#_Toc60312267)

[1.2 实验关键部分 6](#_Toc60312268)

[02数据通路图 9](#_Toc60312269)

[2.1 数据通路图与控制线路 9](#_Toc60312270)

[03各模块详细设计与代码 10](#_Toc60312271)

[3.1 PC及相关模块 10](#_Toc60312272)

[3.2指令存储器与指令寄存器 15](#_Toc60312273)

[3.3 寄存器堆 19](#_Toc60312274)

[3.4 加法器及相关模块 21](#_Toc60312275)

[3.5 数据存储器 25](#_Toc60312276)

[3.6 控制单元及其三个子模块 27](#_Toc60312277)

[3.7 四个分段寄存器 36](#_Toc60312278)

[3.8 三种多路选择器 40](#_Toc60312279)

[04支持的16条指令 43](#_Toc60312280)

[4.1 支持的16条指令 43](#_Toc60312281)

[4.2 指令存储器内的指令设计 49](#_Toc60312282)

[05 仿真波形与分析 49](#_Toc60312283)

[5.1 仿真代码 50](#_Toc60312284)

[5.2 仿真波形分析 52](#_Toc60312285)

[06 下载到FPGA与分析 59](#_Toc60312286)

[6.1 下板步骤 60](#_Toc60312287)

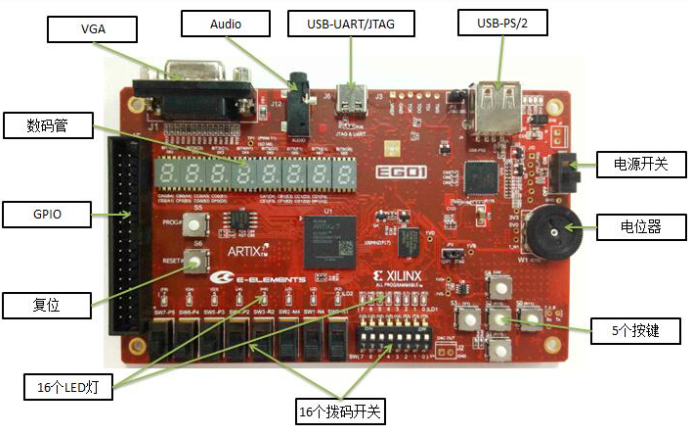
[6.2 有趣的小问题 69](#_Toc60312288)

[07 心得与体会 70](#_Toc60312289)

# 01写在前面

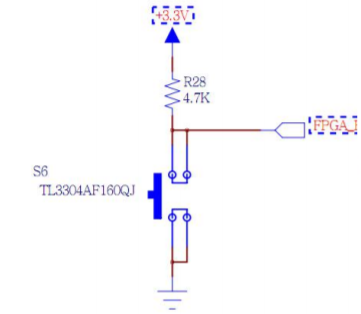
经过多门课程的理论和时间准备，我最终在大三上的第17周完成了多周期流水线CPU的设计和实现，该实验涉及的相关课程很多，主要的有陈田老师的计算机组成原理课程，李建华老师的计算机体系结构课程，徐娟老师的汇编语言课程，刘军老师的数字逻辑课程，需要自己学习的知识有Verilog硬件语言和FPGA实验板的相关知识。

## 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。

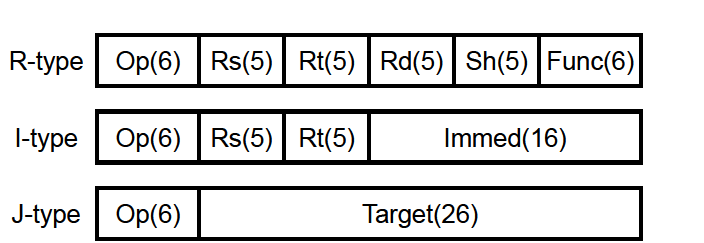
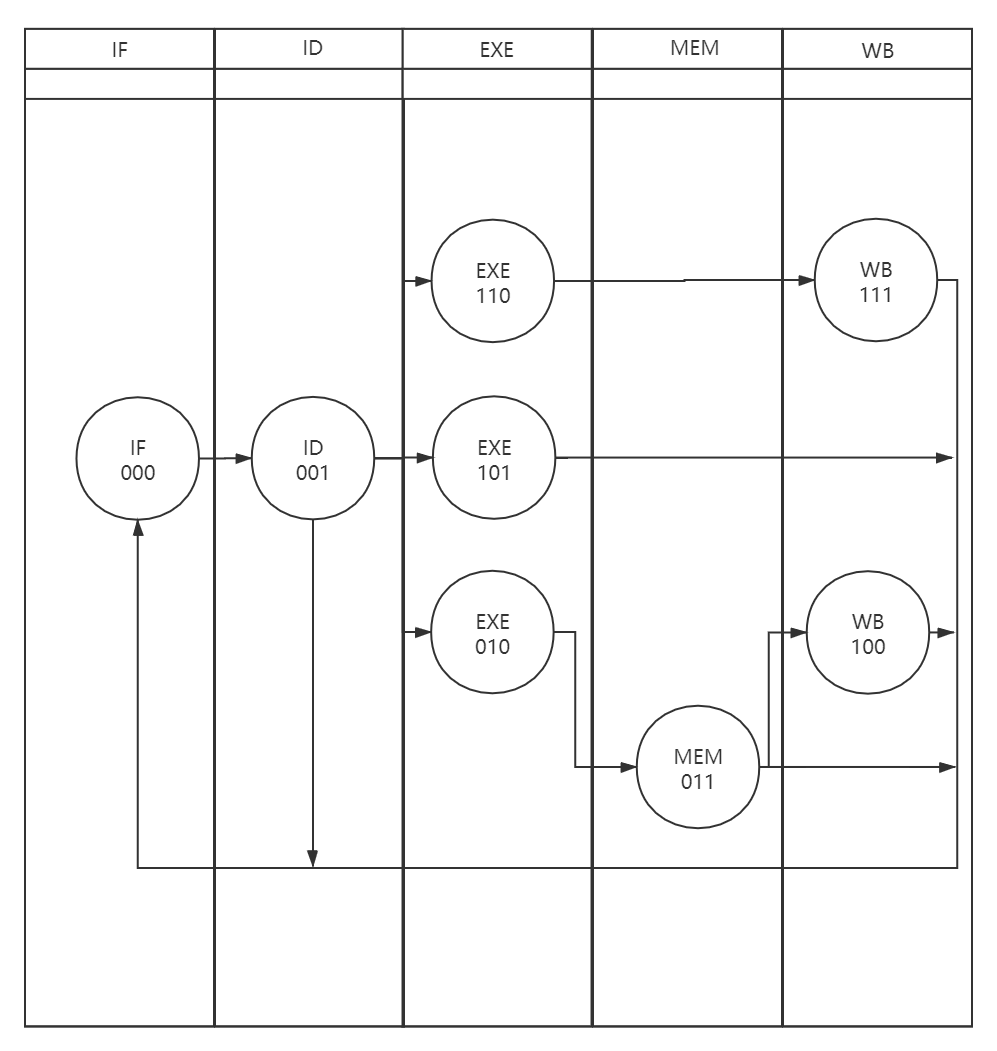
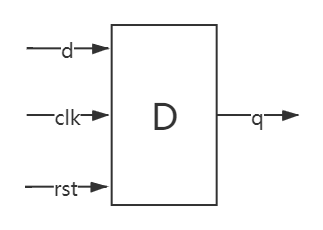
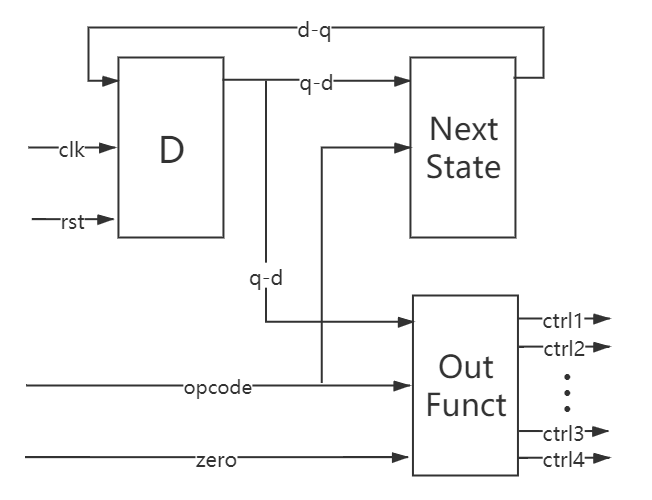
对应的管脚约束如下

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。

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。

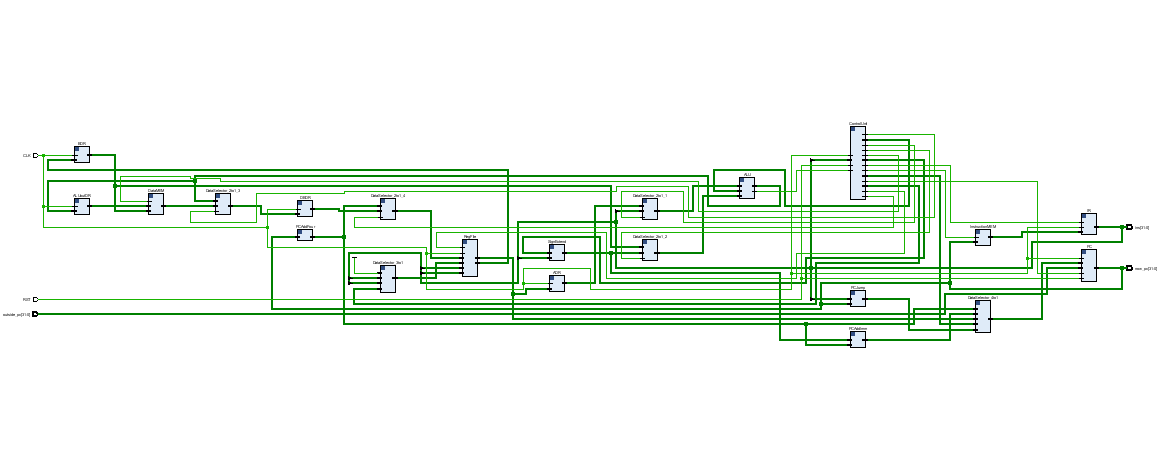
# 02数据通路图

## 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。访间存储器时，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

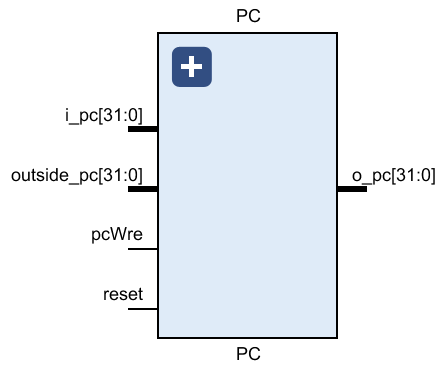
接下来按照在Vivado上面进行的实验步骤进行介绍，完成各个模块的代码后，我们可以在RTL ANALYSIS里面进行Schematic。

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。

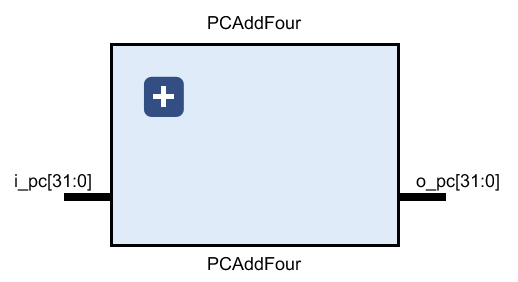
## 3.1 PC及相关模块

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：

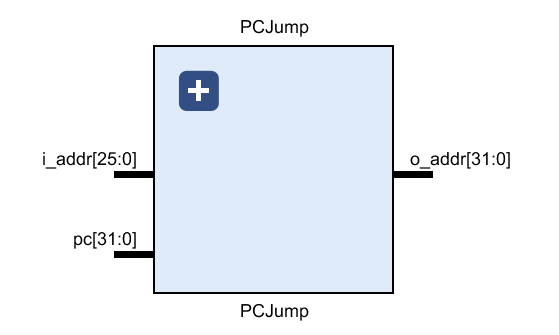
* 顺序存储的下一条指令的地址
* 子程序的地址
* 执行跳转指令后的地址

针对这几种不同的情况，需要加入三个对应的单元

* PCAddFour

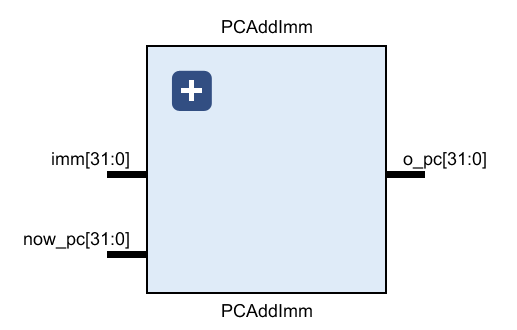
该单元的功能就是实现PC<-PC+4，所以与该单元相关的指令有add、addi、sub、or、ori、and、slt、slti、sll、sw、lw、beq。

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
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

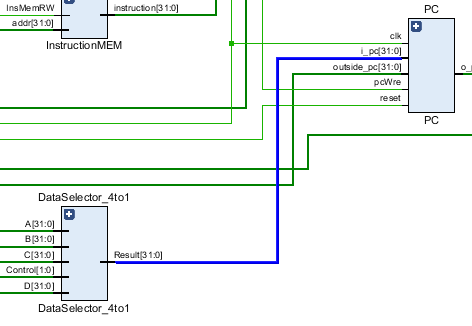
* PCAddImm

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。

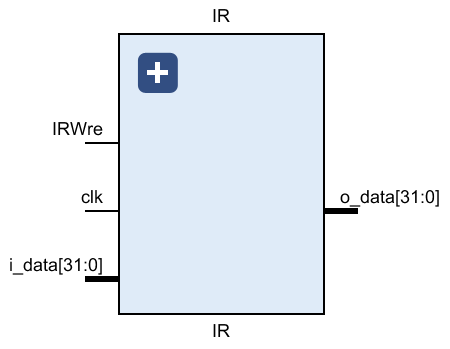
1. `timescale 1ns / 1ps
2. // PC模块的实现
3. // @param clk 时钟信号
4. // @param pcWre信号
5. // @param reset信号
6. // @param i\_pc 输入的pc值
7. // @param o\_pc 输出的pc值
8. // @param outside\_pc ???
9. module PC(clk, i\_pc, pcWre, reset, outside\_pc, o\_pc);
10. input wire clk, pcWre, reset;
11. input wire [31:0] i\_pc, 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;
21. end
22. end
23. 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]。

1. `timescale 1ns / 1ps
2. // 指令存储单元的实现
3. // @param InsMemRW 指令存储单元信号
4. // @param addr pc上指令的地址
5. // @param outside\_pc 获取初始化的pc
6. // @param instruction 取得的指令
7. module InstructionMEM (addr, InsMemRW, instruction);
8. 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;
19. mem[4]=8'b11100111;
20. mem[5]=8'b11100000;
21. mem[6]=8'b00000000;
22. mem[7]=8'b00000000;
24. mem[8]=8'b00001000;
25. mem[9]=8'b00000001;
26. mem[10]=8'b00000000;
27. mem[11]=8'b00000100;
29. mem[12]=8'b00001000;
30. mem[13]=8'b00000010;
31. mem[14]=8'b00000000;
32. mem[15]=8'b00001000;
34. mem[16]=8'b11000000;
35. mem[17]=8'b01000010;
36. mem[18]=8'b00000000;
37. mem[19]=8'b00000000;
39. mem[20]=8'b00000000;
40. mem[21]=8'b01000001;
41. mem[22]=8'b00011000;
42. mem[23]=8'b00000000;
44. mem[24]=8'b00000100;
45. mem[25]=8'b01100001;
46. mem[26]=8'b00011000;
47. mem[27]=8'b00000000;
49. mem[28]=8'b11010000;
50. mem[29]=8'b01000011;
51. mem[30]=8'b11111111;
52. mem[31]=8'b11111110;
54. mem[32]=8'b01001000;
55. mem[33]=8'b00100001;
56. mem[34]=8'b00000000;
57. mem[35]=8'b00000001;
59. mem[36]=8'b01000000;
60. mem[37]=8'b01000001;
61. mem[38]=8'b00011000;
62. mem[39]=8'b00000000;
64. mem[40]=8'b00000000;
65. mem[41]=8'b01000000;
66. mem[42]=8'b00011000;
67. mem[43]=8'b00000000;
69. mem[44]=8'b01000100;
70. mem[45]=8'b01100010;
71. mem[46]=8'b00001000;
72. mem[47]=8'b00000000;
74. mem[48]=8'b01100000;
75. mem[49]=8'b00000010;
76. mem[50]=8'b00001000;
77. mem[51]=8'b10000000;
79. mem[52]=8'b10011000;
80. mem[53]=8'b00100010;
81. mem[54]=8'b00110000;
82. mem[55]=8'b00000000;
84. mem[56]=8'b10011000;
85. mem[57]=8'b01000001;
86. mem[58]=8'b00111000;
87. mem[59]=8'b00000000;
89. mem[60]=8'b10011100;
90. mem[61]=8'b00100110;
91. mem[62]=8'b00000000;
92. mem[63]=8'b00000001;
94. mem[64]=8'b10011100;
95. mem[65]=8'b11000111;
96. mem[66]=8'b00000000;
97. mem[67]=8'b00000001;
99. mem[68]=8'b11101000;
100. mem[69]=8'b00000000;
101. mem[70]=8'b00000000;
102. mem[71]=8'b00000001;
104. mem[72]=8'b11000100;
105. mem[73]=8'b01000100;
106. mem[74]=8'b00000000;
107. mem[75]=8'b00000000;
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
24. // rd != 0 是确保零号寄存器不会改变的作用(MiPS)
25. **if** ((rd != 0) && (RegWre == 1)) begin
26. **register**[rd] = i\_data;
27. end
28. end
29. 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。

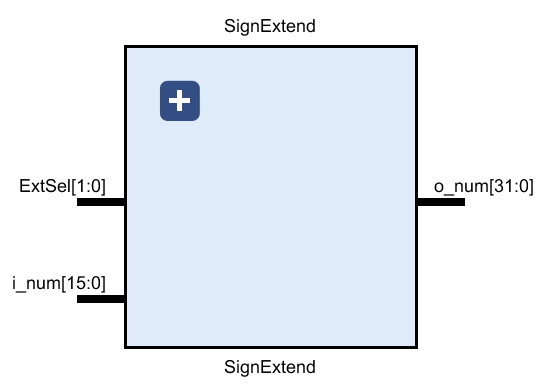
## 3.4 加法器及相关模块

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：

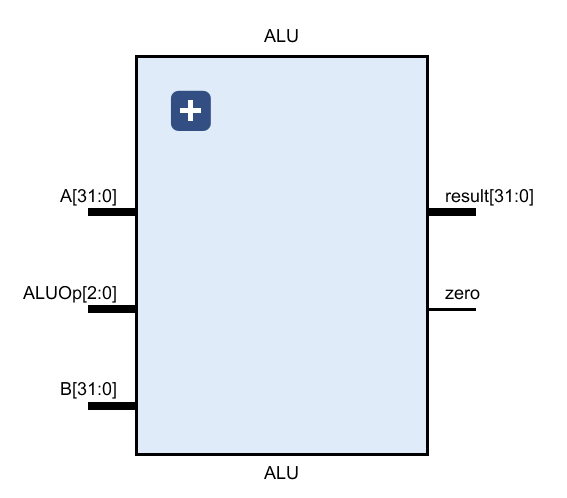
当ALUSrcA为0，对应的二选一的选择器的输出值则来自寄存器堆data1输出，相关指令: add、 sub、 addi、 or、 and、ori、beg、slt、 slti、 sw、lw；当ALUSrcA为1，则来自移位数sa，同时，进行(zero-extend)sa， 即{27(0),sa}, 相关指令: sll

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
14. **case**(ExtSel)
15. // ExtSel 为00时，sa位扩展
16. 2'b00: o\_num <= {{27{0}}, i\_num[10:6]};
17. // ExtSel 为01时，无符号立即数扩展
18. 2'b01: o\_num <= {{16{0}}, i\_num[15:0]};
19. // 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)
20. 2'b10: o\_num <= {{16{i\_num[15]}}, i\_num[15:0]};
21. // 其它情况默认   有符号立即数扩展
22. **default**: o\_num <= {{16{i\_num[15]}}, i\_num[15:0]}; // 默认符号扩展
23. endcase
24. end
25. 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

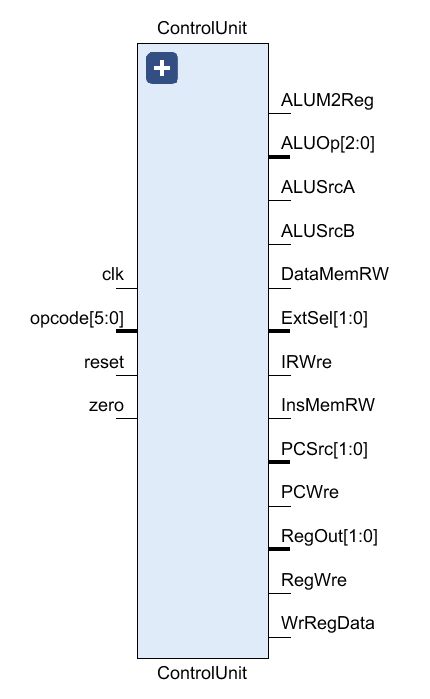
数据存储器的实现与单周期是类似的。定义8位的寄存器数组，将32位的数据分割成4个8位的小段进行存取。不过这里我将D，R两个信号合并为DataMEMRW，当DataMEMRW为1（/WR）时进行写操作，当DataMEMRW为0(/RD)时进行读操作。写与读操作的单位是8位二进制数，一共操作四次。与之相关的指令有lw。

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。

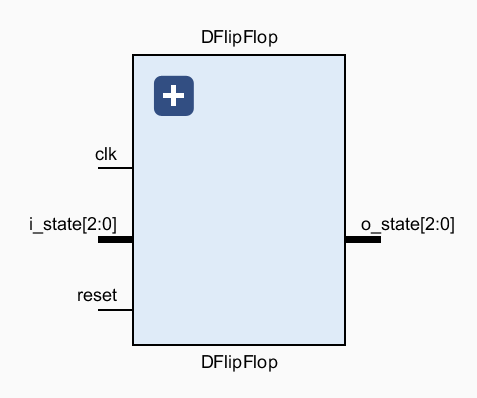
该值经过DBDR后再与来自PC+4的数据进行二选一选择器的选择，然后写回入寄存器组寄存器。

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

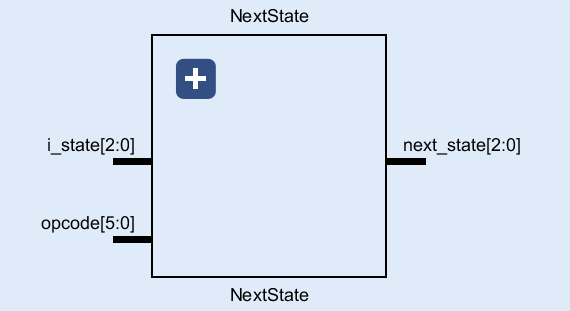
上面解释各个模块的时候已经对各个模块对应的控制信号进行了介绍，所以此处对控制单元以及其总体的架构进行介绍。

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，Contro1Unit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;
11. output reg[2:0]o\_state;
12. always @(posedge clk) begin
13. **if** (reset) o\_state = 3'b000;
14. **else** o\_state = i\_state;
15. end
16. endmodule

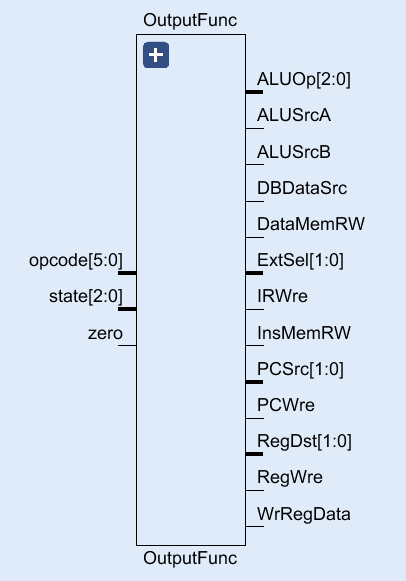
* NextState

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。

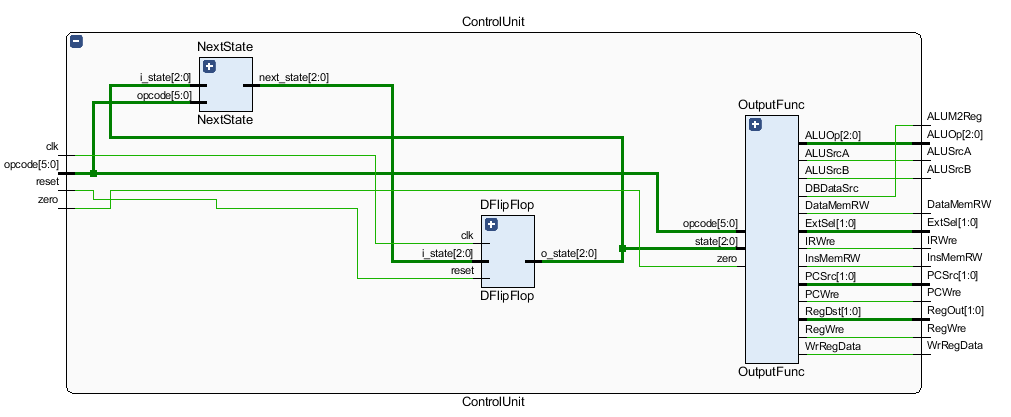
当输入NextState的状态码与操作码发生改变时，使用case产生下一状态，对case的设计如下：

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、1w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1w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。
6. `timescale 1ns / 1ps
7. // NextState模块的实现
8. // @param i\_state 输入的状态
9. // @param opcode 输入的操作码
10. // @param next\_state 下一状态
11. module NextState(i\_state, opcode, next\_state);
12. input [2:0]i\_state;
13. input [5:0]opcode;
14. output reg[2:0]next\_state;
15. parameter [2:0] IF = 3'b000, // IF状态
16. ID = 3'b001, // ID状态
17. aEXE = 3'b110, // 第一条分支的EXE状态
18. bEXE = 3'b101, // 第二条分支的EXE状态
19. cEXE = 3'b010, // 第三条分支的EXE状态
20. MEM = 3'b011, // MEM状态
21. aWB = 3'b111, // 第一个分支的WB状态
22. cWB = 3'b100; // 第三个分支的WB状态
24. always @(i\_state or opcode) begin
25. **case** (i\_state)
26. //所有指令都经过IF-ID
27. IF: next\_state = ID;
28. //四条分叉，其中EXE有三种
29. ID: begin
30. **case** (opcode[5:3])
31. 3'b110: begin
32. **if** (opcode == 6'b110100) next\_state = bEXE; // beq指令
33. **else** next\_state = cEXE; // sw, lw指令
34. end
35. 3'b111: next\_state = IF; // j, jal, jr, halt指令
37. **default**: next\_state = aEXE; // add, sub等指令
38. endcase
39. end
40. //只有一条路
41. aEXE: next\_state = aWB;
42. bEXE: next\_state = IF;
43. cEXE: next\_state = MEM;
44. //
45. MEM: begin
46. **if** (opcode == 6'b110001) next\_state = cWB; // lw指令
47. **else** next\_state = IF; // sw指令
48. end
49. //两种写回都是IF结尾
50. aWB: next\_state = IF;
51. cWB: next\_state = IF;
52. **default**: next\_state = IF;
53. endcase
54. end
55. endmodule

* OutputFunc

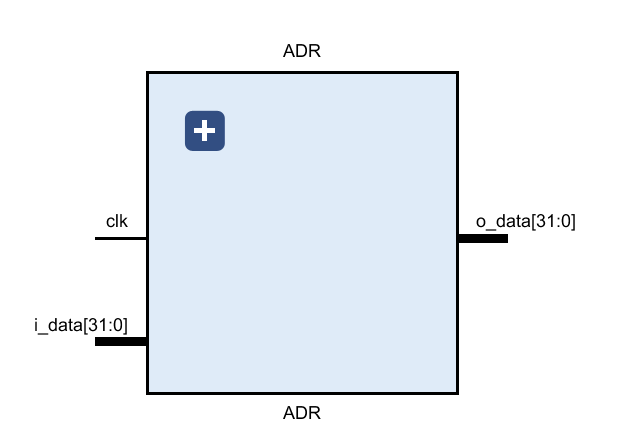
输出模块的大致思路类似于单周期ControlUnit的实现,只不过在多周期中是通过状态来定义控制信号的值。我定义了两个参数列表，一个用于定义各个状态，一个用于定义各种指令。这样的话在实现的时候就没有必要去查ALU操作的opcode了。根据控制信号作用表来定义各个控制信号。在最后，需要在IF阶段将RegWre与DataMEMRW置为0，防止在IF阶段写数据。

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, RegWre, 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;
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
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
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
117. endmodule

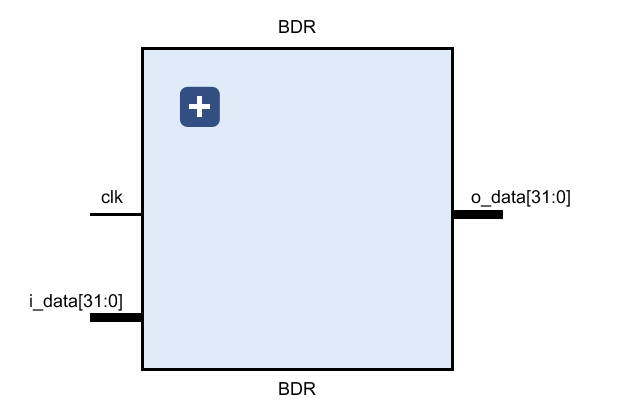
然后通过ControlUnit将它们连结起来：

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;
15. wire [2:0]i\_state, o\_state;
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);
21. endmodule

## 3.7 四个分段寄存器

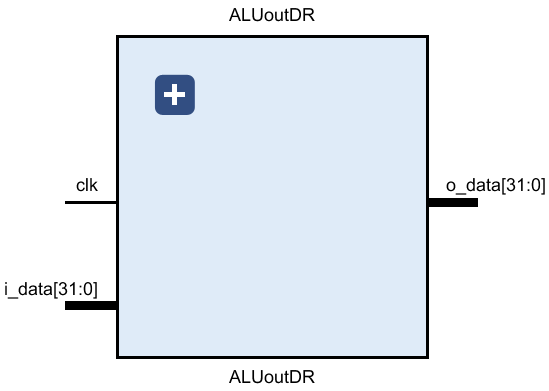
* ADR

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

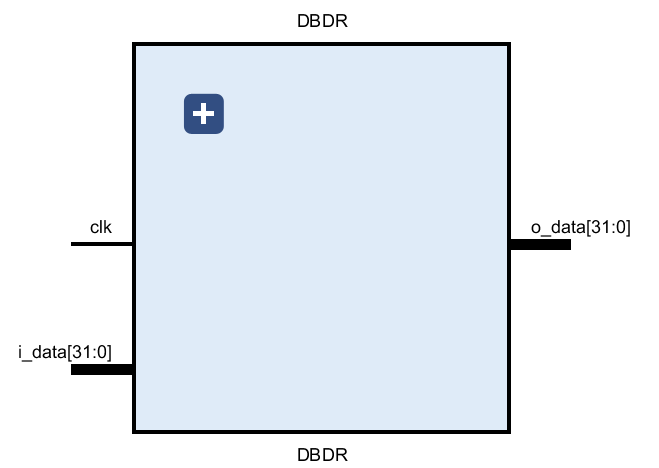
临时存储的寄存器BDR，目的是将输入寄存器的值在一定条件下进行输出（在这里就是在时钟信号的触发下进行输出），具体来说，BDR是将从寄存器堆输出端口2读取的数据在一定条件下输出到加法器输入端口B的寄存器。

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。

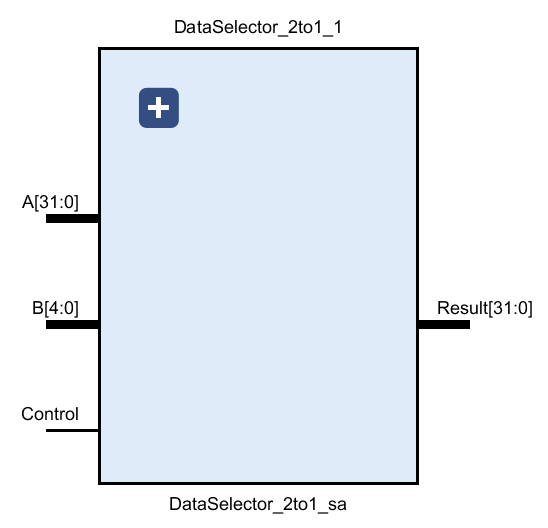
1. `timescale 1ns / 1ps
2. // 切割数据通路
3. // @param i\_data 输入的数据
4. // @param o\_data 输出的数据
5. // @param clk 时钟信号
6. module ALUoutDR(i\_data, clk, o\_data);
7. input clk;
8. input [31:0] i\_data;
9. output reg[31:0] o\_data;
10. //上升沿触发
11. always @(posedge clk) begin
12. o\_data = i\_data;
13. end
14.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

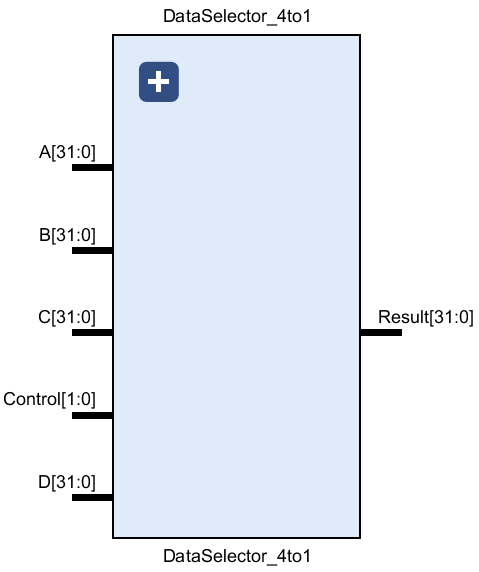
对于二选一选择器，则定义其控制信号为一位，即0，1。在每次输入、时钟信号或者是控制信号变化时，使用case语句选择某个输入输出。

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

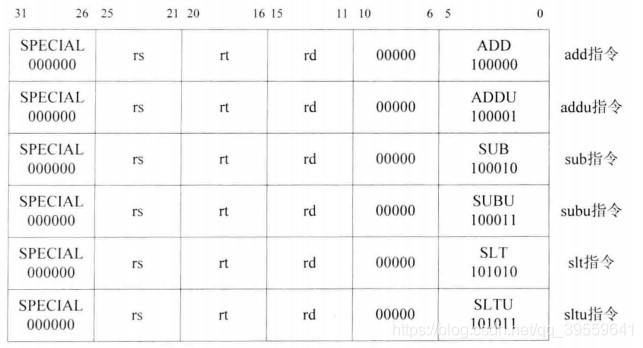
对于四选一选择器，则定义其控制信号为两位，即00，01, 10, 11，在每次输入、时钟信号、控制信号变化时，使用case语句选择某个输入输出。

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
5. // @param C 输入3
6. // @param D 输入4
7. // @param Control 数据选择器的控制信号
8. // @param Result 选择的结果
9. module DataSelector\_4to1(A, B, C, D, Control, Result);
10. input [31:0] A, B, C, D;
11. input [1:0]Control;
12. output reg[31:0] Result;
13. always @(Control or A or B or C or D) begin
14. **case**(Control)
15. 2'b00: Result = A;
16. 2'b01: Result = B;
17. 2'b10: Result = C;
18. 2'b11: Result = D;
19. **default**: Result = 0;
20. endcase
21. end
22. endmodule

# 04支持的16条指令

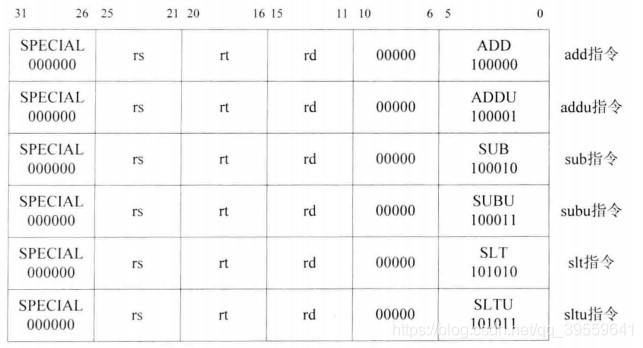
## 4.1 支持的16条指令

* 算数运算指令

1. **当功能码是6b100000时，表示add指令，加法运算。**

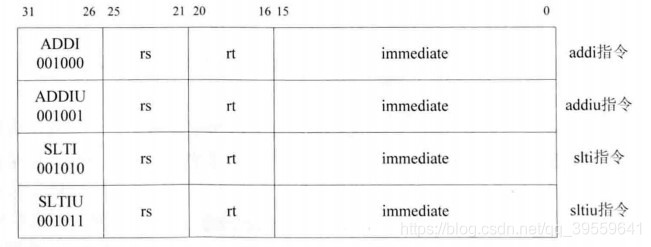
指令用法为：add rd，rs，rt。

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。

1. **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。

1. **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

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

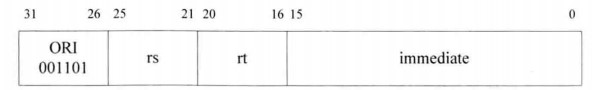
指令用法为：and  rd，rs，rt。

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。

1. **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。

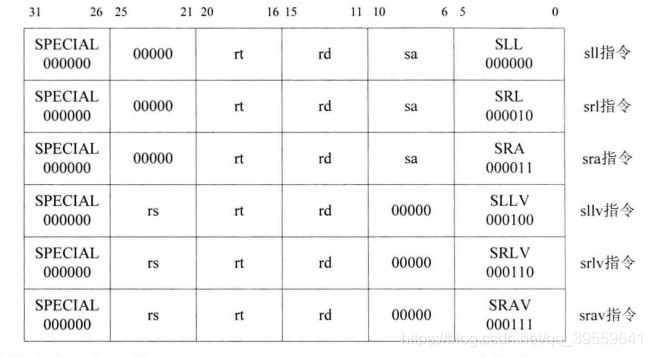
指令作用为：rd ← rs OR rt，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值进行逻辑“或”运算，运算结果保存到地址为rd的通用寄存器中。

1. **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。

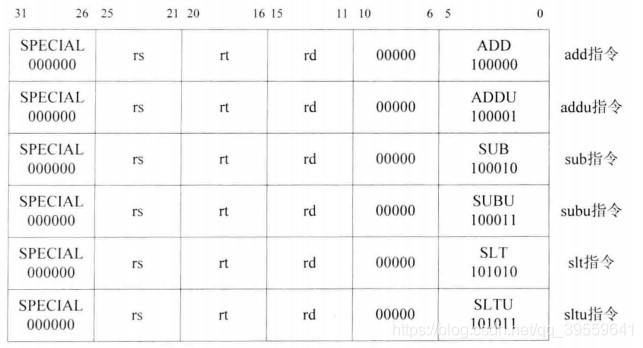
* 移位指令

1. **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。

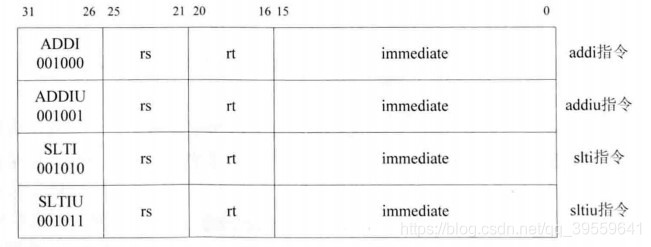
指令作用为：rd ← rt << sa（logic），将地址为rt的通用寄存器的值向左移sa位，空出来的位置使用0填充，结果保存到地址为rd的通用寄存器中。

* 比较指令

1. **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。

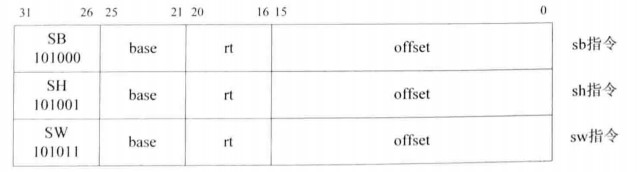
指令作用为：rd ←（rs<rt），将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值按照有符号数进行比较，如果前者小于后者，那么将1保存到地址为rd的通用寄存器中；反之，将0保存到地址为rd的通用寄存器中。

1. **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。

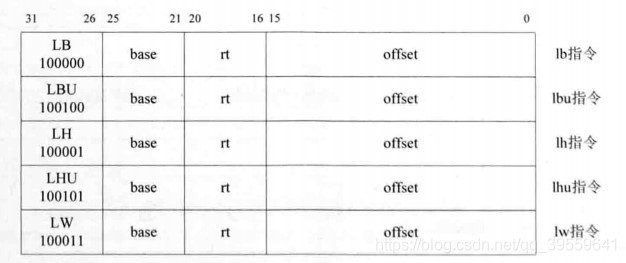
指令作用为：rt ←（rs <（sign\_extended）immediate），将指令中的16位立即数进行符号扩展，与地址为rs的通用寄存器的值按照有符号数进行比较，如果前者大于后者，那么将1保存到地址为rt的通用寄存器中；反之，将0保存到地址为rt的通用寄存器中。

* 存储器读写指令

1. **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）。

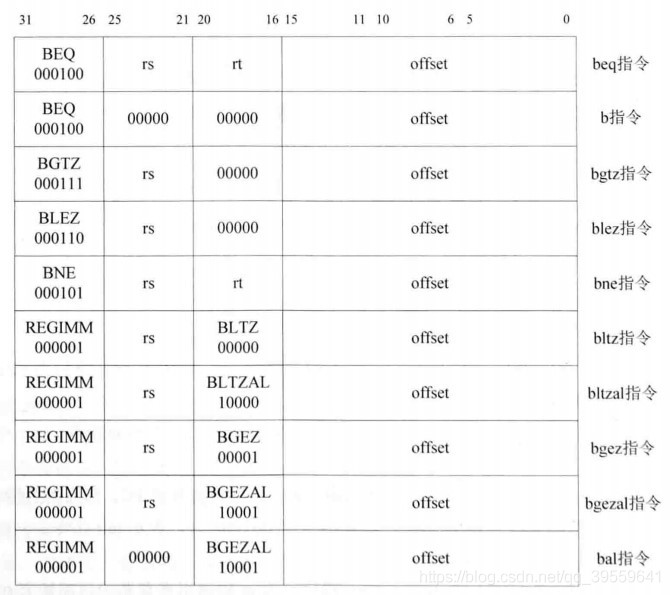
指令作用为：将地址为rt的通用寄存器的值存储到内存中的指定地址。该指令有地址对齐要求，要求计算出来的存储地址的最低两位为00。

1. **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。

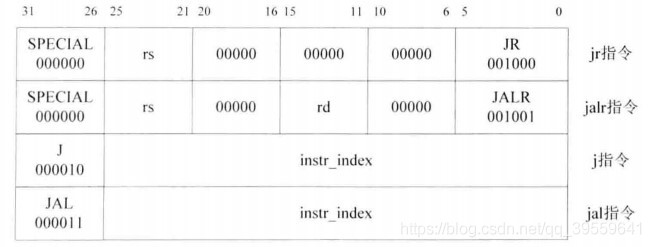
* 分支指令

1. **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。

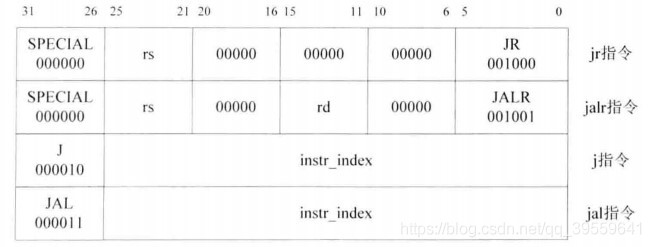
指令作用为：if rs=rt then branch，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值进行比较，如果相等，那么发生转移。

* 跳转指令

1. **当指令中的指令码为6b000010时，表示j指令。**

指令用法为：j target。

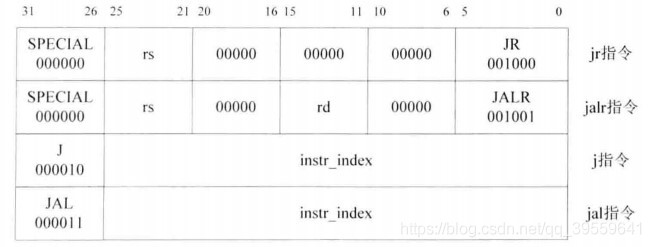
指令作用为：pc ←（pc+4）[31，28]|| target || ‘00'，转移到新的指令地址，其中新指令地址的低28位是指令中的target（也就是上图中的instr\_index）左移两位的值，新指令地址的高4位是跳转指令后面延迟槽指令的地址高4位。

1. **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

1. **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

## 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:

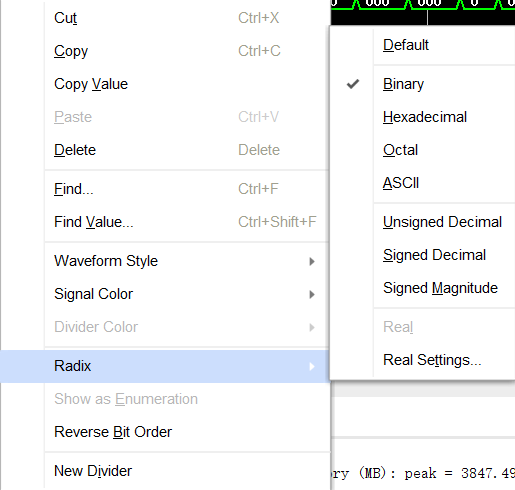
指令情况如下：

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
3. module cpu\_sim;
5. // Inputs
6. reg CLK;
7. reg RST;
8. reg [31:0] outside\_pc;
9. // Outputs
10. wire [31:0] ins, now\_pc;
12. // Instantiate the Unit Under Test (UUT)
13. Main uut (
14. .CLK(CLK),
15. .RST(RST),
16. .outside\_pc(outside\_pc),
17. .ins(ins),
18. .now\_pc(now\_pc)
19. );
21. initial
22. begin
23. // Initialize Inputs
24. CLK = 0;
25. RST = 1;
26. outside\_pc = 0; // 这里设置外部pc
27. #5; // 刚开始设置pc为0
28. CLK = !CLK;
29. #5;
30. RST = 0;
32. forever #10 begin // 产生时钟信号
33. CLK = !CLK;
34. end

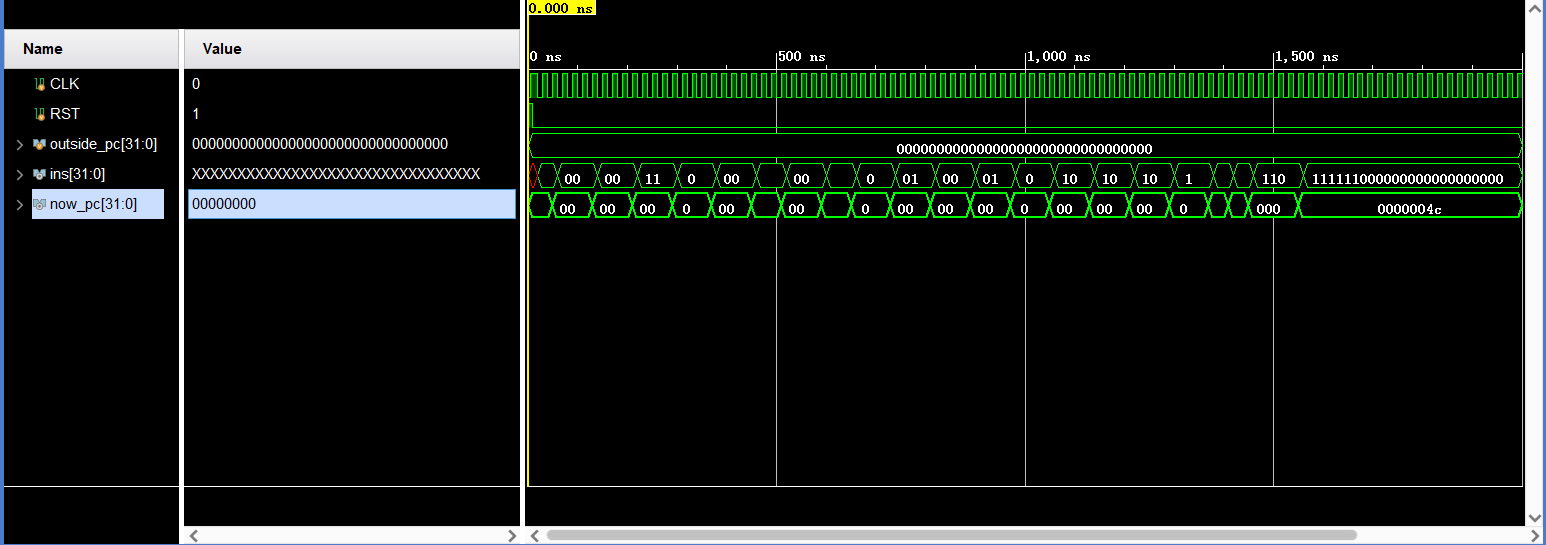
37. end

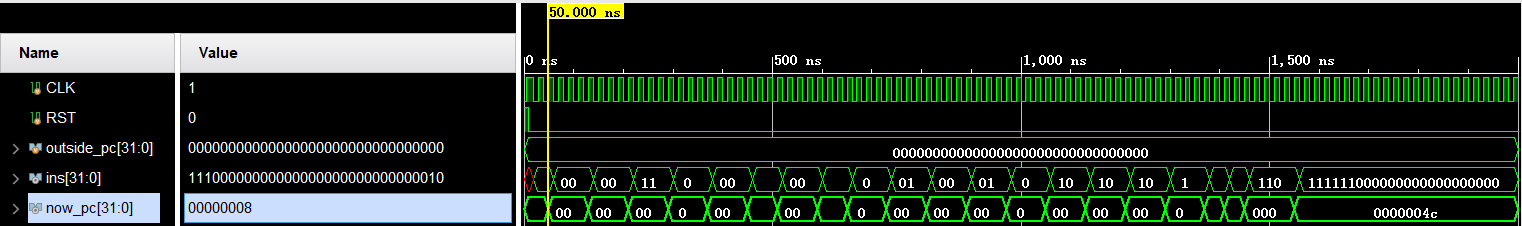
40. endmodule

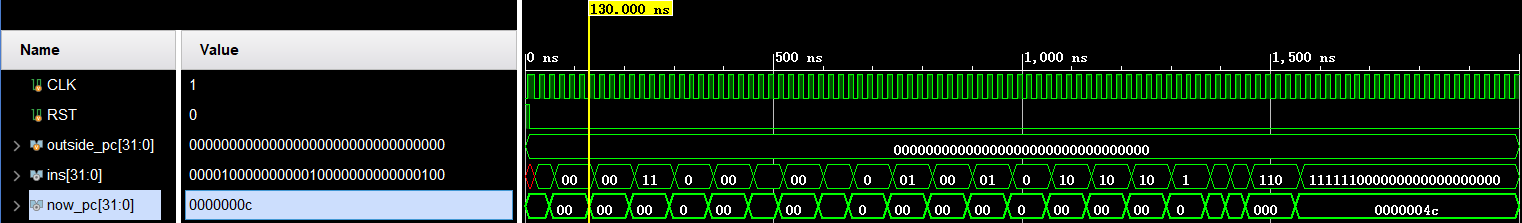
由于设计仿真代码的时候是时钟信号是#10一次,为了方便观察总体波形，我们再运行1000ns即可。

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。

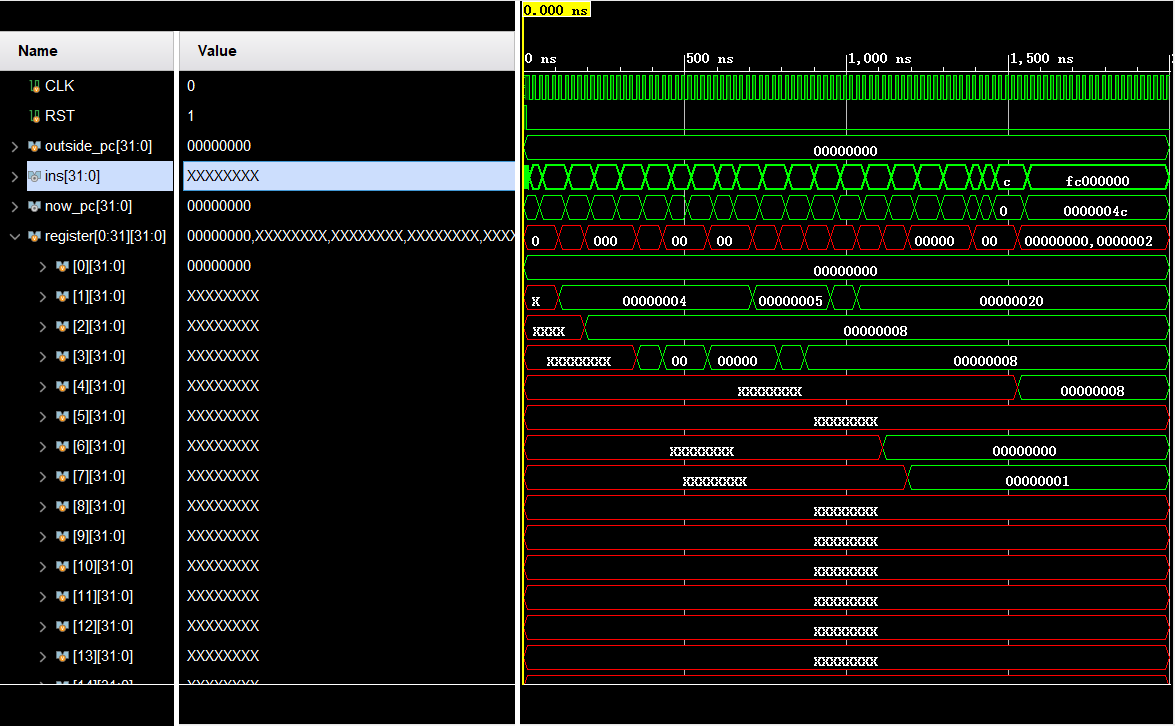
## 5.2 仿真波形分析

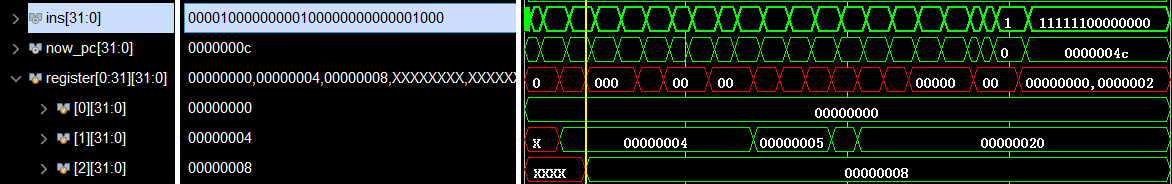
仿真情况与对应分析如下：

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00001000000000010000000000000100，对应第三条指令addi $1,$0,4

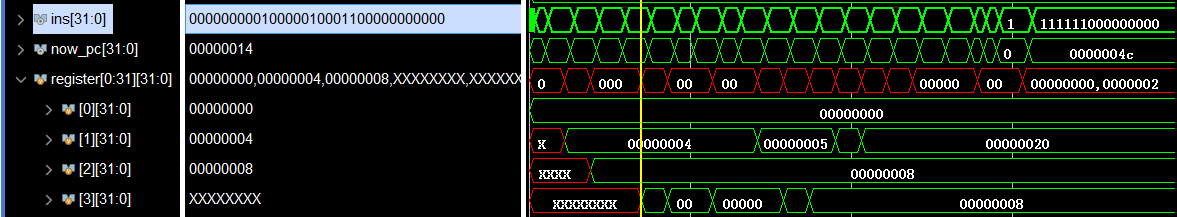
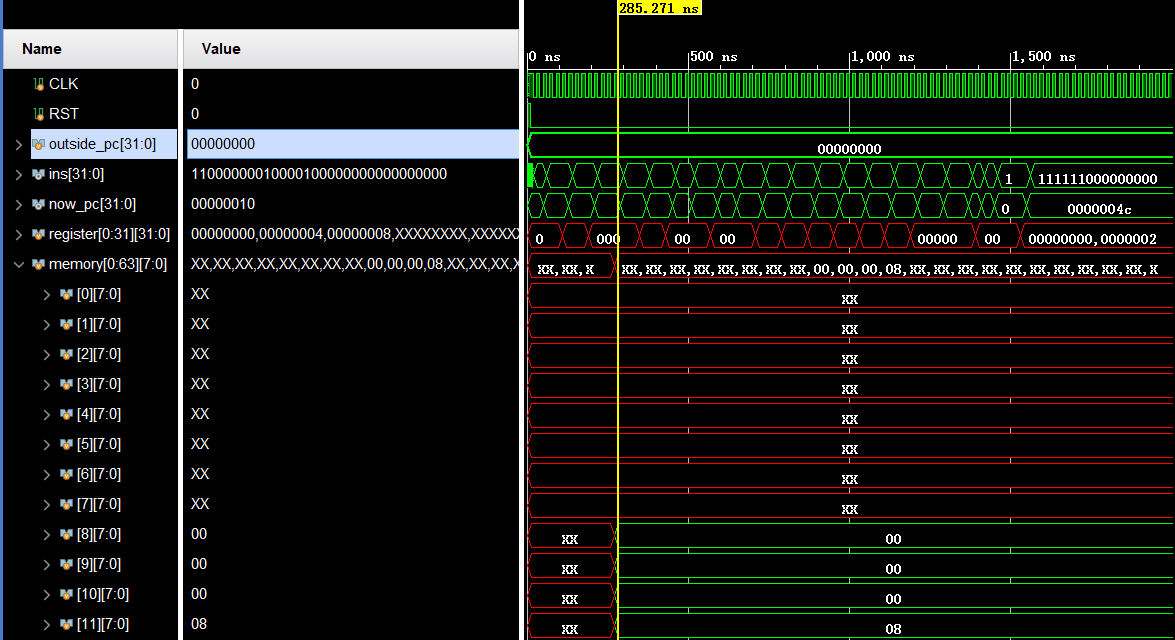
也就是将立即数4作位拓展之后与rs$0相加放入$1，为了方便我们观察寄存器值的变化，以此来检验指令的执行情况，我在scope里面将regster放入并刷新，得到了寄存器的变化波形

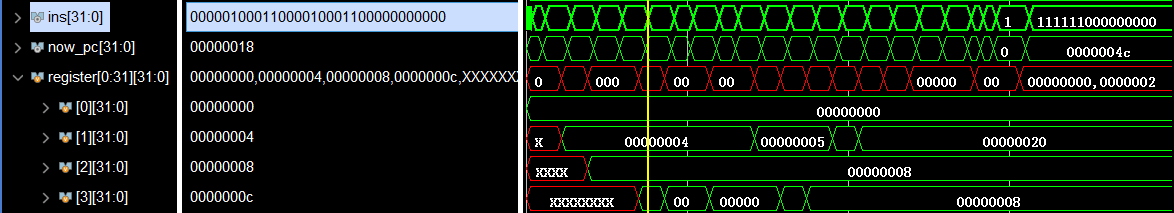
第二条指令，执行完毕后，一号寄存器的值变为了4

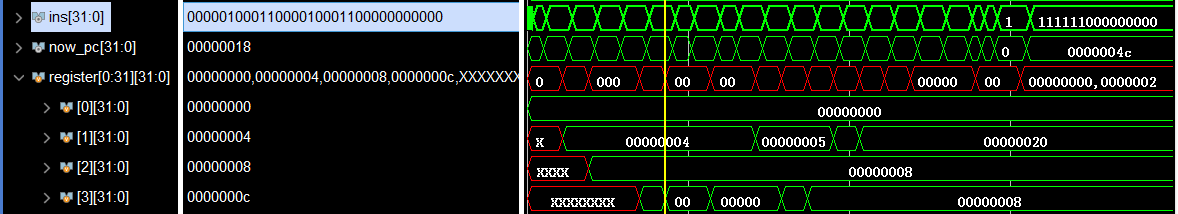
第三条指令addi $2,$0,8,对应的32位二进制位ins为00001000000000100000000000001000，可以看到，二号寄存器的值变为8，

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。

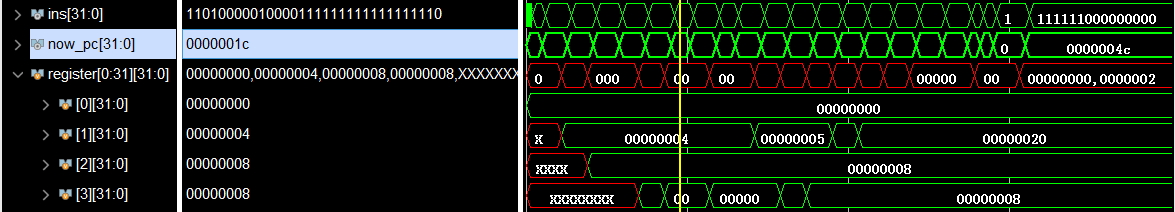
此处涉及到mem，为了方便我们观察和比较，我将memory也加进来，刷新，观察其value，果然，其内容变为8($2)

第五条指令，add $3,$2,$1,对应的二进制指令为00000000010000010001100000000000

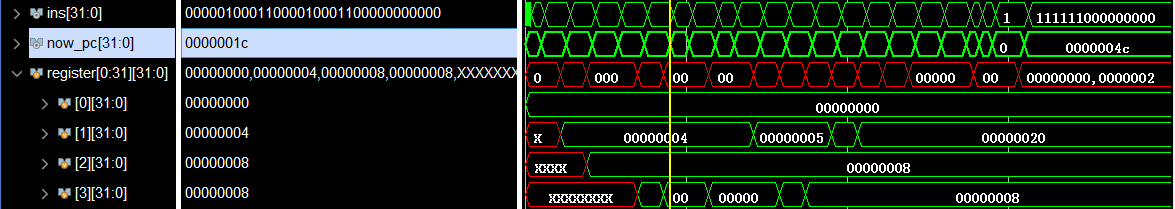
执行完成后，三号寄存器的值变为12

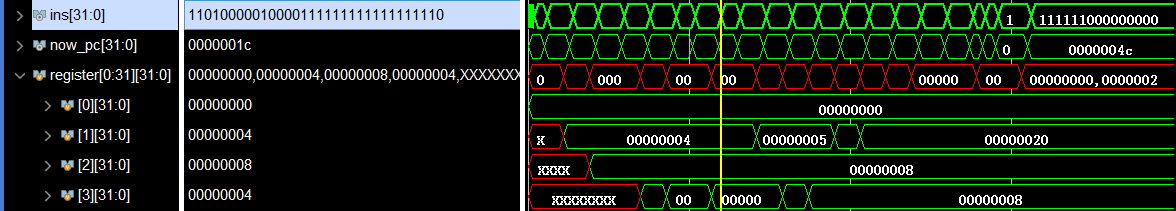
第六条指令，sub $3,$3,$1,对应的二进制指令为00000100011000010001100000000000

执行完成后，三号寄存器的值变为8

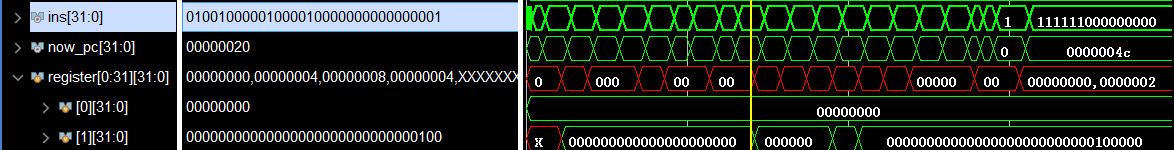
第七条指令，beq $2,$3,-2,对应的二进制指令为11010000010000111111111111111110

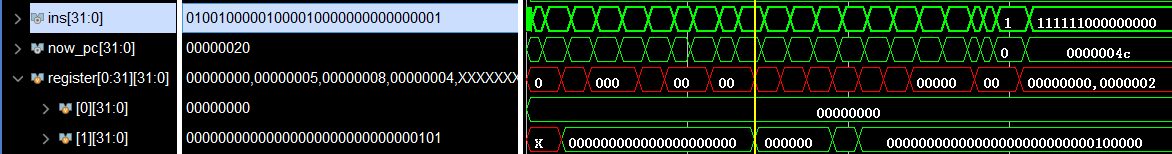
因为$2,$3值相等，都为8，所以跳转到指令sub $3,$3,$1

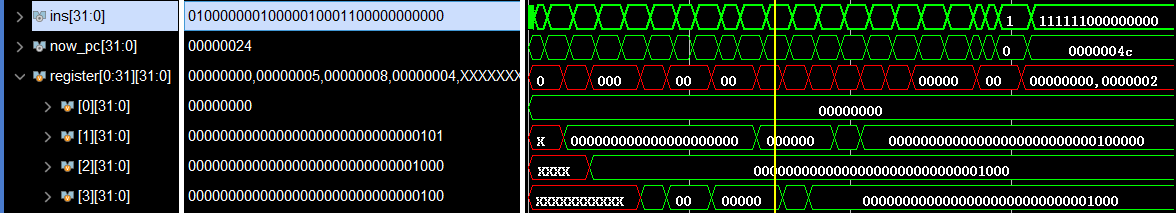
第八条指令sub $3,$3,$1，对应二进制为00000100011000010001100000000000

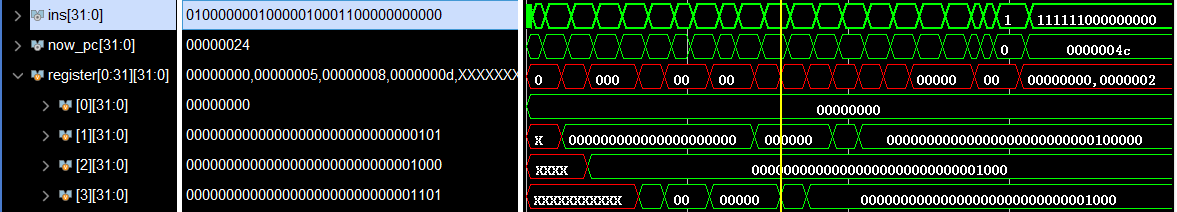
执行完毕后，三号寄存器的值变为4

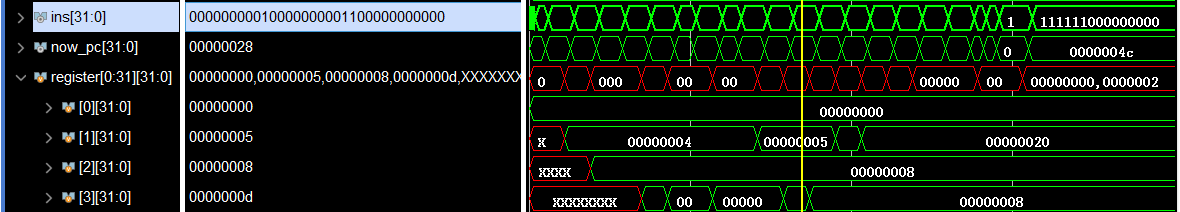
指令九，再次执行beq $2,$3,-2，对应二进制为11010000010000111111111111111110，此时二号寄存器与三号寄存器的值不相等，所以不跳转，继续执行下一条指令ori $1,$1,1。

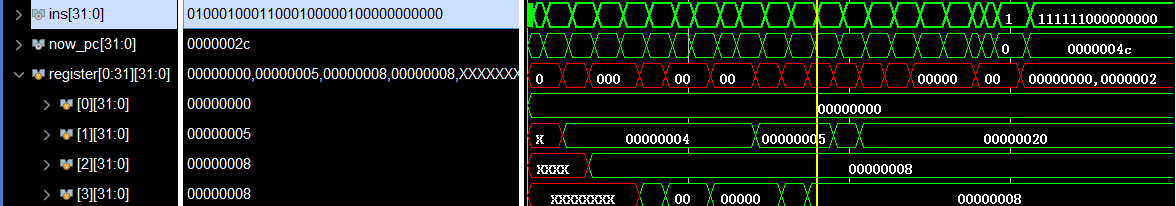
指令十，ori $1,$1,1，对应的二进制是01001000001000010000000000000001，

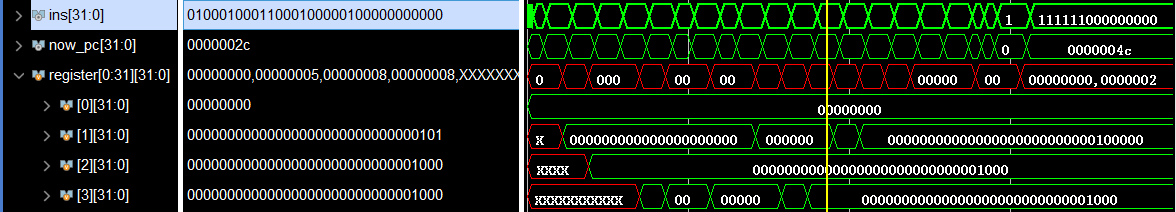
与1做完或运算后，得到0100|0101->0101

指令十一，是or$3,$2,$1,对应的二进制是01000000010000010001100000000000

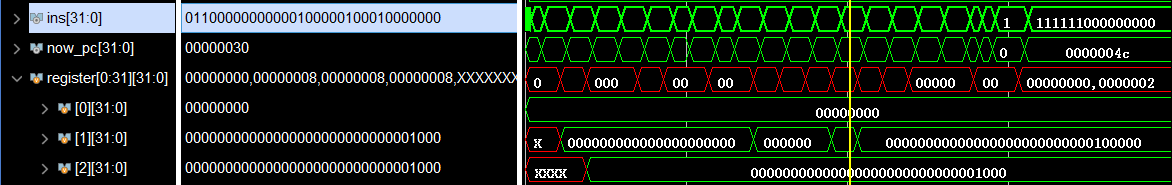
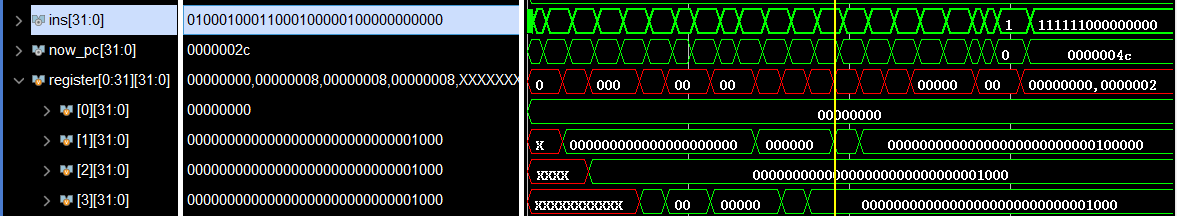
执行完毕后，三号寄存器值变为一号和二号寄存器值的或运算结果。

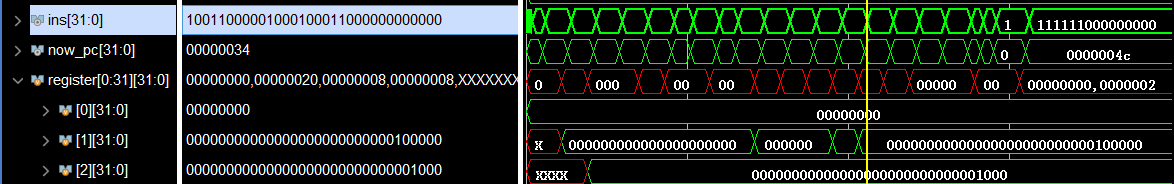
指令十二，add $3,$2,$0，对应二进制为00000000010000000001100000000000，

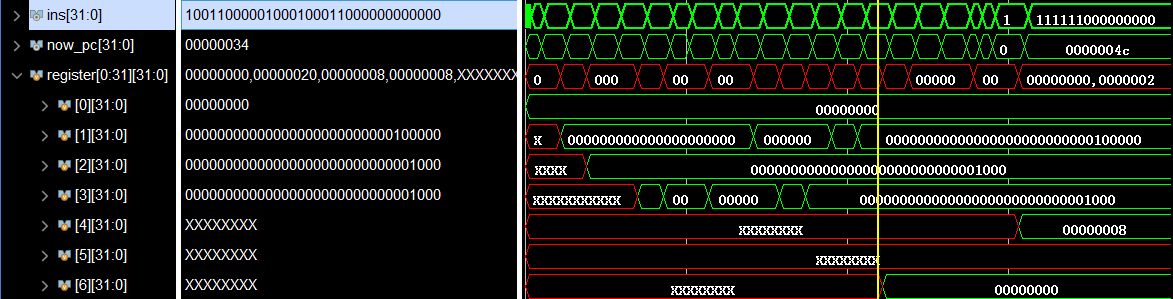
执行完毕后，

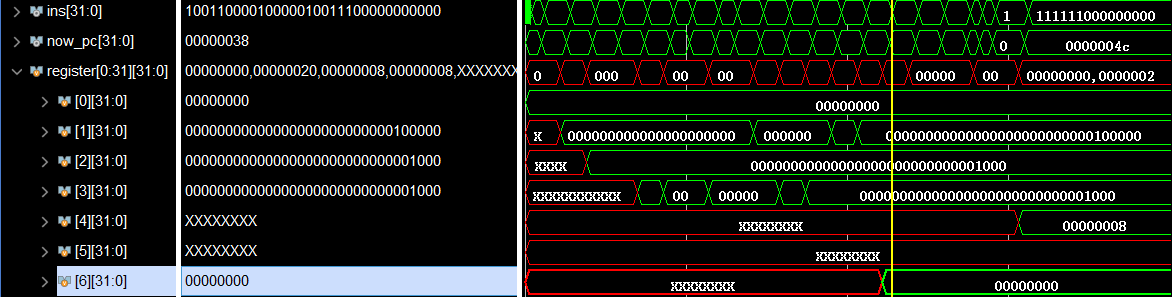
指令十三，and $1,$3,$2,对应的二进制为01000100011000100000100000000000

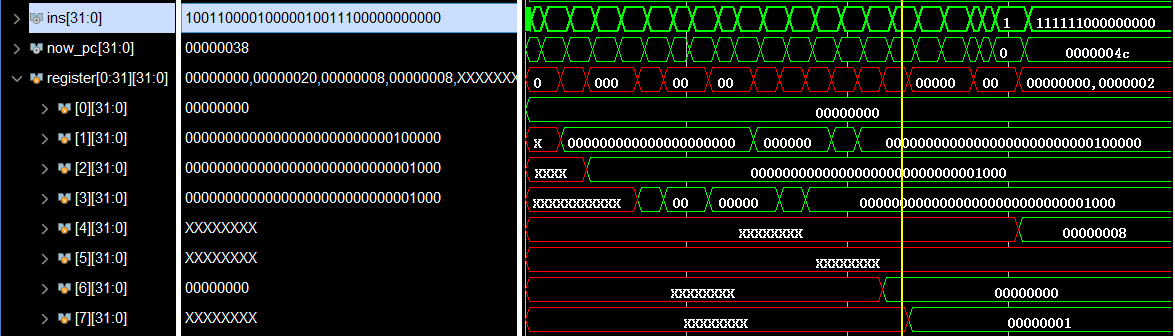
执行完毕后，

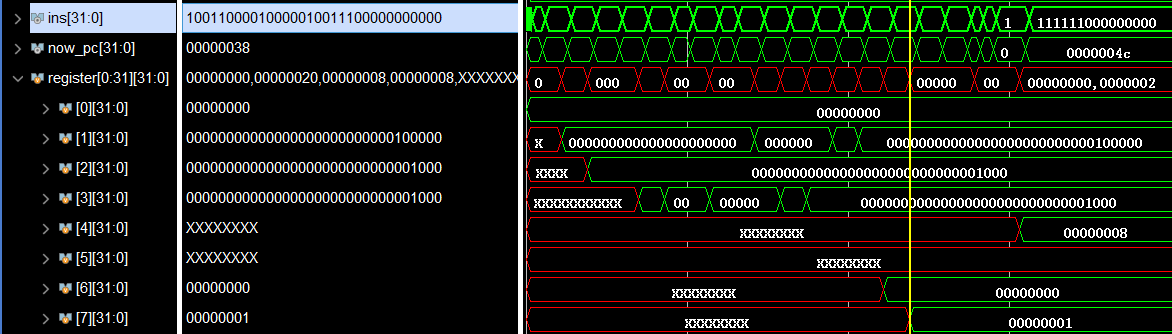
指令十四，sll $1,$2,2,对应二进制为01100000000000100000100010000000，

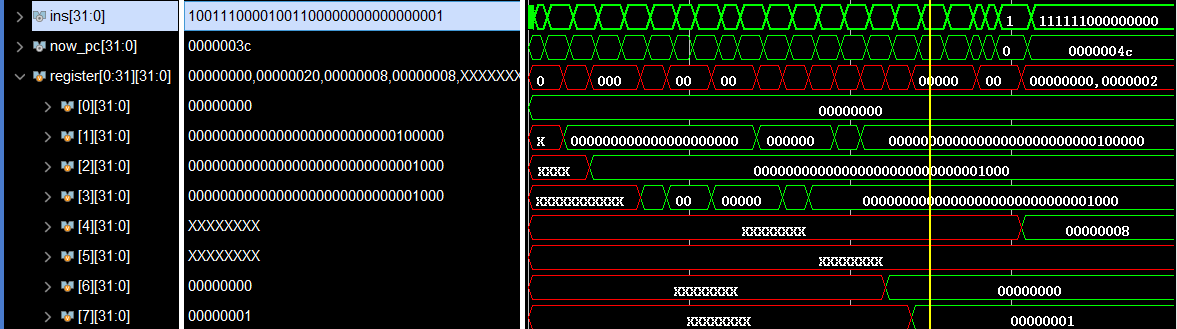
将二号寄存器值左移2位，值给一号寄存器，执行完毕后，得到结果：

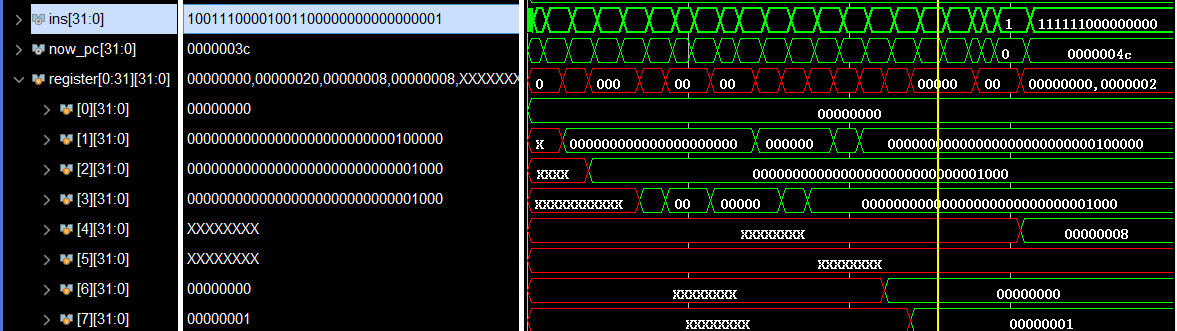
指令十五，slt $6,$1,$2,对应的二进制数是10011000001000100011000000000000，因为一号寄存器的值大于二号寄存器的值，

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。

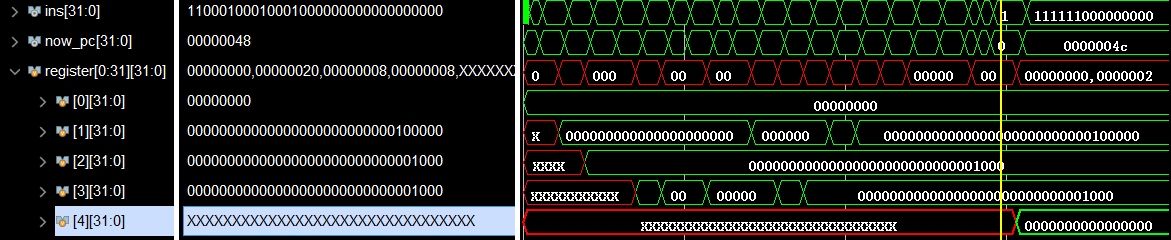
指令十八，slti $7,$6,1,因为六号寄存器的值小于1，

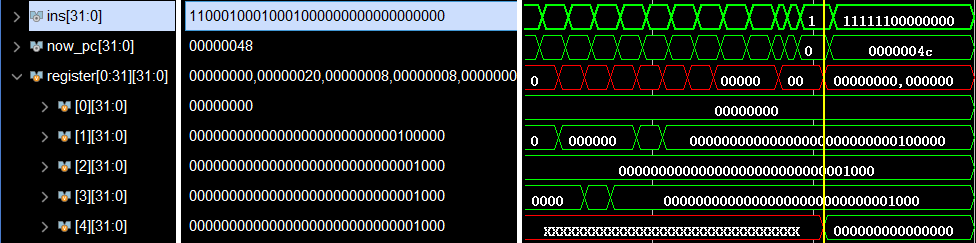
所以$7的值仍为1，

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，

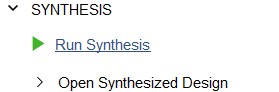
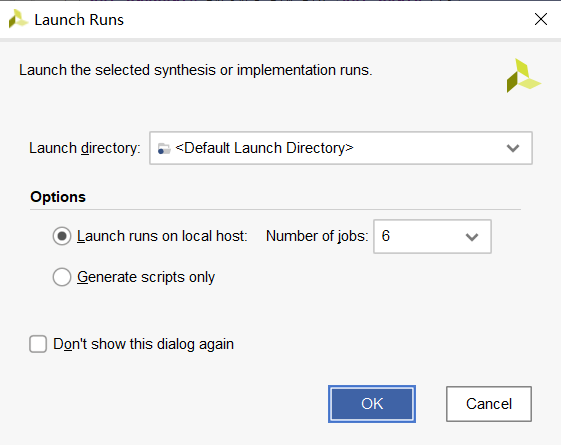
最后执行指令二十二，停机指令，halt，对应二进制为11111100000000000000000000000000

# 06 下载到FPGA与分析

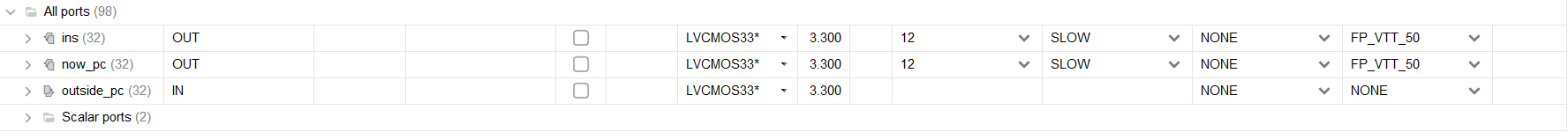
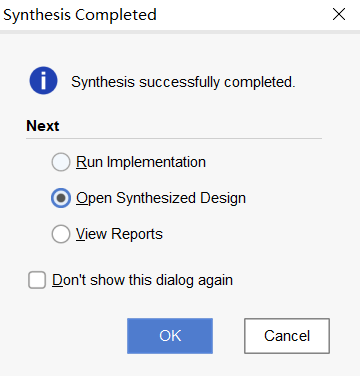
这一部分的难度我感觉应该是整个实验中最大的，因为没有任何的教程，所有需要的相关知识都需要我们自己去检索，所以光是研究怎么去下板就花费了我一两天的时间，至于后续成功下板过程中遇到的问题更是数不胜数，这里先选有一定价值的部分做介绍，最后再附录一些有趣的小问题的介绍。

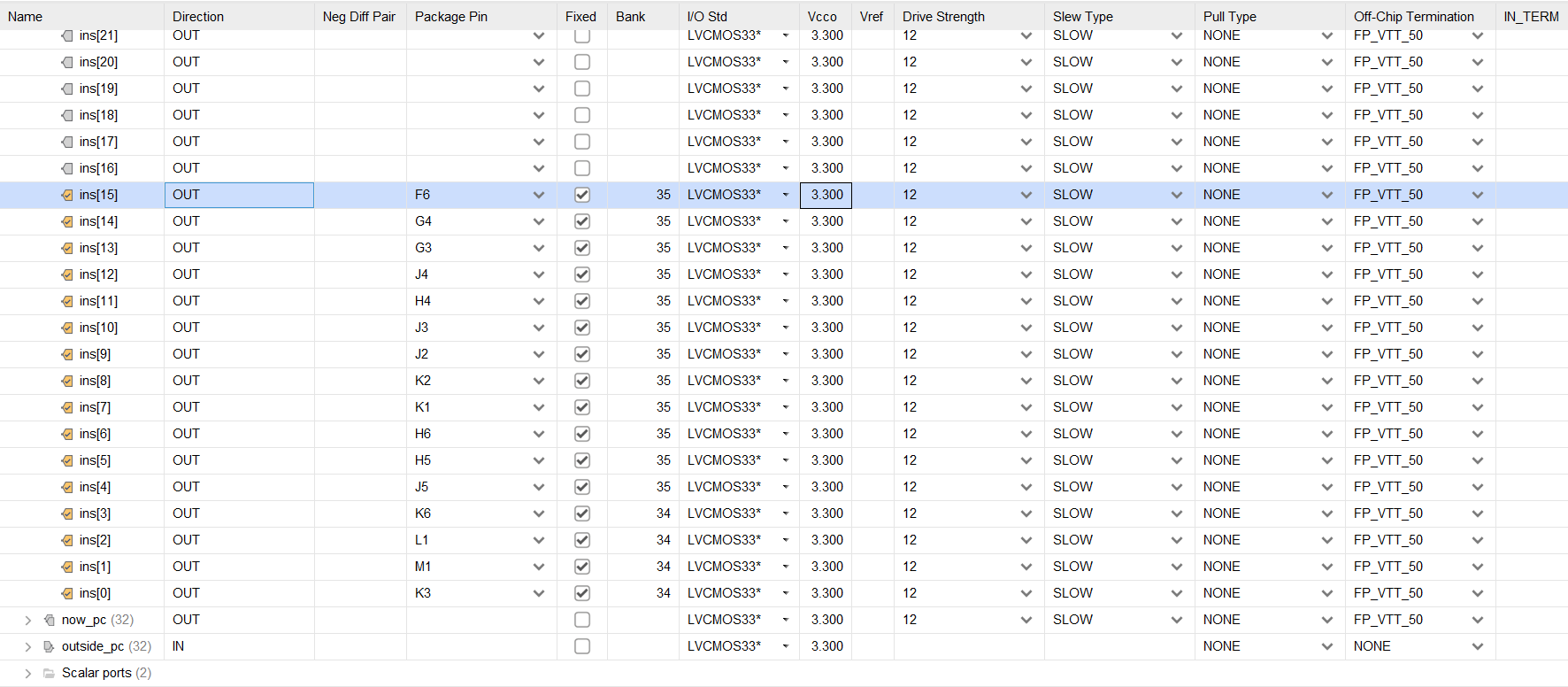
## 6.1 下板步骤

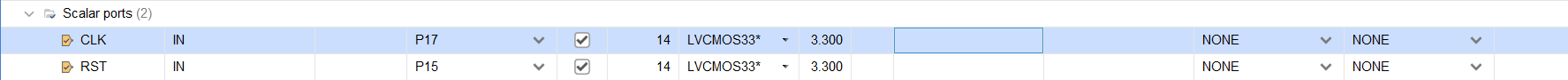
需要解释的是，我没有选择编写约束代码，而是通过可视交互的引脚赋值来是实现的，具体步骤如下：

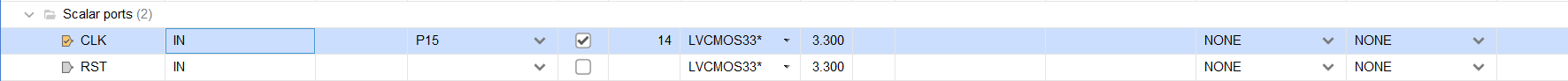
* 运行SYSTHESIS下的Run Synthesis
* 弹窗都是默认值，但是这一部分我有略作了解，这一节最后有趣的小问题部分我将介绍

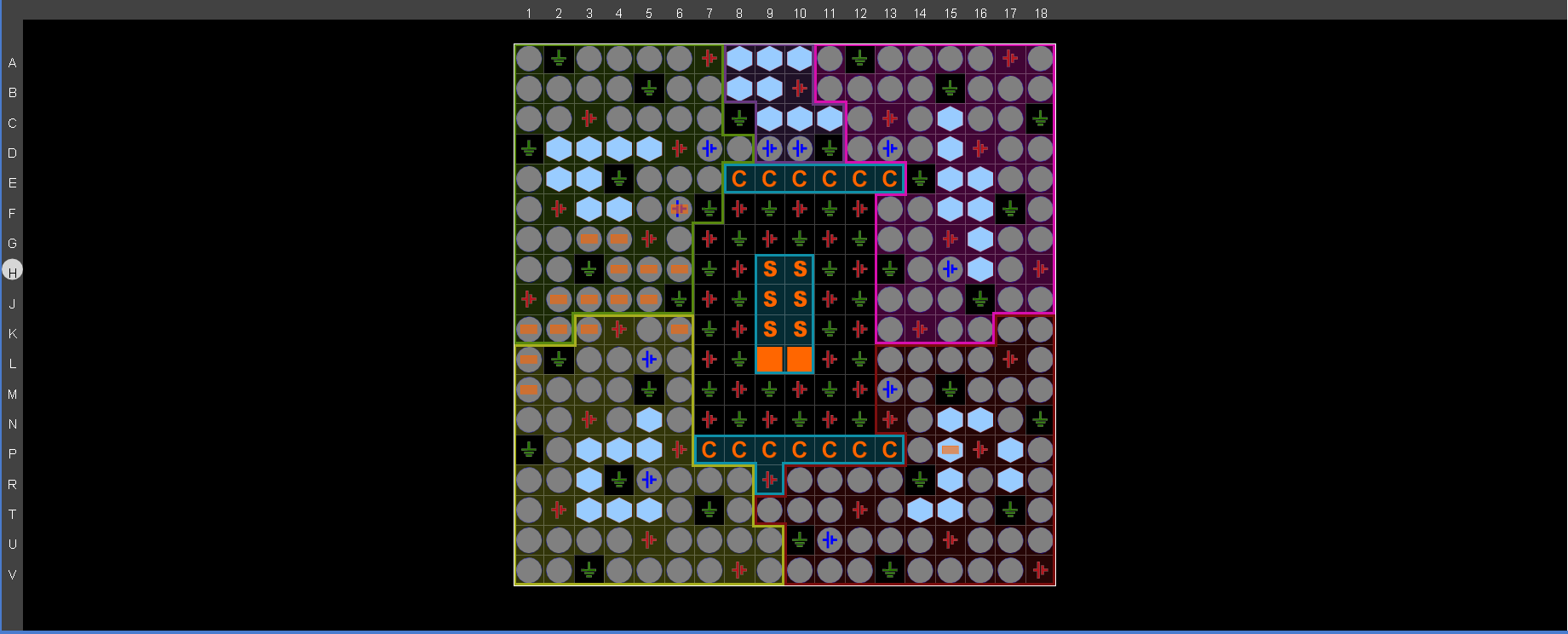
随后耐心等待即可

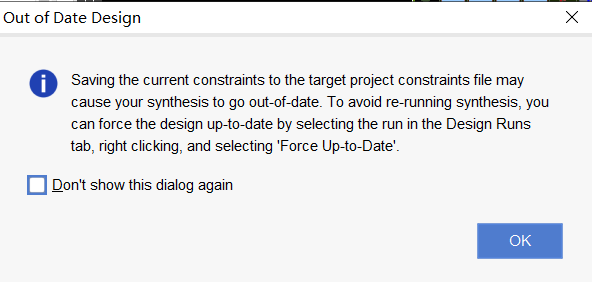
* 运行完后，我们选择Open Synthesis Design进行引脚分配
* 在随后的I/O ports界面

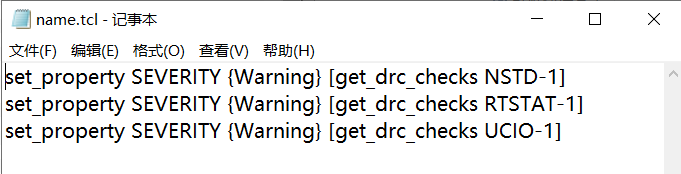
为了能在FPGA上动态的显示，我们可以选择把LED灯的引脚分配给ins或者是now\_pc,为了与我们之前仿真波形和指令更好的进行比较和对应，我们选择ins，由于指令都是32位，但是FPGA板上只有16个LED灯，所以我们干脆就将ins的低16位对应显示在16个LED灯上，并且调整I/O Std到3.3v，对应第一部分对FPGA引脚的介绍，最终的分配情况如下：

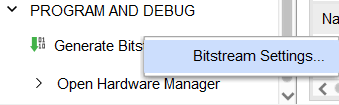
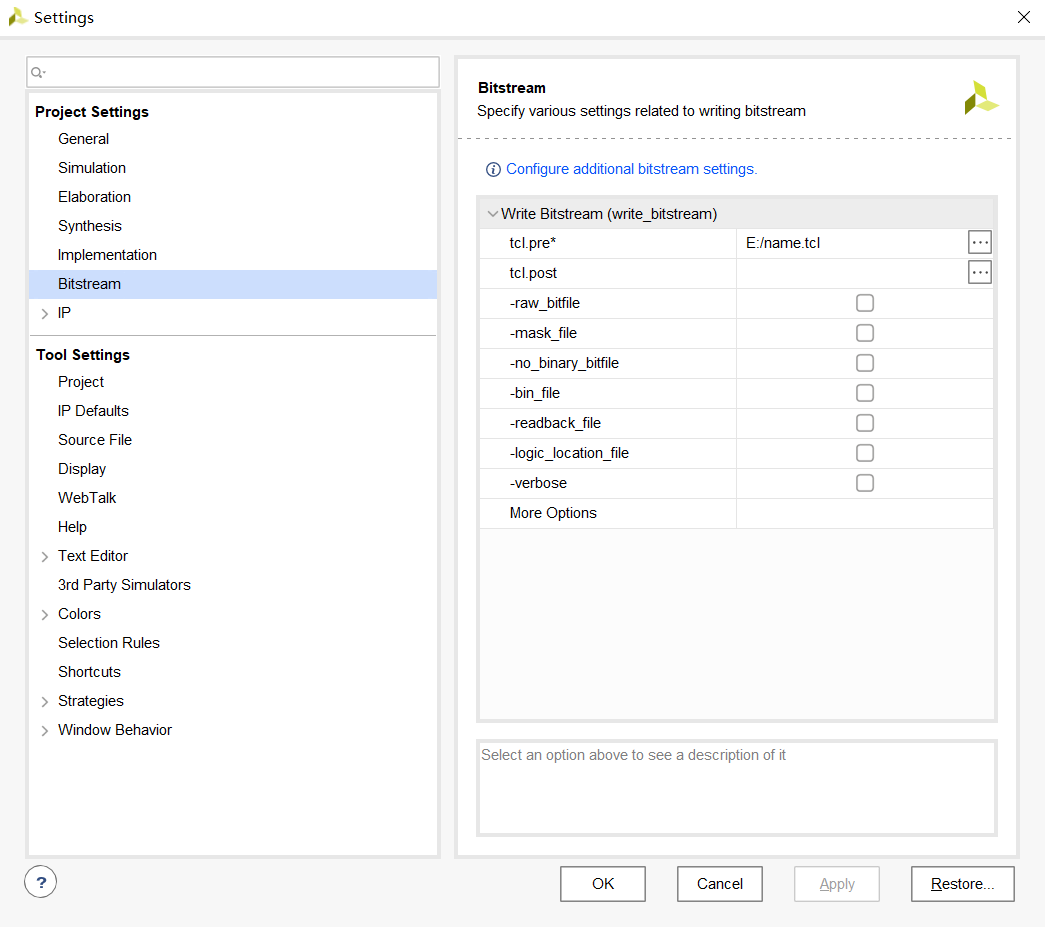
* 在Scalar ports的引脚分配时，我研究发现有两种分配方式，一种是使用系统默认的时钟，给CLK P17,RST P15 ，这种方式下FPGA的时钟信号有板内置的时钟发生，运行速度很快，不方便我们观察和验收。

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。

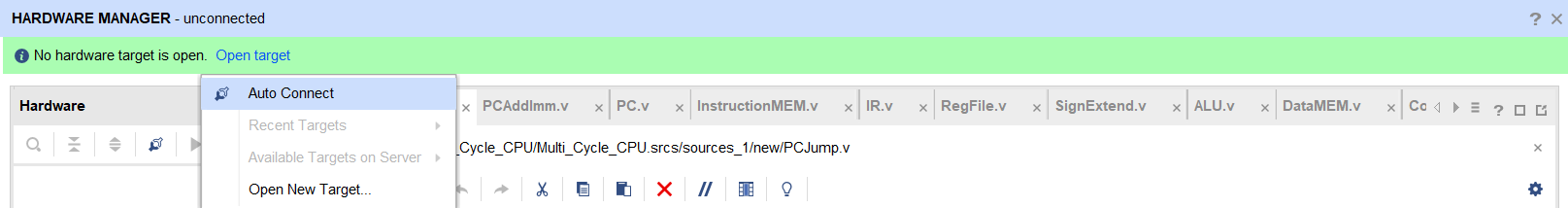
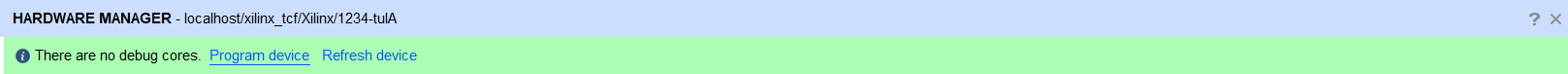
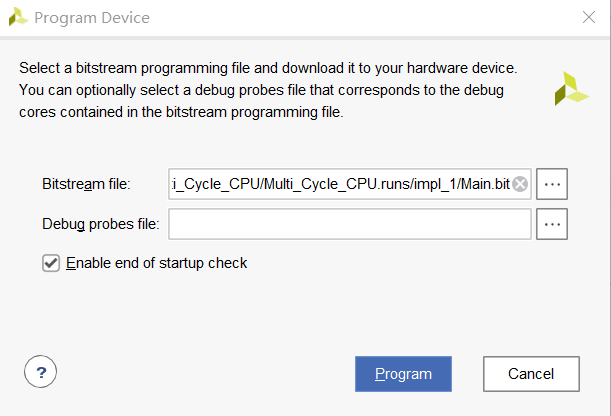
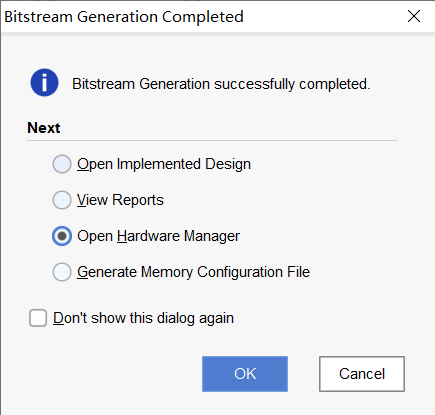
完成所有的分配后，我们可以看到I/O Planning如下：

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。

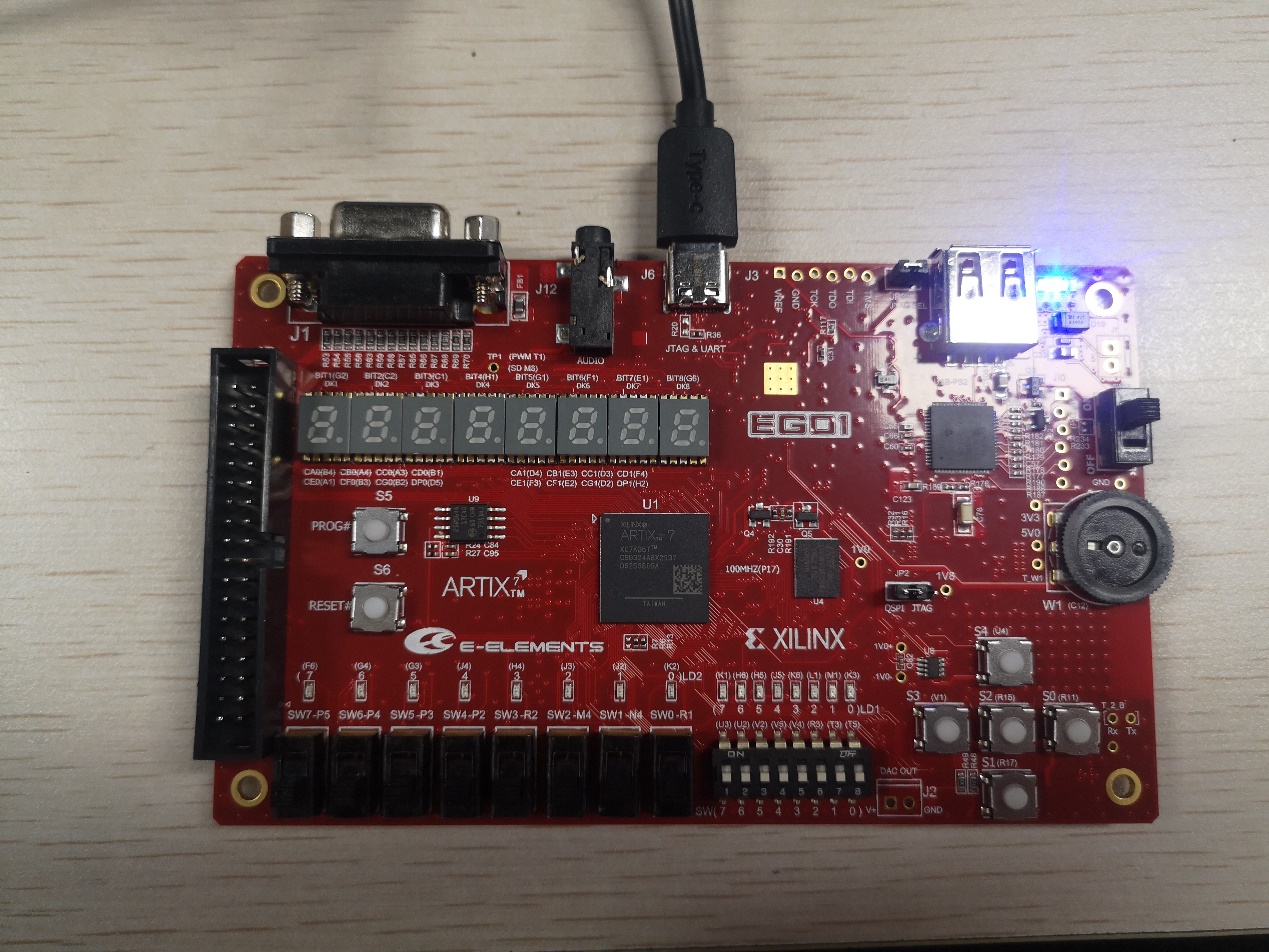
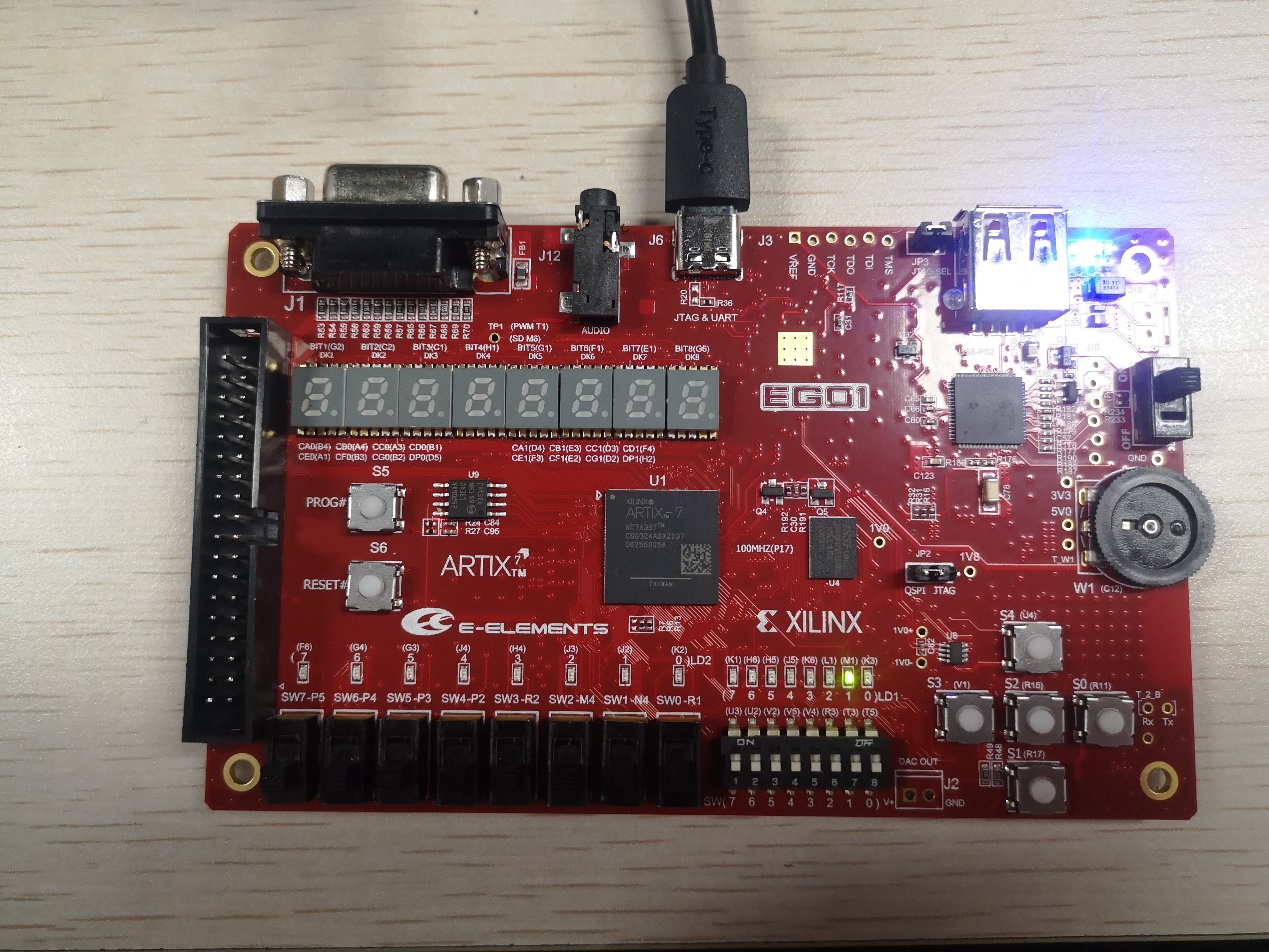
我们点击Program Device就能看到运行了。

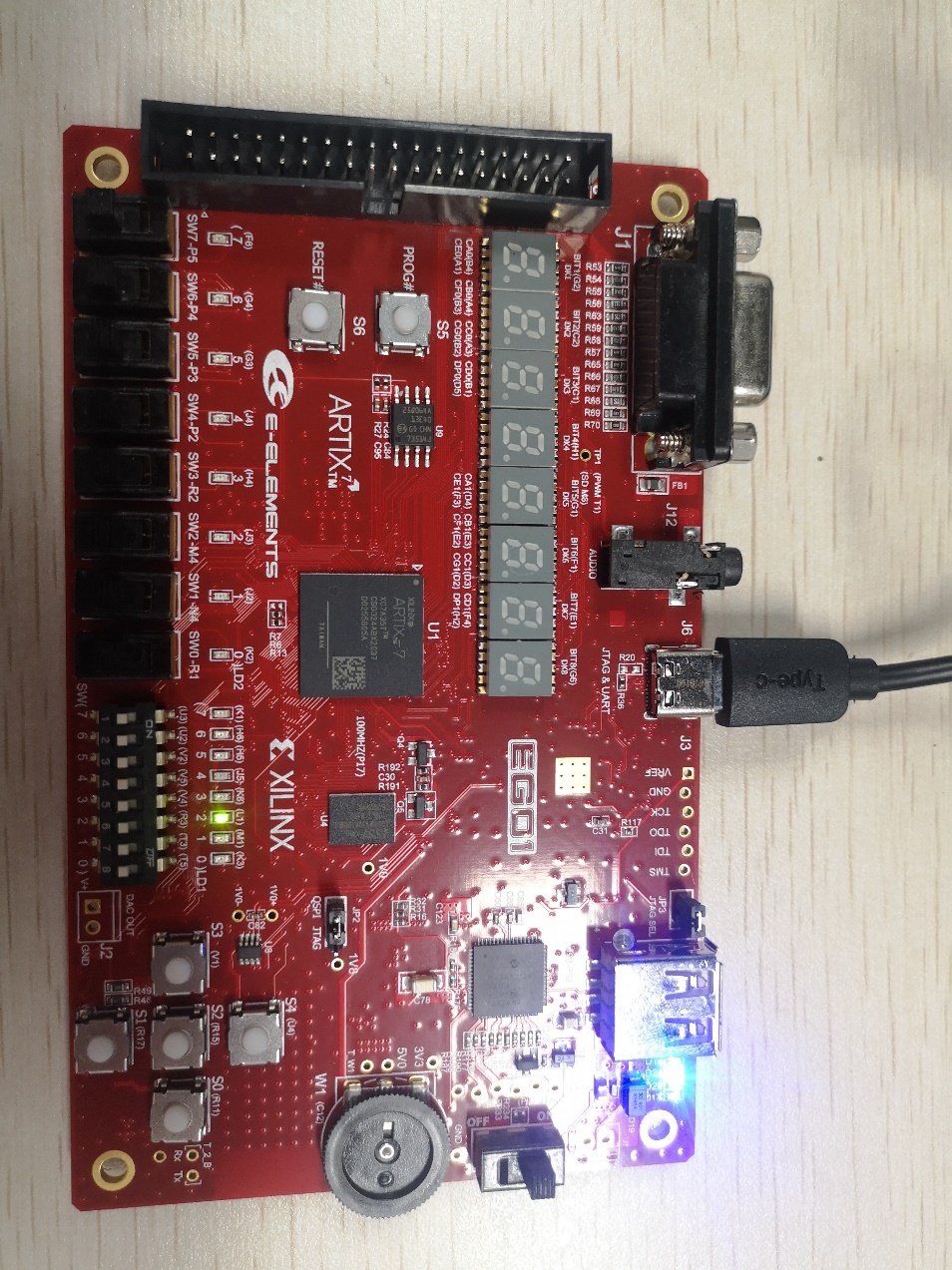
约束代码cpu\_test1.xdc如下：

1. set\_property PACKAGE\_PIN P15 [get\_ports CLK]
2. set\_property IOSTANDARD LVCMOS33 [get\_ports CLK]
3. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[0]}]
4. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[31]}]
5. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[30]}]
6. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[29]}]
7. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[28]}]
8. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[27]}]
9. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[26]}]
10. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[25]}]
11. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[24]}]
12. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[23]}]
13. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[22]}]
14. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[21]}]
15. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[20]}]
16. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[19]}]
17. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[18]}]
18. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[17]}]
19. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[16]}]
20. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[15]}]
21. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[14]}]
22. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[13]}]
23. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[12]}]
24. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[11]}]
25. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[10]}]
26. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[9]}]
27. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[8]}]
28. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[7]}]
29. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[6]}]
30. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[5]}]
31. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[4]}]
32. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[3]}]
33. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[2]}]
34. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[1]}]
35. set\_property IOSTANDARD LVCMOS33 [get\_ports {now\_pc[0]}]
36. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[31]}]
37. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[30]}]
38. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[29]}]
39. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[28]}]
40. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[27]}]
41. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[26]}]
42. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[25]}]
43. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[24]}]
44. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[23]}]
45. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[22]}]
46. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[21]}]
47. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[20]}]
48. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[19]}]
49. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[18]}]
50. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[17]}]
51. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[16]}]
52. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[15]}]
53. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[14]}]
54. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[13]}]
55. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[12]}]
56. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[11]}]
57. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[10]}]
58. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[9]}]
59. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[8]}]
60. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[7]}]
61. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[6]}]
62. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[5]}]
63. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[4]}]
64. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[3]}]
65. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[2]}]
66. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[1]}]
67. set\_property IOSTANDARD LVCMOS33 [get\_ports {outside\_pc[0]}]
68. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[1]}]
69. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[2]}]
70. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[3]}]
71. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[4]}]
72. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[5]}]
73. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[6]}]
74. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[7]}]
75. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[8]}]
76. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[9]}]
77. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[10]}]
78. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[11]}]
79. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[15]}]
80. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[14]}]
81. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[12]}]
82. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[13]}]
83. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[31]}]
84. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[30]}]
85. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[29]}]
86. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[28]}]
87. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[27]}]
88. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[26]}]
89. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[25]}]
90. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[24]}]
91. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[23]}]
92. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[22]}]
93. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[21]}]
94. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[20]}]
95. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[19]}]
96. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[18]}]
97. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[17]}]
98. set\_property IOSTANDARD LVCMOS33 [get\_ports {ins[16]}]
100. set\_property IOSTANDARD LVCMOS33 [get\_ports RST]
102. set\_property PACKAGE\_PIN K3 [get\_ports {ins[0]}]
103. set\_property PACKAGE\_PIN M1 [get\_ports {ins[1]}]
104. set\_property PACKAGE\_PIN L1 [get\_ports {ins[2]}]
105. set\_property PACKAGE\_PIN K6 [get\_ports {ins[3]}]
106. set\_property PACKAGE\_PIN J5 [get\_ports {ins[4]}]
107. set\_property PACKAGE\_PIN H5 [get\_ports {ins[5]}]
108. set\_property PACKAGE\_PIN H6 [get\_ports {ins[6]}]
109. set\_property PACKAGE\_PIN K1 [get\_ports {ins[7]}]
110. set\_property PACKAGE\_PIN K2 [get\_ports {ins[8]}]
111. set\_property PACKAGE\_PIN J2 [get\_ports {ins[9]}]
112. set\_property PACKAGE\_PIN J3 [get\_ports {ins[10]}]
113. set\_property PACKAGE\_PIN H4 [get\_ports {ins[11]}]
114. set\_property PACKAGE\_PIN J4 [get\_ports {ins[12]}]
115. set\_property PACKAGE\_PIN G3 [get\_ports {ins[13]}]
116. set\_property PACKAGE\_PIN G4 [get\_ports {ins[14]}]
117. 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。

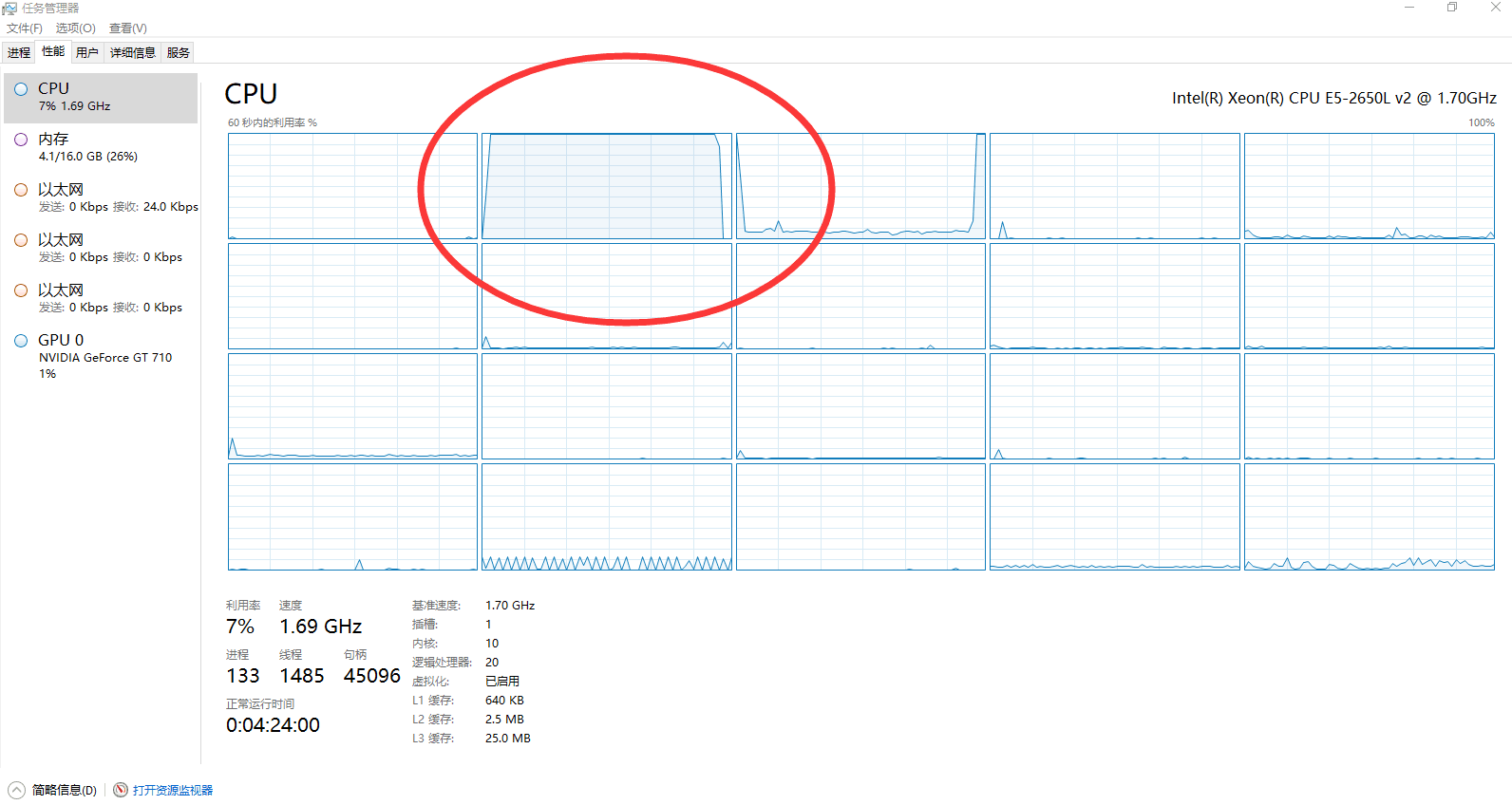
## 6.2 有趣的小问题

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。