|  |
| --- |
| 电子科技大学  **计算机专业类课程** |
| **实验报告** |
| **课程名称：计算机组成原理综合实验**  **学　　院：计算机科学与工程学院**  **专　　业：信息安全**  **学生姓名：蒋程**  **学　　号：2017060201009**  **指导教师：王勇** |
|  |
| **日　　期：　　2019 年 6 月　29　日** |

**实验一**

1. **实验名称：红绿灯控制**
2. **实验学时：4**
3. **实验内容和目的：**

* 实验内容：根据图3-1所示的电路控制表设计一个控制电路，使其能够通过改变开关控制红色、黄色、绿色三种颜色的灯的亮灭。
* 实验目的：熟悉集成开发系统与Verilog HDL硬件描述语言，能够编写简单的控制电路代码并在开发板上实现。

****

图3-1 电路控制信号表

1. **实验原理：**

在Xilinx ISE Design Suite 14.7集成开发系统上创建Verilog模板，并使用Verilog HDL编写模板。主要使用了assign赋值语句和算数和逻辑操作语句。最后根据开发板上对应操作按钮编写约束并在开发板上生成。

1. **实验器材（设备、元器件）**

* PC计算机：Xilinx ISE Design Suite 14.7集成开发系统、digilent.adept.system\_v2.10.2.exe
* FPGA数字电路开发平台：Anvyl（燧石TM）开发板

1. **实验步骤：**
2. 推算红黄绿灯的控制逻辑和开关0，1的逻辑表达式为
   1. Red = switch1 ^ switch 2
   2. Green = switch1 ^ switch 2
   3. Yellow = ~(switch1 ^ switch2)
3. 创建模块单元接口，并设置好输入输出线路，这里为两个input，三个output
4. 编写相关代码逻辑 (这里对异或门进行优化)

module lights(input A,B, output R,G,Y);

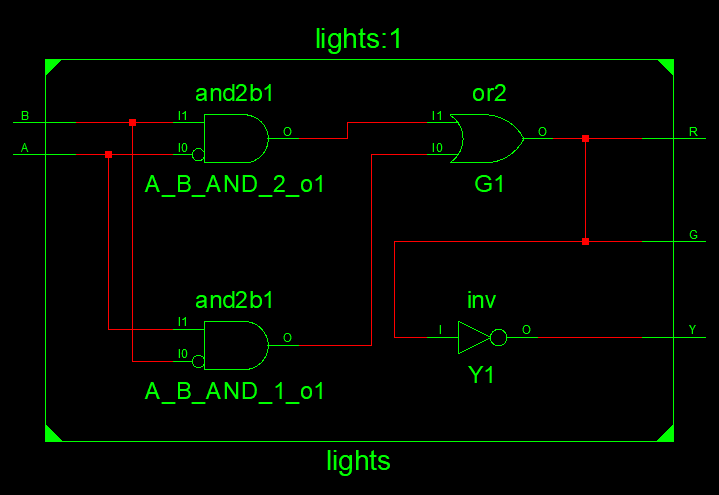
     assign R = (A & ~B)|(~A & B);

     assign G = R;

     assign Y = ~R;

endmodule

1. 检验RTL电路



1. **进行仿真测试：**

* 试代码如下，测试逻辑见代码注释

initial begin

    //初始A均设置为0，此时应该黄灯亮

    A = 0;

    B = 0;

    //250纳秒后，B为1，这个时候红绿灯亮

    #250;

B = 1;

    //250纳秒后，A为1，B为0，这个时候红绿灯亮

    #250;

    A = 1;

    B = 0;

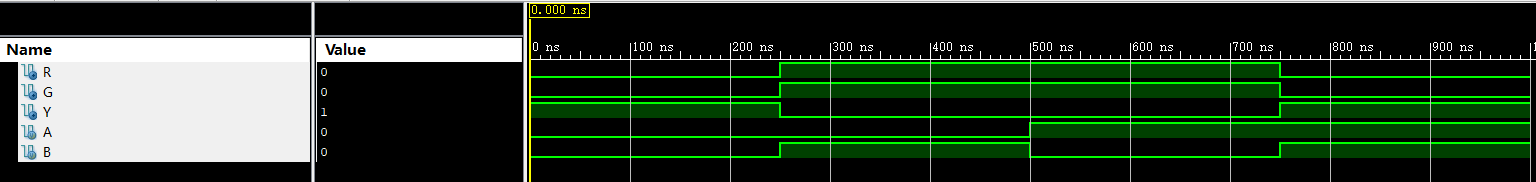
    //250秒后，A为1，B为1，这个时候黄灯亮

    #250;

    B = 1;

end

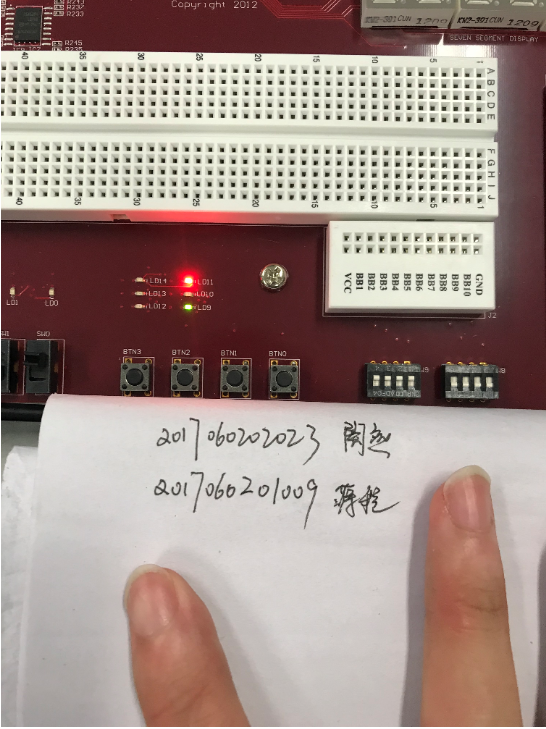
* 仿真测试结果如下(图片较长，请放大查看看)



1. **真机测试：**

* 将A与B分别映射到开发板的V5和U4上。
* RGB分别映射到开发板的T8 R7 U6上

（按下V5或U4）红绿灯亮 （同时(不)按V5和U4）黄灯亮



1. 实验结论、心得体会和改进建议：

* 结论：初步了解了Verilog的使用和简易的逻辑运算在电路中的实现原理。
* 心得体会：异或运算拥有三级门延迟，速度较慢，这里可以通过与或非门进行优化，使得延迟降低到2级门。

**实验二**

1. **实验名称：单周期CPU设计计算部分实现**
2. **实验学时：4**
3. **实验内容和目的：**

* 实验内容：使用特定的开发平台和开发板设计并实现CPU中的一些基本部件：32位二选一多路选择器、5位二选一多路选择器、32位寄存器堆、ALU和(无)符号扩展器。
* 实验目的：熟悉组成CPU的基本部件和CPU的基本结构，能够设计相关基本器件。

1. **实验原理：**

* 32位二选一多路选择器根据sel命令分别对32位的两个传入数据A和B进行选择，输出其中一个数据。其逻辑结构如下。

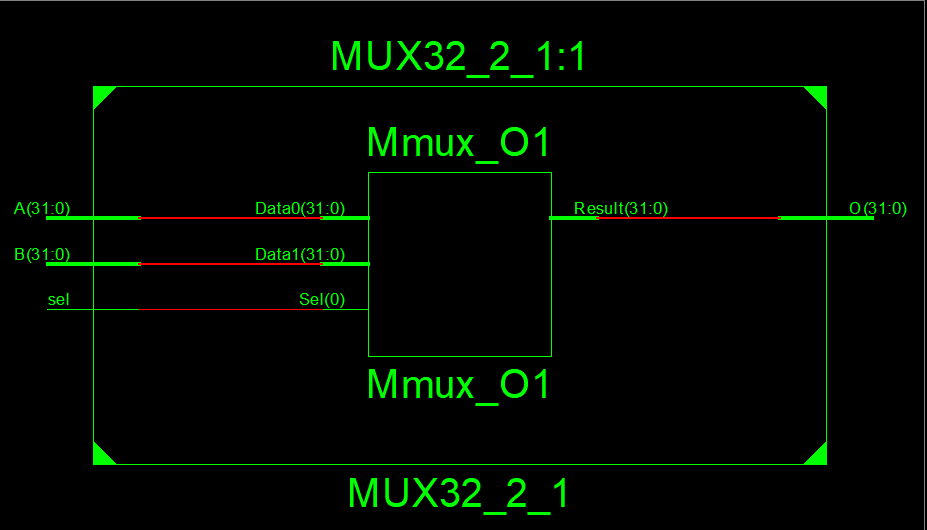
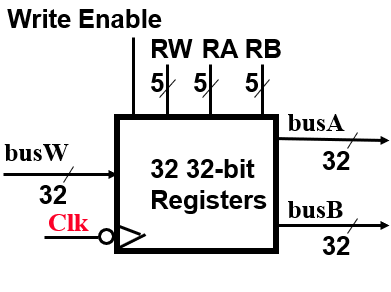
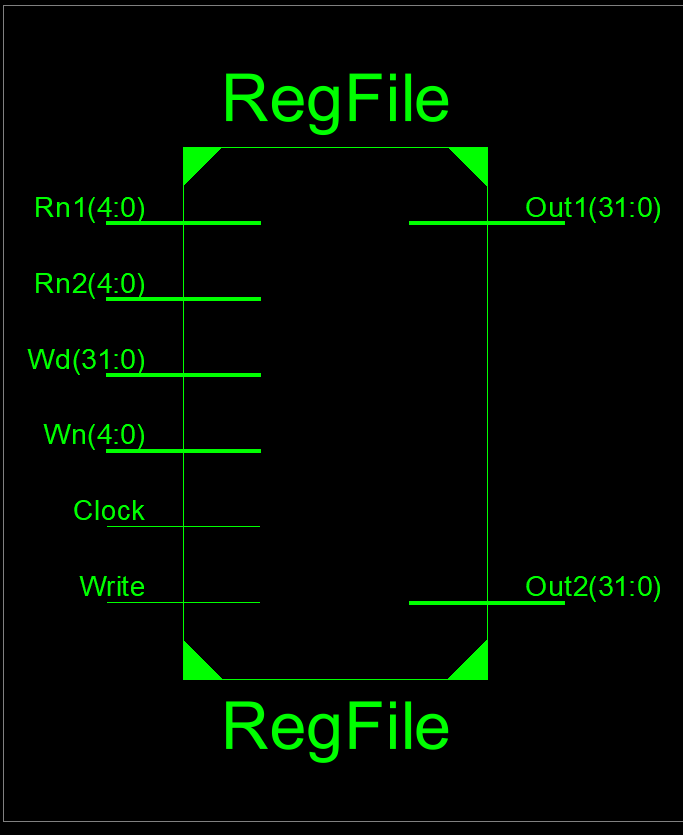


图4-1 二选一多路选择器的RTL图

* 32位寄存器堆由32个32位寄存器组成，其功能是根据时序电路对数据进行存入和读出指定寄存器数据的操作。其逻辑结构如下

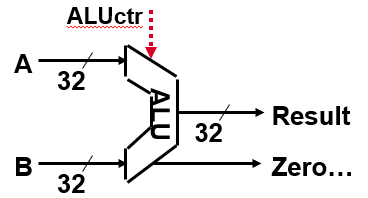


寄存器堆的逻辑结构图（源计组PPT）

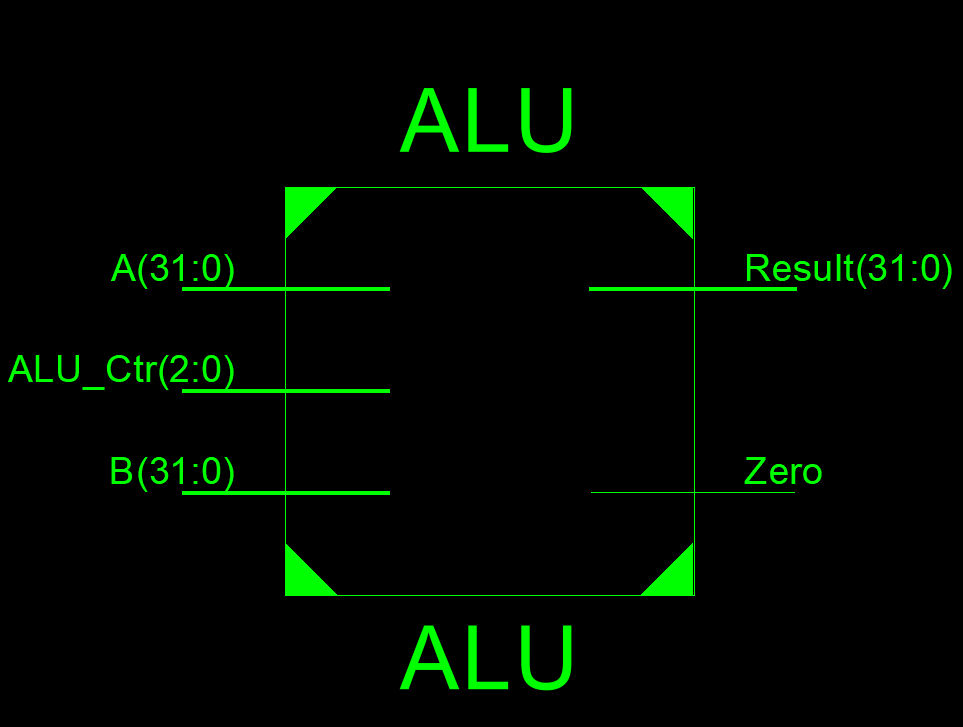


32位寄存器堆的RTL图

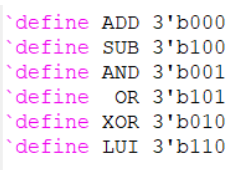
* ALU是CPU中重要的运算器，它可以根据ALU\_op的3位控制命令对传入的两个数据进行加、减、与、或等运算。



ALU的逻辑结构图（源计组PPT）



ALU的RTL结构



定义的ALU\_Ctr指令

* 扩展器是CPU在进行I型指令等操作时的重要部件，其功能是根据传入的16位数据的最高位对其进行符号扩展或无符号扩展为32位的立即数，并将其输出。**这里原本给定的是符号扩展器，但是对于ori操作对于立即数进行的是无符号扩展（且ROM里面有此指令），故这里修改为通用扩展器。**

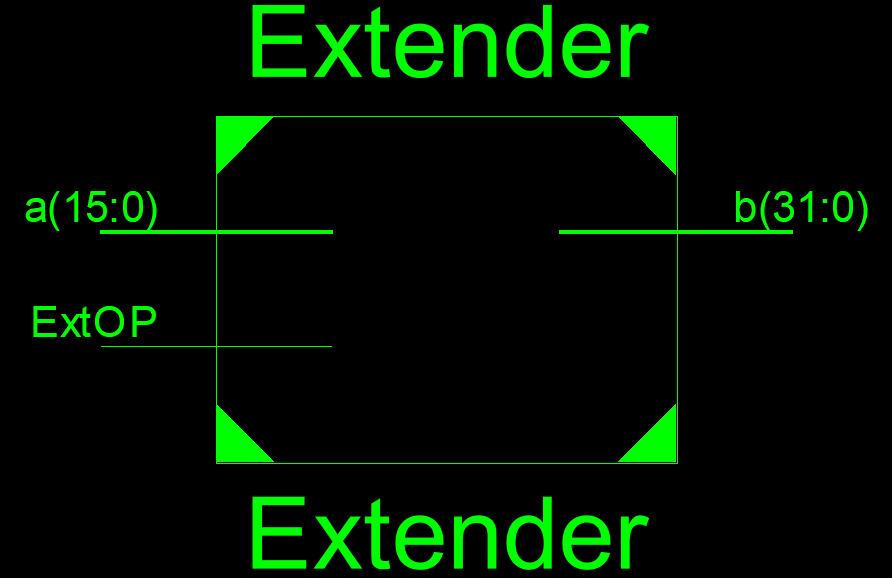


图4-4 符号扩展器的逻辑结构图

1. **实验器材（设备、元器件）**

* PC计算机：Xilinx ISE Design Suite 14.7集成开发系统、digilent.adept.system\_v2.10.2.exe
* FPGA数字电路开发平台：Anvyl（燧石TM）开发板

1. **实验步骤：**
2. **32位2选1选择器：**

* 根据其结构推断32位2选1选择器需要的线路
* 根据逻辑编写相关代码

module MUX32\_2\_1(

    input [31:0] A, B,

    input sel,

    output [31:0] O

);

    assign O = sel? B: A;

endmodule

* 查看32位2选1选择器的RTL结构图（见上图）
* 测试：略，实现比较简单，暂时不需要测试

1. **寄存器堆：**

* 设计寄存器堆的逻辑结构：
* 根据计组PPT提供的原理和结构图，确定寄存器堆的输入和输出的接口。
* 布置相关的寄存器（这里用32位的数组进行替代），并对其进行初始化。
* 定义输入数据源与寄存器的写入逻辑
* 定义相关的输出接口A和B与数据源的线路连接关系。
* 编写相关的代码（实现逻辑见备注）：

module RegFile(

input [4:0]Rn1,Rn2,Wn,

input Write,Clock,

input [31:0]Wd,

output [31:0]Out1,Out2

);

    //创建32个长度为32位的数组模拟寄存器堆

    reg [31:0] Register[1:31];

    //Out1输出Rn1信号指定的寄存器

    assign Out1 = (Rn1 == 0) ? 0: Register[Rn1];

    //Out2输出Rn2信号指定的寄存器

    assign Out2 = (Rn2 == 0) ? 0: Register[Rn2];

    //当始处于时钟上升沿并且处于写使能的时候，向寄存器堆里面写入数据

    always @( posedge Clock ) begin

        if ((Write) && (Wn != 0)) Register[Wn] <= Wd;

    end

    //堆寄存器堆里面的数据进行初始化（全0）

    integer i;

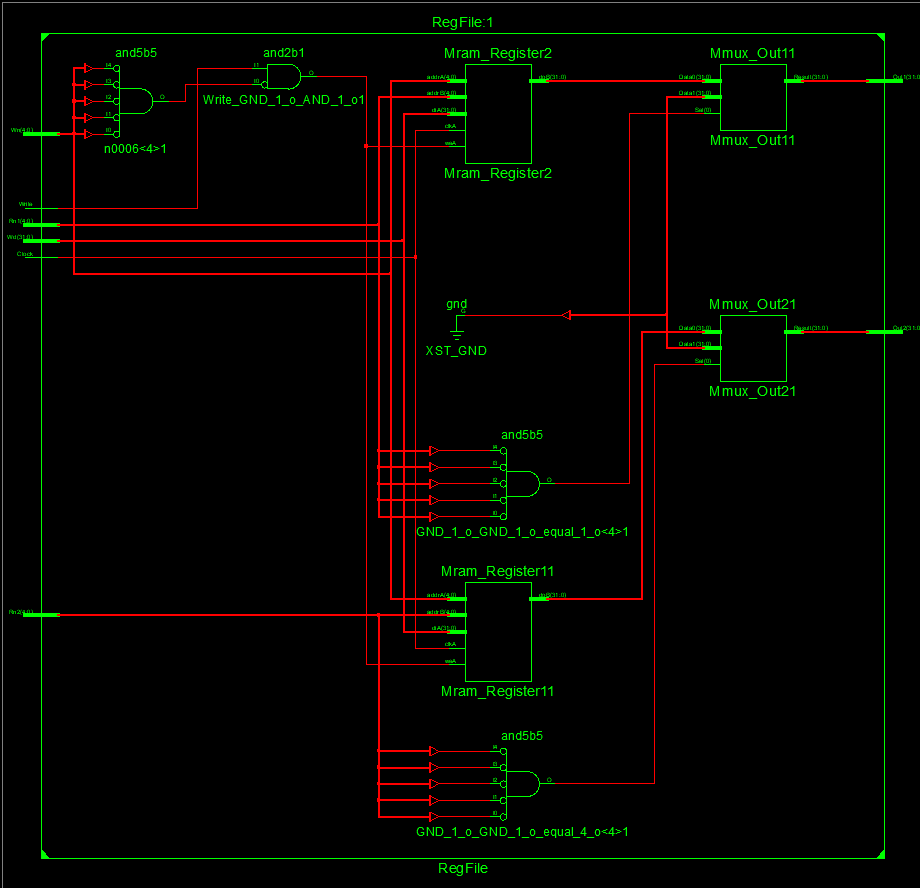
    initial begin

        for (i = 1 ; i <= 31; i = i + 1) Register [i] = 0;

    end

endmodule

* 测试寄存器堆：
* RTL图如下（图片较小，请放大查看）：



* 编写相关仿真代码：
* 仿真测试逻辑，每间隔100纳秒，向寄存器堆里写入指定数据，并且不断更新Rn1和Rn2的值，观察out1和out2的输出结果。

always begin

    #400;

//要写入的寄存器编号，每隔400纳秒，要写入的目标寄存器编号加1

    Wn = Wn + 1;

//这里将写入的Wd取值为4倍的Wn的值

//声明一个Temp临时变量用于0扩展

temp = {28'h0, Wn};

Wd = {temp[29:0], 2'b00};

    //输出端口A,B指定的目标寄存器的编号+1

    Rn1 = Rn1 + 1;

    Rn2 = Rn2 + 1;

end

always begin

    #100;

    Clock = ~Clock;

    Write = Clock == 1? 1: 0;

end

* 校验仿真测试结果（图片较长，请放大查看）：
* 为了方便查看，这里将结果转化位16进制。
* 由图1可知，Rn1和Rn2增加正常，Out1与out2输出的结果与寄存器堆里的指定Rn编号相符，即说明输出结果正确。
* 由图2可知，寄存器堆里存放的数据满足Wd = 4\*Wn，说明写入正确并且数据得到了顺利的保存。

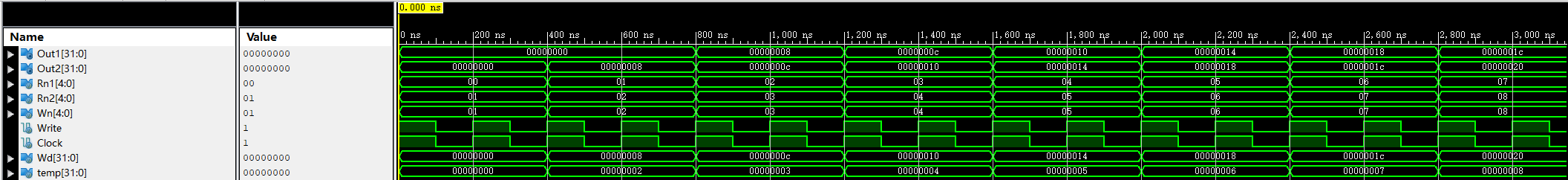


图1仿真结果

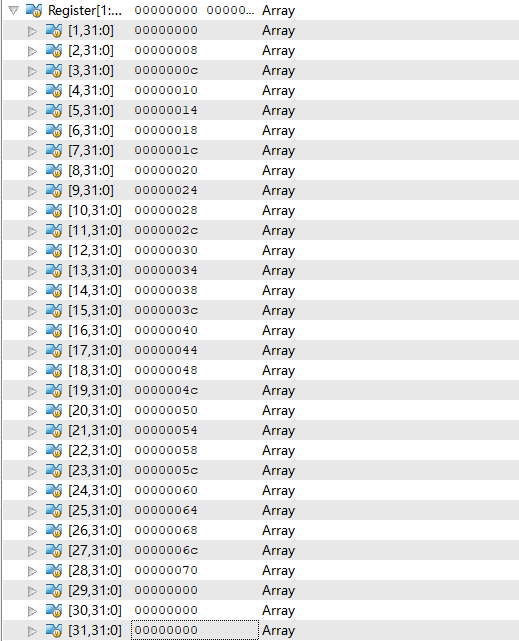


图2寄存器堆的里存放的数据

1. **通用扩展器**

* **通用扩展器的实现逻辑相对较为简单，故这里只放出实现的代码**

module Extender(

input [15:0] a,

     input ExtOP,

     output [31:0] b);

    assign b = ExtOP ? {a[15]? 16'hffff: 16'h0, a}:{a[15]? 16'h0000: 16'h0, a};

endmodule

1. **ALU**

* 设计ALU的逻辑结构
* 根据计组教材PPT提供的原理结构图，确定ALU的输入和输出的接口
* 参考老师提供的指令ROM提供的指令，根据实际情况，确定ALUCtr对应的相关操作。
* 编写相关代码：

`define ADD 3'b000

`define SUB 3'b100

`define AND 3'b001

`define OR 3'b010

`define XOR 3'b101

`define LUI 3'b110

module ALU(

    input [31:0] A, B,

    input [2:0] ALU\_Ctr,

    output reg [31:0] Result,

    output reg Zero);

    always @(A or B or ALU\_Ctr)

    begin

        case (ALU\_Ctr)

            `ADD :Result = A + B;

            `SUB :Result = A - B;

            `AND :Result = A & B;

            `OR :Result = A | B;

            `XOR :Result = A ^ B;

            `LUI :Result = B << 16;

             default: Result = 32'h00000000;

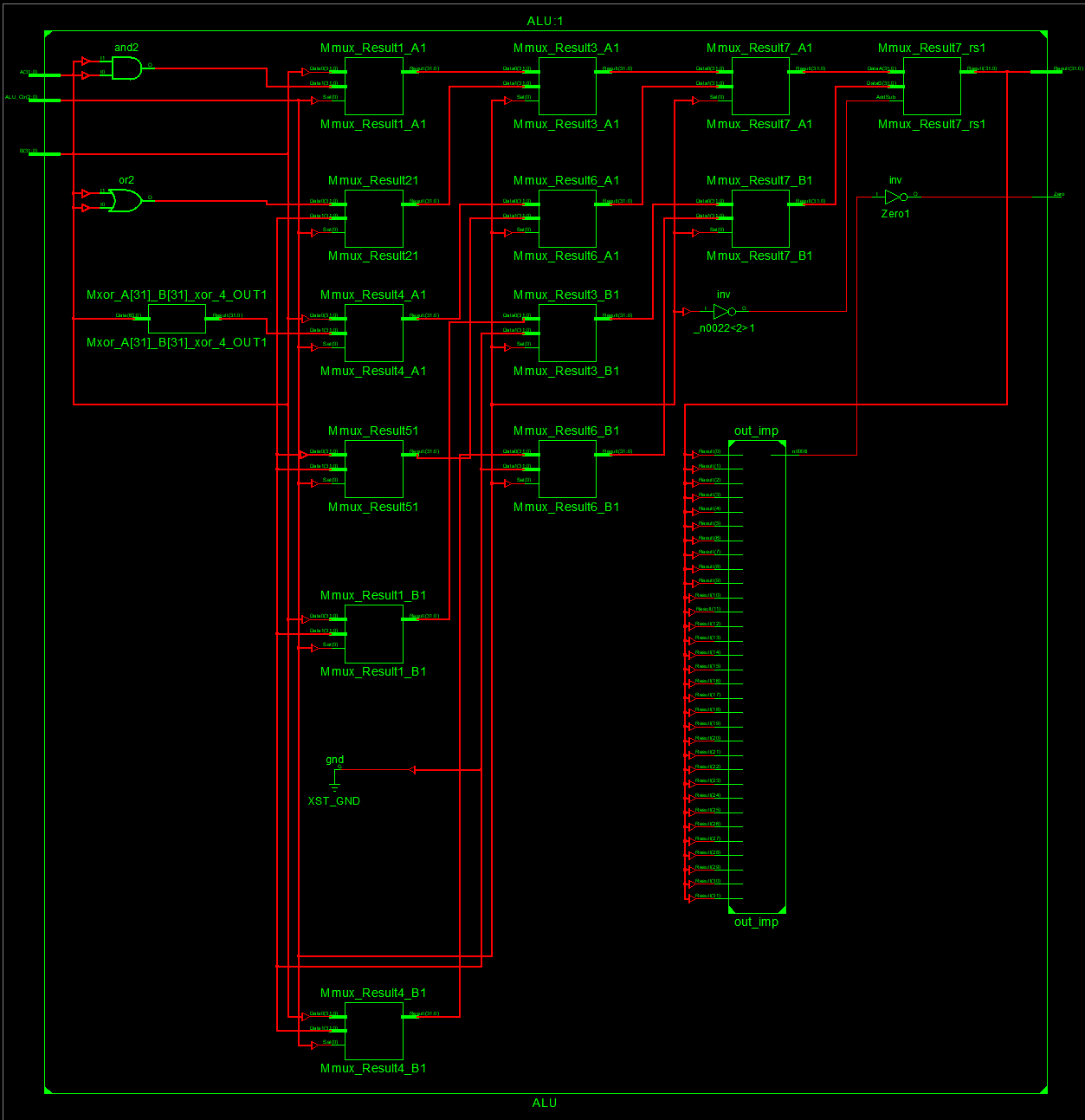
         endcase

    Zero = ~|Result;

    end

endmodule

* 测试ALU
* ALU的RTL结构图如下（图片较小，请放大查看）：



* 编写ALU的仿真测试代码（详细测试内容见注释）：

initial begin

        A = 32'h000000ff;

        B = 32'h000000cc;

        ALU\_Ctr = 111;//初始为111，此时返回全0

        #100;

        ALU\_Ctr = 3'b000;//测试加法

        #100;

        ALU\_Ctr = 3'b100;//测试减法

        #100;

        ALU\_Ctr = 3'b001;//测试与

        #100;

        ALU\_Ctr = 3'b101;//测试或

        #100;

        ALU\_Ctr = 3'b010;//测试异或

        #100;

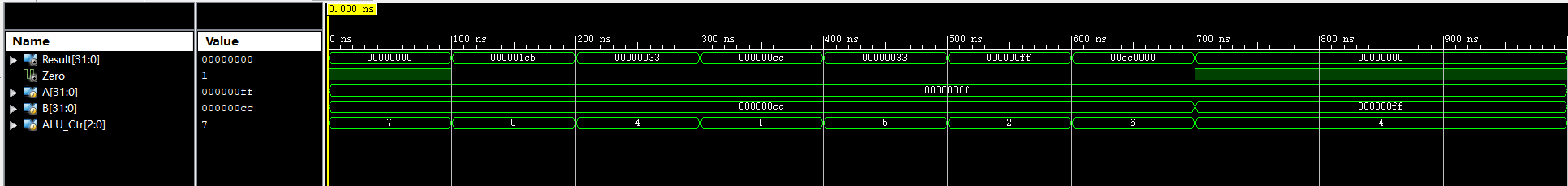
        ALU\_Ctr = 3'b110;//测试B的左移16位

        #100;

        B = 32'h000000ff;

        ALU\_Ctr = 3'b100;//测试0标志位

end

* 检验ALU仿真测试结果（图片较长，请放大查看）：
* 为了方便查看，这里将结果转化为16进制
* 由下图和上面相关测试数据对比可知，上述的相关操作均能够通过测试。
  1. **实验结论、心得体会和改进建议：**

**心得：**

* 做实验千万不要一味的抄老师的代码，之前PPT里面的代码是几个版本的结合体，实际上不互相兼容，这也是之前调不出来的主要原因。ALU所做的所有操作应当结合指令ROM和控制单元，而不是单纯的抄写计组书上面的MIPS CPU的操作逻辑。

**实验三**

1. **实验名称：单周期CPU的设计与实现**
2. **实验学时：4**
3. **实验内容和目的：**

* 实验内容：要求对单周期CPU中控制器进行设计与实现，主要包括Control部件和ALU\_Ctr部件。
* 实验目的：了解CPU中控制器部件的工作原理，能够熟练使用系统开发平台对于设计的控制电路进行仿真与实现，最后能够将部件进行封装。

1. **实验原理：**

* 本次实验要实现的是基于MIPS指令集的单周期CPU，其中控制单元如图4-1所示，包括两部分：Control部件和ALU\_CTR部件。当MIPS指令类型为R型时，op为000000，指令功能由输入进ALU\_CTR部件的func指令字段控制。当MIPS指令类型为I型和J型时，指令的功能由op字段控制。

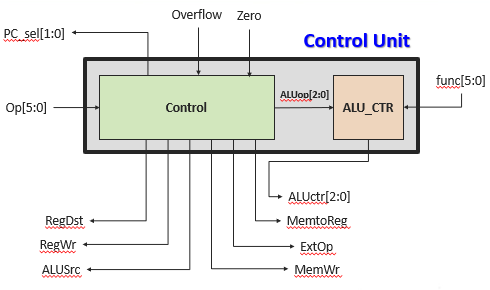
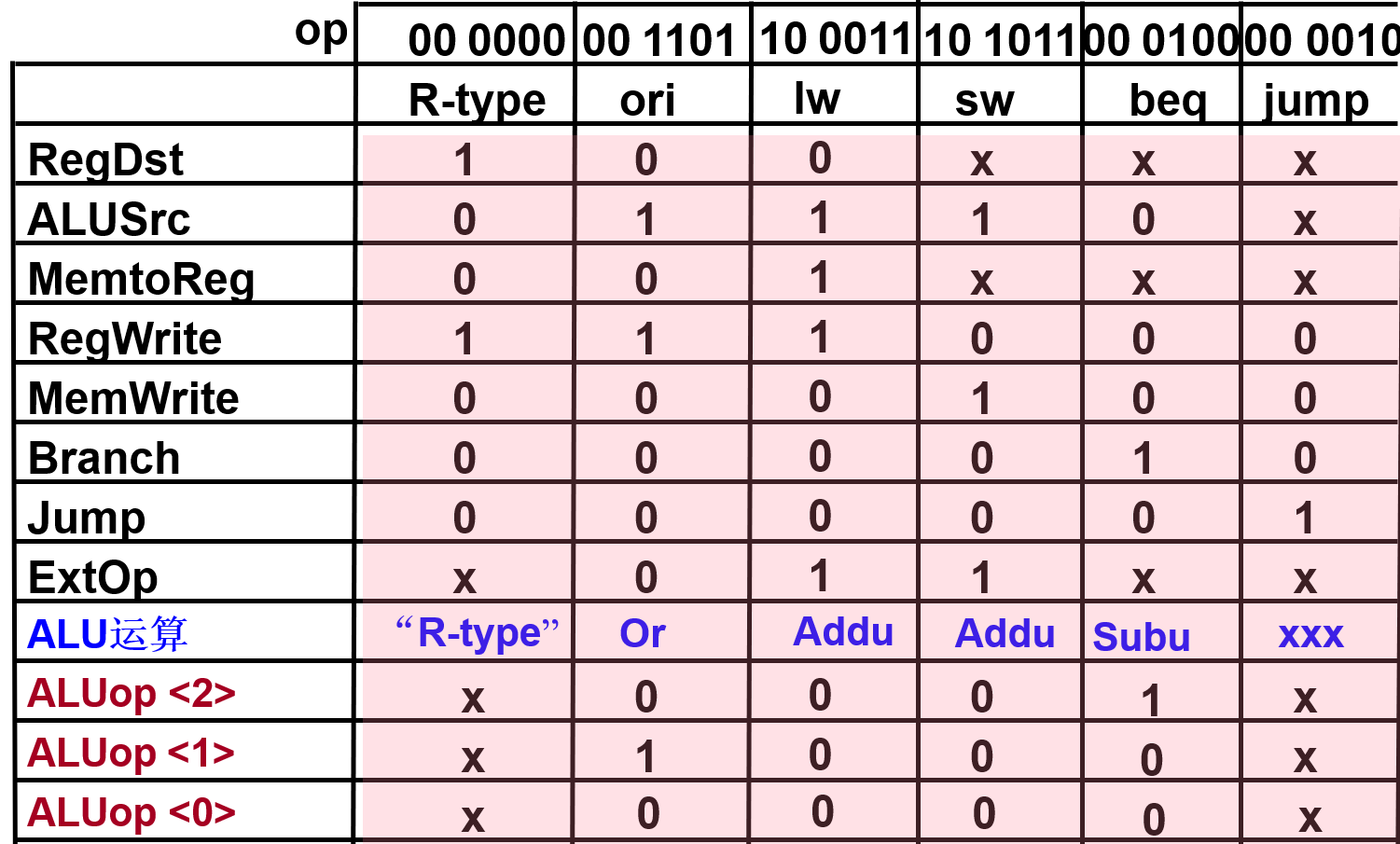


图4-1 控制器单元的结构图

* Control部件和ALU\_CTR部件的设计要能够满足MIPS中的三种指令，其控制字段与指令的关系如图所示：



Control部件中输入与输出的关系(源计组ppt)

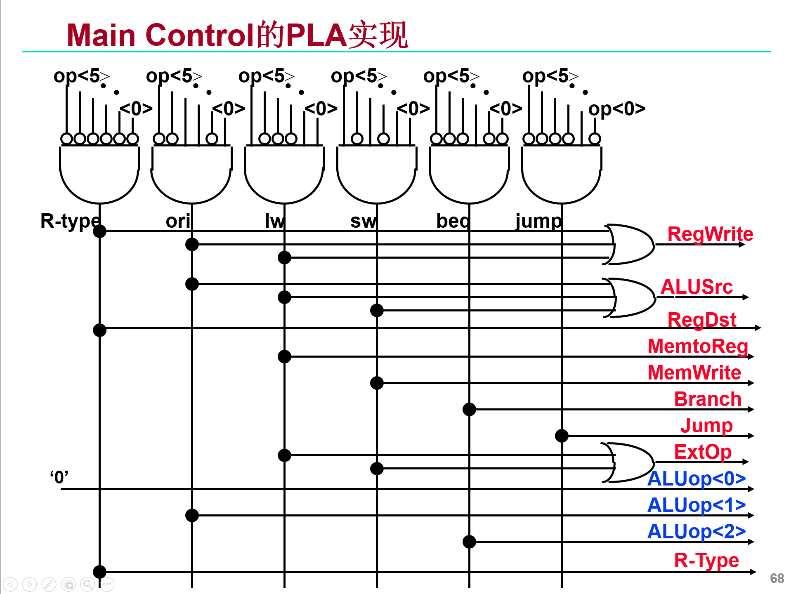
* 根据给出的关系图得出控制信号的逻辑表达式，在集成开发系统上进行编程和仿真并在开发板上进行实现，最后对控制电路进行封装形成Control\_Unit。

1. **实验器材（设备、元器件）**

* PC计算机：Xilinx ISE Design Suite 14.7集成开发系统、digilent.adept.system\_v2.10.2.exe
* FPGA数字电路开发平台：Anvyl（燧石TM）开发板

1. **实验步骤：**
2. **Control：**

* 根据计组PPT上面op指令中非R型指令的部分的指令特征以及对应的信号确定Control负责的功能。
* 参照上图中给定的控制单元的结构分析Control单元输入和输出的接口
* 根据指令ROM中的指令，确定哪些指令会用得到（比如上图并未提供Jump信号的接口但是实际上却用到了，故还需要添加Jump信号的接口）
* 参考计组PPT确定Control输出信号的组合逻辑（原理如下）

****

* **根据上面确定的组合逻辑，编写Control部分的代码**

module Control(

input [5:0] op,

     output RegDst, RegWrite, ALUSrc,

     output MemWrite, MemtoReg,

     output Branch,Jump,ExtOP,

     output [2:0] ALU\_op,R\_type);

    wire i\_Rt = ~|op;

    wire i\_lw = op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0];

    wire i\_sw = op[5] & ~op[4] & ~op[3] & op[2] & ~op[1] & op[0];

    wire i\_beq = op[5] & op[4] & op[3] & op[2] & ~op[1] & ~op[0];

    wire i\_ori = ~op[5] & ~op[4] & op[3] & op[2] & ~op[1] & op[0];

    wire i\_jump = ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & ~op[0];

    assign RegDst = i\_Rt;

    assign RegWrite = i\_Rt | i\_lw | i\_ori;

    assign ALUSrc = i\_lw | i\_sw | i\_ori;

    assign MemWrite = i\_sw;

    assign MemtoReg = i\_lw;

    assign Branch = i\_beq;

    assign Jump = i\_jump;

    assign ExtOP = ~i\_ori;

    assign ALU\_op[2] = i\_beq;

    assign ALU\_op[1] = i\_ori;

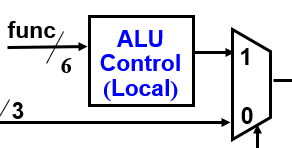
    assign ALU\_op[0] = 0;

    assign R\_type = i\_Rt;

endmodule

1. **AlU\_OP**

* 根据计组PPT上给定的ALU\_OP结构，确定ALU\_OP所需要的接口



* 根据R型指令对于的ALT\_Ctr编写对应的逻辑
* 由于7条指令系统中只有add(000)和sub(100)两条R型指令
* 故实际上只需满足以下条件即可：

ALUctr<0> = 0

ALUctr<1> = 0

ALUctr<2> = func<1>

* ALU\_OP实现代码如下:

module ALUop(

input [5:0] func,

    output [2:0] ALU\_op);

    assign ALU\_op[0] = 0;

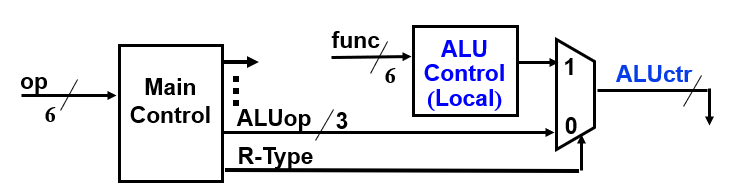
    assign ALU\_op[1] = 0;

    assign ALU\_op[2] = func[1];

endmodule

1. **ControlUnit的封装：**

* ControlUnit的整体结构如图所示：



* 故这里需要一个二选一选择器将2者连接，代码如下

module Control\_Unit(

    input [5:0] op, func,

    output RegDst, RegWrite, ALUSrc,

    output MemWrite,MemtoReg,

    output Branch,Jump,ExtOP,

    output [2:0] ALU\_ctr);

    wire [2:0] ALU\_op1;

    wire [2:0] ALU\_op2;

    wire R\_type;

    Control U0(.op(op),.RegDst(RegDst),.RegWrite(RegWrite),

        .ALUSrc(ALUSrc),.MemWrite(MemWrite),.MemtoReg(MemtoReg),

        .Branch(Branch),.Jump(Jump),.ExtOP(ExtOP),.ALU\_op(ALU\_op1),

        .R\_type(R\_type));

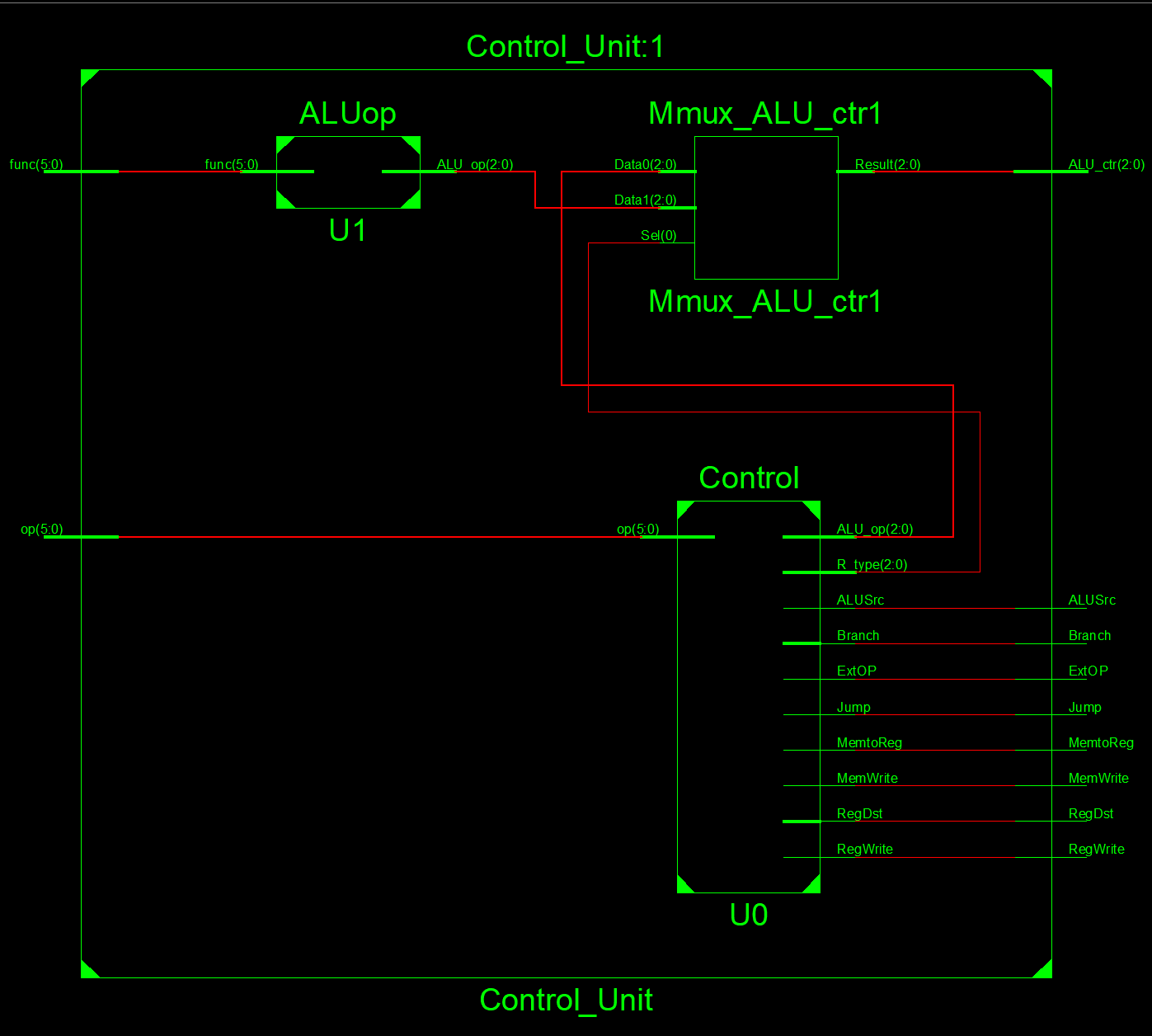
    ALUop U1(.func(func),.ALU\_op(ALU\_op2));

    //这里根据是否是R型指令对ALUctr进行选择输出

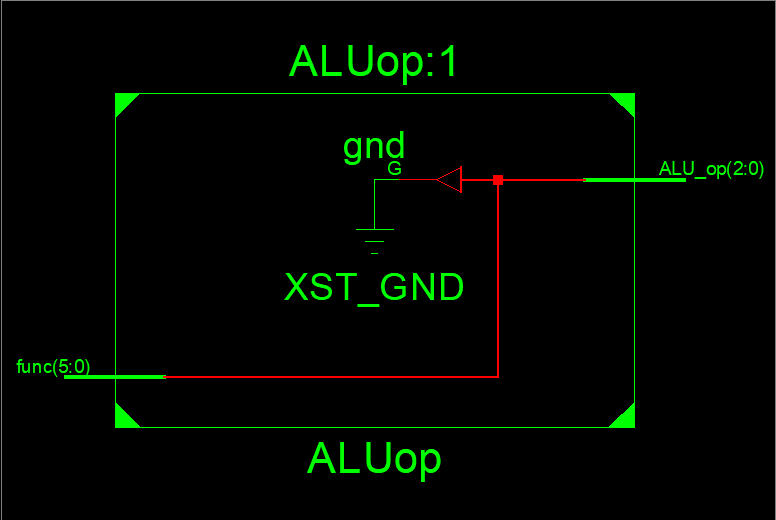
    assign ALU\_ctr = R\_type? ALU\_op2:ALU\_op1;

endmodule

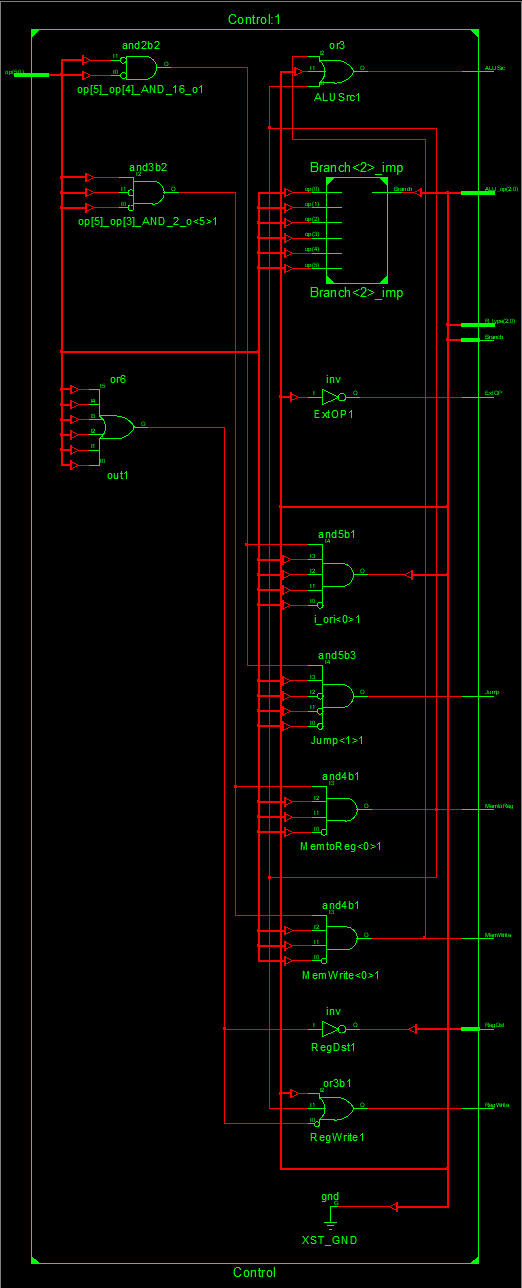
* ControlUnit整体的RTL图：



* ALU\_OP的整体RTL图：



* **Control部分的RTL图（图片过长，请放大查看）：**



* **测试：由于单独测试控制单元相对来说比较麻烦，故这里放在后面测试MIPS\_CPU的时候一起测试。**

1. **实验结论、心得体会和改进建议：**

**心得：**

* 做实验的时候千万不能一味的抄去PPT上面的代码，应该结合实际功能和场景去理解并设计。比如说这里的ControlUnit并没有提供Jump信号，但是实际上指令ROM里面的第一条就是Jump指令，所以需要手动添加。
* 某一个模块在仿真的时候过了并不代表数据正确了，也不代表连接了其他的模块就能够正常工作，比如说老师给的扩展器实际上仅仅为符号扩展器，但是在做ori指令的时候需要做无符号的扩展，所以应当在控制单元里面输出一个ExtOP信号，来保证数据的正确性。当然不这么做仿真跑出来任然有数据且一片绿，但是实际上数据是错误的。
* 在编写Control的时候应当严谨，保证输入的op能够正确的对应到ALU的控制信号和其他控制信号上，因为一般来说只要接上线了就会有数据，但是实际上很有可能op或func对应的控制信号是错误的，但是仿真跑出来任是一片绿色，这些问题就只能够通过人工手动调试一一检查才能够找的出来。

**！！！改进建议：**

* **强烈要求教一下如何使用Verilog 如何DEBUG，调试ControlUnit踩了不少的坑。**

**实验四**

1. **实验名称：单周期CPU的设计与实现**
2. **实验学时：4**
3. **实验内容和目的：**

* 实验内容：设计并实现Fetch部件，即取值电路部件。对之前实验中设计的部件按照单周期计算机CPU结构进行封装。最终实现可运行的MIPS CPU。
* 实验目的：了解并掌握基于MIPS指令集的单周期计算机CPU的结构和工作原理，可以根据单周期计算机结构图和MIPS指令集的相关知识设计并实现一个单周期计算机的CPU。

1. **实验原理：**

* 对取值电路即Fetch部分电路进行封装
* 对MIPS CPU按照单周期计算机结构图进行封装。
* 将CPU,RAM,ROM,SEG,AN等组件封装为一个SOC

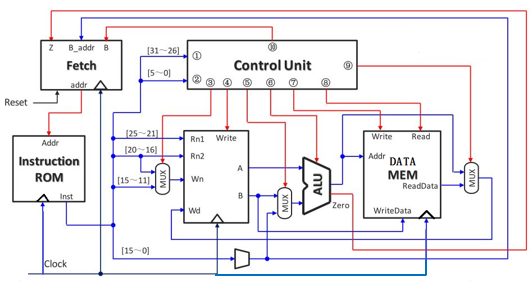


图4-1 单周期计算机结构图

1. **实验器材（设备、元器件）**

* PC计算机：Xilinx ISE Design Suite 14.7集成开发系统、digilent.adept.system\_v2.10.2.exe
* FPGA数字电路开发平台：Anvyl（燧石TM）开发板

1. **实验步骤：**
   1. 设计取址电路Fetch：
      * + 根据计组PPT提供的Fetch模块的结构与所学Fetch的功能，分析Fetch所需要的接口。
        + 根据Fetch的输出的不同指令类型和输入的不同信号，将其拆分为多个内部元件来完成不同地址的生成。
        + 分析Fetch输入的控制信号，确定内部元件之间接口如何连接
        + Fetch的相关代码（各个组件功能见注释）：

module Fetch(

    input Reset,Clock,Jump,Branch,Zero,

    input [31:0] br\_imm,Instruction,

    output [31:0]addr);

wire[31:0] pc\_plus4,br\_imm\_ext,br\_addr\_extra,

br\_addr\_result,jump\_addr\_result,next\_pc;

wire [1:0] select;

    //由输入的信号确定选择信号

assign select = (Jump) ? 2'b10 :(Branch & Zero) ? 2'b01 : 2'b00;

//br\_imm\_ext乘4变成地址

    Left\_2\_Shifter Left2Shifter(br\_imm, br\_addr\_extra);

//pc + 4得到下一条

    ADD32 PCAdder(4,addr,pc\_plus4);

//执行pc + 4 + imm \* 4得到Brach指令的地址

    ADD32 BrachAddrGetter(pc\_plus4,br\_addr\_extra,br\_addr\_result);

//执行PC高四位和Ins的低26位相加获取Jump指令的地址

    Joint\_Addr JumpAddrGetter (addr,Instruction,jump\_addr\_result);

//选择执行哪一种指令，select=11的情况不存在

MUX32\_4\_1 NextPcGetter(pc\_plus4,br\_addr\_result,

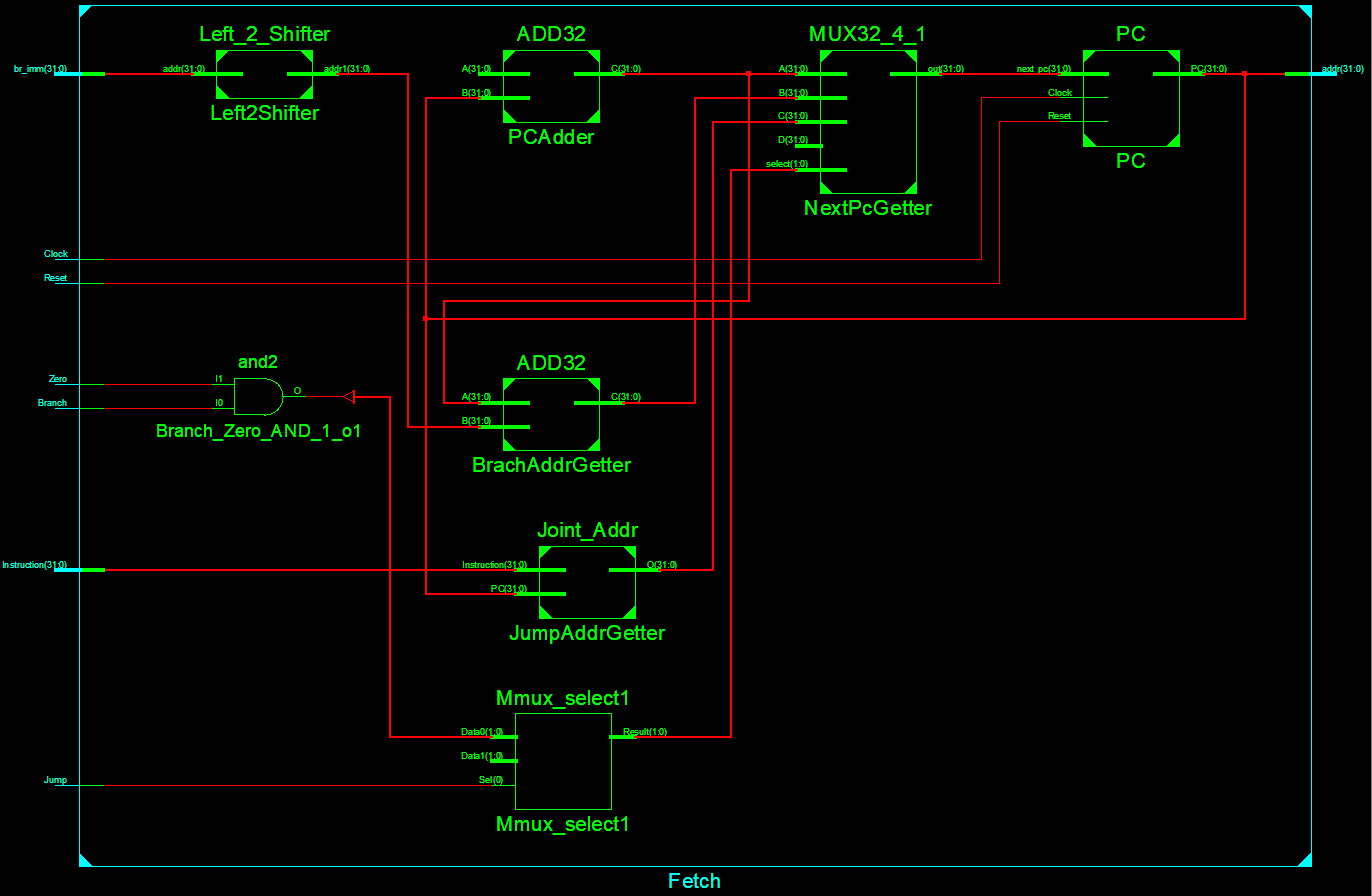
jump\_addr\_result,32'h0,select,next\_pc);

//PC的输出模块

    PC PC(Clock,Reset,next\_pc,addr);

endmodule

* + - * Fetch的RTL结构图：



* + - * Fetch的仿真测试代码（设计逻辑见注释）：

   //这里外挂一个ROM进行测试

    wire [31:0]inst;

    INST\_ROM ROM(addr,inst);

    //由于没有ControlUnit，因此这里需要手动将控制信号初始化

    initial begin

        Reset = 1;

        Clock = 1;

        Jump = 1;

        Branch = 0;

        Zero = 0;

        br\_imm = 0;

        Instruction = 32'b000010\_00000\_00000\_0000\_0000\_0000\_0101;

        #100;

        Reset=0;

        #100;

        Jump = 0;

    end

    always begin

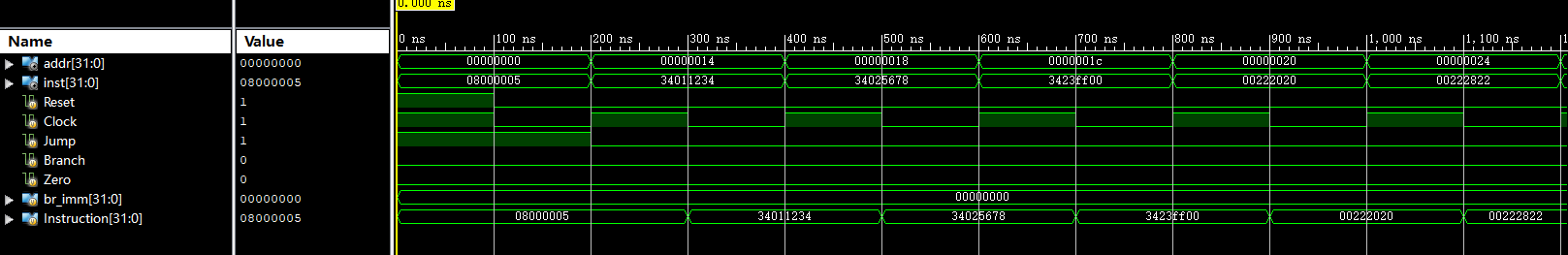
     #100;

     Clock = ~Clock;

     Instruction = inst;

end

* + - * 测试结果如下（16进制显示的数据，图片较小，请放大查看）：



* 1. 封装MIPS\_CPU
     + - 将前面设计好的各个模块封装在一起，构成一个简易的MIPS\_CPU
       - 实现代码如下（组件介绍见注释）：

module MIPS\_CPU(

    input [31:0]Instruction,

    input Reset, Clock,

    //准备写道寄存器堆里面的数据

    input [31:0] DataToWd,

    output [31:0] ALU\_result, Ext\_Imm, addr, Out1,Out2,

    output MemWrite,MemtoReg);

    wire RegDst,RegWrite, ALUSrc, Branch, Zero,Jump,ExtOP ;

    wire [2:0] ALU\_ctr;

    wire [4:0] Wn;

    wire[31:0] ALU\_B;

    //取址电路Fetch

    Fetch Fetch(Reset,Clock,Jump,Branch,Zero,Ext\_Imm,Instruction,addr);

    //控制单元

    Control\_Unit ControlUnit(.op(Instruction[31:26]), .func(Instruction[5:0]),

    .RegDst(RegDst), .RegWrite(RegWrite), .ALUSrc(ALUSrc),

    .MemWrite(MemWrite), .MemtoReg(MemtoReg),

    .Branch(Branch), .Jump(Jump),.ExtOP(ExtOP),.ALU\_ctr(ALU\_ctr));

    //Wn的5位2选一选择器

    MUX5\_2\_1 MUX5\_2\_1(Instruction[20:16], Instruction[15:11], RegDst, Wn);

    //寄存器堆

    RegFile RegFile(Instruction[25:21], Instruction[20:16], Wn, RegWrite,Clock,DataToWd, Out1, Out2);

    //扩展器

    Extender Extender(Instruction[15:0],ExtOP, Ext\_Imm);

    //ALU的第二个输入的二选一选择器

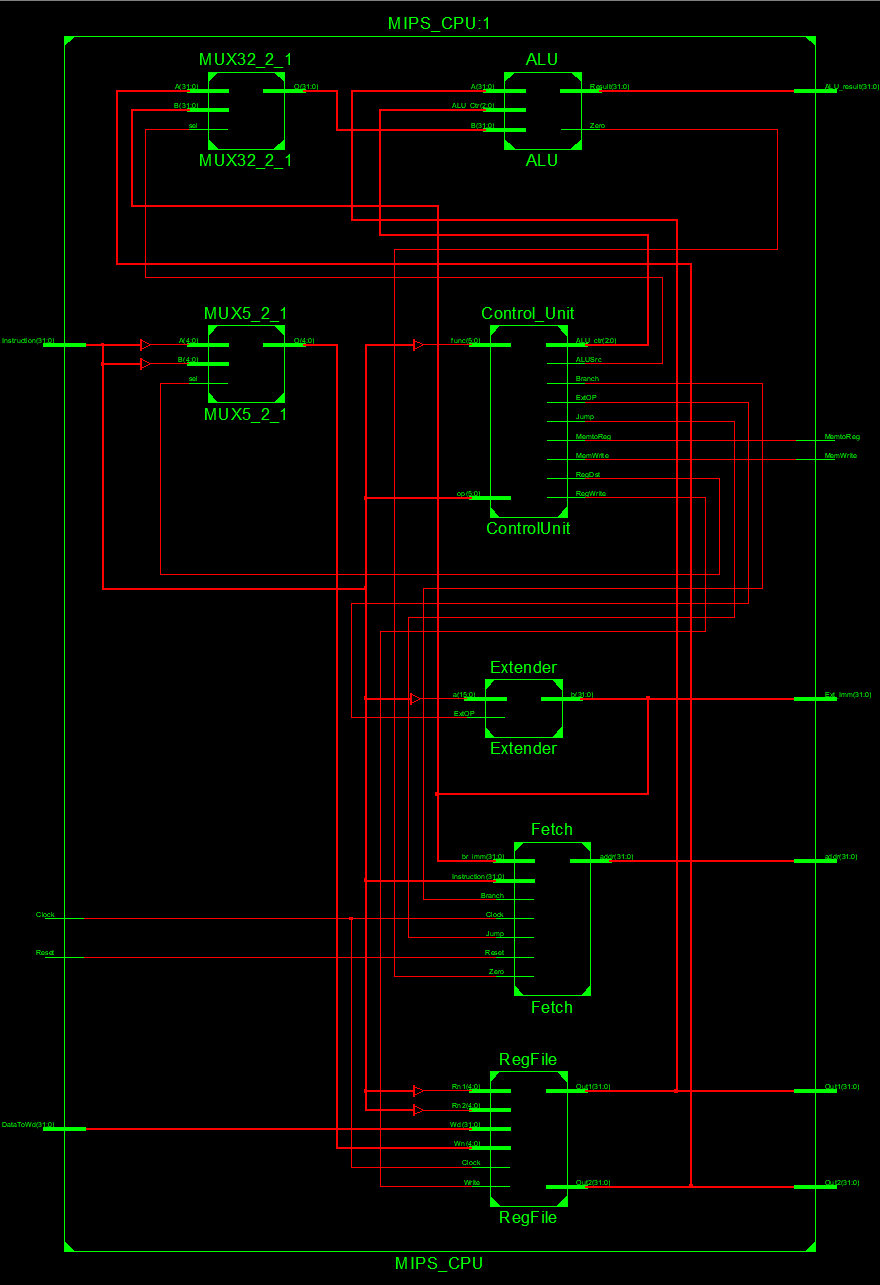
    MUX32\_2\_1 MUX32\_2\_1(Out2, Ext\_Imm, ALUSrc, ALU\_B);

    //ALU

    ALU ALU(Out1, ALU\_B, ALU\_ctr, ALU\_result, Zero);

endmodule

* CPU的RTL结构图：



* CPU的仿真测试代码：

   //这里任然需要外挂一个Rom

    wire [31:0] inst;

    INST\_ROM ROM(addr,inst);

    initial begin

    //由于已经包含ControlUnit了，故这里只需将Reset置0即可

        Instruction = 0;

        Reset = 1;

        Clock = 1;

        DataToWd = 0;

        #100;

        Reset = 0;

        end

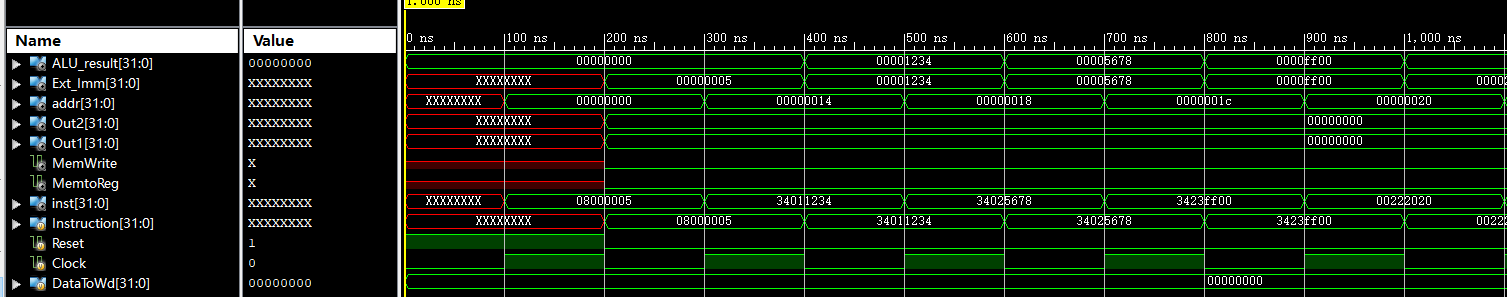
    always begin

        #100;

        Instruction = inst;

        Clock = ~Clock;

end

* CPU的仿真结果（方便观察，数据用的16进制，请放大查看）
* 由于这个没有RAM，且DataToWd一直为0，故寄存器堆输出的结果恒为0，此时寄存器还未和ALU进行关联，ALU的结果也没有写到RAM里。
* 可以看到前面有一段红色的部分，这个实际上是初始的时候，各个组件输出的值并不能够确定，所以值为X,在Reset完成后，就变成0了。
* 可以观察下图看到，CPU已经能够正常的进行取指令了，并且ALU也有结果输出，说明这个时候CPU的主要部分已经能够正常运行了。当然，数据的正确与否任有待考证。
* 
  1. 封装整个SOC
* 将前面设计好的CPU，和ROM，RAM，以及Hex7seg（数码管）进行连接，并将整个结构封装成为一个完整的SOC
* 代码如下：

module SOC(Clock, Reset,

         DISP\_Seg, //数码管显示：PC（高2管），ALU运算结果（低4管）

         AN,          //数码选择信号

         Test\_signal,

         ALU\_Result, Ext\_imm, addr,Out1,Out2,Instruction,

         MemWr,MemtoReg);

    input Clock, Reset;

    output [7:0] DISP\_Seg;

    output [5:0] AN;

    output [31:0] ALU\_Result, Ext\_imm, addr,Out1,Out2,Instruction;

    output MemWr,MemtoReg;

    output [23:0] Test\_signal;

    reg [31:0] ClockNum = 0;

    wire MemWr,Step;

    wire [31:0] ALU\_Result, Ext\_imm,DataToWd,addr,MemOutData;

    assign Test\_signal = {addr[9:2],ALU\_Result[15:0]};

    assign LED0 = Step;

    always @(posedge Clock) ClockNum <= ClockNum + 1;

    //数码管显示

    Hex7seg\_decode seg(Test\_signal,ClockNum[2:0],DISP\_Seg,AN);

    //主存，用来存取数据

    DATA\_MEM RAM(Out2,ALU\_Result,MemWr,Clock,MemOutData);

    //ALU的计算结果或者Ram中读取的值进行二选一，准备输出到寄存器堆的Wd里

    MUX32\_2\_1 mux1 (ALU\_Result, MemOutData,MemtoReg, DataToWd);

    //ROM，通过地址从ROM中获取指令

    INST\_ROM ROM(addr,Instruction);

    //CPU,第一个参数为指令，第二个参数为是否重设PC，准备写到Wd里面的值

    MIPS\_CPU cpu(Instruction,Reset,Clock,DataToWd,

    //ALU计算输出的值，立即数符号扩展的输出值,PC,寄存器读出的out2

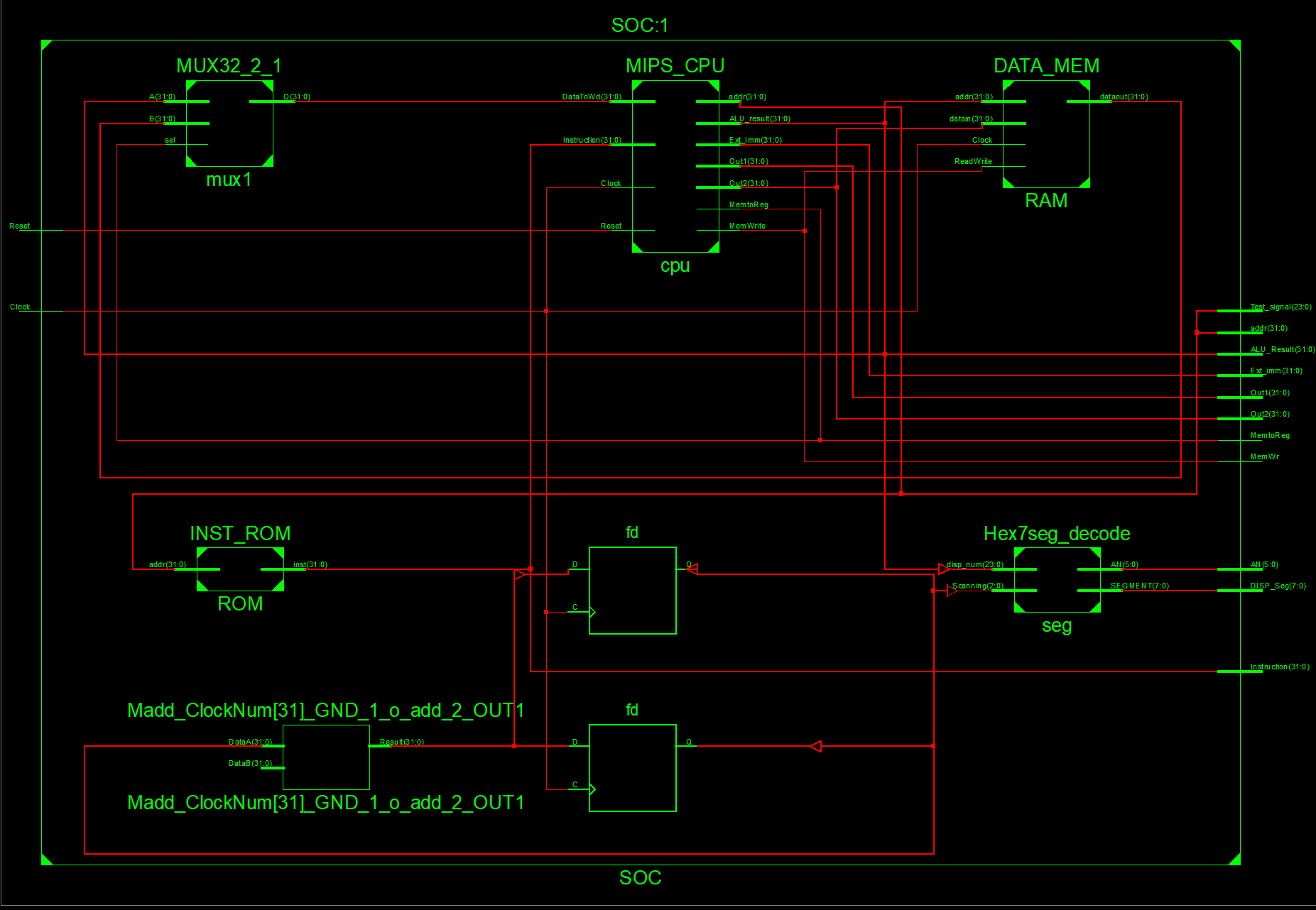
    ALU\_Result,Ext\_imm,addr,Out1,Out2,

    //MEM的两个信号

MemWr,MemtoReg);

endmodule

* SOC的RTL图如下：



* SOC的仿真测试代码如下：

 initial begin

        Reset = 1;

        Clock = 0;

        #200;

Reset = 0;

    end

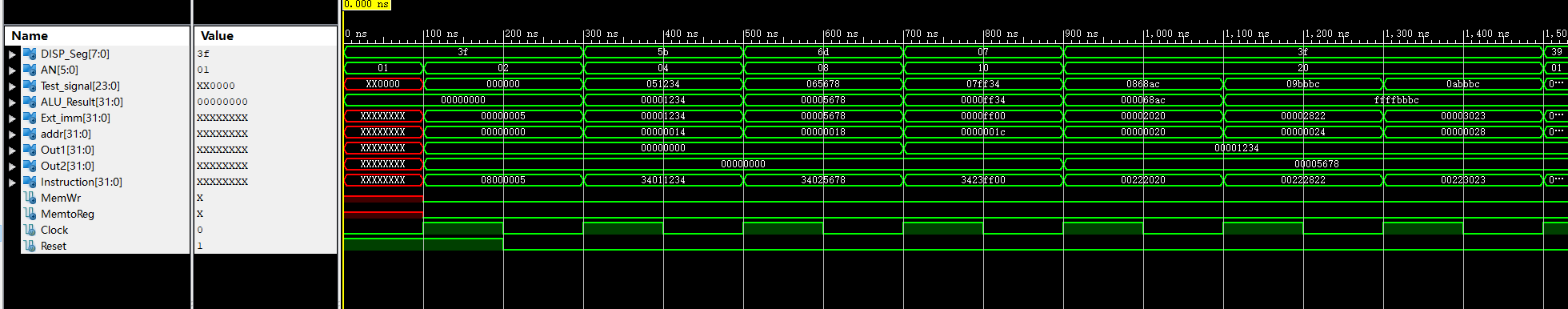
    always begin

        #100;

        Clock = ~Clock;

    end

* SOC的仿真测试结果如下（16进制显示，请放大查看）：



* SOC仿真测试结果分析：
* ROM的指令（用于对照参考实际CPU跑出的结果）：

   wire [31:0] rom [0:31];

    //j 5H

     assign rom[5'h00] = 32'b000010\_00000\_00000\_0000\_0000\_0000\_0101;

     assign rom[5'h01] = 32'hAAA0 ; // Not use

     assign rom[5'h02] = 32'hAAA1 ; // Not use

     assign rom[5'h03] = 32'hAAA2 ; // Not use

     assign rom[5'h04] = 32'hAAA3 ; // Not use

    //ori R1 , R0 , 1234H

     assign rom[5'h05] = 32'b001101\_00000\_00001\_0001\_0010\_0011\_0100;

    //ori R2 , R0 , 5678H

     assign rom[5'h06] = 32'b001101\_00000\_00010\_0101\_0110\_0111\_1000;

    //ori R3 , R1 , FF00H

     assign rom[5'h07] = 32'b001101\_00001\_00011\_1111\_1111\_0000\_0000;

    //add R4 , R1 , R2

     assign rom[5'h08] = 32'b000000\_00001\_00010\_00100\_00000\_100000;

    //sub R5 , R1 , R2

     assign rom[5'h09] = 32'b000000\_00001\_00010\_00101\_00000\_100010;

    //subu R6 , R1 , R2

     assign rom[5'h0A] = 32'b000000\_00001\_00010\_00110\_00000\_100011;

    //slt R7 , R1 , R2

     assign rom[5'h0B] = 32'b000000\_00001\_00010\_00111\_00000\_101010;

    //sltu R8 , R2 , R1

     assign rom[5'h0C] = 32'b000000\_00010\_00001\_01000\_00000\_101011;

    //sw R4 , R0 , 4H

     assign rom[5'h0D] = 32'b101011\_00000\_00100\_0000\_0000\_0000\_0100;

    //lw R9 , R0 , 4H

     assign rom[5'h0E] = 32'b100011\_00000\_01001\_0000000000000100;

    //add R10 , R1 , R9

     assign rom[5'h0F] = 32'b000000\_00001\_01001\_10010\_00000\_100000;

    //beq R1 , R2 , 1234H

     assign rom[5'h10] = 32'b000100\_00001\_00010\_0001\_0010\_0011\_0100;

    //beq R1 , R1 , FFEEH

     assign rom[5'h11] = 32'b000100\_00001\_00001\_1111\_1111\_1110\_1110;

     assign rom[5'h12] = 32'h0 ; // Not use

     assign rom[5'h13] = 32'h0 ; // Not use

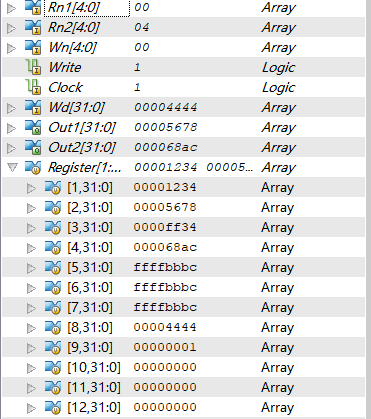
     assign rom[5'h14] = 32'h0 ; // Not use

     assign rom[5'h15] = 32'h0 ; // Not use

     assign rom[5'h16] = 32'h0 ; // Not use

     assign inst = rom[addr[6:2]];

* SOC仿真DEBUG截获的寄存器里的数据：



* 手动验证结果：

1. rom[5’h00]正确执行，(仿真波形图得)PC出现了跳转
2. rom[5’h05]正确执行，1234H与0取或的值正确存放到了R1中
3. rom[5’h06]正确执行，5678H与0取或的值正确存放到了R2中
4. rom[5’h07]正确执行，R1与FF00H取或的值正确存放到了R3
5. rom[5’h08]正确执行，R2与R1相加的值正确存放到了R4中
6. rom[5’h09]正确执行，R2与R1相减的值正确存放到了R5中
7. rom[5’h0A]正确执行，R2与R1相减(无符号)的值正确存放到了R6中
8. rom[5’h0B]正确执行，R2与R1相减的值正确存放到了R7中

……

* 仿真测试结论：
* 初始的时候有一段红色部分，实际上是各个组件输出的值还并未确定，所以此时为x，在Reset完成之后，就被正确的初始化
* 可以看到，这个时候寄存器堆已经有值输出，并且数据正确（见上面的测试结果），即说明RAM正常运行。
* 另外由于寄存器堆得出的结果和指令ROM中对应的结果完全一致，说明ControlUnit编写正确，并且各个控制信号的生成逻辑正确，从而验证了实验三的控制单元的相关编写的正确性。
* 至此，一个完整的单周期的MIPS\_CPU就已经设计完成。

1. **实验结论、心得体会和改进建议：**

**心得：**

* 在CPU的设计过程中踩了不少的坑，从简单到复杂一层一层的包装。从Fetch模块的正常运行，到整个简易CPU的完整工作，调试了很久，也花了不少时间去检查代码，理解代码，再到从自己的理解去修改去校验去设计，学到了不少的东西，对整个MIPSCPU的理解也更加的透彻。
* 实验课程不能仅仅抄写PPT上面的代码即可，其实老师提供的PPT上面已经提供了大量元件的代码，但是实际上因为不同代码之间可能存在版本的差异，不同模块之间的接口实际上是有错的，所以直接抄下来的代码是跑不通的，更主要的应该是自己去理解CPU的组成原理，然后根据自己的情况去调整或者设计，当然也可以根据计组教材提供的完整的MIPSCPU去仿制一个类MIPSCPU。
* 无论是ALUCRT还是OP FUNC，其实各种指令对应的控制信号都是人为制定的，所以在设计CPU的时候，并没有完全照搬书上或者PPT上面的必要，完全可以根据自己的喜好设计相关的组合逻辑，然后自定义相关的指令集和ROM里边的操作指令，这样对于CPU的理解会更加的全面而不局限于教材。
* 还是之前提到的一个问题，不要因为仿真跑出来一片绿色就以为自己测试通过了，很有可能存在指令对应的操作实际上是错误的，但是由于线路连接正常，所以任然有数据计算出来。但是实际上这个时候CPU的控制逻辑是错误的，所以应当更加严谨更加仔细。