计算机组成与体系结构 Lab2
单周期 CPU 设计实验

郑子帆 1120200822

北京理工大学 计算机学院 07112002 班

日期：2023 年 6 月 30 日

摘　　要

本文为北京理工大学《2023 计算机组成与体系结构》课程的 Lab2 实验报告。在本次实验我们设计了单周期 CPU 的模块结构图并用 Verilog 完成了实现。我们以 5 个整数的冒泡排序程序作为测试程序，完成了向机器码的转化和对我们实现的单周期 CPU 的仿真，并 jin′xing′le′jie′guo′fen′ji

1　实验简介 [3]

1.1　实验目的

1. 掌握 RISCV 指令格式与编码，掌握 RISCV 汇编转换成机器代码的方法。
2. 掌握单周期 CPU 的数据通路与控制器设计方法。
3. 掌握硬件描述语言与 EDA 工具。
4. 掌握基本测试与调试方法。

1.2　实验内容

1. 设计并实现单周期处理器，支持包含 RISC-V RV32I 整数指令（LW，SW，ADD，SUB，ORI，BEQ）在内的指令子集等。
2. 测试程序：完成 2 后在该 cpu 上实现对 5 个整数的排序，仿真测试结果与 RARS 中运行结果对比正确。
3. 乐学提交实验报告、工程源代码和测试程序。

1.3 实验环境

1. 硬件描述语言 Verilog HDL。
2. Vivado 2019.2 版本，AMD 官网下载。
3. RISC-V 仿真器：RARS 下载地址：https://github.com/TheThirdOne/rars。
4. RARS 需要配置 Java 环境，jdk 下载地址：https://www.oracle.com/java/technologies/javase-jdk11-downloads. html

2 实验过程

2.1 设计指令集

下图是 RISC-V 的六种指令格式 [1]，在本次实验中非常重要：

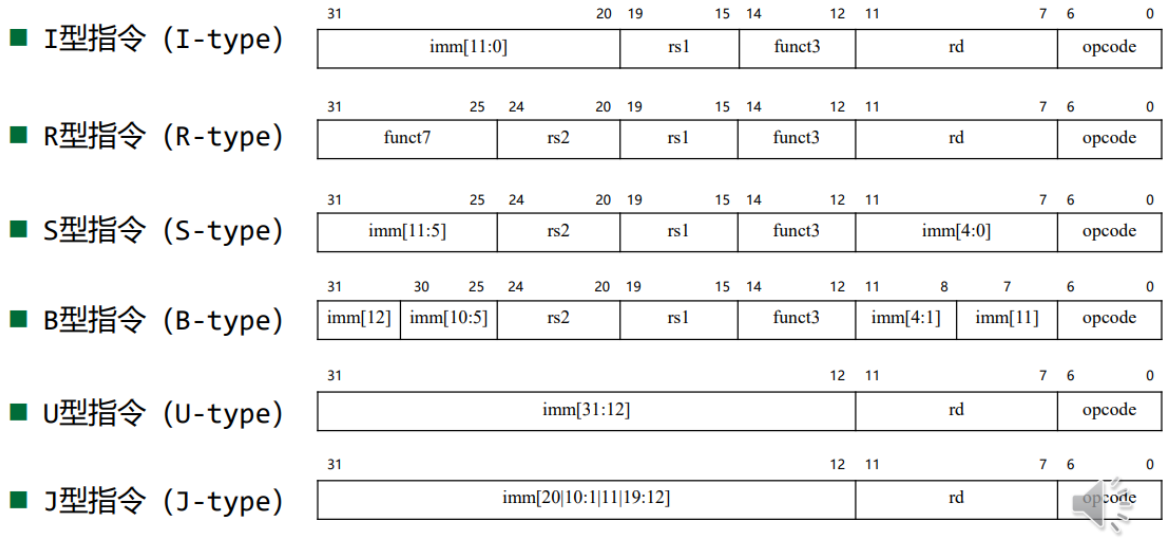

图 1: RISC-V 六种指令格式

首先我们先选择并设计在本实验中我们需要用到的指令，具体如表1。

2.2 CPU 数据通路设计

本实验主要参考了乐学上所提供的资料 [2]，且经过思考发现，需要实现的指令基本都已在资料中呈现，且新加的指令如 blt 等也不需要更改数据通路设计图，故本实验直接采用参考资料中的总设计图如图2。

表 1: 指令功能与数目

序号	操作码	助记符	功能	描述
1	0010011	addi	R[rd]<=R[rs1]+imm; PC<=PC+4	寄存器值加立即数
2	0010011	ori	R[rd]<=R[rs1]|imm; PC<=PC+4	寄存器值或立即数
3	1101111	jal	R[rd]<=PC+4: PC<=PC+imm	无条件转移
4	0000011	lw	R[rd]<=M[R[rs1]+imm]; PC<=PC+4	将内存中数读入寄存器
5	0100011	sw	M[R[rs1]+imm]<=R[rs2]; PC<=PC+4	将寄存器值写入内存
6	0110011	add	R[rd]<=R[rs1]+R[rs2]; PC<=PC+4	寄存器值相加
7	0110011	sub	R[rd]<=R[rs1]-R[rs2]; PC<=PC+4	寄存器值相减
8	1100011	beq	if(R[rs1]==R[rs2]) PC<=imm+PC	相等时跳转
9	1100011	blt	if(R[rs1]<RIrs2]) PC<=imm+PC	小于时跳转
10	1100011	bge	if(R[rs1]>=R[rs2]) PC<=imm+PC	大于等于时跳转
指令总数目			10	

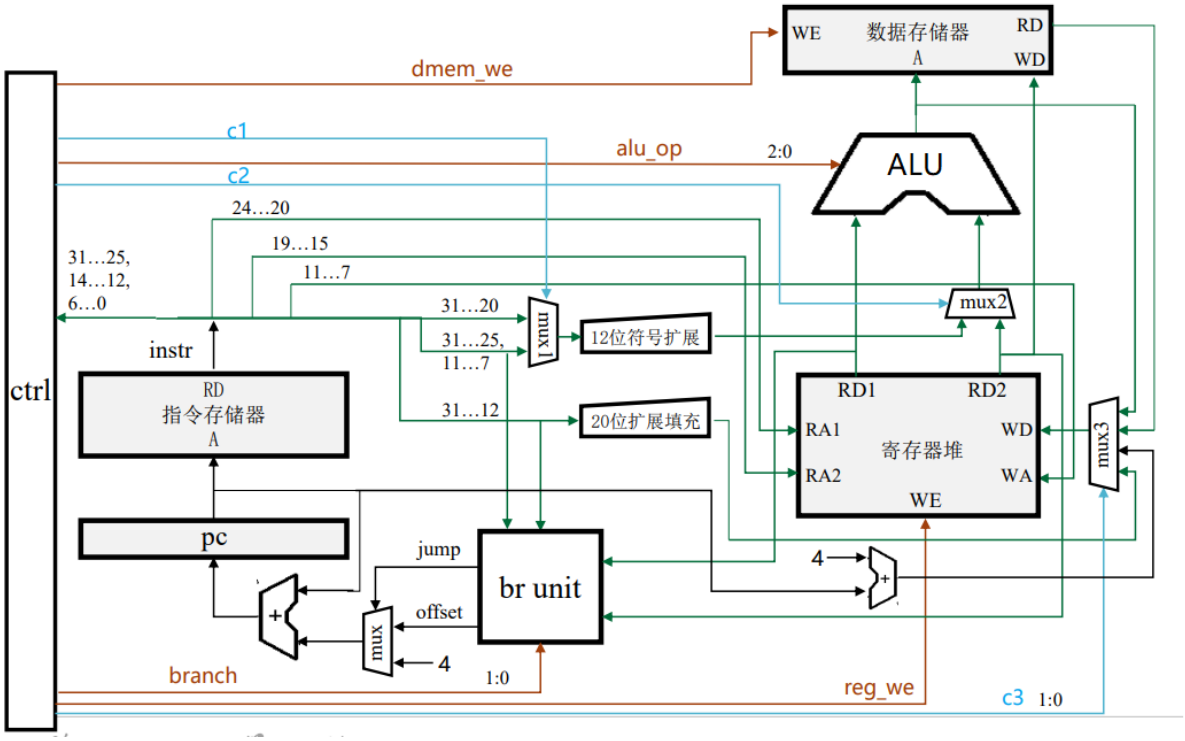

图 2: CPU 总数据通路设计结构图

2.3 各子模块实现与仿真

本部分我们将按照上面的总设计图一一进行实现。

2.3.1 PC

这个模块比较简单，模块的引脚设置如下：

```
module PC(
        input clk,
        input [31:0] nxt_inst,
        output [31:0] now_inst
    );
```

我们在 clk 信号上升沿时触发 PC 更新，将下一条即将要执行的指令的地址传递给 now_inst。

电路图如下：

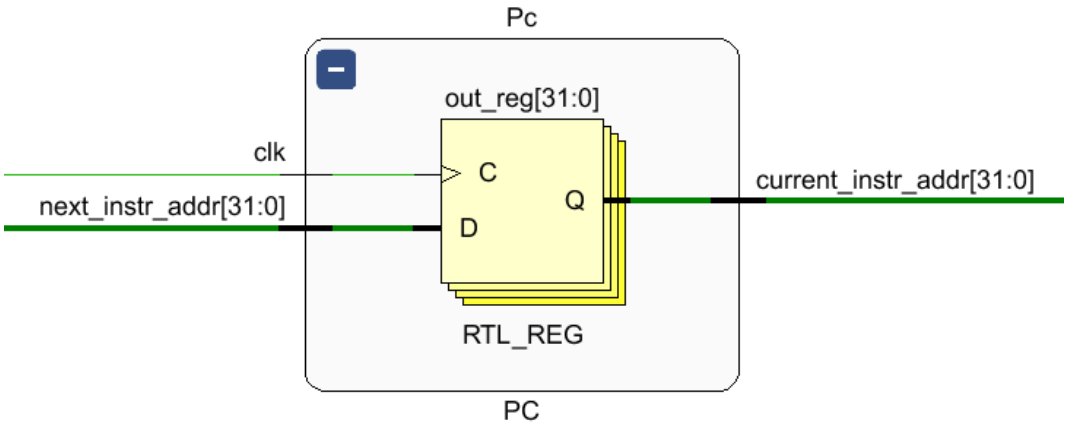

图 3: PC 模块电路图

仿真结果如下：

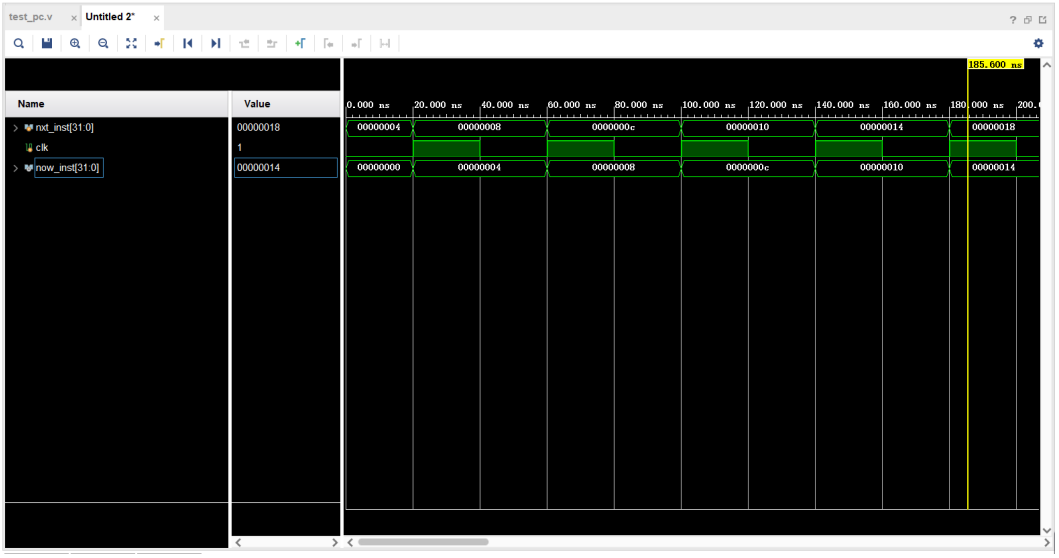

图 4: PC 仿真波形图

2.3.2 InstructionMemory 指令存储器

```
module InstructionMemory(
        input [31:0] in_addr,
        output [31:0] instruction
    );
```

这个模块用于存取指令。我们会预先将指令存在"text.txt" 文件中，该模块负责将 txt 文件中的指令导入并存起来，在 PC 的控制下取出对应地址下的指令。

电路图如下：

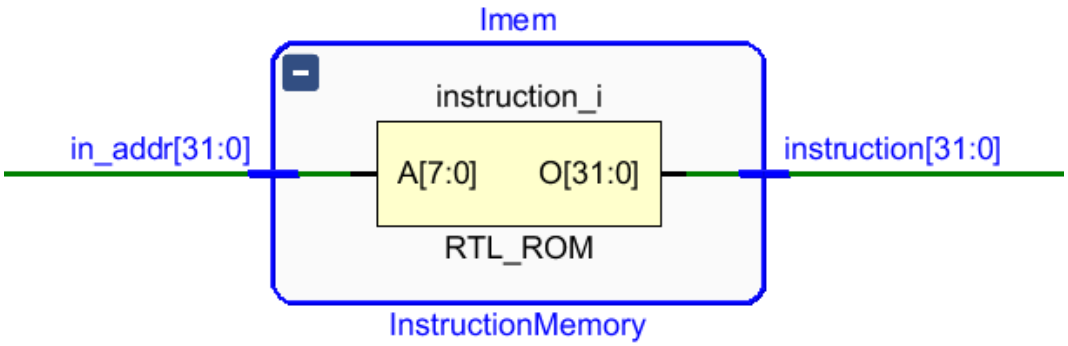

图 5: InstructionMemory 模块电路图

仿真结果如下：

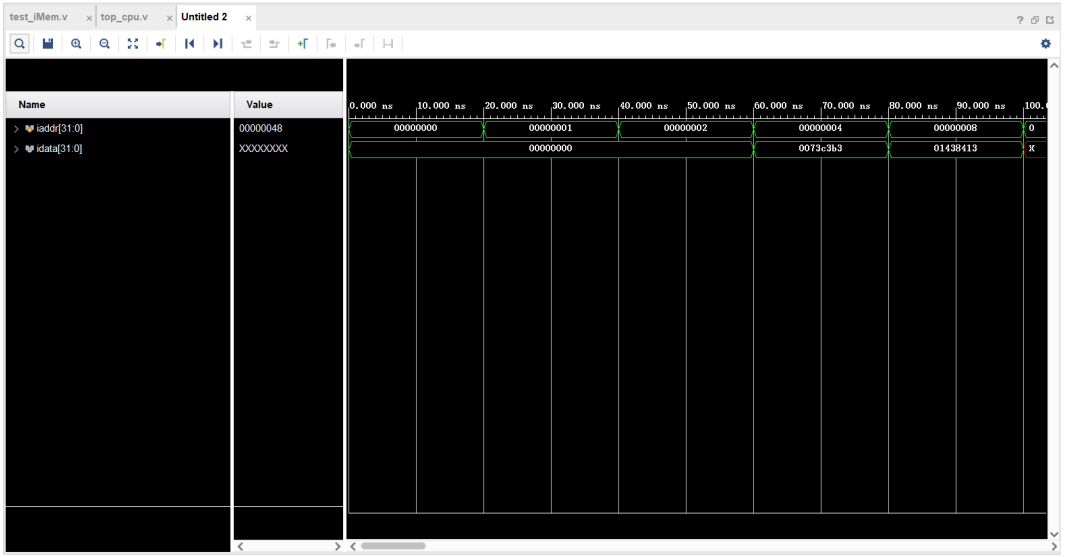

图 6: InstructionMemory 仿真波形图

2.3.3 DataMemory 数据存储器

```
module DataMemory(
        input clk,
        input [31:0] in_addr,
        input we,
        input [31:0] wdata,
        output [31:0] rdata
    );
```

这个模块与上一模块类似，不过这个是存储数据的，在 clk 信号下降沿时触发。首先我们将汇编代码中的数据区编写成 16 进制的文件（可由 RARS 自动生成），然后该模块负责导入并存储这些数据（在本实验中即为 5 个整数）。

这个模块的输入里有数据地址 in_addr 和写使能信号 we，当 we=1 时写入数据，此时 wdata 有效；否则只读出 in_addr 对应的数据。

电路图如下：

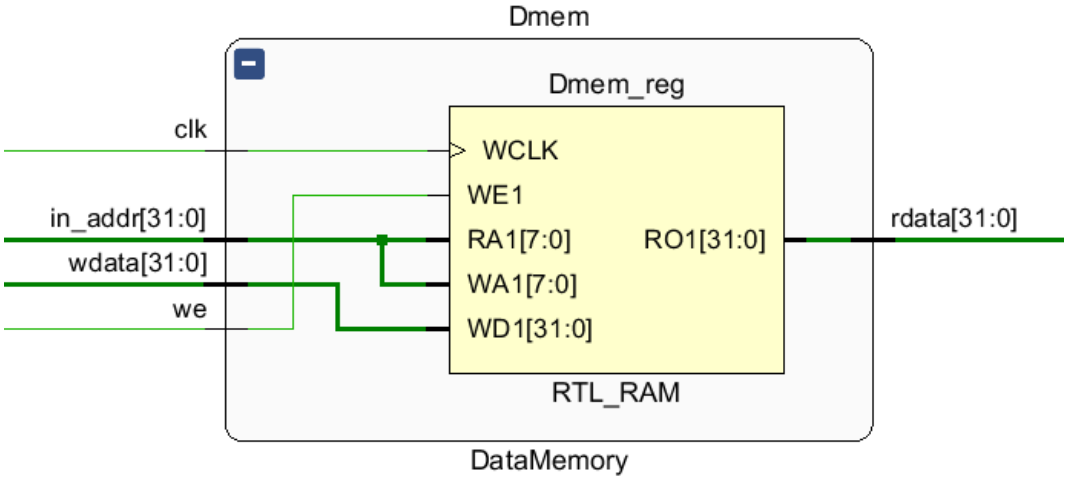

图 7: DataMemory 模块电路图

仿真结果如下：

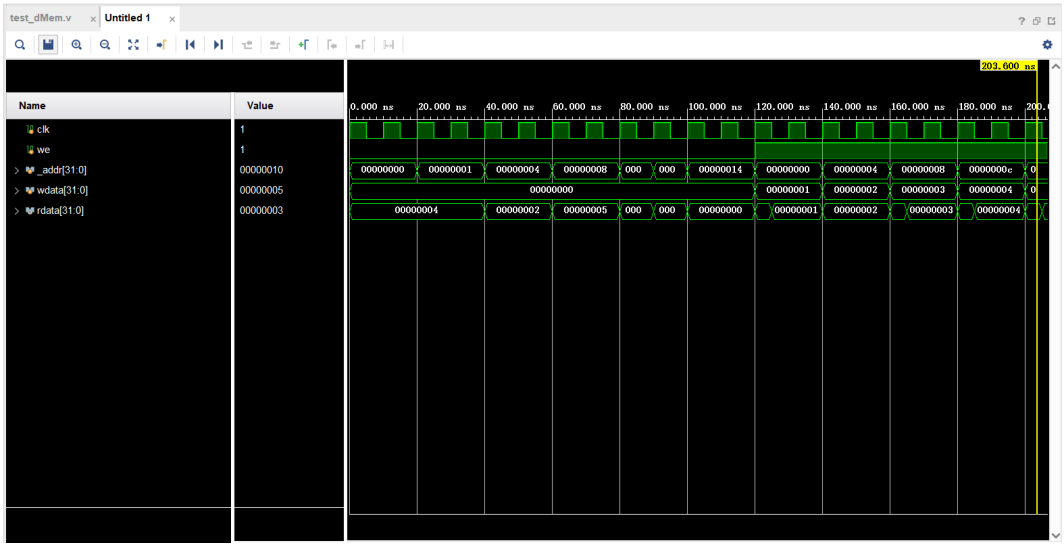

图 8: DataMemory 仿真波形图

2.3.4 RegFile 寄存器堆

```
module RegFile(
        input clk, // 时钟信号
        input rst, // 复位信号
        input reg_we, // 读写信号
        input [4:0] rs_addr, // 读寄存器地址 1
        input [4:0] rt_addr, // 读寄存器地址 2
        input [4:0] wb_addr, // 写寄存器的地址
        input [31:0] wb_data, // 要写入 wb 的数据
        output [31:0] rs_data, // 读出数据 1
        output [31:0] rt_data  // 读出数据 2
    );
```

寄存器堆设计为两读一写功能，一共有 32 个 32 为寄存器。rst 信号下降沿触发寄存器组重置，即所有寄存器置 0。clk 下降沿触发读/写操作：当 we=1 时，写入有效，将选择的数据写入指定的寄存器中；当 we=0 时，读出有效，按照地址将寄存器中的数据读出。

仿真结果如下：

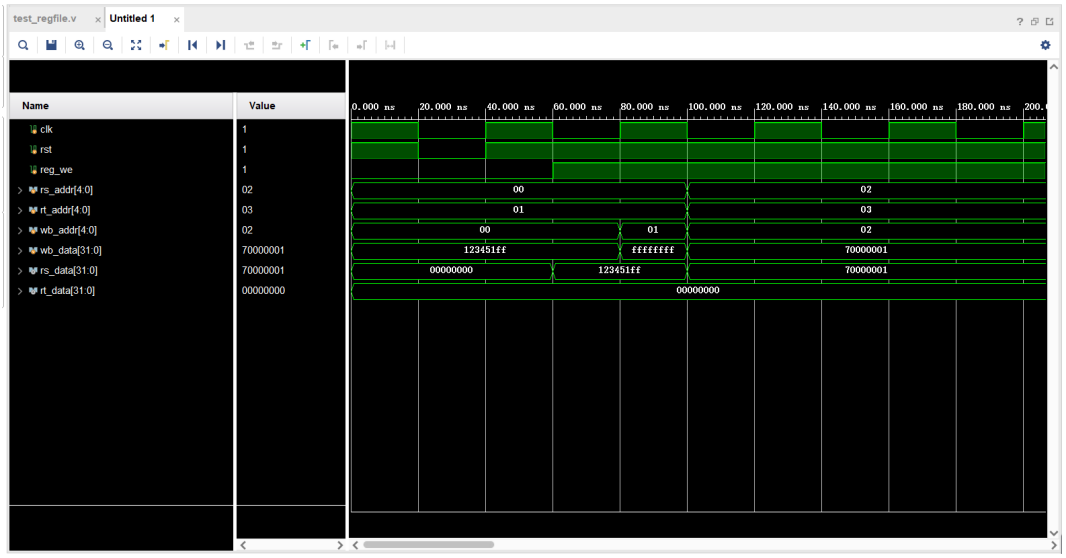

图 9: RegFile 仿真波形图

2.3.5 ALU

在本实验中，我直接使用了 Lab1 中编写的 ALU 代码，没有进行改动，故在此不再赘述。

2.3.6 BranchUnit 分支单元

```
module BrUnit(
        input [11:0] br_offset,    // 条件跳转偏移
        input [19:0] jal_offset,   // 直接跳转偏移
        input [31:0] rs1_data,     // 源数 1
        input [31:0] rs2_data,     // 源数 2
        input [2:0] branch,       // 控制信号
        output reg jump,          // 是否跳转
        output reg [31:0] offset // 偏移量
    );
```

分支单元需要根据控制信号和指令中的偏移来决定需要进行哪种跳转。在本实验中设计了 4 种跳转类型（bge，beq，blt，jal），并且还要预留一类控制信号代表非跳转指令，所以控制信号需要 3 位。

仿真结果如下：

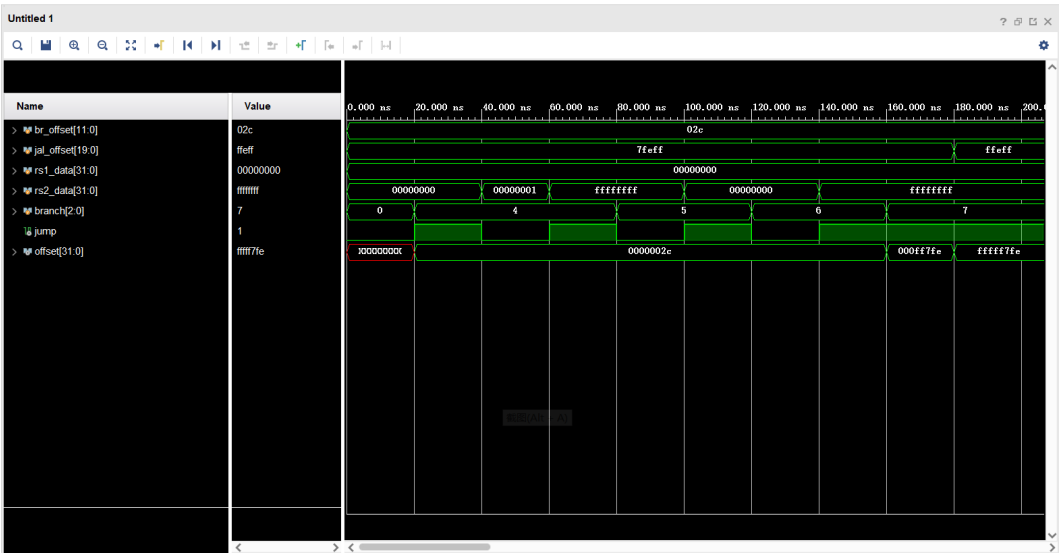

图 10: BranchUnit 仿真波形图

2.3.7 ControlUnit 控制单元

```
module ControlUnit(
        input [6:0] opcode,
        input [2:0] funct3,
        input [6:0] funct7,
        output c1,
        output c2,
        output [1:0] c3,
        output [3:0] alu_op,
        output [2:0] branch,
        output dmem_we,
        output reg_we
    );
```

这个子模块可以说是核心模块，它产生控制其他子模块的控制信号。我们所设计的控制信号 [2] 如图11。

控制信号	位宽	控制对象
dmem_we	1	dmem
reg_we	1	regfile
alu_op	3	alu
branch	2	br unit
c1	1	mux1
c2	1	mux2
c3	2	mux3

图 11: 控制信号设计

具体的电路图如下：

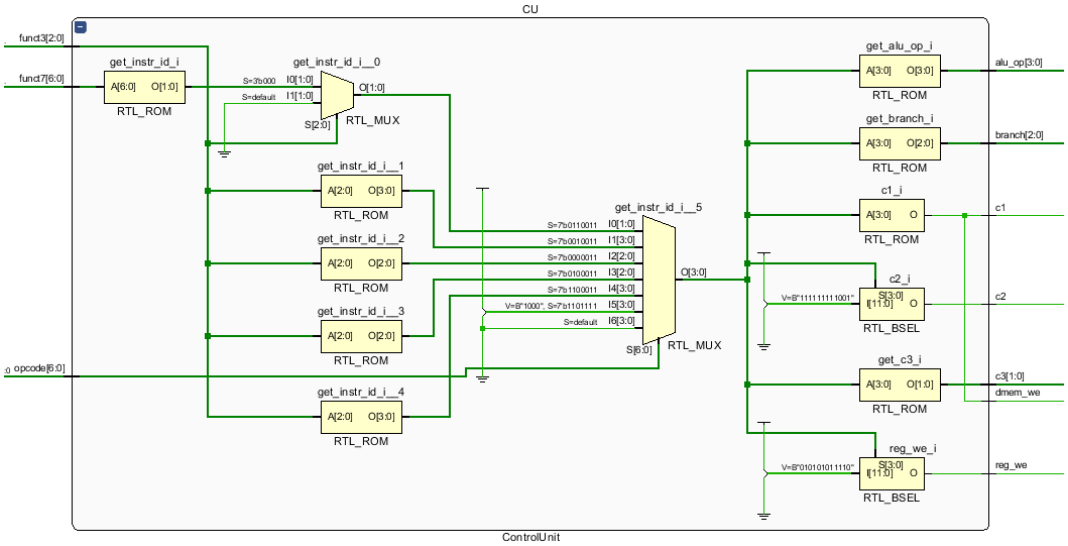

图 12: ControlUnit 仿真波形图

根据指令和控制信号的设计，我们可以设计出下面的控制信号列表：

opcode	指令	dmem_we	reg_we	branch	alu_op	c1	c2	c3
0110011	add	0	1	000	0000	X	0	00
0110011	sub	0	1	000	0001	X	0	00
0010011	addi	0	1	000	0000	0	1	00
0010011	ori	0	1	000	0011	0	1	00
1100011	beq	0	0	101	X	X	X	X
0000011	lw	0	1	000	0000	0	1	01

opcode	指令	dmem_we	reg_we	branch	alu_op	c1	c2	c3
0100011	sw	1	0	000	0000	1	1	X
1101111	jal	0	1	111	X	X	X	10
1100011	bge	0	0	100	X	X	X	X
1100011	blt	0	0	011	X	X	X	X

2.3.8 其他模块

除了上述这些模块，我还按照设计结构图实现了加法器 Adder，多路复用器 MUX 等等，由于比较简单，在这里不再赘述。

2.4 顶层模块 TOP

在完成了各个子模块的实现后，我们需要完成最后一步——顶层模块的实现。顶层模块虽然代码量比较大，但是其实它就是按照图2来模拟连接的过程，即将有关的、相连的子模块的输出和输入进行“连接”。TOP.v 的代码详见附录B。

3 实验测试前准备

3.1 设计冒泡排序汇编代码

根据之前所学知识，我们可以很容易的设计出冒泡排序的汇编代码，并且利用《编译原理》课中相关代码优化知识（如删除归纳变量，强度削弱，常量合并等），得到了如下的汇编代码。

```
.data
list:
    .word 4, 2, 5, 1, 3

.text
bsort:
    xor t2, t2, t2
    addi s0, t2, 20
    ori t0, t2, 0
    jal end1
L1:
    addi t1, t2, 16
    jal end2
```

```
L2:
    lw a0, (t1)
    lw a1, -4(t1)
    bge a0, a1, endif
    sw a1, (t1)
    sw a0, -4(t1)
endif:
    addi t1, t1, -4
end2:
    blt t0, t1, L2
    addi t0, t0, 4
end1:
    blt t0, s0, L1
```

3.2 利用 RARS 转化

利用事先准备好的工具 RARS，通过编译可以获得我们写的汇编代码的相对应的机器码（具体见附录A），运行过程截图如图13和14。

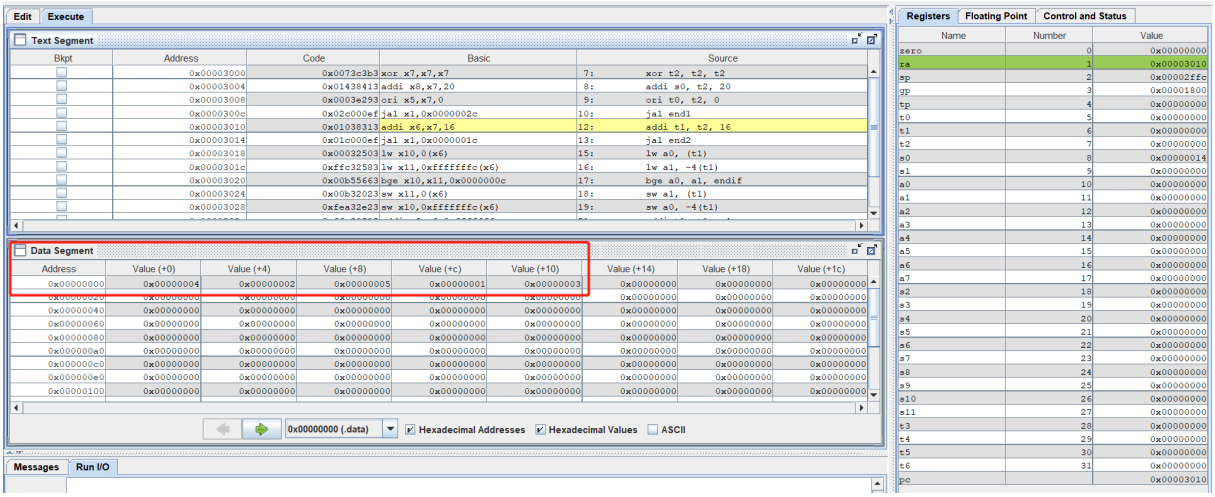

图 13: RARS 运行前

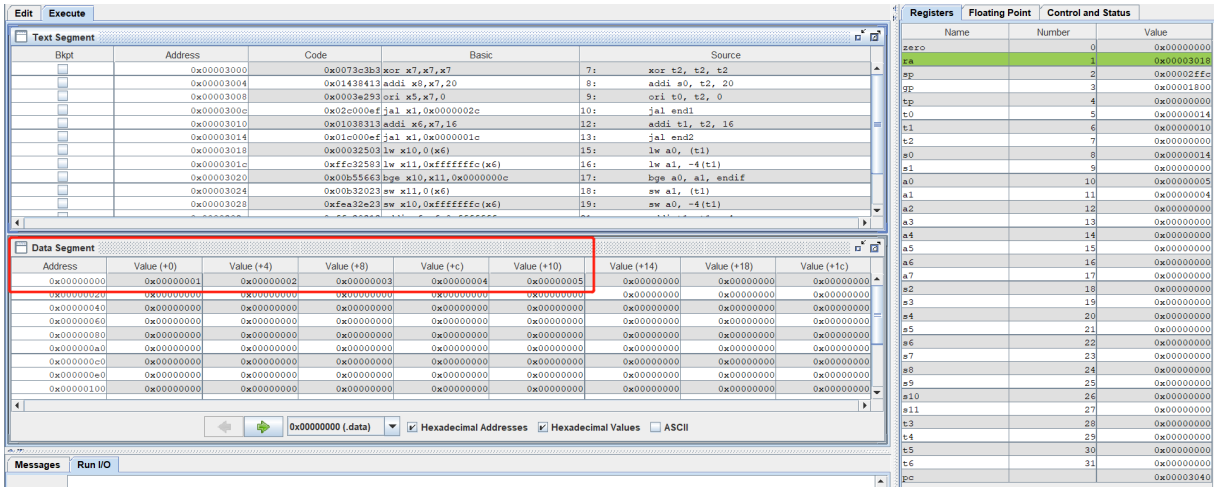

图 14: RARS 运行后

可以看到我们设计的汇编代码能够正确地转化成机器码同时正确地对 5 个整数进行排序。

3.3 指令与对应的机器码

将上面 RARS 编译得到的机器码与 RISC-V 指令集相对照，可以得到表3。

序号	具体指令	funct7	rs2	rs1	funct3	Rd	op
1	xor x7,x7,x7	0000000	00111	00111	000	01110	0110011
2	addi x8,x7,20	0000000	10100	01110	000	10000	010011
3	ori x5,x7,0	0000000	00000	01101	100	01010	010011
4	jal x1,0x2c	0000001	01100	00010	000	00011	101111
5	addi x6,x7,16	0000000	10000	01111	000	01100	010011
6	jal x1, 0x1c	0000000	11100	00001	000	00011	101111
7	lw x10,0(x6)	0000000	00000	00110	100	10100	000011
8	lw x11,0xfc(x6)	1111111	11100	00100	100	10110	000011
9	bge x10,x11,0x0c	0000000	01011	01000	010	11001	100011
10	sw x11,0(x6)	0000000	01011	00110	100	00000	100011
11	sw x10,0xfc(x6)	1111111	01010	00110	101	11000	100011
12	addi x6,x6,0xfc	1111111	11100	00110	000	01100	010011
13	blt x5,x6,0xe8	1111111	00110	00101	000	10011	100011
14	addi x5,x5,4	0000000	00100	00101	000	01010	010011
15	blt x5,x8,0xd8	1111110	01000	00101	001	10011	100011

4 实验仿真与结果分析

4.1 TestBench 的 Verilog 实现

编写 TestBench 程序如下：

```
module TestBench(
    );

     reg clk, rst;
    initial begin
        clk = 0;
        rst = 1;
    end

    always #20 clk = ~clk;

    Top TestCPU(
        .clk(clk),
        .rst(rst)
    );
endmodule
```

4.2 执行指令序列对比

启动 Vivado 的 Simulation，对比每次执行的指令和 RARS 软件执行的指令，如下表（因为篇幅太长，所以只截选了前 30 条）。

序号	单周期 CPU 执行指令	RARS 执行指令
1	0073c3b3	0073c3b3
2	01438413	01438413
3	0003e293	0003e293
4	02c000ef	02c000ef
5	fc82cce3	fc82cce3
6	01038313	01038313
7	01c000ef	01c000ef
8	fe62c4e3	fe62c4e3
9	00032503	00032503

序号	单周期 CPU 执行指令	RARS 执行指令
10	ffc32583	ffc32583
11	00b55663	00b55663
12	ffc30313	ffc30313
13	fe62c4e3	fe62c4e3
14	00032503	00032503
15	ffc32583	ffc32583
16	00b55663	00b55663
17	00b32023	00b32023
18	fea32e23	fea32e23
19	ffc30313	ffc30313
20	fe62c4e3	fe62c4e3
21	00032503	00032503
22	ffc32583	ffc32583
23	00b55663	00b55663
24	00b32023	00b32023
25	fea32e23	fea32e23
26	ffc30313	ffc30313
27	fe62c4e3	fe62c4e3
28	00032503	00032503
29	ffc32583	ffc32583
30	00b55663	00b55663

4.3 仿真截图

点击"Run Simulation" 完成仿真，仿真运行截图如下：

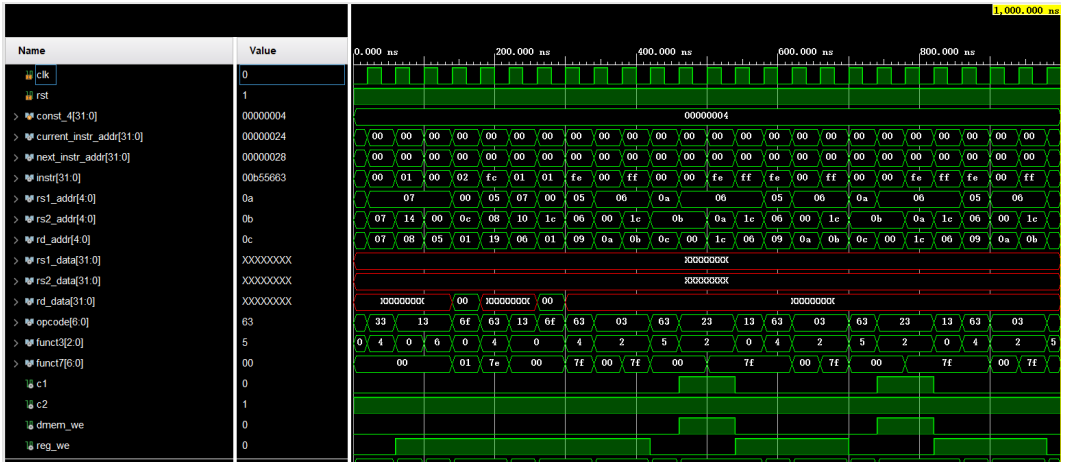

图 15: 仿真运行截图（截选）

经分析可以得到结果与预期一致。

5 实验心得与体会

本次实验相比第一个实验难度徒增，可以说实验一只是实验二的一小部分，自从考试结束到 ddl 当天，我一直都在努力完成这个作业。在花费大量时间和精力的情况下，这次实验让我受益匪浅

首先因为实验和课上所讲的理论知识部分重叠度不是很高，所以我需要先自学一下 RISC-V 指令集，另外上一次使用 Verilog 和 Vivado 还是一年前，所以在完成这个工作量这么大的程序前我还需要复习和学习一下 Verilog 的语法，以及 Vivado 的使用（比如我最开始忘记了如何能让更多变量的波形图显示在仿真结果中，以及如何 display，如何调整仿真模拟时间等等）

作为一个硬件方面的“小白”，其实我已经有一些忘了组合逻辑和时序逻辑的区别，可能“刻板印象”只是组合逻辑更简单，时序逻辑复杂一些，但这次实验也让我亲身体验到了组合和时序的区别，如果处理不当（比如 clk、rst 信号上升沿、下降沿触发等）就会导致仿真结果差别很大。所以我也通过这次实验对于时序逻辑电路的实现有了更深的理解。

当然，我也在完成实验的过程中犯了一些低级错误，比如 PC+1，让我牢记 1 是一条指令，而一条指令有 4 个字节。还有我对于 bge 等指令有一些逻辑上的构思错误，这也教会我以后在编写大工程代码时一定要严谨，想清楚再写。

除了专业知识和技能的提升，我还在这次实验中提升了自己熟练使用 latex 的能力，以及做表格的能力等等。总之，这次工程量大、难度大的实验虽然做的时候很痛苦但是完成之后有一种成就感和如释重负的解脱感，也让我受益匪浅。

最后，感谢马忠梅老师、王娟老师一学期以来的倾囊相授和答疑解惑，祝老师们未来一切顺利！

参考文献

[1] *01RISCV 指令与仿真.pdf*. zh. 2023.

[2] *02RV32I 单周期 CPU 设计.pdf*. zh. 2023.

[3] *实验单周期 CPU 任务书.pdf*. zh. 2023.

A 通过 RARS 转化后的 RISC-V 代码（十六进制）

.text:

```
0073c3b3
01438413
0003e293
02c000ef
01038313
01c000ef
00032503
ffc32583
00b55663
00b32023
fea32e23
ffc30313
fe62c4e3
00428293
fc82cce3
```

.data:

```
00000004
00000002
00000005
00000001
00000003
```

B TOP.v 代码

```
module Top(
    input clk,
    input rst
    );

    integer const_4 = 32'd4;          // 加 4

    // PC
```

```
wire [31:0] current_instr_addr, next_instr_addr;
wire [31:0] instr;

// RegFile
wire [4:0] rs1_addr, rs2_addr, rd_addr;
wire [31:0] rs1_data, rs2_data, rd_data;

// CU
wire [6:0] opcode;
wire [2:0] funct3;
wire [6:0] funct7;
wire c1, c2, dmem_we, reg_we;
wire [3:0] alu_op;
wire [1:0] c3;

// BRU
wire [31:0] act_offset, offset;
wire jump;
wire [19:0] jal_offset;
wire [11:0] br_offset;
wire [2:0] branch;

// MUX1
wire [11:0] imm_I, imm_S;
wire [11:0] imm;

// ALU
wire [31:0] alu_opnd1, alu_opnd2, alu_res;
wire SF, CF, ZF, OF, PF;

// ImmGen
wire [31:0] imm_expand;

// jal
wire [31:0] reg_jal;
```

```
// Dmem
wire [31:0] read_dmem_data;

PC Pc(
    .clk(clk),
    .nxt_inst(next_instr_addr),
    .now_inst(current_instr_addr)
);

InstructionMemory Imem(
    .in_addr(current_instr_addr),
    .instruction(instr)
);

assign jal_offset = instr[31:12];
assign br_offset = {instr[31:25], instr[11:7]};
BrUnit BrU(
    .br_offset(br_offset),
    .jal_offset(jal_offset),
    .rs1_data(rs1_data),
    .rs2_data(rs2_data),
    .branch(branch),
    .jump(jump),
    .offset(offset)
);
MUX2 MUX_BR(
    .src1(const_4),
    .src2(offset),
    .control(jump),
    .out(act_offset)
);

assign opcode = instr[6:0];
assign funct3 = instr[14:12];
assign funct7 = instr[31:25];
ControlUnit CU(
```

```
    .opcode(opcode),
    .funct3(funct3),
    .funct7(funct7),
    .c1(c1),
    .c2(c2),
    .c3(c3),
    .alu_op(alu_op),
    .branch(branch),
    .dmem_we(dmem_we),
    .reg_we(reg_we)
);

assign imm_I = instr[31:20];
assign imm_S = {instr[31:25], instr[11:7]};
MUX1 MUX_1(
    .src1(imm_I),
    .src2(imm_S),
    .control(c1),

    .out(imm)
);

ImmGen IG(
    .in(imm),
    .out(imm_expand)
);

assign alu_opnd1 = rs1_data;
ALU Alu(
    .OP(alu_op),
    .A(alu_opnd1),
    .B(alu_opnd2),
    .F(alu_res),
    .ZF(ZF),
    .CF(CF),
    .OF(OF),
```

```
    .SF(SF),
    .PF(PF)
);

assign rs2_addr = instr[24:20];
assign rs1_addr = instr[19:15];
assign rd_addr = instr[11:7];
RegFile RF(
    .clk(clk),
    .rst(rst),
    .reg_we(reg_we),
    .rs_addr(rs1_addr),
    .rt_addr(rs2_addr),
    .wb_addr(rd_addr),
    .wb_data(rd_data),

    .rs_data(rs1_data),
    .rt_data(rs2_data)
);

MUX2 MUX_2(
    .src1(rs2_data),
    .src2(imm_expand),
    .control(c2),
    .out(alu_opnd2)
);

Adder Add_J(
    .opnd1(current_instr_addr),
    .opnd2(act_offset),
    .res(next_instr_addr)
);

Adder Add_4(
    .opnd1(const_4),
    .opnd2(current_instr_addr),
```

```
        .res(reg_jal)
    );

    DataMemory Dmem(
        .clk(clk),
        .in_addr(alu_res),
        .we(dmem_we),
        .wdata(rs2_data),
        .rdata(read_dmem_data)
    );

    MUX3 MUX_3(
        .src1(alu_res),
        .src2(read_dmem_data),
        .src3(reg_jal),
        .control(c3),

        .out(rd_data)
    );

endmodule
```