计算机组成原理实验报告

目录

1 NPC

1.1 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。

1.2 接口定义

表 1 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

1.3 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。

表 2 宏定义

类别	定义	值	意义
jump_mode	NPC_JUMP_DISABLE	4’b0000	不要跳转
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_NOT_EQUAL	4’b0010	当输入的比较结果不等时跳转
jump_mode	NPC_REG	4’b1111	按照寄存器内地址跳转

类别	定义	值	意义
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
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。

1.4 功能

令跳转基准地址 base = $unsigned(curr_pc)。

若 jump_mode == NPC_JUMP_DISABLED，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_DISABLED 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_DISABLED 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_DISABLED 时相同的步骤。

1.5 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。

2 PC

2.1 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。

2.2 端口定义

表 3 端口定义

端口	类型	位宽	功能
clk	输入	1	时钟信号

端口	类型	位宽	功能
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

2.3 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。

表 4 宏定义

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

2.4 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，但是把最低两位无条件清零。

2.5 注意事项

1. **比较 PC 是无符号数比较**

3 指令存储器

3.1 原理

指令存储器是存储程序指令的地方。

3.2 端口定义

表 5 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
valid	输出	32	PC 是否有效

3.3 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。

表 6 宏定义

类别	定义	值	意义
enable	IM_ENABLED	1'b1	IM 使能
enable	IM_DISABLED	1'b0	IM 非使能
addr	IM_ADDR_WIDTH	13	addr 的位宽
addr	IM_START_ADDRESS	2048	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

3.4 功能

有 IM_SIZE 个 32 位存储器，代表其中存储的指令。内部使用 Block RAM，已经加载好了初值。

无论什么时候，若地址在范围里且对齐，则 valid = 1'b1，否则 valid = 1'b0。

若 valid == 1'b0 或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 == 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）。

3.5 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. 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
4. valid == 0 已经过滤了不合法的情况，所以可以直接取地址的一部分，用这个来找字。
5. valid == 0 时，有地址不对齐的情况，这时输出 nop。这主要是为了帮助中止流水线，也是对取指不对齐异常处理的一部分。等到地址不对齐的那条指令（**取到的是 nop**）进入 M 级，就会被 cp0 检测到，然后进入 ISR。
6. **im 使用 Block RAM，所以使用二倍频时钟信号。**

4 寄存器堆

4.1 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。

4.2 端口定义

表 7 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

4.3 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。

表 8 宏定义

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_DISABLED	1'b0	寄存器堆非使能
输出	RF_OUTPUT_FORMAT	"%d: 0x%08x => 0x%08x"	输出模板

4.4 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。

4.5 注意事项

1. 寄存器可以定义为 reg [31:1] registers [31:0]，把 $0 空出来。

5 比较模块

5.1 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。

5.2 接口定义

表 9 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

端口	类型	位宽	功能
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

5.3 宏定义

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

5.4 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。

6 扩展器

6.1 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。

6.2 接口定义

表 10 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

6.3 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。

表 11 宏定义

类别	定义	值	意义
mode	EXT_MODE_SIGNED	3'b000	符号扩展
mode	EXT_MODE_UNSIGNED	3'b001	无符号扩展
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_MODE_ONE	3'b011	在数字前面填充二进制 1 的扩展

6.4 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。

7 ALU

7.1 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。

7.2 端口定义

表 12 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

端口	类型	位宽	功能
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。

7.3 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。

表 13 宏定义

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

类别	定义	值	意义
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'b11010	寄存器为参数的算术右移位运算
op	ALU_ADDU	5'b00101	不考虑溢出的加法运算
op	ALU_SUBU	5'b00110	不考虑溢出的减法运算
.*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。

7.4 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，若运算是 ADD / SUB，且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。在运算为 ADD / SUB 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。如果它的符号位与它的第二高位相同，就说明没有溢出，否则有溢出。这是因为有符号加减法溢出在 32 位范围内都是上溢或者下溢，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。把实际结果的符号位和第二高位一比较，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。

7.5 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

8 数据存储器

8.1 原理

数据存储器是存储数据的地方。

8.2 端口定义

表 14 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	读取的地址
write_addr	输入	32	写入的地址
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
stop	输入	1	阻止写入
read_result	输出	32	读到的结果
valid	输出	1	读写是否合法

8.3 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。

表 15 宏定义

类别	定义	值	意义
write_enable	DM_WRITE_ENABLE	1'b1	DM 使能
write_enable	DM_WRITE_DISABLE	1'b0	DM 非使能
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
数据存储器	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

8.4 功能

该部件为时序部件。

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，或者地址在 DM_ADDR_LB 和 DM_ADDR_UB 形成的闭区间内，且满足对应模式的对齐要求，则操作合法，否则操作不合法。合法性由 valid 信号表示。

任何时候，若 valid == 1'b0，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。

dm 内部使用 Block RAM。

8.5 注意事项

1. 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。合法性还是由 valid 信号保证的。
2. dm 内部为了能够适应 bram，实际上还是采用了 bitmask 的方式来标记写入哪些字节，然后再根据这个 bitmask 确定对应字的新值。
3. stop 信号是为了更好地让中止流水线的硬件机制给 dm 加钩子，因为一旦有硬件中断，需要关闭 dm 写使能。但是直接关闭 dm 的写使能，又会影响对地址合法性和异常号的判断。而且，在只有 dm 写使能激活才能判断出是异常的情况下，还会使信号震荡，仿真进入死循环。因此，只能通过另一个信号（这里是 stop）来阻止 dm 写入。同时，用 stop 信号还可以把真正的 dm 写使能暴露出来，方便调试。
4. DM_ADDR_WIDTH 和 DM_ADDR_SIZE 要一块改。
5. **CPU 是小端序的。**
6. 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。
7. **DM 使用 Block RAM，所以使用二倍频时钟信号。**

9 设备桥

9.1 原理

设备桥是管理 MMIO 中地址翻译和映射，以及设备的中断请求的部件。它负责把地址进行分配，把 CPU 中需要的 MMIO 地址翻译成设备的内部地址，把 CPU 的设备写使能进行分配。设备的中断请求和 CPU 和设备之间的读写也由设备桥进行桥接。

它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。

9.2 端口定义

表 16 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
write_enable	输入	1	CPU 决定的设备写使能信号
dm_mode	输入	3	dm 的模式
valid	输出	1	读写是否合法
stop	输入	1	阻止写入
addr	输入	32	地址信号
write_enable	输入	1	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[1]	硬件中断请求输出
*device-name*_addr[2]	输出	32	*device-name* 的读写地址
*device-name*_write_enable	输出	32	*device-name* 的写使能
*device-name*_write_data	输出	32	*device-name* 要写入的数据
*device-name*_read_result	输出	32	*device-name* 的读取结果
*device-name*_irq	输出	32[3]	*device-name* 的中断请求信号

注：

1. 下标是 [7:2]，这是为了跟课下要求兼容。
2. 合法的设备名有：timer, uart, switches, led, nixie, buttons。它们视情况可以以全大写或全小写的方式代替 *device-name*。
3. **在顶层模块中可以对设备地址有适当的截取，bridge 不截取。**

9.3 宏定义

、

表 17 宏定义

类别	定义	值	意义
设备地址	BRIDGE_*device-name*_LB	32'h00007f00	*device-name* 的地址下界
设备地址	BRIDGE_*device-name*_UB	32'h00007f0b	*device-name* 的地址上界

9.4 功能

首先确定当前设备。如果地址落在某个设备的地址范围内部，就确定了是哪个设备。

然后确定读写操作合法性，由 valid 信号表示。若 dm_mode == DM_NONE，或读写的是两个 timer 里的一个且 dm_mode == DM_W 且地址按字对齐且不是写入 timer 的 count 寄存器对应的地址，或读写地址在其它设备地址范围内，则读写操作合法，否则读写操作不合法。

如果 valid == 1'b1 且 write_enable == 1'b1 且 stop == 1'b0 且地址落在 *device-name* 的地址范围内，就令 *device-name* 的写使能激活，它的 read_result 为要输出的 read_result，它的内部地址为 addr 作为无符号数减去对应的基址。其余的设备屏蔽写使能，内部地址为 0，read_result 丢弃。

否则，read_result 为 32'b0。

无论什么时候，clk、rst 和 write_data 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，和简化写入数据的接线。

无论什么时候，hwirq = {buttons_irq, nixie_irq, led_irq, switches_irq, uart_irq, timer_irq}。

9.5 注意事项

1. **15 号设备（curr_dev 表示的）代表没有设备。dm 会自己确定地址是否落在它自己的地址范围内，不需要 bridge 关心。**
2. **地址上下界以字节为单位，而且是包括边界的。**

10 CP0

10.1 原理

CP0 是 0 号协处理器的意思，它目前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。

为了能够比较方便地处理异常及其相关转发，以及更好地实现精确异常，CP0 部署在 M 级。

10.2 端口定义

表 18 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

10.3 宏定义

表 19 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）

类别	定义	值	意义
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	（未定义）	cp0 是否工作在跟 MARS 兼容的模式下

10.4 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。

表 20 功能

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。

表 21 功能

小寄存器	位宽	初值	意义
allow_hwirq	6	6’b0	允许对应硬件的中断请求，作为掩码使用

小寄存器	位宽	初值	意义
exl	1	1’b0	异常级别，在 ISR 时为 1
g_allow_hwirq	1	1’b0	全局地允许或禁止硬件的中断请求
in_bds_i	1	1’b0	内部保存的受害指令是否在延迟槽中
hwirq_i	6	6’b000000	内部保存的硬件中断请求情况
exc_i	5	5’d0	内部保存的异常代码
epc_i	32	32’h0	内部保存的从 ISR 中返回需要用到的 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，则说明 cp0 需要处理异常或者中断了，如果两者同时发生，优先处理异常。首先设置 exl 说明进入了 ISR，然后记下 in_bds_i，意思是延迟槽指令受害，还有记下 epc。**记下 epc 时，要按字对齐，因为还有取指时要取的地址没按字对齐的情况。**也就是 exl <= 1'b1; in_bds_i <= in_bds; epc_i <= (in_bds == 1'b1) ? $unsigned({curr_pc[31:2], 2'b0}) - $unsigned(4) : $unsigned({curr_pc[31:2], 2'b0})。

此时，若 have_irq == 1'b1，则 exc_i <= EXC_INT，否则若 have_exc == 1'b1，则 exc_i <= exc。这就是优先处理中断，如果中断和异常同时发生，先在中止流水线的时候把中断丢弃。

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，若 have2handle == 1'b0，则说明没有中断或异常要处理。这时就可以检查 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，若 have2handle == 1'b0 且 write_enable == 1'b0，也就是没有中断或异常要处理，也不写入 cp0 的内部寄存器，则检查其它信号。若 exit_isr == 1'b1，则 exl <= 1'b0。也就是说，让 eret 指令进入 M 级时，cp0 把异常级别设成 1'b0，它自己在模式的意义上退出 ISR。

在每个时钟上升沿，若 ~(have2handle == 1'b0 && write_valid == 1'b1 && addr == 5'd13)，则 hwirq_i <= hwirq。它的意思是，如果不是没有中断和异常要处理而且要写 cause 寄存器的话，就更新它里面的 hwirq_i 小寄存器。这是为了让 CPU 跟硬件更好地同步，也就是同步地记录下 hwirq 的最新情况。

10.5 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 的信号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. 在 MARS 兼容模式下，把 prid 做成 32'h0。
5. 在 MARS 兼容模式下，没有不写 cause 且不需要处理中断就更新 hwirq_i 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_i 紧跟着的后面两位，也就是 cause[9:8]），因为 MARS 有这个机制。
6. **实际上更新 hwirq_i 在进入中断的那个周期，也就是 have2handle == 1'b1 那个周期时更新不很现实，因为进入中断需要一个周期，这个周期后流水线被中止，M 级是 nop。所以，还可以继续更新 hwirq_i，哪怕进入中断那个周期更新了 hwirq_i，也会被最新的值覆盖掉，况且 LOAD_C0 类指令进来最早也需要 3 个周期。have2handle == 1'b0 可以不用判断，因为如**

果 have2handle == 1'b1，write_enable 早就在效果上被屏蔽了。

11 流水线寄存器

11.1 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。

11.2 端口定义

表 22 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

11.3 参数定义

表 23 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

11.4 宏定义

表 24 宏定义

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

11.5 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。

11.6 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，也是为了和整个 CPU 的同步复位机制配合。

12 MUX

12.1 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。

12.2 类别

MUX 有多个类别。有 2 路 MUX、3 路 MUX 以至于多路 MUX。

12.3 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*。

12.4 宏定义

暂无

但是仍然保留 `mux.h` 宏文件并填入模板，以备以后使用。

12.5 参数定义

表 25 参数定义

参数	默认值	功能
BIT_WIDTH	32	输入和输出数据的位宽

12.6 端口定义

表 26 端口定义

端口	类型	位宽	功能
control	输入	[1]	输入控制信号
result	输出	BIT_WIDTH	输出数据
input*n*	输入	BIT_WIDTH	[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**。

12.7 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。

12.8 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。**

13 流水线 CPU 数据通路

13.1 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。

流水线中还可能出现异常和中断，以及对设备的操作。所以，要加上对应的钩子机制，和 CPU 与设备桥的通信总线。

13.2 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。

表 27 分析

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。**

13.2.1 F 级（IF）

表 28 F 级（IF）

数据通路类型	F: npc.curr_pc	F: npc.cmp_result	F: npc.cmp_sig_result
BRANCH	F: pc.curr_pc	D: cmp.cmp	D: cmp.sig_cmp
（其它）	F: pc.curr_pc		
综合	F: pc.curr_pc	D: cmp.cmp	D: cmp.sig_cmp

表 29 F 级（IF）

数据通路类型	F: npc.num	F: npc.jnum	F: npc.reg
BRANCH	D: im.result[15:0]		
JUMP_I		D: im.result[25:0]	
JUMP_R			D: rf.read_result1
（其它）			
综合	D: im.result[15:0]	D: im.result[25:0]	D: rf.read_result1

表 30 F 级（IF）

数据通路类型	F: npc.epc	F: pc.next_pc
JUMP_C0	M: cp0.epc	F: npc.next_pc
（其它）	M: cp0.epc	F: npc.next_pc
综合	M: cp0.epc	F: npc.next_pc

13.2.2 D 级（ID）

表 31 D 级（ID）

数据通路类型	D: ext.num	D: cmp.reg1	D: cmp.reg2
CAL_I	D: im.result[15:0]		
LOAD	D: im.result[15:0]		
BRANCH		D: rf.read_result1	D: rf.read_result2

数据通路类型	D: ext.num	D: cmp.reg1	D: cmp.reg2
CMOV		D: rf.read_result2	
（其它）			
综合	D: im. result[15:0]	D: rf.read_result1	D: rf.read_result2

13.2.3 E 级（EX）

表 32 E **级**（EX）

数据通路类型	E: alu.num1	E: alu.num2
CAL_R	D: rf.read_result1	D: rf.read_result2
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
CMOV	D: rf.read_result1	D: rf.read_result2
（其它）		
综合	D: rf.read_result1	D: rf.read_result2 D: ext.result

表 33 E **级**（EX）

数据通路类型	E: alu.shamt	E: md.dh	E: md.dl
CAL_R	D: im. result[10:6]		
CAL_M		D: rf.read_result1	D: rf.read_result2
STORE_M		D: rf.read_result1	
（其它）			
综合	D: im. result[10:6]	D: rf.read_result1	D: rf.read_result2

13.2.4 M 级（MEM）

表 34 M **级**（MEM）

数据通路类型	M: dm.read_addr	M: dm.write_addr	M: dm.write_data
LOAD	E: alu.result	E: alu.result	E: rf. read_result2
STORE	E: alu.result	E: alu.result	E: rf. read_result2
（其它）			
综合	E: alu.result	E: alu.result	E: rf. read_result2

表 35 M 级（MEM）

数据通路类型	M: cpu_addr	M: cpu_write_data
LOAD	E: alu.result	E: rf.read_result2
STORE	E: alu.result	E: rf.read_result2
（其它）		
综合	E: alu.result	E: rf.read_result2

表 36 M 级（MEM）

数据通路类型	M: cp0.addr	M: cp0.write_data	M: cp0.read_result
LOAD_C0	D: im. result[15:11]		D: rf. read_result2
STORE_C0	D: im .result[15:11]	D: rf. read_result2	

13.2.5 W 级（WB）

表 37 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	

数据通路类型	W: rf.write_data
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

13.2.6 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。

表 38 流水线寄存器

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

流水线级	信号	流水线寄存器名称
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、JUMP_I 和 JUMP_R 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。

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。

13.2.7 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。

表 39 数据通路 MUX

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

13.2.8 注意事项

1. 返回 PC + 8 实际上是通过流水 PC 再加 8 实现的。
2. **D 级流水线寄存器都要接使能信号，W 级以前的流水线寄存器都要接复位信号，因为要插入气泡和复位流水线。**
3. 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。
4. 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。
5. 如果写了（无），那么对应端口的数据为全 0，不过这时对应端口实际上也没有作用。
6. **m_bridge 是根据地址范围判断读取结果是哪个的，具体方法是直接看地址是否属于 dm 的地址范围，如果是就选择 M: dm.read_result 输出，否则选择 M: cpu_read_result 输出。信号来源里的 M: dm.read_result 是原来的 dm.read_result。这是为了方便对 bridge 实现钩子机制，同时简化对 dm 修改地址范围的工作量。**

13.3 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。

dm 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。

rf 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。

epc 同理，在同一个时钟周期内可能会有 STORE_C0 和 JUMP_C0 类指令同时读写 epc 的情况，所以 epc 也需要转发。

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，所以一般要判断 2 次。但是，对 epc 的判断除外。

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，一共有六处：D: rf.read_result1, D: rf.read_result2, E: rf.read_result1, E: rf.read_result2,

M: dm.write_data, M: cp0.epc。其中 E 级的两处是通过流水线寄存器暂存的，M 级的一处是直接存在 cp0 里的。这六处可以分四类。对每类需要构造的转发 MUX 总结如下。

表 40 转发

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

13.3.1 注意事项

1. **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。但是，epc 可以直接被读取，并参加转发，因为它只是一个寄存器，关键路径和流水线寄存器一样长，实际上并不长。**
2. **M: cp0.epc 的转发，也可以在 M 级转发。这是因为新的 epc 可以有来不及写入 cp0 的情况。**
3. **M: cp0.epc 是流水线时间上处在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。**

4. 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——

13.4 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来得及转发。

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。

13.5 异常处理

异常处理的整个过程，有几个大致阶段，对应的硬件机制也按照阶段分类。

中止流水线并摆脱副作用 首先是在流水线寄存器中建立同步复位机制，在能够存储状态的主要部件中建立中止提交机制，以及必要的回退机制，从而能让异常或者中断发生的时候能够中止整个流水线，并摆脱流水线中指令可能的副作用。

进入 ISR 然后，在 npc 中建立强制进入 ISR 的机制，这样就能在异常发生时马上进入 ISR。

传输 epc 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 中返回时需要考虑清空 JUMP_C0 类指令的后一条指令。同时，还需要考虑 JUMP_C0 类指令给 epc 带来的数据转发。

清除、中止和回退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。

14 指令识别机制

14.1 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。

14.2 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。

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。

表 41 宏定义

类别	定义	值	意义
寄存器号	ZERO	5'd0	0 号寄存器（或者表示某指令不需要或不能写入寄存器）
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）

14.3 端口定义

表 42 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	10	当前指令的具体类型

14.4 功能

获取当前指令的具体类型。返回的结果一共 10 位，前 5 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。

14.5 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。

14.6 指令字段

表 43 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

14.6.1 指令类型

表 44 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	

类别	定义	值	意义
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	

类别	定义	值	意义
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

类别	定义	值	意义
数据通路类型	JUMP_C0	4'b1110	按照 cp0 的内容跳转的指令

14.7 注意

1. 临时的数据通路类型都是从上往下长的。

15 控制

15.1 原理

控制是指通过识别指令，控制数据的流通和功能部件的功能，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。控制通过数据通路的分叉和已有的控制信号完成控制。

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。通过指令识别模块，可以分析指令，做到有效的暂停和转发。

处理异常和中断也需要用到控制模块。对 cp0 内部寄存器的控制，需要控制模块发起控制信号。对异常的判断，需要控制模块进行。异常和中断时中止流水线、摆脱副作用、强制进入 ISR，需要控制模块给正常时的控制信号加上钩子。离开 ISR 时对 JUMP_C0 类指令的后一条指令的无效化，和关于 epc 的暂停和转发机制，同样要用到控制模块。

15.2 端口定义

表 45 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

端口	类型	位宽	功能
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

表 46 端口定义

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）

表 47 端口定义

端口	类型	位宽	功能
f_im_valid	输入	1	输入 F: im.valid
e_alu_sig_overflow	输入	1	输入 E: alu.sig_overflow
m_dm_valid	输入	1	输入 M: dm.valid
m_bridge_valid	输入	1	输入 M: bridge.valid
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 （也就是遇到了异常或中断）

表 48 端口定义

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

表 49 端口定义

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

表 50 端口定义

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

表 51 端口定义

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
cw_e_md_stop	输出 1	控制 E: md.stop	
cw_e_md_restore	输出 1	控制 E: md.restore	

表 52 端口定义

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_stop	输出	1	控制 M: dm.stop
cw_m_dm_mode	输出	3	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

表 53 端口定义

端口	类型	位宽	功能
cw_w_rf_write_enable	输出	1	控制 W: rf.write_enable

端口	类型	位宽	功能
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

表 54 端口定义

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	4	控制 fm_w
cw_fm_epc	输出	4	控制 fm_epc

15.3 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。比如暴力转发和标记转发，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。

控制模块内部有 5 条流水线，每条流水线都有自己的意义。在一条流水线上，每个流水线级对应的寄存器，在有对应流水线级流水寄存器使能和复位信号的情况下，都听从对应信号的指令。这样能保证它们和数据通路中流水线寄存器的同步。

但是，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 级都需要指令本身，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。

记录寄存器读取和写入情况的 3 条流水线 这三条流水线都有 E 级、M 级、W 级寄存器。控制模块在内部流水每条指令需要读取的两个寄存器和写入的一个寄

存器，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。

异常 ID 流水线 这条流水线有 D 级、E 级和 M 级寄存器。控制模块在内部流水对应级指令的异常号，该异常号是由上一级指令的执行情况、对应级流水线寄存器的使能情况和复位情况共同得出的。在 M 级，也就是流水线的终点，还会有额外的判断（叫 M^+ 级，但是没有对应的流水线寄存器），从而能捕获到 M 级指令在当级发生的异常。所有的异常 ID 都流水到 M 级判断，这样就能实现精确异常。**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，同时取数范围错误。**

指令是否在延迟槽里的流水线 这条流水线有 D 级、E 级和 M 级寄存器。每级寄存器的内容是该指令是否在延迟槽里。这是给 `M: cp0.in_bds` 准备的。

15.4 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，也需要单独列表。

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。这一节提到的控制信号，都是正常的控制信号。

15.4.1 F 级

表 55 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：

表 56 F 级

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

注意事项

1. **F 级的控制信号是由 D 级指令控制的。**
2. **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。**

15.4.2 D 级（ID）

表 57 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：

表 58 D 级（ID）

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

注意事项

1. **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。**

15.4.3 E 级（EX）

表 59 E 级（EX）

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

CAL_R 类指令类型与 E: alu.op 的关系：

表 60 E 级（EX）

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：

表 61 E 级（EX）

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：

表 62 E 级（EX）

指令类型	E: alu.op
MOVZ	ALU_MOVZ

CAL_M 类指令类型与 E: md.op 的关系：

表 63 E 级（EX）

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：

表 64 E 级（EX）

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：

表 65 E 级（EX）

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

15.4.4 M 级（MEM）

表 66 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

数据通路类型	M: dm.write_enable	M: dm.mode	M: cp0.write_enable
（其它）	1'b0	DM_NONE	1'b0

LOAD 类指令类型与 M: dm.mode 的关系：

表 67 M 级（MEM）

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：

表 68 M 级（MEM）

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

注意事项

1. **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。**
2. **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。**

15.4.5 W 级（WB）

表 69 W 级（WB）

数据通路类型	W: rf.write_enable	W: m_regdata
CAL_R	1'b1	E: alu.result

数据通路类型	W: rf.write_enable	W: m_regdata
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

15.4.6 指令读写寄存器识别

比较显然的一点是数据通路类型大致决定指令要读写的寄存器号。所以，有时识别指令读写的寄存器，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 这一种情况。这种情况只要在转发逻辑上加入相应逻辑即可解决。

表 70 指令读写寄存器识别

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	[4]	ZERO
JUMP_I	ZERO	ZERO	[2]
JUMP_R	RS	ZERO	[3]
CMOV	RS	RT	[1]
CAL_M	RS	RT	ZERO
LOAD_M	ZERO	ZERO	RD

数据通路类型	reg1	reg2	regw
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

注意事项

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_I 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。

15.5 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。对通用寄存器的转发，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。**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 epc 来说，情况比较特殊，因为它是 M 级 cp0 的一个寄存器。特点有以下几点：

1. epc 的转发只需要处理一个寄存器，因为是为了加速 JUMP_C0 指令，才出现的转发。
2. 转发主要是给 F 级的 npc 使用，因为只有 npc 对 epc 有需要转发才能处理的需求。
3. 需要检测出数据通路中的 STORE_C0 是不是写入了 epc，并且把对应寄存器的值转发。这一点跟 rf 的转发类似，不过位置在后面。
4. epc 的关键路径实际上跟流水线寄存器一样长。所以，epc 可以直接从 cp0 导出。
5. **哪怕转发 $0 到 epc，都是有意义的，这意味着 epc 会被写成 32'b0。**
6. epc 可以从 E 级和 M 级转发，因为 D 级读出时牵扯到 rf，不能转发，而且 JUMP_C0 类指令到了 D 级才需要 epc。到了 W 级，STORE_C0 指令已经对 epc 提交了，所以不需要转发了。
7. **epc 是可以同级转发的，因为可能会碰到 D 级是 JUMP_C0 指令，而 M 级是写入 epc 的 STORE_C0 指令，epc 来不及写入的情况。**

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。

表 71 转发控制信号

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

类别	定义	值	意义
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

表 72 转发控制信号

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

表 73 转发控制信号

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

表 74 转发控制信号

类别	定义	值	意义
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

15.5.1 注意事项

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. epc 的值把需求者固定在 M 级，而不是像指令需要 rf 的值那样把需求者在哪一级当成指令在哪一级。这是因为 epc 的需求者可以看成是 cp0，本身就是固定的。这样也方便涉及到使用 epc 的功能扩展。
5. 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。

15.6 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。rf 的暂停机制，是通过 T_{use} 和 T_{new} 机制实现的。epc 的暂停机制，是通过对特例进行分析实现的。

15.6.1 通用寄存器的暂停机制

T_{use} 是指指令到 D 级以后还剩最晚多少个时钟周期就需要新值。T_{new} 是指指令还需要多少个时钟周期才能开始转发。因此只要 $T_{use} < T_{new}$，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。

但是，这种暂停机制没有考虑 md 带来的影响，这种影响会在下一节考虑。

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 T_{use} 和 T_{new} 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。锁定 pc 是通过 pc 使能，清空 E 级各个流水线寄存器是通过同步复位。所以，控制内部的流水线也会插入气泡。同时，异常流水线也会插入气泡。

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。

在 D 级各种数据通路类型的 T_{use} 如下。

表 75 通用寄存器的暂停机制

数据通路类型	T_{use} (read_addr1)	T_{use} (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 T_{new} 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。

表 76 通用寄存器的暂停机制

数据通路类型	T_{new} (E)	T_{new} (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 T_{use} 没有列出的，是因为它没有意义，认为 T_{use} 足够大。T_{new} 同理，但是认为 T_{new} 为 0。

这样，只要算出每个阶段的 T_{use} 和 T_{new}，并且保证发生数据冒险时对两个寄存器，$T_{use} \geq T_{new}$，就能保证转发成功，进而通过这个原理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。

暂停的具体实现是关闭 F 级 `pc` 使能，关闭 D 级流水线寄存器使能，使 E 级流水线寄存器复位。这样就能实现在 E 级插入气泡，同时 F 级和 D 级的指令被堵住，起到暂停的作用。

15.6.2 epc 的暂停机制

由于 epc 只有一种需要转发的情况，所以暂停不需要通过 T_{use} 和 T_{new} 机制来实现，太复杂了。

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 $T_{new} > 0$ 的情况。

这是因为有两重转发需要完成，同时完成这两重转发需要暂停。如果不考虑通用寄存器的数据冲突，那么 JUMP_C0 带来的 epc 的数据冲突只用转发就能解决，虽然 JUMP_C0 需要就在 D 级使用 epc。

但是，如果 E 级是 STORE_C0，那么 W 级有任何指令，都不会发生 STORE_C0 需要的 rf 值转发来不及的情况，也能把这个值接力转发到 epc。同样地，如果 E 级不是 STORE_C0 但 M 级是（两级都有的话显然只需要考虑 E 级），那么能转发给 M 级 STORE_C0 的也就只有 W 级指令，同样不会有转发来不及的情况。

最后只剩下一种大情况：D 级是 JUMP_C0，E 级是 STORE_C0，M 级和 W 级指令未知。W 级指令一定来得及转发；M 级指令 $T_{new} = 0$ 时也来得及转发。但 M 级指令 $T_{new} = 1$（即 $T_{new} > 0$）呢？只看 rf 的数据冒险，当然转发来得及。但是 D 级的 JUMP_C0 马上就要用了，来不及接力转发给 epc 了。所以，在这种小情况下，必须暂停。

解决方案和通常的暂停一样，也是在 E 级插入气泡。这样，D 级是 JUMP_C0，M 级是 STORE_C0，W 级是那条 $T_{new} > 0$ 的指令，就可以接力转发了。

15.6.3 注意事项

1. **T_{new} 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，不能从功能部件直接转发。**
2. 控制内部的流水线也要插入气泡。
3. **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。**

15.7 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 W 级的总共三个地址端口输入指令识别机制相应的结果即可。

15.8 设备访问

15.8.1 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，通过按字节编址的编址方式操作设备的内部寄存器。

读写的地址验证，是 dm 和 bridge 模块分开进行的。这样，dm 和 bridge 自己就知道操作合不合法，当场就能屏蔽不合法的操作，等着异常处理机制来处理。同时，dm 和 bridge 模块分开处理异常，还可以把对内存和设备访问的异常检测，统一到两个设备中去，方便维护。

15.8.2 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 dm 的地址范围决定的。地址在 dm 范围内，就是 dm 的读取结果，否则是 bridge 的读取结果。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。

定义原来的 M: dm.read_result 和 cpu_read_result 对应的选择器选择信号分别为 1'b0 和 1'b1，按照这种选择逻辑选择相应的信号。

15.8.3 注意事项

1. **地址比较要用无符号数比较。**

15.9 异常和中断处理

15.9.1 异常 ID 流水线

控制模块内有一条异常流水线，以流水线的形式保存当前级指令进行操作之前获得的异常 ID。没有异常的异常 ID，以 Int 表示。它的值也正好为 0，代表没有异常。

每级异常流水线寄存器的更新，是参考上一级功能部件的某些代表执行结果的端口和指令类型决定的。和普通的流水线寄存器一样，它们除了需要首先处理同步复位，然后处理使能关闭的情况以外，更新的规则如下表。

表 77 异常 ID 流水线

流水线级	有关端口	更新规则
D	F: im.valid	若 D 级为 JUMP_C0，则为 EXC_NONE； 否则，若 f_im_valid == 1'b0，则为 EXC_ADEL； 否则，为 EXC_NONE
E	（无）	若 D 级为未知指令，则为 EXC_RI； 否则，为 D 级内容
M	E: alu.sig_ overflow	若 E 级为 CAL_R / CAL_I 且 e_alu_sig_overflow == 1'b1，则为 EXC_OV； 否则，若 E 级为 LOAD 且 e_alu_sig_overflow == 1'b1，则为 EXC_ADEL； 否则，若 E 级为 STORE 且 e_alu_sig_overflow == 1'b1，则为 EXC_ADES； 否则，为 E 级内容
M^+	M: dm.valid M: bridge.valid	若 M: dm.valid == 1'b0 且 M: bridge.valid == 1'b0， 则若 m_dm_write_enable == 1'b0， 则为 EXC_ADEL，否则为 EXC_ADES； 否则为 M 级内容

注意事项

1. **这种机制是用更新级的异常，覆盖更老级产生的异常。所以，如果有产生异常就不能更改的情况，需要进行特判，在后面的所有流水级把这种异常固定住。**
2. **EXC_NONE == EXC_INT == 0。这种设计可以保证有中断时，可以直接明白是中断引起的进入 ISR。同时，也方便异常流水线的复位。**
3. **M^+ 级是为了能让 CPU 检测到在 M 级进行操作时产生的异常，因为异常流水线里的 M 级寄存器保存的是 M 级指令进行 M 级操作以前获得的异常号。M^+ 级没有对应的流水线寄存器，就在当级产生结果，然后作为 cp0 的输入。**

15.9.2 检测异常和中断

检测异常和中断，是由异常流水线、设备桥和 cp0 配合完成的。

cp0 的部署位置 cp0 部署在 M 级，因为指令在 W 级运行只是写回寄存器，不会产生异常。因此，所有的异常和中断都可以在 M 级检测，cp0 也部署在 M 级。

异常信号的输入 异常 ID 流水线是为了处理指令运行中的异常信号而设计的，它在 M 级能够得出一条指令在运行到最后获得的异常 ID。

中断信号的输入 中断信号也在 M 级输入，因为 cp0 就在那里，异常和中断由它集中处理。中断信号通过 bridge 输入。

异常和中断的检测 见第 10 节 cp0 的相关功能。cp0 分别把 M^+ 级的异常 ID 和从 bridge 传来的中断信号作为输入，然后按照它的功能判断下个时钟周期是否需要处理异常和中断，判断结果用 M: cp0.have2handle 表示。

15.9.3 异常或中断发生时 CP0 的内部记录控制信号

在 have2handle == 1'b1 时，说明下个时钟周期要进行异常和中断的处理。此时，CP0 需要一些控制信号作为输入，来明白它需要在内部记录什么内容。需要的控制信号如下表。

表 78 异常或中断发生时 CP0 的内部记录控制信号

控制信号名称	位宽	产生值的规则
cw_m_cp0_in_bds	1	若 m_pc_curr_pc != 32'b0， 则为指令是否在延迟槽里的流水线 M 级的值； 否则，若 e_pc_curr_pc != 32'b0， 则为指令是否在延迟槽里的流水线 E 级的值； 否则为指令是否在延迟槽里的流水线 D 级的值
cw_m_cp0_curr_pc	32	若 m_pc_curr_pc != 32'b0， 则为 {m_pc_curr_pc[31:2], 2'b0}； 否则，若 e_pc_curr_pc != 32'b0， 则为 {e_pc_curr_pc[31:2], 2'b0}； 否则为 {d_pc_curr_pc[31:2], 2'b0}

注意事项

1. **cp0 需要找到当前还没有执行完的指令的 pc 值，但 W 级的更改会被提交，所以要从 M 级找起。找的时候要忽略流水线中插入的气泡，而气泡中流水的 curr_pc 是 32'b0，所以碰到气泡时需要往前找。类似地，找 in_bds 时也需要忽略流水线中插入的气泡，找到的 in_bds 也类似，碰到气泡时也一样需要往前找。**
2. **exl == 1'b1 时，并不会屏蔽异常。所以，可能会有在 ISR 中，又重新进入 ISR 的情况。但是，即使在这种情况下，由于 JUMP_C0 不会产生异常，所以哪怕某条指令到了 M 级产生异常，cw_m_cp0_curr_pc 也是合法的值。**
3. **往前找 curr_pc 和 in_bds 时，找到 D 级就可以了，因为只有 JUMP_C0 类指令才能把 D 级清空，但是那时 JUMP_C0 类指令在 E 级，E 级 pc 不是 32'b0。而且，都用流水的 curr_pc 回溯可以保证这两个的值是同一级指令得出来的值。**
4. **cp0 记录 epc 时，根据第 10 节，它会自动检测是否为延迟槽指令。如果是，存入的 epc 会作为无符号数减 32'd4。**
5. **这里给 cp0 存入的 epc 是对齐的，或者说在中断异常发生时 cp0 存入的 epc 一定按照字对齐。但是，为了方便对 cp0 内部寄存器的自由存取，以及把对异常和中断的响应和对 cp0 的控制更好地分离，存入 epc 的工作由控制模块来做。**

15.9.4 精确异常

精确异常是指引起异常的指令的前序指令都完全地、正确地对 CPU 的内部状态提交了更改，但是引起异常的指令本身和其后的指令，执行效果不影响 ISR 返回后所执行指令的执行效果。通俗地讲，就是流水线中的每条指令要么完全执行，要么就像根本没执行一样。用另一种说法，就是把流水线在异常和中断打断时的指令执行流程扁平化，只打断一条指令的执行过程并撤销后续指令（无论是否在流水线中）的执行结果，执行 ISR 后再恢复正常执行。

引起异常的指令叫做受害指令。对中断也可以指定受害指令，就是从 M 级开始向前找到 D 级，第一个非气泡的指令。**受害指令的 pc 会有，但是不一定对齐，也不一定在正常 pc 范围内。**

从精确异常的定义可以看出，如果保证不跳转到 32'b0，那么 cw_m_cp0_curr_pc 就是根据精确异常定义可以得出的受害指令。

引起异常的指令引起的异常会在 M 级被处理，所以在流水线中只有 W 级指令需要提交更改。也就是说，有中断和异常要处理时，提交点正好在 W 级流水线寄存器以后。这是因为 W 级流水线寄存器本身的内容，在下一个时钟上升沿后也会被清空，以免 M 级流水线寄存器的信息流水到 W 级流水线寄存器。但是这个时钟周期 W 级流水线寄存器的内容会被提交，因为 W 级指令是受害指令以前的指令，所以才有以上结果。

15.9.5 进入 ISR

有异常或者中断时，就需要进入 ISR。进入 ISR 是异常和中断处理的重点。当且仅当此时，m_cp0_have2handle == 1'b1，所以也可以把 m_cp0_have2handle 当做要进入 ISR，也就是说下个时钟上升沿后就进入 ISR 的信号。

根据精确异常的原则，进入 ISR 需要撤销流水线中后续指令对流水线状态的更改。组合逻辑并不记住状态，所以重点在于对时序部件的操作。更直白地说，就是对时序部件的寄存器的操作，因为只有它们存储着 CPU 的内部状态。

具体地说，一方面要撤销已经提交的更改，因为有些指令在流水线中就对时序部件的寄存器进行了更改；另一方面要阻止未提交的更改，因为有些指令在这个时钟周期准备更改时序部件的寄存器，但是还没有提交。还有，就是一定要无条件地跳转到 ISR。这三种操作，可以通过对一系列控制信号的操作和超驰来完成，如下表。

表 79 进入 ISR

流水线级	控制信号名称	具体操作	意义
D	cw_d_pff_rst	超驰成 1’b1	放弃对 D 级流水线寄存器的提交
E	cw_e_pff_rst	超驰成 1’b1	放弃对 E 级流水线寄存器的提交
M	cw_m_pff_rst	超驰成 1’b1	放弃对 M 级流水线寄存器的提交
W	cw_w_pff_rst	超驰成 1’b1	放弃对 W 级流水线寄存器的提交

流水线级	控制信号名称	具体操作	意义
E	cw_e_md_restore	M 级为 STORE_M 则设为 1'b1	以便让 M 级的 STORE_C0 在 E 级时的提交被撤销
E	cw_e_md_stop	E 级或 M 级为 CAL_M，以及 E 级为 STORE_M，则设为 1’b1	以便让 W 级以前的 CAL_M 停止运算，以及让 E 级的 STORE_M 停止提交
M	cw_m_dm_stop	设为 1’b1	以便让 M 级指令放弃对 dm 和 bridge 的提交
M	cw_m_cp0 _write_enable	超驰成 1’b0	以便让 M 级指令放弃对 cp0 的提交
F	cw_f_pc_enable	超驰成 1’b1	以便让下一个周期的 pc 变成 IM_ISR_ADDRESS
F	cw_f_npc_jump_mode	超驰成 NPC_ISR	以便跳转到 ISR 起始地址

通过这样的表格描述出来的进入 ISR 的机制是可重入的，也就是说在 ISR 中因为有异常或者中断要处理，重新进入 ISR 是可能的，一样能保证对于 ISR 满足精确异常的原则。实际上，由于 exl == 1'b1 时不屏蔽异常，在 ISR 中有异常要处理是可能的。

15.9.6 从 ISR 返回

从 ISR 返回是一个相对平缓的过程，没有强制清除状态这一比较激烈的步骤。因为按照精确异常的要求，要通过 epc 返回到的指令只要逐步进入流水线就可以了。它分成以下几个按时间顺序说明的步骤：

取 JUMP_C0 指令到 F 级 在 ISR 中取指到 JUMP_C0，就说明需要从 ISR 中返回。

取到 epc 并准备跳转 此时该 JUMP_C0 指令在 D 级，通过转发机制取到 npc。这时，可能由于接力转发无法完成，而暂停一个周期，不过最终的结果是等价的。

屏蔽该 JUMP_C0 指令的延迟槽 此时该 JUMP_C0 指令在 D 级，正准备进入 E 级。由于 JUMP_C0 指令按定义不是跳转指令，所以不应该有延迟槽。因此，应该

使 D 级寄存器的同步复位信号激活，这样此时 F 级取到的该 JUMP_C0 指令的后续指令无法存入 D 级流水线寄存器。这样，就取消了该 JUMP_C0 指令的后续指令，从而起到屏蔽延迟槽的作用。但是，要在暂停结束后这样做。否则，由于暂停会使 E 级流水线寄存器同步复位，JUMP_C0 指令本身会被清除，因此这样的实现是错误的。

取 epc 对应的指令 此时该 JUMP_C0 指令在 E 级，epc 对应的指令在 F 级。这样，就开始填充流水线，CPU 准备恢复正常运行了。**如果取指产生异常，仍然会在 epc 对应的指令进入 M 级时重新被处理。**

把 exl 设为 1'b0 此时该 JUMP_C0 指令在 M 级，正准备进入 W 级；epc 对应的指令在 D 级，正准备进入 E 级。当且仅当 M 级为 JUMP_C0 类指令时，cw_m_cp0_exit_isr == 1'b1，这样下一个时钟周期 cp0 会令 exl = 1'b0，让 cp0 认为退出了 ISR。

开始处理新的异常和中断 此时该 JUMP_C0 指令在 W 级，正要退出流水线；epc 对应的指令在 M 级，正在被 cp0 检查，正准备进入 W 级；cp0 认为自己刚刚退出了 ISR。如果 epc 对应的指令有异常，又会被立刻处理；如果有中断，由于 exl == 1'b0，也会被处理。也就是说，开始处理新的异常和中断了。而且，该 JUMP_C0 指令，无论是否再次进入 ISR，都在提交点以后，会被提交。这就意味着退出 ISR 的过程算结束了。

15.9.7 系统内存空间

整个系统有一个内存空间，由真正的存储器和 MMIO 范围组成。MMIO 范围，就是可以通过 bridge 操作设备内部寄存器的范围。

表 80 系统内存空间

设备	地址下界	地址上界	可读写	允许的访问模式
dm	32'h00000000	32'h00001fff	读写	按字访问（地址字对齐）、按半字访问（地址半字对齐）、按字节访问
im	32'h00003000	32'h00004fff	读写	按字访问（地址按字对齐）

设备	地址下界	地址上界	可读写	允许的访问模式
timer	32'h00007f00	32'h00007f07	读写	按字访问（地址按字对齐）
timer	32'h00007f08	32'h00007f0b	只读	按字访问（地址按字对齐）
uart	32'h00007f10	32'h00007f2b	读写	按字访问（地址按字对齐）
switches	32'h00007f2c	32'h00007f33	只读	按字访问（地址按字对齐）
led	32'h00007f34	32'h00007f37	读写	按字访问（地址按字对齐）
nixie	32'h00007f38	32'h00007f3f	读写	按字访问（地址按字对齐）
buttons	32'h00007f40	32'h00007f43	只读	按字访问（地址按字对齐)

按照不允许的访问模式访问对应的设备，就会让对应设备的 `valid == 1'b0`。

15.9.8 注意事项

1. **为了正确判断 `cw_m_dm_stop`，需要在控制模块内部判断。不能直接超驰 `cw_m_dm_write_enable`。因为这样的话，一些依赖 `cw_m_dm_write_enable` 才能被判断出来的异常就不能被判断出来了。然后，`have2handle = 1'b0`，写使能又恢复，进入一种循环。这会使仿真无限进行。**
2. 为了防止受害指令进入 W 级，需要把 W 级流水线寄存器复位。
3. 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。
4. **如果有 `mult / multu / div / divu` 指令使 `md` 启动乘除法运算，而且该指令在异常发生时已经进入 W 级或离开流水线，那么该指令算成功提交。这样才能不违反精确异常的原则。但是，如果该指令在 E 级或 M 级，那么还是需要撤销该指令对 CPU 内部状态的更改。如果指令成功提交，暂停机制保证了操作 `md` 的三类数据通路类型实际上都是顺序执行的，所以没有数据冒险。效果就是 `md` 的值会更新，不过会出现在 ISR 里，让 ISR 用。这相当于进入 ISR 以前，`md` 指令执行了，而且结果已经得出，跟精确异常的原则相符。**
5. **实际上，`JUMP_C0` 类指令需要清空它后面的一条指令时，需要清空 D 级流水线寄存器。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。**
6. **由于 `JUMP_C0` 类指令到 M 级才把 `exl` 设为 0，所以会出现它没到 M 级的时**

候，或者刚清空流水线时，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，根据精确异常的思想，把它当成 ISR 已经进入或者还没退出来解决。

7. **不要孤立地看 cp0 和控制部件，cp0 提供了异常和中断的报告和处理机制，控制部件给 cp0 提供控制信号。二者结合，异常和中断的处理才能完整、正确。**

16 转发控制模块

16.1 原理

转发控制模块是流水线 CPU 控制机制的一部分，通过评估各指令读写寄存器的情况，实现有效的尽力转发。

16.2 端口定义

表 81 端口定义

端口	类型	位宽	功能
ddptype	输入	10	D 级指令类型
dreg1	输入	5	D 级指令与 rf.read_addr1 对应的寄存器号
dreg2	输入	5	D 级指令与 rf.read_addr2 对应的寄存器号
dregw	输入	5	D 级指令与 rf.write_addr 对应的寄存器号
edptype	输入	10	E 级指令类型
ereg1	输入	5	E 级指令与 rf.read_addr1 对应的寄存器号
ereg2	输入	5	E 级指令与 rf.read_addr2 对应的寄存器号
eregw	输入	5	E 级指令与 rf.write_addr 对应的寄存器号
mdptype	输入	10	M 级指令类型
mreg1	输入	5	M 级指令与 rf.read_addr1 对应的寄存器号
mreg2	输入	5	M 级指令与 rf.read_addr2 对应的寄存器号
mregw	输入	5	M 级指令与 rf.write_addr 对应的寄存器号
wdptype	输入	10	W 级指令类型
wreg1	输入	5	W 级指令与 rf.read_addr1 对应的寄存器号
wreg2	输入	5	W 级指令与 rf.read_addr2 对应的寄存器号
wregw	输入	5	W 级指令与 rf.write_addr 对应的寄存器号

端口	类型	位宽	功能
cw_fm_[de][12]	输出	4	D / E 级与 rf.read_addr[12] 对应的转发 MUX
cw_fm_m	输出	4	M 级与 rf.read_addr2 对应的转发 MUX
cw_fm_epx	输出	4	M 级与 cp0.epc 对应的转发 MUX

16.3 宏定义

暂无。

16.4 功能

见控制中的对应节。

17 暂停控制模块

17.1 原理

暂停控制模块是流水线 CPU 控制机制的一部分，通过评估 T_{use} 和 T_{new} 实现尽量少的暂停。它是控制模块内部的一部分。

17.2 端口定义

表 82 端口定义

端口	类型	位宽	功能
ddptype	输入	10	D 级指令类型
dreg1	输入	5	D 级指令与 rf.read_addr1 对应的寄存器号
dreg2	输入	5	D 级指令与 rf.read_addr2 对应的寄存器号
dregw	输入	5	D 级指令与 rf.write_addr 对应的寄存器号
edptype	输入	10	E 级指令类型
ereg1	输入	5	E 级指令与 rf.read_addr1 对应的寄存器号
ereg2	输入	5	E 级指令与 rf.read_addr2 对应的寄存器号
eregw	输入	5	E 级指令与 rf.write_addr 对应的寄存器号
mdptype	输入	10	M 级指令类型

端口	类型	位宽	功能
mreg1	输入	5	M 级指令与 rf.read_addr1 对应的寄存器号
mreg2	输入	5	M 级指令与 rf.read_addr2 对应的寄存器号
mregw	输入	5	M 级指令与 rf.write_addr 对应的寄存器号
wdptype	输入	10	W 级指令类型
wreg1	输入	5	W 级指令与 rf.read_addr1 对应的寄存器号
wreg2	输入	5	W 级指令与 rf.read_addr2 对应的寄存器号
wregw	输入	5	W 级指令与 rf.write_addr 对应的寄存器号
stall	输出	1	是否需要暂停

17.3 宏定义

暂无。

17.4 功能

见控制中的对应节。

18 中断和异常控制模块

18.1 原理

中断和异常控制模块负责进入 ISR 时，对各种控制信号的设定与超驰。同时，也负责退出 ISR 时对 JUMP_C0 类指令的后继指令的清除。

18.2 端口定义

表 83 端口定义

端口	类型	位宽	功能
dkind	输入	10	D 级指令具体类型
ekind	输入	10	E 级指令具体类型
mkind	输入	10	M 级指令具体类型
wkind	输入	10	W 级指令具体类型

端口	类型	位宽	功能
d_exc	输入	5	D 级指令异常 ID
e_exc	输入	5	E 级指令异常 ID
m_exc	输入	5	M 级指令异常 ID
m_exc_final	输入	5	M^+ 级指令异常 ID
rst	输入	1	同步复位信号
have2handle	输入	1	是否必须处理中断或异常
stall	输入	1	是否需要暂停
cw_d_pff_rst	输入	1	D 级流水线寄存器同步复位信号
cw_m_pff_rst	输入	1	E 级流水线寄存器同步复位信号
cw_w_pff_rst	输入	1	W 级流水线寄存器同步复位信号
cw_f_npc_jump_orig	输入	4	F: npc.jump_mode 原来的值
cw_f_npc_jump_mode	输入	4	F: npc.jump_mode 加钩子后的值
cw_f_pc_enable_orig	输入	1	F: pc.enable 原来的值
cw_f_pc_enable	输出	1	F: pc.enable 加钩子后的值
cw_e_pff_rst_orig	输入	1	E 级流水线寄存器同步复位信号 原来的值
cw_e_pff_rst	输出	1	E 级流水线寄存器同步复位信号 加钩子后的值
cw_e_md_restore	输出	1	E: md.restore 的值
cw_e_md_stop	输出	1	E: md.stop 的值
cw_e_dm_stop	输出	1	E: dm.stop 的值
cw_m_cp0_write_enable_orig	输入	1	M: cp0.write_enable 原来的值
cw_m_cp0_write_enable	输出	1	M: cp0.write_enable 加钩子后的值

18.3 宏定义

暂无。

18.4 功能

见控制中的对应节。

19 CPU

19.1 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。

19.2 端口定义

表 84 端口定义

端口	类型	位宽	功能
clk	输入	1	时钟信号
clk_2x	输入	1	二倍频时钟信号
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
dm_mode	输出	32	M: dm.mode 的值
cpu_write_data	输出	32	CPU 要对设备写入的数据
test_addr	输出	32	M: pc.curr_pc 的值
bridge_stop	输出	1	M: bridge.stop 的值

19.3 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有描述。

19.4 功能

CPU 负责指令的执行和与设备交互。

20 计时器

20.1 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。

20.2 端口定义

表 85 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	2	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

20.3 宏定义

表 86 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

20.4 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, 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。

无论什么时候，addr 为 2'b00、2'b01、2'b10 和 2'b11 时，对应的 read_result 为 {28'b0, allow_irq, mode, enable}、preset、count 和 32'b0。无论什么时候，irq = irq_reg & allow_irq。

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。count 寄存器禁止写入。如果试图写入，会通过 bridge 触发异常。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。

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。
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)；否则，count = 0，并转移到 TIMER_INT，irq <= 1'b1。
4. 若为 TIMER_INT，则 mode == 2'b00 时，enable <= 1'b0，这样可以保持中断信号。否则，irq_reg = 1'b0，关闭中断信号。这两种情况都要同时转移到 TIMER_IDLE。

20.5 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. **对 count 值的判断，源代码里没加 $unsigned，示例代码里也没有，但是应该是 $unsigned。**
4. **allow_irq 因为默认不会使用 timer0 来产生中断，所以初始化为 0。**
5. **写不该写的寄存器和写的模式错误时，写使能信号会被 cpu 屏蔽，所以不用担心误写。**

21 按钮模块

21.1 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。

21.2 端口定义

表 87 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	写使能信号
write_data	输入	32	要写入的数据
read_result	输出	32	读取结果
irq	输出	1	中断请求
user_key	输入	8	8 个用户开关的信号值

21.3 宏定义

暂无

21.4 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 <= ~user_key。

无论什么时候 read_result = {24'b0, stored}。write_enable、write_data、addr、irq 被忽略。

22 LED 控制模块

22.1 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。

22.2 端口定义

表 88 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
write_enable	输入	1	写使能信号
write_data	输入	32	要写入的数据
read_result	输出	32	读取结果
addr	输入	32	地址信号
led_light	输出	32	输出的当前 LED 开关数值
irq	输出	1	中断请求

22.3 宏定义

暂无

22.4 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)，则 stored <= wd。

无论什么时候，led_light = ~stored。

addr、read_result 和 irq 被忽略。

23 七段数码管控制

23.1 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。对应的内部寄存器具体分组如下：

表 89 原理

结构	数据位置	对应数码管	对应段选信号	对应值
{24'b0, tube2}	tube2[3:0]	digital_tube2	digital_tube_sel2	1'b1
{tube1, tube0}	tube1[15:12]	digital_tube1	digital_tube_sel1	4'b1000
{tube1, tube0}	tube1[11:8]	digital_tube1	digital_tube_sel1	4'b0100
{tube1, tube0}	tube1[7:4]	digital_tube1	digital_tube_sel1	4'b0010
{tube1, tube0}	tube1[3:0]	digital_tube1	digital_tube_sel1	4'b0001
{tube1, tube0}	tube0[15:12]	digital_tube0	digital_tube_sel0	4'b1000
{tube1, tube0}	tube0[11:8]	digital_tube0	digital_tube_sel0	4'b0100
{tube1, tube0}	tube0[7:4]	digital_tube0	digital_tube_sel0	4'b0010
{tube1, tube0}	tube0[3:0]	digital_tube0	digital_tube_sel0	4'b0001

23.2 端口定义

表 90 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	要读写的地址
write_enable	输入	1	写使能
write_data	输入	32	要写入的数据
read_result	输出	32	读取的数据
irq	输出	1	中断请求
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

23.3 宏定义

表 91 宏定义

类别	定义	值	意义
计数器	NIXIE_CTR	32'd30000	计数器初值，决定数码管刷新周期

23.4 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 0 和 4'b1111。

每个时钟上升沿，若 rst == 1'b1，则所有寄存器恢复初值。否则，若 write_enable == 1'b1 且 addr 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 addr 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。

表 92 功能

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。令三个 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。

irq 被忽略。

23.5 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. **复位时 phase 设成 4'b1111 能强行把所有数码管都设成 0，更合理。**

24 开关模块

24.1 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。

24.2 端口定义

表 93 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	1	读取地址
write_enable	输出	1	写使能信号
write_data	输入	32	要写入的数据
read_result	输出	32	读取结果
irq	输出	1	中断请求
dip_switch[0-7]	输入	8	8 组开关

25 宏定义

暂无

25.1 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_switch3, dip_switch2, ..., dip_switch0, dip_switch7, ..., dip_switch4}。

每个时钟上升沿，若 rst == 1'b1，stored <= 0。否则，stored <= dip_switch3, dip_switch2, ..., dip_switch0, dip_switch7, ..., dip_switch4}。

若 addr == 1'b0，则读出 stored 的低 32 位。若 addr == 1'b1，则读出 stored 的高 32 位。否则，读出 32'b0。

25.2 注意事项

1. **地址较低的字对应的是上面那排开关。**

26 UART 模块

26.1 原理

UART 模块通过包装已有的 UART 模块并进行微调，使它能够符合 CPU 的接口标准。在这之上建立 UART 通信机制。

26.2 端口定义

表 94 端口定义

端口	类型	位宽	功能
clk	输入	1	时钟信号
rst	输出	1	复位信号
addr	输入	3	操作地址
write_enable	输入	1	写使能
write_data	输入	32	要写入的数据
read_result	输入	32	读取结果
irq	输入	1	中断请求
uart_rxd	输入	1	UART 接收信号
uart_txd	输出	1	UART 发送信号

26.3 功能

UART 模块通过包装已有的 UART 模块存在，是它的兼容层。端口对应表如下。

表 95 功能

原 UART 模块端口名称	对外接线
CLK_I	clk
RST_I	rst
ADD_I	addr[2:0]
STB_I	1'b1
WE_I	write_enable
DAT_I	write_data
DAT_O	read_result

ACK_O	（无）
RxD	uart_rxd
TxD	uart_txd

然后，为了支持中断功能，在原有的 UART 模块中加入了输出信号 IRQ_O，连接到 UART 模块的 irq。IRQ_O 的值直接就是原有 UART 模块的 rs 寄存器，这样就能实现收到一个完整的字符即可中断的功能。

收到中断并进入 ISR 后，若想关闭中断，可以向偏移为 8 的寄存器写值。这样 rs 会变成 0，也就关闭中断了。

由于该模块包装的是原来的 UART 模块，所以详细功能和寄存器定义见原来 UART 模块的文档。

27 时钟分频器模块

27.1 原理

时钟分频器模块是利用 Clock Generator IP 核，把原来 FPGA 25MHz 的时钟变成设定好的时钟频率的。

27.2 端口定义

表 96 端口定义

端口	类型	位宽	功能
CLK_IN	输入	1	输入的时钟信号
CLK_OUT1	输出	1	输出的 30MHz 时钟信号
CLK_OUT2	输出	1	输出的 60MHz 时钟信号

27.3 功能

时钟分频器调整时钟频率，同时使时钟频率稳定。

28 顶层模块

28.1 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、bridge 和各个设备，把它们接在一起，形成综合的模块。

28.2 端口定义

表 97 端口定义

端口	类型	位宽	功能
clk_in	输入	1	FPGA 时钟信号
sys_rstn	输入	1	复位信号
uart_rxd	输入	1	UART 接收信号
uart_txd	输出	1	UART 发送信号
dip_switch0	输入	8	开关信号
dip_switch1	输入	8	开关信号
dip_switch2	输入	8	开关信号
dip_switch3	输入	8	开关信号
dip_switch4	输入	8	开关信号
dip_switch5	输入	8	开关信号
dip_switch6	输入	8	开关信号
dip_switch7	输入	8	开关信号
led_light	输出	32	LED 灯光信号
digital_tube[012]	输出	8	数码管输出信号
digital_tube_sel[01]	输出	4	数码管片选信号
digital_tube_sel2	输出	1	数码管片选信号
user_key	输入	8	用户按键

28.3 接线

CPU 和 bridge 的接线，以及 bridge 和各个设备的接线，都是可以对应的。所以，可以列出对应关系表。

clk_in 通过时钟分频器模块后，变成 clk 和 clk_2x 信号，分别代表时钟和双倍频时钟。它们都接入 CPU，CPU 中指令存储器和数据存储器都需要用到 clk_2x。

sys_rstn 由于是反相过后的信号，所以要在顶层模块内把它再反相一遍，然后再接入各个模块。

表 98 接线

CPU 端口	bridge 端口	数据流动方向
dm_mode	dm_mode	CPU → bridge
bridge_valid	valid	bridge → CPU
bridge_stop	stop	CPU → 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

表 99 接线

bridge 端口	*device-name* 端口	数据流动方向
*device-name*_addr	addr	bridge → *device-name*
*device-name*_write_enable	write_enable	bridge → *device-name*
*device-name*_write_data	write_data	bridge → *device-name*
*device-name*_read_result	read_result	*device-name* → bridge
*device-name*_irq	irq	*device-name* → bridge

对应设备中没有提到的端口，直接接入顶层模块中同名的端口。

28.4 功能

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

29 思考题

29.1 欢迎来到 P8

1. FPGA 是一种集成电路，设计出来是为了被消费者或者设计者在制造以后配置，所以才会有“现场可编程“这一说。FPGA 是由一组可编程逻辑块构成，也有一些可以配置的互联网络，使得逻辑块可以被接在一起。逻辑块可以被综合在一起，形成更复杂的组合逻辑电路。FPGA 里一般也有寄存器或者内存块这种逻辑块。

   好处；适合并行处理数据、在特定任务（如音视频处理）上性能可以比通用处理器高、用户可以决定硬件架构。

   缺陷：通用性一般较低、一般不适合程序性控制。

29.2 联结 ISE 工程与 FPGA

1. 问题：CPU 行为怪异、指令行为不符合预期。解决过程：把 IM 接入二倍频时钟。
2. 问题：Timing Score 过高，提示有的时间限制无法满足。解决过程：降低频率。
3. 问题：删除 `im` 的 IP 核对应的所有文件以后，新建新的 IP 核，ISE 仍然无法找到。解决过程：再删除一遍，ISE 弹出找不到 IP 核相关对话框时勾选意为找不到对应的文件删除的多选框，然后直接新建同名的 `im` IP 核。
4. 问题：综合过程报出一大堆警告。解决过程：仔细查看 Synthesis Report，确定警告都无害再继续。
5. 问题：CPU 无反应。解决过程：把 `sys_rstn` 反相，再当成 `rst`。
6. 问题：乘除法单元无法综合。解决过程：删除乘除法单元。
7. 问题：`bitgen` 提示有些引脚没有在顶层模块中体现，Generate Programming File 失败。解决过程：在顶层模块中体现出所有引脚再生成。

29.3 串口通信

1. 直接把原有的 UART 模块中的 `rs` 信号（标志着已经读取完整一个字节）当做中断信号。同时，若进入 ISR 后想要关闭 UART 中断，则写入偏移为 8 的内部寄存器。因为它能够使 UART 顶层模块认为已经读取了它接收的数据，`rs` 信号也会变回 `1'b0`，从而关闭中断。

29.4 挑战：MMU 与 TLB（选做）

1. 硬件与软件接口有软硬件之间交换信息的协定这一意义。软件可以通过指令操纵 TLB，这是按照一种标准来进行的。

   它也有硬件给软件提供平台、软件赋予硬件功能的意义。硬件通过 MMU 提供的地址转换机制，给软件实现页式虚拟内存。软件通过 TLB 操作（尤其是通过 ISR 进行 TLB 异常处理），赋予了硬件完整的功能。

   同时，它也相当于一种标准或者说约定。TLB 指令、MMU 行为规范和标准 TLB 的规范，都直接体现着联结软件和硬件的一种标准或约定。没有这种约定，硬件和软件的开发者的沟通成本会加大，计算通用性和兼容性也会减弱。

30 测试

1.

```
.text
lui $1, 0x8000
ori $1, 0x0000
lui $2, 0x8000
ori $2, 0x0001

addu $3, $1, $2

# exception
add $4, $1, $2
```

```
loop:
beq $0, $0, loop
nop

.ktext 0x4180
addu $1, $0, $0
addu $2, $0, $0
eret
lui $5, 0xdead
ori $5, 0xbeef

@00003000: $ 1 <= 80000000
@00003004: $ 1 <= 80000000
@00003008: $ 2 <= 80000000
@0000300c: $ 2 <= 80000001
@00003010: $ 3 <= 00000001
@00004180: $ 1 <= 00000000
@00004184: $ 2 <= 00000000
@00003014: $ 4 <= 00000000
```

2.

```
.text
lui $1, 0x8000
ori $1, 0x0000
lui $2, 0x8000
ori $2, 0x0001

addu $3, $1, $2
```

```
# exception
add $4, $1, $2

loop:
beq $0, $0, loop
nop

.ktext 0x4180
addu $1, $0, $0
addu $2, $0, $0
eret
lui $5, 0xdead
ori $5, 0xbeef

@00003000: $ 1 <= 80000000
@00003004: $ 1 <= 80000000
@00003008: $ 2 <= 80000000
@0000300c: $ 2 <= 80000001
@00003010: $ 3 <= 00000001
@00004180: $ 1 <= 00000000
@00004184: $ 2 <= 00000000
@00003014: $ 4 <= 00000000
```

3.

```
.text
lui $1, 0x8000
ori $1, 0x0000
lui $2, 0x8000
ori $2, 0x0000
```

```
add $3, $1, $2

loop:
beq $0, $0, loop
nop

.text 0x4180
mfc0 $4, $14
addiu $4, $4, 4
mtc0 $4, $14
eret

@00003000: $ 1 <= 80000000
@00003004: $ 1 <= 80000000
@00003008: $ 2 <= 80000000
@0000300c: $ 2 <= 80000000
@00004180: $ 4 <= 00003010
@00004184: $ 4 <= 00003014
```

4.

```
.text
lui $1, 0xcafe
ori $1, 0xbabe
lui $2, 0x00c0
ori $2, 0xffee

addu $3, $1, $2

mult $1, $2
```

```
# exception at level M
sw $0, 1($0)

# STORE_Ms at level D and E
mthi $1
mthi $2

# flush pipeline and wait for the mult to complete
nop
nop
nop
nop
nop
nop
nop

# exception at level M
sw $0, 1($0)

mthi $0

mfhi $9
mflo $10

# flush pipeline
nop
nop
nop
nop
nop
```

```
lui $1, 0x8000

mult $1, $1

# exception at level M
add $1, $1, $1

mfhi $9
mflo $10

loop:
beq $0, $0, loop
nop

.ktext 0x4180
mfc0 $4, $14
addiu $4, $4, 4
mtc0 $4, $14
eret

@00003000: $ 1 <= cafe0000
@00003004: $ 1 <= cafebabe
@00003008: $ 2 <= 00c00000
@0000300c: $ 2 <= 00c0ffee
@00003010: $ 3 <= cbbfbaac
@00004180: $ 4 <= 00003018
@00004184: $ 4 <= 0000301c
@00004180: $ 4 <= 00003040
@00004184: $ 4 <= 00003044
```

```
@00003048: $ 9 <= 00000000
@0000304c: $10 <= 8354dea4
@00003064: $ 1 <= 80000000
@00004180: $ 4 <= 0000306c
@00004184: $ 4 <= 00003070
@00003070: $ 9 <= 40000000
@00003074: $10 <= 00000000
```

5.

```
# NOTE: This file is not supposed to be run by MARS.

.text
lui $1, 0x8000
ori $1, 0x0000
lui $2, 0x8000
ori $2, 0x0000
ori $3, 0x7f00
ori $4, 0x7f10
ori $5, 0x0005

# enable all hardware interrupts
ori $8, 0xfc01
mtc0 $8, $12

# allow_irq = 1, mode = 1, enable = 0
ori $6, 0x000a
sw $6, 0($3)

#save the preset of timer0 and load it
sw $5, 4($3)
```

```
lw $7, 4($3)

#enable interrupts
ori $6, 0x000b
sw $6, 0($3)

#nop sled
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop

#allow_irq = 1, mode = 0, enable = 1
lui $6, 0x0000
ori $6, 0x0009
sw $6, 0($3)

#nop sled
nop
nop
nop
nop
```

```
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop

#allow_irq = 1, mode = 0, enable = 1
#try to make two simers interrupt simutaneously
sw $5, 4($3)
addiu $5, $5, -1
sw $5, 4($4)

lui $6, 0x0000
ori $6, 0x0009
sw $6, 0($3)
sw $6, 0($4)

#nop sled
nop
nop
nop
nop
nop
```

```
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop

#stop interrupts in the middle
lui $6, 0x0000
ori $6, 0x0009
sw $6, 0($3)

#allow_irq = 0
lui $6, 0x0000
ori $6, 0x0001
sw $6, 0($3)

#nop sled
nop
nop
nop
nop
nop
nop
nop
```

```
nop
nop
nop
nop
nop
nop
nop
nop
nop

#tail loop
loop:
beq $0, $0, loop
nop

.text 0x4180
#disable counter
lui $6, 0x0000
ori $6, 0x0000
sw $6, 0($3)
sw $6, 0($4)

mfc0 $14, $14
addiu $14, $14, 4
mtc0 $14, $14
lui $26, 0xfeed
ori $26, 0xbeef
eret

@00003000: $ 1 <= 80000000
```

```
@00003004: $ 1 <= 80000000
@00003008: $ 2 <= 80000000
@0000300c: $ 2 <= 80000000
@00003010: $ 3 <= 00007f00
@00003014: $ 4 <= 00007f10
@00003018: $ 5 <= 00000005
@0000301c: $ 8 <= 0000fc01
@00003024: $ 6 <= 0000000a
@00003030: $ 7 <= 00000005
@00003034: $ 6 <= 0000000b
@00004180: $ 6 <= 00000000
@00004184: $ 6 <= 00000000
@00004190: $14 <= 00003058
@00004194: $14 <= 0000305c
@0000419c: $26 <= feed0000
@000041a0: $26 <= feedbeef
@0000306c: $ 6 <= 00000000
@00003070: $ 6 <= 00000009
@00004180: $ 6 <= 00000000
@00004184: $ 6 <= 00000000
@00004190: $14 <= 00003094
@00004194: $14 <= 00003098
@0000419c: $26 <= feed0000
@000041a0: $26 <= feedbeef
@000030bc: $ 5 <= 00000004
@000030c4: $ 6 <= 00000000
@000030c8: $ 6 <= 00000009
@00004180: $ 6 <= 00000000
@00004184: $ 6 <= 00000000
@00004190: $14 <= 000030d0
```

```
@00004194: $14 <= 000030d4
@0000419c: $26 <= feed0000
@000041a0: $26 <= feedbeef
@00003114: $ 6 <= 00000000
@00003118: $ 6 <= 00000009
@00003120: $ 6 <= 00000000
@00003124: $ 6 <= 00000001
```

6.

```
.ktext 0x4180
_entry:
        ori     $k0, $0, 0x1000
        sw      $sp, -4($k0)
        mfc0    $k1, $12
        sw      $k1, -8($k0)

        addiu   $k0, $k0, -256
        move    $sp, $k0

        j       _save_context
        nop

_main_handler:
        mfc0    $k0, $14
        addu    $k0, $k0, 4
        mtc0    $k0, $14
        j       _restore_context
        nop

_restore:
```

```
        eret


_save_context:
        sw      $1, 4($sp)
        sw      $2, 8($sp)
        sw      $3, 12($sp)
        sw      $4, 16($sp)
        sw      $5, 20($sp)
        sw      $6, 24($sp)
        sw      $7, 28($sp)
        sw      $8, 32($sp)
        sw      $9, 36($sp)
        sw      $10, 40($sp)
        sw      $11, 44($sp)
        sw      $12, 48($sp)
        sw      $13, 52($sp)
        sw      $14, 56($sp)
        sw      $15, 60($sp)
        sw      $16, 64($sp)
        sw      $17, 68($sp)
        sw      $18, 72($sp)
        sw      $19, 76($sp)
        sw      $20, 80($sp)
        sw      $21, 84($sp)
        sw      $22, 88($sp)
        sw      $23, 92($sp)
        sw      $24, 96($sp)
        sw      $25, 100($sp)
        sw      $26, 104($sp)
        sw      $27, 108($sp)
```

```
        sw      $28, 112($sp)
        sw      $29, 116($sp)
        sw      $30, 120($sp)
        sw      $31, 124($sp)
        mfhi    $k0
        sw      $k0, 128($sp)
        mflo    $k0
        sw      $k0, 132($sp)
        j       _main_handler
        nop



_restore_context:
        lw      $1, 4($sp)
        lw      $2, 8($sp)
        lw      $3, 12($sp)
        lw      $4, 16($sp)
        lw      $5, 20($sp)
        lw      $6, 24($sp)
        lw      $7, 28($sp)
        lw      $8, 32($sp)
        lw      $9, 36($sp)
        lw      $10, 40($sp)
        lw      $11, 44($sp)
        lw      $12, 48($sp)
        lw      $13, 52($sp)
        lw      $14, 56($sp)
        lw      $15, 60($sp)
        lw      $16, 64($sp)
```

```
        lw      $17, 68($sp)
        lw      $18, 72($sp)
        lw      $19, 76($sp)
        lw      $20, 80($sp)
        lw      $21, 84($sp)
        lw      $22, 88($sp)
        lw      $23, 92($sp)
        lw      $24, 96($sp)
        lw      $25, 100($sp)
        lw      $26, 104($sp)
        lw      $27, 108($sp)
        lw      $28, 112($sp)
        lw      $29, 116($sp)
        lw      $30, 120($sp)
        lw      $31, 124($sp)
        lw      $k0, 128($sp)
        mthi    $k0
        lw      $k0, 132($sp)
        mtlo    $k0
        j       _restore
        nop



.text
        ori     $28, $0, 0x0000
        ori     $29, $0, 0x0000
        lui     $8, 0x7fff
        lui     $9, 0x7fff
```

```
        add     $10, $8, $9
        or      $10, $8, $9
loop:
        beq $0, $0, loop
        nop

@00003000: $28 <= 00000000
@00003004: $29 <= 00000000
@00003008: $ 8 <= 7fff0000
@0000300c: $ 9 <= 7fff0000
@00004180: $26 <= 00001000
@00004184: *00000ffc <= 00000000
@00004188: $27 <= 0000ff13
@0000418c: *00000ff8 <= 0000ff13
@00004190: $26 <= 00000f00
@00004194: $29 <= 00000f00
@000041c0: *00000f04 <= 00000000
@000041c4: *00000f08 <= 00000000
@000041c8: *00000f0c <= 00000000
@000041cc: *00000f10 <= 00000000
@000041d0: *00000f14 <= 00000000
@000041d4: *00000f18 <= 00000000
@000041d8: *00000f1c <= 00000000
@000041dc: *00000f20 <= 7fff0000
@000041e0: *00000f24 <= 7fff0000
@000041e4: *00000f28 <= 00000000
@000041e8: *00000f2c <= 00000000
@000041ec: *00000f30 <= 00000000
@000041f0: *00000f34 <= 00000000
@000041f4: *00000f38 <= 00000000
```

```
@000041f8: *00000f3c <= 00000000
@000041fc: *00000f40 <= 00000000
@00004200: *00000f44 <= 00000000
@00004204: *00000f48 <= 00000000
@00004208: *00000f4c <= 00000000
@0000420c: *00000f50 <= 00000000
@00004210: *00000f54 <= 00000000
@00004214: *00000f58 <= 00000000
@00004218: *00000f5c <= 00000000
@0000421c: *00000f60 <= 00000000
@00004220: *00000f64 <= 00000000
@00004224: *00000f68 <= 00000f00
@00004228: *00000f6c <= 0000ff13
@0000422c: *00000f70 <= 00000000
@00004230: *00000f74 <= 00000f00
@00004234: *00000f78 <= 00000000
@00004238: *00000f7c <= 00000000
@0000423c: $26 <= 00000000
@00004240: *00000f80 <= 00000000
@00004244: $26 <= 00000000
@00004248: *00000f84 <= 00000000
@000041a0: $26 <= 00003010
@000041a4: $ 1 <= 00000000
@000041a8: $ 1 <= 00000004
@000041ac: $26 <= 00003014
@00004254: $ 1 <= 00000000
@00004258: $ 2 <= 00000000
@0000425c: $ 3 <= 00000000
@00004260: $ 4 <= 00000000
@00004264: $ 5 <= 00000000
```

```
@00004268: $ 6 <= 00000000
@0000426c: $ 7 <= 00000000
@00004270: $ 8 <= 7fff0000
@00004274: $ 9 <= 7fff0000
@00004278: $10 <= 00000000
@0000427c: $11 <= 00000000
@00004280: $12 <= 00000000
@00004284: $13 <= 00000000
@00004288: $14 <= 00000000
@0000428c: $15 <= 00000000
@00004290: $16 <= 00000000
@00004294: $17 <= 00000000
@00004298: $18 <= 00000000
@0000429c: $19 <= 00000000
@000042a0: $20 <= 00000000
@000042a4: $21 <= 00000000
@000042a8: $22 <= 00000000
@000042ac: $23 <= 00000000
@000042b0: $24 <= 00000000
@000042b4: $25 <= 00000000
@000042b8: $26 <= 00000f00
@000042bc: $27 <= 0000ff13
@000042c0: $28 <= 00000000
@000042c4: $29 <= 00000f00
@000042c8: $30 <= 00000000
@000042cc: $31 <= 00000000
@000042d0: $26 <= 00000000
@000042d8: $26 <= 00000000
@00003014: $10 <= 7fff0000
```

7.

```
.text
lui $1, 0x8000
ori $1, 0x0000
lui $2, 0x8000
ori $2, 0x0000

add $3, $1, $2

loop:
beq $0, $0, loop
nop

.text 0x4180
mfc0 $4, $14
addiu $4, $4, 4
mtc0 $4, $14
eret
lui $10, 0xdead

@00003000: $ 1 <= 80000000
@00003004: $ 1 <= 80000000
@00003008: $ 2 <= 80000000
@0000300c: $ 2 <= 80000000
@00004180: $ 4 <= 00003010
@00004184: $ 4 <= 00003014
```

31 技巧

31.1 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

31.2 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**
- 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，p5 有时不用改 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
  - **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**

- 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**

- 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`**

31.3 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
  - 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
  - 实在不行就把感觉错了的指令都检查一遍

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
  - 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**
- 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
- 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
- CPU 的初始状态

31.4 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**
- 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**
  - 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
  - 可能需要回退机制
  - 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**

- 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**
  - 分析一下新部件的功能
  - **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

31.5 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