处理器设计文档

NPC

功能

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
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	

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

若 jump_mode == NPC_JUMP_DISABLE，则令 next_pc = $unsigned(curr_pc) + $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(curr_pc) + $signed(4) + $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 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 enable == IM_ENABLED，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。

若 enable == IM_DISABLED，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

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
.*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	2'b10	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

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

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！**

单周期数据通路

功能

单周期数据通路是负责单周期处理器的数据通路。由于它只是在一个周期内做完五步操作，所以能简单一些。数据通路是把相应的数据通路部件按照一定的逻辑关系连接起来得到的、

通过对需要实现的每条指令的分析，可以得出每条指令具体需要什么样的数据通路，然后用这个来知道实现。有些数据的流向是需要指定的，这时就需要控制单元出马，在相应的地方放上 MUX，然后让控制单元控制。

分析

p4 需要实现的 8 条指令为：

```
addu, subu, lui, ori, lw, sw, beq, nop
```

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 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。

最后一行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。

指令	addu	subu	lui
npc.curr_pc	pc.curr_pc	pc.curr_pc	pc.curr_pc
npc.alu_comp_result			
npc.num			
pc.next_pc	npc.next_pc	npc.next_pc	npc.next_pc
im.addr	pc.curr_pc	pc.curr_pc	pc.curr_pc
rf.read_addr1	im.data[25:21]	im.data[25:21]	im.data[25:21]
rf.read_addr2	im.data[20:16]	im.data[20:16]	
rf.write_addr	im.data[15:11]	im.data[15:11]	im.data[20:16]
rf.write_data	alu.result	alu.result	alu.result
alu.num1	rf.read_result1	rf.read_result1	rf.read_result1
alu.num2	rf.read_result2	rf.read_result2	ext.result
ext.num			im.data[15:0]
dm.read_addr			
dm.write_addr			
dm.write_data			

指令	ori	lw	sw
npc.curr_pc	pc.curr_pc	pc.curr_pc	pc.curr_pc
npc.alu_comp_result			
npc.num			
pc.next_pc	npc.next_pc	npc.next_pc	npc.next_pc
im.addr	pc.curr_pc	pc.curr_pc	pc.curr_pc
rf.read_addr1	im.data[25:21]	im.data[25:21]	im.data[25:21]
rf.read_addr2	im.data[20:16]		im.data[20:16]
rf.write_addr	im.data[20:16]	im.data[20:16]	
rf.write_data	alu.result	dm.read_result	
alu.num1	rf.read_result1	rf.read_result1	rf.read_result1
alu.num2	ext.result	ext.result	ext.result
ext.num	im.data[15:0]	im.data[15:0]	im.data[15:0]
dm.read_addr		alu.result	
dm.write_addr			alu.result
dm.write_data			rf.read_result2

指令	beq	nop	综合
npc.curr_pc	pc.curr_pc	pc.curr_pc	pc.curr_pc
npc.alu_comp_result	alu.comp_result		alu.comp_result
npc.num	im.data[15:0]		im.data[15:0]
pc.next_pc	npc.next_pc	npc.next_pc	npc.next_pc
im.addr	pc.curr_pc	pc.curr_pc	pc.curr_pc
rf.read_addr1	im.data[25:21]	im.data[25:21]	im.data[25:21]
rf.read_addr2	im.data[20:16]	im.date[20:16]	im.data[20:16]

rf.write_addr	im.data[20:16]		im.data[20:16], im.data[15:11]
rf.write_data	alu.result		alu.result, dm.read_result
alu.num1	rf.read_result1	rf.read_result1	rf.read_result1
alu.num2	ext.result	rf.read_result2	rf.read_result2, ext.result
ext.num	im.data[15:0]		im.data[15:0]
dm.read_addr		alu.result	alu.result
dm.write_addr			alu.result
dm.write_data			rf.read_result2

这时可以看出如下的端口需要 MUX:

端口	所有的信号来源	MUX 名称
rf.write_addr	im.data[20:16], im.data[15:11]	m_rf_write_addr
rf.write_data	alu.result, dm.read_result	m_rf_write_data
alu.num2	rf.read_result2, ext.result	m_alu_num2

这些 MUX 最终还是让控制部件来识别。MUX 端口的连接顺序（其实也对应着当控制信号从 0 开始递增时会选择的端口）按照上表中的顺序给定。

控制

功能

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。

分析

通过对数据通路的分析，可以得到每条指令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。

指令	addu	subu
m_rf_write_addr	im.data[15:11]	im.data[15:11]
m_rf_write_data	alu.result	alu.result
m_alu_num2	rf.read_result2	rf.read_result2
npc.jump_mode	NPC_JUMP_DISABLED	NPC_JUMP_DISABLED
pc.enable	PC_ENABLED	PC_ENABLED
im.enable	IM_ENABLED	IM_ENABLED
rf.write_enable	RF_WRITE_ENABLED	RF_WRITE_ENABLED
alu.op	ALU_ADD	ALU_SUB
ext.mode	#EXT_MODE_UNSIGNED	#EXT_MODE_UNSIGNED
dm.write_enable	DM_WRITE_DISABLED	DM_WRITE_DISABLED

指令	lui	ori
m_rf_write_addr	im.data[20:16]	im.data[20:16]
m_rf_write_data	alu.result	alu.result
m_alu_num2	ext.result	ext.result
npc.jump_mode	NPC_JUMP_DISABLED	NPC_JUMP_DISABLED
pc.enable	PC_ENABLED	PC_ENABLED
im.enable	IM_ENABLED	IM_ENABLED
rf.write_enable	RF_WRITE_ENABLED	RF_WRITE_ENABLED
alu.op	ALU_OR	ALU_OR
ext.mode	EXT_MODE_PAD	EXT_MODE_UNSIGNED
dm.write_enable	DM_WRITE_DISABLED	DM_WRITE_DISABLED

指令	lw	sw
m_rf_write_addr	im.data[20:16]	im.data[20:16]
m_rf_write_data	dm.read_result	alu.result
m_alu_num2	ext.result	ext.result
npc.jump_mode	NPC_JUMP_DISABLED	NPC_JUMP_DISABLED
pc.enable	PC_ENABLED	PC_ENABLED
im.enable	IM_ENABLED	IM_ENABLED
rf.write_enable	RF_WRITE_ENABLED	RF_WRITE_DISABLED
alu.op	ALU_ADD	ALU_ADD
ext.mode	EXT_MODE_SIGNED	EXT_MODE_SIGNED
dm.write_enable	DM_WRITE_DISABLED	DM_WRITE_ENABLED

指令	beq	nop
m_rf_write_addr	#im.data[20:16]	#im.data[20:16]
m_rf_write_data	#alu.result	#alu.result
m_alu_num2	rf.read_result2	#rf.read_result2
npc.jump_mode	NPC_JUMP_WHEN_EQUAL	NPC_JUMP_DISABLED
pc.enable	PC_ENABLED	PC_ENABLED
im.enable	IM_ENABLED	IM_ENABLED
rf.write_enable	RF_WRITE_DISABLED	RF_WRITE_DISABLED
alu.op	#ALU_OR	#ALU_OR
ext.mode	#EXT_MODE_UNSIGNED	#EXT_MODE_UNSIGNED
dm.write_enable	DM_WRITE_DISABLED	DM_WRITE_DISABLED

对于未知指令，各控制信号的值与 nop 指令的相应值相同。这样相当于直接忽略未知指令。

宏定义

类别	定义	值	意义
宏函数	GET_OP(x)	x[31:26]	得到指令的 op 字段
宏函数	GET_FUNCT(x)	x[25:11]	得到指令的 funct 字段
指令类型	R_TYPE	2'b00	R 型指令
指令类型	I_TYPE	2'b01	I 型指令
指令类型	J_TYPE	2'b10	J 型指令

指令类型	C_TYPE	2'b11	协处理器指令
指令魔数	INSTR_MAGIC_RTYPE_OP	6'b000000	R 型指令 op 字段魔数
指令魔数	INSTR_MAGIC_ADDU_FUNCT	6'b100001	addu 指令 funct 字段魔数
指令魔数	INSTR_MAGIC_SUBU_FUNCT	6'b100011	subu 指令 funct 字段魔数
指令魔数	INSTR_MAGIC_LUI_OP	6'b001111	lui 指令 op 字段魔数
指令魔数	INSTR_MAGIC_ORI_OP	6'b001101	ori 指令 op 字段魔数
指令魔数	INSTR_MAGIC_LW_OP	6'b100011	lw 指令 op 字段魔数
指令魔数	INSTR_MAGIC_SW_OP	6'b101011	sw 指令 op 字段魔数
指令魔数	INSTR_MAGIC_BEQ_OP	6'b000100	beq 指令 op 字段魔数
指令魔数	INSTR_MAGIC_NOP_FUNCT	6'b000000	nop 指令 funct 字段魔数
指令具体类型	INSTR_UNKNOWN	8'd0	未知指令
指令具体类型	INSTR_ADDU	8'd1	addu 指令
指令具体类型	INSTR_SUBU	8'd2	subu 指令
指令具体类型	INSTR_LUI	8'd3	lui 指令
指令具体类型	INSTR_ORI	8'd4	ori 指令
指令具体类型	INSTR_LW	8'd5	lw 指令
指令具体类型	INSTR_SW	8'd6	sw 指令
指令具体类型	INSTR_BEQ	8'd7	beq 指令
指令具体类型	INSTR_NOP	8'd8	nop 指令
cm_rf_write_addr	CM_RF_WRITE_ADDR_IM_DATA_20_16	1'b0	m_rf_write_addr 输入来源为 im.data[20:16]
cm_rf_write_addr	CM_RF_WRITE_ADDR_IM_DATA_15_11	1'b1	m_rf_write_addr 输入来源为 im.data[15:11]
cm_rf_write_data	CM_RF_WRITE_DATA_ALU_RESULT	1'b0	m_rf_write_data 输入来源为 alu.result
cm_rf_write_data	CM_RF_WRITE_DATA_DM_READ_RESULT	1'b1	m_rf_write_data 输入来源为 dm.read_result
cm_alu_num2	CM_ALU_NUM2_RF_READ_RESULT2	1'b0	m_alu_num2 输入来源为 rf.read_result2
cm_alu_num2	CM_ALU_NUM2_EXT_RESULT	1'b1	m_alu_num2 输入来源为 ext.result

端口定义

端口	类型	位宽	功能
curr_instr	输入	32	当前指令
cm_rf_write_addr	输出	1	控制 m_rf_write_addr
cm_rf_write_data	输出	1	控制 m_rf_write_data
cm_alu_num2	输出	1	控制 m_alu_num2
cw_npc_jump_mode	输出	1	控制 npc.jump_mode
cw_pc_enable	输出	1	控制 pc.enable
cw_im_enable	输出	1	控制 im.enable
cw_rf_write_enable	输出	1	控制 rf.write_enable
cw_alu_op	输出	1	控制 alu.op

cw_ext_mode	输出 1	控制 ext.mode
cw_dm_write_enable	输出 1	控制 dm.write_enable

功能

由于 pc.enable 和 im.enable 都是相应的启用值，所以真正实现控制模块的时候，不会控制这两个值，而是把它们都设成对应的常量。

首先，识别指令具体类型。这里只使用一个 8 位的 wire 类型变量存放指令类型，而不是多个 wire 类型变量，这样其实也是一种类似 enum 的做法。识别指令类型主要还是看 op 字段，再更细致一点地去看 R 型指令的 funct 字段。当然也可以先看指令是否为 R 型指令，但是这样逻辑上有点互相缠绕，所以个人认为识别到具体类型比较好。

然后，识别指令类型。这里主要是为了以后的暂停和转发逻辑做准备，代码中不一定实现（虽然一定有宏定义），利用一个 2 位的 wire 类型变量存放指令类型。

最后，做相对应的操作，并输出控制信号。这个识别出指令具体类型以后，按照表格中每个单元格的意义实现即可，主要考虑的是一个判断问题。

注意事项

1. 输入整个指令在电路设计中实际上没用，是为了方便 debug
2. 控制信号和最终 CPU 实现中相应的连接数据通路和控制部分的 wire 有些名字是重复的，在 Verilog 中语法没错误，没有分开
3. **关于各 MUX 选择哪个输入的宏的值是跟 MUX 接线有关的**
4. **谨慎使用宏函数**

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
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
- 对新指令指定相应的控制信号
- 对新指令指定相应的 MUX 控制信号
- 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 `alu`，**注意有符号 / 无符号和运算溢出问题和改 `alu` 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，p4 暂时还不用改 `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 `alu` 的比较方式
- 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

- 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**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
- 如果有新的 `rf` 的值
    - 注意要接线接过来，然后加 MUX
    - 如果是返回地址，**最好是先接过当前 PC，然后无符号数 +4**
    - 注意 `rf` 的值是否写入可以跟 `rf.we` 配合

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**
    - `npc` 看与 `alu` 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**
    - `ext` 看扩展模式对不对，**注意扩展的是哪些数字和扩展模式**
    - `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**
    - `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**
    - `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**

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，跟以前指令的关系
- 如果是 L/S 指令，跟内存的关系
- 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
- 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，那也没办法
- 新部件部署在哪一级?
    - 单周期处理器这样不重要，但是还是要做一遍
    - 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

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 `alu` 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 / 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**
    - 分析一下新部件的功能
    - 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