单周期 CPU 电路设计实验报告

一、 实验目的

a) 实现指定 MIPS 指令的单周期 CPU 电路设计。

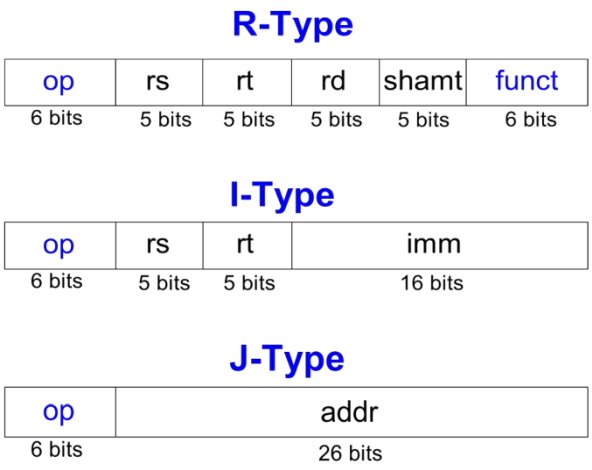
b) 支持的指令：

i. R-type(3 寄存器): add, sub, or, slt,

ii. i-type(2 寄存器，16 位立即数): addi, ori, slti, sw, lw

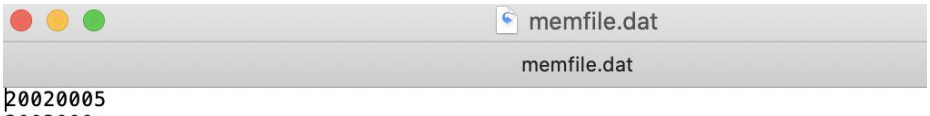
iii. j-type(26 位立即数): j

iv. 空操作: nop



二、 指令设计

a) 根据条件设计了一个寄存器读取指令的文件(memfile.dat)



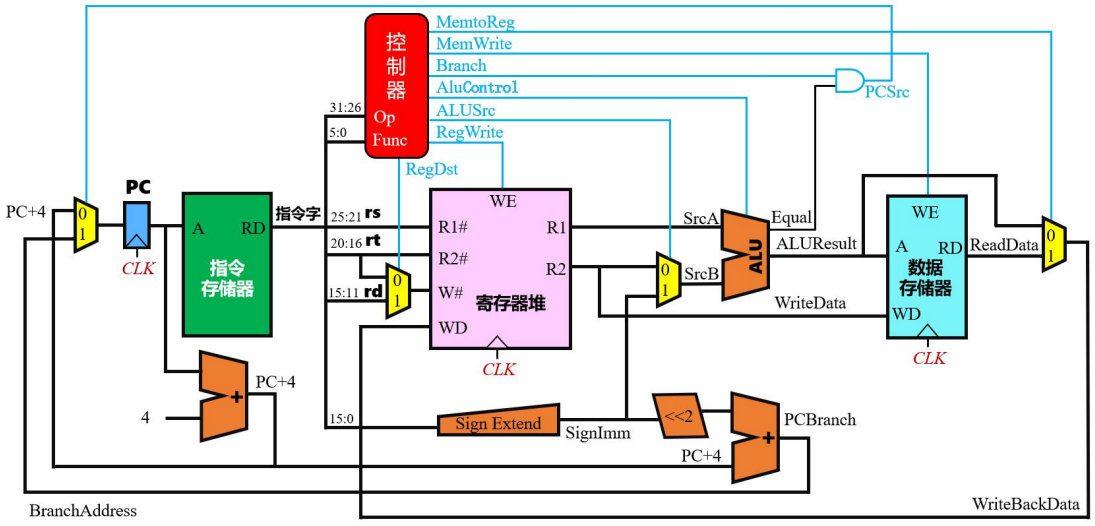

```
20020005
2003000c
2067fff7
30450007
34a80001
28a80004
00e22025
00642824
00a42820
10a7000a
0064202a
10800001
20050000
00e2202a
00853820
00e23822
ac670044
8c020050
08000014
20020001
ac020054
00000000
```

b) 一共 21 条指令，根据每条指令的 8 位 16 进制编码，解码和相应解释如下图。自己对照 mips 指令表做了一个易于观察的表格。

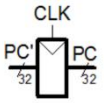
				rs,rt,imme/rs,rt,rd			address
main	20020005	001000 00000 00010 00000 00000 000101	addi	rt = rs+ imme	$2, $0, 5	initialize $2 = 5	0
	2003000c	001000 00000 00011 00000 00000 001100	addi	rt = rs+ imme	$3, $0, 12	initialize $3 = 12	4
	2067fff7	001000 00011 00111 11111 11111 110111	addi	rt = rs+ imme	$7, $3, -9	initialize $7 = 3	8
	30450007	001100 00010 00101 00000 00000 000111	andi	rt = rs and imme	$5, $2, 7	initialize $5 = 5	c
	34a80001	001101 00101 01000 00000 00000 000001	ori	rt = rs or imme	$8, $5, 1	initialize $8 = 5	10
	28a80004	001010 00101 01000 00000 00000 000100	slti	rt = (rs < imme)	$8, $5, 4	initialize $8 = 0	14
	00e22025	000000 00111 00010 00100 00000 100101	or	rd = rs or rt	$4, $7, $2	$4 <= 3 or 5 = 7	18
	00642824	000000 00011 00100 00101 00000 100100	and	rd = rs and rt	$5, $3, $4	$5 <= 12 and 7 = 4	1c
	00a42820	000000 00011 00100 00101 00000 100000	add	rd = rs + rt	$5, $5, $4	$5 = 4 + 7 = b	20
	10a7000a	000100 00101 00111 00000 00000 001010	beq	if rs=rt then branch	$5, $7, end	shouldn't be taken	24
	0064202a	000000 00011 00100 00100 00000 101010	slt	rd = (rs<rt)	$4, $3, $4	$4 = 12 < 7 = 0	28
	10800001	000100 00100 00000 00000 00000 000001	beq	if rs=rt then branch	$4, $0, around	should be taken	2c
	20050000	001000 00000 00101 00000 00000 000000	addi	rt = rs+ imme	$5, $0, 0	shouldn't happen	30
around	00e2202a	000000 00111 00010 00100 00000 101010	slt	rd = (rs<rt)	$4, $7, $2	$4 = 3 < 5 = 1	34
	00853820	000000 00100 00101 00111 00000 100000	add	rd = rs + rt	$7, $4, $5	$7 = 1 + 11 = c	38
	00e23822	000000 00111 00010 00111 00000 100010	sub	rd = rs - rt	$7, $7, $2	$7 = 12 - 5 = 7	3c
	ac670044	101011 00011 00111 00000 00001 000100	sw	[$base+offset]=rt	$7, 68($3)	[80] = 7	40
	8c020050	100011 00000 00010 00000 00001 010000	lw	rt=[$base+offset]	$2, 80($0)	$2 = [80] = 7	44
	08000014	000010 00000 00000 00000 00000 010001	j		end	should be taken	48
	20020001	001000 00000 00010 00000 00000 000001	addi	rt = rs+ imme	$2, $0, 1	shouldn't happen	4c
end	ac020054	101011 00000 00010 00000 00001 010100	sw	[$base+offset]=rt	$2, 84($0)	write adr 84 = 7	50

i. 前 6 条指令都为初始化， addi,addi,addi,andi,ori,slti。
ii. or, and,add
iii. 第一个 beq，由于$5 不等于$7，所以 branch 不会执行
iv. slt
v. 第二个 beq， $4=$0，执行 around 语句。 跳过还在其后的 addi。
vi. slt, add, sub,
vii. sw，在地址为 80 的地方写入数据 7
viii. lw，从地址为 80 的地方给$2 赋值 7
ix. jump 指令的地址要根据每条地址实际的指令去调整，在这里设为 jump 下面第二条指令，即跳过最后一个 addi 指令，到 end。
x. sw，最终在地址 84 的位置写入数据 7

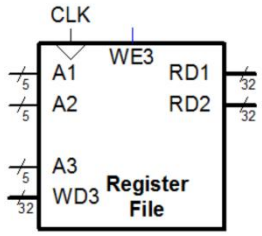
三、模块概述



a) flopr 程序计数器
i. 输出 PC 地址



b) regfile 寄存器



i.

ii. 输入时钟和写使能，读取数据的两个地址，写入的地址，写入的数据，输出读出的两个数据

iii. 根据地址在寄存器中读取，输出读出的数 rd1，rd2

iv. 在写使能的情况下，将输入的写数据写入写地址对应的位置。

```
1  module regfile(
2      input clk,
3      input we,  // write enable
4      input [4:0] ra1, ra2,
5      input [4:0]wa,  // write address
6      input [31:0] wd,  //write data
7      output [31:0] rd1, rd2
8      );
9
10     reg [31:0] rf[31:0];
11
12     always@(posedge clk)
13        if(we)
14           begin
15              rf[wa] <= wd;
16              $display("R%d = %h", wa, wd);
17           end
18     assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
19     assign rd2 = (ra2 != 0) ? rf[ra2]: 0;
20 endmodule
```

c) alu 运算单元

i. 进行逻辑和算数运算，输入 3 位控制符

```
1  module alu(
2      input [31:0] a,b,
3      input [2:0] control,
4      output reg [31:0] result,
5      output reg zero
6      );
```

```
7     always@(*)
8         begin
9           case(control)
10             3'b000: result <= a&b; // and
11             3'b001: result <= a|b;  //or
12             3'b010: begin
13                        result <=  a+b; // add
14                        if(result[31] != a[31] && result[31] != b[31])
15                          $display("add overflow");
16                   end
17             3'b011: result <= b << a;  // sll
18             3'b100: result <= b >>a;  // srl
19             3'b101: result <= a|~b;  // xor
20             3'b110: begin
21                        result <= a-b;  //sub
22                        if((a[31] != b[31] && result[31]==b[31]) || (a!=b
&& result ==0))
23                          $display("sub overflow");
24                   end
25             3'b111: case({a[31],b[31]})  //slt
26                        2'b00: result <= (a<b) ? 1:0;//both positive
27                        2'b01: result <=0; // a is positive, b is negative
28                        2'b10: result <=1;
29                        2'b11: result <= (a>b)? 1:0; // both negative
30                    endcase
31          endcase
32          zero = (result == 0)? 1:0;
33       end
34 endmodule
```

d) signext 符号扩展

i. 

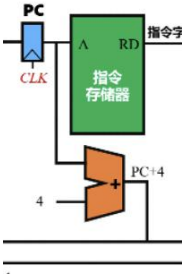
ii. 输入 16 位，输出扩展后的 32 位

iii. 对于 signextsignal，如果是 andi 则为 11，如果是 ori 则为 10，如果都不是则为 00，即把 a 的最高位复制到新的 16 个高位上。

```
1  module signext(
2     input [15:0] a,
3     input [1:0] signextsignal,
4     output reg [31:0] y
5     );
6     always@(*)
7        case(signextsignal)
8           2'b00: y = {{16{a[15]}}, a};
```

```
9           2'b10: y = {16'b0, a};
10          2'b11: y = {16'b1111111111111111, a};
11          default: y = {{16{a[15]}},a};
12      endcase
13 endmodule
```

e) adder 加法器



i. 确定 PC 下一条指令地址

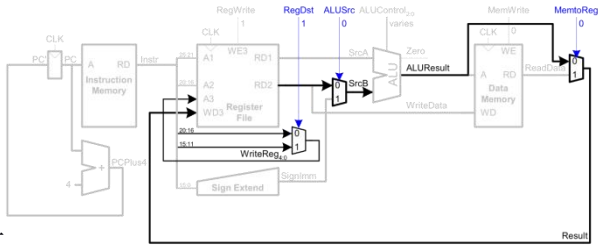
ii. 输入 a,b,输出 a+b

f) mux2, 2-1 复用器

i. 可指定位宽

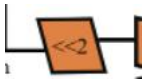
ii. 若输入 s=1, 则输出第二个输入值，否则输出第一个输入值

iii. 图中[0,1]的地方




```
1 module mux2 #(parameter WIDTH=8)
2     (input [WIDTH-1:0] d0, d1,
3     input s,
4     output [WIDTH-1:0] y
5     );
6     assign y = s ? d1 : d0;
7 endmodule
```

g) sl2，shift left 2 bit

i. 


```
1 module sl2(
2     input [31:0] a,
3     output [31:0] y
4     );
5     // move two to the left, same as multiplying by 4\
6     assign y = {a[29:0], 2'b00};
7 endmodule
```

h) instMem 指令存储器

i. 从文件读取指令列表并储存在存储器中，从指令存储器中读取一条指令

ii. 输出指令 32 位 instr。

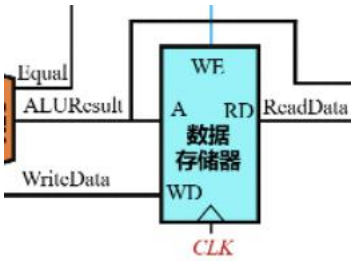
iii. 文件读取要用绝对路径

```
module instMem(
    input [5:0] a,
    output [31:0] instr
    );
    reg [31:0] instRAM[63:0]; // no more than 16 instructions, 32*64

    initial begin
        $readmemh("C:/Users/lenovo/Desktop/ice_pj/memfile.dat",
instRAM);
    end
    assign instr = instRAM[a];
endmodule
```

i) dataMem

i. 把数据写入数据存储器

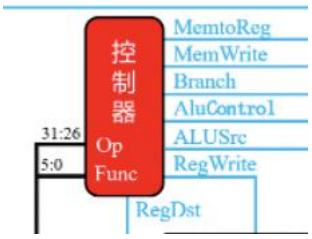


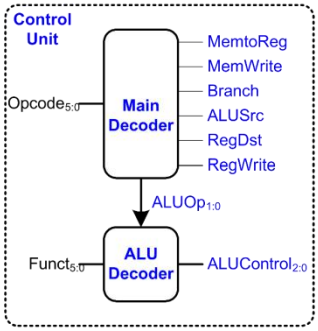

```
module dataMem(
    input clk, we,
    input [31:0] dataadr, writedata,
    output [31:0] readdata
    );

    reg [31:0] dataRAM[63:0];
    assign readdata = dataRAM[dataadr[31:2]]; // word aligned

    always @(posedge clk)
        if(we) dataRAM[dataadr[31:2]] <= writedata;
endmodule
```

j) controller 控制单元





i. maindec 主译码器

1. 从指令的前 6 位译码，输出 MemtoReg, MemWrite, Branch, AluSrc, RegDst, RegWrite 和 两位 AluOP 用于进行进一步的 alu 译码

```
module maindec(
    input [5:0]op,funct,
    output memtoreg, memwrite,
    output wire branch, // if don't set wire, there will be 'Z' in simulation
    output alusrc,
    output regdst, regwrite,
    output jump,
    output [1:0] aluop,
    output [1:0] signextsignal
    );
    reg [8:0] controls;

    assign {regwrite, regdst, alusrc, branch, memwrite, memtoreg,jump, aluop} = controls;

    always@(*)
        case(op)
            6'b000000: case(funct)
                        6'b000000: begin
                                    controls <= 9'b000000010;
                                    $display("Controller nop");
                                   end
                        default:   controls <= 9'b110000010;  //Rtype
                       endcase
            6'b100011: controls <= 9'b101001000; //lw
            6'b101011: controls <= 9'b001010000; //sw
            6'b000100: controls <= 9'b000100001;  //beq
            6'b001000: controls <= 9'b101000000; // addi
            6'b001100: controls <= 9'b101000011;  //andi
            6'b001101: controls <= 9'b101000011;  // ori
            6'b001010: controls <= 9'b101000011;  //slti
            6'b000010: controls <= 9'b000000100;  //j
            default: controls <=9'bxxxxxxxx; //???
```

```
33        endcase
34        assign signextsignal = (op == 6'b001100)? 2'b11:
((op==6'b001101)?2'b10:2'b00);
35 endmodule
```

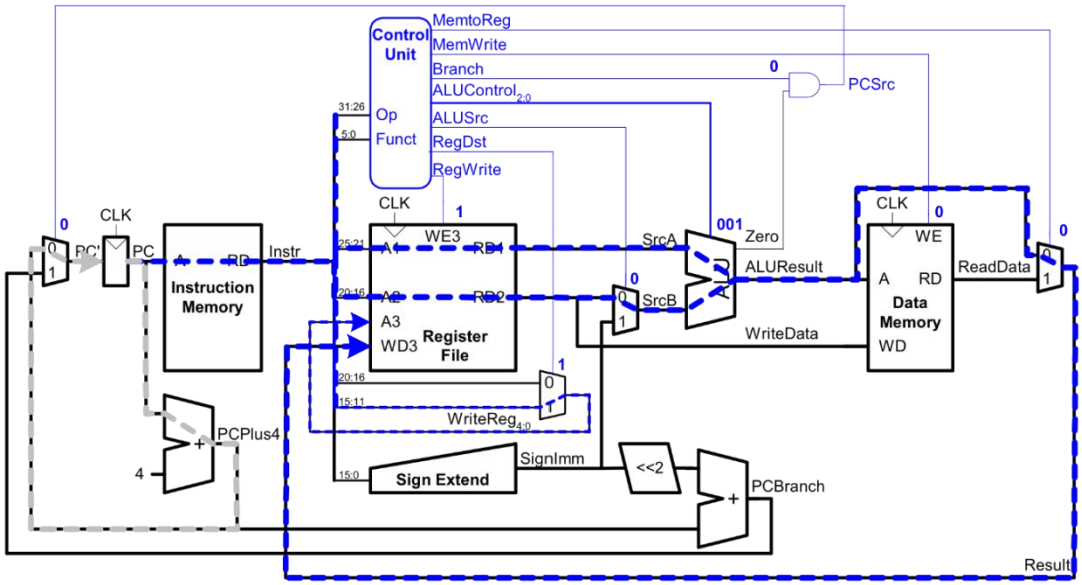
i. aludec alu 译码器

1. 根据 instr 指令的前 6 位和后 6 位，和从主译码器得来的 aluop，输出 3 位的 alucontrol 用于进行 alu 运算

```
1  module aludec(
2      input [5:0] op,funct, // instr[31:26], instr[5:0]
3      input [1:0] aluop,
4      output reg [2:0] alucontrol
5      );
6      always@(*)
7          case(aluop)
8              2'b00: begin
9                     alucontrol <= 3'b010; //add  (for sw, lw addi)
10                    $display("sw/lw/addi");
11                   end
12             2'b01: begin
13                    alucontrol <= 3'b110; // sub  (for beq)
14                    $display("BEQ");
15                   end
16             default:
17                 case(op)
18                    6'b001101: alucontrol <= 3'b001; // ORI
19                    6'b001100: alucontrol <=3'b000; // ANDI
20                    6'b001010: alucontrol <= 3'b111; //SLTI
21
22                    default:
23                       case(funct) // r-type
24                          6'b100000: alucontrol <= 3'b010; //add
25                          6'b100010: alucontrol <= 3'b110;  //sub
26                          6'b100100: alucontrol <= 3'b000;  //and
27                          6'b100101: alucontrol <= 3'b001; //or
28                          6'b101010: alucontrol<=3'b111; //slt
29
30                          default: alucontrol <= 3'bxxx; //???
31                       endcase
32             endcase
33         endcase
34 endmodule
```

k) datapath **执行和连接各个模块**

35 输出 pc 执行地址，alu 运算后的结果 aluout，传到 mips 作为 dataadr 写入存储句的数据的地址，还输出写入数据 writedata。




```
// next pc signal
    flopr #(32) pcreg(clk, reset, pcnext, pc); // input pcnext, output pc = pcnext

    // pcplus4  = pc + 4
    adder      pcadd1(pc, 32'b100, pcplus4);

    // sign extend, signimm = {{16{instr[15]}, instr[15:0]}
    signext    se(instr[15:0], signextsignal, signimm);

    // shift left, signimmsh = signimm << 2
    sl2        immsh(signimm, signimmsh);

    // signimm shift left result + pcplus4 = pcbranch
    adder      pcadd2(pcplus4, signimmsh, pcbranch);

    //beq    if(rs == rt)  pcnextbr = pc+4 + (signimm<<2), else =pc+4
    mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc, pcnextbr);
    //jump
    mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28], instr[25:0], 2'b00},
                      jump, pcnext);

    regfile    rf(clk, regwrite, instr[25:21], instr[20:16],
                  writereg, result, srca, writedata);  // result correspond to wd(writedata)

    //write reg number
    mux2 #(5)   wrmux(instr[20:16], instr[15:11], regdst, writereg);
```

```
27
28    // if (memtoreg) result = readdata, else = aluout
29    mux2 #(32)  resmux(aluout, readdata, memtoreg, result);
30
31    // if(alusrc)   srcb = signimm, else = writedata
32    mux2 #(32)  srcbmux(writedata, signimm, alusrc, srcb);
33
34    alu        alu_(srca, srcb, alucontrol, aluout, zero);
```

l) mips mips **处理器**

i. 是 controller 和 datapath 的上层

ii. 输入时钟和重置信号符和指令 instr，以及读入的数据 readdata。

iii. readdata 在写 memToReg 为 1 时，作为寄存器的写入数据，否则写入 alu 运算后的结果

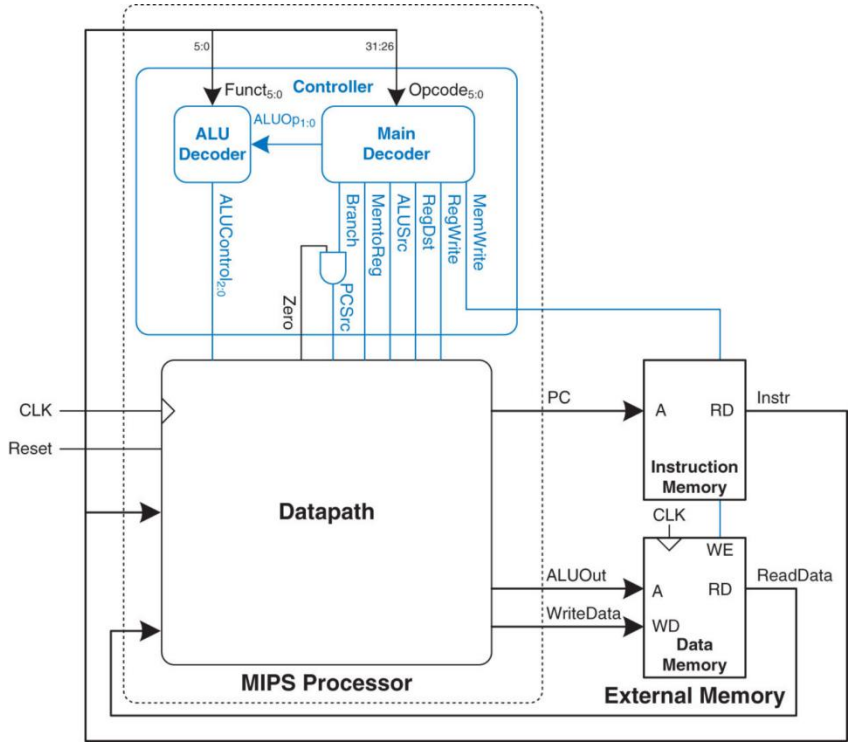
iv. 输出 memwrite，存储器写使能； aluout， writedata

```
1  module mips(
2     input clk, reset,
3     output [31:0] pc,
4     input  [31:0] instr,
5     output       memwrite,
6     output [31:0] aluout, writedata,
7     input  [31:0] readdata
8     );
9
10    wire  memtoreg, branch, pcsrc, zero, alusrc, regdst, regwrite, jump;
11    wire [1:0] signextsignal;
12    wire [2:0] alucontrol;
13
14    controller  c(instr[31:26], instr[5:0], zero,
15                  memtoreg, memwrite, pcsrc, alusrc,
16                  regdst, regwrite, jump, alucontrol, signextsignal);
17
18    datapath    dp(clk, reset, memtoreg, pcsrc, alusrc,
19                  regdst, regwrite, jump,
20                  alucontrol, zero, pc, instr,
21                  aluout, writedata, readdata, signextsignal);
22 endmodule
```

m) top **顶层文件**

i. 连接 mips 和 instMem 指令存储器和 dataMem 数据存储器

ii.




```
1  module top(
2      input clk, reset,
3      output [31:0] writedata, dataadr,
4      output memwrite,
5      output [31:0] pc, instr, readdata
6      );
7      //instantiate processor and memories
8      mips mips(clk, reset, pc, instr, memwrite, dataadr, writedata, readdata);
9
10     instMem imem(pc[7:2], instr);
11     dataMem dmem(clk, memwrite, dataadr, writedata, readdata);
12 endmodule
```

i.

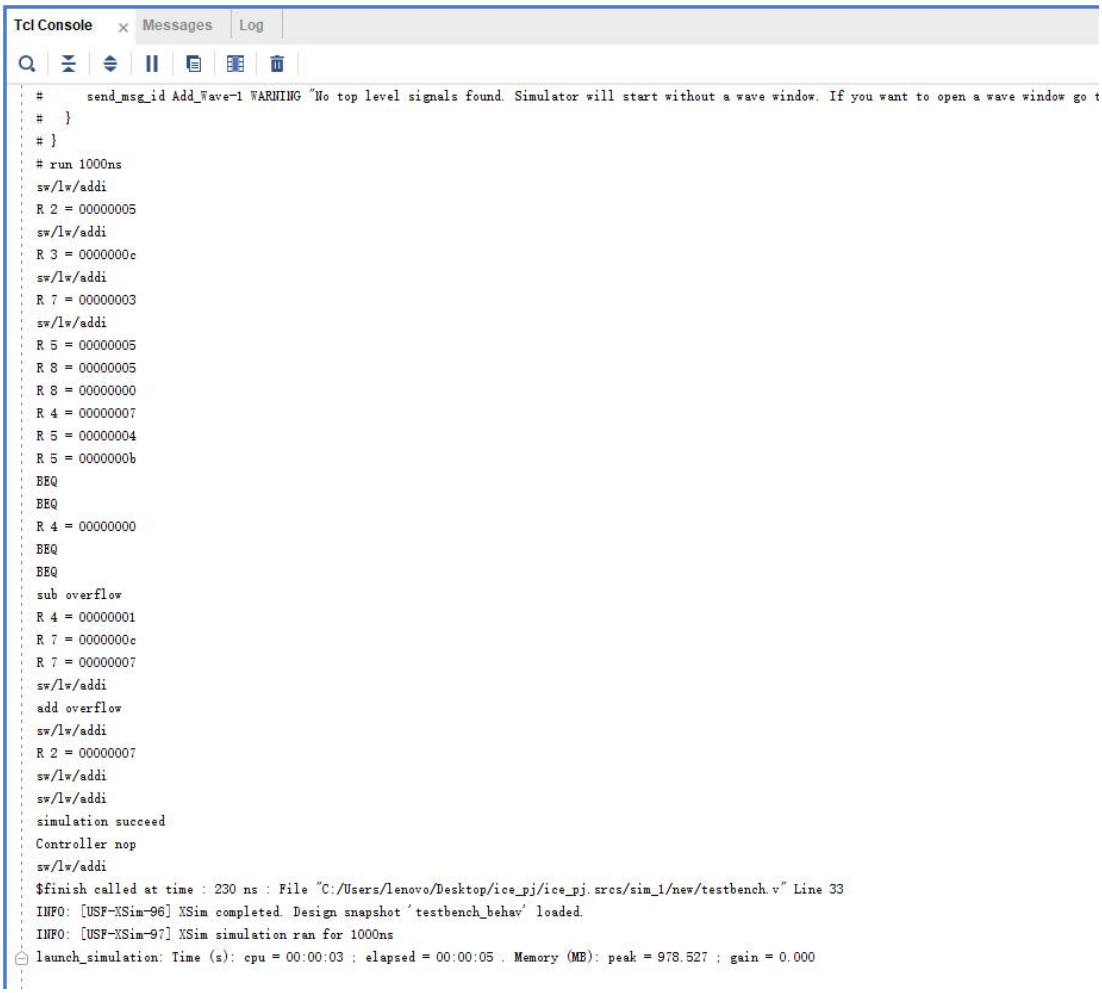
四、 仿真和结果分析

a) testbench

```
1  module testbench(
2      );
3      reg clk;
4      reg reset;
5      wire [31:0] writedata, dataadr;
6      wire memwrite;
7      wire [31:0] pc, instr, readdata;
8
9      top top(clk, reset, writedata, dataadr, memwrite,pc, instr, readdata);
10
11     initial #230 $finish;
12     initial begin
13         reset <= 1;
14         #2; reset <= 0;
15         end
16
```

```
17    always
18        begin
19            clk <= 1;
20            #5; clk <= 0;
21            #5;
22        end
23
24        // check result
25    always@(negedge clk)
26        begin
27            if(memwrite) begin
28                if(dataadr == 84 && writedata == 7) begin
29                    $display ("simulation succeed");
30                    //$stop;
31                end else if (dataadr != 80) begin
32                    $display("simulation failed");
33                    $stop;
34                end
35             end
36        end
37 endmodule
```

b) 仿真终端输出



				rs,rt,imme/rs,rt,rd			address
main	20020005	001000 00000 00010 00000 00000 000101	addi	rt = rs+ imme	$2, $0, 5	initialize $2 = 5	0
	2003000c	001000 00000 00011 00000 00000 001100	addi	rt = rs+ imme	$3, $0, 12	initialize $3 = 12	4
	2067fff7	001000 00011 00111 11111 11111 110111	addi	rt = rs+ imme	$7, $3, -9	initialize $7 = 3	8
	30450007	001100 00010 00101 00000 00000 000111	andi	rt = rs and imme	$5, $2, 7	initialize $5 = 5	c
	34a80001	001101 00101 01000 00000 00000 000001	ori	rt = rs or imme	$8, $5, 1	initialize $8 = 5	10
	28a80004	001010 00101 01000 00000 00000 000100	slti	rt = (rs < imme)	$8, $5, 4	initialize $8 = 0	14
	00e22025	000000 00111 00010 00100 00000 100101	or	rd = rs or rt	$4, $7, $2	$4 <= 3 or 5 = 7	18
	00642824	000000 00011 00100 00101 00000 100100	and	rd = rs and rt	$5, $3, $4	$5 <= 12 and 7 = 4	1c
	00a42820	000000 00011 00100 00101 00000 100000	add	rd = rs + rt	$5, $5, $4	$5 = 4 + 7 = b	20
	10a7000a	000100 00101 00111 00000 00000 001010	beq	if rs=rt then branch	$5, $7, end	shouldn't be taken	24
	0064202a	000000 00011 00100 00100 00000 101010	slt	rd = (rs<rt)	$4, $3, $4	$4 = 12 < 7 = 0	28
	10800001	000100 00100 00000 00000 00000 000001	beq	if rs=rt then branch	$4, $0, around	should be taken	2c
	20050000	001000 00000 00101 00000 00000 000000	addi	rt = rs+ imme	$5, $0, 0	shouldn't happen	30
around	00e2202a	000000 00111 00010 00100 00000 101010	slt	rd = (rs<rt)	$4, $7, $2	$4 = 3 < 5 = 1	34
	00853820	000000 00100 00101 00111 00000 100000	add	rd = rs + rt	$7, $4, $5	$7 = 1 + 11 = c	38
	00e23822	000000 00111 00010 00111 00000 100010	sub	rd = rs - rt	$7, $7, $2	$7 = 12 - 5 = 7	3c
	ac670044	101011 00011 00111 00000 00001 000100	sw	[$base+offset]=rt	$7, 68($3)	[80] = 7	40
	8c020050	100011 00000 00010 00000 00001 010000	lw	rt=[$base+offset]	$2, 80($0)	$2 = [80] = 7	44
	08000014	000010 00000 00000 00000 00000 010001	j		end	should be taken	48
	20020001	001000 00000 00010 00000 00000 000001	addi	rt = rs+ imme	$2, $0, 1	shouldn't happen	4c
end	ac020054	101011 00000 00010 00000 00001 010100	sw	[$base+offset]=rt	$2, 84($0)	write adr 84 = 7	50

i. 通过对比指令数据，第一个$2 赋值为 5，$3=12, $7=3，这些都是 addi，在 aludec 模块进行了 display 输出

ii. 其次的 andi，ori，slti， 使 $5=5, $8=5, $8=0

iii. Or, and, add 分别给 $4, $5,$5 赋值

iv. beq 由于不执行，但在 controller 模块有相应的 display，所以会输出 BEQ，实则不会有跳转语句

v. slt 给$4 赋值

vi. Beq，条件成立输出 BEQ，sub overflow（这个也是针对 beq 命令的输出），跳到下下个指令地址，跳转到 around 地址。

vii. 所以 beq 下面的 addi 没有执行，console 也没有输出

viii. Slt，add，sub 赋值$4，$7，$7

ix. Sw，存储，把 data 存入数据存储器 dataMem 中，80 去掉最后两位是 20，在地址为 20 的 mem 中写入 7。并在 console 输出 sw/lw/addi 和 addoverflow，有 addoverflow 是因为 sw 从 controller 进去会进行 alu 的加法运算，此时 alu 的两个输入值分别是 12 和 68（在仿真图中 150-160ns）。

x. Lw，从数据存储器中读，读出给$2=7， 并在 console 输出 sw/lw/addi 和给$2 赋值的结果

xi. J, jump,跳转到 end 对应的指令地址，所以在下面没有 addi 赋值的 display 输出，否则应该有$2=1

xii. End：sw，把数据 7 写入 datamem 中，地址 84 去掉最后两位是 21。

xiii. 在 testbench 中，基本命令结束后判断

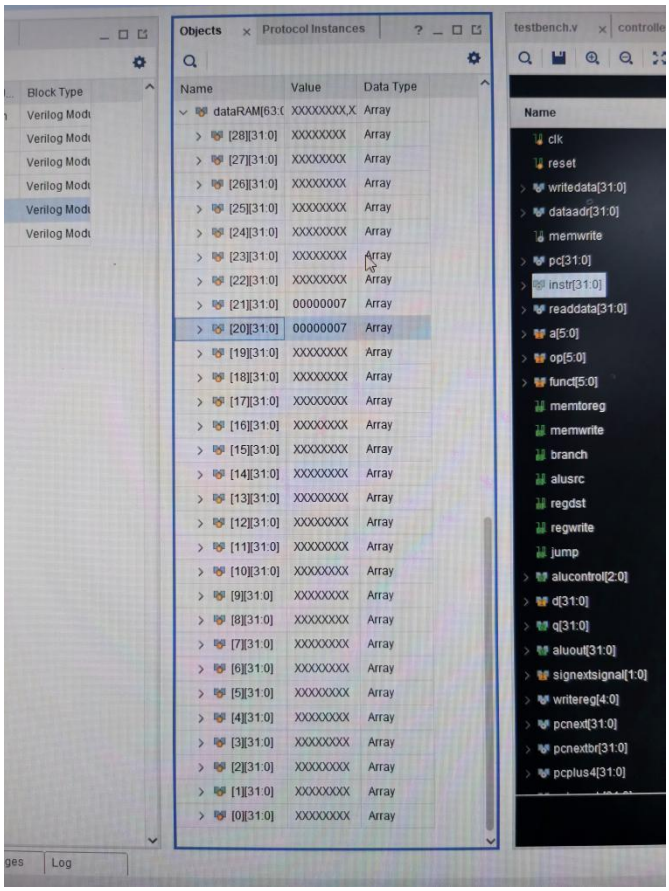
if(dataadr == 84 && writedata == 7) $display ("simulation succeed");

数据地址为 84，值为 7，所以在 console 输出了 simulation succeed，并且我没有设置在仿真成功后停止，为了观察后面的命令

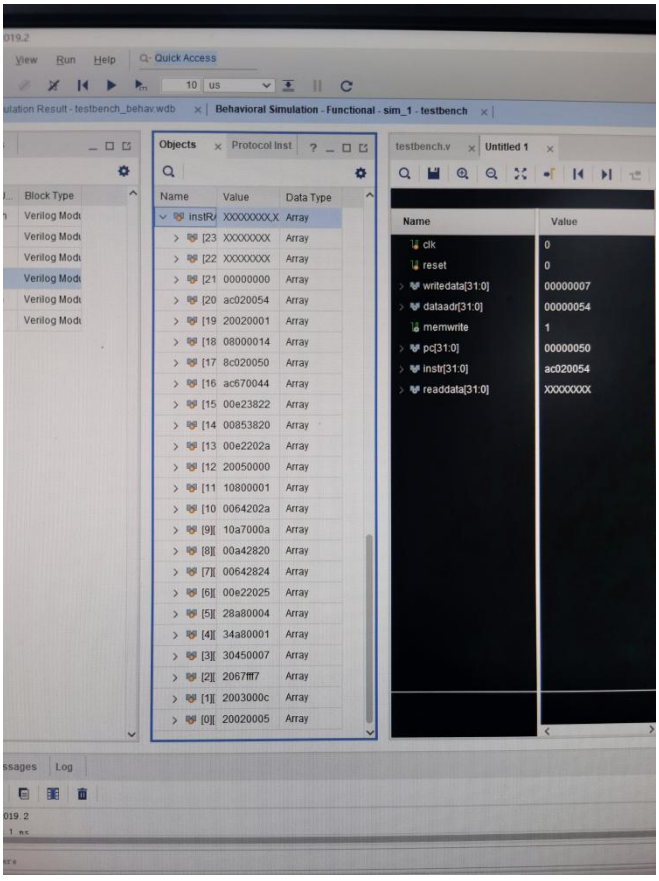
xiv. 最后一条命令，nop，在 controller 中进行了 nop 命令判断，如果有 nop，则 console 输出 Controller nop。

xv. 最后程序结束

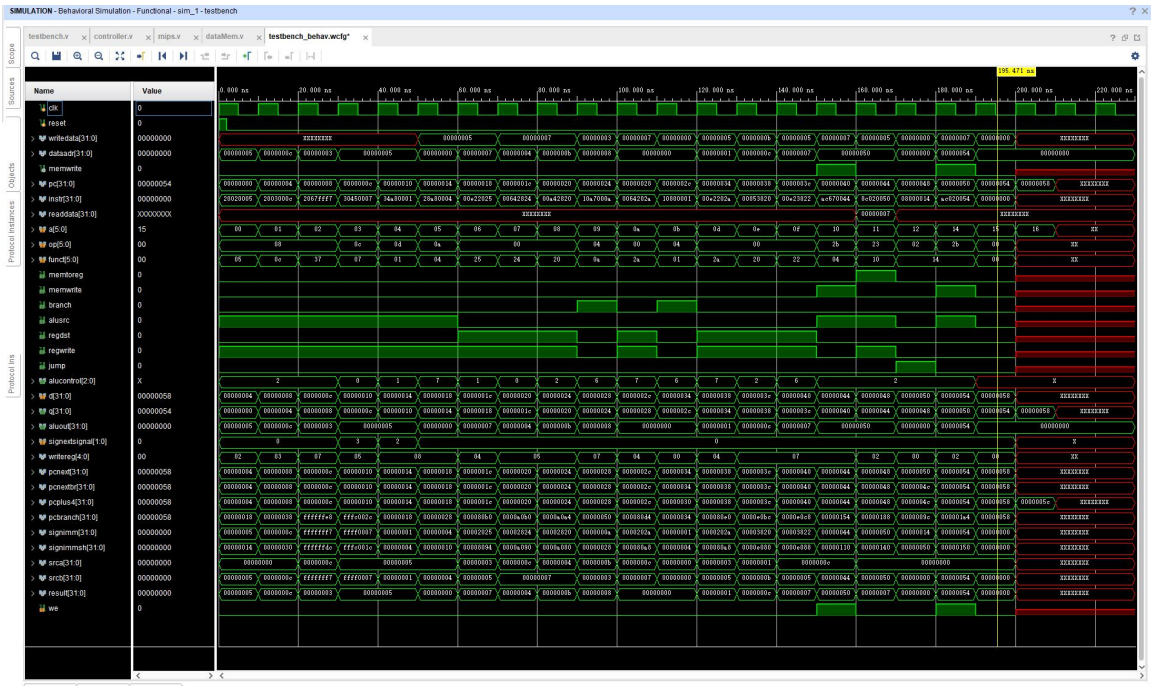
c) dataMem 仿真后的数据见下图，在 20 和 21 两处赋值 7，其他都未赋值。



d) instrMem 仿真后的效果见下图，一共 21 条命令



e) 仿真图验证



1）instr 所示所有的命令

2）Writedata，去的是 instr[20:16]前几位是$2,$3,$7,$5,$8,由于在最开始并未赋值，所以仿真与查找他们数据是都未 XXXX，其后的$8,在上一步已经赋值了 5，所以 writedata 显示了 5.

3）Dataadr，直接显示写入的数据，对应到 console 输出所有=右边的数据。

4）pc 值，以每个 4 递增。

5）Readdata，从 dataMem 读出来的数据，仅有当指令为 sw，也即 memwrite 为 1 时，能够有从存储器读出的数据，所以前面的都为 X，160-170ns 有值，是 7。

6）a 是指令地址，逐个递增。

7）Op，funct 分别是指令的前 6 位和后 6 位。

8）Memtoreg，控制存储器读出数据，仅有指令为 lw 时为 1，对应时间 160-170ns。

9）Memwrite， 控制写入存储器，仅当指令为 sw 时为 1，对应时间为 150-160ns，180-190ns。

10） Branch，控制 beq 命令，对应时间为 90-100ns， 110-120ns，对应命令为 beq。

11） Alusrc，所有 i 类型的命令都为 1，使 alu 运算时，第二个变量能读入 instr 指令的立即数。

12） Regdst，所有的 r 类型的命令都为 1，使 R 类型命令进行运算时，能读到 instr[15:11],也即 rd 数据。

13） Regwrite，寄存器写使能，把数据写入寄存器，除了 sw（存入存储器），beq，j 命令，其他都为 1。

14） Jump，仅当 j 命令时为 1。

15） Alucontrol，alu 控制符号，前面都是 addi，对应 alucontrol2，andi（0），ori（1），slti（7）,or(1),and(0),add(2), beq(6), slt(7), sw(2), lw(2)

16） D,q,分别是 pc 下一个地址和这次地址。

17） Signextsingal，当 andi 时，值为 3，意味着高位补 1,；ori 时，值为 2，高位

补 0.。

18） Writereg， rt 的地址，与 console 输出的 R 后面的地址一致。

19） Signimm， instr 的后 16 位数据，从 5，到 c，到 fffffff7...

20） Signimmsh，signimm 左移两位。

21） Srac， srab， result 是 alu 运算的输入和输出。

22） We， 就是 memwrite，写入寄存器 enable。

2 当时间到 200ns 时，对应 nop 命令，空命令，后面的数据都未 X