自己动手写CPU-实验三

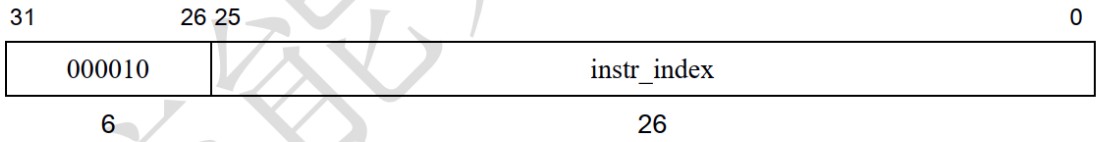
by CPU兴趣小组

转移指令的实现

转移指令编码

下面列出的五条指令是"计算机系统设计"书中第六章实现的五条转移指令, 这五条指令如下图所示

3.6.9 J



汇编格式： J target

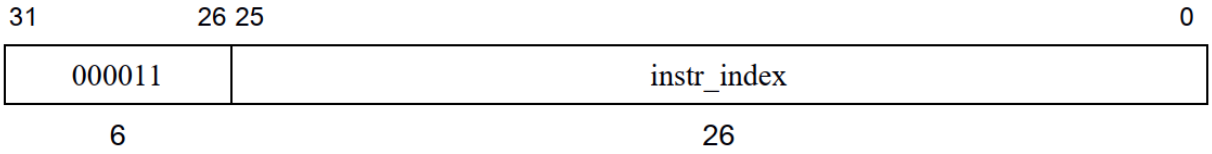
功能描述： 无条件跳转。跳转目标由该分支指令对应的延迟槽指令的 PC 的最高 4 位与立即数 instr_index 左移 2 位后的值拼接得到。

操作定义： I:

I+1: $PC \leftarrow PC_{31..28} \| instr\_index \| 0^2$

例　外： 无

3.6.10 JAL



汇编格式： JAL target

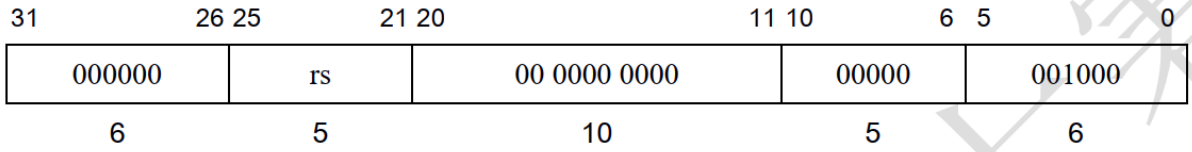
功能描述： 无条件跳转。跳转目标由该分支指令对应的延迟槽指令的 PC 的最高 4 位与立即数 instr_index 左移 2 位后的值拼接得到。同时将该分支对应延迟槽指令之后的指令的 PC 值保存至第 31 号通用寄存器中。

操作定义： I: $GPR[31] \leftarrow PC + 8$

I+1: $PC \leftarrow PC_{31..28} \| instr\_index \| 0^2$

例　外： 无

3.6.11 JR



汇编格式： JR rs

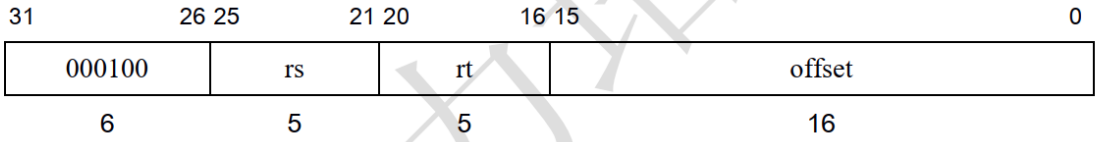
功能描述： 无条件跳转。跳转目标为寄存器 rs 中的值。

操作定义： I: temp ← GPR[rs]

I+1: PC ← temp

例　外： 无

3.6.1 BEQ



汇编格式： BEQ rs, rt, offset

功能描述： 如果寄存器 rs 的值等于寄存器 rt 的值则转移，否则顺序执行。转移目标由立即数 offset 左移 2 位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的 PC 计算得到。

操作定义： I: condition ← GPR[rs] = GPR[rt]

target_offset ← Sign_extend(offset||0^2)

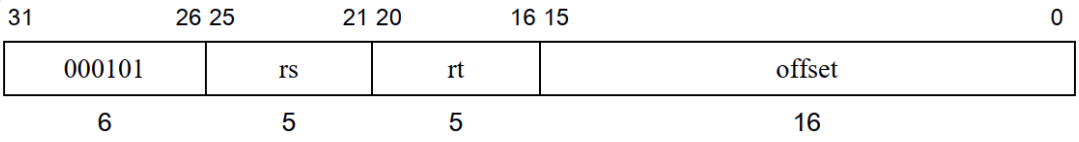
I+1: if condition then

PC ← PC+ target_offset

endif

例　外： 无

3.6.2 BNE



汇编格式： BNE rs, offset

功能描述： 如果寄存器 rs 的值不等于寄存器 rt 的值则转移，否则顺序执行。转移目标由立即数 offset 左移 2 位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的 PC 计算得到。

操作定义： I: condition ← GPR[rs] ≠ GPR[rt]

target_offset ← Sign_extend(offset||0^2)

I+1: if condition then

PC ← PC+ target_offset

endif

例　外： 无

转移指令的分类

转移指令分为无条件转移和有条件转移. 所谓的无条件的转移就是直接跳转, 例如J JAL JR指令, 而BEQ和BNE还需要判断条件, 对rs和rt字段寄存器的值进行比较, 满足条件才发生分支.

由指令编码可以看出, J和JAL指令都是J型指令, 而JR其实是是R型指令, BEQ和BNE是I型指令. 可以看到部分转移指令会读寄存器堆的值, 还有一些转移指令会写寄存器堆, 这些特点都要纳入设计的考虑之中.

原框架需要的修改

取指级

我们常说的pc寄存器（program counter)是在stage_if.sv中进行定义的，对于stage_if取指级，需要根据译码级给出的 `转移目的地址` 和 `是否发生转移` 信号来决定下一个周期的pc值。

译码级

增加模块端口

译码级需要在原来的基础上增加接口: `转移目的地址` ， `是否发生转移` .

增加内部信号

判断是否跳转:

1. 无条件跳转的指令一定会跳转.
2. 有条件的指令, 需要按照指令的要求, 在译码级增加用于比较的电路.

跳转的目的地址

1. J型指令(J和JAL)直接取低位的offset即可.
2. R型指令(JR), 需要用到rs字段的值.
3. I型指令BNE和BEQ, 按照指令集定义实现即可.

TIPS: 这里的实现是不是需要一个MUX，选出最终的跳转地址，输出给pc寄存器?

对于JAL指令的处理

由于JAL指令需要写入通用寄存器堆中的31号寄存器，那么根据Docs文件夹里的指令编码表，在这里我们发现我们需要增加一个位选信号，名称为r31sel，意思是当前指令是转移指令并且会写31号寄存器。定义r31sel之后，根据JAL指令的功能，它会将当前指令的PC的值加8之后写入到31号寄存器，我们把 `PC+8` 这个数值归类到源操作数1中，此时源操作数2的值我们不关心，因为执行级只需要用到一个操作数来存储 `PC+8` 就可以了。

执行级

对于JAL这条指令，它会写入31号寄存器，我们可以把他当作是一种“ALU指令”，虽然这种看法有些不合适，但我们希望尽量复用已经有的端口和信号（例如已经有的exe_i_rfwe,exe_i_rfwa等），我们希望尽量不增加新的信号。于是我们可以增加一种新的alutype叫做JUMP，再增加一种新的aluop叫做ALU_R31,意思是此时ALU执行的操作是写入31号寄存器中，此时会将译码级传递过来的源操作数赋给alures，之后随流水线传递，最终在写回级写入31号寄存器中。

对于其他不需要写通用寄存器堆的转移指令，执行级不需要特殊的改动。

访存级&写回级

TIPS: 如果已经完成了上文中描述的修改工作，这两级还需要进行改动吗?

测试代码1: Jump指令

你需要在im.sv中检查一下目前的指令存储器的内容是否是“Only Simple jump inst test case”对应的内容。

```
        .set noat
        .set noreorder
        .globl main
        .text
main:
   lui  $at, 0x1        # (PC = 0x00)   $at($1) = 0x10000
   j    L2              # (PC = 0x04)   jump to L2(addr = 0x14)
   nop                  # (PC = 0x08)   delayslot instrution

L3:
   jal  L4              # (PC = 0x0C)   jump tp L4(addr = 0x24) $ra($31) = 0x14
   lui  $at, 0x3        # (PC = 0x10)   delayslot instrution, $at = 0x30000

L2:
   la   $v0, L3         # (PC = 0x14)   $v0($2) = L3's addr
   jr   $v0             # (PC = 0x1C)   jump tO L3(addr = 0x0c)
   lui  $at, 0x2        # (PC = 0x20)   delayslot instrution, $at = 0x20000

L4:
   nop
```

测试代码2：Jump and branch共同测试

你需要在im.sv中检查一下目前的指令存储器的内容是否是“Complex Jump and branch inst test case”对应的内容，如果不是的话要把注释修改掉。

```
        .set noat
        .set noreorder
        .globl main
        .text
main:
   lui  $at, 0x1        # (PC = 0x00)   $at($1) = 0x10000
   j    L2              # (PC = 0x04)   jump to L2(addr = 0x14)
   nop                  # (PC = 0x08)   delayslot instrution

L3:
   jal  L4              # (PC = 0x0C)   jump tp L4(addr = 0x24) $ra($31) = 0x14
   lui  $at, 0x3        # (PC = 0x10)   delayslot instrution, $at = 0x30000

L2:
   la   $v0, L3         # (PC = 0x14)   $v0($2) = L3's addr
   jr   $v0             # (PC = 0x1C)   jump tO L3(addr = 0x0c)
   lui  $at, 0x2        # (PC = 0x20)   delayslot instrution, $at = 0x20000

L4:
   bne  $at, $v0, L5    # (PC = 0x24)   branch to L5(addr = 0x30)
   lui  $at, 0x4        # (PC = 0x28)   delayslot instrution, $at = 0x40000

L6:
   lui  $at, 0xA        # (PC = 0x2C)   $at = 0xA0000 the cpu cannot run here

L5:
   lui  $at, 0x5        # (PC = 0x30)   $at = 0x50000
   lui  $v0, 0x6        # (PC = 0x34)   $v0 = 0x60000
   beq  $at, $v0, L6    # (PC = 0x38)   not branch
   nop                  # (PC = 0x3C)   delayslot instrution

loop:
   j    loop            # (PC = 0x40)
   nop                  # (PC = 0x44)   delayslot instrution
```

修改im.sv里的代码

把原来用于测试数据相关的指令注释掉，将第二个test case(simple jump test)取消注释。

有正确波形吗

没有。建议使用一种模拟器，QTspim或者MARZ，研究一下如何使用，然后让模拟器单步执行测试代码，看一下自己的CPU的行为和模拟器的是否一致。MARZ安装教程在CPU群文件->工具类中。