单周期 CPU 设计

一、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

二、实验内容

1. 设计并实现单周期处理器，支持包含 add, addi, sub, lw, sw, beq,blt, jal, ori 的 RISC-V 指令子集。

2. 测试程序：完成 2 后在该 cpu 上实现对 5 个整数的排序，并验证仿真测试结果的正确性。

三、实验步骤

3.1 实验环境说明

- 操作系统: Microsoft Windows 22H2 [10.0.19045.2604]
- 实验环境: Vivado2019.2 + RARS 仿真器
- 编程语言: Verilog HDL + RICS-V 汇编

3.2 实现指令集说明

本次实验一共实现了 9 条指令，覆盖 R 型、I 型、S 型、B 型、J 型，涉及 3 种运算符(+, -, |), 具体说明如下：

表格 1 指令功能与数目

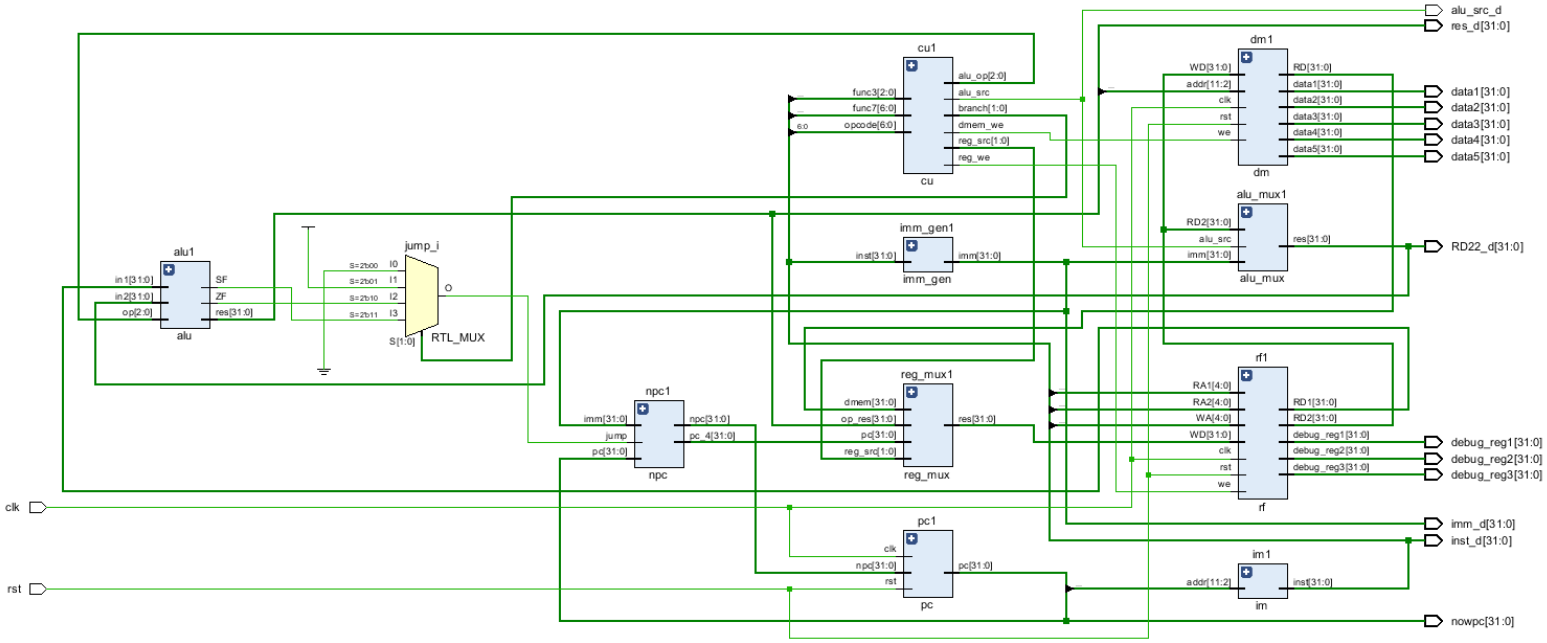
序号	操作码	助记符	功能	描述
1	0110011	add	x[rd]=x[rs1]+x[rs2]	寄存器加法
2	0010011	addi	x[rd]=x[rs1]+sext[imm]	立即数加法
3	0110011	sub	x[rd]=x[rs1]-x[rs2]	寄存器减法
4	0000011	lw	x[rd] = sext(M[x[rs1] +	将内存数据

			sext(offset)][31:0])	加载到寄存器中
5	0100011	sw	M[x[rs1] + sext(offset) = x[rs2][31:0]	将寄存器数据存储到内存中
6	1100011	beq	if (rs1 == rs2) pc += sext(offset)	相等跳转
7	1100011	blt	if (rs1 <$_s$ rs2) pc += sext(offset)	小于跳转
8	1101111	jal	x[1] = pc+2; pc += sext(offset)	无条件跳转并存储返回地址到寄存器中
9	0010011	ori	x[rd] = x[rs1] | sext(immediate)	立即数按位或
指令总数				9

3.3 CPU 数据通路设计

3.3.1 子模块设计总述

一共设计了 11 个子模块，分别为 pc 模块，npc 模块，rf 模块，imm_gen 模块，im 模块，dm 模块，alu_mux 模块，reg_mux 模块，alu 模块，cu 模块，top_cpu 模块。这些模块之间的数据通路如下所示：



3.3.2 具体实现

首先，定义了头文件 macro.vh，包含指令字段值、寄存器/存储器大小、选择信号位数等常量的宏定义.

```
// 寄存器的初始值
`define DEFAULT_VAL 32'h0

// 指令存储器和数据存储器大小
`define IM_SIZE       1023
`define DM_SIZE       1023

// 跳转控制信号位数
`define BRANCH_LEN 2

// 符号选择信号位数
`define ALU_LEN 3

// 使用到的opcode字段值
`define OPCODE_R 7'b0110011    //add,sub
`define OPCODE_I1 7'b0010011    //addi,ori
`define OPCODE_I2 7'b0000011    //lw
`define OPCODE_S 7'b0100011     //sw
`define OPCODE_B 7'b1100011     //beq, blt
`define OPCODE_J 7'b1101111     //jal

// 使用到的func字段值
`define FUNC3_ADDSUB 3'b000     //add,sub
`define FUNC3_ADDI 3'b000       //addi
`define FUNC3_ORI 3'b110        //ori
`define FUNC7_ADD 7'b0000000    //add
`define FUNC7_SUB 7'b0100000    //sub
`define FUNC3_BEQ 3'b000        //beq
`define FUNC3_BLT 3'b100        //blt
```

接下来，将对每个子模块功能与数据 I/O 进行详细解释：(其中，cu 模块是控制模块,top_cpu 是连接所有模块的顶层模块，将在对控制信号进行说明后，再进行解释)

- npc 模块

该模块完成选择下一个 pc 值的功能。因为实现了跳转指令 jal,beq,blt,需要在 pc+4 与 pc+imm (即跳转)之间，根据传入的控制信号进行选择。

  - input
    - jump：1 位控制信号，1 表示需要跳转，pc=pc+imm;0 表示不需要跳转，pc=pc+4
    - imm：32 位，由指令中 12 位数字有符号拓展而来，为跳转时需要用到的立即数(offset)。

- pc：32 位，指针寄存器值。表示前一个指令地址值。

- output
  - npc：32 位，根据 jump 选择后得到的新(下一个)指令地址值
  - pc_4：32 位，pc+4 的值，用于 jal 中跳转前保存返回地址用。
- 代码实现

```
module npc(
    input [31: 0] imm,  // 跳转的pc offset
    input jump,   //是否需要跳转
    input  [31:0] pc,
    output [31:0] npc,
    output [31:0] pc_4
    );
assign pc_4=pc+4;

assign npc = jump==0 ?  pc+4 : pc+imm ;

endmodule
```

- pc 模块

该模块输出当前指令的地址值。根据是否设置 rst 信号，输出 npc 或者默认值。

```
module pc(
    input  clk, //时钟信号
    input  rst, //复位信号
    input [31:0] npc, // 下一个指令地址
    output reg [31:0] pc  //输出的pc值
    );

    always @(posedge clk, posedge rst) begin
        if (rst)
            pc <= `DEFAULT_VAL; //如果复位信号置1，则pc设为默认值
        else
            pc <= npc;  // 否则正常取下一个指令地址
    end
endmodule
```

- im 模块

该模块为指令存储器的抽象，通过输入的地址值，完成取指令数据的功能。使用 verilog 中内置的寄存器数组模拟存储器，故其中指令地址 addr 只取

32 位中的[11:2]位，原因是 macor.vh 中通过宏定义设置指令存储器大小为 1024*32b，addr 表示的是寄存器数组下标。

代码实现如下：

```
module im(
    input [11:2] addr, // 指令存储器由32位寄存器数组表示,直接取到数组的下标; 11由IM_SIZE而来
    output [31:0] inst
    );

reg[31:0] imem[`IM_SIZE:0];
assign inst=imem[addr];   //到对应位置取指令

endmodule
```

- dm 模块

该模块为数据存储器的抽象，同样使用 verilog 中内置的寄存器数组模拟存储器，故其中指令地址 addr 只取 32 位中的[11:2]位。

  - input
    - clk，rst：时钟和复位信号
    - we:1 位控制信号，写使能。只有在 we 值 1 且 clk 上升沿时，才进行数据写入。
    - addr：10 位，待读数据的地址
    - WD：32 位，待写数据的值
  - output
    - RD：32 位读出数据的值
    - 其他 debug 信号(调式用)

代码实现如下

```
module dm(
    input clk,
    input rst,
    input we,
    input [11:2] addr,
    input [31:0] WD,
    output [31:0] RD,
    );
// 存储器
reg[31:0] dmem[`DM_SIZE:0];

assign RD=dmem[addr];

always @(posedge clk) begin
    if (rst==0 && we ) begin
        dmem[addr] <= WD;
    end
end
endmodule
```

- rf 模块

该模块为寄存器堆的抽象。同样使用 verilog 中内置的寄存器数组模拟存储器，一共 32 个寄存器。

  - input
    - clk，rst：时钟和复位信号
    - we:1 位控制信号，写使能。只有在 we 值 1 且 clk 上升沿时，才进行数据写入。
    - RA1，RA2：4 位地址，选择 2 个待读的寄存器
    - WA：4 位地址，选择一个待写入的寄存器
    - WD：32 位，待写数据的值
  - output
    - RD1，RD2：32 位,读出数据的值
    - 其他 debug 信号(调式用)

代码实现如下：

```
module rf(
    input clk,
    input rst,
    input we,           //写使能
    input [4:0] RA1,    //读地址1（寄存器数组的下标）
    input [4:0] RA2,    //读地址2
    input [4:0] WA,     //写地址
    input [31:0] WD,    //写数据

    output [31:0] RD1,  //读出的数据1
    output [31:0] RD2,  //读出的数据2

    );

reg [31:0] regFiles [0:31];     //寄存器堆

always @(posedge clk) begin //时钟上升沿写数据
    if(rst==0 && we && WA>0 ) begin   //写使能置1且非0号寄存器
        regFiles[WA] <= WD;
    end
end

// 读寄存器数据
assign RD1=(RA1==0)? `DEFAULT_VAL: regFiles[RA1];
assign RD2=(RA2==0)? `DEFAULT_VAL: regFiles[RA2];

endmodule
```

- imm_gen 模块

由于 RISC-V 中，不同类型的指令的立即数在指令中分布位置不同，该模块实现对指令中立即数值的提取功能，并直接在该模块中直接将立即数拓展为 32

位。

- input
  - inst：32 位，指令值
- output
  - imm：32 位，提取并拓展位数之后的立即数值

代码实现如下：

```
module imm_gen(
    input [31:0] inst,
    output reg [31:0] imm
    );

always@(*) begin
    case(inst[6:0])
       `OPCODE_I1,          // addi
       `OPCODE_I2: begin    // lw
        // 12位立即数符号拓展
           if(inst[31] == 1)
               imm = {20'hfffff,inst[31:20]};
           else imm =  {20'h0,inst[31:20]};
       end
       `OPCODE_S: begin     //sw
           if(inst[31] == 1)
               imm = {20'hfffff,inst[31:25],inst[11:7]};
           else imm = {20'h0,inst[31:25],inst[11:7]};
       end
       `OPCODE_B: begin     //beq,blt
           if(inst[31] == 1) imm[31:13] = 19'h7ffff;
           else imm[31:13] = 19'h0;
           imm[12:0] = {inst[31],inst[7],inst[30:25],inst[11:8],1'b0};
       end
       `OPCODE_J: begin     //jal
           if(inst[31] == 1)
               imm = {11'h7ff,inst[31],inst[19:12],inst[20],inst[30:21],1'h0};
           else imm = {11'h0,inst[31],inst[19:12],inst[20],inst[30:21],1'h0};
       end
    endcase
end
endmodule
```

- alu_mux 模块

alu 的 2 个输入值中，第二个输入值根据指令类型不同，可以取第二个寄存器或是指令中立即数的值，故该模块根据控制信号 alu_src 完成这一选择功能。

- input
  - alu_src：1 位控制信号。0 时，选择寄存器值 RD2；1 时，选择立即数值 imm
  - RD2：32 位，可能被选择的寄存器值
  - imm：32 位，可能被选择的操作数值

- output
  - res：32 位，选择输入 alu 的结果

代码实现如下：

```
module alu_mux(
    input alu_src,
    input [31:0] RD2,
    input [31:0] imm,
    output reg [31:0] res
    );

always @(*) begin
    case (alu_src)
        0: res=RD2;
        1: res=imm;
    endcase
end
endmodule
```

- reg_mux 模块

在本实验实现的指令集中，写回寄存器的值有 3 种不同的可能：alu 的运算结果 op_res、数据存储器读出的值 dmem，pc+4 的值 pc，故本模块根据控制信号 reg_src 进行该选择功能。

- input
  - reg_src：2 位控制信号。0 选择 alu 结果，1 选择数据存储器，2 选择 pc+4
  - op_res：32 位，为 alu 计算结果值
  - dmem：32 位，为数据存储器读出来的值
  - pc：32 位，为当前 pc+4 的值(对应 npc 模块中的 pc_4)
- output
  - res：选择写回寄存器的结果

代码实现如下：

```
module reg_mux(
    input [1:0] reg_src,
    input [31:0] op_res,
    input [31:0] dmem,
    input [31:0] pc,
    output reg [31:0] res
);

always @(*) begin
    case (reg_src)
        0: res = op_res;
        1: res = dmem;
        2: res = pc;
    endcase
end
endmodule
```

- alu 模块

  实现计算功能的模块。本实验中只涉及 +,-,|三种运算。

  - input
    - op：2 位控制信号。0 选择 +，1 选择 -,2 选择 |
    - in1：32 位，为第一个操作数
    - in2：32 位，为第二个操作数
  - output
    - ZF：1 位零标志位，用于 beq 的跳转判断
    - SF：1 位符号标志位，用于 blt 的跳转判断
    - res：32 位，为计算结果

  代码实现如下：

```
module alu(
    input [`ALU_LEN-1:0] op,
    input [31:0] in1,
    input [31:0] in2,
    output ZF,
    output SF,
    output reg [31:0] res
    );
    assign ZF=(res==0)?1:0;
    assign SF=res[31];

    always @(*) begin
        case (op)
            0: res=in1+in2;
            1: res=in1-in2;
            2: res=in1 | in2;
        endcase
    end
endmodule
```

(P.S. 由于之后的会展示对每条指令的仿真验证结果，覆盖所有路径，故子模块的仿真结果不再列出)

3.4 控制信号说明

对 CU 模块中输出的控制信号说明如下：(func3 和 func7 处的”\”不代表没用对应的 code 值，而是当前指令集中已经能够区分)

表格 2 CPU 模型控制信号列表

操作码	0110011	0010011	0110011	0000011	0100011	1100011	1100011	1101111	0010011
func3	000	\	000	\	\	000	100	\	110
func7	0000000	000	0100000	\	\	\	\	\	\

指令	add	addi	sub	lw	sw	beq	blt	jal	ori
alu_op	0	0	1	0	0	1	1	0	2
branch	0	0	0	0	0	2	3	1	0
reg_src	0	0	0	1	0	2	2	2	0
alu_src	0	1	0	1	1	0	0	0	1
reg_we	1	1	1	1	0	0	0	1	1
dmem_we	0	0	0	0	1	0	0	0	0

3.5 控制模块与顶层模块实现

- cu 模块

采用逻辑硬链接方式，使用 case 语句与 if else 语句，解析指令数据的 opcode，func3,func7 字段对指令类型进行识别，并生成相应的控制信号。

输入为从指令固定位置取出的 opcode，func3，func7 字段，输出为上表列出的所有控制信号。其中每个控制信号的功能已经在子模块实现部分中进行解释。

具体代码实现如下：

```
module cu(
    input [6:0] opcode,
    input [6:0] func7,
    input [2:0] func3,

    //跳转分支指令的选择信号,0 表示不跳转,1 表示 jal,2 表示 beq,3 表示 blt
    output reg [`BRANCH_LEN-1:0] branch,
    // ALU 第二个操作数的选择信号,0 表示第二个寄存器，1 表示立即数
    output reg alu_src,
    // 寄存器写回数据的选择信号，0 表示 alu 结果，1 表示数据存储器，2 表示 pc+4
    output reg [1:0] reg_src,
    // 运算符的选择信号,0 为+，1 为-,2 为|
    output reg [`ALU_LEN-1:0] alu_op,
    // 存储器写使能
    output reg dmem_we,
    // 寄存器堆写使能
    output reg reg_we
    );

// 进行指令译码，并输出控制信号
always@(*) begin
    case(opcode)
       `OPCODE_R: begin //add ,sub
```

```
            if(func7 == `FUNC7_ADD) alu_op = 0;
            else if(func7 == `FUNC7_SUB) alu_op = 1;
            branch = 0;
            reg_src = 0;
            alu_src = 0;
            reg_we = 1;
            dmem_we = 0;
        end
        `OPCODE_I1: begin //addi,ori
            if (func3==`FUNC3_ADDI) alu_op = 0;
            else if(func3==`FUNC3_ORI) alu_op=2;
            branch=0;
            reg_src=0;
            alu_src = 1;
            reg_we = 1;
            dmem_we = 0;
       end
       `OPCODE_I2: begin// lw
            alu_op = 0;
            branch=0;
            reg_src = 1;
            alu_src = 1;
            reg_we = 1;
            dmem_we = 0;
        end
       `OPCODE_S: begin// sw
            alu_op = 0;
            branch=0;
            reg_src = 0;
            alu_src = 1;
            reg_we = 0;
            dmem_we = 1;
        end
       `OPCODE_B: begin
           if(func3 == `FUNC3_BEQ) begin// beq
               alu_op = 1;
               branch=2;
               reg_src = 2;
               alu_src = 0;
               reg_we = 0;
               dmem_we = 0;
            end
            else if(func3 == `FUNC3_BLT) begin //blt
               alu_op = 1;
```

```
                branch=3;
                reg_src = 2;
                alu_src = 0;
                reg_we = 0;
                dmem_we = 0;
            end
        end
        7'b1101111: begin//jal
            alu_op = 0;
            branch=1;
            reg_src = 2;
            alu_src = 0;
            reg_we = 1;
            dmem_we = 0;
        end
    endcase
end
endmodule
```

- top_cpu 模块

该模块为顶层模块，完成在所有子模块中“接线”，调用各个子模块的功能，并且输出一列 debug 信号。(PS. 为了展示，子模块中的输出 debug 信号在之前的代码演示中被暂时删去)

```
module top_cpu(
    input clk,
    input rst,

    // debug
    output [31:0] data1,
    output [31:0] data2,
    output [31:0] data3,
    output [31:0] data4,
    output [31:0] data5,
    output [31:0] debug_reg1,
    output [31:0] debug_reg2,
    output [31:0] debug_reg3,
    output [31:0] nowpc,
    output [31:0] inst_d,
    output [31:0] res_d,
    output [31:0] imm_d,
    output alu_src_d,
    output [31:0] RD22_d
    );
```

定义了一系列 wire 型变量，链接各个子模块中的输入、输出端口，将所有子模块连成一个整体。

```
wire [31:0] pc, npc, pc_4;
wire [31:0] inst;

wire [`BRANCH_LEN-1:0] branch;
wire [`ALU_LEN-1:0] alu_op;
wire [1:0] reg_src;
wire alu_src;
wire ZF,SF;
wire dmem_we,reg_we;
reg jump;

wire [31:0] RD1,RD2,WD;
wire [31:0] imm;    //immgen
wire [31:0] res;    // 计算结果
wire [31:0] RD22;    //alu mux

wire [31:0] RD;

wire [31:0] RF_RB;  // reg mux
```

其中，jump 信号是由 branch 信号进行生成的。结合 ZF 和 SF 标志进行 beq，blt 的跳转判断。

```
always @(*) begin
    case (branch)
        0:  jump=0;
        1:  jump=1;
        2:  jump=ZF?1:0;
        3:  jump=SF?1:0;
    endcase
end
```

各个子模块的连线情况如下所示

```
pc pc1(.clk(clk),
    .rst(rst),
    .npc(npc),
    .pc(pc));

npc npc1(
    .imm(imm),
    .jump(jump),
    .pc(pc),
    .npc(npc),
    .pc_4(pc_4)
);
```

```
im im1(
    .addr(pc[11:2]),
    .inst(inst)
);

cu cu1(
    .opcode(inst[6:0]),
    .func7(inst[31:25]),
    .func3(inst[14:12]),
    .branch(branch),
    .alu_src(alu_src),
    .reg_src(reg_src),
    .alu_op(alu_op),
    .dmem_we(dmem_we),
    .reg_we(reg_we)
);

rf rf1(
    .clk(clk),
    .rst(rst),
    .we(reg_we),
    .RA1(inst[19:15]),
    .RA2(inst[24:20]),
    .WA(inst[11:7]),
    .WD(RF_RB),
    .RD1(RD1),
    .RD2(RD2),
    .debug_reg1(debug_reg1),
    .debug_reg2(debug_reg2),
    .debug_reg3(debug_reg3)
);

imm_gen imm_gen1(
    .inst(inst),
    .imm(imm)
);

alu_mux alu_mux1(
    .alu_src(alu_src),
    .RD2(RD2),
    .imm(imm),
    .res(RD22)
);
```

```
alu alu1(
    .op(alu_op),
    .in1(RD1),
    .in2(RD22),
    .ZF(ZF),
    .SF(SF),
    .res(res)
);

dm dm1(
    .clk(clk),
    .rst(rst),
    .we(dmem_we),
    .addr(res[11:2]),
    .WD(RD2),
    .RD(RD),

    .data1(data1),
    .data2(data2),
    .data3(data3),
    .data4(data4),
    .data5(data5)
);

reg_mux reg_mux1(
    .reg_src(reg_src),
    .op_res(res),
    .dmem(RD),
    .pc(pc_4),
    .res(RF_RB)
);
```

3.6 指令仿真验证

在进行汇编程序测试之前，先对每条指令的波形进行仿真验证，并及时纠错。

- testbench 编写

testbench 中，除了 clk 与 rst，其他定义的 wire 型变量都是用于 debug，为了在波形中展示指令执行的结果。

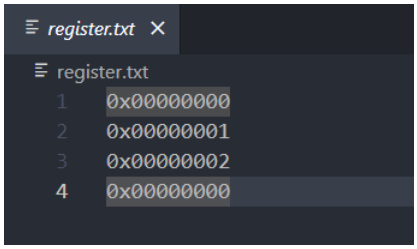
通过$readmemh 系统函数，读入 instructions.txt，register.txt 与 data_memory.txt 中的指令数据、寄存器初始值与内存存入的初始值到 im 模块、

rf 模块、dm 模块中对应的寄存器数组中，完成对存储器与寄存器的模拟。

```
module testbench();
reg clk;
reg rst;

wire [31:0] debug_reg1;
wire [31:0] debug_reg2;
wire [31:0] debug_reg3;
wire [31:0] pc;
wire [31:0] inst_d;
wire [31:0] data1;
wire [31:0] data2;
wire [31:0] data3;
wire [31:0] data4;
wire [31:0] data5;
// wire [31:0] res_d;
// wire [31:0] imm_d;
// wire alu_src_d;
// wire [31:0] RD22_d;

top_cpu top1(
.clk(clk),
.rst(rst),
//debug
.debug_reg1(debug_reg1),
.debug_reg2(debug_reg2),
.debug_reg3(debug_reg3),
.nowpc(pc),
.inst_d(inst_d),
.data1(data1),
.data2(data2),
.data3(data3),
.data4(data4),
.data5(data5)
);

initial begin
    // Load instructions
    $readmemh("D:/VivadoProject/sc-cpu/test/instructions.txt",
top1.im1.imem);
    // Load register initial values
```

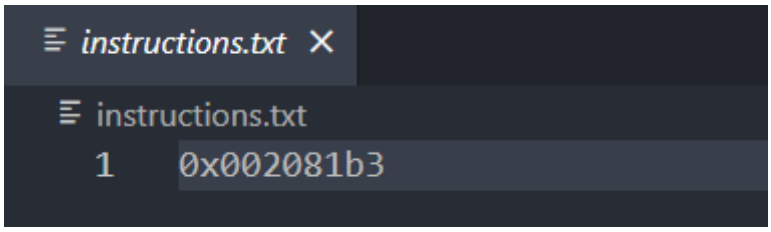
```
    $readmemh("D:/VivadoProject/sc-cpu/test/register.txt",
top1.rf1.regFiles);
    // Load memory data initial values
    $readmemh("D:/VivadoProject/sc-cpu/test/data_memory.txt",
top1.dm1.dmem);
    $display("value: %h",top1.im1.imem[0]);
    rst = 1;
    clk = 0;
    #30 rst = 0; // 30ns 前为 rst 阶段
end

always
    #20 clk = ~clk; // 时钟周期 20ns
endmodule
```

- add
  - register 初始值



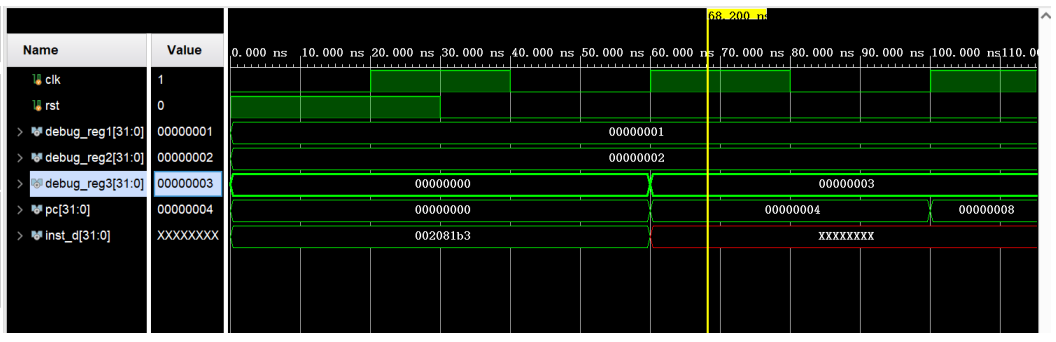
  - 输入指令

```
add x3,x1,x2
```



  - 仿真波形

    在时钟上升沿，成功得出结果并写入 x3



- addi
  - register 初始值

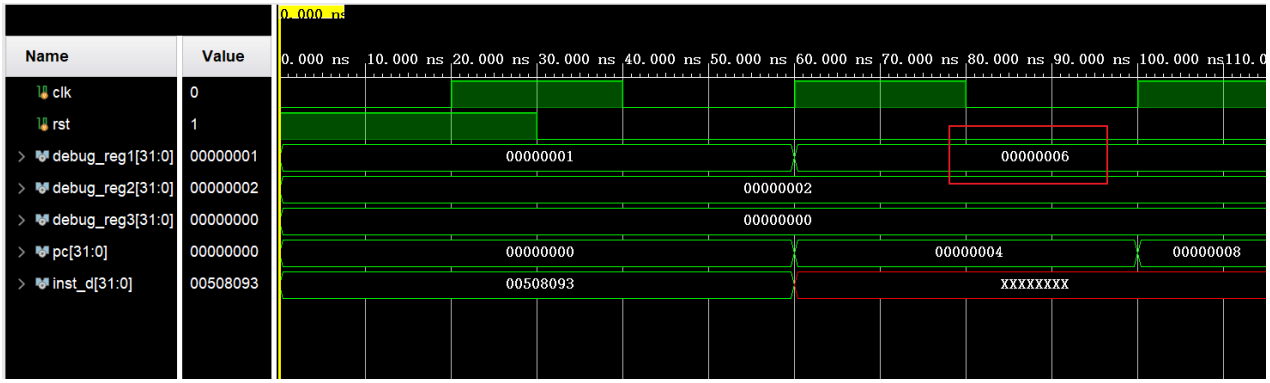
register.txt
```
0x00000000
0x00000001
0x00000002
0x00000000
```

  - 输入指令

```
addi x1, x1,5 （0x00508093）
```

  - 仿真波形

    在时钟上升沿，成功将 6 写入寄存器 x1 中



- sub
  - register 初始值

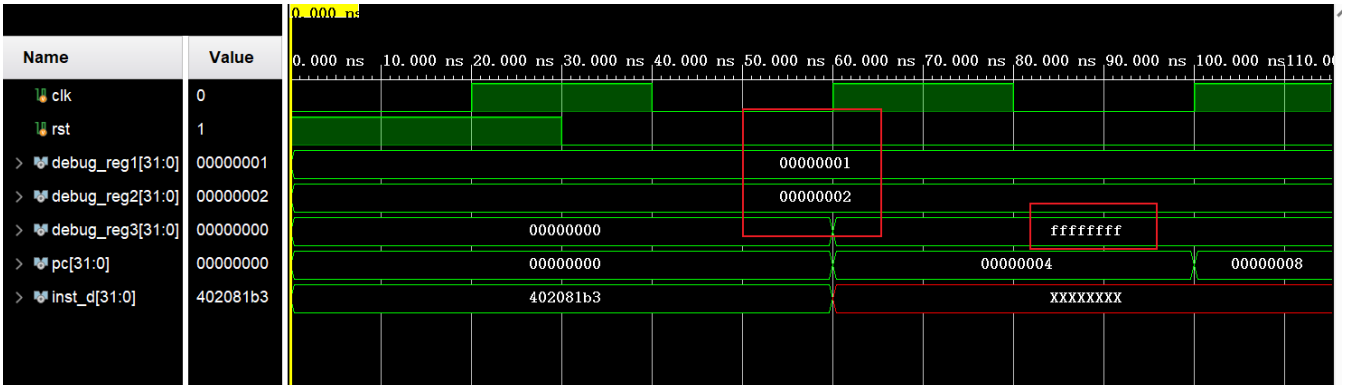
register.txt
```
0x00000000
0x00000001
0x00000002
0x00000000
```

  - 输入指令

```
sub x3,x1,x2 (0x402081b3)
```

  - 仿真波形

    1-2=-1，补码正是 0xffffffff，与 x3 中的结果符合。



- lw
  - register 初始值

register.txt
```
0x00000000
0x00000001
0x00000002
0x00000000
```

  - 存储数据

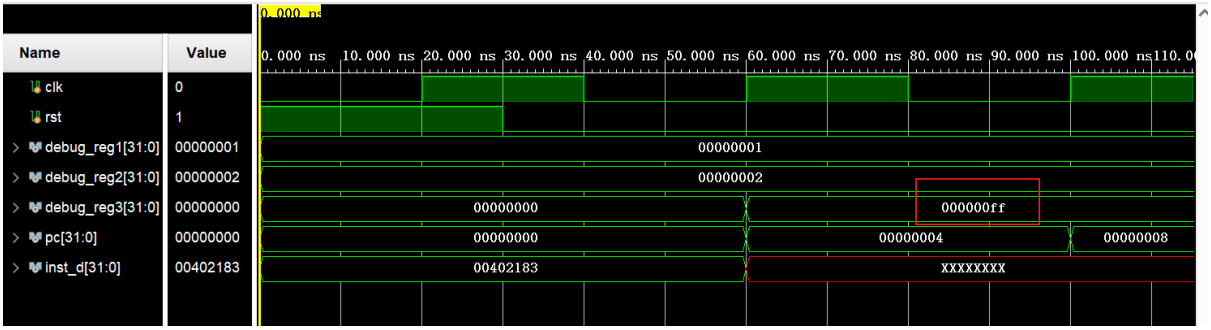
data_memory.txt
```
0x00000010
0x000000ff
```

  - 输入指令

```
lw x3,4(x0)（0x00402183）
```

  - 仿真波形

    存储器地址 0x4 对应的是 0xff，与 x3 中相同，成功加载数据到 x3 中。



- sw
  - register 初始值

register.txt
```
0x00000000
0x00000001
0x00000002
0x00000000
```

  - 存储数据

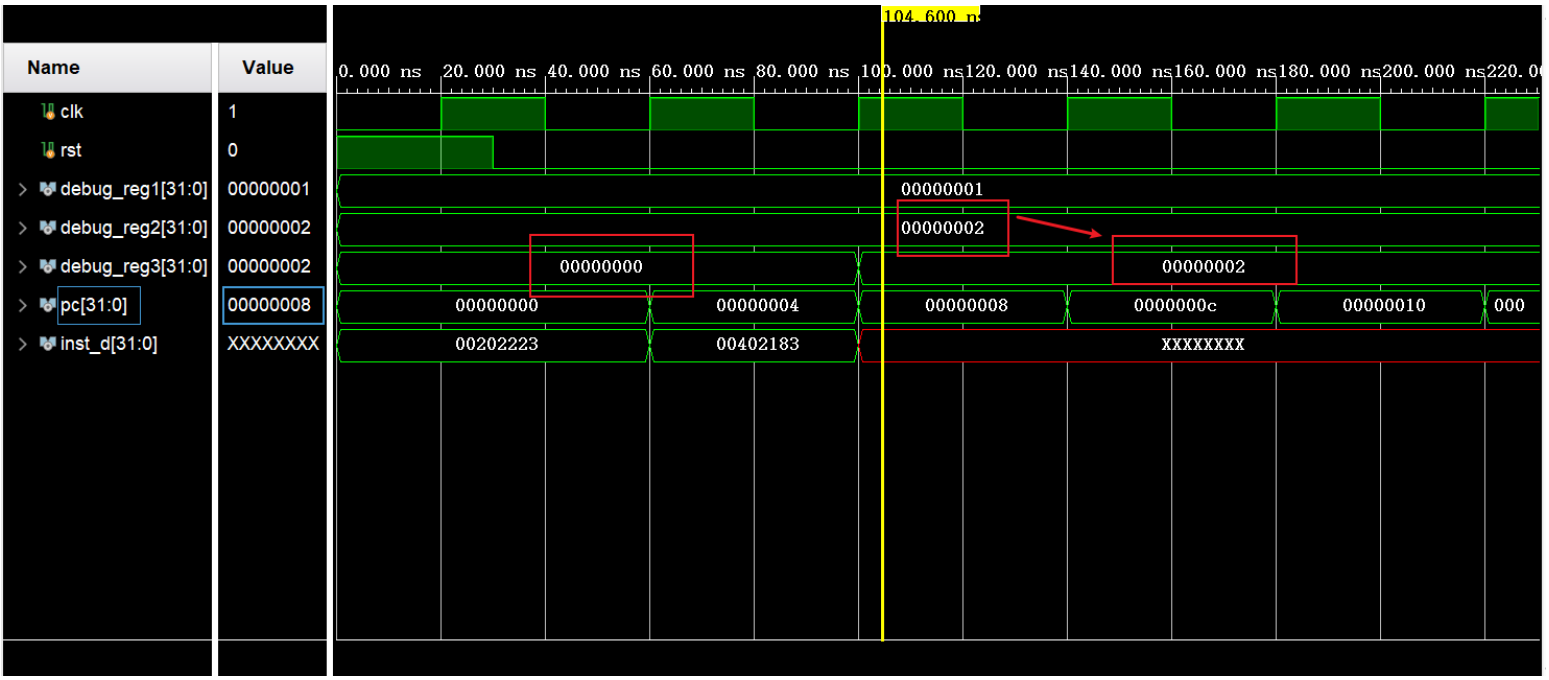
data_memory.txt
```
0x00000010
0x000000ff
```

- 输入指令

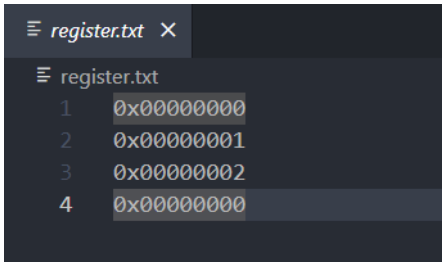
```
sw x2, 4(x0) (0x00202223)
lw x3, 4(x0) (0x00402183)
```

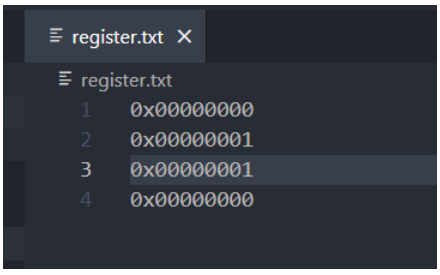
- 仿真波形

  指令执行的动作是先将 x2 的值写入 0x4 地址处，再将 0x4 地址处的值加载进 x3 中。观察波形，最后结果 x3 与 x2 相同，证明功能正确实现。



- beq(2 个输入测试)
  - register 初始值



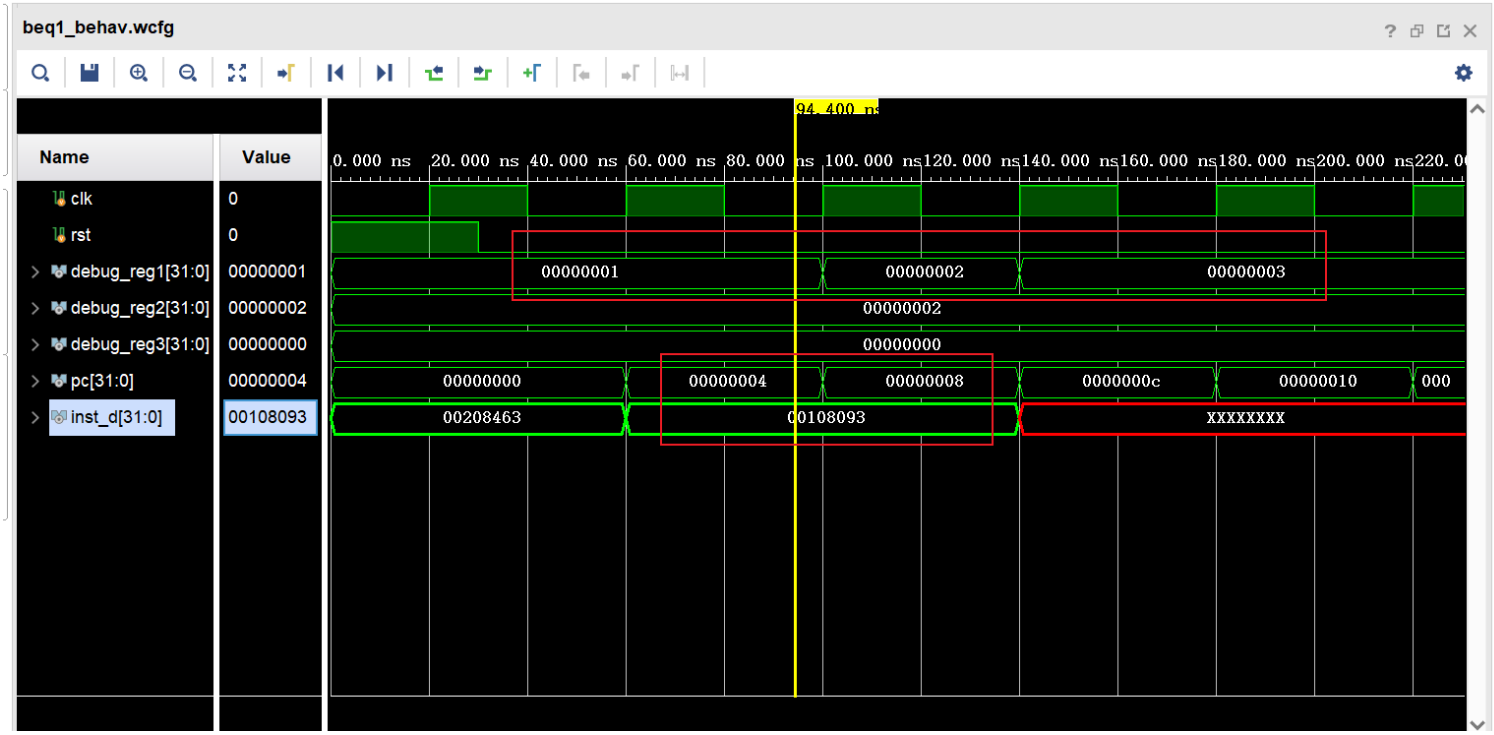


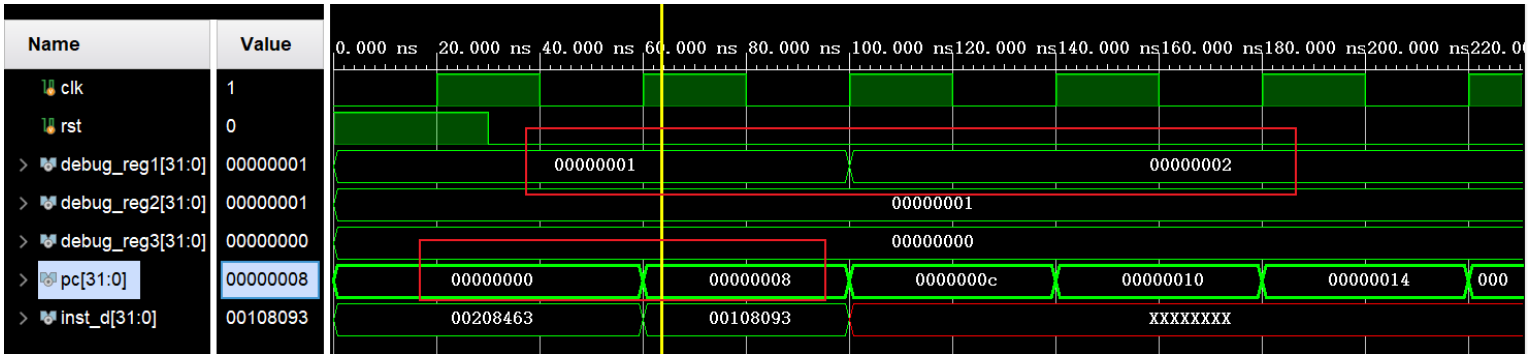
- 输入指令

```
beq x1,x2,L1  (0x00208463)
addi x1,x1,1  (0x00108093)
L1:
addi x1,x1,1  (0x00108093)
```

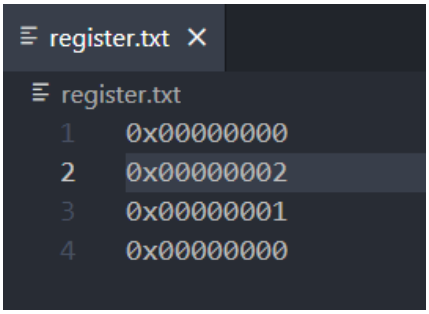
- 仿真结果

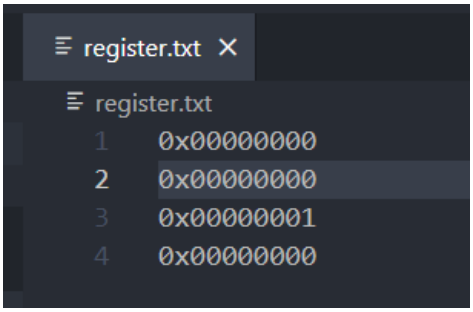
  第一组数据测试的是 x1 和 x2 值不等时不跳转的情况，第二组数据测试的是 x1 和 x2 值相等时跳过 x1=x1+1 的情况，故最后第二个测试结果中，x1 会比第一组的 x1 小 1。并且观察 pc 信号可以发现，第二组 pc 直接从 0x0 跳转到 0x8，跳过了 0x4，也证明了成功实现跳转判断。





- blt(2 个输入测试)
  - register 初始值



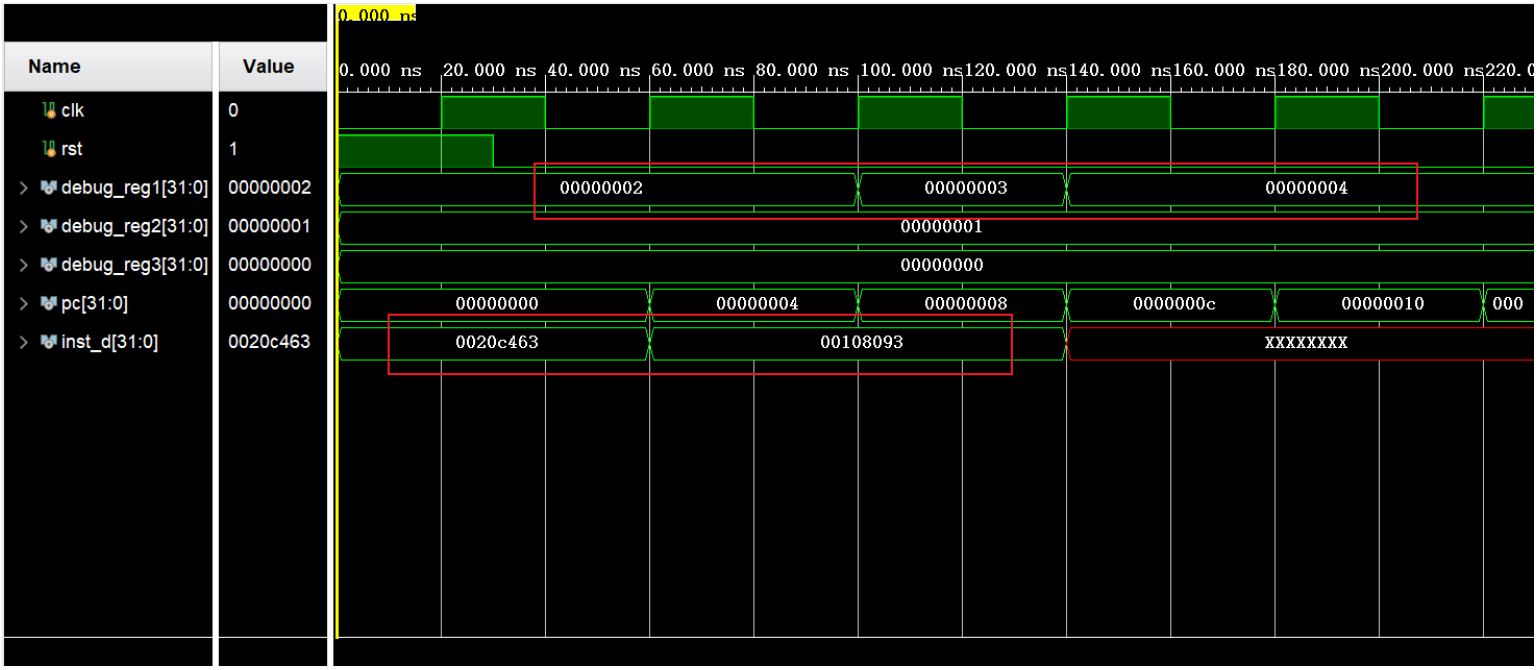


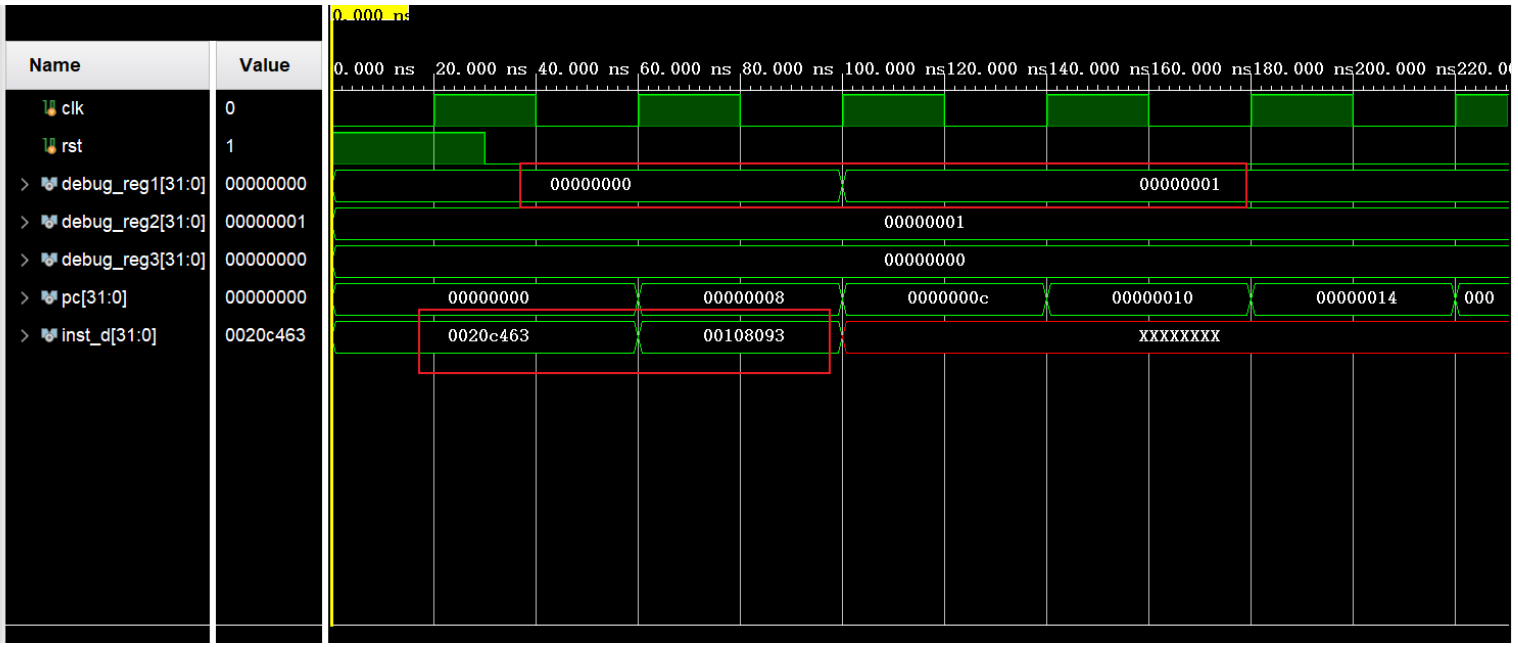
■ 输入指令

```
blt x1,x2,L1  (0x0020c463)
addi x1,x1,1  (0x00108093)
L1:
addi x1,x1,1  (0x00108093)
```

■ 仿真结果

第一组数据测试的是 x1>x2 时不跳转的情况，第二组数据测试的是 x1<x2 值相等时跳过 x1=x1+1 的情况，故最后第二个测试结果中，x1 的值只会加 1，而第一组测试中 x1 的值会加 2。并且观察 pc 信号可以发现，第二组 pc 直接从 0x0 跳转到 0x8，跳过了 0x4，也证明了成功实现跳转判断





- jal
  - register 初始值

register.txt

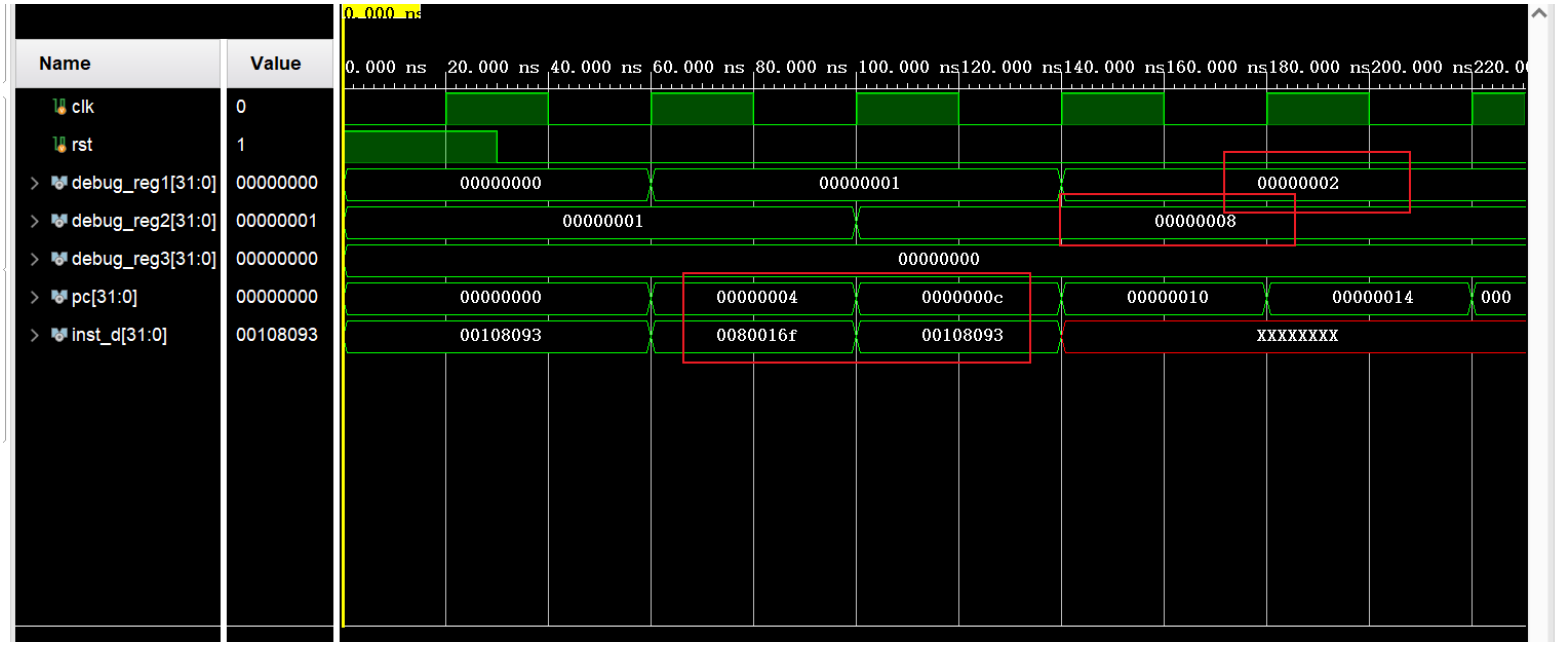
```
0x00000000
0x00000000
0x00000001
0x00000000
```

  - 输入指令

```
addi x1,x1,1  (0x00108093)
jal x2,L1     (0x0080016f)
addi x1,x1,1  (0x00108093)
L1:
addi x1,x1,1  (0x00108093)
```

  - 仿真波形

观察信号 pc 可发现，pc 值直接从 0x4 变为 0xc，跳过 0x8；并且跳转后 x2 的值变为了 0x8，成功实现跳转+返回地址存储。



- ori
  - register 初始值

register.txt

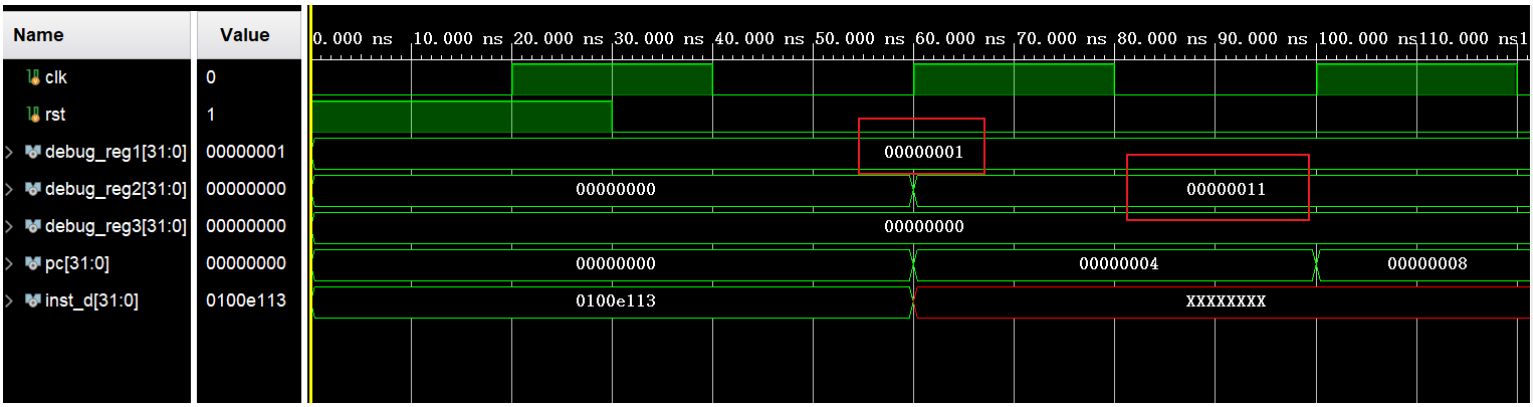
```
0x00000000
0x00000001
0x00000000
0x00000000
```

■ 输入指令

```
ori x2,x1,0x10 (0x0100e113)
```

■ 仿真波形

x1 的值为 0x1，与 0x10 按位与后，值为 0x11，存储到 x2 中，成功验证。



四、实验结果及分析

4.1 RISC-V 汇编程序编写

对于待处理数据的读取，采用数据初始化到数据存储器中的 0 地址开始的方法，并使用 lw 指令直接指定地址读入数据。注意的是，本实验中为了获取数组的长度值到寄存器中，存储器 0 地址存储的是数组字节数 20 (5*4)，0x4 地址开始才是待排序数组的首地址。

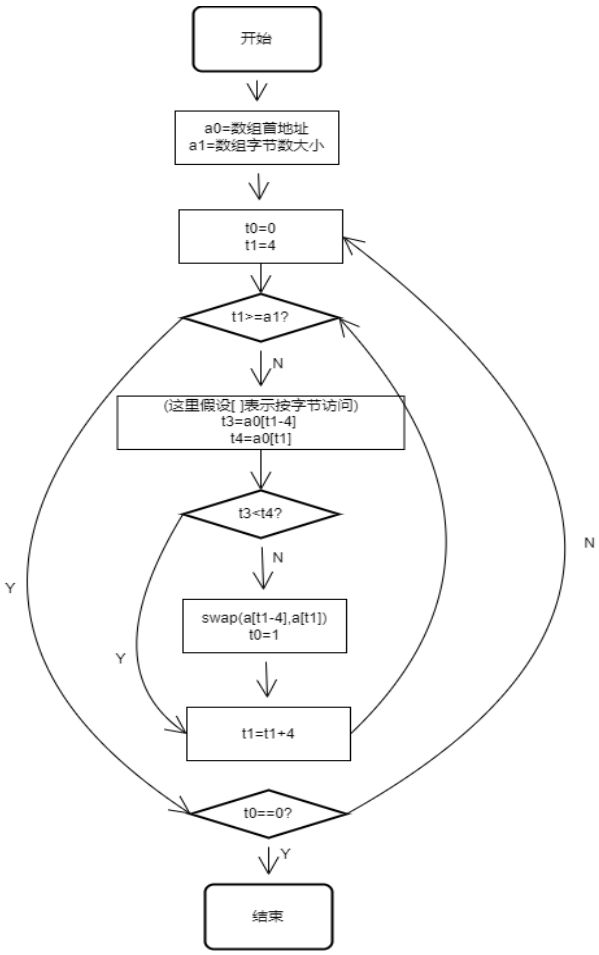
汇编代码实现如下，仅用本实验实现的指令集中的指令实现。输入数字为 5,1,3,4,2,期望得到从小到大的排序。

表格 3

```
.data
list:
    .word 20,5,1,3,4,2   # 20 为数组字节数
.text
    addi a0, x0,4     # a0 为数组元素首地址
    lw a1, 0(x0)     # a1 为待排序数组元素大小(5*4)
loop1_start:
    sub t0,t0,t0     # 标记本轮未发生排序
    addi t1,x0,4      # set t1 to 4
loop2_start:
    blt t1,a1,loop2_body
    jal x1,loop1_end
loop2_body:
    add t2,a0,t1
    lw t3,-4(t2)      # 取出相邻 2 个元素，交换
```

```
    lw t4,0(t2)
    blt t3,t4,loop2_end
    sw t3,0(t2)
    sw t4,-4(t2)
    addi t0,x0,1       # 标记本轮发生了排序
loop2_end:
    addi t1,t1,4
    jal x1,loop2_start
loop1_end:
    beq t0,x0,stop
    jal x1,loop1_start
stop:
```

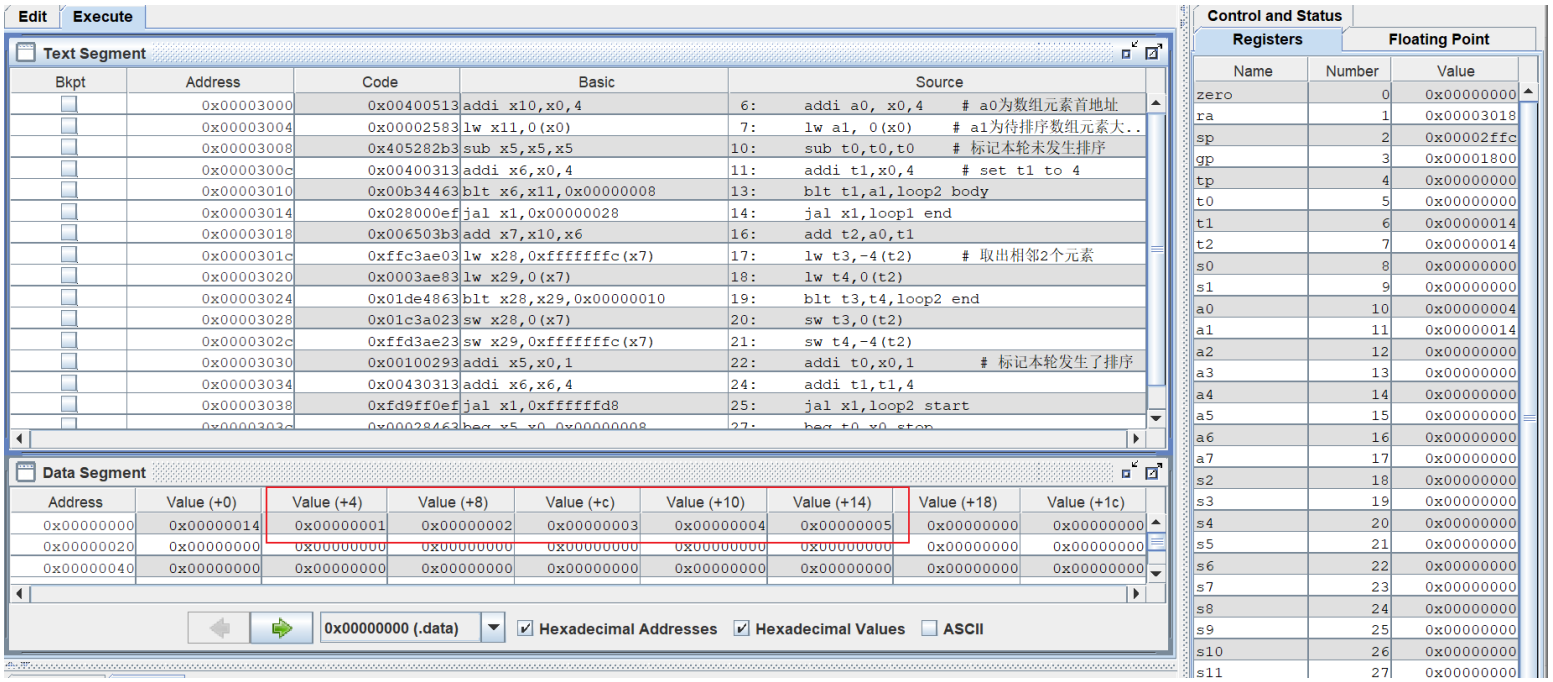
对应的程序流图如下所示



4.2 RARS 仿真调试

在 RARS 中设置内存数据段起始地址为 0 后，编译运行该汇编代码，得到的仿真调试结果如下：

可以看到，运行后的 data segment 中，数组重新排序为 1,2,3,4,5，结果正确。



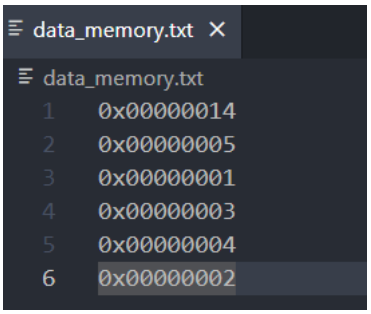
表格 4 指令与对应的机器码

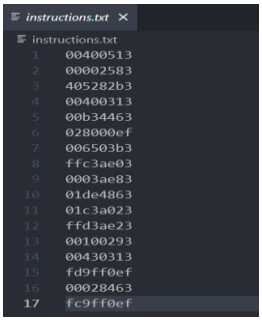
序号	指令格式	具体指令	func7	rs2	rs1	func3	Rd	Op
1	addi rd,rs1,imm12	addi x10,x0,4	\	\	0	00	10	0010011
2	lw rd,imm12(rs1)	lw x11,0(x0)	\	\	0	2	11	0000011
3	sub rd,rs1,rs2	sub x5,x5,x5	32	5	5	0	5	0110011
4	addi rd,rs1,imm12	addi x6,x0,4	\	\	0	0	6	0010011
5	blt rs1,rs2,imm12	blt x6,x11,0x8	\	11	6	4	\	1100011
6	jal rd,imm20	jal x1,0x28	\	\	\	\	1	1101111
7	add rd,rs1,rs2	add x7,x10,x6	0	6	10	0	7	0110011
8	lw rs2,imm12(rs1)	lw x28,0xfffffffc(x7)	\	28	7	2	\	0000011
9	lw rs2,imm12(rs1)	lw x29,0(x7)	\	29	7	2	\	0000011
10	blt rs1,rs2,imm12	blt x28,x29,0x00000010	\	29	28	4	\	1100011
11	sw rs2,imm12(rs1)	sw x28,0(x7)	\	28	7	2	\	0100011

12	sw rs2,imm12(rs1)	sw x29,0xfffffffc0(x7)	\	29	7	2	\	0100011
13	addi rd,rs1,imm12	addi x5,x0,1	\	\	0	0	5	0010011
14	addi rd,rs1,imm12	addi x6,x6,4	\	\	6	0	6	0010011
15	jal rd,imm20	jal x1,0xffffffd8	\	\	\	\	1	1101111
16	beq rs1,rs2,offset	beq x5,x0,0x00000008	\	0	5	0	\	1100011
17	jal rd,imm20	jal x1,0xffffffc8	\	\		\	1	1101111

4.3 Vivado 仿真

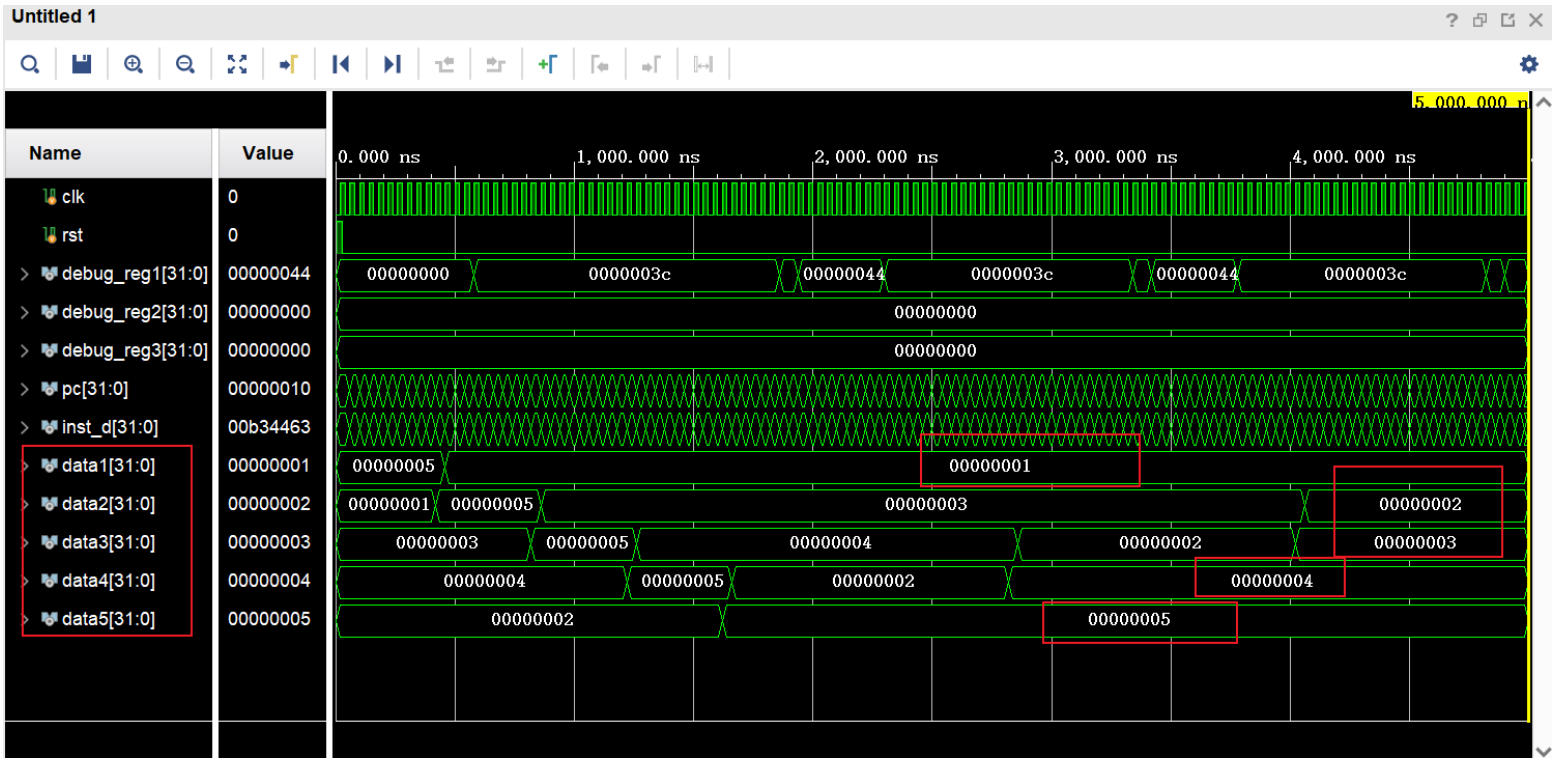
依旧使用验证指令时的 testbench.v，导出 RARS 编译得到的机器码，放入 instructions.txt 中，并将待排序写入到 data_memory.txt 中，如下图





运行 behavior simulation,得到仿真波形图如下

可以观察到，经过 5us 后，内存中数组已经从小到大排序完成，运行结果正确。



由上图，因为完成排序需要执行的指令数量很多，单独一个个在表里列出来直接对比寄存器的值会难以查看、对比，故观察待排序数组在存储单元里值的变化，进行验证。

表格 5 Vivado 仿真结果与 RARS 仿真结果比较

指令序列	单周期处理仿真结果	RARS 仿真结果
00400513 00002583 405282b3 00400313 00b34463 006503b3 ffc3ae03 0003ae83 01de4863 01c3a023	data[0]=5 data[1]=1 data[2]=3 data[3]=4 data[4]=2	data[0]=5 data[1]=1 data[2]=3 data[3]=4 data[4]=2
ffd3ae23	data[0]=5 data[1]=5 data[2]=3 data[3]=4 data[4]=2	data[0]=5 data[1]=5 data[2]=3 data[3]=4 data[4]=2
00100293 00430313 fd9ff0ef 00b34463 006503b3 ffc3ae03 0003ae83 01de4863 01c3a023	data[0]=1 data[1]=5 data[2]=3 data[3]=4 data[4]=2	data[0]=1 data[1]=5 data[2]=3 data[3]=4 data[4]=2
ffd3ae23	data[0]=1	data[0]=1

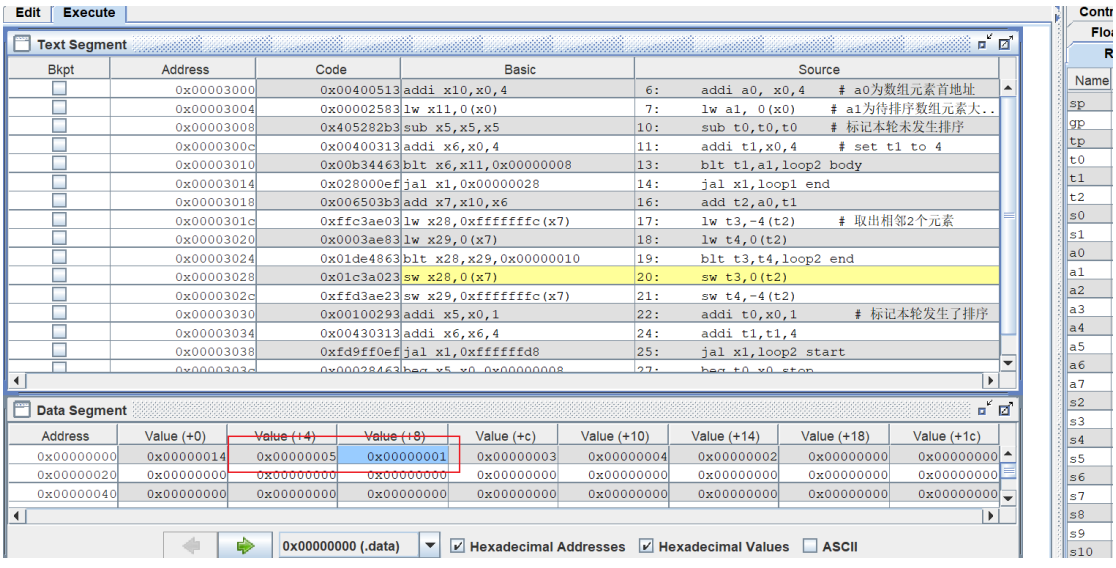
	data[1]=5 data[2]=5 data[3]=4 data[4]=2	data[1]=5 data[2]=5 data[3]=4 data[4]=2
00100293 00400313 fd9ff0ef 00b34463 006503b3 ffc3ae03 0003ae83 01de4863 01c3a023	data[0]=1 data[1]=3 data[2]=5 data[3]=4 data[4]=2	data[0]=1 data[1]=3 data[2]=5 data[3]=4 data[4]=2
ffd3ae23	data[0]=1 data[1]=3 data[2]=5 data[3]=5 data[4]=2	data[0]=1 data[1]=3 data[2]=5 data[3]=5 data[4]=2
00100293 00430313 fd9ff0ef 00b34463 006503b3 ffc3ae03 0003ae83 01de4863 01c3a023	data[0]=1 data[1]=3 data[2]=4 data[3]=5 data[4]=2	data[0]=1 data[1]=3 data[2]=4 data[3]=5 data[4]=2
ffd3ae23	data[0]=1 data[1]=3 data[2]=4 data[3]=5	data[0]=1 data[1]=3 data[2]=4 data[3]=5

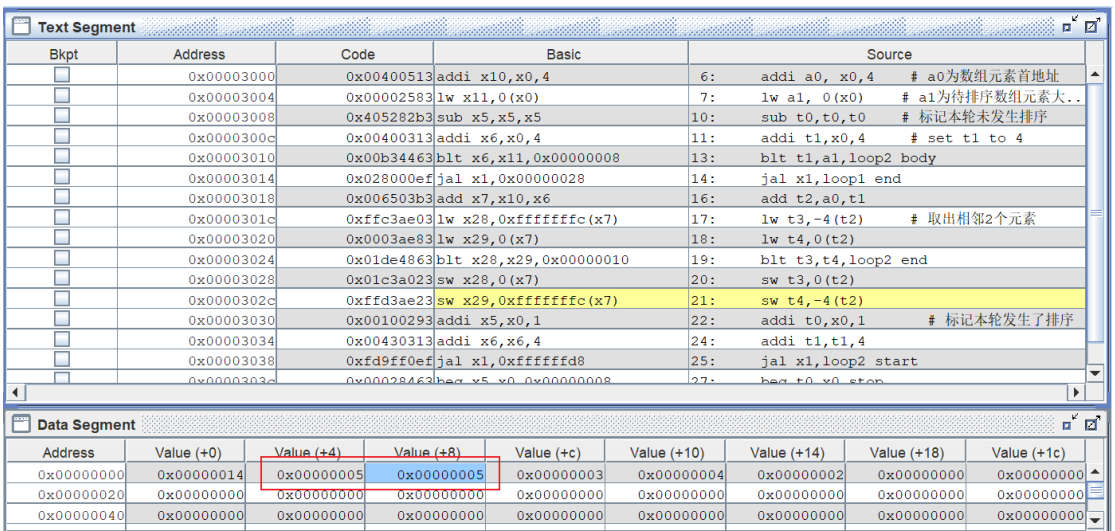
	data[4]=5	data[4]=5
00100293 00430313 fd9ff0ef 00b34463 028000ef 00028463 fc9ff0ef 405282b3 00430313 00b34463 006503b3 ffc3ae03 0003ae83 01de4863 00430313 fd9ff0ef 00b34463 006503b3 ffc3ae03 0003ae83 01de4863 00430313 fd9ff0ef 00b34463 006503b3 ffc3ae03 0003ae83 01de4863 01c3a023	data[0]=1 data[1]=3 data[2]=4 data[3]=2 data[4]=5	data[0]=1 data[1]=3 data[2]=4 data[3]=2 data[4]=5
ffd3ae23	data[0]=1	data[0]=1

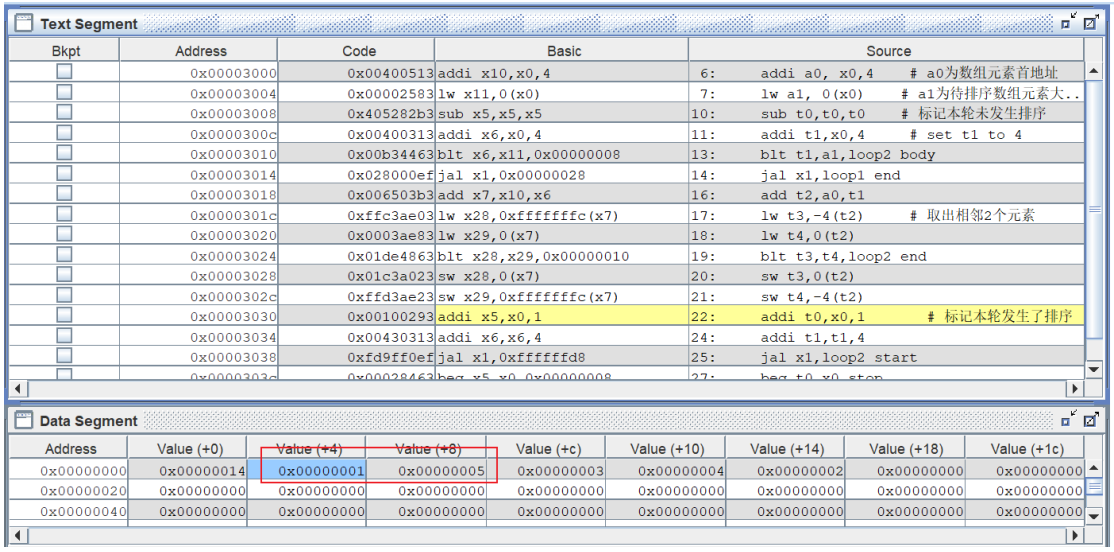
	data[1]=3 data[2]=4 data[3]=4 data[4]=5	data[1]=3 data[2]=4 data[3]=4 data[4]=5
00100293 00430313 fd9ff0ef 00b34463 006503b3 ffc3ae03 0003ae83 01de4863 00430313 fd9ff0ef 00b34463 028000ef 00028463 fc9ff0ef 405282b3 00400313 00b34463 006503b3 ffc3ae03 0003ae83 01de4863 00430313 fd9ff0ef 00b34463 006503b3 ffc3ae03 0003ae83	data[0]=1 data[1]=3 data[2]=2 data[3]=4 data[4]=5	data[0]=1 data[1]=3 data[2]=2 data[3]=4 data[4]=5

01de4863 01c3a023		
ffd3ae23	data[0]=1 data[1]=3 data[2]=3 data[3]=4 data[4]=5	data[0]=1 data[1]=3 data[2]=3 data[3]=4 data[4]=5
00100293 00430313 fd9ff0ef 00b34463 006503b3 ffc3ae03 … 000028463	data[0]=1 data[1]=2 data[2]=3 data[3]=4 data[4]=5	data[0]=1 data[1]=2 data[2]=3 data[3]=4 data[4]=5

附上 RARS 调试过程中的截图以证：





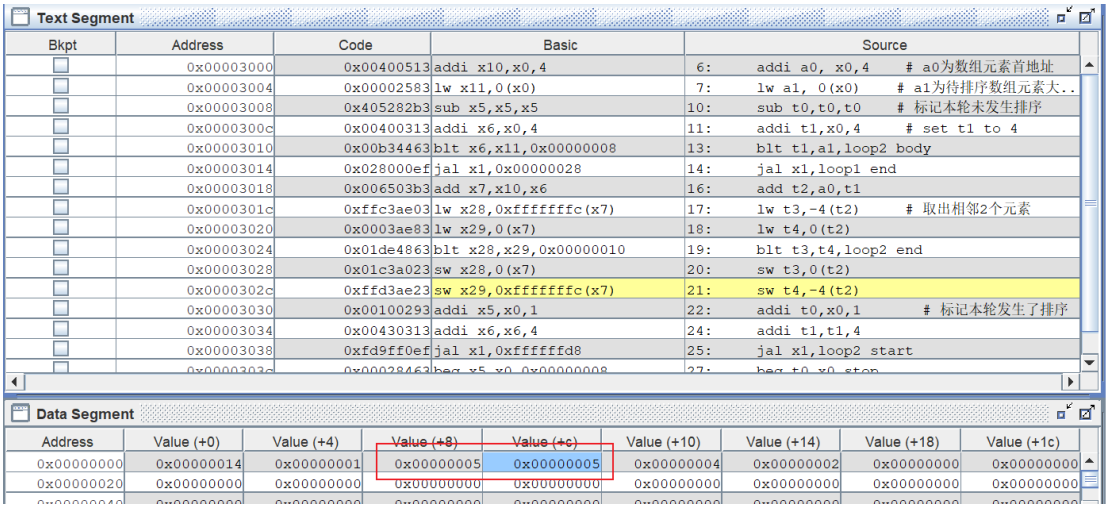


Text Segment

Bkpt	Address	Code	Basic		Source
	0x00003000	0x00400513	addi x10,x0,4	6:	addi a0, x0,4 # a0为数组元素首地址
	0x00003004	0x00002583	lw x11,0(x0)	7:	lw a1, 0(x0) # a1为待排序数组元素大..
	0x00003008	0x405282b3	sub x5,x5,x5	10:	sub t0,t0,t0 # 标记本轮未发生排序
	0x0000300c	0x00400313	addi x6,x0,4	11:	addi t1,x0,4 # set t1 to 4
	0x00003010	0x00b34463	blt x6,x11,0x00000008	13:	blt t1,a1,loop2 body
	0x00003014	0x028000ef	jal x1,0x00000028	14:	jal x1,loop1 end
	0x00003018	0x006503b3	add x7,x10,x6	16:	add t2,a0,t1
	0x0000301c	0xffc3ae03	lw x28,0xfffffffc(x7)	17:	lw t3,-4(t2) # 取出相邻2个元素
	0x00003020	0x0003ae83	lw x29,0(x7)	18:	lw t4,0(t2)
	0x00003024	0x01de4863	blt x28,x29,0x00000010	19:	blt t3,t4,loop2 end
	0x00003028	0x01c3a023	sw x28,0(x7)	20:	sw t3,0(t2)
	0x0000302c	0xffd3ae23	sw x29,0xfffffffc(x7)	21:	sw t4,-4(t2)
	0x00003030	0x00100293	addi x5,x0,1	22:	addi t0,x0,1 # 标记本轮发生了排序
	0x00003034	0x00430313	addi x6,x6,4	24:	addi t1,t1,4
	0x00003038	0xfd9ff0ef	jal x1,0xfffffd8	25:	jal x1,loop2 start

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000014	0x00000001	0x00000005	0x00000003	0x00000004	0x00000002	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

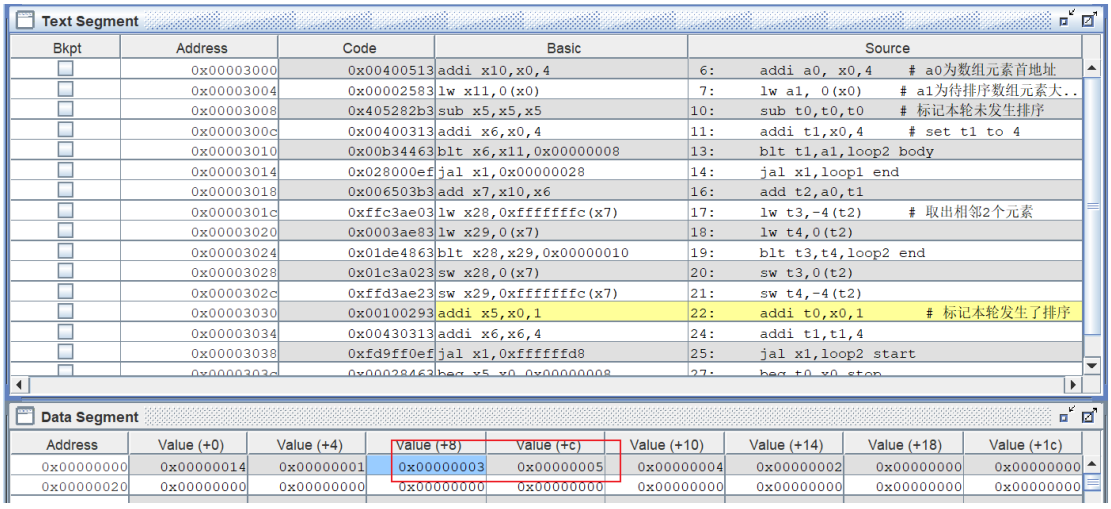


Text Segment

Bkpt	Address	Code	Basic		Source
	0x00003000	0x00400513	addi x10,x0,4	6:	addi a0, x0,4 # a0为数组元素首地址
	0x00003004	0x00002583	lw x11,0(x0)	7:	lw a1, 0(x0) # a1为待排序数组元素大..
	0x00003008	0x405282b3	sub x5,x5,x5	10:	sub t0,t0,t0 # 标记本轮未发生排序
	0x0000300c	0x00400313	addi x6,x0,4	11:	addi t1,x0,4 # set t1 to 4
	0x00003010	0x00b34463	blt x6,x11,0x00000008	13:	blt t1,a1,loop2 body
	0x00003014	0x028000ef	jal x1,0x00000028	14:	jal x1,loop1 end
	0x00003018	0x006503b3	add x7,x10,x6	16:	add t2,a0,t1
	0x0000301c	0xffc3ae03	lw x28,0xfffffffc(x7)	17:	lw t3,-4(t2) # 取出相邻2个元素
	0x00003020	0x0003ae83	lw x29,0(x7)	18:	lw t4,0(t2)
	0x00003024	0x01de4863	blt x28,x29,0x00000010	19:	blt t3,t4,loop2 end
	0x00003028	0x01c3a023	sw x28,0(x7)	20:	sw t3,0(t2)
	0x0000302c	0xffd3ae23	sw x29,0xfffffffc(x7)	21:	sw t4,-4(t2)
	0x00003030	0x00100293	addi x5,x0,1	22:	addi t0,x0,1 # 标记本轮发生了排序
	0x00003034	0x00430313	addi x6,x6,4	24:	addi t1,t1,4
	0x00003038	0xfd9ff0ef	jal x1,0xfffffd8	25:	jal x1,loop2 start

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000014	0x00000001	0x00000005	0x00000005	0x00000004	0x00000002	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

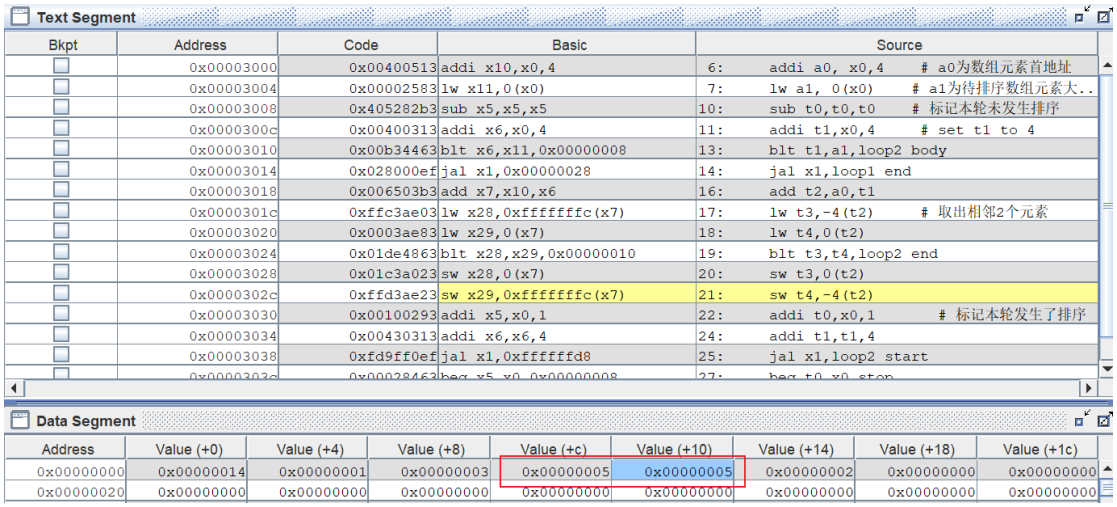


Text Segment

Bkpt	Address	Code	Basic		Source
	0x00003000	0x00400513	addi x10,x0,4	6:	addi a0, x0,4 # a0为数组元素首地址
	0x00003004	0x00002583	lw x11,0(x0)	7:	lw a1, 0(x0) # a1为待排序数组元素大..
	0x00003008	0x405282b3	sub x5,x5,x5	10:	sub t0,t0,t0 # 标记本轮未发生排序
	0x0000300c	0x00400313	addi x6,x0,4	11:	addi t1,x0,4 # set t1 to 4
	0x00003010	0x00b34463	blt x6,x11,0x00000008	13:	blt t1,a1,loop2 body
	0x00003014	0x028000ef	jal x1,0x00000028	14:	jal x1,loop1 end
	0x00003018	0x006503b3	add x7,x10,x6	16:	add t2,a0,t1
	0x0000301c	0xffc3ae03	lw x28,0xfffffffc(x7)	17:	lw t3,-4(t2) # 取出相邻2个元素
	0x00003020	0x0003ae83	lw x29,0(x7)	18:	lw t4,0(t2)
	0x00003024	0x01de4863	blt x28,x29,0x00000010	19:	blt t3,t4,loop2 end
	0x00003028	0x01c3a023	sw x28,0(x7)	20:	sw t3,0(t2)
	0x0000302c	0xffd3ae23	sw x29,0xfffffffc(x7)	21:	sw t4,-4(t2)
	0x00003030	0x00100293	addi x5,x0,1	22:	addi t0,x0,1 # 标记本轮发生了排序
	0x00003034	0x00430313	addi x6,x6,4	24:	addi t1,t1,4
	0x00003038	0xfd9ff0ef	jal x1,0xfffffd8	25:	jal x1,loop2 start

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000014	0x00000001	0x00000003	0x00000005	0x00000004	0x00000002	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

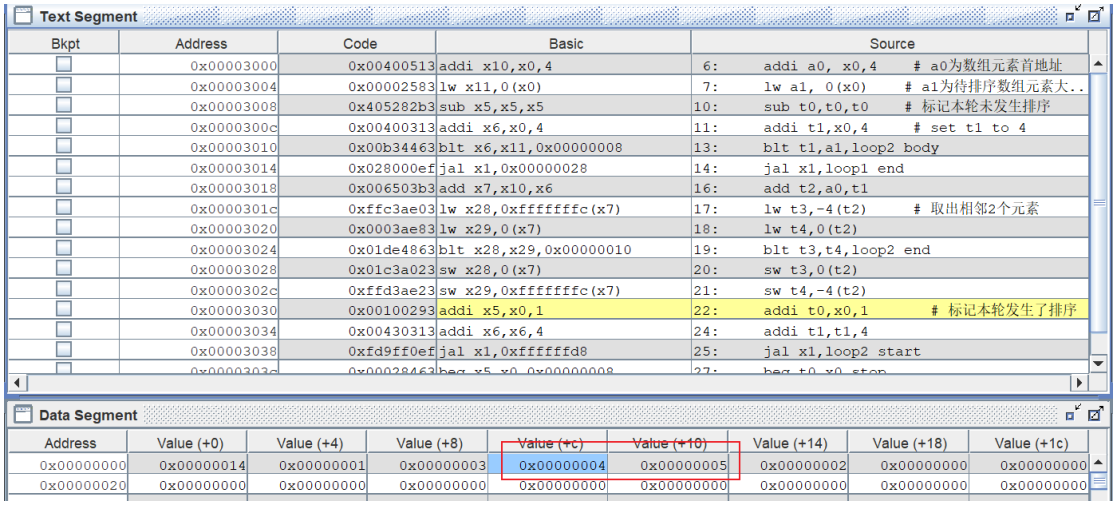


Text Segment

Bkpt	Address	Code	Basic	Source
	0x00003000	0x00400513	addi x10,x0,4	6: addi a0, x0,4 # a0为数组元素首地址
	0x00003004	0x00002583	lw x11,0(x0)	7: lw a1, 0(x0) # a1为待排序数组元素大..
	0x00003008	0x405282b3	sub x5,x5,x5	10: sub t0,t0,t0 # 标记本轮未发生排序
	0x0000300c	0x00400313	addi x6,x0,4	11: addi t1,x0,4 # set t1 to 4
	0x00003010	0x00b34463	blt x6,x11,0x00000008	13: blt t1,a1,loop2 body
	0x00003014	0x028000ef	jal x1,0x00000028	14: jal x1,loop1 end
	0x00003018	0x006503b3	add x7,x10,x6	16: add t2,a0,t1
	0x0000301c	0xffc3ae03	lw x28,0xfffffffc(x7)	17: lw t3,-4(t2) # 取出相邻2个元素
	0x00003020	0x0003ae83	lw x29,0(x7)	18: lw t4,0(t2)
	0x00003024	0x01de4863	blt x28,x29,0x00000010	19: blt t3,t4,loop2 end
	0x00003028	0x01c3a023	sw x28,0(x7)	20: sw t3,0(t2)
	0x0000302c	0xffd3ae23	sw x29,0xfffffffc(x7)	21: sw t4,-4(t2)
	0x00003030	0x00100293	addi x5,x0,1	22: addi t0,x0,1 # 标记本轮发生了排序
	0x00003034	0x00430313	addi x6,x6,4	24: addi t1,t1,4
	0x00003038	0xfd9ff0ef	jal x1,0xffffffd8	25: jal x1,loop2 start
	0x0000303c	0x00028463	beq x5,x0,0x00000008	27: beq t0,x0,stop

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000014	0x00000001	0x00000003	0x00000005	0x00000005	0x00000002	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

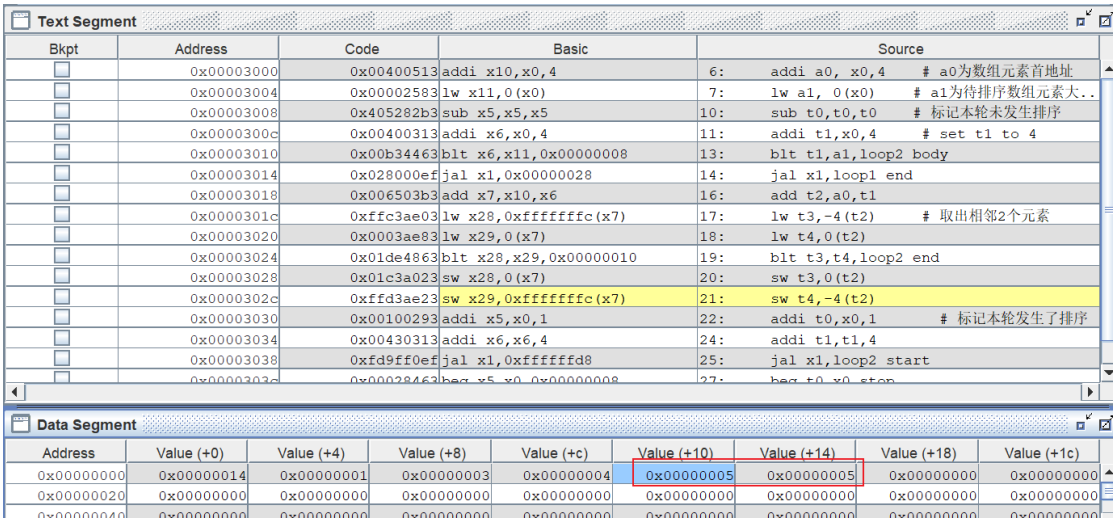


Text Segment

Bkpt	Address	Code	Basic	Source
	0x00003000	0x00400513	addi x10,x0,4	6: addi a0, x0,4 # a0为数组元素首地址
	0x00003004	0x00002583	lw x11,0(x0)	7: lw a1, 0(x0) # a1为待排序数组元素大..
	0x00003008	0x405282b3	sub x5,x5,x5	10: sub t0,t0,t0 # 标记本轮未发生排序
	0x0000300c	0x00400313	addi x6,x0,4	11: addi t1,x0,4 # set t1 to 4
	0x00003010	0x00b34463	blt x6,x11,0x00000008	13: blt t1,a1,loop2 body
	0x00003014	0x028000ef	jal x1,0x00000028	14: jal x1,loop1 end
	0x00003018	0x006503b3	add x7,x10,x6	16: add t2,a0,t1
	0x0000301c	0xffc3ae03	lw x28,0xfffffffc(x7)	17: lw t3,-4(t2) # 取出相邻2个元素
	0x00003020	0x0003ae83	lw x29,0(x7)	18: lw t4,0(t2)
	0x00003024	0x01de4863	blt x28,x29,0x00000010	19: blt t3,t4,loop2 end
	0x00003028	0x01c3a023	sw x28,0(x7)	20: sw t3,0(t2)
	0x0000302c	0xffd3ae23	sw x29,0xfffffffc(x7)	21: sw t4,-4(t2)
	0x00003030	0x00100293	addi x5,x0,1	22: addi t0,x0,1 # 标记本轮发生了排序
	0x00003034	0x00430313	addi x6,x6,4	24: addi t1,t1,4
	0x00003038	0xfd9ff0ef	jal x1,0xffffffd8	25: jal x1,loop2 start
	0x0000303c	0x00028463	beq x5,x0,0x00000008	27: beq t0,x0,stop

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000014	0x00000001	0x00000003	0x00000004	0x00000005	0x00000002	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

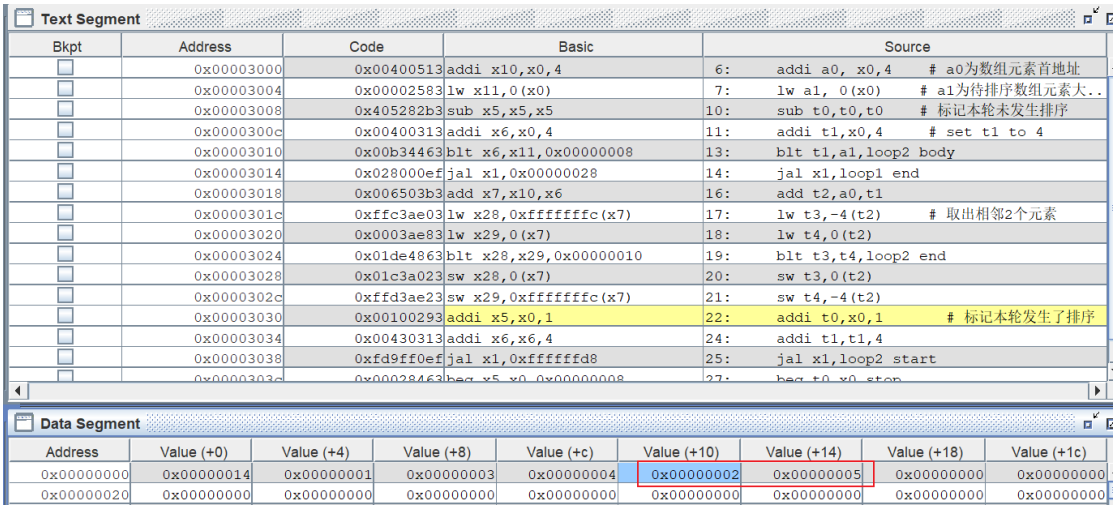


Text Segment

Bkpt	Address	Code	Basic	Source
	0x00003000	0x00400513	addi x10,x0,4	6: addi a0, x0,4 # a0为数组元素首地址
	0x00003004	0x00002583	lw x11,0(x0)	7: lw a1, 0(x0) # a1为待排序数组元素大..
	0x00003008	0x405282b3	sub x5,x5,x5	10: sub t0,t0,t0 # 标记本轮未发生排序
	0x0000300c	0x00400313	addi x6,x0,4	11: addi t1,x0,4 # set t1 to 4
	0x00003010	0x00b34463	blt x6,x11,0x00000008	13: blt t1,a1,loop2 body
	0x00003014	0x028000ef	jal x1,0x00000028	14: jal x1,loop1 end
	0x00003018	0x006503b3	add x7,x10,x6	16: add t2,a0,t1
	0x0000301c	0xffc3ae03	lw x28,0xfffffffc(x7)	17: lw t3,-4(t2) # 取出相邻2个元素
	0x00003020	0x0003ae83	lw x29,0(x7)	18: lw t4,0(t2)
	0x00003024	0x01de4863	blt x28,x29,0x00000010	19: blt t3,t4,loop2 end
	0x00003028	0x01c3a023	sw x28,0(x7)	20: sw t3,0(t2)
	0x0000302c	0xffd3ae23	sw x29,0xfffffffc(x7)	21: sw t4,-4(t2)
	0x00003030	0x00100293	addi x5,x0,1	22: addi t0,x0,1 # 标记本轮发生了排序
	0x00003034	0x00430313	addi x6,x6,4	24: addi t1,t1,4
	0x00003038	0xfd9ff0ef	jal x1,0xffffffd8	25: jal x1,loop2 start
	0x0000303c	0x00028463	beq x5,x0,0x00000008	27: beq t0,x0,stop

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000014	0x00000001	0x00000003	0x00000004	0x00000005	0x00000005	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

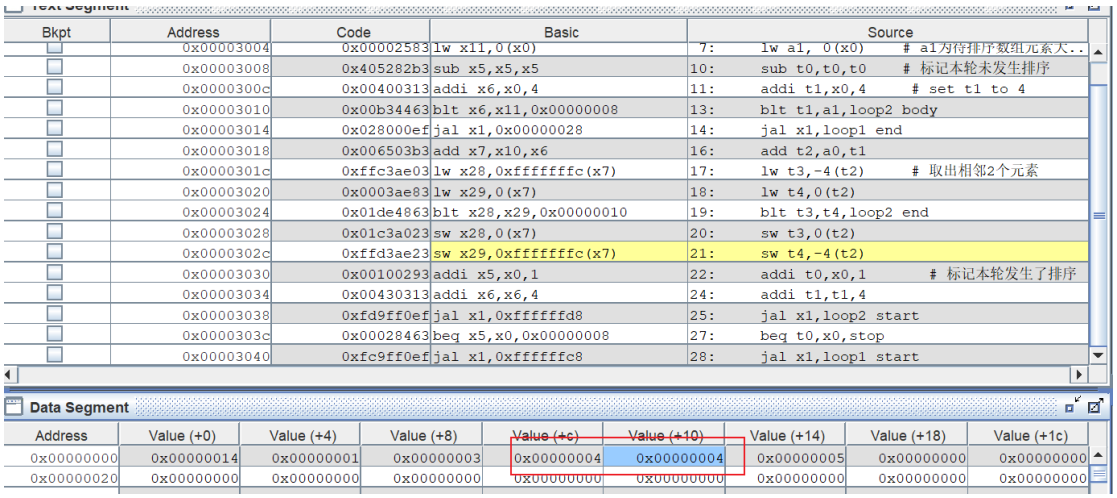


Text Segment

Bkpt	Address	Code	Basic	Source
	0x00003000	0x00400513	addi x10,x0,4	6: addi a0, x0,4 # a0为数组元素首地址
	0x00003004	0x00002583	lw x11,0(x0)	7: lw a1, 0(x0) # a1为待排序数组元素大..
	0x00003008	0x405282b3	sub x5,x5,x5	10: sub t0,t0,t0 # 标记本轮未发生排序
	0x0000300c	0x00400313	addi x6,x0,4	11: addi t1,x0,4 # set t1 to 4
	0x00003010	0x00b34463	blt x6,x11,0x00000008	13: blt t1,a1,loop2 body
	0x00003014	0x028000ef	jal x1,0x00000028	14: jal x1,loop1 end
	0x00003018	0x006503b3	add x7,x10,x6	16: add t2,a0,t1
	0x0000301c	0xffc3ae03	lw x28,0xfffffffc(x7)	17: lw t3,-4(t2) # 取出相邻2个元素
	0x00003020	0x0003ae83	lw x29,0(x7)	18: lw t4,0(t2)
	0x00003024	0x01de4863	blt x28,x29,0x00000010	19: blt t3,t4,loop2 end
	0x00003028	0x01c3a023	sw x28,0(x7)	20: sw t3,0(t2)
	0x0000302c	0xffd3ae23	sw x29,0xfffffffc(x7)	21: sw t4,-4(t2)
	0x00003030	0x00100293	addi x5,x0,1	22: addi t0,x0,1 # 标记本轮发生了排序
	0x00003034	0x00430313	addi x6,x6,4	24: addi t1,t1,4
	0x00003038	0xfd9ff0ef	jal x1,0xfffffffd8	25: jal x1,loop2 start
	0x0000303c	0x00028463	beq x5,x0,0x00000008	27: beq t0,x0,stop

Data Segment

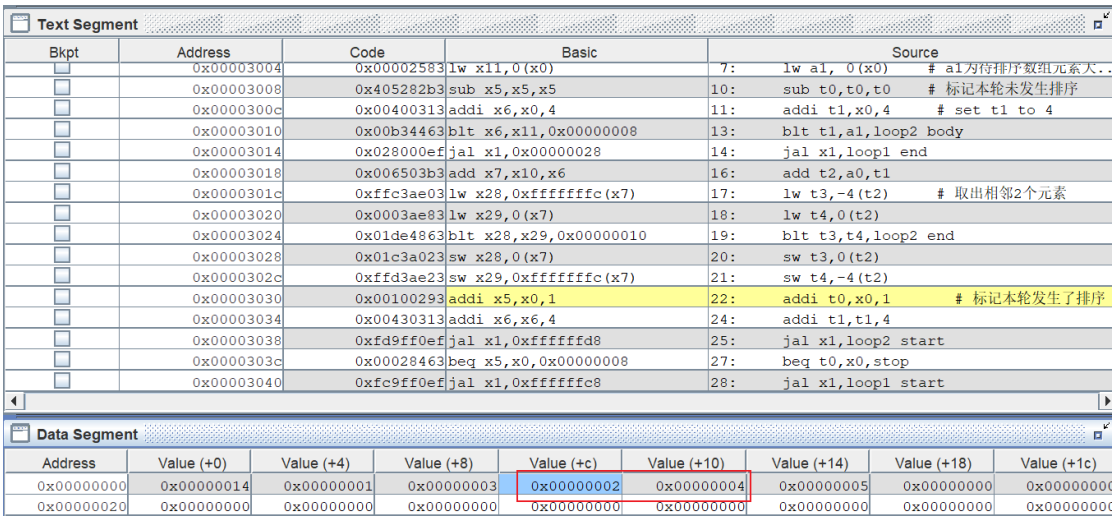
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000014	0x00000001	0x00000003	0x00000004	0x00000002	0x00000005	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000



Bkpt	Address	Code	Basic	Source
	0x00003004	0x00002583	lw x11,0(x0)	7: lw a1, 0(x0) # a1为待排序数组元素大..
	0x00003008	0x405282b3	sub x5,x5,x5	10: sub t0,t0,t0 # 标记本轮未发生排序
	0x0000300c	0x00400313	addi x6,x0,4	11: addi t1,x0,4 # set t1 to 4
	0x00003010	0x00b34463	blt x6,x11,0x00000008	13: blt t1,a1,loop2 body
	0x00003014	0x028000ef	jal x1,0x00000028	14: jal x1,loop1 end
	0x00003018	0x006503b3	add x7,x10,x6	16: add t2,a0,t1
	0x0000301c	0xffc3ae03	lw x28,0xfffffffc(x7)	17: lw t3,-4(t2) # 取出相邻2个元素
	0x00003020	0x0003ae83	lw x29,0(x7)	18: lw t4,0(t2)
	0x00003024	0x01de4863	blt x28,x29,0x00000010	19: blt t3,t4,loop2 end
	0x00003028	0x01c3a023	sw x28,0(x7)	20: sw t3,0(t2)
	0x0000302c	0xffd3ae23	sw x29,0xfffffffc(x7)	21: sw t4,-4(t2)
	0x00003030	0x00100293	addi x5,x0,1	22: addi t0,x0,1 # 标记本轮发生了排序
	0x00003034	0x00430313	addi x6,x6,4	24: addi t1,t1,4
	0x00003038	0xfd9ff0ef	jal x1,0xfffffffd8	25: jal x1,loop2 start
	0x0000303c	0x00028463	beq x5,x0,0x00000008	27: beq t0,x0,stop
	0x00003040	0xfc9ff0ef	jal x1,0xfffffffc8	28: jal x1,loop1 start

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000014	0x00000001	0x00000003	0x00000004	0x00000004	0x00000005	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

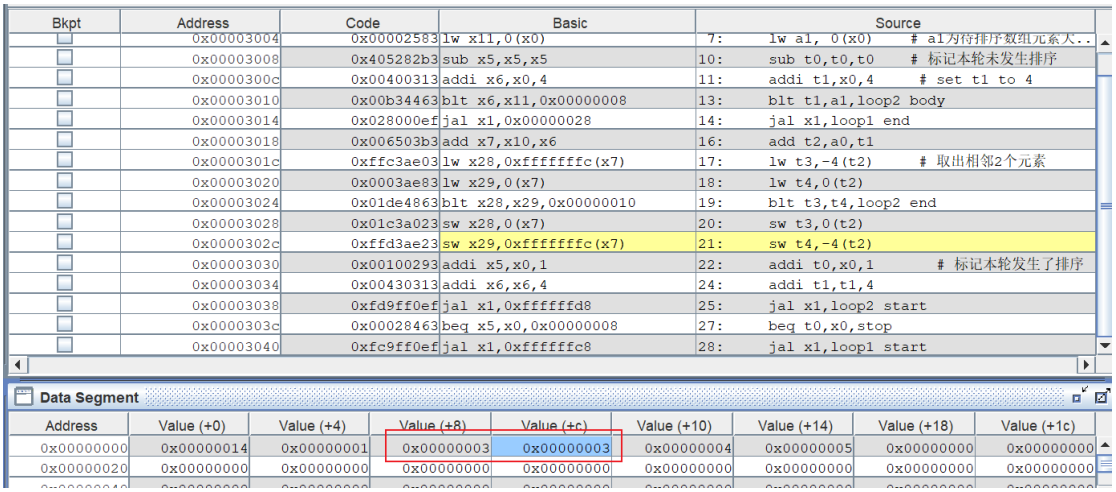


Text Segment

Bkpt	Address	Code	Basic	Source
	0x00003004	0x00002583	lw x11,0(x0)	7: lw a1, 0(x0) # a1为待排序数组元素大..
	0x00003008	0x405282b3	sub x5,x5,x5	10: sub t0,t0,t0 # 标记本轮未发生排序
	0x0000300c	0x00400313	addi x6,x0,4	11: addi t1,x0,4 # set t1 to 4
	0x00003010	0x00b34463	blt x6,x11,0x00000008	13: blt t1,a1,loop2 body
	0x00003014	0x028000ef	jal x1,0x00000028	14: jal x1,loop1 end
	0x00003018	0x006503b3	add x7,x10,x6	16: add t2,a0,t1
	0x0000301c	0xffc3ae03	lw x28,0xfffffffc(x7)	17: lw t3,-4(t2) # 取出相邻2个元素
	0x00003020	0x0003ae83	lw x29,0(x7)	18: lw t4,0(t2)
	0x00003024	0x01de4863	blt x28,x29,0x00000010	19: blt t3,t4,loop2 end
	0x00003028	0x01c3a023	sw x28,0(x7)	20: sw t3,0(t2)
	0x0000302c	0xffd3ae23	sw x29,0xfffffffc(x7)	21: sw t4,-4(t2)
	0x00003030	0x00100293	addi x5,x0,1	22: addi t0,x0,1 # 标记本轮发生了排序
	0x00003034	0x00430313	addi x6,x6,4	24: addi t1,t1,4
	0x00003038	0xfd9ff0ef	jal x1,0xfffffffd8	25: jal x1,loop2 start
	0x0000303c	0x00028463	beq x5,x0,0x00000008	27: beq t0,x0,stop
	0x00003040	0xfc9ff0ef	jal x1,0xfffffffc8	28: jal x1,loop1 start

Data Segment

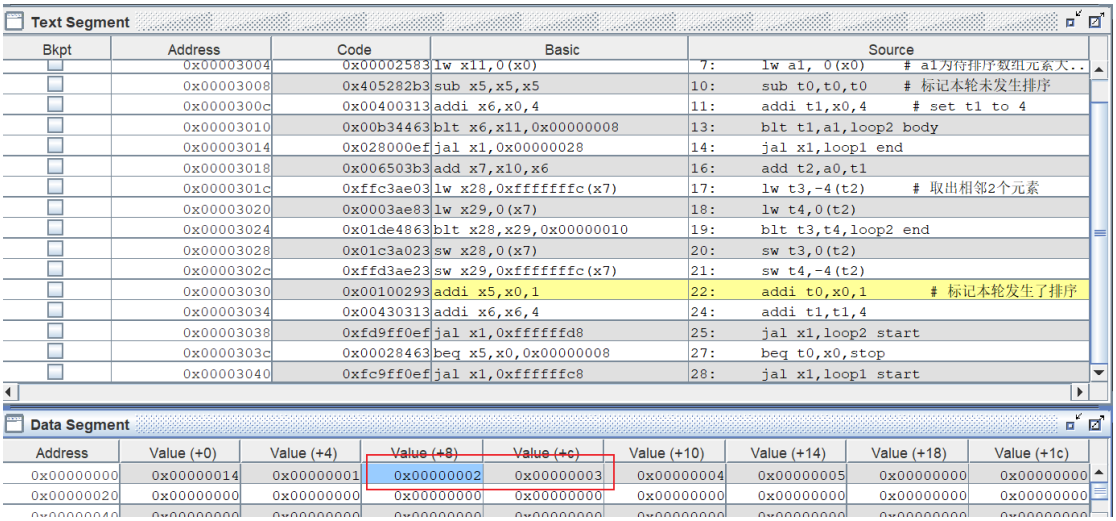
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000014	0x00000001	0x00000003	0x00000002	0x00000004	0x00000005	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000



Bkpt	Address	Code	Basic	Source
	0x00003004	0x00002583	lw x11,0(x0)	7: lw a1, 0(x0) # a1为待排序数组元素大..
	0x00003008	0x405282b3	sub x5,x5,x5	10: sub t0,t0,t0 # 标记本轮未发生排序
	0x0000300c	0x00400313	addi x6,x0,4	11: addi t1,x0,4 # set t1 to 4
	0x00003010	0x00b34463	blt x6,x11,0x00000008	13: blt t1,a1,loop2 body
	0x00003014	0x028000ef	jal x1,0x00000028	14: jal x1,loop1 end
	0x00003018	0x006503b3	add x7,x10,x6	16: add t2,a0,t1
	0x0000301c	0xffc3ae03	lw x28,0xfffffffc(x7)	17: lw t3,-4(t2) # 取出相邻2个元素
	0x00003020	0x0003ae83	lw x29,0(x7)	18: lw t4,0(t2)
	0x00003024	0x01de4863	blt x28,x29,0x00000010	19: blt t3,t4,loop2 end
	0x00003028	0x01c3a023	sw x28,0(x7)	20: sw t3,0(t2)
	0x0000302c	0xffd3ae23	sw x29,0xfffffffc(x7)	21: sw t4,-4(t2)
	0x00003030	0x00100293	addi x5,x0,1	22: addi t0,x0,1 # 标记本轮发生了排序
	0x00003034	0x00430313	addi x6,x6,4	24: addi t1,t1,4
	0x00003038	0xfd9ff0ef	jal x1,0xfffffffd8	25: jal x1,loop2 start
	0x0000303c	0x00028463	beq x5,x0,0x00000008	27: beq t0,x0,stop
	0x00003040	0xfc9ff0ef	jal x1,0xfffffffc8	28: jal x1,loop1 start

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000014	0x00000001	0x00000003	0x00000003	0x00000004	0x00000005	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000



Text Segment

Bkpt	Address	Code	Basic	Source
	0x00003004	0x00002583	lw x11,0(x0)	7: lw a1, 0(x0) # a1为待排序数组元素大..
	0x00003008	0x405282b3	sub x5,x5,x5	10: sub t0,t0,t0 # 标记本轮未发生排序
	0x0000300c	0x00400313	addi x6,x0,4	11: addi t1,x0,4 # set t1 to 4
	0x00003010	0x00b34463	blt x6,x11,0x00000008	13: blt t1,a1,loop2 body
	0x00003014	0x028000ef	jal x1,0x00000028	14: jal x1,loop1 end
	0x00003018	0x006503b3	add x7,x10,x6	16: add t2,a0,t1
	0x0000301c	0xffc3ae03	lw x28,0xfffffffc(x7)	17: lw t3,-4(t2) # 取出相邻2个元素
	0x00003020	0x0003ae83	lw x29,0(x7)	18: lw t4,0(t2)
	0x00003024	0x01de4863	blt x28,x29,0x00000010	19: blt t3,t4,loop2 end
	0x00003028	0x01c3a023	sw x28,0(x7)	20: sw t3,0(t2)
	0x0000302c	0xffd3ae23	sw x29,0xfffffffc(x7)	21: sw t4,-4(t2)
	0x00003030	0x00100293	addi x5,x0,1	22: addi t0,x0,1 # 标记本轮发生了排序
	0x00003034	0x00430313	addi x6,x6,4	24: addi t1,t1,4
	0x00003038	0xfd9ff0ef	jal x1,0xfffffffd8	25: jal x1,loop2 start
	0x0000303c	0x00028463	beq x5,x0,0x00000008	27: beq t0,x0,stop
	0x00003040	0xfc9ff0ef	jal x1,0xfffffffc8	28: jal x1,loop1 start

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000014	0x00000001	0x00000002	0x00000003	0x00000004	0x00000005	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000