# 处理器设计文档

## NPC

### 原理

NPC 是下个 PC 值的意思。它能做到根据当前的 PC 值，计算出下一个 32 位的 PC 值。

一般来说，PC 值的转换是顺序转换。但是，NPC 必须要听控制模块的指令，做到在某些条件下进行符号转换。

### 接口定义

| 端口 | 类型 | 位宽 | 功能 |
| --- | --- | --- | --- |
| curr_pc | 输入 | 32 | 当前 PC |
| jump_mode | 输入 | 4 | 是否可以跳转 |
| cmp_result | 输入 | 2 | cmp 的比较结果 |
| cmp_sig_result | 输入 | 2 | cmp 的有符号比较结果 |
| num | 输入 | 16 | 输入的立即数 |
| jnum | 输入 | 26 | 输入的 J 型指令的立即数 |
| reg_ | 输入 | 32 | 输入的寄存器值 |
| next_pc | 输出 | 32 | 下一个 PC |

### 宏定义

用把宏定义成宏的方法，定义表中值为宏的宏。把一个宏定义成另一个宏，那该宏的意义与定义它的宏一样，表中省略。

| 类别 | 定义 | 值 | 意义 |
| --- | --- | --- | --- |
| jump_mode | NPC_JUMP_DISABLE | 4'b0000 | 不要跳转 |
| jump_mode | NPC_JUMP_DISABLED | NPC_JUMP_DISABLE | |
| jump_mode | NPC_JUMP_WHEN_EQUAL | 4'b0001 | 当输入的比较结果相等时跳转 |
| jump_mode | NPC_JUMP_WHEN_EQUALS_TO | NPC_JUMP_WHEN_EQUAL | |
| jump_mode | NPC_EQUAL | NPC_JUMP_WHEN_EQUAL | |
| jump_mode | NPC_JUMP_WHEN_NOT_EQUAL | 4'b0010 | 当输入的比较结果不等时跳转 |
| jump_mode | NPC_JUMP_WHEN_NOT_EQUALS_TO | NPC_JUMP_WHEN_NOT_EQUAL | |
| jump_mode | NPC_NOT_EQUAL | NOT_JUMP_WHEN_NOT_EQUAL | |
| jump_mode | NPC_REG | 4'b1111 | 按照寄存器内地址跳转 |
| jump_mode | NPC_J | 4'b1110 | 按照 J 型指令的立即数跳转 |
| jump_mode | NPC_LARGER | 4'b0011 | 当输入的比较结果为大于时跳转 |
| jump_mode | NPC_SMALLER | 4'b0100 | 当输入的比较结果为小于时跳转 |

| | | |
|---|---|---|
| jump_mode NPC_LARGER_OR_EQUAL | 4'b0101 | 当输入的比较结果为大于或等于时跳转 |
| jump_mode NPC_SMALLER_OR_EQUAL | 4'b0110 | 当输入的比较结果为小于或等于时跳转 |
| jump_mode NPC_SIG_LARGER | 4'b0111 | 当输入的有符号比较结果为大于时跳转 |
| jump_mode NPC_SIG_SMALLER | 4'b1000 | 当输入的有符号比较结果为小于时跳转 |
| jump_mode NPC_SIG_LARGER_OR_EQUAL | 4'b1001 | 当输入的有符号比较结果为大于或等于时跳转 |
| jump_mode NPC_SIG_SMALLER_OR_EQUAL | 4'b1010 | 当输入的有符号比较结果为小于或等于时跳转 |

comp_result 的相应数值代表的意义，与相应的宏有关，这些宏在 alu.h 中。

### 功能

令跳转基准地址 base = $unsigned(curr_pc)。

若 jump_mode == NPC_JUMP_DISABLE，则令 next_pc = $unsigned(base) + $unsigned(4)。

若 jump_mode == NPC_JUMP_WHEN_EQUAL，则 alu_comp_result == ALU_EQUAL 时，首先把 num 扩展成 32 位有符号立即数，扩展方式是首先把 num 后面加上 2'b0，然后把这 18 位二进制数扩展成 32 位有符号二进制数。然后令 next_pc = $signed(base) + $signed(num)。否则做跟 jump_mode == NPC_JUMP_DISABLE 时相同的步骤。

若 jump_mode 对应的意义有其它的比较，则 cmp_result 或 cmp_sig_result 满足相应条件时，做跟上面相同的步骤。否则做跟 jump_mode == NPC_JUMP_DISABLE 时相同的步骤。

若 jump_mode == NPC_REG，则令 next_pc = reg_。

若 jump_mode == NPC_J，则令 next_pc = {base[31:28], jnum, 2'b0}。

若 jump_mode 为其它值，则做跟 jump_mode == NPC_JUMP_DISABLE 时相同的步骤。

### 注意事项

1. NPC 是在内部进行符号扩展，不用 ext。
2. reg_ 是为了避免和 reg 冲突。
3. base 抽象出来是为了方便调试和维护，它是跟 MIPS 指令集手册相符的。

## PC

### 原理

PC 是程序计数器的意思，负责对当前的指令进行计数。它是标记程序执行到哪里的一种方法，同时输出的信息也被送入指令内存 IM，用来取指。

PC 只负责表示程序执行到哪里，而 PC 的更新由 NPC 模块负责。这样可以做到更简便地处理跳转指令、也对流水线 CPU 插入气泡有帮助。

### 端口定义

| 端口 | 类型 | 位宽 | 功能 |
| --- | --- | --- | --- |
| clk | 输入 | 1 | 时钟信号 |
| next_pc | 输入 | 32 | NPC 计算得来的下一个 PC 地址 |
| enable | 输入 | 1 | PC 使能 |
| curr_pc | 输出 | 32 | PC 地址 |

### 宏定义

用把宏定义成宏的方法，定义表中值为宏的宏。值为宏的宏的意义，与定义它的宏一样，在表中省略。

| 类别 | 定义 | 值 | 意义 |
| --- | --- | --- | --- |
| enable | PC_ENABLED | 1'b1 | PC 使能 |
| enable | PC_ENABLE | PC_ENABLED | |
| enable | PC_DISABLED | 1'b0 | PC 非使能 |
| enable | PC_DISABLE | PC_DISABLED | |
| curr_pc | PC_START_ADDRESS | 32'h00003000 | PC 的起始地址 |

### 功能

该部件是时序部件。

有一个 32 位的寄存器保存当前 PC 的值，初值为 PC_START_ADDRESS。

在每个时钟上升沿，若 enable == PC_ENABLED，则把 PC 部件中保存的当前 PC 的值更新成 next_pc 的值。否则，保存的当前 PC 的值不变。

无论什么时候，输出端口 curr_pc 的值都是 PC 部件中保存的当前 PC 的值。

### 注意事项

1. PC 和 IM 的起始地址是分开定义的，改的时候要注意。

## 程序存储器

### 原理

程序存储器是存储程序指令的地方。为了加载程序指令，它可以通过系统任务读取编译后的指令内容。

为了简便，程序存储器由许多寄存器实现。

### 端口定义

| 端口 | 类型 | 位宽 | 功能 |
| --- | --- | --- | --- |
| addr | 输入 | IM_ADDR_WIDTH | 读地址 |
| enable | 输入 | 1 | 使能信号 |
| result | 输出 | 32 | 读到的结果 |

### 宏定义

用把宏定义成宏的方法，定义表中值为宏的宏。值为宏的宏的意义，与定义它的宏一样，在表中省略。

| 类别 | 定义 | 值 | 意义 |
| --- | --- | --- | --- |
| enable | IM_ENABLE | 1'b1 | IM 使能 |
| enable | IM_ENABLED | IM_ENABLE | |
| enable | IM_DISABLE | 1'b0 | IM 非使能 |
| enable | IM_DISABLED | IM_DISABLE | |
| addr | IM_ADDR_WIDTH | 8 | addr 的位宽 |
| addr | IM_START_ADDRESS | 32 | IM 对外表现的起始地址 |
| 指令存储器 | IM_SIZE | 64 | 能存储指令的个数 |
| 指令存储器 | IM_CODE_FILENAME | "code/code.hex" | 要加载的机器码 |

### 功能

有 IM_SIZE 个 32 位存储器，代表其中存储的指令。它们初值应该使用加载文件的系统任务加载。加载文件名由 IM_CODE_FILENAME 指定。

若 addr 作为无符号数小于 IM_START_ADDRESS，则也返回 32'b0。否则，result 为 addr - IM_START_ADDRESS 这个地址再取 [IM_ADDR_WIDTH - 1:2] 对应的指令（从存储器中取得，是两个无符号数相减）。若相减后的结果超出了已经加载的指令所占的地址空间，则 result 为 32'b0。

### 注意事项

1. IM_ADDR_WIDTH 和 IM_SIZE 需要一块改，因为它们的大小有关系
2. 有 offset 了，注意跟 offset 相减是无符号数相减
3. offset 主要是为了和 MARS 兼容

## 寄存器堆

### 原理

寄存器堆保存着 32 位 32 个通用寄存器，负责存储 CPU 立刻想要的数据，它是存储器层次结构中的最高一级，负责暂存数据。第 0 号寄存器 $0 的值永远是 32'b0，写入不会改变它的值。

由于 MIPS 体系结构中的指令最多读两个寄存器，写一个寄存器，所以寄存器输入两个要读的地址，输出两个要读的数据；输入一个要写的地址和一个要写的数据；同时还有写使能端口。

寄存器的使用没有规定，这一般是软件关心的问题。

### 端口定义

| 端口 | 类型 | 位宽 | 功能 |
| --- | --- | --- | --- |
| clk | 输入 | 1 | 时钟信号 |
| curr_pc | 输入 | 32 | 当前 PC 的值 |
| read_addr1 | 输入 | 5 | 第一个读地址 |
| read_addr2 | 输入 | 5 | 第二个读地址 |
| write_addr | 输入 | 5 | 写地址 |
| write_data | 输入 | 32 | 要写入的数据 |
| write_enable | 输入 | 1 | 写使能 |
| read_result1 | 输出 | 32 | 第一个读地址读出的数据 |

| | | | |
|---|---|---|---|
| read_result2 | 输出 | 32 | 第二个读地址读出的数据 |

### 宏定义

用把宏定义成宏的方法，定义表中值为宏的宏。值为宏的宏的意义，与定义它的宏一样，在表中省略。

| 类别 | 定义 | 值 | 意义 |
|---|---|---|---|
| .*_addr.* | RF_ADDR_ZERO | 5'b0 | 零寄存器的地址 |
| .*_addr.* | RF_ZERO | RF_ADDR_ZERO | |
| write_enable | RF_WRITE_ENABLED | 1'b1 | 寄存器堆使能 |
| write_enable | RF_WRITE_ENABLE | RF_WRITE_ENABLED | |
| write_enable | RF_WRITE_DISABLED | 1'b0 | 寄存器堆非使能 |
| write_enable | RF_WRITE_DISABLE | RF_WRITE_DISABLED | |
| 输出 | RF_OUTPUT_FORMAT | "%d: 0x%08x => 0x%08x" | 输出模版 |

### 功能

该部件为时序部件。

有 31 个 32 位寄存器，代表 $1~$31，它们初值都为 32'b0。$0 实际上不需要寄存器。

在每个时钟上升沿，若 write_enable == RF_WRITE_ENABLED 且 write_addr != RF_ADDR_ZERO，则说明可以执行写操作，且写到的寄存器是可以保存数值的寄存器。此时把 write_addr 指代的寄存器的值更新为 write_data。更新时，以模版中的格式打印出数据变化，第一个参数是当前的模拟时钟的时间，第二个参数是当前 PC 的值，第三个参数是寄存器号，第四个参数是更新后的值。

无论什么时候，若 read_addr1 != RF_ADDR_ZERO，则把 read_addr1 指代的寄存器的值输出到 read_result1 中，否则把 32'b0 输出到 read_result1 中。对 read_addr2 和 read_result2 的相应操作相同。

### 注意事项

1. 暂时还没有内部转发。
2. 寄存器可以定义为 reg [31:1] registers [31:0]，把 $0 空出来。
3. TODO: 应该把正常显示 wrap 起来，等到 ISE 装好后再说

## 比较模块

### 原理

比较模块通过比较两个寄存器的数据，实现分支指令和条件传送指令的提前跳转，提高跳转的效率。

### 接口定义

| 端口 | 类型 | 位宽 | 功能 |
|---|---|---|---|
| reg1 | 输入 | 32 | 第一个寄存器的输入 |
| reg2 | 输入 | 32 | 第二个寄存器的输入 |
| cmp | 输出 | 2 | 无符号比较结果输出 |
| sig_cmp | 输出 | 2 | 有符号比较结果输出 |
| reg2_sig_cmp | 输出 | 2 | reg2 与 0 的有符号比较结果输出 |

### 宏定义

把 CMP_LARGER, CMP_SMALLER, CMP_EQUAL 分别定义成 ALU_LARGER, ALU_SMALLER, ALU_EQUAL。

### 功能

在 comp, sig_comp, reg2comp 三个输出端口分别输出第一个寄存器与第二个寄存器作为无符号数的比较结果、它们作为有符号数的比较结果和第二个寄存器与 0 作为有符号数的比较结果。

## 扩展器

### 功能

扩展器是专门执行扩展整数功能的运算。它能做到小于 32 位的整数向 32 位整数的转换，其中有符号转换，也有无符号转换。

转换器的模式由宏定义的方式指定，有符号扩展，也有无符号扩展，也有其它模式。由于没有明显的层次和类别关系，采用顺序编号和按常见顺序编号的方法。

### 接口定义

| 端口 | 类型 | 位宽 | 功能 |
| --- | --- | --- | --- |
| num | 输入 | 16 | 输入的数字 |
| mode | 输入 | 3 | 模式 |
| result | 输出 | 32 | 扩展的结果 |

### 宏定义

用把宏定义成宏的方法，定义表中值为宏的宏。把一个宏定义成另一个宏，那该宏的意义与定义它的宏一样，表中省略。

| 类别 | 定义 | 值 | 意义 |
| --- | --- | --- | --- |
| mode | EXT_MODE_SIGNED | 3'b000 | 符号扩展 |
| mode | EXT_SIGNED | EXT_MODE_SIGNED | |
| mode | EXT_MODE_UNSIGNED | 3'b001 | 无符号扩展 |
| mode | EXT_UNSIGNED | EXT_MODE_UNSIGNED | |
| mode | EXT_MODE_PAD | 3'b010 | 把输入的 16 位填充到输出结果的高 16 位，输出结果低 16 位置零的扩展 |
| mode | EXT_PAD | EXT_MODE_PAD | |
| mode | EXT_MODE_HIGH_BITS | EXT_MODE_PAD | |
| mode | EXT_HIGH_BITS | EXT_MODE_PAD | |
| mode | EXT_MODE_ONE | 3'b011 | 在数字前面填充二进制 1 的扩展 |
| mode | EXT_ONE | EXT_MODE_ONE | |

### 功能

若 mode 的值为合法操作（即上面“宏定义”一节中列出的操作），按照 mode 中给出的操作计算出结果，并把结果放入 result 中。

若 mode 的值为非法操作，就令 result = 32'b0。

# ALU

## 原理

ALU 是运算控制单元的意思，负责两个 32 位整数的运算。它可以负责各种运算，包括数学运算和逻辑运算。易知它是纯组合逻辑。

由于定义运算的时候需要给运算编码，所以表示运算就有点类似于 C 语言中的 enum。因此，需要对各种运算进行宏定义，以保证系统的可维护性。宏定义也可以把定义的数据空间分隔开，以及对运算按照逻辑进行排序，从而得到对端口运算编码的更好理解。

## 端口定义

| 端口 | 类型 | 位宽 | 功能 |
|---|---|---|---|
| num1 | 输入 | 32 | 第一个操作数 |
| num2 | 输入 | 32 | 第二个操作数 |
| shamt | 输入 | 5 | 移位运算的移位位数 |
| op | 输入 | 5 | 操作符 |
| result | 输出 | 32 | 结果 |
| cmp_result | 输出 | 2 | 作为无符号数的比较结果 |
| sig_cmp_result | 输出 | 2 | 作为有符号数的比较结果 |
| overflow | 输出 | 1 | 计算过程中是否发生溢出 |
| op_invalid | 输出 | 1 | 操作符是否无效 |

由于在硬件层级对数的加减都是无符号数加减法，所以这里的溢出，是指操作过程中出现了做无符号数加减法时结果超出无符号数范围的现象。

## 宏定义

采用操作符最高两位区分类别的方法定义宏。用把宏定义成宏的方法，定义表中值为宏的宏。

| 类别 | 定义 | 值 | 意义 |
|---|---|---|---|
| op | ALU_ADD | 5'b00000 | 加法运算 |
| op | ALU_UNSIGNED_ADD | ALU_ADD | 同上 |
| op | ALU_SUB | 5'b00001 | 减法运算 |
| op | ALU_UNSIGNED_SUB | ALU_SUB | 同上 |
| op | ALU_AND | 5'b10000 | 按位与运算 |
| op | ALU_BITWISE_AND | ALU_AND | 同上 |
| op | ALU_OR | 5'b10001 | 按位或运算 |
| op | ALU_BITWISE_OR | ALU_OR | 同上 |
| op | ALU_NOT | 5'b10010 | 按位非运算 |
| op | ALU_BITWISE_NOT | ALU_NOT | 同上 |
| op | ALU_XOR | 5'b10011 | 按位异或运算 |
| op | ALU_MOVZ | 5b'00010 | 数据转移运算（[1]） |
| op | ALU_NOR | 5'b10100 | 按位或非运算 |

| | | | |
|---|---|---|---|
| op | ALU_SLT | 5'b00011 | 若小于则设置运算 |
| op | ALU_SLTU | 5'b00100 | 无符号的若小于则设置运算 |
| op | ALU_SLL | 5'b10101 | 左移位运算 |
| op | ALU_SRL | 5'b10110 | 逻辑右移位运算 |
| op | ALU_SRA | 5'b10111 | 算数右移位运算 |
| op | ALU_SLLV | 5'b11000 | 寄存器为参数的左移位运算 |
| op | ALU_SRLV | 5'b11001 | 寄存器为参数的逻辑右移位运算 |
| op | ALU_SRAV | 5'b11110 | 寄存器为参数的算数右移位运算 |
| .*cmp_result | ALU_EQUAL | 2b'00 | 等于 |
| .*cmp_result | ALU_EQUAL_TO | ALU_EQUAL | 同上 |
| .*cmp_result | ALU_LARGER | 2b'01 | 大于 |
| .*cmp_result | ALU_LARGER_THAN | ALU_LARGER | 同上 |
| .*cmp_result | ALU_SMALLER | 2b'10 | 小于 |
| .*cmp_result | ALU_SMALLER_THAN | ALU_SMALLER | 同上 |
| overflow | ALU_OVERFLOW | 1'b1 | 溢出 |
| overflow | ALU_NOT_OVERFLOW | 1'b0 | 未溢出 |
| op_invalid | ALU_INVALID_OP | 1'b1 | 操作符无效 |
| op_invalid | ALU_INVALID | ALU_INVALID_OP | 同上 |
| op_invalid | ALU_VALID_OP | 1'b0 | 操作符有效 |
| op_invalid | ALU_VALID | ALU_VALID_OP | 同上 |

注:

1. 数据转移运算只是简单地让结果等于第一个操作数，因为真正转不转移是控制模块判断写入哪个寄存器决定的。

## 功能

若 op 的值为合法操作（即上面“宏定义”一节中列出的操作），按照 op 中给出的操作计算出结果，并把结果放入 result 中。然后把输入的数看成无符号数并比较，若发生上面提到的溢出现象，就令 overflow 为 1'b1，否则为 1'b0。注意不管 num[12] 输入的原来意义是什么，都把它看成无符号数进行计算。

检查溢出的方式是用一个 33 位的中间变量，在加减法时用同样的方法算出该中间变量的值。如果有溢出，那它的最高位应该为 1，否则为 0。在做其它运算时，把这个中间变量变为恒 0。

如果 op 的值为非法操作，就令 op_invalid 为 1'b1，否则为 1'b0。此时令 result 为 32'b0。

.*cmp_result 的值仅由 num[12] 确定，与其它输入无关。.*cmp_result 的比较方式，在端口定义中。比较的输出结果，在宏定义中。不会输出宏定义中没有定义的结果。

若小于则设置运算指的是把 alu.num1 和 alu.num2 作为有符号数比较，若 alu.num1 < alu.num2，则 result = 32'b1，否则 result = 32'b0。无符号的若小于则设置运算是把要比较的两个数作为无符号数比较，之后和若小于则设置运算相同。

左右移位运算如果不说以寄存器为参数，就用 shamt 作为移位位数，否则用 num1 的最后 5 位作为移位位数。所有的移位运算都是对 num2 进行移位。如果当前 op 不对应移位运算，则移位位数为 0。

## 注意事项

1. 添加新运算时注意同时改 `op_invalid` 的输出和 `result` 的输出
2. 如果不确定符号，就加上 `[un|]signed`
3. 由于 ISE 不支持以变量为位数对标量切片，所以只能提前穷举移位位数的 31 种情况，然后进行切片，如果移位位数不属于 $[0, 31]$，那么切片结果是原来的标量

# 乘除法器

## 原理

乘除法器是 MIPS 体系结构中运算代价比较高的部件，需要多个时钟周期进行运算。因此，它必须要有 busy 信号指定它是不是忙，通过是不是忙来让控制模块决定暂停和转发。

两个 32 位数的乘法结果是 64 位，所以需要两个 32 位寄存器。同样地，除法有商和余数，所以也需要两个 32 位寄存器。总结起来，乘除法器需要两个 32 位寄存器。乘法有高低位的区别，把它们叫做 HI 和 LO 寄存器能区分过来。

乘除法需要多个周期，但是模拟时乘除法只要一个周期。因此，需要在乘除法器内部设定一个计时器，模拟需要多个周期乘除法的行为。还没有准备好时，就在 HI 和 LO 寄存器输出代替的值。

## 端口定义

| 端口 | 类型 | 位宽 | 功能 |
|---|---|---|---|
| clk | 输入 | 1 | 时钟信号 |
| dh | 输入 | 32 | 第一个输入的数据 |
| dl | 输入 | 32 | 第二个输入的数据 |
| op | 输入 | 4 | 需要的操作 |
| busy | 输出 | 1 | 乘除法器是否繁忙 |
| invalid | 输出 | 1 | 乘除法操作是否非法 |
| hi | 输出 | 32 | HI 寄存器的值 |
| lo | 输出 | 32 | LO 寄存器的值 |

## 宏定义

各宏的意义如果是对应的操作，就省略。

| 类别 | 定义 | 值 | 意义 |
|---|---|---|---|
| op | MD_NONE | 4'd0000 | |
| op | MD_MFHI | 4'd0001 | |
| op | MD_MFLO | 4'd0010 | |
| op | MD_MTHI | 4'd0011 | |
| op | MD_MTLO | 4'd0100 | |
| op | MD_MULT | 4'd0101 | |
| op | MD_MULTU | 4'd0110 | |
| op | MD_DIV | 4'd0111 | |
| op | MD_DIVU | 4'd1000 | |

## 功能

该部件为时序部件，所有寄存器初值为 0.

有一个 4 位宽的计时器 ctr。

无论什么时候，hi 和 lo 都分别是 hi_reg 和 lo_reg 的内容。无论什么时候，busy 都是 ctr 作为无符号数大于 0 时为 1'b1，否则为 1'b0。invalid 都是 (op_i == MD_DIV || op_i == MD_DIVU) && dl_i == 32'b0 时为 1'b1，否则为 1'b0。检查内部寄存器是因为内部寄存器才是 md 内保存的真正状态，而且在一个时钟周期内检查来得及。

每个时钟上升沿，若计时器 ctr 的值作为无符号数大于 1，则把 ctr 减 1。否则，若 ctr 的值作为无符号数等于 1，则令 hi_reg 和 lo_reg 分别为 dh_i 和 dl_i 寄存器在相应运算下的结果。否则，什么也不做。

每个时钟下降沿，若 op == MD_NONE || $unsigned(ctr) > $unsigned(0)，则什么也不做。

否则，若 op == MD_MTHI || op == MD_MTLO，则把 dh 的值写入相应的寄存器。若 op == MD_MFHI || op == MD_MFLO，也是什么也不做，因为控制模块会自动选择相应的寄存器作为要写入的值。若 op == MD_MULT || op == MD_MULTU || op == MD_DIV || op == MD_DIVU，则把 ctr 设置成对应的数值，而且把内部 dh_i、dl_i 和 op_i 寄存器的值分别设置成 dh、dl和 op。乘法操作ctr初值是 5，除法操作ctr初值是 10。若op` 为其它值，也是什么也不做。

注意：**除法是把 dh 当被除数，dl 当除数。但结果表示的时候，hi 当余数，lo 当商。若 dl == 32'b0，则结果为 64'b0。**

注意：**没有考虑第一条指令是 CAL_M 类指令，但是时钟没有下降沿的情况。**

# 数据存储器

## 原理

数据存储器是存储数据的地方。

为了简便，数据存储器由许多寄存器实现。

## 端口定义

| 端口 | 类型 | 位宽 | 功能 |
| --- | --- | --- | --- |
| clk | 输入 | 1 | 时钟信号 |
| curr_pc | 输入 | 32 | 当前 PC 值 |
| read_addr | 输入 | 32 | 读地址 |
| write_addr | 输入 | 32 | 写地址 |
| write_data | 输入 | 32 | 写数据 |
| write_enable | 输入 | 1 | 写使能信号 |
| mode | 输入 | 3 | 模式选择 |
| read_result | 输出 | 32 | 读到的结果 |
| invalid | 输出 | 1 | 地址是否错误 |

## 宏定义

用把宏定义成宏的方法，定义表中值为宏的宏。值为宏的宏的意义，与定义它的宏一样，在表中省略。

| 类别 | 定义 | 值 | 意义 |
| --- | --- | --- | --- |
| write_enable | DM_WRITE_ENABLE | 1'b1 | DM 使能 |

| | | | |
|---|---|---|---|
| write_enable | DM_WRITE_ENABLED | DM_WRITE_ENABLE | |
| write_enable | DM_WRITE_DISABLE | 1'b0 | DM 非使能 |
| write_enable | DM_WRITE_DISABLED | DM_WRITE_DISABLE | |
| mode | DM_NONE | 3'b000 | 不操作 dm |
| mode | DM_W | 3'b001 | 读取/写入一个字 |
| mode | DM_H | 3'b011 | 读取/写入半个字 |
| mode | DM_HU | 3'b010 | 读取半个字，按无符号数读取 |
| mode | DM_B | 3'b101 | 读取/写入一个字节 |
| mode | DM_BU | 3'b110 | 读取一个字节，按无符号数读取 |
| .*_addr | DM_ADDR_WIDTH | 8 | .*_addr 的位宽 |
| 指令存储器 | DM_SIZE | 64 | 能存储 32 位字的个数 |

### 功能

该部件为时序部件。

有 DM_SIZE 个 32 位存储器，代表其中存储的指令。它们初值都为 32'b0。

首先得出操作地址 op_addr。若 write_enable == DM_ENABLED，则操作地址为写地址，否则操作地址为读地址。

然后确定操作是否合法。若 mode == DM_NONE || (mode == DM_W && op_addr[1:0] == 2'b0) || (mode == DM_H && op_addr[0] == 1'b0) || (mode == DM_HU && op_addr[0] = 1'b0) || mode == DM_B || mode == DM_BU，则操作合法，否则操作不合法。

在每个时钟上升沿，若 write_enable == DM_ENABLED && invalid == 0，则根据操作模式写入相应地址对应的数据。写入半个字和字节分别取 write_data 的低 16 位和低 8 位。同时，打印当前 PC 的值、write_addr 对应的字和它对应字的新值。如果是写入半个字或者字节，也打印对应字的新值。

任何时候，若 invalid == 1'b1，则 read_result == 32'b0。否则，若 mode == DM_NONE，则 read_result == 32'b0。若 mode 为其它 dm 宏的值，则按照相应宏的意义读出数据，读到 read_result 中。若 mode 为其他值，则 read_result == 32'b0。

注意 dm 内部对 write_addr 和 read_addr 都截取了一部分。这样可以把 dm 直接接入数据通路，在数据通路中假定地址是 32 位的；同时 dm 的实现不需要那么多寄存器，更现实。但是实际上这样对地址空间进行了限制。

### 注意事项

1. DM_ADDR_WIDTH 和 DM_ADDR_SIZE 要一块改
2. 地址空间是被截断的，看起来是 32 位，实际上不是
3. **CPU 是小端序的**
4. 为了能打印出写到的字的值，可以把值提前用组合电路算出来

## 流水线寄存器

### 原理

流水线中需要很多寄存器来保存中间状态，而直接使用 always 块写，有不容易管理的缺点。所以更好的方法是设置流水线寄存器。

### 端口定义

| 端口 | 类型 | 位宽 | 功能 |
| --- | --- | --- | --- |
| clk | 输入 | 1 | 时钟信号 |
| enable | 输入 | 1 | 使能 |
| rst | 输入 | 1 | 同步复位信号 |
| i | 输入 | BIT_WIDTH | 输入的数据 |
| o | 输出 | BIT_WIDTH | 输出的数据 |

### 参数定义

| 类别 | 定义 | 默认值 | 意义 |
| --- | --- | --- | --- |
| 寄存器位宽 | BIT_WIDTH | 32 | 寄存器的位宽 |

### 宏定义

| 类别 | 定义 | 值 | 意义 |
| --- | --- | --- | --- |
| enable | PFF_ENABLED | 1'b1 | 使能 |
| enable | PFF_DISABLED | 1'b0 | 使能 |

### 功能

该部件为时序部件。

该部件内部的寄存器初值为全 0。

每个时钟上升沿，如果 `rst == 1'b0`，就令寄存器的值为全 0。否则，如果 `enable == PFF_ENABLED`，则令寄存器的值为 `i` 的值。否则寄存器的值不变。

输出端口 `o` 的值总是寄存器的值。

### 注意事项

1. 复位设成了同步复位，这是为了更好地插入气泡。

## MUX

### 功能

MUX 是多路选择器的意思，是从多个数据源中选择数据的部件。其实它也是数据通路和控制之间的接口，控制部件通过 MUX 来控制数据的流向，实现指令的功能。

### 类别

MUX 有多个类别。有 2 路 MUX、3 路 MUX 以至于多路 MUX。实际上，在单周期 CPU 中只能用到路数比较少的 MUX，多路的 MUX 要等到流水线 CPU 的时候才能用。

### 命名

由于 MUX 有多个类别，所以它也有多个 `module`，也有多个命名。*n* 路 MUX 命名为 `mux`*n*。

### 宏定义

暂无

但是仍然保留 mux.h 宏文件并填入模版，以备以后使用。

### 参数定义

| 参数 | 默认值 | 功能 |
| --- | --- | --- |
| BIT_WIDTH | 32 | 输入和输出数据的位宽 |

### 端口定义

| 端口 | 类型 | 位宽 | 功能 |
| --- | --- | --- | --- |
| control | 输入 | *见注 1* | 输入控制信号 |
| result | 输出 | BIT_WIDTH | 输出数据 |
| input*n* | 输入 | BIT_WIDTH | *见注 2* |

注：

1. 输入控制信号的位宽如下计算：有 *n* 个输入信号，就取最小的使 *2^width* 能够超过 *n* 的 *width*，这就是 control 的位宽。
2. 功能是输入端口，但是个数有 *n* 个。输入端口**从 0 开始计数**。

### 功能

若 control 的值为 *width*'d*n*，则令 result 的值为 input*n* 的值。但是若 *n* 超出了 MUX 的输入端口个数（即路数）或 *n* 为其它值，则令 result 的值为 input0 的值。

### 注意事项

1. BIT_WIDTH 默认为 32，是因为一般传送的数据都是 32 位的。
2. 接线时端口顺序**按照数据通路部分最终总结出来的接线表格中指定的顺序来！**
3. ***n* 为其他值可能是 x 或 z！**

## 流水线 CPU 数据通路

### 原理

流水线技术是通过指令级并行，缩短每级的执行时间从而提高频率的技术。这样可以让关键路径缩短，从而提升频率，因此提高了执行效率。

流水线要注意会出现冒险问题，因此会有暂停和转发机制。暂停和转发实际上是控制的内容，数据通路只需要留出需要的部件即可。

### 分析

p6 需要实现的指令为：

```
addu, subu, add, sub, sll, srl, sra, and, or, nor, xor, slt, sltu, sllv, srlv,
srav
lui, ori, addi, addiu, andi, xori, slti, sltiu
lw, lh, lhu, lb, lbu
sw, sh, sb
beq, bne, blez, bgez, bltz, bgtz
j, jal
jr, jalr
movz
mult, multu, div, divu
mfhi, mflo
mthi, mtlo
```

nop 作为 sll 指令的一种特殊情况存在。

由于不同指令的数据通路可以归类，因此首先需要对数据通路进行分类，之后再对每类数据通路总结连接。数据通路分类表如下。

| 数据通路类型 | 指令 |
| --- | --- |
| UNKNOWN | （未知指令） |
| CAL_R | addu, subu, add, sub, sll, srl, sra, and, or, nor, xor, slt, sltu, sllv, srlv, srav |
| CAL_I | lui, ori, addi, addiu, andi, xori, slti, sltiu |
| LOAD | lw, lh, lhu, lb, lbu |
| STORE | sw, sh, sb |
| BRANCH | beq, bne, blez, bgez, bltz, bgtz |
| JUMP_I | j, jal |
| JUMP_R | jr, jalr |
| CMOV | movz |
| CAL_M | mult, multu, div, divu |
| LOAD_M | mfhi, mflo |
| STORE_M | mthi, mtlo |

通过分析它们的 RTL，可以得到每条指令对应的数据通路连接如下。其中表格某一列的值表示这个输入端口是哪个输出端口的输出。端口用 流水线级：部件.端口名字 格式表示。空白的单元格表示不用关心相对应的端口的值，因为它们会被忽略，不影响指令的正常执行。未知指令只需要屏蔽各个写入的使能，这样就可以避免未知指令的影响，因此不用分析未知指令。

有时部件名称可能和级不对应。这表示相应端口的值是经过流水后的结果。

由于指令分析函数可以分析到指令读写的寄存器，因此把 D 级和 E 级的三个寄存器地址端口交给控制模块控制。这样也能避免在不该写寄存器的指令写寄存器，因为哪怕寄存器写入使能打开，要写入的寄存器也是 ZERO。

注意：**使用延迟槽来简化暂停和转发的分析。**

**F 级（IF）**

数据通路类型|F: npc.curr_pc|F: npc.alu_comp_result|F: npc.num|F: npc.jnum|F: npc.reg_|F: pc.next_pc
---|---|---|---
BRANCH|F: pc.curr_pc|E: alu.comp_result|D: im.result[15:0]|||F: npc.next_pc
JUMP_I|F: pc.curr_pc|||D: im.result[25:0]||F: npc.next_pc
JUMP_R|F: pc.curr_pc||||D: rf.read_result1|F: npc.next_pc
（其它）|F: pc.curr_pc|||||F: npc.next_pc
综合|F: pc.curr_pc|D: cmp.cmp|D: im.result[15:0]|D: im.result[25:0]|D: rf.read_result1|F: npc.next_pc

**D 级（ID）**

数据通路类型|D: ext.num|D: cmp.reg1|D: cmp.reg2
CAL_R|||
CAL_I|D: im.result[15:0]||
LOAD|D: im.result[15:0]||
STORE|||
BRANCH||D: rf.read_result1|D: rf.read_result2
NOP|||
JUMP_I|||
JUMP_R|||
CMOV||D: rf.read_result2
CAL_M|
LOAD_M|
STORE_M|
综合|D: im.result[15:0]|D: rf.read_result1|D: rf.read_result2

**E 级（EX）**

数据通路类型|E: alu.num1|E: alu.num2|E: alu.shamt|E: md.dh|E: md.dl
CAL_R|D: rf.read_result1|D: rf.read_result2|D: im.result[10:6]||
CAL_I|D: rf.read_result1|D: ext.result|||
LOAD|D: rf.read_result1|D: ext.result|||
STORE|D: rf.read_result1|D: ext.result|||
BRANCH|||||
NOP|||||
JAL|||||
JR|||||
CMOV|D: rf.read_result1|D: rf.read_result2|||
CAL_M||||D: rf.read_result1|D: rf.read_result2
LOAD_M|||||
STORE_M||||D: rf.read_result1|D: rf.read_result2
综合|D: rf.read_result1|D: rf.read_result2, D: ext.result|D: im.result[10:6]|D: rf.read_result1|D: rf.read_result2

**M 级（MEM）**

数据通路类型|M: dm.read_addr|M: dm.write_addr|M: dm.write_data
CAL_R|||
CAL_I|||
LOAD|E: alu.result|||
STORE||E: alu.result|E: rf.read_result2
BRANCH|||
NOP|||
JAL|||
JR|||
CMOV|||
CAL_M|||
LOAD_M|||
STORE_M|||
综合|E: alu.result|E: alu.result|E: rf.read_result2

**W 级（WB）**

数据通路类型|W: rf.write_data
CAL_R|E: alu.result
CAL_I|E: alu.result
LOAD|M: dm.read_result
STORE|
BRANCH|
NOP|
JAL|F: $unsigned(pc.curr_pc) + $unsigned(8)
JR|
CMOV|E: alu.result
CAL_M|
LOAD_M|E: md.out
STORE_M|
综合|E: alu.result, M: dm.read_result, F: $unsigned(pc.curr_pc) + $unsigned(8), E: md.out

**流水线寄存器**

由于流水线需要保存每一级流水线的执行结果，所以需要流水线寄存器。需要保存的执行结果，可以从上面数据通路表格中综合出来。为了方便和上面的表格对应，每一级流水线的流水线寄存器都保存上一级流水线的数据。

| 流水线级 | 信号 | 流水线寄存器名称 |
| --- | --- | --- |
| D | im.result | d_im |
| E | rf.read_result1 | e_reg1 |
| E | rf.read_result2 | e_reg2 |
| E | ext.result | e_ext |
| M | alu.result | m_alu |
| M | rf.read_result2 | m_reg2 |
| W | alu.result | w_alu |
| W | dm.read_result | w_dm |

| | | |
|---|---|---|
| W | pc.curr_pc | w_pc |
| W | md.out | w_md |

由于需要的流水线寄存器有跨级的（比如只有 D 级和 W 级），所以需要把漏掉的级补充上。

| 流水线级 | 信号 | 流水线寄存器名称 |
|---|---|---|
| D | pc.curr_pc | d_pc |
| E | pc.curr_pc | e_pc |
| M | pc.curr_pc | m_pc |

这里没有补充 D 级 BRANCH 类指令需要的 alu.comp_result 到 F 级的连接以及 JAL 和 JR 类指令相应数据到 F 级的连接，因为为了正确控制 PC 的转换，它们必须是实时的，不需要流水线寄存器。

注意：**返回 PC + 8 实际上是通过流水 PC 再加 8 实现的。**

注意：**D 级流水线寄存器都要接使能信号，E 级流水线寄存器都要接复位信号，因为要插入气泡。**

## 调试相关功能

为了能够正确地打印出写入寄存器和 dm 时需要的 pc 值，需要流水 pc.curr_pc，一直到 W 级。因此，可能需要新增流水线寄存器，并把相应的 pc 值流水。

注意：**写入寄存器是使用 w 级流水到的 pc 值。**

**数据通路 MUX**

最后是把所有可能的连接综合起来以后，得到的结果。如果有多个可能的连接，就需要一个 MUX。把 MUX 的输出端口连接在相应的输入端口上，MUX 的输入端口要保证所有可能的输入端口都能连接上。

| 端口 | 所有的信号来源 | MUX 名称 |
|---|---|---|
| E: alu.num2 | D: rf.read_result2, D: ext.result | m_alusrc |
| W: rf.write_data | （无）, E: alu.result, M: dm.read_result, D: npc.next_pc, E: md.out | m_regdata |

注意：**都是把信号来源从 0 开始编号，对应 MUX 的 input*n* 接第 *n* 个信号源。**

注意：**如果写了（无），那么相应端口的数据为全 0，不过这时相应端口实际上也没有作用。**

**转发**

需要转发是因为可能出现后面的指令需要使用前面的指令的结果，而前面的指令结果来不及写回（数据冒险）的情况。由于同一个时钟周期只有一条指令读写 dm，所以 dm 不需要转发。但是 rf 在同一个时钟周期内一般会有多条指令读写，所以 rf 需要转发。

转发的原则就是比较新的指令需要读的寄存器和比较老的指令需要写的寄存器一样。对这个条件的判断，在指令识别函数中已经有了。注意一条指令最多读 2 个寄存器，所以要判断 2 次。

转发是通过转发 MUX 来更改数据通路上寄存器的值，从而达到提前更新的目的。首先，数据通路上有寄存器值的地方，一共有五处：D: rf.read_result1, D: rf.read_result2, E: rf.read_result1, E: rf.read_result2, M: dm.write_data。其中 E 级的两处是通过流水线寄存器暂存的。这五处可以分三类。对每类需要构造的转发 MUX 总结如下。

**MUX**

| 端口 | 所有的信号来源 | 名称 |
| --- | --- | --- |
| D: rf.read_result[12] | E: rf.read_result1, E: npc.next_pc, M: npc.next_pc, M: alu.result, W: rf.write_data, M: md.out | fm_d1 |
| E: rf.read_result[12] | M: npc.next_pc, M: alu.result, W: rf.write_data, M: md.out | fm_e1 |
| M: dm.write_data | W: rf.write_data | fm_m |

**注意：不能在 M 级设置 MUX 转发 dm 的数据，因为这样 D 级或 E 级会等待 M 级 dm 的数据，关键路径会变得非常长，极大地降低流水线性能。同样地，也不能在 E 级设置 MUX 转发 alu 的数据。**

转发 MUX 最终是由控制模块控制的。但是控制模块也没法克服有些数据通路不能转发的现实（比如 M: dm.read_result）。这就需要——

#### 暂停

需要暂停是因为有些数据冒险靠转发解决不了，必须要让后面的指令暂停一个时钟周期。暂停的方式是在流水线中插入一个 NOP（这时候也叫气泡），从而让发生数据冒险的指令能够转发。

流水线 CPU 数据通路中能提供的暂停机制有锁定 pc 和清空 E 级各个流水线寄存器。这样就可以在流水线 E 级插入气泡。清空 E 级各个寄存器是通过流水线寄存器的同步复位功能实现的。

# 指令识别机制

## 原理

指令识别机制是为了判断指令的功能而设计的。它可以实现判断指令的具体类型、数据通路类型、需要的控制信号等功能。用这些函数识别出来的数据，就可以判断指令的数据流、转发和暂停相关信息和异常相关信息等。

#### 宏定义

由于有特殊的指令读写固定的寄存器，所以寄存器号也要宏定义。

由于函数的声明需要一定的范式保证健壮性，所以函数的声明本身也要定义。

用把宏定义成宏的方法，定义表中值为宏的宏。值为宏的宏的意义，与定义它的宏一样，在表中省略。

| 类别 | 定义 | 值 | 意义 |
| --- | --- | --- | --- |
| 寄存器号 | ZERO | 5'd0 | 0 号寄存器（或者表示某指令在某函数下对应的寄存器不存在） |
| 寄存器号 | NULL | ZERO | |
| 寄存器号 | RA | 5'd31 | 31 号寄存器（$ra，jal 指令要写入） |
| 函数声明 | ROBUST_FUNCTION | function automatic | automatic 保证函数同时调用时一定使用不同的硬件块 |

## 端口定义

端口 | 类型 | 位宽 | 功能
--- | --- | ---
instr | 输入 | 32 | 要分析的指令
kind | 输出 | 9 | 当前指令的具体类型

**功能**

获取当前指令的具体类型。返回的结果一共 9 位，前 4 位是数据通路类型，后 5 位是具体类型。

若指令的格式符合 MIPS 指令集中的相应格式，则返回对应指令的代码（在宏定义一节中描述）。否则，返回 0。

**宏定义**

用把宏定义成宏的方法，定义表中值为宏的宏。值为宏的宏的意义，与定义它的宏一样，在表中省略。

编码方式为前 4 位为数据通路类型，后 5 位为其下的具体类型。

指令的意义在表示相应指令的情况下省略不写。但如果有相应备注，也会在这栏注明。

##### 指令字段

| 类别 | 定义 | 值 | 意义 |
| --- | --- | --- | --- |
| 指令字段 | OP(x) | (x[31:26]) | 指令的 op 字段 |
| 指令字段 | RS(x) | (x[25:21]) | 指令的 rs 字段 |
| 指令字段 | RT(x) | (x[20:16]) | 指令的 rt 字段 |
| 指令字段 | RD(x) | (x[15:11]) | 指令的 rd 字段 |
| 指令字段 | SHAMT(x) | (x[10:6]) | 指令的 shamt 字段 |
| 指令字段 | FUNCT(x) | (x[5:0]) | 指令的 funct 字段 |
| 指令字段 | IMM(x) | (x[15:0]) | 指令的 imm 字段 |
| 指令字段 | IMM_J(x) | (x[25:0]) | j 指令的 imm 字段 |

##### 指令类型

| 类别 | 定义 | 值 | 意义 |
| --- | --- | --- | --- |
| 指令类型 | UNKNOWN | 9'b0000_00000 | 未知指令 |
| 指令类型 | UNK | UNKNOWN | |
| 指令类型 | ADDU | 9'b0001_00000 | |
| 指令类型 | SUBU | 9'b0001_00001 | |
| 指令类型 | ADD | 9'b0001_00010 | |
| 指令类型 | SUB | 9'b0001_00011 | |
| 指令类型 | SLL | 9'b0001_00100 | |
| 指令类型 | SRL | 9'b0001_00101 | |
| 指令类型 | SRA | 9'b0001_00110 | |
| 指令类型 | AND | 9'b0001_00111 | |
| 指令类型 | OR | 9'b0001_01000 | |
| 指令类型 | NOR | 9'b0001_01001 | |
| 指令类型 | XOR | 9'b0001_01010 | |

```
指令类型 SLT    9'b0001_01011
指令类型 SLTU   9'b0001_01100
指令类型 SLLV   9'b0001_01101
指令类型 SRLV   9'b0001_01110
指令类型 SRAV   9'b0001_01111
指令类型 LUI    9'b0010_00000
指令类型 ORI    9'b0010_00001
指令类型 ADDI   9'b0010_00010
指令类型 ADDIU  9'b0010_00011
指令类型 ANDI   9'b0010_00100
指令类型 XORI   9'b0010_00101
指令类型 SLTI   9'b0010_00110
指令类型 SLTIU  9'b0010_00111
指令类型 LW     9'b0011_00000
指令类型 LH     9'b0011_00001
指令类型 LHU    9'b0011_00010
指令类型 LB     9'b0011_00011
指令类型 LBU    9'b0011_00100
指令类型 SW     9'b0100_00000
指令类型 SH     9'b0100_00001
指令类型 SB     9'b0100_00010
指令类型 BEQ    9'b0101_00000
指令类型 BNE    9'b0101_00001
指令类型 BLEZ   9'b0101_00010
指令类型 BGEZ   9'b0101_00011
指令类型 BLTZ   9'b0101_00100
指令类型 BGTZ   9'b0101_00101
指令类型 J      9'b0110_00000
指令类型 JAL    9'b0110_00001
指令类型 JR     9'b0111_00000
指令类型 JALR   9'b0111_00001
指令类型 MOVZ   9'b1000_00000
指令类型 MULT   9'b1001_00000
指令类型 MULTU  9'b1001_00001
指令类型 DIV    9'b1001_00010
指令类型 DIVU   9'b1001_00011
指令类型 MFHI   9'b1010_00000
指令类型 MFLO   9'b1010_00001
指令类型 MTHI   9'b1011_00000
指令类型 MTLO   9'b1011_00001
```

**注意**

1. 临时的数据通路类型都是从上往下长的。

# 控制

## 原理

控制是指通过识别指令，控制数据的流通，从而让 CPU 执行指定的计算的过程。数据通路只是得到了数据可能的流向，真正要控制还是控制完成。控制通过已有的控制信号和数据通路的分叉完成控制。

在流水线 CPU 中，由于存在结构冒险和数据冒险，所以需要通过暂停和转发解决。暂停控制和转发控制可以放在单独的控制模块中，从而不影响原来单周期时的控制模块。但是，也可以通过改造控制模块的方式集成暂停和转发功能。通过指令识别系列函数（实际上综合时也会被综合成电路），可以分析指令，做到有效的暂停和转发。

## 端口定义

| 端口 | 类型 | 位宽 | 功能 |
|---|---|---|---|
| clk | 输入 | 1 | 时钟信号 |
| d_instr | 输入 | 32 | 当前在 D 级（ID）的指令 |
| rf_read_result2 | 输入 | 32 | rf 的 2 号读取结果 |
| cw_f_pc_enable | 输出 | 1 | 控制 pc 使能 |
| cw_d_pff_enable | 输出 | 1 | 控制 D 级流水线寄存器使能 |
| cw_e_pff_rst | 输出 | 1 | 控制 E 级流水线寄存器复位 |
| cw_f_npc_jump_mode | 输出 | 4 | 控制 npc 的跳转模式 |
| cw_d_ext_mode | 输出 | 3 | 控制 D: ext.mode |
| cw_d_rf_read_addr1 | 输出 | 5 | 控制 D: rf.read_addr1 |
| cw_d_rf_read_addr2 | 输出 | 5 | 控制 D: rf.read_addr2 |
| cw_e_m_alusrc | 输出 | 1 | 控制 E: m_alu_num2 |
| cw_e_alu_op | 输出 | 5 | 控制 E: alu.op |
| cw_e_md_op | 输出 | 3 | 控制 E: md.op |
| cw_m_hilo | 输出 | 1 | 控制 E: m_hilo |
| cw_m_dm_write_enable | 输出 | 1 | 控制 M: dm.write_enable |
| cw_m_dm_mode | 输出 | 1 | 控制 M: dm.mode |
| cw_w_rf_write_enable | 输出 | 1 | 控制 W: rf.write_enable |
| cw_w_m_regdata | 输出 | 3 | 控制 W: m_rf_write_data |
| cw_w_rf_write_addr | 输出 | 5 | 控制 W: rf.write_addr |
| cw_fm_d[12] | 输出 | 3 | 控制 fm_d[12] |
| cw_fm_e[12] | 输出 | 3 | 控制 fm_e[12] |
| cw_fm_m | 输出 | 3 | 控制 fm_w |

## 总体结构

控制模块是时序部件。不设置成组合逻辑部件的原因如下。

1. 哪怕控制本身不设置成时序部件，也需要流水控制信号，这是流水线 CPU 结构上的需要。
2. 控制本身是时序部件，就可以流水更多的信息。最明显的就是指令读写寄存器的信息。比如暴力转发也把指令读写寄存器的信息放在流水线中流水。

3. 保留单周期处理器的控制机制实际上过渡不是那么平滑，因为还有多周期处理器，它的控制是类似状态机的结构。

控制模块在内部流水指令，从而做到比较有效的控制信号发射和数据冒险分析。负责控制信号发射的部分是纯组合逻辑，用函数实现。

同时，控制模块也在内部流水指令需要读取和写入的三个寄存器。因为流水线 CPU 和单周期 CPU 逻辑上应该一样，所以一条指令需要读取和写入的三个寄存器可以直接判断出来，并且流水。这样也可以更方便地处理数据冒险。

## 数据通路和功能控制信号

由于指令的数据通路可以分成几个类型，每种类型中需要的数据通路是一样的，只是某些控制信号不同。而且，流水线是分级的，所以每级控制数据通路形状的信号可以单独列表。

但是，不同的具体指令对不同部件的某些具体操作不同。比如 CAL_R 类指令对 ALU 的具体操作就不同。因此，对这些控制具体操作的信号，需要单独列表。

通过对数据通路形状的分析，可以得到每种数据通路类型需要的控制信号如下。其中表格某一单元格的值有两种情况：若该单元格所在的行最左边的单元格是 MUX，则说明对应的指令需要让该 MUX 的输入端口接入该单元格表示的端口；若该单元格所在的行最左边的单元格是端口，则说明对应的指令需要的控制信号为该单元格表示的控制信号。

若单元格以 # 开头，则说明该控制信号或端口只是为了使控制单元功能明晰而加上的，实际上并不需要关心该控制信号或要接入的端口的值。如果想理解该单元格的值，去掉 # 再按照上一段理解即可。

**F 级**

| **数据通路类型** | **F: npc.jump_mode** |
| --- | --- |
| BRANCH | 视具体指令而定 |
| JUMP_I | NPC_J |
| JUMP_R | NPC_REG |
| （其它） | NPC_JUMP_DISABLED |

BRANCH 类指令类型与 F: npc.jump_mode 的关系：

| **指令类型** | **F: npc.jump_mode** |
| --- | --- |
| BEQ | NPC_EQUAL |
| BNE | NPC_NOT_EQUAL |
| BLEZ | NPC_SIG_SMALLER_OR_EQUAL |
| BGEZ | NPC_SIG_LARGER_OR_EQUAL |
| BLTZ | NPC_SIG_SMALLER |
| BGTZ | NPC_SIG_LARGER |

注意：**F 级的控制信号是由 D 级指令控制的。**

注意：**BRANCH 类指令要跟 0 比较的那些指令，是通过读 $0 比较的，所以能直接进行大小比较。**

**D 级（ID）**

| **数据通路类型** | **D: ext.mode** |
| --- | --- |

CAL_I　　视具体指令而定
LOAD　　EXT_MODE_SIGNED
STORE　　EXT_MODE_SIGNED
（其它）　　#EXT_MODE_SIGNED

CAL_I 类指令类型与 D: ext.mode 的关系：

**指令类型 D: ext.mode**
LUI　EXT_PAD
ORI　EXT_UNSIGNED
ADDI　EXT_SIGNED
ADDIU　EXT_SIGNED
ANDI　EXT_UNSIGNED
XORI　EXT_UNSIGNED
SLTI　EXT_SIGNED
SLTIU　EXT_SIGNED

注意：**SLTIU 扩展立即数的时候确实是按照有符号扩展的，但比较是按照无符号数比较，可以查指令手册。**

**E 级（EX）**

数据通路类型|E: m_alusrc|E: alu.op|E: md.op|E: m_hilo
---|---|---|---
CAL_R|D: rf.read_result2|视具体指令而定|MD_NONE|#E: md.hi
CAL_I|D: ext.result|视具体指令而定|MD_NONE|#E: md.hi
LOAD|D: ext.result|ALU_ADD|MD_NONE|#E: md.hi
STORE|D: ext.result|ALU_ADD|MD_NONE|#E: md.hi
BRANCH|D: rf.read_result2|#ALU_OR|MD_NONE|#E: md.hi
CMOV|D: rf.read_result2|视具体指令而定|MD_NONE|#E: md.hi
CAL_M|#D: rf.read_result2|#ALU_OR|视具体指令而定|#E: md.hi
LOAD_M|#D: rf.read_result2|#ALU_OR|视具体指令而定|视具体指令而定
STORE_M|#D: rf.read_result2|#ALU_OR|视具体指令而定|#E: md.hi
（其它）|#D: rf.read_result2|#ALU_OR|MD_NONE|#E: md.hi

CAL_R 类指令类型与 E: alu.op 的关系：

**指令类型 E: alu.op**
ADDU　ALU_ADD
SUBU　ALU_SUB
ADD　ALU_ADD
SUB　ALU_SUB
AND　ALU_AND
OR　ALU_OR
NOR　ALU_NOR
XOR　ALU_XOR
SLT　ALU_SLT
SLTU　ALU_SLTU

SLL ALU_SLL
SRL ALU_SRL
SRA ALU_SRA
SLLV ALU_SLLV
SRLV ALU_SRLV
SRAV ALU_SRAV

CAL_I 类指令类型与 E: alu.op 的关系:

**指令类型 E: alu.op**
LUI ALU_OR
ORI ALU_OR
ADDI ALU_ADD
ADDIU ALU_ADD
ANDI ALU_AND
XORI ALU_XOR
SLTI ALU_SLT
SLTIU ALU_SLTU

CMOV 类指令类型与 E: alu.op 的关系:

**指令类型 E: alu.op**
MOVZ ALU_MOVZ

CAL_M 类指令类型与 E: md.op 的关系:

**指令类型 E: md.op**
MULT MD_MULT
MULTU MD_MULTU
DIV MD_DIV
DIVU MD_DIVU

LOAD_M 类指令类型与 E: md.op 的关系:

**指令类型 E: md.op**
MFHI MD_MFHI
MFLO MD_MFLO

STORE_M 类指令类型与 E: md.op 的关系:

**指令类型 E: md.op**
MTHI MD_MTHI
MTLO MD_MTLO

LOAD_M 类指令类型与 E: m_hilo 的关系:

**指令类型 E: m_hilo**
MFHI E: md.hi

MFLO  E: md.lo

**M 级（MEM）**

| 数据通路类型 | M: dm.write_enable | M: dm.mode |
| --- | --- | --- |
| LOAD | 1'b0 | 视具体指令而定 |
| STORE | 1'b1 | 视具体指令而定 |
| （其它） | 1'b0 | DM_NONE |

LOAD 类指令类型与 M: dm.mode 的关系：

| 指令类型 | M: dm.mode |
| --- | --- |
| LW | DM_W |
| LH | DM_H |
| LHU | DM_HU |
| LB | DM_B |
| LBU | DM_BU |

STORE 类指令类型与 M: dm.mode 的关系：

| 指令类型 | M: dm.mode |
| --- | --- |
| SW | DM_W |
| SH | DM_H |
| SB | DM_B |

**W 级（WB）**

| 数据通路类型 | W: rf.write_enable | W: m_regdata |
| --- | --- | --- |
| CAL_R | 1'b1 | E: alu.result |
| CAL_I | 1'b1 | E: alu.result |
| LOAD | 1'b1 | E: dm.read_result |
| JUMP_I | 1'b1 | D: npc.next_pc |
| JUMP_R | 1'b1 | D: npc.next_pc |
| CMOV | 1'b1 | E: alu.result |
| LOAD_M | 1'b1 | E: md.out |
| （其它） | 1'b0 | #E: alu.result |

**流水的内容**

流水 E 级、M 级、W 级指令及其要读的两个寄存器和要写的一个寄存器。不流水 D 级指令是为了配合暂停机制，D 级一被暂停，D 级指令只在组合逻辑跟着变化，不需要再在控制模块里改变 D 级指令的值。

**指令读写寄存器识别**

比较显然的一点是数据通路类型决定指令要读写的寄存器号。所以，可以直接用取指令字段的宏来完成。

数据通路类型和指令读写寄存器的关系如下。如果指令不读写哪个寄存器，就用 ZERO 替换，因为 ZERO 不参与转发。这样，对转发正确性也没有影响。其中使用的获取指令字段的宏隐含着用要分析的指令作为参数。

| 数据通路类型 | reg1 | reg2 | regw |
| --- | --- | --- | --- |
| CAL_R | RS | RT | RD |
| CAL_I | RS | RT | RT |
| LOAD | RS | ZERO | RT |
| STORE | RS | RT | ZERO |
| BRANCH | RS | 视指令类型而定（[4]） | ZERO |
| JUMP_I | ZERO | ZERO | 视指令而定（[2]） |
| JUMP_R | RS | ZERO | 视指令而定（[3]） |
| CMOV | RS | RT | 视寄存器值而定（[1]） |
| CAL_M | RS | RT | ZERO |
| LOAD_M | ZERO | ZERO | RD |
| STORE_M | RS | ZERO | ZERO |
| （其它） | ZERO | ZERO | ZERO |

注：

1. 有一点就是 CMOV 类指令。这类指令的一种实现是无条件把要写入的数据看成是 `$rs` 的值，但是**改变要写入的寄存器号**。如果 `$rt == 32'b0`，就写入 `$rd`，否则写入 `$0 / ZERO`。这样，加上把要读写的寄存器号流水的机制，能保证 CMOV 类指令的数据冒险处理不出错。哪怕在 W 级打开了 rf 的写使能，写入 `$0` 也没有影响。
2. JUMP 类指令若为 `jal`，则 `regw == RA`。若为 `j`，则 `regw == ZERO`。
3. JUMP_R 类指令若为 `jr`，则 `regw == ZERO`。若为 `jalr`，则 `regw == RD`。由于 `jr` 指令 RS 字段永远为 0，所以这样分析是正确的。
4. BRANCH 类指令若为 beq 或 bne，则 `reg2 == RT`。若为 `blez, bgez, bltz, bgtz`，则 `reg2 == ZERO`。由于这样会让 cmp 的比较结果变成对应寄存器与 0 的比较，符合指令功能描述，所以这样分析是正确的。

## 转发控制信号

由于流水线 CPU 中存在数据冒险，所以需要转发。由于有了指令识别函数，所以转发是非常抽象的，只需要判断涉及的寄存器号。而且只有两个级是转发的接收端（数据的需求者），因此可以在某一级的角度，一级一级往后排查。

注意：**先检查较新级的数据冒险，再检查较老级的，因为 rf 中的内容最终还是较新级的。**

对 D 级，先检查 E 级，再检查 M 级，再检查 W 级。对 E 级的寄存器，先检查 M 级，再检查 W 级。这样就能保证转发的完整性。

转发控制信号最终需要控制的是转发 MUX，因此转发 MUX 也要进行定义。

下表是所有转发的情况和具体的描述。意义中说的数据通路类型，都是源指令的数据通路类型。

| 类别 | 定义 | 值 | 意义 |
| --- | --- | --- | --- |
| 所有转发 MUX | orig | 0 | 不转发，保持原样 |
| fm_d[12] | E2D_rf | 1 | E 级到 D 级，数据通路类型是 CMOV，要写入的数据在 D 级产生好了，到了 E 级才能转发 |
| fm_d[12] | E2D_npc | 2 | E 级到 D 级，数据通路类型是 JUMP_I / JUMP_R，要写入的数据在 D 级产生好了，到了 E 级才能转发 |

| | | | |
|---|---|---|---|
| fm_d[12] | M2D_npc | 3 | M 级到 D 级，之后同上 |
| fm_d[12] | M2D_alu | 4 | M 级到 D 级，数据通路类型是 CAL_R / CAL_I / CMOV，数据在 D 级或 E 级产生好了，但对 CAL_R / CAL_I 来说，到了 M 级才能转发 |
| fm_d[12] | W2D_rf | 5 | W 级到 D 级，数据通路类型是所有能够写入寄存器的类型，数据在 W 级都可以转发了 |
| fm_d[12] | M2D_md | 6 | M 级到 D 级，数据通路类型是 LOAD_M，数据在 E 级产生好了 |
| fm_d[12] | E2D_md | 7 | E 级到 D 级，数据通路类型是 LOAD_M，数据在 E 级产生好了 |
| fm_e[12] | M2E_npc | 1 | M 级到 E 级，数据通路类型是 JUMP_I / JUMP_R，要写入的数据在 D 级产生好了，到了 E 级才能转发 |
| fm_e[12] | M2E_alu | 2 | M 级到 E 级，数据通路类型是 CAL_R / CAL_I / CMOV，数据在 D 级或 E 级产生好了，但对 CAL_R / CAL_I 来说，到了 M 级才能转发 |
| fm_e[12] | W2E_rf | 3 | W 级到 D 级，数据通路类型是所有能够写入寄存器的类型，数据在 W 级都可以转发了 |
| fm_e[12] | M2E_md | 4 | M 级到 D 级，数据通路类型为 LOAD_M，数据在 E 级产生好了 |
| fm_m | W2M_rf | 1 | W 级到 M 级，数据通路类型是所有能够写入寄存器的类型，数据在 W 级都可以转发了（比如 sw 指令转发 rf 内容） |

注意：**B2A_.* 表示 B 级从 A 级转发。**

注意：**宏的值要和对应转发 MUX 的接线顺序相符。**

注意：**E2D_rf 表示把 E 级的第一个寄存器转发出去，因为用到这条指令的是 CMOV 类指令。**

注意：**fm_m 检查的是要读取的第二个寄存器，因为现在用到的所有写入内存的指令，要写入内存的数据都与相应指令第二个寄存器的读取结果对应。以后甚至可能加上检查要读取的第一个寄存器，不过就要根据指令类型判断了。**

## 暂停控制信号

由于流水线中有些数据冒险通过转发解决不了，所以需要暂停机制。暂停机制的前提是产生数据冒险。暂停机制是通过 Tuse 和 Tnew 机制实现的。

Tuse 是指指令到 D 级以后还剩最晚多少时间就需要新值。Tnew 是指指令还需要多长时间才能开始转发。因此只要 Tuse < Tnew，就需要暂停，因为在流水线中如果没有暂停，两条指令的相对位置是不变的，如果不暂停，就不能解决数据冒险。

数据冒险可以只在 D 级检测和在 E 级解决，因为在 E 级插入气泡，就可以保证 Tuse 和 Tnew 最终回回归正常。

插入气泡是通过锁定 pc 和清空 E 级各个流水线寄存器实现的。但是，控制内部的流水线也要插入气泡。

暂停要分两个寄存器，因为数据冒险也是要分成两个寄存器的情况的。

注意：**Tnew 的计算是要看能够开始转发的时间，而不是生成好要转发数据的时间，因为不是所有转发路径都是可能的。**

注意：**控制内部的流水线也要插入气泡。**

在 D 级各种数据通路类型的 Tuse 如下。

**数据通路类型 Tuse (read_addr1) Tuse (read_addr2)**
UNKNOWN

| | | |
|---|---|---|
| CAL_R | 1 | 1 |
| CAL_I | 1 | 1 |
| LOAD | 1 | |
| STORE | 1 | 2 |
| BRANCH | 0 | 0 |
| JUMP_I | | |
| JUMP_R | 0 | |
| CMOV | 0 | 0 |
| CAL_M | 1 | 1 |
| LOAD_M | | |
| STORE_M | 1 | 1 |

在 E 级和 M 级各种数据通路类型的 Tnew 如下。忽略 W 级，因为所有指令到 W 级时都可以马上转发数据。

| 数据通路类型 | Tnew (E) | Tnew (M) |
|---|---|---|
| UNKNOWN | | |
| CAL_R | 1 | 0 |
| CAL_I | 1 | 0 |
| LOAD | 2 | 1 |
| STORE | | |
| BRANCH | | |
| JUMP_I | 0 | 0 |
| JUMP_R | 0 | 0 |
| CMOV | 0 | 0 |
| NOP | | |
| CAL_M | | |
| LOAD_M | 0 | 0 |
| STORE_M | | |

以上列表中 Tuse 没有列出的，是因为它没有意义，认为 Tuse 足够大。Tnew 同理，认为 Tnew 为 0。

这样，只要算出每个阶段的 Tuse 和 Tnew，并且保证发生数据冒险时对两个寄存器，Tuse >= Tnew，就能控制暂停和转发。当且仅当 `t_use_reg[12]` 小于 `t_new_[em]` 中的任何一个时，需要暂停。

`md.busy == 1'b1` 时，会一直插入气泡，直到 `md.busy == 1'b0`。而且，CAL_M 类指令虽然进行计算，但不写普通寄存器，所以跟其它指令没有转发解决不了的数据冒险，所以不停地插入气泡这种方式是可以解决数据冒险的。而且，跟 dm 类似，md 的 HI 和 LO 寄存器也没有数据冒险。因此，乘除法相关指令和其它指令之间，可以看成解决了需要暂停的问题，虽然 md 需要多个周期运行。

注意：**比较 Tuse 和 Tnew 应该用无符号比较，避免数值最高位是 1 时被看成负数。**

### 寄存器地址控制信号

由于已经有指令识别机制了，所以寄存器的地址控制可以简化。只需要在 D 级和 M 级的三个地址端口输入指令识别机制相应的结果即可。

## CPU

## 原理

CPU 是宏观部件，主要连接起数据通路和控制。该部件主要起的是宏观功能，也就是读取指令并完成计算。

CPU 在模块结构中作为顶层模块而存在。

## 端口定义

| 端口 | 类型 | 位宽 | 功能 |
| --- | --- | --- | --- |
| `clk` | 输入 | 1 | 时钟信号 |

## 接线

按照数据通路和控制部分的定义进行接线。数据通路中的接线方式在数据通路部分的文档中描述，控制部分按照控制部分的文档中描述。控制部分控制数据通路的哪部分，在控制部分的文档中。

## 功能

CPU 需要的外部数据输入是极少的，只有时钟信号、必要的其它信号和指令文件。

## 注意事项

1. 对部件分级是个好习惯，在流水线 CPU 时会有用。
2. TODO: `input rst`?