Lab03　简单的类 MIPS 单周期处理器功能部件的设计与实现（一）

w1049

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　原理分析

2.1　主控单元

主控单元根据指令中的 opCode 产生相应的控制信号，有的直接作用于相应部件，有的（比如 ALUOp）作为 ALU 控制单元的输入。目前的信号如表 1 所示。

表 1: 主控单元的控制信号

信号名	功能
regDst	选择目标寄存器，0 为 rt，1 为 rd
aluSrc	ALU 的第二个输入，0 为 rt，1 为立即数
memToReg	将写入的数据来源，0 为 ALU 结果，1 为内存读取
regWrite	寄存器写使能信号
memRead	内存读使能信号
memWrite	内存写使能信号
aluOp	在 ALU 控制单元（ALUCtr）中进一步处理
branch	当前指令是否为条件分支指令
jump	当前指令是否为跳转指令

opCode 与主控单元输出的关系如表 2 所示。

表 2: 主控单元的真值表

OpCode 指令	000000 R 型指令	000010 j	000100 beq	100011 lw	101011 sw
aluSrc	0	0	0	1	1
aluOp	10	00	01	00	00
branch	0	0	1	0	0
jump	0	1	0	0	0
memRead	0	0	0	1	0
memToReg	0	0	0	1	0
memWrite	0	0	0	0	1
regDst	1	0	0	0	0
regWrite	1	0	0	1	0

2.2 ALU 控制单元

ALU 控制单元从指令中解码出 ALU 需要做什么运算，一部分工作已经由主控单元完成。指令与信号的对应关系如表 3 所示。

表 3: ALU 控制单元信号与指令

指令	aluOp	funct	aluCtr	功能
lw	00	xxxxxx	0010	+
sw	00	xxxxxx	0010	-
beq	01	xxxxxx	0110	-
add	10	100000	0010	+
sub	10	100010	0110	-
and	10	100100	0000	&
or	10	100101	0001	|
slt	10	101010	0111	slt

2.3 ALU

ALU 是算术逻辑单元，有两个操作数输入和一个结果输出，控制信号与运算的对应关系如表 4 所示。

表 4: ALU 功能

aluCtr	功能
0000	and
0001	or
0010	add
0110	sub
0111	set on less than (slt)
1100	nor

3 功能实现

3.1 主控单元

可以使用 case 语句来实现，其功能类似于 C 语言中的 switch 语句，大致代码如下：

```
    always @(opCode) begin
      case (opCode)
      6'b000000: begin // R type
          regDst = 1;
          aluSrc = 0;
          memToReg = 0;
          regWrite = 1;
          memRead = 0;
          memWrite = 0;
          branch = 0;
          aluOp = 2'b10;
          jump = 0;
      end
      // ...
      endcase
    end
```

这里写在 always 块中的语句实际上是组合逻辑，但是等号左边的变量需要定义成 reg 类型，不能直接使用写在 output 中的变量。有很多种做法实现，比如：

```
module A (
    input a,
    output b
    );
    // 使用 assign
    reg B;
    assign b = B;

    // 或者直接定义重名变量
    reg b;

    // 两种方式结果是一致的

    always @(a) begin
        B = a;
    end
endmodule
```

另一种是直接在 output 定义时加上 reg，这与上面两种的结果也是一致的，代码如下：

```
module A (
    input a,
    output reg b // reg 直接写在这里
    );

    always @(a) begin
        B = a;
    end
endmodule
```

我在代码中使用的是更为简洁的最后一种方式。

3.2 ALU 控制单元

在 ALU 控制单元的表 3 中，有些位是不关心的（无关位），被 x 代替。实现时，可以用 casex 语句代替 case 语句，两种语句唯一的差别是 casex 语句将 x 视为无关位，代码如下：

```
  always @(aluOp or funct) begin
    casex ({aluOp, funct})
    8'b00xxxxx: aluCtrOut = 4'b0010;
    8'bx1xxxxx: aluCtrOut = 4'b0110;
    8'b1xxx0000: aluCtrOut = 4'b0010;
    8'b1xxx0010: aluCtrOut = 4'b0110;
    8'b1xxx0100: aluCtrOut = 4'b0000;
    8'b1xxx0101: aluCtrOut = 4'b0001;
    8'b1xxx1010: aluCtrOut = 4'b0111;
    default: aluCtrOut = 4'b0;
    endcase
  end
```

3.3 ALU

ALU 的实现比较简单，根据控制信号进行运算即可。注意 slt 指令是有符号比较，需要使用 $signed() 转换。代码如下：

```
  always @(input1 or input2 or aluCtr) begin
      case (aluCtr)
      4'b0000: aluRes = input1 & input2;
      4'b0001: aluRes = input1 | input2;
      4'b0010: aluRes = input1 + input2;
      4'b0110: aluRes = input1 - input2;
      4'b0111: aluRes = $signed(input1) < $signed(input2) ? 1 : 0;
      4'b1100: aluRes = ~(input1 | input2);
      endcase
      if (aluRes == 0) zero = 1;
      else zero = 0;
  end
```

4 功能仿真

4.1 仿真激励文件

分别对三个模块编写激励文件并进行仿真，测试覆盖所有语句即可。在更改仿真的目标时，需要右键点击想要测试的文件，选择 “Set as Top”。

4.2 仿真结果

4.2.1 主控单元

波形如图 1。

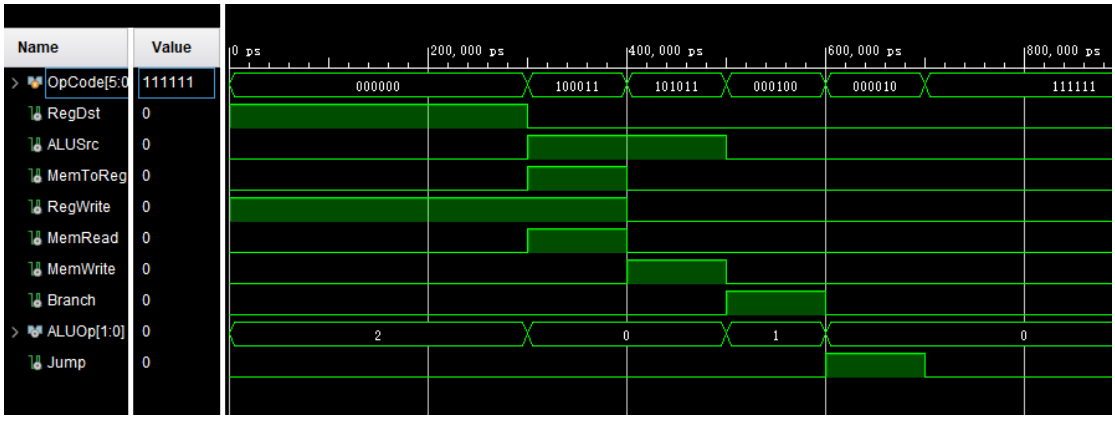

图 1: Ctr 的仿真波形

4.2.2 ALU 控制单元

根据激励文件中的赋值语句的不同，可能出现两种波形，即图 2 和 3。出现差别的原因是，在 casex 中 x 是无关位，实际情况下只会是 0 或者 1；如果在激励文件中直接把变量赋值成 x，那么在仿真时会显示红色的 x，而不是 “0 或 1 中的任意一个”。

还要注意，casex 中的语句是有顺序的，如果有多个语句匹配，那么会执行第一个匹配的语句。编写时我们只考虑了合法情况，也就是 aluOp 只有 00、01、10，而没有 11，所以 10 可以用 1x 代替、01 可以用 x1 代替。但如果仿真激励文件中使用 1x，它可以同时和 1x、x1 匹配，取第一个。

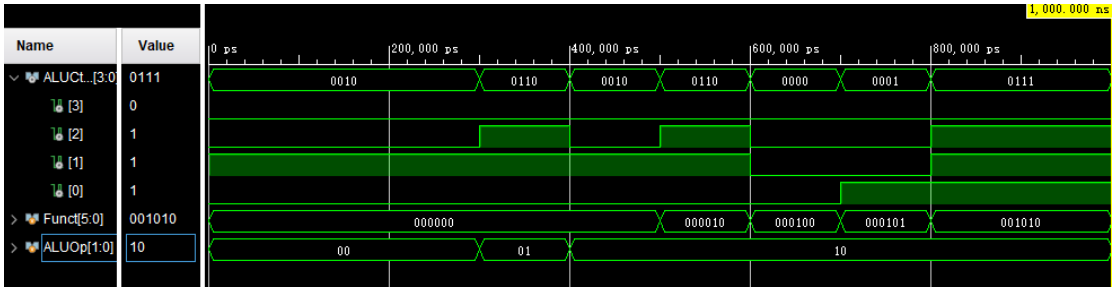

图 2: ALUCtr 的仿真波形 A

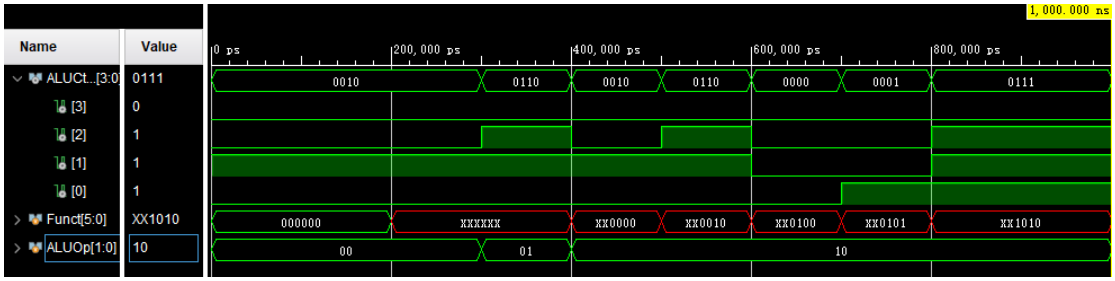

图 3: ALUCtr 的仿真波形 B

4.2.3 主控单元

波形如图 4，其中含有 `zero = 1` 的情况。NOR 运算的波形如图 5。

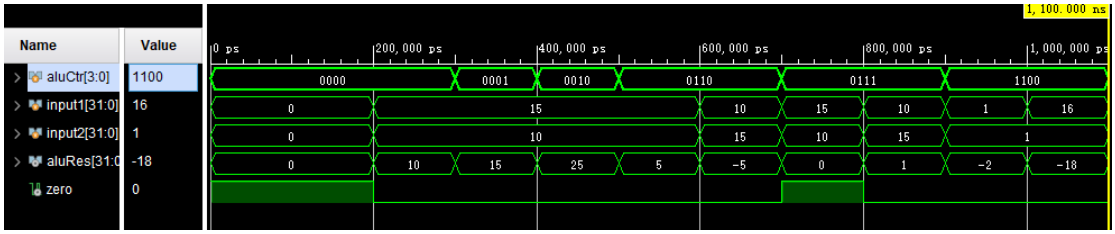

图 4: ALU 仿真波形

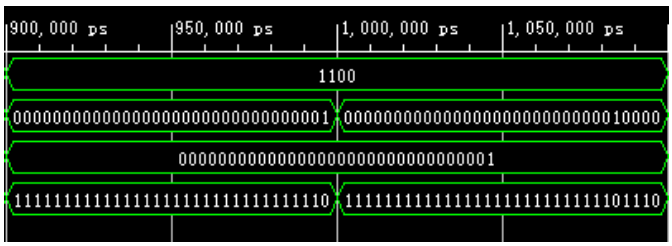

图 5: NOR 运算的二进制显示

5 总结与反思

本次实验开始，实验指导书的内容并不完全，需要自己思考和查询的部分增多了。我也遇到了一些问题，比如：

- 在 `always` 块内，虽然是组合逻辑，却不能直接给 `wire` 类型的输出变量赋值；
- 仿真时使用 x 会让结果不正确；
- 不知道默认比较时使用有符号数还是无符号数。

在查询资料、询问助教和老师后，这些问题一一得到解决。

本次实验完成的 ALU 功能较为简单，运算功能少，如果想要实现有更多指令的处理器，还要修改、扩展本次编写的模块。