实验五 数据通路与控制单元设计

一、实验目的

1 掌握 Verilog 语言和 Vivado、Logisim 开发平台的使用；

2 掌握数据通路与控制单元的设计和测试方法。

二、实验内容

1 数据通路的设计；

2 控制单元的设计。

三、实验要求

1 掌握 Vivado 或 Logisim 开发工具的使用，掌握以上电路的设计和测试方法；

2 记录设计和调试过程（Verilog 代码/电路图/表达式/真值表，Vivado 仿真结果，Logisim 验证结果等）；

3 分析 Vivado 仿真波形/Logism 验证结果，注重输入输出之间的对应关系。

四、实验过程及分析

设计代码

控制单元

```
`timescale 1ns / 1ps

module Controller(
input Zero,
input [5:0] Op,Funct,
output MemToReg, MemWrite,
output [2:0] ALUControl,
output RegDst,RegWrite,
output PCSrc,ALUSrc,
output Jump);
wire Branch;
wire [1:0] ALUOp;
MainDec MainDec_1(Op,MemToReg,MemWrite, Branch,ALUSrc,RegDst,RegWrite,Jump,ALUOp);
ALUDec ALUDec_1(Funct,ALUOp,ALUControl);
assign PCSrc = Branch & Zero;
endmodule
```

声明了一个模块，其输入包括 Zero、Op 和 Funct，输出包括 MemToReg、MemWrite、ALUControl、RegDst、RegWrite、PCSrc、ALUSrc 和 Jump。wire Branch;：这声明了一个名为 Branch 的线。wire [1:0] ALUOp;：这声明了一个 2 位宽的线 ALUOp。MainDec MainDec_1(Op,

MemToReg,MemWrite, Branch, ALUSrc, RegDst, RegWrite, Jump, ALUOp);：这实例化了一个名为 MainDec_1 的 MainDec 模块，并将输入和输出连接到 Controller 模块的相应端口。ALUDec ALUDec_1(Funct, ALUOp, ALUControl);：这实例化了一个名为 ALUDec_1 的 ALUDec 模块，并将输入和输出连接到 Controller 模块的相应端口。

assign PCSrc = Branch & Zero;：这通过逻辑与门计算 PCSrc，其值为 Branch 和 Zero 的逻辑与结果。协调和控制主控制器（MainDec）和 ALU 控制器（ALUDec）的操作。它通过实例化这两个子模块，并根据它们的输出信号计算一些控制信号，如 PCSrc，最终形成一个完整的控制单元。

主译码器

```
`timescale 1ns / 1ps

module MainDec(
input [5:0] Op,
output MemToReg, MemWrite,
output Branch, ALUSrc,
output RegDst, RegWrite,
output Jump,
output [1:0] ALUOp);
reg [8:0] Controls;
assign {RegWrite,RegDst,ALUSrc,Branch,MemWrite,MemToReg,Jump,ALUOp} = Controls;
always@(*)
case(Op)
6'b000000: Controls <= 9'b110000010;
6'b100011: Controls <= 9'b101001000;
6'b101011: Controls <= 9'b001010000;
6'b000100: Controls <= 9'b000100001;
6'b001000: Controls <= 9'b101000000;
6'b000010: Controls <= 9'b000000100;
default: Controls <= 9'bxxxxxxxxx;
endcase
endmodule
```

定义了一个名为 MainDec 的模块，该模块根据输入的 6 位信号 Op 生成一系列控制信号，并将这些控制信号分配给特定的输出端口。

输入为 Op，输出为 MemToReg、MemWrite、Branch、ALUSrc、RegDst、RegWrite、Jump 和 ALUOp。reg [8:0] Controls;：这声明了一个名为 Controls 的 9 位宽寄存器。assign {RegWrite, RegDst, ALUSrc, Branch, MemWrite, MemToReg, Jump, ALUOp} = Controls;：此行将 Controls 寄存器的各个位分配给相应的输出端口。always @(*)：这意味着 always 块内的代码将在任何输入信号发生变化时执行。case(Op)：这开始了基于 Op 值的 case 语句。对于每个 Op 的值，将特定的控制信号赋值给 Controls。default: Controls <= 9'bxxxxxxxxx;用于在没有匹配到任何指定情况时，将“未知”值赋给 Controls。

ALU 译码器

```
`timescale 1ns / 1ps

module ALUDec(
input [5:0] Funct,
input [1:0] ALUOp,
output reg [2:0] ALUControl) ;
always@(*)
case(ALUOp)
2'b00: ALUControl <= 3'b010;
2'b01: ALUControl <= 3'b110;
default: case(Funct)
6'b100000: ALUControl <= 3'b010;
6'b100010: ALUControl <= 3'b110;
6'b100100: ALUControl <= 3'b000;
6'b100101: ALUControl <= 3'b001;
6'b101010: ALUControl <= 3'b111;
default: ALUControl <= 3'bxxx;
endcase
endcase
endmodule
```

定义了一个名为 ALUDec 的模块，该模块根据输入的信号 Funct 和 ALUOp 生成一个 3 位宽的输出信号 ALUControl，用于控制 ALU（算术逻辑单元）的操作。

输入为 Funct 和 ALUOp，输出为 3 位宽的 ALUControl。case 语句中，当 ALUOp 为

2’b00 时，ALUControl 被赋值为 3’b010。当 ALUOp 为 2’b01 时，ALUControl 被赋值为 3’b110。如果 ALUOp 不匹配上述情况，则进入内部的 case(Funct)语句。根据具体的 Funct 值为 ALUControl 赋不同的值。当 Funct 为 6’b100000 时，ALUControl 被赋值为 3’b010。当 Funct 为 6’b100010 时，ALUControl 被赋值为 3’b110。当 Funct 为 6’b100100 时，ALUControl 被赋值为 3’b000。当 Funct 为 6’b100101 时，ALUControl 被赋值为 3’b001。当 Funct 为 6’b101010 时，ALUControl 被赋值为 3’b111。如果 Funct 不匹配上述情况，则 ALUControl 被赋值为 3’bxxx。根据输入的 Funct 和 ALUOp 值生成相应的 ALU 控制信号，用于指导 ALU 执行特定的操作。

模拟代码

```
`timescale 1ns / 1ps


module Controller_tb;
  reg [5:0] Op;
  reg [5:0] Funct;
  reg Zero;
  wire MemToReg;
  wire MemWrite;
  wire PCSrc;
  wire ALUSrc;
  wire RegDst;
  wire RegWrite;
  wire Jump;
  wire [2:0] ALUControl;
  Controller uut (
    .Op(Op),
    .Funct(Funct),
    .Zero(Zero),
    .MemToReg(MemToReg),
    .MemWrite(MemWrite),
    .PCSrc(PCSrc),
    .ALUSrc(ALUSrc),
    .RegDst(RegDst),
    .RegWrite(RegWrite),
    .Jump(Jump),
```

```
    .ALUControl(ALUControl)
);
initial begin
Op = 6'b000000;
Funct = 6'b100100;
Zero = 1'b1;
#10;
Op = 6'b000000;
Funct = 6'b100101;
Zero = 1'b1;
#10;
Op = 6'b000000;
Funct = 6'b101010;
Zero = 1'b1;
#10;
Op = 6'b100011;
Funct = 6'b000000;
Zero = 1'b0;
#10;
Op = 6'b100011;
Funct = 6'b000000;
Zero = 1'b1;
#10;
Op = 6'b111111;
Funct = 6'b100101;
Zero = 1'b1;
#10;
Op = 6'b000000;
Funct = 6'b111111;
Zero = 1'b1;
#10;
$finish;
end
endmodule
```

该模拟代码通过对 Controller 模块的输入信号进行赋值，并在每个时钟周期后等待一段时间（#10 表示等待 10 个时间单位），模拟了多个时钟周期的操作。

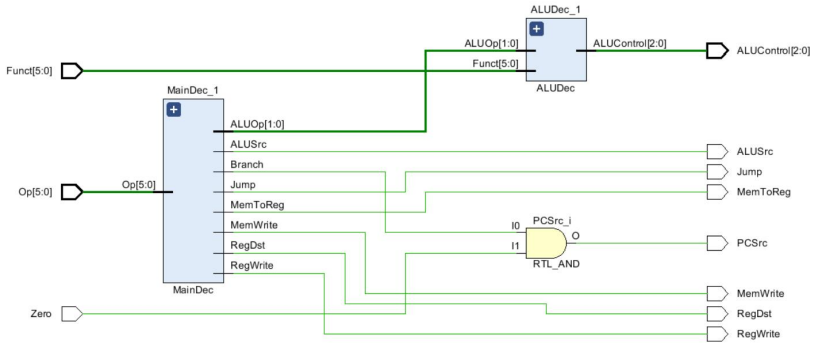
模块实例化：Controller 模块被实例化为 uut，并且输入和输出信号被连接。

initial begin 块：这是一个初始块，包含了对输入信号的赋值和等待时间的指令。每

个#10;指令表示在仿真中等待 10 个时间单位，模拟了一个时钟周期的时间。$finish;：在完成测试后终止仿真。

总体而言，这个测试台通过一系列的输入值模拟了对 Controller 模块的测试。在每个时钟周期后，它等待一段时间，然后更新输入值。仿真将在执行完所有输入值的测试后结束。

RTL 分析



波形图分析

当 OP 为 6’b000000 时，,ALUOp 为 10，此时在 ALUDec 中根据 Funct 的值来改变 ALUControl 的值，分别将 Funct 的值赋为 6’b100100. 6’b100101.6’b101010,可得到 ALUContorl 的值为 0、1、7，将 OP 的值改变为其他值以后，则 ALUDec 的值是根据 ALUOp 的值来变化了，于是这里只将 OP 赋值为 6’b100011，并展示了 Zero 值为 0 和 1 时不同的输出结果。之后分别将 OP 和 Funct 的值设为会产生无效输出的值，并加以展示。这就是输出波形图的所有特征。

五、调试和心得体会

本次实验学习了数据通路的设计和控制单元的设计。通过对 PPT 的学习，我了解了 MIPS 体系结构，包括各种不同的指令集及其执行过程，对于电路图和各端口的作用有了一定的理解，同时学习了控制单元。

在本次模拟代码的设计过程中，起初我有些疑惑如何将 controller 中的两个中间 wire 变量在输出中体现出来，后面经过和同学的讨论后发现没有这个必要，于是在明确了要展示哪几种输出效果之后，很快的完成了模拟代码的编写。