

浙江大学

本科实验报告

课程名称: 计算机组成

姓 名: XXX

学 院: 计算机学院

系:

专 业: 计算机科学与技术

学 号: XXXXXXXXXXXX

指导教师: 马德

2025 年 12 月 4 日

浙江大学实验报告

课程名称: 计算机组成 实验类型: _____

实验项目名称: 单周期处理器

学生姓名: XXX 专业: 计算机科学与技术 学号: XXXXXXXXXX

同组学生姓名: _____ 指导老师: 马德

实验地点: 东 4-509 实验日期: 2025 年 10 月 16 日

目录

Experiment0-CPU 核集成设计	6
一、实验目的和要求	6
1.1 实验目的	6
1.2 实验要求	6
二、实验内容和原理	6
2.1 Computer Organization	6
2.2 Digital circuits vs CPU organization	6
2.3 Data_path	7
2.4 SCPU_ctrl	7
三、主要仪器设备	7
四、操作方法与实验步骤	7
4.1 建立工程	7
4.2 用 RTL 代码描述 CPU 设计 SCPU.v	8
五、实验数据记录和处理	9
5.1 替换验证	9
5.2 实验结果	9
5.2.1 七段数码管	9
六、实验结果与分析	13
6.1 实验结果	13
6.2 实验代码分析	14
6.3 实验结果分析	16
七、讨论、心得	16
Experiment1-CPU 设计之数据通路	16
一、实验目的和要求	16
1.1 实验目的	16
1.2 实验要求	16
二、实验内容和原理	17
2.1 Datapath	17
2.2 Table of RV Registers	19
2.3 RV Instructions	19

2.4 单周期 CPU.....	20
2.5 单周期数据通路结构.....	20
2.6 R-Format Instruction	21
2.7 I-Format Instructions	24
2.8 B-Format Instruction	27
2.9 J-Format Instruction	28
2.10 控制信号定义	30
三、主要仪器设备	30
四、操作方法与实验步骤	30
4.1 任务一：设计实现数据通路	30
4.2 任务二：设计数据通路测试方案并完成测试	33
五、实验数据记录和处理	34
5.1 替换验证	34
六、实验结果与分析	37
6.1 替换验证	37
6.2 代码分析	39
6.3 实验结果分析	41
七、讨论、心得	41
Experiment2-CPU 设计之控制器	42
一、实验目的和要求	42
1.1 实验目的	42
1.2 实验目标及任务	42
二、实验内容和原理	42
2.1 Contrl unit	42
2.2 数据通路结构	43
2.3 add	43
2.4 sw	43
2.5 lw	44
2.6 beq	44
2.7 jal	44
2.8 主控制器信号	45
2.9 ImmSel	45
2.10 RSICV-encode	46
2.11 RSICV-decode	47
2.12 ALU 操作译码-多级译码	48
三、主要仪器设备	48
四、操作方法与实验步骤	48
4.1 任务一：用硬件描述语言设计实现控制器	48
4.2 任务二：设计控制器测试方案并完成测试	49
五、实验数据记录和处理	52
5.1 仿真测试	52
5.2 物理测试	52
六、实验结果与分析	56
6.1 仿真测试	56

6.2 物理测试	56
6.3 代码分析	58
七、讨论、心得	62
Experiment3-CPU 设计之指令扩展	62
一、实验目的和要求	62
1.1 实验目的	62
1.2 实验目标及任务	63
二、实验内容和原理	63
2.1 数据通路	63
2.2 JALR	64
2.3 U-Format Instruction	65
2.4 bne	66
2.4.1 Datapath with bne	66
2.4.2 Control unit	66
2.5 控制信号	67
2.6 ImmSel 信号	67
2.7 RSICV---decode	68
三、主要仪器设备	70
四、操作方法与实验步骤	70
4.1 设计指令扩展后的 ALU	70
4.2 设计指令扩展后的 ImmGen	70
4.3 设计指令扩展后的 SCPU_ctrl	70
4.4 设计指令扩展后的 Datapath	72
4.5 CPU 调试与测试	75
五、实验数据记录和处理	75
5.1 物理验证	75
六、实验结果与分析	79
6.1 物理验证	79
6.2 代码分析	81
6.3 结果分析	87
七、讨论、心得	87
Experiment4-CPU 设计之中断	88
一、实验目的和要求	88
1.1 实验目的	88
1.2 实验目标及任务	88
二、实验内容和原理	88
2.1 中断概念	88
2.2 中断（异常）处理过程	89
2.3 RISC-V 中断结构	90
2.4 RISC-V 中断处理	90
2.5 中断相关指令	92
2.6 RISC-V 中断结构---退出异常	92
2.7 RISC-V 中断结构---异常服务程序	93
2.8 典型处理器中断结构	93

2.9 典型处理器中断结构---ARM 中断	93
2.10 简化中断设计	94
三、主要仪器设备	94
四、操作方法与实验步骤	94
4.1 扩展 CPU 中断功能	94
4.2 任务二：设计 CPU 中断测试方案并完成测试	98
五、实验数据记录和处理	98
六、实验结果与分析	102
6.1 代码分析	102
6.2 物理测试分析	105
七、讨论、心得	110

Experiment0-CPU 核集成设计

一、实验目的和要求

1.1 实验目的

- 复习寄存器传输控制技术
- 掌握 CPU 的核心组成：数据通路与控制器
- 设计数据通路的功能部件
- 进一步了解计算机系统的基本结构
- 熟练掌握 IP 核的使用方法

1.2 实验要求

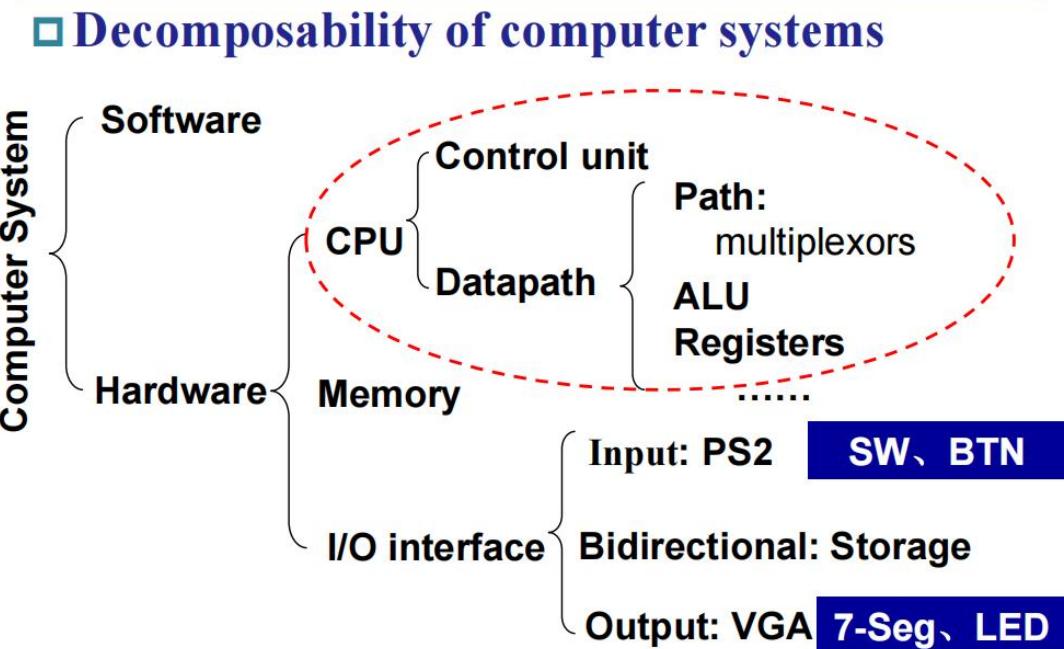
熟悉 SOC 系统的原理，掌握 IP 核集成设计 CPU 的方法

利用数据通路和控制器两个 IP 核集成设计 CPU（根据原理图采用 RTL 代码方式）

二、实验内容和原理

2.1 Computer Organization

Computer Organization

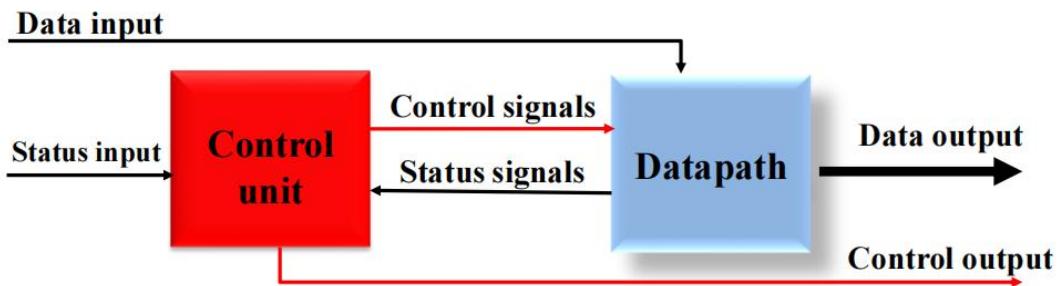


2.2 Digital circuits vs CPU organization

Digital circuits vs CPU organization

□ Digital circuit

- General circuits that controls logical event with logical gates -
-Hardware



□ Computer organization

- Special circuits that processes logical action with instructions
-Software

2.3 Data_path

CPU 主要部件之一

寄存器传输控制对象：通用数据通路

- 基本功能

具有通用计算功能的算术逻辑部件

具有通用目的寄存器

具有通用计数所需的尽可能的路径

2.4 SCPU_ctrl

CPU 主要部件之一

寄存器传输控制技术中的运算和通路控制器

- 基本功能

指令译码

产生操作控制信号：ALU 运算控制

产生指令所需的路径选择

三、主要仪器设备

- 计算机（Intel Core i9-13980, 16GB 内存）系统
- NEXYS A7 开发板
- Xilinx VIVADO2024.2 及以上开发工具

四、操作方法与实验步骤

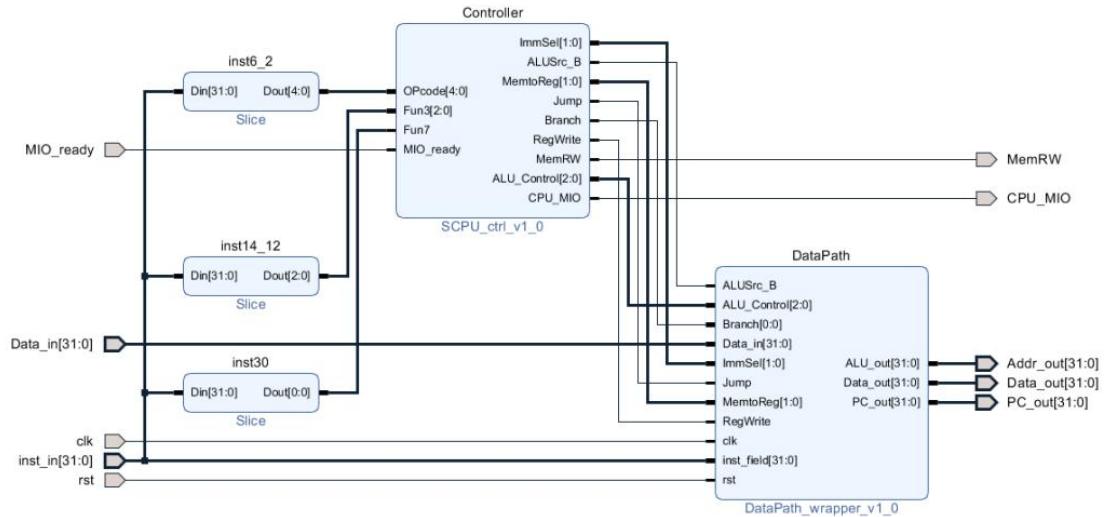
4.1 建立工程

建立工程 OExp04-IP2CPU。

4.2 用 RTL 代码描述 CPU 设计 SCPU.v

根据逻辑原理图，利用 verilog 描述，在 SCPU.v 顶层进行模块调用和连接。

逻辑原理图



所编写代码如下所示：

```
23 module SCPU(
24     input MIO_ready,
25     input [31:0] Data_in,
26     input clk,
27     input [31:0] inst_in,
28     input rst,
29     output MemRW,
30     output CPU_MIO,
31     output [31:0] Addr_out,
32     output [31:0] Data_out,
33     output [31:0] PC_out
34 );
35
36     wire [1:0] SCPU_ctrl_0_ImmSel;
37     wire SCPU_ctrl_0_ALUSrc_B;
38     wire [1:0] SCPU_ctrl_0_MemToReg;
39     wire SCPU_ctrl_0_Jump;
40     wire SCPU_ctrl_0_Branch;
41     wire SCPU_ctrl_0_RegWrite;
42     //wire SCPU_ctrl_0_MemRW;
43     wire [2:0] SCPU_ctrl_0_ALU_Control;
44     //wire SCPU_ctrl_0_CPU_MIO;
45 
```

```

46  :     SCPU_ctrl SCPU_ctrl_0(
47  :         .OPcode(inst_in[6:2]),
48  :         .Fun3(inst_in[14:12]),
49  :         .Fun7(inst_in[30]),
50  :         .MIO_ready(MIO_ready),
51  :         .ImmSel(SCPU_ctrl_0_ImmSel),
52  :         .ALUSrc_B(SCPU_ctrl_0_ALUSrc_B),
53  :         .MemtoReg(SCPU_ctrl_0_MemToReg),
54  :         .Jump(SCPU_ctrl_0_Jump),
55  :         .Branch(SCPU_ctrl_0_Branch),
56  :         .RegWrite(SCPU_ctrl_0_RegWrite),
57  :         .MemRW(MemRW),
58  :         .ALU_Control(SCPU_ctrl_0_ALU_Control),
59  :         .CPU_MIO(CPU_MIO)
60  :     );
61
62
63  :     DataPath DataPath_0(
64  :         .ALUSrc_B(SCPU_ctrl_0_ALUSrc_B),
65  :         .ALU_Control(SCPU_ctrl_0_ALU_Control),
66  :         .Branch(SCPU_ctrl_0_Branch),
67  :         .Data_in(Data_in),
68  :         .ImmSel(SCPU_ctrl_0_ImmSel),
69  :         .Jump(SCPU_ctrl_0_Jump),
70  :         .MemtoReg(SCPU_ctrl_0_MemToReg),
71  :         .RegWrite(SCPU_ctrl_0_RegWrite),
72  :         .clk(clk),
73  :         .rst(rst),
74  :         .inst_field(inst_in),
75  :         .ALU_out(Addr_out),
76  :         .Data_out(Data_out),
77  :         .PC_out(PC_out)
78  :     );
79 endmodule

```

五、实验数据记录和处理

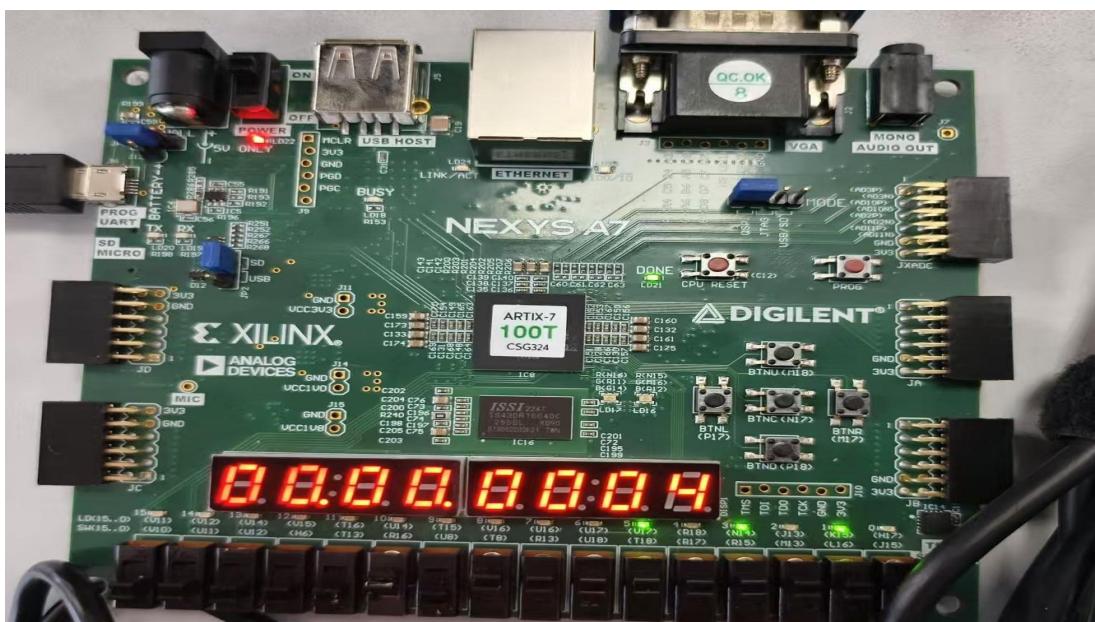
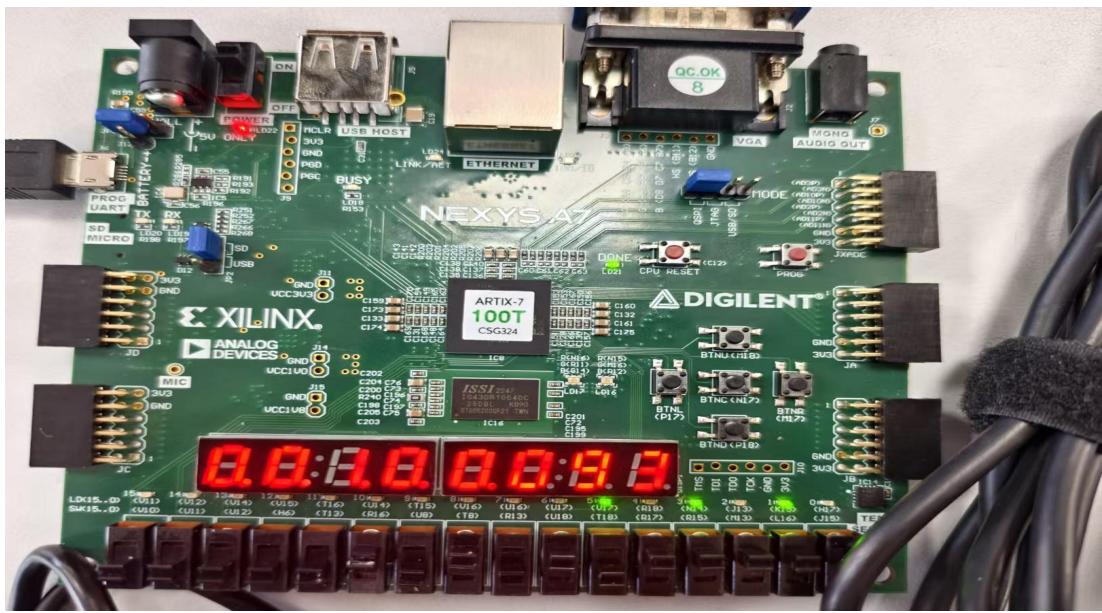
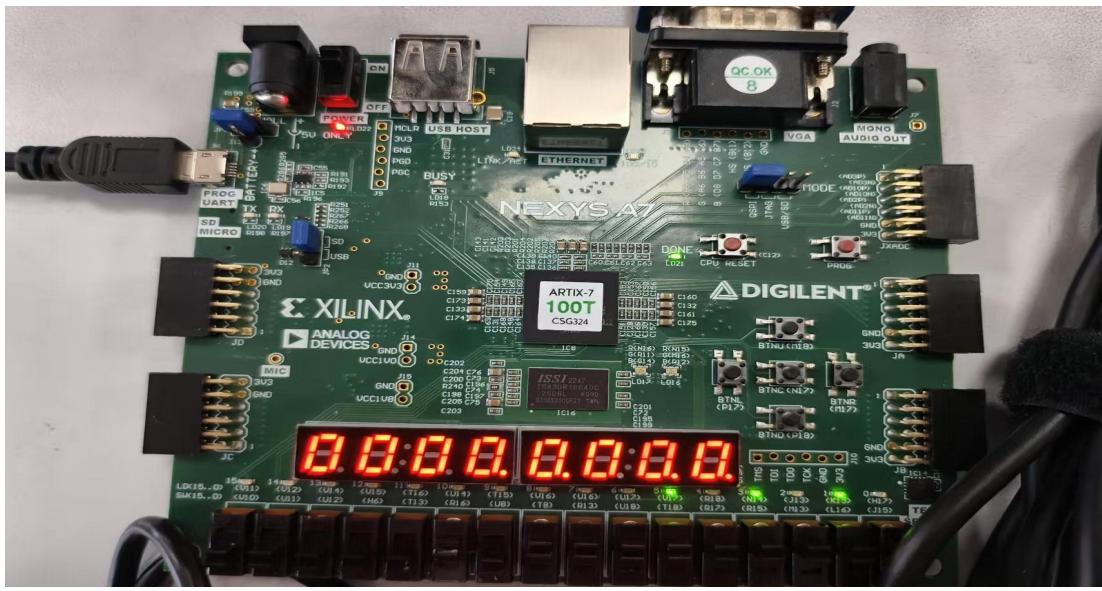
5.1 替换验证

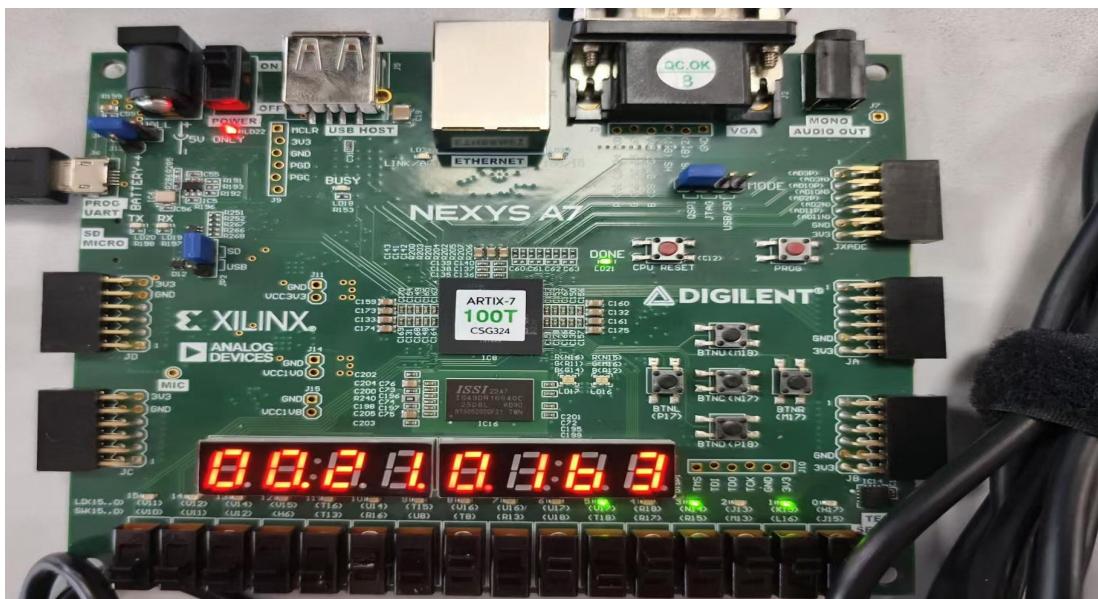
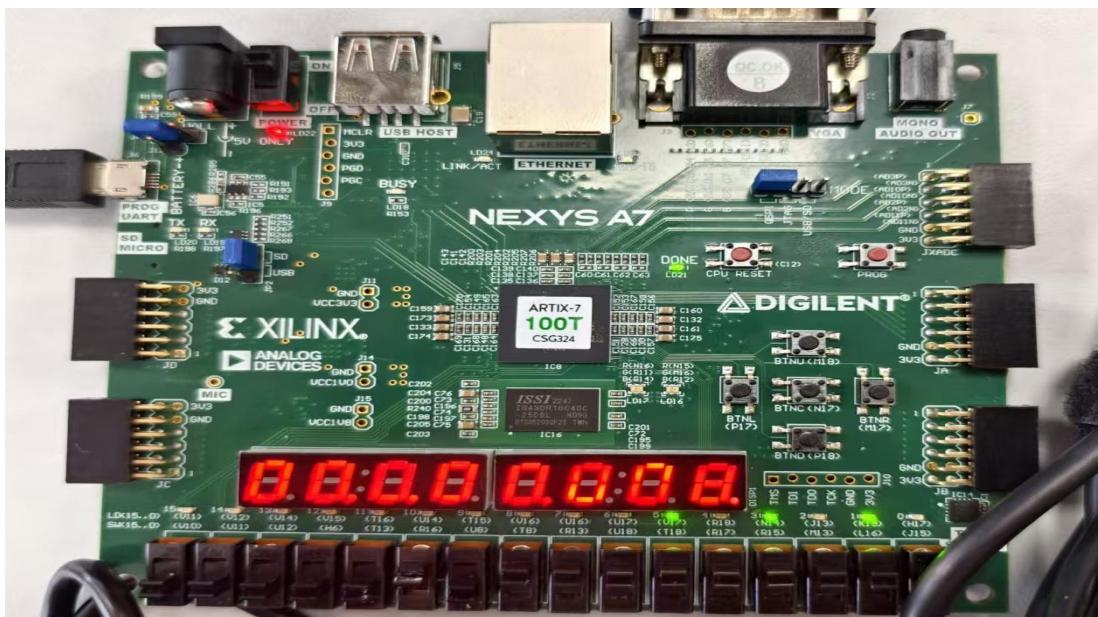
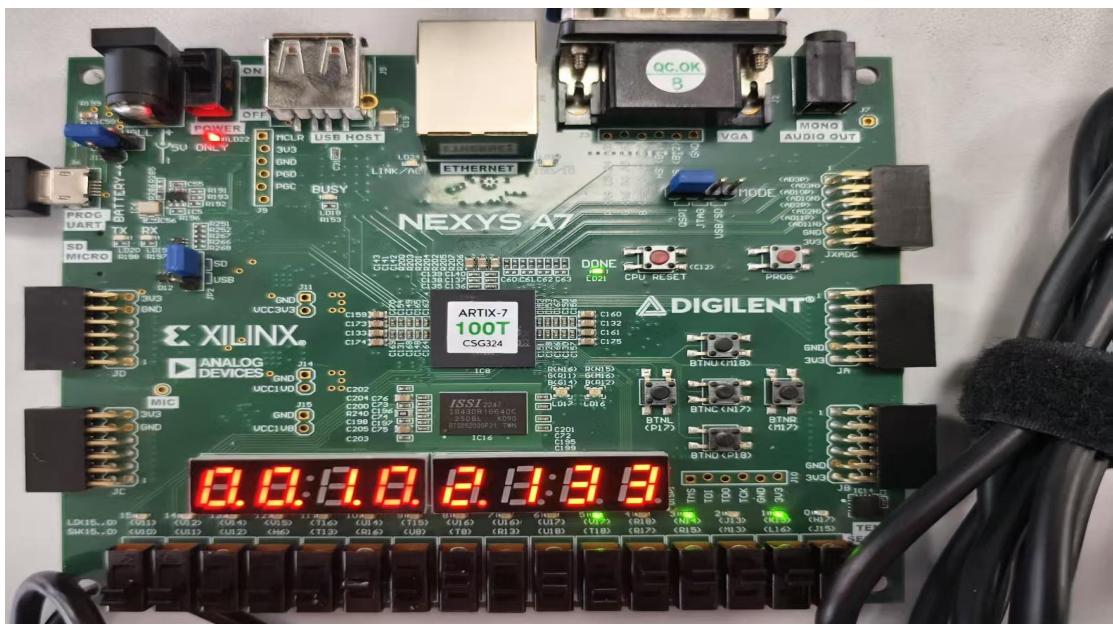
使用 IP 核调用设计的 CPU 模块替换实验 2 中 CPU 调试环境中的 SCPU 模块，综合、实现、生成比特流之后烧录至开发板进行验证。

验证时将 CPU 时钟设置为手动时钟，然后观察开发板上七段数码管显示结果（需调整七段数码管数据源以查看 PC 以及 instruction）及 VGA 输出显示结果。

5.2 实验结果

5.2.1 七段数码管





5.2.2 VGA 输出

```
RV32I Single Cycle CPU

pc: 00000000 inst: 00100093

x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000001 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000000 dmem_addr: 00000001

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

```
RV32I Single Cycle CPU

pc: 00000004 inst: 00102133

x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000001 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000001 dmem_addr: 00000001

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

```
RV32I Single Cycle CPU

pc: 00000008 inst: 002101b3

x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000002 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

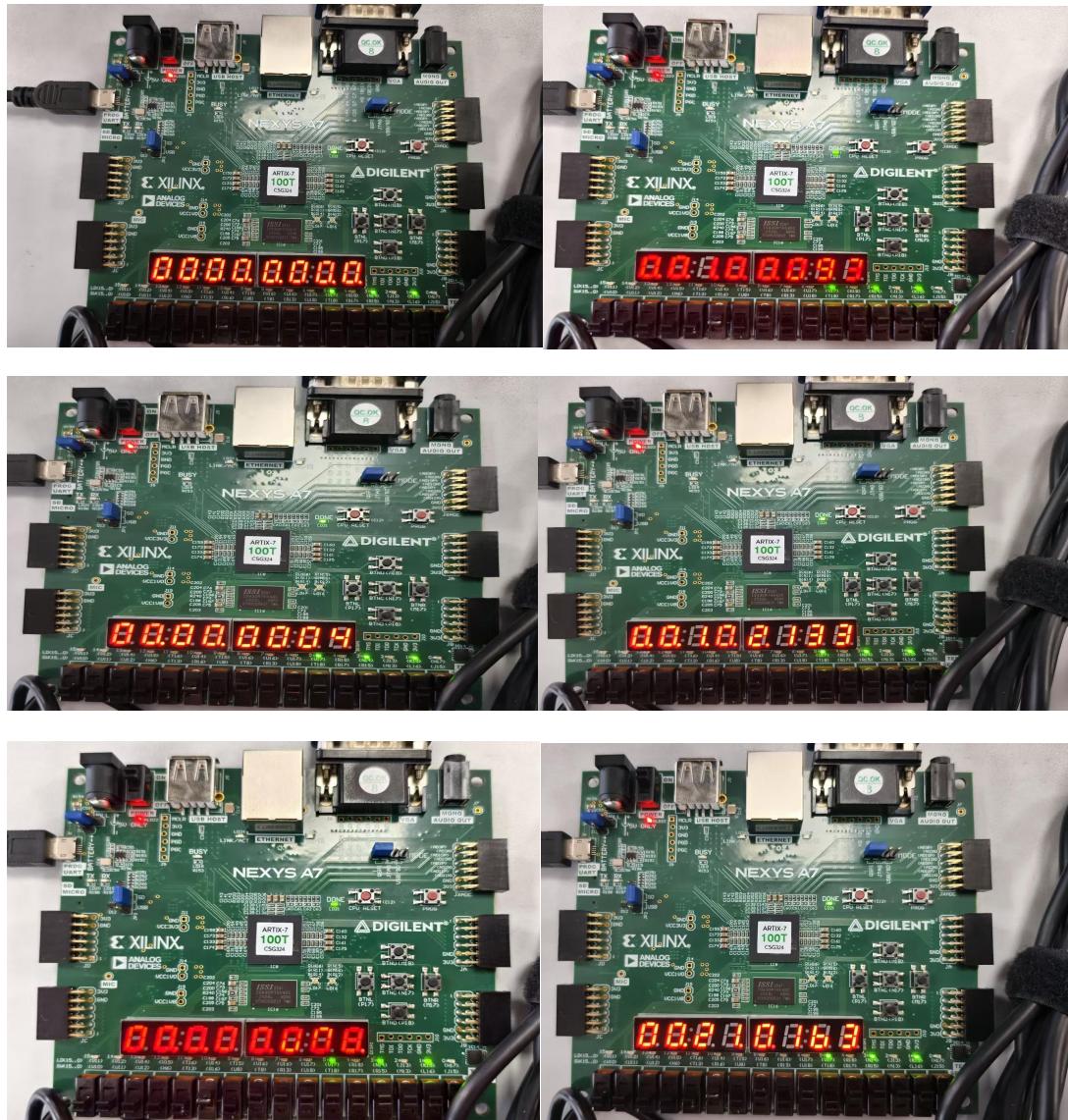
mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000001 dmem_addr: 00000002

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

六、实验结果与分析

6.1 实验结果

6.1.1 七段数码管



根据三步手动时钟运行我们可以看到程序指令地址 PC 可以做到正常、正确地递加，以及三个指令地址对应的指令编码也是正确。说明我们的模块设计功能正确。

6.1.2 VGA 输出

```

RV32I Single Cycle CPU

pc: 00000000 inst: 00100093

x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auiipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000001 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000000 dmem_addr: 00000001

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

```

RV32I Single Cycle CPU

pc: 00000004 inst: 00102133

x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auiipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000001 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000001 dmem_addr: 00000001

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

```

RV32I Single Cycle CPU

pc: 00000008 inst: 002101b3

x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auiipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000002 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000001 dmem_addr: 00000002

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

根据三步手动时钟的 VGA 输出结果，我们可以看到程序计数器可以做到正确地递加，指令地址也均符合我们初始化 ROM 模块的文件内容。说明所设计的模块功能正常。

6.2 实验代码分析

```

23  module SCPU(
24      input MIO_ready,
25      input [31:0] Data_in,
26      input clk,
27      input [31:0] inst_in,
28      input rst,
29      output MemRW,
30      output CPU_MIO,
31      output [31:0] Addr_out,
32      output [31:0] Data_out,
33      output [31:0] PC_out
34  );
35
36  wire [1:0] SCPU_ctrl_0_ImmSel;
37  wire SCPU_ctrl_0_ALUSrc_B;
38  wire [1:0] SCPU_ctrl_0_MemToReg;
39  wire SCPU_ctrl_0_Jump;
40  wire SCPU_ctrl_0_Branch;
41  wire SCPU_ctrl_0_RegWrite;
42  //wire SCPU_ctrl_0_MemRW;
43  wire [2:0] SCPU_ctrl_0_ALU_Control;
44  //wire SCPU_ctrl_0_CPU_MIO;
45

```

该部分代码定义了 SCPU 模块的输入输出端口，并且声明了模块内部需要使用到的中间变量。

```

46  SCPU_ctrl SCPU_ctrl_0(
47      .OPcode(inst_in[6:2]),
48      .Fun3(inst_in[14:12]),
49      .Fun7(inst_in[30]),
50      .MIO_ready(MIO_ready),
51      .ImmSel(SCPU_ctrl_0_ImmSel),
52      .ALUSrc_B(SCPU_ctrl_0_ALUSrc_B),
53      .MemtoReg(SCPU_ctrl_0_MemToReg),
54      .Jump(SCPU_ctrl_0_Jump),
55      .Branch(SCPU_ctrl_0_Branch),
56      .RegWrite(SCPU_ctrl_0_RegWrite),
57      .MemRW(MemRW),
58      .ALU_Control(SCPU_ctrl_0_ALU_Control),
59      .CPU_MIO(CPU_MIO)
60  );
61

```

该部分代码实例化引用 SCPU_ctrl 模块，使用 SCPU 模块的输入端口数据以及我们声明的中间变量作为相应的端口变量。

```

62 |     DataPath DataPath_0(
63 |         .ALUSrc_B(SCPU_ctrl_0_ALUSrc_B),
64 |         .ALU_Control(SCPU_ctrl_0_ALU_Control),
65 |         .Branch(SCPU_ctrl_0_Branch),
66 |         .Data_in(Data_in),
67 |         .ImmSel(SCPU_ctrl_0_ImmSel),
68 |         .Jump(SCPU_ctrl_0_Jump),
69 |         .MemtoReg(SCPU_ctrl_0_MemToReg),
70 |         .RegWrite(SCPU_ctrl_0_RegWrite),
71 |         .clk(clk),
72 |         .rst(rst),
73 |         .inst_field(inst_in),
74 |         .ALU_out(Addr_out),
75 |         .Data_out(Data_out),
76 |         .PC_out(PC_out)
77 |     );
78 |
79 endmodule

```

该部分代码实例化引用 DataPath 模块，使用 SCPU 模块的输出端口以及我们声明的中间变量作为相应的端口变量。

6.3 实验结果分析

根据我们所初始化 ROM 用的程序指令文件，可以发现程序运行正确，说明我们利用 IP 核设计的 CPU 模块可以正常工作。

七、讨论、心得

通过本次基础实验，我理解了 CPU 的两大组成部分：Data_path 和 Ctrl 部分，同时也更加理解了通过抽象化、模块化简化对象设计的思想。

Experiment1-CPU 设计之数据通路

一、实验目的和要求

1.1 实验目的

- 运用寄存器传输控制技术
- 掌握 CPU 的核心：数据通路组成与原理
- 设计数据通路
- 学习测试方案的设计
- 学习测试程序的设计

1.2 实验要求

目标：熟悉 RISC-V RV32I 的指令特点，了解数据通路的原理，设计并测试数据通路

任务一：设计实现数据通路（采用 RTL 实现）

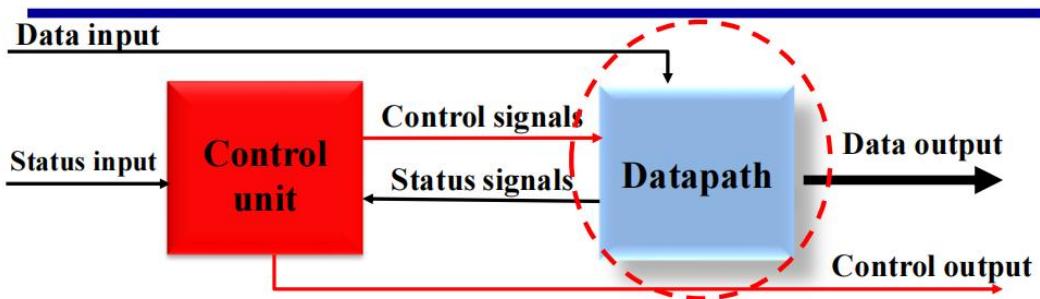
- ALU 和 Regs 调用 Exp01 设计的模块（可直接加 RTL）

- PC 寄存器设计及 PC 通路建立
 - ImmGen 立即数生成模块设计
 - 此实验在 Exp4-0 的基础上完成，替换 Exp4-0 的数据通路核
- 任务二：设计数据通路测试方案并完成测试
- 通路测试：I-格式通路、R-格式通路
 - 部件测试：ALU、Register Files

二、实验内容和原理

2.1 Datapath

Datapath

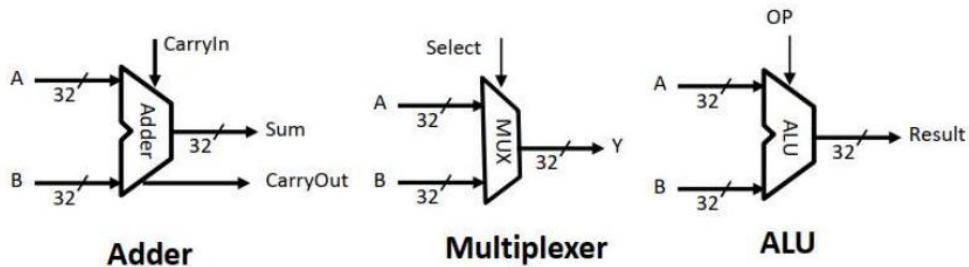


• 数据通路

数据通路作为处理器的一部分，包含了完成处理器所要求的操作所必须的硬件，包括运算单元、寄存器组、状态寄存器等

• 数据通路部件

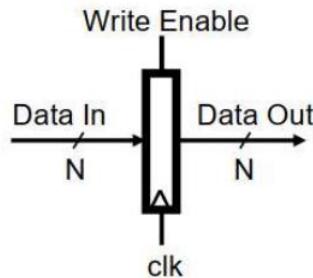
□ 组合逻辑单元：加法器、多路选择器、**ALU**算数运算单元



□ 时序逻辑单元（状态元件）： Register

• Write Enable:

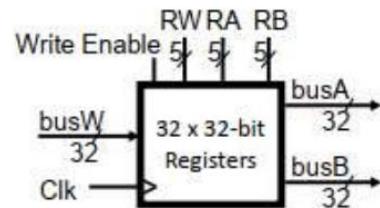
- 置0，数据输出保持原状态不变
- 置1，在有效时钟边沿到来，数据输出为数据输入值



□ 时序逻辑单元：寄存器堆

• Register files: 由32个register构成

- 两个32位的数据输出端口：busA、busB
- 一个32位的数据输入端口：busW



• 寄存器读写操作:

- RA (number) 作为地址选择register存储的数据传输到输出端口busA (读)
- RB (number) 作为地址选择register存储的数据传输到输出端口busB (读)
- RW (number) 作为地址选择register接收输入端口busW的数据，当Write Enable=1且时钟边沿有效时 (写)

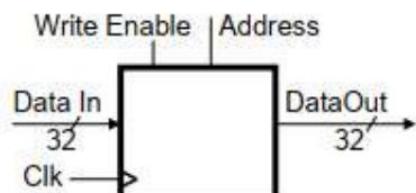
• 时钟 (clk) :

- 写操作时，clk是影响因子，在有效的时钟边沿到来，数据被写入
- 读操作时，clk非影响因子，只要RA、RB有效，经过短暂的器件延迟，数据即从busA、busB输出（此时属于组合逻辑操作）

□ 时序逻辑单元：

• 存储器

- 一个数据输入端口：DataIn
- 一个数据输出端口：DataOut



• 读写操作:

- 读：Address 作为地址选择存储的数据输出到端口DataOut
- 写：Address 作为地址接收端口DataIn输入的数据，当Write Enable = 1且时钟边沿有效

• 时钟(CLK)

- 写操作时，clk是影响因子，在有效的时钟边沿到来，数据被写入
- 读操作时，clk非影响因子，只要Address有效，经过短暂的器件延迟，数据即从DataOut输出（此时属于组合逻辑操作）

每条指令执行时在取指之后会更新以下状态元件：

- **通用寄存器Registers (x0..x31)**

- 寄存器堆为32个32位的寄存器:: `Reg[0]..Reg[31]`
- 指令的`rs1`字段指定了第一个源操作寄存器的读地址
- 指令的`rs2`字段指定了第二个源操作寄存器的读地址
- 指令的`rd`字段指定了目标操作寄存器的写地址
- 寄存器 `x0` 值永远为0；写操作无效

- **Program Counter (PC)**

- 保存当前指令的地址

- **Memory (MEM)**

- 在32位宽的存储空间内保存指令或数据
- 本实验会采用分开的存储用于存储指令 (**IMEM**) 和数据(**DMMEM**)
- 本实验采用的指令存储器只支持只读模式
- 只有Load/store 操作才会访问数据存储器

2.2 Table of RV Registers

Register(s)	Alt.	Description
x0	zero	The zero register, always zero
x1	ra	The return address register, stores where functions should return
x2	sp	The stack pointer, where the stack ends
x5-x7, x28-x31	t0-t6	The temporary registers
x8-x9, x18-x27	s0-s11	The saved registers
x10-x17	a0-a7	The argument registers, a0-a1 are also return value

2.3 RV Instructions

- RISC-V指令分类:

- 1. **RV32I**, 它是 RISC-V 固定不变的基础整数指令集
 - 2. **RV32M**, 乘法和除法
 - 3. **RV32F** 和 **RV32D**, 浮点操作
 - 4. **RV32A**, 原子操作

- RISC-V RV32I 六种基本指令格式：用于寄存器-寄存器操作的 **R** 类型指令，用于短立即数和访存 **load** 操作的 **I** 型指令，用于访存 **store** 操作的 **S** 型指令，用于条件跳转操作的 **B** 类型指令，用于长立即数的 **U** 型指令和用于无条件跳转的 **J** 型指令。

- **本实验主要实现RV32I的常见指令**

31	30	25 24	21	20	19	15 14	12 11	8	7	6	0	R-type
	funct7		rs2		rs1	funct3		rd		opcode		
		imm[11:0]			rs1	funct3		rd		opcode		I-type
	imm[11:5]		rs2		rs1	funct3		imm[4:0]		opcode		S-type
imm[12]	imm[10:5]		rs2		rs1	funct3	imm[4:1]	imm[11]	opcode			B-type
		imm[31:12]						rd		opcode		U-type
imm[20]	imm[10:1]	imm[11]		imm[19:12]				rd		opcode		J-type

opcode: 为操作码；用于表示指令格式和指令操作的字段

rs1: 只读。为第1个源操作数寄存器，寄存器地址（编号）是00000~11111，00~1F；

rs2: 只读。为第2个源操作数寄存器，寄存器地址（编号）是00000~11111，00~1F；

rd: 只写。为目的操作数寄存器，寄存器地址（同上）；

funct3/7: 为功能码，在指令中用来指定指令的功能与操作码配合使用；

immediate: 为立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载（Load）/数据保存（Store）指令的数据地址字节偏移量和分支指令中相对程序计数器（PC）的有符号偏移量；

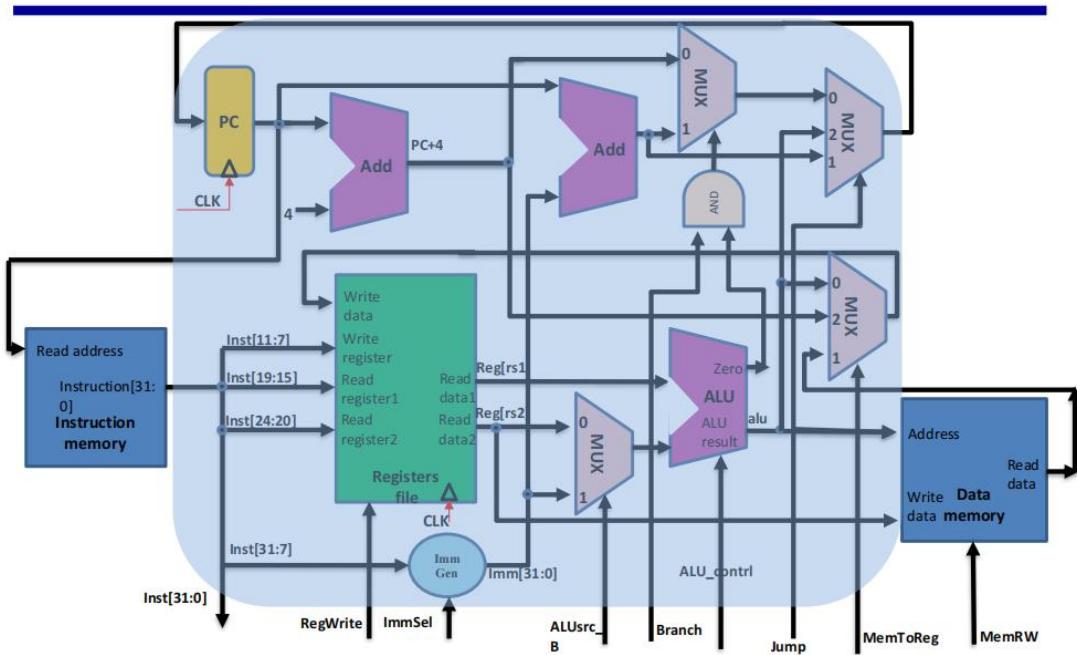
2.4 单周期 CPU

单周期CPU指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即**一条指令用一个时钟周期完成**。单周期CPU，是在一个时钟周期内完成这五个阶段的处理。



- (1) **取指令(IF)**: 根据程序计数器PC中的指令地址，从存储器中取出一条指令，同时，PC根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入PC。
- (2) **指令译码(ID)**: 对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。
- (3) **指令执行(EXE)**: 根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。
- (4) **存储器访问(MEM)**: 所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。
- (5) **结果写回(WB)**: 指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

2.5 单周期数据通路结构



Instruction Memory: 指令存储器,

Readaddress, 指令存储器地址输入端口

Instruction, 指令存储器数据输出端口 (指令代码输出端口)

Data Memory: 数据存储器,

Address, 数据存储器地址输入端口

Writedata, 数据存储器数据输入端口

Readdata, 数据存储器数据输出端口

PC: 程序计数器, 存放指令地址

Registers: 寄存器组

Read Reg1, rs1寄存器地址输入端口

Read Reg2, rs2寄存器地址输入端口

Write Reg, 将数据写入的寄存器端口, 其地址rd字段

Write Data, 写入寄存器的数据输入端口

Read Data1, rs1寄存器数据输出端口

Read Data2, rs2寄存器数据输出端口

ImmGen: 立即数生成单元

ALU: 算术逻辑单元

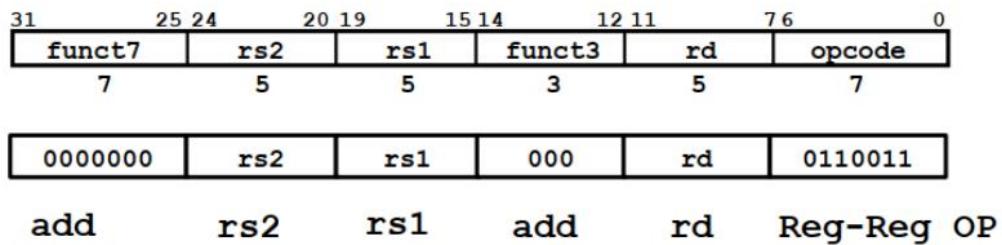
result, ALU运算结果

zero, 运算结果标志, 结果为0, 则zero=1; 否则zero=0

2.6 R-Format Instruction

2.6.1 add

- **Instruction**



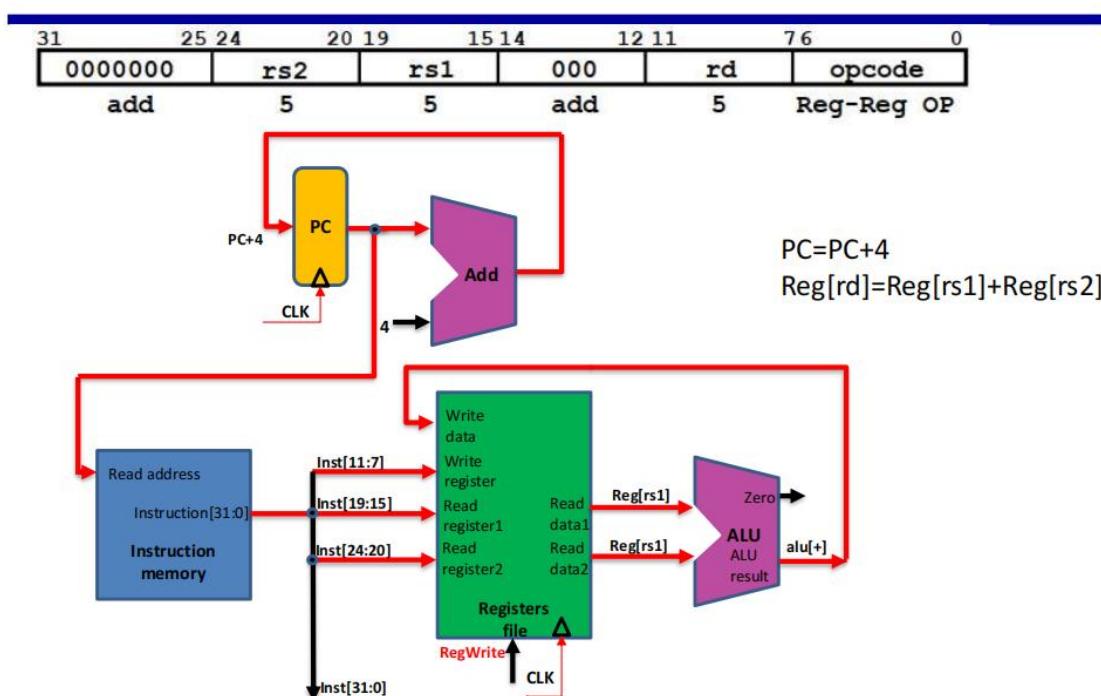
add rd , rs1, rs2

功能: $rd \leftarrow rs1 + rs2$ 。

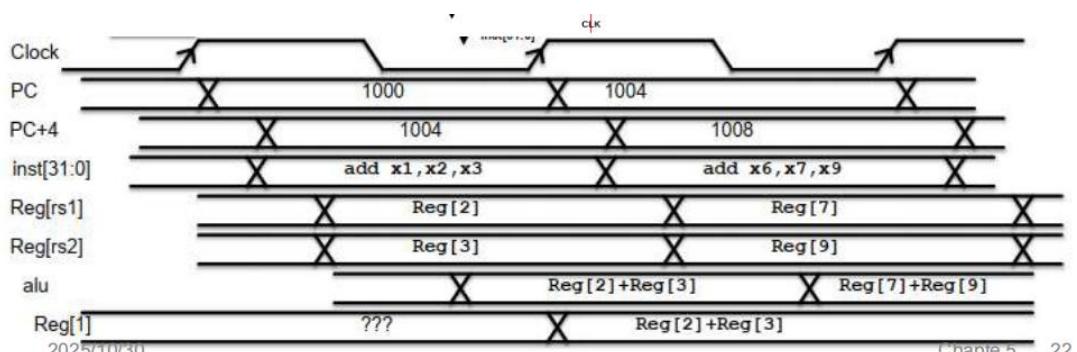
Eg: add x9,x21,x9

0000000_01001_10101_000_01001_0110011

• Datapath for add



• Timing Diagram



2.6.2 Sub

- Instruction

31 7	25 24 rs2	20 19 rs1	15 14 funct3	12 11 rd	7 6 opcode	0
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub

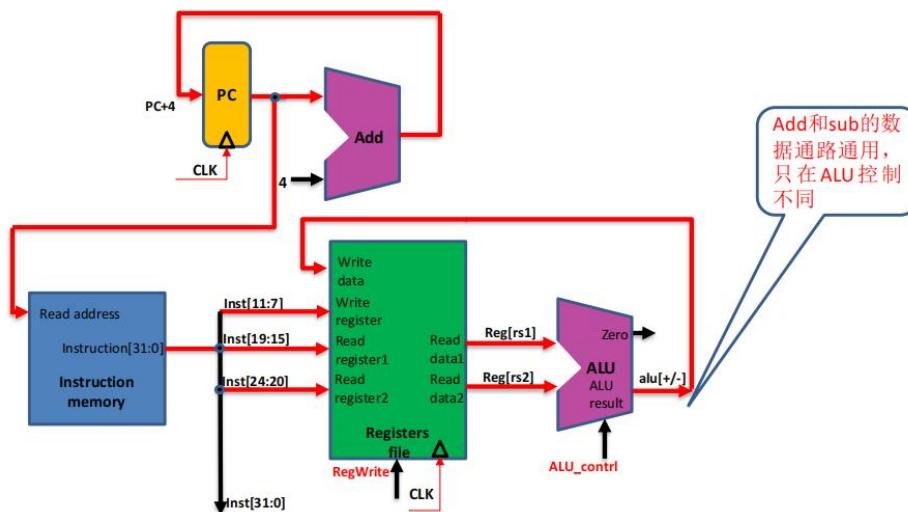
sub rd , rs1 , rs2

功能: $rd \leftarrow rs1 - rs2$

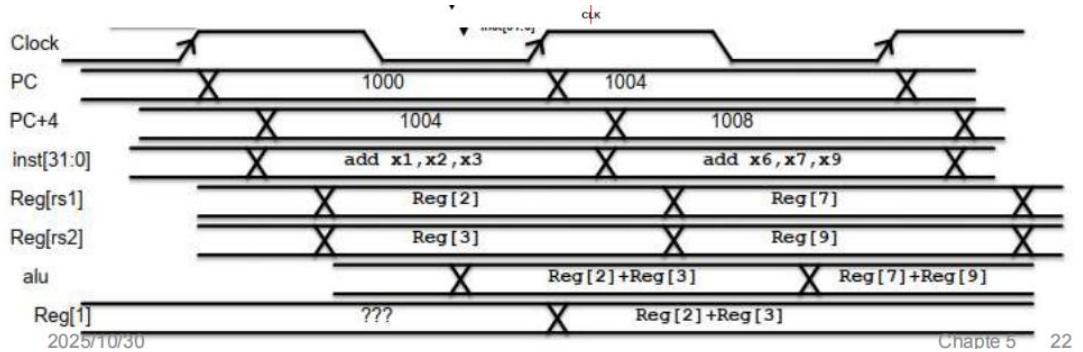
Eg: sub x9,x21,x9

0100000_01001_10101_000_01001_0110011

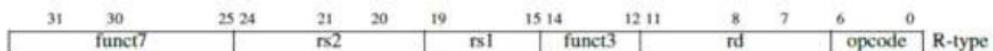
- Datapath for add/sub



- Timing Diagram



2.6.3 Other R-Format Instructions



0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

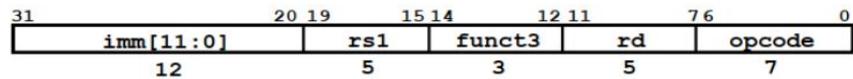
All implemented by decoding funct3 and funct7 fields and selecting appropriate ALU function

所有的R型指令，数据通路通用，只是ALU控制操作不同；而opcode相同，通过funct3和funct7共同决定

2.7 I-Format Instructions

2.7.1 addi

- Instruction



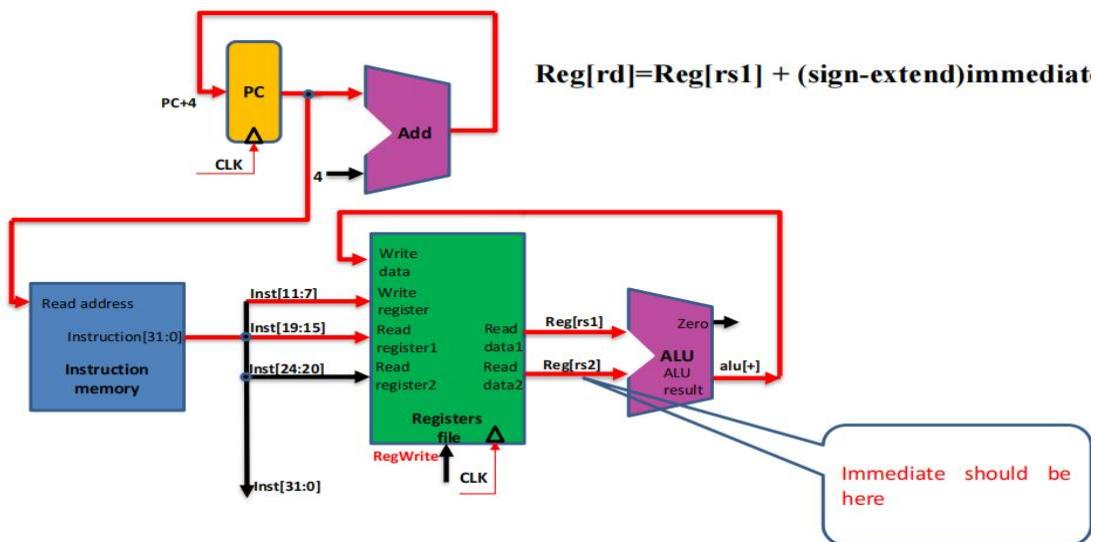
111111001110	00001	000	01111	0010011
imm=-50	rs1=1	add	rd=15	OP-Imm

addi rd , rs1 ,immediate

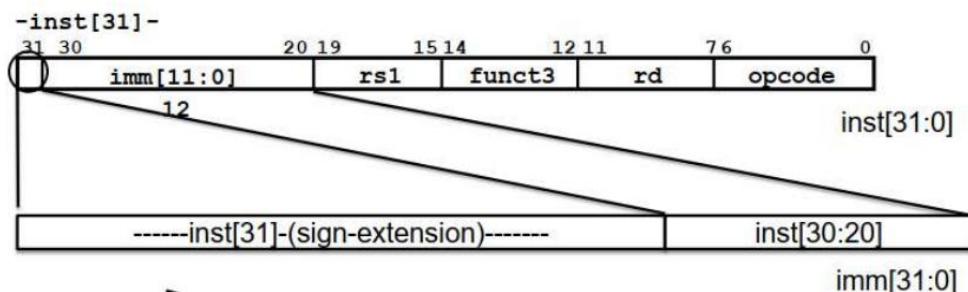
功能: $rd \leftarrow rs1 + (\text{sign-extend})\text{immediate}$; 12位immediate符号扩展再参与“加”运算。

Eg: addi x15,x1,-50
111111001110_00001_000 _01111 _0010011

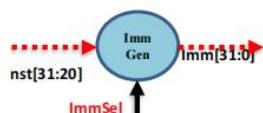
• Datapath added addi



2.7.2 I-Format Immediates



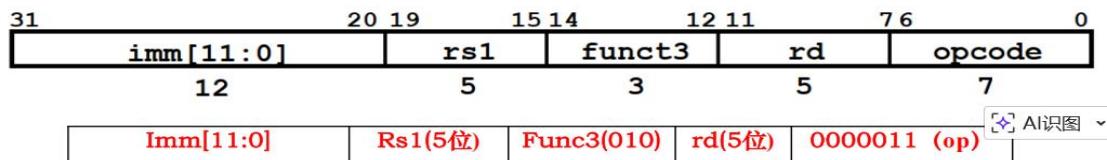
1. High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])



2. Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])

2.7.3 lw

- Instruction

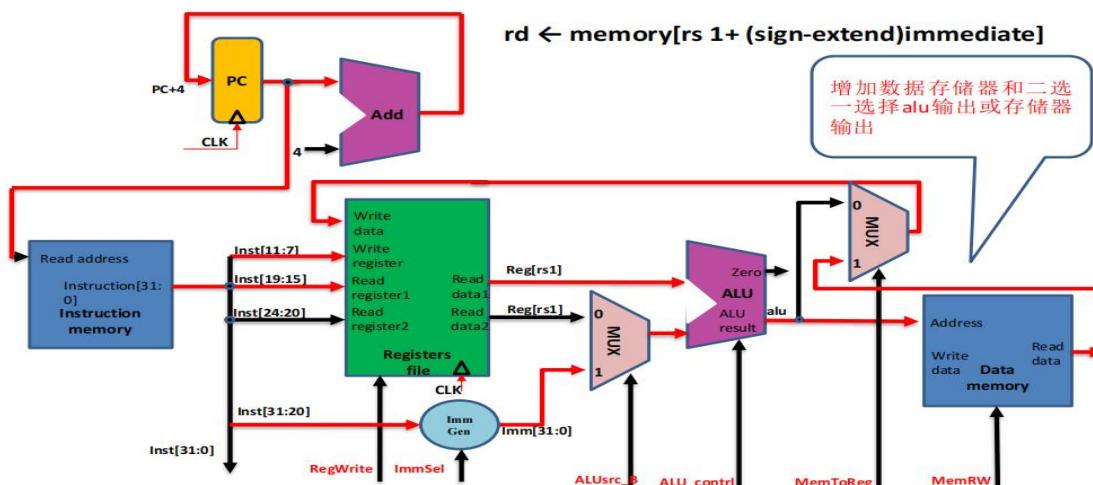


lw rd , immediate(rs1) 读存储器

功能: $rd \leftarrow memory[rs1 + (sign-extend)immediate]$; immediate符号扩展再相加。即读取rs1寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数, 然后保存到rd寄存器中。

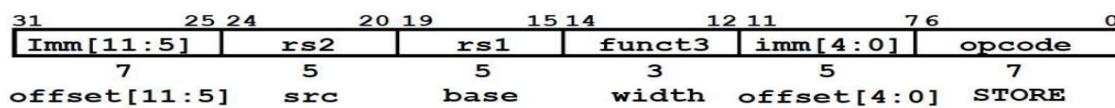
Eg: lw x9,240(x10) x9 = memory[x10+240]
0000111 10000_01010_010_01001_0000011

- Datapath added lw



2.7.4 sw

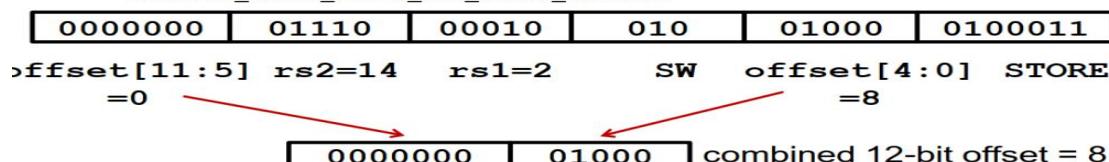
- Instruction



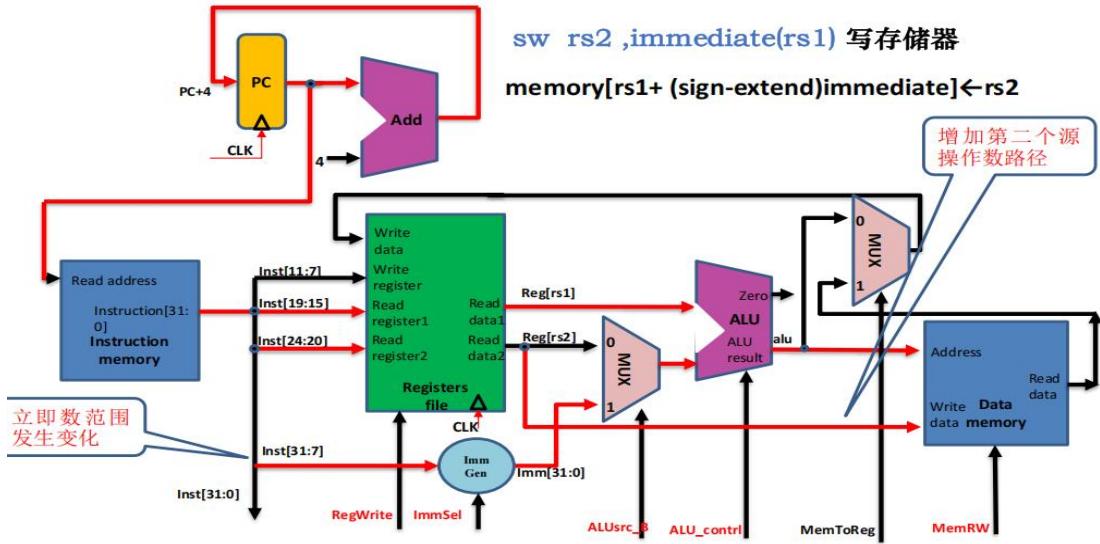
sw rs2 , immediate(rs1) 写存储器

功能: $memory[rs1 + (sign-extend)immediate] \leftarrow rs2$; immediate 符号扩展再相加。即将rs2寄存器的内容保存到rs1寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

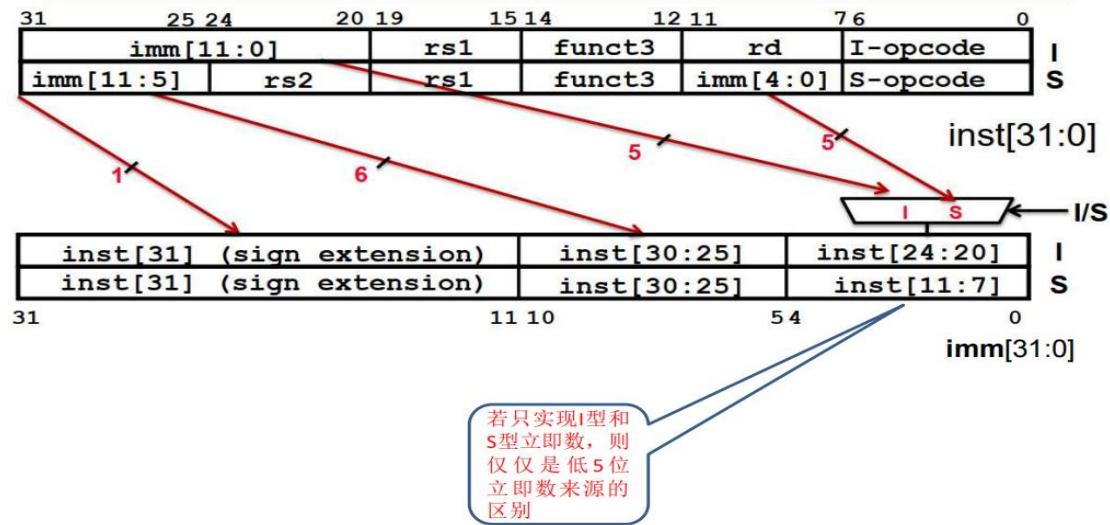
Eg: sw x14, 8(x2)
0000000_01110_00010_010_01000_0100011



• Datapath added sw



2.7.5 I+S Immediate Generation



2.8 B-Format Instruction

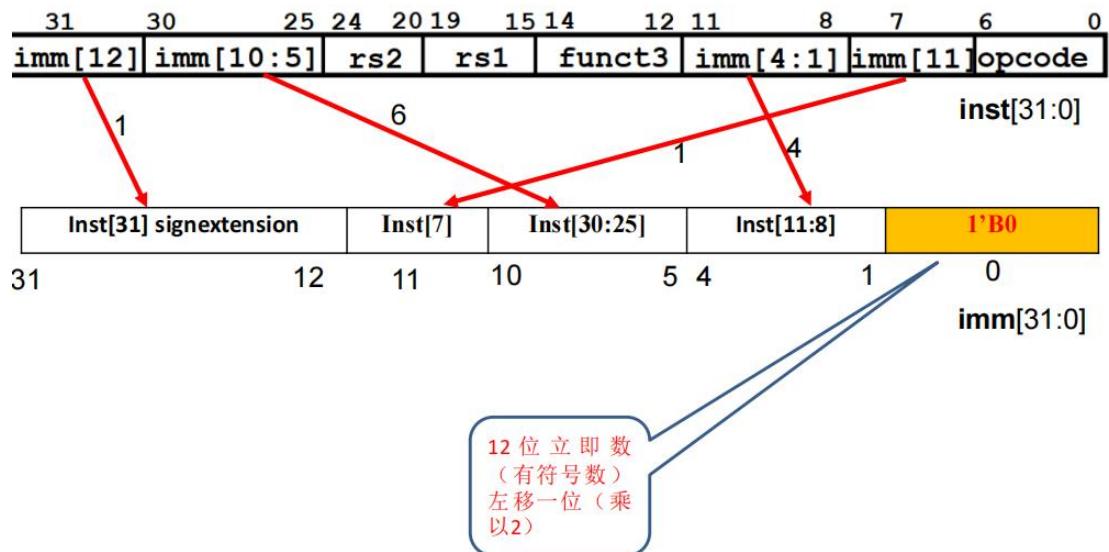
2.8.1 Instruction



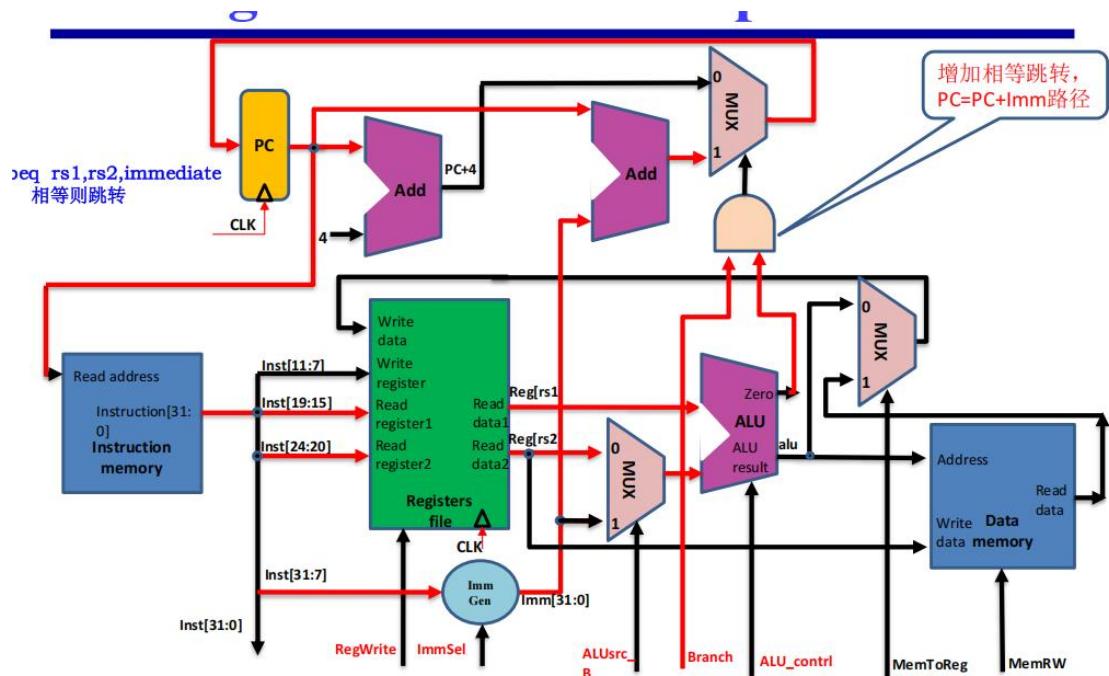
功能: if(rs1==rs2) pc←pc + (sign-extend)immediate<<1 else pc ←pc + 4

Eg `beq x5,x6,100`
`0000 0110 0110 0010 1000 0010 0110 0011`

2.8.2 Branches Immediate Generation

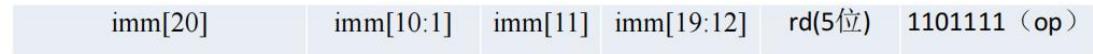
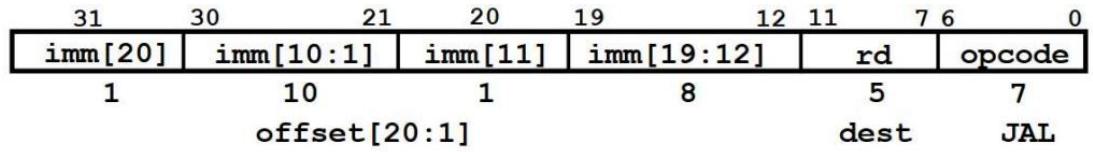


2.8.3 Datapath



2.9 J-Format Instruction

2.9.1 Jal Instruction



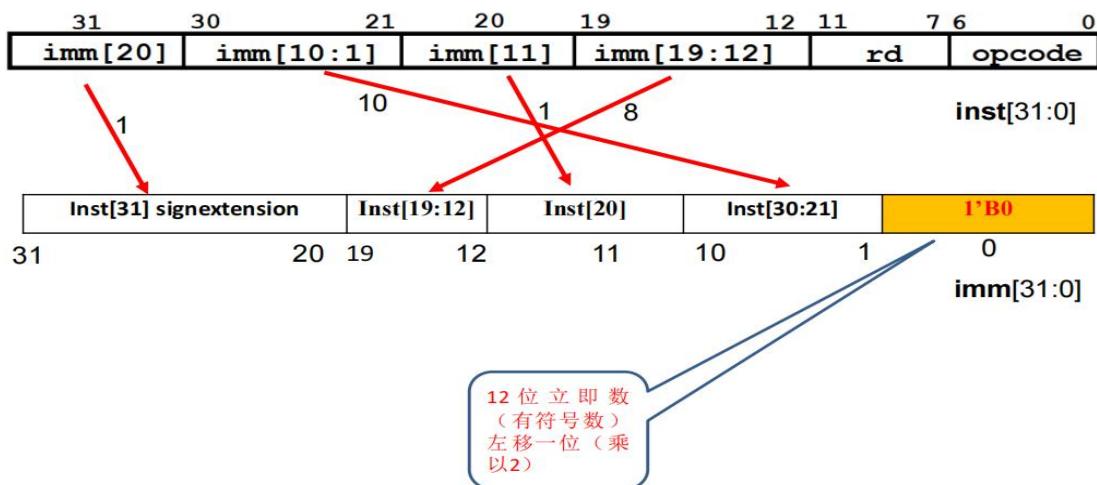
jal rd, immediate 跳转链接

功能: $rd = pc + 4$, $pc \leftarrow pc + (\text{sign-extend})\text{immediate} \ll 1$

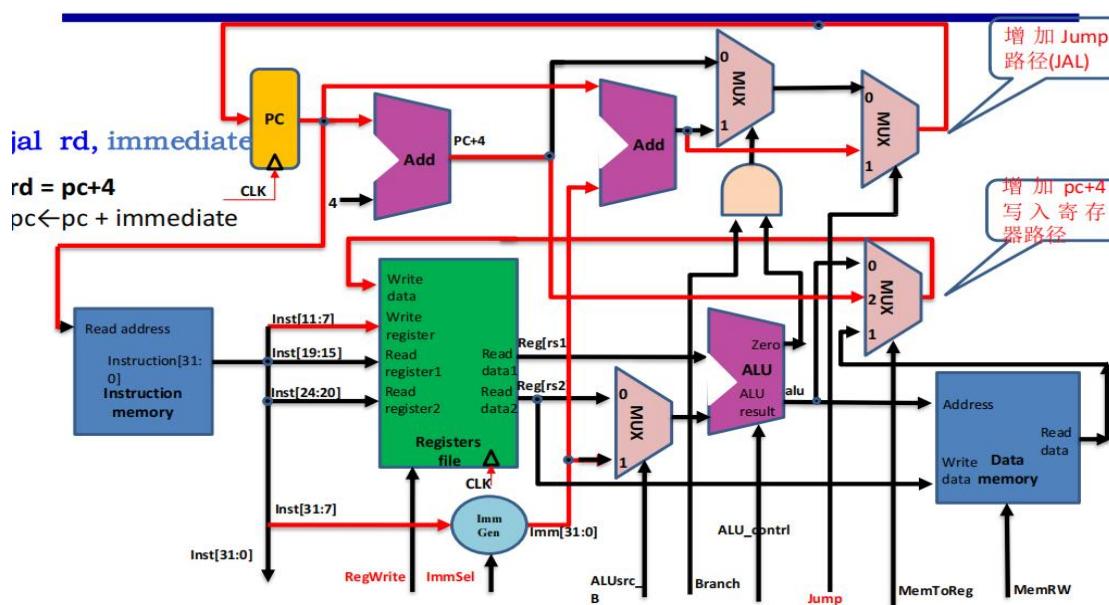
Eg jal x1,100

0000 0110 0100 0000 0000 0000 1110 1111

2.9.2 Jal Immediate Generation



2.9.3 Datapath



2.10 控制信号定义

信号	源数目	功能定义	赋值0时动作	赋值1时动作	赋值2时动作
ALUSrc_B	2	ALU端口B输入选择	选择源操作数寄存器2数据	选择32位立即数（符号扩展后）	-
MemToReg	3	寄存器写入数据选择	选择ALU输出	选择存储器数据	选择PC+4
Branch	2	Beq指令目标地址选择	选择PC+4地址	选择转移目的地址 PC+imm (zero=1)	-
Jump	3	J指令目标地址选择	由Branch决定输出	选择跳转目标地址 PC+imm (JAL)	-
RegWrite	-	寄存器写控制	禁止寄存器写	使能寄存器写	-
MemRW	-	存储器读写控制	存储器读使能，存储器写禁止	存储器写使能，存储器读禁止	-
ALU_Control	000-111	3位ALU操作控制	参考表ALU_Control (详见实验4-2)		
ImmSel	00-11	2位立即数组合控制	参考表ImmSel (详见实验4-2)		

三、主要仪器设备

- 计算机（Intel Core i9-13980, 16GB 内存）系统
- NEXYS A7 开发板
- Xilinx VIVADO2024.2 及以上开发工具

四、操作方法与实验步骤

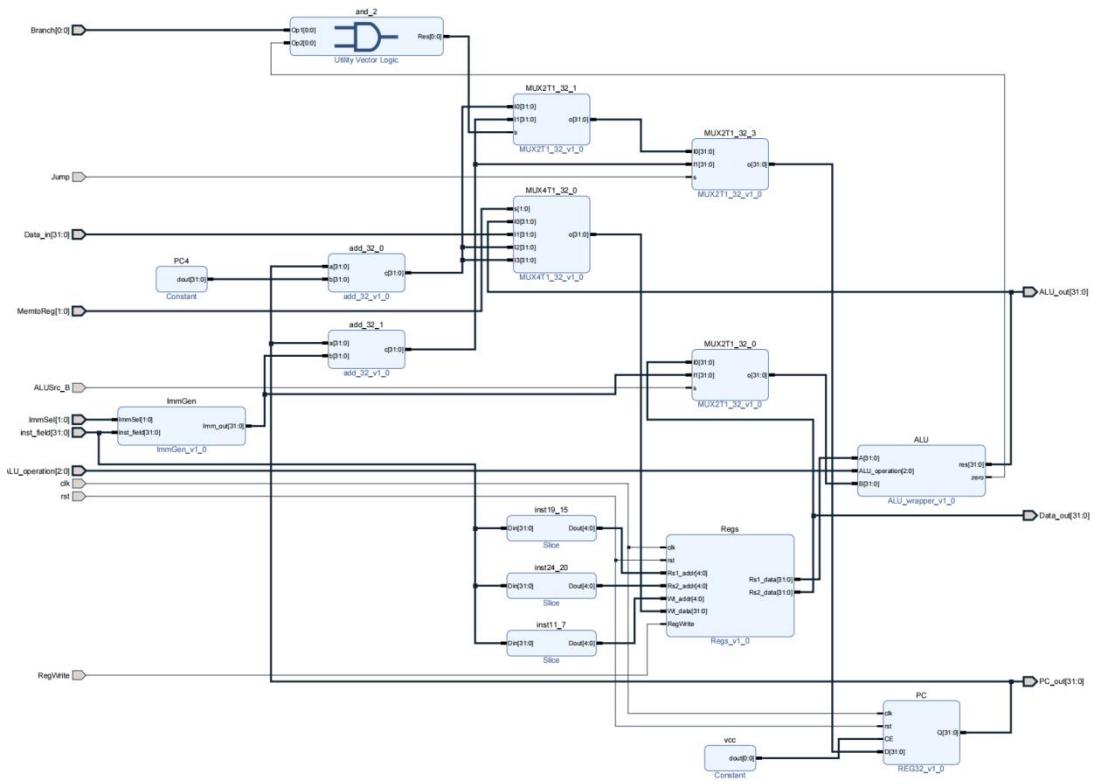
4.1 任务一：设计实现数据通路

4.1.1 设计要点

添加下列模块到当前工程 IP 目录： and32、or32、ADC32、xor32、nor32、sr132、SignalExt_32、mux8to1_32、or_bit_32、add_32、mux2to1_32、mux2to1_5、ALU、Regs、Ext_32、REG32、ImmGe

4.1.2 参考逻辑结构图编写模块代码

- 逻辑结构图



• 关键模块代码

datapath.v 模块代码如下所示：

```

23 module DataPath(
24     input wire Branch,
25     input wire Jump,
26     input wire [31:0] Data_in,
27     input wire [1:0] MemtoReg,
28     input wire ALUSrc_B,
29     input wire [1:0] ImmSel,
30     input wire [31:0] inst_field,
31     input wire [2:0] ALU_Control, // ALU_operation
32     input wire clk,
33     input wire rst,
34     input wire RegWrite,
35     output wire [31:0] ALU_out,
36     output wire [31:0] Data_out,
37     output wire [31:0] PC_out
38 );
39

```

```

40     wire [31:0] ImmGen_0_Imm_out;
41     wire [31:0] Add_32_0_c;
42     wire [31:0] Add_32_1_c;
43     wire [31:0] MUX2T1_32_1_o;
44     wire [31:0] MUX4T1_32_0_o;
45     wire [31:0] MUX2T1_32_3_o;
46     wire [31:0] MUX2T1_32_0_o;
47     wire [31:0] Regs_0_Rs1_data;
48     wire [31:0] Regs_0_Rs2_data;
49     wire ALU_0_zero;
50     wire [31:0] ALU_0_res;
51
52     wire [31:0] PC_Q;
53
54     ImmGen ImmGen_0(
55       .ImmSel(ImmSel),
56       .inst_field(inst_field),
57       .Imm_out(ImmGen_0_Imm_out)
58     );
59
60     Add_32 Add_32_0(
61       .a(PC_Q),
62       .b(32'd0004),
63       .c(Add_32_0_c)
64     );
65
66     Add_32 Add_32_1(
67       .a(PC_Q),
68       .b(ImmGen_0_Imm_out),
69       .c(Add_32_1_c)
70     );
71
72     MUX2T1_32 MUX2T1_32_1(
73       .I0(Add_32_0_c),
74       .I1(Add_32_1_c),
75       .sel/Branch & ALU_0_zero),
76       .O(MUX2T1_32_1_o)
77     );
78
79     MUX4T1_32 MUX4T1_32_0(
80       .S(MemtoReg),
81       .I0(ALU_0_res),
82       .I1(Data_in),
83       .I2(Add_32_0_c),
84       .I3(Add_32_0_c),
85       .O(MUX4T1_32_0_o)
86     );
87
88     MUX2T1_32 MUX2T1_32_3(
89       .I0(MUX2T1_32_1_o),
90       .I1(Add_32_1_c),
91       .sel(Jump),
92       .O(MUX2T1_32_3_o)
93     );
94

```

```

95      MUX2T1_32 MUX2T1_32_0(
96        .IO(Regs_0_Rs2_data),
97        .I1(ImmGen_0_Imm_out),
98        .sel(ALUSrc_B),
99        .O(MUX2T1_32_0_o)
100      );
101

102      Register_32M32b Regs_0(
103        .clk(clk),
104        .rst(rst),
105        .RegWrite(RegWrite),
106        .Rs1_addr(inst_field[19:15]),
107        .Rs2_addr(inst_field[24:20]),
108        .Wt_addr(inst_field[11:7]),
109        .Wt_data(MUX4T1_32_0_O),
110        .Rs1_data(Regs_0_Rs1_data),
111        .Rs2_data(Regs_0_Rs2_data)
112      );
113

114      ALU ALU_0(
115        .A(Regs_0_Rs1_data),
116        .B(MUX2T1_32_0_o),
117        .operation(ALU_Control),
118        .res(ALU_0_res),
119        .zero(ALU_0_zero)
120      );
121

122      REG_32 PC(
123        .clk(clk),
124        .rst(rst),
125        .CE(1'b1),
126        .D(MUX2T1_32_3_o),
127        .Q(PC_Q)
128      );
129

130      assign ALU_out = ALU_0_res;
131      assign Data_out = Regs_0_Rs2_data;
132      assign PC_out = PC_Q;
133
134 endmodule

```

4.2 任务二：设计数据通路测试方案并完成测试

在完成 Datapath.v 顶层模块的设计之后，我们使用自己设计的 Datapath.v 模块替换 Experiment0 实验中 SCPU.v 模块中的 Datapath.v 模块。之后将替换后的 SCPU.v 模块接入到 CPU 测试环境中，运行实验课程提供的参考程序并观察结果。

4.2.1 物理验证-DEMO 接口功能

将比特流烧录至开发板后，调节 SW[8]使得 CPU 切换至单步时钟，之后周期性波动 SW[10]，每一步调节分别为 SW[7:5]010 和 111 时观察七段数码管的显示结果。

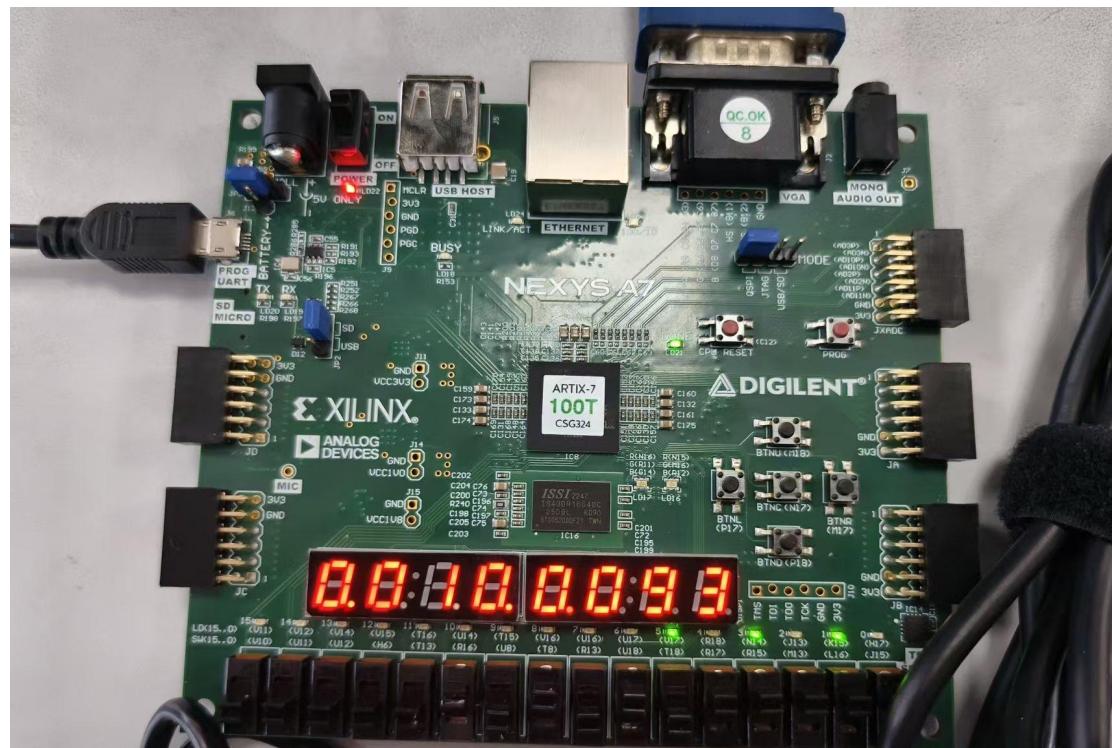
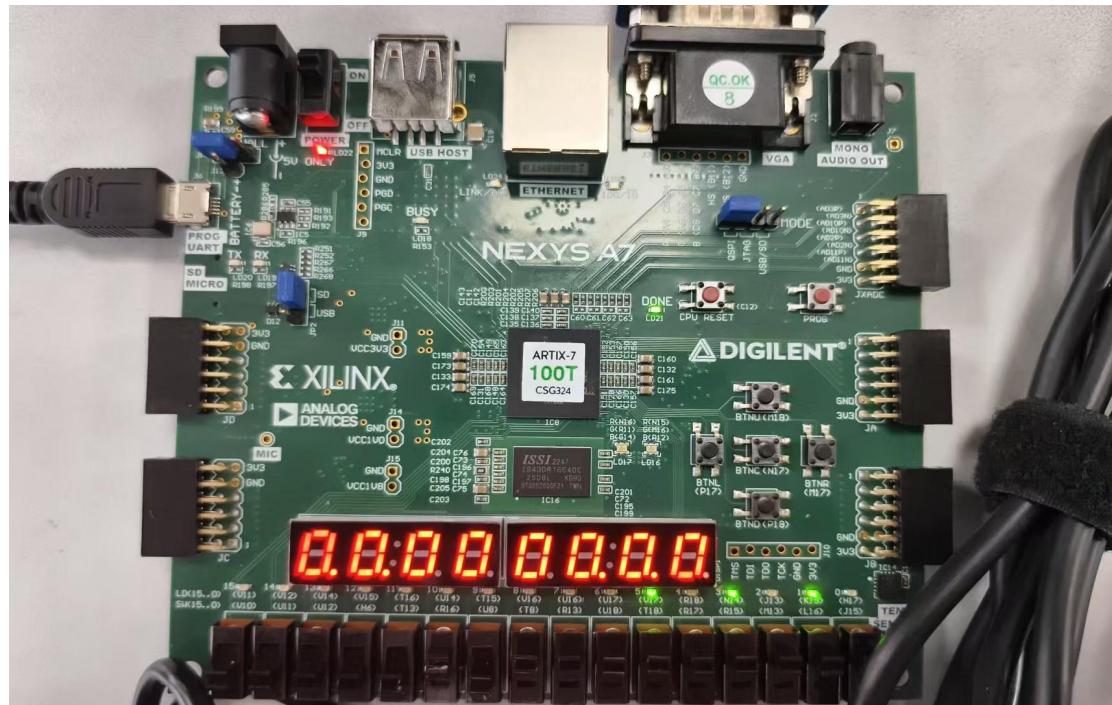
4.2.2 物理验证-VGA 显示输出

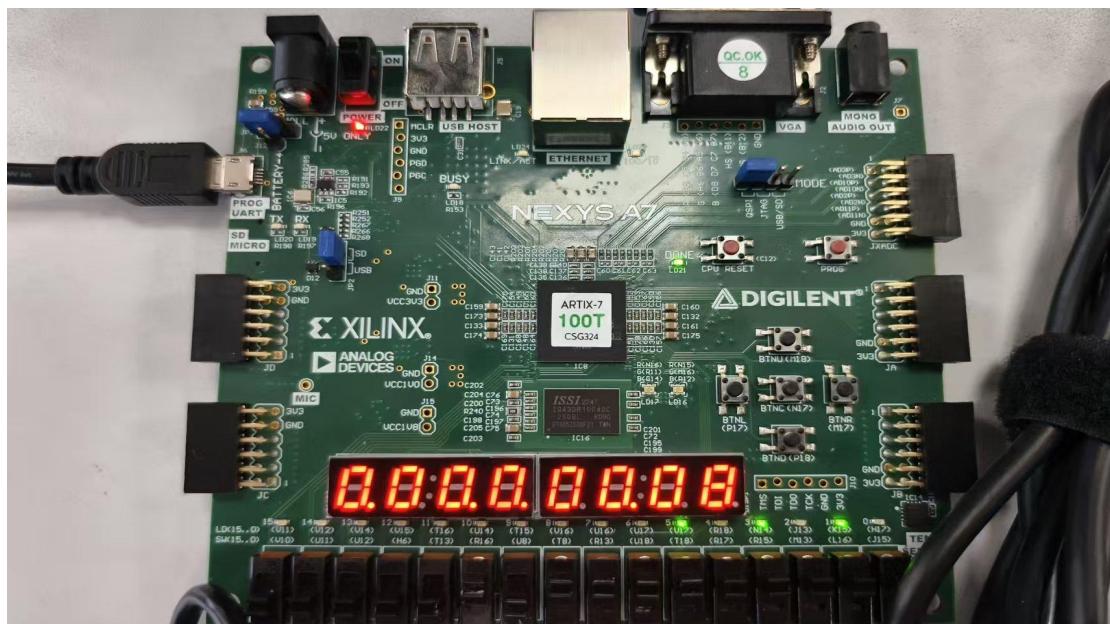
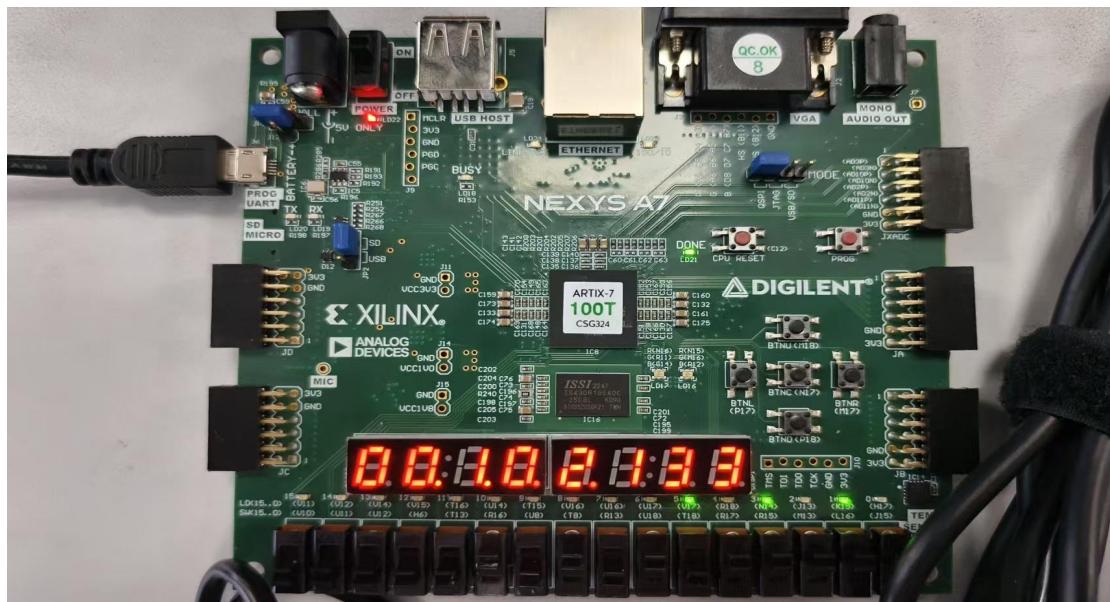
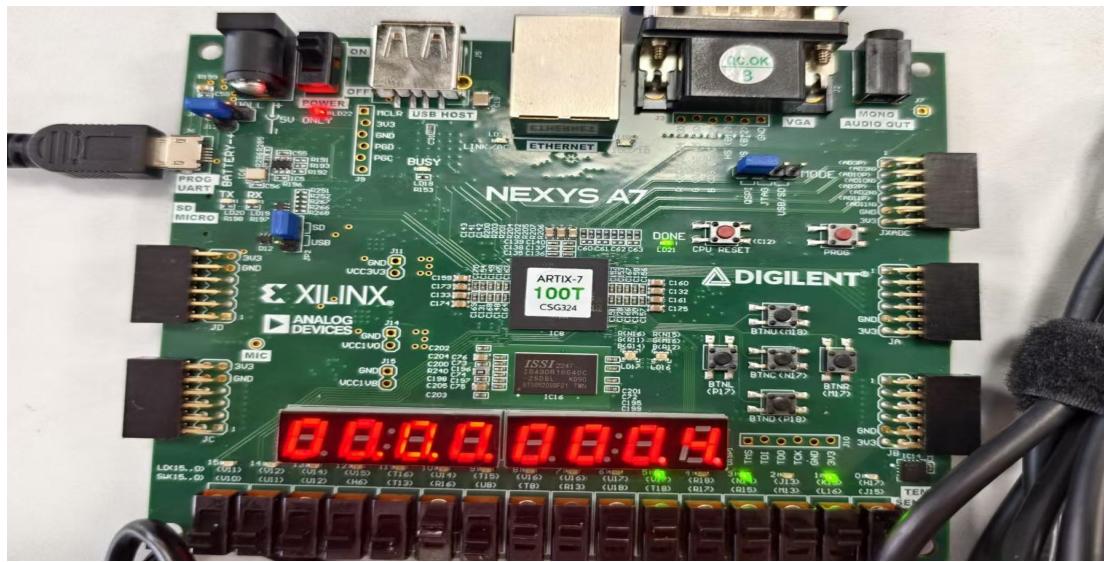
将生成的比特流烧录至开发板后，调节 SW[8]使得 CPU 切换至单步时钟，之后周期性波动 SW[10]，观察 VGA 显示器上左上的“PC”和“inst”参数的值。

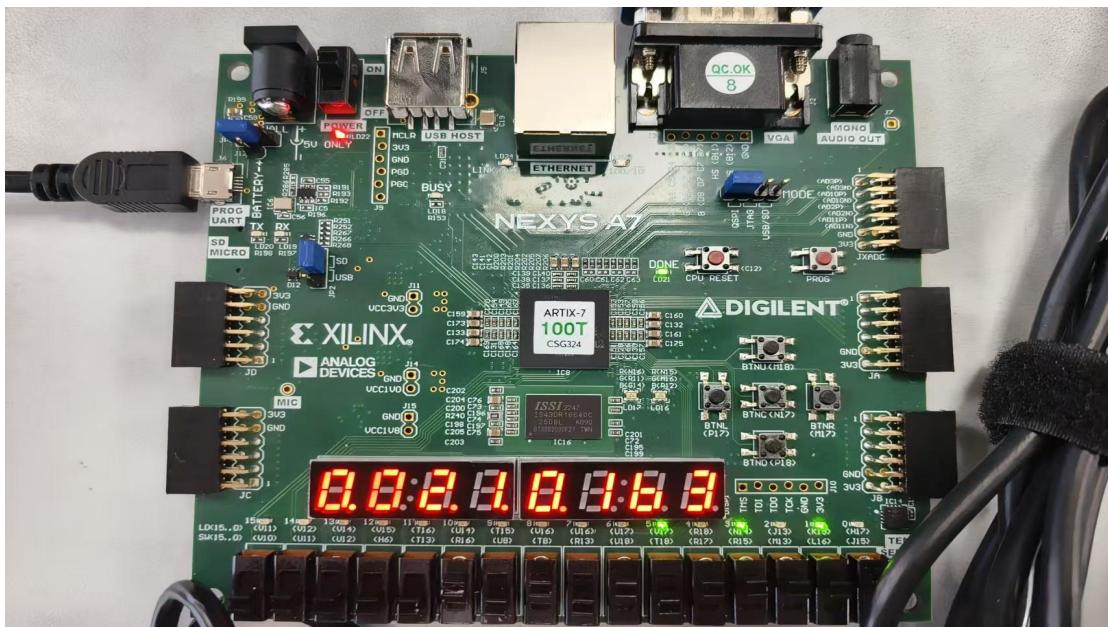
五、实验数据记录和处理

5.1 替换验证

5.2.1 物理验证-DEMO 接口功能







5.2.2 物理验证-VGA 显示输出

```
RV32I Single Cycle CPU
pc: 00000000 inst: 00100093
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0
is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000001 cmp_res: 0
is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000
mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000000 dmem_addr: 00000001
csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

```
RV32I Single Cycle CPU
pc: 00000004 inst: 00102133
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0
is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0
is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000
mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000001 dmem_addr: 00000000
csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

```

R032I Single Cycle CPU
pc: 00000008 inst: 002101b3
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

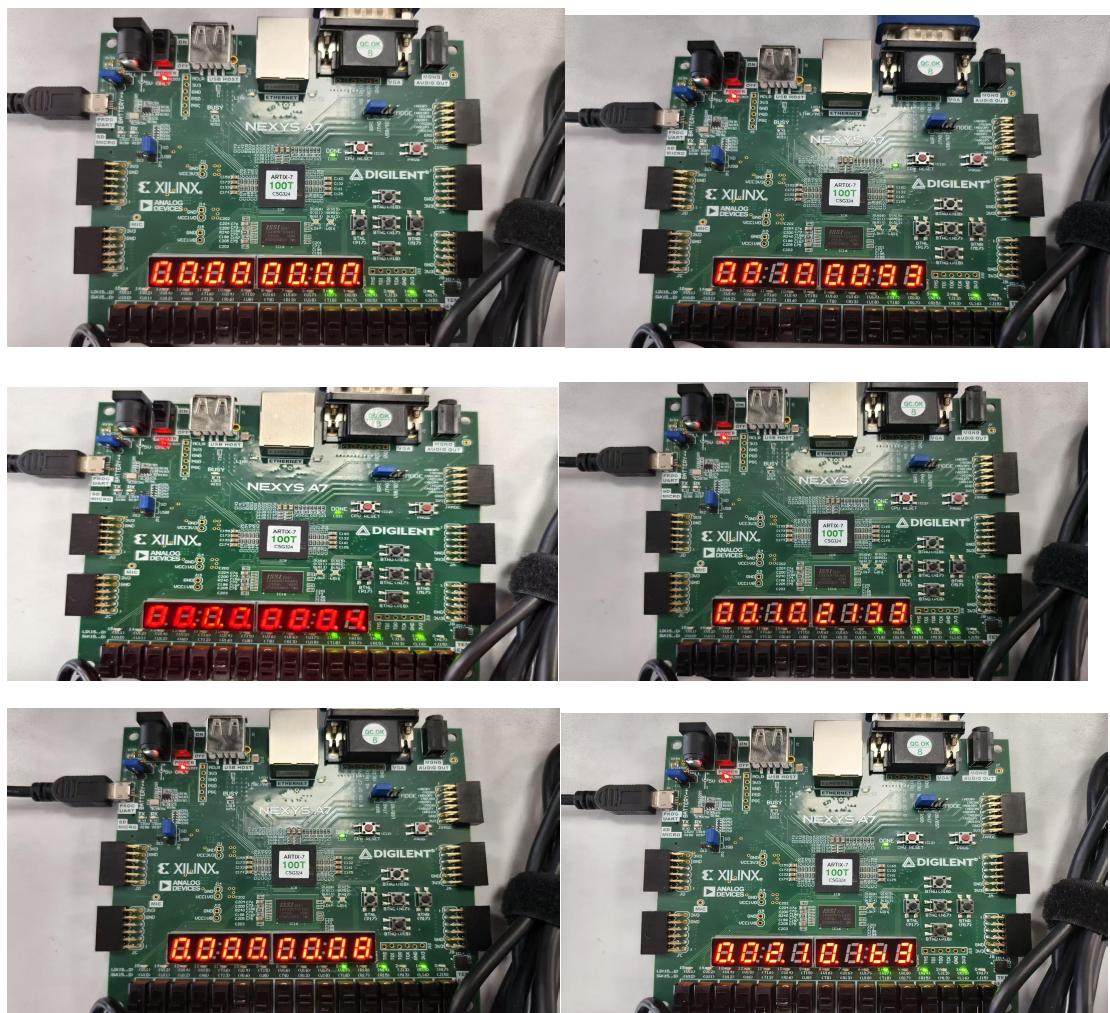
mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000000 dmem_addr: 00000000
csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

六、实验结果与分析

6.1 替换验证

6.2.1 物理验证-DEMO 接口功能



通过三步手动时钟的运行，根据七段数码管现实的程序计数器和指令地址，我们可以看到相关的实验结果都是正确地。

6.2.2 物理验证-VGA 显示输出

```
RV32I Single Cycle CPU

pc: 0000000000000000 inst: 00100093

x0: 0000000000000000 ra: 0000000000000000 sp: 0000000000000000 gp: 0000000000000000 tp: 0000000000000000
t0: 0000000000000000 t1: 0000000000000000 t2: 0000000000000000 s0: 0000000000000000 s1: 0000000000000000
a0: 0000000000000000 a1: 0000000000000000 a2: 0000000000000000 a3: 0000000000000000 a4: 0000000000000000
a5: 0000000000000000 a6: 0000000000000000 a7: 0000000000000000 s2: 0000000000000000 s3: 0000000000000000
s4: 0000000000000000 s5: 0000000000000000 s6: 0000000000000000 s7: 0000000000000000 s8: 0000000000000000
s9: 0000000000000000 s10: 0000000000000000 s11: 0000000000000000 t3: 0000000000000000 t4: 0000000000000000
t5: 0000000000000000 t6: 0000000000000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000001 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000000 dmem_addr: 00000001

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

RV32I Single Cycle CPU

pc: 0000000000000004 inst: 00102133

x0: 0000000000000000 ra: 0000000000000000 sp: 0000000000000000 gp: 0000000000000000 tp: 0000000000000000
t0: 0000000000000000 t1: 0000000000000000 t2: 0000000000000000 s0: 0000000000000000 s1: 0000000000000000
a0: 0000000000000000 a1: 0000000000000000 a2: 0000000000000000 a3: 0000000000000000 a4: 0000000000000000
a5: 0000000000000000 a6: 0000000000000000 a7: 0000000000000000 s2: 0000000000000000 s3: 0000000000000000
s4: 0000000000000000 s5: 0000000000000000 s6: 0000000000000000 s7: 0000000000000000 s8: 0000000000000000
s9: 0000000000000000 s10: 0000000000000000 s11: 0000000000000000 t3: 0000000000000000 t4: 0000000000000000
t5: 0000000000000000 t6: 0000000000000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000001 dmem_addr: 00000000

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

RV32I Single Cycle CPU

pc: 0000000000000008 inst: 002101b3

x0: 0000000000000000 ra: 0000000000000000 sp: 0000000000000000 gp: 0000000000000000 tp: 0000000000000000
t0: 0000000000000000 t1: 0000000000000000 t2: 0000000000000000 s0: 0000000000000000 s1: 0000000000000000
a0: 0000000000000000 a1: 0000000000000000 a2: 0000000000000000 a3: 0000000000000000 a4: 0000000000000000
a5: 0000000000000000 a6: 0000000000000000 a7: 0000000000000000 s2: 0000000000000000 s3: 0000000000000000
s4: 0000000000000000 s5: 0000000000000000 s6: 0000000000000000 s7: 0000000000000000 s8: 0000000000000000
s9: 0000000000000000 s10: 0000000000000000 s11: 0000000000000000 t3: 0000000000000000 t4: 0000000000000000
t5: 0000000000000000 t6: 0000000000000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000000 dmem_addr: 00000000

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

观察三步手动时钟运行的 VGA 显示结果，我们可以看到程序计数器 PC 和指令编码 inst 均符合所设计的 ROM 的内容。

6.2 代码分析

```
23  module DataPath(
24      input wire Branch,
25      input wire Jump,
26      input wire [31:0] Data_in,
27      input wire [1:0] MemtoReg,
28      input wire ALUSrc_B,
29      input wire [1:0] ImmSel,
30      input wire [31:0] inst_field,
31      input wire [2:0] ALU_Control,// ALU_operation
32      input wire clk,
33      input wire rst,
34      input wire RegWrite,
35      output wire [31:0] ALU_out,
36      output wire [31:0] Data_out,
37      output wire [31:0] PC_out
38 );
39
```

该部分代码展示了模块的输入和输出端口。

```
40      wire [31:0] ImmGen_0_Imm_out;
41      wire [31:0] Add_32_0_c;
42      wire [31:0] Add_32_1_c;
43      wire [31:0] MUX2T1_32_1_o;
44      wire [31:0] MUX4T1_32_0_o;
45      wire [31:0] MUX2T1_32_3_o;
46      wire [31:0] MUX2T1_32_0_o;
47      wire [31:0] Regs_0_Rs1_data;
48      wire [31:0] Regs_0_Rs2_data;
49      wire ALU_0_zero;
50      wire [31:0] ALU_0_res;
51
52      wire [31:0] PC_Q;
53
```

该部分代码定义的模块内部使用到的中间变量。

```
54      ImmGen ImmGen_0(
55          .ImmSel(ImmSel),
56          .inst_field(inst_field),
57          .Imm_out(ImmGen_0_Imm_out)
58      );
59
```

该部分代码实例化引用立即数生成模块，产生指令运行所需要的立即数。

```

60      Add_32 Add_32_0(
61          .a(PC_Q),
62          .b(32'>0004),
63          .c(Add_32_0_c)
64      );
65
66      Add_32 Add_32_1(
67          .a(PC_Q),
68          .b(ImmGen_0_Imm_out),
69          .c(Add_32_1_c)
70      );
71

```

该部分代码上下两个 32 位加法器分别计算顺承的下一条指令 (PC+4) 以及跳转时的跳转地址。

```

72      MUX2T1_32 MUX2T1_32_1(
73          .I0(Add_32_0_c),
74          .I1(Add_32_1_c),
75          .sel/Branch & ALU_0_zero),
76          .O(MUX2T1_32_1_o)
77      );
78

```

该部分代码通过对顺承指令地址和跳转指令地址根据相应控制信号的逻辑关系进行选择来实现下一条指令地址的正确计算。

```

79      MUX4T1_32 MUX4T1_32_0(
80          .S(MemtoReg),
81          .I0(ALU_0_res),
82          .I1(Data_in),
83          .I2(Add_32_0_c),
84          .I3(Add_32_0_c),
85          .O(MUX4T1_32_0_o)
86      );
87

```

该部分代码通过以写入处理器寄存器的数据来源为判断信号，对写入寄存器的数据进行选择。

```

88      MUX2T1_32 MUX2T1_32_3(
89          .I0(MUX2T1_32_1_o),
90          .I1(Add_32_1_c),
91          .sel(Jump),
92          .O(MUX2T1_32_3_o)
93      );
94

```

该部分代码通过以 Jump 控制信号为判断信号，对是否为 Jal 指令链接到的寄存器应存放的数据进行选择。

```

95      MUX2T1_32 MUX2T1_32_0(
96          .I0(Regs_0_Rs2_data),
97          .I1(ImmGen_0_Imm_out),
98          .sel(ALUSrc_B),
99          .O(MUX2T1_32_0_o)
100     );
101

```

该部分代码在寄存器 2 的数据以及生成的立即数之间根据 ALUSrc_B 信号进

行选择作为 ALU 计算的数据。

```
102     Register_32M32b Regs_0(
103         .clk(clk),
104         .rst(rst),
105         .RegWrite(RegWrite),
106         .Rs1_addr(inst_field[19:15]),
107         .Rs2_addr(inst_field[24:20]),
108         .Wt_addr(inst_field[11:7]),
109         .Wt_data(MUX4T1_32_0_o),
110         .Rs1_data(Regs_0_Rs1_data),
111         .Rs2_data(Regs_0_Rs2_data)
112     );
113
```

该部分代码实例化引用 32 个 32 位寄存器阵列，作为处理器内部的寄存器。

```
114     ALU ALU_0(
115         .A(Regs_0_Rs1_data),
116         .B(MUX2T1_32_0_o),
117         .operation(ALU_Control),
118         .res(ALU_0_res),
119         .zero(ALU_0_zero)
120     );
121
```

该部分代码实例化引用 ALU 逻辑运算单元进行所需的计算：数据结果的计算、地址的计算。

```
122     REG_32 PC(
123         .clk(clk),
124         .rst(rst),
125         .CE(1'b1),
126         .D(MUX2T1_32_3_o),
127         .Q(PC_Q)
128     );
129
```

该部分代码实例化引用 PC 计算器模块，对下一条指令的地址进行正确的选择。

```
130     assign ALU_out = ALU_0_res;
131     assign Data_out = Regs_0_Rs2_data;
132     assign PC_out = PC_Q;
133
134 endmodule
```

该部分代码最终完成 Datapath.v 模块的相应输出。

6.3 实验结果分析

根据七段数码管以及 VGA 的输出显示，均符合我们所运行的程序的预期，因此可以认为我们设计的处理器的数据通路模块正确，可以正常使用。

七、讨论、心得

通过本次 CPU 设计之数据通路实验，我们从课堂上的相关指令的格式、效果出发，根据逻辑从而设计出相应的电路、选择出适合的组合或者时序模式，最终不断的完善处理器所需要的数据通路。

本实验过程深刻体现了由简到繁的电路设计想法，体现出理论对于实际设计

的指导作用，可以说数据通路的设计和 RSICV 指令集一样精简，非常舒服。

但是在实验过程中，我遇到了连线错误、忘记声明中间变量的问题，还有在设计 PC 计数器时对于时序逻辑的理解并不透彻，看似简单的内容却正是对基础的考量，让我深刻意识到了基础的重要性。我通过仔细阅读综合、实现报错一步一步解决了语法、连线上的错误，之后根据 VGA 显示解决了 PC 计数器的逻辑问题。

Experiment2-CPU 设计之控制器

一、实验目的和要求

1.1 实验目的

1. 运用寄存器传输控制技术
2. 掌握 CPU 的核心：指令执行过程与控制流关系
3. 设计控制器
4. 学习测试方案的设计
5. 学习测试程序的设计

1.2 实验目标及任务

• 目标：熟悉 RISC-V RV32I 的指令特点，了解控制器的原理，设计并测试控制器

• 任务一：用硬件描述语言设计实现控制器

根据 Exp04-1 数据通路及指令编码完成控制信号真值表

此实验在 Exp04-1 的基础上完成，替换 Exp04-1 的控制器核

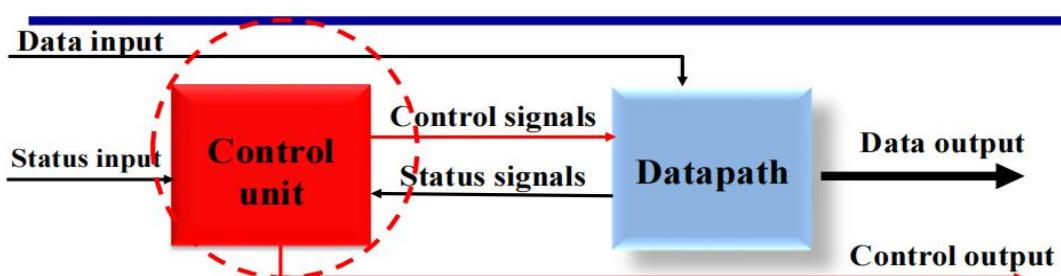
• 任务二：设计控制器测试方案并完成测试

OP 译码测试：R-格式、访存指令、分支指令，转移指令

运算控制测试：Function 译码测试

二、实验内容和原理

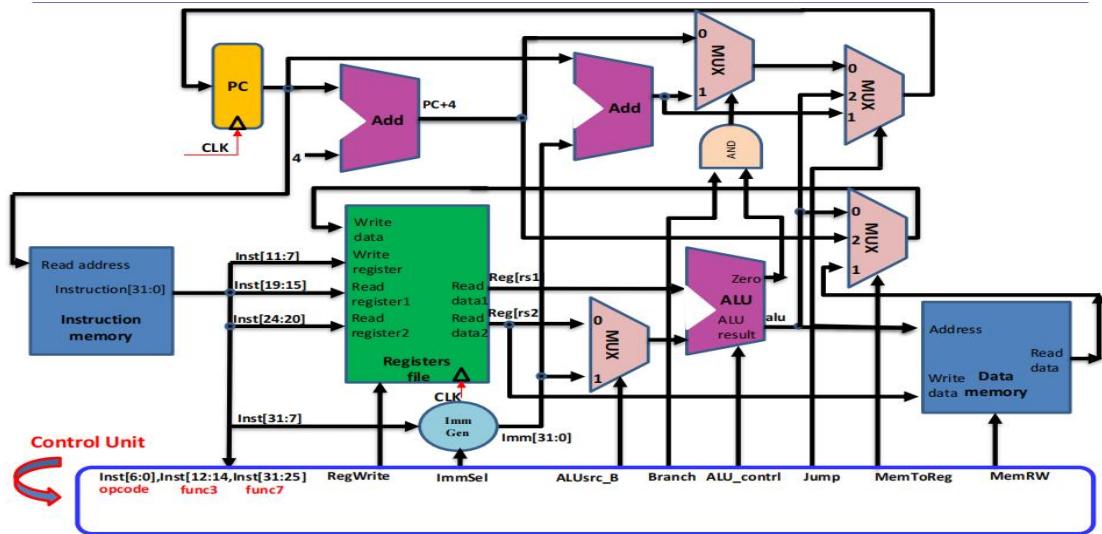
2.1 Control unit



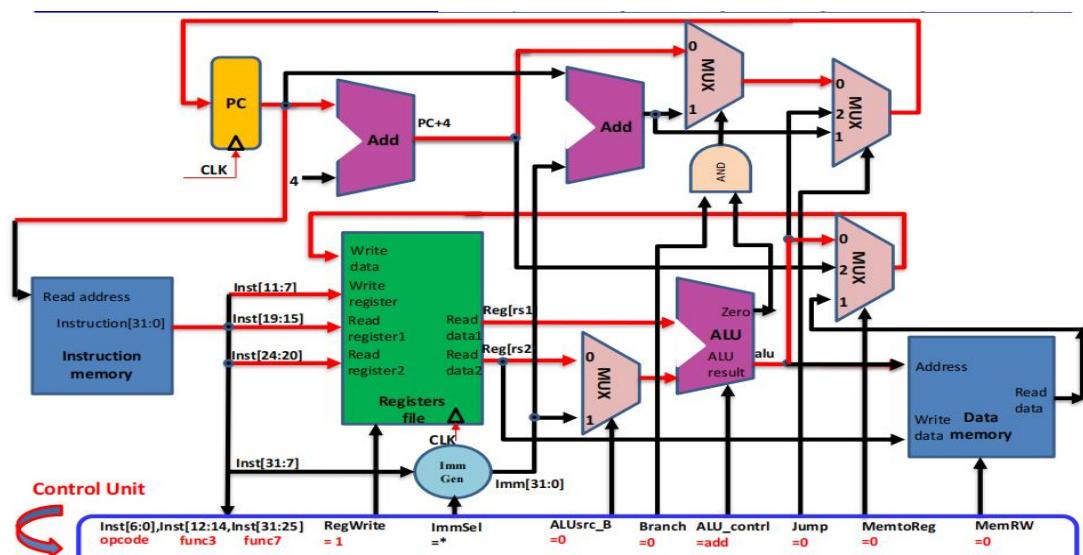
□ 控制单元

□ 控制单元作为处理器的一部分，用以告诉数据通路需要做什么。包含了取指单元（PC 及地址计算单元）、译码单元、控制单元（时序信号形成及微操作控制信号形成电路）、中断等

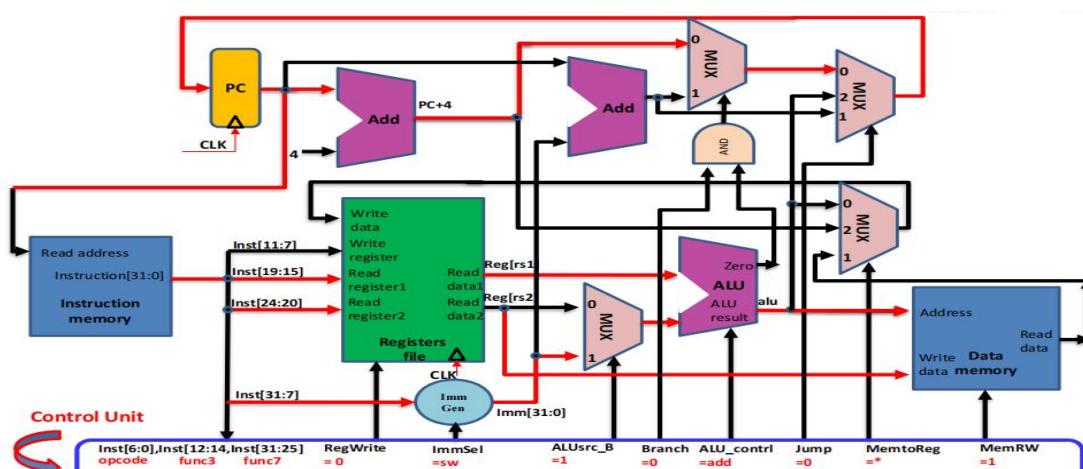
2.2 数据通路结构



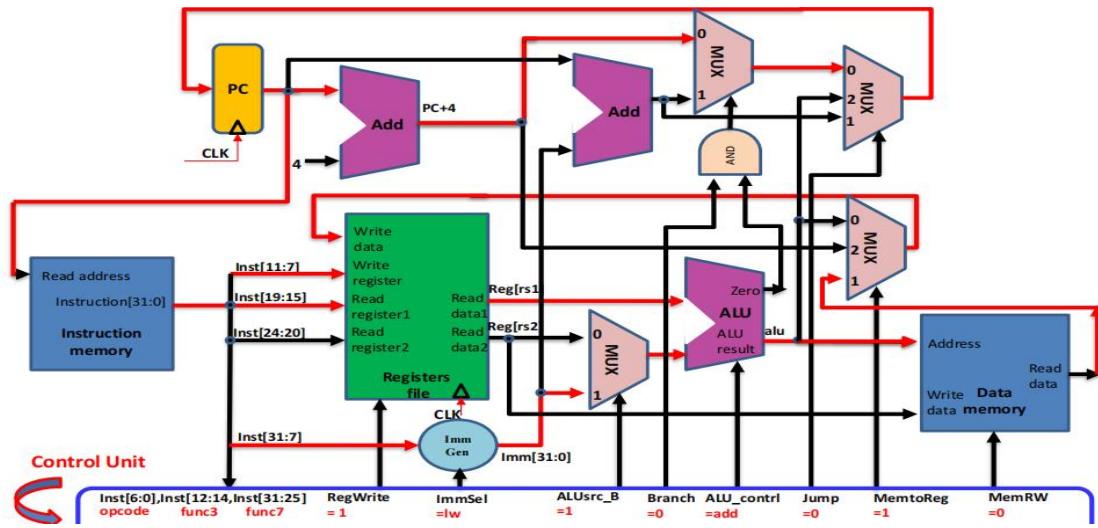
2.3 add



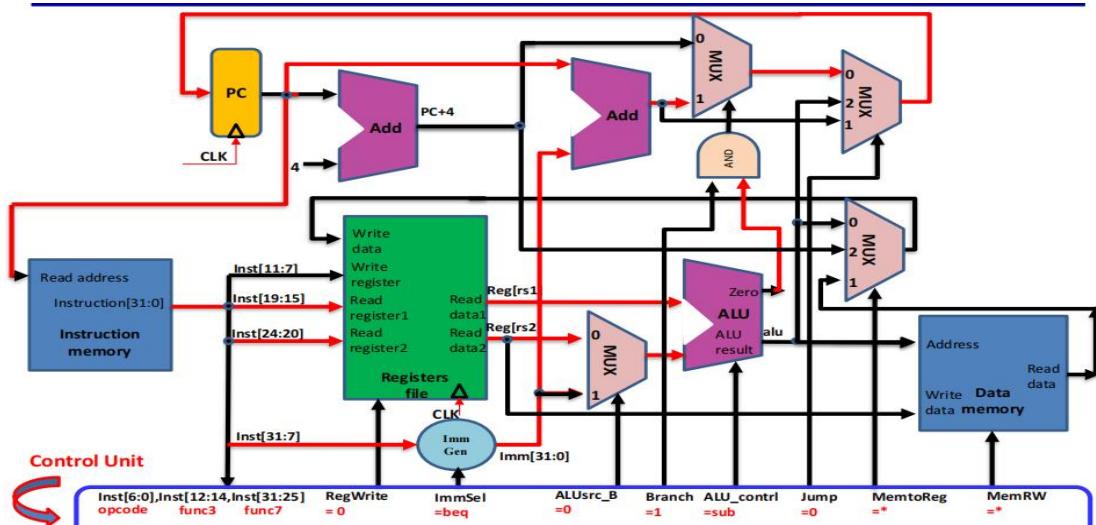
2.4 sw



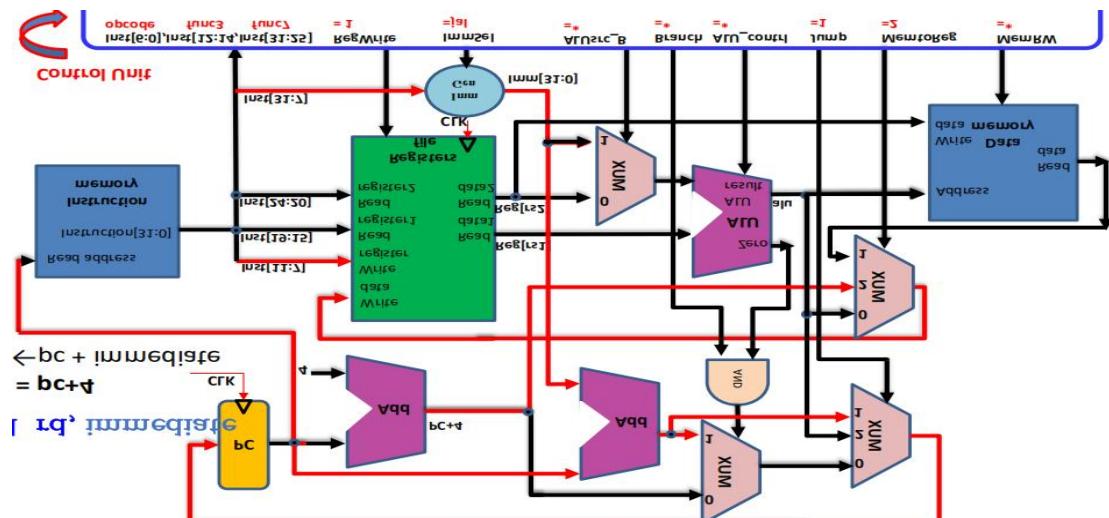
2.5 lw



2.6 beq



2.7 jal



2.8 主控制器信号

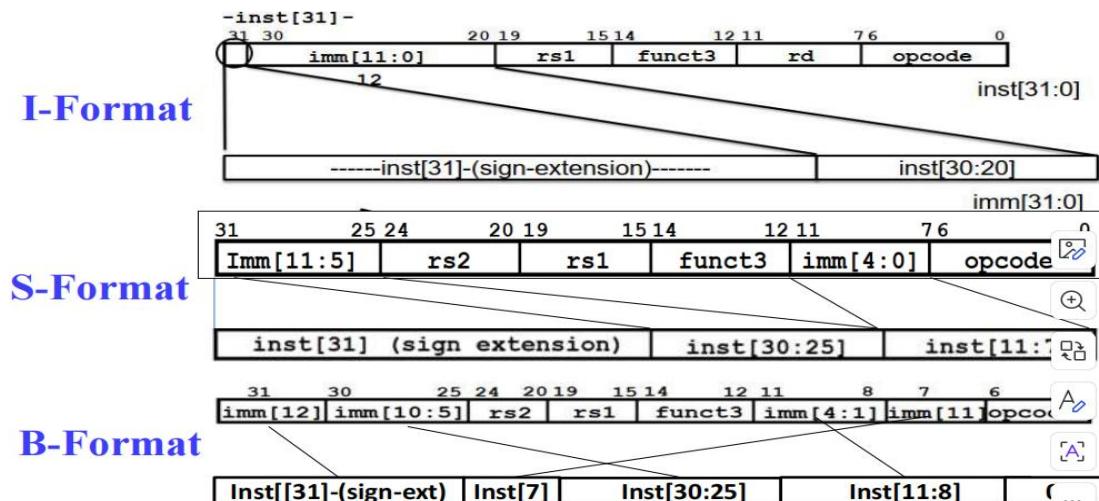
- 定义

信号	源数目	功能定义	赋值0时动作	赋值1时动作	赋值2时动作
ALUSrc_B	2	ALU端口B输入选择	选择源操作数寄存器2数据	选择32位立即数(符号扩展后)	-
MemToReg	3	寄存器写入数据选择	选择ALU输出	选择存储器数据	选择PC+4
Branch	2	Beq指令目标地址选择	选择PC+4地址	选择转移目的地址 PC+imm (zero=1)	-
Jump	3	J指令目标地址选择	由Branch决定输出	选择跳转目标地址 PC+imm (JAL)	-
RegWrite	-	寄存器写控制	禁止寄存器写	使能寄存器写	-
MemRW	-	存储器读写控制	存储器读使能,存储器写禁止	存储器写使能,存储器读禁止	-
ALU_Control	000-111	3位ALU操作控制	参考表ALU_Control		
ImmSel	000-111	3位立即数组合控制	参考表ImmSel		

- 真值表

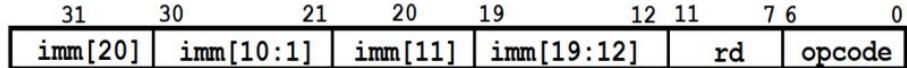
Inst[31:0]	Banch	Jump	ImmSel	ALUSrc_B	ALU_Control	MemRW	RegWrite	MemtoReg
add	0	0	*	Reg	Add	Read	1	ALU
sub	0	0	*	Reg	Sub	Read	1	ALU
(R-R Op)	0	0	*	Reg	(Op)	Read	1	ALU
addi	0	0	I	Imm	Add	Read	1	ALU
lw	0	0	I	Imm	Add	Read	1	Mem
sw	0	0	S	Imm	Add	Write	0	*
beq	0	0	B	Reg	Sub	Read	0	*
beq	1	0	B	Reg	sub	Read	0	*
jal	0	1	J	Imm	*	Read	1	PC+4
lui	0	0	U	Imm	Add	Read	1	ALU

2.9 ImmSel

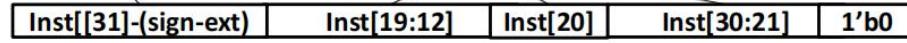




U-Format



J-Format



Instruction type	Instruction opcode[6:0]	Instruction operation	(sign-extend)immediate	Imm Sel
I-type	0000011	Lw;lbu;lh; lb;lhu	(sign-extend) instr[31:20]	00
	0010011	Addi;slti;slti u;xori;ori;a ndi;		
	1100111	jalr		
S-type	0100011	Sw;sb;sh	(sign-extend) instr[31:25],[11:7]	01
B-type	1100011	Beq;bne;blt ;bge;bltu;b geu	(sign-extend) instr[31],[7],[30:25],[11:8], 1'b0	10
J-type	1101111	jal	(sign-extend) instr[31],[19:12],[20],[30:21],1 'b0	11

2.10 RSICV-encode

imm[31:12]	rd	0110111	LUI	AUIPC	inst[30]	inst[14:12]	inst[6:2]		
imm[31:12]	rd	0010011	JAL	JALR	0000000	shamt	rs1 001	rd 0010011	SLL
imm[20:10][11:9:12]	rd	1101111			0000000	shamt	rs1 101	rd 0010011	SRRI
imm[11:0]	rs1	000	rd	BEQ	0100000	shamt	rs1 101	rd 0010011	SRRAI
imm[12:10:5]	rs2	rs1 001	imm[4:1][11]	BNE	0100000	shamt	rs1 000	rd 0110011	ADD
imm[12:10:5]	rs2	rs1 100	imm[4:1][11]	BLT	0000000	rs2	rs1 000	rd 0110011	SUB
imm[12:10:5]	rs2	rs1 101	imm[4:1][11]	BGE	0100000	rs2	rs1 001	rd 0110011	SLL
imm[12:10:5]	rs2	rs1 110	imm[4:1][11]	BGEU	0000000	rs2	rs1 010	rd 0110011	SLT
imm[12:10:5]	rs2	rs1 111	imm[4:1][11]	BLTU	0000000	rs2	rs1 101	rd 0110011	SLTU
imm[11:0]	rs1	000	rd	0000011	LB	0000000	rs2 011	rd 0110011	XOR
imm[11:0]	rs1	001	rd	0000011	LH	0000000	rs2 100	rd 0110011	SRL
imm[11:0]	rs1	010	rd	0000011	LW	0000000	rs2 101	rd 0110011	SRA
imm[11:0]	rs1	100	rd	0000011	LBU	0000000	rs2 101	rd 0110011	OR
imm[11:0]	rs1	101	rd	0000011	LHU	0100000	rs2 110	rd 0110011	AND
imm[11:5]	rs2	rs1 000	imm[4:0]	0100011	SB	0000000	rs2 111	rd 0110011	I fence.i
imm[11:5]	rs2	rs1 001	imm[4:0]	0100011	SLTI	0000000	000	00000	I fence.j
imm[11:5]	rs2	rs1 010	imm[4:0]	0100011	SLTIU	0000000	001	00000	I ecall
imm[11:5]	rs2	rs1 011	imm[4:0]	0100011	XORI	0000000	000	00000	I break
imm[11:5]	rs2	rs1 100	imm[4:0]	0100011	ORI	0000000	000	00000	I csr.w
imm[11:5]	rs2	rs1 110	imm[4:0]	0100011	ANDI	0000000	000	00000	I csr.r
imm[11:0]	rs1	010	rd	0010011	ADDI	0000000	001	00000	I csr.wi
imm[11:0]	rs1	011	rd	0010011	SLTI	0000000	010	00000	I csr.rj
imm[11:0]	rs1	100	rd	0010011	SLTIU	0000000	011	00000	I csr.e
imm[11:0]	rs1	110	rd	0010011	XORI	0000000	100	00000	I csr.wi
imm[11:0]	rs1	111	rd	0010011	ORI	0000000	110	00000	I csr.rj
imm[11:0]	rs1	111	rd	0010011	ANDI	0000000	111	00000	I csr.e

Instruction type encoded using only 9 bits
inst[30], inst[14:12], inst[6:2]

0000	pred	succ	00000	000	00000	0001111	I fence
0000	0000	0000	00000	001	00000	0001111	I fence.i
000000000000			00000	000	00000	1110011	I ecall
000000000001			00000	000	00000	1110011	I break
csr	rs1	001	rd	1110011			I csr.w
csr	rs1	010	rd	1110011			I csr.r
csr	rs1	011	rd	1110011			I csr.e
csr	zimm	101	rd	1110011			I csr.wi
csr	zimm	110	rd	1110011			I csr.rj
csr	zimm	111	rd	1110011			I csr.ei

特殊指令本实验暂不做考虑

2.11 RSICV-decode

Instruction opcode	op	Instruction operation	Funct 3	Funct7	ALUop	Desired ALU action	ALUControl
R-type	0110011	add	000	0000000	10	add	010
		sub	000	0100000		sub	110
		sll	001	0000000		sll	-
		slt	010	0000000		slt	111
		situ	011	0000000		situ	-
		xor	100	0000000		xor	011
		srl	101	0000000		srl	101
		sra	101	0100000		sra	-
		or	110	0000000		or	001
		and	111	0000000		and	000

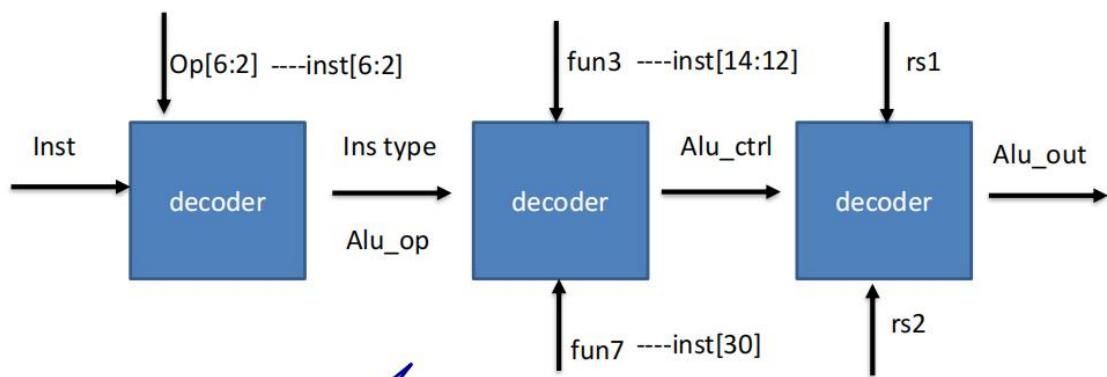
Instruction opcode	op	Instruction operation	Funct 3	Funct7	ALUop	Desired ALU action	ALUControl
S-Type	0100011	sb	000	-	00	add	010
		sh	001	-		add	010
		sw	010	-		add	010
Instruction opcode	op	Instruction operation	Funct 3	Funct7	ALUop	Desired ALU action	ALUControl
B-Type	1100011	Beq	000	-	01	sub	110
		Bne	001	-		sub	110

Instruction opcode	op	Instruction operation	Funct 3	Funct7	ALUop	Desired ALU action	ALUControl
U-Type	0110111	lui	-	-	-	--	-
	0010111	auipc	-	-		-	-
Instruction opcode	op	Instruction operation	Funct 3	Funct7	ALUop	Desired ALU action	ALUControl
J-Type	1101111	jal	-	-	-	-	-

Instruction opcode	op	Instruction operation	Funct 3	Funct7	ALUop	Desired ALU action	ALUControl
I-Type	0000011	Lb	-	-	00	add	010
		Lh	-	-		add	010
		Lw	010	-		add	010
		Lbu	-	-		add	010
		Lhu	-	-		add	010

Instruction opcode	op	Instruction operation	Funct 3	Funct7	ALUop	Desired ALU action	ALUControl
I-Type	0010011	addi	000	-	11	add	010
		slti	010	-		slt	111
		sltiu	011	-		situ	-
		xori	100	-		xor	011
		ori	110	-		or	001
		andi	111	-		and	000
		slli	001	0000000		sll	-
		srl	101	0000000		srl	101
		srai	101	0100000		sra	-

2.12 ALU 操作译码-多级译码



三、主要仪器设备

- 计算机（Intel Core i9-13980, 16GB 内存）系统
- NEXYS A7 开发板
- Xilinx VIVADO2024.2 及以上开发工具

四、操作方法与实验步骤

4.1 任务一：用硬件描述语言设计实现控制器

根据控制器的函数表达式，通过结构描述实现电路。

根据 ALU 操作译码函数表达式，通过结构描述实现电路。

编写模块代码如下所示：

```

23 module SCPU_ctrl(
24     input wire [4:0] OPcode, // inst[6:2]
25     input wire [2:0] Fun3, // inst[14:23]
26     input wire Fun7, // inst[30]
27     input wire MIO_ready, // CPU wait
28     output reg [1:0] ImmSel, // 立即数选择控制
29     output reg ALUSrc_B, // 源操作数2选择
30     output reg [1:0] MemtoReg, // 写回数据选择控制
31     output reg Jump, // jal
32     output reg Branch, // beq
33     output reg RegWrite, // 寄存器写使能
34     output reg MemRW, // 存储器读写使能
35     output reg [2:0] ALU_Control, // ALU控制
36     output reg CPU_MIO // not use
37 );
38
39     wire [3:0] Fun;
40     reg [1:0] ALUop;
41     //reg [10:0] CPU_ctrl_signals;
42
43     assign Fun = {Fun3,Fun7};
44
45     always @ * begin
46         CPU_MIO = MIO_ready;
47         case(OPcode)
48             5'b01100:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b000100010xx;end// R
49             5'b00000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b10110000000;end// lb lh lw ld lbu lhu lwu
50             5'b00100:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b10010001100;end// addi slli xori srli srai ori andi
51             5'b11001:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b10110000000;end// jalr
52             5'b01000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b0001000001;end// S:sh sh sw sd
53             5'b11000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b00000100110;end// SB:beq bne bit bge bitu bgeu // l
54             5'b01101:begin CPU_ctrl_signals = 9'h1ui
55             5'b11011:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b0101001xx1;end// U:S:jal
56             default:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b0000000000;end // ?
57         endcase
58     end
59
60     case(ALUop)
61         2'b00:ALU_Control = 3'b010;// addi计算地址
62         2'b01:ALU_Control = 3'b110;// sub比较条件
63         2'b10:
64             case(Fun)
65                 4'b0000:begin ALU_Control = 3'b010;end// add
66                 4'b0001:begin ALU_Control = 3'b110;end// sub
67                 4'b1110:begin ALU_Control = 3'b000;end// and
68                 4'b1100:begin ALU_Control = 3'b001;end// or
69                 4'b1000:begin ALU_Control = 3'b011;end// xor
70                 4'b0100:begin ALU_Control = 3'b111;end// slt
71                 4'b1010:begin ALU_Control = 3'b101;end// srl
72                 default:begin ALU_Control = 3'bxxx;end
73             endcase
74         2'b11:
75             case(Fun)
76                 3'b000:begin ALU_Control = 3'b010;end// addi
77                 3'b010:begin ALU_Control = 3'b111;end// slli
78                 3'b100:begin ALU_Control = 3'b011;end// xori
79                 3'b110:begin ALU_Control = 3'b001;end// ori
80                 3'b111:begin ALU_Control = 3'b000;end// andi
81                 3'b010:begin ALU_Control = 3'b101;end// srli
82                 default:begin ALU_Control = 3'bxxx;end
83             endcase
84         endcase
85     end
86 endmodule

```

4.2 任务二：设计控制器测试方案并完成测试

4.2.1 仿真测试

根据实验可见所提供仿真激励参考代码，编写控制器模块代码如下所示：

```

23 module SCPU_ctrl_testbench(
24
25     );
26     reg [4:0] OPcode;
27     reg [2:0] Fun3;
28     reg Fun7;
29     reg MIO_ready;
30     wire [1:0] ImmSel;
31     wire ALUSrc_B;
32     wire [1:0] MemtoReg;
33     wire Jump;
34     wire Branch;
35     wire RegWrite;
36     wire MemRW;
37     wire [2:0] ALU_Control;
38     wire CPU_MIO;

```

```

40      SCPU_ctrl SCPU_ctrl_uut(
41          .OPcode(OPcode), // inst[6:2]
42          .Fun3(Fun3), // inst[14:23]
43          .Fun7(Fun7), // inst[30]
44          .MIO_ready(MIO_ready), // CPU wait
45          .ImmSel(ImmSel), // 立即数选择控制
46          .ALUSrc_B(ALUSrc_B), // 源操作数2选择
47          .MemtoReg(MemtoReg), // 写回数据选择控制
48          .Jump(Jump), // jal
49          .Branch(Branch), // beq
50          .RegWrite(RegWrite), // 寄存器写使能
51          .MemRW(MemRW), // 存储器读写使能
52          .ALU_Control(ALU_Control), // ALU控制
53          .CPU_MIO(CPU_MIO) // not use
54      );
55
56      initial begin
57          // Initialize Inputs
58          OPcode = 5'b0;
59          Fun3 = 3'b0;
60          Fun7 = 1'b0;
61          MIO_ready = 1'b0;
62          #40;
63          // Wait 40 ns for global reset to finish. 以上是测试模板代码。
64          // Add stimulus here
65          //检查输出信号和关键信号输出是否满足真值表
66          OPcode = 5'b01100; //ALU指令, 检查 ALUop=2'b10; RegWrite=1
67          Fun3 = 3'b000; Fun7 = 1'b0;//add, 检查ALU_Control=3'b010
68          #20;
69          Fun3 = 3'b000; Fun7 = 1'b1;//sub, 检查ALU_Control=3'b110
70          #20;
71          Fun3 = 3'b111; Fun7 = 1'b0;//and, 检查ALU_Control=3'b000
72          #20;
73          Fun3 = 3'b110; Fun7 = 1'b0;//or, 检查ALU_Control=3'b001
74          #20;
75          Fun3 = 3'b010; Fun7 = 1'b0 ;//slt, 检查ALU_Control=3'b111
76
77          //控制器仿真激励代码参考
78          #20;
79          Fun3 = 3'b101; Fun7 = 1'b0; //srl, 检查ALU_Control=3'b101
80          #20;
81          Fun3 = 3'b100; Fun7 = 1'b0; //xor, 检查ALU_Control=3'b011
82          #20;
83          Fun3 = 3'b111; Fun7 = 1'b1; //间隔
84          #20;
85          #1;
86          OPcode = 5'b00000; //load指令, 检查 ALUop=2'b00,
87          #20; // ALUSrc_B=1, MemtoReg=1, RegWrite=1
88          OPcode = 5'b01000;
89          #20; //store指令, 检查ALUop=2'b00, MemRW=1, ALUSrc_B=1
90          OPcode = 5'b11000;//beq指令, 检查 ALUop=2'b01, Branch=1
91          #20;
92          OPcode = 5'b11011; //jump指令, 检查 Jump=1
93          #40;
94          OPcode = 5'b00100; Fun3 = 3'b000; //I指令, 检查 ALUop=2'b11; RegWrite=1
95          #20;
96          Fun3 = 3'b111;
97          #20;
98          Fun3 = 3'b110;
99          #20;
100         Fun3 = 3'b010;
101         #20;
102         Fun3 = 3'b101;
103         #20;
104         Fun3 = 3'b100;
105         #20;
106         OPcode = 5'h1f; //间隔
107         Fun3 = 3'b000; Fun7 = 1'b0;//间隔
108     end
109 endmodule

```

4.2.2 物理测试

• 控制器继承替换、CPU 集成替换

将 SCPU.v 模块中的 SCPU_ctrl 模块替换为本次实验所设计的 SCPU_ctrl.v 模块；之后将替换后的 SCPU.v 模块引入到实验 2 所配置的 CPU 测试环境中替换其中的 SCPU.v 模块。

运行实验课件提供的 ALU 指令测试程序以进行模块功能检验。

同时此处我们可以充分利用 VGA 显示进行 Debug 以及结果检验。具体做法为从 Datapath.v 模块中引出 32 个 32 位寄存器的数值，输出到 SCPU.v 模块、CSSTE 模块中从而传低到 VGA 模块中。

更新后的相关模块部分代码图下所示：

```
23 ⊖ module Register_32M32b(          84      assign x0 = register[0];
24     input clk,                      85      assign ra = register[1];
25     input rst,                      86      assign sp = register[2];
26     input ReqWrite,                87      assign gp = register[3];
27     input [4:0] Rs1_addr,           88      assign tp = register[4];
28     input [4:0] Rs2_addr,           89      assign t0 = register[5];
29     input [4:0] Wt_addr,            90      assign t1 = register[6];
30     input [31:0] Wt_data,           91      assign t2 = register[7];
31     output [31:0] Rs1_data,         92      assign s0 = register[8];
32     output [31:0] Rs2_data,         93      assign s1 = register[9];
33     output [31:0] ra,              94      assign a0 = register[10];
34     output [31:0] sp,              95      assign a1 = register[11];
35     output [31:0] gp,              96      assign a2 = register[12];
36     output [31:0] tp,              97      assign a3 = register[13];
37     output [31:0] t0,              98      assign a4 = register[14];
38     output [31:0] t1,              99      assign a5 = register[15];
39     output [31:0] t2,              100     assign a6 = register[16];
40     output [31:0] s0,              101     assign a7 = register[17];
41     output [31:0] s1,              102     assign s2 = register[18];
42     output wire [31:0] x0,          103     assign s3 = register[19];
43     output wire [31:0] a0,          104     assign s4 = register[20];
44     output wire [31:0] a1,          105     assign s5 = register[21];
45     output wire [31:0] a2,          106     assign s6 = register[22];
46     output wire [31:0] a3,          107     assign s7 = register[23];
47     output wire [31:0] a4,          108     assign s8 = register[24];
48     output wire [31:0] a5,          109     assign s9 = register[25];
49     output wire [31:0] a6,          110     assign s10 = register[26];
50     output wire [31:0] a7,          111     assign s11 = register[27];
51     output wire [31:0] s2,          112     assign t3 = register[28];
52     output wire [31:0] s3,          113     assign t4 = register[29];
53     output wire [31:0] s4,          114     assign t5 = register[30];
54     output wire [31:0] s5,          115     assign t6 = register[31];
55     output wire [31:0] s6,          116 ⊖ endmodule
56     output wire [31:0] s7,
57     output wire [31:0] s8,
58     output wire [31:0] s9,
59     output wire [31:0] s10,
60     output wire [31:0] s11,
61     output wire [31:0] t3,
62     output wire [31:0] t4,
63     output wire [31:0] t5,
64     output wire [31:0] t6
65 );
```

```

172     SCPU_U1(
173         .MIO_ready(1'b0), // ?
174         .clk(U8_Clk_CPU),
175         .rst(U9_rst),
176         .Data_in(U4_Cpu_data4bus),
177         .inst_in(U2_spo),
178         .CPU_MIO(U1_CPU_MIO),
179         .MemRW(U1_MemRW),
180         .Addr_out(U1_Addr_out),
181         .Data_out(U1_Data_out),
182         .PC_out(U1_PC_out),
183         .x0(U1_x0),
184         .ra(U1_ra),
185         .sp(U1_sp),
186         .gp(U1_gp),
187         .tp(U1_tp),
188         .t0(U1_t0),
189         .t1(U1_t1),
190         .t2(U1_t2),
191         .s0(U1_s0),
192         .s1(U1_s1),
193         .a0(U1_a0),
194         .a1(U1_a1),
195         .a2(U1_a2),
196         .a3(U1_a3),
197         .a4(U1_a4),
198         .a5(U1_a5),
199         .a6(U1_a6),
200         .a7(U1_a7),
201         .s2(U1_s2),
202         .s3(U1_s3),
203         .s4(U1_s4),
204         .s5(U1_s5),
205         .s6(U1_s6),
206         .s7(U1_s7),
207         .s8(U1_s8),
208         .s9(U1_s9),
209         .s10(U1_s10),
210         .s11(U1_s11),
211         .t3(U1_t3),
212         .t4(U1_t4),
213         .t5(U1_t5),
214         .t6(U1_t6)
215     );
285     VGA_U11(
286         .clk_25m(U8_clkdiv[1]),
287         .clk_100m(clk_100mhz),
288         .rst(U9_rst),
289         .pc(U1_PC_out),
290         .inst(U2_spo),
291         .alu_res(U1_Addr_out),
292         .mem_wen(U1_MemRW),
293         .dmem_o_data(U3_douta),
294         .dmem_i_data(U4_ram_data_in),
295         .dmem_addr(U1_Addr_out), ///
296         .x0(U1_x0),
297         .ra(U1_ra),
298         .sp(U1_sp),
299         .gp(U1_gp),
300         .tp(U1_tp),
301         .t0(U1_t0),
302         .t1(U1_t1),
303         .t2(U1_t2),
304         .s0(U1_s0),
305         .s1(U1_s1),
306         .a0(U1_a0),
307         .a1(U1_a1),
308         .a2(U1_a2),
309         .a3(U1_a3),
310         .a4(U1_a4),
311         .a5(U1_a5),
312         .a6(U1_a6),
313         .a7(U1_a7),
314         .s2(U1_s2),
315         .s3(U1_s3),
316         .s4(U1_s4),
317         .s5(U1_s5),
318         .s6(U1_s6),
319         .s7(U1_s7),
320         .s8(U1_s8),
321         .s9(U1_s9),
322         .s10(U1_s10),
323         .s11(U1_s11),
324         .t3(U1_t3),
325         .t4(U1_t4),
326         .t5(U1_t5),
327         .t6(U1_t6),
328         .hs(HSYNC),

```

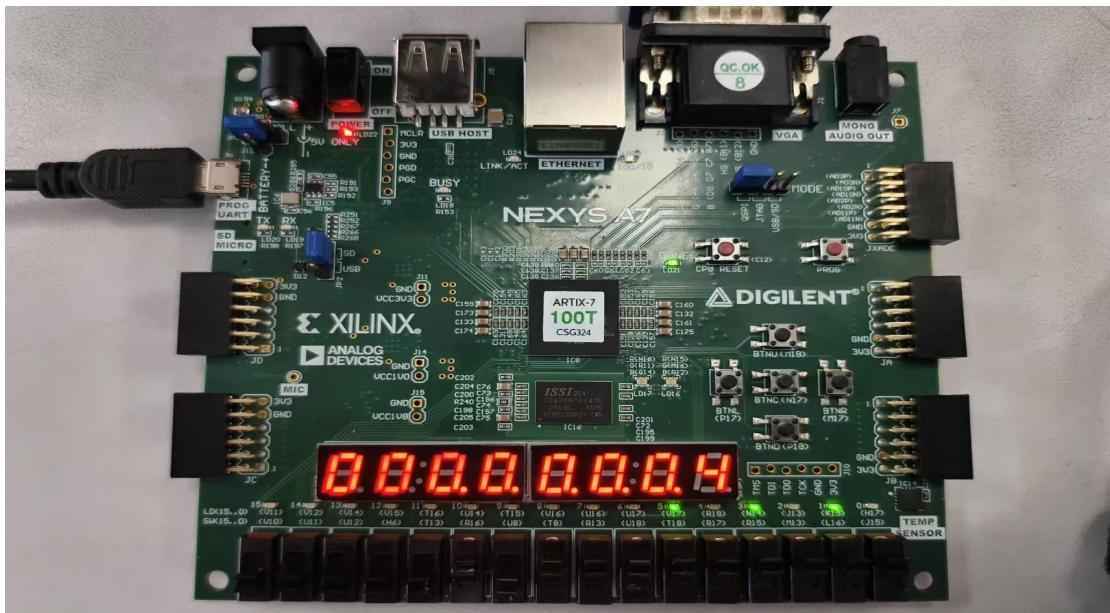
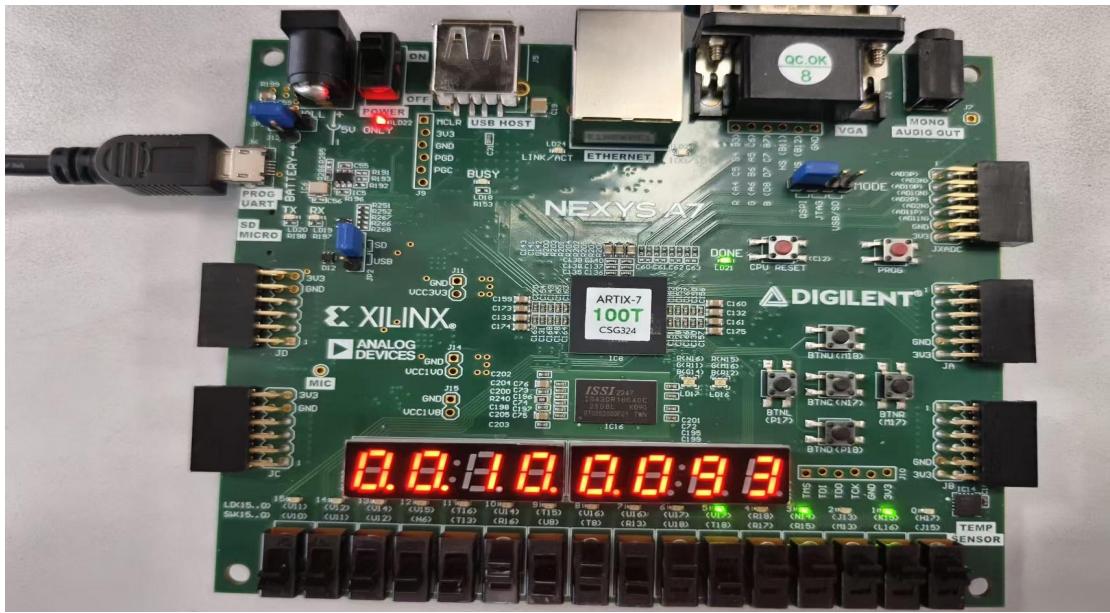
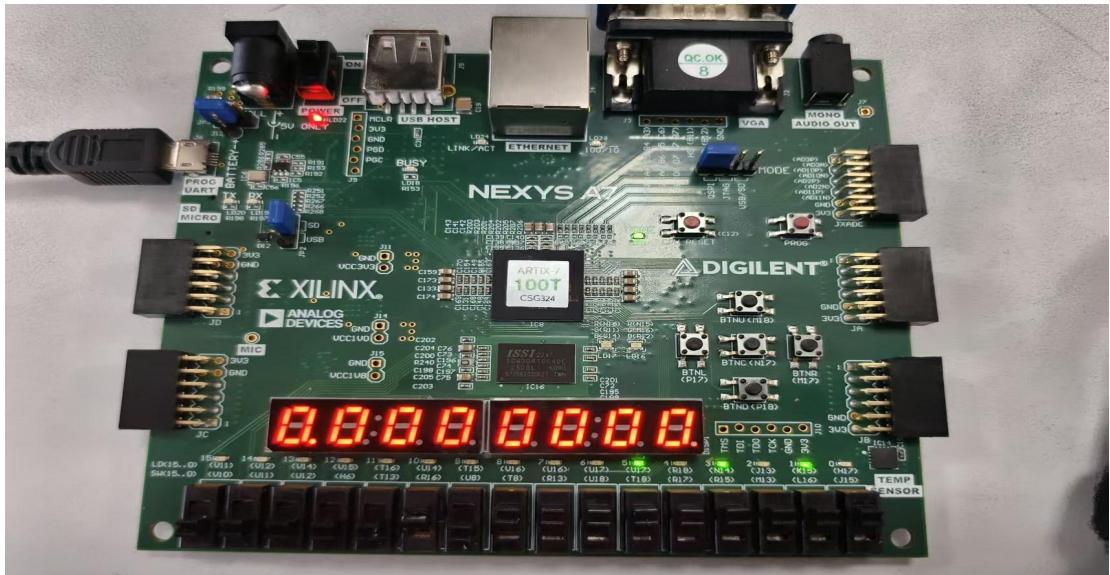
五、实验数据记录和处理

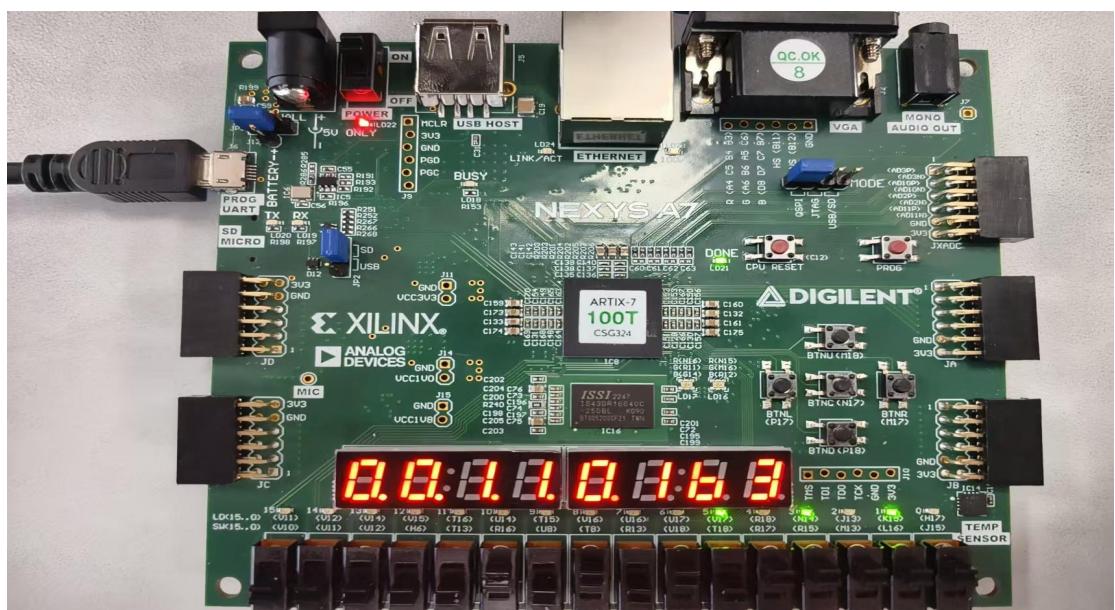
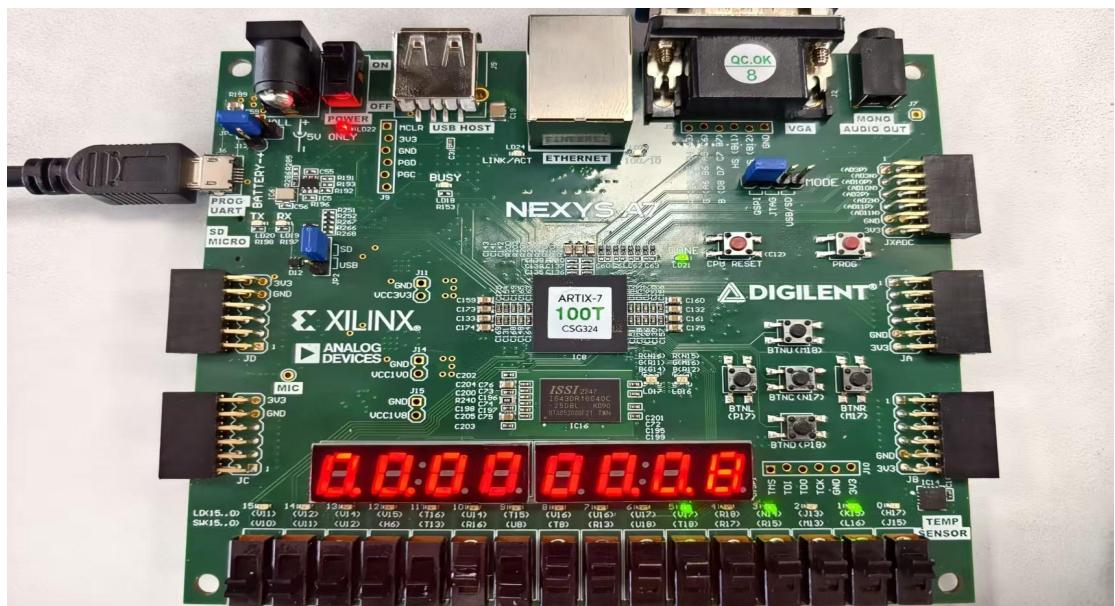
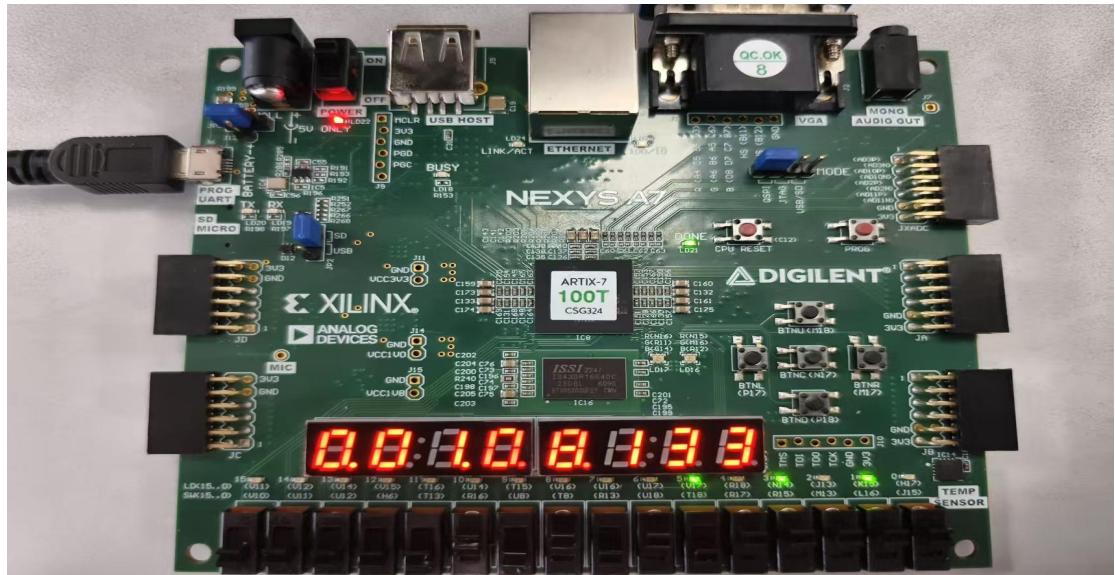
5.1 仿真测试



5.2 物理测试

5.2.1 七段数码管





5.2.2 VGA 显示输出

```
RV32I Single Cycle CPU
pc: 00000000 inst: 00100093
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000001 t6: 00000002

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000001 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000000 dmem_addr: 00000001

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

RV32I Single Cycle CPU
pc: 00000004 inst: 00108133
x0: 00000000 ra: 00000001 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000002 t6: 00000002

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000002 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000001 dmem_addr: 00000002

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

RV32I Single Cycle CPU
pc: 00000008 inst: 001101b3
x0: 00000000 ra: 00000001 sp: 00000002 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000003 t6: 00000002

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000003 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000001 dmem_addr: 00000003

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

```

RU32I Single Cycle CPU

pc: 0000000c    inst: 00315233

x0: 00000000    ra: 00000001    sp: 00000002    gp: 00000003    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000   s11: 00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000005

rs1: 00    rs1_val: 00000000
rs2: 00    rs2_val: 00000000
rd: 00    reg_i_data: 00000000    reg_wen: 0

is_imm: 0    is_auiipc: 0    is_lui: 0    imm: 00000000
a_val: 00000000    b_val: 00000000    alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000000    cmp_res: 0

is_branch: 0    is_jal: 0    is_jalr: 0
do_branch: 0    pc_branch: 00000000

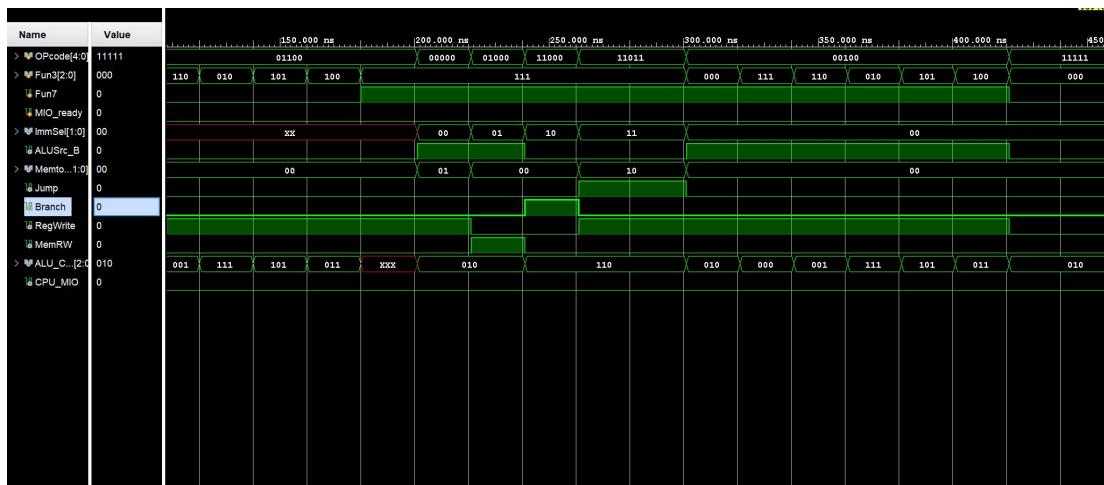
mem_wen: 0    mem_ren: 0
dmem_o_data: 00000000    dmem_i_data: 00000003    dmem_addr: 00000000

csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000    mcause: 00000000    mepc: 00000000    mtval: 00000000
mtvec: 00000000    mie: 00000000    mip: 00000000

```

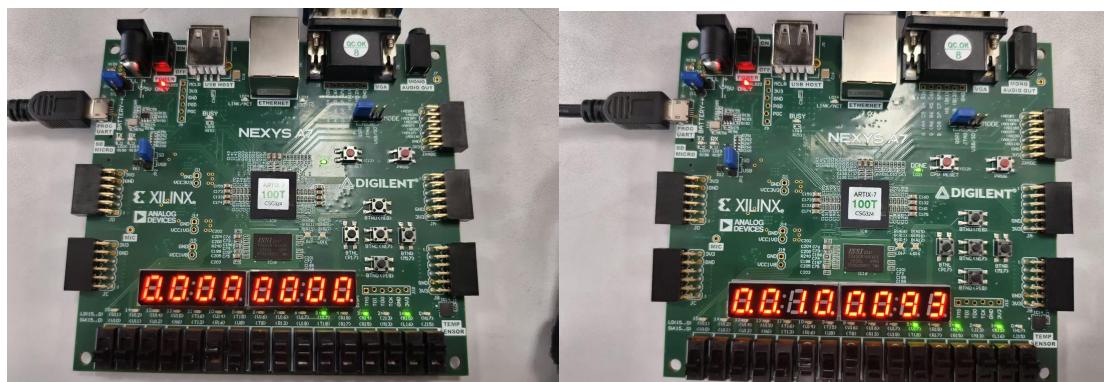
六、实验结果与分析

6.1 仿真测试



根据我们在实验原理部分参考、设计的控制信号真值表，模块仿真所得出的相关控制信号均是正确的。

6.2 物理测试



```

RV32I Single Cycle CPU

pc: 00000000 inst: 00100093

x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000001 t6: 00000002

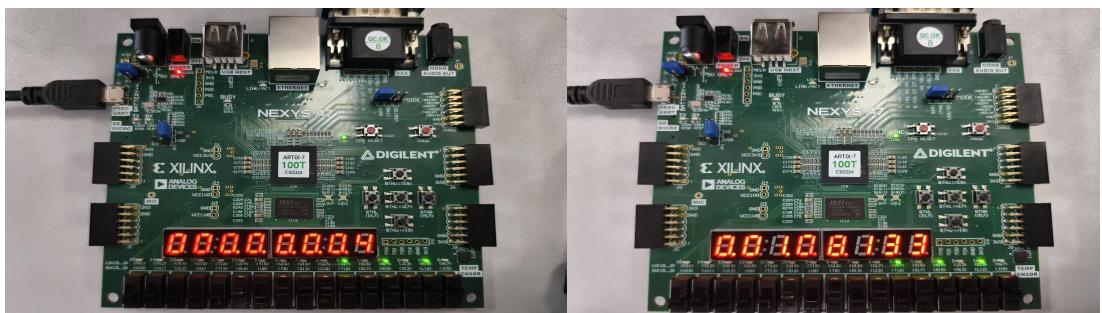
rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auiipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000001 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000000 dmem_addr: 00000001
csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```



```

RV32I Single Cycle CPU

pc: 00000004 inst: 00108133

x0: 00000000 ra: 00000001 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000002 t6: 00000002

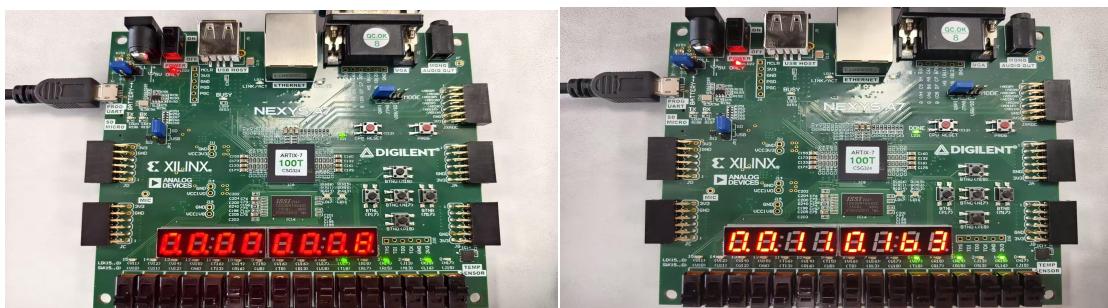
rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auiipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000002 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000001 dmem_addr: 00000002
csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```



```

RU32I Single Cycle CPU

pc: 00000008 inst: 001101b3

x0: 00000000 ra: 00000001 sp: 00000002 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000003 t6: 00000002

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000003 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000001 dmem_addr: 00000003

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

```

RU32I Single Cycle CPU

pc: 0000000c inst: 00315233

x0: 00000000 ra: 00000001 sp: 00000002 gp: 00000003 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000005

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000003 dmem_addr: 00000000

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

我们调节系统时钟为手动单步时钟，可以看到程序计数器 PC 和指令编码 inst 均符合我们设计的 Demo 程序；通过观察 VGA 上相关寄存器的数值，可以看到 0、1、2、3 号寄存器数值分别为 0、1、2、3，此结果与我们设计的 Demo 程序的与预期实验结果相符。

以上仿真测试和物理测试的结果说明我们设计的控制器模块能够正常地工作。

6.3 代码分析

6.3.1 SCPU_ctrl.v

```

23 module SCPU_ctrl(
24     input wire [4:0] OpCode, // inst[6:2]
25     input wire [2:0] Fun3, // inst[14:23]
26     input wire Fun7, // inst[30]
27     input wire MIO_ready, // CPU wait
28     output reg [1:0] ImmSel, // 立即数选择控制
29     output reg ALUSrc_B, // 源操作数2选择
30     output reg [1:0] MemtoReg, // 写回数据选择控制
31     output reg Jump, // jal
32     output reg Branch, // beq
33     output reg RegWrite, // 寄存器写使能
34     output reg MemRW, // 存储器读写使能
35     output reg [2:0] ALU_Control, // ALU控制
36     output reg CPU_MIO // not use
37 );

```

该部分代码展示了 SCPU_ctrl 模块的输入和输出端口。

```
39 :     wire [3:0] Fun;
40 :     reg [1:0] ALUop;
41 :     //reg [10:0] CPU_ctrl_signals;
42 :
43 :     assign Fun = {Fun3,Fun7};
```

该部分代码声明了模块内部使用到的中间变量。

```
46 :     always @ * begin
47 :         CPU_MIO = MIO_ready;
48 :         case (OPcode)
49 :             5'b01100:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b0001000010xx;end// R
50 :             5'b00000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b10110000000;end// lb lh lw ld lbu lhu lwu
51 :             5'b00100:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b100100001100;end// addi slli xorri srlri ori andi
52 :             5'b11001:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b101100000000;end// jalr
53 :             5'b01000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b100010000001;end// S:sb sh sw sd
54 :             5'b11000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b000001000110;end// S:S:beq bne hit bge hitu bgeu // .
55 :             //5'b01101:begin CPU_ctrl_signals = 9'b//lui
56 :             5'b10101:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b0101001xx11;end// U?J:jal
57 :             default:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel} = 11'b00000000000;end // ?
58 :         endcase
59 :
```

该部分代码根据指令的 Opcode 部分对部分相应的控制信号进行编码，并编码中间变量 ALUop 以方便后续的 ALU 控制信号的编码。

```
60 :         case(ALUop)
61 :             2'b00:ALU_Control = 3'b010;// addif算地址
62 :             2'b01:ALU_Control = 3'b110;// sub比较条件
63 :             2'b10:
64 :                 case(Fun)
65 :                     4'b0000:begin ALU_Control = 3'b010;end// add
66 :                     4'b0001:begin ALU_Control = 3'b110;end// sub
67 :                     4'b1110:begin ALU_Control = 3'b000;end// and
68 :                     4'b1100:begin ALU_Control = 3'b001;end// or
69 :                     4'b1000:begin ALU_Control = 3'b011;end// xor
70 :                     4'b0100:begin ALU_Control = 3'b111;end// slt
71 :                     4'b1010:begin ALU_Control = 3'b101;end// srl
72 :                     default:begin ALU_Control = 3'bxxx;end
73 :                 endcase
74 :             2'b11:
75 :                 case(Fun3)
76 :                     3'b000:begin ALU_Control = 3'b010;end// addi
77 :                     3'b011:begin ALU_Control = 3'b111;end// slli
78 :                     3'b100:begin ALU_Control = 3'b011;end// xorri
79 :                     3'b110:begin ALU_Control = 3'b001;end// ori
80 :                     3'b111:begin ALU_Control = 3'b000;end// andi
81 :                     3'b101:begin ALU_Control = 3'b101;end// srlri
82 :                     default:begin ALU_Control = 3'bxxx;end
83 :                 endcase
84 :             end
85 :         end
86 :     endmodule
```

该部分代码根据 ALUop 进一步对 ALU 控制信号进行编码。

6.3.2 SCPU_ctrl_testbench.v

```
23 : module SCPU_ctrl_testbench(
24 :
25 : );
26 :     reg [4:0] OPcode;
27 :     reg [2:0] Fun3;
28 :     reg Fun7;
29 :     reg MIO_ready;
30 :     wire [1:0] ImmSel;
31 :     wire ALUSrc_B;
32 :     wire [1:0] MemtoReg;
33 :     wire Jump;
34 :     wire Branch;
35 :     wire RegWrite;
36 :     wire MemRW;
37 :     wire [2:0] ALU_Control;
38 :     wire CPU_MIO;
```

该部分代码声明了仿真程序所需要使用到的中间变量。

```
40 :     SCPU_ctrl SCPU_ctrl_uut(
41 :         .OPcode(OPcode),// inst[6:2]
42 :         .Fun3(Fun3),// inst[14:23]
43 :         .Fun7(Fun7),// inst[30]
44 :         .MIO_ready(MIO_ready),// CPU wait
45 :         .ImmSel(ImmSel),// 立即数选择控制
46 :         .ALUSrc_B(ALUSrc_B),// 源操作数2选择
47 :         .MemtoReg(MemtoReg),// 写回数据选择控制
48 :         .Jump(Jump),// jal
49 :         .Branch(Branch),// beq
50 :         .RegWrite(RegWrite),// 寄存器写使能
51 :         .MemRW(MemRW),// 存储器读写使能
52 :         .ALU_Control(ALU_Control),// ALU控制
53 :         .CPU_MIO(CPU_MIO)// not use
54 :     );
```

该部分代码实例化引用 SCPU_ctrl.v 模块。

```
56     initial begin
57         // Initialize Inputs
58         OPcode = 5'b0;
59         Fun3 = 3'b0;
60         Fun7 = 1'b0;
61         MIO_ready = 1'b0;
62         #40;
63         // Wait 40 ns for global reset to finish. 以上是测试模板代码。
64         // Add stimulus here
65         //检查输出信号和关键信号输出是否满足真值表
66         OPcode = 5'b01100; //ALU指令, 检查 ALUop=2'b10; RegWrite=1
67         Fun3 = 3'b000; Fun7 = 1'b0;//add, 检查ALU_Control=3'b010
68         #20;
69         Fun3 = 3'b000; Fun7 = 1'b1;//sub, 检查ALU_Control=3'b110
70         #20;
71         Fun3 = 3'b111; Fun7 = 1'b0;//and, 检查ALU_Control=3'b000
72         #20;
73         Fun3 = 3'b110; Fun7 = 1'b0;//or, 检查ALU_Control=3'b001
74         #20;
75         Fun3 = 3'b010; Fun7 = 1'b0 ;//slt, 检查ALU_Control=3'b111
76         //控制器仿真激励代码参考
77         #20;
78         Fun3 = 3'b101; Fun7 = 1'b0; //srl, 检查ALU_Control=3'b101
79         #20;
80         Fun3 = 3'b100; Fun7 = 1'b0; //xor, 检查ALU_Control=3'b011
81         #20;
82         Fun3 = 3'b111; Fun7 = 1'b1; //间隔
83         #20;
84         #1;
```

该部分代码初始化相关变量，并针对 add、sub、and、or、slt、srl、xor 等 R 型指令进行激励检测。

```
86         OPcode = 5'b00000; //load指令, 检查 ALUop=2'b00,
87         #20; // ALUSrc_B=1, MemtoReg=1, RegWrite=1
```

该部分代码激励检测 load 指令。

```
88         OPcode = 5'b01000;
89         #20; //store指令, 检查ALUop=2'b00, MemRW=1, ALUSrc_B=1
```

该部分代码激励检测 stroe 指令。

```
90         OPcode = 5'b11000;//beq指令, 检查 ALUop=2'b01, Branch=1
91         #20;
```

该部分代码激励检测 beq 指令。

```
92         OPcode = 5'b11011; //jump指令, 检查 Jump=1
93         #40;
```

该部分代码激励检测 jump 指令。

```
94         OPcode = 5'b00100; Fun3 = 3'b000; //I指令, 检查 ALUop=2'b11; RegWrite=1
95         #20;
96         Fun3 = 3'b111;
97         #20;
98         Fun3 = 3'b110;
99         #20;
100        Fun3 = 3'b010;
101        #20;
102        Fun3 = 3'b101;
103        #20;
104        Fun3 = 3'b100;
105        #20;
106        OPcode = 5'h1f; //间隔
107        Fun3 = 3'b000; Fun7 = 1'b0;//间隔
108    end
109 endmodule
```

该部分代码激励检测 I 型指令。

6.3.3 VGA 输出信号引出代码

```
23 ⊖ module Register_32M32b(          84      assign x0 = register[0];
24     input clk,                         85      assign ra = register[1];
25     input rst,                         86      assign sp = register[2];
26     input RegWrite,                   87      assign gp = register[3];
27     input [4:0] Rs1_addr,             88      assign tp = register[4];
28     input [4:0] Rs2_addr,             89      assign t0 = register[5];
29     input [4:0] Wt_addr,              90      assign t1 = register[6];
30     input [31:0] Wt_data,            91      assign t2 = register[7];
31     output [31:0] Rs1_data,           92      assign s0 = register[8];
32     output [31:0] Rs2_data,           93      assign s1 = register[9];
33     output [31:0] ra,                94      assign a0 = register[10];
34     output [31:0] sp,                95      assign a1 = register[11];
35     output [31:0] gp,                96      assign a2 = register[12];
36     output [31:0] tp,                97      assign a3 = register[13];
37     output [31:0] t0,                98      assign a4 = register[14];
38     output [31:0] t1,                99      assign a5 = register[15];
39     output [31:0] t2,                100     assign a6 = register[16];
40     output [31:0] s0,                101     assign a7 = register[17];
41     output [31:0] s1,                102     assign s2 = register[18];
42     output wire [31:0] x0,           103     assign s3 = register[19];
43     output wire [31:0] a0,           104     assign s4 = register[20];
44     output wire [31:0] a1,           105     assign s5 = register[21];
45     output wire [31:0] a2,           106     assign s6 = register[22];
46     output wire [31:0] a3,           107     assign s7 = register[23];
47     output wire [31:0] a4,           108     assign s8 = register[24];
48     output wire [31:0] a5,           109     assign s9 = register[25];
49     output wire [31:0] a6,           110     assign s10 = register[26];
50     output wire [31:0] a7,           111     assign s11 = register[27];
51     output wire [31:0] s2,           112     assign t3 = register[28];
52     output wire [31:0] s3,           113     assign t4 = register[29];
53     output wire [31:0] s4,           114     assign t5 = register[30];
54     output wire [31:0] s5,           115     assign t6 = register[31];
55     output wire [31:0] s6,           116 ⊖ endmodule
56     output wire [31:0] s7,
57     output wire [31:0] s8,
58     output wire [31:0] s9,
59     output wire [31:0] s10,
60     output wire [31:0] s11,
61     output wire [31:0] t3,
62     output wire [31:0] t4,
63     output wire [31:0] t5,
64     output wire [31:0] t6
65 );
```

该部分代码具体展示了 32 个 32 位寄存器的数值传出方法，将寄存器变量赋值给相关的引脚输出。

```

172     SCPU_U1(
173         .MIO_ready(1'b0), // ?
174         .clk(U8_Clk_CPU),
175         .rst(U9_rst),
176         .Data_in(U4_Cpu_data4bus),
177         .inst_in(U2_spo),
178         .CPU_MIO(U1_CPU_MIO),
179         .MemRW(U1_MemRW),
180         .Addr_out(U1_Addr_out),
181         .Data_out(U1_Data_out),
182         .PC_out(U1_PC_out),
183         .x0(U1_x0),
184         .ra(U1_ra),
185         .sp(U1_sp),
186         .gp(U1_gp),
187         .tp(U1_tp),
188         .t0(U1_t0),
189         .t1(U1_t1),
190         .t2(U1_t2),
191         .s0(U1_s0),
192         .s1(U1_s1),
193         .a0(U1_a0),
194         .a1(U1_a1),
195         .a2(U1_a2),
196         .a3(U1_a3),
197         .a4(U1_a4),
198         .a5(U1_a5),
199         .a6(U1_a6),
200         .a7(U1_a7),
201         .s2(U1_s2),
202         .s3(U1_s3),
203         .s4(U1_s4),
204         .s5(U1_s5),
205         .s6(U1_s6),
206         .s7(U1_s7),
207         .s8(U1_s8),
208         .s9(U1_s9),
209         .s10(U1_s10),
210         .s11(U1_s11),
211         .t3(U1_t3),
212         .t4(U1_t4),
213         .t5(U1_t5),
214         .t6(U1_t6)
215     );
285     VGA_U11(
286         .clk_25m(U8_clkdiv[1]),
287         .clk_100m(clk_100mhz),
288         .rst(U9_rst),
289         .pc(U1_PC_out),
290         .inst(U2_spo),
291         .alu_res(U1_Addr_out),
292         .mem_wen(U1_MemRW),
293         .dmem_o_data(U3_douta),
294         .dmem_i_data(U4_ram_data_in),
295         .dmem_addr(U1_Addr_out), /**
296         .x0(U1_x0),
297         .ra(U1_ra),
298         .sp(U1_sp),
299         .gp(U1_gp),
300         .tp(U1_tp),
301         .t0(U1_t0),
302         .t1(U1_t1),
303         .t2(U1_t2),
304         .s0(U1_s0),
305         .s1(U1_s1),
306         .a0(U1_a0),
307         .a1(U1_a1),
308         .a2(U1_a2),
309         .a3(U1_a3),
310         .a4(U1_a4),
311         .a5(U1_a5),
312         .a6(U1_a6),
313         .a7(U1_a7),
314         .s2(U1_s2),
315         .s3(U1_s3),
316         .s4(U1_s4),
317         .s5(U1_s5),
318         .s6(U1_s6),
319         .s7(U1_s7),
320         .s8(U1_s8),
321         .s9(U1_s9),
322         .s10(U1_s10),
323         .s11(U1_s11),
324         .t3(U1_t3),
325         .t4(U1_t4),
326         .t5(U1_t5),
327         .t6(U1_t6),
328         .hs(HSYNC),

```

该部分代码展示了 CSSTE 顶部模块，从 SCPU 引出相应的寄存器变量值传输给 VGA 模块的输入引脚，从而实现将寄存器数值显示在 VGA 上的效果。

七、讨论、心得

通过本次实验，我们完成了 CPU 的控制器模块设计来控制我们的数据通路模块，算是初步完成了我们自己的处理器模块设计。

在实验过程中，我遇到了控制信号位数弄错的问题，导致仿真结果十分奇怪；也曾因引出 VGA 引脚涉及模块层数过多而导致频频发生语法错误。面对这样的错误，我们一定要冷静，不能胡乱分析，而是要仔细阅读相关报错，锁定错误出现的范围和位置，进行更加细致的检验。

本次实验极大地帮助我们巩固了时序逻辑的设计方法，包括 case 语法的使用，以及对于时序逻辑执行逻辑的理解，体现出了实验对于理论的巩固与应用作用！

Experiment3-CPU 设计之指令扩展

一、实验目的和要求

1.1 实验目的

- 运用寄存器传输控制技术
- 掌握 CPU 的核心：指令执行过程与控制流关系
- 设计数据通路和控制器
- 设计测试程序

1.2 实验目标及任务

- 目标

熟悉 RISC-V RV32I 的指令特点，了解控制器和数据通路的原理，扩展实验 lab4-2 CPU 指令集，设计并测试 CPU

- 任务一

重新设计数据通路和控制器，在 lab4-2 的基础上完成
兼容 lab4-1、lab4-2 的数据通路和控制器
替换 lab4-1、lab4-2 的数据通路控制器核
扩展不少于下列指令

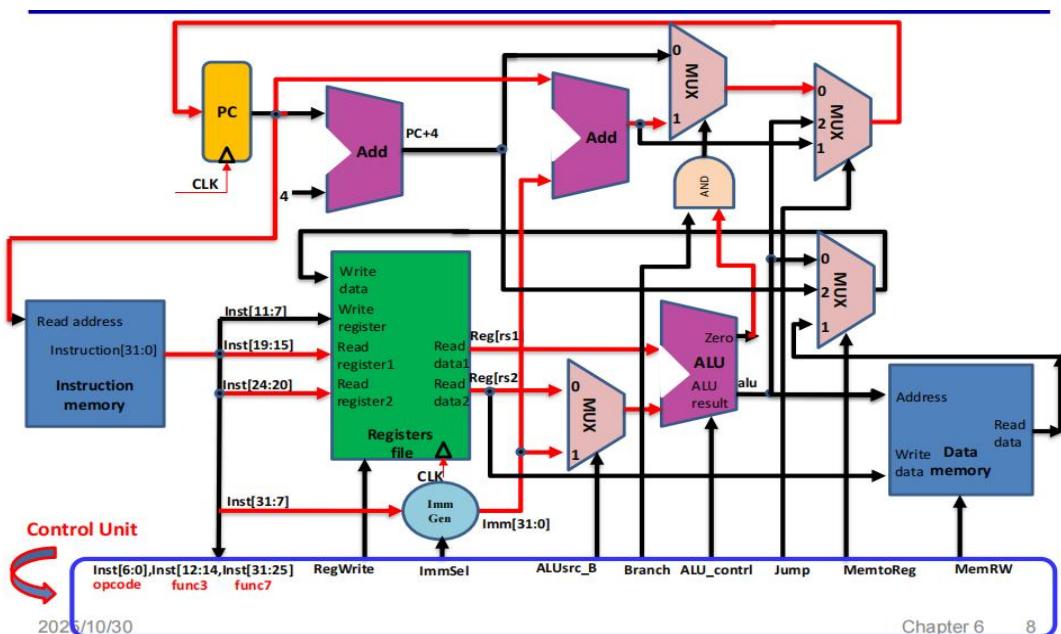
R-Type: add, sub, and, or, xor, slt, **sltu,srl, sra, sll**;
I-Type: addi, andi, ori, xori, slti, **sltiu,srli,srai,slli,lw,jalr**;
S-Type: sw;
B-Type: beq, **bne**;
J-Type: Jal;
U-Type: **lui**;

- 任务二

设计指令集测试方案并完成测试

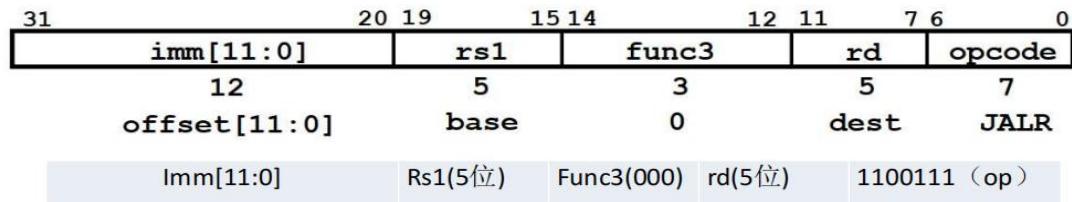
二、实验内容和原理

2.1 数据通路



2.2 JALR

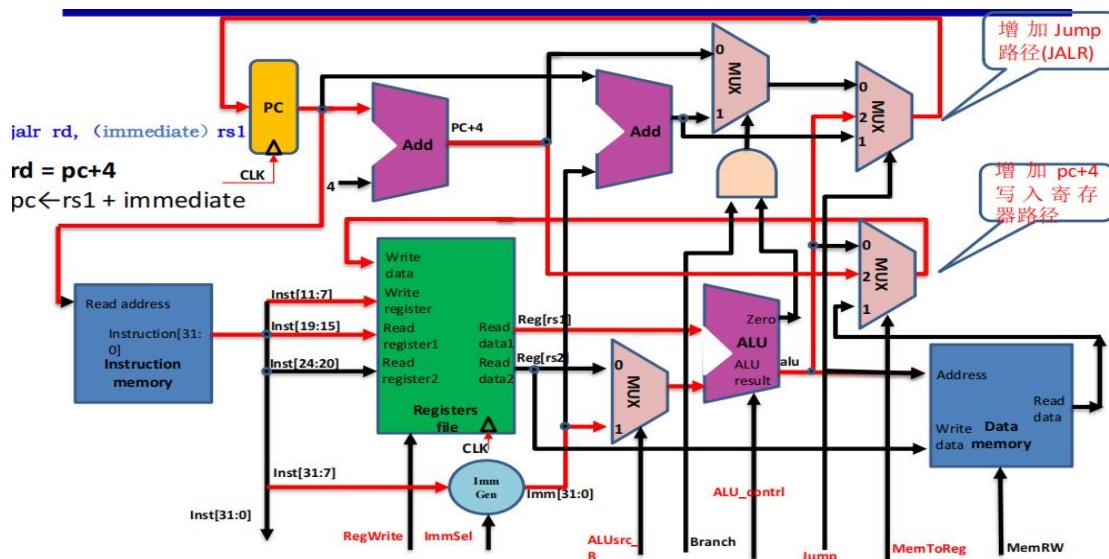
2.2.1 Format



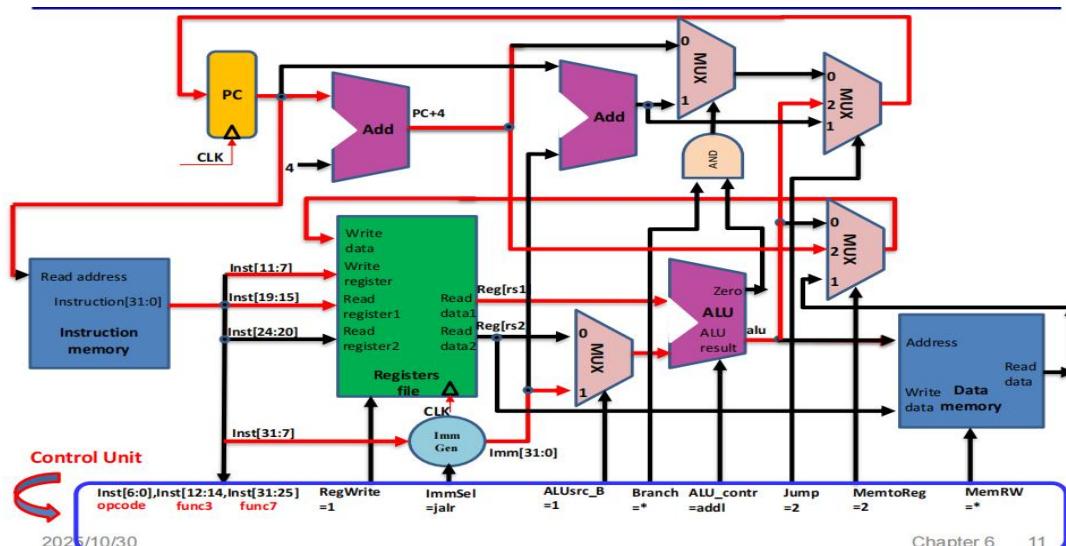
jalr rd, (immediate) rs1 无条件跳转-链接（寄存器地址）

功能: $rd = pc + 4$, $pc \leftarrow rs1 + immediate$

2.2.2 Datapath with JALR

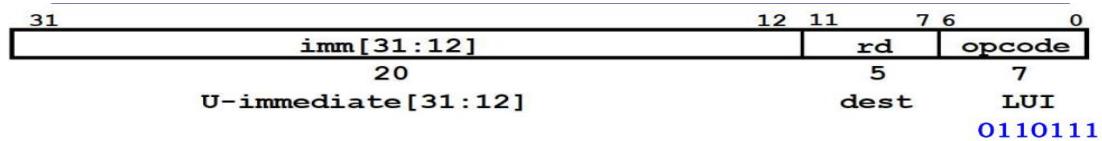


2.2.3 Control unit



2.3 U-Format Instruction

2.3.1 Format



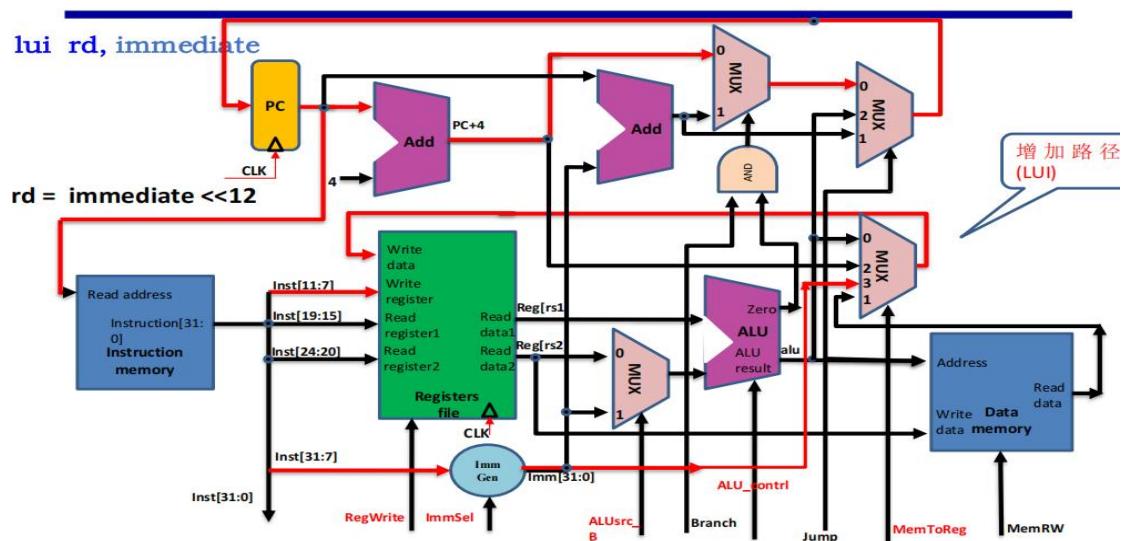
lui rd, immediate

功能: $rd = \text{immediate} \ll 12$

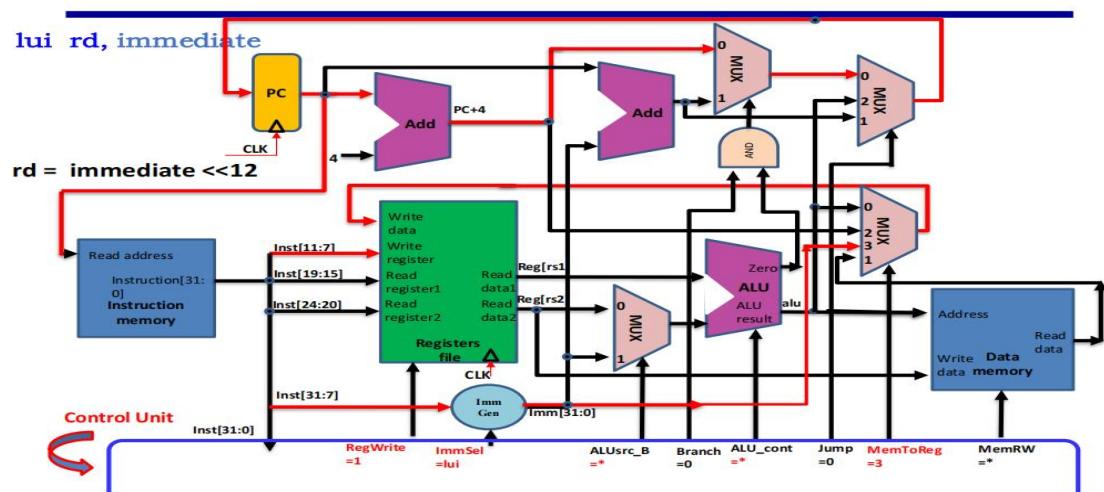
Eg: lui x5,0x12345

X5 = 0x12345000 取左移12位的20位立即数

2.3.2 Datapath with U-format instructions

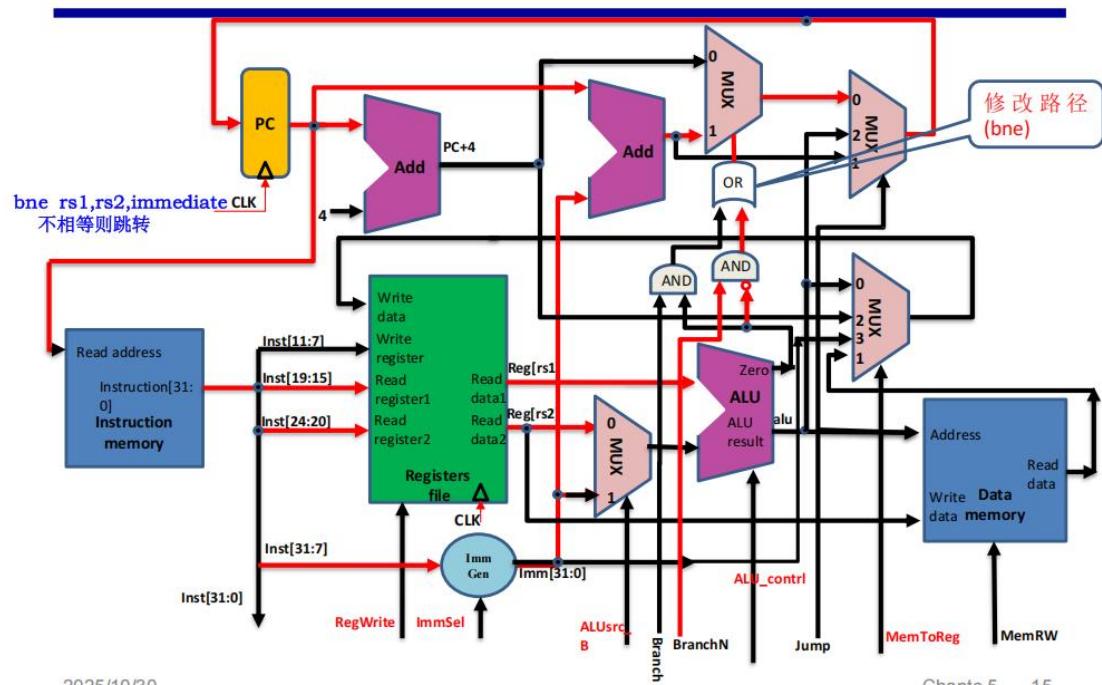


2.3.3 Control unit

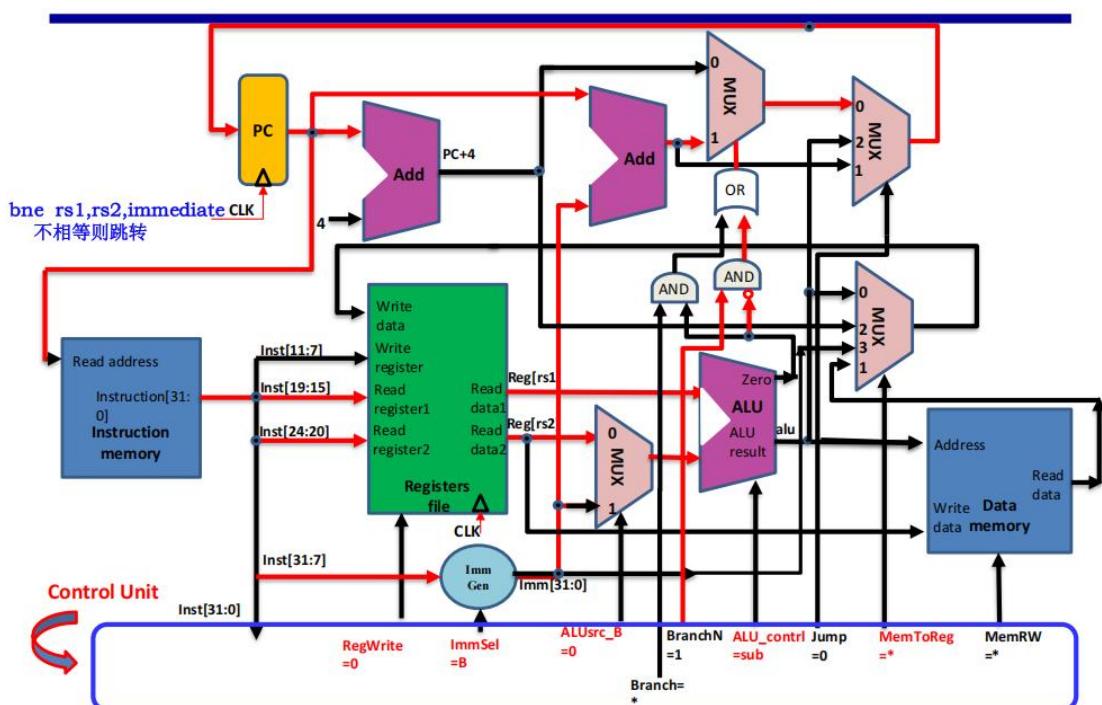


2.4 bne

2.4.1 Datapath with bne



2.4.2 Control unit



2.5 控制信号

2.5.1 定义

信号	源数目	功能定义	赋值0时动作	赋值1时动作	赋值2时动作	赋值3时动作
ALUSrc_B	2	ALU端口B输入选择	选择源操作数寄存器2数据	选择32位立即数(符号扩展后)	-	-
MemToReg	4	寄存器写入数据选择	选择ALU输出	选择存储器数据	选择PC+4	选择imm
Branch	2	Beq指令目标地址选择	选择PC+4地址	选择转移目的地址PC+imm (zero=1)	-	-
BranchN	2	Bne指令目标地址选择	选择PC+4地址	选择转移目的地址PC+imm (zero=0)	-	-
Jump	3	J指令目标地址选择	由Branch决定输出	选择跳转目标地址PC+imm (JAL)	选择跳转目标地址ALU输出(JALR:rs1+imm)	-
RegWrite	-	寄存器写控制	禁止寄存器写	使能寄存器写	-	-
MemRW	-	存储器读写控制	存储器读使能,存储器写禁止	存储器写使能,存储器读禁止	-	-
ALU_Control	0000 - 1111	4位ALU操作控制	参考表ALU_Control			
ImmSel	000-111	3位立即数组合控制	参考表ImmSel			

其中红色部分为兼容扩展指令所需增加的控制信号部分。

2.5.2 真值表

Inst[31:0]	Branch	Branch_N	Jump	ImmSel	ALUSrc_B	ALU_Control	MemRW	RegWrite	MemtoReg
add	0	0	0	*	Reg (0)	Add	Read	1	ALU(0)
sub	0	0	0	*	Reg (0)	Sub	Read	1	ALU(0)
(R-R Op)	0	0	0	*	Reg (0)	(Op)	Read	1	ALU(0)
addi	0	0	0	I	Imm (1)	Add	Read	1	ALU(0)
lw	0	0	0	I	Imm (1)	Add	Read	1	Mem(1)
sw	0	0	0	S	Imm (1)	Add	Write	0	*
beq	0	0	0	B	Reg (0)	sub	*	0	*
beq	1	*	0	B	Reg (0)	sub	*	0	*
bne	0	0	0	B	Reg (0)	sub	*	0	*
bne	*	1	0	B	Reg (0)	sub	*	0	*
jalr	*	*	2	I	Imm (1)	Add	*	1	PC+4(2)
jal	*	*	1	J	Imm (1)	*	*	1	PC+4(2)
lui	0	0	0	U	*	*	*	1	Imm(3)

2.6 ImmSel 信号

Instruction type	Instruction opcode[6:0]	Instruction operation	(sign-extend)immediate	Imm Sel
I-type	0000011	Lw;lbu;lh; lb;ihu	(sign-extend) instr[31:20]	001
	0010011	Addi;slti;slti u;xori;ori;a ndi;slli;srai		
	1100111	jalr		
S-type	0100011	Sw;sb;sh	(sign-extend) instr[31:25],[11:7]	010
B-type	1100011	Beq;bne;blt ;bge;bltu;b geu	(sign-extend) instr[31],[7],[30:25],[11:8], 1'b0	011
J-type	1101111	jal	(sign-extend) instr[31],[19:12],[20],[30:21],1 'b0	100
U-type <small>110100/30</small>	0010111	uiop	instr[31:12],12'h000	000
	0110111	lui		

2.7 RSICV--decode

Instruction opcode	op	Instruction operation	Funct 3	Funct7	ALUop	Desired ALU action	ALUControl
R-type	0110011	add	000	0000000	10	add	0010
		sub	000	0100000		sub	0110
		sll	001	0000000		sll	1110
		slt	010	0000000		slt	0111
		sltu	011	0000000		sltu	1001
		xor	100	0000000		xor	1100
		srl	101	0000000		srl	1101
		sra	101	0100000		sra	1111
		or	110	0000000		or	0001
		and	111	0000000		and	0000

Instruction opcode	op	Instruction operation	Funct 3	Funct7	ALUop	Desired ALU action	ALUControl
S-Type	0100011	sb	000	-	00	add	0010
		sh	001	-		add	0010
		sw	010	-		add	0010

Instruction opcode	op	Instruction operation	Funct 3	Funct7	ALUop	Desired ALU action	ALUControl
B-Type	1100011	Beq	000	-	01	sub	0110
		Bne	001	-		sub	0110
		Blt	100	-		slt	0111
		Bge	101	-		slt	0111
		Bltu	110	-		situ	1001
		Bltu	111	-		sra	1001

Instruction opcode	op	Instruction operation	Funct 3	Funct7	ALUop	Desired ALU action	ALUControl
U-Type	0110111	lui	-	-		-	-
	0010111	auipc	-	-		-	-
Instruction opcode	op	Instruction operation	Funct 3	Funct7	ALUop	Desired ALU action	ALUControl
J-Type	1101111	jal	-	-	-	-	-

Instruction opcode	op	Instruction operation	Funct 3	Funct7	ALUop	Desired ALU action	ALUControl
I-Type	0000011	Lb	-	-	00	add	0010
		Lh	-	-		add	0010
		Lw	-	-		add	0010
		Lbu	-	-		add	0010
		lh	-	-		add	0010
		jalr			00	add	0010

Instruction opcode	op	Instruction operation	Funct 3	Funct7	ALUop	Desired ALU action	ALUControl
I-Type	0010011	addi	000	-	11	add	0010
		slti	010	-		slt	0111
		sltiu	011	-		situ	1001
		xori	100	-		xor	1100
		ori	110	-		or	0001
		andi	111	-		and	0000
		slli	001	0000000		sll	1110
		srai	101	0100000		srl	1101
						sra	1111

三、主要仪器设备

- 计算机（Intel Core i9-13980, 16GB 内存）系统
- NEXYS A7 开发板
- Xilinx VIVADO2024.2 及以上开发工具

四、操作方法与实验步骤

4.1 设计指令扩展后的 ALU

指令扩展要求实现 sltu、sra、sltiu 等指令，其中需要我们实现对于符号数与无符号数运算的区分以及新增的算数右移运算，因此我们需要对 ALU 的功能进行扩展，那么同时我们也需要增加 ALU 控制信号的位数从而实现更多的功能。

新设计的 ALU 模块代码如下所示：

```
23 module MyALU(
24     input wire [31:0] A,
25     input wire [31:0] B,
26     input wire [3:0] operation,
27     output reg [31:0] res,
28     output wire zero
29 );
30
31 always @ (*) begin
32     case(operation)
33         4'b0000:begin res = A & B;end//and:and
34         4'b0001:begin res = A | B;end//or:or
35         4'b0010:begin res = A + B;end//add:add
36         4'b0110:begin res = A - B;end//sub:sub
37         4'b0111:begin res = ($signed(A) < $signed(B));end//set on if less than signed:slt
38         4'b1001:begin res = (A < B);end// set on if less than unsigned:sltu
39         4'b1100:begin res = A ^ B;end// xor:xor
40         4'b1101:begin res = A >> B;end// shift right logical:srl
41         4'b1110:begin res = A << B;end// shift left logical:sll
42         4'b1111:begin res = $signed(A) >>> B;end//shift right arithmetic:sra
43     endcase
44 end
45
46 assign zero = (res == 32'b0);
47 endmodule
```

4.2 设计指令扩展后的 ImmGen

由于新增了 lui 指令，因此我们需要生成相应的 lui 所需的立即数，即高 20 位为有效数字，低 12 位为 0 的立即数。

新设计的 ImmGen 模块代码如下所示；

```
23 module ImmGen_new // 此模块内不转换为半字
24     input wire [2:0] ImmSel, // 立即数操作控制
25     input wire [31:0] inst_field, // 指令数据域[31: 7]
26     output reg [31:0] Imm_out // 立即数输出
27 );
28
29 always @ * begin
30     case(ImmSel)
31         3'b001:begin Imm_out = ({20{inst_field[31]}},inst_field[31:20]); end// Lw,Addi,slti,sltiu,xori,ori,andi,slli,srai,jalr
32         3'b010:begin Imm_out = ({20{inst_field[31]}},inst_field[31:25],inst_field[11:7]); end// sw(s)
33         3'b011:begin Imm_out = ({19{inst_field[31]}},inst_field[31],inst_field[7],inst_field[30:25],inst_field[11:8],1'b0); end// beq,bne
34         3'b100:begin Imm_out = ({11{inst_field[31]}},inst_field[31],inst_field[19:12],inst_field[20],inst_field[30:21],1'b0); end// jal(j)
35         3'b000:begin Imm_out = (inst_field[31:12],12'b000000000000); end // lui
36     endcase
37 end
38 endmodule
```

4.3 设计指令扩展后的 SCPU_ctrl

根据新设计的 ALU 模块没我们发现 ALU 控制信号新增的一位，同时由于

bne 指令的加入，我们需要新加讨论 BranchN 信号以对 bne 指令进行处理，因此我们需要新设计 SCPU_ctrl 模块来保证相关控制信号的正常产生。

新设计的 SCPU_ctrl 模块如下所示：

```
22 module SCPU_ctrl_more(
23     input wire [4:0] OPCODE, // inst[6:2]
24     input wire [2:0] Fun3, // inst[14:23]
25     input wire Fun7, // inst[30]
26     input wire MIO_ready, // CPU wait
27     output reg [2:0] ImmSel, // 立即数选择控制 // need updating
28     output reg ALUSrc_B, // 源操作数2选择
29     output reg [1:0] MemtoReg, // 写回数据选择控制
30     output reg [1:0] Jump, // jal // need updating
31     output reg Branch, // beq
32     output reg BranchN, // need updating
33     output reg RegWrite, // 寄存器写使能
34     output reg MemRW, // 存储器读写使能
35     output reg [3:0] ALU_Control, // ALU控制
36     output reg CPU_MIO // not use
37 );
38
39
40
41
42
43 assign Fun = {Fun3,Fun7};
44
45
46 always @ * begin
47     CPU_MIO = MIO_ready;
48     case(OPCODE)
49         5'b01100:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'b0001000010xxx0;end// 
50         5'b00000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'b10110000000010;end// 
51         5'b01000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'b10010000110010;end// 
52         5'b11001:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'b1101xx1000001x;end// 
53         5'b01000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'b1xx01000000100;end// 
54         5'b01000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 13'b0xx0010001011;end// SB:beq b
55         5'b01101:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'b1111x000xx0000;end// 
56         5'b11011:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'b1101xx10xx100x;end// 
57         default:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'b00000000000000;end // 
58     endcase
59
60     case(ALUop)
61     2'b00:ALU_Control = 4'b0010;// S-type:addi/算地址
62     2'b01:
63         case(Fun3)
64             3'b000:begin BranchN = 1'b0;ALU_Control = 4'b0110;end// B-type:beq sub比较条件
65             3'b001:begin BranchN = 1'b1;ALU_Control = 4'b0110;end// B-type:bne sub比较条件
66         endcase
67     2'b10:
68         case(Fun)
69             4'b0000:begin ALU_Control = 4'b0010;end// add
70             4'b0001:begin ALU_Control = 4'b0110;end// sub
71             4'b0010:begin ALU_Control = 4'b1110;end// sll
72             4'b1110:begin ALU_Control = 4'b0000;end// and
73             4'b1100:begin ALU_Control = 4'b0001;end// or
74             4'b1000:begin ALU_Control = 4'b1100;end// xor
75             4'b0100:begin ALU_Control = 4'b0111;end//slt
76             4'b0110:begin ALU_Control = 4'b1001;end// sltu
77             4'b1010:begin ALU_Control = 4'b1101;end// srl
78             4'b1011:begin ALU_Control = 4'b1111;end//sra
79         default:begin ALU_Control = 3'bxxx;end
80     endcase
81
82     2'b11:
83         case(Fun3)
84             3'b000:begin ALU_Control = 4'b0010;end// addi
85             3'b010:begin ALU_Control = 4'b0111;end// slti
86             3'b011:begin ALU_Control = 4'b1001;end//sltiu
87             3'b100:begin ALU_Control = 4'b1100;end// xori
88             3'b110:begin ALU_Control = 4'b0001;end// ori
89             3'b111:begin ALU_Control = 4'b0000;end// andi
90             3'b001:begin ALU_Control = 4'b1110;end//slli
91         endcase
92         case(Fun7)
93             1'b0:begin ALU_Control = 4'b1101;end// srli
94             1'b1:begin ALU_Control = 4'b1111;end//end
95         endcase
96     endcase
97 endcase
98 end
99
100 endmodule
```

4.4 设计指令扩展后的 Datapath

因为相关模块的控制信号发生了改变，那么自然我们也要对相应的数据通路设计进行相应的修改。

修改后的 Datapath 模块代码如下所示：

```
23 ⊜ module DataPath_more(
24     input wire Branch,
25     input wire BranchN,
26     input wire [1:0] Jump,
27     input wire [31:0] Data_in,
28     input wire [1:0] MemtoReg,
29     input wire ALUSrc_B,
30     input wire [2:0] ImmSel,
31     input wire [31:0] inst_field,
32     input wire [3:0] ALU_Control, // ALU_operation
33     input wire clk,
34     input wire rst,
35     input wire RegWrite,
36     output wire [31:0] ALU_out,
37     output wire [31:0] Data_out,
38     output wire [31:0] PC_out,
39
40     output wire [31:0] x0,
41     output wire [31:0] ra,
42     output wire [31:0] sp,
43     output wire [31:0] gp,
44     output wire [31:0] tp,
45     output wire [31:0] t0,
46     output wire [31:0] t1,
47     output wire [31:0] t2,
48     output wire [31:0] s0,
49     output wire [31:0] s1,
50     output wire [31:0] a0,
51     output wire [31:0] a1,
52     output wire [31:0] a2,
53     output wire [31:0] a3,
54     output wire [31:0] a4,
55     output wire [31:0] a5,
56     output wire [31:0] a6,
57     output wire [31:0] a7,
58     output wire [31:0] s2,
59     output wire [31:0] s3,
60     output wire [31:0] s4,
61     output wire [31:0] s5,
62     output wire [31:0] s6,
63     output wire [31:0] s7,
64     output wire [31:0] s8,
65     output wire [31:0] s9,
66     output wire [31:0] s10,
67     output wire [31:0] s11,
68     output wire [31:0] t3,
69     output wire [31:0] t4,
70     output wire [31:0] t5,
71     output wire [31:0] t6,
72
73     output wire [4:0] rs1,
74     output wire [4:0] rs2,
75     output wire [31:0] rs1_val,
76     output wire [31:0] rs2_val,
77     output wire [4:0] rd,
78     output wire [31:0] reg_i_data,
79     output wire [31:0] imm
80 );
```

```

    ImmGen_new ImmGen_0(
    .ImmSel(ImmSel),
    .inst_field(inst_field),
    .Imm_out(ImmGen_0_Imm_out)
);
100
101 assign imm = ImmGen_0_Imm_out;
103 Add_32 Add_32_0(
104     .b(PC_Q),
105     .a(32'<0004>),
106     .c(Add_32_0_c)
107 );
108
109 Add_32 Add_32_1(
110     .a(PC_Q),
111     .b(ImmGen_0_Imm_out),
112     .c(Add_32_1_c)
113 );
114
115 MUX2T1_32 MUX2T1_32_1(
116     .I0(Add_32_0_c),
117     .I1(Add_32_1_c),
118     .sel((Branch & ALU_0_zero) | (BranchN & ~ALU_0_zero)),
119     .O(MUX2T1_32_1_o)
120 );
121
122 MUX4T1_32 MUX4T1_32_0(
123     .S(MemtoReg),
124     .I0(ALU_0_res),
125     .I1(Data_in),
126     .I2(Add_32_0_c),
127     .I3(ImmGen_0_Imm_out),
128     .O(MUX4T1_32_0_o)
129 );
130
131 MUX2T1_32 MUX2T1_32_0(
132     .I0(Regs_0_Rs2_data),
133     .I1(ImmGen_0_Imm_out),
134     .sel(ALUSrc_B),
135     .O(MUX2T1_32_0_o)
136 );
137
138 assign reg_i_data = MUX4T1_32_0_o;
139
140 Register_32M32b Regs_0(
141     .clk(~clk), // 取反?
142     .rst(rst),
143     .RegWrite(RegWrite),
144     .Rs1_addr(inst_field[19:15]),
145     .Rs2_addr(inst_field[24:20]),
146     .Wt_addr(inst_field[11:7]),
147     .Wt_data(MUX4T1_32_0_o),
148     .Rs1_data(Regs_0_Rs1_data),
149     .Rs2_data(Regs_0_Rs2_data),

```

```

150      .x0(x0),
151      .ra(ra),
152      .sp(sp),
153      .gp(gp),
154      .tp(tp),
155      .t0(t0),
156      .t1(t1),
157      .t2(t2),
158      .s0(s0),
159      .s1(s1),
160      .a0(a0),
161      .a1(a1),
162      .a2(a2),
163      .a3(a3),
164      .a4(a4),
165      .a5(a5),
166      .a6(a6),
167      .a7(a7),
168      .s2(s2),
169      .s3(s3),
170      .s4(s4),
171      .s5(s5),
172      .s6(s6),
173      .s7(s7),
174      .s8(s8),
175      .s9(s9),
176      .s10(s10),
177      .s11(s11),
178      .t3(t3),
179      .t4(t4),
180      .t5(t5),
181      .t6(t6),
182      .rs1_val(rs1_val),
183      .rs2_val(rs2_val) // o
184  );

186      assign rs1 = inst_field[19:15];
187      assign rs2 = inst_field[24:20];
188      assign rd = inst_field[11:7];

190      MUX4T1_32 MUX4T1_32_1(
191      .S(Jump),
192      .I0(MUX2T1_32_1_o),
193      .I1(Add_32_1_c),
194      .I2(ALU_0_res),
195      .I3(MUX2T1_32_1_o),
196      .O(MUX4T1_32_1_o)
197  );

198
199      MyALU MyALU_0( // need to be updated
200      .A(Regs_0_Rs1_data),
201      .B(MUX2T1_32_0_o),
202      .operation(ALU_Control),
203      .res(ALU_0_res),
204      .zero(ALU_0_zero)
205  );

```

```

207     REG_32 PC(
208     .clk(clk),
209     .rst(rst),
210     .CE(1'b1),
211     .D(MUX4T1_32_1_o),
212     .Q(PC_Q)
213 );
214
215     assign ALU_out = ALU_0_res;
216     assign Data_out = Regs_0_Rs2_data;
217     assign PC_out = PC_Q;
218
219 endmodule

```

4.5 CPU 调试与测试

4.5.1 物理验证

使用实验所提供的 Demo 程序 coe 文件初始化指令存储器 ROM，之后将代码生成比特流后烧录至开发板，观察记录实验结果。

同时为了方便查看相关寄存器的数值，我们将 Datapath 模块中输入 32 个 32 位寄存器的时钟引脚设为 CPU 时钟的取反时钟，这样的话将能够在 CPU 时钟的下降沿即时更新寄存器的数值，更加方便我们对于实验数据结果的查看，做到指令与结果的同步。

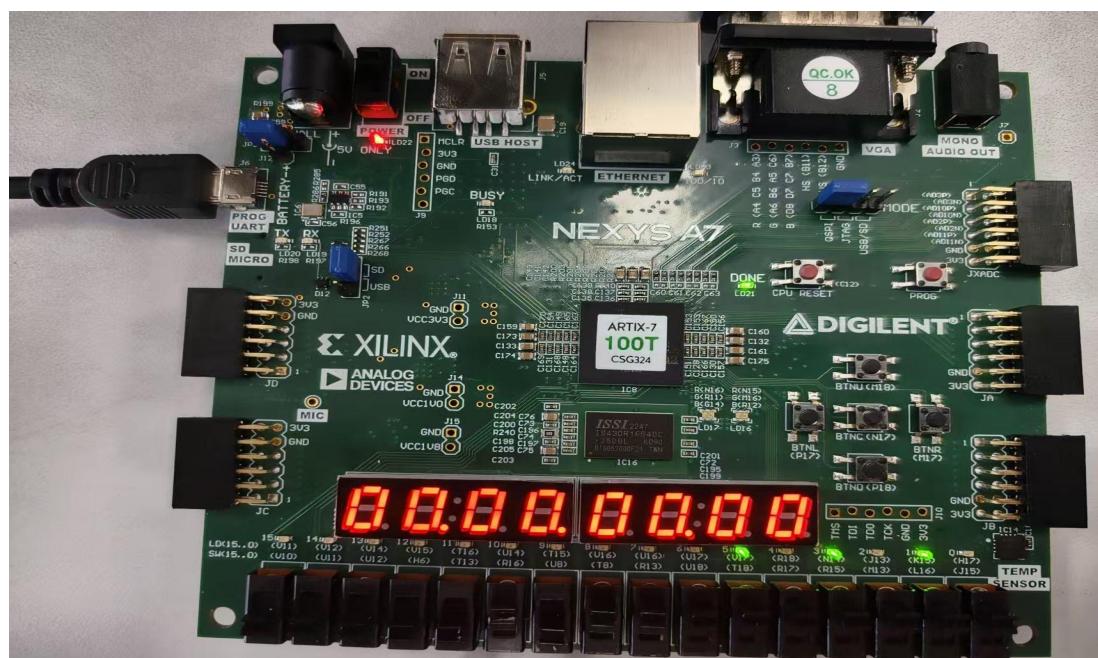
```

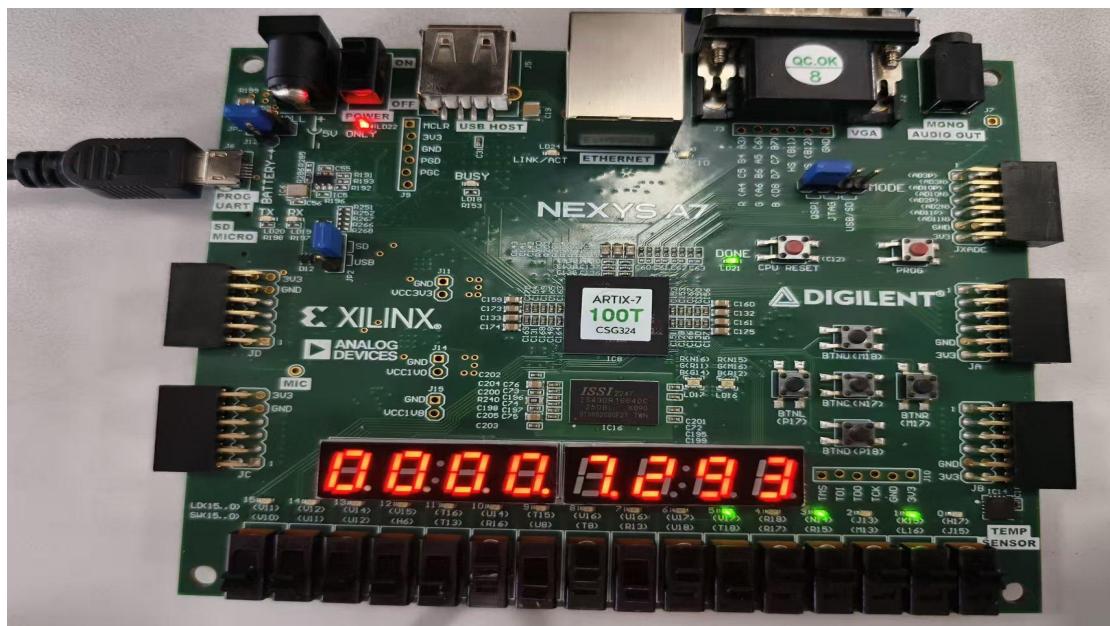
138     assign reg_i_data = MUX4T1_32_0_0;
139
140     Register_32M32b Regs_0(
141         .clk(~clk), // 取反?
142         .rst(rst),

```

五、实验数据记录和处理

5.1 物理验证





```

RV32I Single Cycle CPU

pc: 00000000 inst: 00007293
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

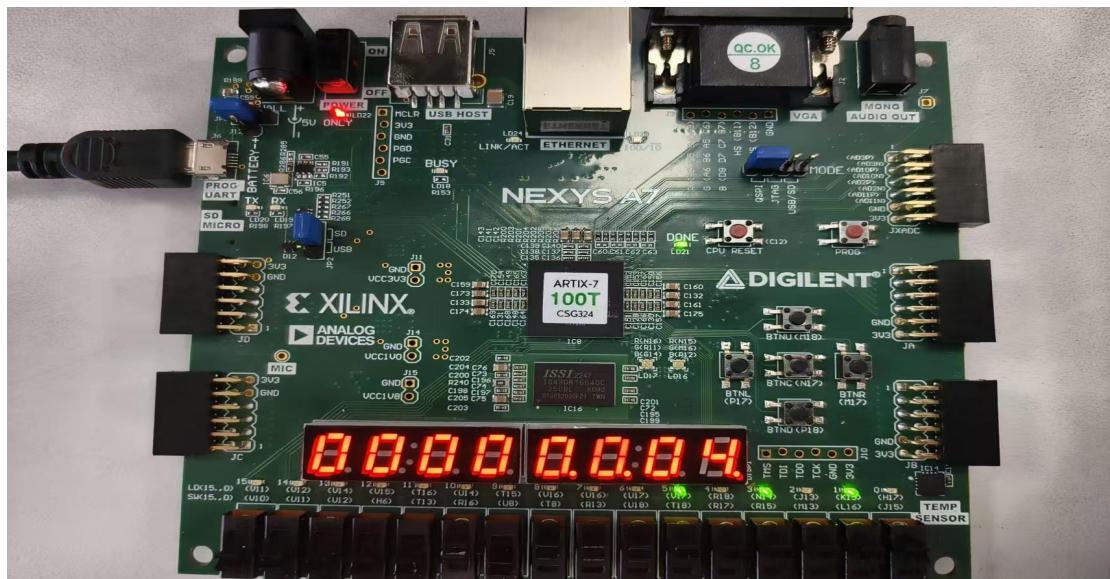
rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 05 reg_i_data: 00000000 reg_wen: 1

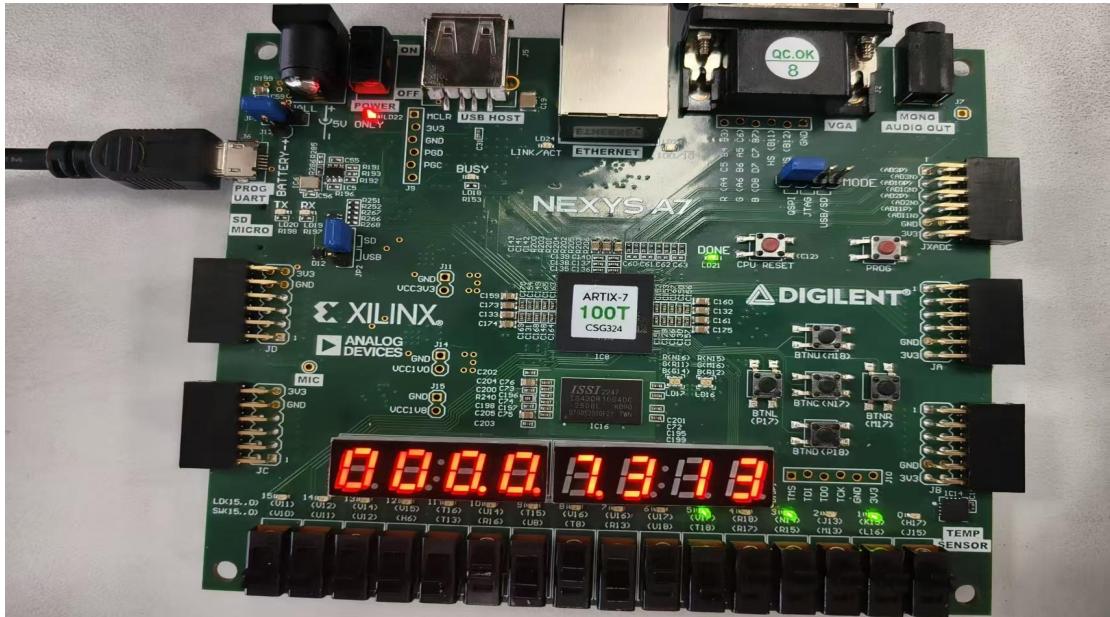
is_imm: 1 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: f0000000 dmem_i_data: 00000000 dmem_addr: 00000000
csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```



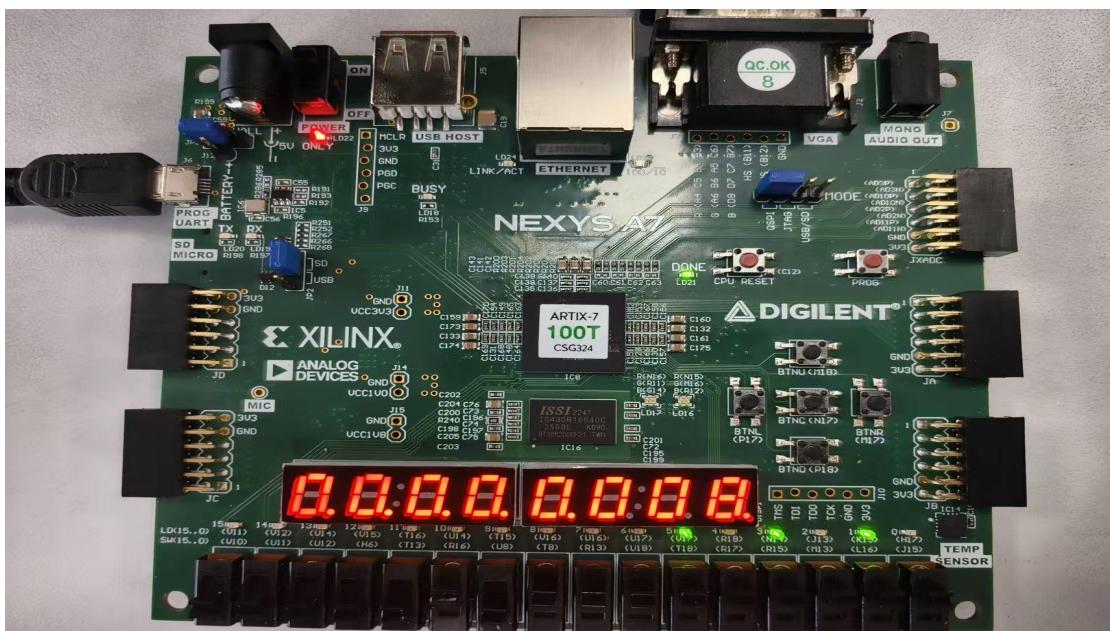


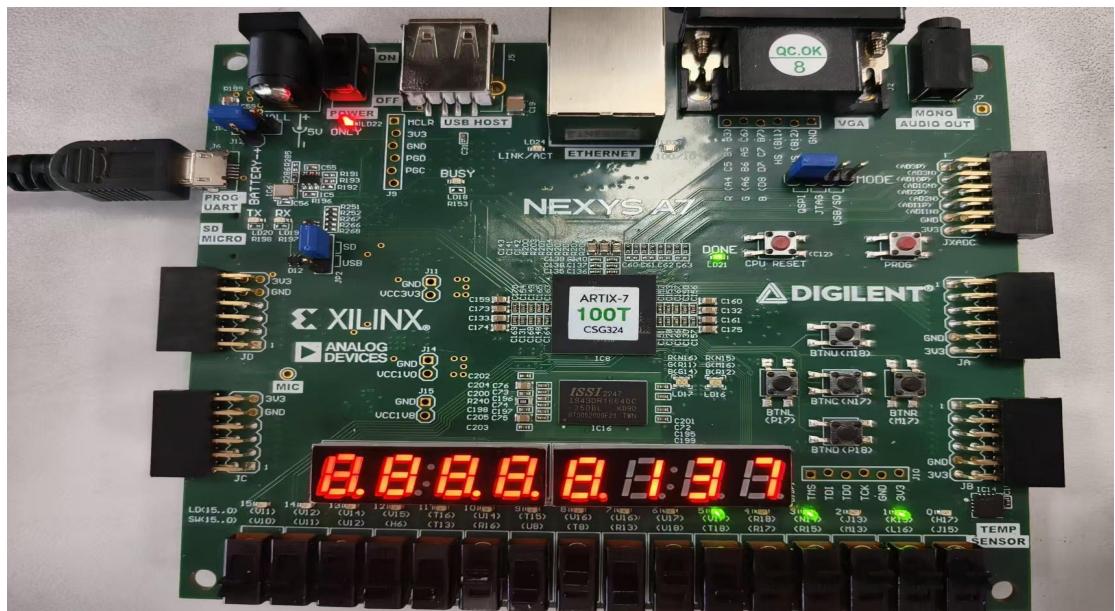
```

RU32I Single Cycle CPU
pc: 00000004 inst: 00007313
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 06 reg_i_data: 00000000 reg_wen: 1
is_imm: 1 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0
is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000
mem_wen: 0 mem_ren: 0
dmem_o_data: f0000000 dmem_i_data: 00000000 dmem_addr: 00000000
csr_wen: 0 csr_ind: 00000000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```





```

RU32I Single Cycle CPU
pc: 00000008    inst: 88888137
x0: 00000000    ra: 00000000    sp: 88888000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000    s11: 00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000000

rs1: 11    rs1_val: 00000000
rs2: 08    rs2_val: 00000000
rd: 02    reg_i_data: 88888000    reg_wen: 1

is_imm: 0    is_auipc: 0    is_lui: 1    imm: 88888000
a_val: 00000000    b_val: 00000000    alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000000    cmp_res: 0

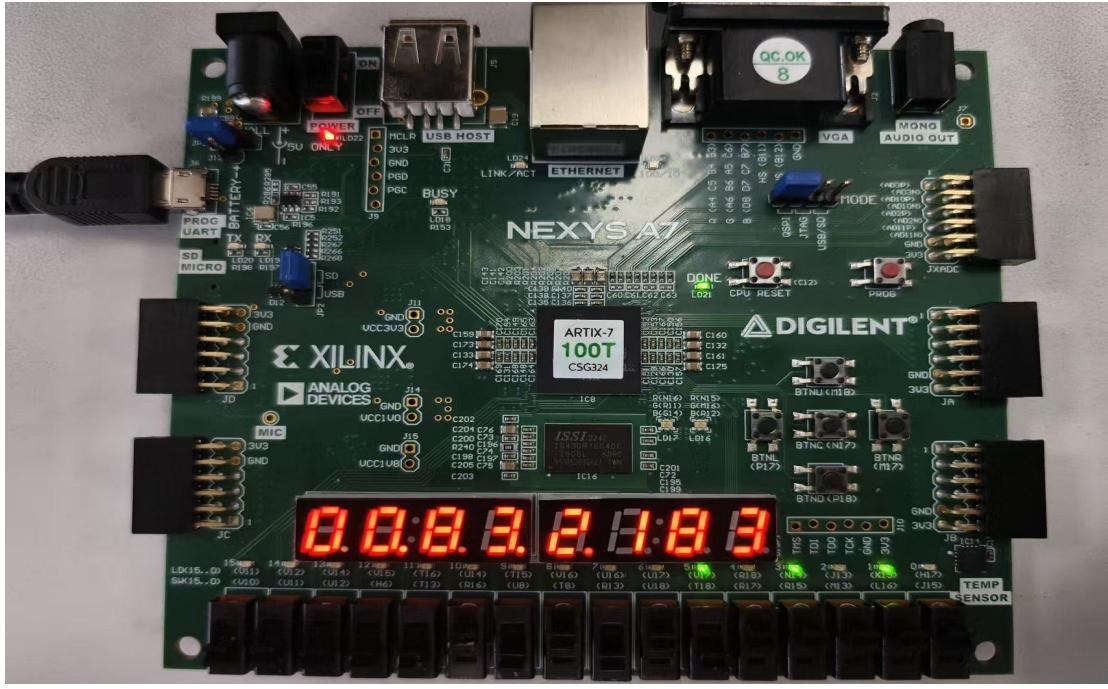
is_branch: 0    is_jal: 0    is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0    mem_ren: 0
dmem_o_data: f0000000    dmem_i_data: 00000000    dmem_addr: 00000000

csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000    mcause: 00000000    mepc: 00000000    mtval: 00000000
mtvec: 00000000    mie: 00000000    mip: 00000000

```





```

RU32I Single Cycle CPU
pc: 0000000c inst: 00832183

x0: 00000000 ra: 00000000 sp: 88888000 gp: 80000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 a8: 00000000 a9: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 06 rs1_val: 00000000
rs2: 08 rs2_val: 00000000
rd: 03 reg_i_data: 80000000 reg_wen: 1

is_imm: 1 is_auipc: 0 is_lui: 0 imm: 00000008
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000008 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

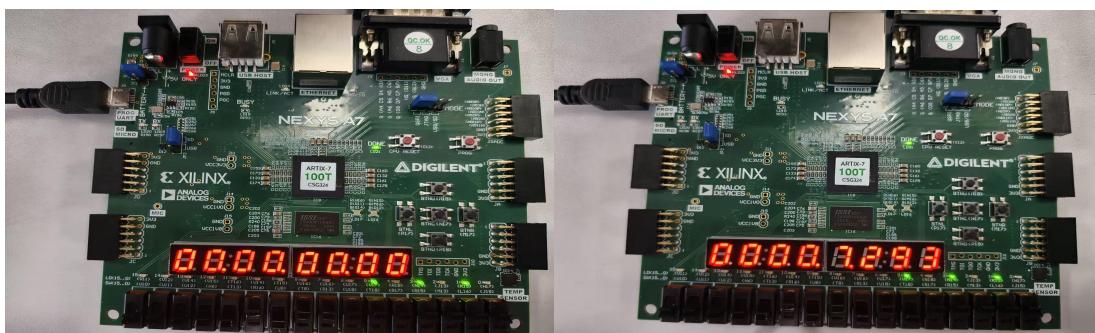
mem_wen: 0 mem_ren: 0
dmem_o_data: 80000000 dmem_i_data: 00000000 dmem_addr: 00000008

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

六、实验结果与分析

6.1 物理验证



```

RU32I Single Cycle CPU

pc: 00000000 inst: 00007293

x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 05 reg_i_data: 00000000 reg_wen: 1

is_imm: 1 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: f0000000 dmem_i_data: 00000000 dmem_addr: 00000000
csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

第一条指令运行后，x5 寄存器值为 0。



```

RU32I Single Cycle CPU

pc: 00000004 inst: 00007313

x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 06 reg_i_data: 00000000 reg_wen: 1

is_imm: 1 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: f0000000 dmem_i_data: 00000000 dmem_addr: 00000000
csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

第二条指令执行之后，x6 寄存器数值为 0。



```

RU32I Single Cycle CPU

pc: 00000008 inst: 88888137

x0: 00000000 ra: 00000000 sp: 88888000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 11 rs1_val: 00000000
rs2: 08 rs2_val: 00000000
rd: 02 reg_i_data: 88888000 reg_wen: 1

is_imm: 0 is_auiipc: 0 is_lui: 1 imm: 88888000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0

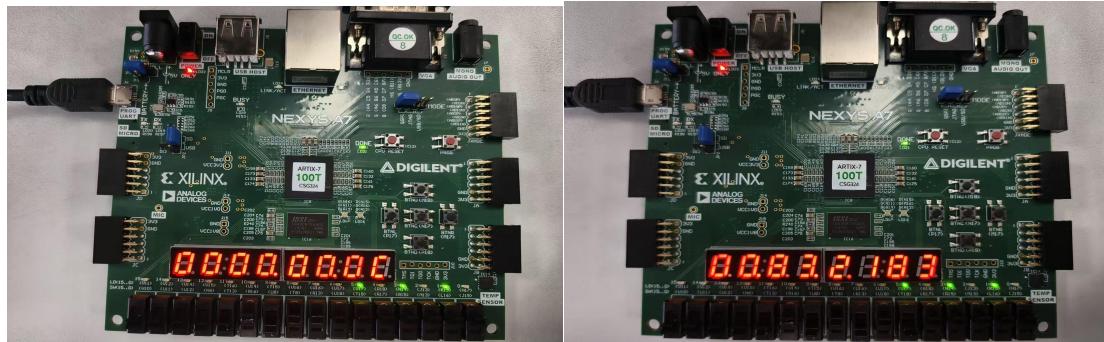
is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: f0000000 dmem_i_data: 00000000 dmem_addr: 00000000

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

第三条指令运行之后，x2 寄存器数值为 0x88888000。



```

RU32I Single Cycle CPU

pc: 0000000c inst: 00832183

x0: 00000000 ra: 00000000 sp: 88888000 gp: 80000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 06 rs1_val: 00000000
rs2: 08 rs2_val: 00000000
rd: 03 reg_i_data: 80000000 reg_wen: 1

is_imm: 1 is_auiipc: 0 is_lui: 0 imm: 00000008
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000008 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 80000000 dmem_i_data: 00000000 dmem_addr: 00000008

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000000 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

第四条指令运行之后，x3 寄存器数值为 0x80000000。

6.2 代码分析

6.2.1 指令扩展后的 ALU

```

23 ⊕ module MyALU(
24     input wire [31:0] A,
25     input wire [31:0] B,
26     input wire [3:0] operation,
27     output reg [31:0] res,
28     output wire zero
29 );
30
31 ⊕     always @ (*) begin
32 ⊕         case(operation)
33             4'b0000:begin res = A & B;end//and:and
34             4'b0001:begin res = A | B;end//or:or
35             4'b0010:begin res = A + B;end//add:add
36             4'b0110:begin res = A - B;end//sub:sub
37             4'b0111:begin res = ($signed(A) < $signed(B));end//set on if less than signed:slt
38             4'b1001:begin res = (A < B);end// set on if less than unsigned:sltu
39             4'b1100:begin res = A ^ B;end// xor:xor
40             4'b1101:begin res = A >> B;end// shift right logical:srl
41             4'b1110:begin res = A << B;end// shift left logical:sll
42             4'b1111:begin res = $signed(A) >>> B;end//shift right arithmetic:sra
43 ⊕         endcase
44 ⊕     end
45
46     assign zero = (res == 32'b0);
47 ⊕ endmodule

```

当我们将 operation 控制信号改为 4 位之后,为了最大程度的减小代码改动量,我们充分利用原来三位控制信号,原有的算数、逻辑操作仅仅增加高一位,对于新增加的操作则使用前所未有的编码进行控制。

6.2.2 指令扩展后的 ImmGen

```

23 ⊕ module ImmGen_new // 此模块内不转换为半字
24     input wire [2:0] ImmSel, // 立即数操作控制
25     input wire [31:0] inst_field, // 指令数据域 [31: 7]
26     output reg [31:0] Imm_out // 立即数输出
27 );
28
29 ⊕     always @ * begin
30         case(ImmSel)
31             3'b001:begin Imm_out = ({20(inst_field[31])},inst_field[31:20]); end// lw,Addi,slti,sltiu,xori,ori,andi,slli,srai,jalr
32             3'b010:begin Imm_out = ({20(inst_field[31])},inst_field[31:25],inst_field[11:7]); end// sw(s)
33             3'b011:begin Imm_out = ({19(inst_field[31])},inst_field[31],inst_field[7],inst_field[30:25],inst_field[11:8],1'b0); end// beq,bne
34             3'b100:begin Imm_out = ({11(inst_field[31])},inst_field[31],inst_field[19:12],inst_field[20],inst_field[30:21],1'b0); end// jal(j)
35             3'b000:begin Imm_out = (inst_field[31:12],12'b000000000000); end // lui
36     endcase
37 ⊕ end
38 ⊕ endmodule

```

该模块代码使用 case 语句对相应的立即数生成控制信号进行响应。RSICV 指令的精简、优格式保证了该模块的简洁与高效。

6.2.3 指令扩展后的 SCPU_ctrl

```

22 ⊕ module SCPU_ctrl_more(
23     input wire [4:0] OPCode, // inst[6:2]
24     input wire [2:0] Fun3, // inst[14:23]
25     input wire Fun7, // inst[30]
26     input wire MIO_ready, // CPU wait
27     output reg [2:0] ImmSel, // 立即数选择控制 // need updating
28     output reg ALUSrc_B, // 源操作数2选择
29     output reg [1:0] MemtoReg, // 写回数据选择控制
30     output reg [1:0] Jump, // jal // need updating
31     output reg Branch, // beq
32     output reg BranchN, // need updating
33     output reg RegWrite, // 寄存器写使能
34     output reg MemRW, // 存储器读写使能
35     output reg [3:0] ALU_Control, // ALU控制
36     output reg CPU_MIO // not use
37 );

```

该部分代码展示了该模块的输入输出端口。

```

39     wire [3:0] Fun;
40     reg [1:0] ALUop;
41     //reg [10:0] CPU_ctrl_signals;
42
43     assign Fun = {Fun3,Fun7};

```

该部分代码声明了模块内部使用的中间变量，并进行相应的赋值。

```
46 ⊖    always @ * begin
47      CPU_MIO = MIO_ready;
48      case(OPcode)
49        5'b01100:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'b00010000010xxx0;end// 
50        5'b00000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'b101100000000010;end// 
51        5'b11001:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'b1101xx10000001x;end// 
52        5'b10000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'b100100000110010;end// 
53        5'b11100:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 13'b0xx001000000100;end// 
54        5'b11000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 13'b0xx001000001011;end// SB:beq b
55        5'b01101:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'bxx11x000xx0000;end// 
56        5'b11011:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'b1101xx01xx100x;end// 
57        default:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN} = 14'b0000000000000000;end // 
58 ⊖ endcase
```

该部分代码根据 opcode 进行第一级译码，对于根据指令类型即可确定的控制信号进行相应的赋值，并准备好用于二级译码的 ALUop 控制信号。

```
~ ⊖
60 ⊖      case(ALUop)
61      2'b00:ALU_Control = 4'b0010;// S-type:addi/算地址
62 ⊖
63 ⊖      2'b01:
64      case(Fun3)
65        3'b000:begin BranchN = 1'b0;ALU_Control = 4'b0110;end// B-type:beq sub比较条件
66      3'b001:begin BranchN = 1'b1;ALU_Control = 4'b0110;end// B-type:bne sub比较条件
67 ⊖ endcase
68 ⊖
69      2'b10:
70      case(Fun)
71        4'b0000:begin ALU_Control = 4'b0010;end// add
72        4'b0001:begin ALU_Control = 4'b0110;end// sub
73        4'b0010:begin ALU_Control = 4'b1110;end// sll
74        4'b0011:begin ALU_Control = 4'b0000;end// and
75        4'b1100:begin ALU_Control = 4'b0001;end// or
76        4'b1000:begin ALU_Control = 4'b1100;end// xor
77        4'b0100:begin ALU_Control = 4'b0111;end// slt
78        4'b0110:begin ALU_Control = 4'b1001;end// sltu
79        4'b1010:begin ALU_Control = 4'b1101;end// srl
80        4'b1011:begin ALU_Control = 4'b1111;end// sra
81      default:begin ALU_Control = 3'bxxx;end
82 ⊖ endcase
```

该部分代码针对 S、B、R 类型的代码进行 ALU 控制信号的翻译，同时针对 R 型指令，由于种类较多，仍需要根据指令的 Function3 和 Function7 部分最终确定 ALU 的控制信号。

```
81 ⊖
82 ⊖      2'b11:
83      case(Fun3)
84        3'b000:begin ALU_Control = 4'b0010;end// addi
85        3'b010:begin ALU_Control = 4'b0111;end// slti
86        3'b011:begin ALU_Control = 4'b1001;end// sltiu
87        3'b100:begin ALU_Control = 4'b1100;end// xori
88        3'b110:begin ALU_Control = 4'b0001;end// ori
89        3'b111:begin ALU_Control = 4'b0000;end// andi
90        3'b001:begin ALU_Control = 4'b1110;end// slli
91      case(Fun7)
92        1'b0:begin ALU_Control = 4'b1101;end// srli
93        1'b1:begin ALU_Control = 4'b1111;end// end
94      endcase
95      default:begin ALU_Control = 3'bxxx;end
96 ⊖ endcase
97 ⊖
98 ⊖ end
99 ⊖
100 ⊖ endmodule
```

该部分代码针对 I 型指令进行 ALU 操作控制信号的翻译，同时根据 Function3 和 Function7 部分进行再次的翻译。

6.2.4 设计指令扩展后的 Datapath

```

23 ⊜ module DataPath_more(
24   input wire Branch,
25   input wire BranchN,
26   input wire [1:0] Jump,
27   input wire [31:0] Data_in,
28   input wire [1:0] MemtoReg,
29   input wire ALUSrc_B,
30   input wire [2:0] ImmSel,
31   input wire [31:0] inst_field,
32   input wire [3:0] ALU_Control, // ALU_operation
33   input wire clk,
34   input wire rst,
35   input wire RegWrite,
36   output wire [31:0] ALU_out,
37   output wire [31:0] Data_out,
38   output wire [31:0] PC_out,
39
40   output wire [31:0] x0,
41   output wire [31:0] ra,
42   output wire [31:0] sp,
43   output wire [31:0] gp,
44   output wire [31:0] tp,
45   output wire [31:0] t0,
46   output wire [31:0] t1,
47   output wire [31:0] t2,
48   output wire [31:0] s0,
49   output wire [31:0] s1,
50   output wire [31:0] a0,
51   output wire [31:0] a1,
52   output wire [31:0] a2,
53   output wire [31:0] a3,
54   output wire [31:0] a4,
55   output wire [31:0] a5,
56   output wire [31:0] a6,
57   output wire [31:0] a7,
58   output wire [31:0] s2,
59   output wire [31:0] s3,
60   output wire [31:0] s4,
61   output wire [31:0] s5,
62   output wire [31:0] s6,
63   output wire [31:0] s7,
64   output wire [31:0] s8,
65   output wire [31:0] s9,
66   output wire [31:0] s10,
67   output wire [31:0] s11,
68   output wire [31:0] t3,
69   output wire [31:0] t4,
70   output wire [31:0] t5,
71   output wire [31:0] t6,
72
73   output wire [4:0] rs1,
74   output wire [4:0] rs2,
75   output wire [31:0] rs1_val,
76   output wire [31:0] rs2_val,
77   output wire [4:0] rd,
78   output wire [31:0] reg_i_data,
79   output wire [31:0] imm
80 );

```

以上部分代码展示了数据通路模块的输入输出端口，而可以看到由于引出 VGA 显示信号，我们的模块的输出端口变得非常的多。

```

95     ImmGen_new ImmGen_0(
96         .ImmSel(ImmSel),
97         .inst_field(inst_field),
98         .Imm_out(ImmGen_0_Imm_out)
99     );
100
101     assign imm = ImmGen_0_Imm_out;

```

该部分代码实例化引用更新后的立即数生成模块。

```

103     Add_32 Add_32_0(
104         .b(PC_Q),
105         .a(32'd0004),
106         .c(Add_32_0_c)
107     );
108
109     Add_32 Add_32_1(
110         .a(PC_Q),
111         .b(ImmGen_0_Imm_out),
112         .c(Add_32_1_c)
113     );
114
115     MUX2T1_32 MUX2T1_32_1(
116         .I0(Add_32_0_c),
117         .I1(Add_32_1_c),
118         .sel((Branch & ALU_0_zero) | (BranchN & ~ALU_0_zero)),
119         .O(MUX2T1_32_1_o)
120     );
121
122     MUX4T1_32 MUX4T1_32_0(
123         .S(MemtoReg),
124         .I0(ALU_0_res),
125         .I1(Data_in),
126         .I2(Add_32_0_c),
127         .I3(ImmGen_0_Imm_out),
128         .O(MUX4T1_32_0_o)
129     );

```

该部分代码计算出顺承的程序计数以及跳转时的程序计数，根据 Branch、BranchN、Jump 控制信号进行选择；并根据 MemtoReg 信号选择适当的写回到寄存器中的数值。

```

131     MUX2T1_32 MUX2T1_32_0(
132         .I0(Regs_0_Rs2_data),
133         .I1(ImmGen_0_Imm_out),
134         .sel(ALUSrc_B),
135         .O(MUX2T1_32_0_o)
136     );
137
138     assign reg_i_data = MUX4T1_32_0_o;

```

该部分代码根据 ALUSrc_B 信号选择作为 ALU 第二个操作数的信号来源。

```

140     Register_32M32b Regs_0(
141         .clk(~clk), // 取反?
142         .rst(rst),
143         .RegWrite(RegWrite),
144         .Rs1_addr(inst_field[19:15]),
145         .Rs2_addr(inst_field[24:20]),
146         .Wt_addr(inst_field[11:7]),
147         .Wt_data(MUX4T1_32_0_O),
148         .Rs1_data(Regs_0_Rs1_data),
149         .Rs2_data(Regs_0_Rs2_data),
150             - - -
150         .x0(x0),
151         .ra(ra),
152         .sp(sp),
153         .gp(gp),
154         .tp(tp),
155         .t0(t0),
156         .t1(t1),
157         .t2(t2),
158         .s0(s0),
159         .s1(s1),
160         .a0(a0),
161         .a1(a1),
162         .a2(a2),
163         .a3(a3),
164         .a4(a4),
165         .a5(a5),
166         .a6(a6),
167         .a7(a7),
168         .s2(s2),
169         .s3(s3),
170         .s4(s4),
171         .s5(s5),
172         .s6(s6),
173         .s7(s7),
174         .s8(s8),
175         .s9(s9),
176         .s10(s10),
177         .s11(s11),
178         .t3(t3),
179         .t4(t4),
180         .t5(t5),
181         .t6(t6),
182         .rs1_val(rs1_val),
183         .rs2_val(rs2_val) // o
184     );

```

该部分代码实例化引用 32 个 32 位寄存器模块作为 CPU 内部的寄存器。

```

186     assign rs1 = inst_field[19:15];
187     assign rs2 = inst_field[24:20];
188     assign rd = inst_field[11:7];

```

该部分代码对 VGA 显示所需要的信号进行相应的赋值。

```

190      MUX4T1_32 MUX4T1_32_1(
191          .S(Jump),
192          .I0(MUX2T1_32_1_o),
193          .I1(Add_32_1_c),
194          .I2(ALU_0_res),
195          .I3(MUX2T1_32_1_o),
196          .O(MUX4T1_32_1_o)
197      );
198
199      MyALU MyALU_0( // need to be updated
200          .A(Regs_0_Rs1_data),
201          .B(MUX2T1_32_0_o),
202          .operation(ALU_Control),
203          .res(ALU_0_res),
204          .zero(ALU_0_zero)
205      );

```

该部分代码根据是否跳转选择恰当的链接程序计数写会到寄存器中；并且实例化引用更新后的 ALU 模块进行相关结果、地址的计算。

```

207      REG_32 PC(
208          .clk(clk),
209          .rst(rst),
210          .CE(1'b1),
211          .D(MUX4T1_32_1_o),
212          .Q(PC_Q)
213      );
214
215      assign ALU_out = ALU_0_res;
216      assign Data_out = Regs_0_Rs2_data;
217      assign PC_out = PC_Q;
218
219 endmodule

```

该部分代码实例化引用程序计算器，对下一条指令的程序计数地址进行计算；最终对部分输出端口进行赋值。

6.3 结果分析

根据我们物理测试的下板结果，我们看到对于新增的 lui 等指令我们的程序均能做出正确的响应，包括写回寄存器正确的数值以及将正确的数据读出、写出到内存中。因此我们可以认为我们设计的程序可以正确地运行。

本次实验在前三次实验所设计的简单指令 CPU 上进行扩展，逐渐丰富我们 CPU 的指令功能。

七、讨论、心得

在实验过程中，我曾被 VGA 信号复杂的跨多模块传递搞得晕头转向；也曾被数据信号更新不彻底带来的错误数据结果搞得不知所措。给人的感觉就是只有自己做起来才会发现这个过程中真的可能有这么多的问题。对于解决这些问题，最好的方法就是仔细检查，去花时间，集中注意力，尤其注意善于利用 Vivado 软件的变量选中高亮功能，但也要小心他的不对大小写进行区分（但是作为变量接口，必须要区分大小写）。

本次实验再次让我看到了实验与理论之间的互相印证、指导关系。实验加深了我对与理论知识的理解，同时理论也让我们的实验有了更加明确的方向。

Experiment4-CPU 设计之中断

一、实验目的和要求

1.1 实验目的

- 深入理解 CPU 结构
- 学习如何提高 CPU 使用效率
- 学习 CPU 中断工作原理
- 设计中断测试程序

1.2 实验目标及任务

• 目标

熟悉 RISC-V 中断的原理，了解引起 CPU 中断产生的原因及其处理方法，扩展包含中断的 CPU

• 任务一：扩展实验 CPU 中断功能

修改设计数据通路和控制器

修改或替换 Exp04-3 的数据通路及控制器

兼容 Exp04-3 数据通路增加中断通路

增加中断控制

扩展 CPU 中断功能

非法指令中断；

外部中断；

ecall

• 任务二：设计 CPU 中断测试方案并完成测试

二、实验内容和原理

2.1 中断概念

中断是指程序执行过程中，当发生某个事件时，中止CPU上现行程序的运行，引出处理该事件的程序执行的过程，此过程都需要打断处理器正常的工作，为此，才提出了“中断”的概念。



- **中断源**:引起中断的事件称为中断源;
- **中断请求**:中断源向CPU提出处理的请求;
- **断点**:发生中断时被打断程序的暂停点;
- **中断响应**:CPU暂停现行程序而转为响应中断请求的过程;
- **中断处理程序**:处理中断源的程序;
- **中断处理**:CPU执行有关的中断处理程序;
- **中断返回**:返回断点的过程;

□ 按照中断信号的来源，可把中断分为外中断和内中断两类：

- **外中断(又称中断)**:指来自处理器和主存之外的中断;
- **内中断(又称异常)**:指来自处理器和主存内部的中断;

□ 中断处理程序主要工作：

- 保护CPU现场
- 处理发生的中断事件
- 恢复正常操作

狭义的中断
和异常均可
归于广义的
异常范畴

2.2 中断（异常）处理过程

当CPU收到中断或者异常的信号时，它会暂停执行当前的程序或任务，通过一定的机制跳转到负责处理这个信号的相关处理程序中，在完成对这个信号的处理后再跳回到刚才被打断的程序或任务中。



2.3 RISC-V 中断结构

□ RISC-V架构工作模式

- 机器模式(Machine Mode)
- 用户模式(User Mode)
- 监督模式(Supervisor Mode)

□ 不同的模式下均可产生异常以及中断

□ RISC-V架构中机器模式是必须具备的模式，因此必须具备机器模式的异常处理机制（本实验只针对机器模式下的异常（中断））

2.4 RISC-V 中断处理

2.4.1 进入异常

RISC-V处理器检测到异常，开始进行异常处理：

- 停止执行当前的程序流，转而从CSR寄存器mtvec定义的PC地址开始执行；
- 更新机器模式异常原因寄存器mcause
- 更新机器模式异常PC寄存器mepc
- 更新机器模式异常值寄存器mtval
- 更新机器模式状态寄存器mstatus

2.4.2 异常入口基址寄存器-**mtvec**

□ RISC-V处理器进入异常后，跳入的PC地址由**mtvec**寄存器指定：



CAUSE表示
中断对应的
异常编号

- MODE = 0;异常响应时，处理器跳转到BASE值指示的PC地址
- MODE = 1;异常响应时，处理器跳转到BASE值指示的PC地址
- MODE = 1;中断响应时，处理器跳转到BASE+4*CAUSE值指示的PC地址

2.4.3 异常原因寄存器-mcause

- RISC-V处理器进入异常后，异常的引发原因由mcause寄存器指定：

31 30		0
interrupt	Exception code	
INT	EC	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned

INT	EC	Description
0	7	Store/AMO access fault
0	10..8	Not supported
0	11	Ecall from M-mode
0	>=12	Reserved
1	2..0	Reserved
1	3	Machine Software Interrupt
1	6..4	Reserved
1	7	Machine Timer Interrupt
1	10..8	Reserved
1	11	Machine External Interrupt
1	>=12	Reserved

2.4.4 异常 PC 寄存器-mepc

- RISC-V处理器进入异常后，异常的返回地址由mepc寄存器保存
- 在进入异常时，硬件将自动更新mepc寄存器的值为当前遇到异常的指令PC值(即当前程序的停止执行点)
- Mepc寄存器作为异常的返回地址，在异常结束后，能够使用它保存的PC值回到之前被停止执行的程序点

2.4.5 异常值寄存器-mtval

- RISC-V处理器进入异常后，异常的存储器访问地址或指令编码由mtval寄存器保存
- 如果是存储器访问造成的异常，如遭遇硬件断点、取指令、读写存储器造成异常，则将存储器访问的地址更新到mtval寄存器中
- 如果是非法指令造成的异常，则将改指令的指令编码更新到mtval寄存器中

2.4.6 异常状态寄存器-mstatus

- RISC-V处理器进入异常后，异常的各种状态由mstatus寄存器指示

Bits	Name	Attributes	Description
2..0	RSV	RZ	Reserved
3	MIE	RW	Global interrupt enable
6..4	RSV	RZ	Reserved
7	MPIE	RW	Previous global interrupt enable
10..8	RSV	RZ	Reserved
12..11	MPP	QRO	Previous privilege mode (hardwired to 11)
31..13	RSV	RZ	Reserved

- MIE = 1;表示机器模式下所有中断全局打开
- MIE = 0;表示机器模式下所有中断全局关闭

2.5 中断相关指令

□ 异常返回

- MRET



□ 环境调用

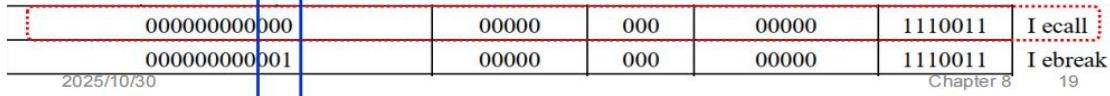
- ecall

- MEPC = ecall指令本身的PC值

□ 断点

- ebreak

- MEPC = ebreak指令本身的PC值



2.6 RISC-V 中断结构---退出异常

- 当异常程序处理完成后，最终要从异常服务程序中退出，并返回主程序。RISCV中定义了一组退出指令MRET，SRET，和URET，对于机器模式，对应MRET。
- 在机器模式下退出异常时候，软件须使用MRET。RISCV架构规定，处理器执行完MRET指令后，硬件行为如下：

2.7 RISC-V 中断结构---异常服务程序

- 处理器进入异常后，即开始从mtvec寄存器定义的PC地址执行新的程序。
- 所执行的新的程序即为异常服务程序，并且程序还可以通过查询mcause中的异常编号决定跳转到更具体的异常服务程序。

2.8 典型处理器中断结构

□ Intel x86中断结构

- 中断向量：000~3FF，占内存最底1KB空间
 - 每个向量由二个16位生成20位中断地址
 - 共256个中断向量，向量编号n=0~255
 - 分硬中断和软中断，响应过程类同，触发方式不同
 - 硬中断响应由控制芯片8259产生中断号n(接口原理课深入学习)

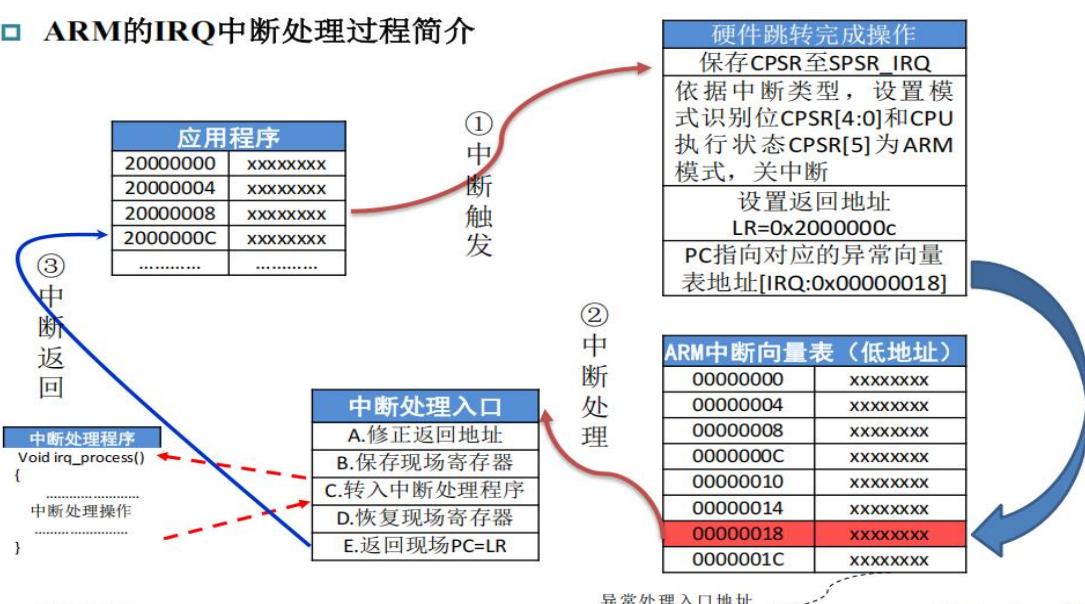
□ ARM中断结构

- 固定向量方式(嵌入式课程深入学习)

异常类型	偏移地址(低)	偏移地址(高)
复位	00000000	FFFF0000
未定义指令	00000004	FFFF0004
软中断	00000008	FFFF0008
预取指令终	0000000C	FFFF000C
数据终止	00000010	FFFF0010
保留	00000014	FFFF0014
中断请求(IRQ)	00000018	FFFF0018
快速中断请求(FIQ)	0000001C	FFFF001C

2.9 典型处理器中断结构---ARM 中断

□ ARM的IRQ中断处理过程简介



2.10 简化中断设计

□ ARM中断向量表

向量地址	ARM异常名称	ARM系统工作模式	本实验定义
0x00000000	复位	超级用户 Svc	复位
0x00000004	未定义指令终止	未定义指令终止 Und	非法指令异常
0x00000008	软中断 (SWI)	超级用户 Svc	ECALL
0x0000000c	Prefetch abort	指令预取终止 Abt	Int外部中断 (硬件)
0x00000010	Data abort	数据访问终止 Abt	Reserved自定义
0x00000014	Reserved	Reserved	Reserved自定义
0x00000018	IRQ	外部中断模式 IRQ	Reserved自定义
0x0000001C	FIQ	快速中断模式 FIQ	Reserved自定义

□ 简化中断设计

- 参考ARM中断向量
 - 实现非法指令异常和外部中断以及ECALL
 - 设计寄存器MEPC

- 1.外部中断 (Int) 触发中断或非法指令 (illegal) 触发异常或ecall系统调用
- 2.响应mtvec寄存器定义的PC值分别针对Int为0x0c; ecall为0x08; illegal为0x04
- 3.mepc寄存器值更新为下一条指令的PC值
- 4.执行异常服务程序
- 5.执行mret指令，返回mepc保存的PC处继续程序流

mtvec在此设计中仅为reg类型变量，功能是存储中断向量

mepc为受clk控制的寄存器，功能是暂存返回PC值

2025/10/30

26

三、主要仪器设备

- 计算机 (Intel Core i9-13980, 16GB 内存) 系统
- NEXYS A7 开发板
- Xilinx VIVADO2024.2 及以上开发工具

四、操作方法与实验步骤

4.1 扩展 CPU 中断功能

对于中断（异常），我们可以概括为指令层面和硬件层面。其中指令层面指的是 mret、ecall 和非法指令，这部分中断需要我们在解码指令的过程中做出相

应的判断。其中 mret、ecall 具有自己的特有指令编码，通过观察发现通过 Opcode 我们就可以判断出此两个指令，那我们应当在第一级解码的时候增设 mret、ecall 识别信号。

而对于外部中断，属于硬件层面，我们完全可以通过数据通路的修改实现。

同时我们增加专用于终端检测的模块，同时增加 mtvec、mepc 寄存器，从而完整的实现 CPU 中断处理。

为了避免在程序计算器模块处理外部中断信号的造成的不定性，我们在程序计数器外部增设外部中断检测模块 INT_Detect 来实现此终端的检测。

4.1.1 修改 SCPU_ctrl

对于非法指令，我们知道 RISC-V 指令均具有简洁以及相对规律的格式，所以 Opcode 和 Function3、Function7 部分不符合要求的指令一定是非法指令，那么我们就可以通过相应指令部分的检测实现对于非法指令的检测。而对于逻辑错误这种非法指令，实际上更多的是由操作系统软件来进行。

修改后的代码部分如下图所示：

```
21 module SCPU_ctrl_more(
22     input wire [4:0] Opcode, // inst[6:2]
23     input wire [2:0] Fun3, // inst[14:23]
24     input wire Fun7, // inst[30]
25     input wire MIO_ready, // CPU wait
26     output reg [2:0] ImmSel, // 立即数选择控制 // need updating
27     output reg ALUSrc_B, // 源操作数2选择
28     output reg [1:0] MemtoReg, // 写回数据选择控制
29     output reg [1:0] Jump, // jal // need updating
30     output reg Branch, // beq
31     output reg BranchN, // need updating
32     output reg RegWrite, // 寄存器写使能
33     output reg MemRW, // 存储器读写使能
34     output reg [3:0] ALU_Control, // ALU控制
35     output reg ill_instr,
36     //output reg ecall,
37     //output reg mret,
38     output reg CPU_MIO // not use
39 );
```



```
41     wire [3:0] Fun;
42     reg [1:0] ALUop;
43     reg em;
44     //reg [10:0] CPU_ctrl_signals;
45
46
47     assign Fun = {Fun3, Fun7};
```



```
50     always @ * begin
51         CPU_MIO = MIC_ready;
52         case(Opcode)
53             5'b01100:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b0001000010xxx000;end//
54             5'b00000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b101100000001000;end//
55             5'b01000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b1001000011001000;end//
56             5'b11001:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b1101xx1000001x00;end//
57             5'b01000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b1kx0100000010000;end//
58             5'b11000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,em,ill_instr} = 14'b0x00000101100;end// Sb: beg bne bit b
59             5'b01101:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b1k11x000xx00000;end//
60             5'b11011:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b1101xx01xx100x00;end//
61             5'b11100:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'bxxxxxx00xxxxx010;end//
62             default:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b0000000000000001;end //
63         endcase
64     // 此步骤只能判断出opcode不合法的指令，其余指令需要下面继续判断
```

4.1.2 修改 Datapath

我们设计内部有 mtvec、mepc 寄存器的终端检测模块处理。

模块代码如下图所示：

```
23  module InterruptDetect (
24      input clk,
25      input rst,
26      input [31:0] PC_next, // 下一条指令
27      input [31:0] PC_add4,
28      input wire INT, // 外部中断信号
29      input wire mret,
30      input wire ecall,
31      input wire ill_instr, // 非法信号
32      output reg [31:0] PC_certain,
33      output wire [31:0] reg_mepc
34  );
35
36      reg [31:0] mepc; // 储存异常指令的下一条指令
37      reg [31:0] mtvec [0:3];
38
39  initial begin
40      mtvec[0] <= 32'h00000000; // reset
41      mtvec[1] <= 32'h00000004; // ill_instr
42      mtvec[2] <= 32'h00000008; // ecall
43      mtvec[3] <= 32'h0000000c; // INT
44 end
45
46      // mepc 寄存器更新 - 时序逻辑
47  always @ (posedge clk or posedge rst) begin
48      if (rst) begin
49          mepc <= 32'h00000020; // 主程序地址
50      end else if (ill_instr) begin
51          // 发生异常时保存返回地址
52          mepc <= PC_add4;
53      end else if (ecall) begin
54          mepc <= PC_next + 32'h00000004;
55      end else if (INT) begin
56          mepc <= PC_next;
57      end
58      // mret 时不更新 mepc
59 end
60
61      // PC 决策 - 组合逻辑，立即响应
62      // 但是其实这些信号判定都是组合逻辑，所以这里时序逻辑应该也没什么
63  always @ (*) begin
64      if (rst) begin
65          PC_certain = mtvec[0]; // 复位
66      end else if (mret) begin
67          PC_certain = mepc; // 异常返回
68      end else if (ill_instr) begin
69          PC_certain = mtvec[1]; // 非法指令
70      end else if (ecall) begin
71          PC_certain = mtvec[2]; // 系统调用
72      end else if (INT) begin
73          PC_certain <= 32'h0000000c; // 外部中断
74      end else begin
75          PC_certain = PC_next; // 正常执行
76      end
77  end
78
79      assign reg_mepc = mepc;
80
81 endmodule
```

同时针对外部中断，我们专门增设外部中断检测模块。具体代码如下所示：

```
22 module INT_Detect(
23     input clk,           // 必须有时钟
24     input rst,           // 复位信号
25     input wire in,       // 原始输入信号
26     output reg out      // 单周期脉冲输出
27 );
28
29     reg in_prev;         // 保存上一个时钟周期的输入值
30
31     // 时序逻辑：采样输入信号
32     always @ (posedge clk or posedge rst) begin
33         if (rst) begin
34             in_prev <= 1'b0;
35             out <= 1'b0;
36         end else begin
37             in_prev <= in;           // 保存当前值，下一周期用
38             // 检测上升沿并产生单周期脉冲
39             if (in && !in_prev) begin
40                 out <= 1'b1;           // 上升沿，输出1
41             end else begin
42                 out <= 1'b0;           // 其他情况输出0
43             end
44         end
45     end
46 endmodule
```

修改后的 Datapath 模块变动部分如下所示：

```
213     InterruptDetect_ID(
214     .clk(clk),
215     .rst(rst),
216     .PC_next(MUX4T1_32_1_o), // 下一条指令
217     .PC_add4(Add_32_0_c),
218     .INT(INT), // 外部中断信号
219     .mret(mret),
220     .ecall(ecall),
221     .ill_instr(ill_instr), // 非法信号
222     .PC_certain(InterruptDetect_D),
223     .reg_mepc(mepc_o)
224 );
```

修改后的 SCPU 模块变动部分如下所示：

```
121     DataPath_more DataPath_more_0(
122     .ALUSrc_B(SCPU_ctrl_0_ALUSrc_B),
123     .ALU_Control(SCPU_ctrl_0_ALU_Control),
124     .Branch(SCPU_ctrl_0_Branch),
125     .BranchN(SCPU_ctrl_0_BranchN),
126     .Data_in(Data_in),
127     .ImmSel(SCPU_ctrl_0_ImmSel),
128     .Jump(SCPU_ctrl_0_Jump),
129     .MemtoReg(SCPU_ctrl_0_MemToReg),
130     .RegWrite(SCPU_ctrl_0_RegWrite),
131     .clk(clk),
132     .rst(rst),
133     .inst_field(inst_in),
134     .ill_instr(SCPU_ctrl_0_ill_instr),
135     .ecall((inst_in == 32'h00000073)),
136     .mret((inst_in == 32'h30200073)),
137     .INT(INT),
138     .ALU_out(Addr_out),
139     .Data_out(Data_out),
140     .PC_out(PC_out),
```

修改后的 CSSTE 模块变动部分如下所示：

```
189     INT_Detect U12(
190         .clk(U8_Clk_CPU),
191         .rst(U9_rst),
192         .in(U9_SW_OK[15]),
193         .out(U12_out)
194     );
```

4.2 任务二：设计 CPU 中断测试方案并完成测试

参考实验文档提供的物理测试 Demo，我们对我们的程序进行检测记录。

五、实验数据记录和处理

```
RV32I Single Cycle CPU

pc: 00000002c    inst: fe62dac2

x0: 00000000    ra: 00000000    sp: 88888000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000   s11: 00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000000

rs1: 05    rs1_val: 00000000
rs2: 06    rs2_val: 00000000
rd: 15    reg_i_data: 00000000    reg_wen: 0

is_imm: 0    is_auipc: 0    is_lui: 0    imm: ffffff4
a_val: 00000000    b_val: 00000000    alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000000    cmp_res: 0

is_branch: 0    is_jal: 0    is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0    mem_ren: 0
dmem_o_data: 00000000    dmem_i_data: 00000000    dmem_addr: 00000000

csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000    mcause: 00000000    mepc: 00000020    mtval: 00000000
mtvec: 00000000    mie: 00000000    mip: 00000000
```

```
RV32I Single Cycle CPU

pc: 00000004    inst: 0c40006f

x0: 00000000    ra: 00000000    sp: 88888000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000   s11: 00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000000

rs1: 00    rs1_val: 00000000
rs2: 04    rs2_val: 00000000
rd: 00    reg_i_data: 00000008    reg_wen: 1

is_imm: 1    is_auipc: 0    is_lui: 0    imm: 000000c4
a_val: 00000000    b_val: 00000000    alu_ctrl: 0    cmp_ctrl: 0
alu_res: 000000c4    cmp_res: 0

is_branch: 0    is_jal: 1    is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0    mem_ren: 0
dmem_o_data: 00000000    dmem_i_data: 00000000    dmem_addr: 000000c4

csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000    mcause: 00000000    mepc: 00000030    mtval: 00000000
mtvec: 00000000    mie: 00000000    mip: 00000000
```

RV32I Single Cycle CPU

```
pc: 0000000c8 inst: 00168693

x0: 00000000 ra: 00000000 sp: 88888000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 0d rs1_val: 00000000
rs2: 01 rs2_val: 00000000
rd: 0d reg_i_data: 00000001 reg_wen: 1
is_imm: 1 is_auiipc: 0 is_lui: 0 imm: 00000001
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000001 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000000 dmem_addr: 00000001
csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000030 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

RV32I Single Cycle CPU

```
pc: 0000000cc inst: 00168693

x0: 00000000 ra: 00000000 sp: 88888000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000001 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 0d rs1_val: 00000001
rs2: 01 rs2_val: 00000000
rd: 0d reg_i_data: 00000002 reg_wen: 1
is_imm: 1 is_auiipc: 0 is_lui: 0 imm: 00000001
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000002 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000000 dmem_addr: 00000002
csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000030 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

RV32I Single Cycle CPU

```
pc: 000000d0 inst: 30200073

x0: 00000000 ra: 00000000 sp: 88888000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000002 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 02 rs2_val: 88888000
rd: 00 reg_i_data: 77778000 reg_wen: 0
is_imm: 0 is_auiipc: 0 is_lui: 0 imm: 00000300
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 77778000 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000000 dmem_addr: 77778000
csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000030 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

RV32I Single Cycle CPU

```
pc: 00000030    inst: 00832183
x0: 00000000    ra: 00000000    sp: 88888000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000002    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000   s11: 00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000000

rs1: 06    rs1_val: 00000000
rs2: 08    rs2_val: 00000000
rd: 03    reg_i_data: 80000000    reg_wen: 1

is_imm: 1    is_auipc: 0    is_lui: 0    imm: 00000008
a_val: 00000000    b_val: 00000000    alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000008    cmp_res: 0

is_branch: 0    is_jal: 0    is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0    mem_ren: 0
dmem_o_data: 80000000    dmem_i_data: 00000000    dmem_addr: 00000000
csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000    mcause: 00000000    mepc: 00000030    mtval: 00000000
mtvec: 00000000    mie: 00000000    mip: 00000000
```

RV32I Single Cycle CPU

```
pc: 00000074    inst: 00000073
x0: 00000000    ra: 80000000    sp: 88888000    gp: 80000000    tp: 00000000
t0: 80000000    t1: 88888000    t2: 80000000    s0: 08888000    s1: 08888000
a0: 80000000    a1: 00000000    a2: 00000000    a3: 00000002    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000   s11: 00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000000

rs1: 00    rs1_val: 80000000
rs2: 00    rs2_val: 80000000
rd: 00    reg_i_data: 00000000    reg_wen: 0

is_imm: 0    is_auipc: 0    is_lui: 0    imm: 00000000
a_val: 00000000    b_val: 00000000    alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000000    cmp_res: 0

is_branch: 0    is_jal: 0    is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0    mem_ren: 0
dmem_o_data: 80000000    dmem_i_data: 00000000    dmem_addr: 00000000
csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000    mcause: 00000000    mepc: 00000030    mtval: 00000000
mtvec: 00000000    mie: 00000000    mip: 00000000
```

RV32I Single Cycle CPU

```
pc: 00000008    inst: 0d80006f
x0: 00000000    ra: 80000000    sp: 88888000    gp: 80000000    tp: 00000000
t0: 80000000    t1: 88888000    t2: 80000000    s0: 08888000    s1: 08888000
a0: 80000000    a1: 00000000    a2: 00000000    a3: 00000002    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000   s11: 00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000000

rs1: 00    rs1_val: 80000000
rs2: 18    rs2_val: 00000000
rd: 00    reg_i_data: 0000000c    reg_wen: 1

is_imm: 1    is_auipc: 0    is_lui: 0    imm: 000000d8
a_val: 00000000    b_val: 00000000    alu_ctrl: 0    cmp_ctrl: 0
alu_res: 000000d8    cmp_res: 0

is_branch: 0    is_jal: 1    is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0    mem_ren: 0
dmem_o_data: 00000000    dmem_i_data: 00000000    dmem_addr: 000000d8
csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000    mcause: 00000000    mepc: 00000078    mtval: 00000000
mtvec: 00000000    mie: 00000000    mip: 00000000
```

```

RU32I Single Cycle CPU

pc: 000000e0    inst: 00128793

x0: 00000000    ra: 80000000    sp: 88888000    gp: 80000000    tp: 00000000
t0: 80000000    t1: 88888000    t2: 80000000    s0: 08888000    s1: 08888000
a0: 80000000    a1: 00000000    a2: 00000000    a3: 00000002    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000   s11: 00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000000

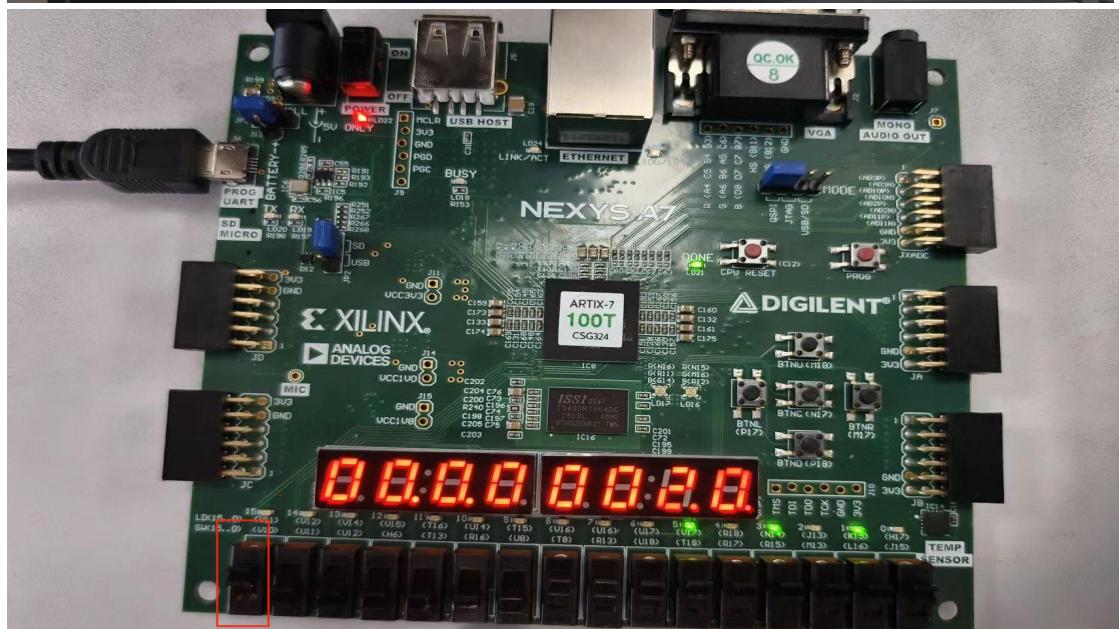
rs1: 05    rs1_val: 80000000
rs2: 01    rs2_val: 80000000
rd: 0f    reg_i_data: 80000001    reg_wen: 1

is_imm: 1    is_auipc: 0    is_lui: 0    imm: 00000001
a_val: 00000000    b_val: 00000000    alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000001    cmp_res: 0

is_branch: 0    is_jal: 0    is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0    mem_ren: 0
dmem_o_data: 80000000    dmem_i_data: 00000000    dmem_addr: B0000001
csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000    mcause: 00000000    mepc: 00000024    mtval: 00000000
mtvec: 00000000    mie: 00000000    mip: 00000000

```



```

RU32I Single Cycle CPU

pc: 0000000c    inst: 0c80006f

x0: 00000000    ra: 00000000    sp: 00000000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000   s11: 00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000000

rs1: 00    rs1_val: 00000000
rs2: 08    rs2_val: 00000000
rd: 00    reg_i_data: 00000010    reg_wen: 1

is_imm: 1    is_auipc: 0    is_lui: 0    imm: 000000c8
a_val: 00000000    b_val: 00000000    alu_ctrl: 0    cmp_ctrl: 0
alu_res: 000000c8    cmp_res: 0

is_branch: 0    is_jal: 1    is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0    mem_ren: 0
dmem_o_data: 00000000    dmem_i_data: 00000000    dmem_addr: 000000c8
csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000    mcause: 00000000    mepc: 00000024    mtval: 00000000
mtvec: 00000000    mie: 00000000    mip: 00000000

```

```

RV32I Single Cycle CPU

pc: 000000d4 inst: 40c70733

x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 0e rs1_val: 00000000
rs2: 0c rs2_val: 00000000
rd: 0e reg_i_data: 00000000 reg_wen: 1

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 0000040e
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 80000000 dmem_i_data: 00000000 dmem_addr: 00000000

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 000000d4 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

```

RV32I Single Cycle CPU

pc: 000000d8 inst: 40c70733

x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 0e rs1_val: 00000000
rs2: 0c rs2_val: 00000000
rd: 0e reg_i_data: 00000000 reg_wen: 1

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 0000040e
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 80000000 dmem_i_data: 00000000 dmem_addr: 00000000

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 000000d4 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

六、实验结果与分析

6.1 代码分析

6.1.1 SCPU_ctrl

```

21 module SCPU_ctrl_more(
22     input wire [4:0] OPcode,// inst[6:2]
23     input wire [2:0] Fun3,// inst[14:23]
24     input wire Fun7,// inst[30]
25     input wire MIO_ready,// CPU wait
26     output reg [2:0] ImmSel,// 立即数选择控制 // need updating
27     output reg ALUSrc_B,// 源操作数2选择
28     output reg [1:0] MemtoReg,// 写回数据选择控制
29     output reg [1:0] Jump,// jal // need updating
30     output reg Branch,// beq
31     output reg BranchN,// need updating
32     output reg RegWrite,// 寄存器写使能
33     output reg MemRW,// 存储器读写使能
34     output reg [3:0] ALU_Control,// ALU控制
35     output reg ill_instr,
36     //output reg ecall,
37     //output reg mret,
38     output reg CPU_MIO// not use
39 );

```

该部分代码展示了模块的输入输出端口。

```
41     wire [3:0] Fun;
42     reg [1:0] ALUop;
43     reg em;
44     //reg [10:0] CPU_ctrl_signals;
45
46
47     assign Fun = {Fun3,Fun7};
```

该部分代码声明了模块内部使用到的中间信号。

```
50     always @ * begin
51       CPU_MIO = MIC_ready;
52       case(OpCode)
53         5'b01100:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b0001000010xxx000;end// 
54         5'b00000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b1011000000001000;end// 
55         5'b00100:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b1001000011001000;end// 
56         5'b11001:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b1101xx1000001x00;end// 
57         5'b01000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b1xx0100000010000;end// 
58         5'b11000:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,em,ill_instr} = 14'b0xx00000101100;end// $b:beg bne bit b
59         5'b1101:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b1111x000xx00000;end// 
60         5'b11011:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b1101xx01xx100x00;end// 
61         5'b11100:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'bxxxxxx000xxxxx010;end// 
62         default:begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,Jump,ALUop,ImmSel,BranchN,em,ill_instr} = 16'b0000000000000001;end // 
63     endcase
64   // 此步骤只能判断出opcode不合法的指令，其余指令需要下面继续判断
```

尤其可以看到该 case 代码增加了对于非法 opcode 的检测。

6.1.2 InterruptDetect

```
23     module InterruptDetect (
24       input clk,
25       input rst,
26       input [31:0] PC_next, // 下一条指令
27       input [31:0] PC_add4,
28       input wire INT, // 外部中断信号
29       input wire mret,
30       input wire ecall,
31       input wire ill_instr, // 非法信号
32       output reg [31:0] PC_certain,
33       output wire [31:0] reg_mepc
34     );
35
36     reg [31:0] mepc; // 储存异常指令的下一条指令
37     reg [31:0] mtvec [0:3];
38   
```

该部分代码声明了该模块的输入输出端口以及使用到的 mepc 寄存器、mtvec 寄存器组。

```
39     initial begin
40       mtvec[0] <= 32'h00000000; // reset
41       mtvec[1] <= 32'h00000004; // ill_instr
42       mtvec[2] <= 32'h00000008; // ecall
43       mtvec[3] <= 32'h0000000c; // INT
44   end
```

该部分代码初始化 mtvec 寄存器组的值，作为相应中断发生时程序需要跳转到的地方。

```

46      // mepc 寄存器更新 - 时序逻辑
47      always @(posedge clk or posedge rst) begin
48          if(rst) begin
49              mepc <= 32'h00000020; // 主程序地址
50          end else if (ill_instr) begin
51              // 发生异常时保存返回地址
52              mepc <= PC_add4;
53          end else if(ecall) begin
54              mepc <= PC_next + 32'h00000004;
55          end else if(INT) begin
56              mepc <= PC_next;
57          end
58      // mret时不更新mepc
59  end

```

该部分使用时序逻辑，做到在中断发生时就即时更新 mepc，实现对于程序中断的退出处理。

```

61      // PC决策 - 组合逻辑，立即响应
62      // 但是其实这些信号判定都是组合逻辑，所以这里时序逻辑应该也没什么
63      always @ (*) begin
64          if(rst) begin
65              PC_certain = mtvec[0]; // 复位
66          end else if (mret) begin
67              PC_certain = mepc; // 异常返回
68          end else if (ill_instr) begin
69              PC_certain = mtvec[1]; // 非法指令
70          end else if (ecall) begin
71              PC_certain = mtvec[2]; // 系统调用
72          end else if (INT) begin
73              PC_certain <= 32'h0000000c; // 外部中断
74          end else begin
75              PC_certain = PC_next; // 正常执行
76          end
77      end
78
79      assign reg_mepc = mepc;
80
81 endmodule

```

该部分代码使用组合逻辑，根据优先级确定出需要传递给程序计算器的 PC 值。

6.2.3 INT_Detect

```

22 module INT_Detect(
23     input clk, // 必须有时钟
24     input rst, // 复位信号
25     input wire in, // 原始输入信号
26     output reg out // 单周期脉冲输出
27 );
28
29     reg in_prev; // 保存上一个时钟周期的输入值
30
31     // 时序逻辑：采样输入信号
32     always @(posedge clk or posedge rst) begin
33         if (rst) begin
34             in_prev <= 1'b0;
35             out <= 1'b0;
36         end else begin
37             in_prev <= in; // 保存当前值，下一周期用
38             // 检测上升沿并产生单周期脉冲
39             if (in && !in_prev) begin
40                 out <= 1'b1; // 上升沿，输出1
41             end else begin
42                 out <= 1'b0; // 其他情况输出0
43             end
44         end
45     end
46 endmodule

```

该部分代码通过时序逻辑时刻监视外部中断信号的有无，在出现外部中断信号时保证一直有外部中断信号传递给 CPU 模块，对外部中断做出相应的处理。

6.1.4 Datapath 改动部分

```
213     InterruptDetect_ID(
214         .clk(clk),
215         .rst(rst),
216         .PC_next(MUX4T1_32_1_o), // 下一条指令
217         .PC_add4(Add_32_0_c),
218         .INT(INT), // 外部中断信号
219         .mret(mret),
220         .ecall(ecall),
221         .ill_instr(ill_instr), // 非法信号
222         .PC_certain(InterruptDetect_D),
223         .reg_mepc(mepc_o)
224     );
```

增加中断检测模块的实例化引用从而正确产生出下一条程序计数值。

6.1.5 SCPU 改动部分

```
121     DataPath_more DataPath_more_0(
122         .ALUSrc_B(SCPU_ctrl_0_ALUSrc_B),
123         .ALU_Control(SCPU_ctrl_0_ALU_Control),
124         .Branch(SCPU_ctrl_0_Branch),
125         .BranchN(SCPU_ctrl_0_BranchN),
126         .Data_in(Data_in),
127         .ImmSel(SCPU_ctrl_0_ImmSel),
128         .Jump(SCPU_ctrl_0_Jump),
129         .MemtoReg(SCPU_ctrl_0_MemToReg),
130         .RegWrite(SCPU_ctrl_0_RegWrite),
131         .clk(clk),
132         .rst(rst),
133         .inst_field(inst_in),
134         .ill_instr(SCPU_ctrl_0_ill_instr),
135         .ecall((inst_in == 32'h00000073)),
136         .mret((inst_in == 32'h30200073)),
137         .INT(INT),
138         .ALU_out(Addr_out),
139         .Data_out(Data_out),
140         .PC_out(PC_out),
```

可以发现 Datapath 增加了 ecall、mret 输入信号，此处我们直接检测指令的 opcode 部分来判断是否发生了相应的中断以简化程序代码的设计。

6.1.6 CSSTE 改动部分

```
189     INT_Detect U12(
190         .clk(U8_Clk_CPU),
191         .rst(U9_rst),
192         .in(U9_SW_OK[15]),
193         .out(U12_out)
194     );
```

增加外部终端检测检测，输出外部中断信号。

6.2 物理测试分析

为避免出现程序在电脑尚未完成烧录之前就已经发生运行，在 program 的过程中我们按住开发板的 reset 键以保证程序不会自己提前运行。

```

RV32I Single Cycle CPU

pc: 00000002c    inst: fe62dae2

x0: 00000000    ra: 00000000    sp: 88888000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000   s11: 00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000000

rs1: 05    rs1_val: 00000000
rs2: 06    rs2_val: 00000000
rd: 15    reg_i_data: 00000000    reg_wen: 0

is_imm: 0    is_auiipc: 0    is_lui: 0    imm: ffffff4
a_val: 00000000    b_val: 00000000    alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000000    cmp_res: 0

is_branch: 0    is_jal: 0    is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0    mem_ren: 0
dmem_o_data: 00000000    dmem_i_data: 00000000    dmem_addr: 00000000

csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000    mcause: 00000000    mepc: 00000020    mtval: 00000000
mtvec: 00000000    mie: 00000000    mip: 00000000

```

程序从 0x00000000 条指令开始执行，此时 mepc 寄存器数值为 0x000000020，为我们专门设计的针对 reset 的处理。实际上 0x00000000 条指令就是跳转到 0x000000020 指令的指令。

程序一直运行直到 0x000000002c 指令，及 Demo 中的非法指令。

```

RV32I Single Cycle CPU

pc: 00000004    inst: 0c40006f

x0: 00000000    ra: 00000000    sp: 88888000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000   s11: 00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000000

rs1: 00    rs1_val: 00000000
rs2: 04    rs2_val: 00000000
rd: 00    reg_i_data: 00000008    reg_wen: 1

is_imm: 1    is_auiipc: 0    is_lui: 0    imm: 000000c4
a_val: 00000000    b_val: 00000000    alu_ctrl: 0    cmp_ctrl: 0
alu_res: 000000c4    cmp_res: 0

is_branch: 0    is_jal: 1    is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0    mem_ren: 0
dmem_o_data: 00000000    dmem_i_data: 00000000    dmem_addr: 000000c4

csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000    mcause: 00000000    mepc: 00000030    mtval: 00000000
mtvec: 00000000    mie: 00000000    mip: 00000000

```

下一条时钟，我们看到程序直接跳转到 0x00000004 条指令执行，并且 mepc 更新为了非法指令的下一条指令地址——0x000000030。

```

RV32I Single Cycle CPU

pc: 000000c8    inst: 00168693

x0: 00000000    ra: 00000000    sp: 88888000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000   s11: 00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000000

rs1: 0d    rs1_val: 00000000
rs2: 01    rs2_val: 00000000
rd: 0d    reg_i_data: 00000001    reg_wen: 1

is_imm: 1    is_auiipc: 0    is_lui: 0    imm: 00000001
a_val: 00000000    b_val: 00000000    alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000001    cmp_res: 0

is_branch: 0    is_jal: 0    is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0    mem_ren: 0
dmem_o_data: 00000000    dmem_i_data: 00000000    dmem_addr: 00000001

csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000    mcause: 00000000    mepc: 00000030    mtval: 00000000
mtvec: 00000000    mie: 00000000    mip: 00000000

```

继续运行，程序跳转到 0x000000c8 条指令，即 Demo 中的非法指令处理部分的第一条指令地址。

```
RV32I Single Cycle CPU

pc: 0000000cc inst: 00168693

x0: 00000000 ra: 00000000 sp: 88888000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000001 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 0d rs1_val: 00000001
rs2: 01 rs2_val: 00000000
rd: 0d reg_i_data: 00000002 reg_wen: 1

is_imm: 1 is_auipc: 0 is_lui: 0 imm: 00000001
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000002 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000000 dmem_addr: 00000002

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000030 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

RV32I Single Cycle CPU

pc: 0000000d0 inst: 30200073

x0: 00000000 ra: 00000000 sp: 88888000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000002 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 00000000
rs2: 02 rs2_val: 88888000
rd: 00 reg_i_data: 77778000 reg_wen: 0

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000300
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 77778000 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000000 dmem_addr: 77778000

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000030 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

此后程序可以正常运行到 0x000000d0 条指令，即 Demo 中的非法指令处理部分的最后一条指令——mret 指令。此指令正是对于中断处理的退出指令。

```

RV32I Single Cycle CPU

pc: 000000030 inst: 00832183

x0: 00000000 ra: 00000000 sp: 88888000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000002 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 06 rs1_val: 00000000
rs2: 08 rs2_val: 00000000
rd: 03 reg_i_data: 80000000 reg_wen: 1

is_imm: 1 is_auipc: 0 is_lui: 0 imm: 00000008
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000008 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 80000000 dmem_i_data: 00000000 dmem_addr: 00000008

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000030 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

之后我们的程序直接跳转到第 0x00000030 条指令，即之前非法指令出现时的下一条指令。

之后程序一直继续运行，正常运行指令中的各种类型的指令。直至遇到第 0x00000074 条指令——Demo 中的 ecall 指令。

```

RV32I Single Cycle CPU

pc: 000000074 inst: 000000073

x0: 00000000 ra: 80000000 sp: 88888000 gp: 80000000 tp: 00000000
t0: 80000000 t1: 88888000 t2: 80000000 s0: 00000000 s1: 00000000
a0: 80000000 a1: 00000000 a2: 00000000 a3: 00000002 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 80000000
rs2: 00 rs2_val: 80000000
rd: 00 reg_i_data: 00000000 reg_wen: 0

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 80000000 dmem_i_data: 00000000 dmem_addr: 00000000

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000030 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

```

RV32I Single Cycle CPU

pc: 00000008 inst: 0d80006f

x0: 00000000 ra: 80000000 sp: 88888000 gp: 80000000 tp: 00000000
t0: 80000000 t1: 88888000 t2: 80000000 s0: 00000000 s1: 00000000
a0: 80000000 a1: 00000000 a2: 00000000 a3: 00000002 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 00 rs1_val: 80000000
rs2: 18 rs2_val: 00000000
rd: 00 reg_i_data: 0000000c reg_wen: 1

is_imm: 1 is_auipc: 0 is_lui: 0 imm: 000000d8
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 000000d8 cmp_res: 0

is_branch: 0 is_jal: 1 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 00000000 dmem_i_data: 00000000 dmem_addr: 000000d8

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 00000078 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000

```

可以看到遇到 ecall 指令之后，我们的程序跳转到了 0x00000008 指令，即设

计的 ecall 中断处理指令，同时 mepc 的值被更新为了 0x00000078——ecall 指令的下一条指令地址。

```
RV32I Single Cycle CPU

pc: 0000000e0  inst: 00128793

x0: 00000000  ra: 00000000  sp: 88888000  gp: 00000000  tp: 00000000
t0: 80000000  t1: 88888000  t2: 00000000  s0: 00000000  s1: 00000000
a0: 00000000  a1: 00000000  a2: 00000000  a3: 00000002  a4: 00000000
a5: 00000000  a6: 00000000  a7: 00000000  s2: 00000000  s3: 00000000
s4: 00000000  s5: 00000000  s6: 00000000  s7: 00000000  s8: 00000000
s9: 00000000  s10: 00000000  s11: 00000000  t3: 00000000  t4: 00000000
t5: 00000000  t6: 00000000

rs1: 05  rs1_val: 80000000
rs2: 01  rs2_val: 80000000
rd: 0f  reg_i_data: 80000001  reg_wen: 1

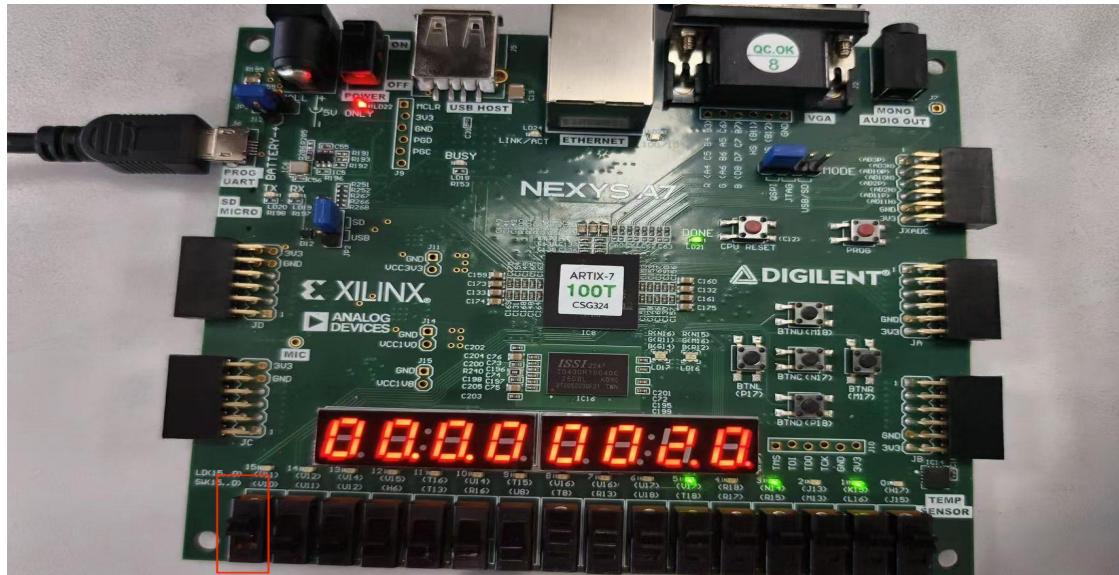
is_imm: 1  is_auiipc: 0  is_lui: 0  imm: 00000001
a_val: 00000000  b_val: 00000000  alu_ctrl: 0  cmp_ctrl: 0
alu_res: 80000001  cmp_res: 0

is_branch: 0  is_jal: 0  is_jalr: 0
do_branch: 0  pc_branch: 00000000

mem_wen: 0  mem_ren: 0
dmem_o_data: 80000000  dmem_i_data: 00000000  dmem_addr: B0000001

csr_wen: 0  csr_ind: 000  csr_ctrl: 0  csr_r_data: 00000000
mstatus: 00000000  mcause: 00000000  mepc: 00000078  mtval: 00000000
mtvec: 00000000  mie: 00000000  mip: 00000000
```

接下来程序直接跳转到 0x0000000e0 指令，即 ecall 中断的具体处理指令。



此时我们将 SW[15]拨至高电平作为外部中断信号。

```
RV32I Single Cycle CPU

pc: 0000000c  inst: 0c80006f

x0: 00000000  ra: 00000000  sp: 00000000  gp: 00000000  tp: 00000000
t0: 00000000  t1: 00000000  t2: 00000000  s0: 00000000  s1: 00000000
a0: 00000000  a1: 00000000  a2: 00000000  a3: 00000000  a4: 00000000
a5: 00000000  a6: 00000000  a7: 00000000  s2: 00000000  s3: 00000000
s4: 00000000  s5: 00000000  s6: 00000000  s7: 00000000  s8: 00000000
s9: 00000000  s10: 00000000  s11: 00000000  t3: 00000000  t4: 00000000
t5: 00000000  t6: 00000000

rs1: 00  rs1_val: 00000000
rs2: 08  rs2_val: 00000000
rd: 00  reg_i_data: 00000010  reg_wen: 1

is_imm: 1  is_auiipc: 0  is_lui: 0  imm: 000000c8
a_val: 00000000  b_val: 00000000  alu_ctrl: 0  cmp_ctrl: 0
alu_res: 000000c8  cmp_res: 0

is_branch: 0  is_jal: 1  is_jalr: 0
do_branch: 0  pc_branch: 00000000

mem_wen: 0  mem_ren: 0
dmem_o_data: 00000000  dmem_i_data: 00000000  dmem_addr: 000000c8

csr_wen: 0  csr_ind: 000  csr_ctrl: 0  csr_r_data: 00000000
mstatus: 00000000  mcause: 00000000  mepc: 00000024  mtval: 00000000
mtvec: 00000000  mie: 00000000  mip: 00000000
```

可以看到我们的程序正确地跳转到了第 0x0000000c 条指令，即外部中断的第一条处理指令；同时寄存器的数值被更新为了当前发生外部中断的指令的下一条指令的地址——0x00000024。

```
RV32I Single Cycle CPU

pc: 000000d4 inst: 40c70733

x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 0e rs1_val: 00000000
rs2: 0c rs2_val: 00000000
rd: 0e reg_i_data: 00000000 reg_wen: 1

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 0000040e
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 80000000 dmem_i_data: 00000000 dmem_addr: 00000000

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 000000d4 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

此后程序正确跳转到 0x000000d4 条指令，即 Demo 中外部中断的主体处理指令。

```
RV32I Single Cycle CPU

pc: 000000d8 inst: 40c70733

x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000

rs1: 0e rs1_val: 00000000
rs2: 0c rs2_val: 00000000
rd: 0e reg_i_data: 00000000 reg_wen: 1

is_imm: 0 is_auipc: 0 is_lui: 0 imm: 0000040e
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000000 cmp_res: 0

is_branch: 0 is_jal: 0 is_jalr: 0
do_branch: 0 pc_branch: 00000000

mem_wen: 0 mem_ren: 0
dmem_o_data: 80000000 dmem_i_data: 00000000 dmem_addr: 00000000

csr_wen: 0 csr_ind: 000 csr_ctrl: 0 csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000 mepc: 000000d4 mtval: 00000000
mtvec: 00000000 mie: 00000000 mip: 00000000
```

之后程序一直运行直至结束。

综上我们可以发现，设计的程序对于 reset、非法指令、ecall 指令、mret 指令和外部中断均能做出正确的处理。说明我设计的 CPU 中断处理程序可以正确运行。

七、讨论、心得

本次实验为拓展 CPU 中断程序设计实验，在实验过程中我们基于课堂知识及理解自行在前面四个实验的基础上为 CPU 模块增加了外部中断处理模块，实现了更为全面的 CPU 功能。

这是我耗时最久、感觉最难的一个实验，确实是上课时针对这方面的知识理解较为薄弱。尤其是在时序逻辑、组合逻辑的选择上，我一开始并没有注意到如此之大的差异。但是事实上通过时序逻辑与组合逻辑的灵活组合，我们可以实现

出更加精确、负责的操作。这个过程也能让我们对于课堂理论知识、计算机处理器的运行过程、原理获得更加深刻的理解。

程序设计实验是课堂知识最好的巩固帮手，同时也是我们课堂知识掌握最好的检测方法。没有实验的理论永远成为不了技术，实验的作用是无比大的！

记得《数字逻辑设计》当时立下的理解计算机运行原理的 flag，但在《计算机组成》课堂中不断深入的学习，我不断深入地感叹计算机实现的神奇！现在的我对于计算机的运行已经有了初步的了解，相信随着更多的课程学习，这条路还可以走的更远！