计组第五次实验报告——Mips体系单周期CPU的功能组成和逻辑实现

姓名：颜铭　学号：2013365　专业：智能科学技术

一、 实验目的：

1. 理解Mips指令结构，熟悉Mips指令集中常用指令的功能和编码，并根据Mips指令类型学习归纳分类方法
2. 了解熟悉Mips体系的处理器架构，学习哈佛结构在存储器中的应用
3. 熟悉并掌握单周期CPU的组成原理和逻辑功能设计方法，为拓展实验学习设计多周期CPU和流水线级CPU打下基础
4. 进一步加强利用Verilog语言进行顶层设计和仿真综合的能力

二、 实验内容:

1. 学习Mips指令集并深入理解Mips常用指令的编码通用规则和功能类型设计。
2. 在源代码确定的二十条指令基础上通过仿真验证指令功能，以此替代上板烧录并报告结果。在实验过程中正确认识到单周期CPU结构的简约性和完备性，要求解析一条读入装载指令、一条写回存储指令，一条以上的跳转指令和10条以上的封装在算术逻辑单元ALU中的基础运算指令。实验重点偏向在搭建CPU的结构的理解上实现简单的内嵌指令程序块。
3. 实验源码给出的指令交互性并不明显并且没有特异的功能实现。为体会CPU的功能设计，实验要求利用汇编语言编写函数实现某一功能，实现不同功能类型的指令的简单逻辑执行。
4. 给出代码模块实现的逻辑框图如下，并利用综合设计进一步得到完整的CPU模块框图：

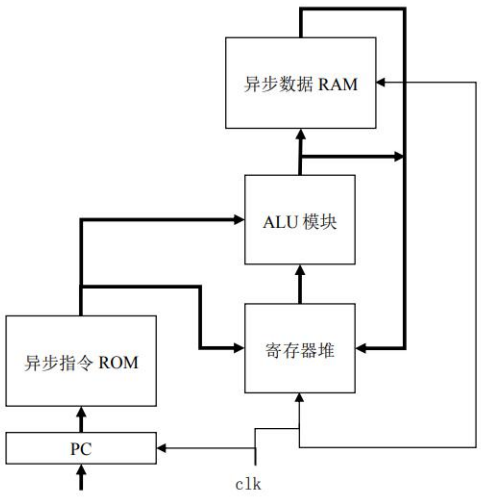


图 1.实验源代码的逻辑框图

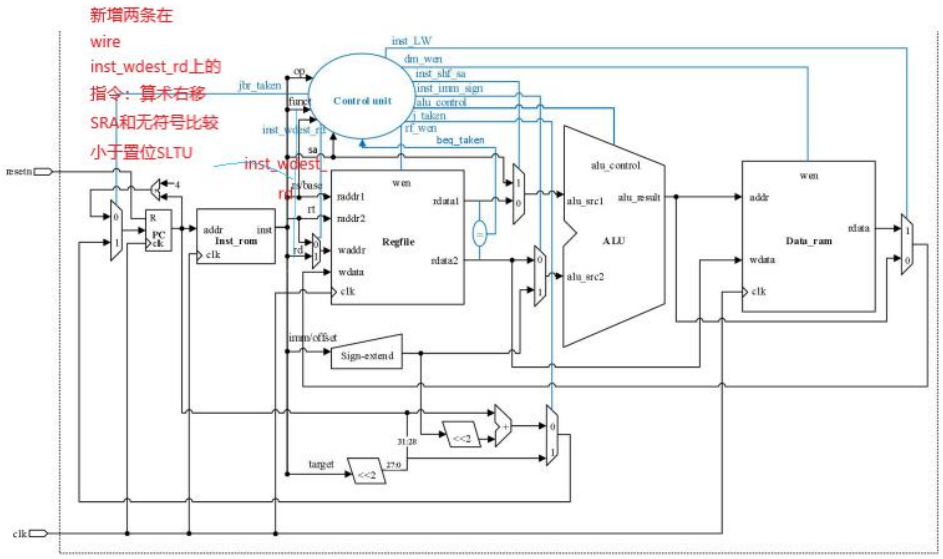


图 2 CPU 的数字系统模块框图

5、本实验根据要求分为三个部分：

①对实验源代码给出的 20 条指令进行归纳并验证其逻辑功能，重点学习运算指令

②为汇编程序准备，学习数据存储器和指令存储器分开的哈佛结构，在指令存储器内嵌指令调试跳转指令和移位指令，为循环和二进制位操作准备功能设计。

③动手实现汇编程序，程序块基于 C++语言中**求解一个非负数的二进制最高位和最低位并判断是否为 2 的整数次幂**的函数，具体分析见实验原理

三、 实验原理

1. 指令基本定义：Mips 指令类型分为三种，R 型指令、I 型指令和 J 型指令

   ① R 型指令的构成为：

   31-26 位为 Opcode 源操作码，默认为 000000，区别于 I 型指令

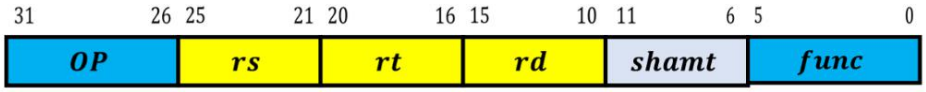
   25-21 位为 rs,源操作数 1；20-16 位为 rt,源操作数 2,支持单目和双目运算

   15-10 位为 rd，目标操作数，作为结果的访问地址，可以输入输出到自身实现运算赋值

   11-6 位为 Shamt(Shf)为偏移量，一般非移位操作默认为 000000

   5-0 位为 Funct,功能函数编码，区分具体的 R 型指令

R型

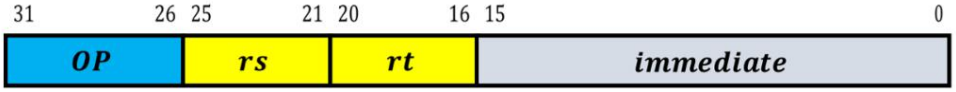


   常见的 R 型指令包括加减运算、逻辑与或非运算、逻辑左右移位运算等

   ② I 型指令的构成为：

31-26 位为 Opcode,源操作码，根据功能设计编码确定
25-21 位为 rs,源操作数 1；20-16 位为 rt,源操作数 2。
15-0 位为立即数 Immediate,是用户定义的操作数值，执行调整运算变量和地址与数据偏移功能

I型



常见的 I 型指令包括立即数加 Addiu，立即数与 Andi，不等转移 Bne，写入存字 Sw，读出取字 Lw 操作

③ J 型指令的构成为：

31-26 位为 Opcode,具体的跳转操作的长短和条件由源操作码确定
25-0 位为 Address,长跳转地址，在原来指针基础上，指针前四位+Address+00(Address<<2)即为新地址。I 型指令中的相等和不等跳转见指令解析

J型

常见的 J 型指令包括跳转指令和调用指令，根据条件可以细分。单周期 CPU 只需实现跳转指令 J

2. 实验指令归纳:

指令类型	汇编指令	指令码	源操作数 1	源操作数 2	源操作数 3	目的寄存器	功能描述
R 型指令	Addu rd,rs,rt	000000|rs|rt|rd 00000|10001	rs	rt		rd	GPR[rd]=GPR[rs]+GPR[rt
	Subu rd,rs,rt	000000|rs|rt|rd|00 000|100011	rs	rt		rd	GPR[rd]=GPR[rs]-GPR[rt]
	Slt rd,rs,rt	000000|rs|rt|rd|00 000|101010	rs	rt		rd	GPR[rd]= (sign(GPR[rs])<sign(GPR[rt])
	And rd,rs,rt	000000|rs|rt|rd|00 000|100100	rs	rt		rd	GPR[rd]=GPR[rs]&GPR[rt]
	Nor rd,rs,rt	000000|rs|rt|rd|00 000|100111	rs	rt		rd	GPR[rd]=~(GPR[rs]|GPR[rt])
	Or rd,rs,rt	000000|rs|rt|rd|00 000|100101	rs	rt		rd	GPR[rd]=GPR[rs]|GPR[rt]
	Xor rd,rs,rt	000000|rs|rt|rd|00 000|100110	rs	rt		rd	GPR[rd]=GPR[rs]^GPR[rt]
	Sll rd,rt,shf	000000|00000|rt|r d|shf|000000		rt		rd	GPR[rd]=zero(GPR[rt])<<shf
	Srl rd,rt,shf	000000|00000|rt|r d|shf|000010		rt		rd	GPR[rd]=zero(GPR[rt])>>shf
	Sra	000000|00000|rt|r		rt		rd	GPR[rd] ← sign(GPR[rt]) >> shf

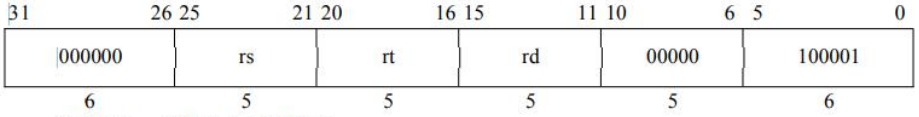
	rd,rt,shf	d|shf|000011					
	Sltu rd,rs,rt	000000|rs|rt|rd|00000|101011	rs	rt		rd	GPR[rd] ← (zero(GPR[rs]) < zero(GPR[rt]))
I 型指令	Addiu rt,rs,imm	001001|rs|rt|imm	rs	sign_ext imm		rt	GPR[rt]=GPR[rs]+ sign_ext (imm)
	Beq rs,rt,offset	000100|rs|rt|offset	rs	rt			if GPR[rs]=GPR[rt] then PC=PC+ sign_ext
	Bne rs,rt,offset	000101|rs|rt|offset	rs	rt			if GPR[rs]≠GPR[rt] then PC=PC+ sign_ext (offset)<<
	Lw rt,offset(b)	100011|b|rt|offset	b	sign_ext (offset)		rt	GPR[rt]=Mem[GPR[b]+sign_ext(offset)]
	Sw rt,offset(b)	101011|b|rt|offset	b	sign_ext (offset)	rt		Mem[GPR[b]+sign_ext(offset)]=GPR[rt]
	Lui rt,imm	001111|00000|rt|imm					GPR[rt]= {imm, 16'd0}
J 型指令	j target	000010|target					PC={PC[31:28],target<<2}

表中 GPR 为数据寄存器依赖 regfile

Mem 为可读数据存储器 data_ram

3. 指令解析

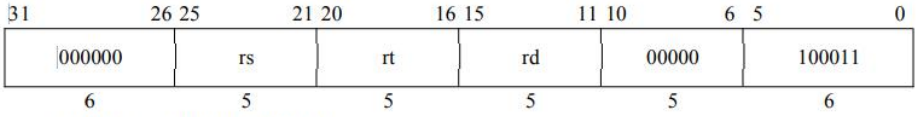
ADDU rd, rs, rt　　R 型



GPR[rd] ← GPR[rs] + GPR[rt]

（2） 无符号减法

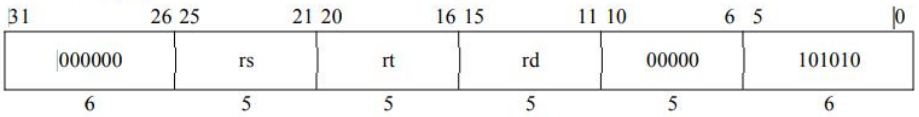
SUBU rd, rs, rt　　R 型



GPR[rd] ← GPR[rs] - GPR[rt]

（3） 有符号比较，小于置位

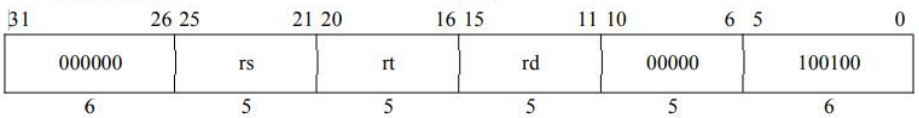
SLT rd, rs, rt　　R 型



GPR[rd] ← (sign(GPR[rs]) < sign(GPR[rt]))

（4） 按位与

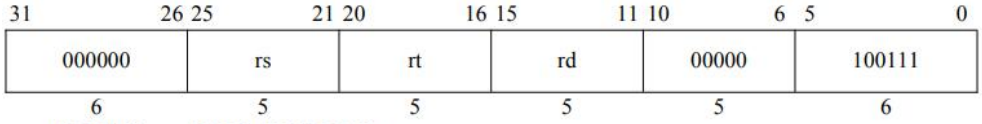
AND rd, rs, rt　　R 型



GPR[rd] ← GPR[rs] & GPR[rt]

（5） 按位或非

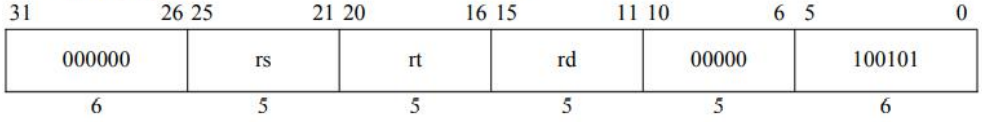
NOR rd, rs, rt　　R 型



GPR[rd] ← ~(GPR[rs] | GPR[rt])

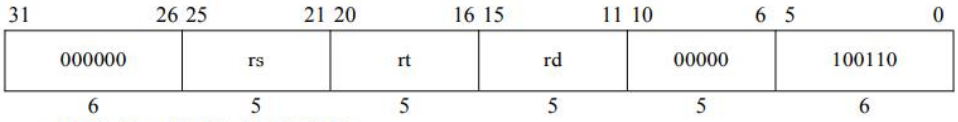
（6） 按位或

OR rd, rs, rt　　R 型



GPR[rd] ← GPR[rs] | GPR[rt]

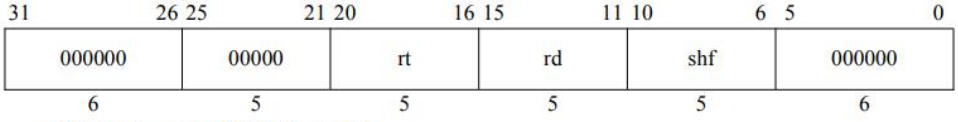
XOR rd, rs, rt　　R 型



GPR[rd] ← GPR[rs] ^ GPR[rt]

（8） 逻辑左移

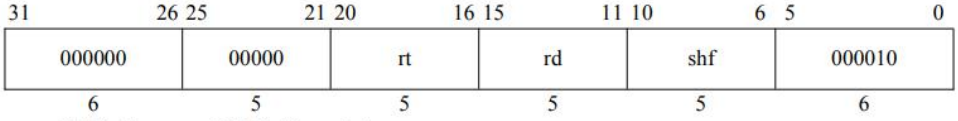
SLL rd, rt, shf　　R 型



GPR[rd] ← zero(GPR[rt]) << shf

（9） 逻辑右移

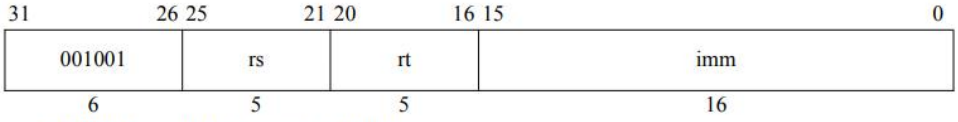
SRL rd, rt, shf　　R 型



GPR[rd] ← zero(GPR[rt]) >> shf

（10） 立即数、无符号加法

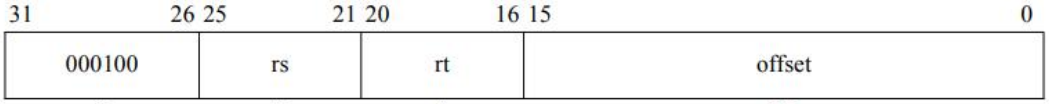
ADDIU rt, rs, imm　　I 型



GPR[rt] ← GPR[rs] + sign_ext(imm)

（11） 相等跳转

BEQ rs, rt, offset　　I 型

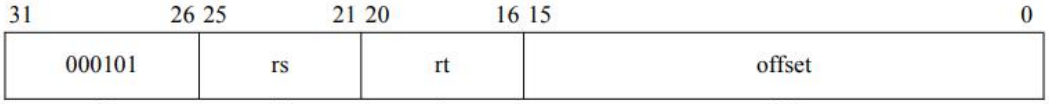


if GPR[rs] = GPR[rt] then PC ← B_PC + sign_ext(offset)<<2

B_PC：分支跳转参与运算的 PC，在不考虑延迟槽时为分支跳转指令的 PC，考虑延迟槽时为延迟槽指令的 PC，即分支跳转指令的 PC+4。

（12） 不等跳转

BNE rs, rt, offset　　I 型

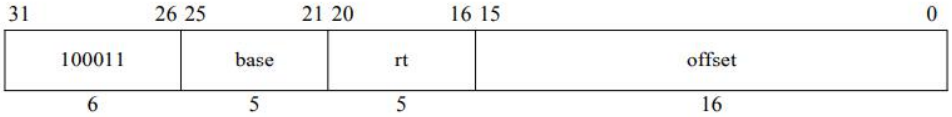


if GPR[rs] ≠ GPR[rt] then PC ← B_PC + sign_ext(offset)<<2

B_PC：分支跳转参与运算的 PC，在不考虑延迟槽时为分支跳转指令的 PC，考虑延迟槽时为延迟槽指令的 PC，即分支跳转指令的 PC+4。

（13） 装载字

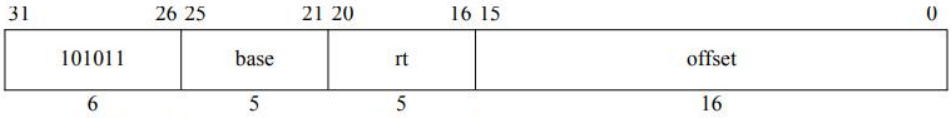
LW rt, offset(base)　　I 型



GPR[rt] ← Mem[GPR[base] + sign_ext(offset)]

（14） 存储字

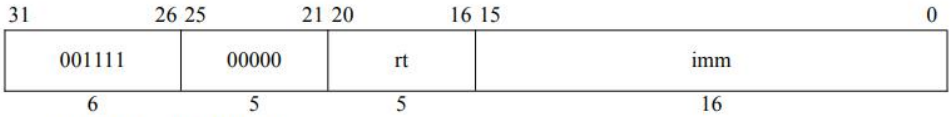
SW rt, offset(base)　　I 型



Mem[GPR[base] + sign_ext(offset)] ← GPR[rt]

（15） 立即数装载高位

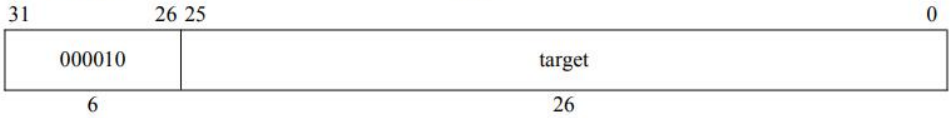
LUI rt, imm　　I 型



GPR[rt] ← {imm,16'd0}

（16） 直接跳转

J target　　J 型

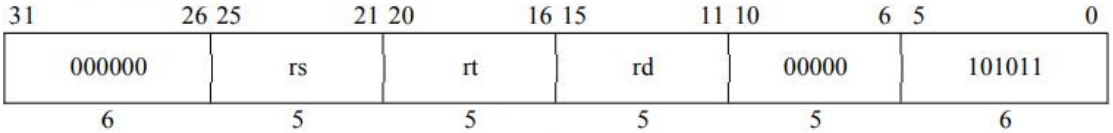


PC ← {B_PC[31:28], target << 2}

B_PC：分支跳转参与运算的 PC，在不考虑延迟槽时为分支跳转指令的 PC，考虑延迟槽时为延迟槽指令的 PC，即分支跳转指令的 PC+4。

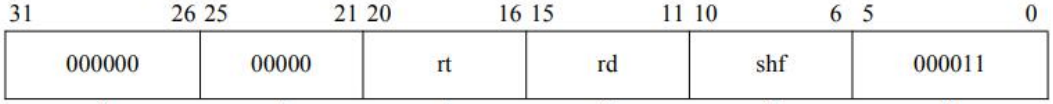
（17） 无符号小于置位

SLTU rd, rs, rt　　R 型



GPR[rd] ← (zero(GPR[rs]) < zero(GPR[rt]))

SRA rd, rt, shf　　R 型



GPR[rd] ← sign(GPR[rt]) >> shf

4. 仿真设计：

根据源码提供的 16 条不同指令归纳特性后对内嵌的 20 条指令的前 19 条指令进行仿真。在运行时钟周期后按顺序访问寄存器内存并称可读数据存储器 data_ram 的记忆化内容访问为打印输出和写进输入操作验证前 5 个寄存器是否符合运算结果，同时多次打印 14H 和 1CH 存储器中的写入内容，验证读写事件

指令地址	汇编指令	结果描述	机器指令的机器码	
			16 进制	二进制
00H	addiu $1,$0,#1	[$1] = 0000_0001H	24010001	0010_0100_0000_0001_0000_0000_0000_0001
04H	sll $2,$1,#4	[$2] = 0000_0010H	00011100	0000_0000_0000_0001_0001_0001_0000_0000
08H	addu $3,$2,$1	[$3] = 0000_0011H	00411821	0000_0000_0100_0001_0001_1000_0010_0001
0CH	srl $4,$2,#2	[$4] = 0000_0004H	00022082	0000_0000_0000_0010_0010_0000_1000_0010
10H	subu $5,$3,$4	[$5] = 0000_000DH	00642823	0000_0000_0110_0100_0010_1000_0010_0011
14H	sw $5,#19($1)	Mem[0000_0014H] = 0000_000DH	AC250013	1010_1100_0010_0101_0000_0000_0001_0011
18H	nor $6,$5,$2	[$6] = FFFF_FFE2H	00A23027	0000_0000_1010_0010_0011_0000_0010_0111
1CH	or $7,$6,$3	[$7] = FFFF_FFF3H	00C33825	0000_0000_1100_0011_0011_1000_0010_0101
20H	xor $8,$7,$6	[$8] = 0000_0011H	00E64026	0000_0000_1110_0110_0100_0000_0010_0110
24H	sw $8,#28($0)	Mem[0000_001CH] = 0000_0011H	AC08001C	1010_1100_0000_1000_0000_0000_0001_1100
28H	slt $9,$6,$7	[$9] = 0000_0001H	00C7482A	0000_0000_1100_0111_0100_1000_0010_1010
2CH	beq $9,$1,#2	跳转到指令 34H	11210002	0001_0001_0010_0001_0000_0000_0000_0010
30H	addiu $1,$0,#4	不执行	24010004	0010_0100_0000_0001_0000_0000_0000_0100
34H	lw $10,#19($1)	[$10] = 0000_000DH	8C2A0013	1000_1100_0010_1010_0000_0000_0001_0011
38H	bne $10,$5,#3	不跳转	15450003	0001_0101_0100_0101_0000_0000_0000_0011
3CH	and $11,$2,$1	[$11] = 0000_0000H	00415824	0000_0000_0100_0001_0101_1000_0010_0100
40H	sw $11,#28($0)	Men[0000_001CH] = 0000_0000H	AC0B001C	1010_1100_0000_1011_0000_0000_0001_1100
44H	sw $4,#16($0)	Mem[0000_0010H] = 0000_0004H	AC040010	1010_1100_0000_0100_0000_0000_0001_0000
48H	lui $12,#12	[$12] = 000C_0000H	3C0C000C	0011_1100_0000_1100_0000_0000_0000_1100
4CH	j 00H	跳转指令 00H	08000000	0000_1000_0000_0000_0000_0000_0000_0000

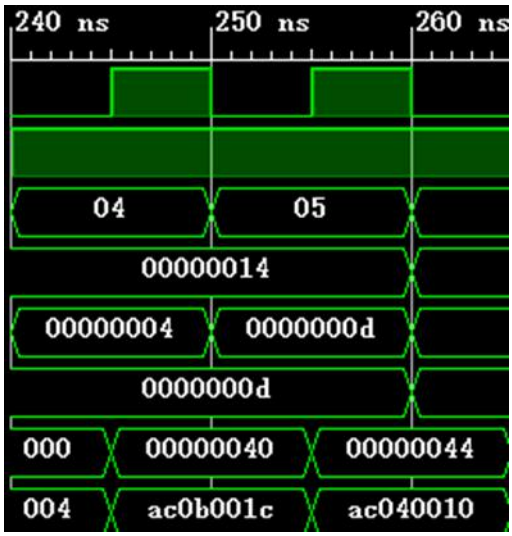
图 3 Mips 单周期 CPU 源代码实现的汇编指令程序

Testbench 有关代码如下:

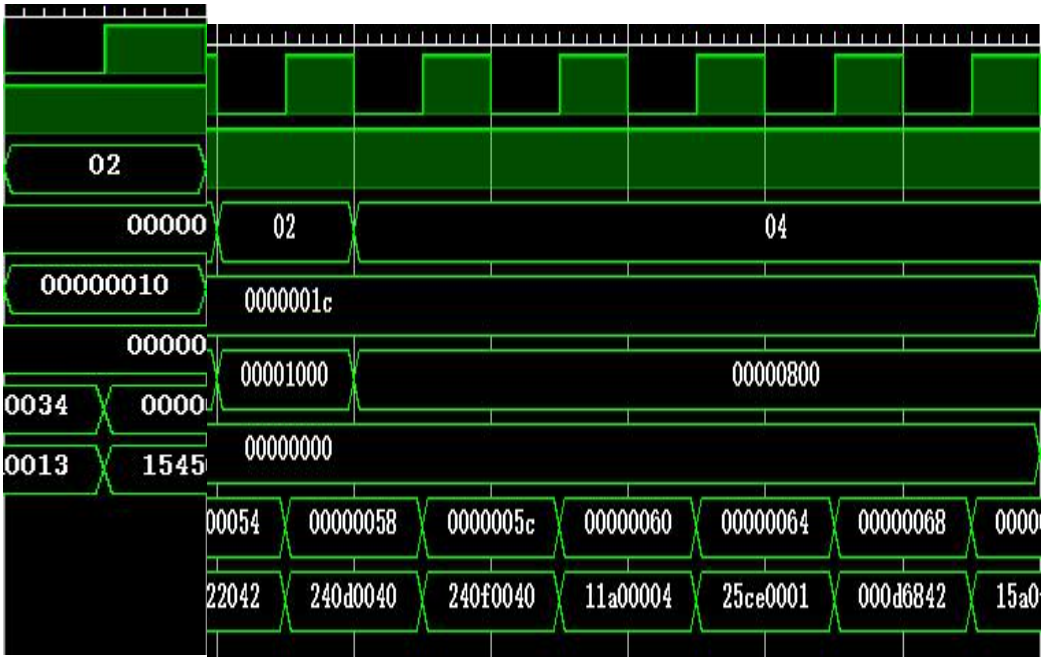
```
    single_cycle_cpu uut (
        .clk(clk),
        .resetn(resetn),
        .rf_addr(rf_addr),
        .mem_addr(mem_addr),
        .rf_data(rf_data),
        .mem_data(mem_data),
        .cpu_pc(cpu_pc),
        .cpu_inst(cpu_inst)
    );

    initial begin
        // Initialize Inputs
        clk = 0;
        resetn = 0;
        rf_addr = 0;
        mem_addr = 0;

        uut.rf_module.rf[15]=5'd18;//调试赋值并监视，便于模拟输入验
证汇编程序，注意只有 reg 类型可以时序赋值
        uut.rf_module.rf[16]=5'd31;

        // Wait 100 ns for global reset to finish
        #100;
      resetn = 1;
        // Add stimulus here
        //调试查看可读数据存储器 rom 即打印写入值监视载入值
        #100;
        mem_addr=20;//打印写入赋值
        #10;
        rf_addr=1; //按顺序查看寄存器中的运算结果
        #10
        rf_addr=2;
        #10;
        rf_addr=3;
        #10;
        rf_addr=4;
        #10;
        rf_addr=5;
        #10;
        rf_addr=8;
        mem_addr=28;//再此打印两次写入后的赋值观察变化
        #50;
```

```
        resetn=0;
        #10;
        rf_addr=10;
```

此外为了验证循环的 goto 语句是否正常，特别更改了第 20 条指令 J 的内容改为 J 15H　　32'h3C0C000C
并访问寄存器 4 查看内容是否被修改



10ns 一个执行周期后调试指针指向第 21 条指令检查循环赋值：



Testbench 相关代码如下:

```
        #10;
        rf_addr=15;//监视调试赋值
        #50;
        resetn=1;
        rf_addr=4;//观察跳转指令执行的线程结果改变
        uut.pc=32'h00000050;//验证 J 型指令功能后验证循环体
```

```
        #10;
        rf_addr=2;//ADDIU 的赋值条件
        #10;
        rf_addr=4;
        #50;
        mem_addr=16;//#监视受调试变量
```

内嵌指令代码 inst_rom.v 修改为

```
  assign inst_rom[19] = 32'h08000015; // 4CH: j      15H          | 跳转指令地址 15H
    //验证 J 型指令跳转和重复指令赋值
    assign inst_rom[20] = 32'h00011300;// 50H : sll $2,$1,#12 / 0000_0001H<---0000_0100H
    assign inst_rom[21] = 32'h00022042;// 54H : srl   $4,$2,#1 / $4=0000_0008H/跳转后一次闭环执行，（$4）随$2 的值移位两位
```

5. 汇编程序设计

验证了单周期 CPU 功能设计的完备性后可以通过 C++语言的函数代码编译汇编程序加强指令逻辑交互性，集中实现汇编程序块的逻辑功能

由于单周期 CPU 的算术逻辑单元 ALU 没有实现复杂的流水线运算除法和取模运算,**判断一个数是否为 2 的整数的幂函数**汇编需要通过循环+移位运算来分别求出二进制数的最高非零位和最低非零位，并通过最高非零位是否大于最低非零位解析二进制数的数码，若为 2 的整数幂则最高非零位和最低非零位的位置重合（非零位也即数码为 1）

具体地，通过右移并通过循环判断是否达到最高位的 0，即数每次右移一位后是否归零，很方便地就可以统计出最高非零位的位置
对于最低位，通过运算 Y&(-Y)得到 2 的最低非零位次幂的二进制表达比如 Y= xxxx_0100, -Y=~Y +1 =XXXX_1100,两者相与得到 0000_0100,即二进制的最低非零位数因子（x 与 X 逻辑相反）

再次通过循环就可以的得到最低非零位因子的最高也是最低位的位数在通过小于置位指令判断最低位位数是否小于最高位位数，若是说明不为 2 的整数次幂，若不是则成立(2 的整数次幂显然最高位和最低位位数一致)

通过分析就可得到 C++程序和相应内嵌汇编程序:

```
        int num; cin>>num;//Addiu $13 $0 #(imm)
        int tmp=num;//Addiu $15 $13 #(0) （$15)
        int neg=-num;//调试输入
        int cnt=0;//调试初始化
        //Beq $13 $0 #4
        while(num!=0){ // Bne $13 $0 #(-2)
```

```
        cnt+=1; // Addiu $14 $14 #1
        num>>=1;//Srl $13 $13 #1
        }
        cout<<cnt<<endl;// Sw $14 #15($0)
        int src=tmp&neg; //  And $16 $16 $15
        int cnts=0; //调试初始化
        // Beq $16 $0 #4
        while(cnts!=0){ //  Bne $16 $0 #(-2)
        cnts+=1; //  Addiu $17 $17 #1
        src>>=1; // Srl $16 $16 #1
        }
        cout<<cnts<<endl; //Sw $17 #16($0)
        flg= cnts<cnt ? 1 :0 ; // Slt $18 $17 $14
        ----------读取仿真信息即可根据调试逻辑值判断是否为2
的整数幂--------
        if (!flg) cout<<"Yes"<<endl;
    else cout<<"No"<<endl;
```

汇编指令的逻辑实现如下:

```
    //$0 默认值为0，即$zero 寄存器
        //求解一个数二进制的最高非零位的位置（从初始低位到高
位共有几位）
        Addiu $13 $0 #(imm) (1)0069 (2)0040 (3)0000 / int
num=(input);
        Beq $13 $0 #4 //若变量寄存器赋值为零则最高非零位默认
为0跳转到I/O接口    /是否执行循环while(num!=0)
        Addiu $14 $14 #1 //$14初始化为0，自加1记录最高非零
位的位置    /int cnt=0; cnt+=1;
        Srl $13 $13 #1 //($13)>>=1
        //!!! 基于移位操作实现的只是根据四舍五入除2后取整的
效果，因此不能将右移1位当做除2也就不能根据能否被2整除判断
数是否为2的整数幂，
        除法和取模指令是较高级的流水线cpu指令,本程序只基于
简单的移位运算指令加循环求解最高非零位和最低非零位信息，也可
以实现判断数是否为2的整数幂，即log2(num)=Z
        //num>>=1; /移位运算逐步统计统计位置
        Bne $13 $0 #-2 //if(num!=0) then goto Addiu $14
$14---cnt+=1;/while循环出口实现,不等条件反向跳转实现60H~68H
的循环执行
        Sw $14 #15（$0)//将最高非零位的位置写入可读存储器打印
值
```

```
        //求解二进制位的最低非零位的位置
        And $16 $16 $15
        //($16)为调试输入的相反数(int neg = -num)，($15)为初始化赋值时输入的副本(int tmp=num),执行($15)and ($16) 即num&(-num)的操作，得到仅有最低非零位的二进制数
        //比如 num = xxxx_0100, -num=~num+1 =XXXX_1100,两者相与得到0000_0100,即二进制的最低非零位数因子。
        //int src=num&(-num);

        Beq $16 $0 #4 //再次执行循环，判断是否存在最低非零位，若没有非零位则默认为0 / 是否执行循环while(src!=0)

        Addiu $17 $17 #1 //($17)为新的计数器 /int cnts=0; cnts+=1;

        Srl $16 $16 #1 //($16)>>=1;移位运算逐步统计位置

        Bne $16 $0 #-2 ;循环出口，移位数为零即停止。否则回到计数一步
        Sw $17 #16($0) //Mem[16] <--- ($17) /写入可读存储器打印值

    SLT $18 $17 $14 //小于置位，($18)为1即说明最低位小于最高位，不是2的整数幂。反之,最低位和最高位位置重合，说明是2的整数幂（0需特判），也可以根据此方法快速生成验证2的幂次
```

得到内嵌指令修改inst_rom.v的代码为

```
    assign inst_rom[22] = 32'h240D0040;//Addiu $13 $0 #(imm) 58H ($13)<--- input
        assign inst_rom[23] = 32'h25AF0000;//---->只需在这里调整保持输入即可/ 5CH  Addiuu $15 $0 #(imm) == Addiu $15 $13 #(0)  ($15)<---($13)
       //assign inst_rom[23] = 32'h240F0040
        assign inst_rom[24] = 32'h11A00004;// 60H   Beq $13 $0 #4
        assign inst_rom[25] = 32'h25CE0001;// 64H   Addiu $14 $14 #1
        assign inst_rom[26] = 32'h000D6842;// 68H   Srl $13 $13 #1
        assign inst_rom[27] = 32'h15A0FFFE;// 6cH   Bne $13 $0 #(-2)
        assign  inst_rom[28] = 32'hAC0E000F;//  70H    Sw  $14 #15($0)
        //循环体：64H - 68H - 6cH
        assign inst_rom[29] = 32'h020F8024;// 74H    And $16 $16
```

```
$15



        assign inst_rom[30] = 32'h12000004;// 78H     Beq $16 $0
#4
        assign inst_rom[31] = 32'h26310001;// 7cH    Addiu $17
$17 #1
        assign inst_rom[32] = 32'h00108042;// 80H    Srl $16 $16
#1
        assign inst_rom[33] = 32'h1600FFFE;// 84H    Bne $16 $0
#(-2)
        //循环体： 7cH - 80H - 84H
        assign  inst_rom[34] = 32'hAC110010;// 88H      Sw $17
#16($0）
    assign inst_rom[35] = 32'h022E902A;// 8CH    Slt $18 $17 $14
```

Testbench 仿真测试代码见:

```
            mem_addr=16;//#监视受调试变量
            #100;
            resetn=0;
            #50;
            resetn=1;
            mem_addr=5;
            uut.rf_module.rf[14]=32'd0;//程序可执行工作区（集中
在中间）的初始化
            uut.rf_module.rf[17]=32'd0;
            uut.rf_module.rf[18]=32'd0;
            #50;
            uut.pc=32'h00000058;//输入调试指针，开始执行汇编程序
            #10;
            rf_addr=13;
            //uut.rf_module.rf[16]=32'hFFFFFF97;
            uut.rf_module.rf[16]=~uut.rf_module.rf[13]+1;
            //求解二进制数的最高非零和最低非零位，并基于非除法运
算判断这个数是否为 2 的整数幂
            #10;
            rf_addr=16;
            #250;
            mem_addr=15;
            #250;
            mem_addr=16;
        rf_addr=18;
```

此外，由于指令的个数超过了 32 个，需要拓展 32 位指令的内嵌存储块大小

细节代码修改为:

```
    assign inst_addr = pc;  // 指令地址：指令长度 32 位
        inst_rom inst_rom_module(          // 指令存储器
            .addr      (inst_addr[7:2]),   // I, 6,指令地址
            .inst      (inst          )    // O, 32,指令
    );
```

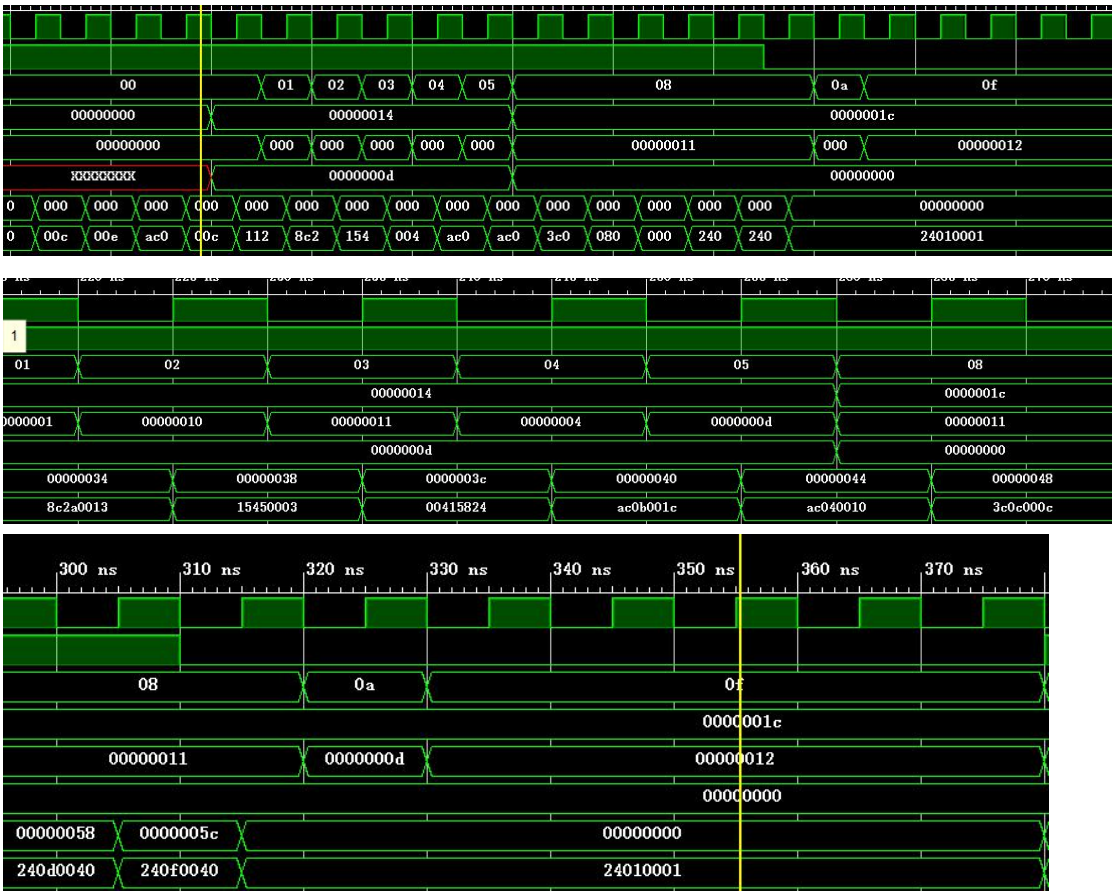
Single_cycle_cpu.v

```
    module inst_rom(
        input      [5 :0] addr, // 指令地址
        output reg [31:0] inst       // 指令
    );
```

Inst_rom.v

四、 实验结果

1. 基于源代码汇编指令的仿真设计波形图



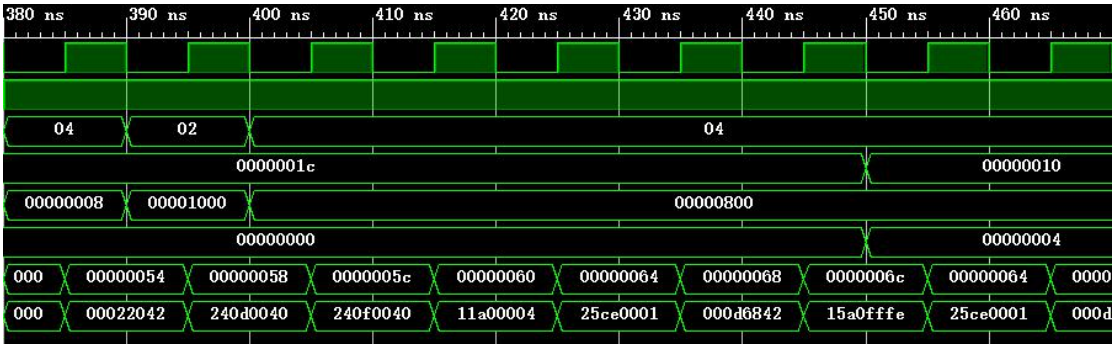
根据汇编指令寄存器 1，2,3,4,5 中依次为 01h,10h,11h,04h,0dh,验证无误。

可读数据寄存器中的 14H 内容为 0dh,而两次前后打印访问可读数据寄存器 1CH 也可以从第二幅图中观察到变化 0011h→0000h

此外第一幅图最先的空阻态表示可读指令寄存器无法被除指令操作外

的手段赋值初始化，根据代码其类型为 wire 无法像 reg 组合或时序赋值

2. 对 J 型指令和移位指令的循环验证与函数级汇编程序的准备



可见 J 型指令成功跳转到了第 22 条指令 54H ： srl $4,$2,#1，相较于 1 中的寄存器 4 号只移 1 位数值在原来基础上乘 2 为 08h

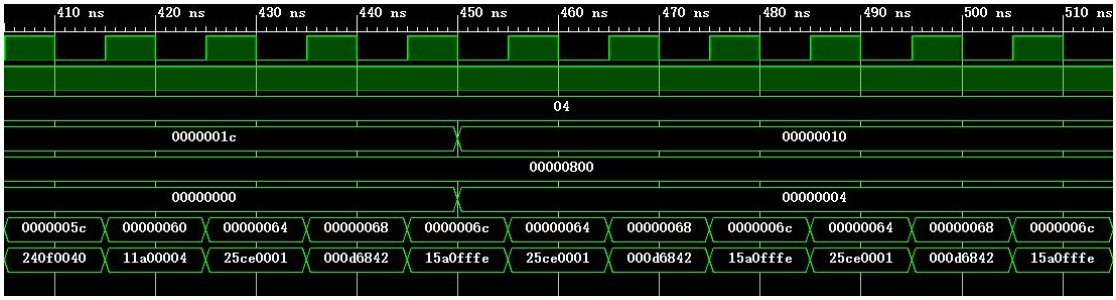
同时，调整调试指针指向第 21 号指令 50H ： sll $2,$1,#12，在执行周期后访问指令关联 02,04 寄存器得到访问内容的比例变换，验证了循环的发生，且比例关系符合移位期望

以上说明了汇编指令的功能组合符合循环体和移位运算求解函数的汇编逻辑要求，可以进一步编写汇编程序实现对 2 的整数幂判断的 Bool 函数

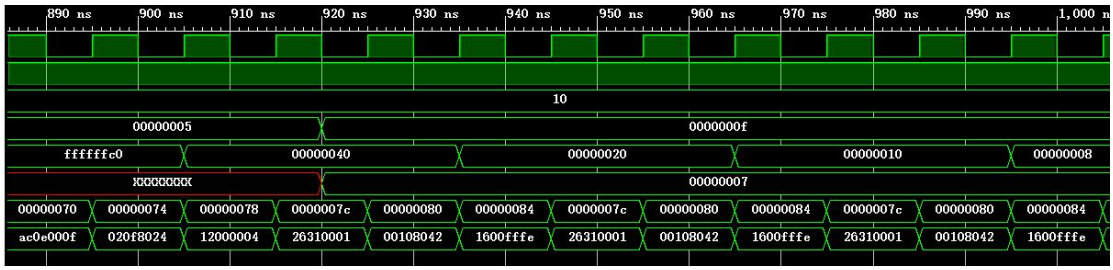
3. 汇编函数实现验证

① 输入执行数据为 0040H 为 2 的整数幂

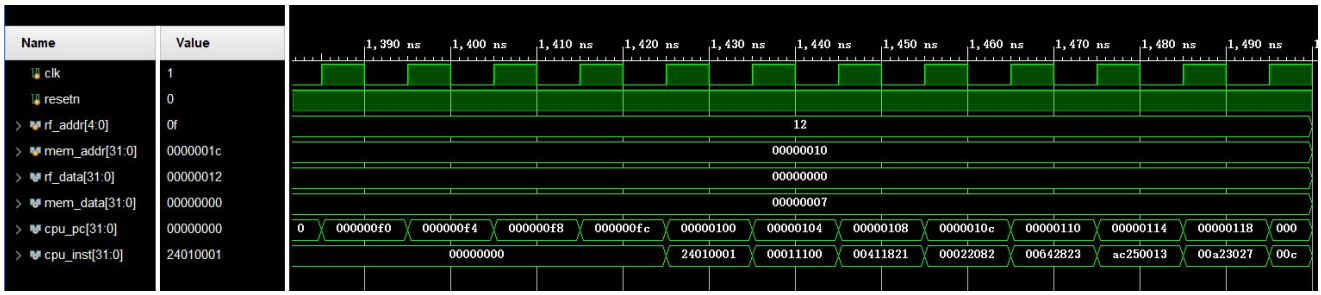
循环 1，观察下方指针的重复



循环 2



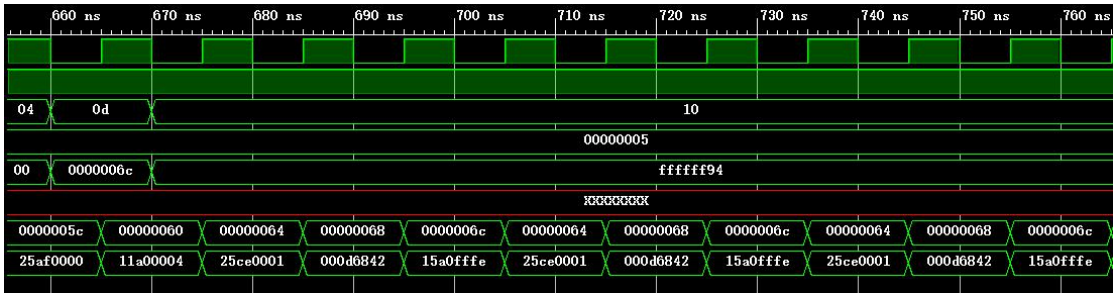
程序结果：



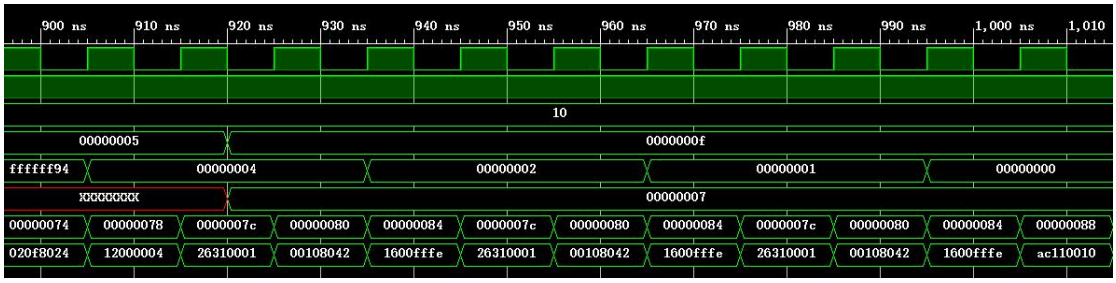
18 号（12H）寄存器存储逻辑判断结果，低电平有效说明该数为 2 的整数次幂
从 15 号（0fH）打印可读数据存储器中存储最高非零位结果为 0007h
从 16 号（10H）打印可读数据存储器中存储最低非零位结果为 0007h
最高位和最低位重合，符合判断条件

② 输入执行数据为 0069H 不是 2 的整数幂

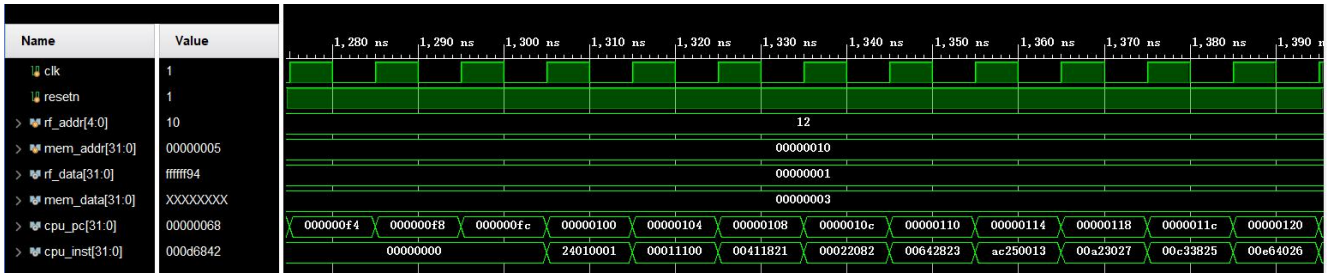
循环 1，观察下方指针的重复



循环 2



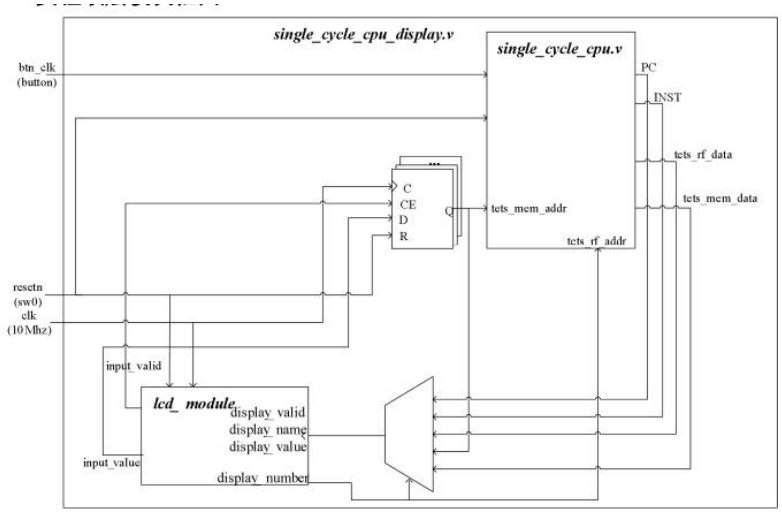
程序结果:



同理，
18 号（12H）寄存器存储逻辑判断结果，低电平有效而该位为高电平说明这个数不是 2 的整数次幂
从 15 号（0fH）打印可读数据存储器中存储最高非零位结果为 0007h
从 16 号（10H）打印可读数据存储器中存储最低非零位结果为 0003h
最低非零位位置小于最高非零位位置，符合对 2 的整数次幂判否条件
综上基于 C++的 Bool 函数判断二进制数是否为 2 的整数次幂的汇编函数程序运

行成功

4. 顶层模块综合设计图



五、 实验总结

1. 本次实验使用 verilog 语言进行顶层模块设计，涉及对之前实验中的存储器单元，算术逻辑单元和寄存器模块，加深了对 RTL 层级语言的逻辑实现的理解，其中的模块嵌套如加法模块调用是前面实验掌握的逻辑层次方法。整体上达到了提高学习使用 verilog 设计数字系统的综合推理思维和验证型完善要求
2. 仿真设计验证源码汇编正确性基础上，对同一地址同一运算多次验证检查代码运行结果。同时设计跳转和移位指令，通过比例变化和指针重复，验证了循环和运算体的完备可行性，为汇编函数程序准备
3. 通过手动编写汇编函数程序，直观感受到了人工编写机器码的耗时费力，更加深了对软件代码语言研究优良编译器的科学理解。函数实现对 2 的整数幂的快速判断，利用的还是移位运算的特殊性，但同理可以扩展为对应进制的整数幂运算结果研判，这在质数分解和对数线性运算中都有重要作用。本实验实现的函数较为简单，期望达到举一反三的效果，为研究高效的计算机综合数字系统的设计功能和执行逻辑打下基础。
4. 由于实验基于参考代码,其具体功能解析见源码注释。另附完整修改后的实验代码