计算机组成课程设计 P4 实验报告

一、 数据通路

1、IFU（取指令单元）

（1）IM： 包括 1024 个 32 位寄存器，初始值为 0，从 code.txt 读取指令存入 IM。

（2）PC： 寄存器，初始地址为 0x00003000。时钟上升沿到来时，若为 beq 且 zero 为 1，则 PC=PC+4+sign_ext(imm[15:0])*4;

若为 jal 指令，则 PC=PC[31:28]||imm[25:0]||00;

若为 jr 指令，则 PC=rs；

否则，PC=PC+4.

（3）输出 PC4：PC+4

Instr：当前指令 32 位机器码，存在 im 中，根据 PC[11:2]，取出 im[PC[11:2]]赋给 instr。（因为 im 为 1024 个存储器，且 PC 每加 4，im 加 1，所以使用 PC[11:2]）

端口定义：

信号名	方向	描述
npcsel[1:0]	I	控制信号，当前指令位 beq 时，npcsel=01； 为 jal 时，npcsel=10; 为 jr 时，npcssel=11； 其余情况下，npcsel=00.

zero	I	当 ALU 两个输入信号相等时，zero 为 1；当不相等时，zero 为 0。
clk	I	时钟信号
reset	I	复位信号
imm[25:0]	I	指令的第 25 至第 0 位，特定情况下代表立即数。
rsvalue[31:0]	I	rs 寄存器的值，用于 jr 指令
instr[31:0]	O	根据 PC 的值从指令寄存器 im 中取出的 32 位指令操作数
PC4[31:0]	O	输出 PC+4

功能定义：

序号	功能名称	功能描述
1	复位	当时钟上升沿到来时，如果复位信号有效，PC 被复位为 0x00003000
2	计算下一个 PC 值	当 npcsel 为 01 且 zero 1 时，PC=PC+4+sign_ext(imm[15:0])*4; 当 npcsel 为 10 时，PC=PC[31:28]||imm[25:0]||00; 当 npcsel 为 11 时，PC=GPR[rs]; 否则，PC=PC+4;
3	取指令	根据当前 PC 值，从指令寄存器中取出 32 位操作数
4	计算 PC4	将 PC+4 赋给 PC4.
5	初始化	im 存储器被初始化为 0;code.txt 存入 im 中;PC 初始化为 0x00003000

2.GRF（寄存器堆）

（1）内部包含 32 个 32 位寄存器，初始值为 0；如果写使能信号有效，则将 busW 存入 rtd 寄存器；0 号寄存器永远为 0.

（2）输出 busA 为 rs 寄存器的值，busB 为 rt 寄存器的值

（3）当写使能信号有效时，将此时写入信息，以@%h: $%d <= %h 格式显示出来。例如：@00003000: $ 5 <= 03030000

端口定义：

信号名	方向	描述
rs[4:0]	I	指令的第 21 到第 25 位
rt[4:0]	I	指令的第 16 到第 20 位
rtd[4:0]	I	当 RegDst 信号为 00 时，指令的第 16 到第 20 位； 当 RegDst 信号为 01 时，指令的第 11 到第 5 位； 当 RegDst 信号为 10 时，为常数 31
RegWrite	I	写使能信号，当为 1 时，将 busW 的值写入 rtd
busW[31:0]	I	回写寄存器值，当 RegWrite 为 1 时，busW 的值写入 rtd
clk	I	时钟信号
reset	I	复位信号
PC4[31:0]	I	PC+4 的值
busA[31:0]	O	从指令中读取的第一个操作数
busB[31:0]	O	从指令中读取的第二个操作数

功能定义：

序号	功能名称	功能描述
1	读取操作数	根据当前 rs、rt 值，读取第一个和第二个操作数，存入 busA 和 busB 中
2	回写	当 RegWrite 信号为 1 且 rtd 不为 0 时，在时钟上升沿，将 busW 的值写入 rtd 寄存器中，并以@%h: $%d <= %h 格式显示
3	复位	当时钟上升沿到来时，如果复位信号有效，则将寄存器全部清零

3.ALU（算术逻辑单元）

（1）根据输入的两个 32 位操作数进行加、减、或运算，

（2）当输入的两个操作数相等时，输出 zero 为 1

端口定义：

信号名	方向	描述
ALUA[31:0]	I	第一个操作数
ALUB[31:0]	I	第二个操作数
ALUctr[1:0]	I	当 ALUctr 等于 00 时，ALU 执行加法运算； 当 ALUctr 等于 01 时，ALU 执行减法运算； 当 ALUctr 等于 10 时，ALU 执行或运算。
ALUout[31:0] (reg)	O	将两个操作数进行运算后的结果
zero (reg)	O	当输入的两个操作数相等时，输出 1； 当输入的两个操作数不相等时，输出 0。

功能定义：

序号	功能名称	功能描述
1	将输入的两个 32 位操作数进行算数逻辑运算	根据 ALUctr 的值，将 ALUA 和 ALUB 两个数进行加法、减法、或运算
2	比较两个数是否相等	当 ALUA 和 ALUB 相等时，zero 信号为 1；否则为 0。

4.DM（数据寄存器）

（1）内部包含 32bit*1024 字大小的 dm 存储器，初始化为 0。当写使能信号有效时，将输入的数据写入对应的地址，并以@%h: *%h <= %h 格式显示，例如：**@00003004: *00001004 <= 00000000**（因为 dm 有 1024 个单元且指令地址加 4dm 加 1，故将 addr[11:2]作为写 dm 存储器的地址）

（2）由根据输入的地址值取出相应的内存中的值并输出。

端口定义：

信号名	方向	描述
addr[31:0]	I	ALU 单元计算的结果，作为读写内存的地址
data[31:0]	I	当写使能信号有效时，将其写入 dm[addr[11:2]]中，并以@%h: *%h <= %h 格式显示
MemWrite	I	写使能信号，当为 1 时，表示写使能有效
clk	I	时钟信号
reset	I	复位信号
PC4[31:0]	I	当前的 PC+4
dataout[31:0]	O	从内存中读取的数据

功能定义：

序号	功能名称	功能描述
1	写入内存数据	当写使能信号有效时，将 data[31:0]写入 dm[addr[11:2]]中，并以@%h: *%h <= %h 格式显示
2	读取内存数据	将 addr[11:2]所代表地址的数据输出
3	复位功能	当时钟上升沿到来时，如果 reset 信号有效，将 dm 清零

5.EXT（扩展单元）

根据选择信号 EXTop，将输入的立即数进行扩展并输出

端口定义：

信号名	方向	描述
imm[15:0]	I	输入的立即数
EXTop[1:0]	I	当 EXTop 为 00 时，将立即数进行无符号扩展； 当 EXTop 为 01 时，将立即数进行有符号扩展； 当 EXTop 为 10 时，将立即数进行高位扩展；
extout[31:0] (reg)	O	扩展后的立即数输出

功能定义：

序号	功能名称	功能描述
1	无符号扩展	当 EXTop 为 00 时，将立即数进行无符号扩展并输出
2	有符号扩展	当 EXTop 为 01 时，将立即数进行有符号扩展并输出
3	高位扩展	当 EXTop 为 10 时，将立即数进行高位扩展并输出

6.MUX（多路选择器）

Mux 中包含三个多路选择器,用来筛选进入 GRF 的 rtd 端口的信号（mux_GRF）、进入 GRF 的 busW 端口的信号（mux_WD）、进入 ALU 的 ALUB 端口的信号（mux_ALU）

mux_ALU 端口定义：

信号名	方向	描述
busB[31:0]	I	GRF 单元的 busB 端口，rt 寄存器中的值
immext[31:0]	I	ext 单元的 extout 端口，扩展之后的立即数
ALUSrc	I	选择信号
ALUB[31:0] (reg)	O	ALU 的 ALUB 端口，第二个操作数

mux_ALU 功能定义：

序号	功能名称	功能描述
1	选择进入 ALUB 的信号	当 ALUSrc 为 0 时，将 busB 输入到 ALUB 接口； 当 ALUSrc 为 1 时，将 immext 输入到 ALUB 接口；

mux_GRF 端口定义：

信号名	方向	描述
rt[4:0]	I	IFU 的 rt 接口
rd[4:0]	I	IFU 的 rd 接口
RegDst[1:0]	I	选择信号
rtd[31:0] (reg)	O	GRF 的 rtd 接口

mux_GRF 功能定义：

序号	功能名称	功能描述
1	选择进入 rtd 的信号	当 RegDst 为 00 时，将 rt 输入到 rtd 接口； 当 RegDst 为 01 时，将 rd 输入到 rtd 接口； 当 RegDst 为 10 时，将常数 31 输入到 rtd 接口；

mux_WD 端口定义：

信号名	方向	描述
ALUout[31:0]	I	ALU 单元的 ALUout 端口
dataout[31:0]	I	dm 单元的 dataout 端口
PC4[31:0]	I	IFU 单元的 PC4 接口
MemtoReg[1:0]	I	选择信号
busW[31:0] (reg)	O	GRF 的 busW 接口

mux_WD 功能定义：

序号	功能名称	功能描述
1	选择进入 busW 的信号	当 MemtoReg 为 00 时，将 ALUout 输入到 busW 接口； 当 MemtoReg 为 01 时，将 dataout 输入到 busW 接口； 当 MemtoReg 为 10 时，将 PC4 输入到 busW 接口；

二、 Controller(控制器)

1.真值表：

func	100001	100011							001000	000000
Op	000000	000000	001101	100011	101011	000100	001111	000011	000000	000000
	addu	subu	ori	lw	sw	beq	lui	jal	jr	nop
RegDst[1:0]	01	01	00	00	x	x	00	10	x	x
ALUSrc	0	0	1	1	1	0	1	x	x	x
MemtoReg[1:0]	00	00	00	01	x	x	00	10	x	x
RegWrite	1	1	1	1	0	0	1	1	0	0
MemWrite	0	0	0	0	1	0	0	0	0	0
npcsel[1:0]	00	00	00	00	00	01	0	10	11	00
EXTop[1:0]	x	x	00	01	01	x	10	x	x	x
ALUctr[1:0]	00(add)	01(sub)	10(ori)	00	00	01	00	x	x	x

2. 信号功能：

信号名	功能描述
RegDst[1:0]	00：将 rt 寄存器的值输入到 GRF 的 rtd 端口 01：将 rd 寄存器的值输入到 GRF 的 rtd 端口 10:常数 31 输入到 GRF 的 rtd 端口
ALUSrc	0：将 GRF 的 busB 输入到 ALU 的 ALUB 接口 1：将 EXT 的 extout 接到 ALU 的 ALUB 接口
MemtoReg[1:0]	00：将 ALU 的 ALUout 接到 GRF 的 busW 接口 01：将 DM 的 dataout 接到 GRF 的 busW 接口 10：将 IFU 的 PC4 输出接到 GRF 的 busW 接口
RegWrite	0：不对 GRF 进行写操作 1：将 GRF 的 busW 写入 rtd 中
MemWrite	0:不对 dm 存储器进行写操作 1：将 data 写入 dm 的 addr 地址中
npcsel[1:0]	00：该指令不是 beq、jal、jr 01：该指令是 beq 10：该指令时 jal 11：该指令是 jr
EXTop[1:0]	00：将立即数进行无符号扩展 01：将立即数进行有符号扩展 10：将立即数进行高位扩展
ALUctr[1:0]	00：进行加法操作 01：进行减法操作 10：进行或操作

三、 测试程序

```
ori $a0,$0,0x100

ori $a1,$a0,0x123

lui $a2,456

lui $a3,0xffff

ori $a3,$a3,0xffff


addu $s0,$a0,$a2

addu $s1,$a0,$a3

addu $s4,$a3,$a3

subu $s2,$a0,$a2

subu $s3,$a0,$a3


sw $a0,0($0)

sw $a1,4($0)

sw $a2,8($0)

sw $a3,12($0)

sw $s0,16($0)

sw $s1,20($0)

sw $s2,24($0)

sw $s3,44($0)

sw $s4,48($0)
```

```
lw $a0,0($0)
lw $a1,12($0)
sw $a0,28($0)
sw $a1,32($0)

ori $a0,$0,1
ori $a1,$0,2
ori $a2,$0,1
beq $a0,$a1,loop1
beq $a0,$a2,loop2
loop1: sw $a0,36($t0)
loop2: sw $a1,40($t0)

jal loop3
sw $s5,64($t0)
ori $a1,$a1,4
jal loop4
loop3:sw $a1,56($t0)
sw $ra,60($t0)
ori $s5,$s5,5
jr $ra
loop4: sw $a1,68($t0)
sw $ra,72($t0)
```

运行结果：

```
@00003000: $ 4 <= 00000100
@00003004: $ 5 <= 00000123
@00003008: $ 6 <= 01c80000
@0000300c: $ 7 <= ffff0000
@00003010: $ 7 <= ffffffff
@00003014: $16 <= 01c80100
@00003018: $17 <= 000000ff
@0000301c: $20 <= fffffffe
@00003020: $18 <= fe380100
@00003024: $19 <= 00000101
@00003028: *00000000 <= 00000100
@0000302c: *00000004 <= 00000123
@00003030: *00000008 <= 01c80000
@00003034: *0000000c <= ffffffff
@00003038: *00000010 <= 01c80100
@0000303c: *00000014 <= 000000ff
@00003040: *00000018 <= fe380100
@00003044: *0000002c <= 00000101
@00003048: *00000030 <= fffffffe
@0000304c: $ 4 <= 00000100
@00003050: $ 5 <= ffffffff
@00003054: *0000001c <= 00000100
@00003058: *00000020 <= ffffffff
@0000305c: $ 4 <= 00000001
@00003060: $ 5 <= 00000002
@00003064: $ 6 <= 00000001
@00003074: *00000028 <= 00000002
@00003078: $31 <= 0000307c
@00003088: *00000038 <= 00000002
@0000308c: *0000003c <= 0000307c
@00003090: $21 <= 00000005
@0000307c: *00000040 <= 00000005
@00003080: $ 5 <= 00000006
@00003084: $31 <= 00003088
@00003098: *00000044 <= 00000006
@0000309c: *00000048 <= 00003088
```

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000001
$a1	5	0x00000006
$a2	6	0x00000001
$a3	7	0xffffffff
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x01c80100
$s1	17	0x000000ff
$s2	18	0xfe380100
$s3	19	0x00000101
$s4	20	0xfffffffe
$s5	21	0x00000005
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00001800
$sp	29	0x00002ffc
$fp	30	0x00000000
$ra	31	0x00003088

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000100	0x00000123	0x01c80000	0xffffffff	0x01c80100	0x000000ff	0xfe380100	0x00000100
0x00000020	0xffffffff	0x00000000	0x00000002	0x00000101	0xfffffffe	0x00000000	0x00000002	0x0000307c
0x00000040	0x00000005	0x00000006	0x00003088	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

四、 思考题

1. 根据你的理解，在下面给出的 DM 的输入示例中，地址信号 addr 位数为什么是[11:2]而不是[9:0]？这个 addr 信号又是从哪里来的？

文件	模块接口定义

dm.v	dm(clk,reset,MemWrite,addr,din,dout);
	input clk; //clock
	input reset; //reset
	input MemWrite; //memory write enable
	input [11:2] addr; //memory's address for write
	input [31:0] din; //write data
	output [31:0] dout; //read data

答：addr 是 ALU 单元的输出 ALUout 端口接过来的，代表的是要读取的 dm 存储器的地址，例如当指令为 lw 时（lw $1,4($2)），addr 中存的是$2 和 4 相加得出的目的地址，该地址是以 4 为偏移单位的，而 dm 中的偏移是以 1 为单位的，故 addr 每加 4，dm 加 1，所以，选取的是 addr[11:2].

2. 在相应的部件中，**reset 的优先级**比其他控制信号（不包括 clk 信号）都要**高**，且相应的设计都是**同步复位**。清零信号 reset 是针对哪些部件进行清零复位操作？这些部件为什么需要清零？

答：针对 PC、GRF、DM 进行的清零复位操作。

复位，代表重新开始运行程序，所以指令的地址应该从头开始，所以，应该将 PC 复位成 0x00003000。

重新运行，要将之前运行指令对内存存储器的改变清除，所以应该将 dm 清零，以避免对接下来的新指令带来影响。

同时，应该将之前指令对 32 个寄存器的影响清除，所以，也应该将 32 个寄存器清零。

3. 列举出用 Verilog 语言设计控制器的几种编码方式（至少三种），并给出代码示例。

第一种（if-else）

```
always@*begin
        if(op==6'b000000&&func==6'b100001)begin                          //addu
           RegDst=2'b01;  RegWrite=1'b1;
           end
        else if(op==6'b000000&&func==6'b100011)begin                     //subu=1;
           RegDst=2'b01;  RegWrite=1'b1;
           end
        else if(op==6'b001101)begin                                      //ori=1;
           ALUSrc=1'b1; RegWrite=1'b1;
           end
        else if(op==6'b101011)begin                                      //lw=1;
           ALUSrc=1'b1; MemtoReg=2'b01;RegWrite=1'b1;EXTop=2'b01;
           end
        else if(op==6'b101011)begin                                      //sw=1;
           ALUSrc=1'b1;MemWrite=1'b1;EXTop=2'b01;
           end
        else if(op==6'b000100)begin                                      // beq=1;
           npcsel=2'b01;
           end
        else if(op==6'b001111)begin                                     //lui=1;
           ALUSrc=1'b1;RegWrite=1'b1;EXTop=2'b10;
           end
        else if(op==6'b000011)begin                                      //jal=1;
           RegDst=2'b10;MemtoReg=2'b10;RegWrite=1'b1;npcsel=2'b10;
           end
        else if(op==6'b000000&&func==6'b001000)begin                    //jr=1;
           npcsel=2'b11;
           end
```

第二种（assign）：

```
assign
{RegDst,ALUSrc,MemtoReg,RegWrite,npcsel,EXTop,ALUctr,MemWrite}=
(op==6'b001101)?13'b0000010000100: //ori
(op==6'b100011)?13'b0010110001000://lw
(op==6'b101011)?13'b0010000001001://sw
(op==6'b000100)?13'b0000000100010://beq
(op==6'b001111)?13'b0010010010000://lui
(op==6'b000011)?13'b1001011000000://jal
(op==6'b000000)?((func==6'b100001)?13'b0100010000000://addu
                 (func==6'b100011)?13'b0100010000010://subu
                 (func==6'b001000)?13'b0000001100000://jr
                 13'b0):13'b0;
```

第三种（宏定义）：

```
parameter      addu=12'b000000100001,subu=12'b000000100011,ori=6'b001101;
parameter      lw=6'b100011,sw=6'b101011,beq=6'b000100,nop=12'b000000000000;
parameter      lui=6'b001111,jal=6'b000011,jr=12'b000000001000;
parameter      rdport=2'b01,PC4port=2'b10,aluport=2'b00,dmport=2'b01,thrityport=2'b10,isbeq=2'b01,isjal=2'b10,isjr=2'b11;
parameter      unsign=2'b00,sign=2'b01,high=2'b10,add=2'b00,sub=2'b01,orr=2'b10;
case({op,func})
 addu:begin
  RegDst=rdport; RegWrite=1;end
 subu:begin
  RegDst=rdport; RegWrite=1;end
 jr:begin
  npcsel=isjr;end
endcase
case(op)
   ori:begin
   ALUSrc=1;RegWrite=1;end
   lw:begin
   ALUSrc=1;MemtoReg=dmport;RegWrite=1;EXTop=sign;end
   sw:begin
   ALUSrc=1;EXTop=sign;MemWrite=1;end
   beq:begin
   npcsel=isbeq;end
   lui:begin
   ALUSrc=1;RegWrite=1;EXTop=high;end
   jal:begin
   RegDst=PC4port;MemtoReg=thrityport;RegWrite=1;npcsel=isjal;end
    endcase
end
```

4. 根据你所列举的编码方式，说明他们的优缺点。

答：if-else 条理更清晰，分界更明显，assign 写法更方便；但是这两种的可读性不如第三种强

5. C 语言是一种弱类型程序设计语言。C 语言中不对计算结果溢出进行处理，这意味着 C 语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持 C 语言，MIPS 指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的 Operation 部分 。

```
temp ← (GPR[rs]31||GPR[rs]) + (GPR[rt]31||GPR[rt])
if temp32 ≠ temp31 then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← temp31..0
GPR[rd] ← GPR[rs] + GPR[rt]
```

addi 与 addiu 的区别在于，当出现溢出时，addiu 忽略溢出，将溢出的最高位舍弃

而 addi 会报告 SignalException(IntegerOverflow)。

如果忽略溢出，则二者等价。

6. 根据自己的设计说明单周期处理器的优缺点。

优点是模块化很强，可以方便的增添指令，各部分各司其职，不存在冲突，正确性高。缺点是，大部分指令并不能用到所有模块，所有指令的执行时间一样，浪费空间，时间。

7. 简要说明 jal、jr 和堆栈的关系。

jal 是跳转到指定地址，并将 PC+4 保存在 31 号寄存器中，在需要多次跳转的函数中，（例如：递归函数）在下一次跳转之前，要将本次的 31 号寄存器的值存入栈中，首先将栈指针下移，然后将 31 号寄存器的值存入栈中。Jr 是函数返回时，将 31 号寄存器中存的 PC 值给当前 PC，跳完之后，要将此时的 31 号寄存器更新，从栈中取出之前保存的 PC 值，赋给 31 号寄存器，栈指针上移。

```
addi $sp,$sp,-12
sw $t0,0($sp)
sw $ra,4($sp)
sw $t1,8($sp)
jal fullarray          #fullaray(index+1);
lw $t0,0($sp)
lw $ra,4($sp)
lw $t1,8($sp)
addi $sp,$sp,12
```