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
epc	输入	32	输入的 EPC 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
			当输入的比较结果为小于

jump_mode NPC_SMALLER	4'b0100	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
jump_mode NPC_ISR	4'b1101	跳转到固定地址 NPC_ISR_ADDR
jump_mode NPC_EPC	4'b1100	跳转到 epc

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 == NPC_ISR，则令 next_pc = IM_ISR_START_ADDRESS。

若 jump_mode == NPC_EPC，则令 next_pc = epc。

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
rst	输入	1	复位信号
next_pc	输入	32	NPC 计算得来的下一个 PC 地址
enable	输入	1	PC 使能
invalid	输出	1	不对齐的 PC 值
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_ADDR	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_ADDR。

在每个时钟上升沿，若 `rst == 1'b1`，则把 PC 部件中保存的当前 PC 的值更新成 PC_START_ADDR。否则，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，但是把最低两位无条件清零。若 PC 最低两位不为 0 或 PC 作为无符号数超出 `im` 中指定的上下界，则 `invalid == 1'b1`。

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。
2. **比较 PC 是无符号数比较**

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
addr	IM_ISR_START_ADDRESS	32	IM 对外表现的 ISR 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.hex"	要加载的机器码
指令存储器	IM_ISR_FILENAME	"code_handler.hex"	要加载的 ISR 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。im 同样存储 ISR，也是用加载文件的系统任务加载，加载文件名由 IM_ISR_FILENAME 指定，加载地址由 ($unsigned(IM_ISR_START_ADDRESS) - $unsigned(IM_START_ADDRESS)) >> $unsigned(2) 指定，一共加载 2047 个字。

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
4. 比较和移位都是无符号数操作，无符号数操作能保证算术移位

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
`rst`	输入	1	同步复位信号
`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 `rst == 1'b1`，则把所有 31 个寄存器的初值都设为 `32'b0`。否则，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。

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
sig_overflow	输出	1	计算过程中是否发生补码溢出

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 或 sig_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。

检查无符号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。

检查有符号溢出的方式是利用另一个的中间变量，它是 33 位的。在加减法时用同样的方法算出该中间变量的值，但是在算以前进行符号扩展，算的时候把这两个数作为有符号数计算。对其它运算，它置为 33'b0。如果它的符号位与 result 的符号位相同，就说明没有溢出，否则有溢出。这是因为有符号加减法溢出都是上溢或者下溢，符号是一定要变化的。而把它们有符号扩展到 33 位再计算，计算结果一定不会溢出。把正确的结果的符号位和实际结果的符号位一比较，就可以知道是否发生溢出了。

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
undo	输出	1	根据当前操作恢复 HI 和 LO 寄存器的值或者停止运算
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`。其它寄存器有 `hi_reg`, `lo_reg`, `old_hi`, `old_lo`, `dh_i`, `dl_i`, `op_i`, `on_rst`。

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 `rst == 1'b1`，则把所有寄存器置为 0，但 `on_rst <= 1'b1`，这是为了屏蔽下一个下降沿的读取 `op` 的操作。否则，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，`hi_reg <= old_hi, lo_reg <= old_lo`。否则，什么也不做。

每个时钟下降沿，若 `on_rst == 1'b1`，则 `on_rst <= 1'b0`，剩下的什么也不做。否则，若 `op == MD_NONE || stop == 1'b1 || restore == 1'b1`，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，op_i <= op，old_hi <= hi_reg; old_lo <= lo_reg。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读取的值。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。

注意事项

1. **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，则不生成新的结果，hi 和 lo 保持原来的结果，这是为了跟 MARS 保持兼容，但是要覆盖保存的旧的 hi 和 lo 寄存器的值为原来的结果，因为这样跟进入 ISR 有关的复位操作在机制不变的情况下逻辑上才能说得过去**
2. **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。**
3. **restore / stop 信号只是名字上有指示，实际上如果出现了这种信号，下一个时钟下降沿就不会更新了，也不需要在下一个时钟上升沿复位**
4. **若 E 级指令为对应的指令，下降沿不会执行 mthi / mtlo / mfhi / mflo，不过进 ISR 时会在下个上升沿恢复；但是若 E 级指令为其它指令，不会做任何撤销操作。这是因为乘除法会在后台进行，若 ISR 中需要乘除法指令则符合逻辑，否则也不影响正常计算，同样符合逻辑。即使 ISR 中做了乘除法计算出 ISR 后再用，需要计算结果的时候暂停机制也能保证暂停，也是符合逻辑的。即使 ISR 中需要乘除法指令，也不会轻易覆盖原来正在执行的乘除法指令的内容，因为还有暂停机制**
5. **不用担心取消了对 ctr 的判断以后会出现操作做到一半被覆盖的问题，因为有 busy 信号撑着，因为暂停机制，下一个能操作 md 的指令进不到 E 级来**
6. **加入 on_rst 寄存器就是为了保证时钟上升沿来的 rst 信号的效果，能持续到下一个时钟上升沿，毕竟下一个时钟下降沿要读 op 然后做对应的操作**

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
rst	输入	1	同步复位信号
curr_pc	输入	32	当前 PC 值
addr	输入	32	读写的地址
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
地址范围	DM_ADDR_UB	32	dm 的地址上界
地址范围	DM_ADDR_LB	32	dm 的地址下界

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 而且 $unsigned(op_addr) >= DM_ADDR_LB && $unsigned(op_addr) <= DM_ADDR_UB，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

5. TODO: `rst`

设备桥

原理

设备桥是通过把设备的寄存器地址映射成数据内存地址，从而做到读写设备的寄存器，进而做到对设备的控制的。设备桥通过 MMIO，把多个设备抽象成一个设备，从而避免了 CPU 识别设备的进一步开销。

设备桥对每个设备都需要知道它的寄存器特征和地址范围，这样才能做到有效地对设备的寄存器进行操作。地址范围不能跟数据内存的地址范围重叠，否则会造成冲突，而且无法对地址进行有效的抽象。

端口定义

端口	类型	位宽	功能
`clk`	输入	1	时钟信号
`rst`	输入	1	复位信号
`addr`	输入	32	地址信号
`write_enable`	输入	1	CPU 决定的设备写使能信号
`write_data`	输入	32	CPU 要写入的数据
`read_result`	输出	32	CPU 读取的数据
`hwirq`	输出	6	硬件中断请求输出

宏定义

类别	定义	值	意义
设备地址	`BRIDGE_TIMER_LB`	`32'h00007f00`	`timer` 的地址下界
设备地址	`BRIDGE_TIMER_UB`	`32'h00007f0b`	`timer` 的地址上界
设备地址	`BRIDGE_UART_LB`	`32'h00007f10`	`uart` 的地址下界
设备地址	`BRIDGE_UART_UB`	`32'h00007f2b`	`uart` 的地址上界
设备地址	`BRIDGE_SWITCHES_LB`	`32'h00007f2c`	`switches` 的地址下界
设备地址	`BRIDGE_SWITCHES_UB`	`32'h00007f33`	`switches` 的地址上界
设备地址	`BRIDGE_LED_LB`	`32'h00007f34`	`led` 的地址下界
设备地址	`BRIDGE_LED_UB`	`32'h00007f37`	`led` 的地址上界
设备地址	`BRIDGE_NIXIE_LB`	`32'h00007f38`	`nixie` 的地址下界
设备地址	`BRIDGE_NIXIE_UB`	`32'h00007f3f`	`nixie` 的地址上界
设备地址	`BRIDGE_BUTTONS_LB`	`32'h00007f40`	`buttons` 的地址下界
设备地址	`BRIDGE_BUTTONS_UB`	`32'h00007f43`	`buttons` 的地址上界

功能

如果地址落在某个设备的地址范围内，就令这个设备的写使能与 `write_enable` 相同，这个设备的 `read_result` 为要输出的值。其余的设备屏蔽写使能。否则，`read_result` 为 `32'b0`。

无论什么时候，`clk` 和 `rst` 的值都接入每个设备。这是为了方便整体时钟信号同步和复位。`hwirq` 的最低位为 `timer` 的 `irq`，倒数第二低位为 `uart` 的 `IRQ_data_complete`。

注意事项

1. **15 号设备（curr_dev 表示的）代表没有设备，14 号设备代表 dm**
2. **地址比较用无符号数比较**
3. **地址上下界以字节为单位，而且是包括边界的**

地址检查器

原理

因为 p7 中引入了异常和中断，所以需要有一种检查地址的机制。但是，p7 里也引入了对设备的读写，而且和对 dm 的读写多少是统一的。所以，为了把对读写地址和模式的检查抽象出来，可以用地址检查器来实现。

端口定义

端口	类型	位宽	功能
addr	输入	32	当前读写 dm 或 bridge 的地址
dm_mode	输入	3	当前 dm 的模式
validity	输出	3	当前地址在当前模式下的有效性

宏定义

地址检查器也使用 dm 和 bridge 的宏。

类别	定义	值	意义
地址有效性	AC_VALID	3'd0	地址有效
地址有效性	AC_BAD	3'd1	读取时地址无效

功能

若当前地址超出范围或者在当前模式下不对齐或者当前模式不适合对应的设备或者当前地址不可写，则判定为无效地址。范围按照 dm.h 和 bridge.h 中确定的值来确定。若 write_enable == 1'b1，则属于写入时地址无效，否则属于读取时地址无效。当前地址既不超出范围又不在当前模式下不对齐时，地址有效。按照地址有效性的各种意义在 validity 端口输出相应的值。

各个小条件的判断逻辑首先假定模式不是 DM_NONE，之后判断 invalid 的时候会统一判断。

注意

1. **地址范围要和 bridge 同步**
2. **如果当前地址是设备地址，则锁定模式为按字读写**
3. **地址要作为无符号数比较**
4. **如果当前模式是 DM_NONE，就不存在超出范围的问题了，所以最后要加上钩子**
5. **不检查 write_enable 的原因是 ac 是组合逻辑，如果检查出错，write_enable 会被禁用，因此只能给 control 查**
6. **对设备读写不对齐是不能被原谅的，往不能写的对应着设备的地址范围写不行，读写模式不是字也不行**
7. **忽略判断模式是 DM_NONE 是为了避免各个小判断的逻辑复杂化，从而出错**

CP0

原理

CP0 是 0 号协处理器的意思，它负责异常和中断的处理，也负责保存和恢复出现异常时的 PC。

为了能够更方便地处理异常及其相关转发，CP0 部署在 M 级。

端口定义

端口	类型	位宽	功能
clk	输入	1	时钟信号
rst	输入	1	同步复位信号
addr	输入	5	内部寄存器的读写地址
write_enable	输入	1	内部寄存器的写使能信号
write_data	输入	32	内部寄存器的写入数据
exit_isr	输入	1	eret 指令清除 exl 的控制信号
in_bds	输入	1	当前指令是否在延迟槽中
hwirq	输入	6	外部设备中断信号
exc	输入	5	内部异常信号
curr_pc	输入	32	受害指令当前 PC 值
read_result	输出	32	内部寄存器的读取结果
epc	输出	32	保存的要跳转回的 pc 值
have2handle	输出	1	必须处理中断或异常

宏定义

类别	定义	值	意义
异常类型	EXC_NONE	5'd0	没有异常
异常类型	EXC_ADEL	5'd4	AdEL 异常（读取时地址错误）
异常类型	EXC_ADES	5'd5	AdES 异常（写入时地址错误）
异常类型	EXC_RI	5'd10	RI 异常（未知指令）
异常类型	EXC_OV	5'd12	Ov 异常（运算溢出）
prid	CP0_PRID	32'h0000002a	cp0 的处理器 ID
工作模式	MARS_COMPAT		cp0 是否工作在跟 MARS 兼容的模式下

功能

该部件为时序部件，没有特殊规定的话，所有寄存器初值均为全 0。但在 MARS 兼容模式下，sr 初值为 32'h0000ff11，意味着打开所有 8 个中断（包括 6 个硬件中断和 2 个软件中断），禁用所有 64 位内核地址空间、监管程序地址空间和用户地址空间的访问，基础模式为用户模式，未实现监管程序模式，exl 为 0，全局中断使能打开。

CP0 内部有四个寄存器，都是 32 位的。它们的概况如下。其中出现常量 0 的部分，是因为寄存器功能没有实现全或者按定义实现导致的，这些部分在普通模式下不允许写入，但在 MARS 兼容模式下允许写入，不会一直保持为 0，但写入的值目前还没有作用。PrID 寄存器为只读，所以它其实不是用寄存器实现的，是用 wire。

编号	寄存器	代码中的名字	结构	备注
12	SR	sr	{16'b0, allow_hwirq, 8'b0, exl, g_allow_hwirq}	

13	Cause	cause	{in_bds_i, 15'b0, hwirq_i, 3'b0, exc_i, 2'b0}	
14	EPC	epc	{epc_i}	实际上也是输出端口
15	PrID	prid	{prid}	直接用 wire 实现

各小寄存器的意义如下。

小寄存器	位宽	初值	意义
allow_hwirq	6	6'b0	允许相应硬件的中断请求，作为掩码使用
exl	1	1'b0	异常级别，在 ISR 时为 1
g_allow_hwirq	1	1'b0	在全局允许相应硬件的中断请求
in_bds_i	1	1'b0	在内部保存的受害指令是否在延迟槽中的信息
hwirq_i	6	6'b000000	在内部保存的硬件中断请求情况
exc_i	5	5'd0	在内部保存的异常代码
epc_i	32	32'h0	在内部保存的从 ISR 中返回的 PC
prid	32	32'h0000002a	处理器 ID

小寄存器定义在寄存器里面，在实现中是用宏表示的。

无论什么时候，若 addr 为相应寄存器的编号，则在 read_result 端口输出相应寄存器的值。若编号超出范围，则输出 32'b0。定义两个表示内部和外部异常或中断请求的 wire：have_irq，have_exc。当且仅当 (hwirq & allow_hwirq) != 0 && g_allow_hwirq == 1'b1 && exl == 1'b0 时，have_irq == 1'b1，否则为 1'b0。当且仅当 exc != EXC_NONE && exl == 1'b0 时，have_exc == 1'b1，否则为 1'b0。have2handle == 1'b1 当且仅当 have_irq || have_exc，否则为 1'b0。

在每个时钟上升沿，首先检查 have2handle。若 have2handle == 1'b1，则 exl <= 1'b1; in_bds_i <= in_bds; epc_i <= (in_bds == 1'b1) ? $unsigned(curr_pc) - $unsigned(4) : $unsigned(curr_pc)。此时，若 have_irq == 1'b1，则 exc_i <= 0，否则若 have_exc == 1'b1，则 exc_i <= exc。在 MARS 兼容模式下，还需要把代表软件中断的两位（[9:8]）清空，无论发生了硬件中断还是软件异常。然后检查 write_enable。若 write_enable == 1'b1 && (addr == 5'd12 || addr == 5'd13 || addr == 5'd14)，则认为试图写入有效，否则进入下一步骤。直接把对应寄存器写入相应的值，但是注意在正常模式下寄存器定义时出现常量 0 的部分会忽略写入，在 MARS 兼容模式下不会。最后检查其它信号。若 exit_isr == 1'b1，则 exl <= 1'b0。若每一步的第一个条件符合，则不需要检查后面的步骤。

在每个时钟上升沿，若在普通模式下，一个附加的步骤（无论如何都要执行）是若不是写入地址有效或者写入的地址不是 5'd13 或者 have2handle 不是 1'b0，则 hwirq_i <= hwirq。这是为了让 CPU 跟硬件更好地同步。

注意事项

1. **中断优先级高于异常，这在功能中也能看出来**
2. **硬件 IRQ 是每个时钟周期都要响应，在 IRQ 内都要响应，不跟着 CPU 的节奏来**
3. **把寄存器做成全可以写入的并且用宏指定小寄存器，这样做是为了跟 MARS 兼容**
4. **把 prid 做成 32'h0 是为了跟 MARS 兼容**
5. **跟 MARS 对拍时记得去掉不写 cause 且不需要处理中断就更新的部分，因为 MARS 没有这个机制；但是要加上发生异常或中断时清除软件中断位（即 hwirq），因为 MARS 有这个机制**
6. **实际上更新 hwirq_i 在进入中断的时候更新不行，因为进入中断后还得更新 hwirq_i；have2handle == 1'b0 可以不用判断，因为如果 have2handle ==**

1'b1，write_enable 早就被屏蔽了

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
rst	输入	1	同步复位信号
enable	输入	1	使能
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'b1`，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

p7 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
mfc0
mtc0
eret
```

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
LOAD_C0	mfc0
STORE_C0	mtc0
JUMP_C0	eret

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.cmp_result	F: npc.cmp_sig_result	F: npc.num	F: npc.jnum
BRANCH	F: pc.curr_pc	D: cmp.cmp	D: cmp.sig_cmp	D: im.result[15:0]	
JUMP_I	F: pc.curr_pc				D: im.result[25:
JUMP_R	F: pc.curr_pc				
JUMP_C0	F: pc.curr_pc				
（其它）	F: pc.curr_pc				
综合	F: pc.curr_pc	D: cmp.cmp	D: cmp.sig_cmp	D: im.result[15:0]	D: im.result[25:

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			
LOAD_C0			
STORE_C0			

JUMP_C0			
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
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_r
LOAD_M					
STORE_M				D: rf.read_result1	D: rf.read_r
LOAD_C0					
STORE_C0					
JUMP_C0					
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_r

M 级（MEM）

数据通路类型	M: dm.read_addr	M: dm.write_addr	M: dm.write_data	M: cpu_addr	M: cpu_write_data	M: a
CAL_R						
CAL_I						
LOAD	E: alu.result			E: alu.result		E: alu.
STORE		E: alu.result	E: rf.read_result2	E: alu.result	E: rf.read_result2	E: alu.
BRANCH						
NOP						

JAL	
JR	
CMOV	
CAL_M	
LOAD_M	
STORE_M	
LOAD_C0	
STORE_C0	
ERET	
综合	E: alu.result E: alu.result E: rf.read_result2 E: alu.result E: rf.read_result2 E: alu.

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	
LOAD_C0	M: cp0.read_result
STORE_C0	
ERET	
综合	E: alu.result, M: dm.read_result, F: $unsigned(pc.curr_pc) + $unsigned(8), E: md.out, M: cp0.read_result

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	pc.curr_pc	d_pc
D	im.result	d_im

```
E       pc.curr_pc      e_pc
E       rf.read_result1 e_reg1
E       rf.read_result2 e_reg2
E       ext.result      e_ext
E       im.result       e_im
M       pc.curr_pc      m_pc
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
W       cp0.read_result w_cp0
```

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

这里没有补充 D 级 BRANCH 类指令需要的 `cmp.cmp` 到 F 级的连接、 JAL 和 JR 类指令相应数据到 F 级的连接和 `cp0.epc` 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。**

注意：**所有级流水线都要接复位信号，因为要做到在异常产生时清空流水线。**

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，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，然后把它看成无符号数，再加 8。**

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
M: dm.read_result	M: dm.read_result, M: cpu_read_result	m_bridge
W: rf.write_data	（无）, E: alu.result, M: dm.read_result, D: npc.next_pc, E: md.out, M: cp0.read_result	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。**

注意：**`m_bridge` 是根据地址范围判断读取结果是哪个的，信号来源里的 `M: dm.read_result` 是原来的 `dm.read_result`。这是为了方便把 `bridge` 实现成钩子机制。**

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。

端口	所有的信号来源	MUX 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`, `M: cp0.read_result`	`fm_d1`
`E: rf.read_result[12]`	`M: npc.next_pc`, `M: alu.result`, `W: rf.write_data`, `M: md.out`, `M: cp0.read_result`	`fm_e1`
`M: dm.write_data`	`W: rf.write_data`	`fm_m`
`M: cp0.epc`	`D: rf.read_result2`, `E: rf.read_result2`, `M: rf.read_result2`	`fm_epc`

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。**

注意：**`M: cp0.epc` 是流水线前面的级转发优先的，这是因为最前面的级写入的 epc 才是逻辑上最终的 epc。**

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。

异常处理

异常处理首先是在流水线寄存器中建立清除机制，让异常或者中断发生的时候能够清空整个流水线。然后，就是传输好 epc，建立好 epc 的转发机制，这样就能正确地从 ISR 中返回。从 ISR 中返回时需要考虑暂停和填充 JUMP_C0 类指令气泡的关系，还有因为 JUMP_C0 类指令新带来的数据转发。还有就是做好回退和禁止写入的机制，这样中断发生的那个时钟周期的下一个时钟上升沿就能正确地抵消所有效果。

清除机制和回退、禁止写入的机制是各个数据通路部件内部实现的，epc 的转发机制在数据通路内部已经有转发 MUX 了。具体的控制由控制模块实现。

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	
指令类型	MFC0	9'b1100_00000	
指令类型	MTC0	9'b1101_00000	
指令类型	ERET	9'b1110_00000	
数据通路类型	UNKNOWN	4'b0000	未知指令
数据通路类型	CAL_R	4'b0001	R 型计算指令
数据通路类型	CAL_I	4'b0010	I 型计算指令
数据通路类型	LOAD	4'b0011	加载指令
数据通路类型	STORE	4'b0100	保存指令
数据通路类型	BRANCH	4'b0101	分支指令
数据通路类型	JUMP_I	4'b0110	带立即数的跳转指令
数据通路类型	JUMP_R	4'b0111	读写寄存器的跳转指令
数据通路类型	CMOV	4'b1000	条件传送指令
数据通路类型	CAL_M	4'b1001	使用 md 的计算指令
数据通路类型	LOAD_M	4'b1010	读取 md 内部寄存器的指令
数据通路类型	STORE_M	4'b1011	写入 md 内部寄存器的指令
数据通路类型	LOAD_C0	4'b1100	读取 cp0 内部寄存器的指令
数据通路类型	STORE_C0	4'b1101	写入 cp0 内部寄存器的指令
数据通路类型	JUMP_C0	4'b1110	按照 cp0 的内容跳转的指令

注意

1. 临时的数据通路类型都是从上往下长的。
2. **只剩下一种临时数据通路类型了（TODO）**

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
rst	输入	1	同步复位信号
d_instr	输入	32	当前在 D 级（ID）的指令
e_instr	输入	32	当前在 E 级（EX）的指令
m_instr	输入	32	当前在 M 级（MEM）的指令
w_instr	输入	32	当前在 W 级（WB）的指令

端口	类型	位宽	功能
rf_read_result2	输入	32	rf 的 2 号读取结果
e_md_busy	输入	1	输入 E: md.busy
m_dm_addr	输入	32	输入 M: dm.read_addr（其实也是 M: dm.write_addr）

端口	类型	位宽	功能
f_pc_invalid	输入	1	输入 F: pc.invalid
e_alu_sig_overflow	输入	1	输入 E: alu.sig_overflow
d_pc_curr_pc	输入	32	输入 D: pc.curr_pc
e_pc_curr_pc	输入	32	输入 E: pc.curr_pc
m_pc_curr_pc	输入	32	输入 M: pc.curr_pc
have2handle	输入	1	cp0 是否必须进入 ISR

端口	类型	位宽	功能
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_d_pff_rst	输出	1	控制 D 级流水线寄存器复位
cw_e_pff_rst	输出	1	控制 E 级流水线寄存器复位
cw_m_pff_rst	输出	1	控制 M 级流水线寄存器复位
cw_w_pff_rst	输出	1	控制 W 级流水线寄存器复位

端口	类型	位宽	功能
cw_f_npc_jump_mode	输出	4	控制 npc 的跳转模式

端口	类型	位宽	功能
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

端口	类型	位宽	功能
cw_e_m_alusrc	输出	1	控制 E：m_alusrc
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_e_m_hilo	输出	1	控制 E：m_hilo

端口	类型	位宽	功能
cw_m_m_bridge	输出	1	控制 M：m_bridge
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_m_cp0_write_enable	输出	1	控制 M：cp0.write_enable
cw_m_cp0_exit_isr	输出	1	控制 M：cp0.exit_isr
cw_m_cp0_in_bds	输出	1	控制 M：cp0.in_bds
cw_m_cp0_exc	输出	5	控制 M：cp0.exc
cw_m_cp0_curr_pc	输出	32	控制 M：cp0.curr_pc

端口	类型	位宽	功能
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

端口	类型	位宽	功能
cw_fm_d[12]	输出	4	控制 fm_d[12]
cw_fm_e[12]	输出	4	控制 fm_e[12]
cw_fm_m	输出	3	控制 fm_w
cw_fm_epc	输出	3	控制 fm_epc

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或者异常信号，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。

但是实际上控制模块在内部并不流水指令，而是把流水指令放到了数据通路中。这是因为数据通路的 E 级和 M 级都需要指令，而且在处理暂停和进入 ISR 时，统一的指令存储和流水线寄存器清除逻辑实际上更方便。

控制模块在内部流水异常信号和指令需要的寄存器，把流水指令的任务交给数据通路，从而做到比较有效的控制信号发射、数据冒险分析和异常收集与分析。负责控制信号发射的部分是纯组合逻辑，用函数实现。

同时，控制模块也在内部计算出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。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。

由于加入了中断和异常处理，而且有些中断和异常处理是需要覆盖当前指令的正常控制信号的，所以跟中断和异常处理有关的控制信号省略不写。

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
JUMP_C0	NPC_EPC
（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	M: cp0.write_enable
LOAD	1'b0	视具体指令而定	1'b0
STORE	1'b1	视具体指令而定	1'b0
LOAD_C0	1'b0	DM_NONE	1'b0
STORE_C0	1'b0	DM_NONE	1'b1

（其它）　1'b0　　DM_NONE　　1'b0

注意：**STORE 类指令类型会把 dm 和 bridge 的写使能都打开，但是它们两个写到无效地址时会无效，而且有 ac 检查地址，所以出错时不会有副作用。**

注意：**M：ac.dm_mode 和 M：dm.write_enable 和 M：dm.mode 相同，实现的时候把它的信号设置成 dm 对应的信号。**

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
LOAD_C0	1'b1	E：cp0.read_result
（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。

为了处理异常和中断，还要把每级指令的异常流水。下一级的异常默认流水这一级保存的异常，但是如果这一级又新增了异常，那么就流水这一级新增的异常。D 级异常流水寄存器默认流水 EXC_NONE。这样可以保证异常不断流水，而且能够做到先处理老异常再处理新异常。

注意：**检测到新异常就流水会让一条指令中更新级的异常覆盖这条指令更老级产生的异常，比如 lw 指令加法溢出了同时不对齐。**

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。

指令读写寄存器识别并没有识别 cp0 的相应寄存器，因为 cp0 大体上和 dm 类似，需要解决数据冒险的只是 STORE_C0 +JUMP_C0 着一种情况。这种情况只要在转发逻辑上加入相应逻辑即可解决。

指令读写寄存器识别也要在进入 ISR 时把所有流水指令读写寄存器号的寄存器全部清空。为了使设计更简洁，也是通过借用数据通路流水线的复位信号实现的。如果某一级数据通路流水线寄存器的复位信号为 1'b1，那么这一级对应的流水指令读写寄存器号的寄存器也要复位。否则，就直接写入上一级数据通路流水线寄存器的值。

注意：**W 级也有这一级数据通路流水线寄存器的复位信号（为了防止出现异常的指令进入下一级），所以也要设置上一段说的复位逻辑。**

数据通路类型	**reg1**	**reg2**	**regw**
CAL_R	RS	RT	RD
CAL_I	RS	ZERO	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
LOAD_C0	ZERO	ZERO	RT
STORE_C0	ZERO	RT	ZERO
JUMP_C0	ZERO	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。

对 M 级的 epc，先检查 D 级，再检查 E 级，这样也是为了保证转发的完整性。现在只有一种情况可以转发，就是 `mtc0 + eret`。如果 `mtc0` 要写入的寄存器号正好是 epc 的寄存器号，就要转发。

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_d[12]	M2D_cp0	8	E 级到 D 级，数据通路类型是 LOAD_C0，数据在 M 级产生好了

类别	定义	值	意义
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 E 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 E 级，数据通路类型为 LOAD_M，数据在 E 级产生好了
fm_e[12]	M2E_cp0	5	M 级到 E 级，数据通路类型是 LOAD_C0，数据在 M 级产生好了

类别	定义	值	意义
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）

类别	定义	值	意义
fm_epc	EPC_D2M_rf	1	D 级到 M 级，数据通路类型是 STORE_C0，数据在 D 级产生好了
fm_epc	EPC_E2M_rf	2	E 级到 M 级，数据通路类型是 STORE_C0，数据在 D 级产生好了
fm_epc	EPC_M2M_rf	3	M 级到 M 级，数据通路类型是 STORE_C0，数据在 M 级产生好了，但是 D 级的 JUMP_C0 类指令马上就需要新的 epc 值了，来不及等到下一个时钟上升沿

注意：

1. **B2A_.* 表示 B 级从 A 级转发。**
2. **宏的值要和对应转发 MUX 的接线顺序相符。**
3. **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，它可以在 D 级完成要写入数据的判断。**
4. *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或设备的指令，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。类似地，fm_epc 检查的也是要读取的第二个寄存器，因为现在用到的所有写入 epc 的指令，要写入 cp0 的数据也是都与相应指令第二个寄存器的读取结果对应。以后可能加上检查要读取的第一个寄存器，不过也是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 最终会回归正常。

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。同时，判断异常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：**控制内部的流水线和判断异常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
LOAD_C0		
STORE_C0		2

JUMP_C0

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
LOAD_C0 1 0
STORE_C0
JUMP_C0

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。

`e_md_busy == 1'b1` 时，会一直插入气泡，直到 `e_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。

由于 cp0 和 dm 类似，只有一种需要转发的情况，所以暂停不需要通过 Tuse 和 Tnew 机制来实现。但是，确实有一种情况需要暂停，就是 D 级为 JUMP_C0、E 级为 STORE_C0 而且写入的寄存器就是 epc，M 级 Tnew 为 1 的情况。这是因为 JUMP_C0 只能在 D 级，STORE_C0 在 M 级或者往后就直接能转发了。如果 STORE_C0 在 E 级，那么 M 级的 Tnew 为 0 时，就能两级转发了。但是，如果 M 级的 Tnew 大于 0，两级转发就构造不起来，但是 D 级的 JUMP_C0 这类指令又马上要用，所以只能暂停了。

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。**

注意：**异常处理需要的 F 级暂停在异常处理一节里。**

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。

设备访问

设备读写

设备读写是通过 bridge 模块进行的。bridge 模块通过把设备的内部寄存器读写映射成类似 dm 的方式，通过按字索引操作设备的内部寄存器。

但是，读写的地址验证，还是通过专门的 ac 模块进行的，bridge、dm 和外部设备只是忽略不合法的读写操作。

控制读入结果

由于有了 bridge 和 dm，所以会有两个并行的读入结果。在这种情况下，选择哪个读入结果是根据地址决定的。这样就把 bridge 和 dm 的结果统一成了 `M: dm.read_result`，使数据冒险的处理更为简便。

由于内存的地址范围是固定的，所以采用只判断内存地址范围的方式来选择 `M: dm.read_result` 其实是哪个结果。如果 `m_dm_addr` 在内存地址的范围之内，就选择原来的 `M: dm.read_result`，否则选择 `cpu_read_result`。选择的 MUX 是 `m_bridge`，定义原来的 `M: dm.read_result` 和 `cpu_read_result` 分别为 `1'b0` 和 `1'b1`，按照这种选择逻辑选择相应的信号。

注意：**地址比较要用无符号数比较。**

异常和中断处理

异常处理

通过异常流水线的机制处理异常。但是，下一级的存储异常的寄存器能够用这一级的异常覆盖这一级存储异常的寄存器原来存储的值。而且，在暂停时，相应级的寄存器也要插入气泡，也就是 EXC_NONE。

由于 cp0 部署在 M 级处理异常，但需要防止受害指令进入 M 级，所以在 D 级、E 级、M 级和 W 级有异常流水寄存器。然后，在 M 级把这一级的异常流水寄存器和这一级产生的异常再进行比较，如果这一级有地址异常（也就是 `M: ac.validity != AC_VALID`），就认为 M 级指令产生了异常，首先处理 M 级指令的异常，把 M 级指令的异常字段设成 `ac.validity` 的非正常情况对应的值，同时根据 dm 是否写使能的原来的信号判断 dm 的读写模式。然后，把最终得到的 M 级指令的异常输出给 cp0。

注意：**为了正确判断 dm 写使能原来的信号，需要在控制模块内部判断。不能用输出的 dm 写使能信号，因为这样能被 have2handle 覆盖。**

如果需要进入 ISR，就把 D 级、E 级、M 级和 W 级的流水线寄存器全部复位，并加上相应的钩子机制，使下个时钟上升沿带来的数据写入被取消，而且把 PC 改成 ISR 的。

注意：**为了防止受害指令进入 M 级，需要把 M 级流水线寄存器复位。**

由于暂停会让 F 级和 D 级的指令停住不动，所以它的优先级最大。由于 JUMP_C0 类指令不是跳转指令，所以不能实现延迟槽，不暂停的时候需要复位 D 级流水线寄存器，通过这种方式插入气泡。这样，存储异常的寄存器的更改逻辑需要一定的调整。

每级存储异常的寄存器更新逻辑如下：

流水线级	更新逻辑
D 级	若 `cw_d_pff_rst == 1'b1`，则值变为 EXC_NONE。若需要暂停，则值不变。否则，若 D 级数据通路类型为 JUMP_C0，则插入气泡。否则，若 `F: pc.invalid == 1'b1`，则写入 EXC_ADEL。否则，写入 EXC_NONE。
E	若 `cw_e_pff_rst == 1'b1`，则值变为 EXC_NONE。若需要暂停，则写入 EXC_NONE。否则，若

级 d_instr_kind == UNKNOWN，则写入 EXC_RI。否则，写入 D 级的值。

M 若 cw_m_pff_rst == 1'b1，则值变为 EXC_NONE。若 ekind 为 ADD / ADDI / SUB 且 E:
级 alu.sig_overflow == 1'b1，则写入 EXC_OV。否则，写入 E 级的值。

因为每级存储异常的流水线寄存器的复位逻辑与每级数据通路中的流水线寄存器是一致的，而每级数据通路中流水线寄存器的复位逻辑是正确的，因此可以这样借用控制信号。

TODO: EXC_RI 好像只是 opcode 未知

中断处理

中断处理是通过 bridge 进行的。bridge 负责捕获硬件的中断，并把它送到 cp0 的端口。如果当前指令发生了异常，也要优先处理中断，因为中断的优先级更高。

注意：**中断的优先级比异常高。**

进入 ISR

cp0 中有专门的输出端口 have2handle 来做这件事。如果在 ISR 外面执行指令突然碰到异常或中断，have2handle 由组合逻辑驱动，就会变成 1'b1。下一个时钟周期的时候，就会处理中断或异常。cp0 会与控制模块写作，保存中断发生时需要的信息。

首先，为了干净地处理中断或异常，需要复位所有的流水线寄存器，然后把新 PC 定位为 ISR 的起始地址。这就需要把原来对 NPC 模式的控制信号 cw_npc_mode 加上钩子，如果出现了中断或异常，就直接改成 NPC_ISR，让 pc 跳转。这就需要四个控制信号：[demw]_pff_rst。其中 e_pff_rst 已经有了，所以需要加上钩子。同时，由于 JUMP_C0 类指令不需要延迟槽，所以 d_pff_rst 在 D 级指令为 JUMP_C0 类指令的时候，也需要为 1'b1。

其次，为了彻底地取消流水线的效果，需要在下个时钟上升沿禁止所有可写入部件的写入。这也需要把原来对 dm、bridge、cp0 和 md 的控制信号屏蔽掉，同样也是加入钩子机制，如果出现了中断或异常，哪怕写入使能也要取消使能。但是，rf 不需要这样做，因为把 cp0 部署在 M 级以后，任何出现异常的指令最晚到 M 级都会被处理，而且 W 级指令只有写回寄存器，不会产生异常，在下一个时钟上升沿正好写入寄存器，完成了最终的指令功能。

对 md，如果 E 级或 M 级指令是 STORE_M 类指令，那么需要把 restore 信号置成 1'b1，因为在下一个时钟下降沿，md 已经开始或者早已结束解析控制信号并存储新值了。因为 E 级或 M 级指令会重新执行。如果 E 级或 M 级指令是 CAL_M 类指令，那么把 stop 信号置成 1'b1，因为有正在进行的运算，需要停止。

注意：**如果有 mult / multu / div / divu 指令执行完了然后 md 在执行，md 执行结束前有指令出现异常，那么 md 的值同样会更新。暂停机制保证了操作 md 的三类数据通路类型都是实际上顺序执行的，所以 md 的值会更新，不过会出现在 ISR 里，让 ISR 用。**

控制信号与其加了钩子后的控制信号的关系如下。

控制信号	加钩子后的控制信号	与原来控制信号和 have2handle 的关系	备注
cw_m_dm_write_enable_orig	cw_m_dm_write_enable	cw_m_dm_write_enable & (~have2handle)	
cw_m_cp0_write_enable_orig	cw_cp0_write_enable	cw_cp0_write_enable_orig & (~have2handle)	[1]
		cw_e_pff_rst_orig 或	

cw_e_pff_rst_orig	cw_e_pff_rst	have2handle	
cw_f_npc_jump_mode_orig	cw_f_npc_jump_mode	(have2handle == 1'b1) ? NPC_ISR : cw_f_npc_jump_mode_orig	[2]
cw_f_pc_enable_orig	cw_f_pc_enable	cw_f_pc_enable_orig 或 have2handle	[3]

注：

1. 实际上不需要加钩子，因为 cp0 内部就是写入的优先级比处理异常和中断的优先级低。
2. 如果 have2handle == 1'b1，那么 cw_f_npc_jump_mode 必须为 NPC_ISR，因为需要强制跳转到 ISR 的起始地址。
3. 为了避免暂停时 pc 不动的问题，暂停时 pc 必须强行跳到 ISR 的起始地址

然后，为了能让 cp0 获得准确的数据，必须把控制模块分析出来的相应的 cp0 控制信号接入 cp0 相应的输入漩口。接入关系及意义如下。

控制模块输出端口	cp0 输入端口	生成方法	意义	备注
cw_m_cp0_write_enable	write_enable	M 级指令为 STORE_C0 类指令	能够让 STORE_C0 类指令写入 cp0 内部寄存器	[5]
cw_m_cp0_exit_isr	exit_isr	M 级指令为 JUMP_C0 类指令	能够让 JUMP_C0 类指令正常去掉中断状态	[1]
cw_m_cp0_in_bds	in_bds	W 级指令为 BRANCH / JUMP_I / JUMP_R 类指令	能够让 cp0 判断出受害指令是否在延迟槽内	[2]
cw_m_cp0_exc	exc	M 级存储异常的流水线寄存器和 M 级的异常情况综合后的结果	能够让 cp0 得到当前发生的异常	[3]
cw_m_cp0_curr_pc	curr_pc	若 M: curr_pc != 0，则为 M: curr_pc；否则一直追溯到 D 级的 curr_pc 为止	能够让 cp0 得到受害指令的 PC 作为 epc	[4]

注：

1. 一般来说 JUMP_C0 类指令应该马上写 cp0 的相应寄存器，从而去掉 exl，以方便之后的 LOAD_C0 类指令。但是，JUMP_C0 类指令由于不是跳转类指令却在 D 级译码，后面一定跟着一个气泡。所以，它到了 M 级的时候，LOAD_C0 类指令最早才到 D 级，下一个时钟上升沿后就可以转发，转发来得及，因此不需要再做 cp0 内部寄存器的转发，直接让 M 级指令为 JUMP_C0 类指令时设置 cw_m_cp0_exit_isr <= 1'b1 即可。
2. 受害指令是 M 级指令，所以如果 M 级指令在延迟槽中，W 级指令一定为以上三种类型。W 级指令为 JUMP_C0 类指令时，后面一定跟着一个气泡，再说 JUMP_C0 类指令没有延迟槽，所以不需要考虑。如果受害指令在延迟槽内，保存的 epc 是受害指令的 PC 作为无符号数再减去 4，因为要恢复到跳转指令，把跳转指令再执行一遍。
3. 异常情况按照处理异常这一节，是一定要综合的。综合以后就能让 cp0 得到 M 级指令最近的异常情况，因为异常原则上都是一级一级往下传递，而且新异常会覆盖老异常。
4. M 级指令出现异常，那么 M 级一定有指令，所以 M: curr_pc != 0，受害指令的 PC 就能被找到。但是，如果出现中断而且 M 级是气泡，M: curr_pc == 0，这是就应该回溯找到没有气泡的

指令。回溯到 D 级就可以，因为一般暂停产生的气泡都是只在 E 级生成，只有一种情况例外，就是 JUMP_C0 类指令。但是它在 D 级生成气泡的时候，自己也到了 E 级，表格中的逻辑照样适合。流水线刚被全部清空，正要进入 ISR 的那个上升沿不会再出现中断。这是因为中断被屏蔽了，况且 ISR 起始地址肯定是关于字对齐的，否则就是硬件设计缺陷了，之后各级由于都是气泡，也不会再出现中断。所以，用表格中的逻辑不用担心找不到不合法的 PC。

5. 这里的信号是原来的信号，注意要像加钩子表格那样加上钩子。

CP0

cp0 主要是负责获取中断时的情况和保存中断的相关信息。cp0 输入了当前的中断和异常情况、epc、受害指令是否再延迟槽内这些相关信息，等到从 ISR 返回时恢复。同时，cp0 也支持读写它内部的寄存器。

cp0 需要通过组合电路判断这个时钟上升沿后 M 级是否出现了异常，以及是否出现了硬件中断。它的内部寄存器有全局中断使能、硬件中断使能掩码和中断层级等设置。硬件中断首先与硬件中断使能掩码相与，然后再根据全局中断使能和中断层级来判断是否进入 ISR。软件异常直接根据全局中断使能和中断层级来判断是否进入 ISR。如果中断层级为 `1'b1`，就不需要再进入 ISR 了。

cp0 首先是在 `have2handle == 1'b1` 的情况下进入 ISR 并设定相应的状态，其次才是处理写请求，最后是处理清除 `exl` 的请求。这样能保证中断机制符合一般的认知，而且不妨碍指令执行逻辑上的顺序性。

从 ISR 返回

从 ISR 返回是用 JUMP_C0 类指令，现在只有一条，就是 ERET 指令。ERET 指令需要读取 `M: cp0.epc` 作为 `F: npc.next_pc`，而且有 STORE_C0 类指令，所以 epc 也要转发，这在前面的转发一节中已经提到了。

虽然 JUMP_C0 类指令也写 cp0 的内部寄存器，但是它们不需要转发，因为它并没有延迟槽，后面会带一个气泡。这样保证了它在 M 级的时候，新指令才会进来，转发来得及，更何况还有暂停机制。实际上，ERET 指令并不算跳转类指令，所以它没有延迟槽。在 JUMP_C0 类指令后面插入气泡的方法，是 D 级指令为 JUMP_C0 类指令时，如果不暂停，在下个时钟上升沿就复位 D 级寄存器。这样，D 级就会插入一个气泡，同时不跟暂停机制冲突，因为暂停时 D 级指令是不动的。实际上，这跟流水线寄存器的复位和暂停优先级冲突了，但是暂时没有好的办法。

从 ISR 返回时，在 D 级的跳转模式是 NPC_EPC，是经过转发的 `M: cp0.epc`。转发也是较新的级优先级较大，因为这样符合逻辑。转发后就不需要清空流水线了，因为正常的指令可以继续执行下去，不像进入 ISR 时需要撤销比 M 级还新的几条指令的执行结果。

注意：**D 级流水线寄存器的复位逻辑是在前面进入 ISR 时实现的，并不在最后。**

注意：**D 级流水线只有不暂停的时候才能复位，否则一暂停会把 D 级的 JUMP_C0 类指令清空。**

注意：**可以在 M 级时判断出 JUMP_C0 类指令再清空 `exl`，因为 JUMP_C0 类指令清空 `exl` 后到了 W 级，会正常执行完，这时正好当时的受害指令到了 M 级，如果还有中断或者异常，该指令及其之后指令仍然会被中断，重新进入 ISR。但是，会出现 `eret` 没到 M 级的时候或者刚清空流水线 ISR 的第一条指令还来不及进入 M 级的时候不持续的硬件中断不会响应的问题，这种问题应该可以忽略，把它当成 ISR 已经进入或者还没退出来解决。**

复位机制

复位机制主要是复位控制模块内部的流水线寄存器。但是控制模块还控制着各级流水线寄存器的复位信号，所以还需要把每级流水线的复位信号再加上钩子。由于内部流水线寄存器的更新代码都把对应级流水线寄存器的复位功能放在优先级最高的位置，所以把控制四级流水线寄存器复位的信号都加上钩子机制即可，单独每级的流水线寄存器和内部的流水线寄存器就不用控制了。

加入对流水线寄存器复位信号的钩子只需要加一个新的逻辑条件 `|| (rst == 1'b1)` 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。但是，为了更好地和外部设备通信，CPU 和 `bridge` 模块相连接，通过除了时钟信号的其它输入输出端口与外部设备通信。

端口定义

端口	类型	位宽	功能
`clk`	输入	1	时钟信号
`rst`	输入	1	复位信号
`cpu_read_result`	输入	32	CPU 从设备得到的读取结果
`hwirq`	输入	6	设备的中断信号
`cpu_addr`	输出	32	CPU 要对设备相应寄存器操作的地址
`dev_write_enable`	输出	1	CPU 对设备的写使能信号
`cpu_write_data`	输出	32	CPU 要对设备写入的数据

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。

按钮模块

原理

按钮模块通过捕获外部输入的 8 个信号，把它转换成相应的硬件中断。输入会自动经过防抖处理，因此在模块内不考虑防抖的问题。由于按钮只是输入，没有输出，所以不考虑向设备内寄存器写入数据的问题。

端口定义

端口	类型	位宽	功能
`clk`	输入	1	时钟信号
`rst`	输入	1	复位信号
`input_`	输入	8	输入的 8 个信号，代表用户输入
`addr`	输入	32	地址信号
`out`	输出	32	读出的数据

宏定义

暂无

参数定义

参数	默认值	功能
BASE_ADDR	0x7fff0100	MMIO 基地址

功能

该部件为时序部件，所有寄存器初值均为 0。

该部件维护一个 8 位寄存器 stored，负责存储在这个时钟周期中用户的输入。

每个时钟上升沿，若 rst == 1'b1，则 stored <= 0。否则，stored <= ~input_。

无论什么时候，若 addr == BASE_ADDR，则令 out = {24'b0, stored}，否则令 out = 32'b0。

LED 控制模块

原理

一共有 32 个 LED，能够使用写入的方式控制。为了能够使高电平持续从而使 LED 常亮，LED 的控制都是采用寄存器保存当前 LED 开关的数值的。

端口定义

端口	类型	位宽	功能
clk	输入	1	时钟信号
rst	输入	1	复位信号
we	输入	1	写使能信号
wd	输入	32	要写入的数据
addr	输入	32	地址信号
led_light	输出	32	输出的当前 LED 开关数值

宏定义

暂无

参数定义

参数	默认值	功能
BASE_ADDR	32'h00007f34	MMIO 基地址

功能

该部件为时序部件，所有寄存器初值均为 0。

维护一个 32 位的 stored 寄存器，保存当前 LED 开关状态。

每个时钟上升沿，若 `rst == 1'b1`，则 `stored <= 0`。否则，若 `(we == 1'b1) && (addr == BASE_ADDR)`，则 `stored <= wd`。

无论什么时候，`led_light = ~stored`。

七段数码管控制

原理

七段数码管通过向内部的 5 个寄存器写入值，来实现对七段数码管的控制。5 个寄存器对应 5 个数码管，每个寄存器有 8 位，让数码管分别显示 0-f。这样，它对外就有两个暴露出来的寄存器。

由于数码管分组，每组只有统一的段选信号和开关，所以需要通过隔一段时间刷新信号的方式进行控制，这样能显示出稳定的图像。具体分组如下：

内部寄存器地址	内部寄存器结构	具体数据位置	对应的数码管	对应的数码管段选信号	对应的数码管段选信号值
`$unsigned(BASE) + $unsigned(4)`	`{24'b0, tube2}`	`tube2[3:0]`	`digital_tube2`	`digital_tube_sel2`	`1'b1`
`$unsigned(BASE)`	`{tube1, tube0}`	`tube1[15:12]`	`digital_tube1`	`digital_tube_sel1`	`4'b1000`
`$unsigned(BASE)`	`{tube1, tube0}`	`tube1[11:8]`	`digital_tube1`	`digital_tube_sel1`	`4'b0100`
`$unsigned(BASE)`	`{tube1, tube0}`	`tube1[7:4]`	`digital_tube1`	`digital_tube_sel1`	`4'b0010`
`$unsigned(BASE)`	`{tube1, tube0}`	`tube1[3:0]`	`digital_tube1`	`digital_tube_sel1`	`4'b0001`
`$unsigned(BASE)`	`{tube1, tube0}`	`tube0[15:12]`	`digital_tube0`	`digital_tube_sel0`	`4'b1000`
`$unsigned(BASE)`	`{tube1, tube0}`	`tube0[11:8]`	`digital_tube0`	`digital_tube_sel0`	`4'b0100`
`$unsigned(BASE)`	`{tube1, tube0}`	`tube0[7:4]`	`digital_tube0`	`digital_tube_sel0`	`4'b0010`
`$unsigned(BASE)`	`{tube1, tube0}`	`tube0[3:0]`	`digital_tube0`	`digital_tube_sel0`	`4'b0001`

端口定义

端口	类型	位宽	功能
`clk`	输入	1	时钟信号
`rst`	输入	1	复位信号
`we`	输入	1	写使能
`wd`	输入	32	要写入的数据
`addr`	输入	32	要读写的地址
`rd`	输出	32	读取的数据
`digital_tube[0-2]`	输出	8	第 `[0=2]` 个数码管的控制信号
`digital_tube_sel[01]`	输出	4	第 `[01]` 个数码管的段选信号

digital_tube_sel2　　输出 1　　第 2 个数码管的段选信号

宏定义

类别	定义	值	意义
计数器	NIXIE_CTR	32'd50000	计数器初值，决定计数器周期

参数定义

参数	默认值	功能
BASE_ADDR	32'h00007f38	MMIO 基地址

功能

该部件为时序部件，所有寄存器若未说明，则初值均为 0。

维护一个 32 位的寄存器 ctr 和一个 4 位的寄存器 phase，以及一个 8 位的寄存器 tube0 和两个 16 位的寄存器 tube1 和 tube2。ctr 和 phase 初值分别为 NIXIE_CTR 和 4'b1000。每个时钟上升沿，若 rst == 1'b1，则所有寄存器恢复初值。否则，若 we == 1'b1 且 wd 属于上表中提到的地址之一，则把上表中对应的寄存器写入 wd 中对应位置的内容。否则，若 $unsigned(ctr) > $unsigned(0)，则 ctr <= $unsigned(ctr) - $unsigned(1)。否则，ctr <= NIXIE_CTR，phase 按照下表变换。

phase 原值	phase 新值
4'b1000	4'b0100
4'b0100	4'b0010
4'b0010	4'b0001
4'b0001	4'b1000
（其它）	4'b1000

无论什么时候，digital_tube_sel2 = 1'b1，digital_tube_sel[10] = phase。令两个 4 位的 wire 变量 data[210] 为按照段选信号和对应的数码管序号得到的内部寄存器 tube[210] 的相应部分。把 data[210] 转换后的相应结果输出到对应的 digital_tube[210] 中。具体的转换逻辑使用数码管译码器实现。

无论什么时候，若 addr 为合法的输入地址，则 rd 为相应的结果。否则，rd = 32'b0。

注意事项

1. 译码器用单独的模块实现，这样能增强模块化。
2. **地址计算和比较用无符号数**
3. **写入数据后刷新是及时的，没有延迟，可能和要求不大一样**

开关模块

原理

开关模块负责控制 FPGA 上 64 个开关，并以一定的形式输出。由于 FPGA 上 64 个开关分成 8 组，每组代表的高低位不同，所以应该按组输入模块，以使模块的接线与 FPGA 的接线相对应。

由于一共有 64 个开关，所以一共有 64 位，分成两个 32 位寄存器读取。

端口定义

端口	类型	位宽	功能
clk	输入	1	时钟信号
rst	输入	1	同步复位信号
addr	输入	32	读取地址
dip_switch[0-7]	输入	8	8 组开关
out	输出	32	读取结果

宏定义

暂无

参数定义

参数	默认值	功能
BASE_ADDR	32'h00007f2c	MMIO 基地址

功能

该部件为时序部件，所有寄存器初值为 0。

维护一个内部寄存器 stored，为 64 位，存储 {dip_switch7, dip_switch6, ..., dip_switch0}。

每个时钟上升沿，若 rst == 1'b1，stored <= 0。否则，stored <= ~{dip_switch7, dip_switch6, ..., dip_switch0}。

无论什么时候，real_addr = $unsigned(addr) - $unsigned(BASE_ADDR)。若 real_addr == 0，则 out = stored[31:0]；若 real_addr = 4，则 out = stored[63:32]；否则 out = 32'h0。

计时器

原理

计时器是产生硬件中断的一种示例部件。它由状态机组成，可以进行定时，并在规定的时间到时产生中断。它内部维护一个计数器和保存控制信号的两个寄存器，从而可以对计数进行控制。CPU 可以通过 bridge 模块更改各寄存器的值。

端口定义

端口	类型	位宽	功能
clk	输入	1	时钟信号
rst	输入	1	同步复位信号
addr	输入	32	地址信号
write_enable	输入	1	写使能信号
write_data	输入	32	要写入内部寄存器的数据
read_result	输出	32	从内部寄存器读出的数据
irq	输出	1	中断请求输出

宏定义

类别	定义	值	意义
状态	TIMER_IDLE	3'b000	空闲状态
状态	TIMER_LOAD	3'b001	装载状态
状态	TIMER_CNT	3'b010	计数状态
状态	TIMER_INT	3'b011	处理中断状态

参数定义

参数	默认值	功能
BASE	32'h00007f00	在 bridge 模块中该模块实例的基址

功能

该部件为时序部件。

内部维护 3 个 32 位寄存器，名字按地址从小到大分别为 ctrl，preset，count。ctrl 的结构为 {28'b0，allow_irq，1'b0，mode，enable}，preset 和 count 分别作为 32 位计数的预设值和当前值。allow_irq，mode 和 enable 分别表示是否允许中断、计数模式和计数器使能。它们的初值都为全 0。每个时钟上升沿，首先检查 rst 是否为 1'b1，若是，则把所有寄存器都恢复初值。

之后，检查写入是否有效。若 write_enable 为 1'b1 且要写入的地址为 preset 或 ctrl 的起始地址，则写入有效，把对应的寄存器写入 write_data 对应的内容。注意 count 寄存器禁止写入。注意写入 CTRL 寄存器时，原来为全 0 的位仍然保持全 0。若地址超出范围或写入无效，则什么也不做。

注意：**地址越界检查由 ac 完成，写入设备的字不对齐时直接被设备忽略，但是会触发异常。**

注意：**count 寄存器禁止写入，若试图写入 count 寄存器，也会被设备忽略，而且会触发异常。**

最后，进行状态转移。一共有四种状态，在宏定义中描述。每种状态中没有描述的情况，默认为什么也不做。

1. 若为 TIMER_IDLE，则当 enable == 1'b1 时，令 irq_reg <= 1'b0，并转移到 TIMER_LOAD。否则，若 allow_irq == 1'b0 时，也令 irq_reg <= 1'b0。
2. 若为 TIMER_LOAD，则令 count <= preset，并转移到 TIMER_CNT。
3. 若为 TIMER_CNT，则 enable == 1'b0 时，转移到 TIMER_IDLE。否则，$unsigned(count) > $unsigned(1) 时，count <= $unsigned(count) - $unsigned(1)；$unsigned(count) == $unsigned(1) 时， count 同样按照上一种情况自减 1，并转移到 TIMER_INT，但是同时在 allow_irq == 1'b1时，令 irq_reg <= 1'b1。
4. 若为 TIMER_INT，则 mode == 1'b0 时， enable <= 1'b0，这样可以保持中断信号，但allow_irq == 1'b0 时，也得令 irq_reg <= 1'b0；mode == 1'b1 时，irq_reg <= 1'b0。mode 的两种值都要转移到 TIMER_IDLE。

无论什么时候，read_result 都是 addr 对应的内部寄存器的值。若 addr 超出范围，则为 32'b0。无论什么时候，irq = irq_reg。

注意事项

1. 改寄存器个数的时候记得同时改 read_result 的判断
2. **地址运算都是无符号的**
3. **allow_irq 因为默认不会用，所以初始化为 0**

4. **写不该写的寄存器和写的模式错误，写使能信号会被 cpu 禁用掉，所以不用担心误写**

UART 控制模块

原理

UART 控制模块是通过使用 FPGA 上自带的 UART 协议相关端口，实现 UART 上的协议从而能够让 FPGA 与外界沟通的模块。UART 上的协议是串行通信的，利用串行的相关协议。

具体的 UART 控制模块是使用现成的 MiniUART 模块，只需要做少量的修改。具体的原理在 MiniUART 模块的文档中。

端口定义

端口	类型	位宽	功能
CLK_I	输入	1	时钟信号
DAT_I	输入	32	要向内部寄存器写入的数据
DAT_O	输出	32	要从内部寄存器读出的数据
RST_I	输入	1	同步复位信号
ADD_I	输入	3	内部地址输入
STB_I	输入	1	有读写操作
WE_I	输入	1	写使能
ACK_O	输出	1	输出响应信号
RxD	输入	1	UART 输入信号
TxD	输出	1	UART 输出信号
IRQ_data_complete	输出	1	数据传送完成的 IRQ

宏定义

类别	定义	值	意义
偏移	OFF_UART_.*	'h0	内部数据寄存器偏移
时钟信息	CLK_FRQ	35000000	FPGA 时钟频率
时钟信息	CYCLE	1000000000	每秒时钟周期数
采样	SAMPLE_FREQUENCE	8	每个字节的采样数
采样	HALF_BIT	(SAMPLE_FREQUENCE/2-1)	输出半个字节计数器的阈值
波特率	BAUD_RCV_.*	(CLK_FRQ/(.* *SAMPLE_FREQUENCE)-1)	在相应波特率下接收每个比特的计数器的阈值
波特率	BAUD_SND_.*	(CLK_FRQ/(.*)-1)	在相应波特率下输出每个字节的计数器的阈值

功能

具体的功能在 MiniUART 模块的文档中。

注意事项

1. **MiniUART 实际上是现成的代码。**

顶层模块

原理

顶层模块是整个工程的顶层模块，负责把用到的所有电路综合起来。顶层模块包括 CPU 和 `bridge`，把它们接在一起，形成综合的模块。

端口定义

端口	类型	位宽	功能
`clk`	输入	1	时钟信号
`rst`	输入	1	复位信号

接线

CPU 和 `bridge` 的接线，除了 `clk` 和 `rst` 之外都是可以对应的。所以，可以列出对应关系表。

CPU 端口	`bridge` 端口	数据流动方向
`clk`	`clk`	外部 → CPU 和 `bridge`
`rst`	`rst`	外部 → CPU 和 `bridge`
`cpu_addr`	`addr`	CPU → `bridge`
`dev_write_enable`	`write_enable`	CPU → `bridge`
`cpu_write_data`	`write_data`	CPU → `bridge`
`cpu_read_result`	`read_result`	`bridge` → CPU
`hwirq`	`hwirq`	`bridge` → CPU

功能

顶层模块按照接线连接起 CPU 和 `bridge`，通过这种方式确定 CPU 和 `bridge` 之间数据流的流向。

技巧

慌了怎么办

1. Don't panic! 做出来是最重要的
2. 深呼吸，专心想实现和调试的事情，不要害怕干不出来
3. 看看哪里的逻辑出错了，**不要逃避！**
4. 用小数据、边界数据、特殊数据测试
5. 踏踏实实想逻辑、定义、算法，必要的时候自己再描述一遍 / 写一遍，不要根据原来做出来的

如何加新指令

1. 看好 RTL，把它转换成数据通路的连线

- 注意流水线分级
- 可能需要引入新的流水线寄存器

2. 如果有多对一的情况，就应该用 MUX

- MUX 是原来的值，改控制信号
- MUX 是新的值，改控制信号，**可能要改 MUX 的位宽和对应接线的位宽**

3. 改好控制信号

- 对指令域进行识别
    - 尽量把新指令归约到原来的 dptype 上，**可以使用 $0，也可以利用其它特殊寄存器，毕竟控制模块里对读写寄存器的指定是 arbitrary 的**
- 如果需要一个新的 dptype
    - 计算好控制信号
    - 计算好 Tuse 和 Tnew
    - 看好如何转发、是否需要改转发路径
- 如果需要改转发路径
    - 确定转发的源和目的
    - **注意数据通路里的转发 MUX 和控制单元里的控制信号需要同时改**
    - **注意关于 cp0.epc 的转发**
- 如果有新的跳转规则
    - 尽量改 npc，让 npc 基于比较结果判断
    - 如果引入了新的比较方式，就需要改 cmp，**注意有符号 / 无符号和运算溢出问题和改 cmp 的接口**，同时也要改数据通路和 npc 的接口
    - 如果要跟立即数比较，**先看一下立即数的扩展模式**，能用 npc 解决的尽量用 npc 解决，p8 一般不用改 cmp
    - 如果根据一个寄存器跳转，那么按照引入了新的比较方式处理，改 cmp 的比较方式
    - 如果跳转时涉及 retaddr，那么有时可以按照 JUMP_[IR] 处理
    - 如果跳转时涉及 cp0.epc，那么可能可以归约到 JUMP_C0 处理，**如果不能，注意转发**
- 如果有新的立即数扩展方式
    - 如果还是 im.result[15:0] 改 ext，**注意有符号 / 无符号的区别**
    - 如果是 im.result 的其它部分，记得加 MUX 信号来源，**注意位宽和有符号 / 无符号的区别**
- 如果有新的寄存器号表示方法
    - 加 MUX 信号来源，**记得改位宽**，控制信号用 sane defaults
    - 可以根据指令类型特判
- 如果 alu 有新的运算
    - **抓好定义**，例如补码的相反数，最小的负数没有相反数
    - **注意地址计算是无符号计算、指令给定了是不是有符号运算要注意**
    - 如果是两个输入的运算，直接写新运算
    - 如果是三个输入的运算，看看能不能省下一个运算源，**有的时候要改控制的输入**，比如条件传送指令需要根据第二个寄存器的值判断 rf.we
        - 如果新值能够比较快地出来，**注意改转发路径，但是为了正确定不改也可以**，比如条件传送指令

    - 如果是输入带附加参数的运算，可以开一个 `alu` 端口，然后在控制器上接过去，也可以通过正常数据通路传过去（不推荐），比如移位运算可以直接在控制器和 `alu` 上开端口
        - 注意一般都有 Python，**可以自动代码生成**

- 如果 `md` 有新的运算
    - **抓好定义**，比如补码的乘除法运算
    - **注意有 / 无符号计算**
    - **注意掌握好 `md` 的内部状态机，`md` 利用了时钟的下降沿**
    - **如果需要检测特殊情况，最好在收到数据后马上检测**，比如检测除法是否除 0
        - **这时错误号可能也要马上更新**
    - **注意暂停机制，现在是把跟 `md` 相关的指令串行化，但是可能有更复杂的暂停控制**
    - 注意 `md` 存的数据恢复到以前时使用的机制
        - **有时可能数据寄存器不改也要保存，为了应对计算过程中可能出现的异常**

- 如果有新的 `dm` 存取方式
    - 如果是特殊的读写范围，那么因为是单周期，可以在 `dm` 上开端口 `mode`，让控制单元控制 `mode`，注意 sane defaults 和**小端序**
    - 如果是同时读写，那么也可以用上面的方法，注意 `dm` 的读写地址端口是分开的，注意开 MUX 的端口和 sane defaults
    - 如果是根据其它来源读写，注意开 MUX 的端口和 sane defaults
        - **注意这样的话转发多了一个新的消费者**
    - 注意在 `ac` 里允许这种方式，**并针对 4 种出错情况做进一步的适配**

- 如果有新的设备
    - 注意把设备的基地址抽象出来
    - 注意在无效地址和无效写入模式的时候忽略操作，因为**`ac` 会检测出来并报告**
    - 注意在 `ac` 里允许新设备的范围，**并针对 4 种出错情况做进一步的适配，尤其要是配好权限**
    - 注意设备的时序，**尤其是状态机的时序，必要的时候可以先画出状态转移图**
    - 如果设备能产生 IRQ，**注意把设备的 IRQ 接线给 `bridge` 接上**
    - 注意设备的功能和硬件的对应关系
    - 注意如果设备有固件之类的 blob，可以写自动生成脚本

- 如果有新的 `cp0` 异常
    - **注意按照异常的具体情况做进一步适配，有时不是单纯地按照级判断**
    - 注意加上新的异常的定义
    - **如果要扩展新的 `cp0` 寄存器，看情况改 `MARS\_COMPAT` 模式里的寄存器值，有的时候其实也不能改**
    - **注意新的 `cp0` 寄存器更新的优先级顺序，一般是异常 > 写入 > 正常更新，但是注意 `hwirq` 这样的寄存器更新逻辑就不一样**
    - **如果有新的 `cp0` 寄存器要读或者用到了 `cp0.epc`，注意双重转发和 `JUMP\_C0` 类指令的 D 级插入气泡效果**

- 如果有新的 `rf` 的值
    - 注意要接线接过来，然后加 MUX

        - **注意补上每级的 `pff` 和对应的 `wire`，一定要声明，否则默认是 1 位的**
    - 如果是返回地址，**最好是先接当前 PC，然后无符号数 +8**
        - **一般这种指令可以归约到 `JUMP_[IR]` 里**
    - 注意 `rf` 的值是否写入可以跟 `rf.we` 配合，**也可以妙用写入 `$0`**
    - **注意 `rf` 的值是否写入一般跟 `cp0.have2handle` 无关**

如何有效调试

1. 定位出错指令

- 平时可以用 `diff`
- 在考场上主要靠看数据和猜
    - 看数据大法
        - 前面一堆 0 位或者 1 位出错的，一般是移位指令
        - 前面数据乱了的，一般是乘除法指令
        - 数据差 1 的，一般是条件设置指令
        - 数据有些位有差别的，可能是读写 `cp0` 的指令
    - 瞎猜大法
        - 最近加了什么指令
        - 哪条指令原理不确定
        - 哪条指令是说了的重点
        - 哪条指令比较复杂，不好实现
        - 课下测试一直没过哪条指令
    - 看 PC 大法
        - 看是 `.text` 还是 ISR
    - 实在不行就把感觉错了的指令都检查一遍
- 因为 p8 涉及到软硬件协同，所以也可以用软件的方法
    - 模块化
    - 防御性编程
    - 踏实掌握指令的意义
    - 做好 debug 机制的建构工作
    - 十分有必要时重写

2. 分析每级的行为

- 先把 RTL 在心里分解成每级
- 然后比较出错指令或者觉得出错指令的差异
- 然后看转发和暂停是不是写对了
    - 首先检查**每级得出的寄存器结果、Tuse 和 Tnew**
    - 然后检查控制器的转发是否写对
        - **注意双重转发**

    - 然后检查数据通路的转发是否正确反映了逻辑
    - 最后检查一下转发相关的接线

- 然后检查每级的行为
    - 先检查控制信号对不对，**尤其是新加的控制信号和它们对应的 defaults**
        - 如果上面检查了的话，转发和暂停检查一下 defaults
    - `npc` 看与 `cmp` 的配合和 `npc` 模式本身的实现，注意**大小比较、有无符号数和指令取立即数的扩展**
    - `rf` 看取寄存器号对不对，不要乱改改折了，**注意 MUX、数据位宽、有无符号数和扩展模式**
        - **检查一下对应的控制信号**
    - `ext` 看扩展模式对不对，**注意扩展的是哪些位和扩展模式**
    - `alu` 看实现的运算对不对，**要踏实地看定义以及和 `rf` 的配合，不要读哪个寄存器都读错了**
        - **如果有第三个参数，检查一下关于第三个参数的逻辑**
    - `md` 看实现的运算和错误检查机制、复位机制对不对，**要踏实地看定义以及内部寄存器的值，注意好时钟周期**
        - 注意 `md` 的内部状态机和错误检测机制
    - `dm` 看实现的读写模式和读写地址对不对，**注意端序和读写地址的对应 MUX，和它们与控制信号的对应关系**
        - 注意可能读写地址要分开转发，**这里的接线需要仔细看然后调一下**
    - `bridge` 看实现的设备的基址和 `hwirq` 的接线对不对，**注意地址不能有重叠的情况，题目不会给地址重叠的情况，因为这样不合理**
        - 注意还有 `hwirq` 到 `cp0` 的接线
    - `ac` 看 4 种情况是不是更新到了能够反映现在设备拓扑的情况，**尤其是设备的 MMIO 地址及其访问权限**
        - 注意 4 种情况一定要全面，可以重叠
    - `cp0` 看 `have2handle` 能不能正确复位流水线、加上钩子，以及新的异常号能不能被识别
        - **注意来了异常会被识别，但是没有异常 `cp0` 应该更新 `hwirq` 等寄存器，下一个时钟周期发中断信号**
        - 注意 `cp0` 没有异常的时候不能乱发 `have2handle`
    - 新加入的设备看实现的设备状态机，还有设备写入值的逻辑
        - **注意写入值如果有错误就忽略，让 `ac` 来检测**
        - **注意设备功能和硬件的对应关系**
        - **注意设备内 `blob` 的生成逻辑**
        - **注意可以使用硬件上的 `debug` 机制**
    - `rf` 还要看实现的钩子对不对，**尤其是根据寄存器值判断的那部分，因为需要控制器配合**
        - **注意要先实现钩子再转发**

3. 分析指令之间的关系

- 跟上一条指令之间的关系
- 跟软硬件协同的关系
- 如果是跳转指令，跟以前指令和对应寄存器的关系
- 如果是 L/S 指令，跟内存的关系
- 如果是乘除法指令，跟乘除法器及其串行化的关系
- 如果是 ISR 指令，跟受害指令、受害指令的状态、cp0 的状态都有关系
- CPU 的初始状态

如何改数据通路

1. 分析为什么要改数据通路

- 改数据通路代价比较大
    - 加入流水线后更是如此，**转发、暂停、流水线寄存器都要重新 evaluate 一遍**
    - 加入异常机制以后更是如此，**出异常、来了异常以后的 hook 机制也都要重新 evaluate 一遍**
- 必须安装新部件吗?
    - 可能有的部件可以通过 hook 来解决
    - 有的部件可以通过改部件本身的方式来解决
    - 可能可以重新把指令的 RTL fit 到现有的数据通路里
    - 可能可以直接 hook 控制机制
    - 如果要求加新部件，那必须加，没有办法
- 新部件部署在哪一级?
    - **直接决定转发、暂停和流水线寄存器的 evaluation**
    - **也决定出异常和来了异常以后的 hook 机制的 evaluation**
    - 对类比同级的 MUX 和部件有帮助
    - 对分清这个部件的功能有描述
- 改了新部件，如何既服务好新指令，又能与原来的指令兼容?
    - sane defaults
    - 弄好回退机制
    - 原来的指令可能需要在控制层面避开新部件带来的影响，不过这一点一般不大可能
    - 对 md 这种自带状态机的部件，弄好状态机

2. 分析怎么改数据通路

- 如果没安装新部件
    - 如果在 npc 这里加上 hook 机制，记得跟 cmp 和控制配合好，**扩展指令的立即数时，源、目的和扩展哪个部分一定要注意**
    - 如果在控制加上 hook 机制，**要注意 sane defaults 和 hook 机制能不能方便之后修改代码**
    - 如果在 `alu / ext / md / dm` 加了新功能，**注意实现是否正确，要紧扣定义**
    - 可能需要在 cp0 处加上新的异常代码 / 寄存器，而且要识别异常，**注意保持 cp0 原来正常的机制**
- 如果安装了新部件

- 重构一遍新指令的数据通路，**注意 sane defaults**
- **evaluate 一遍转发、暂停和流水线寄存器，并且仔细地改转发和暂停规则**
    - **看一下有没有多重转发，有的话可以暂停也可以多重转发，不过一般这不大可能**
- **evaluate 一遍出异常和来了异常以后的 hook 机制**
    - **尤其是产生的异常和来了异常以后的 hook 机制，这跟时序和状态机都有关系**
    - **保证 cp0 原来正常的机制**
- 分析一下新部件的功能
    - 尤其是新设备，**注意软硬件协同**
- **看一下数据通路的更改，注意跟流水线寄存器之间的微妙的关系**
- 如果比较有空，可以稍微测试一下

如何比较方便地改设计

1. 可以在加 hook 机制的时候认为这是必须的
2. 可以在扩展时认为这样可以简化数据通路
3. 可以在部署部件时部署到比较方便实现的级
4. 可以在实现内部流水线时认为这样方便调试
5. 可以在暂停时认为这样可以简化设计