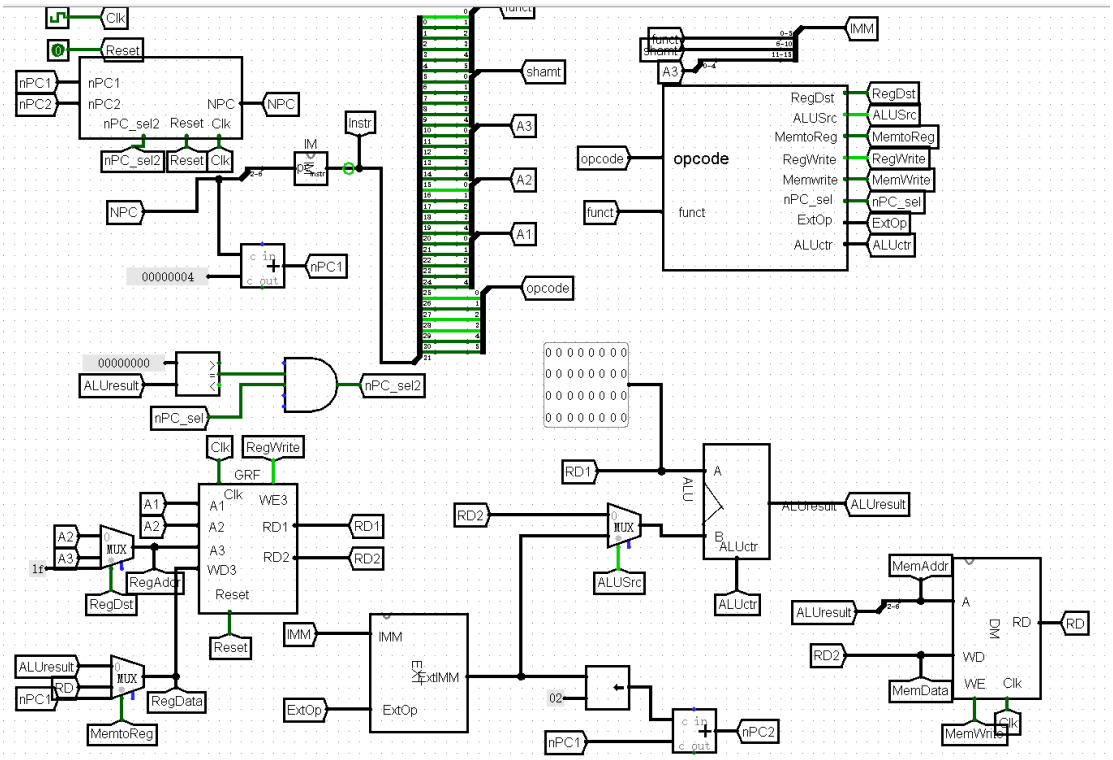
P4 实验设计报告

18373085 张海渝

一. 数据通路



二. 模块规格

1. GetNpc：（程序计数器）

序号	功能名称	方向	功能描述
1	npc1[31:0]	I	为 PC+4
2	npc2[31:0]	I	解释见下
3	npc_sel2[31:0]	I	选择下一 PC 值
4	reset	I	同步复位
5	clk	I	时钟信号
6	nPC[31:0]	O	下一 PC 值

模块解释：

1. 通过选择器进行先选择下一 PC 值。

2. 如果 Reset 信号有效时，用选择器将 PC 值复位。
3. npc2 指的是在 jal,jr,beq 中选择出来的那个 pc 值。

```
module GetNpc(npc1,npc2,npc_sel2,reset,clk,nPC);
   input [31:0]npc1;
   input  [31:0] npc2;
   input npc_sel2;
   input reset;
   input clk;
   output reg [31:0] nPC ;
   initial nPC=32'h00003000;

   always@(posedge clk) begin
      if(reset==1) nPC<=32'h00003000;
      else begin
         if(npc_sel2==1) nPC<=npc2;
         else nPC<=npc1;
      end
   end

endmodule
```

图一：GetNpc 模块代码图

2. IM：（指令获取）

序号	功能名称	方向	功能描述
1	add[9:0]	I	指令地址
2	Instr	O	输出的机器码

模块解释：

1. add[11:2]是当前 PC 值的第 6-2 位(因为 ROM 地址为 5 位，并且 PC 每次加 4，相当于 ROM 加 1)。
2. 用$readmemh 载入机器码。
3. 用寄存器储存机器码。

```
module IM(
   input [9:0] add,
   output [31:0] Instr
    );
   reg [31:0] ROM [1023:0];
   initial  begin
      $readmemh("code.txt",ROM);
   end
   assign Instr=ROM[add];
endmodule
```

图二：IM 模块代码图

3. GRF：（通用寄存器组，也称寄存器文件、寄存器堆）

序号	功能名称	方向	功能描述
1	pc[31]	I	当前 PC 值
2	A1[4:0]	I	机器码的第 25-21 位，寄存器编号
3	A2[4:0]	I	机器码的第 20-16 位，寄存器编号
4	A3[4:0]	I	需要改变的寄存器
5	WD3[31:0]	I	寄存器需要改变为的值
6	reset	I	同步复位信号
7	clk	I	时钟信号
8	WE	I	是否写入寄存器信号
9	RD1[31:0]	O	A1 寄存器中的值
10	RD2[31:0]	O	A2 寄存器中的值

模块解释：

1.0 号寄存器始终为 0，不改变。

2.一共有 32 个带有使能端的寄存器。

3. 每 次 寄 存 器 改 变 时 ， 输

```
    reg [31:0] a [31:0];
    integer i;
    initial begin
        for(i=0;i<=31;i=i+1) begin
            a[i]=0;
        end
    end

    assign RD1=a[A1];
    assign RD2=a[A2];

    always@(posedge clk) begin
        if(reset==1) begin
            for(i=0;i<=31;i=i+1) a[i]<=0;
        end
        else begin
            if(WE==1) begin
                if(A3!=0) begin
                    $display("@%h: $%d <= %h", pc, A3,WD3);
                    a[A3]<=WD3;
                end
            end
        end
    end

endmodule
```

图三：GRF 模块部分代码

4.ALU：（算数逻辑单元）

序号	功能名称	方向	功能描述
1	A[31:0]	I	进行操作的第一个数
2	B[31:0]	I	进行操作的第二个数
3	ALUctr[2:0]	I	进行操作的方式
4	ALUresult[31:0]	O	计算后的结果

模块解释：

1. ALUctr=0：输出 0 值

   ALUctr=1：加法

   ALUctr=2：减法

   ALUctr=3：或（or）操作

   ALUctr=4:输出 B 的值

   ALUctr=其他：输出 0

```
module ALU(A,B,ALUctr,ALUresult);
   input [31:0] A;
   input [31:0] B;
   input [2:0] ALUctr;
   output [31:0] ALUresult;
   assign ALUresult= ALUctr==0?0:
                     ALUctr==1?A+B:
                     ALUctr==2?A-B:
                     ALUctr==3?A|B:
                     ALUctr==4?B:
                     32'b0;
endmodule
```

图四：ALU 模块代码图

4. DM：（数据存储器）

序号	功能名称	方向	功能描述
1	addr[9:0]	I	进行操作的数据的地址
2	din[31:0]	I	数据
3	WE	I	控制是否 RAM 是否工作
4	Clk	I	时钟信号
5	Dout[31:0]	O	读出的数据
6	pc	I	当前 PC 值

模块解释：

1.此模块通过一组寄存器。

2.因为模块容量为 4kb, 所以进行操作的地址为 ALU 计算结果的 11-2 位。

3.用 ans 代表当前 addr 的 32 位表达值。

```
module DM(pc,clk,reset,MemWrite,addr,din,dout);
   input [31:0] pc;
   input clk;
   input reset;
   input MemWrite;
   input [9:0] addr;
   input [31:0] din;
   output [31:0] dout;

   reg [31:0] a [1023:0];
   wire [31:0] ans;
   integer i;
   initial begin
      for(i=0;i<=1023;i=i+1) a[i]=0;
   end
   assign dout=a[addr];
   assign ans={20'b0,addr[9:0],2'b0};
   always@(posedge clk) begin
      if(reset==1) begin
         for(i=0;i<=1023;i=i+1) a[i]<=0;
      end
      else begin
         if(MemWrite==1) begin
            $display("@%h: *%h <= %h",pc, ans,din);
            a[addr]<=din;
         end
      end
   end


endmodule
```

图五：DM 模块代码图

6.EXT：（数据扩展器）

序号	功能名称	方向	功能描述
1	IMM[15:0]	I	进行扩展的数
2	ExtOp[1:0]	I	扩展的方式
3	ExtIMM[31:0]	O	扩展后的数据

模块解释：

1. ExtOp=0：前 16 位 0 扩展

   ExtOp=1：符号扩展至 32 位

ExtOp=2：后 16 位补 0 扩展

2，用 ans（一个寄存器）以及 for 循环的利用实现。

```
reg [31:0] ans;
integer i;
initial  begin
ans=0;
end
assign ExtIMM=ans;
always@(*) begin
   if(ExtOp==0) begin
      for(i=31;i>=0;i=i-1) begin
         if(i>=16) begin
            ans[i]<=0;
         end
         else ans[i]<=IMM[i];
      end
   end
   else if(ExtOp==1) begin
      for(i=31;i>=0;i=i-1) begin
         if(i>=16) begin
            ans[i]<=IMM[15];
         end
         else ans[i]<=IMM[i];
      end
   end
   else if(ExtOp==2) begin
      for(i=31;i>=0;i=i-1) begin
         if(i>=16) begin
            ans[i]<=IMM[i-16];
         end
         else ans[i]<=0;
      end
   end
end


endmodule
```

图六：EXT 模块代码图

7. Control：（控制器）

序号	功能名称	方向	功能描述
1	opcode[5:0]	I	Special
2	funct[5:0]	I	Function
3	RegDst[1:0]	O	选择需要在 GRF 中写入数据的寄存器
4	ALUSrc	O	选择进行 ALU 操作的第二个操作数

5	MemtoReg[1:0]	O	选择存入 GRF 中寄存器的数据
6	RegWrite	O	是否在 GRF 中写入控制信号
7	MemWrite	O	DM 中的 RAM 使能端控制信号
8	nPC_sel	O	选择下一 PC 值信号
9	ExtOp[1:0]	O	数据扩展方式
10	ALUctr[2:0]	O	ALU 进行操作的方式
11	npc_sel3	O	解释见下
12	npc_sel4	O	解释见下

模块解释：

1. 首先通过 opcode 与 funct 选择出此时进行操作的指令方式。

2.通过 if_else 语句实现控制信号的选择。

3. RegDst==2 时选择 31 号寄存器，MemtoReg==2 是选择 PC-4，为了实现 jal 指令。

4.npc_sel3 表示在 beq 和（jal，jr）之中选择是哪一种指令。

5.npc_sel4 表示在 jal 和 jr 之中选择是哪一种指令。

funct	100000	100010					
opcode	000000	000000	001101	100011	101011	000100	001111
	add	sub	ori	lw	sw	beq	lui
RegDst[1:0]	1	1	0	0	x	x	0
ALUScr	0	0	1	1	1	0	1
Memto [1:0]	0	0	0	1	x	x	0
RegWrite	1	1	1	1	0	0	1
MemWrite	0	0	0	0	1	0	0
nPC_sel	0	0	0	0	0	1	0
ExtOp[1:0]	x	x	0	1	1	x	2
ALUctr[2:0]	1	2	3	1	1	2	4
npc_sel3	0	0	0	0	0	0	0
npc_sel4	0	0	0	0	0	0	0

图七：信号输出真值表（Memto 代表 MemtoReg）

funct	**001000**	**000000**	
opcode	000000	000000	000011
	jr	nop	jal
RegDst[1:0]	0	0	2
ALUScr	0	0	0
Memto [1:0]	0	0	2
RegWrite	0	0	1
MemWrite	0	0	0
nPC_sel	1	0	1
ExtOp[1:0]	0	0	0
ALUctr[2:0]	0	0	0
npc_sel3	1	0	1
npc_sel4	1	0	0

图七（续）：信号输出真值表（Memto 代表 MemtoReg）

三. 测试代码 MIPS 测试：代码：

```
            lui  $t1, 0x1111
            ori  $t2, $0, 0x1111
            addu  $t3, $t1, $t2
            addu  $t3, $t2, $t1
            lui  $t2, 0x1111
            beq  $t1, $t2, loop1
            ori  $t4, $0, 0x0001
loop1:
            nop
            ori  $t1, $0, 0x0001
            ori  $t3, $0, 0x0002
loop2:
            addu  $0, $t1, $t2
            addu  $t4, $t1, $t2
            subu  $t3, $t3, $t1
            beq  $t3, $t1, loop2
            jal  dfs
            nop
            ori  $t1, $0, 0x0000
            ori  $t2, $0, 0x1111
```

```
        ori $t1, $0, 0x0000
        ori $t2, $0, 0x1111
        sw $t2, 0($t1)
        lw $t3, 0, ($t1)
        nop
        lui $t1, 0x0000
        beq $t1, $0, done
dfs:
        ori $t1, $0, 0x000c
        lui $t2, 0x0001
        sw $t2, 0($t1)
        lw $t3, 0, ($t1)
        ori $t1, $0, 0x0001
        ori $t3, $0, 0x0002
loop3:
        subu $t3, $t3, $t1
        beq $t3, $t1, loop3
        jr $ra
done:
        nop
```

运行结果：（其中的$gp,$sp 不进行比较）

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0
$t1	9	0
$t2	10	4369
$t3	11	4369
$t4	12	286326785
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	6144
$sp	29	12284
$fp	30	0
$ra	31	12348

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
0	4369	0	0	65536	0	0	0	0
32	0	0	0	0	0	0	0	0
64	0	0	0	0	0	0	0	0
96	0	0	0	0	0	0	0	0
128	0	0	0	0	0	0	0	0
160	0	0	0	0	0	0	0	0
192	0	0	0	0	0	0	0	0
224	0	0	0	0	0	0	0	0
256	0	0	0	0	0	0	0	0
288	0	0	0	0	0	0	0	0
320	0	0	0	0	0	0	0	0
352	0	0	0	0	0	0	0	0
384	0	0	0	0	0	0	0	0
416	0	0	0	0	0	0	0	0

ISE 运行结果：

```
@00003000: $ 9 <= 11110000
@00003004: $10 <= 00001111
@00003008: $11 <= 11111111
@0000300c: $11 <= 11111111
@00003010: $10 <= 11110000
@00003020: $ 9 <= 00000001
@00003024: $11 <= 00000002
@0000302c: $12 <= 11110001
@00003030: $11 <= 00000001
@0000302c: $12 <= 11110001
@00003030: $11 <= 00000000
@00003038: $31 <= 0000303c
@0000305c: $ 9 <= 0000000c
@00003060: $10 <= 00010000
@00003064: *0000000c <= 00010000
@00003068: $11 <= 00010000
@0000306c: $ 9 <= 00000001
@00003070: $11 <= 00000002
@00003074: $11 <= 00000001
@00003074: $11 <= 00000000
@00003040: $ 9 <= 00000000
@00003044: $10 <= 00001111
@00003048: *00000000 <= 00001111
@0000304c: $11 <= 00001111
@00003054: $ 9 <= 00000000
```

	0	1	2	3
31	12348	0	0	0
27	0	0	0	0
23	0	0	0	0
19	0	0	0	0
15	0	0	0	286326785
11	4369	4369	0	0
7	0	0	0	0
3	0	0	0	0

	0	1	2	3
95	0	0	0	0
91	0	0	0	0
87	0	0	0	0
83	0	0	0	0
79	0	0	0	0
75	0	0	0	0
71	0	0	0	0
67	0	0	0	0
63	0	0	0	0
59	0	0	0	0
55	0	0	0	0
51	0	0	0	0
47	0	0	0	0
43	0	0	0	0
39	0	0	0	0
35	0	0	0	0
31	0	0	0	0
27	0	0	0	0
23	0	0	0	0
19	0	0	0	0
15	0	0	0	0
11	0	0	0	0
7	0	0	0	0
3	65536	0	0	4369

四． 思考题

(LO,T2)

1.地址每次增加 4 相当于 DM 中寄存器地址增加 1, 所以从 ALUresult 第二位取，又因为 DM 大小为 4KB，所以一共取 10 位，所以来自于 ALUresult，又因为如果直接于这个模块相连，可以取[11:2]位，所以是[11:2],而不是[9:0]。而且来自于 ALUresult。

2.对 PC 值，GRF 中的寄存器，以及 DM 中的数据储存器，因为在仿真过程中，如果不将这些置为初始值，那么在运算过程中会影响计算结果，导致结果出错。

（LO,T3）：

1.

1.应用 if_else 语句：

```
if(opcode==0) begin
    if(funct==33) begin
        regdst1<=1;
        alusrc1<=0;
        memtoreg1<=0;
        regwrite1<=1;
        memwrite1<=0;
        pc1<=0;
        extop1<=0;
        aluctr1<=1;
        pc2<=0;
        pc3<=0;
    end
    else if(funct==35) begin
        regdst1<=1;
        alusrc1<=0;
        memtoreg1<=0;
        regwrite1<=1;
        memwrite1<=0;
        pc1<=0;
        extop1<=0;
        aluctr1<=2;
        pc2<=0;
        pc3<=0;
    end
```

示例：

2.assign 语句完成操作码和控制信号的值之间的对应。

示例：

```
wire addu,subu,ori,lw,sw,beq,lui,jal,jr,nop;
assign addu=(opcode==6'b000000)&(funct==6'b100000);
assign subu=(opcode==6'b000000)&(funct==6'b100010);
assign jr=(opcode==6'b000000)&(funct==6'b001000);
assign ori=(opcode==6'b001101);
assign lw=(opcode==6'b100011);
assign sw=(opcode==6'b101011);
assign beq=(opcode==6'b000100);
assign lui=(opcode==6'b001111);
assign jal=(opcode==6'b000011);

assign RegDst[0]=(addu|subu);
assign RegDst[1]=(jal);
assign ALUSrc=(ori|lw|sw|lui);
```

3.利用宏定义：

示例：

```
`define addu=(opcode==6'b000000)&(funct==6'b100000);
`define subu=(opcode==6'b000000)&(funct==6'b100010);
`define jr=(opcode==6'b000000)&(funct==6'b001000);
`define ori=(opcode==6'b001101);
`define lw=(opcode==6'b100011);
`define sw=(opcode==6'b101011);
`define beq=(opcode==6'b000100);
`define lui=(opcode==6'b001111);
`define jal=(opcode==6'b000011);

assign RegDst[0]=(`addu|`subu);
assign RegDst[1]=(`jal);
assign ALUSrc=(`ori|`lw|`sw|`lui);
```

2. if_else 语句对于我自己来说用的最熟练，我觉得最不容易出错，但是有一个致命的缺点，过于复杂，容易出错。重复的代码语句多。

利用 assign 语句，有很强的拓展性，利于修改，方便观看，可以很快速的发现问题所在。

宏定义：与 assign 语句类似，方便快捷，逻辑更为简单。

（LO,T5）：

1.

其中 addi 指令：GPR[rt]<-GPR[rs]+ immediate

temp <- (GPR[rs]31<-GPR[rs]) + sign_extend(immediate)

if temp32 ≠ temp31 then SignalException(IntegerOverflow)

else GPR[rt] ← temp31..0

endif

addiu：GPR[rt] <-GPR[rs] + sign_extend(immediate)

在不考虑溢出的时候，可知 addi 直接执行 else 语句，所以等价。

add: temp <- (GPR[rs]31[illegible]PR[rs]) + (GPR[rt]31[illegible]PR[rt])

if temp32 ≠ temp31 then SignalException(IntegerOverflow)

else GPR[rd] ← temp31..0

endif

addu: GPR[rd] ←GPR[rs] + GPR[rt]

可知不考虑溢出的时候 add 也是直接执行 else 语句，所以等价。

2.

优点：不容易发生冲突，一个周期执行一个指令，不会像多周期或者流水线 CPU 那样会出现各种各样的问题，单周期 CPU 的搭建简单一些，并且逻辑也较为简单。

缺点：一个周期只能执行一条指令，满足不了现在的大多数功能，运行较慢，最为致命的一个缺点就是速度效率低。

3.

关系：因为在运用 jal 指令的时候，通常是进入一个函数，而大多数函数都会

递归，就是会重复的调用自己，这个时候就需要将当前执行的这一次函数的数据压入栈中，以免运行的时候会出现错误的信息，当调用 jr 指令回来的时候，通过栈中的值恢复当前需要的一些值，这样才能顺利的进行这一次函数的调用，否则很容易出错。