实验一：单周期CPU

蒋傲凡 518030910275

1 核心代码截图

代码截图如下所示

2 整体仿真结果

仿真结果和提供文件一致，仅在一个地方，即720ns处，ALU的计算结果有出入。根据提供的测试指令，可以计算出ALU的输出值为0000AAAA，仿真结果同样为该值。但提供的文件中该处为00015554，怀疑提供的文件结果存在问题。

具体实验仿真结果截图见下

3 设计思路分析

3.1 ALU部分设计思路

由计算机组成课程上知识可以知道，ALU在接受不同的控制信号下会进行不同的运算操作。在Verilog实现的时候，以此为原理通过对控制信号二进制值的判断，使其完成不同的操作。

加减操作，与、或、异或操作可直接用操作符表示。逻辑移位操作SRL与SLL也可以直接用¡¡和¿¿在Verilog中实现。在LUI操作中，需要将立即数左移十六位，即在尾部添加十六个零。这里我采用了级联的语法实现移位操作。值得注意的是，立即数imm是通过ALU的b端输入，因此LUI指令的操作对象应为b而非a

3.2 CU部分设计思路

首先基于MIPS三类指令 R型，I型，J型的特点，可以通过代码的opcode和func部分判断出具体的指令类型。R型指令需要opcode和func部分联合判断，I型和J型指令则只需通过opcode判断，这样就确定出了各指令。

而控制信号的确定则是建立在对各个MIPS指令的操作过程的熟悉。利用提供的真值表，在补全空缺信息后即可确定各控制信号与指令的联系,表格见附件，截图如下

```
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
```

Figure 1: ALU部分代码

```
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

   //  please complete the deleted code.

   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

// complete by yourself.
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;

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

  // complete by yourself.
  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
```

Figure 2: CU部分代码

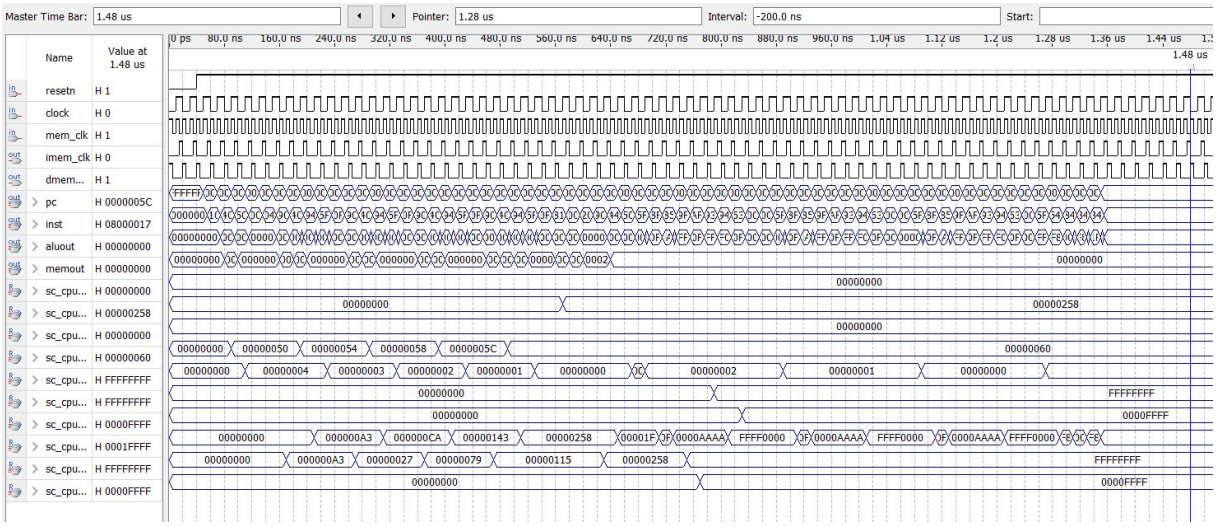

Figure 3: 总体图

指令	指令格式	op	rs	rt	rd	sa	func	z	pcsource [1..0]	aluc [3..0]	shift	aluimm	sext	wmem	wreg	m2reg	regrt	call jal
add	add rd, rs, rt	000000	rs	rt	rd	00000	100000	x	0 0	x 0 0 0	0	0	x	0	1	0	0	0
sub	sub rd, rs, rt	000000	rs	rt	rd	00000	100010	x	0 0	x 1 0 0	0	0	x	0	1	0	0	0
and	and rd, rs, rt	000000	rs	rt	rd	00000	100100	x	0 0	x 0 0 1	0	0	x	0	1	0	0	0
or	or rd, rs, rt	000000	rs	rt	rd	00000	100101	x	0 0	x 1 0 1	0	0	x	0	1	0	0	0
xor	xor rd, rs, rt	000000	rs	rt	rd	00000	100110	x	0 0	x 0 1 0	0	0	x	0	1	0	0	0
sll	sll rd, rt, sa	000000	00000	rt	rd	sa	000000	x	0 0	0 0 1 1	1	0	x	0	1	0	0	0
srl	srl rd, rt, sa	000000	00000	rt	rd	sa	000010	x	0 0	0 1 1 1	1	0	x	0	1	0	0	0
sra	sra rd, rt, sa	000000	00000	rt	rd	sa	000011	x	0 0	1 1 1 1	1	0	x	0	1	0	0	0
jr	jr rs	000000	rs	00000	00000	00000	001000	x	1 0	x x x x	x	x	x	0	0	x	x	x

指令	指令格式	op	rs	rt	rd	sa	func		pcsource [1..0]	aluc [3..0]	shift	aluimm	sext	wmem	wreg	m2reg	regrt	call jal
addi	addi rt, rs, im	001000	rs	rt	imm			x	0 0	x 0 0 0	0	1	1	0	1	0	1	0
andi	andi rt, rs, im	001100	rs	rt	imm			x	0 0	x 0 0 1	0	1	0	0	1	0	1	0
ori	ori rt, rs, im	001101	rs	rt	imm			x	0 0	x 1 0 1	0	1	0	0	1	0	1	0
xori	xori rt, rs, im	001110	rs	rt	imm			x	0 0	x 0 1 0	0	1	0	0	1	0	1	0
lw	lw rt, imm(rs	100011	rs	rt	imm			x	0 0	x 0 0 0	0	1	1	0	1	1	1	0
sw	sw rt, imm(rs	101011	rs	rt	imm			x	0 0	x 0 0 0	0	1	1	1	0	0	0	0
beq	beq rs, rt, im	000100	rs	rt	imm			0	0 0	x 1 0 0	0	0	1	0	0	0	0	0
								1	0 1									
bne	bne rs, rt, im	000101	rs	rt	imm			0	0 1	x 1 0 0	0	0	1	0	0	0	0	0
								1	0 0									
lui	lui rt, imm	001111	00000	rt	imm			x	0 0	x 1 1 0	0	1	0	0	1	0	1	0
j	j addr	000010	addr						1 1	x x x x	x	x	x	0	0	x	x	x
jal	jal addr	000011	addr						1 1	x x x x	x	x	x	0	1	x	x	1

Figure 4: 控制信号真值表

3.3 整体理解

在单周期CPU的实现中，大概将CPU划分为PC、寄存器、ALU模块和CU以实现指令跳转，数据寄存，算术逻辑运算等功能。

3.3.1 指令跳转

由计算机组成可知，CPU内部的PC负责指令的逐条读取。若无跳转则每次PC+4，否则还要再加上跳转的值，这一功能的实现是通过cla32.v完成的，即针对是否跳转两种情况，对PC进行相应的加法。若为分支跳转，则将指令后十六位立即数符号扩展再左移两位得到偏移量offset，用于PC值改变的加法计算。

在下一个PC值的选取上，使用了一个四路的多选器mux4x32.v，pcsource的值可能为00到11四个，分别对应PC+4，分支跳转，jr指令的32位跳转和j型指令的保留PC高四位

3.3.2 算术与逻辑操作

这一部分主要是通过alu.v完成的，向两个数据通道输入指定的值alua和alub，再输入控制信号aluc，输出结果alu和zero信号。在ALU输入数据时，需要通过两个多选器分别确定alua和alub，这是通过mux2x32.v完成的。在a端判断是来自寄存器的ra还是偏移量sa，在b端判断是来自寄存器的data还是立即数immediate

3.3.3 数据寄存

这一部分主要是通过regfile.v完成的。在寄存器读取时，根据指令形式判断两个输出口qa和qb的使用。在数据写入时，首先需要有we信号支持，同时针对特殊的jal指令的PC值寄存的31号寄存器，通过wn进行判断，若为jal指令，wn为11111，否则为指定写入的rt或rd寄存器，这是通过mux2x5.v和判断语句实现的。

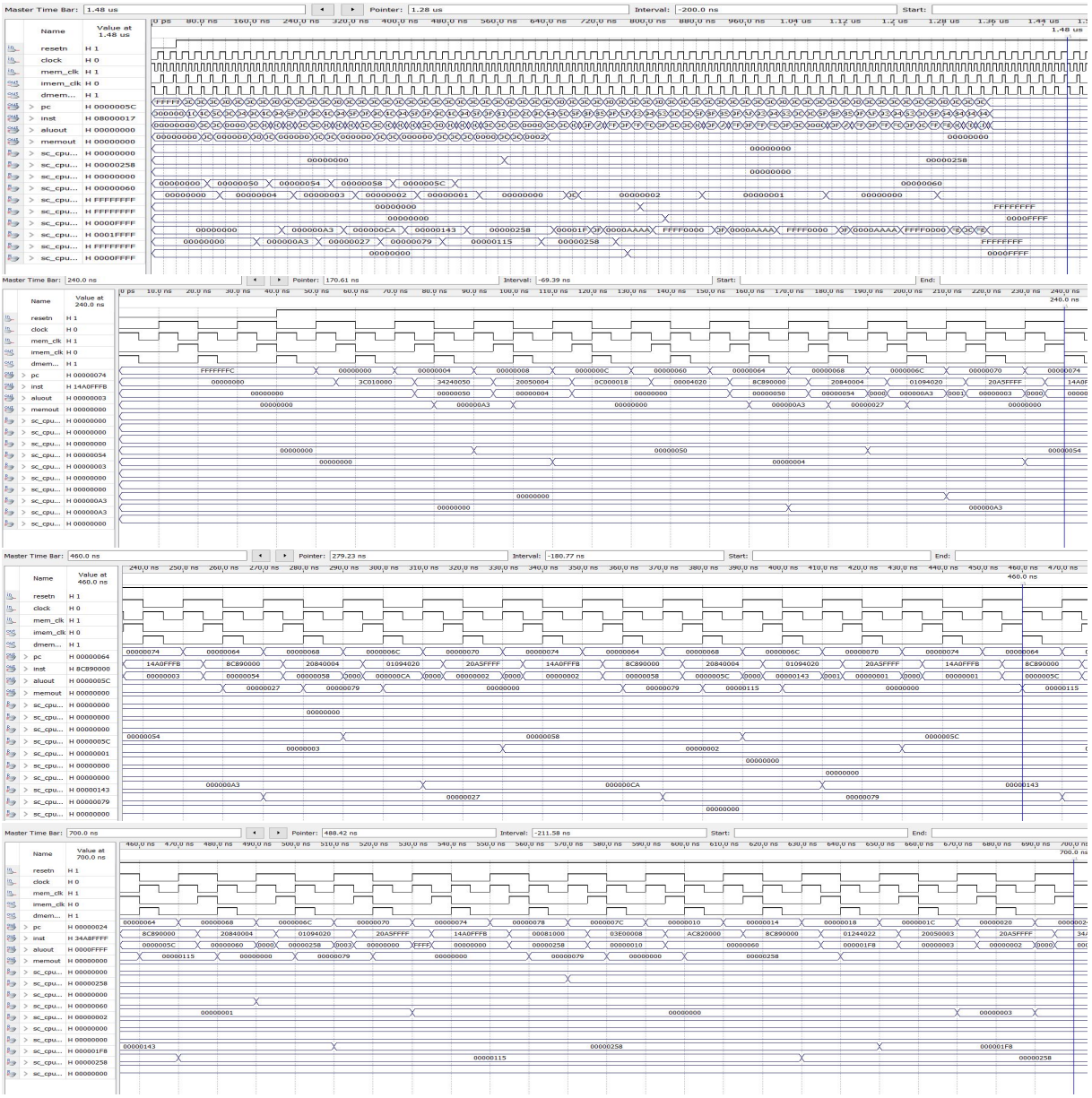

Figure 5: 具体细节图1

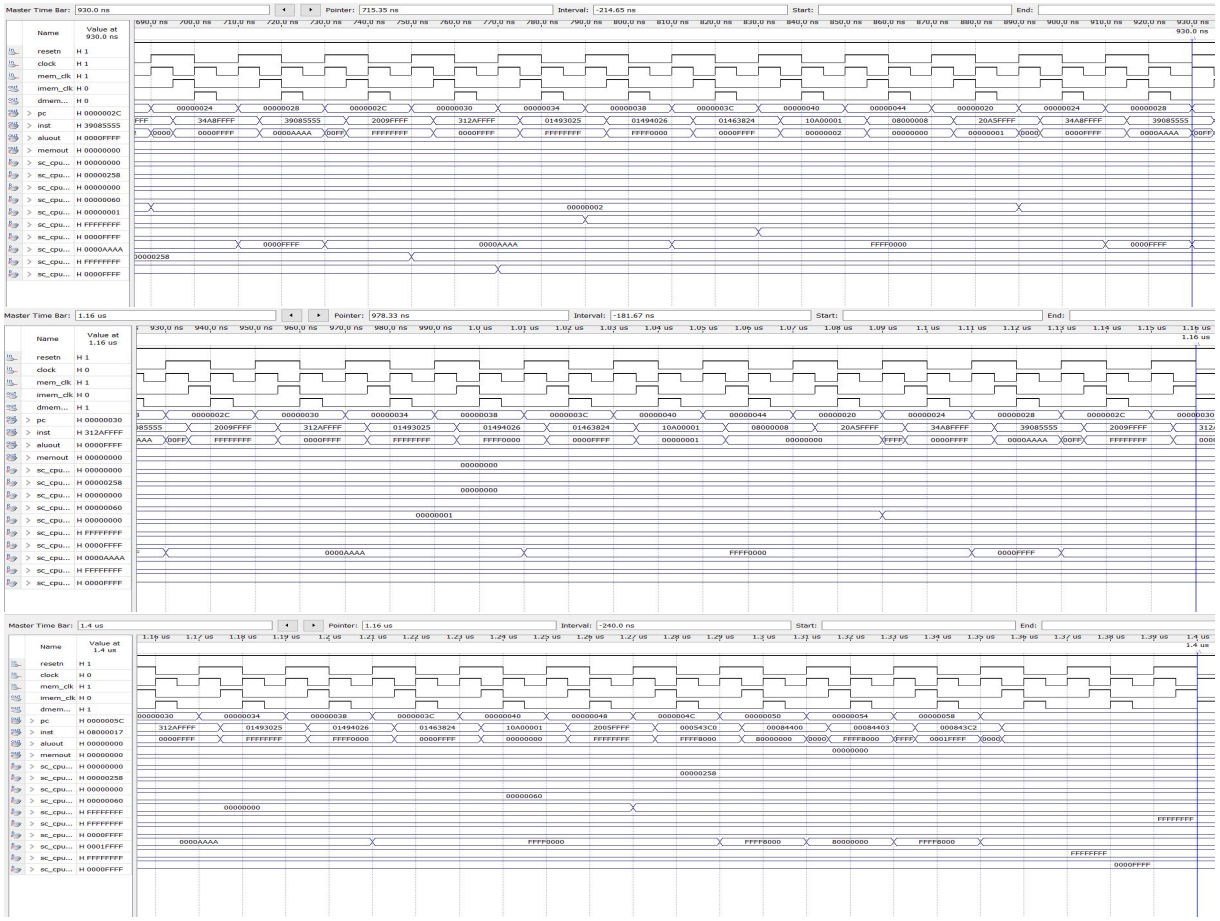

Figure 6: 具体细节图2