

# 目录

<b>1 控制器设计实验</b>	<b>2</b>
1.1 需求 . . . . .	2
1.2 主要实现思路 . . . . .	2
1.3 电路实现 . . . . .	3
1.4 仿真测试 . . . . .	3
<b>2 单周期 CPU 设计实验</b>	<b>4</b>
2.1 需求 . . . . .	4
2.2 电路实现 . . . . .	4
2.3 测试 . . . . .	5
<b>3 用累加和程序验证 CPU 设计</b>	<b>6</b>
3.1 测试步骤 . . . . .	6
3.2 测试 . . . . .	7
<b>4 用冒泡排序程序进行 CPU 设计验证</b>	<b>8</b>
4.1 介绍-冒泡排序 . . . . .	8
4.2 结果验证 . . . . .	9
<b>5 官方测试集测试</b>	<b>10</b>
<b>6 计算机系统基础 PA 程序测试</b>	<b>11</b>
6.1 具体方法 . . . . .	11
6.2 测试结果 . . . . .	12
<b>7 思考题</b>	<b>13</b>
7.1 如何在单 CPU 上实现多任务处理，例如同时执行计算累加和与数据排序两个程序，阐述思路。 . . . . .	13
7.2 在 CPU 的基础上，如何实现键盘输入、TTY 输出部件等输入输出设备的数据访问，构建完整的计算机系统。 . . . . .	13
7.3 阐述如何在单周期 CPU 基础上实现多周期 CPU 和流水线 CPU? . . . . .	13

# 1. 控制器设计实验

## 1.1 需求

根据 RV32I 指令控制信号列表，由操作码 opcode、功能码 func3 和功能码 func7 生成 ExtOp, RegWr, ALUASsrc, ALUBSrc, ALUctr, Branch, MemtoReg, MemWr, MemOp

## 1.2 主要实现思路

主要通过大量人力操作 () 列出真值表来生成每一个控制信号。

ALUctr.	MemoP	ALU con	
op 0110111	func3 X	func7 05?	
op 0110111	func3 X	X	
0010011	101	1	
0110011	000	1	
0110011	101	1.	
6 & 0 & 1 & 8 & 4			
0110110	X	X	
0010011	101	1	
0110011	000	1	
0110011	101	1	
6 & 4 & 3 & 1 & 0			
5 2	11 00 10 00	00	
			2
			xs & xi & xi & 4 & 3 & 2
			0100011 1100011

1

图 1

### 1.3 电路实现

#### 电路的完整实现

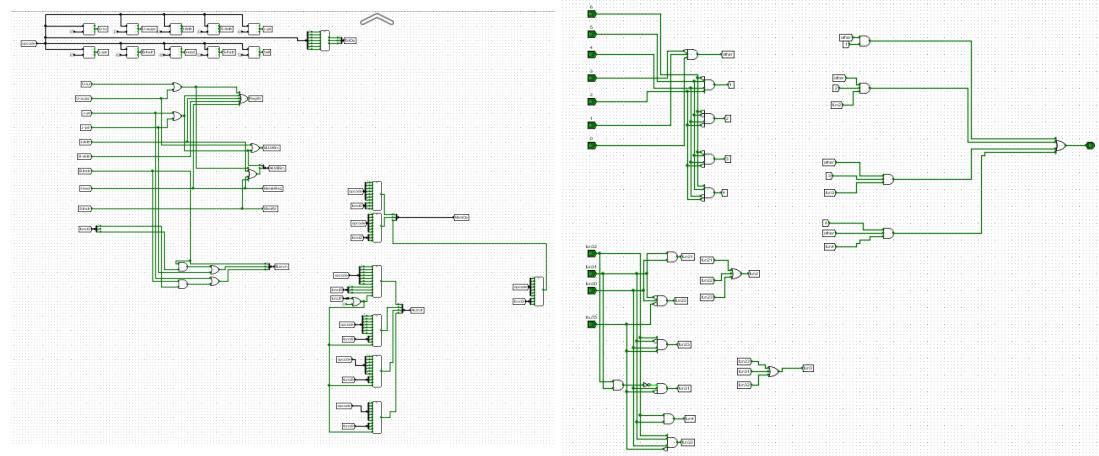


图 2

### 1.4 仿真测试

由于手动仿真测试较为繁琐，并且发现 OJ 平台对于这 37 条指令进行了全测试，故认为能通过仿真测试。

## 2. 单周期 CPU 设计实验

### 2.1 需求

在实验五数据通路部件的基础上，加上控制器部件就能连接成单周期处理器。加载 RV32I 测试程序，观测实验结果，验证单周期处理器的功能。

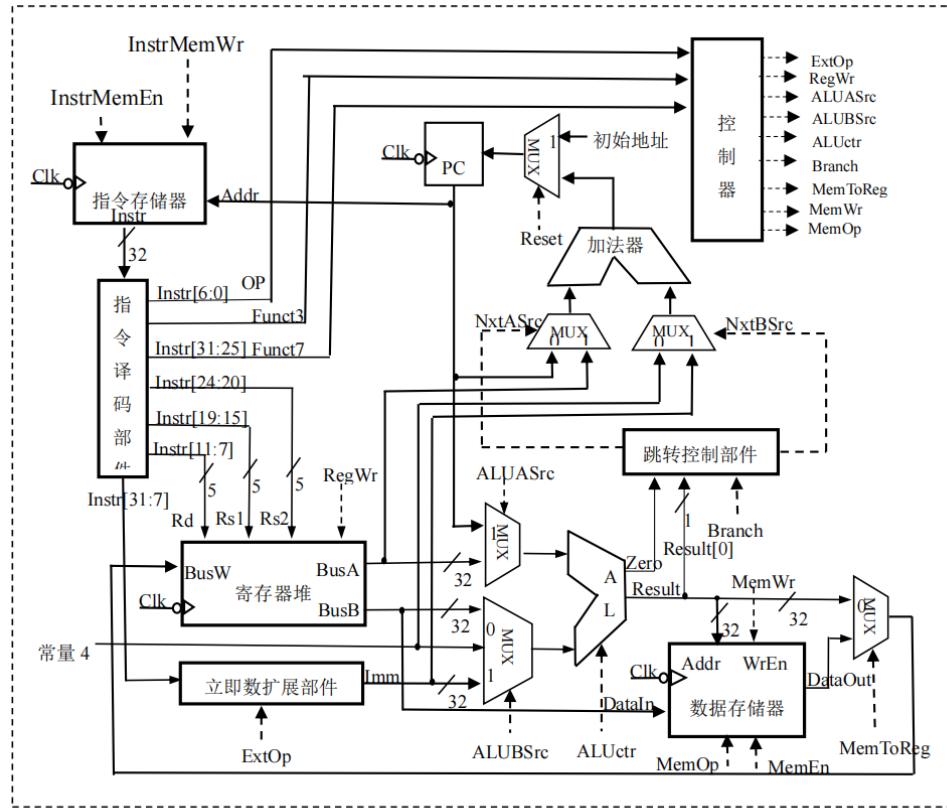
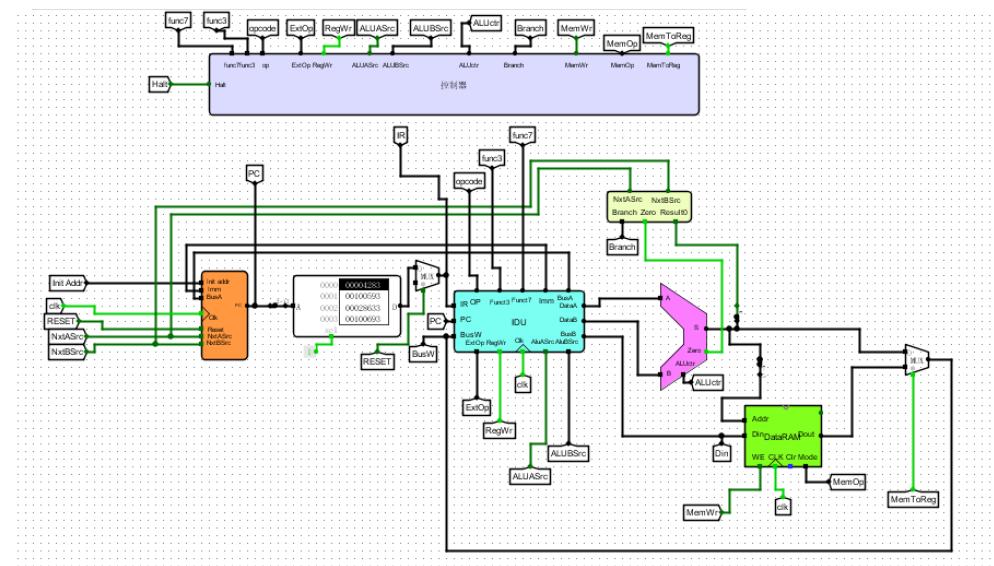


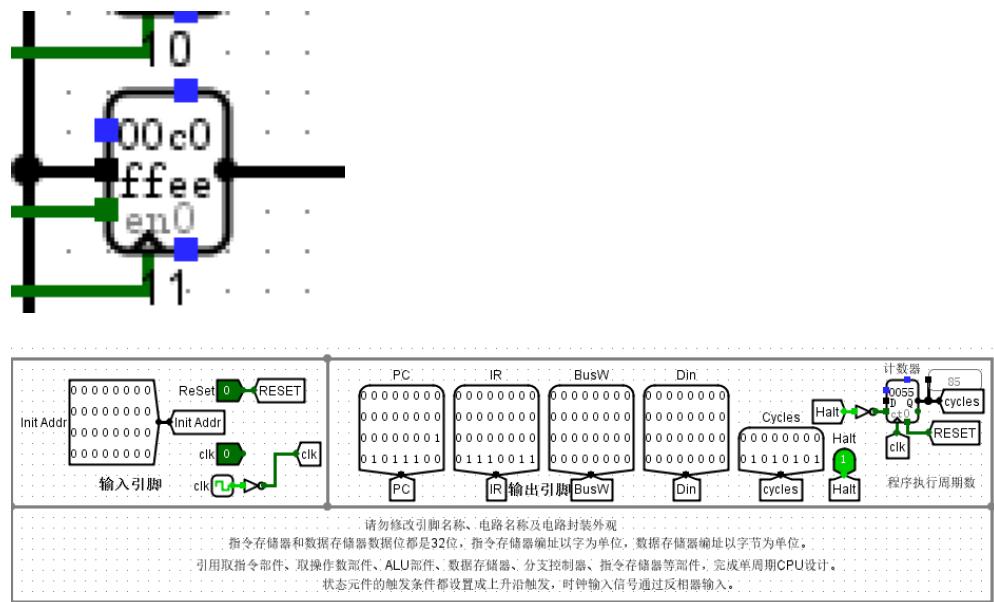
图 6.4 单周期 CPU 原理图

### 2.2 电路实现



## 2.3 测试

在指令寄存器中加载 lab6.2.hex 文件，按照时钟单步执行，观测实验结果，写出寄存器堆和存储器里的数据，验证电路功能。如果指令执行结果验证通过，则在寄存器堆的 10 号寄存器中写入“00c0ffee”；否则在 10 号寄存器写入“deaddead”，并在 3 号寄存器中写入当前测试指令的序号。



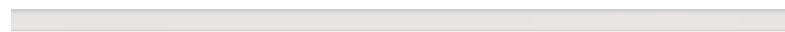
测试通过。

### 3. 用累加和程序验证 CPU 设计

#### 3.1 测试步骤

##### 1) 编写汇编语言源程序

```
main:
    lw    a1,0(x0)      # R[a1]:=n,R[a1]<- Mem[0], 读取参数 n
    beq a1,x0,fail    # if n=0 goto fail
    ori a2,x0,1        # R[a2].i.=1
    xor a3,a3,a3       # R[a3].S.=0
loop:
    add a3, a3, a2      # R[a3]=R[a3]+R[a2]
    beq a2, a1, finish  # if R[a2]=n goto finish
    addi a2, a2, 1       # R[a2]=R[a2]+1
    jal x0, loop        #
finish:
    sw a3, 4(x0)       # Store S to Mem[4],Mem[4]<-R[a3]
pass:
```



```
lui    a0,0xc10
addi   a0,a0,-18  # R[a0]=0x00c0ffee
ecall
# 结束执行
fail:
lui    a0,0xdeade
addi   a0,a0,-339 # R[a0]=0xdeaddead
ecall
# 结束执行
```

##### 2) 将汇编语言源程序转换为机器代码

##### 3) 对机器代码进行调试运行

Text Segment				Registers		
Line	Address	Code	Basic	Name	Number	Value
1	0x00000000	0x00000000 lw a1,0(x0)	5:    lw    a1,0(x0)      # R[a1]:=n,R[a1]<- Mem[0], 读取参数 n	r0	0	0x00000000
2	0x00000004	0x00000000 beq a1,x0,fail	6:    beq a1,x0,fail    # if n=0 goto fail	r1	1	0x00000000
3	0x00000008	0x00000000 ori a2,x0,1	7:    ori a2,x0,1        # R[a2].i.=1	r2	2	0x00000001
4	0x0000000C	0x00000000 xor a3,a3,a3	8:    xor a3,a3,a3       # R[a3].S.=0	r3	3	0x00000000
5	0x00000010	0x00000000 add a3, a3, a2	9:    add a3, a3, a2      # R[a3]=R[a3]+R[a2]	r4	4	0x00000000
6	0x00000014	0x00000000 beq a2, a1, finish	10:   beq a2, a1, finish  # if R[a2]=n goto finish	r5	5	0x00000000
7	0x00000018	0x00000000 addi a2, a2, 1	11:   addi a2, a2, 1       # R[a2]=R[a2]+1	r6	6	0x00000001
8	0x0000001C	0x00000000 jal x0, loop	12:   jal x0, loop        #	r7	7	0x00000000
9	0x00000020	0x00000000 sw a3, 4(x0)	13:   sw a3, 4(x0)       # Store S to Mem[4],Mem[4]<-R[a3]	r8	8	0x00000000
10	0x00000024	0x00000000 addi a0,a0,-18	14:   addi a0,a0,-18     # R[a0]=0x00c0ffee	r9	9	0x00000000
11	0x00000028	0x00000000 lui a0,0xc10	15:   lui a0,0xc10      # R[a0]=0x00c0ffee	r10	10	0x00000000
12	0x0000002C	0x00000000 addi a0,a0,-339	16:   addi a0,a0,-339   # R[a0]=0xdeaddead	r11	11	0x00000000
13	0x00000030	0x00000000 ecall	17:   ecall	r12	12	0x00000000
14	0x00000034	0x00000000 lui a0,0xdeade	18:   lui a0,0xdeade    # R[a0]=0xdeaddead	r13	13	0x00000000
15	0x00000038	0x00000000 addi a0,a0,-18	19:   addi a0,a0,-18     # R[a0]=0xdeaddead	r14	14	0x00000000
16	0x00000040	0x00000000 ecall	20:   ecall	r15	15	0x00000000
17	0x00000044	0x00000000 lui a0,0xdeade	21:   lui a0,0xdeade    # R[a0]=0xdeaddead	r16	16	0x00000000
18	0x00000048	0x00000000 addi a0,a0,-18	22:   addi a0,a0,-18     # R[a0]=0xdeaddead	r17	17	0x00000000
19	0x0000004C	0x00000000 ecall	23:   ecall	r18	18	0x00000000
20	0x00000050	0x00000000 lui a0,0xdeade	24:   lui a0,0xdeade    # R[a0]=0xdeaddead	r19	19	0x00000000
21	0x00000054	0x00000000 addi a0,a0,-18	25:   addi a0,a0,-18     # R[a0]=0xdeaddead	r20	20	0x00000000
22	0x00000058	0x00000000 ecall	26:   ecall	r21	21	0x00000000
23	0x00000060	0x00000000 lui a0,0xdeade	27:   lui a0,0xdeade    # R[a0]=0xdeaddead	r22	22	0x00000000
24	0x00000064	0x00000000 addi a0,a0,-18	28:   addi a0,a0,-18     # R[a0]=0xdeaddead	r23	23	0x00000000

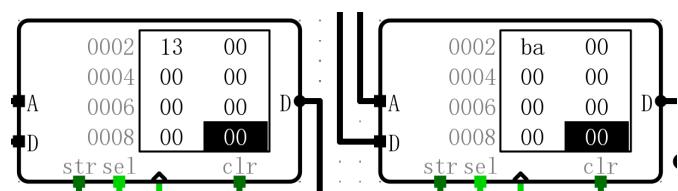
##### 4) 导出机器代码

### 3.2 测试

程序终止结束状态

000b	00000073
000c	deade537
'A	000dead50513D
000e	00000073
sel	

程序结束后，进入数据储存器查看结果



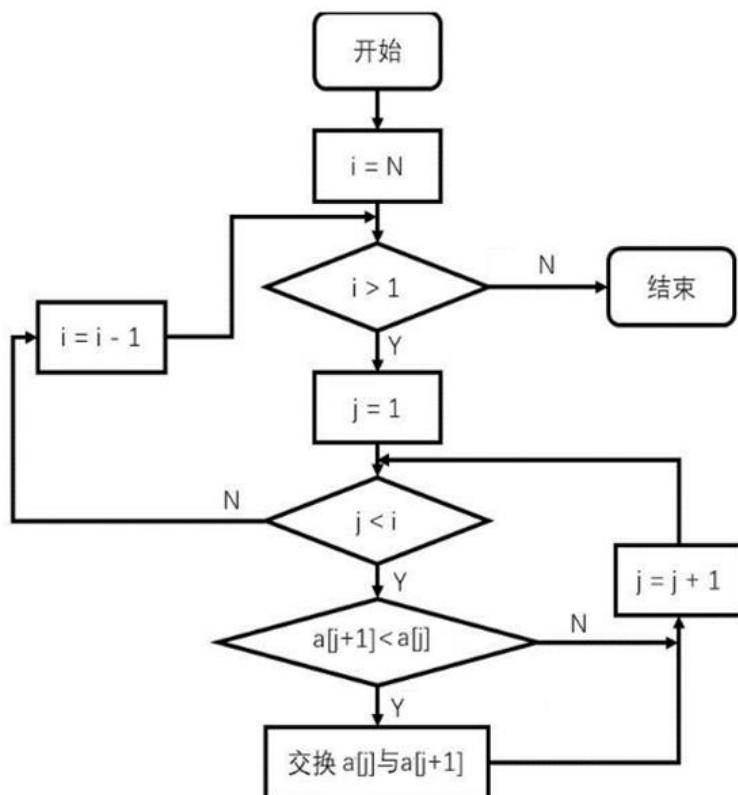
0x13ba(5050) 结果正确

并且，程序可以通过 OJ 平台测试。

## 4. 用冒泡排序程序进行 CPU 设计验证

### 4.1 介绍-冒泡排序

采用冒泡排序对有限数据按照从小到大的顺序排列。冒泡排序算法要点是：对所有相邻记录的关键字值进行比较，如果是逆序 ( $a[j] > a[j+1]$ )，则将其交换，最终达到有序化。其算法基本思想如下：首先，将整个待排序的记录序列划分成有序区和无序区，初始状态有序区为空，无序区包括所有待排序的记录。然后，对无序区从前向后依次将相邻记录的关键字进行比较，若逆序将其交换，从而使得关键字值小的记录向上“冒”（左移），关键字值大的记录向下“落”（右移）。每经过一趟冒泡排序，都使无序区（左边区域）中关键字值最大的记录进入有序区（右边区域），对于由  $n$  个记录组成的记录序列，最多经过  $n-1$  趟冒泡排序，就可以将这  $n$  个记录按关键字从小到大的顺序排列。



## 冒泡排序算法参考代码

冒泡排序算法参考代码如下：

```
for (i=n; i>1; i--) {  
    for (j=1; j<=i-1; j++) {  
        if (a[j]>a[j+1]) {  
            temp=a[j];  
            a[j]=a[j+1];  
            a[j+1]=temp;  
        }  
    }  
}
```

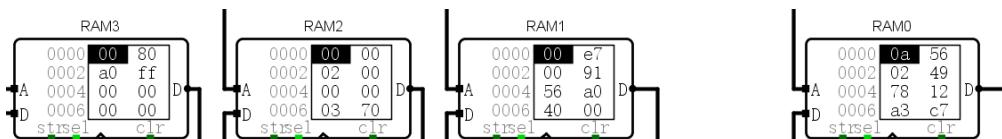
## 冒泡排序的汇编语言

```
#冒泡排序算法  
.text  
main:  
    lw    t0,0(x0) # R[t0]<-Mem[0],t0 保存排序数量 n,待排序的数字个数 n 存在 0x00 处  
    addi a1,x0,1   # a1, 保存常量 1  
    add   a2,t0,x0 # a2, 保存 i, 初始值为 i=n  
L1:  
    addi a3,x0,1   # a3, 保存 j, 初始值为 j=1  
L2:  
    slli a4,a3,2   # a4 保存 a[j]地址  
    lw    a6,0(a4) # 读取第 j 个元素  
    lw    a7,4(a4) # 读取第 j+1 个元素  
    bge  a7,a6,L4  # a[j]>=a[j+1] 跳转,按照带符号数比较  
    sw    a7,0(a4) # 交换存储  
    sw    a6,4(a4) # 交换存储  
L4:  
    addi a3,a3,1   # j=j+1  
    bltu a3,a2,L2  # if j<i then 循环, 序号按照无符号数比较  
L3:  
    sub   a2,a2,a1 # i--  
    bne   a2,a1,L1 # if i>1 then 循环 else 则结束  
    ecall          # 结束执行
```

## 冒泡排序编译仿真执行结果

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000000	0x00000002	0xff009149	0x000005678	0x0000012	0x000340a3	0x007000e7	0x000800ad8
0x00000020	0x00d00205	0x07001234	0x0800e756	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

## 4.2 结果验证



与预期一致，并能通过 OJ 测试。

## 5. 官方测试集测试

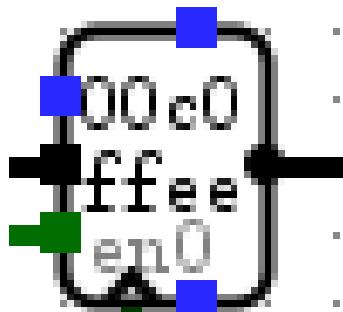
运行以下命令

- 1) \$ git clone https://github.com/riscv/riscv-tests
- 2) \$ cd riscv-tests
- 3) \$ git submodule update --init --recursive
- 4) \$ autoconf
- 5) \$ ./configure --with-xlen=32
- 6) \$ make isa

加载指令测试代码后，选择连续时钟信号，执行程序。

依次加载所有的测试程序，通过官方测试集的验证。

运行结果



测试成功

# 6. 计算机系统基础 PA 程序测试

## 6.1 具体方法

1、准备交叉编译环境。在 Ubuntu 下运行下列命令：

- 1) apt-get install g++-riscv64-linux-gnu
- 2) git clone -b digital <https://github.com/NJU-ProjectN/abstract-machine>
- 3) git clone -b ics2021 <https://github.com/NJU-ProjectN/am-kernels>
- 4) apt install python-is-python3

2、修改文件权限。在 sudo 权限下修改以下文件：

```
--- /usr/riscv64-linux-gnu/include/gnu/stubs.h
+++ /usr/riscv64-linux-gnu/include/gnu/stubs.h
@@@ -5,5 +5,5 @@
#include <bits/wordsize.h>
#if __WORDSIZE == 32 && defined __riscv_float_abi_soft
#ifndef include <gnu/stubs-ilp32.h>
+//# include <gnu/stubs-ilp32.h>
```

```
#endif
```

3、设置环境参数。在 Ubuntu 下执行下列命令：

```
cd ~，进入用户目录，显示 abstract-machine 路径。
```

```
export AM_HOME=~pwd'/abstract-machine，使得 AM_HOME 和 abstract-machine 目录的路径一致。
```

```
cd am-kernels/tests/cpu-tests，在 tests 子目录中找到需要编译测试的 C 程序，如 bubble-sort.c 文件，可根据需要进行修改编辑。
```

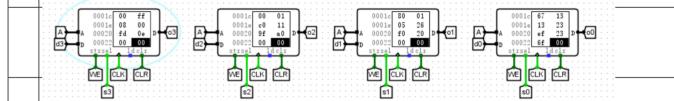
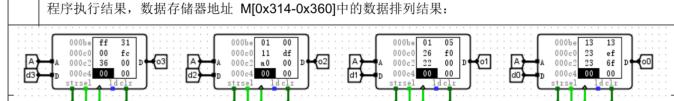
4、生成测试文件。在 am-kernels/tests/cpu-tests 目录下执行下列命令：

```
make ARCH=riscv32-npc ALL=bubble-sort
```

则生成可用于 Logisim 下 CPU 测试的指令镜像文件 bubble-sort-riscv32-npc.bin-logisim-inst.txt 和 4 个按字节分开的数据镜像文件 bubble-sort-riscv32-npc.bin-logisim-data0.txt-bubble-sort-riscv32-npc.bin-logisim-data3.txt。

## 6.2 测试结果

可以通过线下验收。

C-Test C 语言测试程序执行结果	
1	b-sort.c 指令存储器加载: b-sort-riscv32-npc.bin-logisim-inst.txt. 数据存储器分别加载: b-sort-riscv32-npc.bin-logisim-data0.txt ~ b-sort-riscv32-npc.bin-logisim-data3.txt
程序执行周期数:	
程序执行结果, 数据存储器地址 M[0x90-0xDC]中的数据排列结果:	
	
2	q-sort.c 指令存储器加载: q-sort-riscv32-npc.bin-logisim-inst.txt. 数据存储器分别加载: q-sort-riscv32-npc.bin-logisim-data0.txt ~ q-sort-riscv32-npc.bin-logisim-data3.txt
程序执行周期数:	
程序执行结果, 数据存储器地址 M[0x314-0x360]中的数据排列结果:	
	

序号	测试指令	x1 寄存器输出值	x3 寄存器输出值	x10 寄存器输出值	时钟周期数
1	add	00000000/0	0026	00Coffee	458
2	addi	0121	0019	00Coffee	235
3	and	11111111	001b	00Coffee	478
4	andi	00ff00ff	000e	00Coffee	191
5	auipc	0000	0003	12	52
6	beq	0003	0015	1	280
7	bge	0003	0018	1	302
8	bgeu	0003	0018	1	307
9	blt	0003	0015	1	284
10	bltu	0003	0018	1	309
11	bne	0003	0018	1	284
12	jal	0003	0013		48
13	jalr	0000	0007	1	108

## 7. 思考题

### 7.1 如何在单 CPU 上实现多任务处理，例如同时执行计算累加和与数据排序两个程序，阐述思路。

可以采用任务切换的方式进行。在单周期 CPU 上，可以采用一个简单的任务列表，每个任务保存其程序计数器、寄存器状态和内存状态等信息。定义一个时钟中断，定期触发任务切换（比如每执行几条指令后触发一次任务切换）。在时钟中断处理程序中，保存当前任务的状态（如 PC、寄存器值）到任务列表中，然后从任务列表中选择下一个任务的状态进行恢复。当中断发生时，当前任务的 PC、寄存器值需要保存到对应的任务控制块中。需要恢复任务状态时，从任务列表中选择下一个任务，将其 PC 和寄存器值恢复到 CPU 中。

以同时计算累加和与数据排序为例，在初始化时，将计算累加和数据排序的任务分别加载到不同的任务控制块中，并初始化任务列表，然后在程序执行过程中通过时钟中断进行定期切换，使得计算累加和数据排序任务交替执行。

### 7.2 在 CPU 的基础上，如何实现键盘输入、TTY 输出部件等输入输出设备的数据访问，构建完整的计算机系统。

通过在 CPU 中增加存储器地址寄存器（MAR，存储当前访问的内存地址）和存储器数据寄存器（MDR，存储当前访问的内存数据），并使用内存映射 I/O 技术（将输入输出缓冲区映射到 CPU 的内存地址空间，使得 CPU 可以通过读写特定的内存地址来访问这些缓冲区），建立 CPU 和输入输出设备之间的联系。需要修改 CPU 控制器，扩展指令集，增加 IN 指令：从指定的 I/O 地址读取数据到 MDR，然后将数据传送到指定的寄存器；OUT 指令：从指定的寄存器读取数据到 MDR，然后将数据写入指定的 I/O 地址。

键盘输入的数据被转换为二进制存储到缓冲区，CPU 通过 IN 指令读取缓冲区中的数据。TTY 输出数据则通过 OUT 指令将 CPU 寄存器中的数据写入缓冲区，再由 TTY 控制器输出到显示器。

### 7.3 阐述如何在单周期 CPU 基础上实现多周期 CPU 和流水线 CPU？

#### 在单周期 CPU 基础上实现多周期 CPU

划分指令执行步骤，将指令划分为取指令，指令译码，执行，内存访问，写回五个阶段。

增加状态机控制器：使用一个状态机控制器，根据当前指令和状态，控制每个周期执行的步骤。状态机根据时钟信号转移到下一个状态。

修改控制逻辑：每个指令的操作被分解到不同的时钟周期中，通过状态机控制。在每个时钟周期执行不同的操作。

修改寄存器和数据通路：多周期 CPU 需要多个寄存器来保存每个步骤的中间结果。例如，IR（指令寄存器）等。

#### 在单周期 CPU 基础上实现流水线 CPU

分解指令执行阶段：取指令，指令译码，执行，内存访问，写回五个流水段

增加流水线寄存器：在每个阶段之间设置流水线寄存器，用于保存每个指令在不同阶段的中间结果。例如：IF/ID 寄存器、ID/EX 寄存器、EX/MEM 寄存器、MEM/WB 寄存器。

修改数据通路和控制逻辑：修改数据通路以支持流水线操作。

添加控制逻辑以处理数据冒险和控制冒险。