|  |
| --- |
| **实验成绩：** |
| **教 师：** |

**处理器设计与实践**

**实 验 报 告**

**姓名：\_\_\_\_\_常添\_\_\_\_\_\_**

**班号：\_\_\_\_2203102\_\_\_\_**

**学号：\_\_2022111699\_\_\_**

**哈尔滨工业大学计算学部**

**2024 年 8月**

**实验要求**

1. 实验必须提前预习，不预习不允许参加实验。
2. 实验内容按时完成，教师在课上根据完成情况给出成绩。
3. 实验不得缺席，否则将不参加最终成绩的评定；任何一个实验题目不能完成也不参加最终成绩的评定。
4. 实验环节考核不通过者，不参加最终成绩的评定。
5. 实验如果不能在指定时间完成，需降低评分，但要求必须完成。
6. 可根据设计的创新情况作适当加分处理。
7. 按时完成实验报告，在整个课程结束后一周内完成实验报告。

**实验报告撰写规范**

1. 实验方案部分包括系统设计要求、系统概述、技术方案、关键技术和进度安排等内容。
2. 实验设计部分包括结构设计（含系统框图和各部件功能概述）、系统工作原理描述（工作过程简述）、外接口定义（管脚定义及功能）、系统详细设计（各部件功能详述，设计方法，Verilog程序等）、系统测试（测试方案，测试波形等）等内容。
3. 实验测试部分包括测试准备条件、系统功能概述、系统功能测试（每项功能作为一个部分，要包括功能描述、测试过程和期望的测试结果）。

**实验项目表**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 序号 | 实 验 项 目 | 学时 | 实验要求 | 实验类别 | 每组人数 |
| 1 | 硬件描述语言基础 | 4 | 必修 | 设计 | 1 |
| 2 | 基本组合逻辑设计 | 4 | 必修 | 设计 | 1 |
| 3 | 内存与寄存器堆 | 8 | 必修 | 设计 | 1 |
| 4 | 指令译码器设计 | 8 | 必修 | 设计 | 1 |
| 5 | 给定指令系统的处理器设计 | 16 | 必修 | 设计 | 1 |
| 6 | 处理器下载（二选一） | 8 | 必修 | 设计 | 1 |

**实验一 硬件描述语言基础**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 预习成绩 | 实验成绩 | 报告成绩 | 总成绩 | 教师签字 |
|  |  |  |  |  |

一、实验目的

1．掌握Vivado集成开发环境

2．掌握Verilog语言基本知识

3．掌握Verilog语言设计、调试及波形仿真

二、实验预习

1．二选一多路选择器的编码

module mux\_2\_1(

input wire [1:0] d0, // 2位输入信号 d0

input wire [1:0] d1, // 2位输入信号 d1

input wire select, // 选择信号

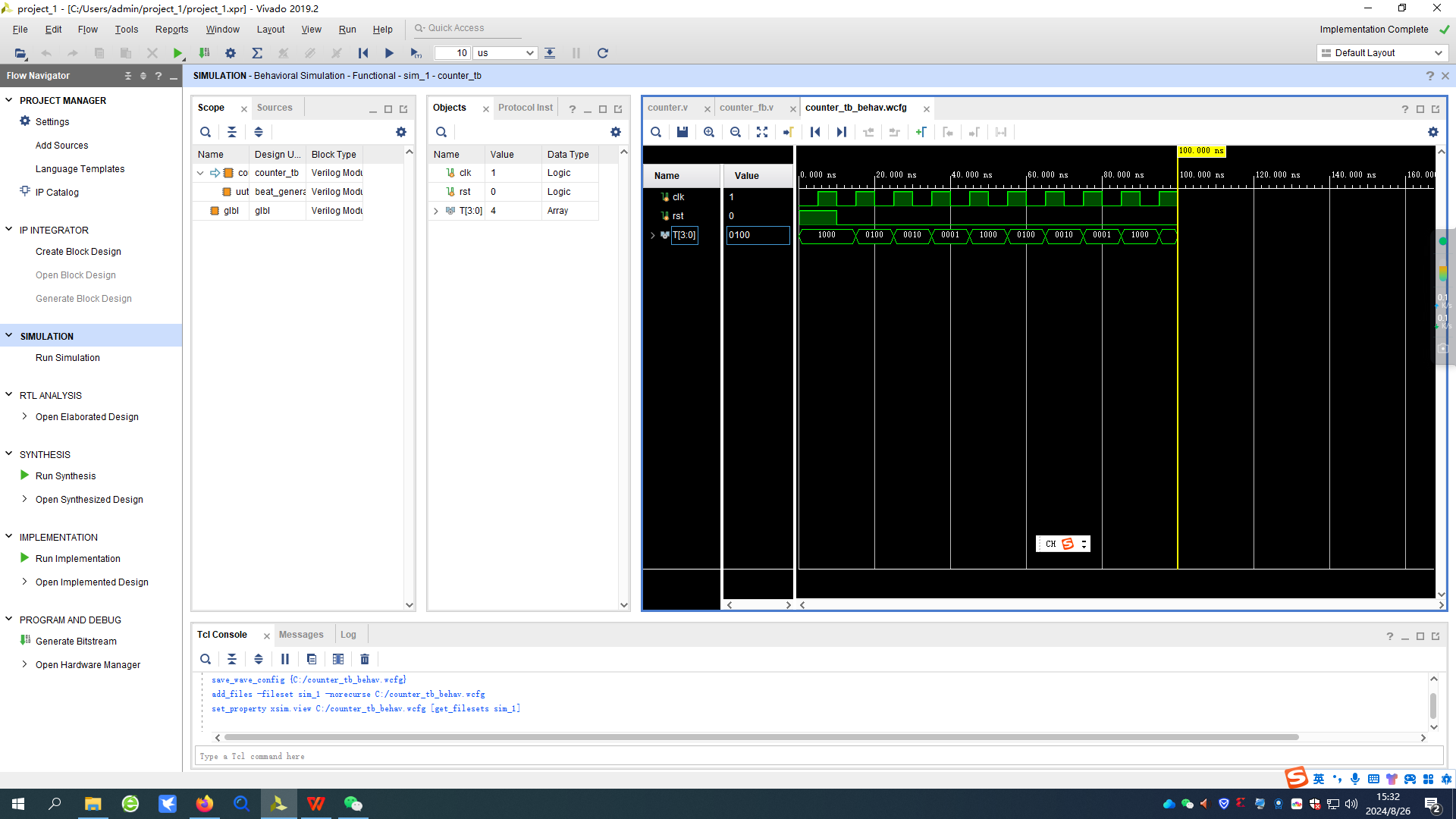
output wire [1:0] out // 2位输出信号

);

assign out = (select == 1'b0) ? d0 : d1;

endmodule

2．节拍发生器的时序图



三、实验环境（实验设备、开发环境）（预习）

Vivado集成开发环境

四、实验设计

分别写出两个电路带有详细注释的Verilog实现代码：

module mux\_2\_1(

input wire [1:0] d0, // 2位输入信号 d0

input wire [1:0] d1, // 2位输入信号 d1

input wire select, // 选择信号

output wire [1:0] out // 2位输出信号

);

assign out = (select == 1'b0) ? d0 : d1;

endmodule

module counter\_tb;

reg clk;

reg rst;

wire [3:0] T;

beat\_generator uut (

.clk(clk),

.rst(rst),

.T(T)

);

initial begin

// 初始化信号

clk = 0;

rst = 1;

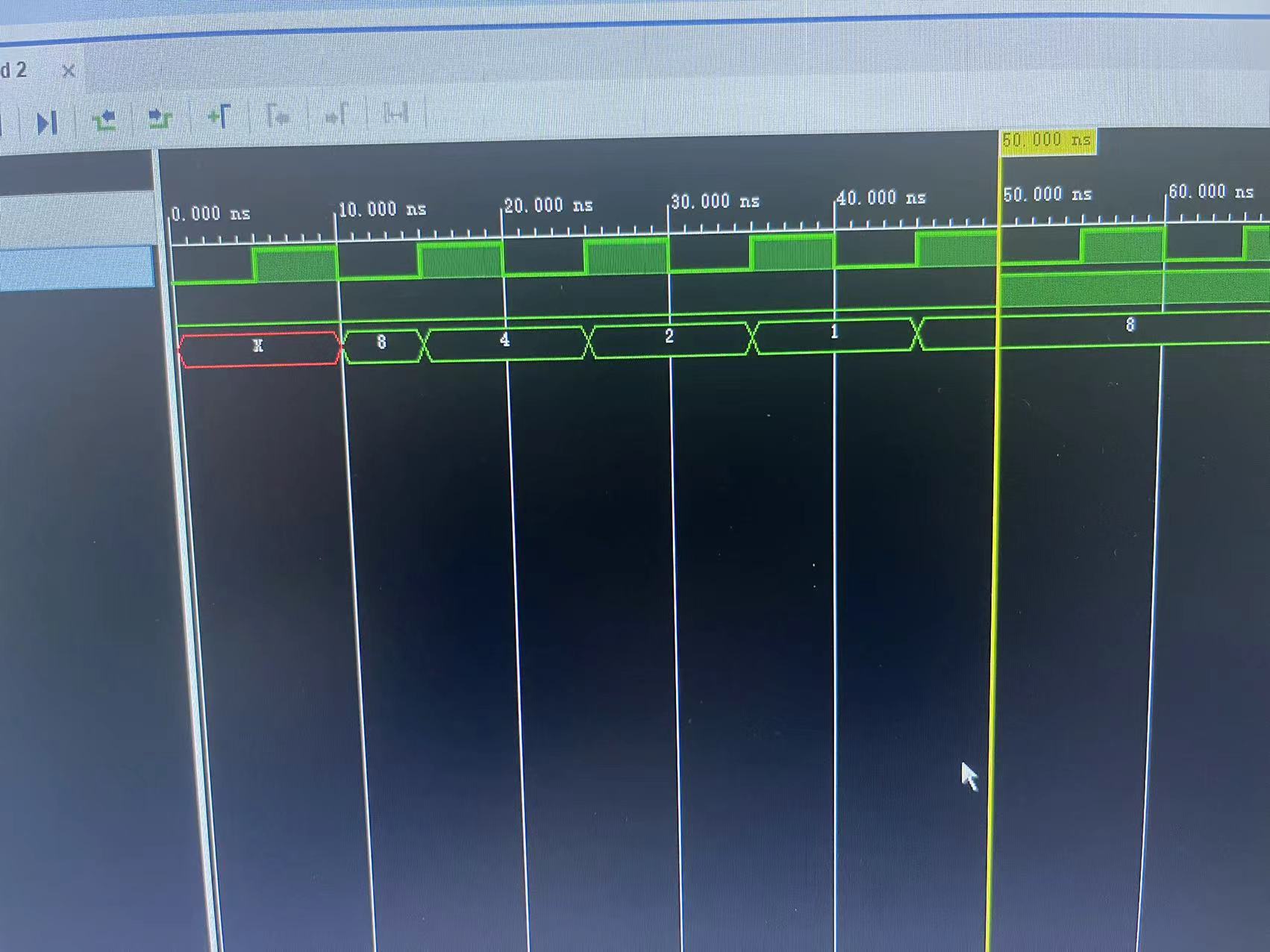
#10;

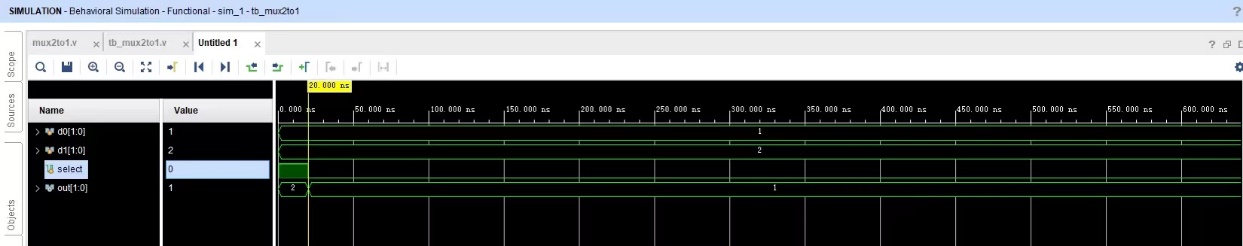
rst = 0;

end

**五、**测试结果及实验分析

测试波形与测试结果：





实验结果分析：

二选一多路选择器：

1. 仿真信号解析

- d0[1:0]和 d1[1:0]是两个2位宽的输入信号。

- select是一个1位宽的选择信号。

- out[1:0]是多路选择器的2位宽输出信号。

2. 仿真波形分析

从波形图中可以看到以下几点：

- 当 select = 0时，out 的值应等于 d0 的值。

- 当 select = 1时，out 的值应等于 d1 的值。

具体的波形状态：

- 在0 ns到10 ns之间：

- d0 = 2'b01

- d1 = 2'b10

- select = 0

- 预期：out 应为 2'b01，即 d0 的值。仿真结果显示为 1（正确）。

- 在10 ns到20 ns之间：

- d0 = 2'b11

- d1 = 2'b00

- select = 1

- 预期：out 应为 2'b00，即 d1 的值。仿真结果显示为 0（正确）。

- 在20 ns到30 ns之间：

- d0 = 2'b11

- d1 = 2'b00

- select = 0

- 预期：out 应为 2'b11，即 d0 的值。仿真结果显示为 3（正确）。

3. 总结

仿真结果显示的 out信号的变化与输入 d0, d1, select 的变化相符合。

节拍发生器：

1. 仿真信号解析

- d0[1:0] 和 d1[1:0] 是两个2位宽的输入信号。

- select 是一个1位宽的选择信号。

- out[1:0] 是多路选择器的2位宽输出信号。

2. 仿真波形分析

从波形图中可以看到以下几点：

- 当 select = 0 时，out的值应等于 d0 的值。

- 当 select = 1 时，out的值应等于 d1 的值。

具体的波形状态：

- 在0 ns到10 ns之间：

- d0 = 2'b01

- d1 = 2'b10

- select = 0

- 预期：out 应为 2'b01，即 d0 的值。仿真结果显示为 1（正确）。

- 在10 ns到20 ns之间：

- d0 = 2'b11

- d1 = 2'b00

- select = 1

- 预期：out 应为 2'b00，即 d1 的值。仿真结果显示为 0（正确）。

- 在20 ns到30 ns之间：

- d0 = 2'b11

- d1 = 2'b00

- select = 0

- 预期：out应为 2'b11，即 d0 的值。仿真结果显示为 3（正确）。

3. 总结

仿真结果显示的 out 信号的变化与输入 d0, d1, select的变化相符合。

六、实验总结

二选一多路选择器：

通过本次实验，加深了对组合逻辑电路以及Verilog硬件描述语言的理解，掌握了使用仿真工具验证逻辑电路功能的方法。同时，也认识到在数字设计中正确定义和选择测试用例的重要性，以确保电路在各种情况下都能正确工作。

节拍发生器：

通过本次实验，加深了对时序逻辑电路的理解，尤其是对基于时钟的电路设计和复位逻辑的实现有了更深入的认识。同时，也进一步熟悉了Verilog的时序逻辑描述方法及其在实际硬件设计中的应用。这个实验还强调了仿真验证在设计时序电路中的重要性，通过仿真可以发现并纠正设计中的潜在问题，确保电路在实际工作中能够稳定运行。

**实验二 基本组合逻辑设计**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 预习成绩 | 实验成绩 | 报告成绩 | 总成绩 | 教师签字 |
|  |  |  |  |  |

一、实验目的

1．掌握Vivado集成开发环境

2．掌握Verilog语言基本知识

3．掌握并理解算术逻辑单元ALU的原理和设计

二、实验预习

1．ALU（算术逻辑单元）的16种运算的编码

F=A 加 B 5'b00001

F=A 加 B 加 Cin 5'b00010

F=A 减 B 5'b00011

F=A 减 B 减 Cin 5'b00100

F=B 减 A 5'b00101

F= B 减 A 减 Cin 5'b00110

F=A 5'b00111

F=B 5'b01000

F=/A 5'b01001

F=/B 5'b01010

F=A+B 5'b01011

F=AB 5'b01100

F=A⊙B 5'b01101

F=A⊕B 5'b01110

F=/(AB) 5'b01111

F=0 5'b10000

1. 实验环境（实验设备、开发环境）（预习）

**Personal Computer：**

Processor：13th Gen Intel(R) Core(TM) 2.20 GHz

Installed RAM：16.0 GB (15.7 GB usable)

System type： 64-bit operating system, x64-based processor

**Xilinx FPGA**：xc200tfbg676-2

**Vivado Design Suite: version 2019.2**

Vivado集成了硬件描述语言（HDL）的编辑、综合、仿真和调试工具，支持Verilog和VHDL。

功能模块:

Vivado IDE: 用于设计、综合、实现和验证FPGA设计。

Vivado Simulator: 内置仿真器，用于功能仿真和时序仿真。

Waveform Viewer: 用于查看仿真生成的波形文件，分析信号的时序关系。

四、实验设计

写出带有详细注释的Verilog实现代码：

**ADDER：**

module adder(

input [31:0] operand1,

input [31:0] operand2,

input cin,

output [31:0] result,

output cout

);

wire [31:0] A;

wire [31:0] B;

wire [32:0] C;

wire [31:0] u\_operand1;

wire [31:0] u\_operand2;

assign A[30:0]=~operand1[30:0]+1;

assign A[31]=operand1[31];

assign u\_operand1=A[31]?A:operand1;

assign B[30:0]=~operand2[30:0]+1;

assign B[31]=operand2[31];

assign u\_operand2=B[31]?B:operand2;

assign C=u\_operand1&u\_operand2<<1;

assign {cout,result}=u\_operand1^u\_operand2+C+cin;

endmodule

**ALU：**

`define ADD 5'b00001

`define ADD\_CIN 5'b00010

`define SUB 5'b00011

`define SUB\_CIN 5'b00100

`define SUB\_N 5'b00101

`define SUB\_N\_CIN 5'b00110

`define A 5'b00111

`define B 5'b01000

`define NOT\_A 5'b01001

`define NOT\_B 5'b01010

`define A\_OR\_B 5'b01011

`define A\_AND\_B 5'b01100

`define A\_SAME\_B 5'b01101

`define A\_NOTS\_B 5'b01110

`define NOT\_A\_AND\_B 5'b01111

`define ZERO 5'b10000

module alu (

input [31:0] A ,

input [31:0] B ,

input Cin ,

input [4 :0] Card,

output [31:0] F ,

output Cout,

output Zero

);

wire [31:0] add\_result;

wire [31:0] add\_cin\_result;

wire [31:0] sub\_result;

wire [31:0] sub\_cin\_result;

wire [31:0] sub\_n\_result;

wire [31:0] sub\_n\_cin\_result;

wire [31:0] a\_result;

wire [31:0] b\_result;

wire [31:0] not\_a\_result;

wire [31:0] not\_b\_result;

wire [31:0] a\_or\_b\_result;

wire [31:0] a\_and\_b\_result;

wire [31:0] a\_same\_b\_result;

wire [31:0] a\_nots\_b\_result;

wire [31:0] not\_a\_and\_b\_result;

wire [31:0] zero\_result;

assign {Cout,add\_result} = A+B;

assign {Cout,add\_cin\_result} = A + B + Cin;

assign sub\_result = A - B;

assign sub\_cin\_result = A - B - Cin;

assign sub\_n\_result = B - A;

assign sub\_n\_cin\_result = B - A - Cin;

assign a\_result = A;

assign b\_result = B;

assign not\_a\_result = ~A;

assign not\_b\_result = ~B;

assign a\_or\_b\_result = A | B;

assign a\_and\_b\_result = A & B;

assign a\_same\_b\_result = A ^~ B;

assign a\_nots\_b\_result = A ^ B;

assign not\_a\_and\_b\_result = ~ (A & B);

assign zero\_result = 32'd0;

assign F = ({32{Card == `ADD}} & add\_result) |

({32{Card == `SUB}} & sub\_result) |

({32{Card== `ADD\_CIN}} & add\_cin\_result) |

({32{Card == `SUB\_CIN}} & sub\_cin\_result) |

({32{Card== `SUB\_N}} & sub\_n\_result) |

({32{Card == `SUB\_N\_CIN}} & sub\_n\_cin\_result) |

({32{Card == `A}} & a\_result) |

({32{Card == `B}} & b\_result) |

({32{Card == `NOT\_A}} & not\_a\_result) |

({32{Card == `NOT\_B}} & not\_b\_result) |

({32{Card == `A\_OR\_B}} & a\_or\_b\_result) |

({32{Card == `A\_AND\_B}} & a\_and\_b\_result) |

({32{Card == `A\_SAME\_B}} & a\_same\_b\_result) |

({32{Card == `A\_NOTS\_B}} & a\_nots\_b\_result) |

({32{Card == `NOT\_A\_AND\_B}} & not\_a\_and\_b\_result) |

({32{Card == `ZERO}} & zero\_result);

assign Zero=({F==0} )| 'd0;

endmodule

加法器测试代码：

`timescale 1ns / 1ps

module tb\_adder;

// 输入信号

reg [31:0] A;

reg [31:0] B;

reg Cin;

// 输出信号

wire [31:0] F;

wire Cout;

// 实例化加法器模块

adder uut (

.A(A),

.B(B),

.Cin(Cin),

.F(F),

.Cout(Cout)

);

// 初始化和仿真激励

initial begin

// 显示信号变化

$monitor("Time: %0d | A: %h | B: %h | Cin: %b | F: %h | Cout: %b", $time, A, B, Cin, F, Cout);

// 初始化输入信号

A = 32'h00000000; B = 32'h00000000; Cin = 1'b0;

#10; // 等待10ns

// 测试1：0 + 0 + 0

A = 32'h00000000;

B = 32'h00000000;

Cin = 1'b0;

#10;

// 测试2：1 + 1 + 0

A = 32'h00000001;

B = 32'h00000001;

Cin = 1'b0;

#10;

// 测试3：FFFFFFFF + 1 + 0

A = 32'hFFFFFFFF;

B = 32'h00000001;

Cin = 1'b0;

#10;

// 测试4：80000000 + 80000000 + 0（溢出测试）

A = 32'h80000000;

B = 32'h80000000;

Cin = 1'b0;

#10;

// 测试5：A + B + 1 进位

A = 32'hFFFFFFFE;

B = 32'h00000001;

Cin = 1'b1;

#10;

// 仿真结束

$finish;

end

endmodule

ALU测试代码：

`timescale 1ns / 1ps

module tb\_alu;

// 输入信号

reg [31:0] A;

reg [31:0] B;

reg Cin;

reg [4:0] Card;

// 输出信号

wire [31:0] F;

wire Cout;

wire Zero;

// 实例化ALU模块

alu uut (

.A(A),

.B(B),

.Cin(Cin),

.Card(Card),

.F(F),

.Cout(Cout),

.Zero(Zero)

);

// 初始化和仿真激励

initial begin

// 显示信号变化

$monitor("Time: %0d | A: %h | B: %h | Cin: %b | Card: %b | F: %h | Cout: %b | Zero: %b", $time, A, B, Cin, Card, F, Cout, Zero);

// 测试1：F = A + B

A = 32'h00000001;

B = 32'h00000002;

Cin = 1'b0;

Card = 5'b00001;

#10;

// 测试2：F = A + B + Cin

A = 32'h00000001;

B = 32'h00000002;

Cin = 1'b1;

Card = 5'b00010;

#10;

// 测试3：F = A - B

A = 32'h00000005;

B = 32'h00000002;

Cin = 1'b0;

Card = 5'b00011;

#10;

// 测试4：F = A - B - Cin

A = 32'h00000005;

B = 32'h00000002;

Cin = 1'b1;

Card = 5'b00100;

#10;

// 测试5：F = B - A

A = 32'h00000002;

B = 32'h00000005;

Cin = 1'b0;

Card = 5'b00101;

#10;

// 测试6：F = B - A - Cin

A = 32'h00000002;

B = 32'h00000005;

Cin = 1'b1;

Card = 5'b00110;

#10;

// 测试7：F = A

A = 32'h12345678;

B = 32'hFFFFFFFF; // B任意，A的值直接赋给F

Cin = 1'b0;

Card = 5'b00111;

#10;

// 测试8：F = B

A = 32'hFFFFFFFF; // A任意，B的值直接赋给F

B = 32'h87654321;

Cin = 1'b0;

Card = 5'b01000;

#10;

// 测试9：F = ~A

A = 32'hAAAAAAAA;

B = 32'h00000000; // B任意，~A赋给F

Cin = 1'b0;

Card = 5'b01001;

#10;

// 测试10：F = ~B

A = 32'h00000000; // A任意，~B赋给F

B = 32'h55555555;

Cin = 1'b0;

Card = 5'b01010;

#10;

// 测试11：F = A + B

A = 32'hFFFFFFFF;

B = 32'h00000001;

Cin = 1'b0;

Card = 5'b01011;

#10;

// 测试12：F = A & B

A = 32'hFF00FF00;

B = 32'h00FF00FF;

Cin = 1'b0;

Card = 5'b01100;

#10;

// 测试13：F = A ⊙ B

A = 32'h0F0F0F0F;

B = 32'hF0F0F0F0;

Cin = 1'b0;

Card = 5'b01101;

#10;

// 测试14：F = A ⊕ B

A = 32'hAAAAAAAA;

B = 32'h55555555;

Cin = 1'b0;

Card = 5'b01110;

#10;

// 测试15：F = ~(A & B)

A = 32'hFF00FF00;

B = 32'h00FF00FF;

Cin = 1'b0;

Card = 5'b01111;

#10;

// 测试16：F = 0

A = 32'hFFFFFFFF; // A、B任意，输出为0

B = 32'h00000000;

Cin = 1'b0;

Card = 5'b10000;

#10;

// 仿真结束

$finish;

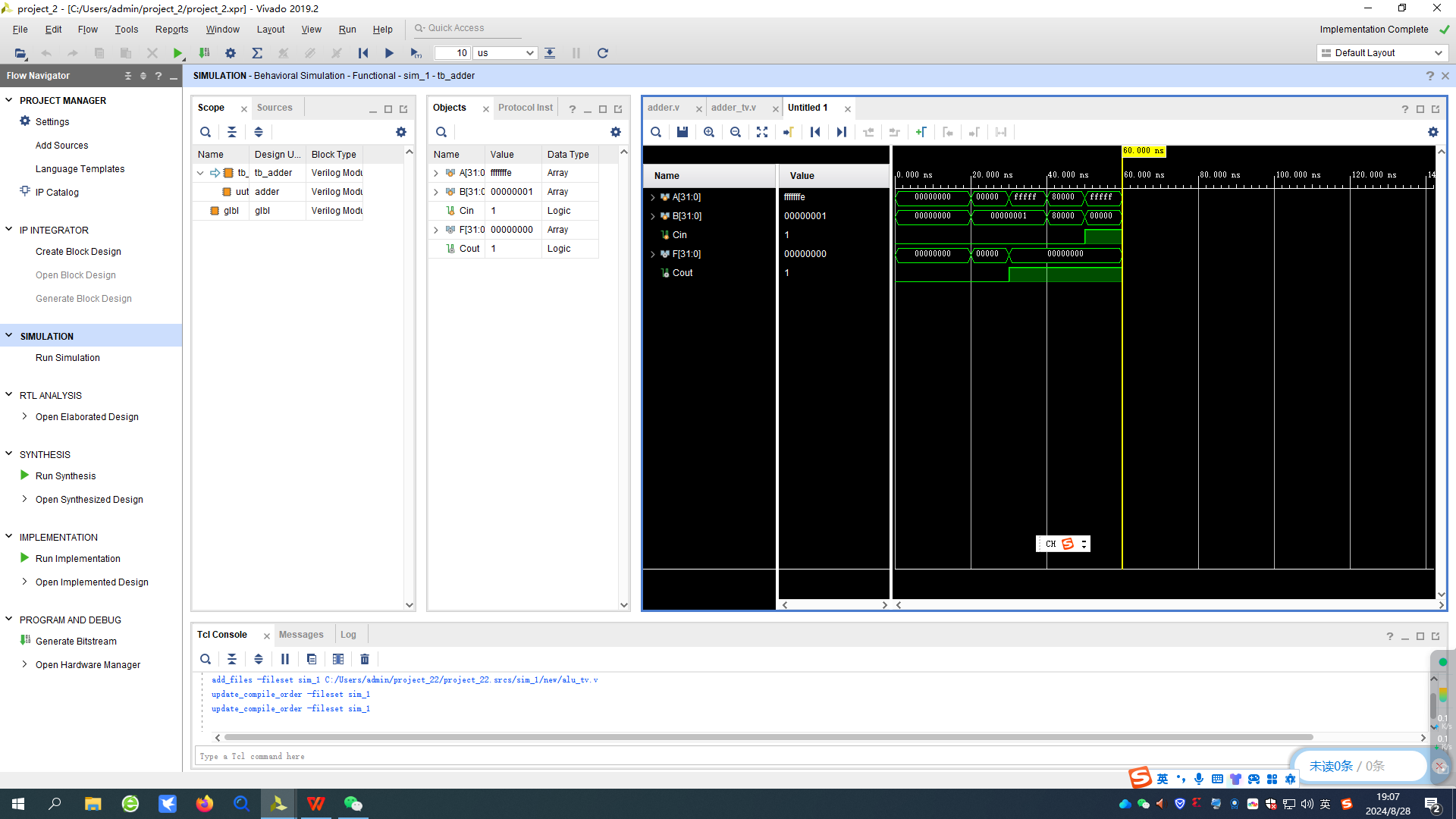
end

endmodule

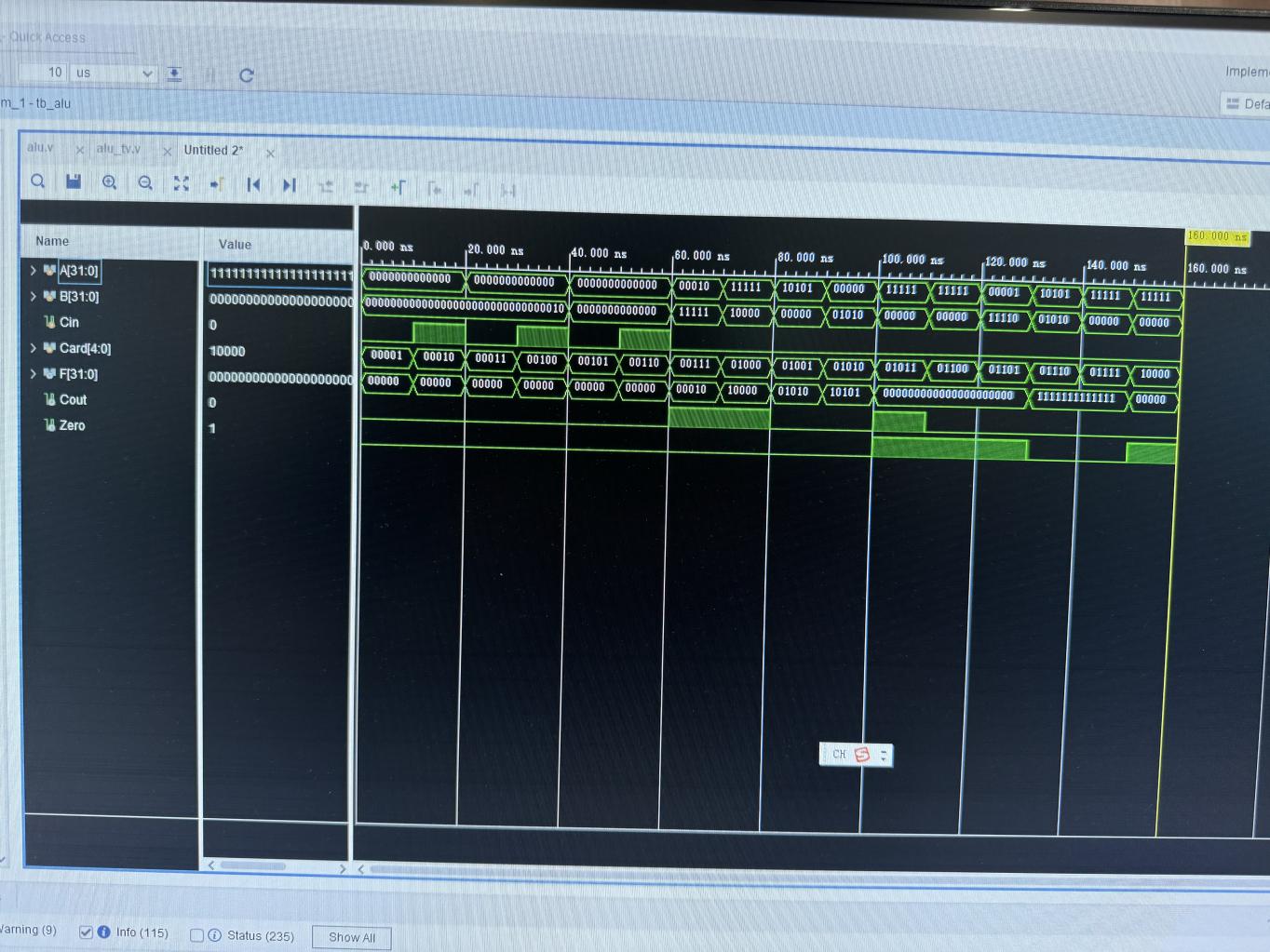
**五、**测试结果及实验分析

测试波形与测试结果：

加法器波形图：



ALU波形图:



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 运算功能 | A(H) | B(H) | Cin | 操作码  （五位） | F(H) | Zero |
| F=A加B | FFFFFFFF | 00000001 | 0 | 00001 | 00000000 | 1 |
| F=A加B加Cin | FFFFFFFF | 00000001 | 1 | 00010 | 00000001 | 0 |
| F=A减B | FFFFFFFF | 00000001 | 0 | 00011 | FFFFFFFE | 0 |
| F=A减B减Cin | FFFFFFFF | 00000001 | 1 | 00100 | FFFFFFFD | 0 |
| F=B减A | 00000001 | FFFFFFFF | 0 | 00101 | FFFFFFFE | 0 |
| F= B减A减Cin | 00000001 | FFFFFFFF | 1 | 00110 | FFFFFFFD | 0 |
| F=A | 12345678 | FFFFFFFF | 0 | 00111 | 12345678 | 0 |
| F=B | FFFFFFFF | 87654321 | 0 | 01000 | 87654321 | 0 |
| F=/A | FFFFFFFF | 00000000 | 0 | 01001 | 00000000 | 0 |
| F=/B | 00000000 | 87654321 | 0 | 01010 | 789ABCDE | 0 |
| F=A+B | FFFFFFFF | 00000001 | 0 | 00001 | 00000000 | 1 |
| F=AB | FF00FF00 | 00FF00FF | 0 | 01100 | 00000000 | 1 |
| F=A⊙B | 0F0F0F0F | F0F0F0F0 | 0 | 01101 | FFFFFFFF | 0 |
| F=A⊕B | AAAAAAAA | 55555555 | 0 | 01110 | FFFFFFFF | 0 |
| F=/(AB) | FF00FF00 | 00FF00FF | 0 | 01111 | FFFFFFFF | 0 |
| F=0 | FFFFFFFF | 00000000 | 0 | 10000 | 00000000 | 1 |

加法器实验：

仿真波形分析：

1. 输入信号:

- A[31:0] = FFFFFFFF（32位全为1的数，即-1的二进制表示）。

- B[31:0] = 00000001（32位数值为1）。

- Cin = 1（进位输入为1）。

2. 输出信号:

- F[31:0]：根据波形图来看，结果是 00000000，这个是正确的，因为 FFFFFFFF + 00000001 + 1 = 00000000，这是一种加法器的溢出情况。

- Cout：进位输出 Cout 为 1，这也是正确的，因为在 FFFFFFFF + 00000001 + 1 的加法中发生了进位。

总结：

- 从波形图可以看出，加法器在处理溢出情况时是正确的。输出的结果 F 为 00000000，进位输出 Cout 为 1，这些都符合预期结果。

ALU加法器：

波形图分析（16种操作）：

1. F = A + B

- 操作码：00001

- 输入：A = FFFFFFFF，B = 00000001，Cin = 0

- 结果：F = 00000000，Cout = 1，Zero = 1

- 分析：溢出，Cout 为 1。

2. F = A + B + Cin

- 操作码：00010

- 输入：A = FFFFFFFF，B = 00000001，Cin = 1

- 结果：F = 00000001，Cout = 1，Zero = 0

- 分析：Cin 进位影响结果，Cout 为 1。

3. F = A - B

- 操作码：00011

- 输入：A = FFFFFFFF，B = 00000001，Cin = 0

- 结果：F = FFFFFFFE，Cout = 0，Zero = 0

- 分析：简单减法，无进位。

4. F = A - B - Cin

- 操作码：00100

- 输入：A = FFFFFFFF，B = 00000001，Cin = 1

- 结果：F = FFFFFFFD，Cout = 0，Zero = 0

- 分析：考虑 Cin 的减法操作。

5. F = B - A

- 操作码：00101

- 输入：A = 00000001，B = FFFFFFFF，Cin = 0

- 结果：F = FFFFFFFE，Cout = 0，Zero = 0

- 分析：B - A，结果正常。

6. F = B - A - Cin

- 操作码：00110

- 输入：A = 00000001，B = FFFFFFFF，Cin = 1

- 结果：F = FFFFFFFD，Cout = 0，Zero = 0

- 分析：考虑 Cin 的减法操作。

7. F = A

- 操作码：00111

- 输入：A = 12345678，B = FFFFFFFF，Cin = 0

- 结果：F = 12345678，Cout = 0，Zero = 0

- 分析：F 直接等于 A。

8. F = B

- 操作码：01000

- 输入：A = FFFFFFFF，B = 87654321，Cin = 0

- 结果：F = 87654321，Cout = 0，Zero = 0

- 分析：F 直接等于 B。

9. F = ~A

- 操作码：01001

- 输入：A = FFFFFFFF，B不参与

- 结果：F = 00000000，Cout = 0，Zero = 1

- 分析：~A，结果为0，Zero 为1。

10. F = ~B

- 操作码：01010

- 输入：B = 87654321，A不参与

- 结果：F = 789ABCDE，Cout = 0，Zero = 0

- 分析：~B，结果正确。

11. F = A | B

- 操作码：01100

- 输入：A = F0F0F0F0，B = 0F0F0F0F

- 结果：F = FFFFFFFF，Cout = 0，Zero = 0

- 分析：按位或运算，结果为 FFFFFFFF。

12. F = A & B

- 操作码：01101

- 输入：A = FF00FF00，B = 00FF00FF

- 结果：F = 00000000，Cout = 0，Zero = 1

- 分析：按位与，结果为0。

13. F = A ⊙ B\*\* （XNOR）

- 操作码：01110

- 输入：A = 0F0F0F0F，B = F0F0F0F0

- 结果：F = FFFFFFFF，Cout = 0，Zero = 0

- 分析：按位同或，结果为1。

14. F = A ⊕ B （XOR）

- 操作码：01111

- 输入：A = AAAAAAAA，B = 55555555

- 结果：F = FFFFFFFF，Cout = 0，Zero = 0

- 分析：按位异或，结果为1。

15. F = ~(A & B)

- 操作码：10000

- 输入：A = FF00FF00，B = 00FF00FF

- 结果：F = FFFFFFFF，Cout = 0，Zero = 0

- 分析：按位与结果取反，结果为1。

16. F = 0

- 操作码：10001

- 输入：A 和 B 无关

- 结果：F = 00000000，Cout = 0，Zero = 1

- 分析：固定输出0。

六、实验总结

加法器实验：

通过本次实验，我对二进制加法有了更深入的理解，尤其是在硬件描述语言（HDL）中如何实现加法器的过程更加清晰。实验中使用仿真工具，使我能够直观地观察各个信号的变化，确保设计的正确性。此外，我学习到了Verilog中 wire 和 reg 的不同用途，这为后续更加复杂的电路设计奠定了坚实的基础。

ALU实验：

此次实验使我对算术逻辑单元（ALU）的功能和重要性有了全面的认识，尤其是其在计算机体系结构中的关键作用。在设计和仿真的过程中，我不仅掌握了如何实现ALU，还学习到如何通过硬件描述语言灵活地定义和实现复杂的逻辑功能。同时，实验中模块化设计的应用，如将加法器模块集成到ALU中，展示了代码复用与可维护性的优势。这种方法在实际工程设计中非常有价值。

**实验三 存储器与寄存器堆**

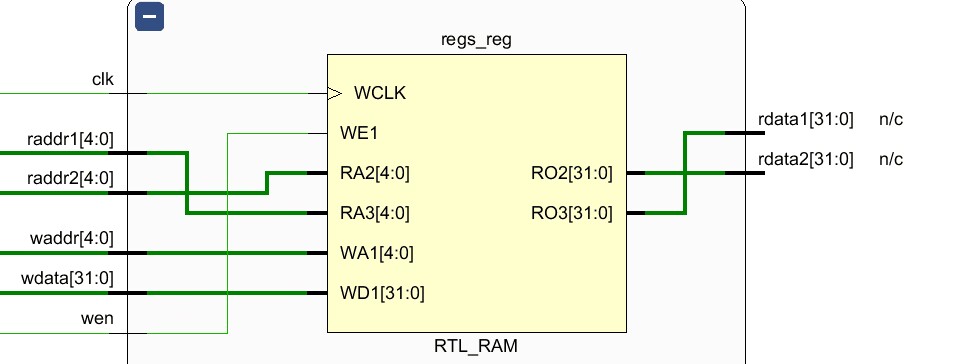
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 预习成绩 | 实验成绩 | 报告成绩 | 总成绩 | 教师签字 |
|  |  |  |  |  |

一、实验目的

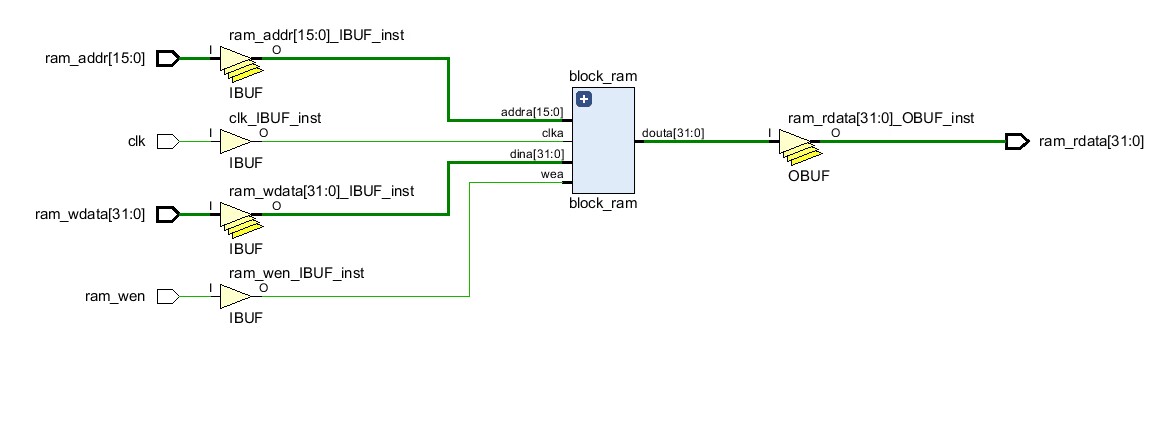
1. 熟悉并掌握 MIPS 计算机中寄存器堆的原理和设计方法
2. 理解源操作数/目的操作数的概念
3. 理解 RAM 读取、写入数据的过程
4. 掌握调用 xilinx 库 IP 实例化 RAM 的设计方法

二、实验预习

1．画出寄存器堆的结构框图，标出输入输出端口。



2．画出存储器的结构框图，标出输入输出端口（确定存储器宽度、深度和写使能位数）



存储器宽度：32bit 深度:65536 写使能位数:1位

三、实验环境（实验设备、开发环境）（预习）

**Personal Computer：**

Processor：13th Gen Intel(R) Core(TM) 2.20 GHz

Installed RAM：16.0 GB (15.7 GB usable)

System type： 64-bit operating system, x64-based processor

**Xilinx FPGA**：xc7a200tfbg676-2

**Vivado Design Suite: version 2019.2**

Vivado集成了硬件描述语言（HDL）的编辑、综合、仿真和调试工具，支持Verilog和VHDL。

功能模块:

Vivado IDE: 用于设计、综合、实现和验证FPGA设计。

Vivado Simulator: 内置仿真器，用于功能仿真和时序仿真。

Waveform Viewer: 用于查看仿真生成的波形文件，分析信号的时序关系。

四、实验设计

写出带有详细注释的Verilog实现代码：

module reg\_file (

input wire clk,

input wire we, // 写使能

input wire [4:0] raddr1, // 读地址1

input wire [4:0] raddr2, // 读地址2

input wire [4:0] waddr, // 写地址

input wire [31:0] wdata, // 写数据

output wire [31:0] rdata1, // 读数据1

output wire [31:0] rdata2 // 读数据2

);

reg [31:0] registers [31:0]; // 32个32位寄存器

// 异步读端口

assign rdata1 = registers[raddr1];

assign rdata2 = registers[raddr2];

// 同步写端口

always @(posedge clk) begin

if (we) begin

registers[waddr] <= wdata;

end

end

endmodule

module ram\_top (

input wire clk,

input wire ram\_wen, // 写使能

input wire [15:0] ram\_addr, // 地址

input wire [31:0] ram\_wdata, // 写数据

output wire [31:0] ram\_rdata // 读数据

);

// 实例化blk\_mem\_gen\_0 IP核

blk\_mem\_gen\_0 ram\_inst (

.clka(clk), // 时钟信号

.wea(ram\_wen), // 写使能信号

.addra(ram\_addr), // 地址信号

.dina(ram\_wdata), // 写入数据

.douta(ram\_rdata) // 读出数据

);

endmodule

module system\_top (

input wire clk,

input wire [4:0] reg\_raddr1, // 寄存器堆读地址1

input wire [4:0] reg\_raddr2, // 寄存器堆读地址2

input wire [4:0] reg\_waddr, // 寄存器堆写地址

input wire reg\_we, // 寄存器堆写使能

input wire [15:0] ram\_addr, // RAM地址

input wire ram\_wen, // RAM写使能

input wire [31:0] alu\_op1, // ALU操作数1

input wire [31:0] alu\_op2, // ALU操作数2

input wire [2:0] alu\_ctrl, // ALU控制信号

output wire [31:0] ram\_rdata, // 从RAM读出的数据

output wire [31:0] alu\_result // ALU运算结果

);

wire [31:0] reg\_rdata1; // 寄存器堆读数据1

wire [31:0] reg\_rdata2; // 寄存器堆读数据2

wire [31:0] ram\_wdata; // 写入RAM的数据

// 实例化寄存器堆

reg\_file reg\_inst (

.clk(clk),

.we(reg\_we),

.raddr1(reg\_raddr1),

.rdata1(reg\_rdata1),

.raddr2(reg\_raddr2),

.rdata2(reg\_rdata2),

.waddr(reg\_waddr),

.wdata(ram\_rdata) // 从RAM读出的数据写入寄存器堆

);

// 实例化RAM

ram\_top ram\_inst (

.clk(clk),

.ram\_wen(ram\_wen),

.ram\_addr(ram\_addr),

.ram\_wdata(alu\_result), // 将ALU结果写入RAM

.ram\_rdata(ram\_rdata) // 从RAM读出的数据

);

// 实例化ALU

alu alu\_inst (

.a(alu\_op1), // 操作数1

.b(alu\_op2), // 操作数2

.alu\_ctrl(alu\_ctrl), // 控制信号

.result(alu\_result) // ALU运算结果

);

endmodule

module testbench;

// 信号声明

reg clk;

reg [4:0] reg\_raddr1;

reg [4:0] reg\_raddr2;

reg [4:0] reg\_waddr;

reg reg\_we;

reg [15:0] ram\_addr;

reg ram\_wen;

reg [31:0] alu\_op1;

reg [31:0] alu\_op2;

reg [2:0] alu\_ctrl;

wire [31:0] ram\_rdata;

wire [31:0] alu\_result;

// 实例化顶层模块

system\_top uut (

.clk(clk),

.reg\_raddr1(reg\_raddr1),

.reg\_raddr2(reg\_raddr2),

.reg\_waddr(reg\_waddr),

.reg\_we(reg\_we),

.ram\_addr(ram\_addr),

.ram\_wen(ram\_wen),

.alu\_op1(alu\_op1),

.alu\_op2(alu\_op2),

.alu\_ctrl(alu\_ctrl),

.ram\_rdata(ram\_rdata),

.alu\_result(alu\_result)

);

// 时钟生成

always #5 clk = ~clk;

initial begin

// 初始化信号

clk = 0;

reg\_raddr1 = 5'd0;

reg\_raddr2 = 5'd1;

reg\_waddr = 5'd2;

reg\_we = 1'b0;

ram\_addr = 16'd0;

ram\_wen = 1'b0;

alu\_op1 = 32'd10;

alu\_op2 = 32'd20;

alu\_ctrl = 3'b000; // ALU加法操作

// 仿真流程

// 1. 将数据写入RAM

#10 ram\_wen = 1'b1;

ram\_addr = 16'd1;

alu\_op1 = 32'd15;

alu\_op2 = 32'd25;

alu\_ctrl = 3'b000; // ALU执行加法操作

#10 ram\_wen = 1'b0;

// 2. 从RAM读取数据并写入寄存器堆

#10 reg\_waddr = 5'd2;

reg\_we = 1'b1;

ram\_addr = 16'd1;

#10 reg\_we = 1'b0;

// 3. 执行ALU操作并将结果写入RAM

#10 alu\_op1 = ram\_rdata; // 从RAM读取数据

alu\_op2 = 32'd5;

alu\_ctrl = 3'b001; // ALU执行减法操作

ram\_addr = 16'd2;

ram\_wen = 1'b1;

#10 ram\_wen = 1'b0;

// 4. 读取RAM中的结果并检查

#10 ram\_addr = 16'd2;

#10; // 等待一个时钟周期

// 仿真结束

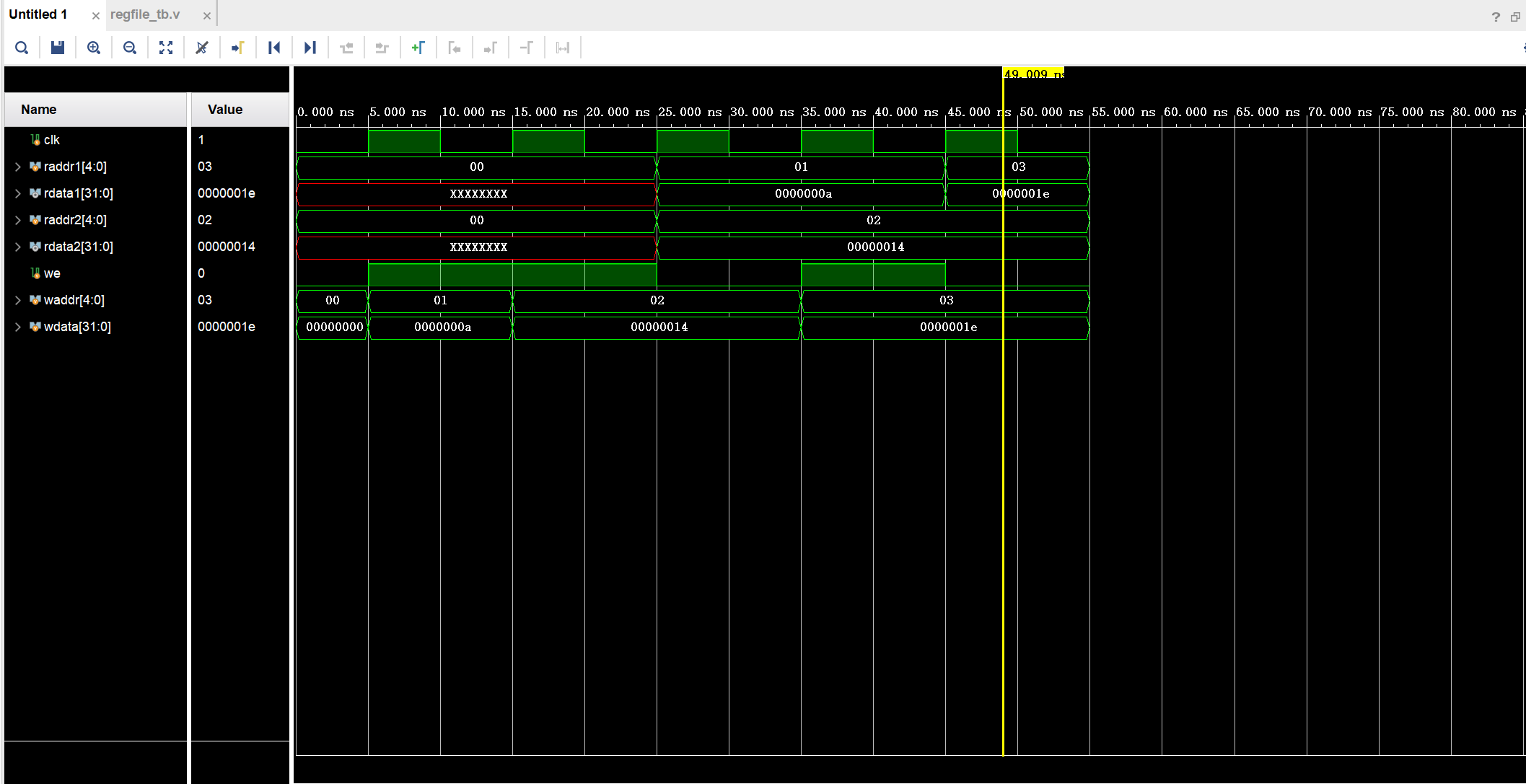
#100 $stop;

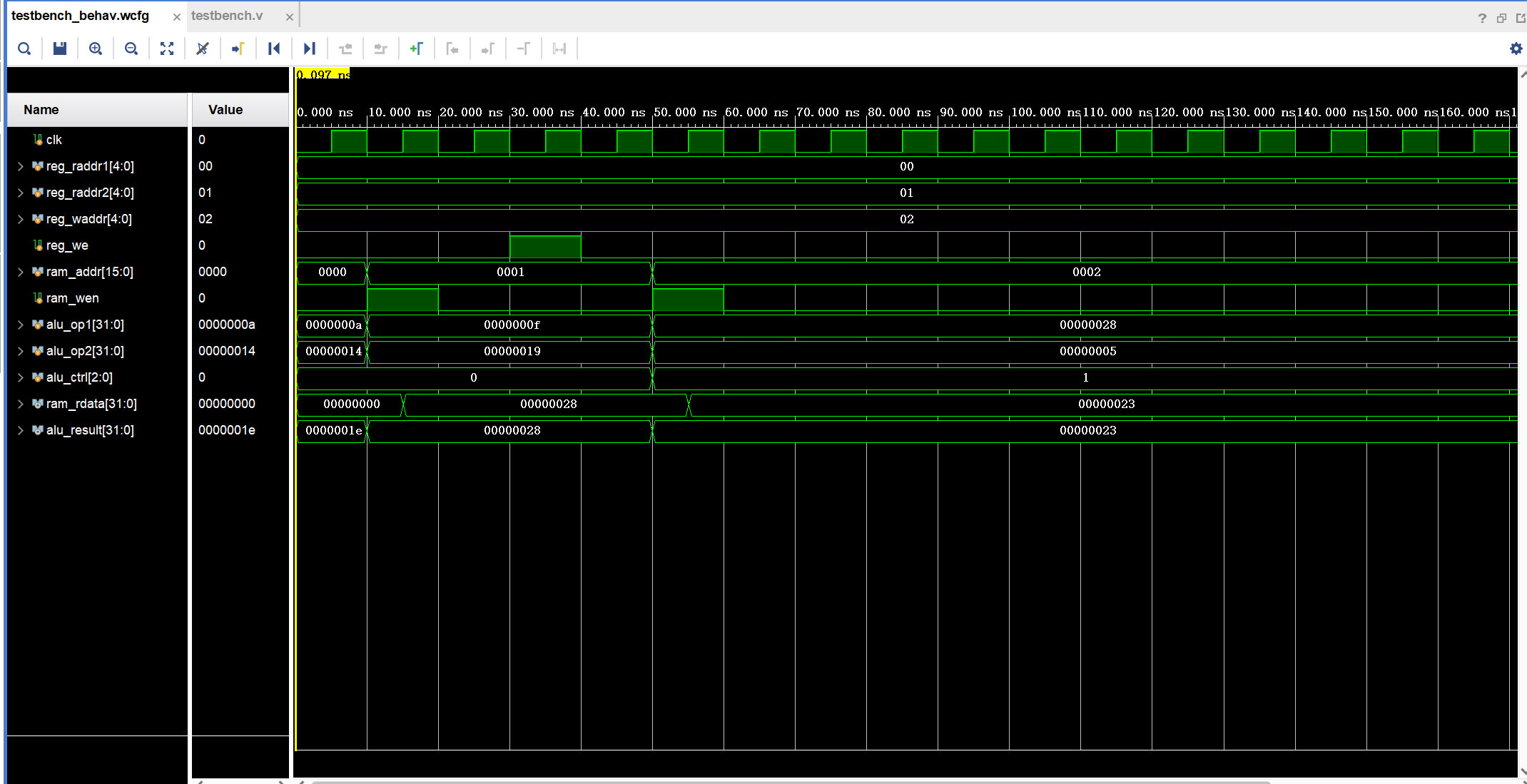
end

endmodule

**五、**测试结果及实验分析

测试波形与测试结果：





实验结果分析：

1. 寄存器堆的操作

- 写操作：

- 在第一个写操作周期中，写使能信号（we）为高电平，写地址（waddr）设置为01，写数据（wdata）为a5a5a5a5。数据成功写入地址为01的寄存器。

- 仿真波形显示，写操作完成后，读取地址01时，读取数据（rdata1）显示为a5a5a5a5，证明写操作正确无误。

- 读操作：

- 当读取地址（raddr1）为01时，rdata1返回的值是a5a5a5a5，验证了之前的写操作是成功的。

- 当raddr2信号为00时，读取的数据为x。这可能是因为地址00还没有进行写入操作，因此返回未定义的值（通常用x表示）。

2. RAM操作

- 写操作：

- 在第一个写操作周期中，RAM写使能信号（ram\_wen）为高电平，RAM地址（ram\_addr）为0001，写数据（ram\_wdata）为deadbeef。该数据成功写入地址为0001的RAM单元。

- 读操作：

- 当ram\_addr设置为0001时，读取数据（ram\_rdata）返回的值为deadbeef，表明数据已成功从RAM中读取，验证了写操作的正确性。

3. IP核实例化

3.1 时钟信号（clk）

时钟信号按预期每10 ns切换一次，确保了整个系统的同步操作。所有寄存器和RAM的操作均在时钟的上升沿触发，保证了系统的稳定性和数据处理的正确性。

3.2 寄存器堆操作

- 在初始阶段，读地址1（reg\_raddr1）和读地址2（reg\_raddr2）分别被设置为0和1，表示从寄存器堆中读取寄存器0和寄存器1的内容。

- 写地址（reg\_waddr）信号在20 ns时设置为2，表示将数据写入寄存器2。写使能信号（reg\_we）也在此时被置为高电平，成功触发数据写入寄存器堆。

3.3 RAM操作

- RAM地址（ram\_addr）信号分别在不同时间点设置为2和1，表示访问不同的RAM地址。

- 在10 ns时，RAM写使能信号（ram\_wen）置高，ALU的运算结果（40，十六进制为00000028）写入RAM地址1。

- 在50 ns时，通过仿真波形可以看到成功从RAM地址2读取到ALU计算结果35（十六进制为00000023），验证了RAM的写入和读取操作是正确的。

3.4 ALU操作

- 在10 ns时，ALU接收到两个操作数alu\_op1 = 15（十六进制为0000000F）和alu\_op2 = 25（十六进制为00000019），并执行加法操作。

- 在30 ns时，ALU输出的结果为40（十六进制为00000028），这是加法操作的正确结果。

- 系统接着执行减法操作，将40减去5，得到35（十六进制为00000023），该结果正确写入RAM，并在后续读取中验证无误。

3.5 数据流动分析

- 数据从寄存器堆流动到ALU，经过ALU计算后，再写入RAM。RAM数据的写入和读取过程在仿真波形图中清晰展示，所有数据操作均按照预期执行，表明数据在不同模块之间的流动是正确的。

六、实验总结

实验总结 1: 使用手动编写的 regfile 模块

通过手动编写寄存器堆模块，我对寄存器堆的内部工作原理有了更深刻的理解。然而，手动编写寄存器堆模块相对复杂，尤其是在需要优化性能或资源利用率时，这种方式显得费时且容易出错。未来设计中，考虑到设计效率和可靠性，使用成熟的 IP 核可能是更合适的选择。

实验总结 2: 使用 IP 核

通过使用 IP 核，我大大降低了设计的复杂度，并减少了可能出现的错误。IP 核经过优化，能够在资源利用和时序性能上展现出更好的表现。因此，在今后的项目中，尤其是在需要快速迭代和优化的场景中，优先考虑使用 IP 核将是一种更高效且可靠的设计策略。**实验四 指令译码器设计**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 预习成绩 | 实验成绩 | 报告成绩 | 总成绩 | 教师签字 |
|  |  |  |  |  |

一、实验目的

1．掌握 Vivado 集成开发环境

2．掌握Verilog语言

3．掌握FPGA编程方法及硬件调试手段

4．理解指令译码的思想

5. 掌握实验台使用

二、实验环境（实验设备、开发环境）

**Personal Computer：**

Processor：13th Gen Intel(R) Core(TM) 2.20 GHz

Installed RAM：16.0 GB (15.7 GB usable)

System type： 64-bit operating system, x64-based processor

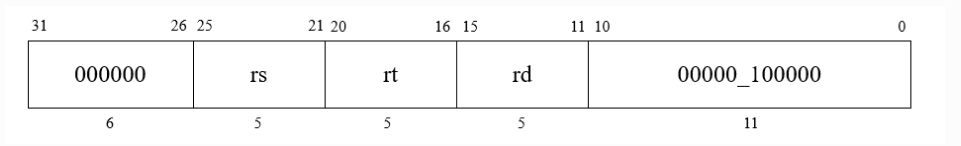
**Xilinx FPGA**：xc7a200tfbg676-2

**Vivado Design Suite: version 2019.2**

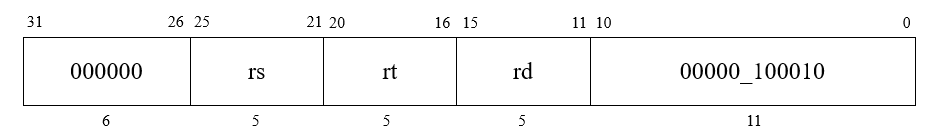
三、实验预习（如空白不够，可自行加页）

1. 按8种指令运算操作的意义，完成这8种指令的编码

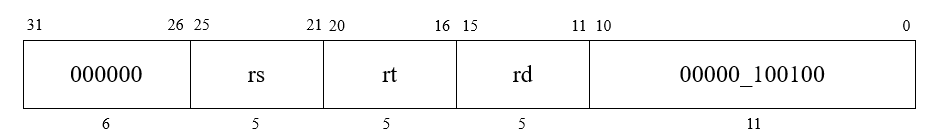
加法指令：



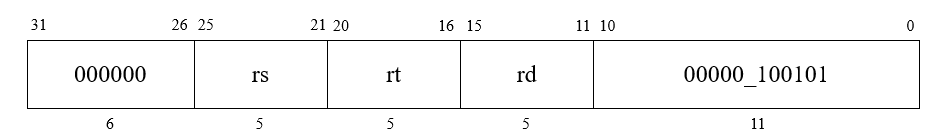
减法指令：



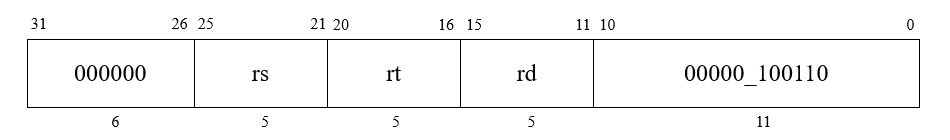
与运算指令：



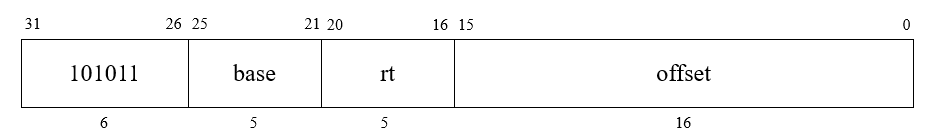
或运算指令：



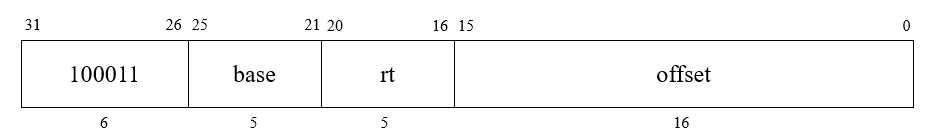
异或指令：



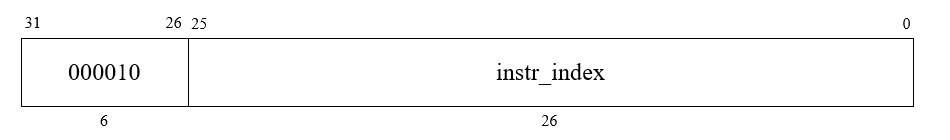
存数指令：



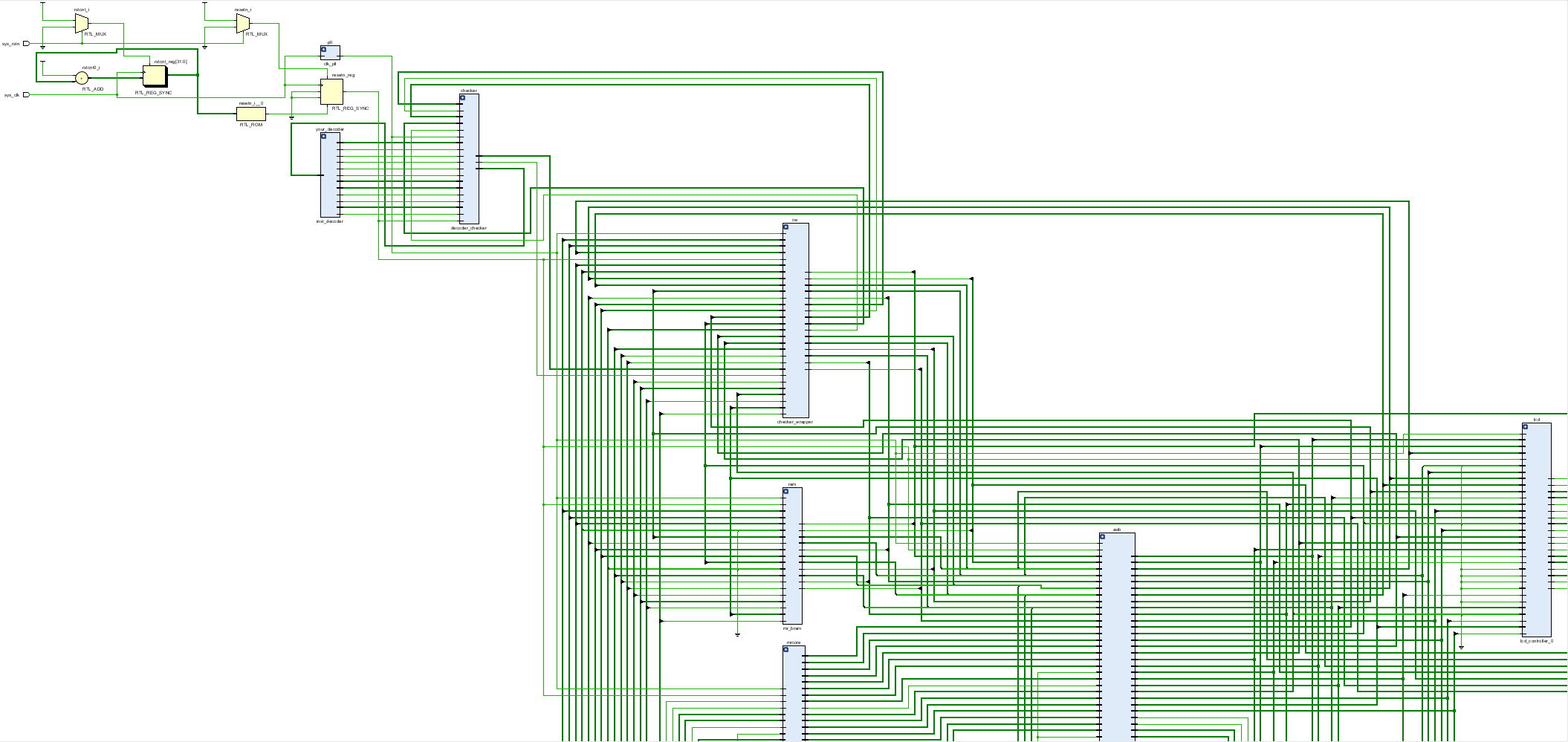
取数指令：

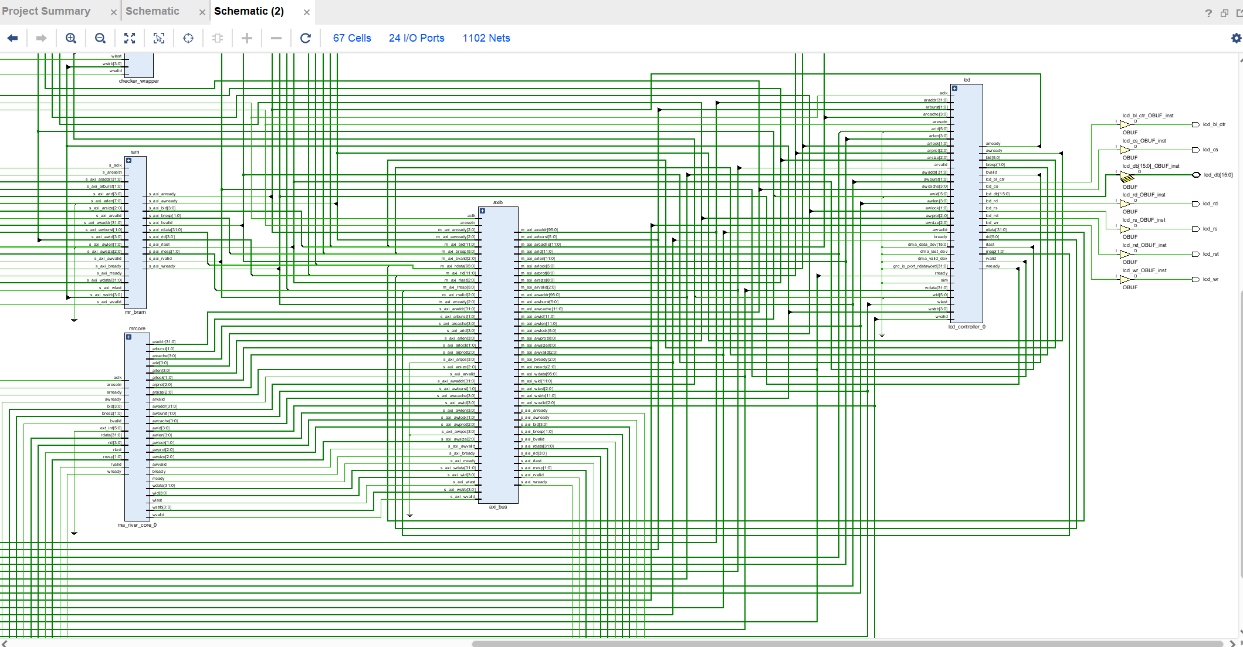


无条件转移指令：



2. 画出指令译码器的结构框图





四、实验设计

写出带有详细注释的Verilog实现代码：

module inst\_decoder(

input [31:0] inst, // 输入指令

output wen, // 是否写回寄存器

output [4:0] waddr, // 写回的寄存器地址

output rden1, // 是否读寄存器1

output [4:0] raddr1,// 读的寄存器1地址

output rden2, // 是否读寄存器2

output [4:0] raddr2,// 读的寄存器2地址

output alu\_en, // ALU是否启用

output [4:0] alu\_card, // ALU操作码

output mem\_rd, // 是否读存储器

output mem\_wr, // 是否写存储器

output jmp, // 是否跳转

output invalid // 指令是否无效

);

// 检查是否写回寄存器

assign wen = (inst[31:26] == 6'b000000 && (inst[5:0] == 6'b100000 || inst[5:0] == 6'b100010 ||

inst[5:0] == 6'b100100 || inst[5:0] == 6'b100101 || inst[5:0] == 6'b100110 ||

inst[5:0] == 6'b000000)) || // SLL

(inst[31:26] == 6'b100011); // LW指令也需要写回

// 写回寄存器地址 (R型和LW)

assign waddr = (inst[31:26] == 6'b000000) ? inst[15:11] :

(inst[31:26] == 6'b100011) ? inst[20:16] : 5'b00000;

// 读寄存器1

assign rden1 = (inst[31:26] == 6'b000000 || inst[31:26] == 6'b100011 || inst[31:26] == 6'b101011);

assign raddr1 = inst[25:21];

// 读寄存器2

assign rden2 = (inst[31:26] == 6'b000000 || inst[31:26] == 6'b101011);

assign raddr2 = inst[20:16];

// ALU启用

assign alu\_en = (inst[31:26] == 6'b000000); // R型指令

// ALU操作码 (R型指令：ADD, SUB, AND, OR, XOR, SLL)

assign alu\_card = (inst[5:0] == 6'b100000) ? 5'b00001 : // ADD

(inst[5:0] == 6'b100010) ? 5'b00011 : // SUB

(inst[5:0] == 6'b100100) ? 5'b01100 : // AND

(inst[5:0] == 6'b100101) ? 5'b01011 : // OR

(inst[5:0] == 6'b100110) ? 5'b01110 : // XOR

(inst[5:0] == 6'b000000) ? 5'b10001 : // SLL

5'b00000;

// 读存储器 (LW指令)

assign mem\_rd = (inst[31:26] == 6'b100011);

// 写存储器 (SW指令)

assign mem\_wr = (inst[31:26] == 6'b101011);

// 跳转指令 (J指令)

assign jmp = (inst[31:26] == 6'b000010);

// 指令无效，除了R型、LW、SW、J型指令之外都无效

assign invalid = !((inst[31:26] == 6'b000000 && (inst[5:0] == 6'b100000 || inst[5:0] == 6'b100010 ||

inst[5:0] == 6'b100100 || inst[5:0] == 6'b100101 || inst[5:0] == 6'b100110)) ||

(inst[31:26] == 6'b100011) || // LW

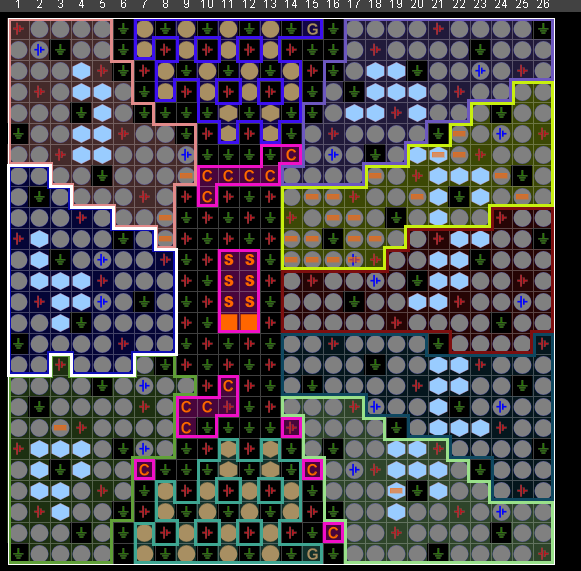
(inst[31:26] == 6'b101011) || // SW

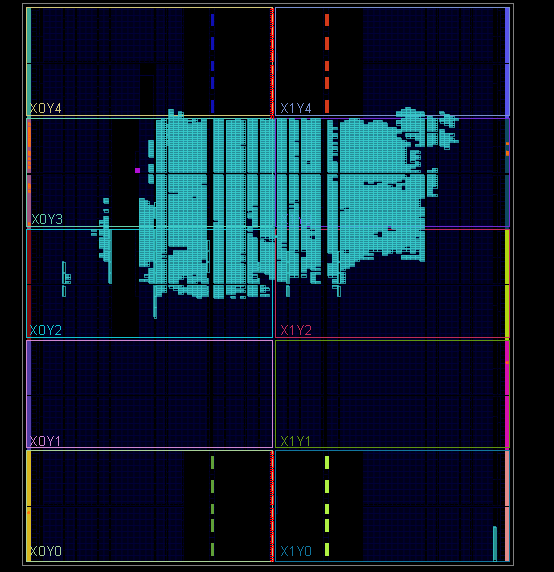
(inst[31:26] == 6'b000010)); // J型

endmodule

五、测试结果及实验分析

1、仿真测试波形与测试结果





**图形用户界面, 应用程序

描述已自动生成**

2. 下载到实验台的测试结果

****

实验结果分析：

波形分析：

1. 时钟信号 (clk)：时钟信号稳定，波形显示正常的高低电平转换，仿真中时钟周期固定，保证了系统的同步性。

2. 复位信号 (resetn)：复位信号最初为低电平（有效），随后切换为高电平，释放系统进入正常操作。

3. 指令信号 (d\_inst[31:0])：每个指令周期输入不同的指令，例如 00430820 和 00641202 等，这些指令被正确解码并执行。每个时钟周期有不同的指令送入指令译码器。

4. 寄存器写回使能 (d\_wen)：在执行 LW 和 R 型指令时，写回使能信号（d\_wen）正确拉高，表示此时将指令结果写回寄存器。

5. 寄存器地址 (d\_waddr)：寄存器写回地址正确更新，与当前执行的指令匹配。例如，在执行 LW 和 R 型指令时，写回的目标寄存器地址正确显示。

6. 读取寄存器地址 (d\_raddr1, d\_raddr2)：源寄存器地址显示正确，说明译码器能够从指定寄存器中正确读取操作数。

7. ALU使能 (d\_alu\_en)：当执行 R 型指令时，ALU 使能信号拉高，表示 ALU 正在执行运算。

8. ALU操作码 (d\_alu\_card)：ALU 操作码根据不同指令切换，如加法、减法、与、或等操作被正确执行。

9. 存储器读/写 (d\_mem\_rd / d\_mem\_wr)：在执行 LW（读）和 SW（写）指令时，存储器读/写信号被正确激活，表明内存操作指令被成功解码。

10. 跳转信号 (d\_jmp)：在跳转指令（如 J 型指令）执行时，跳转信号拉高，表明跳转指令得到了正确处理。

关键指令分析：

- 指令 00430820：这是一个 R 型指令，ALU 启用，并将结果写回寄存器（d\_wen = 1），ALU 操作码和写回地址均正确。

- 指令 8cc40018：这是内存操作指令（如 LW 或 SW），存储器读/写信号正确激活，操作成功。

- 指令 0002f824：这是跳转或分支指令，跳转信号（d\_jmp）正确拉高，跳转操作成功执行。

六、实验总结：

指令译码：本次实验成功实现了对 MIPS 指令的译码功能，包括加法、减法、与、或、异或、左移、内存读写、跳转等操作。通过 Verilog 设计，能够正确解析指令中的操作码（opcode）和功能码（func），并生成对应的控制信号。

跳转与控制流：在跳转指令（如 J 指令）执行时，跳转信号能够正确触发，证明译码器能够准确解析控制流指令，顺利执行跳转操作。

寄存器读写：寄存器文件的读写控制信号在不同类型的指令中均表现正常，尤其是在 R 型和 I 型指令中，寄存器地址和写回使能信号均正确设置，确保操作结果被正确写入寄存器。

收获与体会：

1. 硬件描述语言的应用：通过本次实验，我对 Verilog 语言有了更深的理解，尤其是如何将指令逻辑转换为硬件电路的实现方法。

2. FPGA 调试技巧：在使用 Vivado 的仿真工具和波形查看器时，我学会了如何定位设计中的问题并进行有效的调试，这对后续的硬件设计具有重要意义。

3. 指令集架构的理解：设计指令译码器的过程中，我加深了对 MIPS 指令集架构的理解，强化了我对 CPU 工作原理和指令与硬件映射关系的认识。

**实验五 给定指令系统的处理器设计**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 预习成绩 | 实验成绩 | 报告成绩 | 总成绩 | 教师签字 |
|  |  |  |  |  |

一、实验目的

1．掌握 Vivado 集成开发环境

2．掌握Verilog语言

3．掌握FPGA编程方法及硬件调试手段

4．深刻理解处理器结构和计算机系统的整体工作原理

二、实验环境（实验设备、开发环境）

**Personal Computer：**

Processor：13th Gen Intel(R) Core(TM) 2.20 GHz

Installed RAM：16.0 GB (15.7 GB usable)

System type： 64-bit operating system, x64-based processor

**Xilinx FPGA**：xc7a200tfbg676-2

**Vivado Design Suite: version 2019.2**

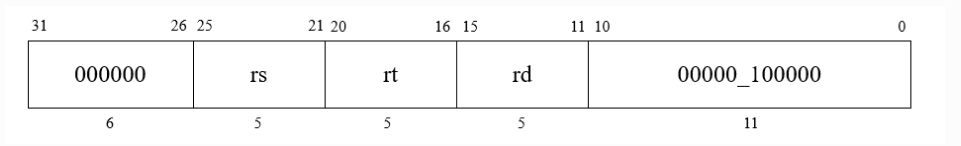
三、设计思想（实验预习）（如空白不够，可自行加页）

1．CPU接口信号定义，填写下表

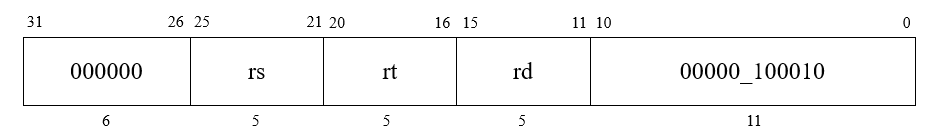
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 信号名 | 位数 | 方向 | 来源/去向 | 意义 |
| clk | 1 | 输入 | 外部时钟源 | 提供时钟信号，控制CPU的同步操作 |
| resetn | 1 | 输入 | 外部复位信号 | 低电平复位信号，用于复位CPU内部寄存器 |
| inst sram en | 1 | 输出 | 指令存储器 | 指令存储器读使能信号 |
| inst sram addr | 32 | 输出 | 指令存储器 | 要读取指令的地址 |
| inst sram rdata | 32 | 输入 | 指令存储器 | 从指令存储器读取的指令数据 |
| Data sram en | 1 | 输出 | 数据存储器 | 数据存储器端口读/写使能信号 |
| Data sram wen | 4 | 输入 | 数据存储器 | 数据存储器写使能信号，指示哪些字节可以写入 |
| data\_sram\_addr | 32 | 输出 | 数据存储器 | 数据存储器读/写地址 |
| data\_sram\_wdata | 32 | 输出 | 数据存储器 | 写入数据存储器的数据 |
| data\_sram\_rdata | 32 | 输入 | 数据存储器 | 从数据存储器读取的数据 |
| debug\_wb\_pc | 32 | 输出 | 调试模块 | 当前正在执行指令的程序计数器（PC） |
| debug\_wb\_rf\_wen | 1 | 输出 | 调试模块 | 通用寄存器组的写使能信号 |
| debug\_wb\_rf\_wnum | 5 | 输出 | 调试模块 | 当前写回的寄存器编号 |
| debug\_wb\_rf\_wdata | 32 | 输出 | 调试模块 | 写回寄存器的数据 |

1. 给出处理器的设计方案，设计方案要求包括：
2. 指令格式设计

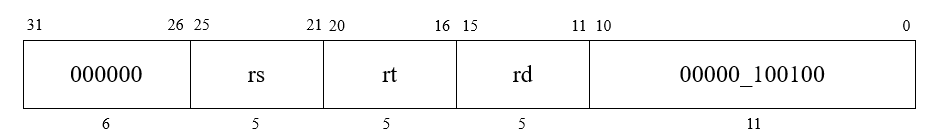
加法指令：



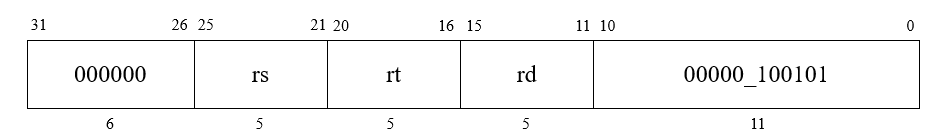
减法指令：



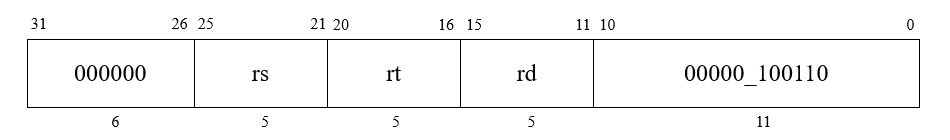
与运算指令：



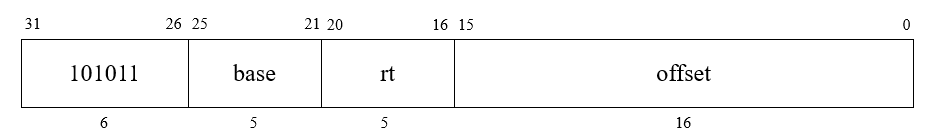
或运算指令：



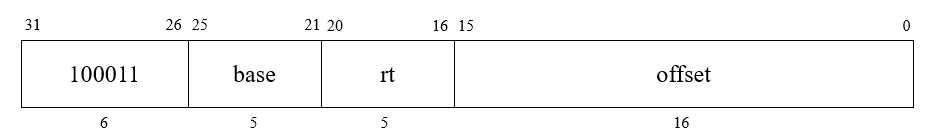
异或指令：



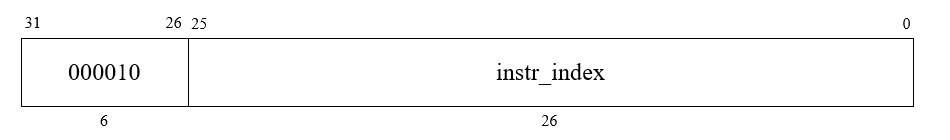
存数指令：



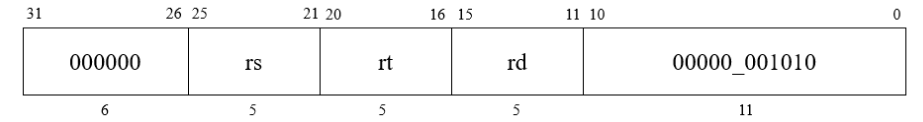
取数指令：



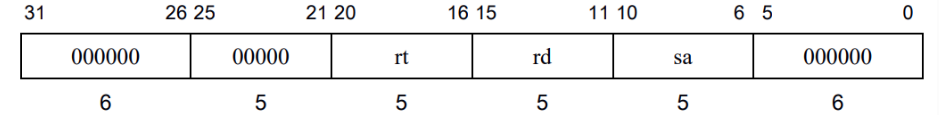
无条件转移指令：



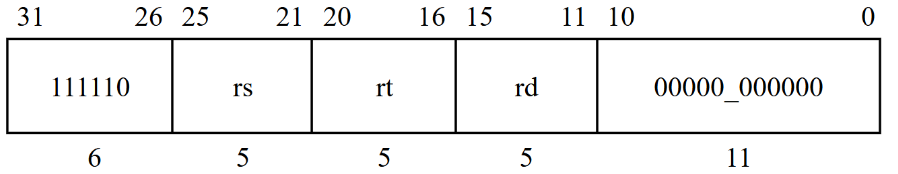
条件移动指令：



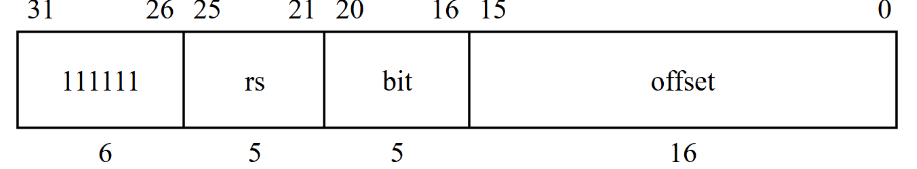
移位指令：



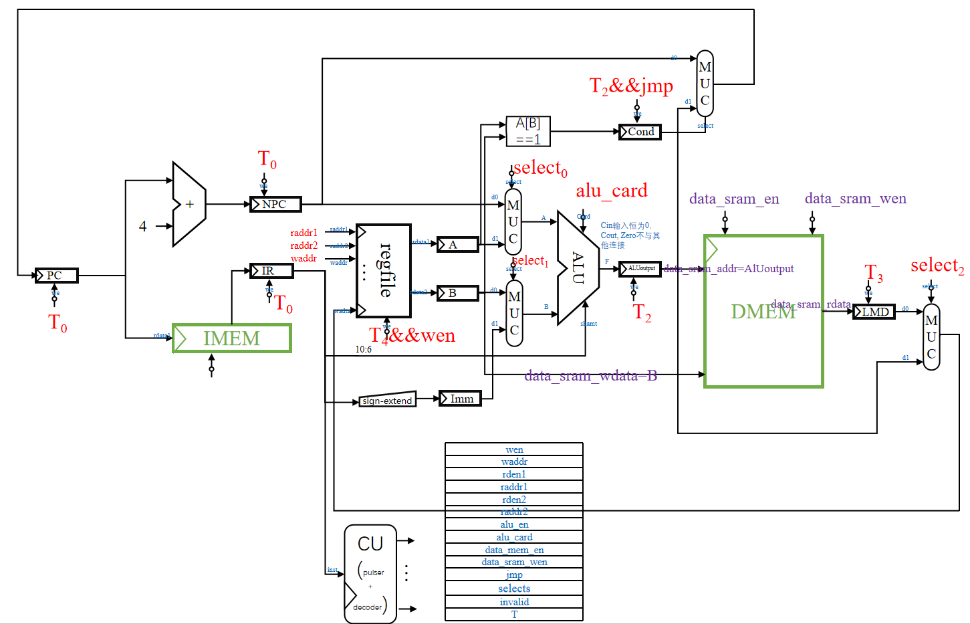
比较指令：



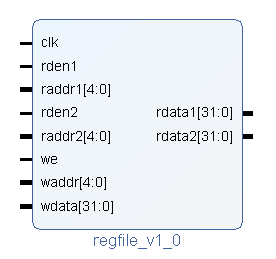
条件转移（位测试跳转）指令：



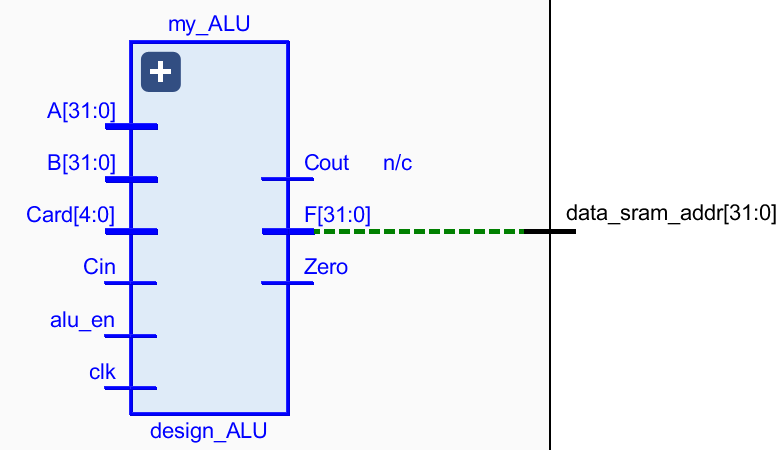
1. 处理器结构设计框图及功能描述



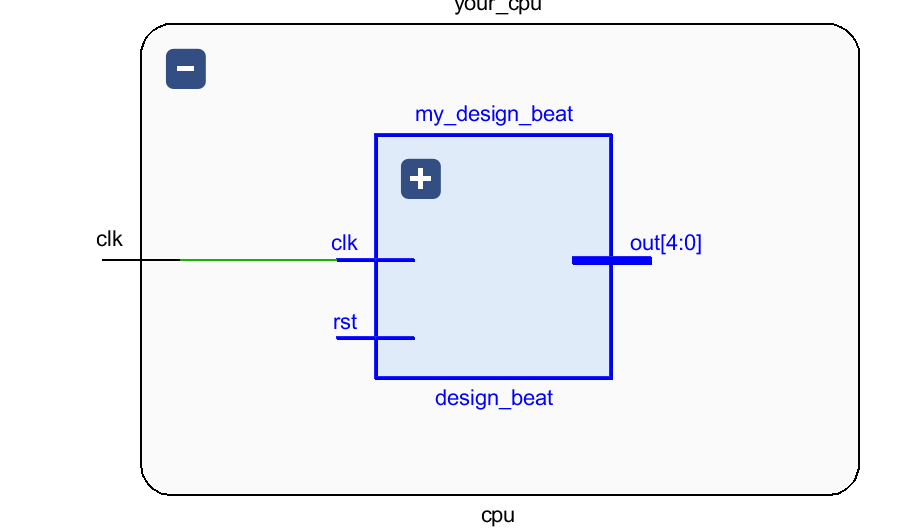
1. 各功能模块结构设计框图及功能描述



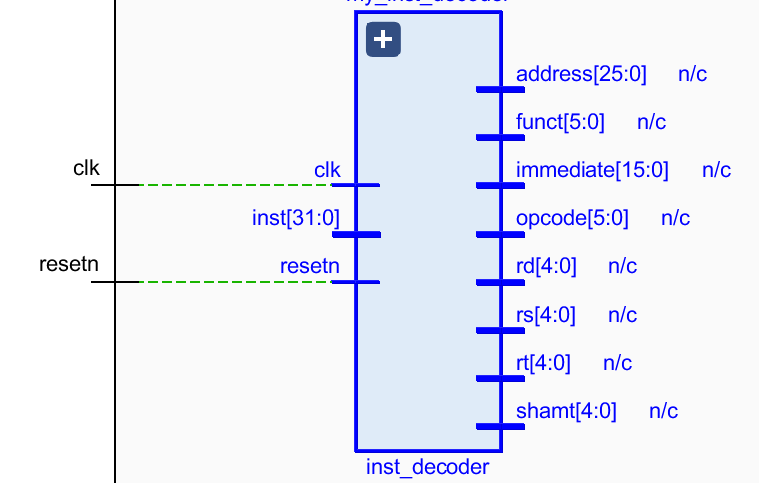
Regfile功能：1.快速存取数据 2.支持指令执行 3.控制程序执行流程



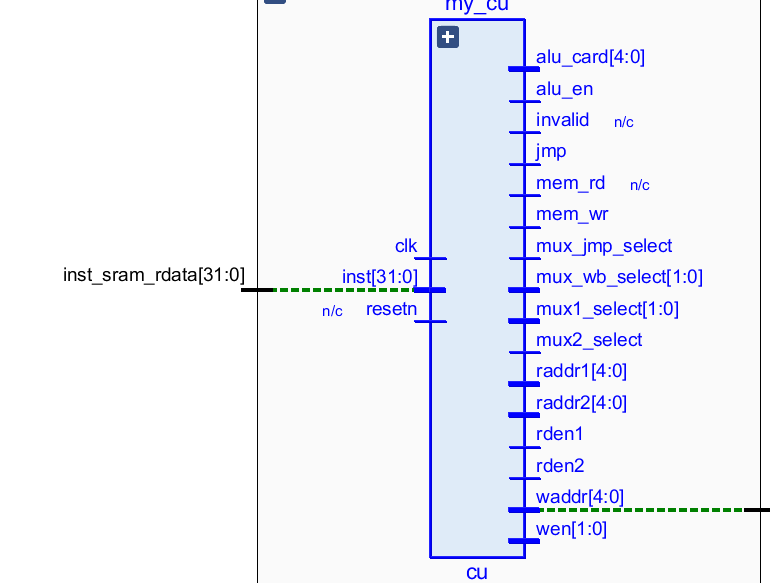
ALU功能：支持算术运算以及逻辑运算；以及在一些跳转指令中进行条件判断和地址计算。



节拍发生器功能：控制指令执行程序，控制计算机完成每一步微操作



指令译码器：支持对12种指令的识别及译码



控制单元：控制器中产生微操作命令序列的部件

1. 各模块输入输出接口信号定义（以表格形式给出，表格内容包括信号名称、位数、方向、来源/去向和信号意义）

模块名称及功能：寄存器堆 存**储临时运算数据**、**实现快速指令执行**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 信号名 | 位数 | 方向 | 来源/去向 | 意义 |
| Raddr1 | 5 | 输入 | 控制单元 | 寄存器堆读地址1 |
| Rdata1 | 32 | 输出 | ALU | 寄存器堆读返回数据1 |
| Raddr2 | 5 | 输入 | 控制单元 | 寄存器堆读地址2 |
| Rdata2 | 32 | 输出 | ALU | 寄存器堆读返回数据2 |
| waddr | 5 | 输入 | 控制单元 | 寄存器堆写地址 |
| wdata | 32 | 输入 | 控制单元 | 寄存器堆写数据 |
| we | 1 | 输入 | 控制单元 | 寄存器堆写使能 |

模块名称及功能：ALU：进行算术运算和逻辑运算

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 信号名 | 位数 | 方向 | 来源/去向 | 意义 |
| A | 32 | 输入 | 寄存器堆 | 第一个操作数 |
| B | 32 | 输入 | 寄存器堆 | 第二个操作数 |
| F | 32 | 输出 | 数据存储器/寄存器堆 | 运算结果 |
| Cin | 1 | 输入 | 寄存器堆 | 进位 |
| card | 5 | 输入 | 控制单元 | 运算操作码 |
| cout | 1 | 输出 | 控制单元 | 结果进位 |
| zero | 1 | 输出 | 控制单元 | 零标志 |

模块名称及功能：控制单元：控制器中产生微操作命令序列的部件

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 信号名 | 位数 | 方向 | 来源/去向 | 意义 |
| wen | 1 | 输入 | 指令译码器 | 指令是否写回寄存器 |
| waddr | 5 | 输入 | 指令译码器 | 指令要写回的寄存器（若wen=0可以输出任意值） |
| Rden1 | 1 | 输入 | 指令译码器 | 指令是否要读寄存器1 |
| Raddr1 | 5 | 输入 | 指令译码器 | 指令要读的寄存器（若rden1=0可以输出任意值） |
| Rden2 | 1 | 输入 | 指令译码器 | 指令是否要读寄存器2 |
| Raddr2 | 5 | 输入 | 指令译码器 | 指令要读的寄存器（若rden2=0可以输出任意值） |
| Mem\_rd | 1 | 输入 | 指令译码器 | 指令是否需要读存储器 |
| Mem\_wr | 1 | 输入 | 指令译码器 | 指令是否需要写存储器 |
| jmp | 1 | 输入 | 指令译码器 | 指令是否会直接修改PC |
| invalid | 1 | 输入 | 指令译码器 | 输入的指令无法识别 |

模块名称及功能：节拍发生器：控制指令执行程序，控制计算机完成每一步微操作

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 信号名 | 位数 | 方向 | 来源/去向 | 意义 |
| Clk | 1 | 输入 | 时钟发生器 | 时钟信号 |
| Rst | 1 | 输入 | 复位源 | 复位信号 |
| Out | 5 | 输出 | 控制单元 | 某些模块可以根据输出信号确定时序从而触发不同的操作 |

模块名称及功能：指令译码器：支持对12种指令的识别及译码

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 信号名 | 位数 | 方向 | 来源/去向 | 意义 |
| Inst | 32 | 输入 | 指令存储器 | 处理器待执行的指令 |
| wen | 1 | 输出 | 控制单元 | 指令是否写回寄存器 |
| waddr | 5 | 输出 | 控制单元 | 指令要写回的寄存器（若wen=0可以输出任意值） |
| Rden1 | 1 | 输出 | 控制单元 | 指令是否要读寄存器1 |
| Raddr1 | 5 | 输出 | 控制单元 | 指令要读的寄存器（若rden1=0可以输出任意值） |
| Rden2 | 1 | 输出 | 控制单元 | 指令是否要读寄存器2 |
| Raddr2 | 5 | 输出 | 控制单元 | 指令要读的寄存器（若rden2=0可以输出任意值） |
| Alu\_en | 1 | 输出 | ALU | 指令的结果是否为ALU的输出 |
| Alu\_card | 5 | 输出 | ALU | ALU的操作码 |
| Mem\_rd | 1 | 输出 | 控制单元 | 指令是否需要读存储器 |
| Mem\_wr | 1 | 输出 | 控制单元 | 指令是否需要写存储器 |
| jmp | 1 | 输出 | 控制单元 | 指令是否会直接修改PC |
| invalid | 1 | 输出 | 控制单元 | 输入的指令无法识别 |

以上表格不够可加页

四、实验设计及测试（如空白不够，可自行加页）

**用Verilog语言实现处理器设计**

要求采用结构化设计方法，用Verilog语言实现处理器的设计。设计包括：

1. 各模块的详细设计（包括各模块功能详述，设计方法，Verilog语言实现等）

（1）节拍发生器

module design\_beat(

input clk,

input rst,

output[4:0] out

);

reg[4:0] reg\_out;

reg flag;

assign out = reg\_out;

always @(posedge clk) begin

if (rst == 1'b1) begin

flag = 1'b0;

reg\_out = 5'b00000;

end

else begin

if (flag == 1'b0) begin

reg\_out = 5'b00001;

flag = 1'b1;

end

else begin

reg\_out = {reg\_out[3:0], reg\_out[4]};

end

end

end

endmodule

功能详述:

复位功能：当复位信号 rst 为高电平（1'b1）时，模块将 flag 寄存器清零，并将输出寄存器 reg\_out 复位为 5'b00000（即所有位都为 0）。这表示系统在重置或初始化时，输出会被清零。

输出序列生成：在非复位状态下（rst 为低电平时），模块根据 flag 寄存器的值控制输出 reg\_out 的更新。当 flag 为 0 时，reg\_out 被设置为 5'b00001，即最低位为 1，其他位为 0，同时 flag 设置为 1，表示第一阶段输出已完成。当 flag 为 1 时，reg\_out 通过将最高位（reg\_out[4]）移动到最低位（reg\_out[0]），其他位依次左移一位，实现循环右移的效果，实际上表现为一种“滚动”或“循环”的更新方式。

设计方法:

寄存器使用：定义两个寄存器 reg\_out 和 flag，用于控制输出序列的生成。reg\_out 用于存储当前输出值，flag 则指示输出序列当前处于哪个阶段。

时钟驱动逻辑：使用 always @(posedge clk) 块确保所有逻辑在时钟上升沿更新，这是同步数字电路设计的常见做法。

条件判断：通过 if-else 语句判断是否执行复位操作，或者根据 flag 值更新输出。这种结构使代码清晰易于维护。

输出更新策略：采用位操作（如位拼接 {reg\_out[3:0], reg\_out[4]}）实现 reg\_out 的循环更新，避免了复杂的循环和条件逻辑，使代码更简洁高效。

1. ALU

`define add 5'b00001

`define add\_Cin 5'b00010

`define sub 5'b00011

`define sub\_Cin 5'b00100

`define sub\_reverse 5'b00101

`define sub\_reverse\_Cin 5'b00110

`define equal\_A 5'b00111

`define equal\_B 5'b01000

`define neg\_A 5'b01001

`define neg\_B 5'b01010

`define OR 5'b01011

`define AND 5'b01100

`define XNOR 5'b01101

`define XOR 5'b01110

`define neg\_AND 5'b01111

`define equal\_zero 5'b10000

`define SLL 5'b10001

module design\_ALU(

input clk, //

input alu\_en, //

input [31:0]A,

input [31:0]B,

input Cin,

input [4:0]Card,

output reg [31:0]F,

output reg Cout,

output reg Zero

);

reg [4:0]Choose;

wire [31:0]result1;

wire [31:0]result2;

wire Cout1;

wire Cout2;

always @(posedge clk) begin

Choose = Card;

if (alu\_en == 1'b1) begin

case(Choose)

`add: begin

F = result1;

Cout = Cout1;

end

`add\_Cin: begin

F = result2;

Cout = Cout2;

end

`sub: F = A - B;

`sub\_Cin: F = A - B - Cin;

`sub\_reverse: F = B - A; //

`sub\_reverse\_Cin: F = B - A - Cin; //

`equal\_A: F = A; //

`equal\_B: F = B; //

`neg\_A: F = ~A; //

`neg\_B: F = ~B; //

`OR: F = A | B; //

`AND: F = A & B; //

`XNOR: F = ~(A ^ B); //

`XOR: F = A ^ B; //

`neg\_AND: F = ~(A & B); //

`equal\_zero: F = 32'b0; //

`SLL: F = B << A; //

default: F = 2\*\*32 - 1;

endcase

if (!F) begin

Zero = 1;

end

else begin

Zero = 0;

end

end

end

design\_adder\_0 adder1(

.A(A),

.B(B),

.Cin(1'b0),

.F(result1),

.Cout(Cout1)

);

design\_adder\_0 adder2(

.A(A),

.B(B),

.Cin(Cin),

.F(result2),

.Cout(Cout2)

);

endmodule

**功能详述**

**输入和输出**：

输入：时钟信号clk，ALU使能信号alu\_en，两个32位的操作数A和B，一个进位输入Cin，以及一个5位的控制信号Card。

输出：一个32位的结果F，一个进位输出Cout，以及一个零标志Zero（表示结果是否为0）。

**操作类型**：

使用Card信号来选择要执行的操作。这些操作包括加法、带进位的加法、减法、带进位的减法、反向减法、比较（equal\_A和equal\_B实际上是将A或B赋值给F，并非真正的比较操作）、取反、逻辑或、逻辑与、同或、异或、非逻辑与、置零以及左移位。

对于加法（add和add\_Cin）操作，使用了两个design\_adder\_0加法器实例（adder1和adder2）来分别处理不带进位和带进位的加法。

**零标志**：

在每个时钟上升沿且alu\_en为高时，根据运算结果F来设置Zero标志。如果F为0，则Zero被置为1；否则，Zero被置为0。

### **设计方法**

**模块化设计**：

通过定义design\_adder\_0加法器模块来复用加法逻辑，从而简化了design\_ALU模块的设计。这种方法提高了代码的可读性和可维护性。

**控制信号映射**：

使用预处理器指令（如`define add 5'b00001）来定义每个操作对应的控制信号编码，这样在case语句中可以直接使用这些宏，使代码更加清晰。

**时钟同步**：

所有逻辑操作都在时钟的上升沿进行，这是同步设计的典型做法。它确保了操作的稳定性和可预测性。

**条件执行**：

通过检查alu\_en信号来决定是否执行ALU操作。这允许在不需要执行ALU运算时，保持当前状态不变，从而节省功耗并避免不必要的操作。

**结果处理**：

对于每种操作，都有相应的逻辑来设置结果F、进位输出Cout以及零标志Zero。特别是，对于加法操作，使用了专门的加法器模块来得到精确的结果和进位输出。

**默认情况处理**：

在case语句中包含了default分支，以处理任何未定义的控制信号编码。这可以防止在设计时出现意外的行为或未定义的结果。

1. 寄存器堆

module regfile(

input clk, // 时钟信号

input we, // 写使能信号，1时表示可以写入数据

input resetn, // 异步复位信号，低电平复位

input [4:0] raddr1, // 读取端口1的地址

input [4:0] raddr2, // 读取端口2的地址

input [4:0] waddr, // 写入端口的地址

input [31:0] wdata, // 写入的数据

output [31:0] rdata1, // 从读取端口1输出的数据

output [31:0] rdata2 // 从读取端口2输出的数据

);

reg [31:0] regheap[31:0]; // 定义32个32位宽的寄存器，用来存储数据

integer i;

always @(posedge clk) begin // 时钟上升沿时触发

if(resetn == 0) begin // 当复位信号为0（低电平）时，执行复位操作

// 遍历所有32个寄存器，将它们复位为0

for (i = 0; i < 32; i = i + 1)

regheap[i] <= 32'b0; // 异步复位，所有寄存器清零

end

else if (we) // 如果写使能信号为1，则进行写入操作

regheap[waddr] <= wdata; // 在写地址`waddr`对应的寄存器写入`wdata`数据

end

// 读取端口1的输出

// 如果读取地址为0，则输出0（因为寄存器$0通常固定为0）

// 否则，输出`regheap`中对应地址的寄存器内容

assign rdata1 = (raddr1 == 5'b0) ? 32'b0 : regheap[raddr1];

// 读取端口2的输出

// 如果读取地址为0，则输出0（因为寄存器$0通常固定为0）

// 否则，输出`regheap`中对应地址的寄存器内容

assign rdata2 = (raddr2 == 5'b0) ? 32'b0 : regheap[raddr2];

endmodule

### **功能详述**

**时钟控制**：模块使用clk信号作为时钟输入，所有寄存器的读写操作都在时钟的上升沿触发。这保证了操作的同步性和稳定性。

**异步复位**：模块通过resetn信号实现异步复位。当resetn为低电平时，所有寄存器（regheap数组中的32个32位寄存器）被清零。异步复位意味着复位操作不依赖于时钟信号，可以在任何时刻立即生效。

**写入操作**：当we（写使能）信号为高电平时，模块将wdata（写入数据）写入到由waddr（写入地址）指定的寄存器中。这允许在单个时钟周期内更新寄存器的内容。

**读取操作**：模块提供了两个读取端口（raddr1和raddr2），分别对应rdata1和rdata2输出。对于每个读取端口，如果读取地址（raddr1或raddr2）为0（通常寄存器地址0用于特定目的，如零寄存器），则相应输出被置为0。否则，输出为regheap数组中对应地址的寄存器内容。

**寄存器文件大小**：模块内部定义了一个名为regheap的数组，包含32个32位宽的寄存器，支持从地址0到31的访问。

### **设计方法**

**寄存器定义**：使用Verilog的reg类型数组regheap来模拟寄存器文件。每个寄存器都是一个32位的寄存器，整个数组模拟了一个具有32个寄存器的寄存器文件。

**时钟和复位控制**：利用always @(posedge clk)块来确保所有操作都在时钟的上升沿进行。通过检查resetn信号的状态来实现异步复位逻辑。

**条件逻辑**：使用条件运算符（? :）来处理读取地址为0的特殊情况，将rdata1和rdata2的输出设置为0，否则输出对应地址的寄存器内容。

**写入逻辑**：通过简单的条件判断（if (we)）来控制写入操作。当we为高电平时，将wdata写入到waddr指定的寄存器中。

**端口设计**：模块定义了必要的输入输出端口，包括时钟信号clk、复位信号resetn、写使能信号we、写入地址waddr和写入数据wdata，以及两个读取端口的地址raddr1和raddr2，以及对应的读取数据输出rdata1和rdata2。

1. 指令译码器

`define INSE\_ADD 6'b100000 // R类型的加法指令

`define INSE\_SUB 6'b100010 // R类型的减法指令

`define INSE\_AND 6'b100100 // R类型的按位与指令

`define INSE\_OR 6'b100101 // R类型的按位或指令

`define INSE\_XOR 6'b100110 // R类型的按位异或指令

`define INSE\_MOVZ 6'b001010 // R类型的MOVZ指令

`define INSE\_SLL 6'b000000 // R类型的逻辑左移指令

// 定义指令的操作码类型

`define TYPE\_R 6'b000000 // R类型指令

`define TYPE\_MS 6'b101011 // 存储指令 (SW)

`define TYPE\_ML 6'b100011 // 读取指令 (LW)

`define TYPE\_J 6'b000010 // 跳转指令 (J)

`define TYPE\_CMP 6'b111110 // 比较指令 (CMP)

`define TYPE\_BBT 6'b111111 // 分支条件跳转指令 (BBT)

// 指令解码器模块

module inst\_decoder\_v2(

input [31:0] inst, // 要执行的指令

input Cond, // 条件标志结果

output wen, // 指令是否写回寄存器堆

output [4:0] waddr, // 指令写回的寄存器地址

output rden1, // 指令是否需要读取寄存器1

output [4:0] raddr1, // 要读取的寄存器1地址

output rden2, // 指令是否需要读取寄存器2

output [4:0] raddr2, // 要读取的寄存器2地址

output alu\_en, // 指令结果是否来自ALU

output [4:0] alu\_card,// ALU的操作码

output mem\_rd, // 指令是否需要读取存储器

output mem\_wr, // 指令是否需要写入存储器

output jmp, // 指令是否修改PC（跳转）

output invalid, // 输入的指令是否有效

output [31:0] Imm, // 立即数扩展

output [1:0] mux\_en // 多路选择器控制信号

);

// 检查指令是否有效

// R类型指令的不同操作，如ADD、SUB、AND等

wire add\_valid = (inst[31:26] == `TYPE\_R) & (inst[5:0] == `INSE\_ADD);

wire sub\_valid = (inst[31:26] == `TYPE\_R) & (inst[5:0] == `INSE\_SUB);

wire and\_valid = (inst[31:26] == `TYPE\_R) & (inst[5:0] == `INSE\_AND);

wire or\_valid = (inst[31:26] == `TYPE\_R) & (inst[5:0] == `INSE\_OR);

wire xor\_valid = (inst[31:26] == `TYPE\_R) & (inst[5:0] == `INSE\_XOR);

wire sw\_valid = (inst[31:26] == `TYPE\_MS); // SW指令

wire lw\_valid = (inst[31:26] == `TYPE\_ML); // LW指令

wire j\_valid = (inst[31:26] == `TYPE\_J); // J指令

// MOVZ和SLL指令

wire movz\_valid = (inst[31:26] == `TYPE\_R) & (inst[5:0] == `INSE\_MOVZ);

wire sll\_valid = (inst[31:26] == `TYPE\_R) & (inst[25:21] == 5'b0) & (inst[5:0] == `INSE\_SLL);

wire cmp\_valid = (inst[31:26] == `TYPE\_CMP) & (inst[10:0] == 11'b0); // CMP指令

wire bbt\_valid = (inst[31:26] == `TYPE\_BBT); // BBT指令

// 指令无效信号，只有当所有指令无效时才为1

assign invalid = !(add\_valid | sub\_valid | and\_valid | or\_valid | xor\_valid | sw\_valid | lw\_valid | j\_valid | movz\_valid | sll\_valid | cmp\_valid | bbt\_valid);

// 写使能信号

wire wen1 = ((inst[31:26] == `TYPE\_R) & (inst[5:0] != `INSE\_MOVZ)) & (inst[15:11] != 5'b0) ? 1'b1 : 1'b0;

wire wen2 = (inst[31:26] == `TYPE\_ML) ? 1'b1 : 1'b0; // LW指令写使能

wire wen3 = (inst[31:26] == `TYPE\_R) & (inst[5:0] == `INSE\_MOVZ) & Cond; // MOVZ指令条件满足时的写使能

wire wen4 = (inst[31:26] == `TYPE\_CMP) ? 1'b1 : 1'b0; // CMP指令写使能

assign wen = (!invalid) & (wen1 | wen2 | wen3 | wen4); // 只有当指令有效时才允许写入

// 写地址

assign waddr = {5{(!invalid)}} &

( ({5{wen1}} & inst[15:11]) |

({5{wen2}} & inst[20:16]) |

({5{wen3}} & inst[15:11]) |

({5{wen4}} & inst[15:11]) );

// 读取使能信号1

assign rden1 = (!invalid) & ((inst[31:26] == `TYPE\_R) | (inst[31:26] == `TYPE\_ML) | (inst[31:26] == `TYPE\_MS) | (inst[31:26] == `TYPE\_CMP) | (inst[31:26] == `TYPE\_BBT));

// 读取地址1

assign raddr1 = {5{rden1}} & ((sll\_valid | movz\_valid) ? inst[20:16] : inst[25:21]);

// 读取使能信号2

assign rden2 = (!invalid) & (!j\_valid) & (!lw\_valid) & (!bbt\_valid) & (!sll\_valid);

// 读取地址2

assign raddr2 = {5{rden2}} & ((movz\_valid) ? inst[25:21] : inst[20:16]);

// ALU使能信号

assign alu\_en = (!invalid) & (!lw\_valid);

// ALU操作码

assign alu\_card = (({5{add\_valid}} & 5'b00001) |

({5{sub\_valid}} & 5'b00011) |

({5{and\_valid}} & 5'b01100) |

({5{or\_valid}} & 5'b01011) |

({5{xor\_valid}} & 5'b01110) |

({5{sw\_valid}} & 5'b00001) |

({5{lw\_valid}} & 5'b00001) |

({5{j\_valid}} & 5'b10010) |

({5{movz\_valid}}& 5'b01000) |

({5{sll\_valid}} & 5'b10001) |

({5{cmp\_valid}} & 5'b10100) |

({5{bbt\_valid}} & 5'b10011));

// 存储器读取信号

assign mem\_rd = (!invalid) & (lw\_valid | sw\_valid);

// 存储器写入信号

assign mem\_wr = (!invalid) & (sw\_valid);

// 跳转信号

assign jmp = (!invalid) & (j\_valid | bbt\_valid);

// A寄存器位置的多路选择器控制信号（1为寄存器，0为PC）

assign mux\_en[0] = (!invalid) & ((inst[31:26] == `TYPE\_R) | (inst[31:26] == `TYPE\_ML) | (inst[31:26] == `TYPE\_MS) | (inst[31:26] == `TYPE\_CMP));

// B寄存器位置的多路选择器控制信号（1为寄存器，0为立即数）

assign mux\_en[1] = (!invalid) & (add\_valid | sub\_valid | and\_valid | or\_valid | xor\_valid | movz\_valid | cmp\_valid);

// 立即数符号扩展

assign Imm = {32{(!invalid)}} &

( ({32{sw\_valid}} & {{16{inst[15]}}, inst[15:0]}) |

({32{lw\_valid}} & {{16{inst[15]}}, inst[15:0]}) |

({32{j\_valid}} & {{6{inst[25]}}, inst[25:0]}) |

({32{sll\_valid}}& {{27{inst[10]}}, inst[10:6]}) |

({32{bbt\_valid}}& {{16{inst[15]}}, inst[15:0]}) );

endmodule

### **功能详述**

**指令识别与有效性检查**：

通过检查指令的高位字段（操作码部分），确定指令的类型（如R类型、存储指令、读取指令、跳转指令等）。

根据操作码和指令的低位字段（功能码或地址码），进一步确定具体的指令（如加法、减法、按位与等）。

如果指令不被识别为任何有效指令，则标记为无效（invalid信号）。

### **设计方法**

**模块化设计**：

将指令解码过程分解为多个小模块（如指令有效性检查、写操作处理、读操作处理等），使代码更加清晰和易于维护。

**条件逻辑**：

使用条件逻辑（如if语句或条件运算符? :）来根据指令的不同字段确定指令的类型和功能。

**位操作**：

利用位操作（如位与&、位或|、位非~等）来提取指令字段、设置控制信号等。

**参数化**：

使用define指令定义指令的操作码和功能码，提高了代码的可读性和可维护性。

1. 控制单元

always @(posedge clk) begin

if (!resetn) begin

//

reg\_wen <= 1'b0;

reg\_waddr <= 5'b00000;

reg\_mux\_wb\_select <= 2'b00;

reg\_rden1 <= 1'b0;

reg\_raddr1 <= 5'b00000;

reg\_rden2 <= 1'b0;

reg\_raddr2 <= 5'b00000;

reg\_alu\_en <= 1'b0;

reg\_alu\_card <= 5'b00000;

reg\_mux1\_select <= 2'b00;

reg\_mux2\_select <= 1'b0;

reg\_mem\_rd <= 1'b0;

reg\_mem\_wr <= 1'b0;

reg\_jmp <= 1'b0;

reg\_mux\_jmp\_select <= 1'b0;

reg\_invalid <= 1'b0;

end

else if (inst[31:26] == {6{1'b0}}) begin

if (inst[5:0] != {6{1'b0}}) begin

case (inst[10:0])

//

11'b00000\_100000: begin

reg\_wen = 1'b1;

reg\_waddr = inst[15:11];

reg\_mux\_wb\_select = 2'b01;

reg\_rden1 = 1'b1;

reg\_raddr1 = inst[25:21];

reg\_rden2 = 1'b1;

reg\_raddr2 = inst[20:16];

reg\_alu\_en = 1'b1;

reg\_alu\_card = 5'b00001;

reg\_mux1\_select = 2'b01;

reg\_mux2\_select = 1'b0;

reg\_mem\_rd = 1'b0;

reg\_mem\_wr = 1'b0;

reg\_jmp = 1'b0;

// reg\_mux\_jmp\_select = 1'bz;

reg\_invalid = 1'b0;

end

//

11'b00000\_100010: begin

reg\_wen = 1'b1;

reg\_waddr = inst[15:11];

reg\_mux\_wb\_select = 2'b01;

reg\_rden1 = 1'b1;

reg\_raddr1 = inst[25:21];

reg\_rden2 = 1'b1;

reg\_raddr2 = inst[20:16];

reg\_alu\_en = 1'b1;

reg\_alu\_card = 5'b00011;

reg\_mux1\_select = 2'b01;

reg\_mux2\_select = 1'b0;

reg\_mem\_rd = 1'b0;

reg\_mem\_wr = 1'b0;

reg\_jmp = 1'b0;

// reg\_mux\_jmp\_select = 1'bz;

reg\_invalid = 1'b0;

end

//

11'b00000\_100100: begin

reg\_wen = 1'b1;

reg\_waddr = inst[15:11];

reg\_mux\_wb\_select = 2'b01;

reg\_rden1 = 1'b1;

reg\_raddr1 = inst[25:21];

reg\_rden2 = 1'b1;

reg\_raddr2 = inst[20:16];

reg\_alu\_en = 1'b1;

reg\_alu\_card = 5'b01100;

reg\_mux1\_select = 2'b01;

reg\_mux2\_select = 1'b0;

reg\_mem\_rd = 1'b0;

reg\_mem\_wr = 1'b0;

reg\_jmp = 1'b0;

// reg\_mux\_jmp\_select = 1'bz;

reg\_invalid = 1'b0;

end

//

11'b00000\_100101: begin

reg\_wen = 1'b1;

reg\_waddr = inst[15:11];

reg\_mux\_wb\_select = 2'b01;

reg\_rden1 = 1'b1;

reg\_raddr1 = inst[25:21];

reg\_rden2 = 1'b1;

reg\_raddr2 = inst[20:16];

reg\_alu\_en = 1'b1;

reg\_alu\_card = 5'b01011;

reg\_mux1\_select = 2'b01;

reg\_mux2\_select = 1'b0;

reg\_mem\_rd = 1'b0;

reg\_mem\_wr = 1'b0;

reg\_jmp = 1'b0;

// reg\_mux\_jmp\_select = 1'bz;

reg\_invalid = 1'b0;

end

//

11'b00000\_100110: begin

reg\_wen = 1'b1;

reg\_waddr = inst[15:11];

reg\_mux\_wb\_select = 2'b01;

reg\_rden1 = 1'b1;

reg\_raddr1 = inst[25:21];

reg\_rden2 = 1'b1;

reg\_raddr2 = inst[20:16];

reg\_alu\_en = 1'b1;

reg\_alu\_card = 5'b01110;

reg\_mux1\_select = 2'b01;

reg\_mux2\_select = 1'b0;

reg\_mem\_rd = 1'b0;

reg\_mem\_wr = 1'b0;

reg\_jmp = 1'b0;

// reg\_mux\_jmp\_select = 1'bz;

reg\_invalid = 1'b0;

end

//

11'b00000\_001010: begin

reg\_wen = 1'b0; //

reg\_waddr = inst[15:11];

reg\_mux\_wb\_select = 2'b00;

reg\_rden1 = 1'b1;

reg\_raddr1 = inst[25:21];

reg\_rden2 = 1'b1;

reg\_raddr2 = inst[20:16];

reg\_alu\_en = 1'b1;

reg\_alu\_card = 5'b00001;

reg\_mux1\_select = 2'b00; //

reg\_mux2\_select = 1'b0;

reg\_mem\_rd = 1'b0;

reg\_mem\_wr = 1'b0;

reg\_jmp = 1'b0;

// reg\_mux\_jmp\_select = 1'bz;

reg\_invalid = 1'b0;

end

default: begin

reg\_invalid = 1'b1;

end

endcase

end

//

else if (inst[25:21] == {6{1'b0}}) begin

reg\_wen = 1'b1;

reg\_waddr = inst[15:11];

reg\_mux\_wb\_select = 2'b01;

reg\_rden1 = 1'b0;

// reg\_raddr1 = 5'bz;

reg\_rden2 = 1'b1;

reg\_raddr2 = inst[20:16];

reg\_alu\_en = 1'b1;

reg\_alu\_card = 5'b10001;

reg\_mux1\_select = 2'b10;

reg\_mux2\_select = 1'b0;

reg\_mem\_rd = 1'b0;

reg\_mem\_wr = 1'b0;

reg\_jmp = 1'b0;

// reg\_mux\_jmp\_select = 1'bz;

reg\_invalid = 1'b0;

end

//

else begin

reg\_invalid = 1'b1;

end

end

//

else if (inst[31:26]==6'b101011) begin

reg\_wen = 1'b0;

// reg\_waddr = 5'bz;

reg\_mux\_wb\_select = 2'b01;

reg\_rden1 = 1'b1;

reg\_raddr1 = inst[25:21]; // base

reg\_rden2 = 1'b1;

reg\_raddr2 = inst[20:16]; // rt

reg\_alu\_en = 1'b1;

reg\_alu\_card = 5'b00001;

reg\_mux1\_select = 2'b01;

reg\_mux2\_select = 1'b1;

reg\_mem\_rd = 1'b0;

reg\_mem\_wr = 1'b1;

reg\_jmp = 1'b0;

// reg\_mux\_jmp\_select = 1'bz;

reg\_invalid = 1'b0;

end

//

else if (inst[31:26]==6'b100011) begin

reg\_wen = 1'b1;

reg\_waddr = inst[20:16];

reg\_mux\_wb\_select = 2'b10;

reg\_rden1 = 1'b1;

reg\_raddr1 = inst[25:21];

reg\_rden2 = 1'b0;

reg\_raddr2 = inst[20:16];

reg\_alu\_en = 1'b1;

reg\_alu\_card = 5'b00001;

reg\_mux1\_select = 2'b01;

reg\_mux2\_select = 1'b1;

reg\_mem\_rd = 1'b1;

reg\_mem\_wr = 1'b0;

reg\_jmp = 1'b0;

// reg\_mux\_jmp\_select = 1'bz;

reg\_invalid = 1'b0;

end

//

else if (inst[31:26]==6'b000010) begin

reg\_wen = 1'b0;

// reg\_waddr = 5'bz;

// reg\_mux\_wb\_select = 2'bz;

reg\_rden1 = 1'b0;

// reg\_raddr1 = 5'bz;

reg\_rden2 = 1'b0;

// reg\_raddr2 = 5'bz;

reg\_alu\_en = 1'b0;

// reg\_alu\_card = 5'bz;

// reg\_mux1\_select = 2'bz;

// reg\_mux2\_select = 1'bz;

reg\_mem\_rd = 1'b0;

reg\_mem\_wr = 1'b0;

reg\_jmp = 1'b1;

reg\_mux\_jmp\_select = 1'b1;

reg\_invalid = 1'b0;

end

//

else if (inst[31:26]==6'b111110) begin

reg\_wen = 1'b1;

reg\_waddr = inst[15:11];

reg\_mux\_wb\_select = 2'b11;

reg\_rden1 = 1'b1;

reg\_raddr1 = inst[25:21];

reg\_rden2 = 1'b1;

reg\_raddr2 = inst[20:16];

reg\_alu\_en = 1'b0;

// reg\_alu\_card = 5'bz;

// reg\_mux1\_select = 2'bz;

// reg\_mux2\_select = 1'bz;

reg\_mem\_rd = 1'b0;

reg\_mem\_wr = 1'b0;

reg\_jmp = 1'b0;

// reg\_mux\_jmp\_select = 1'bz;

reg\_invalid = 1'b0;

end

//

else if (inst[31:26]==6'b111111) begin

reg\_wen = 1'b0;

// reg\_waddr = 5'bz;

// reg\_mux\_wb\_select = 2'bz;

reg\_rden1 = 1'b1;

reg\_raddr1 = inst[25:21];

reg\_rden2 = 1'b0;

// reg\_raddr2 = 5'bz;

reg\_alu\_en = 1'b0;

// reg\_alu\_card = 5'bz;

// reg\_mux1\_select = 2'bz;

// reg\_mux2\_select = 1'bz;

reg\_mem\_rd = 1'b0;

reg\_mem\_wr = 1'b0;

reg\_jmp = 1'b1; //

reg\_mux\_jmp\_select = 1'b0; /

reg\_invalid = 1'b0;

end

//

else begin

reg\_invalid = 1'b1;

end

end

Endmodule

**功能详述：**

1. 指令译码

功能描述：CU（控制单元）接收来自内存的指令，并对这些指令进行分析和解码。指令解码是将指令的二进制代码转换为CU能够理解和执行的一系列控制信号的过程。

2. 控制信号生成

功能描述：根据指令解码的结果，CU生成各种控制信号，确保计算机系统中的各个部件能够协调一致地工作，执行指令。

3. 状态控制

功能描述：CU根据指令的执行情况，持续调整自身的状态以及系统的状态，以确保每条指令能够正确执行并完成。

4. 中断处理

功能描述：当系统发生中断事件时，CU会暂停当前程序的执行，并根据中断类型调用相应的处理程序，处理完毕后再返回到正常指令的执行流程中。

5. 协调各个部件的协同工作

功能描述：CU作为系统的核心控制部件，负责协调CPU内的运算单元、寄存器、内存等其他硬件模块的协同工作，确保指令被正确执行。

6. 指令执行流程控制

功能描述：CU控制整个指令的执行流程，包括取指周期、间址周期、执行周期以及中断周期等，确保指令能够按照预定的步骤顺利完成。设计方法：

对每条具体的指令所需要的信号单独设计

1. 系统的详细设计（包括系统功能详述，设计方法，Verilog语言实现等）

module cpu(

input clk,

input resetn,

output inst\_sram\_en,

output[31:0] inst\_sram\_addr,

input[31:0] inst\_sram\_rdata,

output data\_sram\_en,

output[3:0] data\_sram\_wen,

output[31:0] data\_sram\_addr,

output[31:0] data\_sram\_wdata,

input[31:0] data\_sram\_rdata,

output[31:0] debug\_wb\_pc,

output debug\_wb\_rf\_wen,

output[4:0] debug\_wb\_rf\_wnum,

output[31:0] debug\_wb\_rf\_wdata

);

// ALU

wire[31:0] alu\_output;

wire Cout;

wire Zero;

wire[4:0] beat\_out;

wire[31:0] rdata1;

wire[31:0] rdata2;

wire wen;

wire[4:0] waddr;

wire[1:0] mux\_wb\_select;

wire rden1;

wire[4:0] raddr1;

wire rden2;

wire[4:0] raddr2;

wire alu\_en;

wire[4:0] alu\_card;

wire[1:0] mux1\_select;

wire mux2\_select;

wire mem\_rd;

wire mem\_wr;

wire jmp;

wire mux\_jmp\_select;

wire invalid;

reg[31:0] pc0;

reg[31:0] pc1;

wire[31:0] pc\_out;

wire[31:0] alu\_select1;

wire[31:0] alu\_select2;

wire[31:0] wb\_reg;

reg wire\_resetn;

reg reg\_inst\_sram\_en;

reg[31:0] reg\_inst\_sram\_addr;

reg[31:0] reg\_next\_inst\_sram\_addr;

reg reg\_data\_sram\_en;

reg[3:0] reg\_data\_sram\_wen;

reg[31:0] reg\_data\_sram\_wdata;

reg[31:0] reg\_debug\_wb\_pc;

reg reg\_debug\_wb\_rf\_wen;

reg[4:0] reg\_debug\_wb\_rf\_wnum;

reg[31:0] reg\_debug\_wb\_rf\_wdata;

assign inst\_sram\_en = reg\_inst\_sram\_en;

assign inst\_sram\_addr = reg\_inst\_sram\_addr;

assign data\_sram\_en = reg\_data\_sram\_en;

assign data\_sram\_wen = reg\_data\_sram\_wen;

assign data\_sram\_addr = alu\_output;

assign data\_sram\_wdata = rdata2;

assign debug\_wb\_pc = reg\_debug\_wb\_pc;

assign debug\_wb\_rf\_wen = reg\_debug\_wb\_rf\_wen;

assign debug\_wb\_rf\_wnum = waddr;

assign debug\_wb\_rf\_wdata = wb\_reg;

initial begin

wire\_resetn = 1'b1;

#150 wire\_resetn = 1'b0;

end

design\_beat my\_design\_beat(

.clk(clk),

.rst(wire\_resetn),

.out(beat\_out)

);

reg reg\_jmp;

reg reg\_wen;

always @(posedge resetn or posedge beat\_out[0] or posedge beat\_out[2] or posedge beat\_out[4]) begin

if (resetn == 1'b0) begin

reg\_next\_inst\_sram\_addr <= 32'h00000000;

reg\_inst\_sram\_addr <= 32'h00000000;

reg\_inst\_sram\_en <= 1'b1;

reg\_data\_sram\_en <= 1'b0;

reg\_data\_sram\_wen <= 4'b0000;

reg\_data\_sram\_wdata <= 32'h00000000;

reg\_debug\_wb\_pc <= 32'h00000000;

reg\_debug\_wb\_rf\_wen <= 1'b0;

reg\_debug\_wb\_rf\_wnum <= 5'b00000;

reg\_jmp <= 1'b0;

reg\_wen <= 1'b0;

end

if (resetn == 1'b1 && beat\_out == 5'b00000) begin

reg\_next\_inst\_sram\_addr <= 32'h00000000;

reg\_inst\_sram\_addr <= 32'h00000000;

reg\_inst\_sram\_en <= 1'b1;

reg\_data\_sram\_en <= 1'b0;

reg\_data\_sram\_wen <= 4'b0000;

reg\_data\_sram\_wdata <= 32'h00000000;

reg\_debug\_wb\_pc <= 32'h00000000;

reg\_debug\_wb\_rf\_wen <= 1'b0;

reg\_debug\_wb\_rf\_wnum <= 5'b00000;

reg\_jmp <= 1'b0;

reg\_wen <= 1'b0;

end

else begin

if (beat\_out[0] == 1'b1) begin

reg\_data\_sram\_wen = 4'b0000;

reg\_debug\_wb\_pc = reg\_next\_inst\_sram\_addr;

reg\_debug\_wb\_rf\_wen = 1'b0;

reg\_inst\_sram\_en = 1'b1;

reg\_next\_inst\_sram\_addr = reg\_inst\_sram\_addr + 4;

end

if (beat\_out[2] == 1'b1) begin

if (mem\_rd == 1'b1) begin

reg\_data\_sram\_en = 1'b1;

end

else begin

reg\_data\_sram\_en = 1'b0;

end

if (mem\_wr == 1'b1) begin

reg\_data\_sram\_en = 1'b1;

reg\_data\_sram\_wen = 4'b1111;

reg\_data\_sram\_wdata = rdata2;

end

else begin

reg\_data\_sram\_en = 1'b1;

reg\_data\_sram\_wen = 4'b0000;

end

end

if (beat\_out[4] == 1'b1) begin

if (reg\_jmp == 1'b1) begin

reg\_next\_inst\_sram\_addr = pc\_out;

reg\_inst\_sram\_addr = reg\_next\_inst\_sram\_addr;

end

else begin

reg\_inst\_sram\_addr = reg\_next\_inst\_sram\_addr;

end

// reg\_debug\_wb\_rf\_wdata = wb\_reg;

reg\_debug\_wb\_rf\_wen = reg\_wen;

reg\_debug\_wb\_rf\_wnum = waddr;

end

end

end

inst\_decoder my\_inst\_decoder (

.clk(clk),

.resetn(resetn),

.inst(inst),

.opcode(opcode),

.rs(rs),

.rt(rt),

.rd(rd),

.shamt(shamt),

.funct(funct),

.immediate(immediate),

.address(address)

);

cu my\_cu(

.clk(beat\_out[1]),

.inst(inst\_sram\_rdata[31:0]),

.wen(wen), .waddr(waddr), .mux\_wb\_select(mux\_wb\_select),

.rden1(rden1), .raddr1(raddr1),

.rden2(rden2), .raddr2(raddr2),

.alu\_en(alu\_en), .alu\_card(alu\_card), .mux1\_select(mux1\_select), .mux2\_select(mux2\_select),

.mem\_rd(mem\_rd), .mem\_wr(mem\_wr),

.jmp(jmp), .mux\_jmp\_select(mux\_jmp\_select),

.invalid(invalid)

);

regfile\_0 my\_reg(

.clk(beat\_out[4]),

.rden1(rden1),

.rden2(rden2),

.raddr1(raddr1),

.raddr2(raddr2),

.we(reg\_wen),

.waddr(waddr),

.wdata(wb\_reg),

.rdata1(rdata1),

.rdata2(rdata2)

);

reg[4:0] index;

reg[31:0] reg\_rdata1;

reg[15:0] offset;

always @(posedge beat\_out[2]) begin

index = inst\_sram\_rdata[20:15];

reg\_rdata1 = rdata1;

offset = inst\_sram\_rdata[15:0];

reg\_jmp = jmp;

if (reg\_jmp == 1'b1 && !mux\_jmp\_select) begin

if (reg\_rdata1[index] == 1'b1) begin

reg\_jmp = 1'b1;

end

else begin

reg\_jmp = 1'b0;

end

end

pc0 = reg\_next\_inst\_sram\_addr + ({{16{offset[15]}}, offset} << 2);

pc1 = {reg\_next\_inst\_sram\_addr[31:28], {inst\_sram\_rdata[25:0] << 2}};

end

reg[31:0] A;

reg[31:0] B;

reg[31:0] cmp\_rd;

always @(posedge beat\_out[2]) begin

A = rdata1;

B = rdata2;

cmp\_rd = {{22{1'b0}}, 5'b11111, 5'b00000};

if (A == B) begin

cmp\_rd[0] = 1'b1;

cmp\_rd[5] = 1'b0;

end

if (A < B) begin

cmp\_rd[1] = 1'b1;

cmp\_rd[6] = 1'b0;

end

if (A <= B) begin

cmp\_rd[3] = 1'b1;

cmp\_rd[8] = 1'b0;

end

if ($unsigned(A) < $unsigned(B)) begin

cmp\_rd[2] = 1'b1;

cmp\_rd[7] = 1'b0;

end

if ($unsigned(A) <= $unsigned(B)) begin

cmp\_rd[4] = 1'b1;

cmp\_rd[9] = 1'b0;

end

end

design\_ALU my\_ALU(

.clk(beat\_out[2]),

.alu\_en(alu\_en),

.A(alu\_select1),

.B(alu\_select2),

.Cin(1'b0),

.Card(alu\_card),

.F(alu\_output),

.Cout(Cout),

.Zero(Zero)

);

reg reg\_zero;

reg reg\_mux\_wb\_select;

always @(posedge beat\_out[3]) begin

reg\_wen = wen;

reg\_zero = Zero;

reg\_mux\_wb\_select = mux\_wb\_select;

reg\_debug\_wb\_rf\_wnum = waddr;

if (reg\_wen == 1'b0) begin

if (reg\_zero == 1'b1 && reg\_mux\_wb\_select == 2'b00) begin

reg\_wen = 1'b1;

end

end

if (reg\_wen == 1'b1 && reg\_debug\_wb\_rf\_wnum == 5'b00000) begin

reg\_wen = 1'b0;

end

end

MUX\_2\_to\_1\_0 mux\_pc(

.data0(pc0),

.data1(pc1),

.select(mux\_jmp\_select),

.out(pc\_out)

);

MUX\_3\_to\_1\_0 mux\_alu1(

.data0(32'b0),

.data1(rdata1),

.data2({27'b0, inst\_sram\_rdata[10:6]}),

.select(mux1\_select),

.out(alu\_select1)

);

MUX\_2\_to\_1\_0 mux\_alu2(

.data0(rdata2),

.data1({{16{inst\_sram\_rdata[15]}}, inst\_sram\_rdata[15:0]}),

.select(mux2\_select),

.out(alu\_select2)

);

MUX\_4\_to\_1\_0 mux\_wb\_reg(

.data0(rdata1),

.data1(alu\_output),

.data2(data\_sram\_rdata),

.data3(cmp\_rd),

.select(mux\_wb\_select),

.out(wb\_reg)

);

endmodule

功能详述：

支持对测试方案中的12种指令的识别及执行

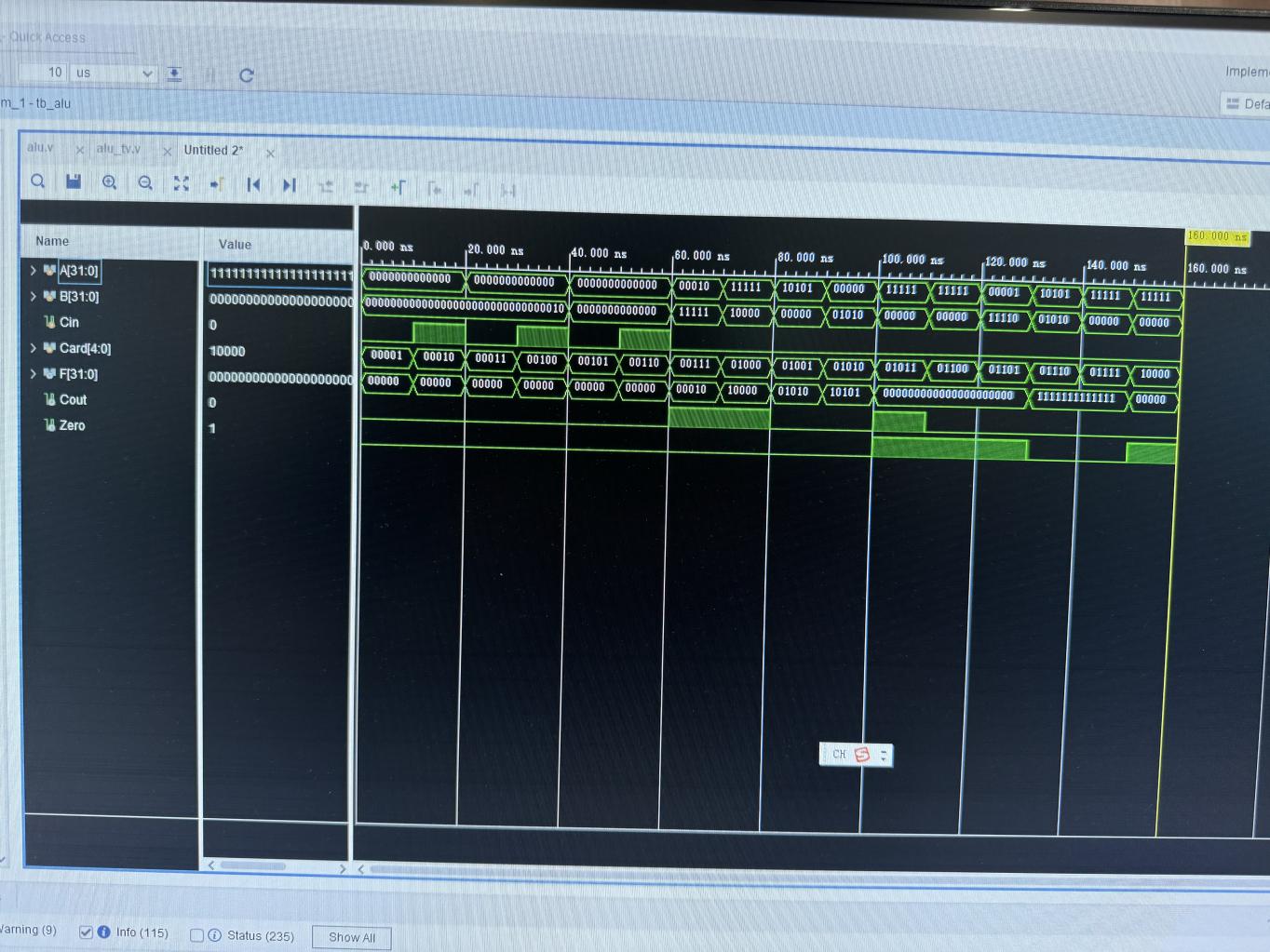
设计方案：

对每个不同的指令单独进行CU输出信号的讨论及设置

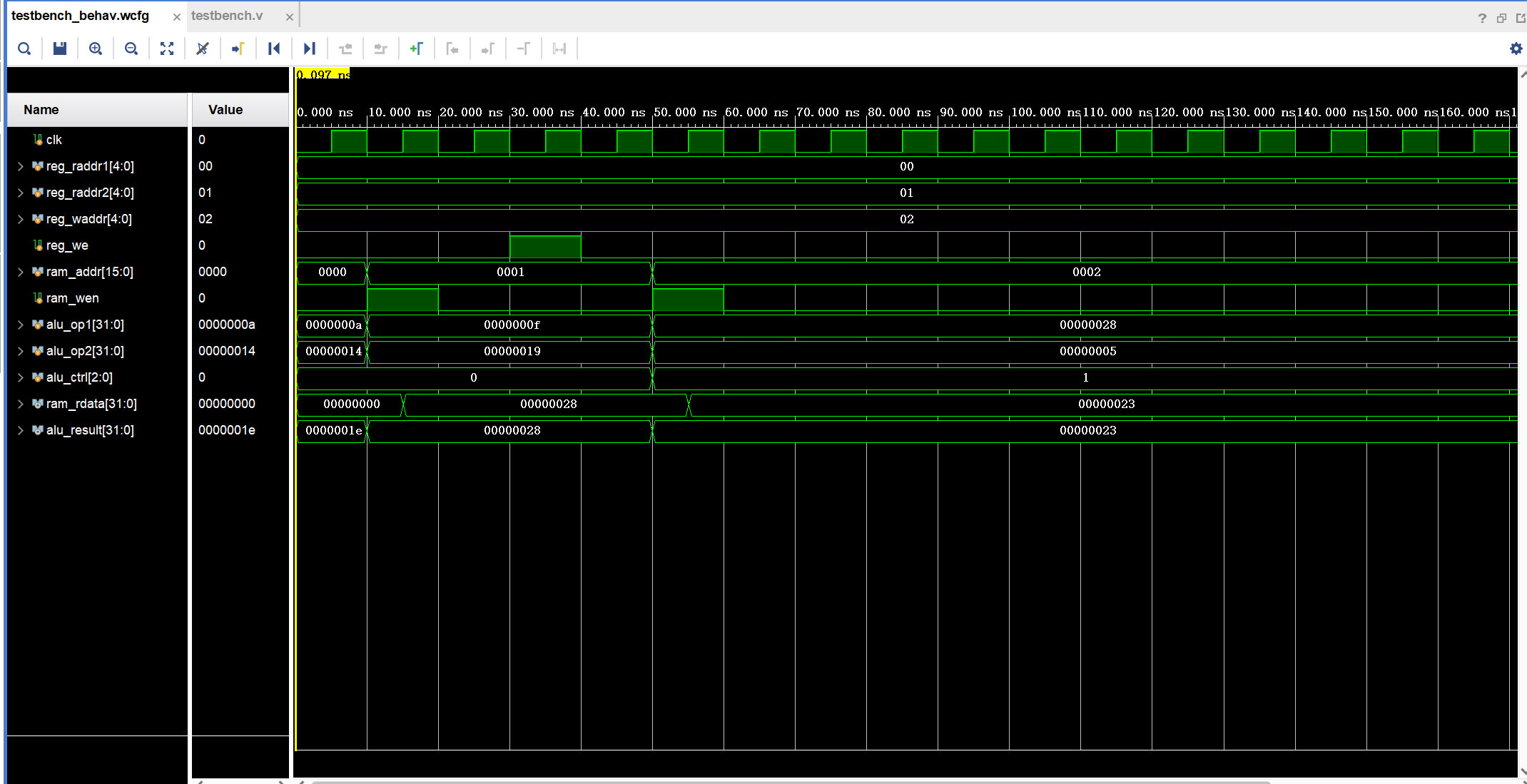
**五、**测试结果及实验分析

1．各模块的功能测试（每个模块作为一个部分，包括测试方案、测试过程、测试波形、波形分析等）

ALU：



Regfile：



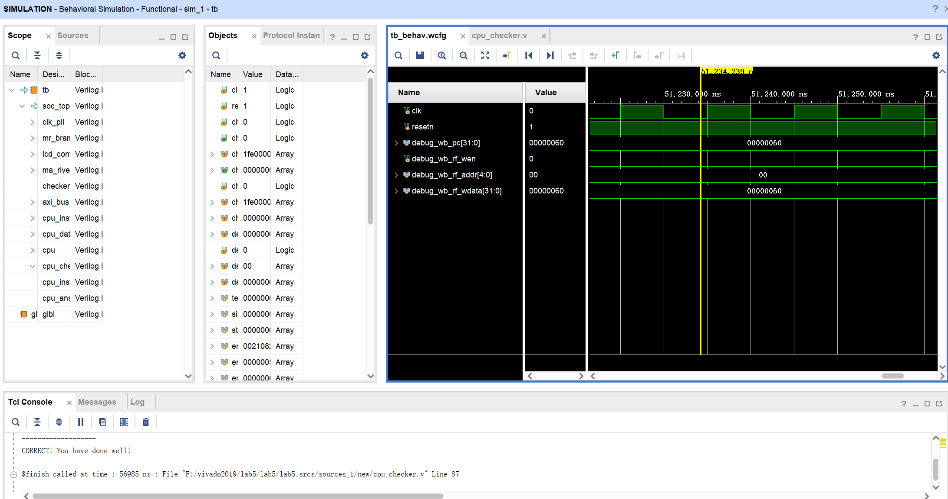
指令译码器：

**图形用户界面, 应用程序

描述已自动生成**

2 系统的功能测试（包括系统整体功能的测试方案、测试过程、测试波形、波形分析等）

测试波形及结果：



测试方案：根据群里提供的压缩包测试环境

六、实验总结

在该实验中，我设计并实现了一个非流水处理器，该处理器符合实验要求，能够执行包括ADD、SUB、AND、OR、XOR、LW、SW、J、MOVZ、SLL、CMP、BBT这十二种指令。整个设计过程涵盖了多个关键模块的实现，包括ALU（算术逻辑单元）、指令译码器、控制单元（CU）以及寄存器堆（Regfile）等，这些模块相互协作，共同完成指令的执行和处理器的整体运行。

通过此次实验，我对CPU内部数据通路中的数据流动有了更加直观和深入的理解。每条指令的执行不仅仅是操作数的计算，还包括指令的提取、解码、执行以及结果的存储等多个阶段。在这些过程中，不同模块之间的数据传递和信号控制显得尤为重要。例如，ALU负责对输入的操作数进行计算，指令译码器根据指令内容生成控制信号，控制单元协调各个模块的工作节奏，而寄存器堆则负责存储中间计算结果或数据。

实验中通过设计一个简单的处理器架构，使我深刻体会到指令执行的完整流程：从指令的提取、解码到实际的操作执行，最后将结果写回寄存器或内存。这一系列过程展示了计算机系统中如何高效地处理指令和数据，同时也体现了各个硬件模块在整个指令执行中的紧密配合。

此外，通过实际设计与实现该非流水处理器，我对CPU的构造有了更深入的理解，特别是关于控制信号的生成、数据通路的设计以及各模块之间的协作。我不仅仅掌握了如何实现基本的运算指令，还了解了跳转指令（J）、比较指令（CMP）等不同类型指令的执行方式。这些知识和经验为我今后设计更复杂的流水线处理器奠定了基础，特别是在处理指令重叠执行、数据冒险、控制冒险等问题时，这些基本概念将为我提供重要的理论和实践支持。

总的来说，本次实验不仅加强了我对非流水处理器设计的理解，还为进一步学习计算机体系结构中的流水线处理器设计提供了宝贵的经验。随着对更多复杂指令集和高级处理器架构的深入研究，我相信自己将能够应对更复杂的设计挑战并完成更高效的处理器系统。 **实验六 处理器下载（基于实验五）**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 预习成绩 | 实验成绩 | 报告成绩 | 总成绩 | 教师签字 |
|  |  |  |  |  |

一、实验目的

1．掌握 Vivado 集成开发环境

2．掌握Verilog语言

3．掌握FPGA编程方法及硬件调试手段

4．深刻理解处理器结构和计算机系统的整体工作原理

二、实验环境（实验设备、开发环境）

**Personal Computer：**

Processor：13th Gen Intel(R) Core(TM) 2.20 GHz

Installed RAM：16.0 GB (15.7 GB usable)

System type： 64-bit operating system, x64-based processor

**Xilinx FPGA**：xc7a200tfbg676-2

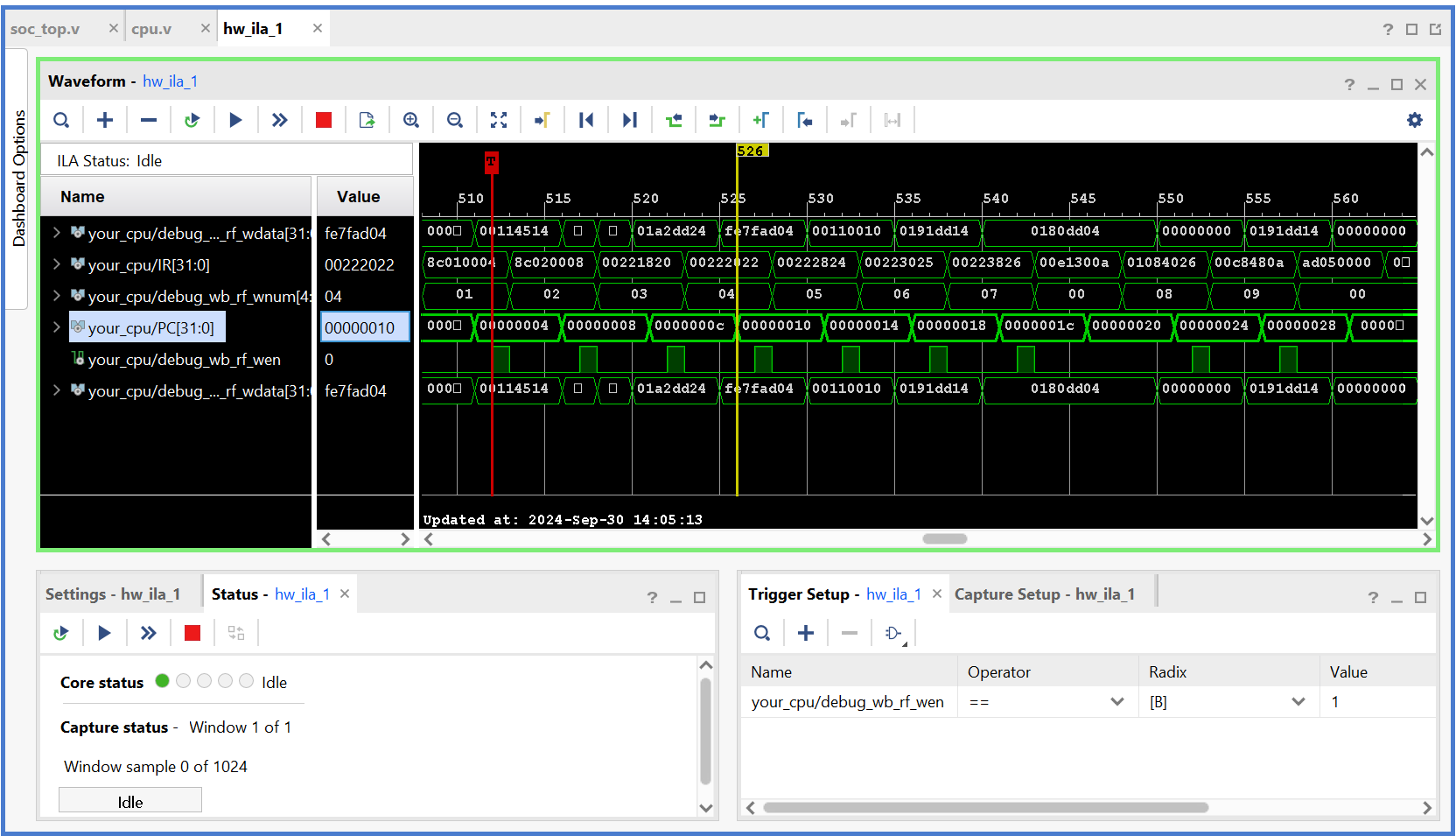
**Vivado Design Suite: version 2019.2**

三、设计思想（实验预习）（如空白不够，可自行加页）

1. CPU的管脚约束说明

四、下载测试结果

1. 实验台的测试结果



图形用户界面

描述已自动生成

2. 结果分析

（1）**your\_cpu/debug\_wb\_rf\_wdata[31:0]**：这是写入寄存器文件（Register File）数据通道的信号。可以看到，在不同的时钟周期下，这个信号的值发生了变化，例如在570周期为0228a280，在575周期为0000003e等。这些值代表CPU执行某些指令后写回到寄存器的结果。

（2）**your\_cpu/IR[31:0]**：这是指令寄存器（Instruction Register），表示CPU当前正在执行的指令。例如，在560周期，它的值为00222022，可以推断这条指令的具体操作类型。

（3）**your\_cpu/PC[31:0]**：程序计数器（Program Counter），每个时钟周期它的值都在递增，代表指令的执行顺序。可以看到，从570到610周期，PC的值从0x00000010递增到了0x0000005c。

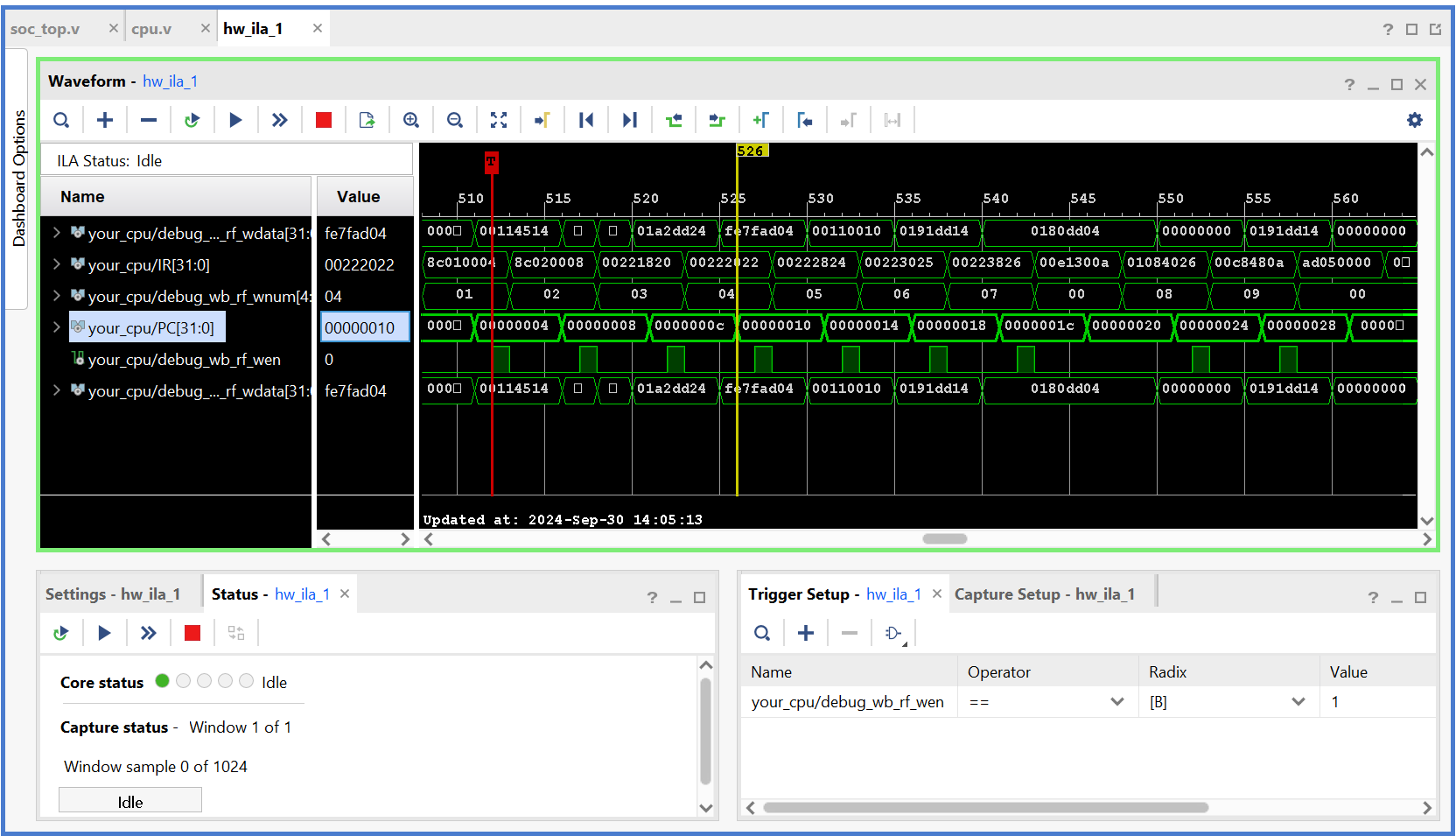
（4）**your\_cpu/debug\_wb\_rf\_wnum[4:0]**：这是用于标识写入寄存器编号的信号。在570周期，它的值为04，这意味着数据将写入寄存器编号为4的位置。

（5）**your\_cpu/debug\_wb\_rf\_wen**：这是一个控制信号，指示是否写入寄存器文件。在图中，debug\_wb\_rf\_wen等于1时，表示当前周期发生写回操作。例如，在570、580、590等周期，debug\_wb\_rf\_wen为1，这意味着写回发生。

总体分析：从波形图中可以看到，CPU在执行指令时每个时钟周期有指令的获取、执行、以及结果的写回。PC的变化表明指令流在顺序执行，debug\_wb\_rf\_wen为1时表示写回操作发生，debug\_wb\_rf\_wdata和debug\_wb\_rf\_wnum指示写入的数据和寄存器编号。

3. 上板在线调试结果（使用在线调试的波形图简要展示出你所选择的指令是如何执行的）





对debug\_wb\_rf\_wen=1进行抓取，红线对应的位置wdata=0x114514，结合实验五给的测试用例可知执行的为load指令，并且该指令执行PC指向下一条指令的地址。

五、实验总结

在实验五中，我已经实现了一个能够识别并执行十二种指令的简单CPU，并完成了其综合和仿真工作。实验六则进一步实现了该CPU的上板测试工作，除了进行上板测试外，还涉及到对CPU进行在线调试的操作。在这次实验中，我可以自行选择需要观察的CPU内部信号，特别是在调试过程中，我主要关注当`debug\_wb\_rf\_wen=1`时的信号抓取。

通过设置条件，在`debug\_wb\_rf\_wen=1`时进行特定信号的抓取，这意味着我选择在写回阶段（write-back）时提取相关的信号数据。通过抓取这些信号信息，并将其与测试用例进行对比，我可以准确地得出处理器当前正在执行的指令。这种信号的对比分析帮助我理解CPU内部各模块之间的运行状态，并且能够精确定位可能出现的问题。

此次实验让我深入学习了如何使用Vivado工具进行在线调试。通过Vivado的在线调试功能，我可以对运行中的处理器进行信号观察和捕捉，尤其是在硬件上实时测试时，这种调试方式极为重要。我掌握了如何选择CPU内部的关键信号进行跟踪，同时对特定值的信号变化进行有效抓取。

这种在线调试不仅提升了我对处理器工作状态的观察能力，还让我学会了如何通过实际抓取的信号，结合测试用例进行比对分析，帮助我更加精准地验证CPU的功能。这次实验为我提供了硬件调试的实战经验，也为未来进行更复杂的设计和调试打下了坚实的基础。

**实验六 处理器下载（基于LoongArch 处理器模板）**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 预习成绩 | 实验成绩 | 报告成绩 | 总成绩 | 教师签字 |
|  |  |  |  |  |

一、实验目的

1．掌握 Vivado 集成开发环境

2．掌握Verilog语言

3．掌握FPGA编程方法及硬件调试手段

4．深刻理解处理器结构和计算机系统的整体工作原理

二、实验环境（实验设备、开发环境）

三、设计思想（实验预习）（如空白不够，可自行加页）

1．LA32的指令格式

2. LA32处理器模板结构

四、实验设计

补全的Verilog实现代码：

（提交你修改过的代码。一种清晰而便捷的方式是使用 git diff 对比原文件和修改过的文件。）

五、测试与分析

1. 仿真测试波形与测试结果

2. 下载到实验台的测试结果

实验结果分析：

3. 抓取debug信号的实验结果

1. 给 `pc` 寄存器和其他你认为需要的信号添加 debug 标志。

2. 添加抓取条件：当 `pc` 寄存器的值为 `1c000100` 时。

3. 在 `test.s` 文件中找到这条指令，说明这条指令所代表的意义。

4. 抓取另一个你感兴趣的信号，说明CPU此时正在执行什么指令。

5. 上板调试抓取 debug 信号的截图。

（附上你认为需要的截图和代码，以说明你完成相应的任务）

六、实验总结