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	3	是否可以跳转
alu_comp_result	输入	2	ALU 的比较结果
num	输入	16	输入的立即数
jnum	输入	26	输入的 J 型指令的立即数
reg_	输入	32	输入的寄存器值
next_pc	输出	32	下一个 PC

宏定义

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

类别	定义	值	意义
jump_mode	NPC_JUMP_DISABLE	3'b000	不要跳转
jump_mode	NPC_JUMP_DISABLED	NPC_JUMP_DISABLE	
jump_mode	NPC_JUMP_WHEN_EQUAL	3'b001	当 ALU 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_JUMP_WHEN_NOT_EQUAL	3'b010	当 ALU 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_REG	3'b111	按照寄存器内地址跳转
jump_mode	NPC_J	3'b110	按照 J 型指令的立即数跳转

alu_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 == NPC_JUMP_WHEN_NOT_EQUAL，则 alu_comp_result != ALU_EQUAL 时，做跟上面相同的步骤。否则做跟 jump_mode == NPC_JUMP_DISABLE 时相同的步骤。

若 jump_mode == NPC_REG，则令 next_pc = reg_。

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

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

注意事项

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

PC

原理

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

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

端口定义

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

宏定义

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

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

功能

该部件是时序部件。

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

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

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

注意事项

1. PC 和 IM 的起始地址是分开定义的，改的时候要注意。

程序存储器

原理

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

为了简便，程序存储器由许多寄存器实现。

端口定义

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

宏定义

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

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

功能

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

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

注意事项

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

寄存器堆

原理

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

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

寄存器的使用没有规定，这一般是软件关心的问题。

端口定义

端口	类型	位宽	功能
clk	输入	1	时钟信号
curr_pc	输入	32	当前 PC 的值
read_addr1	输入	5	第一个读地址
read_addr2	输入	5	第二个读地址
write_addr	输入	5	写地址
write_data	输入	32	要写入的数据
write_enable	输入	1	写使能
read_result1	输出	32	第一个读地址读出的数据
read_result2	输出	32	第二个读地址读出的数据

宏定义

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

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

功能

该部件为时序部件。

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

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

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

注意事项

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

比较模块

原理

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

接口定义

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

宏定义

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

功能

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

扩展器

功能

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

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

接口定义

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

宏定义

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

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

功能

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

若 mode 的值为非法操作，就令 result = 32'b0。

ALU

原理

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

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

端口定义

端口	类型	位宽	功能
num1	输入	32	第一个操作数
num2	输入	32	第二个操作数
op	输入	5	操作符

result	输出 32	结果
comp_result	输出 2	作为无符号数的比较结果
sig_comp_result	输出 2	作为有符号数的比较结果
overflow	输出 1	计算过程中是否发生溢出
op_invalid	输出 1	操作符是否无效

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

宏定义

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

类别	定义	值	意义
op	ALU_ADD	5b'00000	加法运算
op	ALU_UNSIGNED_ADD	ALU_ADD	同上
op	ALU_SUB	5b'00001	减法运算
op	ALU_UNSIGNED_SUB	ALU_SUB	同上
op	ALU_AND	5b'10000	按位与运算
op	ALU_BITWISE_AND	ALU_AND	同上
op	ALU_OR	5b'10001	按位或运算
op	ALU_BITWISE_OR	ALU_OR	同上
op	ALU_NOT	5b'10010	按位非运算
op	ALU_BITWISE_NOT	ALU_NOT	同上
op	ALU_XOR	5b'10011	按位异或运算
op	ALU_MOVZ	5b'00010	数据转移运算（[1]）
.*comp_result	ALU_EQUAL	2b'00	等于
.*comp_result	ALU_EQUAL_TO	ALU_EQUAL	同上
.*comp_result	ALU_LARGER	2b'01	大于
.*comp_result	ALU_LARGER_THAN	ALU_LARGER	同上
.*comp_result	ALU_SMALLER	2b'10	小于
.*comp_result	ALU_SMALLER_THAN	ALU_SMALLER	同上
overflow	ALU_OVERFLOW	1'b1	溢出
overflow	ALU_NOT_OVERFLOW	1'b0	未溢出
op_invalid	ALU_INVALID_OP	1'b1	操作符无效
op_invalid	ALU_INVALID	ALU_INVALID_OP	同上
op_invalid	ALU_VALID_OP	1'b0	操作符有效
op_invalid	ALU_VALID	ALU_VALID_OP	同上

注：

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

功能

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

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

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

.*comp_result 的值仅由 num[12] 确定，与其它输入无关。.*comp_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。

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

数据存储器

原理

数据存储器是存储数据的地方。

为了渐变，数据存储器由许多寄存器实现。

端口定义

端口	类型	位宽	功能
clk	输入	1	时钟信号
curr_pc	输入	32	当前 PC 值
read_addr	输入	32	读地址
write_addr	输入	32	写地址
write_data	输入	32	写数据
write_enable	输入	1	写使能信号
read_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
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	
.*_addr	DM_ADDR_WIDTH	8	.*_addr 的位宽
指令存储器	DM_SIZE	64	能存储 32 位字的个数

功能

该部件为时序部件。

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

在每个时钟上升沿，若 write_enable == DM_ENABLED，则 write_addr[DM_ADDR_WIDTH - 1:1] 这个地址对应的 32 位字写入 write_data 对应的值。同时，打印当前 PC 的值、write_addr、write_addr 这个地址对应的 32 位字原来的值、它的新值。

任何时候，read_result 的值为 read_addr[DM_ADDR_WIDTH - 1:1] 对应的地址的值。

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. 现在还没有按照半个字或者字节寻址，所以暂时不加入 mode 端口
2. 直接忽略地址后两位
3. DM_ADDR_WIDTH 和 DM_ADDR_SIZE 要一块改
4. 地址空间是被截断的，看起来是 32 位，实际上不是

流水线寄存器

原理

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

端口定义

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

参数定义

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

宏定义

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

功能

该部件为时序部件。

该部件内部的寄存器初值为全 0。

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

输出端口 `o` 的值总是寄存器的值。

注意事项

1. 复位设成了同步复位，这是为了更好地插入气泡。

MUX

功能

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

类别

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

命名

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

宏定义

暂无

但是仍然保留 `mux.h` 宏文件并填入模版，以备以后使用。

参数定义

参数	默认值	功能
`BIT_WIDTH`	32	输入和输出数据的位宽

端口定义

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

注：

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

功能

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

注意事项

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

流水线 CPU 数据通路

原理

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

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

分析

p5 需要实现的指令为：

`addu, subu, lui, ori, lw, sw, beq, nop, j, jal, jr, movz`

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
CAL_I	lui, ori
LOAD	lw
STORE	sw
BRANCH	beq
NOP	nop
JAL	j, jal
JR	jr
CMOV	movz

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

有时部件名称可能和级不对应。这表示相应端口的值是经过流水后的结果。

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

注意：使用延迟槽来简化暂停和转发的分析。

F 级（IF）

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

D 级（ID）

数据通路类型|D: ext.num|D: cmp.reg1|D: cmp.reg2
CAL_R|||
CAL_I|D: im.result[15:0]||
LOAD|D: im.result[15:0]||
STORE|||
BRANCH||D: rf.read_result1|D: rf.read_result2
NOP|||
JAL|||
JR|||
CMOV||D: rf.read_result2
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
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
BRANCH||
NOP||
JAL||
JR||
CMOV|D: rf.read_result1|D: rf.read_result2
综合|D: rf.read_result1|D: rf.read_result2, D: ext.result

M 级（MEM）

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

W 级（WB）

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

流水线寄存器

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

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

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

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

M	pc.curr_pc	m_pc

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

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

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

调试相关功能

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

注意：**写入寄存器是使用 w 级流水到的 pc 值。**

数据通路 MUX

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

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

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

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

转发

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

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

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

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	fm_d1
E: rf.read_result[12]	M: npc.next_pc, M: alu.result, W: rf.write_data	fm_e1
M: dm.write_data	W: rf.write_data	fm_m

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

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

暂停

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

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

指令识别机制

原理

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

宏定义

由于有特殊的指令读写固定的寄存器，所以寄存器号也要宏定义。

由于函数的声明需要一定的范式保证健壮性，所以函数的声明本身也要定义。

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

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

端口定义

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

功能

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

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

宏定义

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

编码方式为前 4 位为数据通路类型，后 5 位为其下的具体类型。

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

指令字段

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

指令类型

类别	定义	值	意义
指令类型	UNKNOWN	9'b0000_00000	未知指令
指令类型	UNK	UNKNOWN	
指令类型	ADDU	9'b0001_00000	
指令类型	SUBU	9'b0001_00001	
指令类型	LUI	9'b0010_00000	
指令类型	ORI	9'b0010_00001	
指令类型	LW	9'b0011_00000	
指令类型	SW	9'b0100_00000	
指令类型	BEQ	9'b0101_00000	
指令类型	J	9'b0110_00000	
指令类型	JAL	9'b0110_00001	
指令类	JR	9'b0111_00000	

型			
指令类型	MOVZ	9'b1000_00000	
指令类型	NOP	9'b1111_00000	nop 指令是 sll 指令的一个特例，所以临时开一栏，在扩充指令时会去掉这一类

注意

1. 临时的数据通路类型都是从上往下长的。

控制

原理

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

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

端口定义

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

总体结构

控制模块是时序部件。不设置成组合逻辑部件的原因如下。

1. 哪怕控制本身不设置成时序部件，也需要流水控制信号，这是流水线 CPU 结构上的需要。
2. 控制本身是时序部件，就可以流水更多的信息。最明显的就是指令读写寄存器的信息。比如暴力转发也把指令读写寄存器的信息放在流水线中流水。
3. 保留单周期处理器的控制机制实际上过渡不是那么平滑，因为还有多周期处理器，它的控制是类似状态机的结构。

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

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

数据通路和功能控制信号

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

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

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

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

F 级

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

BRANCH 类指令类型与 F: npc.jump_mode 的关系：

指令类型	**F: npc.jump_mode**
BEQ	NPC_JUMP_WHEN_EQUAL

注意：**F 级的控制信号是由 D 级指令控制的。**

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_MODE_PAD
ORI　EXT_MODE_UNSIGNED

E 级（EX）

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

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

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

CMOV 类指令类型与 E: alu.op 的关系

指令类型 E: alu.op
MOVZ　ALU_CMOV

M 级（MEM）

数据通路类型 M: dm.write_enable
STORE　1'b1
（其它）　1'b0

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	1'b1	D: npc.next_pc
CMOV	1'b1	E: alu.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。

指令读写寄存器识别

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

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

数据通路类型	reg1	reg2	regw
CAL_R	RS	RT	RD
CAL_I	RS	RT	RT
LOAD	RS	ZERO	RT
STORE	RS	RT	ZERO
BRANCH	RS	RT	ZERO
JUMP	ZERO	ZERO	视指令而定（[2]）
JUMPR	RS	ZERO	ZERO
CMOV	RS	RT	视寄存器值而定（[1]）
NOP	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`。

转发控制信号

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

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

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

转发控制信号最终需要控制的是转发 MUX，因此转发 MUX 也要进行定义。

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

类别	定义	值	意义
所有转发 MUX	orig	0	不转发，保持原样
fm_d[12]	E2D_rf	1	E 级到 D 级，数据通路类型是 CMOV，要写入的数据在 D 级产生好了，到了 E 级才能转发
fm_d[12]	E2D_npc	2	E 级到 D 级，数据通路类型是 JUMP_I，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_e[12]	M2E_npc	1	M 级到 E 级，数据通路类型是 JUMP_I，要写入的数据在 D 级产生好了，到了 E 级才能转发
fm_e[12]	M2E_alu	2	M 级到 E 级，数据通路类型是 CAL_R / CAL_I / CMOV，数据在 D 级或 E 级产生好了，但对 CAL_R / CAL_I 来说，到了 M 级才能转发
fm_e[12]	W2E_rf	3	W 级到 D 级，数据通路类型是所有能够写入寄存器的类型，数据在 W 级都可以转发了
fm_m	W2M_rf	1	W 级到 M 级，数据通路类型是所有能够写入寄存器的类型，数据在 W 级都可以转发了（比如 sw 指令转发 rf 内容）

注意：**B2A_.* 表示 B 级从 A 级转发。**

注意：**宏的值要和对应转发 MUX 的接线顺序相符。**

注意：**E2D_rf 表示把 E 级的第一个寄存器转发出去。**

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

暂停控制信号

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

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

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

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

暂停要分两个寄存器，因为数据冒险也是要分成两个寄存器的情况的。

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

注意：**控制内部的流水线也要插入气泡。**

在 D 级各种数据通路类型的 Tuse 如下。

数据通路类型	Tuse (read_addr1)	Tuse (read_addr2)
UNKNOWN		
CAL_R	1	1
CAL_I	1	1
LOAD	1	
STORE	1	2
BRANCH	0	0
JUMP_I		
JUMP_R	0	
CMOV	0	0
NOP		

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		
JAL	0	0
JR	0	
CMOV	0	0
NOP		

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。

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

寄存器地址控制信号

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

CPU

原理

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

CPU 在模块结构中作为顶层模块而存在。

端口定义

端口 类型 位宽　功能
`clk` 输入 1　　时钟信号

接线

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

功能

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

注意事项

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

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**

- 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`**

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

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

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