实验报告一

实验名称： 单周期 CPU **日期：** 2022.11.9

班级： 10012007 **学号：** 2020302977 **姓名：** 张建龙

一、 实验要求：

1. 设计单周期 CPU 的基本模块 PC,IM,GPR,ALU,DM
2. 按照 addu 指令的功能把基本模块进行连接，形成一个能执行 addu 指令的单周期 CPU。
3. 使单周期 CPU 支持 R 型指令。

在实验一的基础上增加实现以下指令：

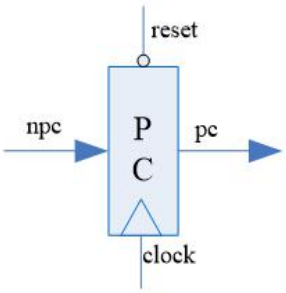
1. addi，addiu，andi，ori，lui；
2. lw，sw；
3. beq，j，jal，jr 。

由于第二次实验的内容是在第一次实验的基础上整合的，因此报告中的代码都是最终实验的结果，可能会缺少过程性的步骤。

二、 实验过程：

1. 基本模块

（1）pc 模块



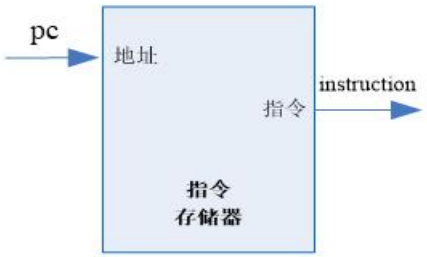

```
module pc(pc,clock,reset,npc);
output reg [31:0] pc;
input clock;
input reset;
input [31:0] npc;

always @(posedge clock,negedge reset) begin
    if(~reset)
    begin
      pc<=32'h0000_3000;
      //npc<=32'h0000_3000;
    end
    else
    begin
      pc<=npc;
    end
end
endmodule
```

clock 上升沿有效；

reset 低电平有效，异步复位

异步复位:它是指无论时钟沿是否到来，只要复位信号有效，就对系统进行复位，复位为：32'h0000_3000

（2）IM模块

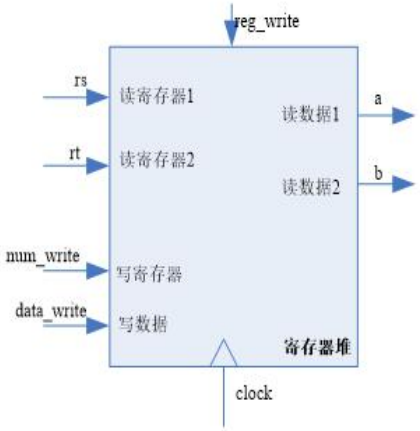



```
module im(instruction,pc);
output [31:0] instruction;
input [31:0] pc;
reg [31:0] ins_memory[1023:0]; //4k指令

assign instruction=ins_memory[pc[11:0]>

endmodule
```

采用组合逻辑,通过 assign 连续赋值

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>>2

（3）gpr 模块




```
module gpr(a,b,clock,reg_write,num_write,rs,rt,data_write);
output  [31:0] a;
output  [31:0] b;
input clock;
input reg_write;
input [4:0] rs; //读寄存器1
input [4:0] rt; //读寄存器2
input [4:0] num_write; //写寄存器
input [31:0] data_write; //写数据
reg [31:0] gp_registers[31:0];  //32个寄存器
always @(posedge clock) begin
    if(reg_write&&num_write!=5'b0)
    begin
        gp_registers[num_write]=data_write;
    end
end
   assign a=gp_registers[rs];
   assign b=gp_registers[rt];
endmodule
```

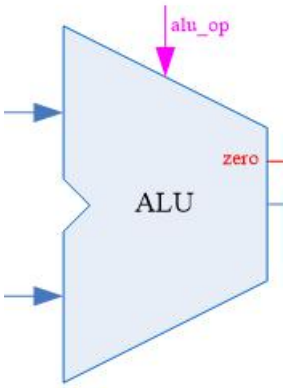
0 号寄存器读出的值永远是 0
reg_write 高电平有效

这是寄存器堆模块,顾名思义,可以通过这个模块读、写数据。

根据上图可知,reg_write 指示读信号,有 clock 即时序逻辑,每当 clock 上升沿时测试 reg_write 信号.

在 clock 的上升沿，如果 reg_write 有效，寄存器堆[num_write] = data_write

（4）ALU 模块



```
`include "include.v"
module alu(c,a,b,alu_ctrl,zero);
output  zero;
output reg[31:0] c;
input [31:0] a;
input [31:0] b;
input[5:0]alu_ctrl;

wire[31:0]addu_result,subu_result,add_result,and_result,or_result,slt_result,lui_result;

assign addu_result=a+b;

assign subu_result=a-b;

assign add_result=a+b;

assign and_result=a&b;

assign or_result=a|b;

assign slt_result=($signed(a)<$signed(b))?32'h1:32'h0;

assign lui_result={b[15:0],{16{1'b0}}};


assign zero=(a-b==0)?1'b0:1'b1;
always @* begin
        case (alu_ctrl)
                `add_op:c=add_result;
                `subu_op:c=subu_result;
                `addu_op:c=addu_result;
                `and_op:c=and_result;
                `or_op:c=or_result;
                `slt_op:c=slt_result;
                `lui_op:c=lui_result;
               default: c=0;
        endcase
end

endmodule
```

需要注意的是,虽然是 addu 无符号数相加

“无符号”是一个误导，其本意是不考虑溢出。

但无符号与有符号相加差别是是否溢出

mips 并不判断,如果不考虑溢出，则 add 与 addu 等价。

所以根本不用声明为 unsigned,这样反而会错

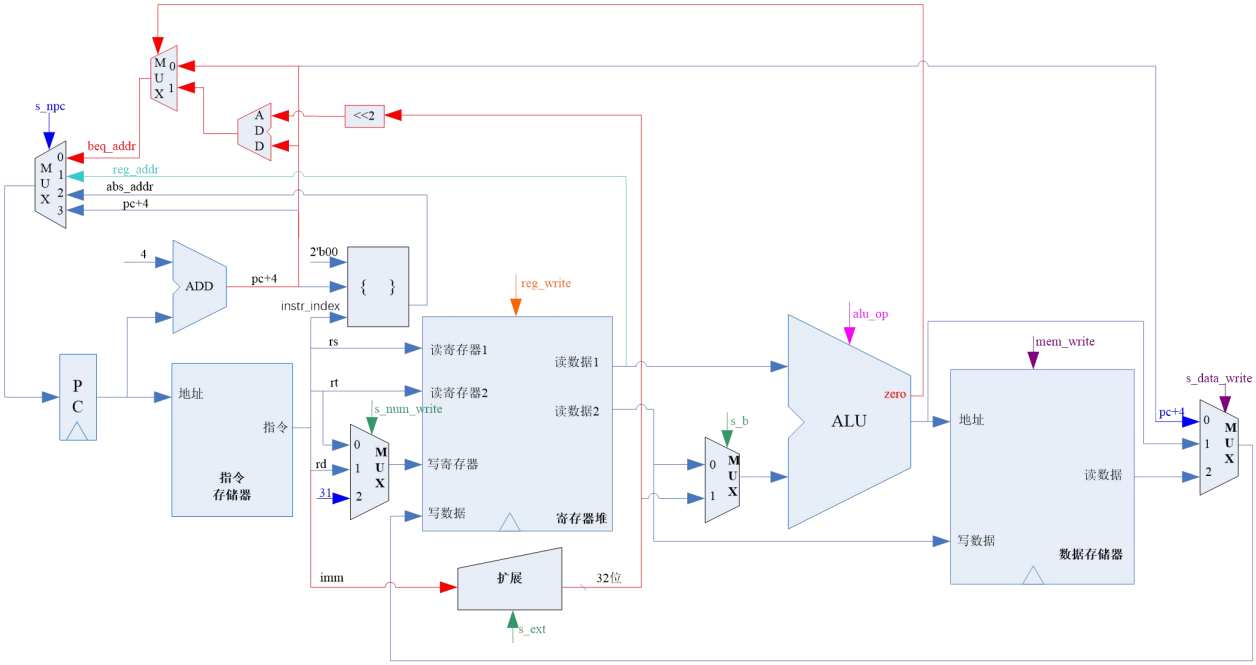
通过 ctrl 模块对 aluop 进行赋值，选择不同的运算，加减与或，slt

zero 用于判断跳转指令中的 beq

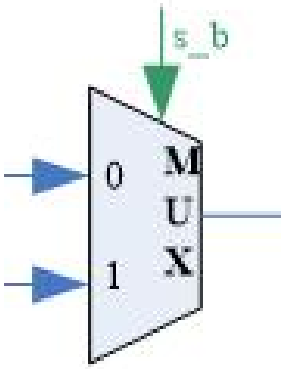
（5）ctrl 模块（控制模块）

这里对各种模块进行控制，控制各个多路选择器，还有控制存储器，寄存器，读写，ALU 进行哪种操作，符号扩展还是 0 扩展等等，相当于一个指挥部

说明：先通过 op 字段来判断，若 op 为 000000，则可能是 R 型指令或者寄存器跳转
总之通过 op 字段和 funct 字段，可以唯一确定指令是哪种类型，并且也可以精确到执行哪个操作，
然后根据下面的数据通路对每个控制端口
s_npc,reg_write,s_num_write,s_ext,s_b,aluop,mem_write,s_data_write 进行赋值



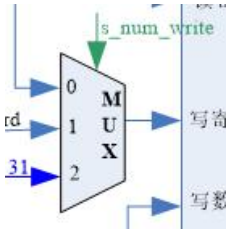
（6）Mux2to1 模块（也就是选择 ALU 第二个操作数的模块，一个二选一选择器）




```
module mux2to1_32(
    input[31:0] num1,
    input[31:0]  num2,
    input sel,
    output[31:0]  result
);
assign result=(sel==1'b0)?num1:
                num2;

endmodule

module mux3to1_32(
    input[31:0] num1,
    input[31:0]  num2,
    input[31:0] num3,
    input[1:0] sel,
    output reg[31:0]  result
);
always@*begin
    case (sel)
        2'b00:result=num1;
        2'b01:result=num2;
        2'b10:result=num3;
        2'b11:result=32'b0;
    endcase
end
```

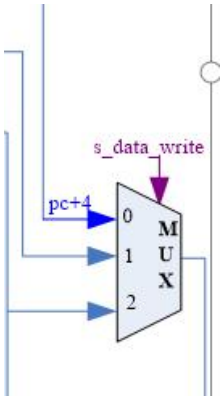
普通的 R 型指令和 beq 指令 ALU 的第二个输入应该是寄存器 rt 的内容，s_b=0
其他指令第二个输入是立即数或者 offset 经过符号扩展或者 0 扩展后的内容，s_b=1

(7)mux3to1 模块（也就是选择要往哪个寄存器进行写操作）



R 型指令要对 rd 寄存器进行写操作，num_write=1
jal 指令要对 31 号寄存器进行写操作
I 型指令和取数操作是对 rt 寄存器进行写操作

（8）datawrite 模块（选择往寄存器写的数据具体是什么）

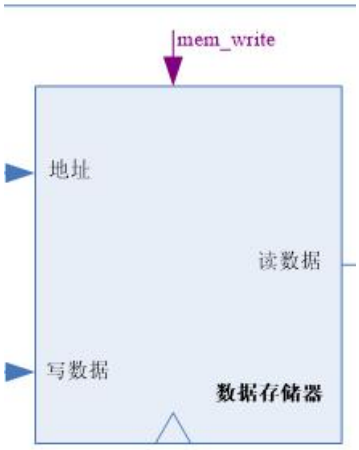


沿用之前的通用多选器模块

R 型，I 型指令都是将 ALU 的运算结果写入寄存器，取数操作是将存储器的值放入寄存器中

Jal 是将 PC+4 放入 31 号寄存器

（9）dm 模块

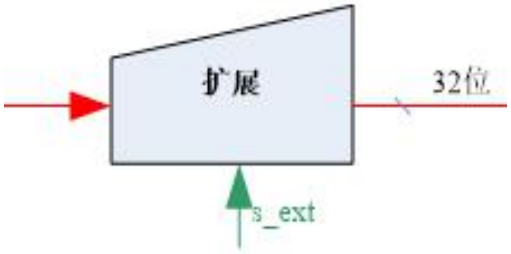



```
module dm(data_out,clock,mem_write,address,data_in);
output [31:0] data_out;
input clock;
input mem_write;
input [31:0] address;
input [31:0] data_in;
reg [31:0] data_memory[1023:0];
assign data_out = data_memory[address[11:2]];
    always@(posedge clock )
        if(mem_write) data_memory[address[11:2]]<= data_in;

endmodule
```

数据存储器模块的功能是建立一个 4kB 大小的存储器以及读写存储器。
对于读操作，data_out 一直输出 address 地址处的值。
对于写操作，在 clock 上升沿且 mem_write 信号有效时 data_in 被写入 address 地址处。

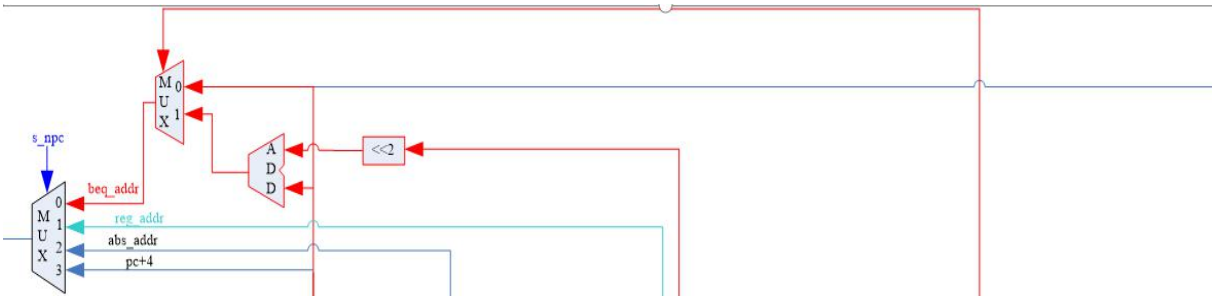
（10）ext 模块（扩展模块）

```
module ext(
    input s_ext,
    input [15:0] Imm16,
    output  reg [31:0] Imm32
);

always@(*)
Imm32=s_ext? {{16{Imm16[15]}},Imm16}:{{16{1'b0}},Imm16};

endmodule
```



s_ext 为 0 时进行 0 扩展，为 1 时进行符号扩展

（11）npc 模块（得到下一个 pc 值）



Beq 指令 s_npc 为 00

Jr 指令，寄存器跳转 s_npc 为 01

j 和 jal 指令 s_npc 为 10

其他指令都是正常的+4

```
wire[1:0]s_npc;
assign abs_addr={pc[31:28],instruction[25:0],2'b0};
assign reg_addr=a;
assign beq_pc={{16{instruction[15]}},instruction[15:0]<<2}+pc+4;
pc PC(
.pc(pc),
.clock(clock),
.reset(reset),
.npc(npc)
);
mux2to1_32 beq2to1(
    .num1(beq_pc),
    .num2(pc+4),
    .sel(zero),
    .result(beq_addr)
);
```

2. s_cycle_cpu 模块（顶层模块）

见源码

3. 指令分析与验证

（1）R 型指令

位段	31 26	25 21	20 16	15 11	10 6	5 0
编码	op(6位)	rs(5位)	rt(5位)	rd(5位)	shamt(5位)	funct(6位)
addu rd, rs, rt GPR[rd]←GPR[rs] + GPR[rt]	000000	rs	rt	rd	00000	100001
subu rd, rs, rt GPR[rd]←GPR[rs] - GPR[rt]	000000	rs	rt	rd	00000	100011
add rd, rs, rt （不考虑溢出） GPR[rd]←GPR[rs] + GPR[rt]	000000	rs	rt	rd	00000	100000
and rd, rs, rt GPR[rd]←GPR[rs] & GPR[rt]	000000	rs	rt	rd	00000	100100
or rd, rs, rt GPR[rd]←GPR[rs] | GPR[rt]	000000	rs	rt	rd	00000	100101
slt rd, rs, rt （有符号数比较） GPR[rd]←GPR[rs] < GPR[rt]	000000	rs	rt	rd	00000	101010

测试代码

见 testbench

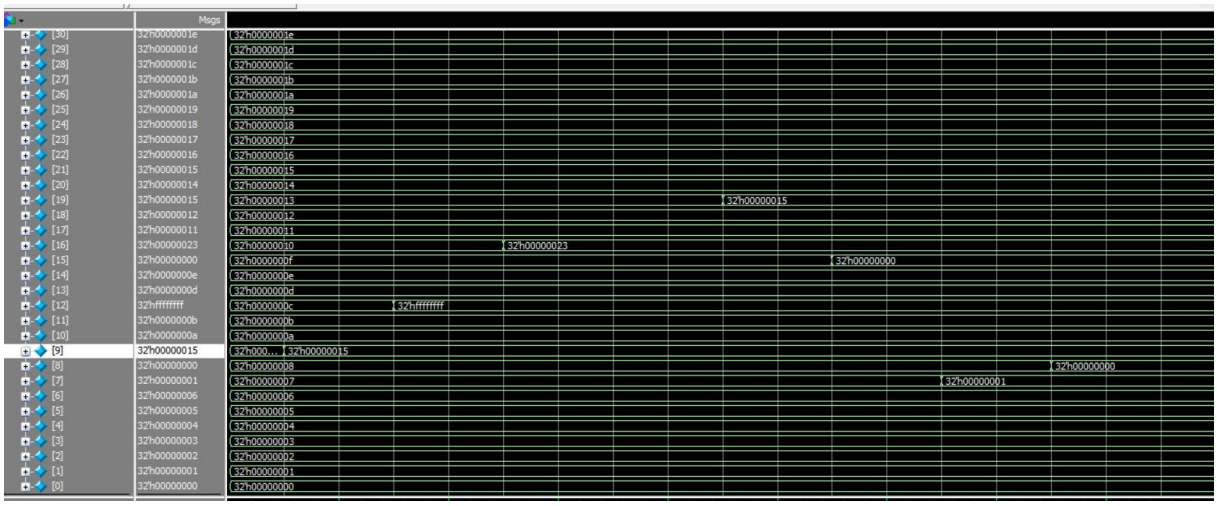
```
addi $v0,$v1,0x00003004
and $t1,$t1,$t1
and $t1,$t1,$t1
and $t1,$t1,$t1
and $t1,$t1,$t1
test1:
addi $t1,$t1,100
j test2
subi $t1,$t1,100
test2:
#jal test1
addi $t2,$t3,100
and $s1,$s1,$s1
and $s1,$s1,$s1
and $s1,$s1,$s1
and $s1,$s1,$s1
#jr $v0
beq $t6,$t7,end
and $s1,$s1,$s1
and $s1,$s1,$s1
and $s1,$s1,$s1
and $s1,$s1,$s1
end:
and $s1,$s1,$s1
and $s1,$s1,$s1
```

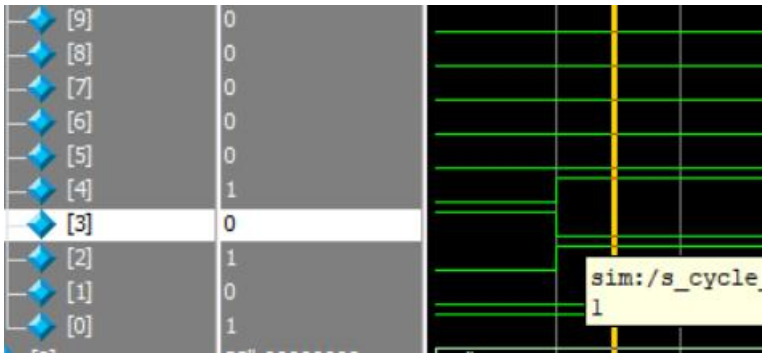
存取指令

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
014b4821    //将10号和11号寄存器中的值add后存入9号寄存器中
01ae6023    //将13号和14号寄存器中的值add后存入12号寄存器中
02328020
00a62024
02959825
0135782a
0043382a
0062402a
```

波形分析





在第一个时钟周期后，9 号寄存器的值变为了 21（10+11），与预期相符合

```
# gp_registers[ 0] = 00000000
# gp_registers[ 1] = 00000001
# gp_registers[ 2] = 00000002
# gp_registers[ 3] = 00000003
# gp_registers[ 4] = 00000004
# gp_registers[ 5] = 00000005
# gp_registers[ 6] = 00000006
# gp_registers[ 7] = 00000001
# gp_registers[ 8] = 00000000
# gp_registers[ 9] = 00000015
# gp_registers[10] = 0000000a
# gp_registers[11] = 0000000b
# gp_registers[12] = ffffffff
# gp_registers[13] = 0000000d
# gp_registers[14] = 0000000e
# gp_registers[15] = 00000000
# gp_registers[16] = 00000023
# gp_registers[17] = 00000011
# gp_registers[18] = 00000012
# gp_registers[19] = 00000015
# gp_registers[20] = 00000014
# gp_registers[21] = 00000015
# gp_registers[22] = 00000016
# gp_registers[23] = 00000017
# gp_registers[24] = 00000018
# gp_registers[25] = 00000019
# gp_registers[26] = 0000001a
# gp_registers[27] = 0000001b
# gp_registers[28] = 0000001c
# gp_registers[29] = 0000001d
# gp_registers[30] = 0000001e
# gp_registers[31] = 0000001f
```

（2）I 型指令

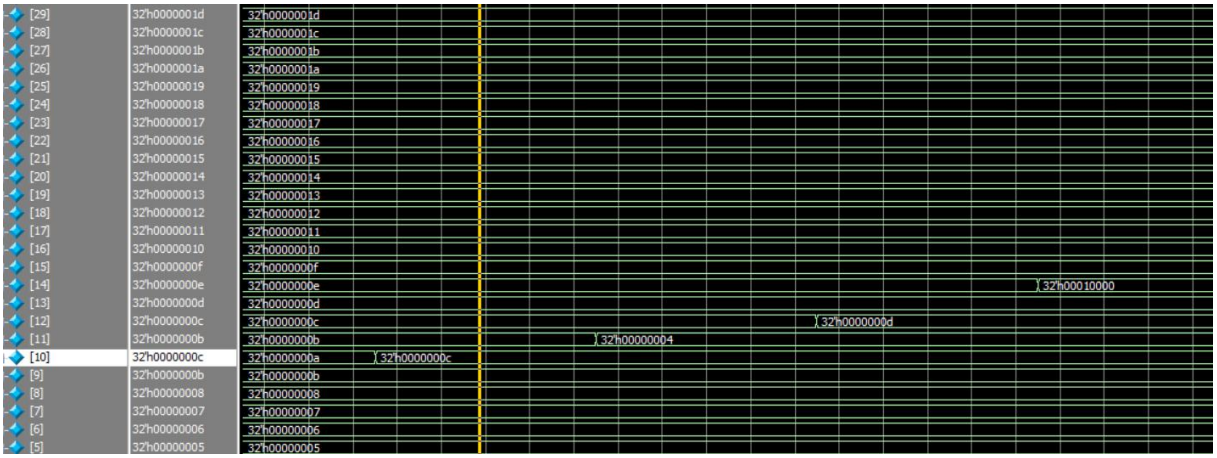
指令格式	描述	31 26	25 21	20 16	15 0
		op(6位)	rs(5位)	rt(5位)	imm(16位)
addi rt, rs, imm	GPR[rt] ←GPR[rs] + sign_extend(imm)	001000	rs	rt	imm
addiu rt, rs, imm	GPR[rt] ←GPR[rs] + sign_extend(imm)	001001	rs	rt	imm
andi rt, rs, imm	GPR[rt] ←GPR[rs] & zero_extend(imm)	001100	rs	rt	imm
ori rt, rs, imm	GPR[rt] ←GPR[rs] | zero_extend(imm)	001101	rs	rt	imm
lui rt, imm	GPR[rt]←{imm, 0^{16} }	001111	00000	rt	imm

测试文件同上

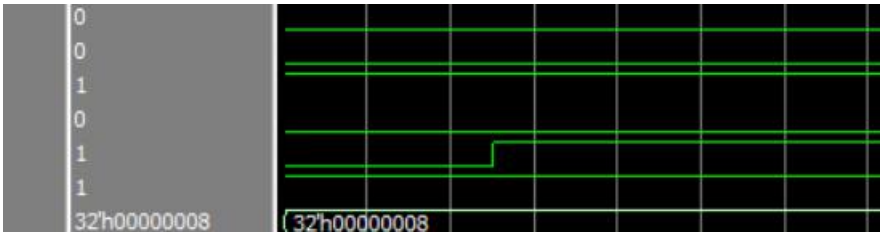
具体指令如下

Code	Basic	
0x21490001	addi $9, $10, 0x00000001	1: addi $t1 $t2, 1
0x256a0001	addiu $10, $11, 0x00000001	2: addiu $t2 $t3, 1
0x318b0064	andi $11, $12, 0x00000064	3: andi $t3 $t4, 100
0x35ac0001	ori $12, $13, 0x00000001	4: ori $t4, $t5, 1
0x3c0e0001	lui $14, 0x00000001	5: lui $t6, 1

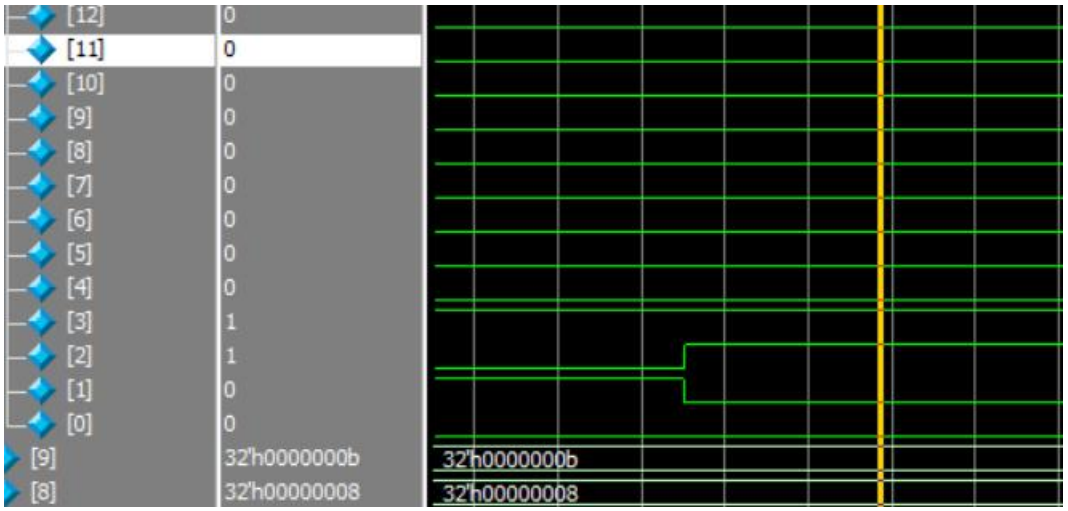
波形分析



第一条指令将 9 号寄存器的值变为 10 号+1，也就是 11 如下



第二条指令将 10 号寄存器的值变为 11 号+1，也就是 12 如下



最终结果如下

```
VSIM 4> run -all
# gp_registers[ 0] = 00000000
# gp_registers[ 1] = 00000001
# gp_registers[ 2] = 00000002
# gp_registers[ 3] = 00000003
# gp_registers[ 4] = 00000004
# gp_registers[ 5] = 00000005
# gp_registers[ 6] = 00000006
# gp_registers[ 7] = 00000007
# gp_registers[ 8] = 00000008
# gp_registers[ 9] = 0000000b
# gp_registers[10] = 0000000c
# gp_registers[11] = 00000004
# gp_registers[12] = 0000000d
# gp_registers[13] = 0000000d
# gp_registers[14] = 00010000
# gp_registers[15] = 0000000f
# gp_registers[16] = 00000010
# gp_registers[17] = 00000011
# gp_registers[18] = 00000012
# gp_registers[19] = 00000013
# gp_registers[20] = 00000014
# gp_registers[21] = 00000015
# gp_registers[22] = 00000016
# gp_registers[23] = 00000017
# gp_registers[24] = 00000018
# gp_registers[25] = 00000019
# gp_registers[26] = 0000001a
# gp_registers[27] = 0000001b
# gp_registers[28] = 0000001c
# gp_registers[29] = 0000001d
# gp_registers[30] = 0000001e
# gp_registers[31] = 0000001f
# ** Note: $stop    : C:/Users/Administrator/Desktop/cpu/s_cycle_cpu_test.v(28)
#    Time: 1050 ns  Iteration: 0  Instance: /s_cycle_cpu_test
# Break in Module s_cycle_cpu_test at C:/Users/Administrator/Desktop/cpu/s_cycle
```

（3）MEM 型指令和跳转指令

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

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

测试代码

三、 遇到的问题和解决方法：

（1）在第一次实验时，没有把所有模块打包，以为直接提交顶层模块可以利用前四次提交的基本模块

JSON格式错误!
Traceback (most recent call last): File "/kernel/kernel.zip/run.py", line 55, in run return _run(job, testcase) File "/kernel/kernel.zip/run.py", line 69, in _run submit_temp, submit_result = config['runner'](job, tb, submit) TypeError: cannot unpack non-iterable NoneType object {"verdict": "Runtime Error", "score": "0", "rank": {"rank": "-1"}, "HTML": "enable", "detail": "RUN INFORMATION\n** Warning: setting ADDR_NO_RANDOMIZE failed - Success.\nError loading design\n\nReading pref.tcl\n\n# 2020.1\n\n# vsim -c -do \"run -all\" s_cycle_cpu_test \n# Start time: 12:59:38 on May 30,2022\n# Loading work.s_cycle_cpu_test\n# Loading work.s_cycle_cpu\n# ** Error: (vsim-3033) Instantiation of 'pc' failed. The design unit was not found.\n# Time: 0 ps Iteration: 0 Instance: /s_cycle_cpu_test/S_CYCLE_CPU File: addu.v Line: 11\n# Searched libraries:\n# /tmp/tmp7mr0ik_q/work\n# ** Error: (vsim-3033) Instantiation of 'im' failed. The design unit was not found.\n# Time: 0 ps Iteration: 0 Instance: /s_cycle_cpu_test/S_CYCLE_CPU File: addu.v Line: 15\n# Searched libraries:\n# /tmp/tmp7mr0ik_q/work\n# ** Error: (vsim-3033) Instantiation of 'gpr' failed. The design unit was not found.\n# Time: 0 ps Iteration: 0 Instance: /s_cycle_cpu_test/S_CYCLE_CPU File: addu.v Line: 29\n# Searched libraries:\n# /tmp/tmp7mr0ik_q/work\n# ** Error: (vsim-3033) Instantiation of 'alu' failed. The design unit was not found.\n# Time: 0 ps Iteration: 0 Instance: /s_cycle_cpu_test/S_CYCLE_CPU File: addu.v Line: 31\n# Searched libraries:\n# /tmp/tmp7mr0ik_q/work\n# Error loading design\n# End time: 12:59:38 on May 30,2022, Elapsed time: 0:00:00\n# Errors: 4, Warnings: 0\n\n"}

后来把这段错误翻译了之后发现实例化失败，后来意识到应该一起打包的

（2）在进行第二个实验的时候，遇到了超时问题（死循环）

得分0.00 最后一次提交时间:2022-05-30 22:13:46
运行时间过长!

```
6'b000011:
 begin
reg_write=1; s_npc=2'b10;s_data_write=2'b00;s_num_write=2'b10;
```

后来看着指令格式对每条指令逐条分析，发现寄存器跳转（000011）没有将 num_write 值赋为正确值，也就是说没有将 PC+4 写入 31 号寄存器中，在 beq 指令的控制指令赋值的时候，要仔细地分析好各个控制信号的要求，我因为想当然的认为 alu 的两个输入源操作数就是两个寄存器，导致 zero 赋值出错，导致反复调了很长时间。

（3）还有就是比如跳转指令用不到数据存储器之类的模块，就没有对控制端口进行赋值，以为这样就不会进行写或者读操作了，导致发生了不必要的错误

```
# GPR.rs = 00, GPR.rt = 00, GPR.a = 00000000, GPR.b = 00000000
# DM.mem_write = 1, DM.address = 00000c15, DM.data_in = 00000000
# GPR.regwrite = 0
# ------------------------------------
# time = 4960
# CLOCK = 1, RESET = 1, pc = 00003054, instruction = 08000c15
# GPR.rs = 00, GPR.rt = 00, GPR.a = 00000000, GPR.b = 00000000
# DM.mem_write = 1, DM.address = 00000c15, DM.data_in = 00000000
# GPR.regwrite = 0
# ------------------------------------
# time = 4970
```

```
# GPR.rs = 00, GPR.rt = 00, GI
# DM.mem_write = 0
# GPR.regwrite = 0
# ------------------------------
# time = 4960
# CLOCK = 1, RESET = 1, pc =
# GPR.rs = 00, GPR.rt = 00, GI
# DM.mem_write = 0
# GPR.regwrite = 0
# ------------------------------
# time = 4970
```

前后比对发现，如果不重新赋值的话，该变量还会保留执行上条指令时的值

后来对 ctrl 模块对每种指令的各个端口都进行了赋值

（4）还有语言方面的错误，这种错误贯穿在第一次和第二次实验中，比如阻塞赋值和非阻塞赋值，reg，assign，wire 语句运用的不规范，导致在实验的过程中发生了很多很多莫名其妙的错误，虽然是一个小错误，但在找的时候就像大海捞针一样，很费时间，慢慢的，通过 CSDN 查各种语句的用法，后来这类型的错误就慢慢减少了

四、 实验总结：

通过这两次实验课的学习,更加深刻理解了 CPU 各个模块的构建,以及每个模块之间的联系，与理论学习相辅相成，进一步理解了指令是如何执行的

同时对 verliog 更加掌握了，对于 assign，wire，reg 之类的判断也更加清晰了