中山大学数据科学与计算机学院本科生实验报告

（2019 学年秋季学期）

课程名称：**计算机组成原理实验**　　任课教师：郭雪梅　　助教：汪庭葳、刘洋旗

年级&班级	2018 级计科一班	专业(方向)	计算机类
学号	18340013	姓名	陈琮昊
电话	15734867907	Email	1062080021@qq.com
开始日期	2019.11.20	完成日期	2019.11.22

一、实验题目：

实验 7 控制器实验

二、实验目的：

1.掌握单周期 CPU 控制器的工作原理及其设计方法。

2.掌握各个控制信号的作用和生成过程。

三、实验内容

1. 实验步骤：

（I）编写存储器模块：该模块有4位地址输入和32位数据输出，首先将16种类型的16条指令写入存储单元中，然后根据4位地址输入选择相应的单元指令内容，将数据写入到输出变量中。地址输入数据、存储单元编号与指令的对应规则如下表所示：

表 1 寄存器堆模块约束

地址输入	存储编号	单元数据	含义
0000	0	00000000001000100001100000100000	R-类型 add
0001	1	00000000001000100001100000100001	R-类型 addu
0010	2	00000000001000100001100000101010	R-类型 slt
0011	3	00000000001000100001100000101011	R-类型 sltu
0100	4	00000000001000000000000000001000	R-类型 jr
0101	5	00100000001000100001100000101010	I-类型 addi
0110	6	00100100001000100001100000101010	I-类型 addiu
0111	7	00111100000000100001100000101010	I-类型 lui
1000	8	10001100001000100001100000101010	I-类型 lw
1001	9	10101100001000100001100000101010	I-类型 sw
1010	10	00010000001000100001100000101010	I-类型 beq
1011	11	00010100001000100001100000101010	I-类型 bne
1100	12	00101000001000100001100000101010	I-类型 slti
1101	13	00101100001000100001100000101010	I-类型 addi
1110	14	00001000001000100001100000101010	J-类型 j

1111	15	00001100001000100001100000101010	J-类型 jal

由于该实验主要目的是对所设计的控制器进行测试，故可以不使用 IP 核设计指令存储器，直接将代码添加到 design sources 实现即可。根据输入，读取相应单元的的值输出，对相应存储单元进行写操作。

（II） 控制器模块：

控制器模块输入为指令的操作码opCode段，输出各个复用器、存储器读写等的信号，控制数据通路的正常进行。根据指令的不同，输出不同的信号即可。不同指令对应的各个控制信号总览如下：

input	opcode[5:0]
output	regDst
output	aluSrc
output	memToReg
output	regWrite
output	memRead
output	memWrite
output	branch
output	aluop[1:0]
output	jmp

（III）ALU 控制译码模块：ALU 主要执行5 种操作：与，或，加，减，小于设置。这五种操作可以使用四位的编码表示：0000，0001，0010，01110，0111。指令不同，则对应的ALU 运算不同，所以该模块需要根据指令来控制 ALU 进行正确的运算。lw，sw，addi 指令均要求 ALU 执行加操作，则可分为一类，编码 00；beq 指令要求ALU 执行减操作，则分为一类，编码01；最后一类是 R 型指令，可以编码为 10；但不同的 R 型指令对应不同的 ALU 运算，故需要再通过指令的功能码进一步确定 ALU 的运算。最终该模块即实现2 位操作码以及6 位功能码输出4 位 ALU 控制信号码。指令与ALU 操作码及功能码的对应关系如下：

指令	2位操作码	指令功能	6位功能码	ALU运算	ALU控制信号
lw	00	取字	XXXXXX	加	0010
sw	00	存字	XXXXXX	加	0010
beq	01	相等则跳转	XXXXXX	减	0110
R-type	10	加	100000	加	0010
R-type	10	减	100010	减	0110
R-type	10	与	100100	与	0000
R-type	10	或	100101	或	0001
R-type	10	小于则置1	101010	小于设置	0011

（IV）编写主模块：主模块有 4 位地址输入和 16 位输出，主模块首先调用寄存器堆模块生成 32 位指令信号，然后调用控制器模块生成相应的控制信号，将这 16 位控制信号赋值到输出变量中。

（V）依次综合项目源文件、实施活动的设计合成的网表。综合的目的是检查代码是否存在语法错误，生成网表后可以绑定相应的管脚。

（VI）添加相关仿真文件进行仿真。

2. 实验原理：控制器的基本任务是完成当前指令的翻译和执行，即将当前指令的功能转换成可以控制的硬件逻辑部件工作的命令序列，完成数据传送和各种处理操作。本实验根据MIPS的31种类型的指令以及根据该指令结构所设计出的数据通路模拟其中控制单元生成控制信号的过程。其中MIPS的31条指令类型如表1所示：

表2　MIPS的31种指令

助记符	指　令　格　式						示　例	示例含义	操作及解释
BIT #	31..26	25..21	20..16	15..11	10..6	5..0			
R-类型	**op**	**rs**	**rt**	**rd**	**shamt**	**func**			
add	000000	rs	rt	rd	00000	100000	add $1,$2,$3	$1=$2+S3	(rd)←(rs)+(rt); rs=$2,rt=$3,rd=$1
addu	000000	rs	rt	rd	00000	100001	addu $1,$2,$3	$1=$2+S3	(rd)←(rs)+(rt); rs=$2,rt=$3,rd=$1,无符号数
sub	000000	rs	rt	rd	00000	100010	sub $1,$2,$3	$1=$2-S3	(rd)←(rs)-(rt); rs=$2,rt=$3,rd=$1
subu	000000	rs	rt	rd	00000	100011	subu $1,$2,$3	$1=$2-S3	(rd)←(rs)-(rt); rs=$2,rt=$3,rd=$1,无符号数
and	000000	rs	rt	rd	00000	100100	and $1,$2,$3	$1=$2&S3	(rd)←(rs)&(rt); rs=$2,rt=$3,rd=$1
or	000000	rs	rt	rd	00000	100101	or $1,$2,$3	$1=$2|S3	(rd)←(rs) | (rt); rs=$2,rt=$3,rd=$1
xor	000000	rs	rt	rd	00000	100110	xor $1,$2,$3	$1=$2^S3	(rd)←(rs)^(rt); rs=$2,rt=$3,rd=$1
nor	000000	rs	rt	rd	00000	100111	nor $1,$2,$3	$1= ~($2 | S3)	(rd)←~((rs) | (rt)); rs=$2,rt=$3,rd=$1
slt	000000	rs	rt	rd	00000	101010	slt $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs< rt) rd=1 else rd=0;rs＝$2，rt=$3, rd=$1
sltu	000000	rs	rt	rd	00000	101011	sltu $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs< rt) rd=1 else rd=0;rs＝$2，rt=$3, rd=$1,无符号数
sll	000000	00000	rt	rd	shamt	000000	sll $1,$2,10	$1=$2<<10	(rd)←(rt)<<shamt,rt=$2,rd=$1,shamt=10
srl	000000	00000	rt	rd	shamt	000010	srl $1,$2,10	$1=$2>>10	(rd) ← (rt)>>shamt, rt=$2, rd=$1, shamt=10,(逻辑右移)
sra	000000	00000	rt	rd	shamt	000011	sra $1,$2,10	$1=$2>>10	(rd) ← (rt)>>shamt, rt=$2, rd=$1, shamt=10,(算术右移，注意符号位保留)
sllv	000000	rs	rt	rd	00000	000100	sllv $1,$2,$3	$1=$2<<$3	(rd)←(rt)<<(rs), rs=$3,rt=$2,rd=$1
srlv	000000	rs	rt	rd	00000	000110	srlv $1,$2,$3	$1=$2>>$3	(rd) ← (rt)>>(rs), rs=$3,rt=$2,rd=$1, (逻辑右移)
srav	000000	rs	rt	rd	00000	000111	srav $1,$2,$3	$1=$2>>$3	(rd) ← (rt)>>(rs), rs=$3,rt=$2,rd=$1, (算术右移，注意符号位保留)
jr	000000	rs	00000	00000	00000	001000	jr $31	goto $31	(PC)←(rs)
I-类型	**op**	**rs**	**rt**	**immediate**					
addi	001000	rs	rt	immediate			addi $1,$2,10	$1=$2+10	(rt) ← (rs)+(sign-extend)immediate,rt=$1,rs=$2
addiu	001001	rs	rt	immediate			addiu $1,$2,10	$1=$2+10	(rt) ← (rs)+(sign-extend)immediate,rt=$1,rs=$2
andi	001100	rs	rt	immediate			andi $1,$2,10	$1=$2&10	(rt) ← (rs)&(zero-extend)immediate,rt=$1,rs=$2
ori	001101	rs	rt	immediate			ori $1,$2,10	$1=$2|10	(rt)←(rs)|(zero-extend)immediate,rt=$1,rs=$2
xori	001110	rs	rt	immediate			xori $1,$2,10	$1=$2^10	(rt) ← (rs)^(zero-extend)immediate,rt=$1,rs=$2

lui	001111	00000	rt	immediate	lui $1,10	$1=10*65536	(rt)←immediate<<16 & 0FFFF0000H，将 16 位立即数放到目的寄存器高 16 位，目的寄存器的低 16 位填 0
lw	100011	rs	rt	offset	lw $1,10($2)	$1=Memory[$2+10]	(rt)←Memory[(rs)+(sign_extend)offset], rt=$1,rs=$2
sw	101011	rs	rt	offset	sw $1,10($2)	Memory[$2+10] =$1	Memory[(rs)+(sign_extend)offset]←(rt), rt=$1,rs=$2
beq	000100	rs	rt	offset	beq $1,$2,40	if($1=$2) goto PC+4+40	if ((rt)=(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2), rs=$1, rt=$2
bne	000101	rs	rt	offset	bne $1,$2,40	if($1≠$2) goto PC+4+40	if ((rt)≠(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2) , rs=$1, rt=$2
slti	001010	rs	rt	immediate	slti $1,$2,10	if($2<10) $1=1 else $1=0	if ((rs)<(Sign-Extend)immediate) then (rt)←1; else (rt)←0, rs=$2, rt=$1
sltiu	001011	rs	rt	immediate	sltiu $1,$2,10	if($2<10) $1=1 else $1=0	if ((rs)<(Zero-Extend)immediate) then (rt)←1; else (rt)←0, rs=$2, rt=$1
J-类型	**op**	**address**					
j	000010	address			j 10000	goto 10000	(PC)←((Zero-Extend) address<<2), address=10000/4
jal	000011	address			jal 10000	$31=PC+4 goto 10000	($31)←(PC)+4; (PC)←((Zero-Extend) address<<2), address=10000/4

由上表可知该指令主要分为三大类，分别为 R-类型、I-类型和 J-类型。每类指令对应的控制信号也不相同，根据这些指令所设计的数据通路如图 1 所示：

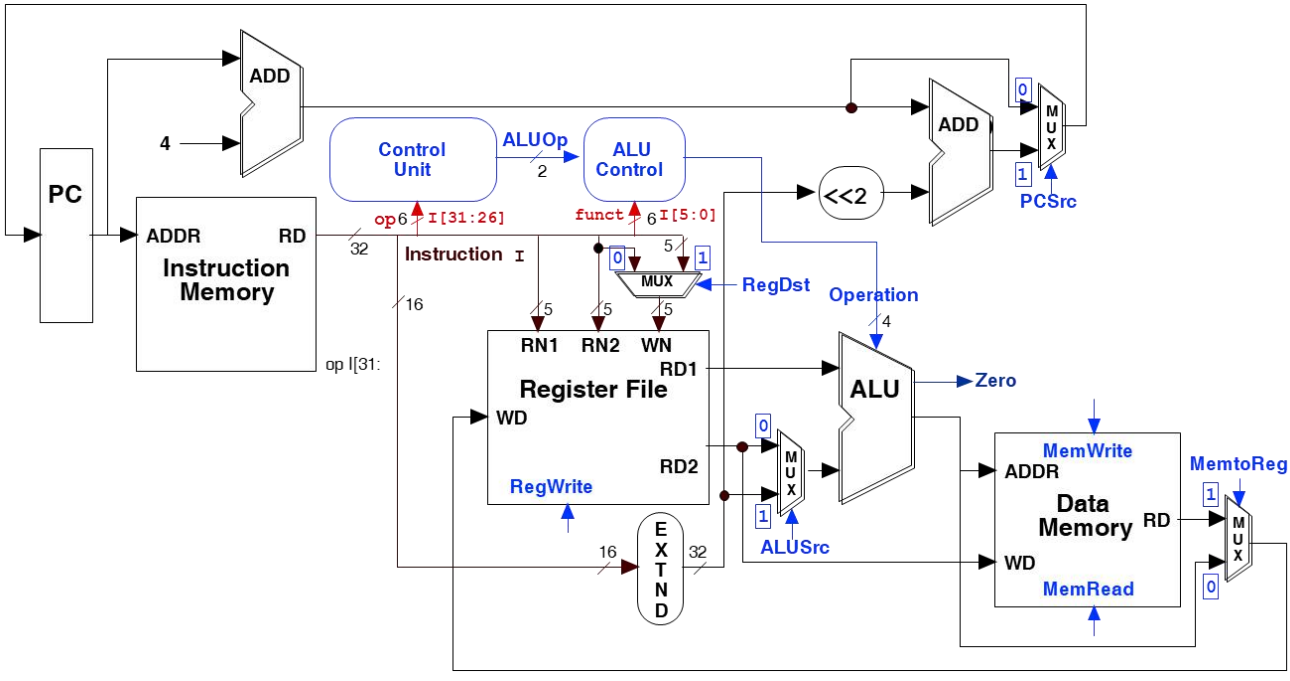


图 1 单周期 CPU 数据通路

本次实验是主要模拟指令生成控制信号的过程，本次实验根据输入指令从寄存器堆中选取相应的寄存器，然后读取该寄存器获得相应 32 位指令，取其高 5 位和低 5 位送入控制器单元，由控制单元进行译码，输出相应的控制信号。该控制器组成原理图如下图所示：
Control（控制单元）：根据指令中的指令码（op）和功能码（funct）的不同组合输出相应的控制信号。

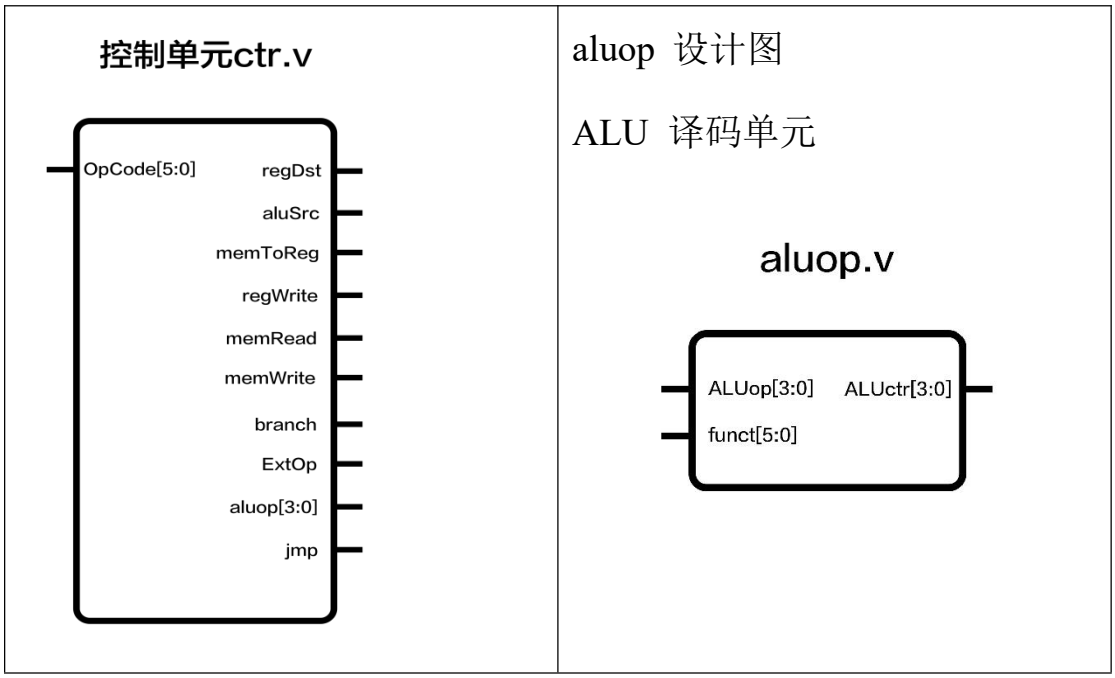
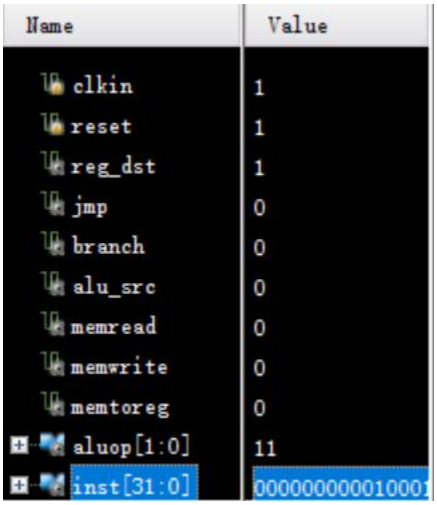


图 2　控制器组成原理图

该控制器的存储模块有 16 个存储单元，每一个单元都预先分配了一条指令信息，4 位地址输入读取相应单元指令。控制器主要实现对输入的 10 位指令操作码的解析。

四、实验结果：

仿真结果如下：以下为显示的参数：



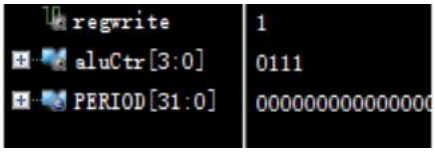
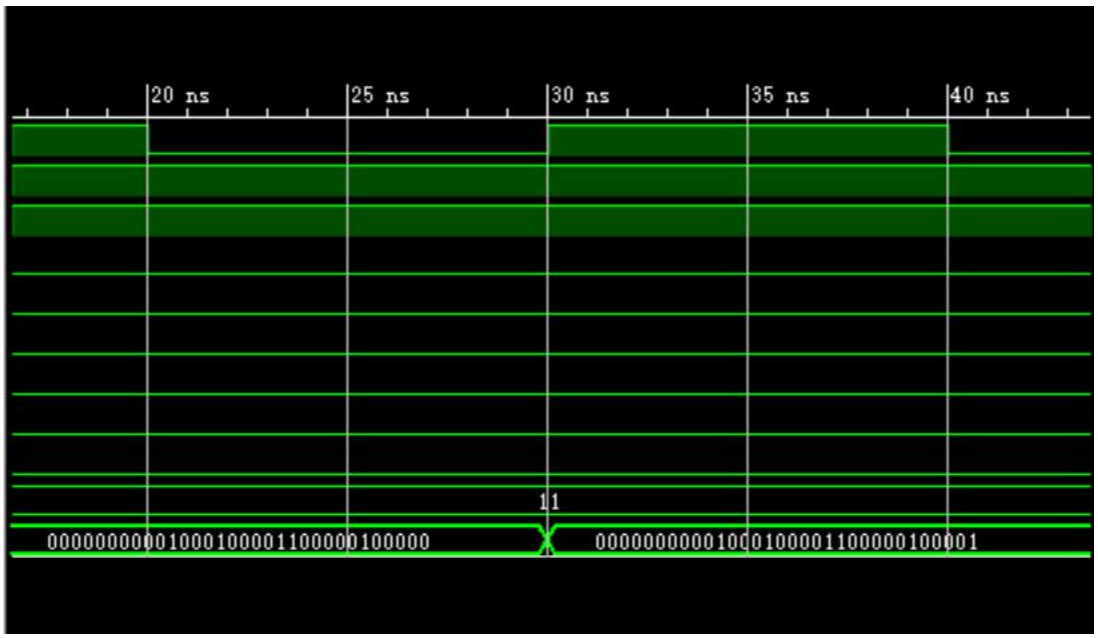
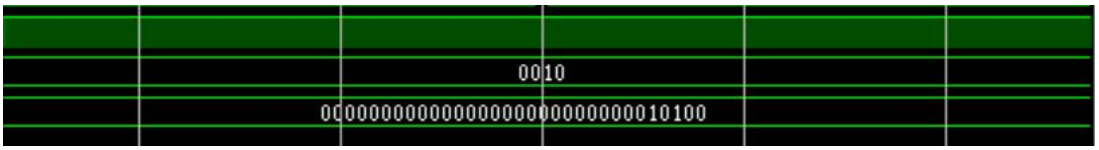


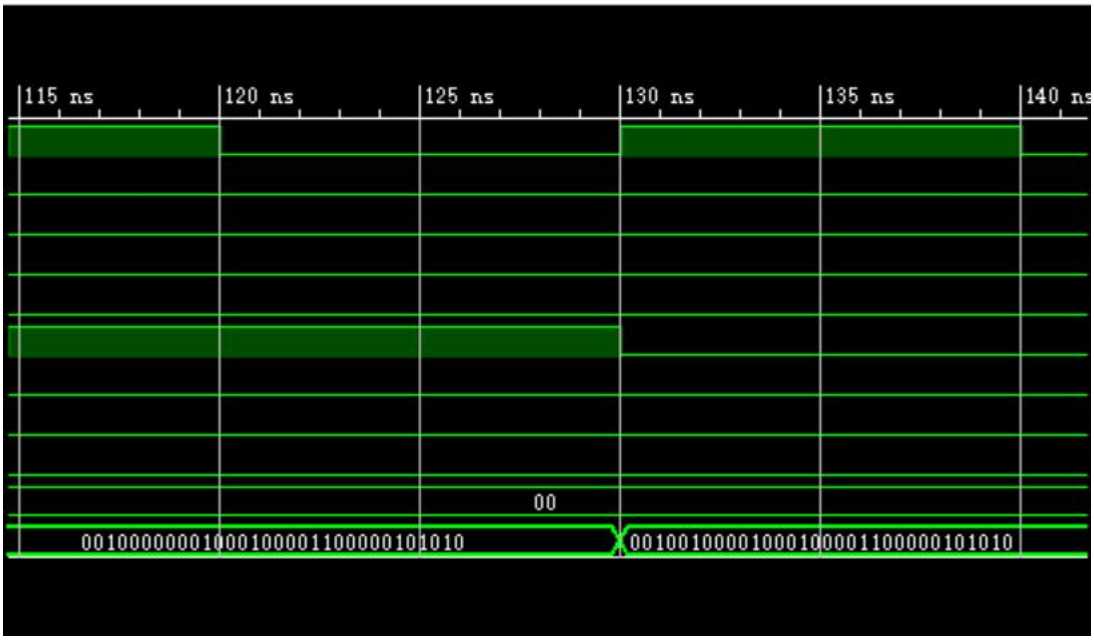
图 3　　图 4

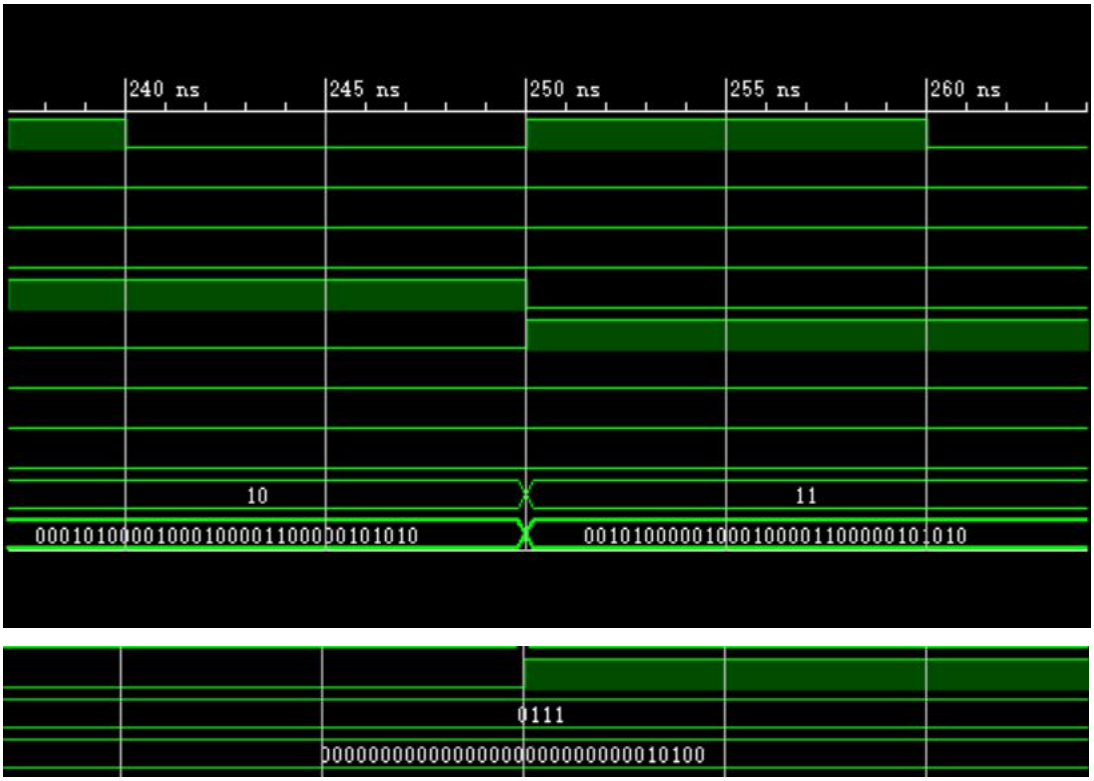
下面是仿真结果（列出了一部分）：

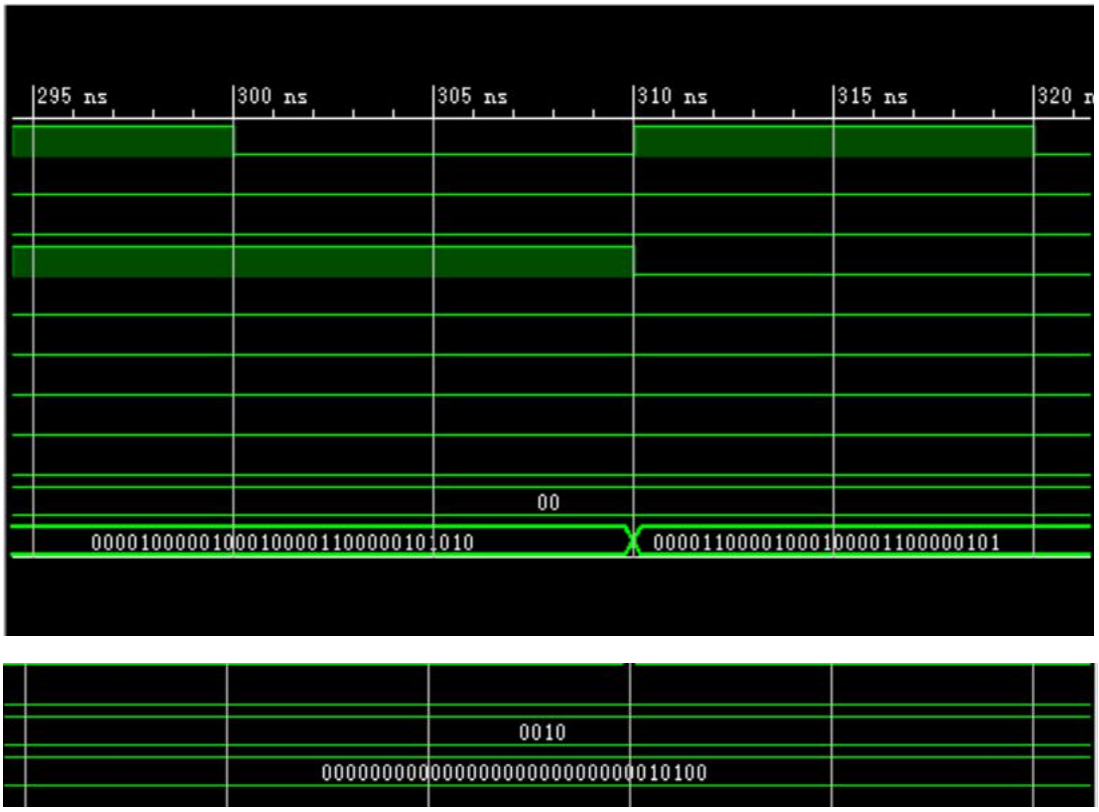
每两张为一组，第一张显示的是图 3 的对应内容，第二张显示的是图 4 的对应内容。

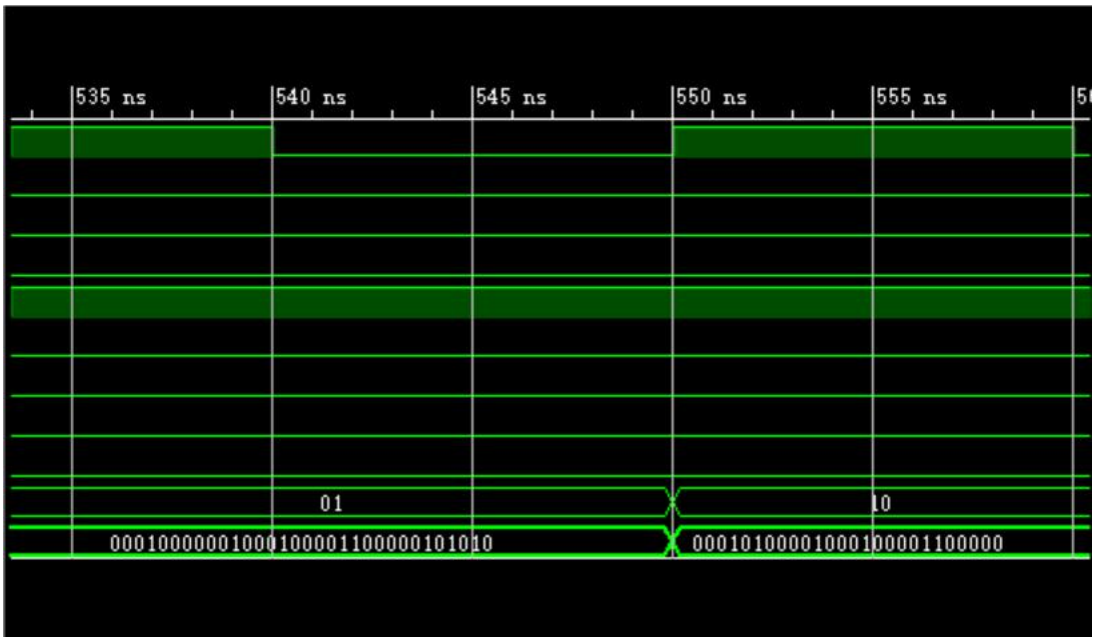







240 ns
245 ns
250 ns
255 ns
260 ns
10
11
00010100001000100001100000101010
00101000001000100001100000101010
0111
00000000000000000000000000010100


295 ns
300 ns
305 ns
310 ns
315 ns
320 n
00
00001000001000100001100000101010
00001100001000100001100000101
0010
00000000000000000000000000010100



进行比对可知实验结果正确。

五、实验感想：本次实验主要是设计控制器，有一点比较好的就是不需要烧板，只进行仿真即可。按照给出的代码进行初步仿真没有任何问题；但要进行下一步的仿真需要修改代码的 top.v 等相关部分来成功输出 32 位指令译码等相关内容，这部分是难点，需要多次调试以及熟悉 Verilog 语言。

附录：

(I)top.v：

```
module top(
input clkin, input reset,
output reg_dst, jmp, branch, alu_src, memread, memwrite, memtoreg,[1:0] aluop,
output [31:0] inst,
output regwrite,
output[3:0] aluCtr
);
```

```
// 指令寄存器 Addr
reg[3:0] Addr;
// CPU 控制信号线
// ALU 控制信号线
// 例化寄存器
regFile regFile ( .clk(clkin), .Addr(Addr), .instruction(inst)
);
// 实例化控制器模块
ctr mainctr(
.opCode(inst[31:26]),
.regDst(reg_dst),
.aluSrc(alu_src),
.memToReg(memtoreg),
.regWrite(regwrite),
.memRead(memread),
.memWrite(memwrite),
.branch(branch),
.aluop(aluop),
.jmp(jmp));
//  实例化 ALU 控制模块
aluctr aluctr1(
.ALUOp(aluop),
.funct(inst[5:0]),
.ALUCtr(aluCtr));
```

```
endmodule
```

(II)regFile.v:

```
module regFile( input clk,
input [3:0] Addr,       //寄存器地址编码
output reg [31:0] instruction//  寄存器的值
);
//寄存器地址都是 4 位二进制数，因为寄存器只有 16 个，4 位就能表示所有寄存器
reg [31:0] regs [0:15]; //  寄存器组
integer i = 0;
always @( posedge clk ) //  时钟上升沿操作
begin
instruction=regs[i] ; //   取指令
i = i + 1;
if(i == 16) i = 0;
      end
initial
          begin
              regs[0]    = 32'b00000000001000100001100000100000;// sltu
              regs[1]    = 32'b00000000001000100001100000100001;// slt
              regs[2]    = 32'b00000000001000100001100000101010;// addu
              regs[3]    = 32'b00000000001000100001100000101011;// add
               regs[4]    = 32'b00000000001000000000000000001000;// jr
               regs[5]    = 32'b00100000001000100001100000101010;// addi
               regs[6]    = 32'b00100100001000100001100000101010;// addiu
```

```
regs[7]    = 32'b00111100000000100001100000101010;// lui
regs[8]    = 32'b10001100001000100001100000101010;// lw
regs[9]    = 32'b10101100001000100001100000101010;// sw
regs[10]   = 32'b00010000001000100001100000101010;//beq
regs[11]   = 32'b00010100001000100001100000101010;//bne
regs[12]   = 32'b00101000001000100001100000101010;// slti
regs[13]   = 32'b00101100001000100001100000101010;// addi
regs[14]   = 32'b00001000001000100001100000101010;//j
regs[15]   = 32'b00001100001000100001100000101010;//jal
end
endmodule
```

(III)ctr.v:

```
module ctr(
input [5:0] opCode, output reg regDst, output reg aluSrc, output reg memToReg, output reg regWrite, output reg memRead, output reg memWrite, output reg branch,
output reg ExtOp, //符号扩展方式，1 为 sign-extend，0 为 zero-extend
output reg[3:0] aluop, // 经过 ALU 控制译码决定 ALU 功能
output reg jmp
);
always@(opCode) begin
// 操作码改变时改变控制信号
case(opCode) 6'b000010: begin
regDst = 0;    aluSrc = 0; memToReg = 0;
regWrite = 0; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0000; jmp = 1; ExtOp = 1;
```

```
end // 'J 型' 指令操作码: 000010，无需 ALU
6'b000000: begin
regDst = 1;    aluSrc = 0; memToReg = 0;
regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b1111; jmp = 0; ExtOp = 1;
end // 'R 型'指令操作码: 000000
//'I'型指令操作码
6'b100011: begin
regDst = 0;    aluSrc = 1; memToReg = 1;
regWrite = 1; memRead = 1; memWrite = 0; branch = 0; aluop = 4'b0000; jmp = 0; ExtOp = 1;
end // 'lw' 指令操作码: 100011
6'b101011: begin
regDst = 0;    aluSrc = 1; memToReg = 0;
regWrite = 0; memRead = 0; memWrite = 1; branch = 0; aluop = 4'b0000; jmp = 0; ExtOp = 1;
end // 'sw' 指令操作码: 101011
6'b000100: begin
regDst = 0;    aluSrc = 0; memToReg = 0;
regWrite = 0; memRead = 0; memWrite = 0; branch = 1; aluop = 4'b0001; jmp = 0; ExtOp = 1;
end // 'beq' 指令操作码: 000100
6'b000101: begin
regDst = 0;    aluSrc = 0; memToReg = 0;
regWrite = 0; memRead = 0; memWrite = 0; branch = 1; aluop = 4'b0110; jmp = 0; ExtOp = 1;
end // 'bne' 指令操作码: 000101
6'b000101: begin
regDst = 0;    aluSrc = 0; memToReg = 0;
```

regWrite = 0; memRead = 0; memWrite = 0; branch = 1; aluop = 4'b0001; jmp = 0; ExtOp = 1;

end // 'bne' 指令操作码: 000101

6'b001000: begin

regDst = 0; aluSrc = 1; memToReg = 0;

regWrite = 1; memRead = 0; memWrite = 0;

branch = 0; aluop = 4'b0000; jmp = 0;

ExtOp = 1;

end // 'addi' 指令操作码: 001000

6'b001100: begin

regDst = 0; aluSrc = 1; memToReg = 0;

regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0100; jmp = 0; ExtOp = 0;

end // 'andi' 指令操作码: 001100

6'b001101: begin

regDst = 0; aluSrc = 1; memToReg = 0;

regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0010; jmp = 0; ExtOp = 0;

end // 'ori' 指令操作码: 001101

6'b001110: begin

regDst = 0; aluSrc = 1; memToReg = 0;

regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b1100; jmp = 0; ExtOp = 0;

end // 'xori' 指令操作码: 001110

6'b001010: begin

regDst = 0; aluSrc = 1; memToReg = 0;

regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0011; jmp = 0; ExtOp = 1;

end // 'slti' 指令操作码: 001010

```
6'b001111: begin
regDst = 0;    aluSrc = 1; memToReg = 0;
regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b1011; jmp = 0; ExtOp = 1;
end // 'lui'  指令操作码: 001111
default: begin
regDst = 0;    aluSrc = 0; memToReg = 0;
regWrite = 0; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0000; jmp = 0; ExtOp = 0;
end //  默认设置
endcase end
endmodule

module aluctr(
input [1:0] ALUOp, input [5:0] funct, output reg [3:0]    ALUCtr
);
```

(IV)aluctr.v:

```
always @(ALUOp or funct) //    如果操作码或者功能码变化执行操作
casex({ALUOp, funct}) //  拼接操作码和功能码便于下一步的判断
8'b00xxxxxx: ALUCtr = 4'b0010; // lw，sw,addi
8'b01xxxxxx: ALUCtr = 4'b0110; // beq
8'b1xxx0000: ALUCtr = 4'b0010; // add
8'b1xxx0010: ALUCtr = 4'b0110; // sub
8'b1xxx0100: ALUCtr = 4'b0000; // and
8'b1xxx0101: ALUCtr = 4'b0001; // or
8'b1xxx1010: ALUCtr = 4'b0111; // slt
```

```
endcase endmodule
```

(V) topsim.v:

```
module topsim;
// Inputs
reg clkin;
reg reset;
wire reg_dst, jmp, branch, alu_src, memread, memwrite, memtoreg;
wire[1:0] aluop;
wire [31:0] inst;
wire regwrite;
wire[3:0] aluCtr;
// Instantiate the Unit Under Test (UUT)
top uut (
.clkin(clkin),
.reset(reset),
.reg_dst(reg_dst),
.jmp(jmp),
.branch(branch),
.alu_src(alu_src),
.memread(memread),
.memwrite(memwrite),
.memtoreg(memtoreg),
.aluop(aluop),
.inst(inst),
```

```
.regwrite(regwrite),
.aluCtr(aluCtr)
);
initial begin
// Initialize Inputs
clkin = 0;
reset = 1;
// Wait 100 ns for global reset to finish
#100;
reset = 0;
end
parameter PERIOD = 20;
always begin
clkin = 1'b0;
#(PERIOD / 2) clkin = 1'b1;
#(PERIOD / 2) ;
end
endmodule
```

(VI)aluctrsim.v:

```
module aluctrsim;
// Inputs
reg [1:0] ALUOp;
reg [5:0] funct;
// Outputs
```

```
wire [3:0] ALUctr;

// Instantiate the Unit Under Test (UUT)

aluctr uut (

.ALUOp(ALUOp),

.funct(funct),

.ALUCtr(ALUctr)

);

initial begin

// Initialize Inputs

ALUOp = 0;

funct = 0;

// Wait 100 ns for global reset to finish

#100;

// Add stimulus here

ALUOp = 2'b01;

funct = 0;

#100;

ALUOp = 2'b10;

funct = 6'b100000;

#100;

ALUOp = 2'b10;

funct = 6'b101010;

end

endmodule
```

(VII)ctrsim.v:

```
module ctrsim;

// Inputs

reg [5:0] opCode;

// Outputs

wire regDst;

wire aluSrc;

wire memToReg;

wire regWrite;

wire memRead;

wire memWrite;

wire branch;

wire [1:0] aluop;

wire jmp;

// Instantiate the Unit Under Test (UUT)

ctr uut (

.opCode(opCode),

.regDst(regDst),

.aluSrc(aluSrc),

.memToReg(memToReg),

.regWrite(regWrite),

.memRead(memRead),

.memWrite(memWrite),

.branch(branch),
```

```
.aluop(aluop),
.jmp(jmp)
);
initial begin
// Initialize Inputs
opCode = 6'b000000;
#100;
opCode = 6'b000010;
// Wait 100 ns for global reset to finish
#100;
opCode = 6'b110000;
#100;
opCode = 6'b100011;
#100;
opCode = 6'b000100;
#100;
opCode = 6'b001000;
end
endmodule
```