CPU 设计文档

一、模块规格

1. IFU

(1)模块接口

表 1 IFU 模块接口

信号名	方向	描述
Imm[15:0]	I	接收跳转指令的 offset
PCSrc[1:0]	I	控制 PC 是否跳转的控制信号 00:正常执行 PC + 4 01:按 J 型指令的 index 跳转 10:按 beq 指令的 offset 跳转
Reset	I	复位信号，1 时清零 PC
jimm[25:0]	I	J 指令的跳转 index
Instr[31:0]	O	当前指令
PC[31:0]	O	当前 PC 值

(2)功能定义

表 2 IFU 功能定义

序号	功能名称	功能描述
1	复位	当复位信号有效时，PC 设置为 0x00000000
2	取指令	PC 从 IM 中取出指令
3	计算下一条指令的地址	如果当前指令不是 beq，PC←PC+4 如果当前指令是 beq，ALU 执行比较大小结果为 0，则 PC←PC+4 如果当前指令是 beq，ALU 执行比较大小结果为 1，PC←PC+4+sign_extend(offset||00) 如果当前指令是 j/jal，PC←PC31..28||instr_index||02

2. ALU

(1)模块接口

表 3 ALU 模块接口

信号名	方向	描述
A[31:0]	I	接收第一个操作数
B[31:0]	I	接收第二个操作数
ALUOp[2:0]	I	决定 ALU 进行何种操作类型的控制信号 000：无符号加法 001：无符号减法 010：或运算 011：比较大小 100：异或运算
Res[31:0]	O	ALU 输出结果

(2)功能定义

表 4 ALU 功能定义

序号	功能名称	功能描述
1	无符号加法	Res = A + B
2	无符号减法	Res = A - B
3	或运算	Res = A or B
4	比较大小	Res = (A == B)?1:0
5	异或运算	Res = A xor B

3.DM

(1)模块接口

表 5 DM 模块接口

信号名	方向	描述
Reset	I	为 1 时清空 DM
Addr[4:0]	I	读/写存储器的地址
Data[31:0]	I	写存储器的数据
MemWrite	I	为 1 时写存储器
MemRead	I	为 1 时读存储器

ReadData	0	从存储器中读出的值

(2)功能定义

表 6 DM 功能定义

序号	功能名称	功能描述
1	读存储器	MemRead 为 1 时输出存储器 Addr 位置对应的值
2	写存储器	MemWrite 为 1 时向存储器 Addr 位置写入对应的值
3	复位	Reset 为 1 时清零存储器

4. GRF

(1)模块接口

表 7 GRF 模块接口

信号名	方向	描述
Reset	I	Reset 为 1 时清零寄存器
RegRead1[4:0]	I	第一个需要读取内容的寄存器编号
RegRead2[4:0]	I	第二个需要读取内容的寄存器编号
RegWrite	I	寄存器写入的使能信号
RegAddr[4:0]	I	需要写入数据的寄存器编号
RegData[31:0]	I	需要写入寄存器的数据
ReadData1[31:0]	O	输出编号为 RegRead1 的寄存器的值
ReadData2[31:0]	O	输出编号为 RegRead2 的寄存器的值

(2)功能定义

表 8 GRF 功能定义

序号	功能名称	功能描述
1	读取寄存器的值	能够将编号为 RegRead1、RegRead2 的寄存器的值分别通过 ReadData1， ReadData2 输出
2	写寄存器	在 RegWrite 信号为 1 时，将 RegData 写入编号为 RegAddr 的寄存器中
3	复位	Reset 为 1 时清空所有寄存器

二、Controller

表 9 controller 真值表

funct	100001	100011	----------								
op	000000	000000	001101	100011	101011	000100	001111	000010	000011	100001	001110
	addu	**subu**	**ori**	**lw**	**sw**	**beq**	**lui**	**j**	**jal**	**lh**	**xori**
RegDst[1:0]	01	01	00	00	x	x	00	x	10	00	00
RegWrite	1	1	1	1	0	0	1	0	1	1	1
ALUOp[2:0]	000	001	010	000	000	011	000	x	000	000	100
ALUSrcB[2:0]	000	000	001	010	010	x	011	x	100	010	001
MemtoReg[1:0]	10	10	10	00	x	x	10	x	10	01	10
MemRead	0	0	0	1	0	0	0	0	0	1	0
MemWrite	0	0	0	0	1	0	0	0	0	0	0
isbeq	0	0	0	0	0	1	0	0	0	0	0
ALUA[1:0]	00	00	00	00	00	00	01	00	10	00	00
isj	0	0	0	0	0	0	0	1	1	0	0

布尔表达式：

RegDst1 = jal

RegDst0 = addu + subu

RegWrite = addu + subu + ori + lw + lui + jal + lh + xori

ALUOp2 = xori

ALUOp1 = ori + beq

ALUOp0 = subu + beq

ALUSrcB2 = jal

ALUSrcB1 = lw + sw + lui + lh

ALUSrcB0 = ori + xori + lui

MemtoReg1 = ~(lw+lh) = ~lw~lh（以~代表非）

MemtoReg0 = ~lh

MemRead = lh + lw

MemWrite = sw

isbeq = beq

ALUA1 = jal

ALUA0 = lui

isj = j

表 10 主控制模块端口定义

信号名	方向	描述
op[5:0]	I	接收指令的 31..26 位
funct[5:0]	I	接收指令的 5..0 位
addu	0	为 1 时说明当前指令为 addu 指令
subu	0	为 1 时说明当前指令为 subu 指令
ori	0	为 1 时说明当前指令为 ori 指令
lw	0	为 1 时说明当前指令为 lw 指令
sw	0	为 1 时说明当前指令为 sw 指令
beq	0	为 1 时说明当前指令为 beq 指令
lui	0	为 1 时说明当前指令为 lui 指令
j	0	为 1 时说明当前指令为 j 指令
lh	0	为 1 时说明当前指令为 lh 指令
xori	0	为 1 时说明当前指令为 xori 指令

表 11 副控制模块端口定义

信号名	方向	描述
addu	I	为 1 时说明当前指令为 addu 指令
subu	I	为 1 时说明当前指令为 subu 指令
ori	I	为 1 时说明当前指令为 ori 指令
lw	I	为 1 时说明当前指令为 lw 指令
sw	I	为 1 时说明当前指令为 sw 指令
beq	I	为 1 时说明当前指令为 beq 指令

lui	I	为 1 时说明当前指令为 lui 指令
j	I	为 1 时说明当前指令为 j 指令
lh	I	为 1 时说明当前指令为 lh 指令
xori	I	为 1 时说明当前指令为 xori 指令
RegDst[1:0]	0	控制寄存器写入地址的信号 00：写入 Rt 寄存器 01：写入 Rd 寄存器 10: 写入 31 号寄存器
RegWrite	0	控制寄存器写入的使能信号， 1 时写寄存器
ALUOp[2:0]	0	ALU 进行何种操作的控制信号， ALU 模块已进行介绍
ALUSrcB[2:0]	0	ALU 的第二个端口接入何种操作数的控制信号 000：RegData2 001:zero_extend(imm) 010:sign_extend(imm) 011:imm||0^16 100:4
MemtoReg[1:0]	0	控制寄存器写入何种数据的控制信号， 00：将 DM 的数据写入寄存器 01：将 DM 的数据经过取半字后写入寄存器 10：将 ALU 的结果写寄存器
MemRead	0	为 1 时读存储器
MemWrite	0	为 1 时写存储器
isbeq	0	为 1 时当前信号为 beq
ALUA[1:0]	0	ALU 的第一个端口接入何种操作数的控制信号 00:RegData1 01:0 10:PC

isj	0	为 1 时当前信号为 j

三、测试程序

程序代码及期望结果：

```
ori $t0, $0, 1 #t0 寄存器值变为 1

ori $t1, $0, 2 #t1 寄存器值变为 2

###

lui $a0, 123  #a0 变为 0x007b0000

lui $a1, 0xffff  # a1 变为 0xffff0000

ori $a1, $a1, 0xffff   #a1 变为 0xffffffff

####a1 负数-1

addu $t3, $t0, $t1  # t3 寄存器值变为 3

addu $t3, $t1, $a1  # t3 寄存器值变为 1

addu $t3, $a1, $a1  # t3 寄存器值变为 0xfffffffe

#####

subu $t4, $t1, $t0  # t4 寄存器值变为 1

subu $t4, $t0, $a1  # t4 寄存器值变为 2

subu $t4, $t3, $a1  # t4 寄存器值变为 0xffffffff

####

sw $t0, 0($0) # DM 第一个字存入 1

sw $t1, 4($0) # DM 第二个字存入 2

##

lw $t5, 0($0) # t5 寄存器值变为 1

lw $t6, 4($0) # t6 寄存器值变为 2
```

```
##
beq $t0, $t5, then  # 跳转至 then
beq $t0, $t6, no #无效
addu $t0, $t0, $t0  #无效
then:
addu $t1, $t1, $t1  # t1 寄存器值变为 4
no:
addu $t4, $t4, $t4  # t4 寄存器值变为 0xfffffffe
###
j here  # 跳转至 here
addu $t4, $t4, $t4  # 无效
here:
addu $t1, $t1, $t1  # t1 寄存器值变为 8
jal that # 跳转至 that，且将下条指令地址存入$31
addu $t1, $t1, $t1  # 无效
that:
lh $t5, 0($0) # t5 寄存器值变为 1
xori $t2, $t1, 0x81 # t2 寄存器值变为 0x00000089
xori $t2, $t1, 0x444   # t2 寄存器值变为 0x0000044c
```

四、思考题

（一）

1. 若 PC（程序计数器）位数为 30 位，试分析其与 32 位 PC 的优劣。

30 位 PC 等于 32 位 PC 最后两位 0 舍弃掉，因此保证了 PC 是 4 的倍数，不会产生异常情况，但换算到 IM 的地址时会麻烦。

2. 现在我们的模块中 IM 使用 ROM， DM 使用 RAM， GRF 使用寄存器，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。

合理。

我们只需要读取指令，因此用 ROM 即可。DM 我们既需要读也需要写，因此用 RAM。GRF 要求它在时钟下跳沿存入数据，其他时候输出保持稳定，这恰好时寄存器的功能。

（二）

1.结合上文给出的样例真值表，给出 RegDst， ALUSrc， MemtoReg，RegWrite, nPC_Sel, ExtOp 与 op 和 func 有关的布尔表达式（表达式中只能使用"与、或、非"3 种基本逻辑运算。）

RegDst = $\overline{Op_5}\,\overline{Op_4}\,\overline{Op_3}\,\overline{Op_2}\,\overline{Op_1}\,\overline{Op_0}$

ALUSrc = $\overline{Op_5}\,\overline{Op_4}\,Op3\,Op2\,\overline{Op_1}\,Op0 + Op5\,\overline{Op_4}\,\overline{Op_2}\,Op1\,Op0$

MemtoReg = $Op5\,\overline{Op_4}\,\overline{Op_3}\,\overline{Op_2}\,Op1\,Op0$

RegWrite = $\overline{Op_5}\,\overline{Op_4}\,\overline{Op_3}\,\overline{Op_2}\,\overline{Op_1}\,\overline{Op_0} + \overline{Op_5}\,Op3\,Op2\,\overline{Op_1} + Op5\,\overline{Op_3}\,\overline{Op_2}\,Op1$

MemWrite = $Op5\,\overline{Op_4}\,Op3\,\overline{Op_2}\,Op1\,Op0$

nPC_sel = $\overline{Op_5}\,\overline{Op_4}\,\overline{Op_3}\,Op2\,\overline{Op_1}\,\overline{Op_0}$

ExtOp = $Op5\,\overline{Op_4}\,\overline{Op_2}\,Op1\,Op0$

2.充分利用真值表中的 X 可以将以上控制信号化简为最简单的表达式，请给出化简后的形式。

RegDst = $\overline{Op_5}\,\overline{Op_4}\,\overline{Op_3}\,\overline{Op_1}\,\overline{Op_0}$

ALUSrc = $\overline{Op_5}\,\overline{Op_4}\,Op3\,Op2\,\overline{Op_1}\,Op0 + Op5\,\overline{Op_4}\,\overline{Op_2}\,Op1\,Op0$

MemtoReg = $Op5\,\overline{Op_4}\,\overline{Op_2}\,Op1\,Op0$

RegWrite = $\overline{Op_5}\,\overline{Op_4}\,\overline{Op_3}\,\overline{Op_2}\,\overline{Op_1}\,\overline{Op_0} + \overline{Op_5}\,Op3\,Op2\,\overline{Op_1} + Op5\,\overline{Op_3}\,\overline{Op_2}\,Op1$

MemWrite = $Op5\,\overline{Op_4}\,Op3\,\overline{Op_2}\,Op1\,Op0$

nPC_sel = $\overline{Op_5}\,\overline{Op_4}\,\overline{Op_3}\,Op2\,\overline{Op_1}\,\overline{Op_0}$

ExtOp = $Op5\,\overline{Op_4}\,\overline{Op_2}\,Op1\,Op0$

3.事实上，实现 nop 空指令，我们并不需要将它加入控制信号真值表，为什么？请给出你的理由。

Nop 会被已有的 controller 判为 R 型指令，rs=0,rt=0, rd=0，即使写寄存器使能信号为 1，它表示的也是将 0 写入$0，$0 不可修改，故不会产生实际效果。

（三）

1.前文提到，“可能需要手工修改指令码中的数据偏移”，但实际上只需再增加一个 DM 片选信号,就可以解决这个问题。请阅读相关资料并设计一个 DM 改造方案使得无需手工修改数据偏移。

当 Mars 设置为 Text at address 0 时,Data 段从 0x00002000 开始，一直到 0x00003000，故有 32*32 个字，需要 32 个 RAM(设计要求 RAM 为 32*32bit)，经过一个 5 位译码器后产生 32 位片选信号连接到每个 RAM 的 chip select 口。

2.除了编写程序进行测试外，还有一种验证 CPU 设计正确性的办法——形式验证。 **形式验证**的含义是根据某个或某些形式规范或属性，使用数学的方法证明其正确性或非正确性。请搜索“形式验证（Formal Verification)"了解相关内容后，简要阐述相比与测试，形式验证的优劣。

优点：

1. 由于形式验证是借用数学上的方法将待验证电路和功能描述直接进行比较，因此我们不必考虑如何获得测试的数据。
2. 形式验证是对指定描述的所有可能的情况进行验证，而不是仅仅对其中的一个子集进行多次试验（测试很难穷尽所有可能）。
3. 形式验证可以进行从系统级到门级的验证，而且验证时间短，有利于尽早、尽快地发现和改正电路设计中的错误，有可能缩短设计周期。

缺点：

不能有效的验证电路的一些性能，例如功耗、延时。