Lab 5 流水线 CPU report

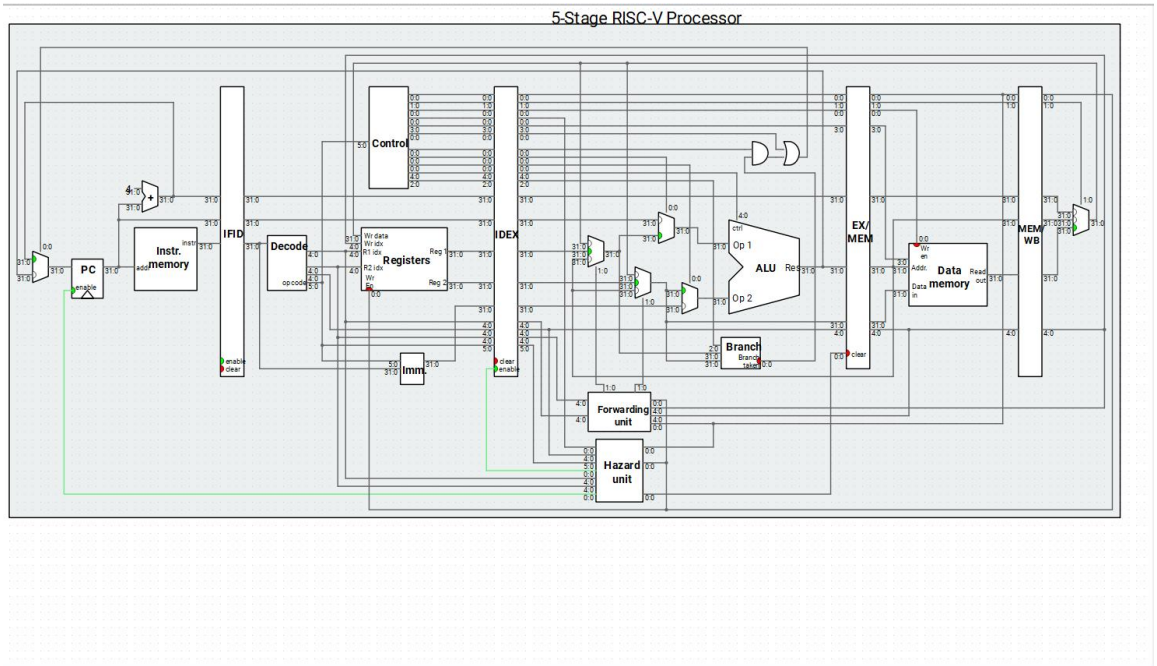
PB21111681 朱炜荣

实验目的与内容

1. 理解五级流水线 CPU 的结构和工作原理
2. 掌握流水线 CPU 的设计和调试方法，特别是五级流水线中的数据相关和控制相关的处理
3. 掌握使用 Verilog 来设计和描述流水线 CPU 的数据通路和控制器的方法

逻辑设计

1. 仿照的数据通路如下图所示：



2. 流水线的实现过程为在每一个时钟周期各个模块都在处理指令代码。所以程序中并没有有限状态机，也不存在状态转移。每个时钟周期 IM 以 PC 输出的数值作为地址读取指令，并将对应的 PC、NPC、IR 存入到段间寄存器中，以此类推，共存在四个段间寄存器来存储未来可能需要用到的数值。
3. 以下为较为核心的代码，即 CPU 的数据通路连线中的两个冲突相关处理单元的实现。

数据前递单元：

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2023/05/18 10:42:54
// Design Name:
// Module Name: MEM_WB
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Forward_Unit(
    input [31:0] ir_w,
    input [31:0] ir_m,
    input [31:0] ir_e,
    input we_w,
    input we_m,
    output reg [1:0] rf_choice_1,
    output reg [1:0] rf_choice_2
);
// wire [6:0] op_w;
// wire [6:0] op_m;
wire [6:0] op_e = ir_e[6:0];
wire [4:0] rd_w = ir_w[11:7];
wire [4:0] rd_m = ir_m[11:7];
// wire [4:0] rs1_m = ir_m[19:15];
// wire [4:0] rs2_m = ir_m[24:20];
wire [4:0] rs1_e = ir_e[19:15];
wire [4:0] rs2_e = ir_e[24:20];
always@(*)
begin
    rf_choice_1 = 2'b0;
    rf_choice_2 = 2'b0;
    case(op_e)
        7'b0010011:begin    //addi
```

```
            if(rd_w == rs1_e && we_w != 0)   rf_choice_1 = 2'b10;
            if(rd_m == rs1_e && we_m != 0)   rf_choice_1 = 2'b01;
        end
        7'b1100111:begin    //jalr
            if(rd_w == rs1_e && we_w != 0)   rf_choice_1 = 2'b10;
            if(rd_m == rs1_e && we_m != 0)   rf_choice_1 = 2'b01;
        end
        7'b0000011:begin    //lw
            if(rd_w == rs1_e && we_w != 0)   rf_choice_1 = 2'b10;
            if(rd_m == rs1_e && we_m != 0)   rf_choice_1 = 2'b01;
        end
        7'b1100011:begin    //beq
            if(rd_w == rs1_e && we_w != 0)   rf_choice_1 = 2'b10;
            if(rd_w == rs2_e && we_w != 0)   rf_choice_2 = 2'b10;
            if(rd_m == rs1_e && we_m != 0)   rf_choice_1 = 2'b01;
            if(rd_m == rs2_e && we_m != 0)   rf_choice_2 = 2'b01;
        end

        7'b0110011:begin    //add
            if(rd_w == rs1_e && we_w != 0)   rf_choice_1 = 2'b10;
            if(rd_w == rs2_e && we_w != 0)   rf_choice_2 = 2'b10;
            if(rd_m == rs1_e && we_m != 0)   rf_choice_1 = 2'b01;
            if(rd_m == rs2_e && we_m != 0)   rf_choice_2 = 2'b01;
        end

        7'b0100011:  begin  //sw
            if(rd_w == rs1_e && we_w != 0)   rf_choice_1 = 2'b10;
            if(rd_w == rs2_e && we_w != 0)   rf_choice_2 = 2'b10;
            if(rd_m == rs1_e && we_m != 0)   rf_choice_1 = 2'b01;
            if(rd_m == rs2_e && we_m != 0)   rf_choice_2 = 2'b01;
        end
    endcase
end
endmodule
```

数据前递的处理方法是，对于当前指令需要的源寄存器，分别与 MEM 和 WB 阶段指令的目的寄存器进行对比，同时需要判断是否是写回信号，否则会出现把 sw 地址读回的离奇情况。然后通过控制信号来选择 MUX 的数据从而达到前递的效果。

控制冲突单元：

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2023/05/18 10:42:54
// Design Name:
```

```
// Module Name: MEM_WB
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Hazard_Unit(
    input we_w,
    input we_m,
    input [31:0] ir_e,
    input [31:0] ir_d,
    input judge,
    output reg [2:0] bubbles,
    output reg [2:0] clear
);
wire [6:0] op_e = ir_e[6:0];
wire [6:0] op_d = ir_d[6:0];
wire [4:0] rd_e = ir_e[11:7];
wire [4:0] rs1_d = ir_d[19:15];
wire [4:0] rs2_d = ir_d[24:20];


always@(*)begin
    bubbles = 3'b111;
    clear = 3'b111;
    if(op_e == 7'b0000011)begin     //lw
        if(op_d == 7'b0010011 || op_d == 7'b000011)begin    //addi lw
            if(rd_e == rs1_d)begin
                bubbles = 3'b001;
                clear = 3'b101;
            end
        end
        else if(op_d == 7'b1100011 || op_d == 7'b0110011 || op_d == 0100011)begin    //beq sw add
            if(rd_e == rs1_d || rd_e == rs2_d)begin
                bubbles = 3'b001;
                clear = 3'b101;
            end
        end
```

```
    end
    if(op_e == 7'b1100011)begin     //beq
        if(judge)begin
            bubbles = 3'b111;
            clear = 3'b001;
        end
    end
    if(op_e == 7'b1100111 || op_e == 7'b1101111)begin
        if(judge)begin
            bubbles = 3'b111;
            clear = 3'b001;
        end
    end
end


Endmodule
```

而对于冲突检测，主要分为两个大类：flush 和 stall。

stall 的出现是因为 lw 指令需要再过一个周期才能将数据从数据存储器中读出，所以对于后来指令需要其数据的需要多等一个周期。所以要先判断 IR_M 是否为 lw，然后再判断 IR_E 中是否需要处理本应被写回寄存器的数值。其中 addi lw 等类型只有一个源寄存器，而 add beq sw 则需要两个源寄存器。一旦判断成功，控制信号便会 ID/EX 段间寄存器，PC 和 IF/ID 保持当前值一个周期。

flush 则是因为跳转成功时不应该处理顺序执行的两条指令，需要清除原有的操作转而去执行新 PC 处的指令。在程序中判断当前指令是否为 beq jal jalr 类型，如果为 beq 则还需要判断跳转是否成功。之后如果成功则清空 IF/ID 和 ID/EX 的段间寄存器。

仿真结果与分析

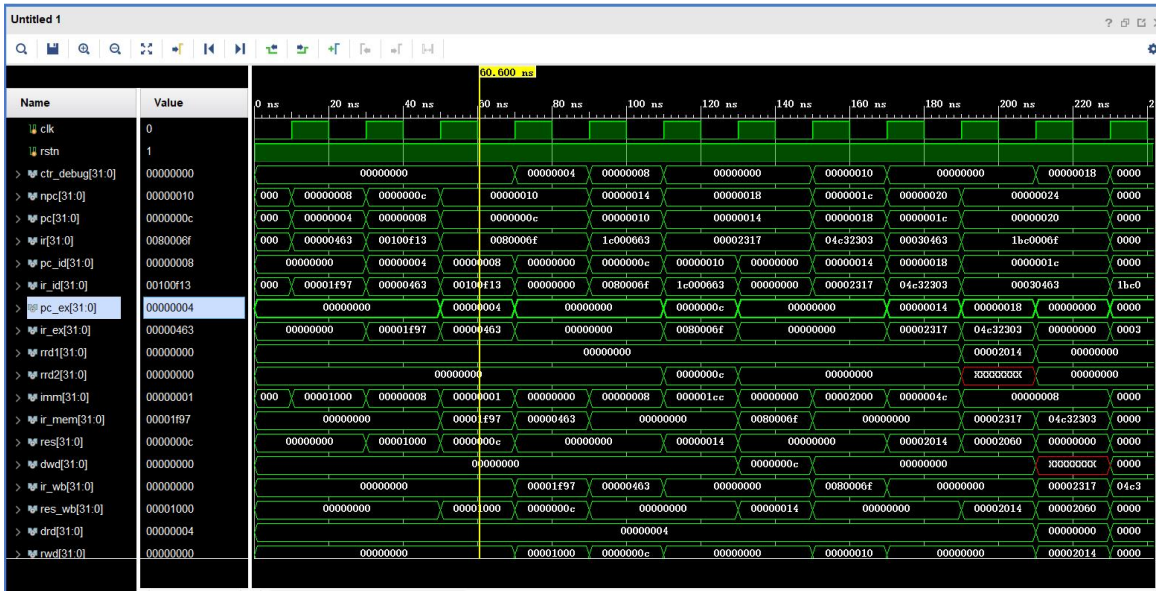
仿真文件如下：

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2023/05/21 10:50:58
// Design Name:
// Module Name: test_branch
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
```

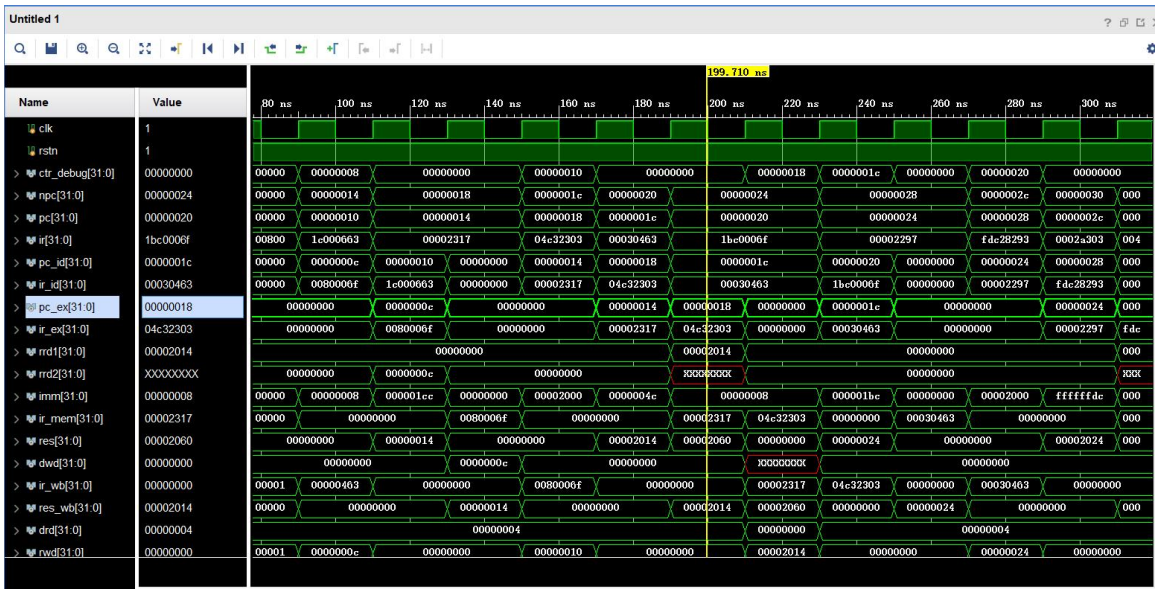
```
13. //
14. // Dependencies:
15. //
16. // Revision:
17. // Revision 0.01 - File Created
18. // Additional Comments:
19. //
20. //////////////////////////////////////////////////////////////////////////////////
21.
22.
23. module test_branch();
24. reg clk = 0;
25. reg rstn = 1;
26.
27.
28. wire [31:0]     ctr_debug;
29. wire [31:0]     npc,pc,ir,
30.                 pc_id,ir_id,pc_ex,ir_ex,rrd1,rrd2,imm,
31.                 ir_mem,res,dwd,ir_wb,res_wb,drd,rwd;
32. wire [31:0]     drd0,rrd0;
33. reg [2:0]      dra0 = 3'b0;
34. wire [4:0]      rra0;
35.
36. initial
37. begin
38.     forever #10 clk = ~clk;
39. end
40.
41. initial
42. begin
43.     forever #10 dra0 = dra0 + 3'b1;
44. end
45.
46. pipe_line_cpu cpu_v4(
47.     .cpu_clk(clk),      // 控制CPU 运行的时钟
48.     .cpu_rstn(rstn),    // 控制CPU 复位的信号
49.     /* ***以下根据需要修改*** */
50.     // 用于D 与R 指令的调试接口
51.     .dra0({7'b0,dra0}),             // 数据存储器的输入地址
52.     .drd0(drd0),            // 数据存储器的输出
53.     .rra0(rra0),            // 寄存器堆的输入地址，唯一的五位位宽
54.     .rrd0(rrd0),            // 寄存器堆的输出数据
55.     // 自由定义部分
56.     .ctr_debug(ctr_debug),
57.     // IF段
58.     .npc(npc),              // 下一个pc值
59.     .pc(pc),                // IF 段前pc
```

```
60.         .ir(ir),                // IF 段取出的指令
61.         // IF/ID 段间
62.         .pc_id(pc_id),          // IF 段?给 ID 段的 pc
63.         .ir_id(ir_id),          // IF 段?给 ID 段的 ir
64.         // ID/EX 段间
65.         .pc_ex(pc_ex),          // ID 段?给 EX 段的 pc
66.         .ir_ex(ir_ex),          // ID 段?给 EX 段的 ir
67.         .rrd1(rrd1),            // 寄存器堆输出?1
68.         .rrd2(rrd2),            // 寄存器堆输出?2
69.         .imm(imm),
70.         // EX/MEM 段间
71.         .ir_mem(ir_mem),        // EX 段?给 MEM 段的 ir
72.         .res(res),              // ALU output
73.         .dwd(dwd),              // 要写入数据存储器的数?
74.         // MEM/WB 段间
75.         .ir_wb(ir_wb),          // MEM 段?给 WB 段的 ir
76.         .drd(drd),              // 数据存储器读出的数据
77.         .res_wb(res_wb),        // ALU 的计算结果继续传?
78.         // WB 段写?
79.         .rwd(rwd)               // ?要写回到寄存器堆的数?
80.     );
81.
82. endmodule
```

仿真文件的结果如下所示（以指令集测试程序为例）：



在 pc_e 运行到 00000004 时，跳转指令判断成功，发生了 flush,于是接下来两个周期的 pc_e 全部为 00000000，然后才是跳转成功后去的 0000000c。



而 pc_e 为 00000018 时，出现了不能利用数据前递处理的 lw，所以发生了 stall，即下一个时钟周期的 pc_e 变为 00000000，等待一个时钟周期之后才能获得 lw 从数据存储器中读出的数值。

下面为指令含义：

Text Segment

Bkpt	Address	Code	Basic	Source		
	0x00000000	0x00001f97	auipc x31,1	35:	auipc t6, 1	
	0x00000004	0x00000463	beq x0,x0,0x00000008	36:	beq zero, zero, pass_beq	#test_beq
	0x00000008	0x00100f13	addi x30,x0,1	37:	addi t5, zero, 1	
	0x0000000c	0x0080006f	jal x0,0x00000008	39:	jal zero, pass_jal	#test_jal
	0x00000010	0x1c000663	beq x0,x0,0x000001cc	40:	beq zero, zero, wrong_answer	
	0x00000014	0x00002317	auipc x6,2	43:	lw t1, test_lw	#test_lw
	0x00000018	0x04c32303	lw x6,0x0000004c(x6)			
	0x0000001c	0x00030463	beq x6,x0,0x00000008	44:	beq t1, zero, pass_lw	
	0x00000020	0x1bc0006f	jal x0,0x000001bc	45:	jal zero, wrong_answer	
	0x00000024	0x00002297	auipc x5,2	47:	la t0, test_add	#test_add
	0x00000028	0xfdc28293	addi x5,x5,0xffffffdc			
	0x0000002c	0x0002a303	lw x6,0(x5)	48:	lw t1, (t0)	

测试结果与分析

对于流水线 CPU 我共检验了三个不同的汇编程序，分别为：lab3 时写的指令集测试和冒泡排序、以及助教提供的相关检测汇编程序。

下面分别展示在三个程序的运行情况：

1. 指令集测试：

用 rars 跑完程序之后表示状态量的 PC 和寄存器的情况如下：

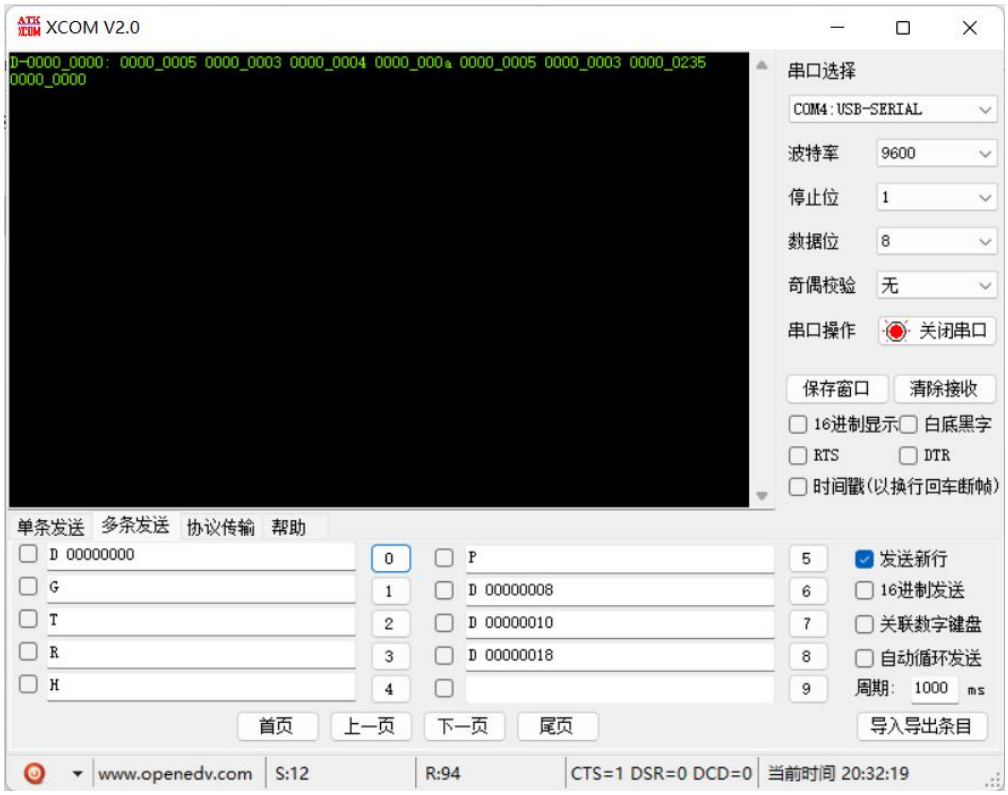
Registers | Floating Point | Control and Status

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x00003ffc
gp	3	0x00001800
tp	4	0x00000000
t0	5	0x000001cc
t1	6	0x000001cc
t2	7	0x00000002
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x00000001
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0xffffffff
t4	29	0x00000003
t5	30	0x00000000
t6	31	0x000001b8
pc		0x000001c8

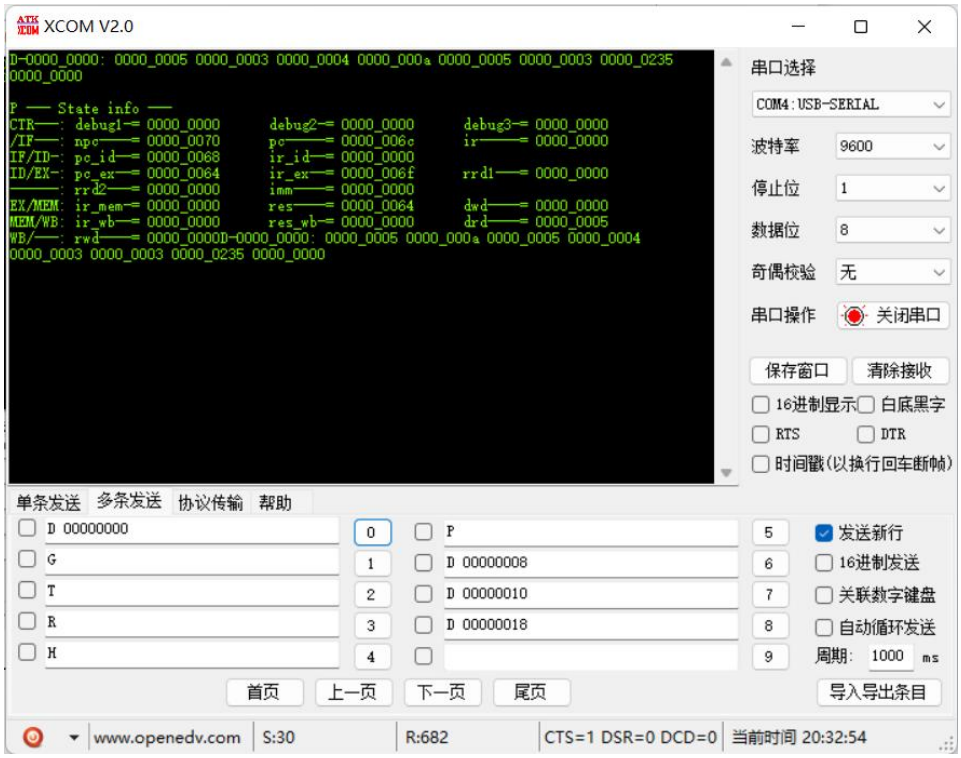
串口调试所得结果如下：

2. 冒泡排序程序：

排序之前的数据存储器数据展示如下：



排序之后我们得到如下的结果：



其中第一个数为要排序的数组长度，然后排序的结果为降序 a 5 4 3 3,235 为第 6 个数所以没有纳入排序范围。

3. 助教提供的相关程序：

助教程序在单周期 cpu 检验后的寄存器状态如下：

Registers | Floating Point | Control and Status

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000001
sp	2	0x00000002
gp	3	0x00000003
tp	4	0x00000004
t0	5	0x00000005
t1	6	0x00000015
t2	7	0x00000007
s0	8	0x00000008
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x0000004c
a2	12	0x0000004c
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00002000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00000050

利用串口调试所得结果与预期一致：

总结

1. 本次实验有惊无险的度过了。在实验的过程中，我充分理解了五级流水线和单周期 CPU 之间相同和不同的地方。两者在 ALU 的使用上有不小的区别。在单周期中 ALU 在跳转指令中用于比较控制条件是否成立，而流水线中则需要其来计算跳转的地址，自己也因为这个写错了很多的控制信号。以及写实验的过程中愈发的理解了规范定义变量名的优势与便捷。相较于单周期 cpu，这次的连线失误减少了很多。同时也充分理解了冲突相关以及对应的解决方法。
2. 吐槽：老师能否将实验文档中的数据通路再补充的详细一些，以及实验文档的冲突相关部分能否多些文字说明，而不是靠学生们去普通班那里找资料。