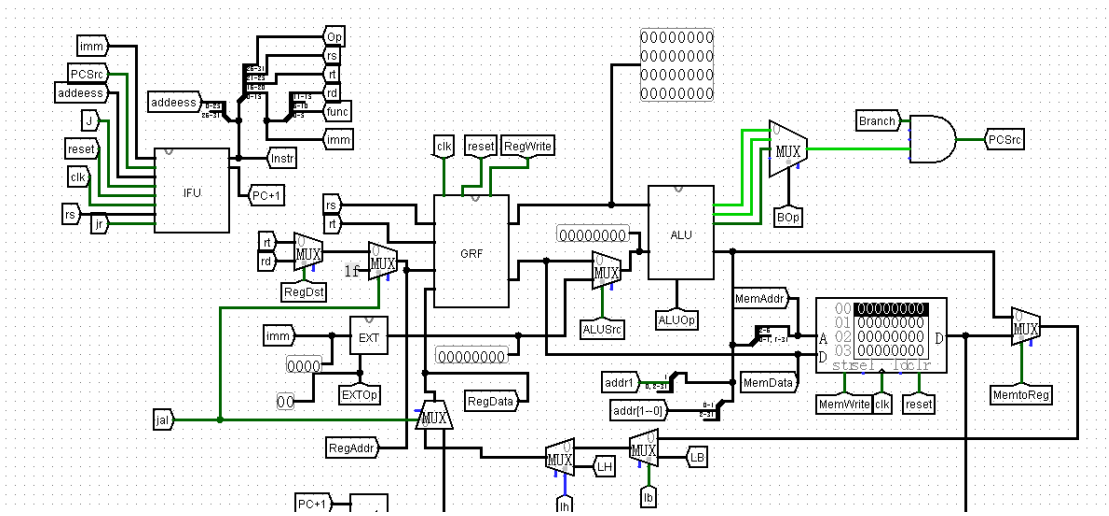
P4 实验报告

一、 数据通路

P4 设计的数据通路与 P3logisim 的电路图基本一致。



二、 模块定义

1. IFU 模块定义:

(1)基本描述

IFU 主要功能是完成取指令功能。IFU 内部包括 PC(程序计数器)、IM(指令存储器)以及其他相关逻辑。IFU 除了能执行顺序取指令外，还能根据 BEQ 指令的执行情况决定顺序取指令还是转移取指令。

(2)模块接口

表 1 模块接口

信号名	方向	描述
PCBranch	I	输入 PC 要跳转几条指令
Branch	I	输入确定 PC 是否要跳转至 PC+4+PCBranch
Reset	I	是否清零
Instr	O	输出指令二进制编码

（3）功能定义

表 2 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 被设置为 0x00003000
2	取指令	根据 PC 从 IM 中取出指令。
3	计算下一条指令地址	如果当前 PCbranch 有效，那么 PC←PC+4+Branch 如果当前 PCBranch 为 0，那么 PC←PC+4

2.GRF 模块定义:

(1)基本描述

GRF 主要功能是完成寄存器的写入和读出操作。GRF 内部包括 32 个寄存器以及其他相关逻辑。其中 0 号寄存器的值始终保持为 0。GRF 可以根据输入来判断写入哪个寄存器或者读出哪个寄存器。

(2)模块接口

表 3 模块接口

信号名	方向	描述
reset	I	复位信号，将 32 个寄存器中的值全部清零
RegWrite	I	写使能信号
rs	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其中存储的数据读出到 RD1
rt	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其中存储的数据读出到 RD2
rt/rd	I	5 位地址输入信号，指定 32 个寄存器中的一个作为写入的目标寄存器
WData	I	32 位数据输入信号

RD1	O	输出 A1 指定的寄存器中的 32 位数据
RD2	O	输出 A1 指定的寄存器中的 32 位数据

（3）功能定义

表 4 功能定义

序号	功能名称	功能描述
1	复位	reset 信号有效时，所有寄存器存储的数值清零
2	读数据	读出 rs，rt 地址对应寄存器中所存储的数据到 RD1，RD2
3	写数据	当 RegWrite 有效且时钟上升沿来临时，将 WData 写入 WAddr 所对应的寄存器中

3. ALU 模块定义:

(1)基本描述

ALU 是算数逻辑单元，读入两个 32 位的数，之后根据 ALUop 来判断是什么运算。其中包括加、减、比较大小、与或等运算。

(2)模块接口

表 5 模块接口

信号名	方向	描述
RData1	I	第一个值
RData2	I	第二个值
ALUop	I	判断进行什么运算
ALUOut	O	运算结果

（3）功能定义

表 6 功能定义

序号	功能名称	功能描述

1	与	A&B
2	或	A|B
3	加	A+B
4	减	A-B

4. EXT 模块定义:

(1)基本描述

EXT 是位数拓展操作，它可以把一个 16 位的数拓展至 32 位。

(2)模块接口

表 7 模块接口

信号名	方向	描述
EXTIn	I	一个 16 位的数
EXTOp	I	判断拓展的形式
EXTOut	O	输出拓展后的数

（3）功能定义

表 8 功能定义

序号	功能名称	功能描述
1	无符号拓展	把输入的 16 位数拓展至无符号的 32 位数
2	有符号拓展	把输入的 16 位数拓展至有符号的 32 位数
3	拓展至高 16 位	将原有的 16 位数简单赋值到 32 位，新数的高 16 位，低 16 位用 0 填充

5. DM 模块定义:

(1)基本描述

DM 是数据存储器，它可以存储数据，并在想读出来的时候把他读出来。

(2)模块接口

表 9 模块接口

信号名	方向	描述
MemAddr	I	要写入的地址
MemData	I	要写入的数据
MemWrite	I	是否写入数据
MemOut	O	读出的数据

（3）功能定义

表 10 功能定义

序号	功能名称	功能描述
1	写入数据	确定要写入的数据存储的地址，进而 MemWrite 为 1 时，写入数据；为 0 时不写入
2	读出数据	根据存储器相应地址输出数据

6.Controller 模块定义：

(1)基本描述

Controller 是控制器，我们读入指令的高 6 位和 function 位来判断是什么指令，进而，我们可以确定各个板块的使能端控制端的值。

(2)模块接口

表 9 模块接口

信号名	方向	描述
op[5:0]	I	高 6 位
func[5:0]	I	Function
RegWrite	O	要不要写入寄存器
RegDst	O	写寄存器的目标寄存器号来源
ALUSrc	O	选择是运算扩展后的立即数还是寄存器内的值
Branch	O	与 ALU 零输出结合决定 PC 是否跳转

MemWrite	0	要不要写入存储器
MemtoReg	0	写入寄存器的数据来源：ALU 计算后的或者 DM 输出的
EXTOp	0	判断进行拓展的方式
ALUOp	0	判断 ALU 要进行什么运算
j	0	判断是否为 j 指令
jal	0	判断是否为 jal 指令
jr	0	判断是否为 jr 指令

（3）功能定义

表 10 功能定义

序号	功能名称	功能描述
1	判断是什么指令	我们根据传进来的两组数据，根据 MIPS 对应规则，可以知道他是什么指令。
2	判断是否要写入寄存器	如果 RegWrite 为 1 的话，就可以写入寄存器；否则不写入
3	判断要写入哪个寄存器	如果 RegDst 为 1 的话，就写入 11-15 位的寄存器，否则就写入 16-20 位的寄存器
4	判断要运算立即数还是寄存器的值	如果 ALUSrc 为 1 的话，就会选择立即数进行运算，否则，选择读出的寄存器的值进行运算
5	判断要不要跳转	如果当前指令需要跳转，比如 beq，那么 Branch 就为 1，否则为 0
6	判断要不要写入寄存器	如果 MemWrite 为 1 的话，就

		写入存储器，否则，不写入
7	判断写入的数据是哪个值	如果要写入寄存器的值是 ALU 算出来的时候，那么 MemtoReg 就为 0，否则为 1
8	判断要进行拓展的方式	比如如果 ExtOp 为 1，就进行有符号拓展
9	判断要进行那个运算	生成 ALUOp 来确定要让 ALU 进行那种运算。

三、控制器设置

1. 真值表:

func	100001	100011							000011
op	000000	000000	001101	100011	101011	000100	001111	000011	000000
	addu	subu	ori	lw	sw	beq	lui	jal	jr
EXTOp	xx	xx	00	01	01	01	10	00	00
MemtoReg	0	0	0	1	0	0	0	0	0
MemWrite	0	0	0	0	1	0	0	0	0
Branch	0	0	0	0	0	1	0	0	0
ALUOp	010	011	001	010	010	xxx	100	000	000
ALUSrc	0	0	1	1	1	0	1	0	0
RegDst	1	1	1	1	0	0	1	0	0
RegWrite	1	1	1	1	0	0	1	1	0
j	0	0	0	0	0	0	0	1	0

jal	0	0	0	0	0	0	0	1	0
jr	0	0	0	0	0	0	0	0	1

2、表达式:

EXTOp[0] = lw||sw||beq EXTOp[1] = lui

EXTOp[0] =

op[0]&op[1]&!op[2]&!op[3]&!op[4]&op[5]+

op[0]&op[1]&!op[2]&

op[3]&!op[4]&op[5]+ !op[0]&!op[1]&op[2]&!op[3]&!o

p[4]&!op[5]

EXTOp[1] = op[0]&op[1]&op[2]&op[3]&!op[4]&!op[5]

MemtoReg = lw

MemtoReg =

op[0]&op[1]&!op[2]&!op[3]&!op[4]&!op[5]

Memwrite = sw

MemWrite = op[0]&op[1]&!op[2]&op[3]&!op[4]&op[5]

Branch = beq

Branch = !op[0]&!op[1]&op[2]&!op[3]&!op[4]&!op[5]

ALUOp[0] = subu||ori ALUOp[1] =

addu||subu||lw||sw ALUOp[2] = lui

ALUOp[0]

= !op[0]&!op[1]&!op[2]&!op[3]&!op[4]&!op[5]&fun[0

]&fun[1]&!fun[2]&!fun[3]&!fun[4]&fun[5]+ !op[0]&!

op[1]&op[2]&!op[3]&!op[4]&!op[5]

ALUOp[1]

= !op[0]&!op[1]&!op[2]&!op[3]&!op[4]&!op[5]

+op[0]&op[1]&!op[2]&!op[3]&!op[4]&!op[5]+

op[0]&op[1]&!op[2]&op[3]&!op[4]&op[5]

ALUOp[2] = op[0]&op[1]&op[2]&op[3]&!op[4]&!op[5]

ALUSrc = ori||lw||sw||lui

```
 ALUSrc = op[0]&!op[1]&op[2]&op[3]&!op[4]&!op[5]+
op[0]&op[1]&!op[2]&!op[3]&!op[4]&op[5]+
op[0]&op[1]&!op[2]&op[3]&!op[4]&op[5]+
op[0]&op[1]&op[2]&op[3]&!op[4]&!op[5]
 RegDst = addu||subu
 RegDst
= !op[0]&!op[1]&!op[2]&!op[3]&!op[4]&!op[5]
 RegWrite = addu||subu||ori||lw||lui
 RegWrite
=       !op[0]&!op[1]&!op[2]&!op[3]&!op[4]&!op[5]+
op[0]&!op[1]&op[2]&op[3]&!op[4]&!op[5]+
op[0]&op[1]&!op[2]&!op[3]&!op[4]&!op[5]+
op[0]&op[1]&op[2]&op[3]&!op[4]&!op[5]
```

3、具体实现方式

```
output lh_Enable,
output lbu_Enable
);
reg [18:0] control = 0;
assign{EXTOp,MemtoReg,MemWrite,Branch,ALUOp,ALUSrc,RegDst,RegWrite,j,jal,jr,BOp,lb_Enable,lh_Enable,lbu_Enable}=c
always @(*)
begin
   case(Op)
     6'b000000:
     begin
        case(func)
           6'b000000:  control<=19'b 0000000000000000000;//nop
           6'b100001:  control<=19'b 0000001001100000000;//addu
           6'b100011:  control<=19'b 0000001101100000000;//subu
           6'b001000:  control<=19'b 0000000000000100000;//jr
           default: control<=19'b 0000000000000000000;
        endcase
     end
     6'b001101:  control<=19'b 0000000110100000000;//ori
     6'b100011:  control<=19'b 0110001010100000000;//lw
     6'b101011:  control<=19'b 0101001010000000000;//sw
     6'b000100:  control<=19'b 0100100000000000000;//beq
     6'b001111:  control<=19'b 1000010010100000000;//lui
     6'b000011:  control<=19'b 0000000000111000000;//jal
     6'b000010:  control<=19'b 0000000000010000000;//j
```

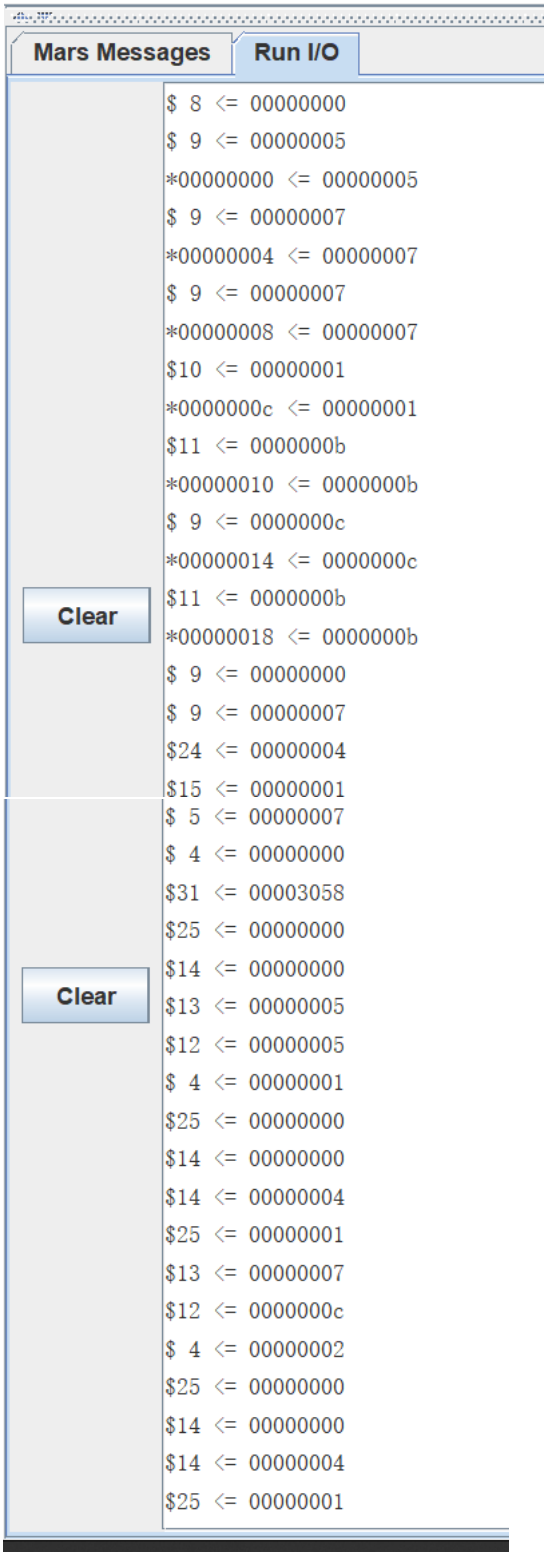
四、测试程序

```
ori $t0 $t0 0          #t0<=0
ori $t1 $t1 5       #t1<=5
sw $t1 0($t0)          #存入 t1
```

```
ori $t1 $t1 2       #t1<=2
sw $t1 4($t0)           #存入 t1
ori $t1 7                   #t1 赋值为 7
sw $t1 8($t0)           #存入 7
ori $t2 1                   #t2 赋值 1
sw $t2 12($t0)          #存入 1
ori $t3 11              #t3 赋值 11
sw $t3 16($t0)          #存入 11
addu $t1 $t2 $t3    #t1<=12
sw $t1 20($t0)          #存入 12
sub $t3 $t1 $t2         #t3 赋值为 11
sw $t3 24($t0)          #存入 11
ori $t1 $0 0            #t1 赋值为 0
ori $t1 $t1 7           #t2 赋值为 7
ori $t8 $t8 4           #t8 赋值为 4
ori $t7 $t7 1           #t7 赋值为 1
ori $a1 $t1 0           #a1=7
ori $a0 $0 0            #a0=0
jal sum                     #跳入 sum
subu $t4 $t4 $t7        #t4-1
j end                           #跳到结束
sum:
  for_min:
     beq $a0 $a1 min_end
```

```
        ori $t9 $0 0        #t9 循环
        ori $t6 $0 0      #t6 负责+4
        for:
        beq $t9 $a0 for_end
           addu $t6 $t6 $t8        #t6+4
           addu $t9 $t9 $t7        #t9+1
           j for
        for_end:
        lw $t5 0($t6)                  #取出值
        addu $t4 $t4 $t5           #加，$4 表示结果
        addu $a0 $a0 $t7           #for 循环用
        j for_min
    min_end:
        jr $ra                              #跳回
end:
lui $1,1
```

期望运行结果：


Mars Messages
Run I/O
$ 8 <= 00000000
$ 9 <= 00000005
*00000000 <= 00000005
$ 9 <= 00000007
*00000004 <= 00000007
$ 9 <= 00000007
*00000008 <= 00000007
$10 <= 00000001
*0000000c <= 00000001
$11 <= 0000000b
*00000010 <= 0000000b
$ 9 <= 0000000c
*00000014 <= 0000000c
$11 <= 0000000b
*00000018 <= 0000000b
$ 9 <= 00000000
$ 9 <= 00000007
$24 <= 00000004
$15 <= 00000001
Clear
$ 5 <= 00000007
$ 4 <= 00000000
$31 <= 00003058
$25 <= 00000000
$14 <= 00000000
$13 <= 00000005
$12 <= 00000005
$ 4 <= 00000001
$25 <= 00000000
$14 <= 00000000
$14 <= 00000004
$25 <= 00000001
$13 <= 00000007
$12 <= 0000000c
$ 4 <= 00000002
$25 <= 00000000
$14 <= 00000000
$14 <= 00000004
$25 <= 00000001
Clear

CPU 运行结果：

```
@00003000: $ 8 <= 00000000
@00003004: $ 9 <= 00000005
@00003008: *00000000 <= 00000005
@0000300c: $ 9 <= 00000007
@00003010: *00000004 <= 00000007
@00003014: $ 9 <= 00000007
@00003018: *00000008 <= 00000007
@0000301c: $10 <= 00000001
@00003020: *0000000c <= 00000001
@00003024: $11 <= 0000000b
@00003028: *00000010 <= 0000000b
@0000302c: $ 9 <= 0000000c
@00003030: *00000014 <= 0000000c
@00003038: *00000018 <= 0000000b
@0000303c: $ 9 <= 00000000
@00003040: $ 9 <= 00000007
@00003044: $24 <= 00000004
@00003048: $15 <= 00000001
@0000304c: $ 5 <= 00000007
@00003050: $ 4 <= 00000000
@00003054: $31 <= 00003058
@00003064: $25 <= 00000000
@00003068: $14 <= 00000000
@0000307c: $13 <= 00000005
@00003080: $12 <= 00000005
@00003084: $ 4 <= 00000001
@00003064: $25 <= 00000000
@00003068: $14 <= 00000000
@00003070: $14 <= 00000004
```

```
@0000307c: $13 <= 00000005
@00003080: $12 <= 00000005
@00003084: $ 4 <= 00000001
@00003064: $25 <= 00000000
@00003068: $14 <= 00000000
@00003070: $14 <= 00000004
@00003074: $25 <= 00000001
@0000307c: $13 <= 00000007
@00003080: $12 <= 0000000c
@00003084: $ 4 <= 00000002
@00003064: $25 <= 00000000
@00003068: $14 <= 00000000
@00003070: $14 <= 00000004
@00003074: $25 <= 00000001
ISim>
```

上面显示验证为输出直接与 Mars 输出对比，过程中 testbench 中的值也进行过对比。

所以运行结果与 MARS 相同，所以对这 10 条指令没有问题。