计算机组成原理实验报告

一、CPU 设计方案综述

总体设计综述

本 **CPU** 为 **Logisim** 实现的单周期 **CPU**，支持指令集包含 `addu,subu,ori,lw,sw,beq,lui,nop` 。为了实现这些功能，**CPU** 主要包含了 **IM,PC,GRF,Controller,Extender,Comp,Decoder** 。

关键模块定义

GRF

信号名	方向	信号描述
`Clk`	*I*	时钟信号
`Reset`	*I*	异步复位信号
`RD`	*I*	`RegDst` 信号
`MR`	*I*	`MemToReg` 信号
`Link`	*I*	`Link` 信号
`LR`	*I*	`LinkReg` 信号
`Byte`	*I*	写入 `Byte` 信号
`Half`	*I*	写入 `Half` 信号
`Sign`	*I*	符号信号
`WE`	*I*	写入使能信号
`Offset`	*I*	偏移量
`rs[4:0]`	*I*	`rs` 的地址
`rt[4:0]`	*I*	`rt` 的地址
`rd[4:0]`	*I*	`rd` 的地址
`AluData[31:0]`	*I*	经过 `ALU` 运算后的数据
`MemData[31:0]`	*I*	`DM` 中读出的数据
`nPCData[31:0]`	*I*	`PC` 中读出的地址
`RD1[31:0]`	*O*	读出 `rs` 寄存器的数据
`RD2[31:0]`	*O*	读出 `rt` 寄存器的数据
`RegData[31:0]`	*O*	写入的数据
`RegAddr[4:0]`	*O*	写入的寄存器地址

序号	功能名称	功能描述
1	复位	当复位信号有效，则寄存器堆的寄存器全复位为 `0x00000000`
2	读寄存器	将对应的地址的寄存器输出数值
3	`AluData` 写入寄存器	将运算后的数值写入进对应的地址的寄存器内
4	`MemData` 写入寄存器	将存储器读出的数值写入进对应的地址的寄存器内
5	`nPCData` 写入寄存器	将指令地址写入进对应的地址的寄存器内

ALU

信号名	方向	信号描述
`A[31:0]`	I	A 运算数
`B[31:0]`	I	B 运算数
`imm[31:0]`	I	立即数
`S[4:0]`	I	移位数
`AS`	I	控制参与运算数信号
`Ctrl[3:0]`	I	控制 ALU 的操作信号
`Out[31:0]`	O	运算结果

序号	`Ctrl`	功能名称
1	`0000`	无符号加
2	`0001`	无符号减
3	`0010`	逻辑左移
4	`0011`	逻辑右移
5	`0100`	算术右移
6	`0101`	按位与
7	`0110`	按位或
8	`0111`	按位异或
9	`1000`	无符号比较
10	`1001`	符号比较

EXT

信号名	方向	信号描述
`Ext_OP[1:0]`	I	控制扩展方式
`imm16[15:0]`	I	16 位立即数
`imm26[25:0]`	I	26 位立即数
`Out[31:0]`	O	扩展后数据

序号	`Ext_OP`	功能名称
1	`00`	零扩展 16 位数据
2	`01`	符号扩展 16 位数据
3	`10`	扩展后把 16 位数据移至到高位
4	`11`	零扩展 26 位数据

Controller

信号名	方向	信号描述
`Opcode[5:0]`	I	输入 `Opcode` 指令
`Funct[5:0]`	I	输入 `Funct` 指令
`RegDst`	O	选择 `rd` 和 `rt` 作为写入目标寄存器
`AluSrc`	O	选择运算的数值
`RegWrite`	O	寄存器堆写入使能
`MemToReg`	O	选择数据写入寄存器
`MemWrite`	O	`DM` 写入使能
`Branch[2:0]`	O	检测条件偏移指令
`Jump`	O	检测 `J` 指令信号
`Link`	O	将 `PC + 4` 写入寄存器信号
`LinkReg`	O	写入地址进寄存器并有指定寄存器地址信号
`Return`	O	将寄存器的地址写入 `PC` 信号
`Byte`	O	传输 `Byte` 使能信号
`Half`	O	传输 `Half` 使能信号
`Sign`	O	传输 `Sign` 使能信号
`AluCtrl[3:0]`	O	控制运算方式
`Ext_Op[1:0]`	O	控制扩展方式

funct	100000	100010	-	-	-	-	-	000000	-	001000	001001	-	-	-	-	000000
opcode	000000	000000	001101	100011	101011	000100	001111	000000	000011	000000	000000	100000	100001	101000	101001	000000
Instruction	addu	subu	ori	lw	sw	beq	lui	nop	jal	jr	jalr	lb	lh	sb	sh	sll
RegDst	1	1	0	0	x	x	0	1	0	x	0	0	0	x	x	1
AluSrc	0	0	1	1	1	1	1	0	0	0	0	1	1	1	1	0
RegWrite	1	1	1	1	0	0	1	1	1	0	1	1	1	0	0	1
MemToReg	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0
MemWrite	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0
Branch[2:0]	x	x	x	x	x	equal	x	x	x	x	x	x	x	x	x	x
Jump	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0
Link	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
LinkReg	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
Return	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
Byte	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
Half	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
Sign	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
ALUCtrl[3:0]	addu	subu	or	addu	addu	compu	addu	sll	x	x	x	addu	addu	addu	addu	sll
Ext_Op[1:0]	x	x	sign16	sign16	sign16	sign16	upper	x	zero26	x	x	sign16	sign16	sign16	sign16	x

Comp

信号名	方向	信号描述
`In[2:0]`	I	输入状态
`bgt`	O	检测大于状态
`beq`	O	检测等于状态
`blt`	O	检测小于状态

StoreType

信号名	方向	信号描述
`RegData[31:0]`	I	寄存器数据
`MemData[31:0]`	I	存储器数据
`Byte`	I	`Byte` 使能扩展操作信号
`Half`	I	`Half` 使能扩展操作信号
`Offset`	I	偏移量
`Out[31:0]`	O	输出数据

序号	功能名称	功能描述
1	`sb`	取寄存器数据的 `[7:0]` 位，再将存储器的数据取出根据偏移量拼接
2	`sh`	取寄存器数据的 `[15:0]` 位，再将存储器的数据取出根据偏移量拼接
3	`sw`	取寄存器数据的 `[31:0]` 位，并直接输出

nPC_Sel

信号名	方向	信号描述
`Branch`	I	检测是否是分支指令
`Jump`	I	检测是否是跳转指令
`BranchAddr`	I	分支指令地址
`JumpAddr`	I	跳转指令地址
`PC + 4`	I	下一个指令地址
`nPC`	O	输出下一个地址

序号	功能名称	功能描述
1	跳转	跳转至指定地址

Instr Decoder

信号名	方向	信号描述
`Instr[31:0]`	I	指令
`opcode[5:0]`	O	`opcode` 指令
`rs[4:0]`	O	`rs` 寄存器地址
`rt[4:0]`	O	`rt` 寄存器地址
`rd[4:0]`	O	`rd` 寄存器地址
`shamt[4:0]`	O	移位数
`funct[5:0]`	O	`funct` 指令
`imm16[15:0]`	O	16 位立即数
`imm26[25:0]`	O	26 位立即数

二、测试方案

典型测试样例

汇编代码

```
    ori $t0, $0, 0
    ori $t1, $0, 1
    subu $t2, $t0, $t1
label:
    addu $t2, $t2, $t1
    beq $t2, $t0, label
    lui $t2, 0xffff
    sw $t2, ($0)
    lw $t1, ($0)
    beq $t1, $t2, jump
    addu $t1, $t1, $t2
jump:
    nop
```

机器码

```
v2.0 raw
34080000
34090001
01095023
01495021
1148fffe
3c0affff
ac0a0000
8c090000
112a0001
012a4821
00000000
```

期望结果

```
$8  <= 0x00000000
$9  <= 0x00000001
$10 <= 0xffffffff
$10 <= 0x00000000
$10 <= 0x00000001
$10 <= 0xffff0000
*00000000 <= 0xffff0000
$9  <= 0xffff0000
```

汇编代码

```
ori $a0, $0, 0x1000
ori $t0, $0, 0x3018
sw $t0, ($a0)
lb $t1, 1($a0)
lh $t2, 2($a0)
lw $t3, ($a0)
ori $t0, $0, 0xffff
sw $t0, 4($a0)
lb $t1, 5($a0)
lh $t2, 4($a0)
lw $t3, 4($a0)
lui $t0, 0x1234
sw $t0, -4($a0)
lb $t1, -1($a0)
lh $t2, -2($a0)
lw $t3, -4($a0)
```

机器码

```
v2.0 raw
34041000
34083018
ac880000
80890001
848a0002
8c8b0000
3408ffff
ac880004
80890005
848a0004
8c8b0004
3c081234
ac88fffc
8089ffff
848afffe
8c8bfffc
```

期望结果

```
$4 <= 00001000
$8 <= 00003018
*00001000 <= 00003018
$9 <= 00000030
$10 <= 00000000
$11 <= 00003018
$8 <= 0000ffff
*00001004 <= 0000ffff
$9 <= ffffffff
$10 <= ffffffff
$11 <= 0000ffff
$8 <= 12340000
*00000ffc <= 12340000
$9 <= 00000012
$10 <= 00001234
$11 <= 12340000
```

汇编代码

```
ori $a0, $0, 0x0000
ori $t0, $0, 0x3018
sw $t0, ($a0)
ori $t0, $0, 0x1234
sb $t0, 2($a0)
sh $t0, 6($a0)
```

机器码

```
34040000
34083018
ac880000
34081234
a0880002
a4880006
```

期望结果

```
$4 <= 00000000
$8 <= 00003018
*00000000 <= 00003018
$8 <= 00001234
*00000000 <= 00343018
*00000004 <= 12340000
```

思考题

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

   IM 只需读出指令，利用只读存储器 ROM 来储存指令是合理的，而且 ROM 基本上就是组合逻辑的操作，所以可以快速读取指令。

   DM 需写入数据，并且空间较大，所以使用 RAM 是合理的，利用 $MemWrite$ 和时钟信号来控制输入可以确保数据的传输方向。

   GRF 需频繁操作，速度必须要快，所以使用 $Register$ 是合理的，GRF 处理的数据一般是暂时的，储存空间也比较小，所以经常需要更新。

2. 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。

   因为 `nop` 不会对 CPU 进行任何操作，不加入控制信号也不会对 CPU 有多大的影响。

3. 上文提到，**MARS**不能导出**PC**与**DM**起始地址均为**0**的机器码。实际上，可以通过为**DM**增添片选信号，来避免手工修改的麻烦，请查阅相关资料进行了解，并阐释为了解决这个问题，你最终采用的方法。

   将数据地址减去偏移量即可得到正确的地址。

4. 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，简要阐述相比于测试，形式验证的优劣之处。

   形式验证会比测试更加严谨，测试验证可能会少考虑一些数据，而形式验证会通过数理逻辑的抽象方式推出各种测试结果，归纳了所有测试数据，所以形势验证推出的结果会比测试验证来得可靠。

   形式验证的验证方式抽象且复杂，对于小型处理器可以使用测试验证枚举所有数据来测试会相对容易得多。