转移指令的实现

一、延迟槽的定义及说明

在之前设计流水线的时候，存在三种相关，分别为数据相关、结构相关、控制相关。其中控制相关是指流水线中的转移指令或者其他需要改写 PC 的指令造成的相关。这些指令改写了 PC 值，所以导致后面进入流水线的几条指令无效。比如：如果转移指令在流水线的执行阶段进行条件转移条件判断，在发生转移时，会导致当前处于取指、译码阶段的指令无效，需要重新取指。

如图：

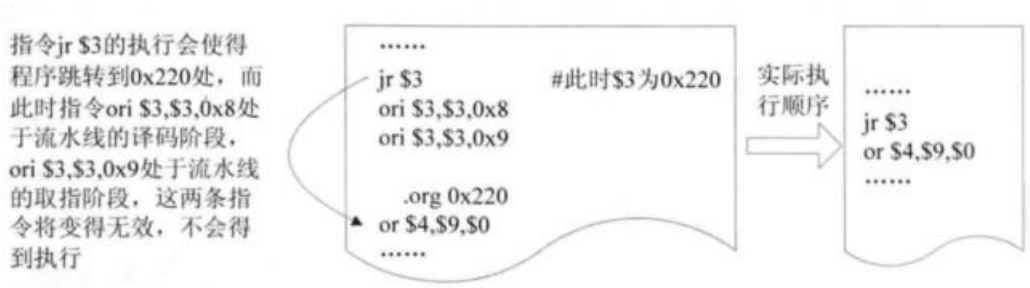


图 8-1 转移指令会使得其后面已经进入流水线的几条指令无效

尽管转移指令后面的指令确实不需要执行，但是这样却白白浪费了两个时钟周期，当转移指令很多的时候，会大大影响流水线性能。

因此我们引入延迟槽概念，规定转移指令后面的指令位置为“延迟槽”，延迟槽中的指令被称为“延迟槽指令”。该指令总是被执行，与是否被转移没有关系。引入延迟槽后的指令执行顺序如下图：

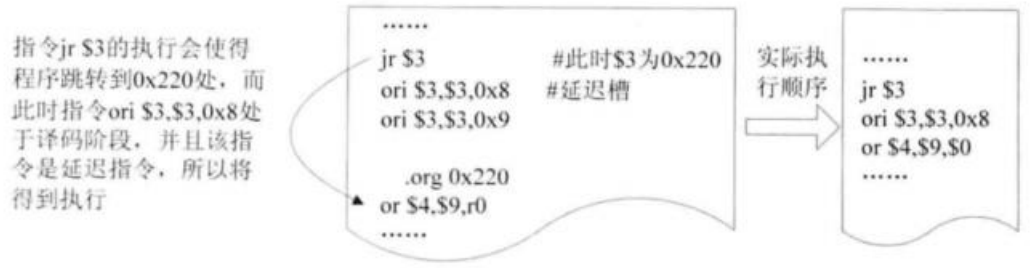


图 8-2 引入延迟槽以减少转移带来的损失

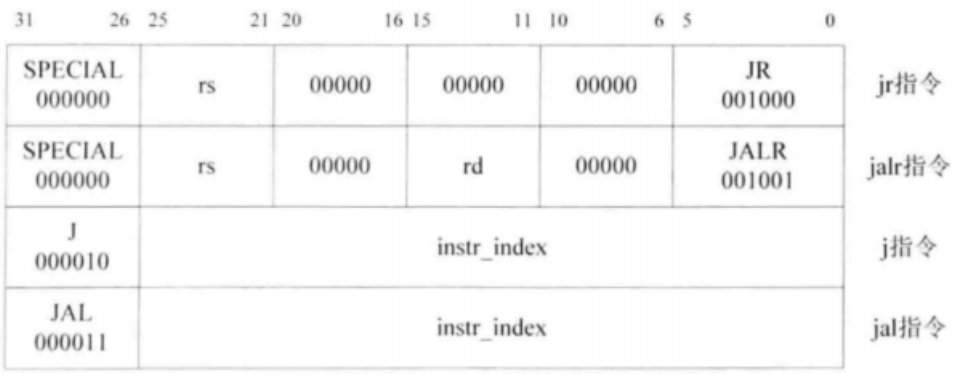
但是我们又发现一个问题，因为是在执行阶段开始判断转移，因此前面有两个阶段的指令已经被读取，使用延迟槽仍然会使得已经进入取指阶段的指令无效，也就是仍然浪费一个时钟周期。因此我们计划在译码阶段进行转移判断。

那为什么不能用两个延迟槽呢？还有延迟槽的作用到底是什么呢？（我的疑问）

二、指令说明

2.1 跳转指令

跳转指令格式如下：



31 26	25 21	20 16	15 11	10 6	5 0	
SPECIAL 000000	rs	00000	00000	00000	JR 001000	jr指令
SPECIAL 000000	rs	00000	rd	00000	JALR 001001	jalr指令
J 000010	instr_index					j指令
JAL 000011	instr_index					jal指令

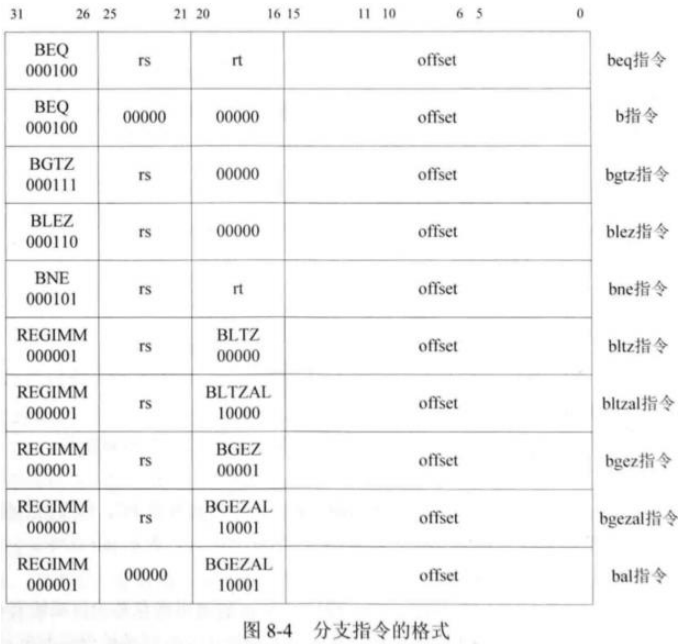
图 8-3 跳转指令的格式

其中 jalr 指令表示格式：jalr rs 或者 jalr rd,rs 表示将地址为 rs 的通用寄存器的值赋给寄存器 PC，作为新的指令地址，同时跳转到指令后面第二条指令的地址作为返回地址保存到地址为 rd 的通用寄存器，如果没有在指令中指明 rd，那么默认将返回地址保存到寄存器$31.

J 指令表示格式：j target，作用为：pc->（pc+4）[31, 28] || target || ‘00’，新指令地址的低 28 位是 instr_index 左移两位的值，新指令地址的高 4 位是跳转指令后面延迟槽指令的地址高 4 位。

2.2 分支指令

分支指令格式如下图：



31 26	25 21	20 16	15 11	10 6	5 0	
BEQ 000100	rs	rt	offset			beq指令
BEQ 000100	00000	00000	offset			b指令
BGTZ 000111	rs	00000	offset			bgtz指令
BLEZ 000110	rs	00000	offset			blez指令
BNE 000101	rs	rt	offset			bne指令
REGIMM 000001	rs	BLTZ 00000	offset			bltz指令
REGIMM 000001	rs	BLTZAL 10000	offset			bltzal指令
REGIMM 000001	rs	BGEZ 00001	offset			bgez指令
REGIMM 000001	rs	BGEZAL 10001	offset			bgezal指令
REGIMM 000001	00000	BGEZAL 10001	offset			bal指令

图 8-4 分支指令的格式

从图中知道，前五条指令可以直接根据指令码进行判断，后五条指令指令码都是REGIMM，需要根据16-20bit进一步判断。并且所有分支指令的第0-15bit都是offset，如果发生转移将其左移两位再扩展成32位，然后与延迟槽指令的地址相加，得到的结果就是转移目的地址。

转移目标地址 = (signed_extend)(offset || '00') + (pc+4)

- 当指令中的指令码为6'b000100时，表示beq指令。

指令用法为：beq rs, rt, offset。

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

- 当指令中的指令码为6'b000100，且16-25bit为0时，表示b指令。

指令用法为：b offset。

指令作用为：无条件转移，从图8-4可知，b指令可以认为是beq指令的特殊情况，当beq指令的rs、rt都等于0时，即为b指令，所以在OpenMIPS实现的时候不需要特意实现b指令，只需要实现beq指令即可。

- 当指令中的指令码为6'b000111时，表示bgtz指令。

指令用法为：bgtz rs, offset。

指令作用为：if rs > 0 then branch，如果地址为rs的通用寄存器的值大于零，那么发生转移。

- 当指令中的指令码为6'b000110时，表示blez指令。

指令用法为：blez rs, offset。

指令作用为：if rs ≤ 0 then branch，如果地址为rs的通用寄存器的值小于等于零，那么发生转移。

- 当指令中的指令码为6'b000101时，表示bne指令。

指令用法为：bne rs, rt, offset。

指令作用为：if rs ≠ rt then branch，如果地址为rs的通用寄存器的值不等于地址为rt的通用寄存器的值，那么发生转移。

- 当指令中的指令码为REGIMM，且第16～20bit为5'b00000时，表示bltz指令。

指令用法为：bltz rs, offset。

指令作用为：if rs < 0 then branch，如果地址为rs的通用寄存器的值小于0，那么发生转移。

- 当指令中的指令码为REGIMM，且第16～20bit为5'b10000时，表示bltzal指令。

指令用法为：bltzal rs, offset。

指令作用为：if rs < 0 then branch，如果地址为rs的通用寄存器的值小于0，那么发生转移，并且将转移指令后面第2条指令的地址作为返回地址，保存到通用寄存器$31。

● 当指令中的指令码为 REGIMM，且第 16～20bit 为 5'b00001 时，表示 bgez 指令。

指令用法为：bgez rs, offset。

指令作用为：if rs ≥ 0 then branch，如果地址为 rs 的通用寄存器的值大于等于 0，那么发生转移。

● 当指令中的指令码为 REGIMM，且第 16～20bit 为 5'b10001 时，表示 bgezal 指令。

指令用法为：bgezal rs, offset。

指令作用为：if rs ≥ 0 then branch，如果地址为 rs 的通用寄存器的值大于等于 0，那么发生转移，并且将转移指令后面第 2 条指令的地址作为返回地址，保存到通用寄存器$31。

● 当指令中的指令码为 REGIMM，且第 21～25bit 为 0，第 16～20bit 为 5'b10001 时，表示 bal 指令。

指令用法为：bal offset。

指令作用为：无条件转移，并且将转移指令后面第 2 条指令的地址作为返回地址，保存到通用寄存器$31。从图 8-4 的指令格式可知，bal 指令是 bgezal 指令的特殊情况，当 bgezal 指令的 rs 为 0 时，就是 bal 指令，所以在 OpenMIPS 实现时，不用特意考虑 bal 指令，只要实现 bgezal 指令即可。

跳转指令和分支指令有点复杂，这也是汇编中子程序设计中比较复杂和难以理解的地方，因此我把书中的内容全部截图过来了。

值得注意的是，所有的分支指令在转移到目标地址前都要先执行延迟槽中的指令。

现在我有个疑问，分支指令和转移指令有什么区别？例如 b 指令和 j 指令有什么区别？还有为什么要保存转移指令后面第二条指令的地址？

三、转移指令的实现

3.1 修改数据流图

我们将在译码阶段进行转移条件的判断，数据流图修改如下：

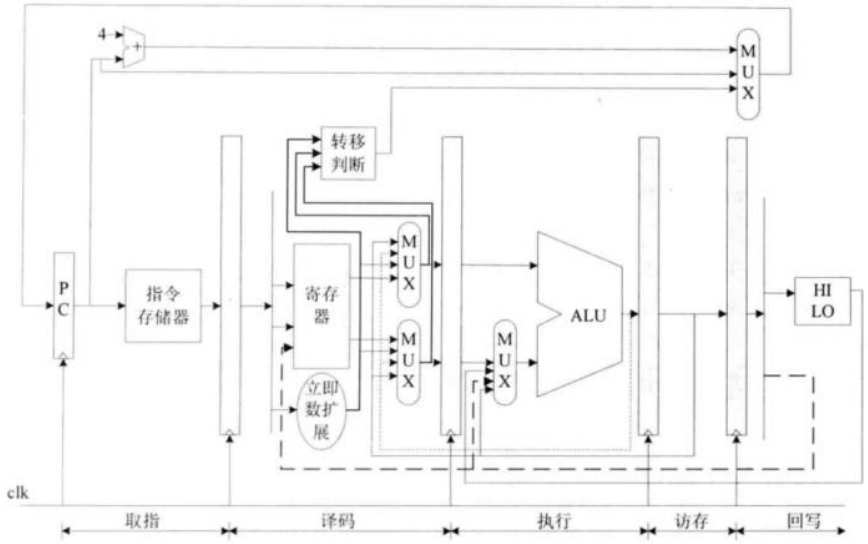


图 8-5 为实现转移指令而修改的数据流图

此时 PC 的取值有三种情况：PC+4，保持不变或者变成转移判断的结果。

3.2 系统结构的修改

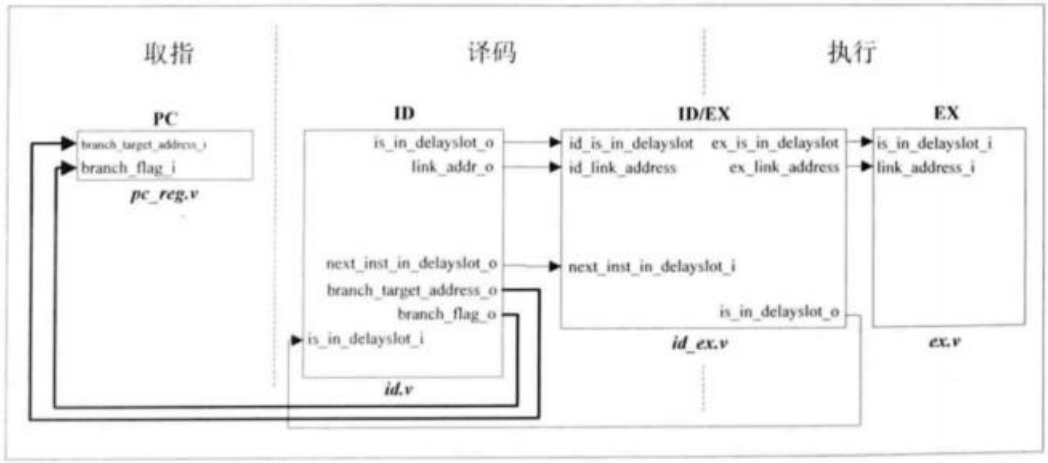


图 8-6 为实现转移指令而对系统结构所做的修改

3.3 修改 PC 模块

增加的接口如下表：

表 8-1 PC 模块增加的接口描述

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	branch_flag_i	1	输入	是否发生转移
2	branch_target_address_i	32	输入	转移到的目标地址

代码如下：

```
always @(posedge clk) begin
    if (ce == `ChipDisable) begin
        pc <= 32'h0000_0000;
    end else if (stall[0] == `NoStop) begin
        if (branch_flag_i == `Branch) begin
            pc <= branch_target_address_i;
        end else begin
            pc <= pc + 4'h4;
        end
    end
    $monitor("PC stall[0]==",stall[0]);
end
```

3.4 修改 ID 模块

增加的接口如下表所示：

表 8-2 ID 模块新增加的接口描述

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	branch_flag_o	1	输出	是否发生转移
2	branch_target_address_o	32	输出	转移到的目标地址
3	is_in_delayslot_o	1	输出	当前处于译码阶段的指令是否位于延迟槽
4	link_addr_o	32	输出	转移指令要保存的返回地址
5	next_inst_in_delayslot_o	1	输出	下一条进入译码阶段的指令是否位于延迟槽
6	is_in_delayslot_i	1	输入	当前处于译码阶段的指令是否位于延迟槽

确定指令转移的过程：

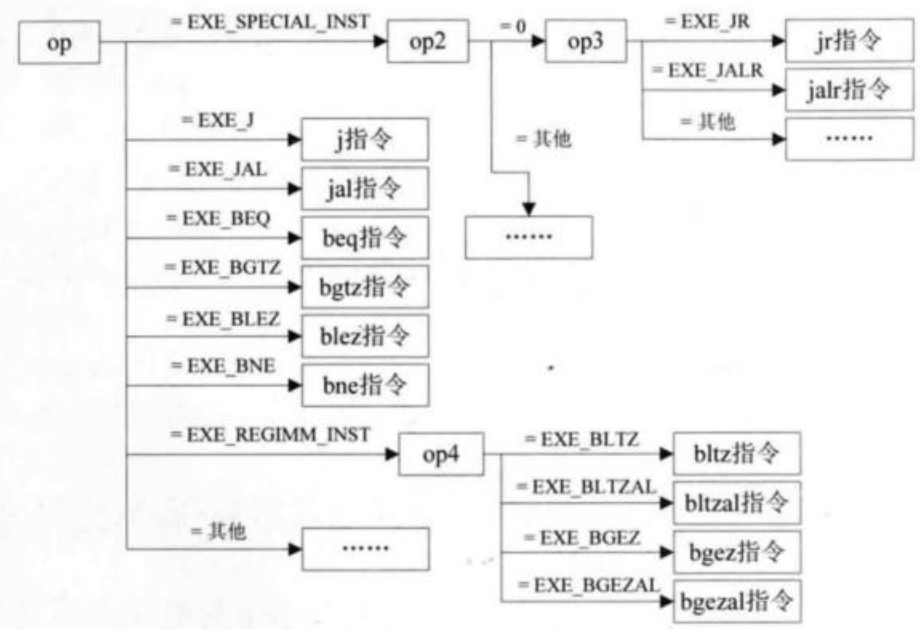


图 8-7 确定转移指令的过程

有几个问题：

（1） 为什么将 BNE 的类型赋值为`EXE_BLEZ_OP，类似的有 BLTZ 赋值为`EXE_BGEZAL，BLTZAL 赋值为`EXE_BGEZAL 等。实际上 aluop 这个运算子类型的值是为了在 EX 执行阶段方便进行运算而定义的，但是本章的跳转和分支无需运算，故这里的所有 aluop 都可以定义为`EXE_NOP_OP.

（2） 转移目标地址怎么计算，为什么那样计算？

（3） 保存的地址为什么是延迟槽之后第二条指令？这样有什么作用和意义？

3.5 修改 ID/EX 模块

增加的接口如下表所示：

表 8-3 ID/EX 模块新增加的接口描述

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	id_is_in_delayslot	1	输入	当前处于译码阶段的指令是否位于延迟槽
2	id_link_address	32	输入	处于译码阶段的转移指令要保存的返回地址
3	next_inst_in_delayslot_i	1	输入	下一条进入译码阶段的指令是否位于延迟槽
4	ex_is_in_delayslot	1	输出	当前处于执行阶段的指令是否位于延迟槽
5	ex_link_address	32	输出	处于执行阶段的转移指令要保存的返回地址
6	is_in_delayslot_o	1	输出	当前处于译码阶段的指令是否位于延迟槽

该部分代码很简单，当流水线译码阶段没有暂停时，ID/EX 模块在时钟上升沿将新增加的输入传递到对应的输出。

```
end else if (stall[2] == `NoStop) begin
    ex_aluop <= id_aluop;
    ex_alusel <= id_alusel;
    ex_reg1 <= id_reg1;
    ex_reg2 <= id_reg2;
    ex_wd <= id_wd;
    ex_wreg <= id_wreg;
    ex_link_address <= id_link_address;
    ex_is_in_delayslot <= id_is_in_delayslot;
    is_in_delayslot_o <= next_inst_in_delayslot_i;
end
```

3.6 修改 EX 模块

增加的接口如下表所示：

表 8-4 EX 模块新增加的接口描述

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	is_in_delayslot_i	1	输入	当前处于执行阶段的指令是否位于延迟槽
2	link_address_i	32	输入	处于执行阶段的转移指令要保存的返回地址

```
`EXE_RES_JUMP_BRANCH: begin
    wdata_o <= link_address_i;
end
```

注意一点，此处没有用到输入的信号 is_in_delayslot_i，该信号表示当前处于执行阶段的指令是否为延迟槽指令，这个信号在异常处理中会使用，这里暂时不需要。

3.7 修改 OpenMIPS 模块

细心一点添加接口即可。

四、测试结果

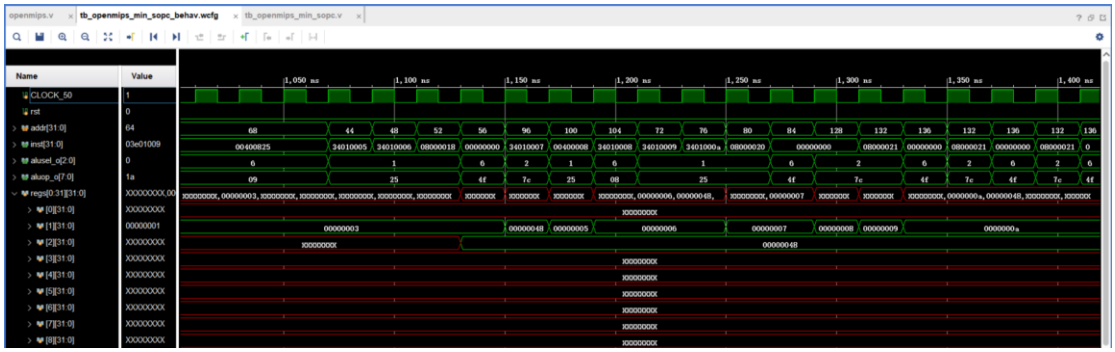
4.1 测试跳转指令

我们使用测试文件：

```
   .org 0x0
   .set noat
   .set noreorder
   .set nomacro
   .global _start
_start:
   ori  $1,$0,0x0001   # $1 = 0x1
   j    0x20
   ori  $1,$0,0x0002   # $1 = 0x2
   ori  $1,$0,0x1111
   ori  $1,$0,0x1100

   .org 0x20
   ori  $1,$0,0x0003   # $1 = 0x3
   jal  0x40
   div  $zero,$31,$1   # $31 = 0x2c, $1 = 0x3
                       # HI = 0x2, LO = 0xe
   ori  $1,$0,0x0005   # r1 = 0x5
   ori  $1,$0,0x0006   # r1 = 0x6
   j    0x60
   nop

   .org 0x40

   jalr $2,$31
   or   $1,$2,$0        # $1 = 0x48
   ori  $1,$0,0x0009    # $1 = 0x9
   ori  $1,$0,0x000a    # $1 = 0xa
   j 0x80
   nop
```

```
   .org 0x60
   ori  $1,$0,0x0007    # $1 = 0x7
   jr   $2
   ori  $1,$0,0x0008    # $1 = 0x8
   ori  $1,$0,0x1111
   ori  $1,$0,0x1100

   .org 0x80
   nop

_loop:
   j _loop
   nop
```

测试结果为，结果正确。



4.2 测试分支指令

使用测试文件：

```
   .org 0x0
   .set noat
   .set noreorder
   .set nomacro
   .global _start
_start:
   ori  $3,$0,0x8000
   sll  $3,16              # $3 = 0x80000000
   ori  $1,$0,0x0001       # $1 = 0x1
   b    s1
   ori  $1,$0,0x0002       # $1 = 0x2
1:
   ori  $1,$0,0x1111
   ori  $1,$0,0x1100

   .org 0x20
s1:
   ori  $1,$0,0x0003       # $1 = 0x3
   bal  s2
   div  $zero,$31,$1       # $31 = 0x2c, $1 =0x3
                           # HI = 0x2, LO = 0xe
   ori  $1,$0,0x1100
   ori  $1,$0,0x1111
   bne  $1,$0,s3
   nop
   ori  $1,$0,0x1100
   ori  $1,$0,0x1111


   .org 0x50
s2:
   ori  $1,$0,0x0004       # $1 = 0x4
   beq  $3,$3,s3
   or   $1,$31,$0          # $1 = 0x2c
   ori  $1,$0,0x1111
   ori  $1,$0,0x1100
2:
   ori  $1,$0,0x0007       # $1 = 0x7
   ori  $1,$0,0x0008       # $1 = 0x8
   bgtz $1,s4
   ori  $1,$0,0x0009       # $1 = 0x9
   ori  $1,$0,0x1111
   ori  $1,$0,0x1100

   .org 0x80
s3:
   ori  $1,$0,0x0005       # $1 = 0x5
   BGEZ $1,2b
   ori  $1,$0,0x0006       # $1 = 0x6
   ori  $1,$0,0x1111
   ori  $1,$0,0x1100
```

```
   .org 0x100
s4:
   ori  $1,$0,0x000a       # $1 = 0xa
   BGEZAL $3,s3
   or   $1,$0,$31          # $1 = 0x10c
   ori  $1,$0,0x000b       # $1 = 0xb
   ori  $1,$0,0x000c       # $1 = 0xc
   ori  $1,$0,0x000d       # $1 = 0xd
   ori  $1,$0,0x000e       # $1 = 0xe
   bltz $3,s5
   ori  $1,$0,0x000f       # $1 = 0xf
   ori  $1,$0,0x1100


   .org 0x130
s5:
   ori  $1,$0,0x0010       # $1 = 0x10
   blez $1,2b
   ori  $1,$0,0x0011       # $1 = 0x11
   ori  $1,$0,0x0012       # $1 = 0x12
   ori  $1,$0,0x0013       # $1 = 0x13
   bltzal $3,s6
   or   $1,$0,$31          # $1 = 0x14c
   ori  $1,$0,0x1100


   .org 0x160
s6:
   ori $1,$0,0x0014        # $1 = 0x14
   nop

_loop:
   j _loop
   nop
```

测试结果如下，结果正确。