计算机系统结构实验

lab3: 类 MIPS 单周期处理器功能部件的设计与实现(一)

周旭东 521021910829

2023 April

摘要

在理解 MIPS 指令结构的基础之上，学习主控制部件或单元、ALU 控制器单元、ALU 单元的原理，并根据 MIPS 指令集设计支持九条指令（beq,lw,sw,add,sub,and,or,slt,jump）的三个单元模块，随后对实现的各个模块进行仿真测试。

目录

1 实验目的

1. 理解主控制部件或单元、ALU 控制器单元、ALU 单元的原理
2. 熟悉所需的 Mips 指令集
3. 使用 Verilog HD 设计与实现主控制器部件 (Ctr)
4. 使用 Verilog 设计与实现 ALU 控制器部件 (ALUCtr)
5. ALU 功能部件的实现
6. 使用 Vivado 进行功能模块的行为仿真

2 实验原理

2.1 MIPS 指令结构

MIPS 是一种经典的 RISC（精简指令集计算机）体系结构，图 1给出其指令格式的三种类型。

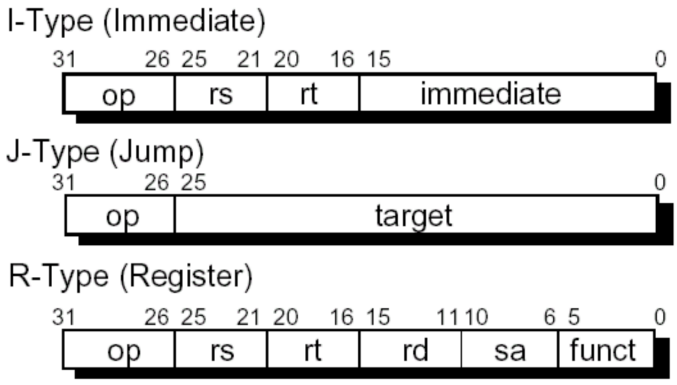

图 1: MIPS 指令的三种格式

R-Type 指令格式：R-Type 指令格式用于表示寄存器之间的操作，如加、减、与、或等。其格式如下：

- op：操作码，用于指定该指令的操作类型；
- rs、rt、rd：寄存器编号，分别表示源操作数寄存器、目标操作数寄存器和结果寄存器的编号；
- sa：移位量，表示在进行移位操作时移动的位数；

- funct：函数码，用于指定具体的操作类型，如加、减、与、或等。

I-Type 指令格式：I-Type 指令格式用于表示立即数和寄存器之间的操作，如加载、存储、分支等。

- op：操作码，用于指定该指令的操作类型；
- rs、rt：寄存器编号，分别表示源操作数寄存器和目标操作数寄存器的编号；
- immediate：立即数，表示该指令要操作的立即数值。

J-Type 指令格式：J-Type 指令格式用于表示跳转指令。其格式如下：

- op：操作码，用于指定该指令的操作类型；
- target：跳转目标地址，表示该指令要跳转到的地址。

助记符	指令格式						示例	操作及其解释
Bit #	31..26	25..21	20..16	15..11	10..6	5..0		
R–type	op	rs	rt	rd	shamt	func		
add	000000	rs	rt	rd	0	100000	add $1,$2,$3	rd <– rs + rt ； 其中rs＝$2，rt=$3, rd=$1
sub	000000	rs	rt	rd	0	100010	sub $1,$2,$3	rd <– rs – rt ； 其中rs＝$2，rt=$3, rd=$1
and	000000	rs	rt	rd	0	100100	and $1,$2,$3	rd <– rs & rt ； 其中rs＝$2，rt=$3, rd=$1
or	000000	rs	rt	rd	0	100101	or $1,$2,$3	rd <– rs | rt ； 其中rs＝$2，rt=$3, rd=$1
slt	000000	rs	rt	rd	0	101010	slt $1,$2,$3	if (rs < rt) rd=1 else rd=0 ； 其中rs＝$2，rt=$3, rd=$1
I–type	op	rs	rt	immediate				
lw	100011	rs	rt	immediate			lw $1,10($2)	rt <– memory[rs + (sign–extend)immediate] ； rt=$1,rs=$2
sw	101011	rs	rt	immediate			sw $1,10($2)	memory[rs + (sign–extend)immediate] <– rt ； rt=$1,rs=$2
beq	000100	rs	rt	immediate			beq $1,$2,10	if (rs == rt) PC <– PC+4 + (sign–extend)immediate<<2
J–type	op	address						
j	000010	address					j 10000	PC <– (PC+4)[31..28],address,0,0 ； address=10000/4

表 1: 支持的 9 条指令及其具体信息

表 1列出了本次实验需要支持的九条指令，其中，有五条指令为 R-Type，三条指令为 I-Type，一条指令为 J-Type。

2.2 主控制器 Control Unit 原理

主控制部件（Control Unit）是计算机中一个重要的组成部分，它主要负责控制 CPU 中的各个部件以及指令执行的流程。主控制部件根据指令的操作码（opcode）来确定该指令的操作类型，并产生相应的控制信号，以控制指令执行的流程，例如控制存储器的读写、寄存器的读写、ALU 的操作等。

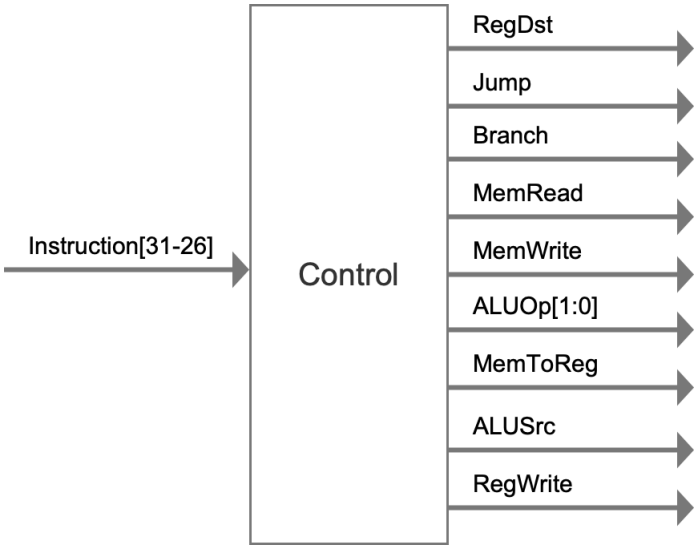

图 2: 主控制器结构

图 2展示了主控制器的结构。主控制器模块通过指令第 26-31 位获取代码指令及类型信息，根据指令给出各个位置的控制信号，并向 ALU 发出 ALUOp 信号，指示其进行相应运算操作。

RegDst 信号用于选择目标寄存器的编号。Jump 信号用于指示 CPU 是否需要进行无条件跳转。Branch 信号用于控制分支操作，判断分支是否需要被执行。MemRead 信号和 MemWrite 信号用于控制存储器的读和写操作。ALUOp 信号用于选择 ALU 运算类型，决定 CPU 要执行哪种运算。ALUSrc 信号用于选择 ALU 第二个操作数的来源，可以是寄存器或者立即数。MemToReg 信号用于选择从存储器中读取数据还是从 ALU 中读取数据写入寄存器。RegWrite 信号用于控制寄存器的写操作。

表 2定义了主控制器模块各个输入输出信号量的定义并给出了相应功能的描述。

输入/输出量	信号线	描述	定义的信号量
输入	Instruction（5位）	指令的第 31-26 位，决定指令类型	opCode
输出	RegDst	目标寄存器选择信号	regDst
	Jump	无条件跳转信号	jump
	Branch	分支控制信号	branch
	MemRead	存储器读信号	memRead
	MemWrite	存储器写信号	memWrite
	ALUOp（2位）	ALU运算类型	aluOp
	ALUSrc	ALU 第二个操作数来源选择信号	aluSrc
	MemToReg	从存储器写到寄存器选择信号	memToReg
	RegWrite	寄存器写信号	regWrite

表 2: 主控制器各信号量及其定义

支持的九条指令及其对应控制信号由表 3给出。

Inst	OpCode	ALUop	RegDst	ALUsrc	MemTReg	RegWrite	MemRead	MemWrite	Branch	Jump
lw	100011	00	0	1	1	1	1	0	0	0
sw	101011	00	0	1	0	0	0	1	0	0
beq	000100	01	0	0	0	0	0	0	1	0
j	000010	00	0	0	0	0	0	0	0	1
Rtype	000000	10	1	0	0	1	0	0	0	0

表 3: 不同指令对应信号

2.3 ALU 控制器原理

ALU 控制器单元（ALU Control Unit）是控制 ALU（算术逻辑单元）操作的部件，它接收主控制部件发出的指令操作码，并根据操作码中的功能字段（function field）产生相应的控制信号，以控制 ALU 执行相应的操作。例如，当指令是加法指令时，ALU 控制器单元会将 ALU 的控制信号设置为加法操作，从而实现加法运算。

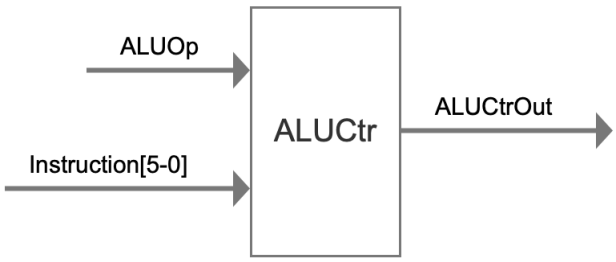

图 3: ALU 控制器结构

图 3展示了 ALU 控制器的结构。ALU 控制器通过主控制器给出的 ALUOp，结合指令第 0-5 位给出的 funct 信息，确定 16 条指令具体对应的 ALU 运算类型（R type 的解析无法在 Ctr 中完成）。

ALUCtr 模块接收两个输入信号，分别为 ALUOp 和 Instruction。其中，ALUOp 是 2 位的信号线，用来指定 ALU（算术逻辑单元）的运算类型指令；Instruction[5-0] 则是指令的第 0-5 位，用来提供 ALU 运算的具体操作码 funct。

在接收到这两个输入信号后，该电路会输出一个 4 位的信号线 ALUCtrOut，用于指示解析后的 ALU 具体运算类型。同时，该电路还会定义两个信号量，分别为 aluOp 和 funct，用来表示输入信号的具体值。其中，aluOp 表示 ALUOp 的值，funct 表示 Instruction 中的 funct 值。

表 4定义了 ALU 控制器模块各个输入输出信号量的定义并给出了相应功能的描述。

输入/输出量	信号线	描述	定义的信号量
输入	ALUOp（2位）	ALU运算类型指令	aluOp
	Instruction[5-0]	指令第0-5位，给出funct	funct
输出	ALUCtrOut（4位）	解析后的ALU具体运算类型	aluCtrOut

表 4: ALU 控制器各信号量及其定义

2.4 逻辑运算单元 ALU 原理

ALU 单元（Arithmetic Logic Unit）是 CPU 中一个重要的功能部件，主要用于执行算术和逻辑操作。ALU 通常由多个加法器、多路选择器、逻辑门等部件组成，它能够执行诸如加、减、与、或等算术和逻辑运算。ALU 接收来自寄存器的操作数，并根据 ALU 控制器单元发出的控制信号执行相应的运算，最终将结果写回寄存器或者其他目的地。

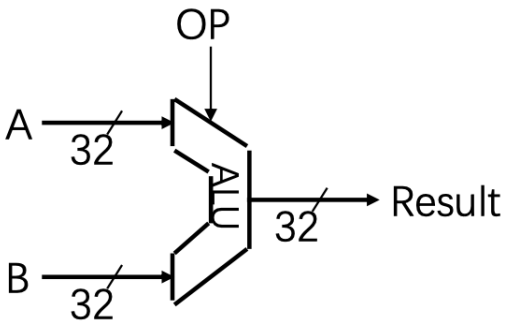

图 4: ALU 结构

图 4展示了 ALU 的结构。ALU 接收两个 32 位的输入数据（InputA 和 InputB），根据控制信号（ALUOp 和 aluCtr）选择相应的算术或逻辑运算类型，然后执行运算并输出运算结果（Output）。控制信号还会产生一个标志信号（ZeroSign），指示运算结果是否为零。

表 5定义了 ALU 模块各个输入输出信号量的定义并给出了相应功能的描述。

输入/输出量	信号线	描述	定义的信号量
输入	ALUOp（4位）	ALU运算类型指令	input1
	InputA（32位）	第一个32位输入	input2
	InputB（32位）	第二个32位输入	aluCtr
输出	Output（32位）	进行相应运算后的32位输出	aluRes
	ZeroSign	标志运算结果是否为0	zero

表 5: ALU 各信号量及其定义

表 6定义了 ALU 模块对各个 ALUOp 信号下的运算方式及其代码实现。其中，slt 比较两个有符号整数 input1 和 input2 的大小，如果 input1 小于 input2，则结果为 1，否则结果为 0。具体地，$signed() 是一个 SystemVerilog 函数，用于将无符号数转换为有符号数，以便进行带符号的比较。因此，$signed(input1) 和 $signed(input2) 将无符号输入转换为有符号数。然后，这两个有符号数被比较，比较结果被存储在 ALURes 变量中。

ALUCtr	ALU操作	实现代码
0010	add	ALURes = input1 + input2
0110	sub	ALURes = input1 - input2
0001	or	ALURes = input1 | input2
0000	and	ALURes = input1 & input2
1100	nor	ALURes = ~(input1 | input2)
0111	slt	ALURes = ($signed(input1) < $signed(input2))

表 6: 各 ALUOp 对应的 ALU 操作及其实现

3 功能实现

3.1 Ctr 模块

根据 Control Unit 原理描述与指令表格设计 Ctr 模块代码，模块内部定义了 10 个寄存器，用来存储不同 opCode 信号值对应的输出信号。

使用 opCode 作为 always 条件，每当得到信号则进行解析。根据 opCode 信号值的不同，给定义的寄存器赋上不同的值，从而确定输出信号的值。最后，使用 assign 语句将每个寄存器的值分别赋给对应的输出信号。代码实现如下：

```
module Ctr(
    input [5:0] opCode,
    output regDst,
    output aluSrc,
    output memToReg,
    output regWrite,
    output memRead,
    output memWrite,
    output branch,
```

```
10      output [1:0] aluOp,
11      output jump
12      );
13
14      reg RegDst;
15      reg ALUSrc;
16      reg MemToReg;
17      reg RegWrite;
18      reg MemRead;
19      reg MemWrite;
20      reg Branch;
21      reg [1:0] ALUOp;
22      reg Jump;
23
24      always @(opCode)
25      begin
26          case(opCode)
27          6'b000000: //R type
28          begin
29              RegDst = 1;
30              ALUSrc = 0;
31              MemToReg = 0;
32              RegWrite = 1;
33              MemRead = 0;
34              MemWrite = 0;
35              Branch = 0;
36              ALUOp = 2'b10;
37              Jump = 0;
38          end
39
40          //beq command
41          6'b000100:
42          begin
43              RegDst = 0;
44              ALUSrc = 0;
45              MemToReg = 0;
46              RegWrite = 1;
47              MemRead = 0;
48              MemWrite = 0;
49              Branch = 1;
50              ALUOp = 2'b01;
51              Jump = 0;
```

```
52          end
53
54          //lw command
55          6'b100011:
56          begin
57              RegDst = 0;
58              ALUSrc = 1;
59              MemToReg = 1;
60              RegWrite = 1;
61              MemRead = 1;
62              MemWrite = 0;
63              Branch = 0;
64              ALUOp = 2'b00;
65              Jump = 0;
66          end
67
68          //sw command
69          6'b101011:
70          begin
71              RegDst = 0;
72              ALUSrc = 1;
73              MemToReg = 0;
74              RegWrite = 0;
75              MemRead = 0;
76              MemWrite = 1;
77              Branch = 0;
78              ALUOp = 2'b00;
79              Jump = 0;
80          end
81
82          //add Jump
83          6'b000010:
84          begin
85              RegDst = 0;
86              ALUSrc = 0;
87              MemToReg = 0;
88              RegWrite = 0;
89              MemRead = 0;
90              MemWrite = 0;
91              Branch = 0;
92              ALUOp = 2'b00;
93              Jump = 1;
```

```
94          end
95
96          default:
97          begin
98              RegDst = 0;
99              ALUSrc = 0;
100             MemToReg = 0;
101             RegWrite = 0;
102             MemRead = 0;
103             MemWrite = 0;
104             Branch = 0;
105             ALUOp = 2'b00;
106             Jump = 0;
107         end
108         endcase
109     end
110
111     assign regDst = RegDst;
112     assign aluSrc = ALUSrc;
113     assign memToReg = MemToReg;
114     assign regWrite = RegWrite;
115     assign memRead = MemRead;
116     assign memWrite = MemWrite;
117     assign branch = Branch;
118     assign aluOp = ALUOp;
119     assign jump = Jump;
120
121 endmodule
```

3.2 ALUCtr 模块

根据 ALUCtr 原理描述与指令表格设计 ALUCtr 模块代码。在模块内部，有一个 4 位的寄存器 ALUCtrOut，用于存储解析后的 ALU 控制信号。

always 块对输入信号进行组合逻辑判断，并将结果存储在 ALUCtrOut 寄存器中。使用 casex 语句，对不同的 aluOp 和 funct 组合进行了匹配，确定了对应的 ALU 控制信号类型。如果没有匹配成功，则默认输出 4'b0000。最后，通过 assign 语句将 ALUCtrOut 赋值给输出端口 aluCtrOut。代码实现如下：

```
1 module ALUCtr(
2     input [1:0] aluOp,
```

```
3       input [5:0] funct,
4       output [3:0] aluCtrOut
5       );
6       reg [3:0] ALUCtrOut;
7
8       always@(aluOp or funct)
9       begin
10          casex({aluOp, funct})
11              8'b00xxxxxx://lw, sw
12              begin
13              ALUCtrOut = 4'b0010;
14              end
15
16              8'bx1xxxxxx://beq
17              begin
18              ALUCtrOut = 4'b0110;
19              end
20
21              8'b1xxx0000://add
22              begin
23              ALUCtrOut = 4'b0010;
24              end
25
26              8'b1xxx0010://sub
27              begin
28              ALUCtrOut = 4'b0110;
29              end
30
31              8'b1xxx0100://and
32              begin
33              ALUCtrOut = 4'b0000;
34              end
35
36              8'b1xxx0101://or
37              begin
38              ALUCtrOut = 4'b0001;
39              end
40
41              8'b1xxx1010://stl
42              begin
43              ALUCtrOut = 4'b0111;
44              end
```

```
        endcase
    end

    assign aluCtrOut = ALUCtrOut;
endmodule
```

3.3 ALU 模块

根据 ALU 原理描述与指令表格设计 ALU 模块代码。在模块内部，有一个 32 位的寄存器 ALURes 用于存储运算结果和寄存器 Zero 用于存储标志 0。

在 always 块中，根据给定的 aluCtr 执行所需的算术逻辑运算，并将结果存储在 ALURes 寄存器中。如果 ALURes 等于零，则将零标志设置为 1；否则，将零标志设置为 0。最后，通过 assign 语句将 ALURes 和 Zero 分配给相应的输出端口 aluRes 和 zero。代码实现如下：

```
用中文介绍如下代码
module ALU(
    input [31:0] input1,
    input [13:0] input2,
    input [3:0] aluCtr,
    output zero,
    output [31:0] aluRes
    );
    reg Zero;
    reg [31:0] ALURes;

    always @ (input1 or input2 or aluCtr)
    begin
        //add
        if(aluCtr == 4'b0010)
            ALURes = input1 + input2;

        //sub
        else if (aluCtr == 4'b0110)
            ALURes = input1 - input2;

        //or
        else if (aluCtr == 4'b0001)
            ALURes = input1 | input2;

        //and
        else if (aluCtr == 4'b0000)
```

```
28              ALURes = input1 & input2;
29
30          //nor
31          else if (aluCtr == 4'b1100)
32              ALURes = ~(input1 | input2);
33
34          //slt
35          else if (aluCtr == 4'b0111)
36              ALURes = ($signed(input1) < $signed(input2));
37
38          if(ALURes==0)
39              Zero = 1;
40          else
41              Zero = 0;
42      end
43
44     assign aluRes=ALURes;
45     assign zero=Zero;
46
47 endmodule
```

4 实例化和测试

4.1 主控制器测试

编写激励程序如下：

```
1  module Ctr_tb(
2
3      );
4      reg [5 : 0] OpCode;
5      wire RegDst;
6      wire ALUSrc;
7      wire MemToReg;
8      wire RegWrite;
9      wire MemRead;
10     wire MemWrite;
11     wire Branch;
12     wire [1 : 0] ALUOp;
13     wire Jump;
14
```

```
    Ctr u0(
        .opCode(OpCode),
        .regDst(RegDst),
        .aluSrc(ALUSrc),
        .memToReg(MemToReg),
        .regWrite(RegWrite),
        .memRead(MemRead),
        .memWrite(MemWrite),
        .branch(Branch),
        .aluOp(ALUOp),
        .jump(Jump)
    );

    initial begin
        // Initialize Inputs
        OpCode = 0;

        // Wait 100 ns for global reset to finish
        #100;

        #100 OpCode = 6'b000000;    // R-Type

        #100 OpCode = 6'b100011;    // lw

        #100 OpCode = 6'b101011;    // sw

        #100 OpCode = 6'b000100;    // beq

        #100 OpCode = 6'b000010;    // jump

        #100 OpCode = 6'b111111;    // default

    end
endmodule
```

即指令顺序为 R-Type, lw, sw, beq, jump, 其他指令，仿真测试如图 5所示，对应信号值符合预期，即说明该模块设计良好。

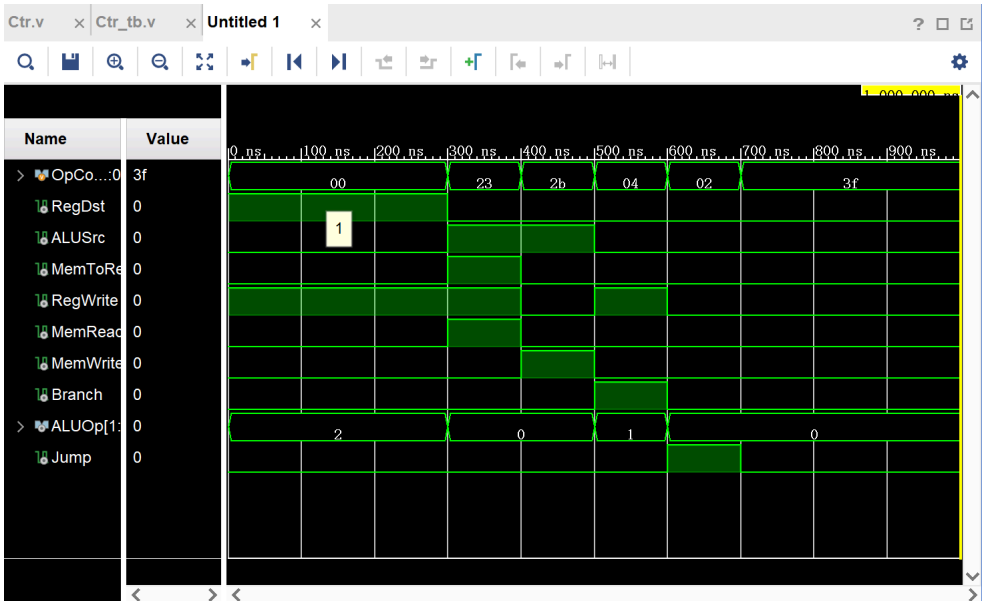

图 5: Ctr 模块测试结果

4.2 ALU 控制器测试

编写激励程序如下：

```
module ALUCtr_tb(

    );
    reg [1:0] ALUOp;
    reg [5:0] Funct;
    wire [3:0] ALUCtrOut;

    ALUCtr u0(
        .aluOp(ALUOp),
        .funct(Funct),
        .aluCtrOut(ALUCtrOut)
        );

    initial begin
        ALUOp = 0;
        Funct = 0;
        #100;

        //0010
```

```
        ALUOp = 2'b00;
        Funct = 6'b000000;
        #100;

        //0110
        ALUOp = 2'b01;
        Funct = 6'b000000;
        #100;

        //0010
        ALUOp = 2'b10;
        Funct = 6'b000000;
        #100;

        //0110
        ALUOp = 2'b10;
        Funct = 6'b000010;
        #100;

        //0000
        ALUOp = 2'b10;
        Funct = 6'b000100;
        #100;

        //0001
        ALUOp = 2'b10;
        Funct = 6'b000101;
        #100;

        //0111
        ALUOp = 2'b10;
        Funct = 6'b001010;
        #100;
    end

endmodule
```

即给定指令顺序为 lw/sw, beq, add, sub, and, or, slt, 对应的 ALUCtrOut 顺序应为 0010, 0110, 0010, 0110, 0000, 0001, 0111。仿真图 6表明，所得结果符合预期，证明该模块正确。

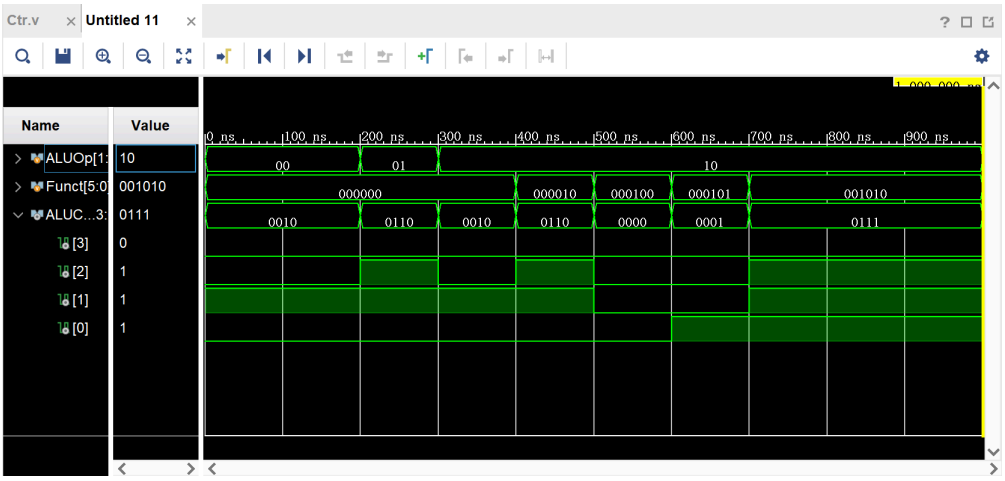

图 6: ALUCtr 模块测试结果

4.3 ALU 测试

编写激励程序如下：

```
module ALU_tb(

    );
    wire Zero;
    wire [31:0] ALURes;
    reg [31:0] Input1;
    reg [31:0] Input2;
    reg [3:0] ALUCtr;

    ALU u0(
        .aluRes(ALURes),
        .input1(Input1),
        .input2(Input2),
        .aluCtr(ALUCtr),
        .zero(Zero)
    );

    initial begin
        ALUCtr = 0;
        Input1 = 0;
        Input2 = 0;
        #100;
```

```
23
24          ALUCtr = 4'b0000;
25          Input1 = 15;
26          Input2 = 10;
27          #100;
28
29          ALUCtr = 4'b0001;
30          Input1 = 15;
31          Input2 = 10;
32          #100;
33
34          ALUCtr = 4'b0010;
35          Input1 = 15;
36          Input2 = 10;
37          #100;
38
39          ALUCtr = 4'b0110;
40          Input1 = 15;
41          Input2 = 10;
42          #100;
43
44          ALUCtr = 4'b0110;
45          Input1 = 10;
46          Input2 = 15;
47          #100;
48
49          ALUCtr = 4'b0111;
50          Input1 = 15;
51          Input2 = 10;
52          #100;
53
54          ALUCtr = 4'b0111;
55          Input1 = 10;
56          Input2 = 15;
57          #100;
58
59          ALUCtr = 4'b1100;
60          Input1 = 1;
61          Input2 = 1;
62          #100;
63
64          ALUCtr = 4'b1100;
```

```
65          Input1 = 16;
66          Input2 = 1;
67          #100;
68      end
69  endmodule
```

给定的指令顺序为 and, or, add, sub, sub, slt, slt, nor, nor。进行仿真结果如图 7，结果与预期相同。例如，对 nor 指令部分对二进制如图 8进行观察，结果显然正确。

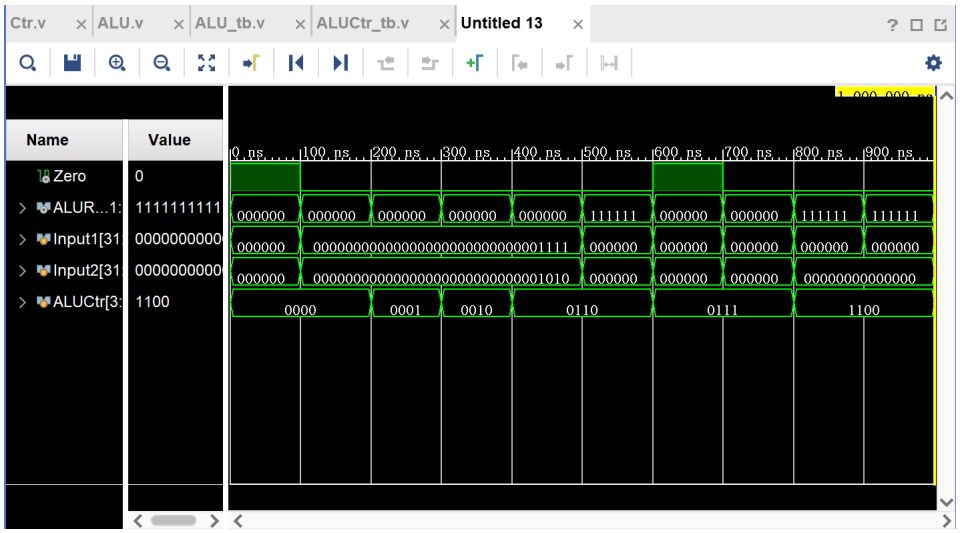

图 7: ALUCtr 模块测试结果

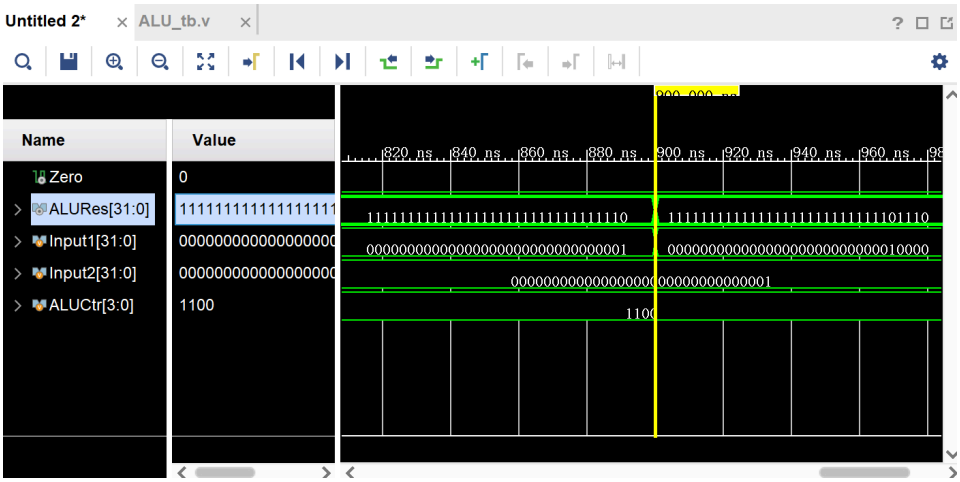

图 8: ALUCtr 模块 NOR 指令测试结果

5 总结与反思

实验 3 在理解 MIPS 指令结构的基础之上，学习了主控制部件或单元、ALU 控制器单元、ALU 单元的原理：

- MIPS 指令格式的三种类型 R-Type, I-Type, J-Type
- 主控制部件根据指令的操作码（opcode）来确定该指令的操作类型，并产生相应的控制信号，以控制指令执行的流程
- ALU 控制模块接收主控制部件发出的指令操作码，并根据操作码中的功能字段（function field）产生相应的控制信号，以控制 ALU 执行相应的操作。
- ALU 接收来自寄存器的操作数，并根据 ALU 控制器单元发出的控制信号执行相应的运算，最终将结果写回寄存器或者其他目的地。

并根据 MIPS 指令集设计支持九条指令（beq,lw,sw,add,sub,and,or,slt,jump）的三个单元模块，随后对实现的各个模块进行仿真测试。

实验 3 是构建 MIPS 单周期处理器的前置任务和先修工作，较为简单。但需要对照指令表，在理解指令解析逻辑的基础上进行模块设计，虽然目前指令不多，但进行信息整理有利于推进复杂的工作。在本次实验中，总结如下经验：

在进行模块设计时，先将对应信息用表格整理好，有利于避免疏漏和错误，也方便检查和调试。

在进行仿真测试时，发生如图 9所示报错，这一般是由于语法错误造成。

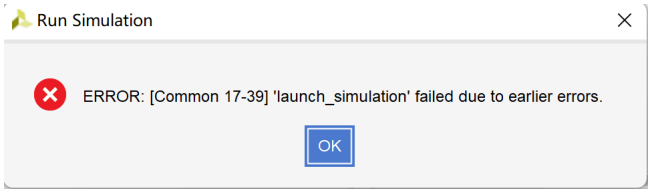

图 9: 仿真过程报错信息

6 致谢

刘雨桐老师为实验提供了良好的讲解，在课程中时常做出对于任务难度和时间上的提醒，并十分体谅学生，能理解我们的困难并给予帮助，深表感谢。

实验过程中，助教蔡明昕、黄正翔老师多次解决我的困惑和实验中遇到的问题，并且在困难的时候给予我鼓励和帮助，对此深表感激。

感谢黄小平老师及实验室提供的资源和硬件支持。

感谢在实验过程中给予我帮助的同学。

邓倩妮老师为使得教学效果更好，将课程体系由 RISC-V 改为 MIPS，在实验中提供很多帮助，对此表示感谢。