单周期CPU设计

一、CPU设计方案综述

（一）总体设计概述

本CPU位Logisim实现的32位单周期MIPS-CPU，支持指令集包含`{addu，subu，ori，lw，sw，beq，jal，jr，lui，nop，j，lh，lb，sh，sb}`。为实现相关功能，CPU主要包含PC、NPC、IM、GRF、EXT、ALU、DM、CTRL模块。遵循形式化建模综合方法完成设计与实现。

（二）数据通路模块定义

1.PC

程序计数器。就是一个32位寄存器，接收NPC的信号，指向当前执行指令的地址。

2.NPC

计算下一条指令的地址

信号名	方向	描述
[31:0] PC	Input	32位输入，当前PC值
[15:0] IMM	Input	26位立即数（imm16和imm26的综合考量）
[31:0] RA	Input	jr指令中，所选寄存器32位值输入，目标地址值
[1:0] Op	Input	选择不同NPC输出： `2'b00`：输出顺序地址PC+4 `2'b01`：输出指令beq所得地址 `2'b10`：输出指令jal所得地址 `2'b11`：输出指令jr所得地址
Zero	Input	beq指令中，rs和rt的比较结果： `0`：不相等 `1`：相等
[31:0] PC4	Output	jal指令中，将PC+4存入`$ra`内所需输出
[31:0] NPC	Output	下一条指令目标地址

3.IM

指令内存。该CPU将指令和数据分开存储。就是一个ROM。设置最大指令条数数为 2^{16}，因而输入端A取PC输出信号2-17位。

起始地址：**0x0000_0000**

4.GRF

通用寄存器堆

信号名	方向	描述
[4:0] A1	Input	输入rs段要读取的寄存器编号
[4:0] A2	Input	输入rt段要读取的寄存器编号
[4:0] A3	Input	输入rd段要写入的寄存器编号
[31:0] WD	Input	需要写回的值
WE	Input	写入使能端
Clk	Input	时钟信号端
Rst	Input	寄存器复位端
[31:0] RD1	Output	输出A1所选对应寄存器的值
[31:0] RD2	Output	输出A2所选对应寄存器的值

5.EXT

实现不同位扩展功能。

信号名	方向	描述
[15:0] I	Input	输入16位立即数
[1:0] EXTOp	Input	选择扩展方式： `2'b00`：无符号扩展 `2'b01`：有符号扩展 `2'b10`：加载到高位
[31:0] O	Output	输出扩展后的32位数

6.ALU

算数逻辑单元。

信号名	方向	描述
[31:0] A	Input	输入数1
[31:0] B	Input	输入数2
[1:0] ALUOp	Input	运算选择器： `2'b00`：addu加法运算 A + B `2'b01`：subu减法运算 A - B `2'b10`：ori或运算 A | B
[31:0] Y	Output	输出运算结果
Zero	Output	A和B比较结果： 0：不相等 1：相等

7.DM

数据内存。双端模式RAM实现（RAM 的 **Data Interface** 属性设置为 **Separate load and store ports**），容量为$2^{16} \times 32bit$。

起始地址：**0x0000_0000**

信号名	方向	描述
[31:0] A	Input	
[31:0] WD	Input	
DMWr	Input	
Clk	Input	
Rst	Input	
[1:0] SSel	Input	
[1:0] LSel	Input	
[31:0] RD	Output	

（三）数据通路连接总表

部件	PC	NPC				IM	RF				EXT	ALU		DM	
输入信号	DI	PC	IMM	RA	Zero	A	A1	A2	A3	WD		A	B	A	WD
addu	NPC.NPC	PC.DO				PC.DO	IM.D[25:21]	IM.D[20:16]	IM.D[15:11]	ALU.C		RF.RD1	RF.RD2		
subu	NPC.NPC	PC.DO				PC.DO	IM.D[25:21]	IM.D[20:16]	IM.D[15:11]	ALU.C		RF.RD1	RF.RD2		
ori	NPC.NPC	PC.DO				PC.DO	IM.D[25:21]		IM.D[20:16]	ALU.C		RF.RD1	RF.RD2		
lw	NPC.NPC	PC.DO				PC.DO	IM.D[25:21]		IM.D[20:16]	DM.RD	IM.D[15:00]	RF.RD1	EXT.O	ALU.C	
sw	NPC.NPC	PC.DO				PC.DO	IM.D[25:21]	IM.D[20:16]			IM.D[15:00]	RF.RD1	EXT.O	ALU.C	RF.RD2
beq	NPC.NPC	PC.DO	IM.D[15:00]		ALU.Zero	PC.DO	IM.D[25:21]	IM.D[20:16]				RF.RD1	RF.RD2		
jal	NPC.NPC	PC.DO	IM.D[25:00]			PC.DO			0x1F	NPC.PC4					
jr	NPC.NPC	PC.DO		RF.RD1		PC.DO	IM.D[25:21]								
lui	NPC.NPC	PC.DO				PC.DO			IM.D[20:16]	EXT.O	IM.D[15:00]				
nop	NPC.NPC	PC.DO				PC.DO									
j	NPC.NPC	PC.DO	IM.D[25:00]			PC.DO									
add															
addi															
sll															
sllv															
lh	NPC.NPC	PC.DO				PC.DO	IM.D[25:21]		IM.D[20:16]	DM.RD	IM.D[15:00]	RF.RD1	EXT.O	ALU.C	
lb	NPC.NPC	PC.DO				PC.DO	IM.D[25:21]		IM.D[20:16]	DM.RD	IM.D[15:00]	RF.RD1	EXT.O	ALU.C	
sh	NPC.NPC	PC.DO				PC.DO	IM.D[25:21]	IM.D[20:16]			IM.D[15:00]	RF.RD1	EXT.O	ALU.C	RF.RD2
sb	NPC.NPC	PC.DO				PC.DO	IM.D[25:21]	IM.D[20:16]			IM.D[15:00]	RF.RD1	EXT.O	ALU.C	RF.RD2
COMPLETE	NPC.NPC	PC.DO	IM.D[25:00]	RF.RD1	ALU.Zero	PC.DO	IM.D[25:21]	IM.D[20:16]	IM.D[20:16] IM.D[15:11] 0x1F	NPC.PC4 DM.RD ALU.C EXT.O	IM.D[15:00]	RF.RD1	RF.RD2 EXT.O	ALU.C	RF.RD2

（四）控制器CTRL模块定义

1.控制信号定义

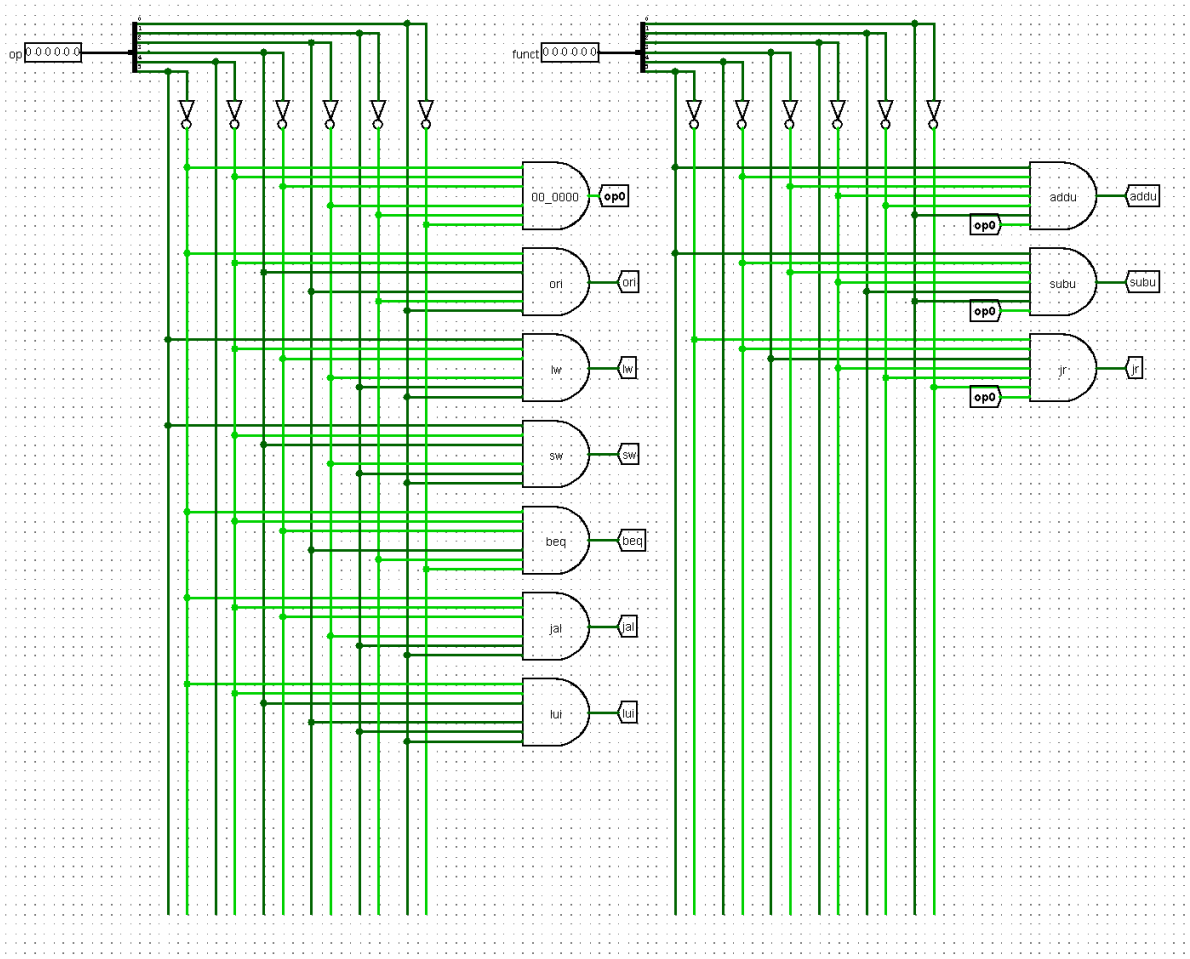
控制信号	描述
[1:0] NPCOp	执行跳转指令时控制选择NPC输出值： （即NPC的Op信号） `2'b00`：输出顺序地址PC+4 `2'b01`：输出指令beq所得地址 `2'b10`：输出指令jal所得地址 `2'b11`：输出指令jr所得地址
[1:0] M1Sel	选择GRF模块A3（写入寄存器）的输入信号： `2'b00`：接入IM.D[20:16]信号（也是A2处信号） `2'b01`：接入IM.D[15:11]信号 `2'b10`：接入常量0x1F，即`$ra` `2'b11`：空置
[1:0] M2Sel	选择GRF模块WD（写入内容共）的输入信号： `2'b00`：接入NPC.PC4，jal指令存入PC+4的地址 `2'b01`：接入DM.RD，内存数据的回写 `2'b10`：接入ALU.C，计算数据回写 `2'b11`：接入EXT.O，位扩展数据回写（lui）
RFWr	GRF的写入使能端
[1:0] EXTOp	位扩展方式： `2'b00`：零扩展 `2'b01`：符号扩展 `2'b10`：加载至高16位 `2'b11`：
M3Sel	选择ALU模块B的输入信号： `0`：接入RF.RD2，接收寄存器取出内容 `1`：接入EXT.O，接收位扩展后的16位立即数内容
[1:0] ALUOp	ALU的运算模式： `2'b00`：加法运算 `2'b01`：减法运算 `2'b10`：或运算 `2'b11`：空置
DMWr	DM的写入使能端

2.控制器逻辑真值表

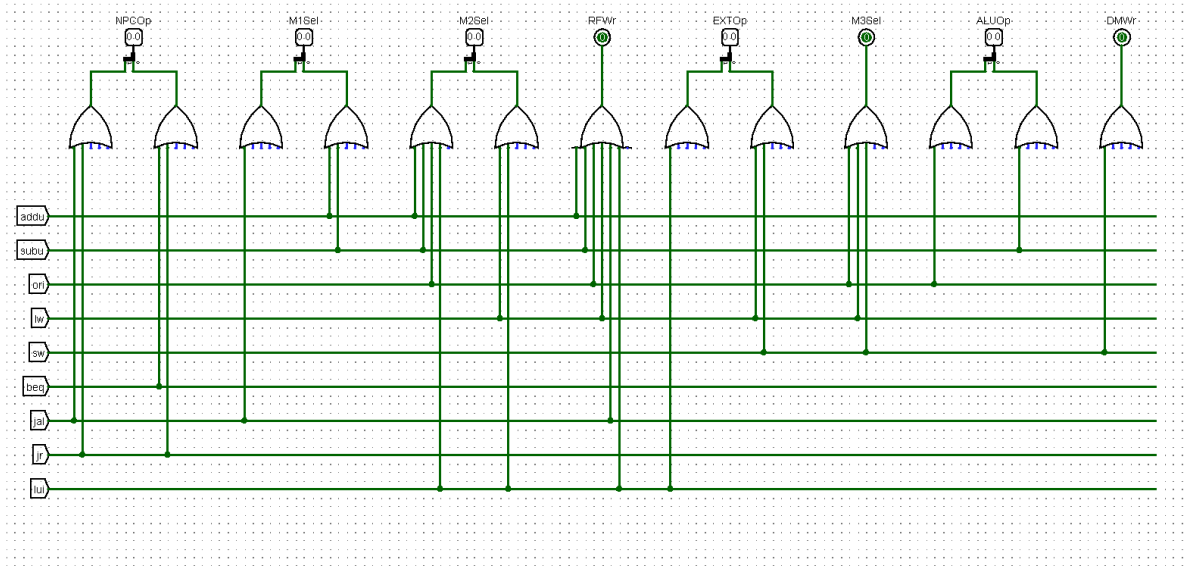
func	10_0001	10_0011						00_1000						
op	00_0000	00_0000	00_1101	10_0011	10_1011	00_0100	00_0011	00_0000	00_1111	00_0010	10_0001	10_0000	10_1001	10_1000
	addu	subu	ori	lw	sw	beq	jal	jr	lui	j	lh	lb	sh	sb
NPCOp[1]	0	0	0	0	0	0	1	1	0	1	0	0	0	0
NPCOp[0]	0	0	0	0	0	1	0	1	0	0	0	0	0	0
M1Sel[1]	0	0	0	0	x	x	1	x	0	x	0	0	x	x
M1Sel[0]	1	1	0	0	x	x	0	x	0	x	0	0	x	x
M2Sel[1]	1	1	1	0	x	x	0	x	1	x	0	0	x	x
M2Sel[0]	0	0	0	1	x	x	0	x	1	x	1	1	x	x
RFWr	1	1	1	1	0	0	1	0	1	0	1	1	0	0
EXTOp[1]	x	x	0	0	0	x	x	x	1	x	0	0	0	0
EXTOp[0]	x	x	0	1	1	x	x	x	0	x	1	1	1	1
M3Sel	0	0	1	1	1	0	x	x	x	x	1	1	1	1
ALUOp[1]	0	0	1	0	0	x	x	x	x	x	0	0	0	0
ALUOp[0]	0	1	0	0	0	x	x	x	x	x	0	0	0	0
DMWr	0	0	0	0	1	0	0	0	0	0	0	0	1	1
SSel[1]	0	0	0	0	0	0	0	0	0	0	0	0	0	1
SSel[0]	0	0	0	0	0	0	0	0	0	0	0	0	1	0
LSel[1]	0	0	0	0	0	0	0	0	0	0	0	1	0	0
LSel[0]	0	0	0	0	0	0	0	0	0	0	1	0	0	0

3.控制器实现结构

与逻辑产生指令信号



或逻辑产生控制信号



二、测试方案

（一）手动测试

1. 测试到lw指令

```
v2.0 raw
3404007b
348501c8
3c06007b
3c07ffff
34e7ffff
00868021
00000000
34080000
ad100000
ad040004
ad050008
ad06000c
8d09000c
8d0a0008
8d0b0004
8d0c0000
```

2.测试到jr指令

```
ori $t0, $0, 123
ori $t1, $0, 125
subu $t2, $t1, $t0

ori $s0, $0, 0
ori $t3, $0, 2
loop:
addu $s0, $s0, $t3
beq $s0, $t2, loop

ori $v0, $v0, 0
lui $a0, 1
jal no_fun
sw $a0, 0($v0)


no_fun:
sw $a0, 4($v0)
jr $ra
```

```
v2.0 raw
3408007b
3409007d
01285023
34100000
340b0002
020b8021
120afffe
34420000
3c040001
0c00000b  //此处原为0c000c0b，修正PC指向后为此
```

```
ac440000
ac440004
03e00008
```

3.测试到sb（自动化生成）

```
ori $4,$0,104
ori $2,$0,23
ori $7,$0,22
ori $24,$0,121
ori $20,$0,44
ori $21,$0,83
ori $27,$0,98
ori $23,$0,85
ori $12,$0,62
ori $15,$0,6
sb $20,24($2)
sw $7,-88($4)
sb $27,8($7)
sb $20,-63($4)
lw $9,18($7)
beq $3,$20,branch2
jal branch3
branch3:
lui $5,32815
sh $2,31($24)
beq $4,$16,branch4
lh $17,82($4)
lb $14,-30($20)
beq $0,$12,branch5
sh $0,141($23)
lui $5,30807
jal branch6
addu $10,$21,$8
lb $8,-101($24)
addu $8,$2,$12
addu $9,$24,$15
jal branch8
addu $22,$18,$25
beq $12,$27,branch9
beq $15,$19,branch10
lw $8,22($7)
beq $22,$25,branch11
nop
j branch12
sb $2,-33($20)
branch5:
sw $26,258($27)
branch9:
j branch13
branch11:
subu $6,$11,$27
addu $14,$6,$19
sw $10,31($23)
branch6:
branch13:
```

```
lh $10,44($27)
branch14:
addu $25,$21,$14
sh $11,102($15)
branch1:
lui $8,40234
nop
jal branch16
sw $3,342($12)
branch2:
branch12:
sh $25,93($24)
branch8:
jal branch18
sb $0,16($27)
branch10:
branch17:
j branch19
jal branch21
branch18:
j branch22
branch19:
addu $18,$25,$9
branch15:
branch22:
sh $19,-13($21)
sh $23,174($12)
sw $10,237($21)
sw $25,218($12)
lh $5,-27($24)
nop
branch7:
branch16:
nop
j branch23
sh $3,124($4)
branch20:
lb $9,48($12)
branch4:
lh $26,-1($2)
beq $19,$0,branch24
branch23:
branch24:
branch25:
subu $17,$20,$25
branch21:
lb $14,-21($7)
```

（二）自动化测试

呜呜呜没时间写啊，各种命令行处理和代码调试耗时很长呢。。

1.自动化生成特定指令的MIPS（别人写的）

```
#include<fstream>
#include<vector>
#include<algorithm>
#include<cstdlib>
#include<ctime>
using namespace std;
ofstream cout("test_stg_mips.txt");
struct op
{
    int type,r1,r2,r3;
    op(){}
    op(int a1,int a2,int a3,int a4){
        type=a1,r1=a2,r2=a3,r3=a4;
    }
};
int size_im,size_dm,size_op,small_reg, ra;
op a[10010];
vector<int> branch[10010];
int is_small[40],small[40],val[40],cnt;
int r(int a,int b,int except1)//由于1号寄存器使用时可能出现错误，所以生成指令不包含1号寄存器
{
    int t0=rand()*2+rand()%2;
    int res=t0%(b-a+1)+a;
    return ((res==1)&&except1==1)?0:res;
}
int t[10010];
void get_small()//随机选取若干个寄存器用于lw和sw，他们的值小于dm大小且不变
{
    int i;
    for(i=1;i<=26;i++) t[i]=i;
    for(i=1;i<=26;i++) swap(t[i],t[rand()%30+1]);
    for(i=1;i<=small_reg;i++) small[i]=t[i]+1,is_small[t[i]+1]=1;
}
int nosmall()//随机一个不用于lw和sw的寄存器
{
    int u=r(0,27,1);
    while(is_small[u]==1){u=r(0,27,1);}
    return u;
}
void print(int x)
{
    if(a[x].type==0) cout<<"nop"<<endl;
    else if(a[x].type==1) cout<<"addu $"<<a[x].r1<<",$"<<a[x].r2<<",$"
<<a[x].r3<<endl;
    else if(a[x].type==2) cout<<"subu $"<<a[x].r1<<",$"<<a[x].r2<<",$"
<<a[x].r3<<endl;
    else if(a[x].type==3) cout<<"lui $"<<a[x].r1<<","<<a[x].r3<<endl;
    else if(a[x].type==4) cout<<"ori $"<<a[x].r1<<",$"<<a[x].r2<<","
<<a[x].r3<<endl;
    else if(a[x].type==5) cout<<"lw $"<<a[x].r1<<","<<a[x].r3<<"($"
<<a[x].r2<<")"<<endl;
    else if(a[x].type==6) cout<<"sw $"<<a[x].r1<<","<<a[x].r3<<"($"
<<a[x].r2<<")"<<endl;
```

```
    else if(a[x].type==7) cout<<"beq $"<<a[x].r1<<",$"<<a[x].r2<<",branch"
<<a[x].r3<<endl;
    //else if(a[x].type==8) cout<<"sll $"<<a[x].r1<<",$"<<a[x].r2<<","
<<a[x].r3<<endl;
    //else if(a[x].type==9) cout<<"slt $"<<a[x].r1<<",$"<<a[x].r2<<",$"
<<a[x].r3<<endl;
    //else if(a[x].type==10) cout<<"addiu $"<<a[x].r1<<",$"<<a[x].r2<<","
<<a[x].r3<<endl;
    else if(a[x].type==11) cout<<"lb $"<<a[x].r1<<","<<a[x].r3<<"($"
<<a[x].r2<<")"<<endl;
    else if(a[x].type==12) cout<<"sb $"<<a[x].r1<<","<<a[x].r3<<"($"
<<a[x].r2<<")"<<endl;
    else if(a[x].type==13) cout<<"lh $"<<a[x].r1<<","<<a[x].r3<<"($"
<<a[x].r2<<")"<<endl;
    else if(a[x].type==14) cout<<"sh $"<<a[x].r1<<","<<a[x].r3<<"($"
<<a[x].r2<<")"<<endl;

    //else if(a[x].type==15) cout<<"bne $"<<a[x].r1<<",$"<<a[x].r2<<",branch"
<<a[x].r3<<endl;
    else if(a[x].type==16) cout<<"j "<<"branch"<<a[x].r3<<endl;
    else if(a[x].type==17) cout<<"jal "<<"branch"<<a[x].r3<<endl;
    else if(a[x].type==18) cout<<"jr $ra"<<endl;
    //else if(a[x].type==19) cout<<"lhu $"<<a[x].r1<<","<<a[x].r3<<"($"
<<a[x].r2<<")"<<endl;
    //else if(a[x].type==20) cout<<"lbu $"<<a[x].r1<<","<<a[x].r3<<"($"
<<a[x].r2<<")"<<endl;

}
int main()
{
    // 0nop 1addu 2subu 3lui 4ori 5lw 6sw 7beq 8sll 9slt /10addiu 11lb 12sb 13lh
14sh /15bne
    // 16j 17jal 18jr 19lhu 20lbu

    int i,j;
    srand(time(0));
    size_im=100;//生成的指令的大小
    size_dm=128;//DM的大小
    size_op=18;//支持的指令集的大小
    small_reg=10;//最前面small_reg个指令都是ori，先对寄存器进行赋值
    get_small();
    for(i=1;i<=small_reg;i++)
        a[i]=(op(4,small[i],0,val[small[i]]=r(0,size_dm-1,0)));
    for(i=small_reg+1;i<=size_im;i++)
    {
        int op0=r(0,size_op-1,0),r1,r2,r3;
        if(op0==0) a[i]=(op(0,0,0,0));
        else if(op0==1||op0==2||op0==9){//addu与subu
            a[i]=op(op0,nosmall(),r(0,27,1),r(0,27,1));
        }
        else if(op0==3||op0==4||op0==10){//lui与ori
            a[i]=(op(op0,nosmall(),r(0,27,1),r(0,65535,0)));
        }
        else if(op0==8){
            a[i]=(op(op0,nosmall(), r(0, 27, 1), r(0, 31, 0)));
        }
        else if(op0==5){//lw
            r1=r(0,size_dm-1,0)*4;
```

```
            r2=r(1,small_reg,0);
            a[i]=(op(op0,nosmall(),small[r2],r1-val[small[r2]]));
        }
        else if(op0==13 || op0 == 19){// lh lhu
            r1=r(0, size_dm-1, 0)*2;
            r2=r(1, small_reg, 0);
            a[i]=(op(op0,nosmall(),small[r2],r1-val[small[r2]]));
        }
        else if(op0==6){//sw
            r1=r(0,size_dm-1,0)*4;
            r2=r(1,small_reg,0);
            a[i]=(op(op0,r(0,27,1),small[r2],r1-val[small[r2]]));
        }
        else if(op0==14){
            r1=r(0,size_dm-1,0)*2;
            r2=r(1,small_reg,0);
            a[i]=(op(op0,r(0,27,1),small[r2],r1-val[small[r2]]));
        }
        else if(op0==11 || op0 == 20){//lb lbu
            r1=r(0, size_dm-1, 0);
            r2=r(1, small_reg, 0);
            a[i]=(op(op0, nosmall(), small[r2], r1-val[small[r2]]));
        }
        else if(op0==12){//sb
            r1=r(0, size_dm-1, 0);
            r2=r(1, small_reg, 0);
            a[i]=(op(op0, r(0, 27, 1), small[r2], r1-val[small[r2]]));
        }
        else if(op0==7||op0==15|| op0 == 16 || op0 == 17){//beq j jal由于往上跳转可
能出现死循环，所以只生成往下跳转
            r3=r(i,size_im,0);
            a[i]=(op(op0,r(0,27,1),r(0,27,1),++cnt));
            branch[r3].push_back(cnt);
            if(op0 == 17) ra = r3;

        }
        else if(op0 == 18 && ra)
        {
            a[i]=op(op0,nosmall(),r(0,27,1),r(0,27,1));
            ra = 0;
        }
    }
    for(i=1;i<=size_im;i++)
    {
        print(i);
        for(j=0;j<branch[i].size();j++)
            cout<<"branch"<<branch[i][j]<<":"<<endl;
    }
}
```

2.生成PC偏移处理后的ROM导入文件

就是在导入文件中添加相应个数的 `00000000`

```
#include<stdio.h>
#include<string.h>
```

```c
int main(){
    FILE *fp;
    char ch;
    fp = fopen("test_stg.txt","r");

    freopen("test_stg_final.txt","w",stdout);

    printf("v2.0 raw\n");
    for(int i=0;i<0x00003000/4;++i)
        printf("00000000\n");
    while( (ch=fgetc(fp))!=EOF )
        putchar(ch);

    return 0;
}
```

（三）一些注意事项

1. `ori` 等指令中的立即数为16位，超出该范围将启动溢出处理，因而基本测试时，立即数应在 `0x0000_0000` 到 `0x0000_ffff` 之间（即0~65535）

预期	实际

`ori $a0, $0, 0x0001ffff`	`lui $at, 0x00000001`
	`ori $at, $at, 0x0000ffff`
	`or $a0, $0, $at`

`ori $a0, $0, -1`（即为 `0xffffffff`）	`lui $at, 0xffffffff`
	`ori $at, $at, 0x0000ffff`
	`or $a0, $0, $at`

计算类指令功能测试

- 寄存器数据方面，可以考虑以下情况：
  - 00 及附近的数：-2, -1, 0, 1, 2−2,−1,0,1,2
  - 3232 位数边界附近的数： -2147483648, -2147483647, 2147483646, 2147483647−2147483648,−2147483647,2147483646,2147483647
  - 3232 位数范围内的一些随机数：-1000786109, 1919156834, ... −1000786109,1919156834,...
- 无符号立即数方面，可以考虑以下情况：
  - 00 及附近的数：0, 1, 2, 30,1,2,3
  - 1616 位无符号数边界附近的数：65533, 65534, 6553565533,65534,65535
  - 1616 位无符号数范围内的一些随机数：25779, 42528, ...25779,42528,...
- 符号立即数 (P3 不涉及) 方面，可以考虑以下情况：
  - 00 及附近的数：-2, -1, 0, 1, 2−2,−1,0,1,2
  - 1616 位符号数边界附近的数：-32768, -32767, 32766, 32767−32768,−32767,32766,32767
  - 1616 位符号数范围内的一些随机数：-5329, 25299, ...−5329,25299,...
- 特别的，可注意测试目标寄存器是 $0$0 的情况。

存取类指令功能测试

- offset 方面，可以考虑以下情况：
  - offset 是正数
  - offset 是零
  - offset 是负数
- $base 寄存器方面，可以考虑以下情况：
  - $base 寄存器中的值是正数
  - $base 寄存器中的值是零
  - $base 寄存器中的值是负数
- 特别的，对于 sw 指令，建议存入的 word 中，每个 byte 都不是零。
- 特别的，对于 lw 指令，可注意测试目标寄存器是 $0 的情况。

跳转类指令功能测试

- 对于非比较相关的部分，可以考虑以下情况：
  - 跳转，且目标在此跳转指令之前
  - 跳转，且目标是此跳转指令
  - 跳转，且目标在此跳转指令之后
  - 不跳转，且目标在此跳转指令之前
  - 不跳转，且目标是此跳转指令
  - 不跳转，且目标在此跳转指令之后
- 对于比较相关的部分，本质上依旧是构造寄存器数据，处理类似 “计算类指令功能测试”。

三、思考题

1.现在我们的模块中 IM 使用 ROM， DM 使用 RAM， GRF 使用 Register，这种做法合理吗?

既然指令和数据分开存放，那这种设计挺合理的。IM指令不可修改，DM数据可存取，GRF本就是寄存器堆。

2.事实上，实现 nop 空指令，我们并不需要将它加入控制信号真值表，为什么？请给出你的理由。

`nop`指令不进行任何操作，不改变GRF和DM内容。不加入控制信号真值表，`nop`指令时`RFWr`和`DMWr`均为0，便万事大吉了。

3.上文提到，MARS 不能导出 PC 与 DM 起始地址均为 0 的机器码。实际上，可以通过为 DM 增添片选信号，来避免手工修改的麻烦，请查阅相关资料进行了解，并阐释为了解决这个问题，你最终采用的方法。

- MARS上PC初始为`0x00003000`只会在使用指令地址时产生问题，例如`jal` `j`指令。分析指令构成，修改导出机器码相应位置的值。自动化操作未来可期（没时间写了呜呜呜）
- 或者初始化PC为`0x00003000`（自建一个可初始化的寄存器），且导入指令时从ROM的`0x00003000`处导入

4.除了编写程序进行测试外，还有一种验证 CPU 设计正确性的办法——形式验证。形式验证的含义是根据某个或某些形式规范或属性，使用数学的方法证明其正确性或非正确性。请搜索“形式验证（Formal Verification)”了解相关内容后，简要阐述相比于测试，形式验证的优劣之处。

模拟仿真	形式验证
需要测试向量	不需测试向量
完整模型，部分验证	部分模型，完整验证
输入驱动，比较输出	输出驱动，预测行为
不完整	完整，全输入空间
验证输入空间的点，一次检查一个输出点	验证属性，一次检查一组同属性输出点
难点：输入激励是否足够	难点：属性是否完备