计算机组成实验1

单周期CPU核心模块设计仿真

518030910283 王航宇

一、 实验目的和要求

1、 采用Verilog硬件描述语言在Quartus II EDA设计平台中，基于Intel cyclone II系列FPGA完成具有执行20条MIPS基本指令的单周期CPU模块的设计。根据提供的单周期 CPU示例程序的Verilog代码文件将设计代码补充完整，实现该模块的电路设计。

1、 利用实验提供的标准测试程序代码，完成单周期CPU模块的功能仿真测试，验证CPU执行所设计的20条RISC指令功能的正确性。

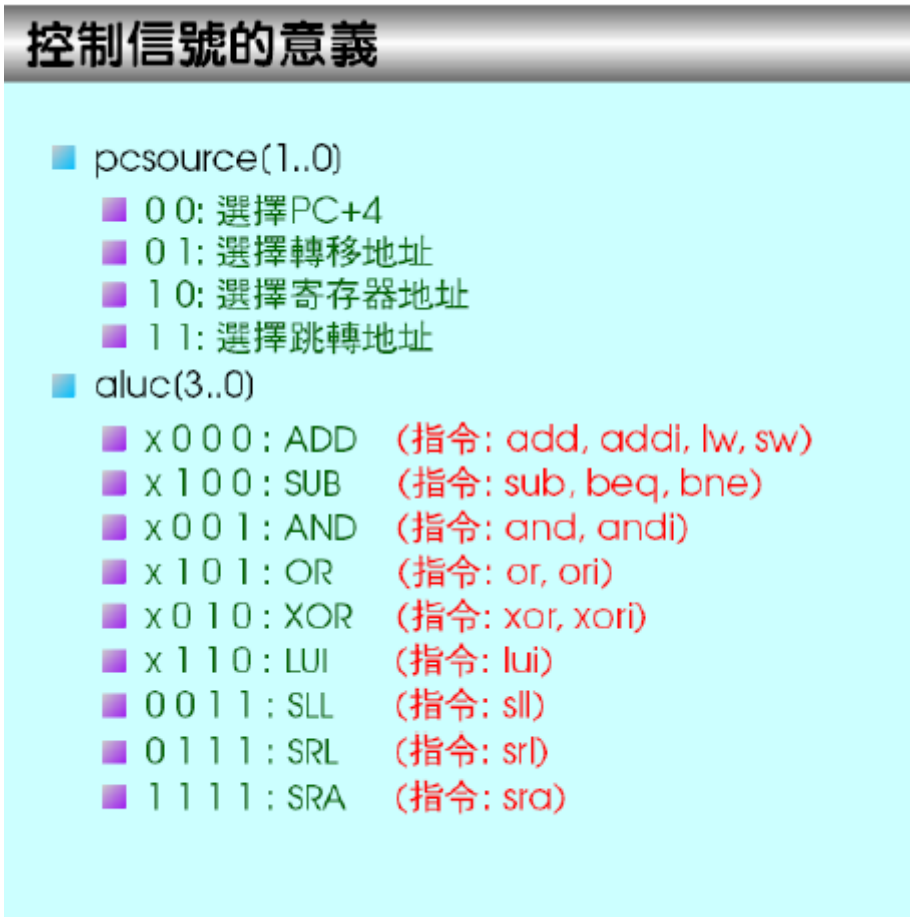
从而理解计算机五大组成部分的协调工作原理，理解存储程序自动执行的原理和掌握运算器、存储器、控制器的设计和实现原理。

二、 核心代码

alu.c补充代码

```
module alu (a,b,aluc,s,z);
   input [31:0] a,b;
   input [3:0] aluc;
   output [31:0] s;
   output        z;
   reg [31:0] s;
   reg        z;
   always @ (a or b or aluc)
      begin                                 // event
         casex (aluc)
             4'bx000: s = a + b;            //x000 ADD
             4'bx100: s = a - b;            //x100 SUB
             4'bx001: s = a & b;            //x001 AND
             4'bx101: s = a | b;            //x101 OR
             4'bx010: s = a ^ b;            //x010 XOR
             4'bx110: s = {b[15:0],16'b0};  //x110 LUI: imm << 16bit
             4'b0011: s = b << a;           //0011 SLL: rd <- (rt << sa)
             4'b0111: s = b >> a;           //0111 SRL: rd <- (rt >> sa) (logical)
             4'b1111: s = $signed(b) >>> a; //1111 SRA: rd <- (rt >> sa) (arithmetic)
             default: s = 0;
         endcase
         if (s == 0 )  z = 1;
            else z = 0;
      end
endmodule
```

根据给定的aluc信号，判断是哪一种运算，写出对应运算的实现代码，具体对应如下图



sc_cu.v补充代码

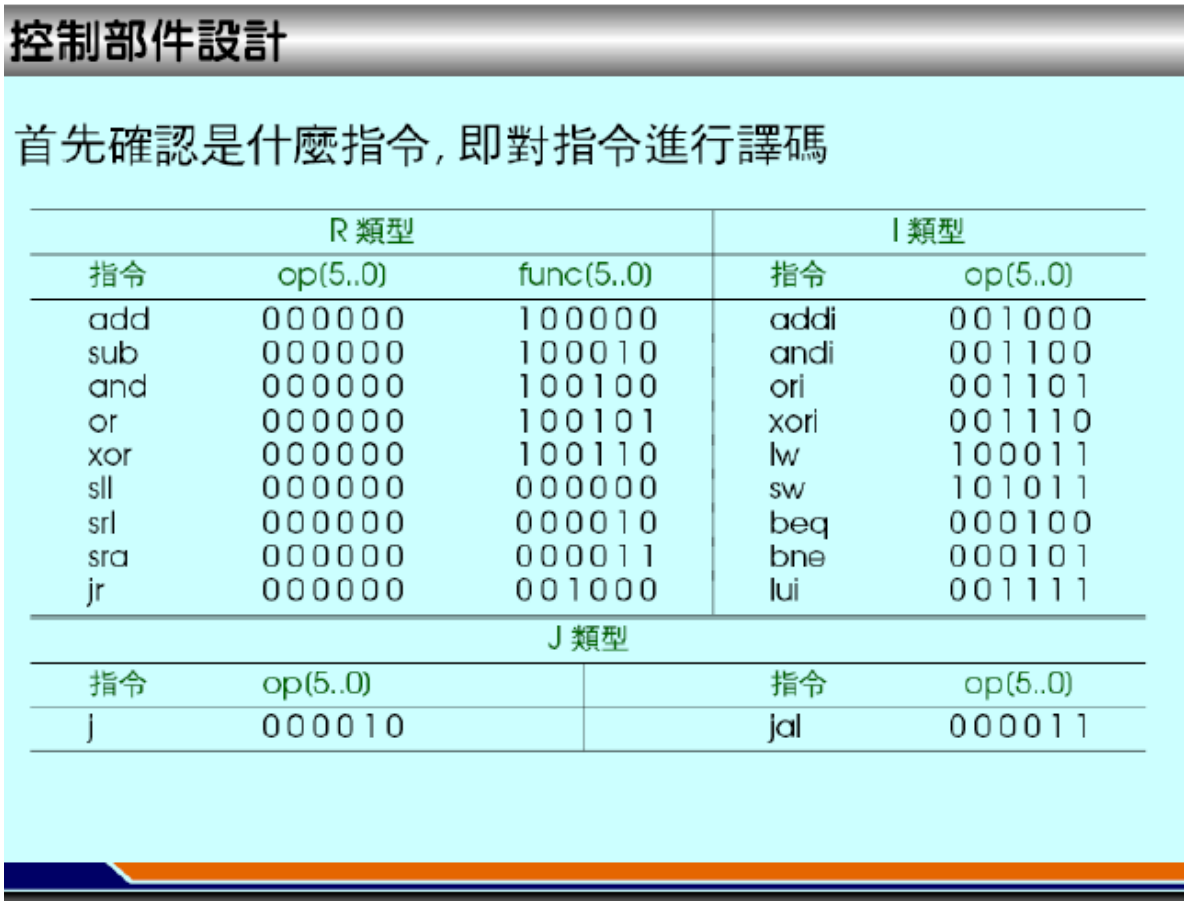
```
module sc_cu (op, func, z, wmem, wreg, regrt, m2reg, aluc, shift,
              aluimm, pcsource, jal, sext);
   input  [5:0] op,func;
   input        z;
   output       wreg,regrt,jal,m2reg,shift,aluimm,sext,wmem;
   output [3:0] aluc;
   output [1:0] pcsource;
   wire r_type = ~|op;
   wire i_add = r_type & func[5] & ~func[4] & ~func[3] &
                ~func[2] & ~func[1] & ~func[0];          //100000
   wire i_sub = r_type & func[5] & ~func[4] & ~func[3] &
                ~func[2] &  func[1] & ~func[0];          //100010


   wire i_and = r_type & func[5] & ~func[4] & ~func[3] &
                func[2] & ~func[1] & ~func[0];           //100100
   wire i_or  = r_type & func[5] & ~func[4] & ~func[3] &
                func[2] & ~func[1] & func[0];            //100101
   wire i_xor = r_type & func[5] & ~func[4] & ~func[3] &
                func[2] & func[1] & ~func[0];            //100110
   wire i_sll = r_type & ~func[5] & ~func[4] & ~func[3] &
                ~func[2] & ~func[1] & ~func[0];          //000000
   wire i_srl = r_type & ~func[5] & ~func[4] & ~func[3] &
                ~func[2] & func[1] & ~func[0];           //000010
   wire i_sra = r_type & ~func[5] & ~func[4] & ~func[3] &
                ~func[2] & func[1] & func[0];            //000011
   wire i_jr  = r_type & ~func[5] & ~func[4] & func[3] &
                ~func[2] & ~func[1] & ~func[0];          //001000

   wire i_addi = ~op[5] & ~op[4] &  op[3] & ~op[2] & ~op[1] & ~op[0]; //001000
   wire i_andi = ~op[5] & ~op[4] &  op[3] &  op[2] & ~op[1] & ~op[0]; //001100
```

```
    wire i_ori  = ~op[5] & ~op[4] &  op[3] & op[2] & ~op[1] & op[0];  //001101
    wire i_xori = ~op[5] & ~op[4] &  op[3] & op[2] & op[1] & ~op[0];  //001110
    wire i_lw   = op[5] & ~op[4] &  ~op[3] & ~op[2] & op[1] & op[0];  //100011
    wire i_sw   = op[5] & ~op[4] &  op[3] & ~op[2] & op[1] & op[0];   //101011
    wire i_beq  = ~op[5] & ~op[4] &  ~op[3] & op[2] & ~op[1] & ~op[0];//000100
    wire i_bne  = ~op[5] & ~op[4] &  ~op[3] & op[2] & ~op[1] & op[0]; //000101
    wire i_lui  = ~op[5] & ~op[4] &  op[3] & op[2] & op[1] & op[0];   //001111
    wire i_j    = ~op[5] & ~op[4] &  ~op[3] & ~op[2] & op[1] & ~op[0];//000010
    wire i_jal  = ~op[5] & ~op[4] &  ~op[3] & ~op[2] & op[1] & op[0]; //000011


    assign pcsource[1] = i_jr | i_j | i_jal;
    assign pcsource[0] = ( i_beq & z ) | (i_bne & ~z) | i_j | i_jal ;

    assign wreg = i_add | i_sub | i_and | i_or   | i_xor  |
                  i_sll | i_srl | i_sra | i_addi | i_andi |
                  i_ori | i_xori | i_lw | i_lui  | i_jal;

    assign aluc[3] = i_sra;
    assign aluc[2] = i_sub | i_beq | i_bne | i_or | i_ori | i_lui | i_srl | i_sra;
    assign aluc[1] = i_xor | i_sll | i_srl | i_sra | i_xori | i_lui;
    assign aluc[0] = i_and | i_andi | i_or | i_ori | i_sll | i_srl | i_sra;
    assign shift   = i_sll | i_srl | i_sra ;

    assign aluimm  = i_addi | i_andi | i_ori | i_xori | i_lw | i_sw | i_lui;
    assign sext    = i_addi | i_lw | i_sw | i_beq | i_bne;
    assign wmem    = i_sw;
    assign m2reg   = i_lw;
    assign regrt   = i_addi | i_andi | i_ori | i_xori | i_lw | i_lui;
    assign jal     = i_jal;

endmodule
```

首先判断指令类型，r_type用于判断指令是否是R型指令。

如果是R型指令，根据对应的func处代码写出逻辑表达式，生成表示指令的中间变量

如果是I型或是J型指令，根据op处代码写出逻辑表达式，也生成表示指令的中间变量

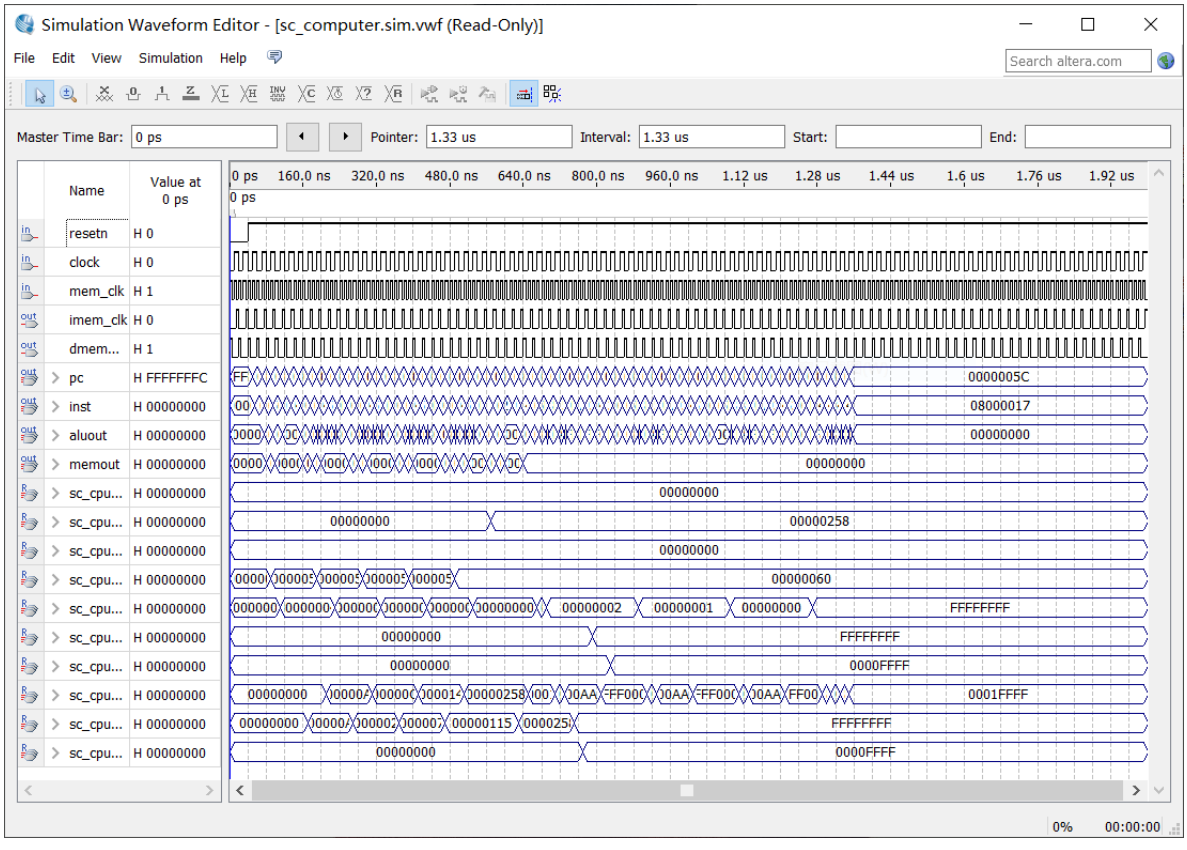


然后，根据不同指令对于不同控制信号的需要列出表格，写出控制信号的逻辑表达式

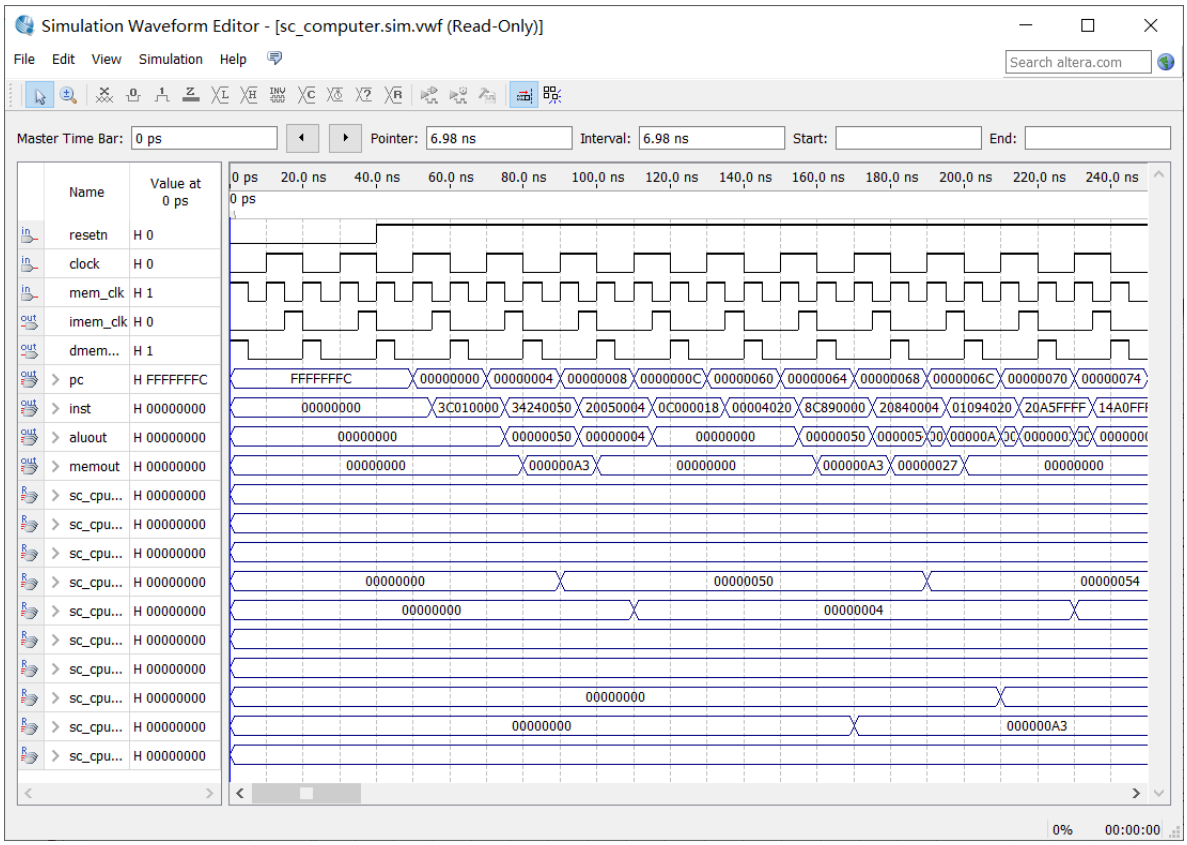
指令	指令格式	aluc [3..0]	shift	aluimm	sext	wmem	wreg	m2reg	regrt	call jal
add	add rd, rs, rt	x 0 0 0	0	0	x	0	1	0	0	0
sub	sub rd, rs, rt	x 1 0 0	0	0	x	0	1	0	0	0
and	and rd, rs, rt	x 0 0 1	0	0	x	0	1	0	0	0
or	or rd, rs, rt	x 1 0 1	0	0	x	0	1	0	0	0
xor	xor rd, rs, rt	x 0 1 0	0	0	x	0	1	0	0	0
sll	sll rd, rt, sa	0 0 1 1	1	0	x	0	1	0	0	0
srl	srl rd, rt, sa	0 1 1 1	1	0	x	0	1	0	0	0
sra	sra rd, rt, sa	1 1 1 1	1	0	x	0	1	0	0	0
jr	jr rs	x x x x	x	x	x	0	0	x	x	x
addi	addi rt, rs, imm	x 0 0 0	0	1	1	0	1	0	1	0
andi	andi rt, rs, imm	x 0 0 1	0	1	0	0	1	0	1	0
ori	ori rt, rs, imm	x 1 0 1	0	1	0	0	1	0	1	0
xori	xori rt, rs, imm	x 0 1 0	0	1	0	0	1	0	1	0
lw	lw rt, imm(rs)	x 0 0 0	0	1	1	0	1	1	1	0
sw	sw rt, imm(rs)	x 0 0 0	0	1	1	1	0	0	0	0
beq	beq rs, rt, imm	x 1 0 0	0	0	1	0	0	0	0	0
bne	bne rs, rt, imm	x 1 0 0	0	0	1	0	0	0	0	0
lui	lui rt, imm	x 1 1 0	x	1	x	0	1	0	1	0
						0				
j	j addr	x x x x	x	x	x	0	0	x	x	x
jal	jal addr	x x x x	x	x	x	0	1	x	x	1

三、 仿真结果

总仿真图



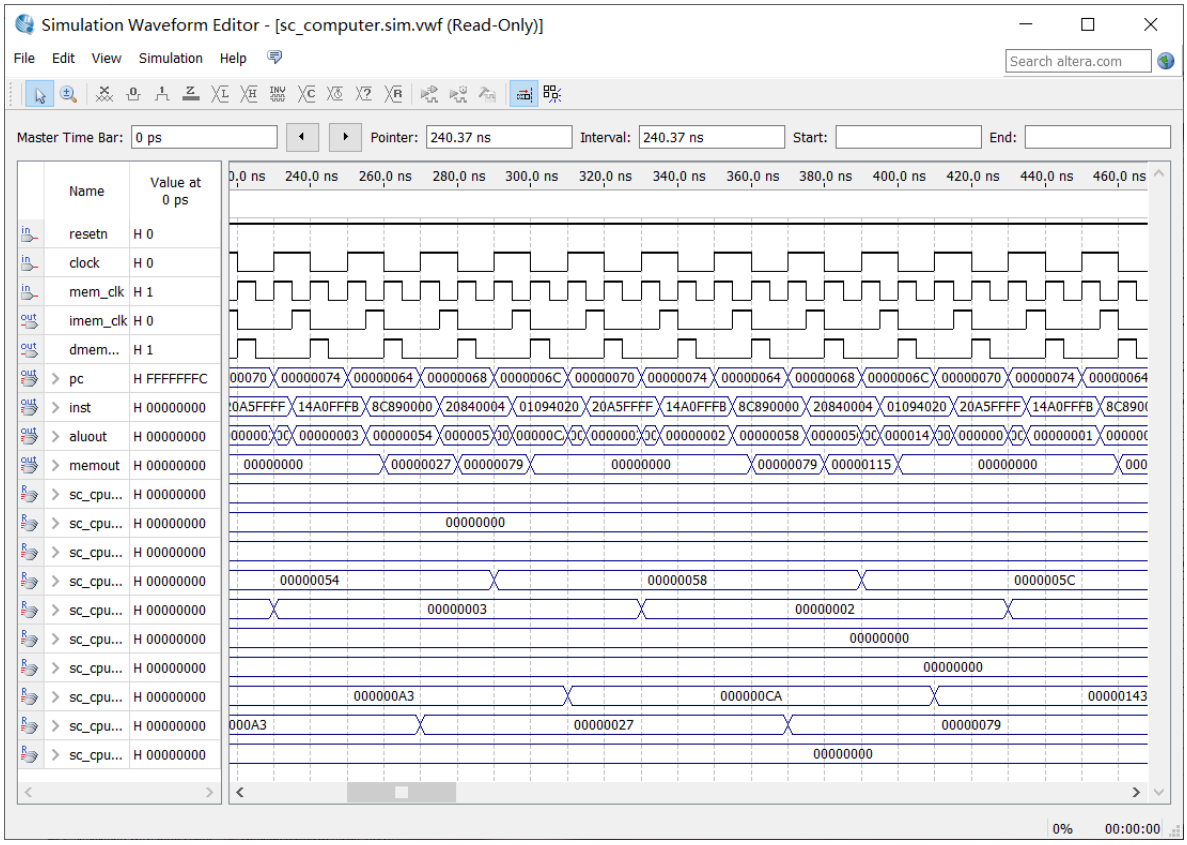
仿真细节图1



可以看到，PC在resetn置1且每个clock上升沿时加4，然后inst在每个imem_clk上升沿时取址。

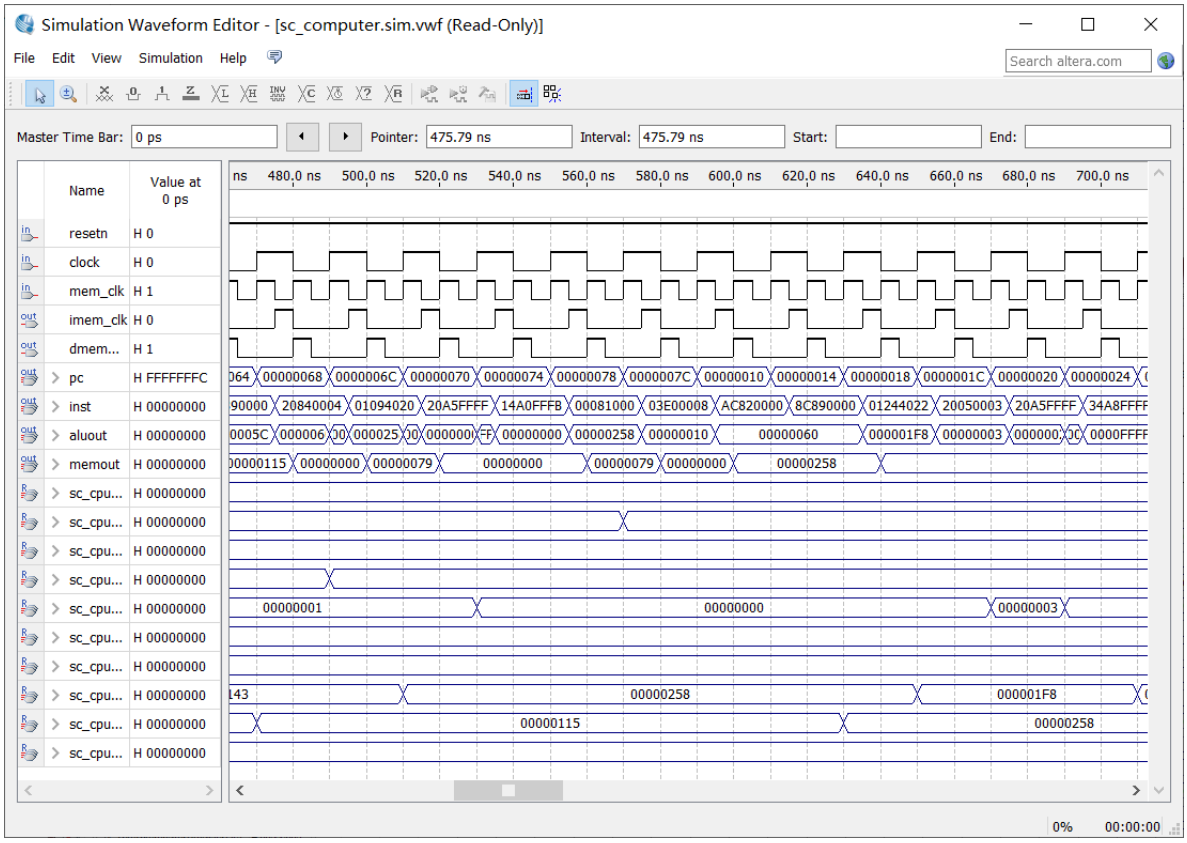
事实上，imem_clk和dmem_clk分贝作为指令ROM和数据RAM的同步时钟

图二



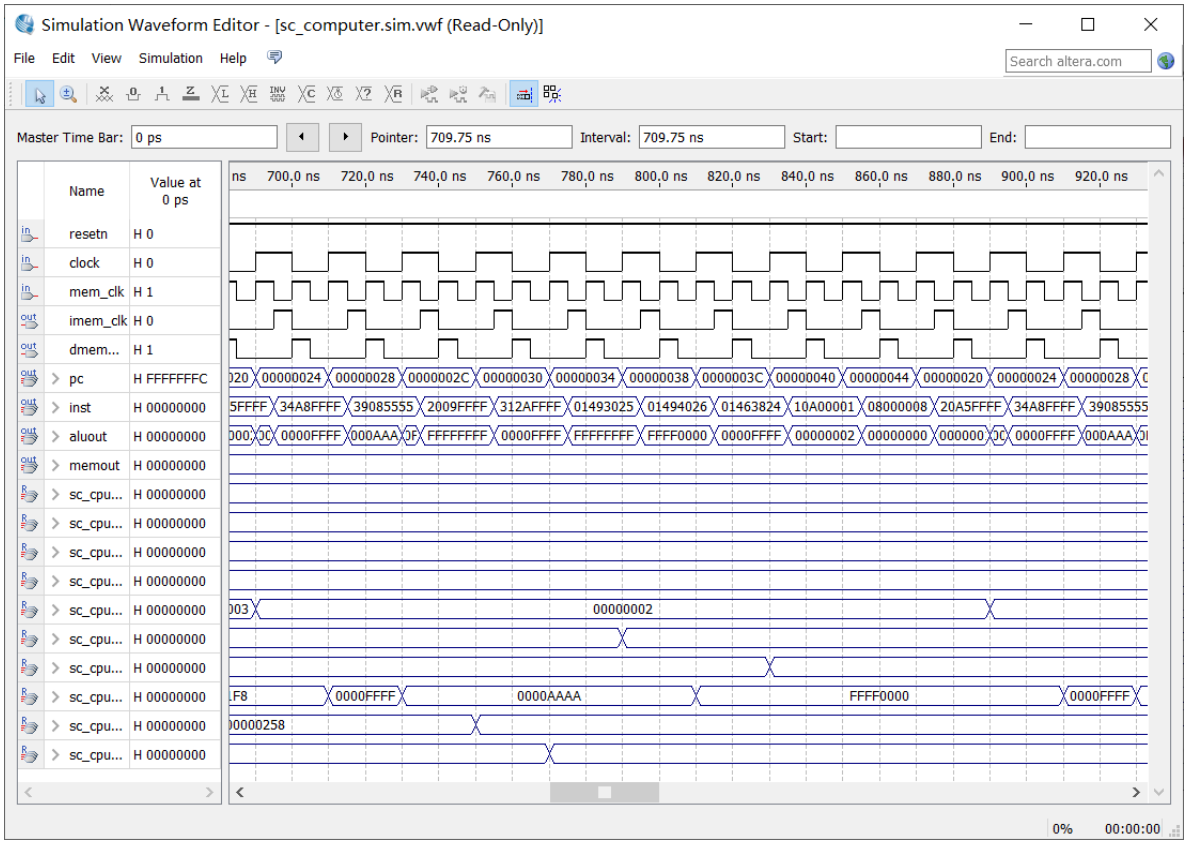
可以看到420ns时，pc=00000070，inst=20A5FFFF，aluout=00000001，memout=00000000

图三



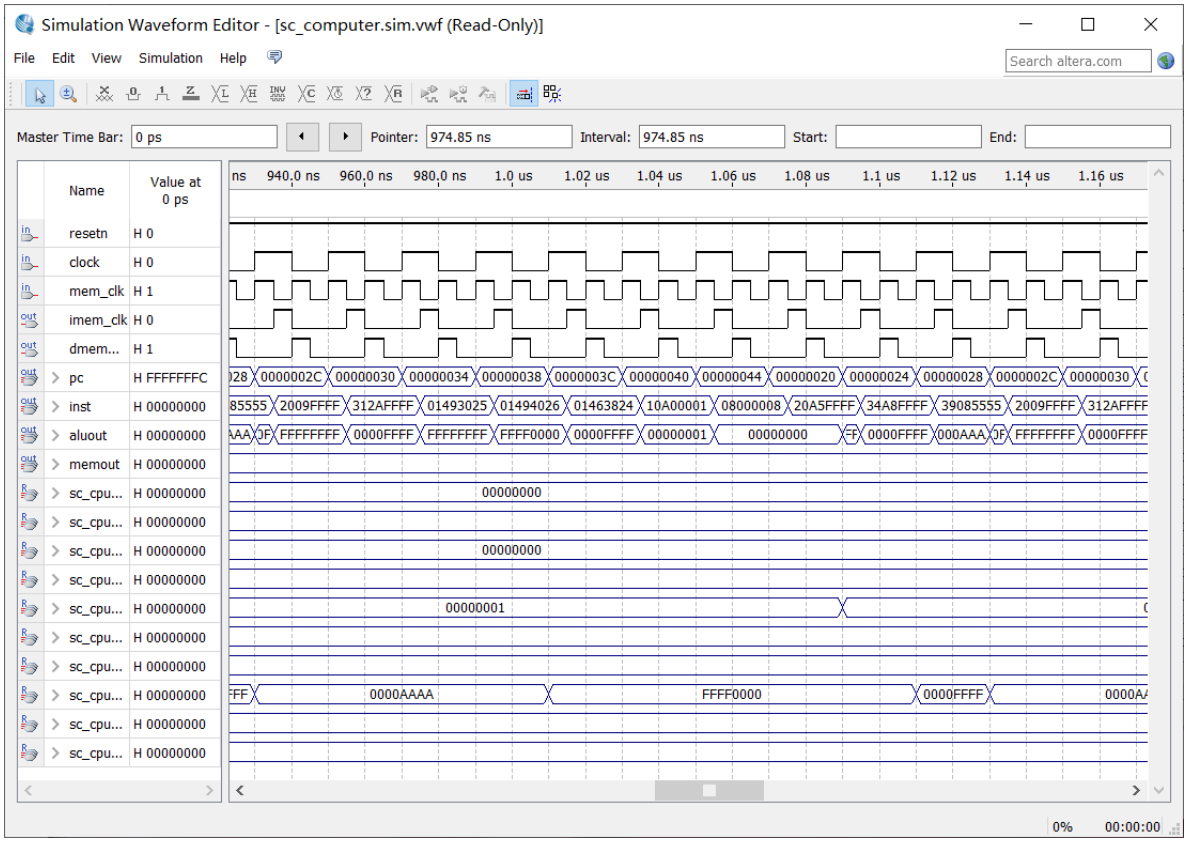
680ns时，pc=00000020，inst=20A5FFFF，aluout=00000002，memout=00000000

图四



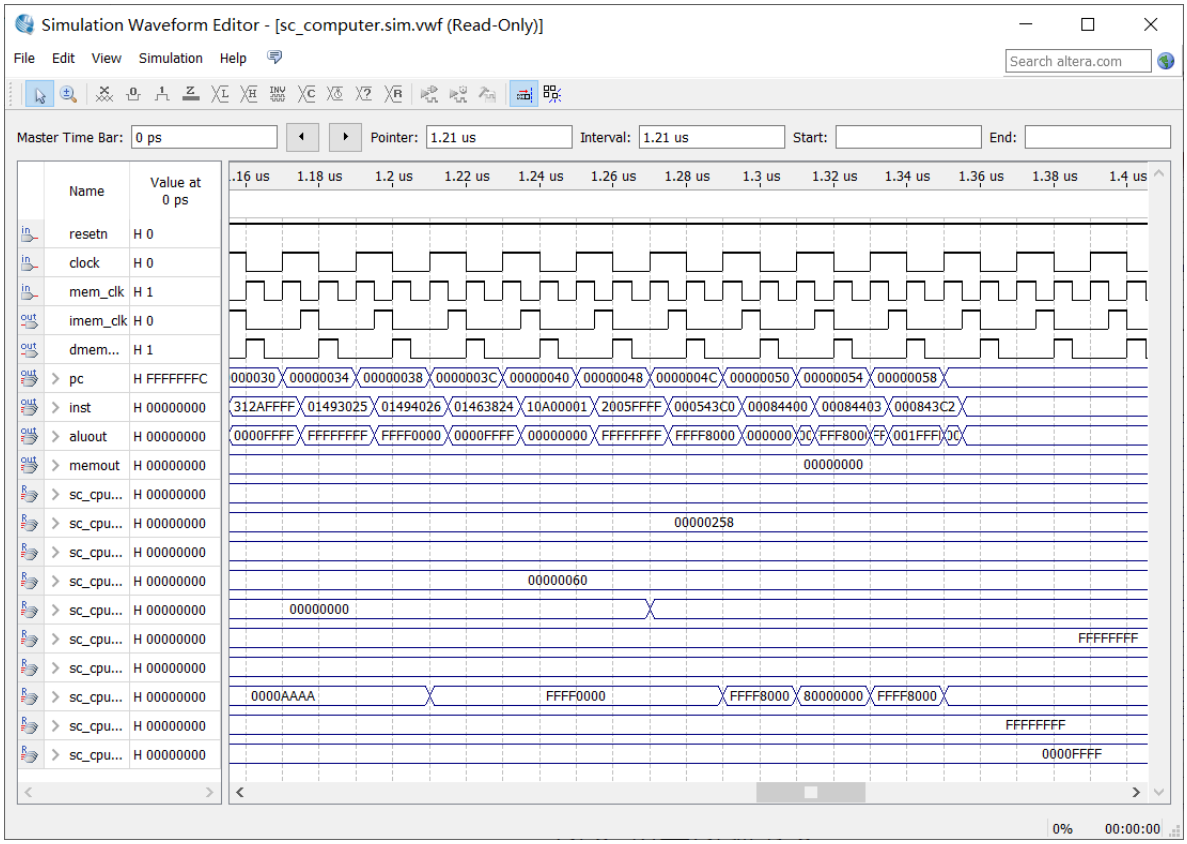
900ns时，pc=00000024，inst=34A8FFFF，aluout=0000FFFF，memout=00000000

图五



1.08us时，pc=00000020，inst=20A5FFFF，aluout=00000000，memout=00000000

图六

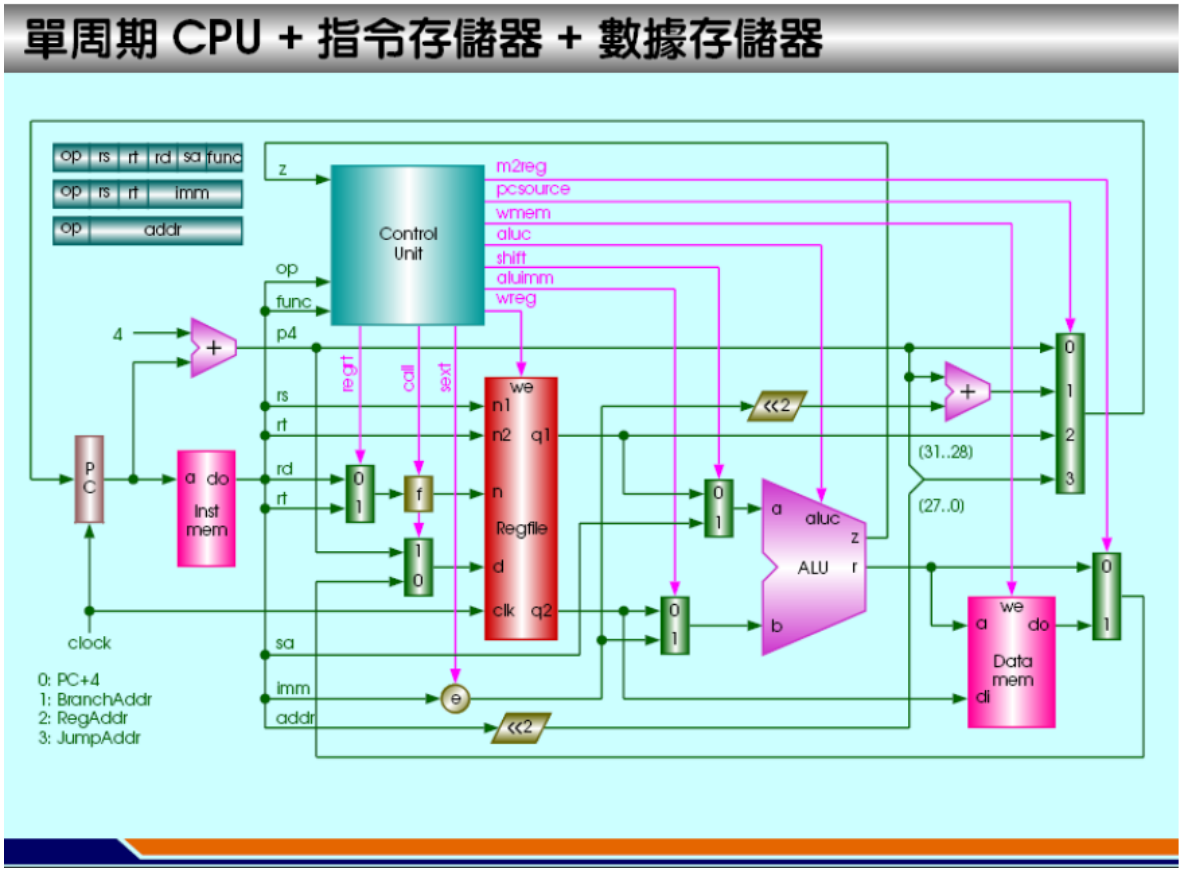


1.355us时，程序跳转到死循环，pc=00000058，inst=000843C2，aluout=0001FFFF

四、 设计思路分析

单周期CPU指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。

CPU在处理指令时，一般需要经过以下几个步骤：取指令IF、指令译码ID、指令执行EXE、存储器访问MEM、结果写回WB。单周期CPU，是在一个时钟周期内完成这五个阶段的处理。



上图就是单周期CPU数据通路和控制线路图，其中指令和数据存储在不同的存储器中。

从顶层文件sc_computer中可知，此单周期sc_computer由CPU、指令ROM、数据RAM共三个模块组成。三个模块之间由定义为wire型连接信号线的多路数据连接信号线通过各个模块的输入输出端口（信号）进行互联。同层同名的信号线连通不同模块的端口，一个模块的输出信号可做为一个或者多个模块的输入信号。

详细理解cpu模块下子模块功能：

alu:al_unit：输入为a,b和aluc，根据控制信号aluc对a，b两个数进行运算操作，并把最终结果赋给输出s，如果s为零，输出z置1，否则z置0

mux2x32:alu_a：用于选择ALU操作数a。如果控制信号shift为1，那么输出alua就等于sa，否则alua等于ra

mux2x32:alu_b：用于选择ALU操作数b。如果控制信号aluimm为1，那么输出alub就等于immediate，否则alub等于data

cla32:br_adr：用于计算跳转指令时下一条指令的地址。令adr的值等于p4+offset

sc_cu:cu：根据op和func段指令和ALU输出的z控制信号，生成用于之后操作的控制信号

- pcsource用于选择下一条指令的地址
- wreg是寄存器堆的写使能信号
- aluc控制ALU的运算
- shift控制ALU操作数a的值

- aluimm控制ALU操作数b的值
- sext控制符号扩展和零扩展
- wmem控制数据存储器写入
- m2reg控制寄存器写入
- regrt控制写入的目标寄存器
- jal控制jal指令的跳转

dff32:ip：给PC设计一个D寄存器

mux2x32:link：用于选择下一条指令是否需要跳转。如果控制信号jal为1，那么输出res就等于p4，否则res就等于alu_mem的值

mux4x32:pcsource：根据控制信号pcsource的值，选择下一条指令的地址

cla32:pcplus4：p4=pc+4，为下一条指令的地址

mux2x5:reg_wn：用于选择是取rt还是rd作为写入的目标寄存器地址。如果控制信号regrt为1，输出reg_dest就取rt，否则就取rd

mux2x32:result：用于选择是ALU的计算结果还是数据存储器的数据。如果控制信号m2reg为1，alu_mem就取mem的值，否则就取alu的值

regfile:rf：(1)输入寄存器rna、rnb，输出读取的内容qa、qb；

(2)根据时钟信号clk、clrn和写信号we，在指定寄存器wn处写入数据d；

(3)根据时钟信号和复位信号clrn，将所有32个寄存器清零