计算机组成原理 实验报告

姓名：廖洲洲　学号：PB17081504　实验日期：2019-5-17

一、实验题目：

Lab5 多周期 MIPS-CPU

二、实验目的：

设计实现多周期 MIPS-CPU，可执行如下指令：

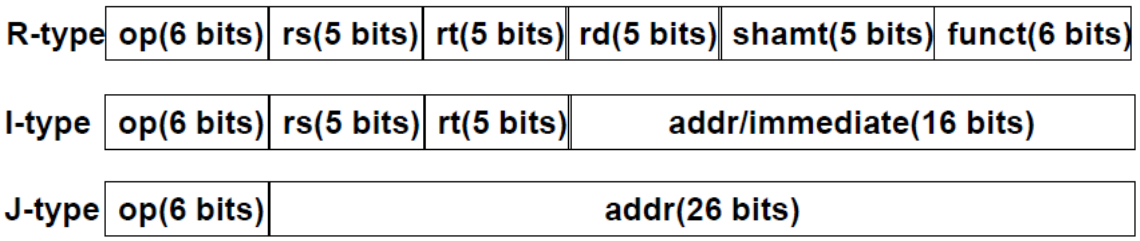
- add, sub, and, or, xor, nor, slt
- addi, andi, ori, xori, slti
- lw, sw
- beq, bne, j

三、实验平台：

Vivado

四、实验过程：

1) MIPS 机器语言格式（R 型，I 型，J 型）



MIPS 指令解码（操作码 op 对应的功能）

opcode		bits 28..26							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	*SPECIAL* δ	*REGIMM* δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0* δ	*COP1* δ	*COP2* θδ	*COP1X*[1] δ	BEQL ϕ	BNEL ϕ	BLEZL ϕ	BGTZL ϕ
3	011	β	β	β	β	*SPECIAL2* δ	JALX ε	ε	*SPECIAL3*[2] δΘ
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	β
5	101	SB	SH	SWL	SW	β	β	SWR	CACHE
6	110	LL	LWC1	LWC2 θ	PREF	β	LDC1	LDC2 θ	β
7	111	SC	SWC1	SWC2 θ	*	β	SDC1	SDC2 θ	β

R 型指令中 funct 字段对应的功能

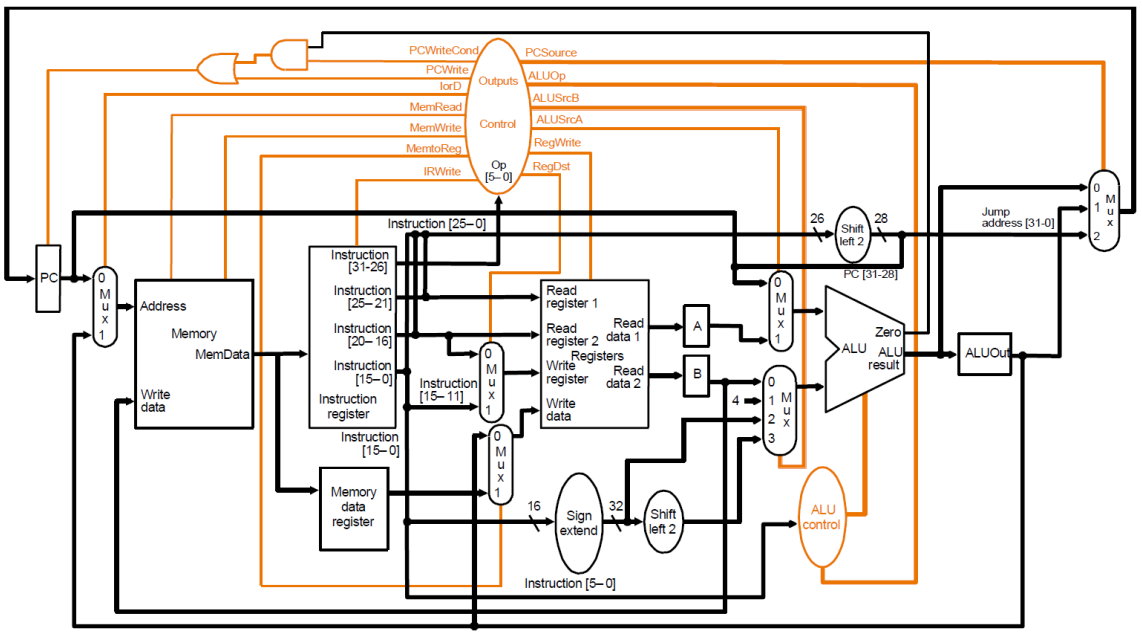
function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	SLL[1]	*MOVCI* δ	*SRL* δ	SRA	SLLV	*	*SRLV* δ	SRAV
1	001	JR[2]	JALR[2]	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	β	*	β	β
3	011	MULT	MULTU	DIV	DIVU	β	β	β	β
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	β	β	β	β
6	110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	111	β	*	β	β	β	*	β	β

2) 寄存器使用约定

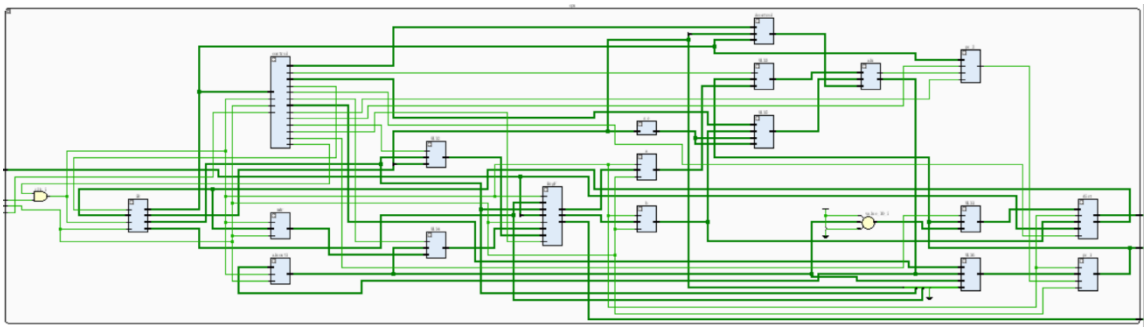
REGISTER	NAME	USAGE
$0	$zero	常量0(constant value 0)
$1	$at	保留给汇编器(Reserved for assembler)
$2-$3	$v0-$v1	函数调用返回值(values for results and expression evaluation)
$4-$7	$a0-$a3	函数调用参数(arguments)
$8-$15	$t0-$t7	暂时的(或随便用的)
$16-$23	$s0-$s7	保存的(或如果用，需要SAVE/RESTORE的)(saved)
$24-$25	$t8-$t9	暂时的(或随便用的)
$28	$gp	全局指针(Global Pointer)
$29	$sp	堆栈指针(Stack Pointer)
$30	$fp	帧指针(Frame Pointer)
$31	$ra	返回地址(return address)

3) 数据通路的建立

数据通路基本与 ppt 上的数据通路相一致，与该图不一致的时在取完一条指令后，我们对 pc 不是加 4，而是加 1.



设计图：



核心代码：（在顶层将各个模块相连接）

```
assign clk = run_clk & clk_input;

assign  pc = pc_out;

PC_Module pc_1(pc_src,pc_we,rst,clk,pc_out);

PCWE pc_2(Instruction_op,PCWrite,PCWriteCond,Zero,pc_we);

MUX_2_32 MUX1(pc_out,(ALU_out>>2),lorD,address);

dist_mem_gen_0

dist(address[7:0],B_out,addr,clk,MemWrite,Mem_data,mem_data_out);//rea

d

IR_Module

iR(Mem_data,IRWrite,clk,rst,Instruction_op,Instruction_25_21,Instructi

on_20_16,Instruction_15_0);

MUX_2_5

MUX2(Instruction_20_16,Instruction_15_0[15:11],RegDst,Write_register);

RegFile

RegF(Instruction_25_21,Instruction_20_16,addr[4:0],Write_data_regf,Wri

te_register,RegWrite,rst,clk,RD0,RD1,reg_data);

ALU_A a(RD0,clk,rst,A_out);

ALU_B b(RD1,clk,rst,B_out);

MUX_2_32 MUX3(pc_out,A_out,ALUSrcA,ALU_1);

MemoryDataR mdr(Mem_data,rst,clk,memdataout);

MUX_2_32 MUX4(ALU_out,memdataout,MemtoReg,Write_data_regf);

Sign_extend s_e(Instruction_15_0,I);

assign I_shift = I;

MUX_4_32 MUX5(B_out,I,I_shift,ALUSrcB,ALU_2);

ALU alu(ALU_1,ALU_2,alu_c,alu_result,Zero);

ALU_OUT aluout1(alu_result,clk,rst,ALU_out);
```

```
ALU_control
Acontrol(Instruction_op,Instruction_15_0[5:0],ALUOp,alu_c);
s_l_2({6'b000000,Instruction_25_21,Instruction_20_16,Instruction_15_0}
,Jaddress);
assign Jaddress =
{6'b000000,Instruction_25_21,Instruction_20_16,Instruction_15_0};
MUX_3_32 MUX6(alu_result,ALU_out,Jaddress,PCSource,pc_src);
ALL_Control
control(Instruction_op,clk,rst,PCWriteCond,PCWrite,lorD,MemRead,MemWri
te,MemtoReg,IRWrite,PCSource,ALUOp,ALUSrcB,ALUSrcA,RegWrite,RegDst,cur
rent_state,run,run_clk);
```

4) RegFile 的读写

RegFile 可以进行异步读和同步写操作，即读操作不依赖于时钟，只要输入个要读数据的地址，即可通过输出数据端口得到对应的数据，这里设置了三个读数据端口。数据的写依赖于时钟，且只有当写使能有效时才能写成功。

核心代码：

```
always @(posedge clk or posedge rst) begin
    if (rst) begin
        // reset
        for(i = 0;i<NUM;i=i+1)
            register[i] <= 0;     //初始化操作
    end
    else if (!rst) begin
        if(we)begin
            if(wd!=0)
                register[wd] <= wa;//写操作
        end
    end
end

assign rd0 = register[ra0];   //读操作
```

```
assign rd1 = register[ra1];
assign rd2 = register[ra2];
```

5) ALU 控制

输入两个数据，输出经过计算后的结果和判 0 标志。ALU 功能由 4 位控制信号控制。（异步）

0000：&　0001：|　0010：xor　0011：nor　0100：add

0101：sub　0110：slt

核心代码：

```
 assign Zero = (y==0)?1:0;  //判 0
reg [32:0] temp;
always @(a,b,s)
begin
case(s)
    4'b0000:   //&
    begin
        y<=a&b;
    end
    4'b0001:    //|
    begin
        y<=a|b;
    end
    4'b0010:  //xor
    begin
        y<=a^b;
    end
    4'b0011:    //nor
    begin
            y<=~(a|b);
    end
    4'b0100:    //add
    begin
        /*temp={1'b0,a}+{1'b0,b};
```

```
    y=temp[31:0];*/
    y <= a+b;
end
4'b0101:    //sub
 begin
    /*temp={1'b0,a}-{1'b0,b};
    y=temp[31:0];*/
    y <= a-b;
end
4'b0110:  //slt
begin
    if(a[31]==1&&b[31]==1)
            if(a[30:0]>b[30:0])
                    y <= 1;
            else begin
                    y <= 0;
            end
    else if(a[31]==1&&b[31]==0)
    begin
            y <= 1;
    end
    else if(a[31]==0&&b[31]==1)
    begin
            y <= 0;
    end
    else begin
            if(a[30:0]<b[30:0])
                    y <= 1;
            else begin
                    y <= 0;
            end
```

```
            end
        end
        default:begin
            y <= 0;
        end
    endcase
    end
```

6) 主控制单元的设计

总共设计了 13 个控制信号，对应的功能即在何阶段有效如下：

1. PcWriteCond:控制 Pc 的写回，其判定当前指令是否是 beq 或 bne 指令，故只有在执行 beq 或 bne 指令时其有效。
2. PcWrite：也是控制 Pc 写回的信号，但是其只有在取值或跳转阶段有效，也就是说一旦取值完成，我们就可以对 Pc 值进行更新。
3. lord：存储器前面的多选器的控制信号，当其为 0 时，输入地址为 Pc 值，当其为 1 时，输入地址为计算后的地址。因此只有在 load 和 send 阶段其值为 1，其他时间均为 0.
4. MemWrite：控制存储器的写，当其为 1 时，存储器可进行写操作。故其在存储器的写回阶段有效。
5. MemtoReg：寄存器堆写数据口前的多选器控制信号，当其为 1 时，写入的数据为从存储器中加载的数据，当其为 0 时，输入的数据来自 alu 计算后的数据。故其在 load 指令的写回阶段有效。
6. RegDst：寄存器堆前的写寄存器号，当其是 1 时，写寄存器号为 I[15:11],当其为 0 时，写寄存器号为 I[20:16],故若是 R 型指令，则其值为 0，在其他情况，其值为 1 。
7. RegWrite：寄存器堆写控制信号，当其值为 1 时，可以对寄存器进行写操作，当其值为 0 时，不能堆寄存器值进行写操作。因此其在 load 指令，R 型指令，I 型指令的 WB 写回阶段有效。
8. IRWrite：指令寄存器更新信号，当其值为 1 时可以对指令进行更新，故其只在 IF 取值阶段有效。
9. PCSource：PC 多选器的控制信号，当其值为 00 时，PC 来自前一条指令加 1，当其值为 01 时，PC 来自 beq 或 bne 指令中的跳转地址，当其值为 10 时，PC 来自 JMP 指令的跳转地址。

10. ALUOp：给 ALUctr 的控制信号，用于协助 ALUctr 产生 alu 控制信号，在取值阶段和 lw、sw 指令的地址计算阶段其值均为 00，当其值为 10 时，说明其要根据 R 型的功能判断 alu 控制信号。当其值为 01 时，单纯令 alu 执行减操作，故其在 beq 和 bne 为 01.当其值为 11 时，说明要根据 I 型指令来判断 alu 功能。
11. ALUSrcB：ALUB 数据端口前的多选器的控制信号。
12. ALUSrcA：ALUA 数据端口前的多选器的控制信号。
13. MemRead：控制存储器的读信号，这里设置其恒为 1.

核心代码：

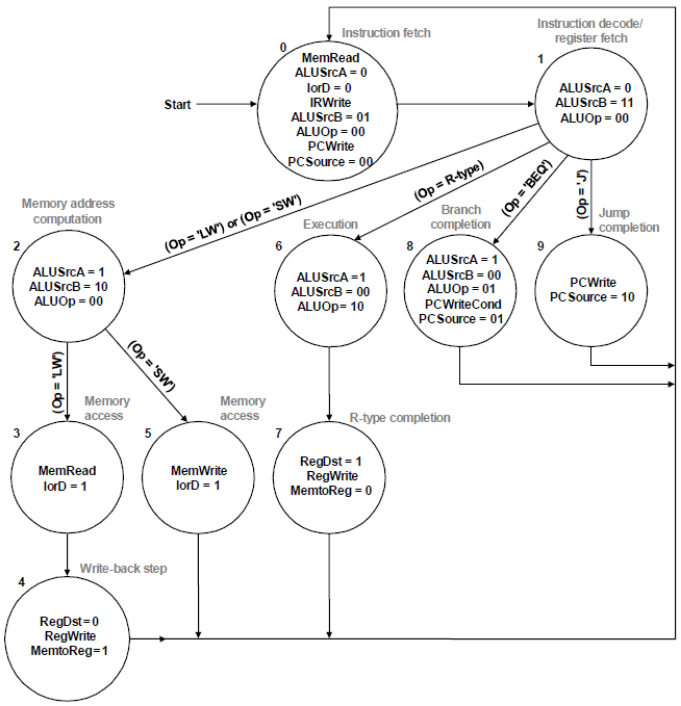
```
always @(posedge clk) begin  //下一状态的控制信号使用的是上一个时钟中修改的
  //PCWriteCond
if(next_state == BEQC)
  PCWriteCond = 1;  //PCwrite 只有在 beq bne 中有效
else begin
  PCWriteCond = 0;
end

//PCWRITE  只在取值和跳转时有效
if(next_state == IF || next_state == JMC)
  PCWrite = 1;
else begin
  PCWrite = 0;
end

//lord 只有在 load 和 send 阶段有效，lord 为 1 说明是对数据存储器读写
if(next_state == LOAD ||next_state == SEND)
  lorD = 1;
else begin
  lorD = 0;
end
```

```
//memwrite 在 sw 的写回有效
if(next_state == SEND)
  MemWrite = 1;
else begin
  MemWrite = 0;
end

//memtoreg lw_wb 为 1，即内存中数据写入，0 则 alu 结果写入
if(next_state == LW_WB)
  MemtoReg = 1;
else begin
  MemtoReg = 0;
end
//regdst
if(next_state == R_WB)//IWB is zero
  RegDst = 1;
else begin
  RegDst = 0;
end

//RegWrite
if(next_state == LW_WB|| next_state == R_WB||next_state==I_WB)
  RegWrite = 1;
else begin
  RegWrite = 0;
end

//IRWrite
if(next_state == IF)
  IRWrite = 1;
else begin
```

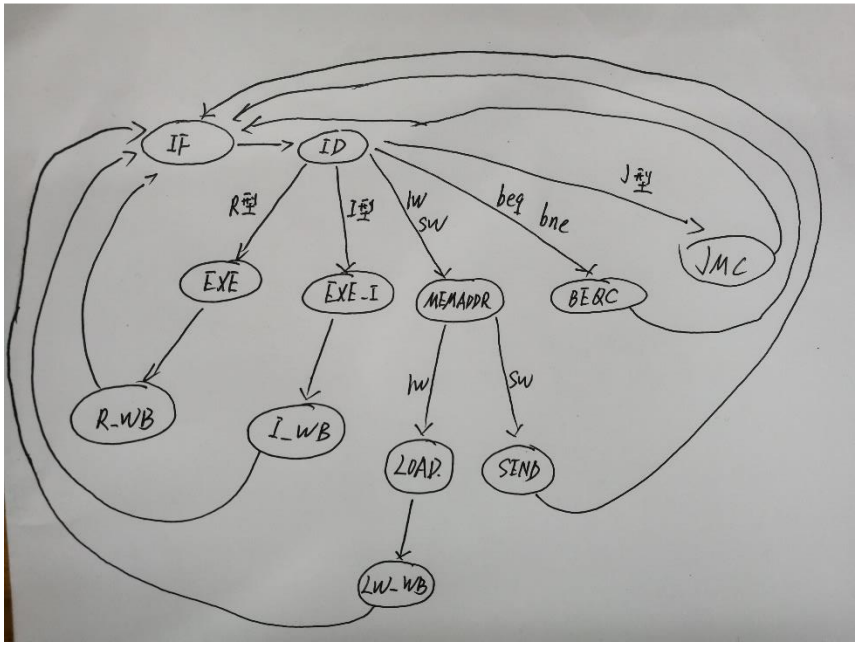
```
  IRWrite = 0;
end

//pcsource
if(next_state == IF)
  PCSource = 2'b00;
else if(next_state == BEQC)
  PCSource = 2'b01;
else if(next_state == JMC)
  PCSource = 2'b10;
else begin
  PCSource = 2'b11;
end

//ALUOP
if(next_state == IF || next_state == MEMADRESS || next_state ==
ID)
  ALUOp = 2'b00;
else if(next_state == EXE)begin
  ALUOp = 2'b10;
end
else if(next_state == BEQC)
  ALUOp = 2'b01;  //减
else if(next_state == EXE_I)begin
  ALUOp = 2'b11;
end
else begin
  ALUOp = 2'b00;
end

//ALUSrcB
```

```
if(next_state == MEMADRESS || next_state == EXE_I)
  ALUSrcB = 2'b10;
else if(next_state == IF)
  ALUSrcB = 2'b01;
else if(next_state == ID)
  ALUSrcB = 2'b11;
else begin
  ALUSrcB = 2'b00;
end

//ALUSRCA
if(next_state == ID||next_state == IF)
  ALUSrcA = 0;
else begin
  ALUSrcA = 1;
end
//MemRead==1
MemRead = 1;
end
```

7） 主控制中状态的转换



这是 ppt 上的状态转换，其中也包含了信号的输出，但是本次实验我们是通过对单个信号单独分析实现的，因此在状态转换中不需要考虑控制信号的输出。下面是本次实验使用的状态转换图。



核心代码：

```
always@(current_state or Op) begin
      next_state = 4'b0000;
      case(current_state)
      INVALID:begin
            next_state = IF;
      end
      IF:begin
            next_state = ID;
      end
      ID:begin
            if(Op == 6'b100011||Op == 6'b101011)//lw or sw
            begin
                  next_state = MEMADRESS;  //计算内存地址
            end
            else if(Op==6'b000000) begin  //R 型
                  next_state = EXE;
            end
```

```
            else if(Op == 6'b000100||Op == 6'b000101) begin //beq and
bne
                  next_state = BEQC;
            end
            else if(Op == 6'b000010) begin  //j型 跳转
                  next_state = JMC;
            end
            else if(Op[5:3] == 3'b001) begin  //I型
                  next_state = EXE_I;
            end
      end
      MEMADRESS:begin
            if(Op == 6'b100011)// lw3
            begin
                  next_state = LOAD;  //lw 下次进入加载阶段
            end
            else begin
                  next_state = SEND;  //sw3 进入写阶段
            end
      end
      LOAD:begin
            next_state = LW_WB;  //lw4 进入写寄存器阶段
      end
      LW_WB:begin
            next_state = IF;   //lw5 结束,再次取指
      end
      SEND:begin     //sw4 写回，结束了，下一阶段取指
            next_state = IF;
      end
      EXE:begin //R3 执行，下一阶段写回
            next_state = R_WB;
```

```
        end
        R_WB:begin //R4 R 结束，下一阶段取指
              next_state = IF;
        end
        BEQC:begin //beq3 bne3 结束，下一阶段取值
              next_state = IF;
        end
        JMC:begin   //JMC3 结束
              next_state = IF;
        end
        EXE_I:begin  //I3 i 型执行，下一阶段写回
              next_state = I_WB;
        end
        I_WB:begin  //I4 写回，结束
              next_state = IF;
        end
        default:begin
              next_state = INVALID;
        end
        endcase
    end
```

8) 指令的执行过程

a) R 型指令的执行过程

1. 取值和 PC+1
2. 取两个源操作数寄存器
3. ALU 操作
4. 结果写回目的寄存器

b) I 型指令的执行过程

1. 取值和 PC+1
2. 取一个操作数和立即数扩展
3. ALU 操作
4. 结果写回目的寄存器

c) lw 指令的执行过程

1. 取值和 PC+1
2. 读寄存器$t2
3. ALU 完成$t2 与扩展后的 16 位立即数相加，得到要取数据的地址
4. 将地址送往 MEM，读出数据
5. 内存中的数据送往目的寄存器

d) sw 指令的执行过程

1. 取值和 PC+1
2. 读寄存器，得到要存入内存的数据
3. 将另一寄存器的数和扩展后的 16 位立即数相加，得到内存地址
4. 将数据存入内存

e) beq 指令的执行过程

1. 取值和 PC+1
2. 读寄存器$t1 和$t2
3. ALU 将$t1 和$t2 相减，PC+1 与进行符号扩展后的 16 位 offset 相加，作为分支目标地址
4. ALU 的 Zero 确定应该送往 PC 的值

f) jump 指令的实现

1. 取值和 PC+1
2. 译码
3. 无条件跳转到目标地址

9) DDU 模块的实现

将 pc 的后八位从 cpu 连接出来在 led 的低八位显示，在 DDU 中根据按键产生一个地址，将这个地址送入到 cpu 中，然后从中读出对应这个地址的内存中和寄存器堆中的值，再根据控制按钮选择不同的数以 16 进制输出在显示数码管上。当单步执行的开关为 1 时，若按下一次执行的开关，会产生一个 run 信号，这个信号只在一个周期内有效，从而控制 cpu 单步执行。

核心代码：

```
module RUN(
input cont,
input step,
```

```
input clk,
input rst,
output run
    );
wire clk_out;
//run = 1/run = 0;
cout_1ms c3(clk,rst,clk_out);  //1000HZ prevent the cycle too short
reg real_run;
reg flag;
always @(posedge clk_out or posedge rst) begin
    if (rst) begin
        // reset
        real_run <= 0;
        flag <= 0;
    end
    else begin
        if(step == 1 && flag == 0)
        begin
            real_run <= 1;
            flag <= 1;
        end
        else if(step == 0 && flag == 1)
        begin
            flag <= 0;
            real_run <= 0;
        end
        else begin
            real_run <= 0;
        end
    end
end
```

```
assign run = real_run | cont;

endmodule
```

10) 测试代码

```
j _start

.data
    .word 0,8,1,6,0xffffffff8,1,3,5,0

_start:
            addi $t0,$0,3       #t0=3
        addi $t1,$0,5    #t1=5
            addi $t2,$0,1       #t2=1

        add  $s0,$t1,$t0  #s0=t1+t0=8  ц
        lw   $s1,12($0)
        bne  $s1,$s0,_fail

        and  $s0,$t1,$t0  #s0=t1&t0=1
        lw   $s1,16($0)
        bne  $s1,$s0,_fail

        xor  $s0,$t1,$t0  #s0=t1^t0=6
        lw   $s1,20($0)
        bne  $s1,$s0,_fail

        nor  $s0,$t1,$t0  #s0=t1 nor t0=0xffffffff8
        lw   $s1,24($0)
        bne  $s1,$s0,_fail

        slt  $s0,$t0,$t1  #s0=1
        lw   $s1,28($0)
```

```
        bne   $s1,$s0,_fail

        andi  $s0,$t0,7   #s0=3
        lw    $s1,32($0)
        bne   $s1,$s0,_fail

        ori   $s0,$t1,4   #s0=5
        lw    $s1,36($0)
        bne   $s1,$s0,_fail

        sw    $t1,40($0)
        lw    $s1,40($0)
        beq   $t1,$s1,_sucess

_fail:
            sw    $0,8($0)
        j     _fail

_sucess:
      sw    $t2,8($0)
      j     _sucess
```

五、实验结果：

cpu 运行过程中$s0 中的数依次为：

最终内存中地址为 0X8 的数为 1：

测试成功

六、心得体会

- 通过这次实验用 verilog 实现了一个 MIPS 多周期 CPU，让我对 MIPS 指令集有了更多的了解，理解了 cpu 中数据通路的建立，同时加深了我对 CPU 各类指令执行过程的了解。

七、代码

```
module MainCPU(
input cont,step,mem,inc,dec,clk_in,rst,
output [15:0]led,
output [7:0]an,
output [6:0]seg,
output dp
    );
wire run;
wire [7:0]address;
wire locked;
wire [31:0]mem_data;
wire clk_10hz;

wire clk_50mhz;
wire [7:0]pc;

wire [31:0]reg_data;
clk_wiz_0
c(.clk_out1(clk_50mhz),.reset(rst),.locked(locked),.clk_in1(cl
k_in));
DISPLAY_UNIT
Displayunit(cont,step,mem,inc,dec,pc,mem_data,reg_data,clk_50m
hz,rst,run,addr,led,an,seg,dp);
cout_10hz c2(clk_50mhz,rst,clk_10hz);
CPU cpu(run,address,clk_10hz,rst,pc,mem_data,reg_data);

endmodule
```

```
module CPU(
input run,
input [7:0]address,
input clk_input,rst,
output [31:0]pc,
output [31:0]out_data,
output [31:0]reg_data,
output    Zero,PCWrite,lorD,
MemWrite,IRWrite,RegDst,RegWrite,ALUSrcA,MemtoReg,current_stat
e,
output [5:0]IR_op,
output [1:0]PCSource,
output [1:0]ALUSrcB*/
    );
//declaration
wire
[31:0]pc_src,pc_out,B_out,Mem_data,Write_data_regf,RD0,RD1,alu
_o,A_out,A_out_1,B_out_1,ALU_1,memdataout,I,I_shift,ALU_2,alu_
result;
wire [31:0]Jaddress;
wire [31:0]ALU_out,address;
wire [15:0]IR_15_0;
wire [3:0] alu_c;
wire [1:0] ALUOp;
wire [1:0] ALUSrcB;
wire [1:0]PCSource;
wire run_clk;
```

```
wire
PCWriteCond,Zero,PCWrite,lorD,MemWrite,IRWrite,RegDst,RegWrite
,ALUSrcA,MemtoReg;
wire clk,MemRead;
wire [3:0]current_state;
wire [5:0]IR_op;
wire [4:0]IR_25_21,IR_20_16,Write_register;
//
wire pc_we;
assign clk = run_clk & clk_input;
assign  pc = pc_out;
PC_Module pc_1(pc_src,pc_we,rst,clk,pc_out);
PCWE pc_2(IR_op,PCWrite,PCWriteCond,Zero,pc_we);
MUX_2_32 MUX1(pc_out,(ALU_out>>2),lorD,address);

dist_mem_gen_0
dist(address[7:0],B_out,addr,clk,MemWrite,Mem_data,out_data);//
/read

IR_Module
iR(Mem_data,IRWrite,clk,rst,IR_op,IR_25_21,IR_20_16,IR_15_0);
MUX_2_5 MUX2(IR_20_16,IR_15_0[15:11],RegDst,Write_register);
RegFile
RegF(IR_25_21,IR_20_16,addr[4:0],Write_data_regf,Write_registe
r,RegWrite,rst,clk,RD0,RD1,reg_data);
ALU_A a(RD0,clk,rst,A_out);
ALU_B b(RD1,clk,rst,B_out);
//DataLate A(A_out_1,clk,A_out);
//DataLate B(B_out_1,clk,B_out);
```

```
MUX_2_32 MUX3(pc_out,A_out,ALUSrcA,ALU_1);
MemoryDataR mdr(Mem_data,rst,clk,memdataout);
MUX_2_32 MUX4(ALU_out,memdataout,MemtoReg,Write_data_regf);
Sign_extend s_e(IR_15_0,I);
//Shift_left_2 s_l_1(I,I_shift);
assign I_shift = I;
ALU_control Acontrol(IR_op,IR_15_0[5:0],ALUOp,alu_c);
//Shift_left_2
s_l_2({6'b000000,IR_25_21,IR_20_16,IR_15_0},Jaddress);
assign Jaddress = {6'b000000,IR_25_21,IR_20_16,IR_15_0};
MUX_3_32 MUX6(alu_result,ALU_out,Jaddress,PCSource,pc_src);
ALL_Control
control(IR_op,clk,rst,PCWriteCond,PCWrite,lorD,MemRead,MemWrit
e,MemtoReg,IRWrite,PCSource,ALUOp,ALUSrcB,ALUSrcA,RegWrite,Reg
Dst,current_state,run,run_clk);

//MUX_4_32 MUX5(B_out,4,I,I_shift,ALUSrcB,ALU_2);
MUX_4_32
MUX5(B_out,I,I_shift,ALUSrcB,ALU_2);//1?0?6?0?5?0?6?0?2° ü?0?2
?0?1?0?8?0?3?0?8?0?7?0?1?0?3?0?9?0?9
ALU alu(ALU_1,ALU_2,alu_c,alu_result,Zero);
ALU_OUT aluout1(alu_result,clk,rst,ALU_out);
//DataLate ALUOut(alu_o,clk,ALU_out);
endmodule




/*module MUX_2_8(
input [7:0]value_0,
```

```
input [7:0]value_1,
input choose,
output reg[7:0]value
    );
always@(*)
begin
    case(choose)
    1'b1:
          value = value_1;
    1'b0:
          value = value_0;
    endcase
end
endmodule
*/
module MemoryDataR(
input [31:0]memdata,
input rst,
input clk,
output [31:0]memdataout
    );
reg [31:0]memd;
assign memdataout = memd;
always @(posedge clk or posedge rst) begin
    if(rst)
          memd <= 0;
    else begin
          memd <= memdata;
    end
```

```
end
endmodule


module RegFile(ra0,ra1,ra2,wa,wd,we,rst,clk,rd0,rd1,rd2
    );
parameter ADDR = 5;
parameter NUM = 32;
parameter SIZE = 32;
input [ADDR-1:0]ra0;
input [ADDR-1:0]ra1;
input [ADDR-1:0]ra2;
input [SIZE-1:0]wa;
input [ADDR-1:0]wd;
input we;
input rst;
input clk;
output [SIZE-1:0]rd0;
output [SIZE-1:0]rd1;
output [SIZE-1:0]rd2;
//output [SIZE-1:0]rd3;
reg [SIZE-1:0]register[0:NUM-1];
integer i;

always @(posedge clk or posedge rst) begin
    if (rst) begin
          // reset
          for(i = 0;i<NUM;i=i+1)
                register[i] <= 0;  //初始化操作
```

```
    end
    else if (!rst) begin
          if(we)begin
                if(wd!=0)
                      register[wd] <= wa;  //写操作
          end
    end
end

assign rd0 = register[ra0];  //读操作
assign rd1 = register[ra1];
assign rd2 = register[ra2];
//assign rd3 = 0;
endmodule


module Sign_extend(
input [15:0]in,
output [31:0]out
    );
assign out = in[15]==0?{16'b0,in}:{16'b1,in};
endmodule

/*module Zero_extend(
input [15:0]in,
output [31:0]out
    );
assign out ={16'b0,in};
endmodule
```

```
*/
/*module Shift_left_2(
input [31:0]in,
output [31:0]out
    );
assign out = in<<2;
endmodule
*/
module ALU_A(
input [31:0]in,
input clk,
input rst,
output reg[31:0]out
    );
always @(posedge clk or posedge rst) begin
    if(rst == 1)
          out = 0;
    else begin
          out = in;
    end
end
endmodule


module IR_Module(
input [31:0]IR_in,
input IRWrite,
input clk,
input rst,
```

```
output [5:0]IR_op,
output [4:0]IR_25_21,
output [4:0]IR_20_16,
output [15:0]IR_15_0
);
reg [31:0]IR;
assign IR_op = IR[31:26];
assign IR_25_21 = IR[25:21];
assign IR_20_16 = IR[20:16];
assign IR_15_0 = IR[15:0];
always @(negedge clk or posedge rst) begin//decrease
    if(rst)
          IR <= 0;
    else
    begin
          if(IRWrite==1)
                IR <= IR_in;
          else begin
                IR <= IR;
          end
    end
end
endmodule

module ALU_B(
input [31:0]in,
input clk,
input rst,
output reg[31:0]out
```

```
    );
always @(posedge clk or posedge rst) begin
    if(rst == 1)
          out = 0;
    else begin
          out = in;
    end
end
endmodule

module ALU(
    input [31:0]a,
    input [31:0]b,
    input [3:0]s,
    output reg [31:0]y,
    output Zero

    );
    assign Zero = (y==0)?1:0;  //判0
    reg [32:0] temp;
    always @(a,b,s)
    begin
    case(s)
        4'b0000:   //&
        begin
            y<=a&b;
        end
        4'b0001:    //|
        begin
```

```
    y<=a|b;
end
4'b0010:  //xor
begin
    y<=a^b;
end
4'b0011:    //nor
begin
        y<=~(a|b);
end
4'b0100:    //add
begin
    /*temp={1'b0,a}+{1'b0,b};
    y=temp[31:0];*/
    y <= a+b;
end
4'b0101:    //sub
begin
    /*temp={1'b0,a}-{1'b0,b};
    y=temp[31:0];*/
    y <= a-b;
end
4'b0110:      //slt
begin
  if(a[31]==1&&b[31]==1)
        if(a[30:0]>b[30:0])
              y <= 1;
        else begin
              y <= 0;
```

```
                end
            else if(a[31]==1&&b[31]==0)
            begin
                y <= 1;
            end
            else if(a[31]==0&&b[31]==1)
            begin
                y <= 0;
            end
            else begin
                if(a[30:0]<b[30:0])
                    y <= 1;
                else begin
                    y <= 0;
                end
            end
          end
          default:begin
            y <= 0;
          end
    endcase
    end
endmodule

module PC_Module(
input [31:0]pc_value,
input pc_we,
input rst,
input clk,
```

```
output reg[31:0]pc_out
    );
always @(posedge clk or posedge rst) begin
    if (rst) begin
          // reset
          pc_out = 0;
    end
    else begin
          if(pc_we==1)
                pc_out = pc_value;
          else begin
                pc_out = pc_out;
           end
    end
end
endmodule



module ALU_OUT(
input [31:0]ALU_in,
input clk,
input rst,
output reg[31:0]ALU_out
    );
always @(posedge clk or posedge rst) begin
    if(rst == 1)
          ALU_out <= 0;
    else begin
```

```
            ALU_out <= ALU_in;
    end
end
endmodule

module MUX_2_32(
input [31:0]value_0,
input [31:0]value_1,
input choose,
output reg[31:0]value
    );
always@(*)
begin
    case(choose)
    1'b1:
            value = value_1;
    1'b0:
            value = value_0;
    endcase
end
endmodule

module MUX_2_5(
input [4:0]value_0,
input [4:0]value_1,
input choose,
output reg[4:0]value
    );
always@(*)
```

```
begin
    case(choose)
    1'b1:
          value = value_1;
    1'b0:
          value = value_0;
    endcase
end
endmodule

//ALUOP: 00 01 10 11
module ALU_control(
input [5:0] Op,
input [5:0] IR,
input [1:0] ALUOp,
output reg[3:0]alu_control
    );
parameter [3:0]add = 4'b0100,
sub = 4'b0101,
and_ = 4'b0000,
or_ = 4'b0001,
xor_ = 4'b0010,
nor_ = 4'b0011,
slt = 4'b0110,
wrong = 4'b1000;
always @(*) begin
    case(ALUOp)
    2'b00:alu_control <= 4'b0100;//add
    2'b10:begin
```

```
case(IR)//funct
6'b100000://add
begin
      alu_control <= 4'b0100;
end
6'b100010://sub
begin
      alu_control <= 4'b0101;
end
6'b100100://and
begin
      alu_control <= 4'b0000;
end
6'b100101://or
begin
      alu_control <= 4'b0001;
end
6'b100110://xor
begin
      alu_control <= 4'b0010;
end
6'b100111://nor
begin
      alu_control <= 4'b0011;
end
6'b101010://slt
begin
      alu_control <= 4'b0110;
end
```

```
        default:begin
                alu_control <= 4'b1000;
        end
        endcase
end
2'b01:begin
        alu_control <= 4'b0101;
end
2'b11:begin
        case(Op)
        6'b001000:
        begin
                alu_control <= 4'b0100;
        end
        6'b001100:
        begin
                alu_control <= 4'b0000;
        end
        6'b001101:
        begin
                alu_control <= 4'b0001;
        end
        6'b001110:
        begin
                alu_control <= 4'b0010;
        end
        6'b001010:
        begin
                alu_control <= 4'b0110;
```

```
            end
            default:begin
                  alu_control <= 4'b0100;
            end
            endcase
    end
    endcase
end

endmodule

module MUX_3_32(
input [31:0]value_0,
input [31:0]value_1,
input [31:0]value_2,
input [1:0]choose,
output reg[31:0]value
    );
always@(*)
begin
    case(choose)
    2'b10:
          value = value_2;
    2'b01:
          value = value_1;
    2'b00:
          value = value_0;
    default:begin end
    endcase
```

```
end
endmodule

module MUX_4_32(
input [31:0]value_0,
input [31:0]value_2,
input [31:0]value_3,
input [1:0]choose,
output reg[31:0]value
    );
always@(*)
begin
    case(choose)
    2'b11:
          value = value_3;
    2'b10:
          value = value_2;
    2'b01:
          value = 32'b001;
    2'b00:
          value = value_0;
    endcase
end
endmodule



module PCWE(Op,pc_we,PCWriteCond,Zero,pcwe);
input [5:0]Op;
```

```
input pc_we;
input PCWriteCond;
input Zero;
output reg pcwe;

always@(*)
begin
    if(Op == 6'b000100)//BEQ
          pcwe <= pc_we|(PCWriteCond&Zero);
    else if(Op == 6'b000101)//bne
          pcwe <= pc_we|(PCWriteCond&(~Zero));
    else begin
          pcwe <= pc_we;
    end
end
endmodule

module ALL_Control(
Op,clk,rst,PCWriteCond,PCWrite,lorD,MemRead,MemWrite,MemtoReg,
IRWrite,PCSource,ALUOp,ALUSrcB,ALUSrcA,RegWrite,RegDst,current
_state,run,run_clk
    );
input clk;
input rst;
input [5:0]Op;
output reg PCWriteCond;
output reg PCWrite;
output reg lorD;
output reg MemRead;
```

```
output reg MemWrite;
output reg MemtoReg;
output reg RegDst;
output reg RegWrite;
output reg IRWrite;
output reg [1:0]PCSource;
output reg[1:0]ALUOp;
output reg[1:0]ALUSrcB;
output reg ALUSrcA;
input run;
output reg run_clk;
parameter [3:0]IF = 4'b0000,
                    ID = 4'b0001,
                    MEMADRESS = 4'b0010,
                    LOAD = 4'b0011,
                    LW_WB = 4'b0100,
                    SEND = 4'b0101,
                    EXE = 4'b0110,
                    R_WB = 4'b0111,
                    BEQC = 4'b1000,
                    JMC = 4'b1001,
                    EXE_I = 4'b1010,
                    I_WB = 4'b1011,
                    INVALID = 4'b1111;
output reg[3:0] current_state;
reg [3:0] next_state;

always @(posedge run or posedge clk or posedge rst) begin
    if(rst)
```

```
    begin
          run_clk <= 0;
    end
    else begin
          if(run == 1)
                run_clk <= 1;
          else begin
                if(next_state == IF)
                      run_clk <= 0;//一条指令运行完毕，又到了取
指阶段，则让时钟为0
          end
    end
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
          // reset
          current_state <= INVALID;//IF state
    end
    else begin
          current_state <= next_state;
    end
end

always@(current_state or Op) begin
    next_state = 4'b0000;
    case(current_state)
    INVALID:begin
          next_state = IF;
```

```
    end
    IF:begin
          next_state = ID;
    end
    ID:begin
          if(Op == 6'b100011||Op == 6'b101011)//lw or sw
          begin
                next_state = MEMADRESS;  //计算内存地址
          end
          else if(Op==6'b000000) begin  //R 型
                next_state = EXE;
          end
          else if(Op == 6'b000100||Op == 6'b000101) begin
//beq and bne
                next_state = BEQC;
          end
          else if(Op == 6'b000010) begin  //j 型 跳转
                next_state = JMC;
          end
          else if(Op[5:3] == 3'b001) begin  //I 型
                next_state = EXE_I;
          end
    end
    MEMADRESS:begin
          if(Op == 6'b100011)// lw3
          begin
                next_state = LOAD;  //lw 下次进入加载阶段
          end
          else begin
```

```
            next_state = SEND;  //sw3 进入写阶段
      end
end
LOAD:begin
      next_state = LW_WB;  //lw4 进入写寄存器阶段
end
LW_WB:begin
      next_state = IF;   //lw5 结束,再次取指
end
SEND:begin     //sw4 写回，结束了，下一阶段取指
      next_state = IF;
end
EXE:begin //R3 执行，下一阶段写回
      next_state = R_WB;
end
R_WB:begin //R4 R 结束，下一阶段取指
      next_state = IF;
end
BEQC:begin //beq3 bne3 结束，下一阶段取值
      next_state = IF;
end
JMC:begin   //JMC3 结束
      next_state = IF;
end
EXE_I:begin  //I3 i 型执行，下一阶段写回
      next_state = I_WB;
end
I_WB:begin  //I4 写回，结束
      next_state = IF;
```

```
    end
    default:begin
          next_state = INVALID;
    end
    endcase
end

always @(posedge clk) begin  //下一状态的控制信号使用的是上一个时钟中修改的
    //PCWriteCond
if(next_state == BEQC)
    PCWriteCond = 1;  //PCwrite 只有在 beq bne 中有效
else begin
    PCWriteCond = 0;
end

//PCWRITE  只在取值和跳转时有效
if(next_state == IF || next_state == JMC)
    PCWrite = 1;
else begin
    PCWrite = 0;
end

//lord 只有在 load 和 send 阶段有效，lord 为 1 说明是对数据存储器读写
if(next_state == LOAD ||next_state == SEND)
    lorD = 1;
else begin
    lorD = 0;
```

```
end

//memwrite 在 sw 的写回有效
if(next_state == SEND)
    MemWrite = 1;
else begin
    MemWrite = 0;
end

//memtoreg lw_wb 为 1，即内存中数据写入，0 则 alu 结果写入
if(next_state == LW_WB)
    MemtoReg = 1;
else begin
    MemtoReg = 0;
end
//regdst
if(next_state == R_WB)//IWB is zero
    RegDst = 1;
else begin
    RegDst = 0;
end

//RegWrite
if(next_state == LW_WB|| next_state == R_WB||next_state==I_WB)
    RegWrite = 1;
else begin
    RegWrite = 0;
end
```

```
//IRWrite
if(next_state == IF)
    IRWrite = 1;
else begin
    IRWrite = 0;
end

//pcsource
if(next_state == IF)
    PCSource = 2'b00;
else if(next_state == BEQC)
    PCSource = 2'b01;
else if(next_state == JMC)
    PCSource = 2'b10;
else begin
    PCSource = 2'b11;
end

//ALUOP
if(next_state == IF || next_state == MEMADRESS || next_state
== ID)
    ALUOp = 2'b00;
else if(next_state == EXE)begin
    ALUOp = 2'b10;
end
else if(next_state == BEQC)
    ALUOp = 2'b01;  //减
else if(next_state == EXE_I)begin
    ALUOp = 2'b11;
```

```
end
else begin
    ALUOp = 2'b00;
end

//ALUSrcB
if(next_state == MEMADRESS || next_state == EXE_I)
    ALUSrcB = 2'b10;
else if(next_state == IF)
    ALUSrcB = 2'b01;
else if(next_state == ID)
    ALUSrcB = 2'b11;
else begin
    ALUSrcB = 2'b00;
end

//ALUSRCA
if(next_state == ID||next_state == IF)
    ALUSrcA = 0;
else begin
    ALUSrcA = 1;
end
//MemRead==1
MemRead = 1;
end
endmodule

module DISPLAY_UNIT(
```

```
cont,step,mem,inc,dec,pc,mem_data,reg_data,clk,rst,run,addr,le
d,an,seg,dp
    );
input cont;
input step;//always@step
input mem;
input inc;
input dec;
input [7:0]pc;
input [31:0]mem_data;
input [31:0]reg_data;
input clk;
input rst;
output run;
output [7:0]addr;
output [15:0]led;
output [7:0]an;
output [6:0]seg;
output dp;


wire [31:0]data;
wire clk_10hz;
assign data = mem == 1?mem_data:reg_data;
RUN r(cont,step,clk,rst,run);
cout_5hz c1(clk,rst,clk_10hz);
Address a(inc,dec,clk_10hz,rst,addr);
Display d(pc,data,addr,clk,rst,led,an,seg,dp);
```

```
endmodule


//module about run
module RUN(
input cont,
input step,
input clk,
input rst,
output run
    );
wire clk_out;
//run = 1/run = 0;
cout_1ms c3(clk,rst,clk_out);  //1000HZ prevent the cycle too
short
reg real_run;
reg flag;
always @(posedge clk_out or posedge rst) begin
    if (rst) begin
          // reset
          real_run <= 0;
          flag <= 0;
    end
    else begin
          if(step == 1 && flag == 0)
          begin
                real_run <= 1;
                flag <= 1;
          end
          else if(step == 0 && flag == 1)
```

```
            begin
                  flag <= 0;
                  real_run <= 0;
            end
            else begin
                  real_run <= 0;
            end
      end
end
assign run = real_run | cont;

endmodule

//address
module Address(
input inc,
input dec,
input clk,
input rst,
output reg [7:0]addr
    );

always@(posedge clk or posedge rst)
begin
    if(rst == 1)
          addr = 8'b0;
    else begin
          if(inc == 1)
                addr = addr + 1;
```

```
            else if(dec == 1)begin
                  addr = addr - 1;
            end
            else begin
                  addr = addr;
            end
    end
end

endmodule

//display parameter
module Display(
input [7:0]pc,
input [31:0]data,
input [7:0]address,
input clk,
input rst,
output [15:0]led,
output reg[7:0]an,
output [6:0]seg,
output dp
    );
wire clk_1ms;
cout_1ms t(clk,rst,clk_1ms);
assign led[15:8] = address;
assign led[7:0] = pc;
reg [3:0]in;
reg [2:0]count;
```

```
Change change(in,seg,dp);
always @(posedge clk_1ms or posedge rst) begin
    if (rst) begin
          // reset
          count <= 0;
          an <= 8'b11111111;
    end
    else begin
        an[0] = 1;
        an[1] = 1;
        an[2] = 1;
        an[3] = 1;
        an[4] = 1;
        an[5] = 1;
        an[6] = 1;
        an[7] = 1;
          an[count] = 0;
          count = count + 1;
    end
end

always@(count)
begin
    if(rst)
          in = 0;
    else begin
          case(count)  //set in for the pre cycle
          3'b000:begin
                in = data[31:28];
```

```
            end
            3'b111:begin
                  in = data[27:24];
            end
            3'b110:begin
                  in = data[23:20];
            end
            3'b101:begin
                  in = data[19:16];
            end
            3'b100:begin
                  in = data[15:12];
            end
            3'b011:begin
                  in = data[11:8];
            end
            3'b010:begin
                  in = data[7:4];
            end
            3'b001:begin
                  in = data[3:0];
            end
            endcase
      end
end



endmodule
```

```
module cout_1ms(  //the time of one cycle doesn't be
confirmed,so it may need to change to follow the time of one
period
    input clk,
    input rst_n,
    output reg Q
    );
    reg   [24:0]      count;
    always@(posedge clk or posedge rst_n)
    begin
        if(rst_n)
            count    <= 25'd0;
        else if(count >= 25'd50000)
            count    <= 25'd0;
        else
            count    <= count + 25'd1;
    Q =  (count >= 25'd25000) ? 1'b1 : 1'b0;
    end
endmodule

module Change(
input [3:0]in,
output reg[6:0]seg,
output reg dp
    );

always@(*)
begin
    case(in)
```

```
    4'b0:{dp,seg} = 8'hC0;
    4'b1:{dp,seg} = 8'hF9;
    4'b10:{dp,seg} = 8'hA4;
    4'b11:{dp,seg} = 8'hB0;
    4'b100:{dp,seg} = 8'h99;
    4'b101:{dp,seg} = 8'h92;
    4'b110:{dp,seg} = 8'h82;
    4'b111:{dp,seg} = 8'hF8;
    4'b1000:{dp,seg} = 8'h80;
    4'b1001:{dp,seg} = 8'h90;
    4'b1010:{dp,seg} = 8'h88;
    4'b1011:{dp,seg} = 8'h83;
    4'b1100:{dp,seg} = 8'hC6;
    4'b1101:{dp,seg} = 8'hA1;
    4'b1110:{dp,seg} = 8'h84;
    4'b1111:{dp,seg} = 8'h8E;
    endcase
end

module cout_10hz(
    input clk,
    input rst_n,
    output reg Q
    );
    reg   [24:0]      count;
    always@(posedge clk or negedge rst_n)
    begin
        if(rst_n)
            count    <= 25'd0;
```

```
        else if(count >= 25'd5000000)
            count    <= 25'd0;
        else
            count    <= count + 25'd1;
    Q =  (count >= 25'd2500000) ? 1'b1 : 1'b0;
    end
endmodule


module cout_5hz(
    input clk,
    input rst_n,
    output reg Q
    );
    reg   [24:0]      count;
    always@(posedge clk or negedge rst_n)
    begin
        if(rst_n)
            count    <= 25'd0;
        else if(count >= 25'd10000000)
            count    <= 25'd0;
        else
            count    <= count + 25'd1;
    Q =  (count >= 25'd5000000) ? 1'b1 : 1'b0;
    end
endmodule
```