实验二

**北京理工大学 计算机学院**

# 确定指令格式和编码

这里通过查阅RISCV文档的指令格式与编码，得知RISCV整数指令分为五类：寄存器型、立即数型、存储型、分支型、跳转型。每条指令均为32位。

为了完成实验要求的排序测试，这里共设计了18条指令，指令如下：

表格 1指令功能与数目

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 序号 | 操作码 | 3位功能码 | 7位功能码 | 助记符 | 功能 | 描述 |
| 1 | 011\_0011 | 000 | 000\_0000 | add | R[rd] = R[rs1] + R[rs2]  PC = PC + 1 | 寄存器加法 |
| 2 | 011\_0011 | 000 | 001\_0100 | sub | R[rd] = R[rs1]+R[rs2]  PC = PC + 1 | 寄存器减法 |
| 3 | 011\_0011 | 100 | 000\_0000 | xor | R[rd] = R[rs1]^R[rs2]  PC = PC + 1 | 寄存器异或 |
| 4 | 011\_0011 | 110 | 000\_0000 | or | R[rd] = R[rs1]|R[rs2]  PC = PC + 1 | 寄存器或 |
| 5 | 011\_0011 | 111 | 000\_0000 | and | R[rd] = R[rs1]&R[rs2]  PC = PC + 1 | 寄存器与 |
| 6 | 011\_0011 | 010 | 000\_0000 | slt | R[rd] = (R[rs1] < R[rs2]) ? 1 : 0  PC = PC + 1 | 小于置位 |
| 7 | 001\_0011 | 000 | -- | Addi | R[rd] = R[rs1] + imm  PC = PC + 1 | 常数加法 |
| 8 | 001\_0011 | 100 | -- | xori | R[rd] = R[rs1] ^ imm  PC = PC + 1 | 常数异或 |
| 9 | 001\_0011 | 110 | -- | ori | R[rd] = R[rs1] | imm  PC = PC + 1 | 常数或 |
| 10 | 001\_0011 | 111 | -- | andi | R[rd] = R[rs1] & imm  PC = PC + 1 | 常数与 |
| 11 | 001\_0011 | 010 | -- | slti | R[rd] = (R[rs1] < imm) ? 1 : 0  PC = PC + 1 | 常数小于置位 |
| 12 | 000\_0011 | 010 | -- | lw | R[rd] = M[R[rs1] + imm]  PC = PC + 1 | 从内存加载字 |
| 13 | 010\_0011 | 010 | -- | sw | M[R[rs1] + imm] = R[rs2] | 向内存存储字 |
| 14 | 110\_0011 | 000 | -- | beq | If(R[rs1] == R[rs2]) PC = PC + imm | 相等分支跳转 |
| 15 | 110\_0011 | 001 | -- | bne | If(R[rs1] != R[rs2]) PC = PC + imm | 不等分支跳转 |
| 16 | 110\_0011 | 100 | -- | blt | If(R[rs1] < R[rs2]) PC = PC + imm | 小于跳转 |
| 17 | 110\_0011 | 101 | -- | bge | If(R[rs1] >= R[rs2]) PC = PC + imm | 大于等于跳转 |
| 18 | 110\_1111 | -- | -- | jal | PC = PC + imm | 无条件跳转 |
| 指令总数目 | | | | | 18 | |

# CPU数据通路

CPU数据通路总体结构大体参照任务书中所给出的。这里先给出每个模块的设计，然后将模块连接起来形成单周期CPU。

## 寄存器堆

这里实现了两读一写寄存器堆。原理较为简单，Verilog代码如下：

module regfile(

input clk,

input rst,

input write\_enable, // 写使能

input [4:0] ra\_addr, // 读寄存器A地址

input [4:0] rb\_addr, // 读寄存器B地址

input [4:0] rw\_addr, // 写寄存器地址

input [31:0] rw\_data, // 写寄存器时数据

output [31:0] ra\_data, // 读寄存器A时数据

output [31:0] rb\_data // 读寄存器B时数据

);

// 寄存器堆

reg [31:0] regs[31:0];

integer i;

always @(posedge clk or negedge rst) begin

if (!rst) begin // 复位，将每个寄存器置为0

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

regs[i] <= 32'b0;

end

else if (write\_enable) // 写寄存器

regs[rw\_addr] = rw\_data;

end

// 读寄存器

assign ra\_data = regs[ra\_addr];

assign rb\_data = regs[rb\_addr];

endmodule

仿真代码如下：

module regfile\_sim();

reg clk;

reg rst;

reg write\_enable;

reg [4:0] ra\_addr;

reg [4:0] rb\_addr;

reg [4:0] rw\_addr;

reg [31:0] rw\_data;

wire [31:0] ra\_data;

wire [31:0] rb\_data;

regfile regfile\_sim(

.clk(clk),

.rst(rst),

.write\_enable(write\_enable),

.ra\_addr(ra\_addr),

.rb\_addr(rb\_addr),

.rw\_addr(rw\_addr),

.ra\_data(ra\_data),

.rb\_data(rb\_data),

.rw\_data(rw\_data)

);

initial begin

clk = 1'b0; #10;

rst = 1'b0; #10;

rst = 1'b1; #10;

rw\_addr = 5'b00000; rw\_data = 32'h0000\_1234; write\_enable = 1'b1; #10;

rw\_addr = 5'b00001; rw\_data = 32'h0000\_4321; write\_enable = 1'b1; #10;

write\_enable = 1'b0; #10;

ra\_addr = 5'b00000; rb\_addr = 5'b00001; #10;

$finish;

end

always #5 clk = ~clk;

endmodule

电视游戏的萤幕截图

描述已自动生成仿真波形如下，可以看到结果符合预期。

## 算数逻辑单元ALU

这里根据实验一的代码进行稍加改动，主要是添加了时钟信号和置位信号。并且为了方便进行比较判断，增加了新的选择信号，删除了标志位。Verilog代码如下：

module alu(

input clk,

input rst,

input [31:0] num1, // 输入的第一个数

input [31:0] num2, // 输入的第二个数

input [3:0] select, // 输入的选择信号，用于选择运算类型

output [31:0] res, // 输出的运算结果

output flag // 根据输入的选择信号，判断条件是否成立

);

reg [31:0] res\_reg;

reg flag\_reg;

// 计算过程

always@(posedge clk or negedge rst) begin

if (!rst) begin

res\_reg = 32'b0;

flag\_reg = 1'b0;

end

else begin // 没有复位时进行的操作

case(select)

4'b0000: begin

res\_reg = num1 + num2;

flag\_reg = 1'b0;

end

4'b0001: begin

res\_reg = num1 - num2;

flag\_reg = 1'b0;

end

4'b0010: begin

res\_reg = num1 & num2;

flag\_reg = 1'b0;

end

4'b0011: begin

res\_reg = num1 | num2;

flag\_reg = 1'b0;

end

4'b0100: begin

res\_reg = num1 ^ num2;

flag\_reg = 1'b0;

end

4'b0101: begin

res\_reg = ~(num1 | num2);

flag\_reg = 1'b0;

end

4'b0110: begin

res\_reg = num1 << num2;

flag\_reg = 1'b0;

end

4'b0111: begin

res\_reg = num1 >> num2;

flag\_reg = 1'b0;

end

4'b1000: begin

flag\_reg = num1 < num2;

end

4'b1001: begin

flag\_reg = num1 >= num2;

end

4'b1010: begin

flag\_reg = num1 == num2;

end

4'b1011: begin

flag\_reg = num1 != num2;

end

4'b1111: begin

flag\_reg = 1'b1;

end

endcase

end

end

// assign res = double\_sign\_res[31:0];

assign res = res\_reg;

assign flag = flag\_reg;

endmodule

仿真代码如下：

module alu\_sim();

reg clk;

reg rst;

reg [31:0] num1;

reg [31:0] num2;

reg [3:0] select;

wire [31:0] res;

wire flag;

alu alu\_sim(

.clk(clk),

.rst(rst),

.num1(num1),

.num2(num2),

.select(select),

.flag(flag),

.res(res)

);

initial begin

clk = 1'b0; #10;

rst = 1'b0; #10;

rst = 1'b1; #10;

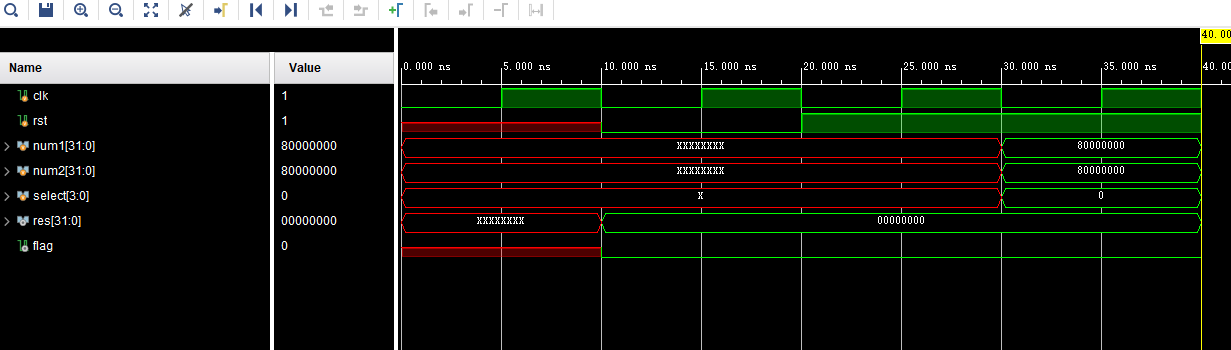
select = 4'b0000; num1 = 32'h8000\_0000; num2 = 32'h8000\_0000; # 10;

$finish;

end

always #5 clk = ~clk;

endmodule

这里对加法进行了测试。仿真波形如下，可以看到结果符合预期。

## 数据内存

这部分和寄存器堆部分类似，但是由于内存一般是字节寻址的，所以这里存储单元设计为1字节。为了方便读写数据，这里内存访问单元为4字节。Verilog代码如下：

module data\_memory(

input clk,

input rst,

input write\_enable, // 写使能

input [31:0] addr, // 读写内存地址

input [31:0] write\_data, // 写入的数据

output [31:0] read\_data // 读出的数据

);

// 内存，总大小为1KB

reg [7:0] ram[1023:0];

integer i;

always @(posedge clk or negedge rst) begin

if (!rst) begin // 复位，将每个字节置为0

for(i = 0; i < 1024; i = i + 1)

ram[i] = 32'b0;

$readmemh("mem.txt", ram);

end

else if (write\_enable) begin // 写内存

ram[addr] <= write\_data[7:0];

ram[addr+1] <= write\_data[15:8];

ram[addr+2] <= write\_data[23:16];

ram[addr+3] <= write\_data[31:24];

end

end

// 读内存

assign read\_data = {ram[addr+3], ram[addr+2], ram[addr+1], ram[addr]};

endmodule

仿真代码如下：

module data\_memory\_sim();

reg clk;

reg rst;

reg write\_enable;

reg [31:0] addr;

reg [31:0] write\_data;

wire [31:0] read\_data;

data\_memory data\_memory\_sim(

.clk(clk),

.rst(rst),

.write\_enable(write\_enable),

.addr(addr),

.write\_data(write\_data),

.read\_data(read\_data)

);

initial begin

clk = 1'b0; #10;

rst = 1'b0; #10;

rst = 1'b1; #10;

addr = 32'h00000\_0000; write\_data = 32'h0000\_1234; write\_enable = 1'b1; #10;

write\_enable = 1'b0; #10;

addr = 32'b00000\_0000;;

$finish;

end

always #5 clk = ~clk;

endmodule

日程表

中度可信度描述已自动生成仿真结果如下，可以看到符合预期。

## 取指单元

该部分的主要功能是解析存储在指令内存中的每条指令，获取其操作码、3位功能码、7位功能码、目的寄存器编号、源寄存器1编号、源寄存器2编号、立即数。

module ifu(

input clk,

input rst,

input [31:0] pc,

output [6:0] op\_code, // 操作码

output [2:0] func3, // 3位操作码

output [6:0] func7, // 7位操作码

output [4:0] rd, // 目的寄存器

output [4:0] rs1, // 源寄存器1

output [4:0] rs2, // 源寄存器2

output [31:0] imm // 扩展到32的立即数

);

// 指令寄存器

reg [31:0] ir;

// 存放指令的内存

reg [31:0] iram[63:0];

reg [6:0] op\_code\_reg;

reg [2:0] func3\_reg;

reg [6:0] func7\_reg;

reg [4:0] rd\_reg;

reg [4:0] rs1\_reg;

reg [4:0] rs2\_reg;

reg signed [31:0] imm\_reg;

integer i;

always @(posedge clk or negedge rst) begin

if (!rst) begin // 复位，将指令读入指令存储器

for(i = 0; i < 64; i = i + 1)

iram[i] = 32'b0;

$readmemh("instruction.txt", iram);

ir = 32'b0;

end

else begin

ir = iram[pc];

op\_code\_reg = ir[6:0];

// 根据操作码获取指令的各个部分

case(op\_code\_reg)

7'b011\_0011: begin // 寄存器型指令

imm\_reg = 32'b0;

func3\_reg = ir[14:12];

func7\_reg = ir[31:25];

rs1\_reg = ir[19:15];

rs2\_reg = ir[24:20];

rd\_reg = ir[11:7];

end

7'b001\_0011: begin // 立即数型指令

imm\_reg = {{20{ir[31]}}, ir[31:20]};

func3\_reg = ir[14:12];

func7\_reg = 7'b0;

rs1\_reg = ir[19:15];

rs2\_reg = 5'b0;

rd\_reg = ir[11:7];

end

7'b000\_0011: begin // lw指令

imm\_reg = {{20{ir[31]}}, ir[31:20]};

func3\_reg = ir[14:12];

func7\_reg = 7'b0;

rs1\_reg = ir[19:15];

rs2\_reg = 5'b0;

rd\_reg = ir[11:7];

end

7'b010\_0011: begin // sw指令

imm\_reg = {{20{ir[31]}}, ir[31:25], ir[11:7]};

func3\_reg = ir[14:12];

func7\_reg = 7'b0;

rs1\_reg = ir[19:15];

rs2\_reg = ir[24:20];

rd\_reg = 5'b0;

end

7'b110\_0011: begin // 分支指令

imm\_reg = {{20{ir[31]}}, ir[31:25], ir[11:7]};

func3\_reg = ir[14:12];

func7\_reg = 7'b0;

rs1\_reg = ir[19:15];

rs2\_reg = ir[24:20];

rd\_reg = 5'b0;

end

7'b110\_1111: begin // 跳转指令

imm\_reg = {{12{ir[31]}}, ir[31:12]};

func3\_reg = 3'b0;

func7\_reg = 7'b0;

rs1\_reg = 5'b0;

rs2\_reg = 5'b0;

rd\_reg = ir[11:7];

end

endcase

end

end

assign op\_code = op\_code\_reg;

assign imm = imm\_reg;

assign func3 = func3\_reg;

assign func7 = func7\_reg;

assign rs1 = rs1\_reg;

assign rs2 = rs2\_reg;

assign rd = rd\_reg;

endmodule

这里在置位时向指令内存中写入数据，然后进行仿真，仿真代码如下：

module ifu\_sim();

reg clk;

reg rst;

reg [31:0] pc;

wire [6:0] op\_code;

wire [2:0] func3;

wire [6:0] func7;

wire [4:0] rd;

wire [4:0] rs1;

wire [4:0] rs2;

wire [31:0] imm;

ifu ifu\_sim(

.clk(clk),

.rst(rst),

.pc(pc),

.op\_code(op\_code),

.func3(func3),

.func7(func7),

.rd(rd),

.rs1(rs1),

.rs2(rs2),

.imm(imm)

);

initial begin

clk = 1'b0; #10;

rst = 1'b0; #10;

rst = 1'b1; #10;

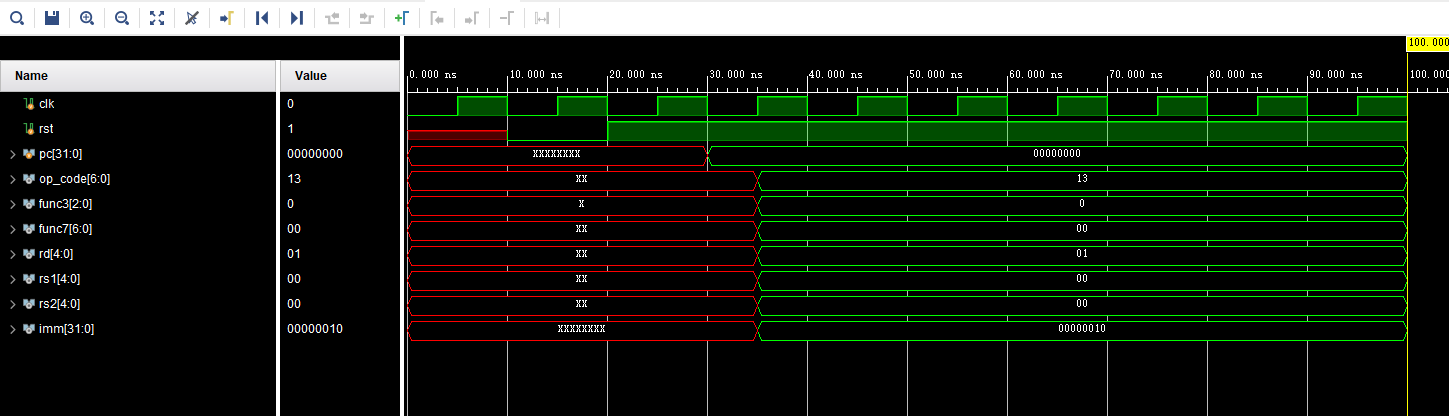
pc = 32'b0; #30;

$finish;

end

always #5 clk = ~clk;

endmodule

仿真结果如下，指令解析结果符合预期。

## PC

这部分的主要功能是形成下一条指令的地址。为了方便起见，这里使用了ALU模块的输出flag来确定下条指令是相对偏移还是继续加一。功能较为简单，这里没有进行仿真。其Verilog代码如下：

module pc(

input clk,

input rst,

input flag,

input signed [31:0] imm,

output [31:0] new\_pc

);

reg [31:0] pc\_value;

always @(posedge clk or negedge rst) begin

if(!rst)

pc\_value = 32'b0;

else begin

// 改变pc

if (flag) begin

pc\_value = pc\_value + $signed(imm >>> 2);

end

else

pc\_value = pc\_value + 32'b1;

$display("pc: %d", pc\_value);

end

end

assign new\_pc = pc\_value;

endmodule

## 组合各模块，形成单周期CPU

这里设计的单周期CPU将每条指令执行分为四个阶段：

* 解析指令：调用ifu模块，解析每条指令的各个字段
* 进行计算：调用alu模块，进行计算或判断
* 写入寄存器堆或数据内存
* 形成下一条指令的PC值

这里由于寄存器和数据内存的读取是通过组合电路实现的，所以没有考虑在上述阶段中。

module cpu(

input clk, // 时钟信号

input rst // 置位信号

);

wire reg\_write\_enable;

wire [4:0] ra\_addr;

wire [4:0] rb\_addr;

wire [4:0] rw\_addr;

wire [31:0] rw\_data;

wire [31:0] ra\_data;

wire [31:0] rb\_data;

wire mem\_write\_enable;

wire [31:0] mem\_read\_data;

wire [31:0] alu\_num2;

wire [3:0] alu\_select;

wire [31:0] alu\_res;

wire flag;

wire [6:0] op\_code;

wire [2:0] func3;

wire [6:0] func7;

wire [31:0] imm;

wire num2\_src, mem\_to\_reg;

wire [31:0] pc\_value;

reg regfile\_clk, data\_memory\_clk, alu\_clk, ifu\_clk, pc\_clk;

integer i = 0;

always @(posedge clk) begin

if (rst == 1) begin

if (i == 0) begin

ifu\_clk = 1;

i = i + 1;

end

else if (i == 1) begin

alu\_clk = 1;

i = i + 1;

end

else if (i == 2) begin

regfile\_clk = 1;

data\_memory\_clk = 1;

i = i + 1;

end

else if (i == 3) begin

pc\_clk = 1;

i = 0;

end

end

end

always @(negedge clk) begin

regfile\_clk = 0;

data\_memory\_clk = 0;

alu\_clk = 0;

ifu\_clk = 0;

pc\_clk = 0;

end

// 实例化各模块

// 寄存器堆

regfile RegFile(

.clk(regfile\_clk),

.rst(rst),

.write\_enable(reg\_write\_enable),

.ra\_addr(ra\_addr),

.rb\_addr(rb\_addr),

.rw\_addr(rw\_addr),

.rw\_data(rw\_data),

.ra\_data(ra\_data),

.rb\_data(rb\_data)

);

// 数据内存

data\_memory DataMemory(

.clk(data\_memory\_clk),

.rst(rst),

.write\_enable(mem\_write\_enable),

.addr(alu\_res),

.write\_data(rb\_data),

.read\_data(mem\_read\_data)

);

// 算数逻辑单元

alu ALU(

.clk(alu\_clk),

.rst(rst),

.num1(ra\_data),

.num2(alu\_num2),

.select(alu\_select),

.res(alu\_res),

.flag(flag)

);

// 取指单元

ifu IFU(

.clk(ifu\_clk),

.rst(rst),

.pc(pc\_value),

// .flag(flag),

.op\_code(op\_code),

.func3(func3),

.func7(func7),

.rd(rw\_addr),

.rs1(ra\_addr),

.rs2(rb\_addr),

.imm(imm)

);

// 控制单元

cu CU(

.op\_code(op\_code),

.func3(func3),

.func7(func7),

.num2\_src(num2\_src),

.mem\_to\_reg(mem\_to\_reg),

.reg\_write\_enable(reg\_write\_enable),

.mem\_write\_enable(mem\_write\_enable),

.alu\_select(alu\_select)

);

// PC

pc PC(

.clk(pc\_clk),

.rst(rst),

.flag(flag),

.imm(imm),

.new\_pc(pc\_value)

);

assign alu\_num2 = num2\_src ? imm : rb\_data;

assign rw\_data = mem\_to\_reg ? mem\_read\_data : alu\_res;

endmodule

图示

描述已自动生成模块连接图如下：

# 控制信号表

这里根据每条指令的功能含义填写控制信号表。

* Num2\_src：ALU的num2选择信号，为1表示从立即数输入，为0表示从寄存器堆rb输出输入。
* Mem\_to\_reg：寄存器堆写入数据的选择信号，为1表示从内存写入寄存器，为0表示从ALU输出写入寄存器。
* Reg\_write\_enable：寄存器堆写入使能信号。
* Mem\_write\_enable：数据内存写入使能信号。
* Alu\_select[3:0]：ALU的计算功能选择信号，为0表示加法，为1表示减法，为2表示按位与，为3表示按位或，为4表示按位异或，为5表示按位或非，为6表示左移，为7表示右移，为8表示小于比较，为9表示大于等于比较，为10表示等于比较，为11表示不等于比较，为15表示flag恒为1。

表格 2 CPU模型控制信号列表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 序号 | 操作码 | 3位功能码 | 7位功能码 | 助记符 | num2\_src | mem\_to\_reg | reg\_write\_enable | mem\_write\_enable | alu\_select[3:0] |
| 1 | 011\_0011 | 000 | 000\_0000 | add | 0 | 0 | 1 | 0 | 0000 |
| 2 | 011\_0011 | 000 | 001\_0100 | sub | 0 | 0 | 1 | 0 | 0001 |
| 3 | 011\_0011 | 100 | 000\_0000 | xor | 0 | 0 | 1 | 0 | 0100 |
| 4 | 011\_0011 | 110 | 000\_0000 | or | 0 | 0 | 1 | 0 | 0011 |
| 5 | 011\_0011 | 111 | 000\_0000 | and | 0 | 0 | 1 | 0 | 0010 |
| 6 | 011\_0011 | 010 | 000\_0000 | slt | 0 | 0 | 1 | 0 | 1000 |
| 7 | 001\_0011 | 000 | -- | addi | 1 | 0 | 1 | 0 | 0000 |
| 8 | 001\_0011 | 100 | -- | xori | 1 | 0 | 1 | 0 | 0100 |
| 9 | 001\_0011 | 110 | -- | ori | 1 | 0 | 1 | 0 | 0011 |
| 10 | 001\_0011 | 111 | -- | andi | 1 | 0 | 1 | 0 | 0010 |
| 11 | 001\_0011 | 010 | -- | slti | 1 | 0 | 1 | 0 | 1000 |
| 12 | 000\_0011 | 010 | -- | lw | 1 | 1 | 1 | 0 | 0000 |
| 13 | 010\_0011 | 010 | -- | sw | 1 | 0 | 0 | 1 | 0000 |
| 14 | 110\_0011 | 000 | -- | beq | 0 | 0 | 0 | 0 | 1010 |
| 15 | 110\_0011 | 001 | -- | bne | 0 | 0 | 0 | 0 | 1011 |
| 16 | 110\_0011 | 100 | -- | blt | 0 | 0 | 0 | 0 | 1000 |
| 17 | 110\_0011 | 101 | -- | bge | 0 | 0 | 0 | 0 | 1001 |
| 18 | 110\_1111 | -- | -- | jal | 0 | 0 | 0 | 0 | 1111 |

# 逻辑表达式

实验中采用了组合逻辑硬链接方式，其主要部分位于cu（控制单元）模块中。该模块的主要功能是根据每条指令操作码、3位功能码、7位功能码生成相应的控制信号，控制信号包括ALU的num2来源的选择信号、寄存器堆写入数据来源的选择信号、寄存器堆写入使能、数据内存写入使能、ALU计算功能的选择信号，Verilog代码如下：

module cu(

input [6:0] op\_code, // 操作码

input [2:0] func3, // 3位功能码

input [6:0] func7, // 7位功能码

output num2\_src, // ALU的num2选择信号，为1表示从立即数输入，为0表示从寄存器堆rb输出输入

output mem\_to\_reg, // 寄存器堆写入数据的选择信号，为1表示从内存写入寄存器，为0表示从alu输出写入寄存器

output reg\_write\_enable,// 寄存器堆写入使能

output mem\_write\_enable,// 数据内存写入使能

output [3:0] alu\_select // ALU的计算功能选择信号

);

assign num2\_src = ((op\_code == 7'b000\_0011) && (func3 == 3'b010))

|| ((op\_code == 7'b010\_0011) && (func3 == 3'b010))

|| (op\_code == 7'b001\_0011);

assign mem\_to\_reg = (op\_code == 7'b000\_0011) && (func3 == 3'b010);

assign reg\_write\_enable = ((op\_code == 7'b000\_0011) && (func3 == 3'b010))

|| (op\_code == 7'b011\_0011)

|| (op\_code == 7'b001\_0011);

assign mem\_write\_enable = ((op\_code == 7'b010\_0011) && (func3 == 3'b010));

assign alu\_select = (((op\_code == 7'b011\_0011) && (func3 == 3'b000) && (func7 == 7'b000\_0000))

|| ((op\_code == 7'b001\_0011) && (func3 == 3'b000))

|| ((op\_code == 7'b000\_0011) && (func3 == 3'b010))

|| ((op\_code == 7'b010\_0011) && (func3 == 3'b010))) ? 4'b0000 : // 加法

((op\_code == 7'b011\_0011) && (func3 == 3'b000) && (func7 == 7'b001\_0100)) ? 4'b0001 : // 减法

(((op\_code == 7'b011\_0011) && (func3 == 3'b111) && (func7 == 7'b000\_0000))

|| ((op\_code == 7'b001\_0011) && (func3 == 3'b111))) ? 4'b0010 : // 与

(((op\_code == 7'b011\_0011) && (func3 == 3'b110) && (func7 == 7'b000\_0000))

|| ((op\_code == 7'b001\_0011) && (func3 == 3'b110))) ? 4'b0011 : // 或

(((op\_code == 7'b011\_0011) && (func3 == 3'b100) && (func7 == 7'b000\_0000))

|| ((op\_code == 7'b001\_0011) && (func3 == 3'b100))) ? 4'b0100 : // 异或

(((op\_code == 7'b011\_0011) && (func3 == 3'b010) && (func7 == 7'b000\_0000))

|| ((op\_code == 7'b001\_0011) && (func3 == 3'b010))

|| ((op\_code == 7'b110\_0011) && (func3 == 3'b100))) ? 4'b1000 : // 小于

((op\_code == 7'b110\_0011) && (func3 == 3'b101)) ? 4'b1001 : // 大于等于

((op\_code == 7'b110\_0011) && (func3 == 3'b000)) ? 4'b1010 : // 等于

((op\_code == 7'b110\_0011) && (func3 == 3'b001)) ? 4'b1011 : // 不等于

(op\_code == 7'b110\_1111) ? 4'b1111 : // 无条件跳转

4'b0000; // 无效编码

endmodule

由于这里主要是硬连逻辑，所以没有进行仿真测试。

# 测试程序

这里使用冒泡排序来进行含有5个元素数组的排序。数组元素存放在DataMemory中，并从0x0开始存放，每个元素4字节。

程序流程图如下：

图示

描述已自动生成 根据流程图编写代码如下：

表格 3

|  |
| --- |
| RISC-V汇编源代码粘贴处： |
| *# x1寄存器存放常数 ((5-1)\*4)*      addi x1, x0, 16  *# x2 寄存器存放数组起始内存地址，这里为 0*      addi x2, x0, 0  *# x3寄存器存放外层循环变量 i = 20*      addi x3, x1, 0  loop:  *# x4 寄存器存放内层循坏变量 j = 0*      addi x4, x0, 0  inner\_loop:  *# x5 寄存器存放当前访问内存数组元素的地址*      add x5, x2, x4  *# x6 存放当前访问内存数组元素值*      lw x6, 0(x5)  *# x7 存放下一个内存数组元素值*      lw x7, 4(x5)      blt x6, x7, inner\_loop\_end  *# 交换两个相邻数组元素*      sw x7, 0(x5)      sw x6, 4(x5)  inner\_loop\_end: *# 内循环结束处理，j+4*      addi x4, x4, 4      blt x4, x3, inner\_loop  loop\_end: *# 外循环结束处理，i-4*  addi x3, x3, -4  bge x3, x0, loop |

图形用户界面, 文本, 应用程序

描述已自动生成 将上述汇编代码放入RARS中，进行编译，得到的机器码如下：

根据机器码填写指令与对应的机器码表格。

表格 4指令与对应的机器码

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 序号 | 指令格式 | 具体指令 | Func7 | Rs2 | Rs1 | Func3 | Rd | Op | Imm |
| 1 | Addi rd, rs1, imm | Addi x1, x0, 16 | -- | -- | 00000 | 000 | 00001 | 001\_0011 | 16 |
| 2 | Addi rd, rs1, imm | Addi x2, x0, 0 | -- | -- | 00000 | 000 | 00010 | 001\_0011 | 0 |
| 3 | Addi rd, rs1, imm | Addi x3, x1, 0 | -- | -- | 00001 | 000 | 00011 | 001\_0011 | 0 |
| 4 | Addi rd, rs1, imm | Addi x4, x0, 0 | -- | -- | 00000 | 000 | 00100 | 001\_0011 | 0 |
| 5 | Add rd, rs1, rs2 | Add x5, x2, x4 | 000\_0000 | 00010 | 00010 | 000 | 00101 | 011\_0011 | -- |
| 6 | Lw rd, imm(rs1) | Lw x6, 0(x5) | -- | -- | 00101 | 010 | 00110 | 000\_0011 | 0 |
| 7 | Lw rd, imm(rs1) | Lw x7, 4(x5) | -- | -- | 00101 | 010 | 00111 | 000\_0011 | 4 |
| 8 | Blt rs1, rs2, imm | Blt x6, x7, 0x0000\_000c | -- | 00111 | 00101 | 100 | -- | 110\_0011 | 0x0000\_000c |
| 9 | Sw rs2, imm(rs1) | Sw x7, 0(x5) | -- | 00111 | 00101 | 010 | -- | 110\_0011 | 0 |
| 10 | Sw rs2, imm(rs1) | Sw x6, 4(x5) | -- | 00110 | 00101 | 010 | -- | 110\_0011 | 4 |
| 11 | Addi rd, rs1, imm | Addi x4, x4, 4 | -- | -- | 00100 | 000 | 00100 | 001\_0011 | 0 |
| 12 | Blt rs1, rs2, imm | Blt x4, x3, 0xffff\_ffe4 | -- | 00011 | 00100 | 100 | -- | 110\_0011 | 0xffff\_ffe4 |
| 13 | Addi rd, rs1, imm | Addi x3, x3, 0xffff\_fffc | -- | -- | 00011 | 000 | 00011 | 001\_0011 | 0xffff\_fffc |
| 14 | Bge rs1, rs2, imm | Bge x3, x0, 0xffff\_ffd8 | -- | 00000 | 00011 | 101 | -- | 110\_0011 | 0xffff\_ffd8 |

# Vivado仿真和RARS执行比较

## RARS软件配置

* 启动：java -jar rars1\_6.jar
* 图形用户界面, 文本, 应用程序

  描述已自动生成修改内存配置：由于我们要访问和修改内存地址为0x0处附近数据，所以要修改配置。方法为Settings -> Memory Configuration
* 向内存输入数据：在Data Segment窗口输入数据。

## 表格 描述已自动生成Vivado指令和数据准备

* 通过RARS将汇编文件转换为十六进制格式的机器码。并放在“cpu.sim\sim\_1\behav\xsim\instruction.txt”文件夹下以方便使用相对路径。
* 图形用户界面, 应用程序

  描述已自动生成在ifu模块中使用“$readmemh(“instruction.txt”, iram)”向指令内存写入指令。
* 在data\_memory模块中使用“$readmemh(“mem.txt”, iram)”向数据内存中写入要排序的数据。这里使用的数据为5、4、3、2、1，进行从小到大的排序。

## 仿真比较

由于执行的指令数目较多，这里仅对执行的前20条指令进行列出。

表格 5 Vivado仿真结果与RARS仿真结果比较

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 序号 | 指令序列 | Vivado仿真结果 | | | | | | | | | | | |
| x1 | x2 | x3 | x4 | x5 | x6 | x7 | m0 | m1 | m2 | m3 | m4 |
| 1 | addi x1, x0, 16 | 0x10 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 5 | 4 | 3 | 2 | 1 |
| 2 | addi x2, x0, 0 | 0x10 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 5 | 4 | 3 | 2 | 1 |
| 3 | addi x3, x1, 0 | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 5 | 4 | 3 | 2 | 1 |
| 4 | addi x4, x0, 0 | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 5 | 4 | 3 | 2 | 1 |
| 5 | add x5, x2, x4 | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 5 | 4 | 3 | 2 | 1 |
| 6 | lw x6, 0(x5) | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x5 | 0x0 | 5 | 4 | 3 | 2 | 1 |
| 7 | lw x7, 4(x5) | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x5 | 0x4 | 5 | 4 | 3 | 2 | 1 |
| 8 | blt x6, x7, 0xc | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x5 | 0x4 | 5 | 4 | 3 | 2 | 1 |
| 9 | sw x7, 0(x5) | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x5 | 0x4 | 4 | 4 | 3 | 2 | 1 |
| 10 | sw x6, 4(x5) | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x5 | 0x4 | 4 | 5 | 3 | 2 | 1 |
| 11 | addi x4, x4, 4 | 0x10 | 0x10 | 0x0 | 0x4 | 0x0 | 0x5 | 0x4 | 4 | 5 | 3 | 2 | 1 |
| 12 | blt x4, x3, 0xffff\_ffe4 | 0x10 | 0x10 | 0x0 | 0x4 | 0x0 | 0x5 | 0x4 | 4 | 5 | 3 | 2 | 1 |
| 13 | add x5, x2, x4 | 0x10 | 0x10 | 0x0 | 0x4 | 0x4 | 0x5 | 0x4 | 4 | 5 | 3 | 2 | 1 |
| 14 | lw x6, 0(x5) | 0x10 | 0x10 | 0x0 | 0x4 | 0x4 | 0x5 | 0x4 | 4 | 5 | 3 | 2 | 1 |
| 15 | lw x7, 4(x5) | 0x10 | 0x10 | 0x0 | 0x4 | 0x4 | 0x5 | 0x3 | 4 | 5 | 3 | 2 | 1 |
| 16 | blt x6, x7, 0xc | 0x10 | 0x10 | 0x0 | 0x4 | 0x4 | 0x5 | 0x3 | 4 | 5 | 3 | 2 | 1 |
| 17 | sw x7, 0(x5) | 0x10 | 0x10 | 0x0 | 0x4 | 0x4 | 0x5 | 0x3 | 4 | 3 | 3 | 2 | 1 |
| 18 | sw x6, 4(x5) | 0x10 | 0x10 | 0x0 | 0x4 | 0x4 | 0x5 | 0x3 | 4 | 3 | 5 | 2 | 1 |
| 19 | addi x4, x4, 4 | 0x10 | 0x10 | 0x0 | 0x8 | 0x4 | 0x5 | 0x3 | 4 | 3 | 5 | 2 | 1 |
| 20 | blt x4, x3, 0xffff\_ffe4 | 0x10 | 0x10 | 0x0 | 0x8 | 0x4 | 0x5 | 0x3 | 4 | 3 | 5 | 2 | 1 |
| 序号 | 指令序列 | RARS仿真结果 | | | | | | | | | | | |
| x1 | x2 | x3 | x4 | x5 | x6 | x7 | m0 | m1 | m2 | m3 | m4 |
| 1 | addi x1, x0, 16 | 0x10 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 5 | 4 | 3 | 2 | 1 |
| 2 | addi x2, x0, 0 | 0x10 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 5 | 4 | 3 | 2 | 1 |
| 3 | addi x3, x1, 0 | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 5 | 4 | 3 | 2 | 1 |
| 4 | addi x4, x0, 0 | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 5 | 4 | 3 | 2 | 1 |
| 5 | add x5, x2, x4 | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 5 | 4 | 3 | 2 | 1 |
| 6 | lw x6, 0(x5) | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x5 | 0x0 | 5 | 4 | 3 | 2 | 1 |
| 7 | lw x7, 4(x5) | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x5 | 0x4 | 5 | 4 | 3 | 2 | 1 |
| 8 | blt x6, x7, 0xc | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x5 | 0x4 | 5 | 4 | 3 | 2 | 1 |
| 9 | sw x7, 0(x5) | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x5 | 0x4 | 4 | 4 | 3 | 2 | 1 |
| 10 | sw x6, 4(x5) | 0x10 | 0x10 | 0x0 | 0x0 | 0x0 | 0x5 | 0x4 | 4 | 5 | 3 | 2 | 1 |
| 11 | addi x4, x4, 4 | 0x10 | 0x10 | 0x0 | 0x4 | 0x0 | 0x5 | 0x4 | 4 | 5 | 3 | 2 | 1 |
| 12 | blt x4, x3, 0xffff\_ffe4 | 0x10 | 0x10 | 0x0 | 0x4 | 0x0 | 0x5 | 0x4 | 4 | 5 | 3 | 2 | 1 |
| 13 | add x5, x2, x4 | 0x10 | 0x10 | 0x0 | 0x4 | 0x4 | 0x5 | 0x4 | 4 | 5 | 3 | 2 | 1 |
| 14 | lw x6, 0(x5) | 0x10 | 0x10 | 0x0 | 0x4 | 0x4 | 0x5 | 0x4 | 4 | 5 | 3 | 2 | 1 |
| 15 | lw x7, 4(x5) | 0x10 | 0x10 | 0x0 | 0x4 | 0x4 | 0x5 | 0x3 | 4 | 5 | 3 | 2 | 1 |
| 16 | blt x6, x7, 0xc | 0x10 | 0x10 | 0x0 | 0x4 | 0x4 | 0x5 | 0x3 | 4 | 5 | 3 | 2 | 1 |
| 17 | sw x7, 0(x5) | 0x10 | 0x10 | 0x0 | 0x4 | 0x4 | 0x5 | 0x3 | 4 | 3 | 3 | 2 | 1 |
| 18 | sw x6, 4(x5) | 0x10 | 0x10 | 0x0 | 0x4 | 0x4 | 0x5 | 0x3 | 4 | 3 | 5 | 2 | 1 |
| 19 | addi x4, x4, 4 | 0x10 | 0x10 | 0x0 | 0x8 | 0x4 | 0x5 | 0x3 | 4 | 3 | 5 | 2 | 1 |
| 20 | blt x4, x3, 0xffff\_ffe4 | 0x10 | 0x10 | 0x0 | 0x8 | 0x4 | 0x5 | 0x3 | 4 | 3 | 5 | 2 | 1 |

Vivado仿真波形图如下：

电脑萤幕画面

描述已自动生成电脑萤幕画面

描述已自动生成图形用户界面

描述已自动生成

图形用户界面

描述已自动生成电脑萤幕画面

描述已自动生成日程表

描述已自动生成 可以看到，最终数组按照从小到大排序完毕。

# 错误记录和解决

仿真出现时出现了Z：最开始我检查了设计文件代码，检查了很久没有解决，最终我检查了仿真文件代码，发现问题出现在仿真文件没有连接模块输出。

在这次实验中，为了方便指令的解析，我把指令放在了reg[31:0] iram[63:0]类型的变量中。但是分支指令的imm是以字节为单位。所以需要将imm/4才能正确访问到要跳转到的指令位置。在这个过程中，遇到了如下问题：

* Imm右移二位：在verilog中，“>>”表示逻辑右移，“>>>”表示算数右移。这与Java中的规定正好相反，导致了我浪费了许多时间寻找问题。
* 在分支跳转指令中，imm是pc的相对偏移，是一个有符号数，所以在声明模块时需要使用signed标识。也可以使用$signed()函数来进行转化。

数据依赖问题：最开始进行设计时，没有考虑到这个问题。导致出现了寄存器写入位置错误、指令没有跳转等莫名其妙的问题。最终我把指令周期分解为：分析指令、计算、寄存器写入或内存写入、形成下一条PC共四个阶段来解决该问题。

# 心得体会

在做实验一的时候，我觉得实验二可能并没有我想的那么困难，于是就开始按照任务书和先导PPT进行实现，先导PPT中给出了寄存器堆和ALU的实现，尤其是数据通路的图片让我有了大体思路。

在实现过程中，我遇到了一些问题，这些问题往往需要通过仿真和$display()来进行“单步调试”。解决问题过程中，我回想起了上学期在《计算机系统导论》这门课程中第4章的内容，而实验二的单周期CPU是那个章节里最基础的部分。虽然原理简单，但是在自己实现过程中确实会遇到许多意料之外的问题，需要一个一个解决。

最后感谢宿老师在我大三一学年教授我《计算机系统导论》和《计算机组成与体系结构》两门课程，让我对计算机的硬件和底层有了一定的了解。