

1 实验要求

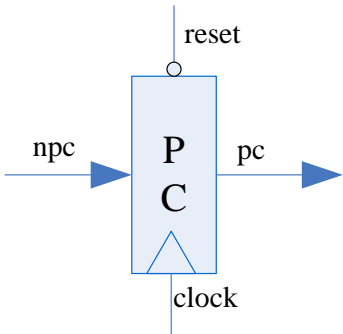
- 1) 设计单周期 cpu 的基本模块，包括：pc 模块（程序计数器），im 模块（指令存储器），gpr 模块（通用寄存器），alu 模块（算术逻辑单元），dm 模块（数据存储器）。
- 2) 连接基本模块，实现能够执行 addu 指令的单周期 CPU。
- 3) 在 1) 和 2) 的基础上完善功能，实现能够执行以下 R 型指令的单周期 CPU：addu, subu, add, and, or, slt。
- 4) 在 3) 的基础上完善功能，增加实现以下 I 型指令：addi, addiu, andi, ori, lui。
- 5) 在 4) 的基础上完善功能，增加实现以下和数据存储器相关的 MEM 型指令：lw, sw。
- 6) 在 5) 的基础上完善功能，增加实现以下跳转指令：beq, j, jal, jr 。

2 实验过程

2.1 pc 模块

2.1.1 功能描述

采用异步复位的方式，复位信号有效时，pc 的值复位为 0x0000_0030。复位信号无效时，在时钟信号上升沿完成自加。



2.1.2 端口说明

信号名	方向	描述
pc	O	当前的 pc
clock	I	时钟信号，上升沿有效
reset	I	复位信号，低电平有效

npc		npc = pc+4
-----	--	------------

2.1.3 实现过程

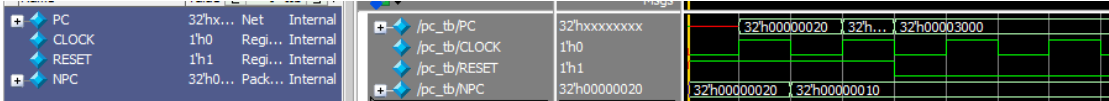
1) 代码

```
1  module pc(pc,clock,reset,npc);
2
3      output [31:0] pc;
4      input clock; //时钟信号, 上升沿有效
5      input reset; //复位信号, 低电平有效
6      input [31:0] npc; // npc = pc+4, 即下一条指令的pc
7
8      reg [31:0] temp; //临时变量
9      always @(posedge clock, negedge reset) //异步复位
10     begin
11         if (!reset)
12             temp <= 32'h00003000; //复位为00003000
13         else
14             temp <= npc;
15     end
16     assign pc = temp;
17
18 endmodule
```

分析: pc 的复位采用异步复位的方式, 无论时钟信号是否处于上升沿, 只要复位信号有效, 就会复位。老师也在课堂上清楚地说明了与同步复位的区别。二者区别在于: 同步复位在时钟信号发生变化且复位信号有效时进行复位。而异步复位当复位信号有效就进行复位。

以下是 testbench 部分截图以及仿真截图:

```
initial
begin
    NPC = 32'h00000020;
    CLOCK = 0;
    RESET = 1;
    #10 NPC = 32'h00000010;
    #10 RESET = 0;
end
always
#5 CLOCK = ~CLOCK;
```



Signal	Value	Internal
PC	32'hx...	Net
CLOCK	1'h0	Regi...
RESET	1'h1	Regi...
NPC	32'h0...	Pack...

Signal	Value	Internal
/pc_tb/PC	32'hxxxxxxxx	
/pc_tb/CLOCK	1'h0	
/pc_tb/RESET	1'h1	
/pc_tb/NPC	32'h00000020	

分析:

设 npc 初值为 0x00000020，初始时钟信号 0，复位信号为 1，此时 pc 没有值，因为时钟信号为 0，复位信号也无效。

5 个时间单位后，由于时钟信号上升沿到来，pc 置为 npc，此时 pc 为 0x000020。

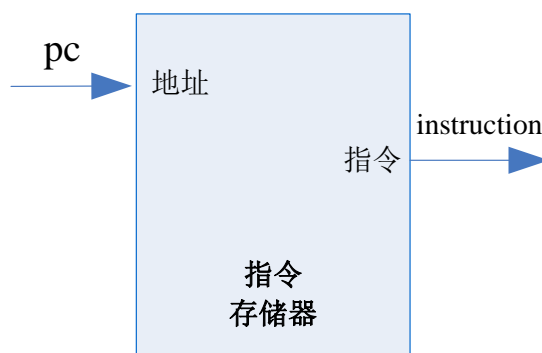
再过 15 个时间单位后，此时复位信号有效，pc 复位为 0x00000030。

2) 由于 pc 要在 always 中赋值，因此需定义为 reg 类型，我尝试定义一个 reg 类型的临时变量 temp，发现可以先用 temp 在 always 中赋值，随后再把 temp 的值传给 pc。

2.2 im 模块

2.2.1 功能描述

将 pc 的低 12 位作为地址，在指令存储器中找到相应的指令传给 instruction。



2.2.2 端口说明

信号名	方向	描述
pc	I	当前的 pc，低 12 位地址有效
instruction	O	取出的指令

2.2.3 实现过程

1) 代码

```
1  module im(instruction,pc);
2
3      output [31:0] instruction;
4      input [31:0] pc;
5
6      reg [31:0] ins_memory[1023:0]; //4kB指令存储器
7
8
9      assign instruction = ins_memory[pc[11:0]>>2]; //逻辑右移后的是真实地址
10
11  endmodule
```

分析：ins_memory 大小为 4KB，32 是位宽，1024 是存储器的深度，可以理解为 1024 个存储器连在一起，而每个存储器的位宽是 8。im 模块的输入地址 pc 是 32 位，但指令存储器 ins_memory 只有 4kB（即 1KW），所以取 pc 的低 12 位作为 ins_memory 的地址。另一方面，虽然 MIPS 指令都是固定长度的 32 位（一个字），但是 MIPS 是按字节进行编址的，所以字地址为 $pc \gg 2$ 。

以下是 testbench 部分截图以及仿真截图：

```
initial
begin
for(i=0; i<1024; i=i+1)
    IM.ins_memory[i] = i;
PC = 32'h00000000;
#10 PC = 32'hfffffff;
#10 $stop;
end
```

/im_tb/PC	32'hfffffff	32'h00000000	32'hfffffff
/im_tb/INSTRUCTION	32'h000003ff	32'h00000000	32'h000003ff
/im_tb/i	32'h00000400	32'h00000400	

分析：

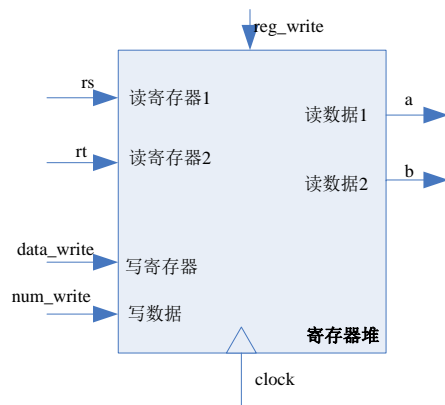
对指令存储器进行了简单的初始化，随机赋值两个 pc 测试一下结果。这部分比较简单。

2) 反思：课堂上讲指令存储器大小是 4KB 时，我其实不太理解，之后查了一下资料得以理解。

2.3 gpr 模块

2.3.1 功能描述

通用寄存器模块。0 号寄存器读出的值永远是 0，reg_write 高电平有效



2.3.2 端口说明

信号名	方向	描述
a	O	寄存器 1 的值
b	O	寄存器 2 的值
clock	I	时钟信号，上升沿有效
reg_write	I	写使能信号，高电平有效
rs	I	读寄存器 1 编号
rt	I	读寄存器 2 编号
num_write	I	写寄存器编号
data_write	I	写数据
gp_registers	O	32 个寄存器

2.3.3 实现过程

1) 代码

```

1  module gpr(a,b,clock,reg_write,num_write,rs,rt,data_write):
2
3      output [31:0] a; //读寄存器1的值
4      output [31:0] b; //读寄存器2的值
5      input clock; //时钟信号, 上升沿有效
6      input reg_write; //写使能信号, 高电平有效
7      input [4:0] rs; //读寄存器1编号
8      input [4:0] rt; //读寄存器2编号
9      input [4:0] num_write; //写寄存器编号
10     input [31:0] data_write; //写数据
11
12     reg [31:0] gp_registers[31:0]; //32个寄存器
13     integer i; //初始化用的临时变量
14
15     initial
16     begin
17         for (i = 0; i < 32; i = i + 1) //寄存器初始化
18             gp_registers[i] = 32'b0;
19     end
20
21     assign a = gp_registers[rs]; //读取rs的值
22     assign b = gp_registers[rt]; //读取rt的值
23
24     always @(posedge clock)
25     begin
26         if (reg_write == 1 && num_write != 0) //在时钟信号的上升沿写使能信号有效且写寄存器编号不为0
27             gp_registers[num_write] <= data_write;
28     end
29
30 endmodule

```

分析：一开始不理解为什么 num_write 非零才能写数据，后来想起理论课上讲过的要保证 0 号寄存器的值始终为 0，才解决了疑惑。关于寄存器的初始化可以采用 for 循环，也可以采用 txt 文件加载的方式。

以下是 testbench 部分截图以及仿真截图：

```

initial
begin
    CLOCK = 0;
    RS = 5'b0;
    RT = 5'b0;
    REG_WRITE = 0;
    NUM_WRITE = 5'b0;
    DATA_WRITE = 32'hffffffff;
    #10 REG_WRITE = 1;
    NUM_WRITE = 00001;
    #10 $stop;
end

always
begin
    #5 CLOCK = ~CLOCK;
end

initial
    $monitor("s=%32b",GPR.gp_registers[00001]);

```

A	32'h0...	Net	Internal	/gpr_tb/A	32'h00000000	32'h00000000				
B	32'h0...	Net	Internal	/gpr_tb/B	32'h00000000	32'h00000000				
CLOCK	1'h1	Regi...	Internal	/gpr_tb/CLOCK	1'h1					
REG_WRITE	1'h1	Regi...	Internal	/gpr_tb/REG_WRITE	1'h1					
RS	5'h00	Pack...	Internal	/gpr_tb/RS	5'h00	5'h00				
RT	5'h00	Pack...	Internal	/gpr_tb/RT	5'h00	5'h00				
NUM_WRITE	5'h01	Pack...	Internal	/gpr_tb/NUM_WRITE	5'h01	5'h00				
DATA_WRITE	32'hff...	Pack...	Internal	/gpr_tb/DATA_WRITE	32'hffffffff	32'hffffffff				

VSIM 11> run -all

```
# s=00000000000000000000000000000000
```

```
# s=11111111111111111111111111111111
```

分析：

初始化时令时钟信号为 0，rs 和 rt 都为 0，写使能信号为 0，num_write 为 0，data_write 为 0xffffffff，此时 1 号寄存器的值为 0（写到这里就感觉到自己考虑的不严谨，应该设置一组 reg_write=1, 而 num_write=0 的数据，观察结果）。

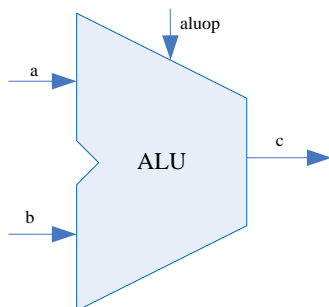
15 个时钟单位后，令 reg_write 为 1，num_write 为 5'b01，处于时钟信号的上升沿，此时通过 monitor 查看 1 号寄存器的值为 0xffffffff，说明成功写入。

2) 实验反思：开始想查看寄存器的值，却不知道在 tb 中怎么写，后来和同学讨论后得知可以跨模块调用，加上课堂上学习的 monitor 就能知道寄存器的中的内容，更有利于排除错误。

2.4 ALU 模块

2.4.1 功能描述

实现简单的 alu 功能，即实现两个数相加。



2.4.2 端口说明

信号名	方向	描述
c	O	$c = a + b$
a	I	输入 1

b	l	输入 2
---	---	------

2.4.3 实现过程

1) 代码

```
1 module alu(c,a,b);
2
3     output [31:0] c;
4     input [31:0] a;
5     input [31:0] b;
6
7     assign c = a + b;
8
9 endmodule
```

第一次实验 alu 模块相对简单。

以下是 testbench 部分截图以及仿真截图：

```
initial
begin
    A = 32'h00000030;
    B = 32'h30000000;
    #10 A = 32'h00000040;
    B = 32'h40000000;
    #10 $stop;
end
```

/alu_tb/C	32'h40000040	32h30000030		32h40000040	
/alu_tb/A	32'h00000040	32h00000030		32h00000040	
/alu_tb/B	32'h40000000	32h30000000		32h40000000	

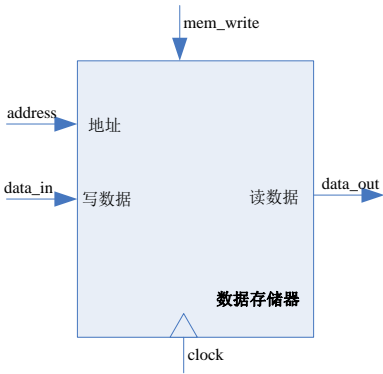
赋值了两遍，简单看下结果。

2) 实验反思：这个模块是实验一中最简单的，因此没怎么花时间。

2.5 DM 模块

2.5.1 功能描述

建立一个 4kB 大小的存储器以及读写存储器。



2.5.2 端口说明

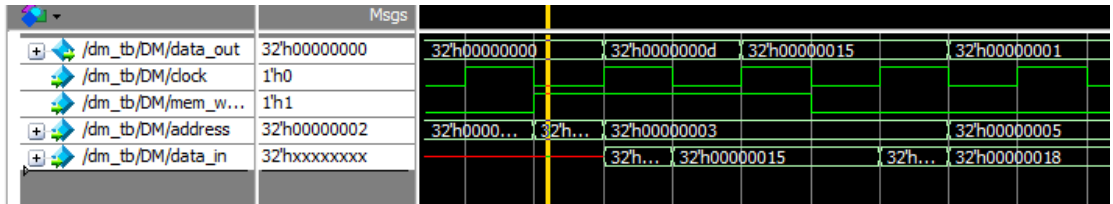
信号名	方向	描述
data_out[31:0]	O	读出的数据
clock	I	时钟信号，上升沿有效
mem_write	I	写使能信号 1：有效； 0：无效。
address[31:0]	I	读写地址
data_in[31:0]	I	要写入的数据

2.5.3 实现过程

1) 代码

```
1 module dm(data_out,clock,mem_write,address,data_in);
2
3     output [31:0] data_out;
4     input clock;
5     input mem_write;
6     input [31:0] address;
7     input [31:0] data_in;
8
9     reg [31:0] data_memory[1023:0]; //4K数据存储器
10
11     assign data_out = data_memory[address>>2];
12     always @(posedge clock)
13     begin
14         if (mem_write)
15             data_memory[address[11:2]] <= data_in;
16     end
17
18 endmodule
```

以下是 testbench 部分截图以及仿真截图：



2) 实验反思：该模块是实现寄存器堆的功能，对所给的 rs,rt,rd 寄存器编号，对 rt, rd 寄存器进行写操作。实现对寄存器的读写功能。

2.6 能够执行 addu 指令的单周期 CPU

2.6.1 功能描述

顶层模块，按照 pc 值从指令存储器中取出指令，按照指令定义，从寄存器堆中取出 GPR[rs]和 GPR[rt]，用 alu 模块实现 GPR[rs]+GPR[rt]，将结果存入寄存器 GPR[rd] 中。指令执行的同时，通过 pc+4 计算下条指令的地址 npc。

2.6.2 端口说明

信号名	方向	描述
clock	1	时钟信号，上升沿有效
reset	0	复位信号，低电平有效

2.6.3 实现过程

1) 代码

```

1  module s_cycle_cpu(clock,reset);
2
3      input clock; //时钟信号, 高电平有效
4      input reset; //复位信号, 低电平有效
5
6      wire [31:0] npc; //下一条指令的pc
7      wire [31:0] pc; //当前指令的pc
8      wire [31:0] instruction; //指令
9      wire [4:0] rs; //读寄存器1编号
10     wire [4:0] rt; //读寄存器2编号
11     wire [4:0] rd; //写寄存器编号
12     wire reg_write; //写使能信号
13     wire [31:0] a; //读寄存器1的值
14     wire [31:0] b; //读寄存器2的值
15     wire [31:0] c; //写数据
16
17     pc <= PC(pc(pc),.clock(clock),.reset(reset),.npc(npc));
18     assign npc = pc + 4;
19     im IM(.instruction(instruction),.pc(pc));
20     assign rs = instruction[25:21];
21     assign rt = instruction[20:16];
22     assign rd = instruction[15:11];
23     assign reg_write = 1;
24     gpr GPR(.a(a),.b(b),.clock(clock),.reg_write(reg_write),.num_write(rd),.rs(rs),.rt(rt),.data_write(c));
25     alu ALU(.c(c),.a(a),.b(b));
26
27 endmodule

```

分析: 用 mars 写了个 addu 指令导出, 准备写入指令存储器文件, 却发现编译错误, tb 代码如下:

```

1  `timescale 10ns/1ns
2
3  module s_cycle_cpu_tb;
4
5      reg CLOCK;
6      reg RESET;
7
8      integer i;
9
10     s_cycle_cpu CPU(.clock(CLOCK),.reset(RESET));
11
12     always
13     #5 CLOCK = ~CLOCK;
14
15     initial
16     $readmemb("C:\Users\admin\Desktop\jizu\1.txt",CPU.IM.ins_memory);
17
18     initial
19     begin
20         CLOCK = 0;
21         RESET = 1;
22         #5 RESET = 0;
23         #5 RESET = 1;
24         #100
25         for(i=0;i<10;i=i+1)
26             $display("display gp_registers[%2d] = %8h",i,CPU.GPR.gp_registers[i]);
27         $finish;
28     end
29
30 endmodule

```

```
** Error: (vlog-13067) C:/Users/admin/Desktop/VerilogCPU-master/modules/s_cycle_cpu_tb.v(13.4): Syntax error, unexpected non-printable character with the hex value '0xe3'.

** Error: (vlog-13067) C:/Users/admin/Desktop/VerilogCPU-master/modules/s_cycle_cpu_tb.v(13.5): Syntax error, unexpected non-printable character with the hex value '0x80'.

** Error: (vlog-13067) C:/Users/admin/Desktop/VerilogCPU-master/modules/s_cycle_cpu_tb.v(13.6): Syntax error, unexpected non-printable character with the hex value '0x80'.
```

2) 这个问题，经过查询发现是 notepad++ 的编码格式被更改。

```
initial
    $readmemb("C:\Users\admin\Desktop\jizu\1.txt",CPU.IM.ins_memory);

initial
begin
    CLOCK = 0;
    RESET = 1;
    #5 RESET = 0;
    #5 RESET = 1;
    #100
    for(i=0;i<13;i=i+1)
        $display("display gp_registers[%2d] = %8h",i,CPU.GPR.gp_registers[i]);
    $finish;
end

assign CPU.GPR.reg_write = 1;

# display gp_registers[ 5] = 00000005
# display gp_registers[ 6] = 00000006
# display gp_registers[ 7] = 00000007
# display gp_registers[ 8] = 00000008
# display gp_registers[ 9] = 00000009
# display gp_registers[10] = 0000000a
# display gp_registers[11] = 0000000b
# display gp_registers[12] = 0000000c
```

根据 mars 导出的指令来看，应该是将 10 号寄存器和 11 号寄存器的值相加结果保存到 9 号寄存器。猜测可能与时钟信号有关，寄存器只在时钟信号上升沿写入。

2.7 能够执行 R 型指令的单周期 CPU

2.7.1 功能描述

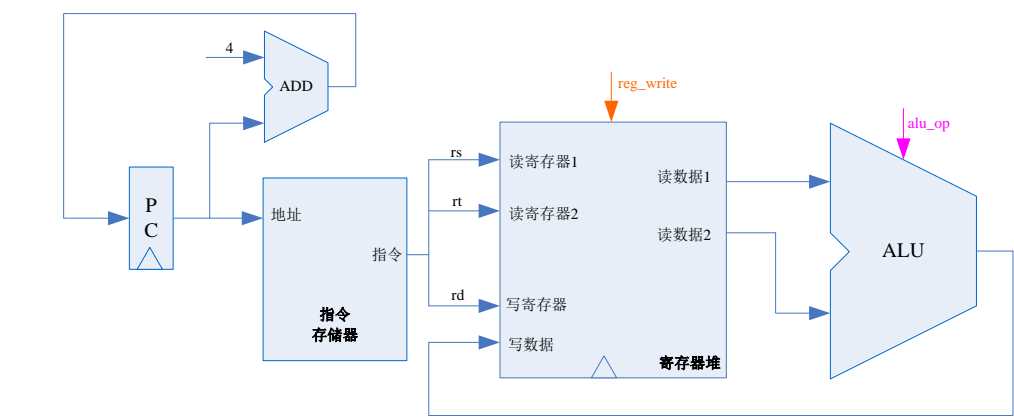
实现能够执行以下 R 型指令的单周期 CPU: addu, subu, add, and, or, slt

2.7.2 指令分析

R 型指令格式如下：

位段	3126	2521	2016	1511	106	50
编码	操作码(6位)	rs(5位)	rt(5位)	rd(5位)	shamt(5位)	功能码(6位)
addu rd, rs, rt	000000	rs	rt	rd	00000	100001
GPR[rd]←GPR[rs] + GPR[rt]						
subu rd, rs, rt	000000	rs	rt	rd	00000	100011
GPR[rd]←GPR[rs] - GPR[rt]						
add rd, rs, rt	000000	rs	rt	rd	00000	100000
GPR[rd]←GPR[rs] + GPR[rt]						
and rd, rs, rt	000000	rs	rt	rd	00000	100100
GPR[rd]←GPR[rs] & GPR[rt]						
or rd, rs, rt	000000	rs	rt	rd	00000	100101
GPR[rd]←GPR[rs] GPR[rt]						
slt rd, rs, rt	000000	rs	rt	rd	00000	101010
GPR[rd]←(GPR[rs] < GPR[rt])						

数据通路图如下：



分析：

根据 pc 的低 12 位找到对应地址中存放的指令，并传送给 instruction，根据 instruction 的对应字段找出源寄存器，并读出内容送到 alu 单元进行运算，alu 单元根据选择信号 aluop 来选择运算，运算结果在写使能信号 reg_write 有效时写入目的寄存器。

与前一个实验相比，指令格式相同（各字段功能相同），数据通路也相同；alu 模块增加了减、与、或、比较（<）功能；alu 模块增加了一个选择信号 aluop，决定执行哪种操作；增加了 ctrl 模块，产生控制信号。

2.7.3 实现过程

1) 代码

以下是一开始写的第一版代码：

ctrl 模块：

```

1  module ctrl(reg_write,aluop,op,funct);
2
3      output reg_write;
4      output [2:0] aluop;
5
6      input [5:0] op;
7      input [5:0] funct;
8
9      assign reg_write = (op == 000000)? 1:0;
10     assign aluop = (op == 000000)? (funct == 100001)? 001:
11         (funct == 100011)? 011:
12         (funct == 100000)? 000:
13         (funct == 100100)? 100:
14         (funct == 100101)? 101:
15         (funct == 101010)? 101:
16         000;
17
18
19 endmodule

```

alu 模块:

```

1  module alu(c,a,b,aluop);
2
3      output reg [31:0] c;
4      input [31:0] a;
5      input [31:0] b;
6      input [2:0] aluop;
7
8      always @(a, b, aluop)
9      case (aluop)
10         3'b000: c = a + b;
11         3'b001: c = a + b;
12         3'b011: c = a - b;
13         3'b100: c = a & b;
14         3'b101: c = a | b;
15         3'b010: c = (a < b)? 32'd1 : 32'd0;
16         default c = a + b;
17     endcase
18
19 endmodule

```

代码的可读性差，没有使用宏定义，也没有注意有符号数加法和无符号数加法的区别，与前一个实验最后碰到的问题一样，实例化 cpu 没办法传参数，后来发现应该是利用文件初始化指令存储器和寄存器的时候，文件路径没有打\\。

以下是更改后的代码:

ctrl 模块:

```

1  `include "header.v"
2  module ctrl(reg_write,aluop,op,funct);
3
4      output reg_write; //写使能信号
5      output [2:0] aluop; //选择信号，根据功能码选择不同的运算
6      input [5:0] op; //操作码
7      input [5:0] funct; //功能码
8
9      assign reg_write = (op == `op_R)? 1 : 0; //op为R型指令的操作码时，写使能信号才有效
10     assign aluop = (op == `op_R)? /*(funct == `funct_addu)? `ADD: */ //缺省为add
11         (funct == `funct_subu)? `SUB:
12         /*(funct == `funct_add)? `ADD: */
13         (funct == `funct_and)? `AND:
14         (funct == `funct_or)? `OR:
15         (funct == `funct_slt)? `SLT:
16         `ADD;
17
18
19 endmodule

```

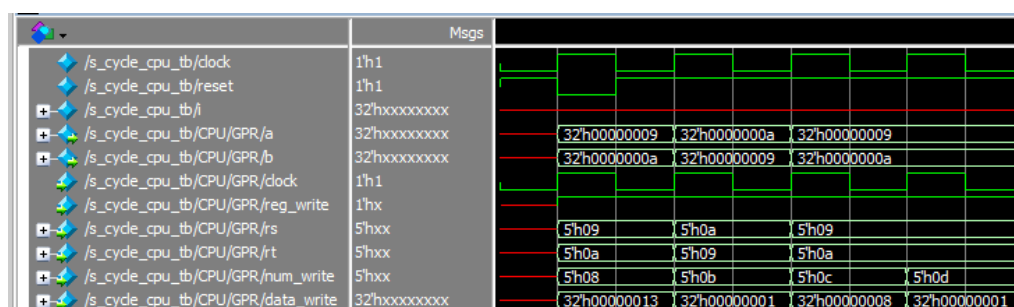
顶层模块:

```

1  module s_cycle_cpu(clock,reset);
2
3      //输入
4      input clock;
5      input reset;
6
7      wire [31:0] npc; //npc = pc + 4
8      wire [31:0] pc; //当前指令的pc
9      wire [31:0] instruction;
10     wire [4:0] rs; //读寄存器1
11     wire [4:0] rt; //读寄存器2
12     wire [4:0] rd; //写寄存器
13     wire reg_write; //写使能信号
14     wire [31:0] a; //读寄存器1的值
15     wire [31:0] b; //读寄存器2的值
16     wire [31:0] c; //待写入数据
17     wire [2:0] aluop; //选择信号
18     wire [5:0] op; //操作码
19     wire [5:0] funct; //功能码
20
21     //实例化
22     pc PC(.pc(pc),.clock(clock),.reset(reset),.npc(npc));
23     im IM(.instruction(instruction),.pc(pc));
24     gpr GPR(.a(a),.b(b),.clock(clock),.reg_write(reg_write),.num_write(rd),.rs(rs),.rt(rt),.data_write(c));
25     alu ALU(.c(c),.a(a),.b(b),.aluop(aluop));
26     ctrl CTRL(.reg_write(reg_write),.aluop(aluop),.op(op),.funct(funct));
27     assign npc = pc + 4;
28     assign op = instruction [31:26];
29     assign rs = instruction [25:21];
30     assign rt = instruction [20:16];
31     assign rd = instruction [15:11];
32     assign funct = instruction [5:0];
33
34
35 endmodule

```

仿真波形如下：



分析：

reset 信号有效后，pc 复位，开始按指令存储器中的指令顺序执行，为了方便查看寄存器中值，采用 display 方法显示到控制台：

```

# display gp_registers[ 8] = 00000013
# display gp_registers[ 9] = 00000009
# display gp_registers[10] = 0000000a
# display gp_registers[11] = 00000001
# display gp_registers[12] = 00000008
# display gp_registers[13] = 00000001

```

采用 mars 写了四条语句用来测试结果：

```

add $t0,$t1,$t2
subu $t3,$t2,$t1
and $t4,$t1,$t2
slt $t5,$t1,$t2

```

mars 中的 t0 寄存器对应 gp_registers[8]，其他按顺序对应，可以看到，简单

分析：

- 1) 操作码决定指令类型；
- 2) 指令低十六位为立即数字段，十六位立即数需要扩展为 32 位，而且需要根据指令类型确定是符号扩展还是无符号扩展；
- 3) ALU 的两个源操作数是寄存器 GPR[rs]和立即数；
- 4) ALU 结果写入寄存器 GPR[rt]。

与 R 型指令的区别在于：

1. 增加器件

- 1) 新增 2 选 1 数据选择器——选择不同的目标寄存器（选 rt/rd）
- 2) 新增 2 选 1 数据选择器——源操作数选择（立即数/寄存器）
- 3) 立即数扩展——（符号扩展/零扩展）

2. 增加控制信号

- 1) regdst 为 1，选择 rd 是目标寄存器。
- 2) alusrc 为 1，选择立即数作为第二源操作数。

2.8.3 实现过程

1) 代码

ext 模块：

```
1 module ext (immediate, ExtSel, extended_immediate);
2
3     input [15:0] immediate; //16位立即数
4     input ExtSel; //选择信号
5     output [31:0] extended_immediate; //扩展后的立即数
6
7     assign extended_immediate = (ExtSel ? {{16{immediate[15]}}, immediate[15:0]} //根据选择信号判断是符号扩展还是零扩展
8                                     : {{16{1'b0}}, immediate[15:0]});
9 endmodule
```

ctrl 模块：

```

1  `include "header.v"
2  module ctrl(reg_write,aluop,op,funct,regdst,extop,alusrc);
3
4      output reg reg_write; //写使能信号
5      output reg [2:0] aluop; //选择信号
6      input [5:0] op; //操作码
7      input [5:0] funct; //功能码
8      output reg extop; //符号扩展信号, 高电平有效
9      output reg alusrc; //选择立即数的控制信号, 高电平有效
10     output reg regdst; //选择寄存器的控制信号, 高电平有效
11
12     always @(*)
13     begin
14         reg_write = 0; regdst = 0; extop = 0; alusrc = 1; aluop = 'ADD;
15         case(op)
16         `op_R:
17             begin
18                 reg_write = 1; regdst = 1; extop = 0; alusrc = 0;
19                 case (funct)
20                 `funct_addu: aluop = 'ADD;
21                 `funct_subu: aluop = 'SUB;
22                 `funct_add: aluop = 'ADD;
23                 `funct_and: aluop = 'AND;
24                 `funct_or: aluop = 'OR;
25                 `funct_slt: aluop = 'SLT;
26                 endcase
27             end
28         `op_addi: begin reg_write = 1; regdst = 1; extop = 1; alusrc = 1; aluop = 'ADD; end //根据不同指令, 置选择信号的值
29         `op_addiu: begin reg_write = 1; regdst = 0; extop = 1; alusrc = 1; aluop = 'ADD; end
30         `op_andi: begin reg_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = 'AND; end
31         `op_ori: begin reg_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = 'OR; end
32         `op_lui: begin reg_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = 'LUI; end
33         endcase
34     end
35 endmodule
36
37

```

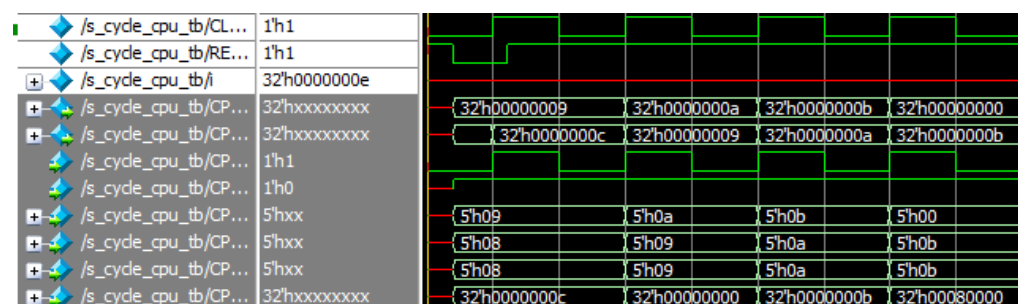
顶层模块:

```

9      wire [31:0] pc;
10     wire [31:0] instruction;
11     wire [4:0] rs; //读寄存器1
12     wire [4:0] rt; //读寄存器2
13     wire [4:0] rd; //写寄存器
14     wire reg_write; //写使能信号
15     wire [31:0] a; //读寄存器1的值
16     wire [31:0] b; //读寄存器2的值
17     wire [31:0] bl; //第二源操作数
18     wire [31:0] c; //待写入数据
19     wire [2:0] aluop; //选择信号
20     wire [5:0] op; //操作码
21     wire [5:0] funct; //功能码
22     wire regdst; //regdst为1, 选择rd是目标寄存器
23     wire extop; //extop为1, 选择符号扩展
24     wire alusrc; //alusrc为1, 选择立即数作为第二源操作数
25     wire [15:0] imm; //立即数
26     wire [31:0] eximm; //扩展后的立即数
27     wire [4:0] num_write; //写寄存器
28
29     pc <= PC(.pc(pc),.clock(clock),.reset(reset),.npc(npc));
30     im <= IM(.instruction(instruction),.pc(pc));
31     ctrl CTRL(.reg_write(reg_write),.aluop(aluop),.op(op),.funct(funct),.regdst(regdst),.extop(extop),.alusrc(alusrc));
32     ext <= EXT(.immediate(imm),.ExtSel(extop),.extended_immediate(eximm));
33     gpr <= GPR(.a(a),.b(b),.clock(clock),.reg_write(reg_write),.num_write(num_write),.rs(rs),.rt(rt),.data_write(c));
34     alu <= ALU(.c(c),.a(a),.b(bl),.aluop(aluop));
35
36     assign npc = pc + 4;
37     assign op = instruction [31:26];
38     assign rs = instruction [25:21];
39     assign rt = instruction [20:16];
40     assign rd = instruction [15:11];
41     assign funct = instruction [5:0];
42     assign imm = instruction [15:0];
43     assign num_write = regdst ? rd : rt;
44     assign bl = alusrc ? eximm : b;

```

仿真波形如下:



分析：

采用 display 方法打印寄存器的值来查看结果：

```
# display gp_registers[ 8] = 0000000c
# display gp_registers[ 9] = 00000000
# display gp_registers[10] = 0000000b
# display gp_registers[11] = 00080000
# display gp_registers[12] = 0000000c
# display gp_registers[13] = 0000000d
```

利用 mars 编写的汇编指令如下：

```
1 addi $t0,$t1,3
2 andi $t1,$t2,1
3 ori  $t2,$t3,1
4 lui  $t3,8
```

t0 对应 8 号寄存器，可以看到运算结果正确。

2) 实验反思：

ctrl 模块设计的主要思想是根据不同指令的要求设置一个选择信号，这样在顶层模块使用一个判断语句就能实现二选一选择器。

2.9 添加 MEM 型指令

2.9.1 功能描述

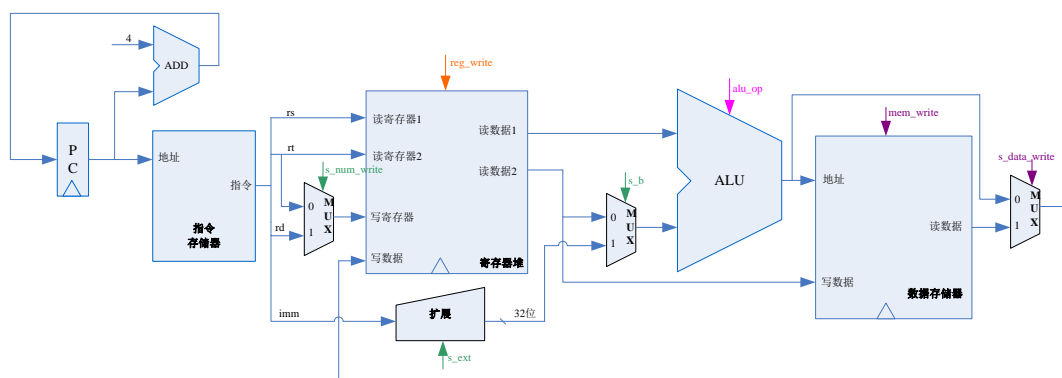
在 2.8 的基础上完善功能，增加实现以下和数据存储器相关的 MEM 型指令：
lw, sw。

2.9.2 指令分析

I 型指令格式如下：

指令格式	描述	31	26	25	21	20	16	15	0
		op(6位)				rs(5位)			
sw rt, offset(base)	$Addr \leftarrow GPR[base] + sign_ext(offset)$ $memory[Addr] \leftarrow GPR[rt]$	101011				rs(base)			
lw rt, offset(base)	$Addr \leftarrow GPR[base] + sign_ext(offset)$ $GPR[rt] \leftarrow memory[Addr]$	100011				rs(base)			

数据通路图如下：



分析:

lw 指令特点:

- 1) 指令类型仅由 op 决定; 目标寄存器为 rt
- 2) 唯一一条从存储器中读取数据的指令
- 3) 指令中的 16 位立即数为偏移量 offset, 需符号扩展成 32 位 (符号扩展, 16 位有符号数)
- 4) rs 和符号扩展后的立即数作为 ALU 的数据输入, ALU 的计算结果为要访问的存储单元的地址 (用这个地址把数据取出来放到 rt 中)

sw 指令特点:

- 1) 指令类型仅由 op 决定
- 2) 唯一一条向存储器写数据的指令
- 3) 指令中的 16 位立即数为偏移量 offset, 需符号扩展成 32 位
- 4) rs 和符号扩展后的立即数作为 ALU 的数据输入, ALU 的计算结果为要访问的存储单元的地址

与 I 型指令相比:

1. 增加器件
 - 1) 数据存储器—实验一最开始实现的
 - 2) 2 选 1 数据选择器—运算类指令和 Iw 指令写入的目的寄存器的结果来源不同。选择器的控制信号为 MemtoReg (寄存器中的数据来自 ALU 计算结果/数据 RAM)
2. 增加控制信号
 - 1) memread 选择写入寄存器的是存储器数据还是运算器数据 (1 存储器 0 ALU)

2) memwrite——存储器写允许信号

2.9.3 实现过程

1) 代码

ext 模块:

```

1 module ext (immediate, ExtSel, extended_immediate);
2
3     input [15:0] immediate; //16位立即数
4     input ExtSel; //选择信号
5     output [31:0] extended_immediate; //扩展后的立即数
6
7     assign extended_immediate = (ExtSel ? ({16{immediate[15]}}, immediate[15:0]) : ({16{1'b0}}, immediate[15:0])); //根据选择信号判断是符号扩展还是零扩展
8
9 endmodule

```

ctrl 模块:

```

1 `include "header.v"
2 module ctrl(reg_write,aluop,op,funct,regdst,extop,alusrc);
3
4     output reg reg_write; //写使能信号
5     output reg [2:0] aluop; //选择信号
6     input [5:0] op; //操作码
7     input [5:0] funct; //功能码
8     output reg extop; //符号扩展信号, 高电平有效
9     output reg alusrc; //选择立即数的控制信号, 高电平有效
10    output reg regdst; //选择寄存器的控制信号, 高电平有效
11
12    always @(*)
13    begin
14        reg_write = 0; regdst = 0; extop = 0; alusrc = 1; aluop = `ADD;
15        case(op)
16        `op_R:
17            begin
18                reg_write = 1; regdst = 1; extop = 0; alusrc = 0;
19                case (funct)
20                `funct_addu: aluop = `ADD;
21                `funct_subu: aluop = `SUB;
22                `funct_add: aluop = `ADD;
23                `funct_and: aluop = `AND;
24                `funct_or: aluop = `OR;
25                `funct_slt: aluop = `SLT;
26                endcase
27            end
28        `op_addi: begin reg_write = 1; regdst = 0; extop = 1; alusrc = 1; aluop = `ADD; end //根据不同指令, 置选择信号的值
29        `op_addiu: begin reg_write = 1; regdst = 0; extop = 1; alusrc = 1; aluop = `ADD; end
30        `op_andi: begin reg_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = `AND; end
31        `op_ori: begin reg_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = `OR; end
32        `op_lui: begin reg_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = `LUI; end
33        endcase
34    end
35 end
36
37 endmodule

```

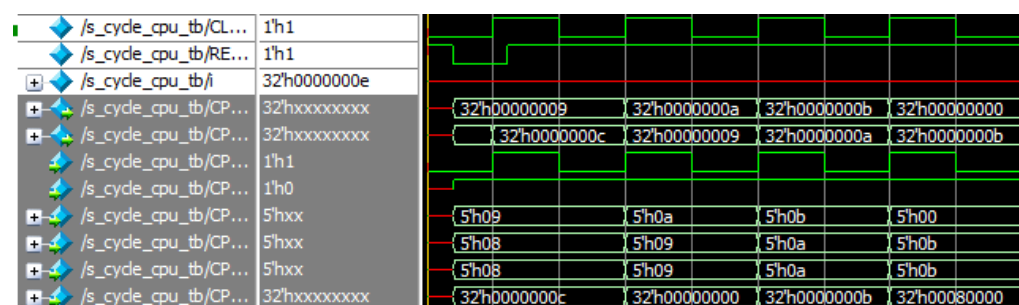
顶层模块:

```

9      wire [31:0] pc;
10     wire [31:0] instruction;
11     wire [4:0] rs; //读寄存器1
12     wire [4:0] rt; //读寄存器2
13     wire [4:0] rd; //写寄存器
14     wire reg_write; //写使能信号
15     wire [31:0] a; //读寄存器1的值
16     wire [31:0] b; //读寄存器2的值
17     wire [31:0] bl; //第二源操作数
18     wire [31:0] c; //待写入数据
19     wire [2:0] aluop; //选择信号
20     wire [5:0] op; //操作码
21     wire [5:0] funct; //功能码
22     wire regdst; //regdst为1, 选择rd是目标寄存器
23     wire extop; //extop为1, 选择符号扩展
24     wire alusrc; //alusrc为1, 选择立即数作为第二源操作数
25     wire [15:0] imm; //立即数
26     wire [31:0] eximm; //扩展后的立即数
27     wire [4:0] num_write; //写寄存器
28
29     pc <- PC(pc, .clock(clock), .reset(reset), .npc(npc));
30     im <- IM(instruction, .pc(pc));
31     ctrl <- CTRL(.reg_write(reg_write), .aluop(aluop), .op(op), .funct(funct), .regdst(regdst), .extop(extop), .alusrc(alusrc));
32     ext <- EXT(.immediate(imm), .ExtSel(extop), .extended_immediate(eximm));
33     gpr <- GPR(.a(a), .b(b), .clock(clock), .reg_write(reg_write), .num_write(num_write), .rs(rs), .rt(rt), .data_write(c));
34     alu <- ALU(.c(c), .a(a), .b(bl), .aluop(aluop));
35
36     assign npc = pc + 4;
37     assign op = instruction [31:26];
38     assign rs = instruction [25:21];
39     assign rt = instruction [20:16];
40     assign rd = instruction [15:11];
41     assign funct = instruction [5:0];
42     assign imm = instruction [15:0];
43     assign num_write = regdst ? rd : rt;
44     assign bl = alusrc ? eximm : b;

```

仿真波形如下：



分析：

采用 display 方法打印寄存器的值来查看结果：

```

# display gp_registers[ 8] = 0000000c
# display gp_registers[ 9] = 00000000
# display gp_registers[10] = 0000000b
# display gp_registers[11] = 00080000
# display gp_registers[12] = 0000000c
# display gp_registers[13] = 0000000d

```

利用 mars 编写的汇编指令如下：

```

1  addi $t0, $t1, 3
2  andi $t1, $t2, 1
3  ori  $t2, $t3, 1
4  lui  $t3, 8

```

t0 对应 8 号寄存器，可以看到运算结果正确。

2) 实验反思：

ctrl 模块设计的主要思想是根据不同指令的要求设置一个选择信号,这样在顶层模块使用一个判断语句就能实现二选一选择器。

2.10 添加 J 型指令

2.10.1 功能描述

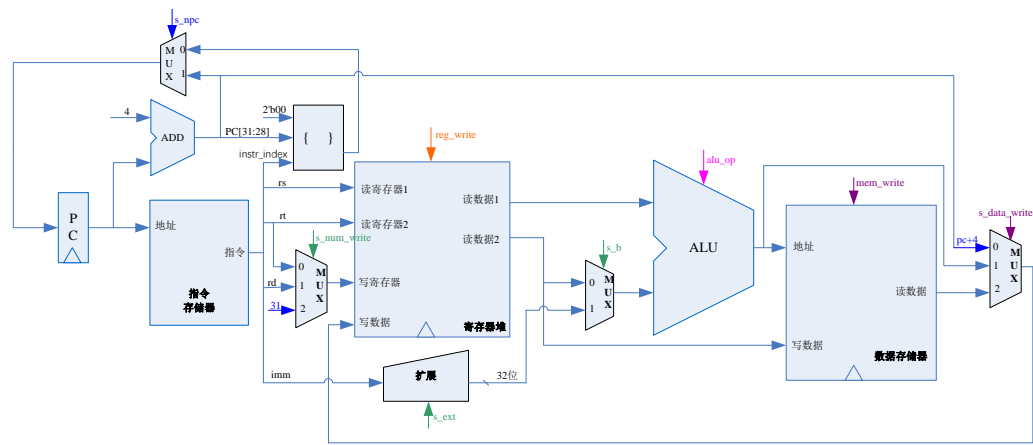
在 2.9 的基础上完善功能,增加实现以下跳转指令: beq, j, jal, jr 。

2.10.2 指令分析

J 型指令格式如下:

指令格式	描述	31 26	25 21	20 16	15 11	10 6	5 0
		6位	5位	5位	5位	5位	6位
beq rs, rt, offset	if (GPR[rs] == GPR[rt]) PC ←PC + 4 +(sign_extend(offset)<<2) else PC ←PC + 4	000100	rs	rt	offset		
j target	PC ←{PC[31..28], instr_index ,2'b00}	000010	instr_index				
jal target	①GPR[31] ←PC + 4 ②PC ←{PC[31..28], instr_index ,2'b00}	000011	instr_index				
jr rs	PC ←GPR[rs]	000000	rs	00000	00000	00000	001000

数据通路图如下:



beq 指令特点:

- 1) 指令类型仅由 op 决定
- 2) R[rs] 与 R[rt] 相减, 并以差为 0 作为继续执行的条件

3) 指令中的 16 位立即数为偏移量 offset，需乘以 4 后做符号扩展成 32 位；然后和 PC+4 相加作为新的 PC 值。该指令在差为 0 的时候会改变 PC 值

bne 指令特点：

- 1) 指令类型仅由 op 决定
- 2) R[rs] 与 R[rt] 相减，并以差不为 0 作为继续执行的条件
- 3) 指令中的 16 位立即数为偏移量 offset，需乘以 4 后做符号扩展成 32 位；然后和 PC+4 加作为新的 PC 值。该指令在差不为 0 的时候会改变 PC 值

j 指令特点：

- 1) 指令类型仅由 op 决定
- 2) 指令中的 26 位立即数为地址，需乘以 4 后作为新的 PC 值。该指令会改变 PC 值

jal 指令特点：

- 1) 指令类型仅由 op 决定
- 2) 将 PC + 4 的值存放到 \$31 中
- 3) 指令中的 26 位立即数为地址，需乘以 4 后作为新的 PC 值。该指令会改变 PC 值

分析：

与 mem 型指令区别：

1. 增加器件
 - 1) npc 模块
 - 2) 三选一选择器——主要新增 PC+4 选择分支，考虑 jal 指令
2. 增加控制信号
 - 1) zero 信号——alu 输出，表明是 beq 指令
 - 2) s 信号——ctrl 模块输出，确定当前指令类型

2.10.3 实现过程

- 1) 代码

npc 模块:

```
module npc(npc_t,instr_index,offset,a,zero,s);

    output reg [31:0] npc;
    input [31:0] npc_t; //npc_t = pc+4
    input [25:0] instr_index;
    input [31:0] offset; //指令低16位符号扩展
    input [31:0] a; //alu模块a输出
    input zero; //alu模块zero输出
    input [1:0] s; //ctrl模块产生, 确定当前指令类型

    always @(*)
    begin
        npc = npc_t;
        case(s)
            `NONE: npc = npc_t;
            `BEQ:
                begin
                    if(zero)
                        npc = npc_t + {offset[29:0], 2'b00};
                    else
                        npc = npc_t;
                end
            `J_JAL: npc = {npc_t[31:28], instr_index, 2'b00};
            `JR: npc = a;

            default: npc = 32'hxxxxxxxx;
        endcase
    end

endmodule
```

alu 模块:

```
always @(*)
begin
    zero = 0;
    case (aluop)
        `ADD: c <= a + b;
        `SUB: c <= a - b;
        `AND: c <= a & b;
        `OR: c <= a | b;
        `SLT: c <= $signed(a) < $signed(b) ? 32'd1 : 32'd0;
        `LUI: c <= b << 16;
        `EQB: c <= b;
        default: c <= b;
    endcase
    zero <= (c == 0) ? 1 : 0;
end
```

分析: 新增了 zero 标志位, alu 运算结果为 0 时, 输出 zero 信号至 npc 模块。

顶层模块:

顶层模块中主要新增了 r31 的判断表达式, r31 用于 jal 指令的执行, 执行 jal 时要先将 PC+4 保存到 31 号寄存器当中。

num_write 是写寄存器, b1 的第二源操作数。

3 实验总结

对于实验一：

1. 实验一实验总体比较简单，感觉作为入门来说是很不错的实验，帮助我们复习了 verilog 的基本语法，同时也通过模块化实现 addu 指令学习了 verilog 模块化的思想。

2. 这次实验又让感受到了语言还得在应用中学习，这次实验真的学到了很多。老师也给我们总结了一些需要注意的点，比如：

parameter 的用法：parameter 是常量，所以不能在运行时修改它的值，也就是说在组合逻辑或者时序逻辑中不能赋值。parameter 可以在模块中声明，也可以在声明模块的时候进行声明。

模块实例化的不同方法：方法一 fun #(xx) Fun(t1, t2);

方法二 fun #(xx) Fun(. t1(t1), . t2(t2));

\$monitor 和 \$display 的用法和区别：\$monitor 对持续监测指定变量，当变量的值发生变化时，就立即显示对应的输出语句。\$display 用于检测指定变量，运行到 \$display 语句时显示对应的输出语句。

阻塞赋值语句 (=)，非阻塞赋值语句 (=>) 的用法：

非阻塞赋值方式：

块结束后才完成赋值操作，值不是立刻改变，常用于 always 结构。

阻塞赋值方式：

赋值语句执行完后，块结束，值在赋值语句执行完后立刻改变。

对于实验二：

1. 写 testbench 熟练了许多，也发现了上一次实验存在的问题，对 questa sim 的操作也更加熟练，学会了在 memory_list 中可以看寄存器和存储器的内容，方便定位错误，此外还了解了编译错误可能是跨模块调用出现的问题。

2. 这次实验对三种指令的理论知识又进行了复习，理解上更加深刻了。

3. 这次实验的难度感觉还是比较大的，借助实验一的基础慢慢摸索实现单周期 CPU 的各种功能，也遇见了许多问题，但是通过查找资料以及询问其他同学解决了许多困惑，让我体会到了发现问题、解决问题的乐趣。

4. 这次的实验不仅提高了我发现问题、解决问题的能力，同时也让我在思考问题的时候思维更加细致严谨，不再像以前那样容易出小错误，代码也比之前规范了，思维也比以前清晰了，同时，老师的讲解非常细致，全面的向我们说明了实验过程中可能遇到的问题和难点，让我们在实验过程中能够更高效地完成任务，也让我们更好地吸收知识，在老师的帮助下学到了很多。