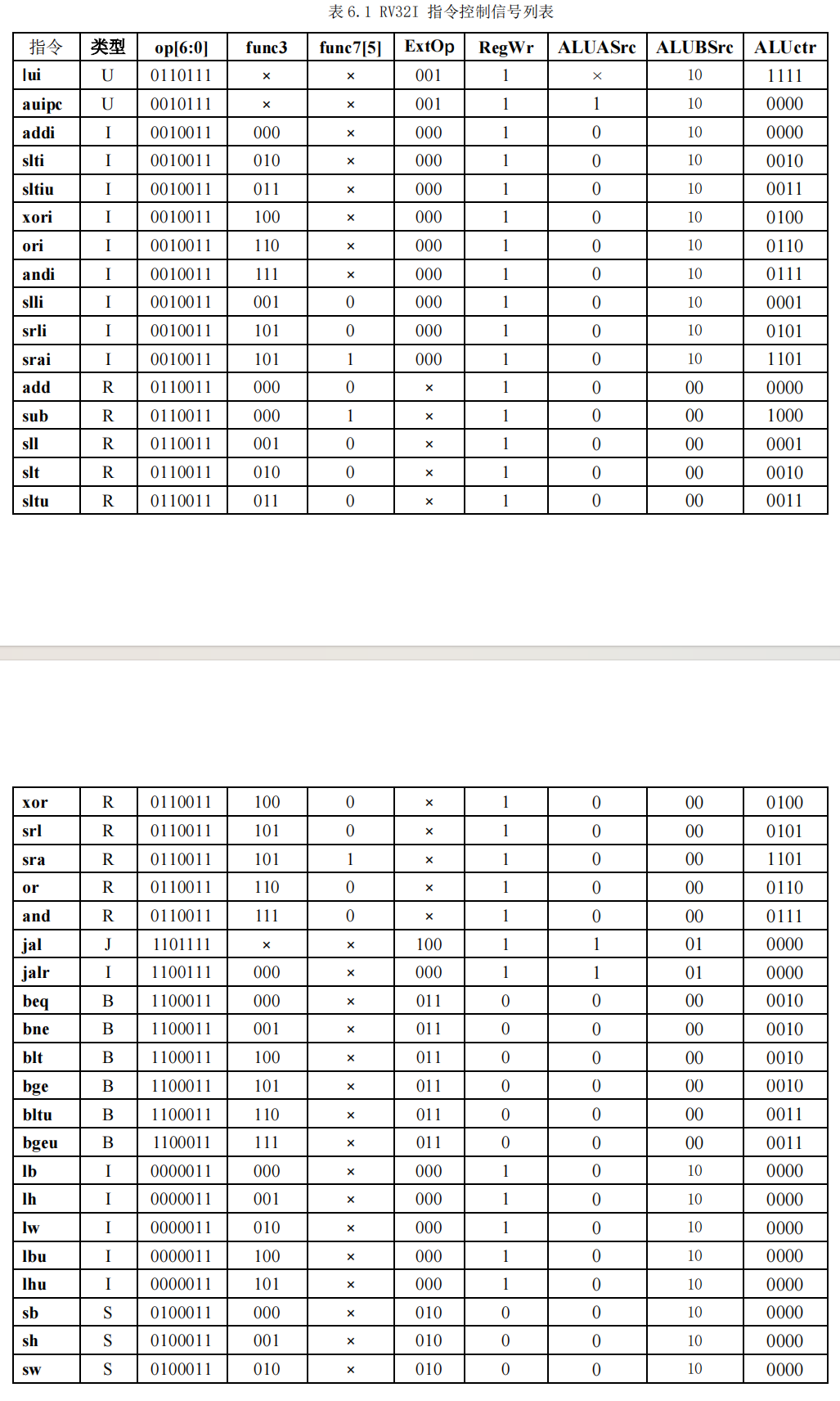
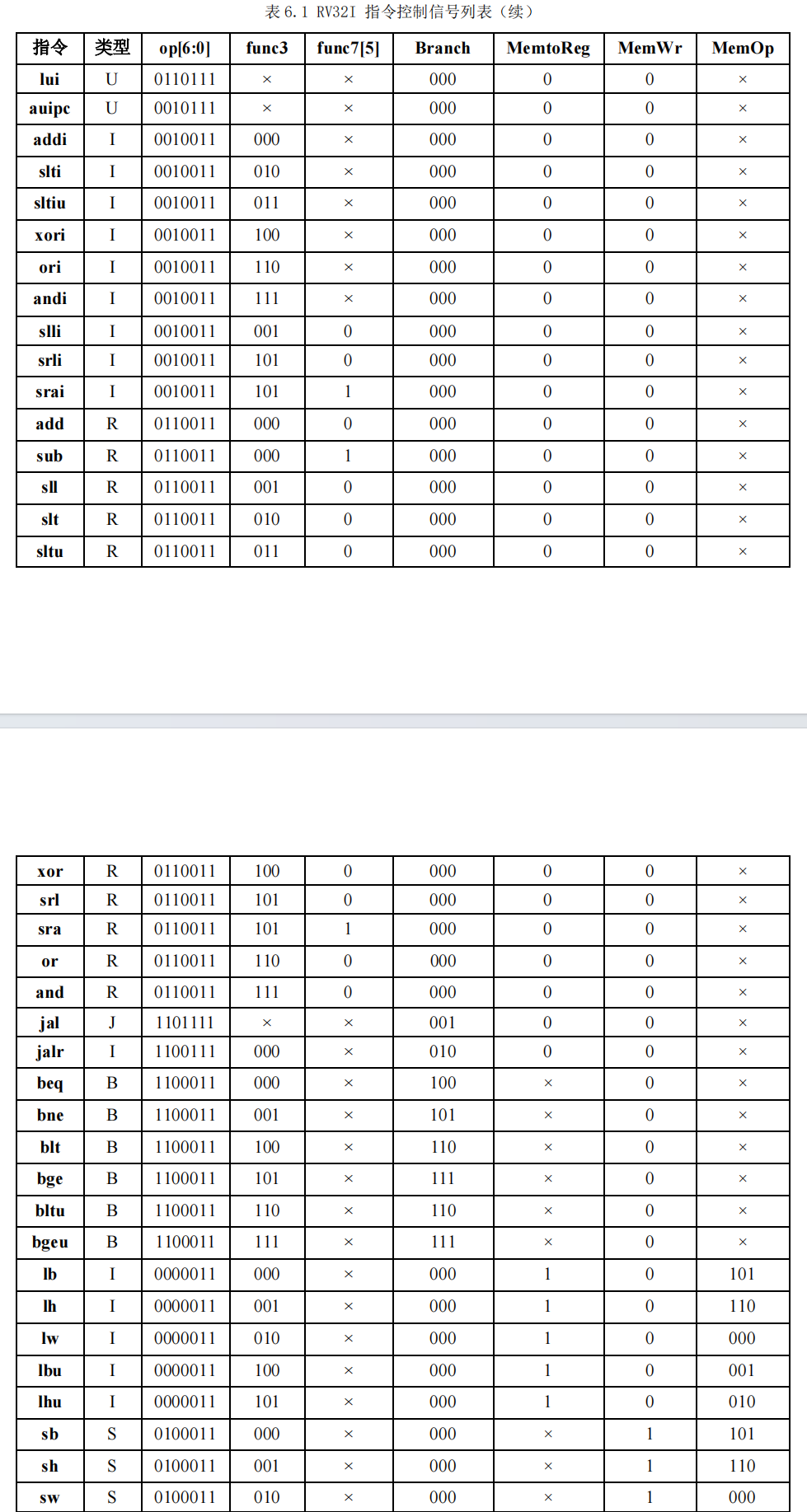
**实验6：单周期CPU设计与测试**

**6.1 控制器设计实验**

（1）原理

观察手册中的指令表如下：





可以看出，jal时候ExtOP[2]=1，B或S类ExtOP[1]=1，lui、auipc和B类ExtOP[0]=1；

RegWr在B或S类为0；

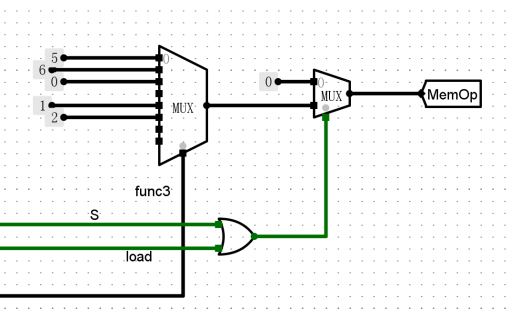
ALUAsrc在jal、jalr、auipc时为1；

ALUBsrc[0]在jal、jalr下为1，ALUBsrc[1]在U(lui, auipc）、I(I-load, I-Arith)、S类下为1；

MemToReg：load指令下为1

MemWr：S类为1

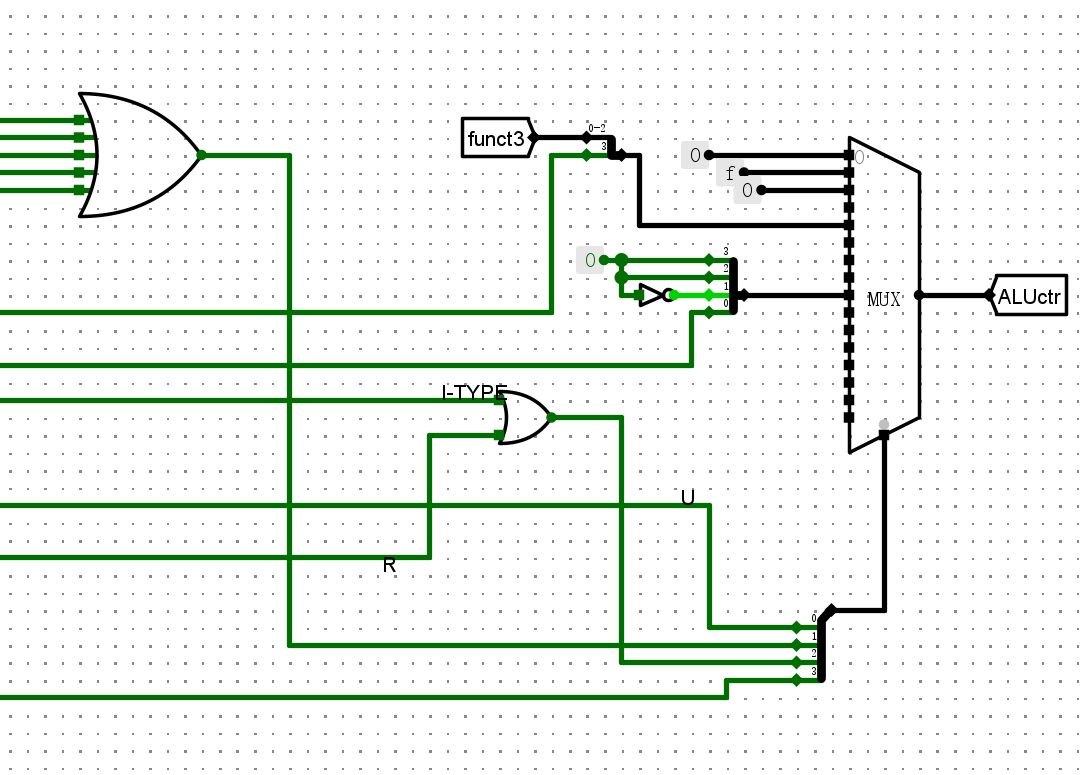
MemOp[0]：load或S下根据func3对应，见表格对应，原理图如下；其他情况直接取0；



Branch[2]在B类下为1；Branch[1]在(B & func3[2]=1)or(jalr)下为1；Branch[0]在(B & func3[0]=1)or(jal)下为1；

ALUCtr本关使用逻辑如下：

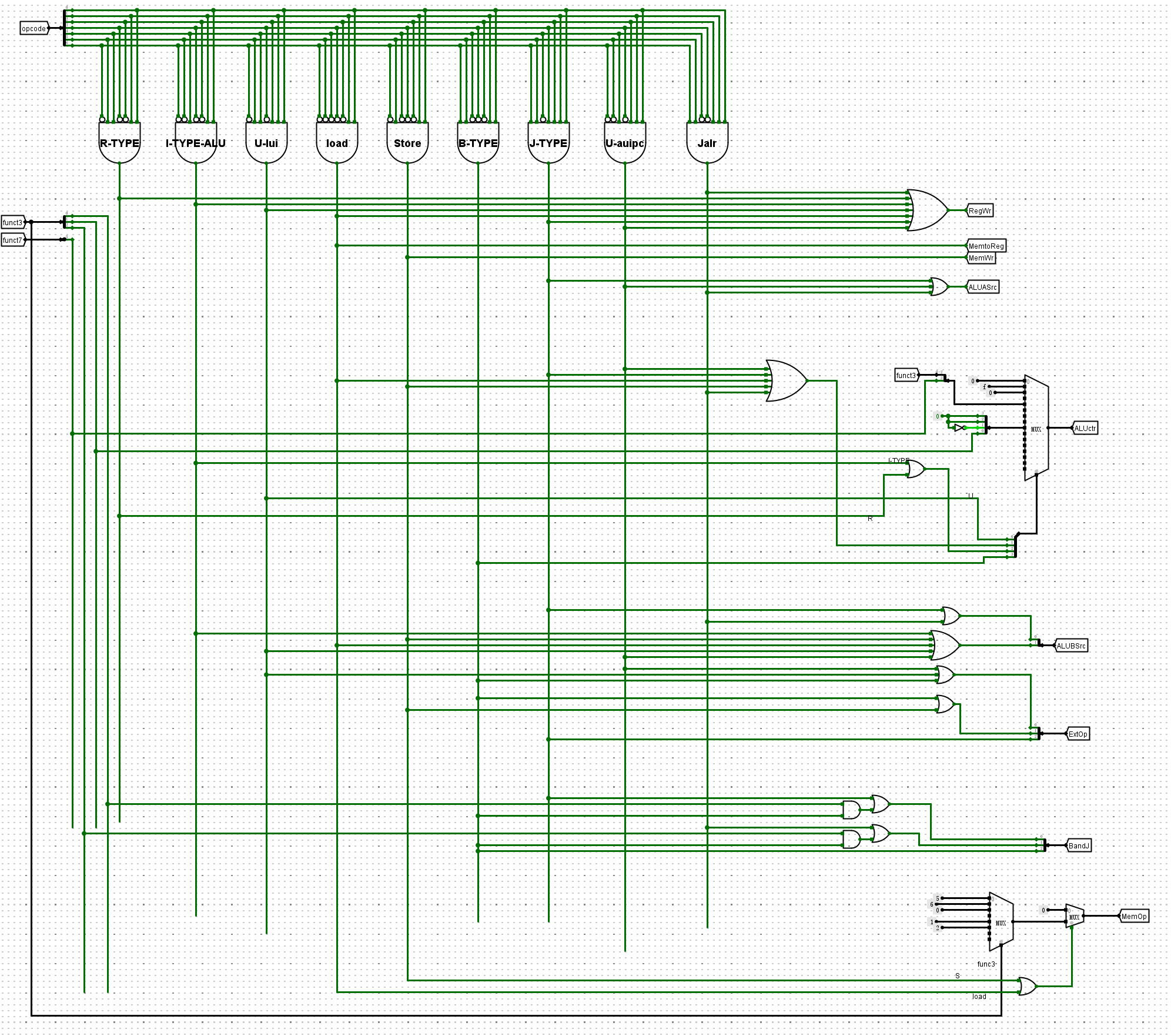
只有lui是1111，S、load类、auipc、jal、jalr的为0000，R类高位func7[5]低位func3，I-Arith类为func3（高位补0），B类'001func3[1]'。电路原理如下：



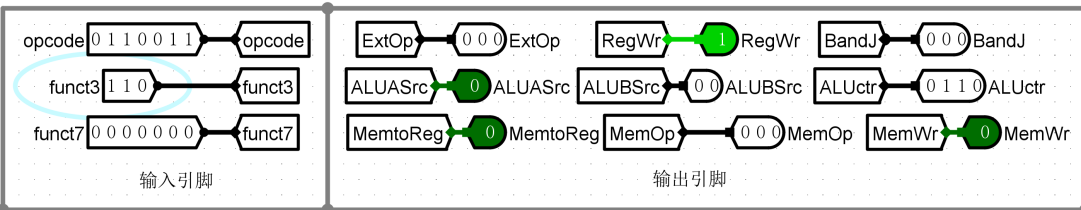
（但是本逻辑在单周期CPU实验平台测试上出现逻辑错误，原因暂不明确，于是在6.2中进行逻辑修改）

（2）Logisim电路图

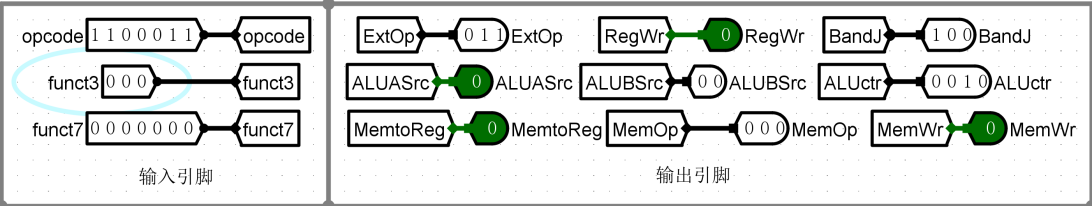
由上面逻辑，得电路图如下：

（3）实验验证

or：



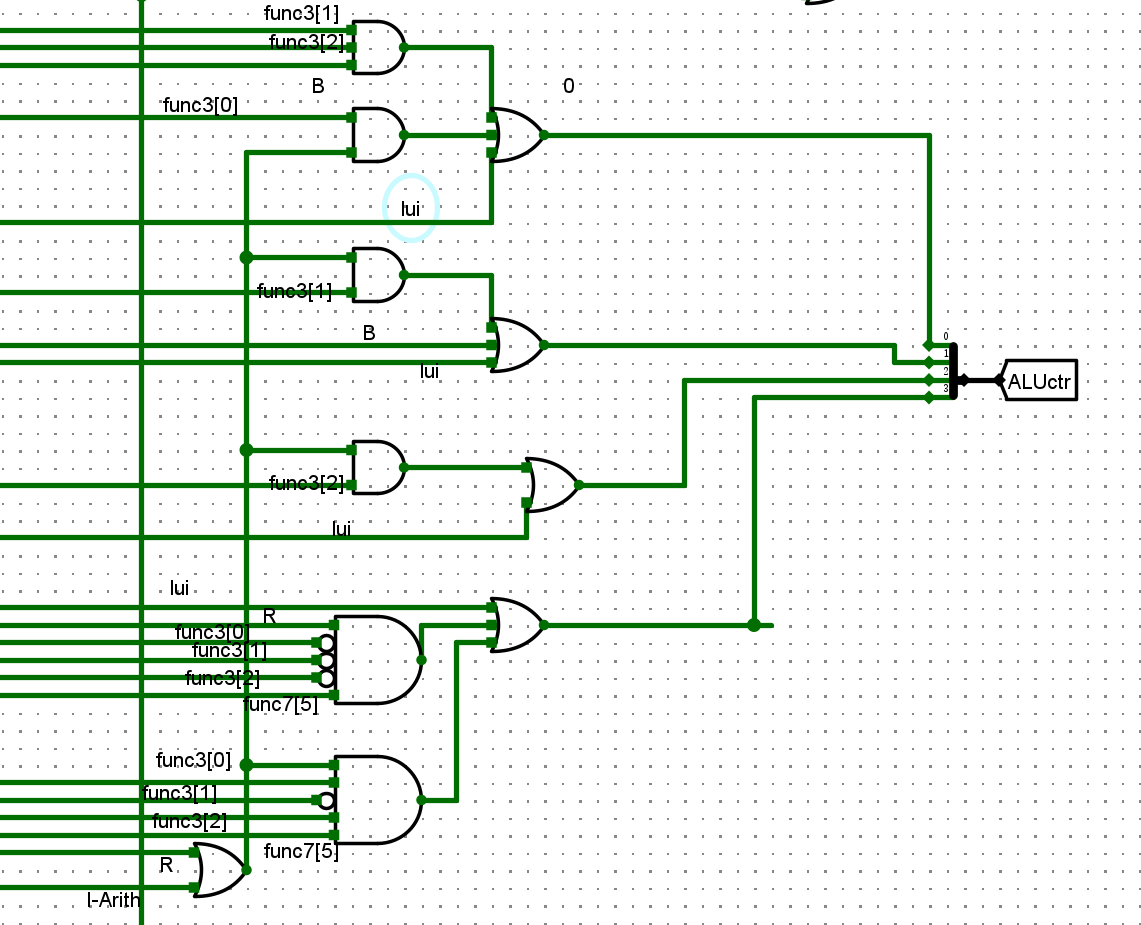
beq：



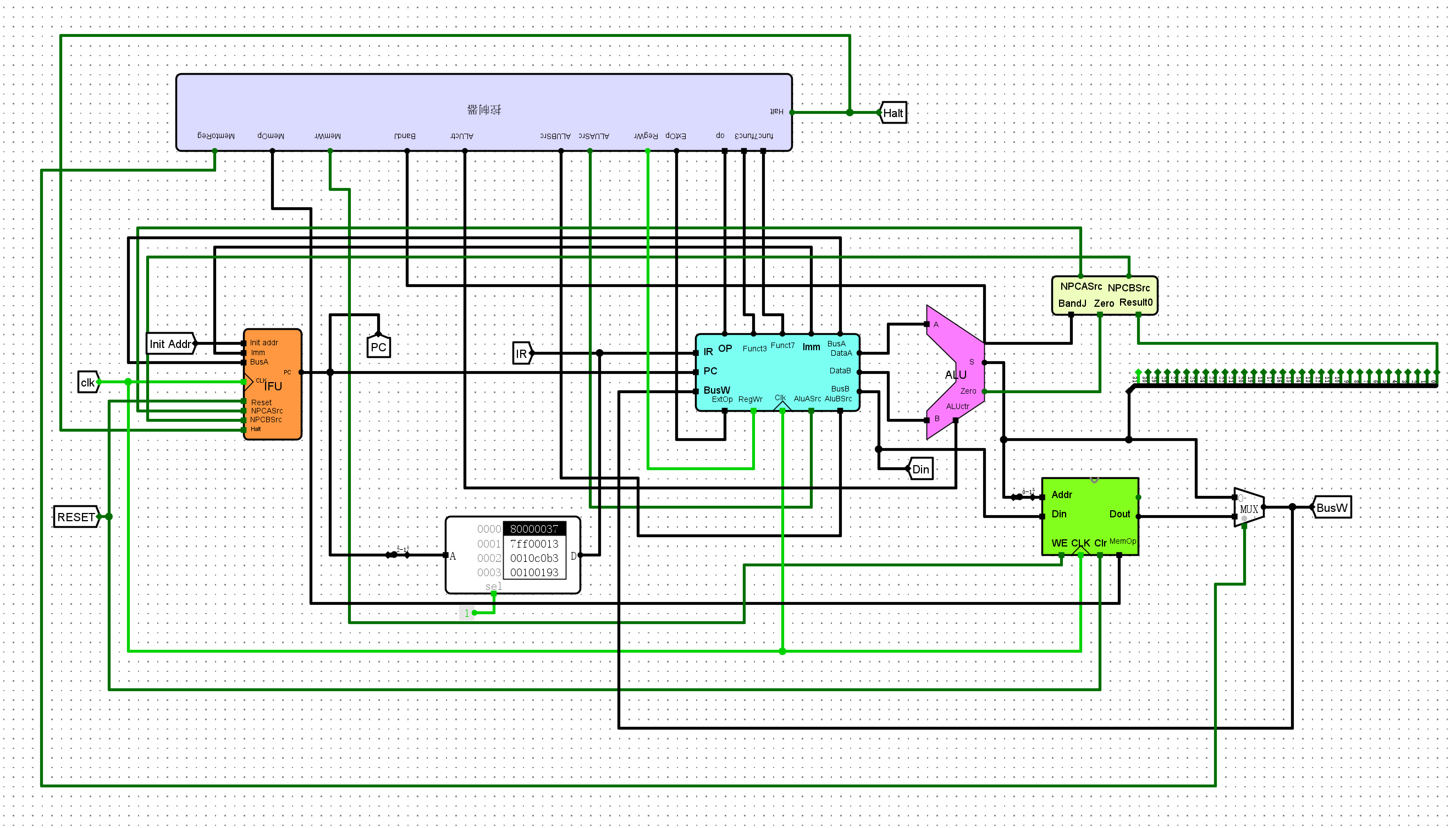
**6.2 单周期CPU设计实验**

（1）原理

基本上与实验5.5一样，接口接到6.1的控制器即可。控制器ALUCtr部分修改电路如下，其中R或I-Arith都对ALUCtr有所控制，B与func3[1,2]或lui对ALUCtr[0]有所控制，B或lui在没有func3下对ALUCtr[1]有所控制：



（2）Logisim电路图



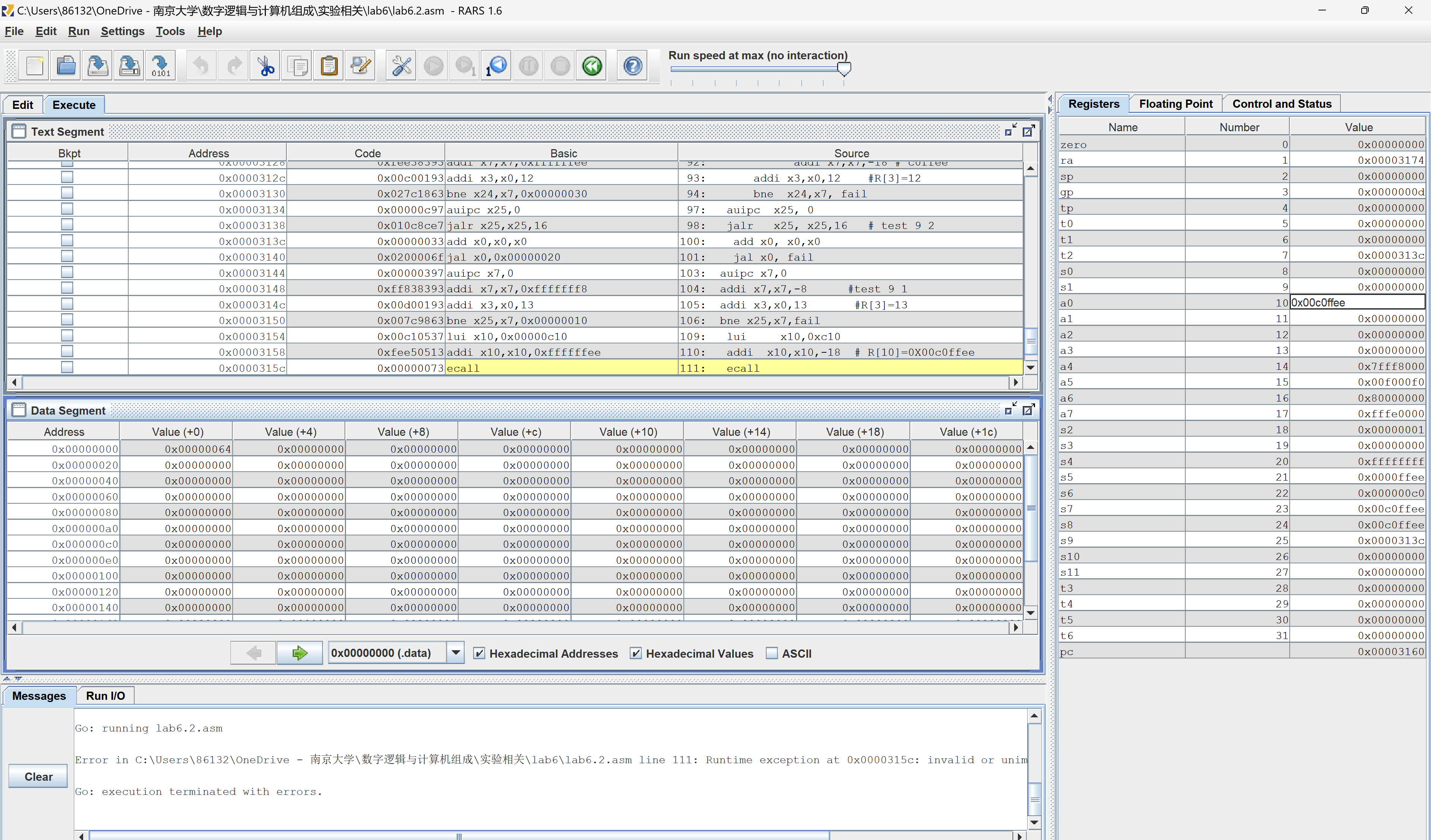
（3）实验验证

见6.3及之后

**6.3 用累加和程序验证CPU设计**

（1）原理

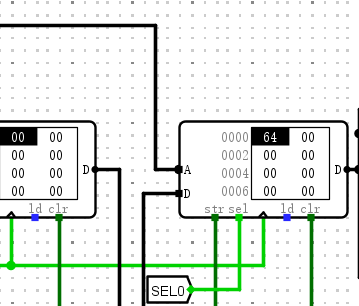
已经给好1+2+...+n的汇编指令，使用rars1\_6.jar进行生成16进制指令，取n=100（0x64），结果如下：



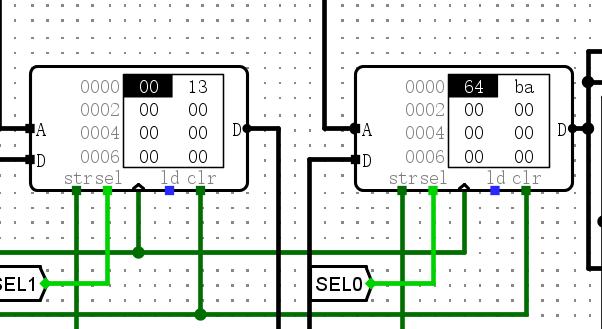
根据手册指导过程，生成的cumsum.hex成立，可运行。

（2）实验验证

在0号RAM存储n，如下：



加载镜像，输出如下，结果0x13ba：



与手册一致，验证结束。

**6.4 用冒泡排序程序进行CPU设计验证**

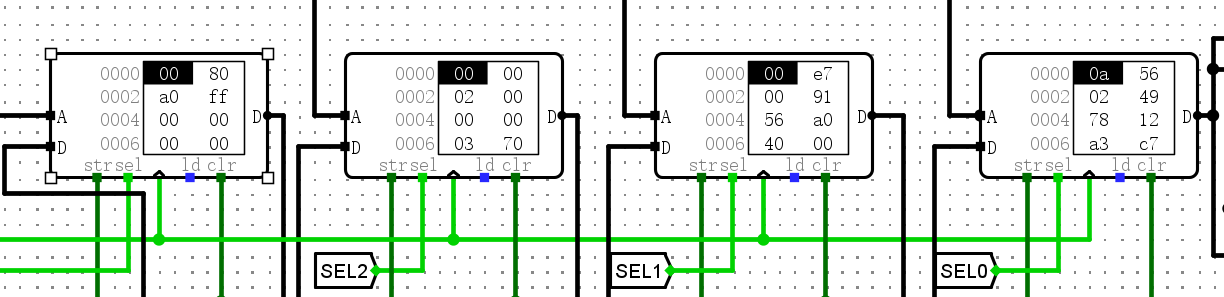
（1）原理

已经给好的汇编语句和数据集，方法同6.3，导出6.4的指令的hex文件，再在DataRAM上分别加载4个待排序的数据镜像，运行之后即可得出结果。

（2）实验验证

直接用lab6.4.hex在指令存储器上，RAM3~0分别加载lab6.4\_d.hex0 ~

lab6.4\_d.hex3，时钟连续运行后结果如图，与手册一致。



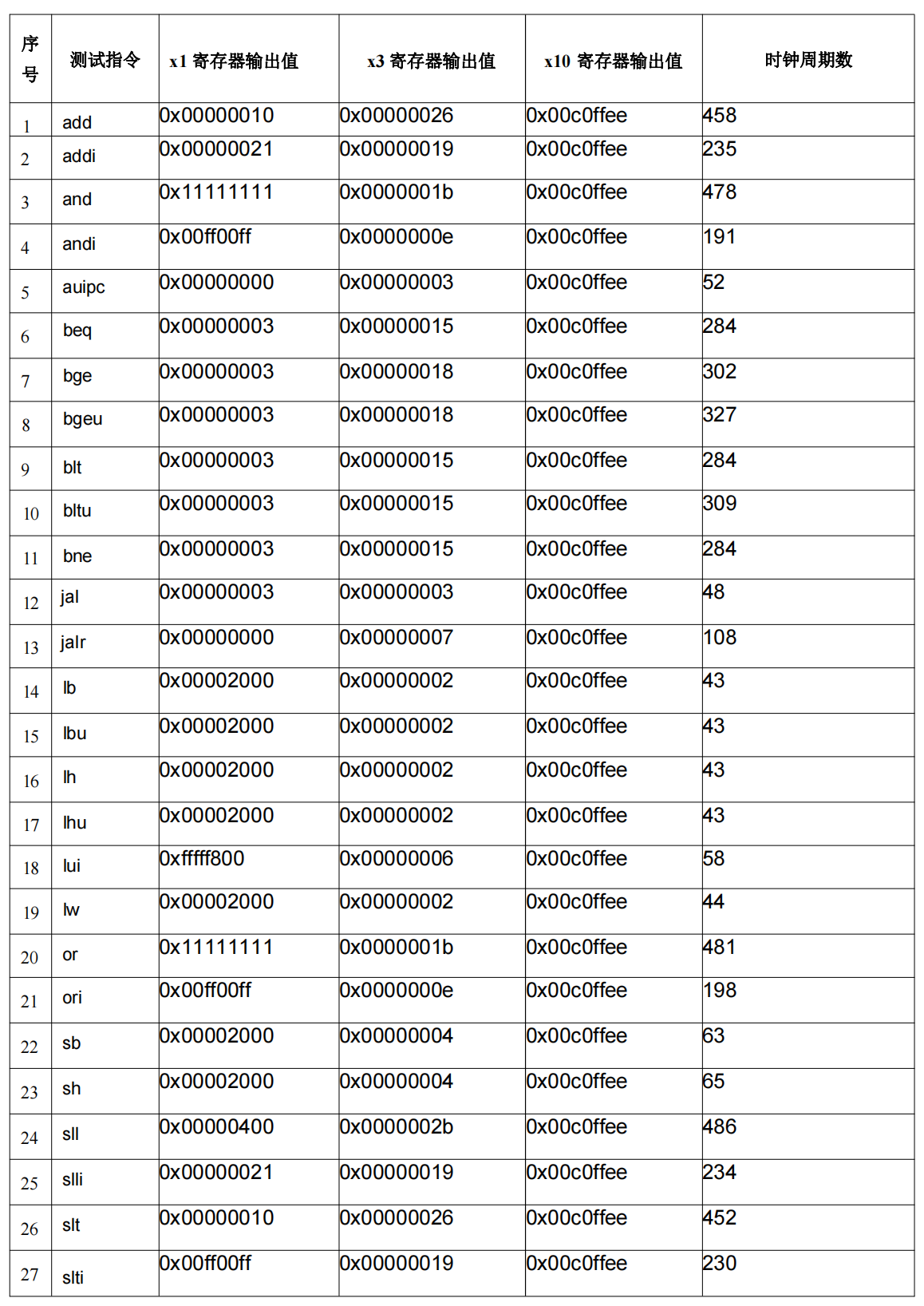
**6.5 官方测试集测试**

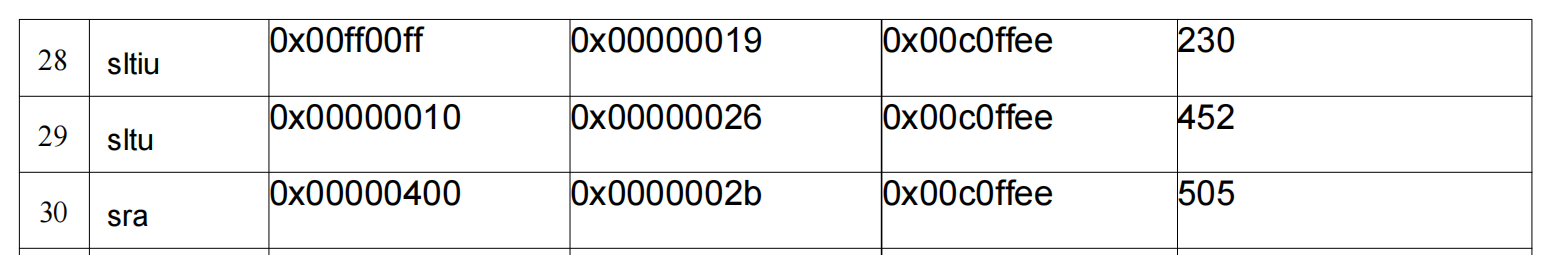
（1）原理

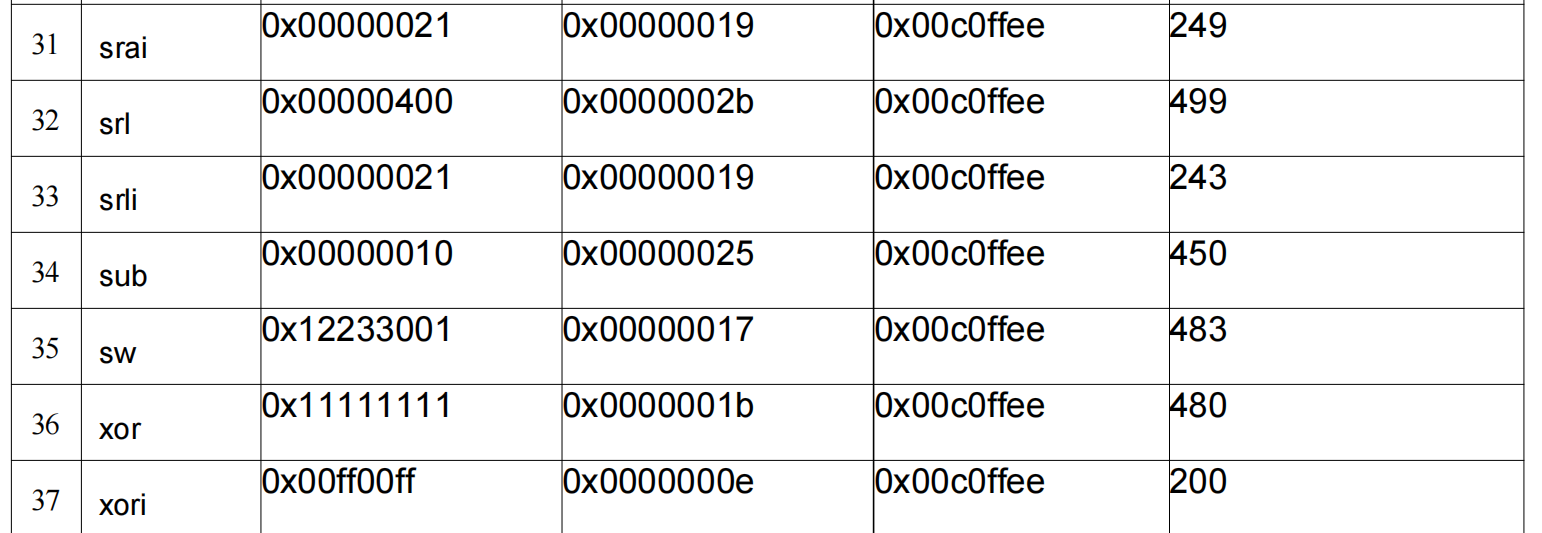
官方测试集针对不同的 RISC-V 指令变种都提供了测试。在本实验中主要使用rv32ui，也就是 RV32的基本指令集，实验中采用的环境是无虚拟地址的环境，即只使用物理地址访问内存。如果指令测试通过，则在a0寄存器中的数据为 0x00c0ffee。

（2）实验验证

这里使用线下验收的填表，运行指令用testcase文件夹中的hex文件，没有对需要RAM的进行加载0-3的数据文件，只是在指令处加载的没有'\_d'后缀的。运行结果在[线下验收表](实验六 线下验收表_fin.pdf)中，如下：







**6.6 计算机系统基础PA程序测试**

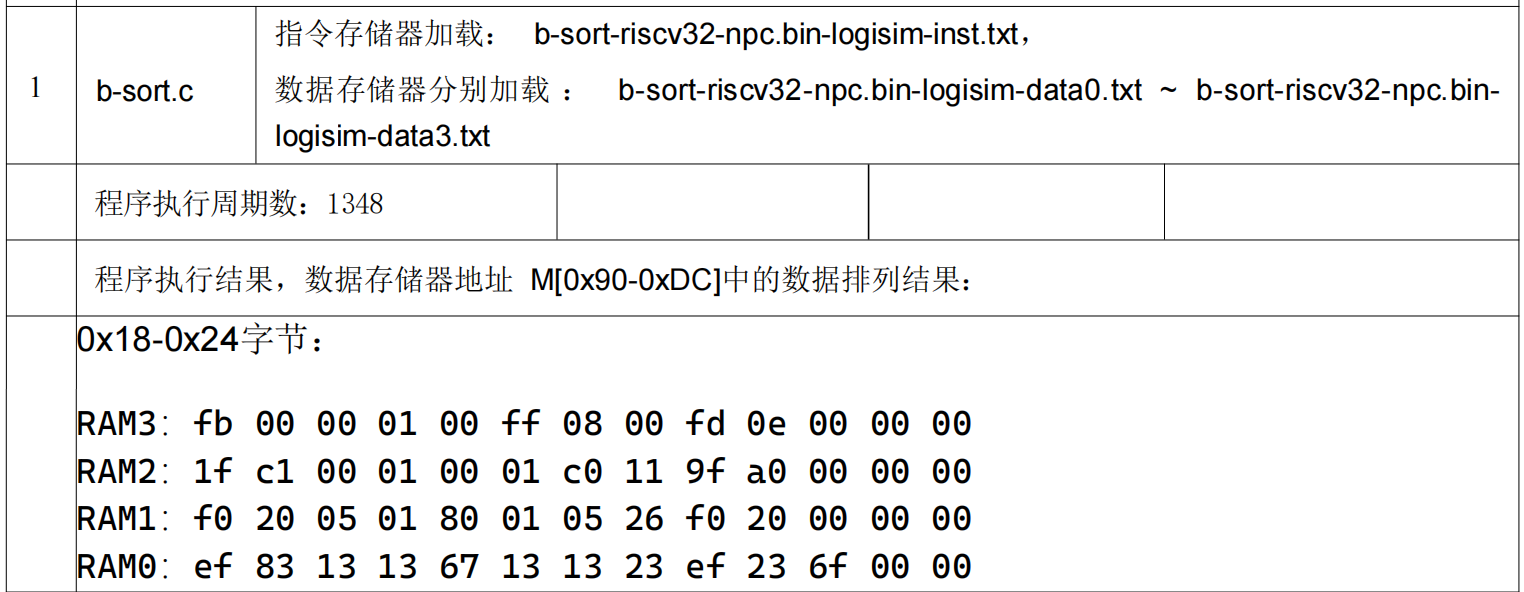
（1）原理

使用g++-riscv64-linux-gnu把C语言程序转化为汇编指令，生成指令文件和存储数据文件，直接在电路中可运行。由于安装risc-v gcc工具链有困难，下面用线下验收的C Test - offine文件夹中的指令和数据进行验证。

（2）实验验证

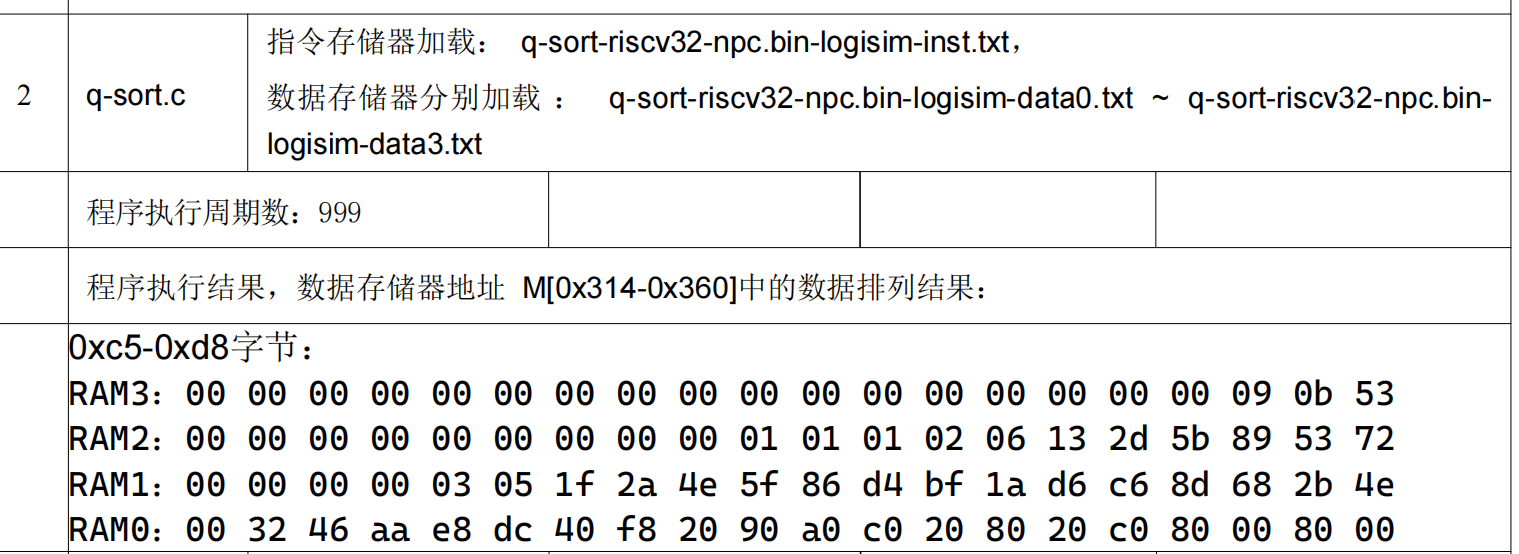
i. b-sort.c程序运行

指令存储器加载b-sort-riscv32-npc.bin-logisim-inst.txt文件，四个RAM加载b-sort-riscv32-npc.bin-logisim-data3.txt~b-sort-riscv32-npc.bin-logisim-data0.txt。输出结果如下，同样在[线下验收表](实验六 线下验收表_fin.pdf)中：



ii. q-sort.c程序运行

同上，加载对应文件即可，输出结果如下，同样在[线下验收表](实验六 线下验收表_fin.pdf)中：



**6.7 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。

可以通过**时间分片或切换运行**来模拟并发执行。本质上是将CPU的执行时间分配给多个任务，在任务之间切换，使得每个任务看起来是同时运行的。

下面以同时执行计算累加和与数据排序两个程序为例：

基本思路是，首先定义独立运行的两个或多个任务；然后调度器按固定时间间隔切换任务，通过保存和恢复任务状态，实现任务切换。

伪代码如下：

def scheduler ([ ]tasks) :

for task in tasks : // 遍历所有任务

try:

next(task) // 执行当前任务的一部分

except StopIteration:

tasks.remove(task) // 移除已完成的任务

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）**添加适当的输入输出设备接口：**比如提供键盘控制器，用于检测按键状态并将按键编码传送到数据总线上；针对TTY（Teletypewriter）输出，可以提供一个显示控制器，将字符数据发送到输出设备（如显示屏或打印机）。实现缓冲区，以便处理数据流。

2）在总线上为每个设备分配唯一的I/O地址。通过地址总线选择设备，通过数据总线与设备交换数据。

3）指令集中增加特定的指令集进行I/O操作

4）处理指令：在控制单元中识别并处理指令，根据指令生成对应的执行和输入输出信号，再进行读写操作。I/O设备通过内存与CPU交互，数据通常先被读入内存，然后CPU再从内存中读取数据进行处理。

3. 阐述如何在单周期CPU基础上实现多周期CPU和流水线CPU？

**· 多周期CPU：**通过将指令拆分成多个阶段，解决了单周期CPU中时钟周期过长的问题，提高了资源利用率。

多周期CPU通过将每条指令的执行划分为多个时钟周期完成，解决了单周期CPU中时钟周期过长的问题。

（1）设计思路

将指令的执行分解为多个步骤（如取指、译码、执行、访存、写回等），在每个时钟周期内完成一个步骤。

（2）可能的调整

利用有限状态机，根据当前阶段生成控制信号。增加寄存器（如阶段寄存器）以保存阶段间的数据。

（3）实现步骤

将指令执行划分为若干阶段。例如：从内存中读取指令；解析指令，生成控制信号；进行运算或计算地址；访问数据存储器；将结果写回寄存器。

使用一个有限状态机控制在每个时钟周期转移到下一阶段的执行流程

**· 流水线CPU：**在多周期CPU的基础上，通过并行化多个指令的执行阶段，实现了更高的吞吐率，但增加了硬件和控制逻辑的复杂性。

流水线CPU通过将多条指令分阶段重叠执行，实现指令的并行处理，提高了吞吐率。

（1）设计思路

类似于工厂中的流水线，将指令执行过程划分为多个阶段。在每个时钟周期内，流水线中的每个阶段同时处理不同的指令。各阶段可以同时工作，时钟周期受限于最慢阶段。

（2）可能的调整

增加流水线寄存器，在各阶段之间保存中间结果。为每个阶段提供独立的硬件资源（如独立的加法器数值存储）。生成阶段性控制信号，协调不同指令在流水线中的执行，但是需要添加冲突检测和处理机制。

（3）实现步骤

与多周期CPU类似，划分阶段同上，在每两个阶段之间插入寄存器，用于保存上一阶段的输出。

但是可能出现数据和控制冒险：

数据冒险是，前一条指令的结果未写回寄存器，而下一条指令已经使用该结果。这时可能需要旁路（转发）技术或流水线暂停。

控制冒险是，分支指令导致流水线需要重新调整。这是需要分支预测或延迟槽。