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 端口定义

表 1 端口定义

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 的有符号比较结果
num	输入	16	输入的立即数
next_pc	输出	32	下一个 PC

1.3 宏定义

表 2 宏定义

类别	值	意义
jump_mode	3’b000	不要跳转
jump_mode	3’b001	当 ALU 输入的比较结果相等时跳转

alu_comp_result 的相应数值代表的意义，与相应的宏有关。

1.4 功能

若 jump_mode == 3'b000，则令 next_pc = $unsigned(curr_pc) + $unsigned(4)。

若 jump_mode == 3'b001，则 alu_comp_result == 2'b00 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 == 3'b000 时相同的步骤。

若 jump_mode 为其它值，则 next_pc 为 x。

1.5 注意事项

1. NPC 是在内部进行符号扩展，不用 ext。

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。

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
rst	输入	1	异步复位信号
next_pc	输入	32	NPC 计算得来的下一个 PC 地址
curr_pc	输出	32	PC 地址

2.3 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。在每个时钟上升沿，把 PC 部件中保存的当前 PC 的值更新成 next_pc 的值。

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。

3 程序存储器

3.1 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。

3.2 端口定义

表 4 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

3.3 功能

有 32 个 32 位存储器，代表其中存储的指令。它们初值由 ROM 中加载值的方式给定。无论什么时候，`result = ROM[addr[6:2]]`。

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 5 端口定义

端口	类型	位宽	功能
clk	输入	1	时钟信号
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 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 == 1'b1 且 write_addr != 5'b0，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。

无论什么时候，把 read_addr1 指代的寄存器的值输出到 read_result1 中。对 read_addr2 和 read_result2 的相应操作相同。

5 扩展器

5.1 功能

扩展器是专门执行扩展整数功能的运算。它能做到 16 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、无符号扩展和其它模式。

5.2 端口定义

表 6 端口定义

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

5.3 宏定义

表 7 宏定义

类别	值	意义
mode	3'b000	符号扩展
mode	3'b001	无符号扩展
mode	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

5.4 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 = x。

6 ALU

6.1 原理

ALU 负责两个 32 位整数的运算。它可以负责数学运算和逻辑运算。易知它是纯组合逻辑。

6.2 端口定义

表 8 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	3	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

6.3 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 9 宏定义

类别	值	意义
op	3b'000	加法运算
op	3b'001	减法运算
op	3b'010	按位与运算
op	3b'011	按位或运算
.*comp_result	2b'00	等于
.*comp_result	2b'01	大于
.*comp_result	2b'10	小于

6.4 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。

如果 op 的值为非法操作，就令 result 为 x。

.*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。

7 数据存储器

7.1 原理

数据存储器是存储数据的地方。

为了简便，数据存储器由许多寄存器实现。

7.2 端口定义

表 10 端口定义

端口	类型	位宽	功能
clk	输入	1	时钟信号
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

7.3 功能

该部件为时序部件。

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

在每个时钟上升沿，若 write_enable == 1'b1，则 write_addr [6:2] 这个地址对应的 32 位字写入 write_data 对应的值。

任何时候，read_result 的值为 read_addr[6:2] 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存储空间，更现实。但是实际上这样对地址空间进行了限制。

7.4 注意事项

1. 直接忽略地址后两位

2. 地址空间是被截断的，看起来是 32 位，实际上不是
3. 用到的 RAM 是分开读写端口的

8 单周期数据通路

8.1 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。

为了简便，PC、NPC 和指令存储器结合在一起，形成 IFU。

8.2 分析

p3 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。

表 11 分析

指令	addu	subu	lui
npc.curr_pc	pc.curr_pc	pc.curr_pc	pc.curr_pc
npc.alu_comp_result			

指令	addu	subu	lui
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			

表 12 分析

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

指令	ori	lw	sw
dm.write_data			rf.read_result2

表 13 分析

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:

表 14 分析

端口	所有的信号来源	MUX 名称
rf.write_addr	im.data[20:16], im.data[15:11]	无（见下）
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。

实际上,为了扩展,MUX 可能会留出更多的空位,位数也会增加。`rf.write_addr` 没有对应的 MUX，这是因为最终确定 `rf.write_addr`，是由控制单元确定的，这虽然牺牲了一定的设计优雅程度，但是增强了设计的灵活性。

9 控制单元

9.1 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单元完成。控制通过已有的控制信号和数据通路的分叉完成控制。

9.2 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。

表 15 分析

指令	addu	subu
rf_write_addr	curr_instr[15:11]	curr_instr[15:11]
m_rf_write_data	alu.result	alu.result
m_alu_num2	rf.read_result2	rf.read_result2
npc.jump_mode	3'b000	3'b000

指令	addu	subu
rf.write_enable	1'b1	1'b1
alu.op	3'b000	3'b001
ext.mode	2'b00	2'b00
dm.write_enable	1'b0	1'b0

表 16 分析

指令	lui	ori
m_rf_write_addr	curr_instr[20:16]	curr_instr[20:16]
m_rf_write_data	alu.result	alu.result
m_alu_num2	ext.result	ext.result
npc.jump_mode	3'b000	3'b000
rf.write_enable	1'b0	1'b0
alu.op	3'b011	3'b011
ext.mode	2'b10	2'b00
dm.write_enable	1'b0	1'b0

表 17 分析

指令	lw	sw
m_rf_write_addr	curr_instr[20:16]	curr_instr[20:16]
m_rf_write_data	dm.read_result	alu.result
m_alu_num2	ext.result	ext.result
npc.jump_mode	3'b000	3'b000
rf.write_enable	1'b1	1'b0
alu.op	3'b000	3'b000
ext.mode	2'b01	2'b01
dm.write_enable	1'b0	1'b1

表 18 分析

指令	beq	nop
m_rf_write_addr	#curr_instr[20:16]	#curr_instr[20:16]

指令	beq	nop
m_rf_write_data	#alu.result	#alu.result
m_alu_num2	rf.read_result2	#rf.read_result2
npc.jump_mode	3'b001	3'b000
rf.write_enable	1'b0	1'b0
alu.op	3'b011	3'b011
ext.mode	#2'b00	#2'b00
dm.write_enable	1'b0	1'b0

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。

9.3 宏定义

表 19 宏定义

类别	值	意义
指令魔数	6’b000000	R 型指令 op 字段魔数
指令魔数	6’b100001	addu 指令 funct 字段魔数
指令魔数	6’b100011	subu 指令 funct 字段魔数
指令魔数	6’b001111	lui 指令 op 字段魔数
指令魔数	6’b001101	ori 指令 op 字段魔数
指令魔数	6’b100011	lw 指令 op 字段魔数
指令魔数	6’b101011	sw 指令 op 字段魔数
指令魔数	6’b000100	beq 指令 op 字段魔数
指令魔数	6’b000000	nop 指令 funct 字段魔数

9.4 端口定义

表 20 端口定义

端口	类型	位宽	功能
curr_instr	输入	32	当前指令
cw_rf_read_addr1	输出	5	控制 rf.read_addr1

端口	类型	位宽	功能
cw_rf_read_addr2	输出	5	控制 rf.read_addr2
cw_rf_write_addr	输出	1	控制 rf.write_addr
cm_rf_write_data	输出	2	控制 m_rf_write_data
cm_alu_num2	输出	1	控制 m_alu_num2
cw_npc_jump_mode	输出	3	控制 npc.jump_mode
cw_rf_write_enable	输出	1	控制 rf.write_enable
cw_alu_op	输出	3	控制 alu.op
cw_ext_mode	输出	2	控制 ext.mode
cw_dm_write_enable	输出	1	控制 dm.write_enable

9.5 功能

首先，识别指令具体类型。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，所以可能识别到具体类型比较好。

然后，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。

9.6 真值表

最终生成的控制信号真值表如下：

表 21 真值表

指令类型	op	func
addu	6'b000000	6'b100001
subu	6'b000000	6'b100011
lui	6'b001111	6'bxxxxxx
ori	6'b001101	
lw	6'b100011	
sw	6'b101011	
beq	6'b000100	

表 22 真值表

信号	通用名	addu	subu	lui	ori	lw	sw	beq	nop
（无）	RegDst	1	1	0	0	0	X	X	X
cm_alu_num2	ALUSrc	0	0	1	1	1	1	X	X
cm_rf_write_data[0]	MemtoReg	0	0	0	0	1	X	X	X
cw_rf_write_enable	RegWrite	1	1	1	1	1	0	0	0
cw_dm_write_enable	MemWrite	0	0	0	0	0	1	0	0
cw_npc_jump_mode[0]	nPC_sel	0	0	0	0	0	0	1	0
cw_ext_op[1]	ExtOp[1]	X	X	1	0	0	0	X	X
cw_ext_op[0]	ExtOp[0]	X	X	0	0	1	1	X	X
cw_alu_op[1]	ALUctr[1]	0	0	1	1	0	0	X	X
cw_alu_op[0]	ALUctr[0]	0	1	1	1	0	0	X	X

10 CPU

10.1 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。

CPU 在模块结构中作为顶层模块而存在。

10.2 端口定义

表 23 端口定义

端口	类型	位宽	功能
rst	输入	1	复位信号
Instr	输出	32	32 位指令信号
RegWrite	输出	1	GRF 写入控制信号
RegAddr	输出	5	GRF 写入地址
RegData	输出	32	GRF 写入数据
MemWrite	输出	1	DM 写入控制信号

端口	类型	位宽	功能
MemAddr	输出	5	DM 写入地址
MemData	输出	32	DM 写入数据

10.3 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那部分，在控制部分的文档中。

10.4 功能

CPU 需要的外部数据输入是极少的，主要是输出用于评测。

11 测试程序

```
ori $1, 0x2 # $1 <= 0x00000002
ori $2, 0x2 # $2 <= 0x00000002
addu $3, $1, $2 # $3 <= 0x00000004

lui $4, 0xffff # $4 <= 0xffff0000
ori $4, 0xffff # $4 <= 0xffffffff
subu $5, $0, $4 # $5 <= 0x00000001

sw $4, 4($0) # *0x00000004 <= 0xffffffff
lw $6, 4($0) # $6 <= 0xffffffff

beq $4, $5, bad # no jump
```

```
lui $7, 0xdead # $7 <= 0xdead0000
lui $8, 0xbeef # $8 <= 0xbeef0000
bad:

beq $0, $0, front # jump
back:
ori $9, 0xbabe # $9 <= 0x0000babe

front:
beq $1, $1, back # jump
```

期望结果在注释中。

12 思考题

12.1 模块规格

1. 优点：PC 默认按照字对齐，NPC 模块和 IM 模块更好设计。而且，跳转指令也可以省略后面需要填充的两个 0。
   缺点：在之后实现写入 PC 的指令，和检测无效 PC（PC 没有按照字对齐）时会有比较大的困难，写入 PC 无法保留低两位。

2. IM 使用 ROM 是因为指令不是需要更改的，而且指令需要自动化加载，所以比较合理。DM 需要可读可写，而且使用现成的数据内存也比较合理，同时这样可以简化 DM 的容量扩展。GRF 除了可以使用寄存器，也可以使用 DM，因为二者在指定读写地址这方面类似。但是，对于 `$0`，需要特殊处理。综上所述，这种做法比较合理。
   其实，也可以考虑用 DM 实现 GRF，当然在真正制造 CPU 时，不能这么做。IM 和 DM 分开，其实类似哈佛架构，而不是标准的冯诺依曼架构。这样能够简化体系结构，但是不支持指令的修改，同时寻址也不统一。可以把 IM 改成用 RAM，来达到统一寻址的目的。而且，在真正的系统中，是有 cache 的，这

里 IM 和 DM 都没有 cache，是一种简化。

12.2 控制器设计

1. 用等价的 Verilog 表达式表述。

```
assign rtype = ~op[5] & ~op[4] & ~op[3] & ~op[2] & ~op
   [1] & ~op[0];
assign addu = rtype & func[5] & ~func[4] & ~func[3] & ~
   func[2] & ~func[1] & func[0];
assign subu = rtype & func[5] & ~func[4] & ~func[3] & ~
   func[2] & func[1] & func[0];
assign lui = ~op[5] & ~op[4] & op[3] & op[2] & op[1] &
   op[0];
assign ori = ~op[5] & ~op[4] & op[3] & op[2] & ~op[1] &
   op[0];
assign lw = op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] &
   op[0];
assign sw = op[5] & ~op[4] & op[3] & ~op[2] & op[1] & op
   [0];
assign beq = ~op[5] & ~op[4] & ~op[3] & op[2] & ~op[1] &
    ~op[0];
assign RegDst = addu | subu;
assign ALUSrc = lui | ori | lw | sw;
assign MemtoReg = lw;
assign RegWrite = addu | subu | lui | ori | lw | sw;
assign nPC_sel = beq;
assign ExtOp[1] = lui;
assign ExtOp[0] = lw | sw;
assign ALUctr[1] = lui | ori;
```

```
assign ALUctr[0] = subu | lui | ori;
```

2.

```
assign RegDst = rtype & func[5] & ~func[4] & ~func[3] &
   ~func[2] & func[0];
assign ALUSrc = (~op[5] & ~op[4] & op[3] & op[2] & op
   [0]) | (op[5] & ~op[4] & ~op[2] & op[1] & op[0]);
assign MemtoReg = op[5] & ~op[4] & ~op[3] & ~op[2] & op
   [1] & op[0];
assign RegWrite = (rtype & func[5] & ~func[4] & ~func[3]
    & ~func[2] & func[0]) | (~op[5] & ~op[4] & op[3] &
   op[2] & ~op[0]) | (op[5] & ~op[4] & ~op[2] & op[1] &
   op[0]);
assign nPC_sel = ~op[5] & ~op[4] & ~op[3] & op[2] & ~op
   [1] & ~op[0];
assign ExtOp[1] = ~op[5] & ~op[4] & op[3] & op[2] & op
   [1] & op[0];
assign ExtOp[0] = op[5] & ~op[4] & ~op[2] & op[1] & op
   [0];
assign ALUCtr[1] = ~op[5] & ~op[4] & op[3] & op[2] & op
   [0];
assign ALUCtr[0] = (rtype & func[5] & ~func[4] & ~func
   [3] & ~func[2] & func[1] & func[0] & ~op[5] & ~op[4]
   & op[3] & op[2] & op[0];
```

3. 因为如果不把 nop 指令加入控制信号真值表，或者根据最简表达式，所有更改 DM 和 GRF 的信号都不会激活，PC 也会正常改变。所以 nop 指令会没有效果，与预期相符。

12.3 测试 CPU

1. 可以通过判断 PC 的数值来操控 DM 片选，从而可以实现对 IM 的读取。若要读取的地址作为无符号数大于 `0x3000`，则片选信号激活，把地址减去对应的偏移，让新地址信号作用于 IM，读出 IM 的值。

2. 优点：证明相对可理解、可自动化、证明的结果比较可靠、对新特性的容忍程度较高、理论性较强。缺点：模型和实际实现需要保证对应、需要数学工具、实践性较弱。

13 技巧

13.1 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

13.2 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，p3 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

13.3 如何有效调试

1. 定位出错指令

- 平时可以用对拍
- 在考场上主要靠看数据和猜
  - 看数据大法
    - 前面一堆 0 位或者 1 位出错的，一般是移位指令
    - 前面数据乱了的，一般是乘除法指令
    - 数据差 1 的，一般是条件设置指令
    - 数据错了的，一般是跳转指令，少部分是控制信号
  - 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

13.4 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
- 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

13.5 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