

实验 4 : 单周期CPU设计

实验 4 : 单周期CPU设计

Part 1 基本信息

小组成员信息

声明

工程哈希

Part 2 实验目的及原理

实验目的

译码真值表

主控制模块

指令类型译码

控制信号译码

ALU控制模块

ALU操作类型译码

ALU控制信号译码

组织结构图

Part 3 模块实现

主要功能模块

N位二选一模块 single_mux #(N)

32位加法器模块 single_add #(N)

PC程序计数器模块 single_pc #(N)

PC+4模块 single_pc_plus_4

寄存器堆模块 single_gpr

控制模块 single_ctrl

ALU控制模块 single_alu_ctrl

16位至32位带符号扩展模块 single_signext

ALU模块 single_alu

调试验证模块

100ms时钟模块 clk_100ms #(DIV_1s)

防抖模块 pedebounce

16位四选一模块 MUX4to1b2

时钟计数器模块 clkdiv

数字显示模块 dispnum

顶层模块以及引脚约束

顶层模块

引脚约束

Part 4 实验验证

实验验证使用的代码

实验验证使用的数据存储器的初始值：

物理测试结果

运算结果

\$t*寄存器的使用

表格A

表格B

COE文件路径

Part 1 基本信息

小组成员信息

序号	姓名	学号
1	钟添芸	3180103009
2	江昊翰	3180101995
3	徐震	3180105004

序号	姓名	学号
4	麦昌楷	3180101982

声明

- 本工程目标平台与物理测试平台为Sword Kintex7实验平台
- 本工程主要代码采用Verilog编写，同时辅助有Schematic图形
- 本工程综合编译环境为ISE-14.7 (nt64)
- 本工程Simulation环境为Windows10 (64bit)

工程哈希

6be1cb7127e3a22087181a97c37d3d02

Part 2 实验目的及原理

实验目的

1. 实现一个单周期CPU，提供对MIPS常见指令集的支持，包括部分R型语句、lw、sw、beq以及j语句。
2. 在实验板上验证该单周期CPU的正确性。

译码真值表

主控制模块

指令类型译码

信号	R型	lw	sw	beq	j
OP[5]	0	1	1	0	0
OP[4]	0	0	0	0	0
OP[3]	0	0	1	0	0
OP[2]	0	0	0	1	0
OP[1]	0	1	1	0	1
OP[0]	0	1	1	0	0

控制信号译码

指令类型	RegDst	ALUsrc	MemtoReg	RegWrite	MemWrite	MemRead	Branch	Jump	ALUop
R	1	0	0	1	0	0	0	0	10
lw	0	1	1	1	0	1	0	0	00
sw	x	1	x	0	1	0	0	0	00
beq	x	0	x	0	0	0	1	0	01
j	x	x	x	x	x	x	x	1	xx

ALU控制模块

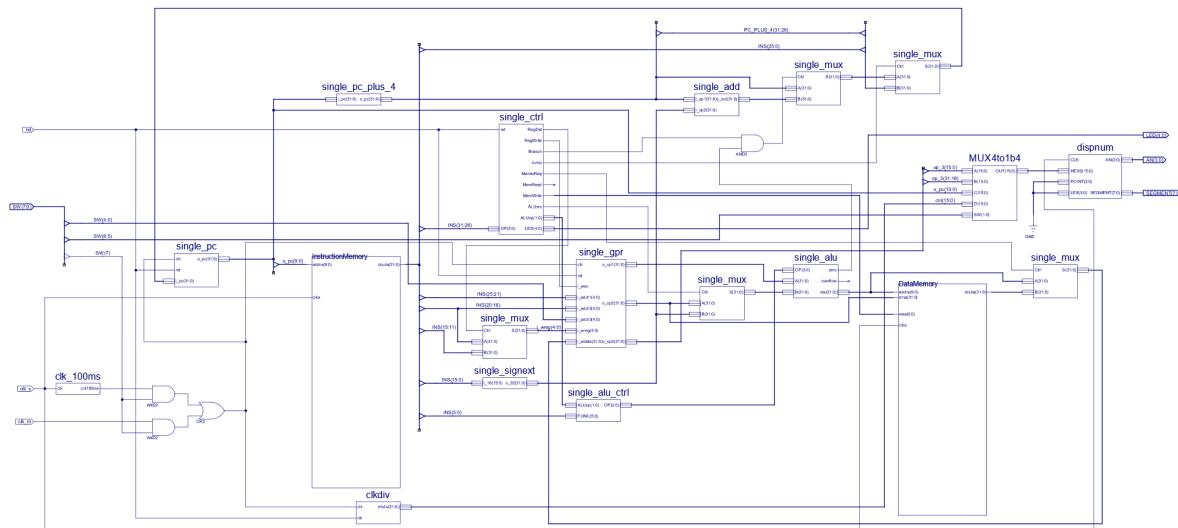
ALU操作类型译码

操作	ALUop[1]	ALUop[0]	FUNC[5]	FUNC[4]	FUNC[3]	FUNC[2]	FUNC[1]	FUNC[0]
加	0	0	x	x	x	x	x	x
减	x	1	x	x	x	x	x	x
加	1	x	x	x	0	0	0	0
减	1	x	x	x	0	0	1	0
与	1	x	x	x	0	1	0	0
或	1	x	x	x	0	1	0	1
小于置1	1	x	x	x	1	0	1	0

ALU控制信号译码

操作	OP[3]	OP[2]	OP[1]	OP[0]
加	0	0	1	0
减	0	1	1	0
与	0	0	0	0
或	0	0	0	1
小于置1	0	1	1	1

组织结构图



Part 3 模块实现

主要功能模块

N位二选一模块 single_mux #(N)

该模块接受两个N位的信号，通过控制信号选择其中一个输出。

```
``` verilog
1 module single_mux(
2 A,
3 B,
```

```

4 Ctrl,
5 S
6);
7
8 parameter N = 32; //默认输入输出宽度为32位，可调整
9 input wire Ctrl; //控制信号
10 input wire [N-1:0] A, B; //输入
11 output wire [N-1:0] S; //输出
12
13 assign S = (Ctrl == 1'b0) ? A : B; //控制信号置0时输出A，置1时输出B
14
15 endmodule
```

```

32位加法器模块 single_add #(N)

该模块接受两个32位的输入，并输出它们的和，用于计算beq指令的跳转地址

```

``` verilog
1 module single_add(
2 i_op1,
3 i_op2,
4 o_out
5);
6
7 input wire [31:0] i_op1, i_op2; //两个输入
8 output wire [31:0] o_out; //输出它们的和
9
10 assign o_out = i_op1 + i_op2; //单纯的加法
11
12 endmodule
```

```

PC程序计数器模块 single_pc #(N)

该模块实现程序计数器的功能，有一个参数N

```

``` verilog
1 module single_pc(
2 i_pc,
3 o_pc,
4 clk,
5 rst
6);
7
8 parameter N = 32; //参数，决定了程序计数器的位数
9
10 input wire clk, rst; //时钟信号与复位信号
11 input wire [N-1:0] i_pc; //新的PC
12 output wire [N-1:0] o_pc; //当前的PC
13 reg [N-1:0] t_pc; //存放当前PC的寄存器
14
15 assign o_pc = rst ? {N{1'b1}} : t_pc; //重置信号将PC重置到-1，这样，读取指令使用的PC将从0开始
16
17 always @(posedge clk) //每个时钟周期，更新程序计数器
18 t_pc <= i_pc;
19
20 endmodule
```

```

PC+4模块 single_pc_plus_4

```
``` verilog
1 module single_pc_plus_4(
2 i_pc,
3 o_pc
4);
5
6 parameter N = 32;
7
8 input wire [N-1:0] i_pc; //输入：PC
9 output wire [N-1:0] o_pc; //输出：PC+4
10
11 assign o_pc = i_pc + 1; //由于我们的存储器使用字地址，因此不需要+4，而是使用+1
12
13 endmodule
```
```

寄存器堆模块 single_gpr

这个模块中提供对32个寄存器的读写方法，并且提供一个额外的用于测试的接口。

```
``` verilog
1 module single_gpr(
2 clk, rst,
3 i_addr1, i_addr2, i_addr3,
4 i_wreg, i_wdata, i_wen,
5 o_op1, o_op2, o_op3
6);
7
8 input wire clk, rst, i_wen; //时钟信号、复位信号和写入使能信号
9 input wire [4:0] i_addr1, i_addr2, i_addr3, i_wreg; //三个读取地址，一个写入地址
10 input wire [31:0] i_wdata; //写入的数据
11 output wire [31:0] o_op1, o_op2, o_op3; //读取的数据
12
13 reg [31:0] mem[31:0];
14
15 assign o_op1 = mem[i_addr1]; //通过addr信号提供的地址直接取出对应的寄存器中的数据
16 assign o_op2 = mem[i_addr2];
17 assign o_op3 = mem[i_addr3];
18
19 always @(posedge clk or posedge rst)
20 if (rst == 1'b1) mem[0] = {32{1'b0}}; //复位时，将$zero重置为0x00000000
21 else if (i_wen) mem[i_wreg] = (i_wreg == {5{1'b0}}) ? {32{1'b0}} :
22 i_wdata; // $zero寄存器无法被写入
23
24 endmodule
```

```

控制模块 single_ctrl

该模块对指令INS[31:26]段译码，得到指令类型并输出各种控制信号

```
``` verilog
1 module single_ctrl(
2 OP,
3 ALUop,
4 RegDst,
```

```

5 RegWrite,
6 Branch,
7 Jump,
8 MemtoReg,
9 MemRead,
10 MemWrite,
11 ALUsrc,
12 rst,
13 LED
14);
15
16 input wire rst; //复位信号
17 input wire [5:0] OP; //操作码
18
19 //输出各种控制信号
20 output wire [1:0] ALUop; //ALU控制模块的二次译码信号
21 output wire RegDst, RegWrite, Branch, Jump, MemtoReg, MemRead, MemWrite,
22 ALUsrc;
23 output wire [4:0] LED; //将指令类型通过LED信号进行输出
24
25 wire R, LW, SW, BEQ;
26
27 //对指令类型进行译码
28 assign R = ~OP[0] & ~OP[1] & ~OP[2] & ~OP[3] & ~OP[4] & ~OP[5];
29 assign LW = OP[0] & OP[1] & ~OP[2] & ~OP[3] & ~OP[4] & OP[5];
30 assign SW = OP[0] & OP[1] & ~OP[2] & OP[3] & ~OP[4] & OP[5];
31 assign BEQ = ~OP[0] & ~OP[1] & OP[2] & ~OP[3] & ~OP[4] & ~OP[5];
32 assign J = ~OP[0] & OP[1] & ~OP[2] & ~OP[3] & ~OP[4] & ~OP[5];
33
34 //将指令类型的译码结果通过LED进行输出
35 assign LED = {J, BEQ, LW, SW, R};
36
37 //知道指令类型之后，可以参照真值表对各种信号进行译码
38 assign RegDst = ~rst & R;
39 assign ALUsrc = ~rst & SW | LW;
40 assign MemtoReg = ~rst & LW;
41 assign RegWrite = ~rst & R | LW;
42 assign MemRead = ~rst & LW;
43 assign MemWrite = ~rst & SW;
44 assign Branch = ~rst & BEQ;
45 assign ALUop = ~rst & {R, BEQ};
46 assign Jump = ~rst & J;
47 endmodule
```

```

ALU控制模块 single_alu_ctrl

该模块利用控制模块提供的ALUop以及指令INS[5:0]段进行二次译码，输出ALU模块的操作码。

```

``` verilog
1 module single_alu_ctrl(
2 ALUop,
3 FUNC,
4 OP
5);
6
7 input wire [1:0] ALUop;
8 input wire [5:0] FUNC;
9 output wire [3:0] OP;
10
11 wire LS, BEQ, R;
12 wire OP_ADD, OP_SUB, OP_AND, OP_OR, OP_SLT;
```

```

```

13
14    //利用ALUop得到指令类型
15    assign LS  = ~ALUop[0] & ~ALUop[1];
16    assign BEQ =  ALUop[0] & ~ALUop[1];
17    assign R   = ~ALUop[0] &  ALUop[1];
18
19    //得到指令类型之后，依照真值表得到ALU的操作类型
20    assign OP_ADD = LS | (~FUNC[3] & ~FUNC[2] & ~FUNC[1] & ~FUNC[0] & R);
21    assign OP_SUB = BEQ | (~FUNC[3] & ~FUNC[2] & FUNC[1] & ~FUNC[0] & R);
22    assign OP_OR  = ~FUNC[3] & FUNC[2] & ~FUNC[1] & FUNC[0] & R;
23    assign OP_AND = ~FUNC[3] & FUNC[2] & ~FUNC[1] & ~FUNC[0] & R;
24    assign OP_SLT =  FUNC[3] & ~FUNC[2] & FUNC[1] & ~FUNC[0] & R;
25
26    //已知操作类型之后，将操作类型翻译为操作码
27    assign OP[3] = 0;
28    assign OP[2] =  OP_SUB | OP_SLT;
29    assign OP[1] = ~OP_OR & ~OP_AND;
30    assign OP[0] =  OP_OR | OP_SLT;
31
32 endmodule
...

```

16位至32位带符号扩展模块 single_signext

这个模块将16位的输入带符号扩展到32位

```

``` verilog
1 module single_signext(
2 i_16,
3 o_32
4);
5
6 input wire[15:0] i_16; //输入：16位的数字
7 output reg[31:0] o_32; //输出：32位的数字
8
9 always @(i_16)
10 o_32 <= {{16{i_16[15]}}, i_16[15:0]}; //单纯的带符号扩展
11
12 endmodule
...

```

#### #### ALU模块 single\_alu

这个模块对32位的数字输入提供8种不同的运算模式，并能够判断输出是否为0

```

``` verilog
1 module single_alu(
2     OP,
3     A,
4     B,
5     res,
6     zero,
7     overflow
8 );
9
10    //两个常数，用于大小比较并置1/置0运算
11    parameter num_one  = 32'h00000001;
12    parameter num_zero = 32'h00000000;
13
14    input wire [31:0] A, B;
15    input wire [3 :0] OP;

```

```

16     output reg [31:0] res;
17     output wire zero, overflow;
18
19     wire [31:0] res_add, res_sub, res_and, res_or, res_nor, res_slt, res_xor,
      res_srl;
20
21     //一次进行所有运算
22     assign res_add = A + B;                                //加法运算
23     assign res_sub = A - B;                               //减法运算
24     assign res_and = A & B;                             //按位与
25     assign res_or = A | B;                            //按位或
26     assign res_nor = ~(A | B);                         //按位或非
27     assign res_slt = (A < B) ? num_one : num_zero;    //大小比较
28     assign res_xor = A ^ B;                           //按位异或
29     assign res_srl = B << 1;                          //左移1位
30
31     //利用操作码OP来选择输出结果
32     always @*
33         case (OP)
34             4'b0000: res = res_and;
35             4'b0001: res = res_or;
36             4'b0010: res = res_add;
37             4'b0011: res = res_xor;
38             4'b0100: res = res_nor;
39             4'b0101: res = res_srl;
40             4'b0110: res = res_sub;
41             4'b0111: res = res_slt;
42             default: res = res_add;
43         endcase
44
45     assign zero = (res == 0) ? 1 : 0; //zero信号输出结果是否为0
46 endmodule
```

```

### ### 调试验证模块

#### #### 100ms时钟模块 clk\_100ms #(DIV\_1s)

该模块提供一个1秒的时钟周期，根据参数DIV\_1S的不同，也能提供其他周期大小的时钟。

```

``` verilog
1  module clk_100ms(
2      input wire clk,
3      output reg clk100ms
4  );
5
6  parameter DIV_1S = 1000; //参数，时钟周期的大小
7
8  reg [31:0] cnt;          //时钟计数器
9
10 always @ (posedge clk)
11 begin
12     if (cnt < 50_000_000/DIV_1S) //未达到半周期，更新时钟计数器
13     begin
14         cnt <= cnt + 1;
15     end
16     else
17     begin
18         cnt <= 0;           //达到半周期，时钟计数器置0
19         clk100ms <= ~clk100ms; //更新输出的时钟信号
20     end
```

```

```
21 end
22 endmodule
```
```

防抖模块 pedebounce

该模块可以防止按按钮时产生的抖动

```
``` verilog  
1 module pbdebounce(
2 input wire clk_1ms,
3 input wire button,
4 output reg pbreg
5);
6
7 reg [7:0] pbshift;
8
9 //需要按下按键的时间充分长才能激活输出信号，松开也是同理
10 always@(posedge clk_1ms) begin
11 pbshift = pbshift << 1;
12 pbshift[0] = button;
13 if (pbshift == 8'b0)
14 pbreg=0;
15 if (pbshift == 8'hFF)
16 pbreg=1;
17 end
18 endmodule
```
```

16位四选一模块 MUX4to1b2

该模块通过SW信号来从四个输入信号种选择一个输出

```
``` verilog  
1 module MUX4to1b4(
2 A,B,C,D,SW,
3 OUT
4);
5
6 input wire [15:0] A, B, C, D;
7 input wire [1:0] SW;
8 output reg [15:0] OUT;
9
10 //利用SW来选择输出，常见操作
11 always @*
12 case (SW)
13 2'b00: OUT = A;
14 2'b01: OUT = B;
15 2'b10: OUT = C;
16 2'b11: OUT = D;
17 default: OUT = {16{1'b0}};
18 endcase
19
20 endmodule
```
```

时钟计数器模块 clkdiv

使用一个分时器来记录TICK的数量

```

``` verilog
1 module clkdiv(
2 input clk,
3 input rst,
4 output reg[31:0]clkdiv=0
5);
6 always @(posedge clk or posedge rst)begin
7 if (rst) clkdiv<=0;
8 else clkdiv <=clkdiv+1'b1;
9 end
10 endmodule
```

```

数字显示模块 dispnum

该模块将16位的数字输出到4个七段数码管上，详情请参考实验二。

顶层模块以及引脚约束

顶层模块

```

``` verilog
1 module TOP_single_CPU(
2 clk_s, //系统时钟信号
3 SW, //开关， SW[7]选择时钟（开启为系统时钟，关闭为手动时钟）， SW[6:5]选择数
 码管显示， SW[4:0]选择寄存器
4 LED, //LED输出， LED[4]: J, LED[3]:BEQ, LED[2]:LW, LED[1]:SW,
 LED[0]:R
5 SEGMENT, //七段数码管输出
6 AN, //七段数码管使能
7 K_COL, //键盘的行信号
8 K_ROW //键盘的列信号
9);
10
11 input wire clk_s;
12 input wire [7:0] SW;
13 output wire [4:0] LED;
14 output wire [7:0] SEGMENT;
15 output wire [3:0] AN;
16
17 input wire [1:0]K_COL; // 键盘的行信号
18 wire [1:0]K_COL_DE; // 键盘的行信号译码，即列信号输入
19 output wire [4:0]K_ROW;
20 wire clk_m, rst;
21 assign clk_m = ~K_COL_DE[0]; // 手动时钟信号
22 assign rst = ~K_COL_DE[1]; // 复位信号
23
24 wire clk;
25 wire [5:0] wreg;
26 wire [1:0] ALUop;
27 wire [3:0] OP;
28 wire [15:0] cnt;
29 wire [15:0] DIS;
30 wire [31:0] i_pc, o_pc, o_pc_added, pc_branch;
31 wire [31:0] INS;
32 wire [31:0] Sext_32, Sext_32_ls2;
33 wire [27:0] addr_j_ls2;
34 wire [31:0] op_1, op_2, op_3;
35 wire [31:0] A, B, ALUout, DMout;
36 wire [31:0] addr_b, addr_j;
```

```

```

37      wire [31:0] wdata_reg, wdata_mem;
38      wire RegDst, RegWrite, Branch, Jump, MemtoReg, MemRead, MemWrite,
    ALUsrc, PCBranch, zero;
39
40      wire clk1000ms;
41      wire clk1ms;
42
43
44      assign K_ROW = 5'b00000; //只是使用第一行的按钮
45
46      //分时模块，提供两种不同周期的时钟信号
47      clk_100ms #(DIV_1S(1)) c100(.clk(clk_s), .clk100ms(clk100ms));
48      clk_100ms #(DIV_1S(1000)) c1(.clk(clk_s), .clk100ms(clk1ms));
49
50      //防抖动模块，防止按钮信号因为电路接触问题带来的扰动
51      pbdebounce db1(.clk_1ms(clk1ms), .button(K_COL[0]),
    .pbreg(K_COL_DE[0]));
52      pbdebounce db2(.clk_1ms(clk1ms), .button(K_COL[1]),
    .pbreg(K_COL_DE[1]));
53
54
55      assign clk = (SW[7] == 1'b1) ? clk100ms : clk_m; //通过SW[7]可以控制程序
计数器的时钟
56      assign PCBranch = Branch & zero; //控制是否执行分支的信号
57      assign A = op_1; //ALU的第一个输入直接来
来自于寄存器堆
58      assign wdata_mem = op_2; //数据存储器的写入信号直
接来自于寄存器堆
59      assign addr_j = {o_pc[31:26], INS[25:0]}; //j指令的跳转地址的低26
位由指令给出，其余部分继承自PC
60
61      //利用指令计数器，在每一个时钟周期更新当前取出的指令
62      single_pc #(N(32)) pc(.i_pc(i_pc), .o_pc(o_pc), .clk(clk), .rst(rst));
63
64      //给PC信号+4，此处用的是字地址，因此是+1
65      single_pc_plus_4 #(N(32)) pc_plus_4(.i_pc(o_pc), .o_pc(o_pc_added));
66
67      //这是一个单接口ROM，我们通过PC来将指令取出，为了保证时序使用系统时钟
68      InstructionMemory imem(.addr(o_pc[9:0]), .clka(clk_s), .douta(INS));
69
70      //控制模块，接受指令高六位进行译码，输出各种信号，其中LED信号为测试用
71      single_ctrl ctrl(.rst(rst), .OP(INS[31:26]),
    .RegDst(RegDst), .RegWrite(RegWrite), .Branch/Branch),
    .Jump(Jump),
    .MemtoReg(MemtoReg), .MemRead(Memread),
    .MemWrite(MemWrite),
    .ALUsrc(ALUsrc), .ALUop(ALUop),
    .LED(LED));
72
73
74      //ALU控制模块，接受ALUop以及指令低六位进行译码，给出ALU的操作码
75      single_alu_ctrl alu_ctrl(.ALUop(ALUop), .FUNC(INS[5:0]), .OP(OP));
76
77      //ALU模块，通过操作码OP进行操控，可以实现8个基本功能
78      single_alu alu(.OP(OP), .A(A), .B(B), .res(ALUout), .zero(zero));
79
80      //寄存器堆，额外接受一个测试信号addr3，并输出对应的寄存器中的值，由SW进行手动控制
81      single_gpr RegFiles(.clk(clk), .rst(rst),
    .i_addr1(INS[25:21]), .i_addr2(INS[20:16]),
    .i_addr3(SW[4:0]),
    .i_wreg(wreg), .i_wdata(wdata_reg),
    .i_wen(RegWrite),
    .o_op1(op_1), .o_op2(op_2), .o_op3(op_3));
82
83
84      //这是一个单接口RAM，用作数据存储器，可以通过dina来写入。当wea为0时，为读取模式；
85      //wea为1时，为写入模式

```

```

90     DataMemory dmem(.addr(a(ALUout), .dina(wdata_mem), .wea(MemWrite),
91     .clka(clk_s), .douta(DMout));
92
93     //带符号扩展模块，将beq指令输入的地址扩展到32位
94     single_signext signext(.i_16(INS[15:0]), .o_32(Sext_32));
95
96     //各种MUX选择模块
97     single_mux #(N(5)) mux_wreg(.A(INS[20:16]), .B(INS[15:11]), .S(wreg),
98     .Ctrl(RegDst));
99     single_mux #(N(32)) mux_alui(.A(op_2), .B(Sext_32), .S(B),
100    .Ctrl(ALUsrc));
101   single_mux #(N(32)) mux_pc_b(.A(o_pc_added), .B(addr_b), .S(pc_branch),
102    .Ctrl(PCBranch));
103   single_mux #(N(32)) mux_pc_j(.A(pc_branch), .B(addr_j), .S(i_pc),
104    .Ctrl(Jump));
105   single_mux #(N(32)) mux_data(.A(ALUout), .B(DMout), .S(wdata_reg),
106    .Ctrl(MemtoReg));
107
108   //我们这里不使用+4，因此不用移位
109   // single_srl_2 #(N(32)) srl_Sext_32(.i_in(Sext_32),
110   // .o_out(Sext_32_ls2));
111   // single_srl_2 #(N(28)) srl_addr_j(.i_in(INS[25:0]),
112   // .o_out(addr_j_ls2));
113
114   //加法器模块，将PC+4的值加上偏移量来得到beq指令的跳转地址
115   single_add addr_b_addr(.i_op1(o_pc_added), .i_op2(Sext_32),
116   .o_out(addr_b));
117
118 endmodule
```

```

#### #### 引脚约束

```

``` verilog
1 # 系统时钟
2 NET "clk_s"      LOC = AC18 | IOSTANDARD = LVCMOS18;
3
4 # 拨动开关，高触发
5 NET "SW[0]"       LOC = AA10 | IOSTANDARD = LVCMOS15;
6 NET "SW[1]"       LOC = AB10 | IOSTANDARD = LVCMOS15;
7 NET "SW[2]"       LOC = AA13 | IOSTANDARD = LVCMOS15;
8 NET "SW[3]"       LOC = AA12 | IOSTANDARD = LVCMOS15;
9 NET "SW[4]"       LOC = Y13  | IOSTANDARD = LVCMOS15;
10 NET "SW[5]"      LOC = Y12  | IOSTANDARD = LVCMOS15;
11 NET "SW[6]"      LOC = AD11 | IOSTANDARD = LVCMOS15;
12 NET "SW[7]"      LOC = AD10 | IOSTANDARD = LVCMOS15;
13
14 # 七段数码管输出
15 NET "SEGMENT[0]" LOC = AB22 | IOSTANDARD = LVCMOS33;#a
16 NET "SEGMENT[1]" LOC = AD24 | IOSTANDARD = LVCMOS33;#b
17 NET "SEGMENT[2]" LOC = AD23 | IOSTANDARD = LVCMOS33;
18 NET "SEGMENT[3]" LOC = Y21  | IOSTANDARD = LVCMOS33;
19 NET "SEGMENT[4]" LOC = W20  | IOSTANDARD = LVCMOS33;
```

```

```

20 NET "SEGMENT[5]" LOC = AC24 | IOSTANDARD = LVCMOS33;
21 NET "SEGMENT[6]" LOC = AC23 | IOSTANDARD = LVCMOS33;#g
22 NET "SEGMENT[7]" LOC = AA22 | IOSTANDARD = LVCMOS33;#point
23 # 七段数码管使能
24 NET "AN[0]" LOC = AD21 | IOSTANDARD = LVCMOS33;
25 NET "AN[1]" LOC = AC21 | IOSTANDARD = LVCMOS33;
26 NET "AN[2]" LOC = AB21 | IOSTANDARD = LVCMOS33;
27 NET "AN[3]" LOC = AC22 | IOSTANDARD = LVCMOS33;
28
29 # Arduino小板的LED灯光，从最左边开始
30 NET "LED[0]" LOC = W23 | IOSTANDARD = LVCMOS33 ;
31 NET "LED[1]" LOC = AB26 | IOSTANDARD = LVCMOS33 ;
32 NET "LED[2]" LOC = Y25 | IOSTANDARD = LVCMOS33 ;
33 NET "LED[3]" LOC = AA23 | IOSTANDARD = LVCMOS33 ;
34 NET "LED[4]" LOC = Y23 | IOSTANDARD = LVCMOS33 ;
35
36 # 阵列键盘的列，按下按键后信号从1变为0，为低触发
37 NET "K_COL[0]" LOC = V18 | IOSTANDARD = LVCMOS18;
38 NET "K_COL[1]" LOC = V19 | IOSTANDARD = LVCMOS18;
39
40 # 阵列键盘的行，在TOP模块中被全部赋值为零
41 NET "K_ROW[0]" LOC = V17 | IOSTANDARD = LVCMOS18;
42 NET "K_ROW[1]" LOC = W18 | IOSTANDARD = LVCMOS18;
43 NET "K_ROW[2]" LOC = W19 | IOSTANDARD = LVCMOS18;
44 NET "K_ROW[3]" LOC = W15 | IOSTANDARD = LVCMOS18;
45 NET "K_ROW[4]" LOC = W16 | IOSTANDARD = LVCMOS18;
```

```

Part 4 实验验证

实验验证使用的代码

实验验证使用的数据存储器的初始值：

物理测试结果

注1：本物理测试中，我们关注每条指令**执行前**的PC值与LED值以及**执行后**造成的寄存器改变

注2：实验截图中从左至右依次编号为图1，图2，...，并附有相应解说文字

注3：小标题的“Run X”代表MIPS汇编源码中PC为X的源码执行情况

注4：Run 0 ~ Run 5为程序初始化后执行到第一次循环结束前的情况

Run 0

- MIPS 指令：`lw $at, 0($zero)`
- 机器码：100011 00000 00001 0000000000000000
- 指令类型：lw
- 预期现象：执行前：LED第3灯亮，PC = 0；执行后：`$at = 0x00000001`；
- 实验截图：

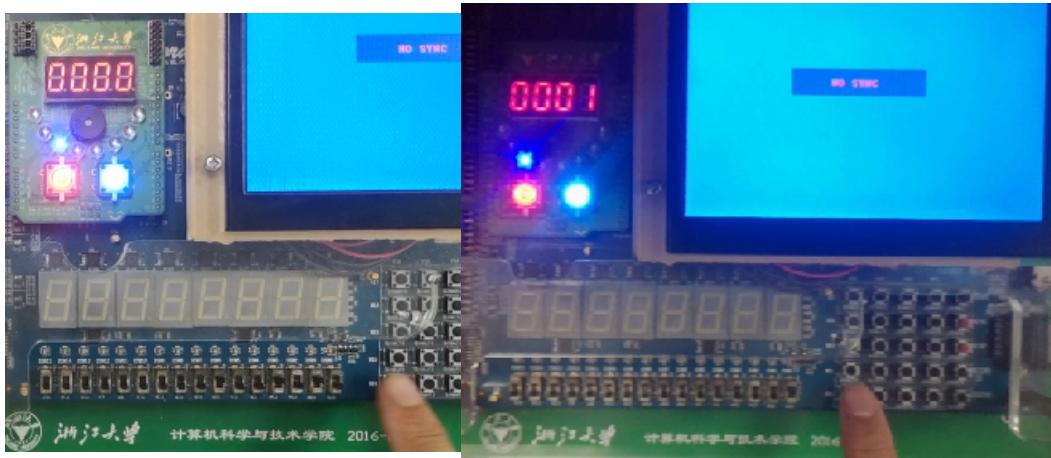


图1：执行前，PC = 1，LED第3灯亮

图2：执行后，寄存器\$at低 = 0x0001

Run 1

- MIPS指令：`lw $v0, 12($zero)`
- 机器码：100011 00000 00010 0000000000001100
- 指令类型：lw
- 预期结果：执行前：LED第3灯亮，PC = 1；执行后：`$v0 = 0x00000000`；
- 实验截图：



图1：PC = 1，LED第3灯亮

图2：寄存器\$V0低 = 0x0000

Run 2

- MIPS指令：`lw $a0, 4($zero)`
- 机器码：100011 00000 00100 000000000000000100
- 指令类型：`lw`
- 预期结果：执行前：LED第3灯亮，PC = 2；执行后：`$a0 = 0x00000065`；
- 实验截图：



图1：PC = 2，LED第3灯亮

图2：寄存器\$a0低 = 0x0065

Run 3

- MIPS指令：`add $v1, $at, $v0`
- 机器码：000000 00001 00010 00011 00000 100000
- 指令类型：`R`
- 预期结果：执行前：LED第1灯亮，PC = 3；执行后：`$v1 = 0x00000001`；

- 实验截图：



图1：PC = 3，LED第1灯亮

图2：寄存器\$v0低 = 0x0001

Run 4

- MIPS指令：`add $v0, $v1, $v0`
- 机器码：000000 00011 00010 00010 00000 100000
- 指令类型：R
- 预期结果：执行前：LED第1灯亮，PC = 4；执行后： $$v0 = 0x00000001$ ；
- 实验截图：

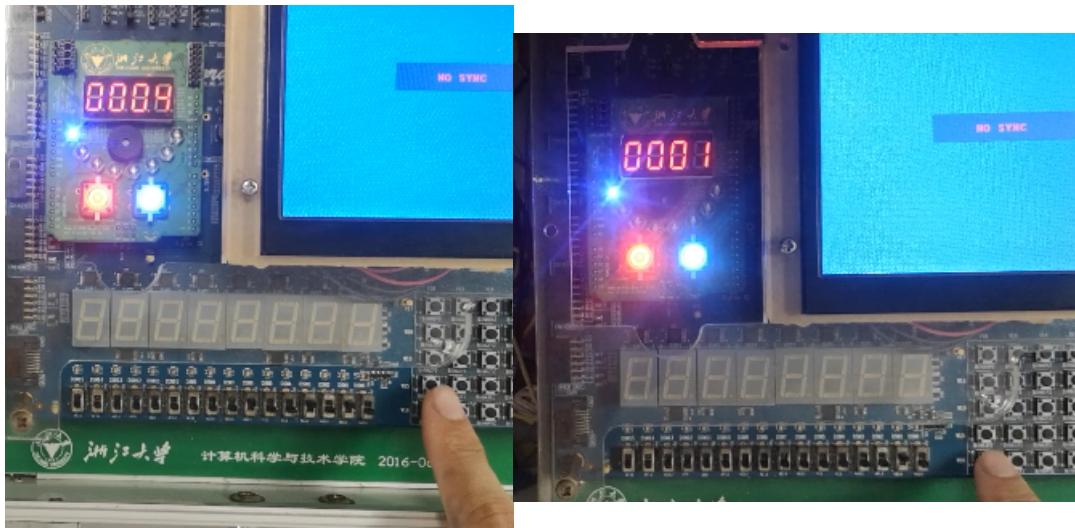


图1：PC = 4，LED第1灯亮

图2：寄存器\$v0低 = 0x0001

Run 5

- MIPS指令：`add $v1, $v1, $at`
- 机器码：000000 00011 00001 00011 00000 100000

- 指令类型 : R
- 预期结果 : 执行前 : LED第1灯亮 , PC = 5 ; 执行后 : \$v1 = 0x00000002 ;
- 实验截图 :

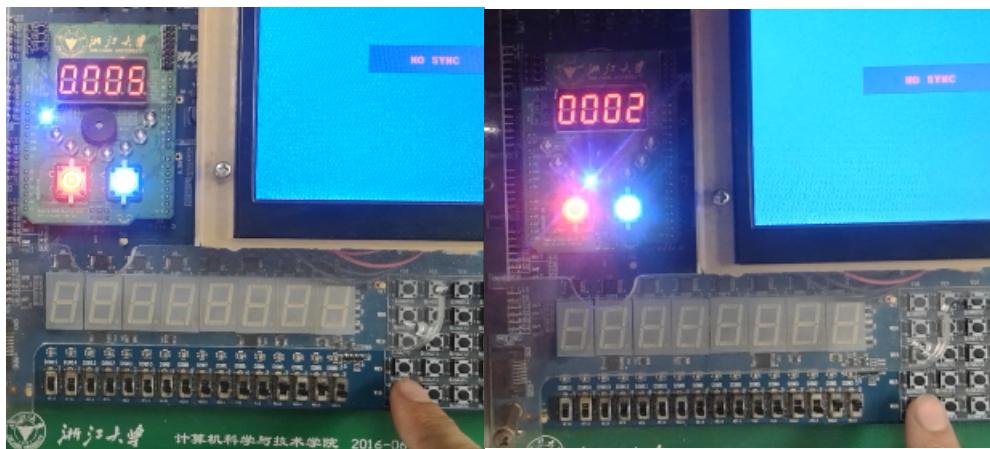


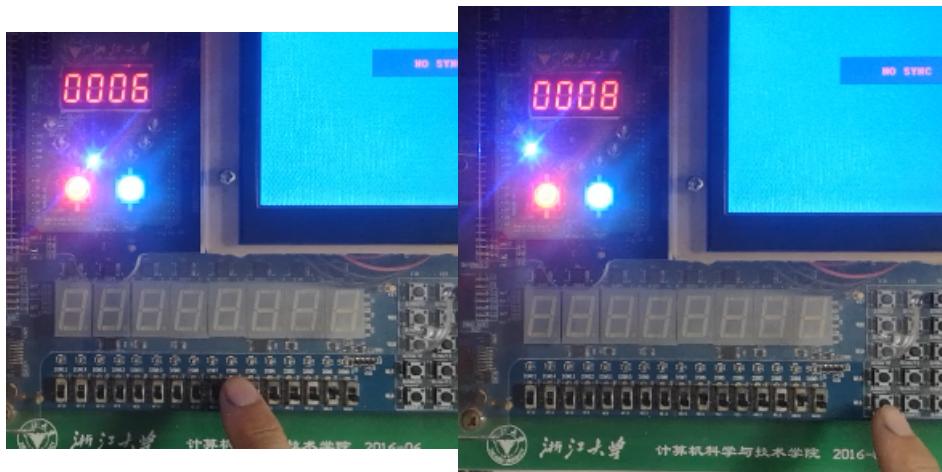
图1 : PC = 5 , LED第1灯亮

图2 : 寄存器\$V1低 = 0x0002

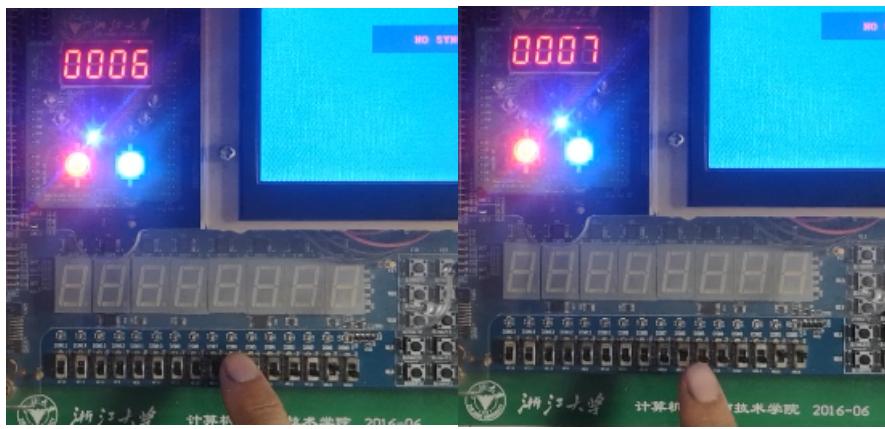
Run 6

- MIPS指令 : `beq $v1, $a0, 4`
- 机器码 : 000100 00100 00011 0000000000000000
- 指令类型 : L
- 预期现象 : 没跳出循环 : PC = 6->7 ; 跳出循环 : PC = 6->8
- 实验截图 :

跳出循环:



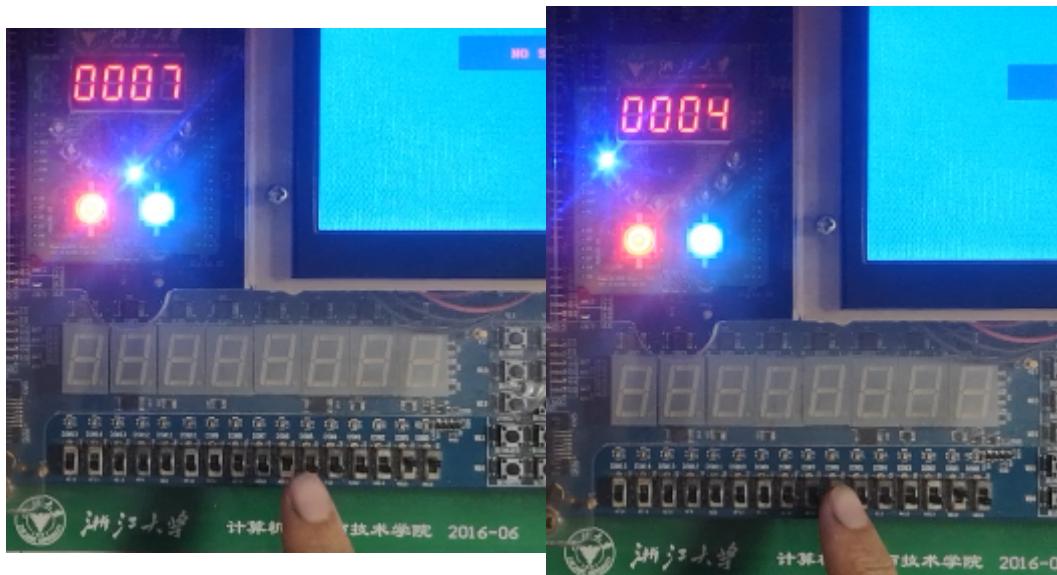
没跳出循环 :



Run 7

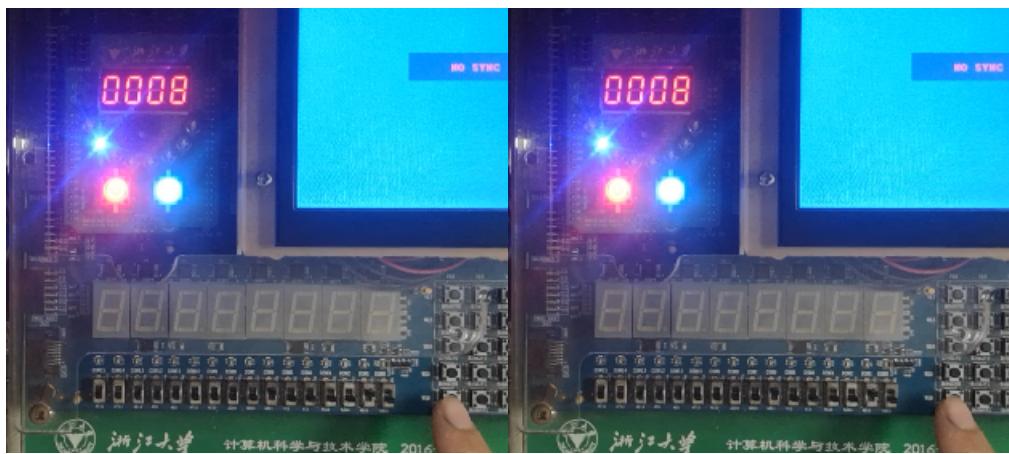
- MIPS指令：`j 16`
- 机器码：000010 0000000000000000000000000100
- 指令类型：J
- 预期现象：PC = 7->4；

PC：



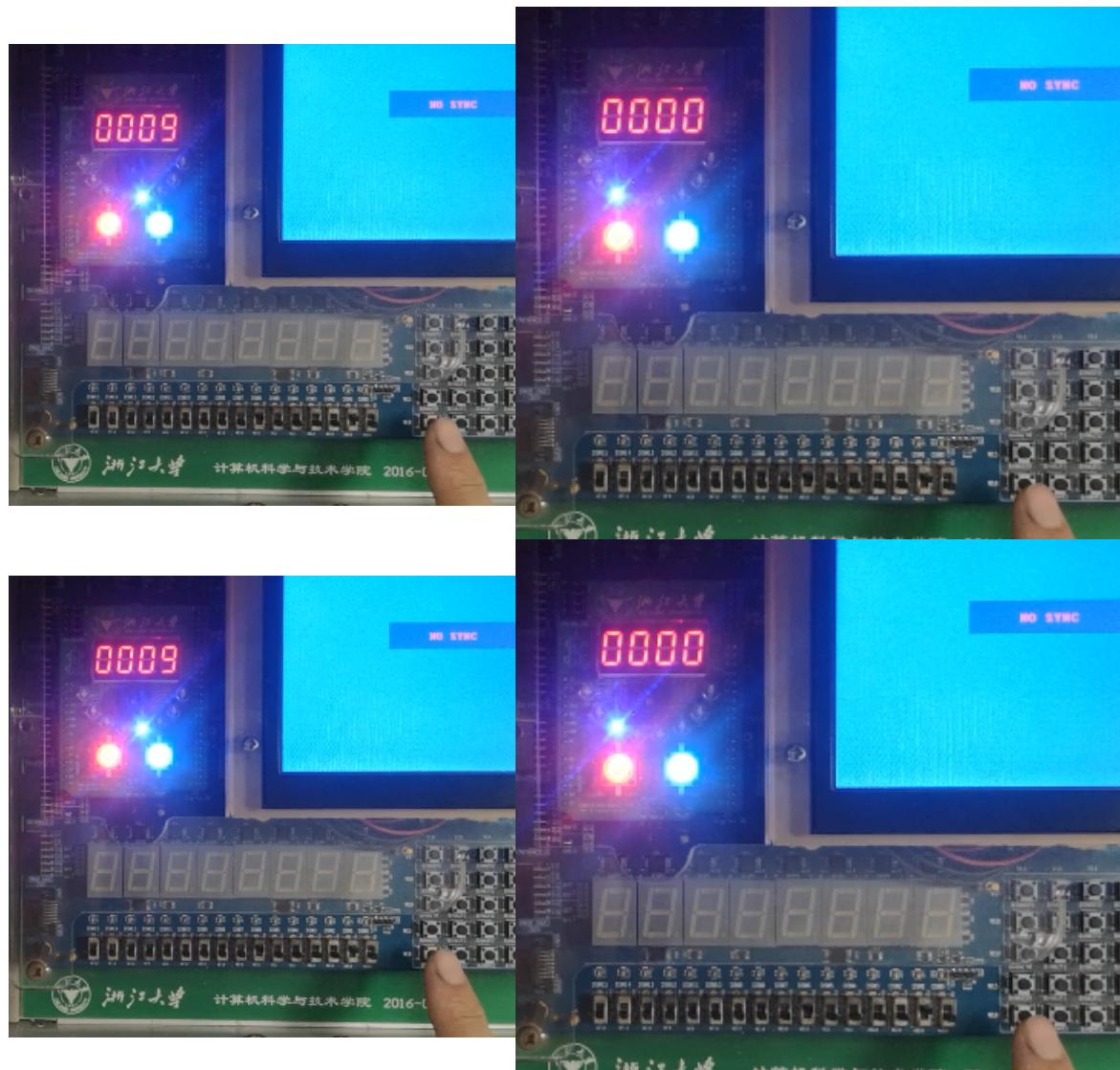
Run 8

- MIPS指令：`sw $v0, 8($zero)`
- 机器码：101011 00000 00010 0000000000001000
- 指令类型：L
- 预期现象：PC = 8->9；
- 实验截图：



Run 9

- MIPS指令：`j 0`
- 机器码：000010 00000000000000000000000000000000
- 指令类型：J
- 预期现象：PC = 9->0；
- 实验截图：



运算结果

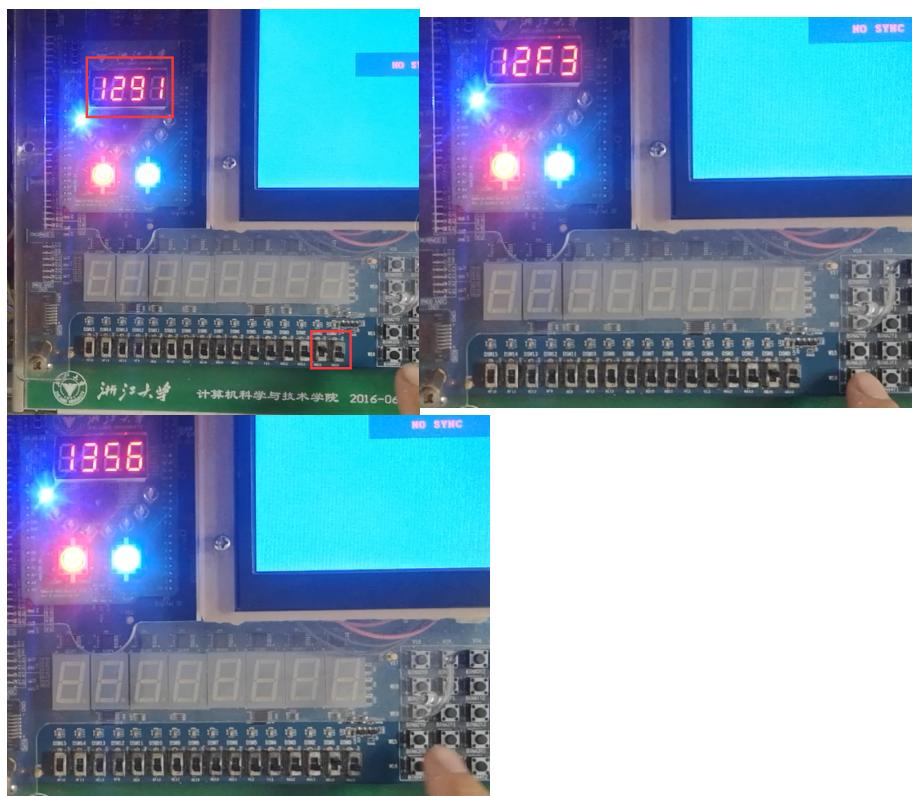
我们等待程序运行到100次循环完全结束的阶段，观察程序是否对下列算式进行了正确求和：

$$\sum_{i=1}^{100} i = 5050$$

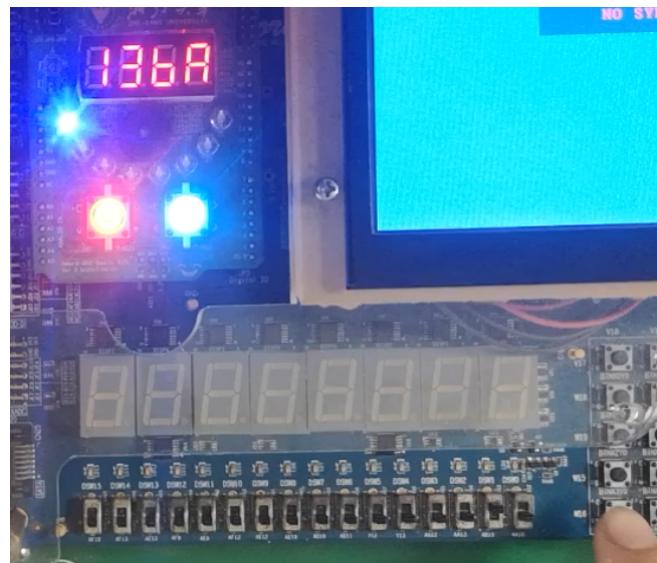
(1)

容易注意到最终结果 $5050_{(Dec)}$ = $13BA_{(Hex)}$

我们显示的是\$2寄存器的值：

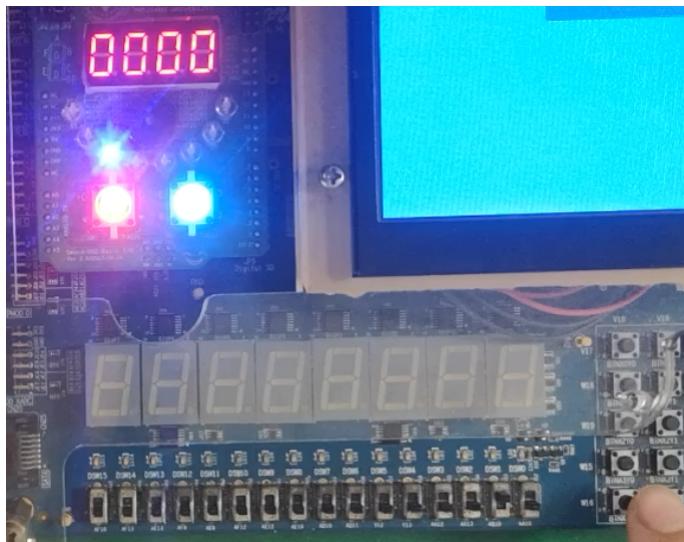


在循环到100次后程序算出了求和的总结果：



并成功执行了在 `beq` 语句处跳出到最后一条语句：

j 0 : 00001000000000000000000000000000



\$t* 寄存器的使用

我们增加了一份汇编源码：

```
```asm
1 lw $t1, 0($t0)
2 lw $t2, 12($t0)
3 lw $t4, 4($t0)
4 add $t3, $t1, $t2
5 add $t2, $t3, $t2
6 add $t3, $t3, $t1
7 beq $t3, $t4, 4
8 j 16
9 sw $t2, 8($t0)
10 j 0
```
```

其中使用的寄存器为 \$t0, \$t1, \$t2, \$t3, \$t4

分别对应于实验PPT中的\$0, \$1, \$2, \$3, \$4

其汇编后的机器码（.coe文件）为：

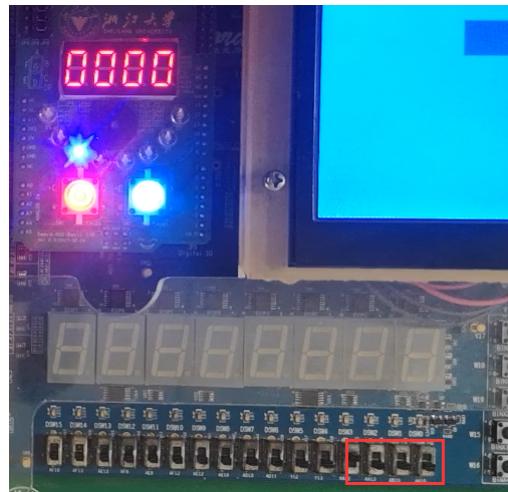
我们进行t寄存器的测试有以下几点考虑：

1. 在PPT中使用的\$at等寄存器并不符合MIPS标准，为系统独用寄存器
 2. 而t类型寄存器是客户程序可随意使用的
 3. 使用t类型可以验证我们的寄存器值是否得到了正确的初始化

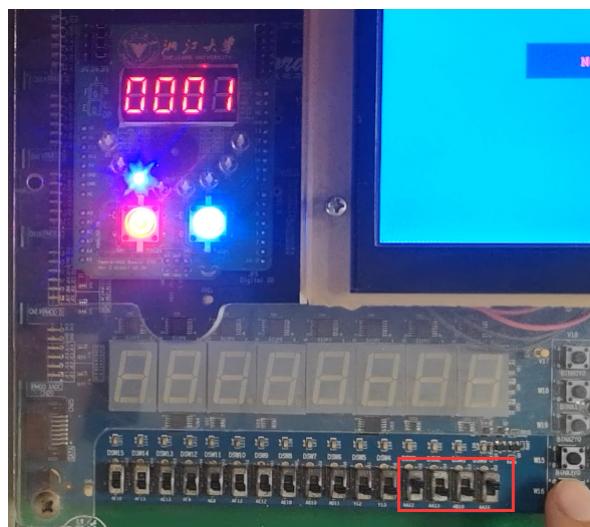
4. 并在一定程度上说明我们的CPU对一般性的寄存器也有着正常的调用能力

可见，我们的t寄存器正确储存了相关的值：

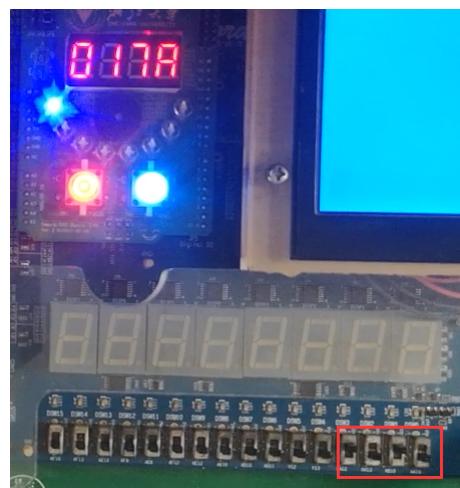
t₀：常数0



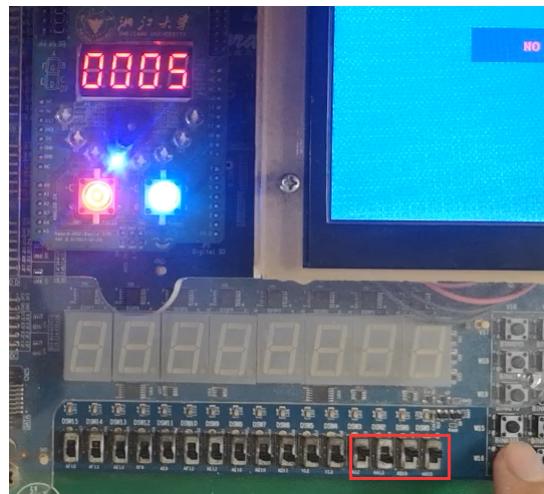
t₁：常数1



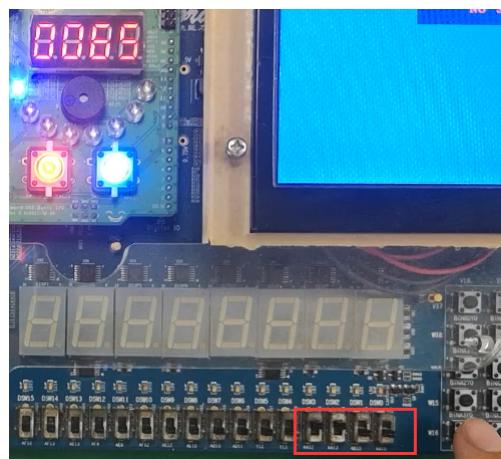
t₂：求和结果寄存器



t₃：求和循环变量



t4 : 常数65(Hex)=101(Dex)



表格A

| 指令 | 机器码 | 分割后机器码 |
|----------------------|--|---|
| lw \$at, 0(\$zero) | 100011000000000010000000000000000000 | 100011-00000-00001-000000000000000000 |
| lw \$v0, 12(\$zero) | 100011000000001000000000000000001100 | 100011-00000-00010-00000000000000001100 |
| lw \$a0, 4(\$zero) | 10001100000001000000000000000000100 | 100011-00000-00100-0000000000000000100 |
| add \$v1, \$at, \$v0 | 00000000001000100001100000100000 | 000000-00001-00010-00011-00000-100000 |
| add \$v0, \$v1, \$v0 | 000000000011000100001000000100000 | 000000-00011-00010-00010-00000-100000 |
| add \$v1, \$v1, \$at | 000000000011000010001100000100000 | 000000-00011-00001-00011-00000-100000 |
| beq \$v1, \$a0, 4 | 000100001000001100000000000000000001 | 000100-00100-00011-00000000000000000001 |
| j 16 | 000010000000000000000000000000000000100 | 000010-00000000000000000000000000000000100 |
| sw \$v0, 8(\$zero) | 101011000000000100000000000000001000 | 101011-00000-00010-00000000000000001000 |
| j 0 | 00001000 | 000010-00 |

表格B

初始状态：所有寄存器均为0
(字节取址)

第一次循环：

| 指令 | 相关寄存器 | PC | PC+4(按word计数为PC+1) | clockcnt |
|----------------------|----------------------|----|--------------------|----------|
| lw \$at, 0(\$zero) | \$at=1; | 0 | 1 | 1 |
| lw \$v0, 12(\$zero) | \$v0=0 | 1 | 2 | 2 |
| lw \$a0, 4(\$zero) | \$a0=0x65 | 2 | 3 | 3 |
| add \$v1, \$at, \$v0 | \$v1=1;\$at=1;\$v0=0 | 3 | 4 | 4 |
| add \$v0, \$v1, \$v0 | \$v0=1;\$v1=1; | 4 | 5 | 5 |
| add \$v1, \$v1, \$at | \$v1=2;\$at=1 | 5 | 6 | 6 |
| beq \$v1, \$a0, 4 | \$v1=2;\$a0=0x65 | 6 | 7 | 7 |

循环结束时：

| 指令 | 相关寄存器 | PC | PC+4(按word计数为PC+1) | clockcnt |
|-------------------------|------------------------|----|--------------------|----------|
| add \$v0, \$v1,
\$v0 | \$v0=0x13BA;\$v1=0x64; | 4 | 5 | 0x18F |
| add \$v1, \$v1,
\$at | \$v1=65;\$at=1 | 5 | 6 | 0x190 |
| beq \$v1, \$a0, 4 | \$v1=0x65;\$a0=0x65 | 6 | 7 | 0x191 |
| j 16 | | 7 | 8 | 0x192 |
| sw \$v0, 8(\$zero) | \$v0=0x13BA | 8 | 9 | 0x193 |
| j 0 | | 9 | 0xA | 0x194 |

COE文件路径

`./ipcore_dir/ipcore_dir/machine_code.coe` : 指令机器码coe文件

`./ipcore_dir/ipcore_dir/memory_store.coe`：内存初始化coe文件

注：我们同时提供了使用t类型寄存器相关的指令机器码文件，可在同一目录下查询得到。