实验 6：单周期 CPU 设计与测试

一、实验目的

1. 分析单周期 CPU 的控制信号，掌握 RV32I 控制器的设计方法。
2. 掌握 RISC-V 汇编语言程序的基本设计方法。
3. 运用 RARS 编译、生成二进制可执行文件。
4. 加载可执行文件、验证 CPU 设计。
5. 理解汇编语言程序与机器语言代码之间的对应关系。

二、实验环境

Logisim：https://github.com/Logisim-Ita/Logisim

RISC-V 模拟器工具 RARS：https://github.com/thethirdone/rars

三、实验内容

CPU中控制指令执行的部件是控制器。控制器输入的是指令操作码op和功能码，输出的是控制信号。控制器的主要设计步骤如下。

(1)根据每条指令的功能，分析控制信号的取值，并在表中列出。

(2)根据列出的指令和控制信号之间的关系，写出每个控制信号的逻辑表达式。

(3)实现取指令部件，设计时序信号，接连模块，实现CPU的综合。

在实现 CPU 的过程中需要对每一个环节进行详细的测试才能够保证系统整体的可靠性。

1、控制器设计实验

整体模块设计：

RV32I 指令集中包含 47 条基础指令，涵盖了整数运算、存储器访问、控制转移和系统控制几个大类。 本次实验中需要实现除了系统控制类的 10 条指令外的 37 条指令，分为整数运算指令（21 条）、控制转移指 令（8 条）和存储器访问指令（8 条）。信号表：

表 6.1 RV32I 指令控制信号列表

指令	类型	op[6:0]	func3	func7[5]	ExtOp	RegWr	ALUASrc	ALUBSrc	ALUctr
lui	U	0110111	×	×	001	1	×	10	1111
auipc	U	0010111	×	×	001	1	1	10	0000
addi	I	0010011	000	×	000	1	0	10	0000
slti	I	0010011	010	×	000	1	0	10	0010
sltiu	I	0010011	011	×	000	1	0	10	0011
xori	I	0010011	100	×	000	1	0	10	0100
ori	I	0010011	110	×	000	1	0	10	0110
andi	I	0010011	111	×	000	1	0	10	0111
slli	I	0010011	001	0	000	1	0	10	0001
srli	I	0010011	101	0	000	1	0	10	0101
srai	I	0010011	101	1	000	1	0	10	1101
add	R	0110011	000	0	×	1	0	00	0000
sub	R	0110011	000	1	×	1	0	00	1000
sll	R	0110011	001	0	×	1	0	00	0001
slt	R	0110011	010	0	×	1	0	00	0010
sltu	R	0110011	011	0	×	1	0	00	0011

xor	R	0110011	100	0	×	1	0	00	0100
srl	R	0110011	101	0	×	1	0	00	0101
sra	R	0110011	101	1	×	1	0	00	1101
or	R	0110011	110	0	×	1	0	00	0110
and	R	0110011	111	0	×	1	0	00	0111
jal	J	1101111	×	×	100	1	1	01	0000
jalr	I	1100111	000	×	000	1	1	01	0000
beq	B	1100011	000	×	011	0	0	00	0010
bne	B	1100011	001	×	011	0	0	00	0010
blt	B	1100011	100	×	011	0	0	00	0010
bge	B	1100011	101	×	011	0	0	00	0010
bltu	B	1100011	110	×	011	0	0	00	0011
bgeu	B	1100011	111	×	011	0	0	00	0011
lb	I	0000011	000	×	000	1	0	10	0000
lh	I	0000011	001	×	000	1	0	10	0000
lw	I	0000011	010	×	000	1	0	10	0000
lbu	I	0000011	100	×	000	1	0	10	0000
lhu	I	0000011	101	×	000	1	0	10	0000
sb	S	0100011	000	×	010	0	0	10	0000
sh	S	0100011	001	×	010	0	0	10	0000
sw	S	0100011	010	×	010	0	0	10	0000

表 6.1 RV32I 指令控制信号列表（续）

指令	类型	op[6:0]	func3	func7[5]	Branch	MemtoReg	MemWr	MemOp
lui	U	0110111	×	×	000	0	0	×
auipc	U	0010111	×	×	000	0	0	×
addi	I	0010011	000	×	000	0	0	×
slti	I	0010011	010	×	000	0	0	×
sltiu	I	0010011	011	×	000	0	0	×
xori	I	0010011	100	×	000	0	0	×
ori	I	0010011	110	×	000	0	0	×
andi	I	0010011	111	×	000	0	0	×
slli	I	0010011	001	0	000	0	0	×
srli	I	0010011	101	0	000	0	0	×
srai	I	0010011	101	1	000	0	0	×
add	R	0110011	000	0	000	0	0	×
sub	R	0110011	000	1	000	0	0	×
sll	R	0110011	001	0	000	0	0	×
slt	R	0110011	010	0	000	0	0	×
sltu	R	0110011	011	0	000	0	0	×

xor	R	0110011	100	0	000	0	0	×
srl	R	0110011	101	0	000	0	0	×
sra	R	0110011	101	1	000	0	0	×
or	R	0110011	110	0	000	0	0	×
and	R	0110011	111	0	000	0	0	×
jal	J	1101111	×	×	001	0	0	×
jalr	I	1100111	000	×	010	0	0	×
beq	B	1100011	000	×	100	×	0	×
bne	B	1100011	001	×	101	×	0	×
blt	B	1100011	100	×	110	×	0	×
bge	B	1100011	101	×	111	×	0	×
bltu	B	1100011	110	×	110	×	0	×
bgeu	B	1100011	111	×	111	×	0	×
lb	I	0000011	000	×	000	1	0	101
lh	I	0000011	001	×	000	1	0	110
lw	I	0000011	010	×	000	1	0	000
lbu	I	0000011	100	×	000	1	0	001
lhu	I	0000011	101	×	000	1	0	010
sb	S	0100011	000	×	000	×	1	101
sh	S	0100011	001	×	000	×	1	110
sw	S	0100011	010	×	000	×	1	000

原理图：

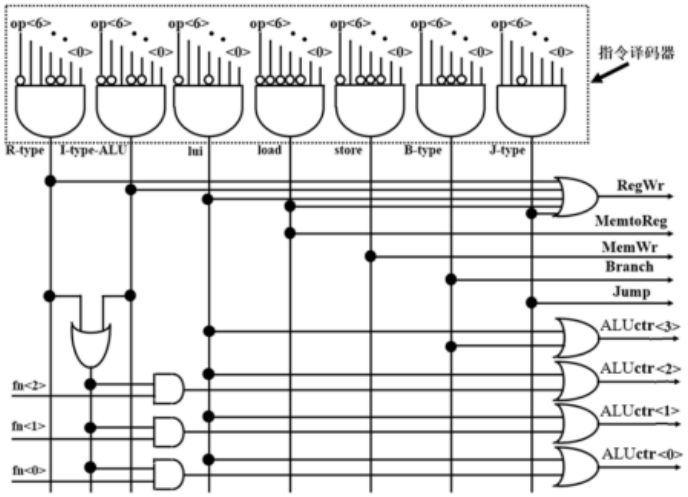
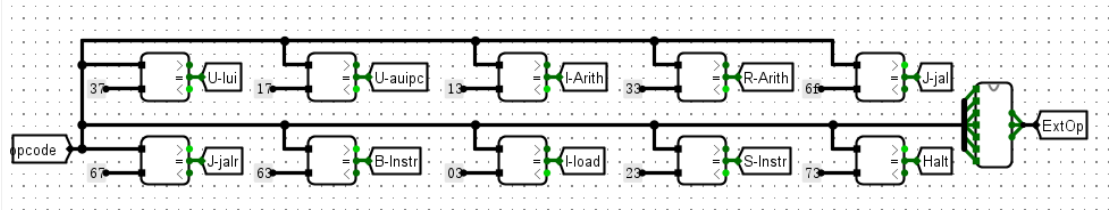
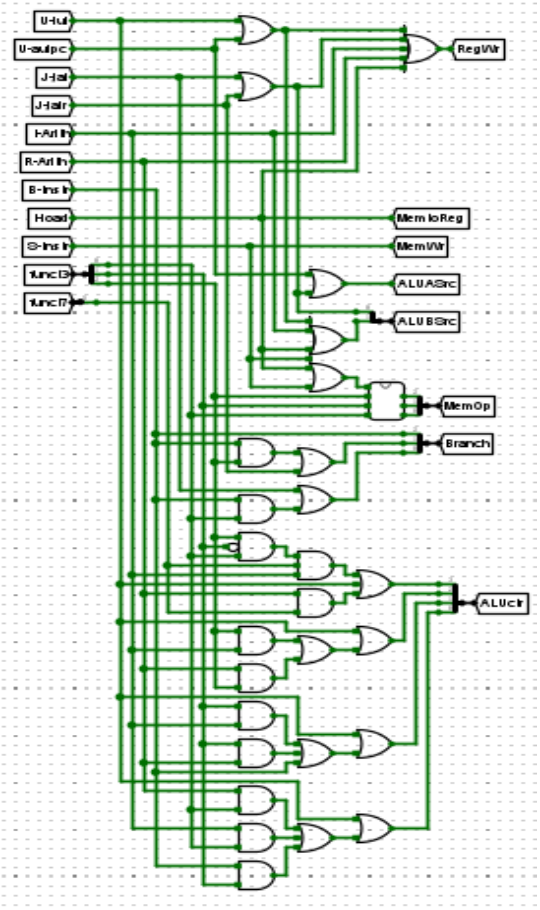


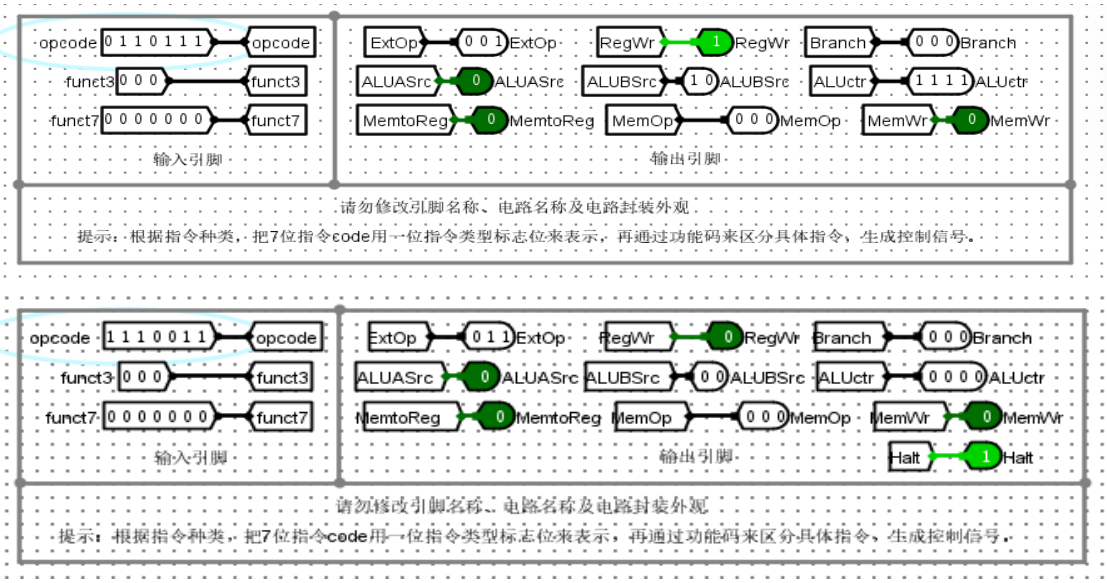
图 6.1 控制器电路示意图

电路图：





仿真测试图：



2、单周期 cpu 设计实验

整体模块设计：

在单周期 CPU 中，每条的指令都需要在一个时钟周期内完成。本次实验中，以时钟下降沿为状态元件 的有效触发，寄存器和存储器在时钟信号下降沿时同步实现写入操作；而读取操作，则是组合逻辑，输入地 址改变后，输出数据立即改变。

为了保证 CPU 的正常执行，在系统中需要考虑复位信号 Reset、中止信号 Halt，还需要考虑状态元件的 片选信号 sel 等。

复位信号 Reset 用来初始化系统状态，保证 CPU 每次执行程序时，都能从相同的状态开始。当复位信 号有效时，PC 寄存器的输入端为程序段初始地址，数据存储器清零端有效。

CPU 一旦开始执行程序，下地址计算部件就不断计算下条指令的地址，PC 寄存器持续更新。为了观测 当前程序执行结果，在程序执行结束时，需要中止当前程序的执行。本次实验中，使用 ecall 指令作为程序 停止执行的指令，在汇编测试程序中，以 ecall 指令作为结束语句，ecall 指令的操作码为 1110011（0x73）。 因此修改控制器设计，增加一个输出信号 Halt，当操作码 opcode 为 0x73 时，中止信号 Halt 有效，赋值为 1。当中止信号有效时，使得 PC 寄存器的使能端无效，暂停 PC 输出。

单周期 CPU 中的状态元件有三个：PC 寄存器、指令存储器和数据存储器。PC 寄存器在每个时钟周期 都需要修改，因此 PC 寄存器的片选信号设置为高电平有效，且要始终有效。指令存储器 ROM 的片选信号 Sel 设置为高电平有效，且要始终有效。数据存储器中每一片字节存储器 RAM 片选信号设置都设置为高电 平有效，但是每一片 RAM 的片选信号的输入值需根据数据存储器读写格式控制信号 MemOp 和最低 2 位 地址来决定。 根据图 6.4 所示的单周期 CPU 原理图，CPU 由不同组件构成，分工协作完成功能。

原理图：

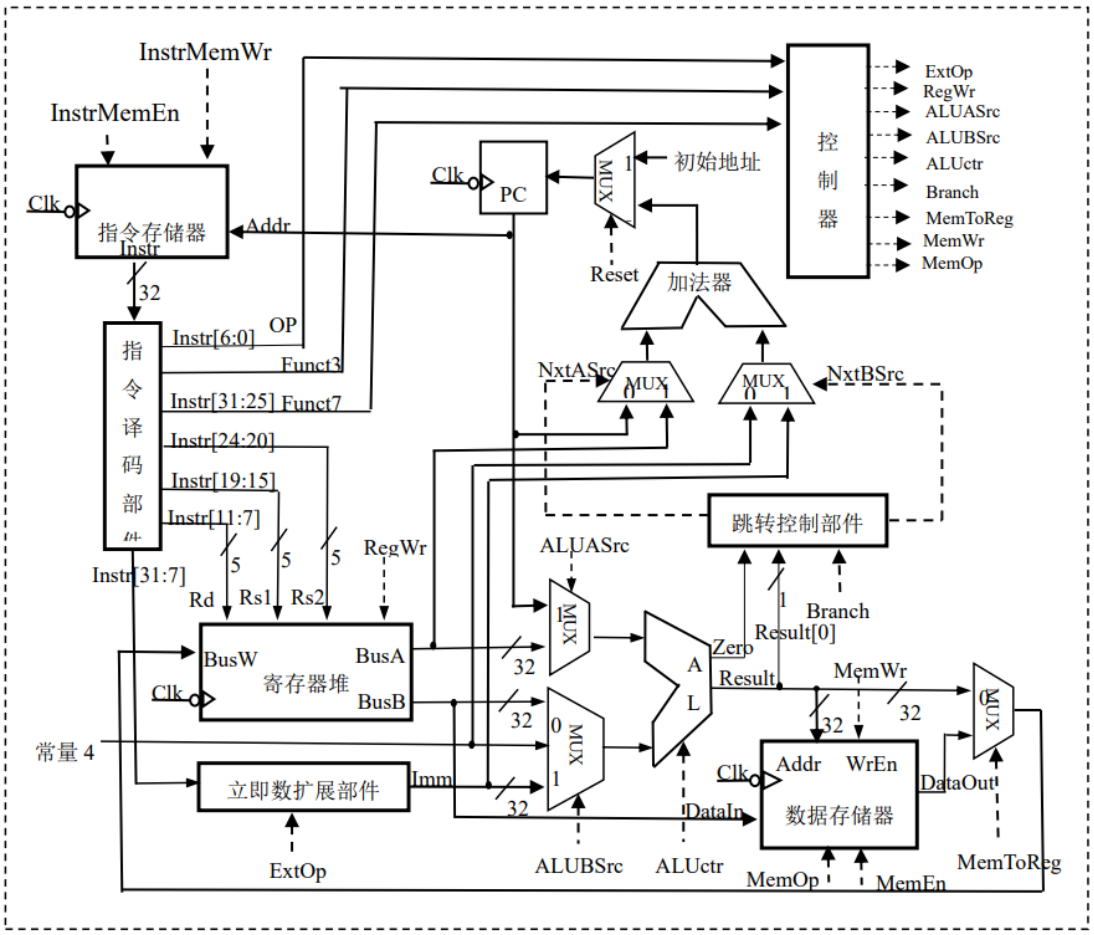
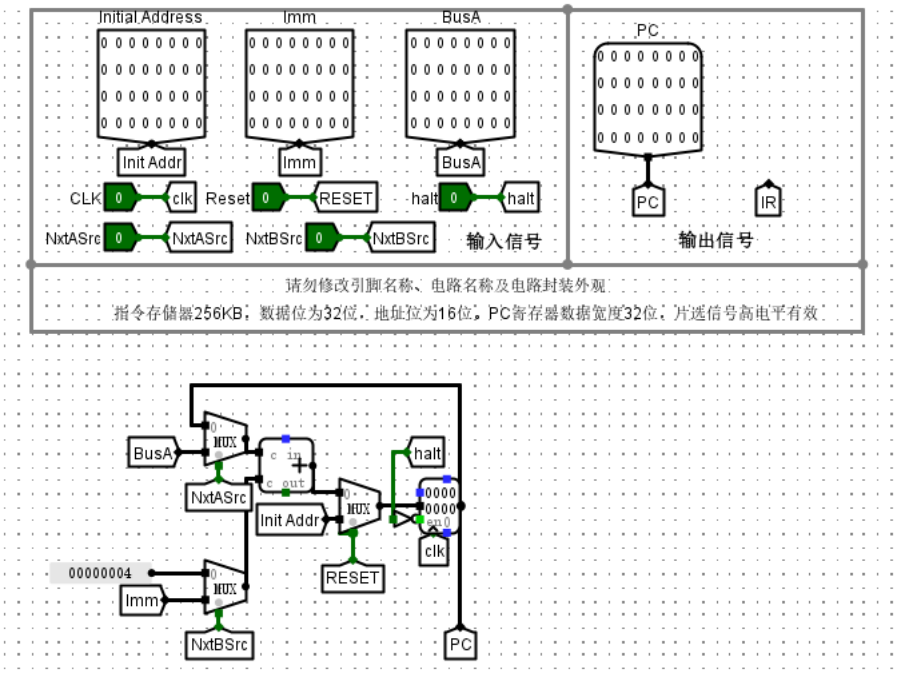
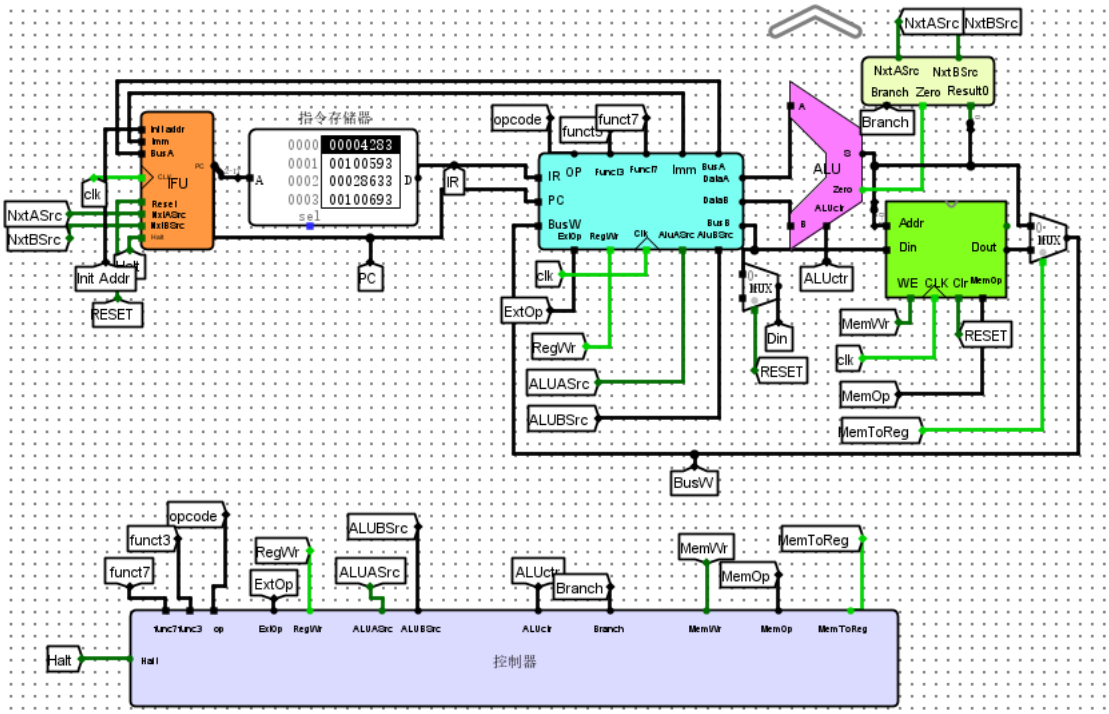


图 6.4 单周期 CPU 原理图

电路图：（修改 IFU 子电路，加入 Halt 信号）



单周期 cpu 电路



仿真测试图:

—— 实际输出 ——

PC	IR	BusW	Din	Cycles
00000000	00000000	00000000	00000000	0000
00000000	80000037	80000000	00000000	0000
00000004	7ff00013	000007ff	00000000	0001
00000008	0010c0b3	00000000	00000000	0002
0000000c	00100193	00000001	00000000	0003
00000010	14101863	00000000	00000000	0004
00000014	800000b7	80000000	00000000	0005
00000018	ffff8137	ffff8000	00000000	0006
0000001c	00208733	7fff8000	ffff8000	0007
00000020	7fff83b7	7fff8000	00000000	0008
00000024	00200193	00000002	ffff8000	0009
00000028	12771c63	00000000	7fff8000	000a
0000002c	0ff010b7	0ff01000	00000000	000b
00000030	ff008093	0ff00ff0	00000000	000c
00000034	f0f0f137	f0f0f000	00000000	000d
00000038	0f010113	f0f0f0f0	00000000	000e
0000003c	0020f7b3	00f000f0	f0f0f0f0	000f
00000040	00f003b7	00f00000	00f000f0	0010
00000044	0f038393	00f000f0	00000000	0011
00000048	00300193	00000003	00000002	0012
0000004c	10779a63	00000000	00f000f0	0013
00000050	00100093	00000001	0ff00ff0	0014
00000054	03f00113	0000003f	00000000	0015
00000058	00209833	80000000	0000003f	0016
0000005c	800003b7	80000000	00000000	0017
00000060	00400193	00000004	00000000	0018
00000064	0e781e63	00000000	80000000	0019
00000068	800000b7	80000000	00000000	001a
0000006c	00e00113	0000000e	7fff8000	001b
00000070	4020d8b3	fffe0000	0000000e	001c
00000074	fffe03b7	fffe0000	00000000	001d
00000078	00500193	00000005	00000000	001e
0000007c	0e789263	00000000	fffe0000	001f
00000080	800000b7	80000000	00000000	0020
00000084	7ff0a913	00000001	00000000	0021
00000088	00100393	00000001	80000000	0022
0000008c	00600193	00000006	00000000	0023
00000090	0c791863	00000000	00000001	0024
00000094	800000b7	80000000	00000000	0025
00000098	7ff0b993	00000000	00000000	0026
0000009c	00000393	00000000	00000000	0027
000000a0	00700193	00000007	00000000	0028
000000a4	0a799e63	00000000	00000000	0029
000000a8	00000097	000000a8	00000000	002a
000000ac	0cc08093	00000174	00000000	002b
000000b0	0ee00113	000000ee	7fff8000	002c
000000b4	00208023	00000174	000000ee	002d

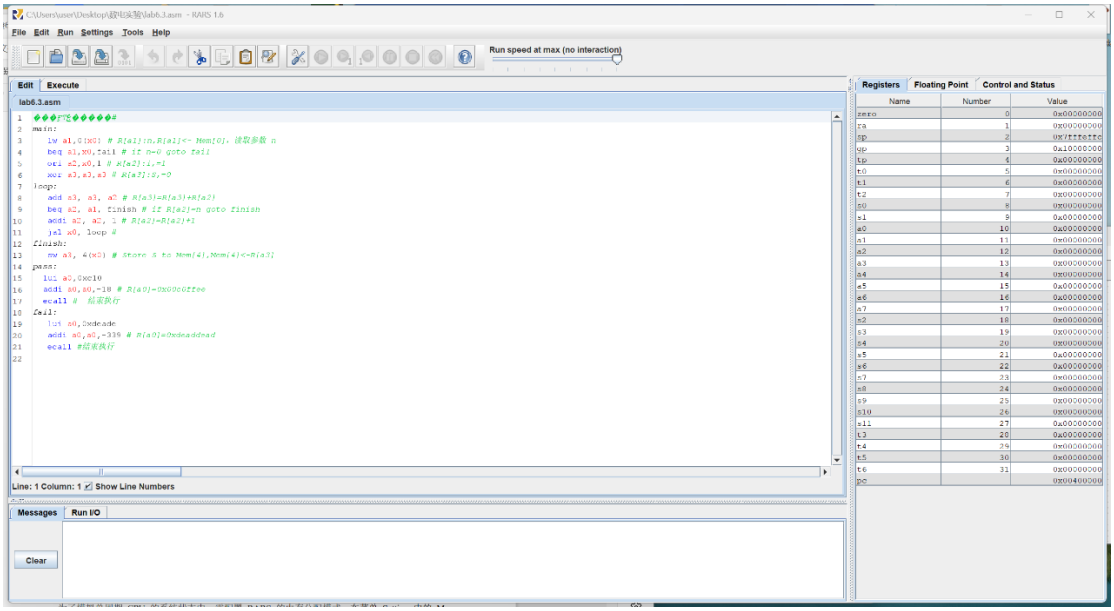
000000b8	0ff00113	000000ff	00000000	002e
000000bc	002080a3	00000175	000000ff	002f
000000c0	0c000113	000000c0	00000000	0030
000000c4	00208123	00000176	000000c0	0031
000000c8	00000113	00000000	00000000	0032
000000cc	002081a3	00000177	00000000	0033
000000d0	00108a03	ffffffff	00000174	0034
000000d4	fff00393	ffffffff	00000000	0035
000000d8	00800193	00000008	00000000	0036
000000dc	087a1263	00000000	ffffffff	0037
000000e0	0000da83	0000ffee	00000000	0038
000000e4	000103b7	00010000	00000000	0039
000000e8	fee38393	0000ffee	7fff8000	003a
000000ec	00900193	00000009	00000000	003b
000000f0	067a9863	00000000	0000ffee	003c
000000f4	01509223	00000178	0000ffee	003d
000000f8	00209b03	000000c0	00000000	003e
000000fc	0c000393	000000c0	00000000	003f
00000100	00a00193	0000000a	00000000	0040
00000104	047b1e63	00000000	000000c0	0041
00000108	01609323	0000017a	000000c0	0042
0000010c	0000ab83	00c0ffee	00000000	0043
00000110	00c103b7	00c10000	00000000	0044
00000114	fee38393	00c0ffee	7fff8000	0045
00000118	00b00193	0000000b	00000000	0046
0000011c	047b9263	00000000	00c0ffee	0047
00000120	0040ac03	00c0ffee	00000000	0048
00000124	00c103b7	00c10000	00000000	0049
00000128	fee38393	00c0ffee	7fff8000	004a
0000012c	00c00193	0000000c	00000000	004b
00000130	027c1863	00000000	00c0ffee	004c
00000134	00000c97	00000134	00000000	004d
00000138	010c8ce7	0000013c	80000000	004e
00000144	00000397	00000144	00000000	004f
00000148	ff838393	0000013c	00c0ffee	0050
0000014c	00d00193	0000000d	00000000	0051
00000150	007c9863	00000000	0000013c	0052
00000154	00c10537	00c10000	00000000	0053
00000158	fee50513	00c0ffee	7fff8000	0054

3、 用累加和程序验证 CPU 设计

整体模块设计：

在单周期 CPU 设计中，指令存储器 ROM 和数据存储器 DataRam 是独立部署分开实现的，因此可以各 自独立编址。因此在编写 RISC-V 汇编语言程序时，指令存储器地址和数据存储器都可以从地址 0 号单元开 始读取。

实验过程：（在 RARS 加载汇编语言程序）

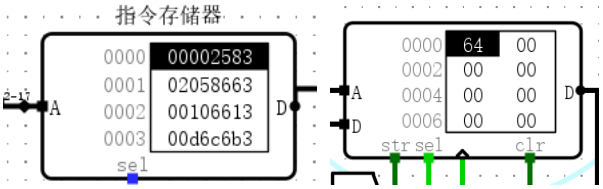


(程序运行结果)：

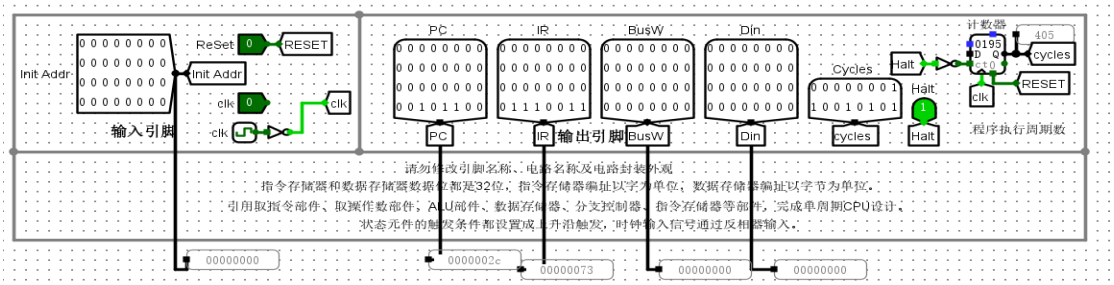
Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x00000000	0x00000064	0x000013ba	0x00000000	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

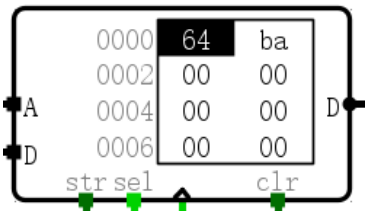
在指令存储器加载镜像指令：lab6.3.hex,并在数据存储器的最低字节 RAM 写入 0x64



选择时钟连续，CPU 开始执行机器代码：



检查数据存储器：



四、用冒泡排序程序进行 CPU 设计验证

整体模块设计：

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 进行比效，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。

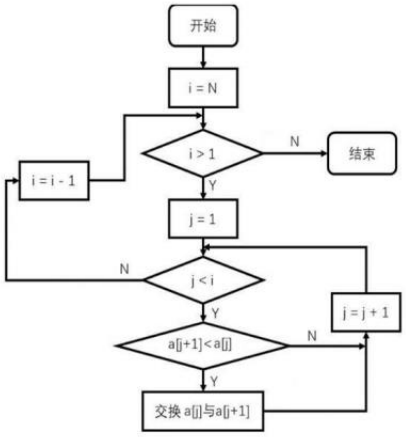
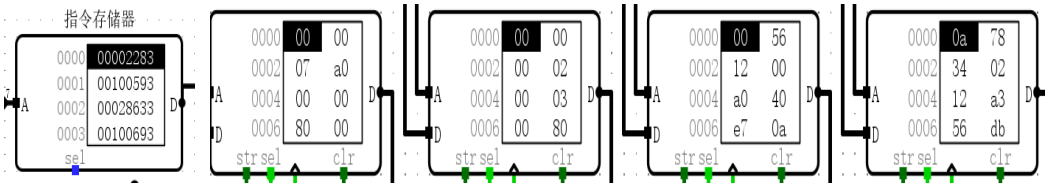


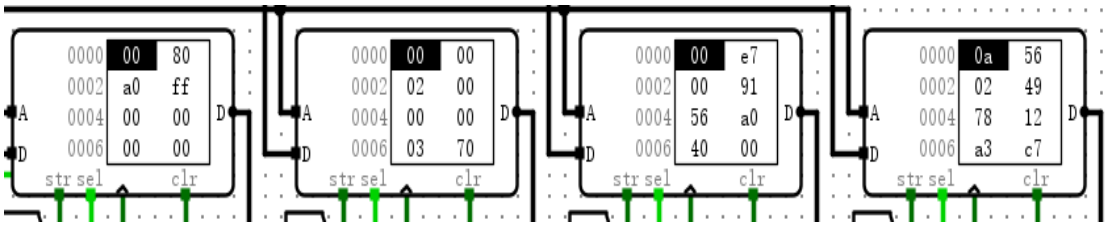
图 6.15 冒泡排序算法流程图

原理图：

实验过程：在指令存储器中加载镜像文件 lab6.4.hex，在数据存储器 RAM0~RAM3 中分别加载镜像文件 lab6.4_d.hex0~lab6.4_d.hex3。

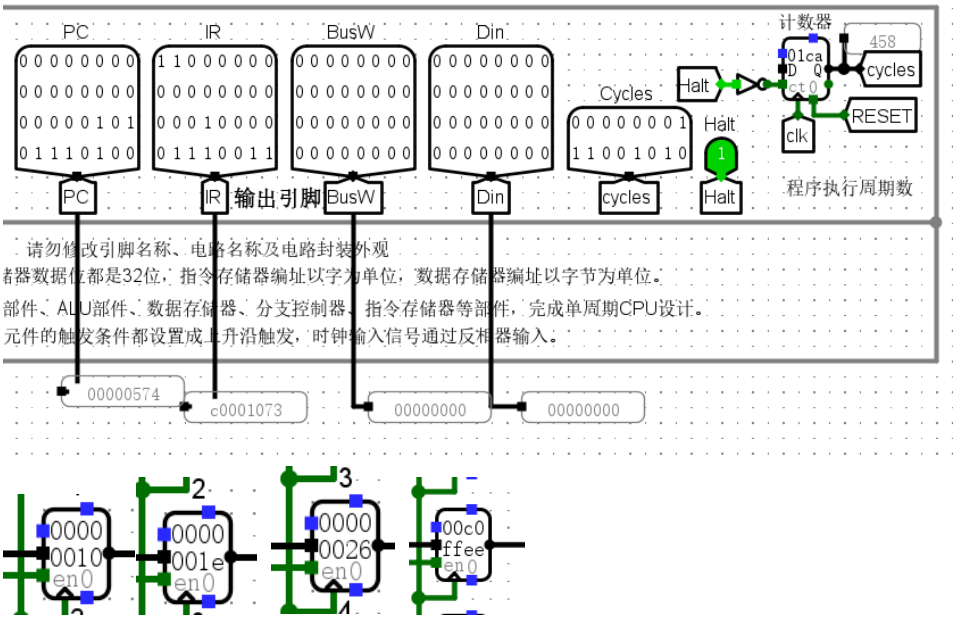


实验结果：

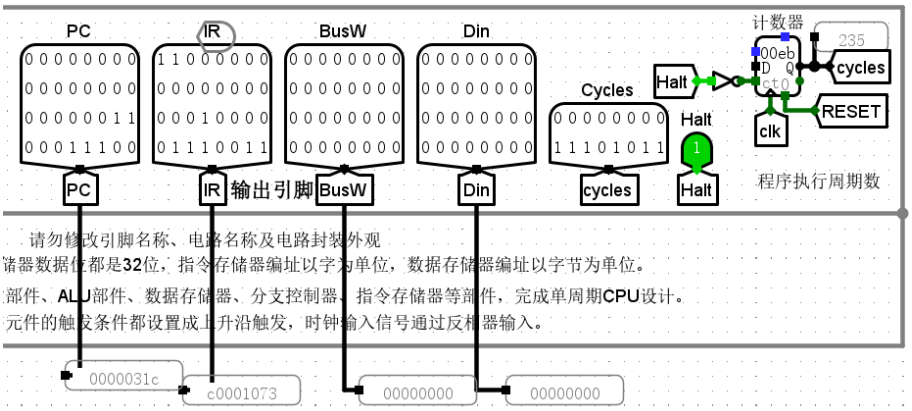


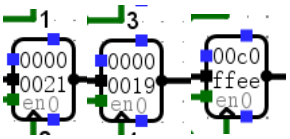
5、官方测试集测试

实验过程：（add 测试集）

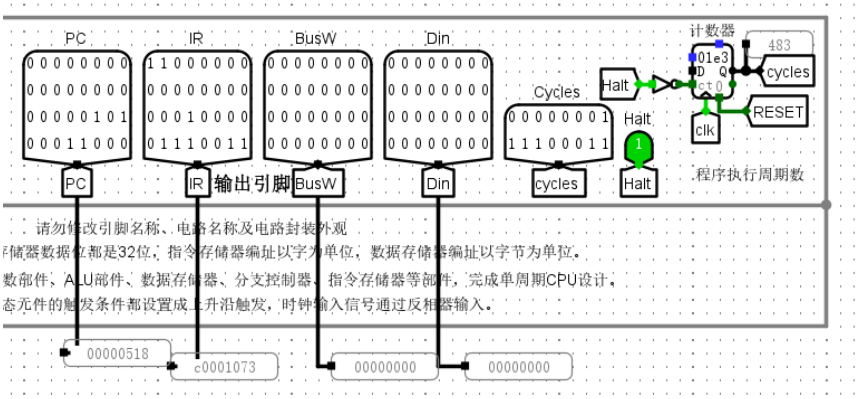


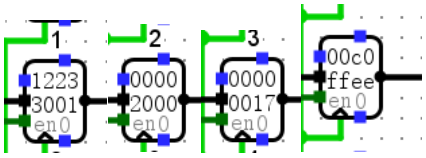
（addi 测试集）





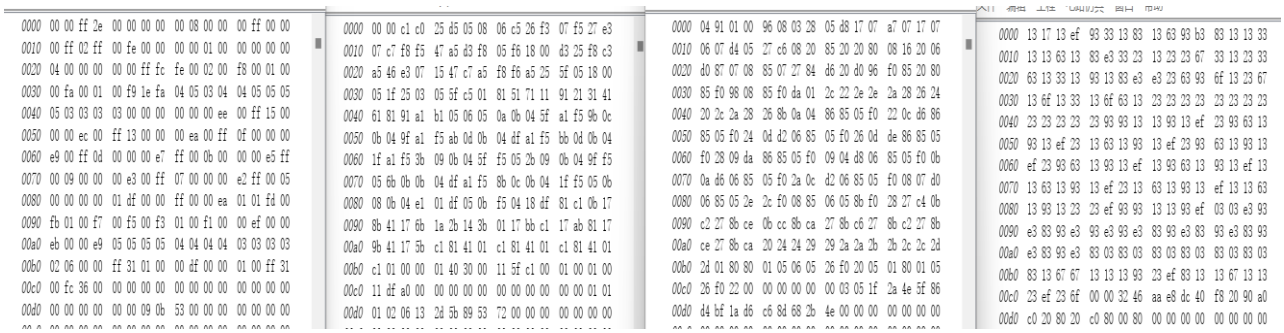
（store 测试集）



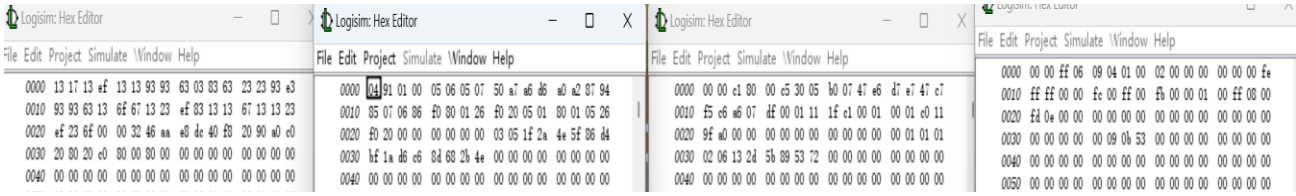


6、计算机系统基础 PA 程序测试

实验过程：（qsort 快排测试集）（从左至右是 RAM3-RAM0 的数据集）



（bsort 冒泡排序测试集）



四、错误现象及分析

实验连线较多，一开始经常会出现线路交叉的情况，后来注意观察线路有没有圆圈，且在线路交叉的地方预留多一点的空间，线路就会变得整洁了很多，出错的情况也少多了。

五、思考题

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。

思路：（部分通过网上查阅资料获取）

在单 CPU 系统上实现多任务处理，可以通过时间分片或协作多任务处理来实现，这些方法使得多个任务可以在单 CPU 上看起来像是“同时”执行。

时间分片是一种抢占式多任务处理方法，在这种方法中，操作系统将 CPU 的时间划分成多个时间片，并在这些时间片之间切换任务。每个任务在其时间片内运行，然后被中断，接着另一个任务运行。通过快速地切换任务，给用户一种多个任务同时运行的错觉。

实现步骤：

操作系统内核：

创建一个简单的调度器可以是一个循环调度器或优先级调度器。

使用一个时钟中断来触发任务切换。例如，每隔 10 毫秒触发一次中断。

任务切换：

保存当前任务的状态（CPU 寄存器、程序计数器等）。

恢复下一个任务的状态。

任务管理：

维护一个任务队列或任务列表，其中包含所有要执行的任务。

在每个时间片结束时，从任务队列中选择下一个任务执行。

```
// 简单的任务结构体
typedef struct {
    void (*taskFunction)();
    int state;
    // 其他任务相关的状态信息
} Task;

Task tasks[MAX_TASKS];
int currentTask = 0;

void timerInterruptHandler() {
    // 保存当前任务的状态
    saveTaskState(&tasks[currentTask]);

    // 选择下一个任务
    currentTask = (currentTask + 1) % MAX_TASKS;

    // 恢复下一个任务的状态
    restoreTaskState(&tasks[currentTask]);
}

void runScheduler() {
    while (1) {
        tasks[currentTask].taskFunction();
    }
}

// 初始化任务并启动调度器
void init() {
    // 初始化任务队列
    tasks[0].taskFunction = task1;
    tasks[1].taskFunction = task2;
    // 初始化其他任务...

    // 设置定时器中断
    setupTimerInterrupt(timerInterruptHandler);

    // 启动调度器
    runScheduler();
}
```

协作多任务处理是另一种多任务处理方法，在这种方法中，任务主动放弃 CPU 使用权，使得操作系统可以调度其他任务运行。

实现步骤：

任务定义：

将每个任务实现为一个循环，每次循环迭代执行一部分任务，然后主动放弃控制权。

任务调度：

操作系统维护一个任务列表，循环调度任务。

任务切换：

在每个任务中，明确调用一个“yield”函数来放弃 CPU 控制权。

```
void task1() {
    while (1) {
        // 执行一部分任务
        doTask1Part();

        // 主动放弃控制权
        yield();
    }
}

void task2() {
    while (1) {
        // 执行一部分任务
        doTask2Part();

        // 主动放弃控制权
        yield();
    }
}

void runScheduler() {
    while (1) {
        for (int i = 0; i < MAX_TASKS; i++) {
            if (tasks[i].state == READY) {
                tasks[i].taskFunction();
            }
        }
    }
}

// 初始化任务并启动调度器
void init() {
    // 初始化任务队列
    tasks[0].taskFunction = task1;
    tasks[1].taskFunction = task2;
    // 初始化其他任务...

    // 启动调度器
    runScheduler();
}
```

具体例子：（同时执行计算累加和数据排序）

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_TASKS 2
typedef void (*TaskFunction)();
typedef struct {
    TaskFunction taskFunction;
    int isCompleted;
} Task;
Task tasks[MAX_TASKS];
int currentTask = 0;
void calculateSum() {
    static int sum = 0;
    for (int i = 1; i <= 100; i++) {
        sum += i;// 模拟一个小的时间片，进行任务切换
        if (i % 10 == 0) {
            printf("Partial sum: %d\n", sum);
            yield();
```

```
            }
        }
        tasks[0].isCompleted = 1;
        printf("Total sum: %d\n", sum);
}
void sortArray() {
        static int array[] = {5, 3, 2, 4, 1};
        static int n = sizeof(array) / sizeof(array[0]);
        for (int i = 0; i < n-1; i++) {
            for (int j = 0; j < n-i-1; j++) {
                if (array[j] > array[j+1]) {
                    int temp = array[j];
                    array[j] = array[j+1];
                    array[j+1] = temp;
                }
            }// 模拟一个小的时间片，进行任务切换
            printf("Array after pass %d: ", i+1);
            for (int k = 0; k < n; k++) {
                printf("%d ", array[k]);
            }
            printf("\n");
            yield();
        }
        tasks[1].isCompleted = 1;
}
void yield() {
        currentTask = (currentTask + 1) % MAX_TASKS;
        if (!tasks[currentTask].isCompleted) {
            tasks[currentTask].taskFunction();
        }
}
void runScheduler() {
        while (!tasks[0].isCompleted || !tasks[1].isCompleted) {
            tasks[currentTask].taskFunction();
        }
}
int main() {
        tasks[0].taskFunction = calculateSum;
        tasks[0].isCompleted = 0;
        tasks[1].taskFunction = sortArray;
        tasks[1].isCompleted = 0;
        // 启动调度器
        runScheduler();
        return 0;}
```

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。

1. 连接设备到总线：将键盘、TTY 等设备连接到计算机系统的总线上，可以使用适当的接口标准连接键盘，使用串口或并口连接 TTY 等设备。
2. 设备地址分配：为每个设备分配一个唯一的地址，以便 CPU 能够识别和访问这些设备。这通常通过在总线上使用地址线进行设置来完成。
3. 编写设备驱动程序：为每个设备编写相应的设备驱动程序，驱动程序负责与设备通信并处理数据传输。驱动程序需要了解设备的通信协议和操作方式。
4. 设备注册：在系统启动时，设备驱动程序需要注册到操作系统中，告诉操作系统这些设备的存在和如何访问它们。
5. 数据传输：当需要从设备中读取数据时，CPU 通过访问相应的设备地址来触发设备驱动程序执行相应的读取操作；当需要向设备中写入数据时，CPU 也通过访问相应的设备地址来触发设备驱动程序执行相应的写入操作。
6. 中断处理：为了提高效率，通常会使用中断机制来处理设备数据的读写。设备在有数据传输时会向 CPU 发送中断信号，CPU 在接收到中断信号后会暂停当前执行的任务，转而执行与该设备相关的中断处理程序。

3、阐述如何在单周期 CPU 基础上实现多周期 CPU 和流水线 CPU

实现多周期 CPU：

1. 指令分解：将每条指令分解成几个基本的操作步骤，取指令（IF），指令解码（ID），执行（EX），内存访问（MEM）和写回（WB）。
2. 状态机设计：设计一个有限状态机，用于控制在每个时钟周期内执行哪个步骤。状态机会根据当前的操作和指令类型决定下一个状态。
3. 硬件资源共享：在多周期 CPU 中，可以共享硬件资源。例如，ALU 可以在不同的指令执行阶段被重复使用，而不是为每种类型的指令单独设计。
4. 控制逻辑修改：修改控制逻辑以支持多个步骤的执行。控制信号不再是针对整个指令的，而是针对每个步骤的。
5. 数据通路调整：调整数据通路以允许在不同的时钟周期内传输数据。这可能涉及到增加更多的寄存器来保存中间结果，以及设计多用途的功能单元。

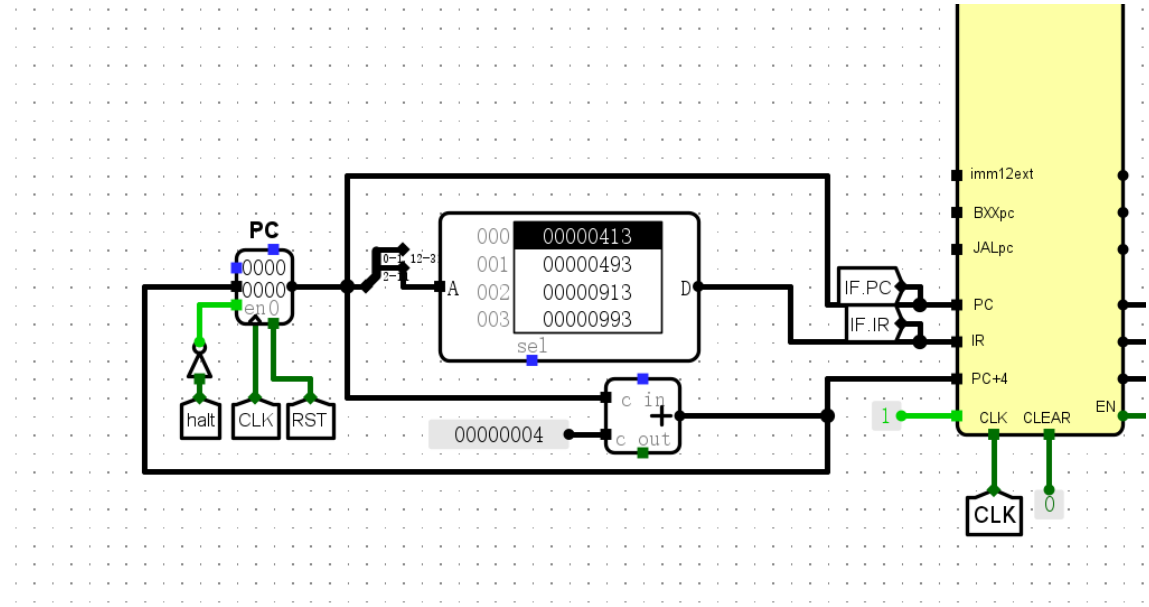
实现流水线 CPU：

6. 指令流水线划分：将指令执行过程划分为多个流水线阶段，通常包括取指、译码、执行、访存、写回等阶段。每个阶段负责执行指令执行过程中的一个子任务。
7. 流水线寄存器设计：在每个流水线阶段之间插入流水线寄存器，用于保存各个阶段的中间结果。这些寄存器有助于隔离各个阶段，确保流水线的正常运行。
8. 重新设计控制信号：设计并引入适当的控制信号，用于控制流水线的各个阶段，并确保指令在流水线中正确流动。
9. 处理数据冲突：由于流水线中的指令重叠执行，可能会出现数据冲突。可以采用数据前推、数据旁路等技术来解决数据冲突，以确保流水线的正常运行。
10. 处理控制冲突和异常：与多周期 CPU 类似，流水线 CPU 也需要处理控制冲突和异常情况。可以采用类似的方法来处理这些情况，例如插入气泡、延迟槽等。

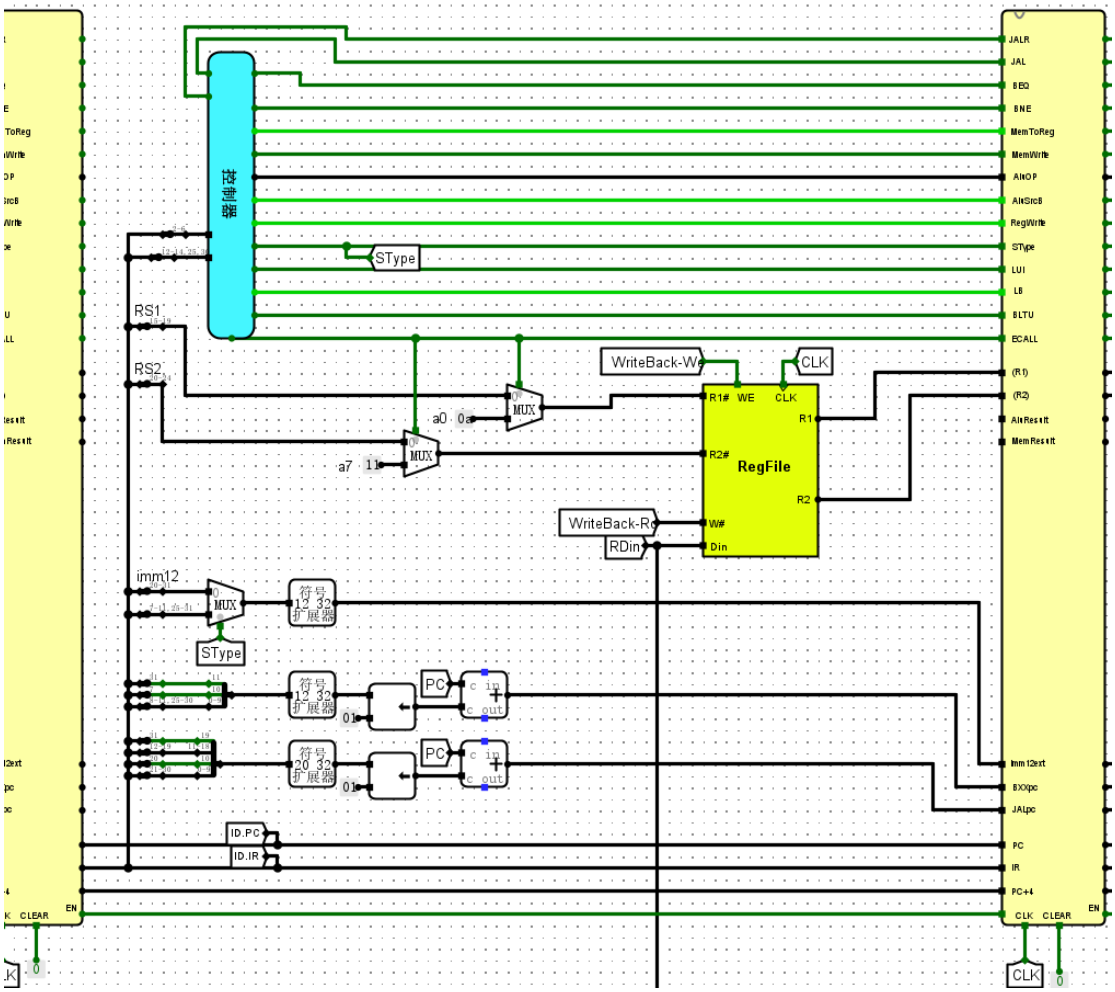
下展示简陋的模拟理想流水线：

电路图：（分成五个部分展现）

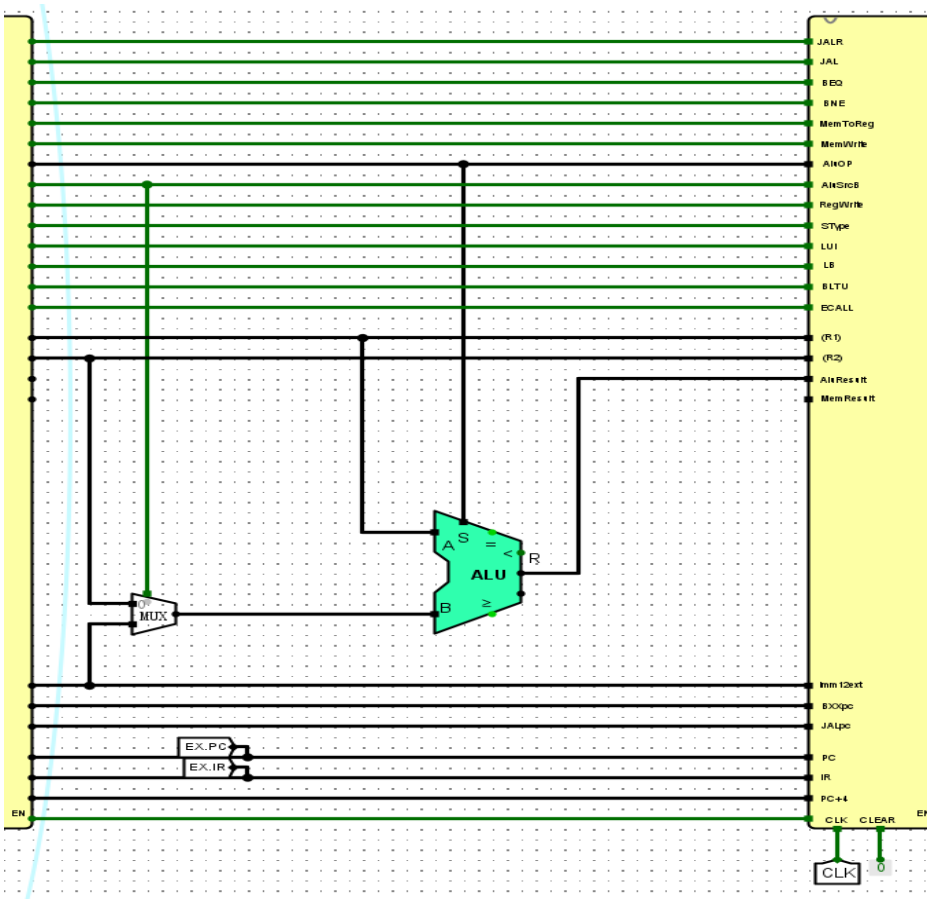
IF 段:


PC
halt
CLK
RST
000
001
002
003
00000413
00000493
00000913
00000993
A
D
sel
00000004
c_in
c_out
IF.PC
IF.IR
imm12ext
BXXpc
JALpc
PC
IR
PC+4
CLK
CLEAR
EN
1
0

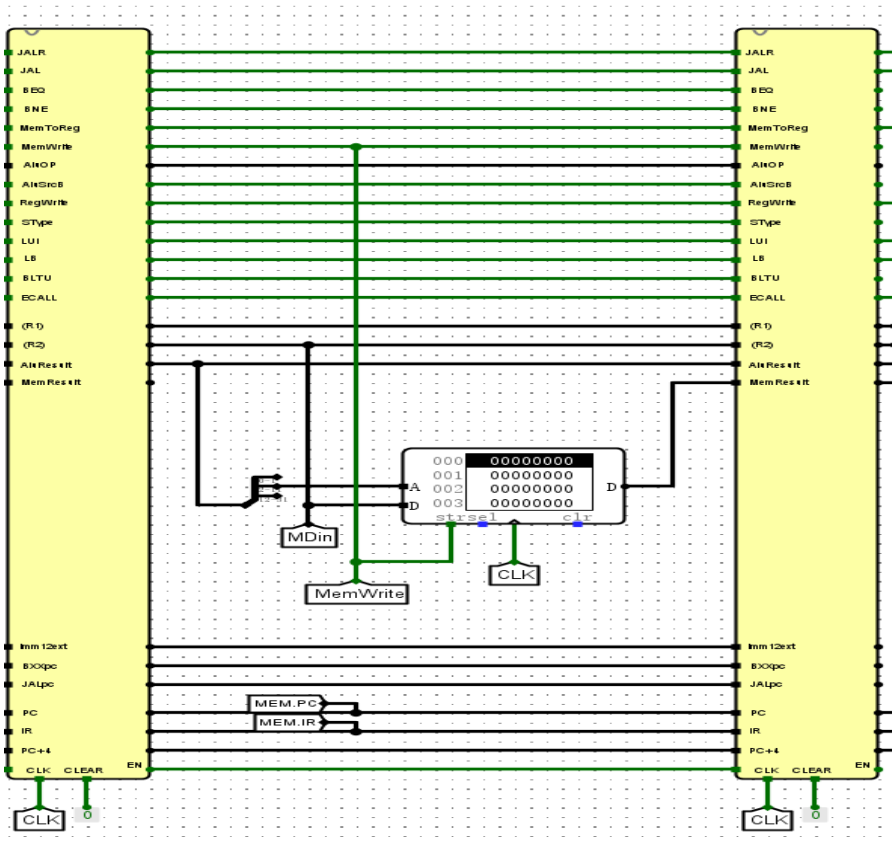
ID 段:


控制器
SType
RS1
RS2
WriteBack-We
CLK
MUX
RegFile
R1#
WE
R1
R2#
R2
WriteBack-Rd
RDin
W#
Din
imm12
符号 12 32 扩展器
符号 20 32 扩展器
PC
c_in
c_out
ID.PC
ID.IR
JALR
JAL
BEQ
BNE
MemToReg
MemWrite
AluOP
AluSrcB
RegWrite
LUI
LB
BLTU
ECALL
AluResult
MemResult
imm12ext
BXXpc
JALpc
PC
IR
PC+4
CLK
CLEAR
EN

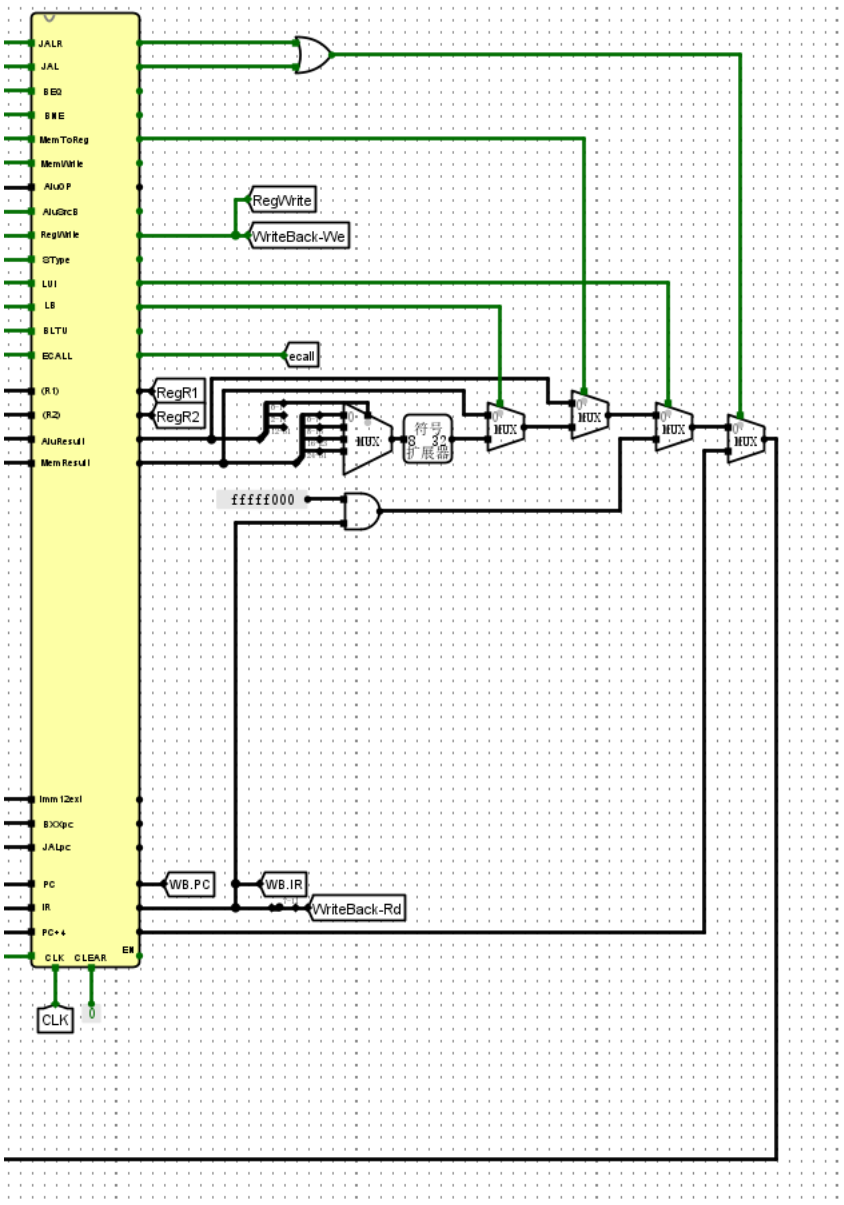
EX 段:


JALR
JAL
BEQ
BNE
MemToReg
MemWrite
AluOP
AluSrcB
RegWrite
SType
LUI
LB
BLTU
ECALL
(R1)
(R2)
AluResult
MemResult
A
S
=
<
R
ALU
B
≥
MUX
Imm12ext
BJXpc
JALpc
PC
IR
PC+4
EX.PC
EX.IR
EN
CLK
CLEAR
CLK
0

MEM 段:


JALR
JAL
BEQ
BNE
MemToReg
MemWrite
AluOP
AluSrcB
RegWrite
SType
LUI
LB
BLTU
ECALL
(R1)
(R2)
AluResult
MemResult
000 00000000
001 00000000
002 00000000
003 00000000
A
D
D
strsel
clr
MDin
MemWrite
CLK
Imm12ext
BJXpc
JALpc
PC
IR
PC+4
MEM.PC
MEM.IR
EN
CLK
CLEAR
CLK
0

WB 段：


JALR
JAL
BEQ
BNE
MemToReg
MemWrite
AluOP
AluSrcB
RegWrite
SType
LUI
LB
BLTU
ECALL
(R1)
(R2)
AluResult
MemResult
Imm12ext
BXXpc
JALpc
PC
IR
PC+4
CLK
CLEAR
EN
RegWrite
WriteBack-We
ecall
RegR1
RegR2
MUX
符号扩展器
8 32
fffff000
WB.PC
WB.IR
WriteBack-Rd
CLK
0

（Din 写回）