# 《单周期CPU设计与测试》实验报告

姓名：许霁烨

学号：211275024

## 控制器设计实验

### 实验整体方案的设计

控制器的主要功能是根据指令的的**操作码 opcode**、**功能码 func3** 和**功能码 func7** 来生成对应的控制信号。控制信号的具体表格如下：

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 RV321 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 |

下面将对根据上述的表格设计每个控制信号的逻辑电路。

## ExtOp

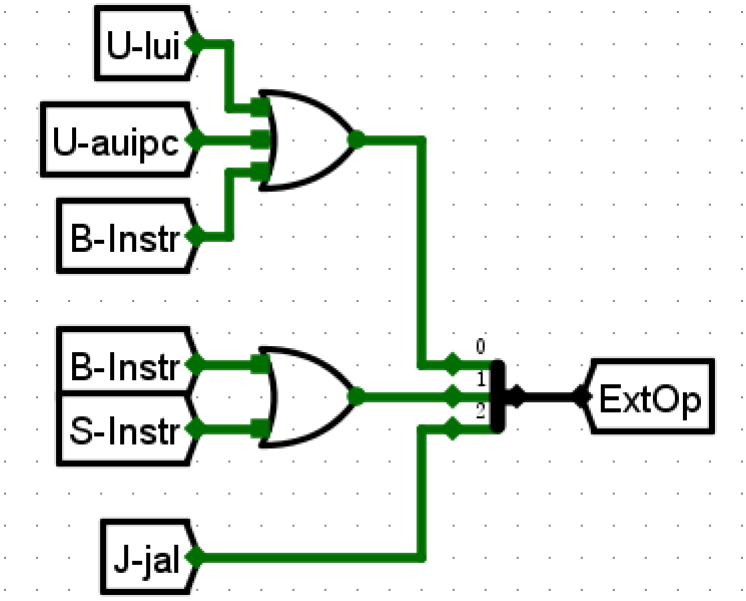
可以发现;

ExtOp[2]:仅有jal是1

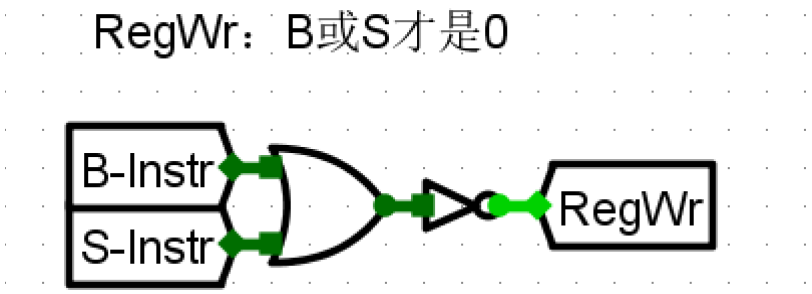
ExtOp[1]:B或S是1

ExtOp[0]:lui auipc B是1

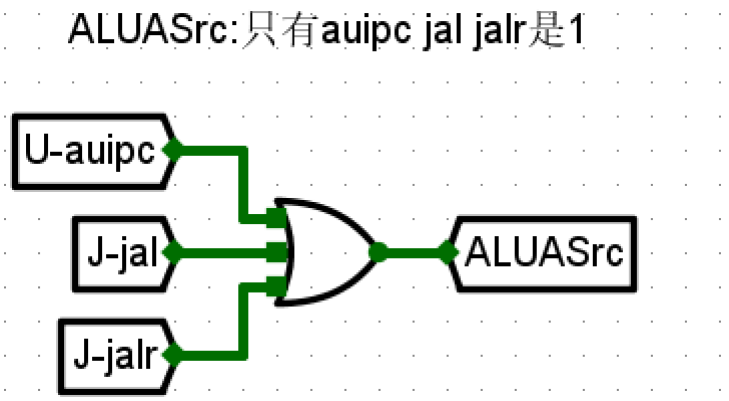
因此设计电路



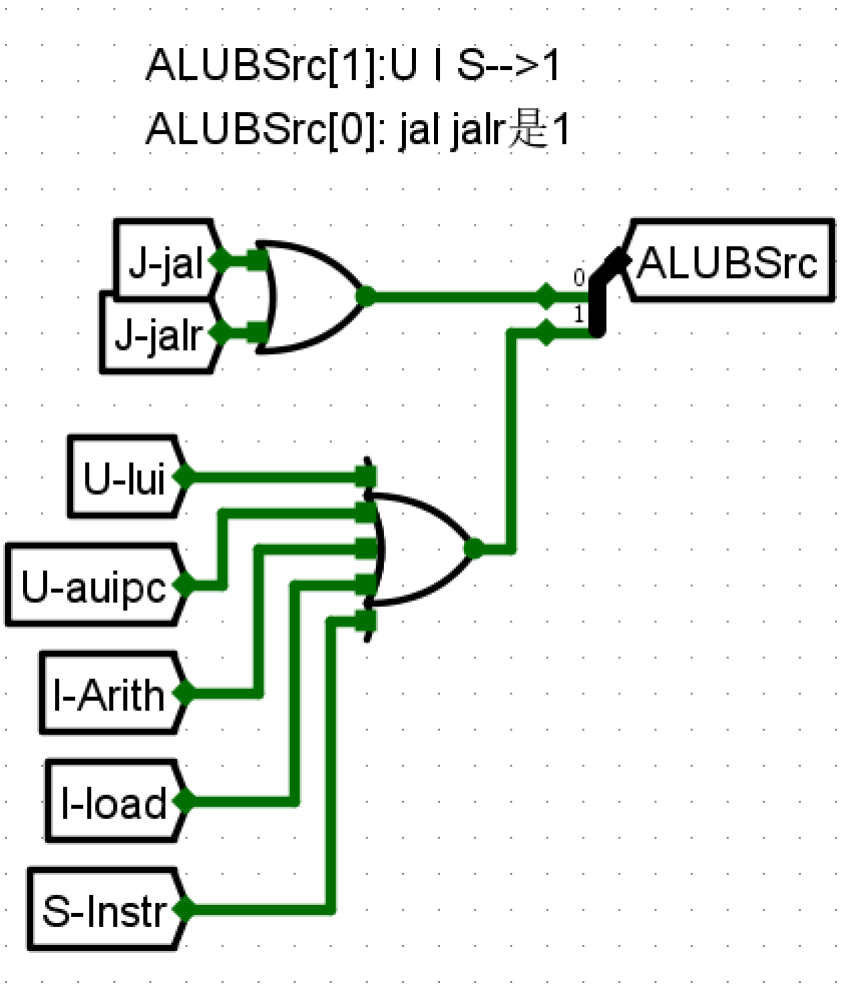

## RegWr




## ALUASrc



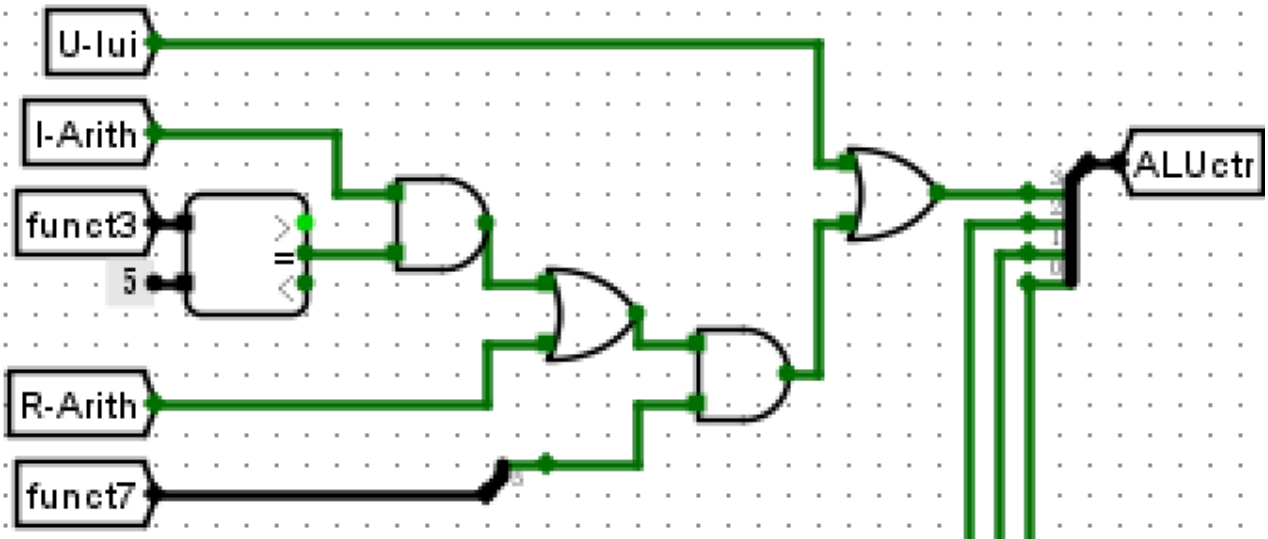
## ALUBSrc


ALUBSrc[1]:U I S-->1
ALUBSrc[0]: jal jalr是1
J-jal
J-jalr
0
1
ALUBSrc
U-lui
U-auipc
I-Arith
I-load
S-Instr

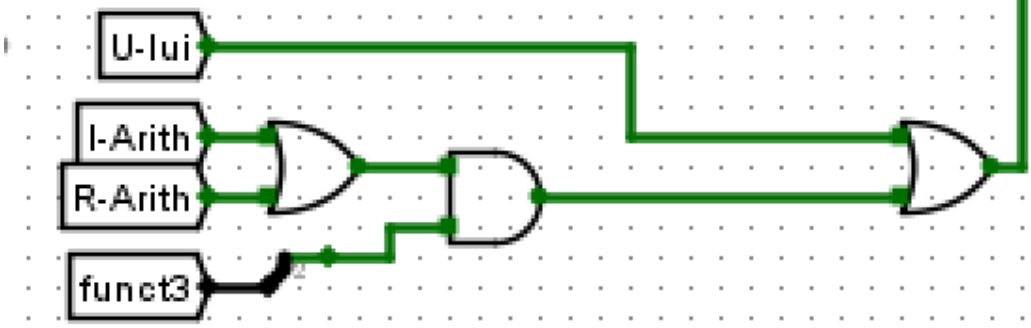
## ALUctr

ALUctr:

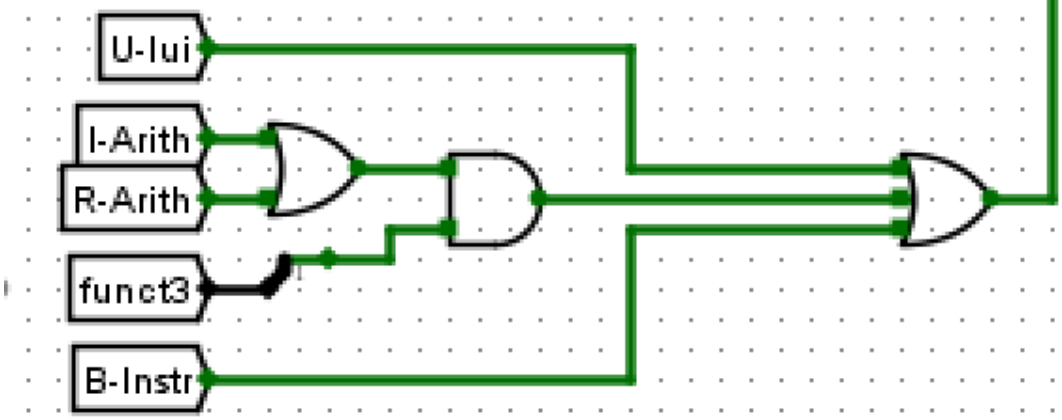
ALUctr[3]: lui (I+func3=101或R)&func75=1




ALUctr[2] lui 或IArith RArith中func3[2]==1的

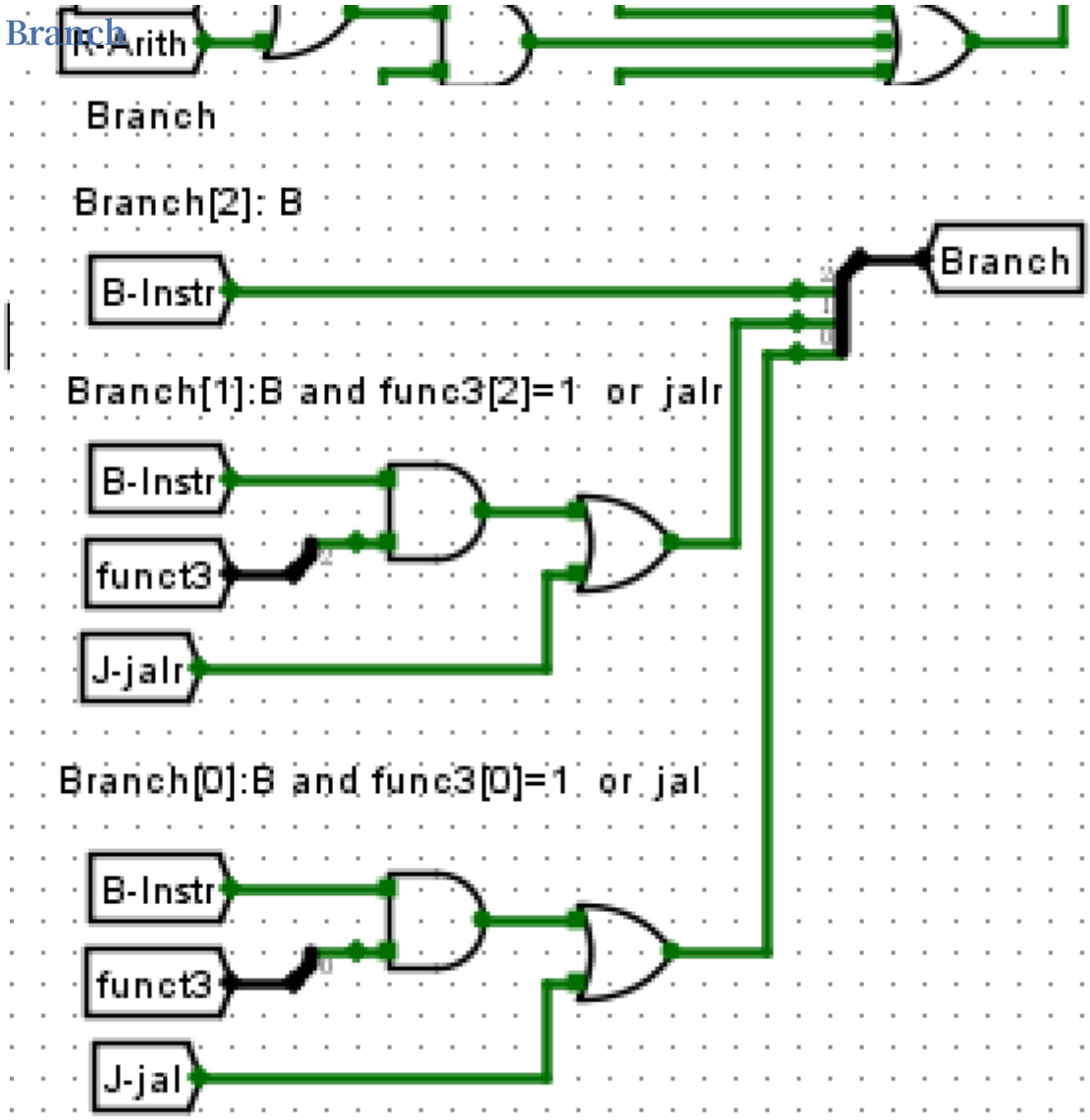



ALUctr[1] lui 或IArith RArith中func3[1]==1 或 B

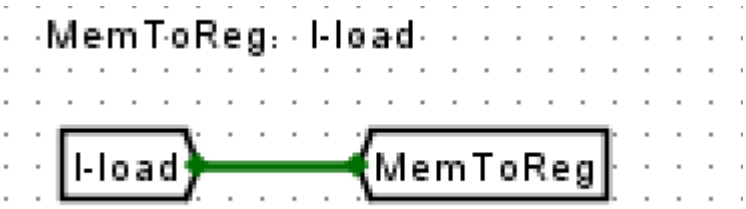



ALUctr[0] lui 或IArith RArith中func3[0]==1 或 Band func3[2:1]==11

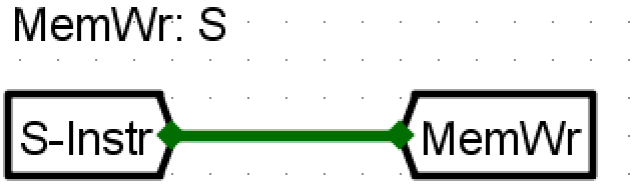
## Branch



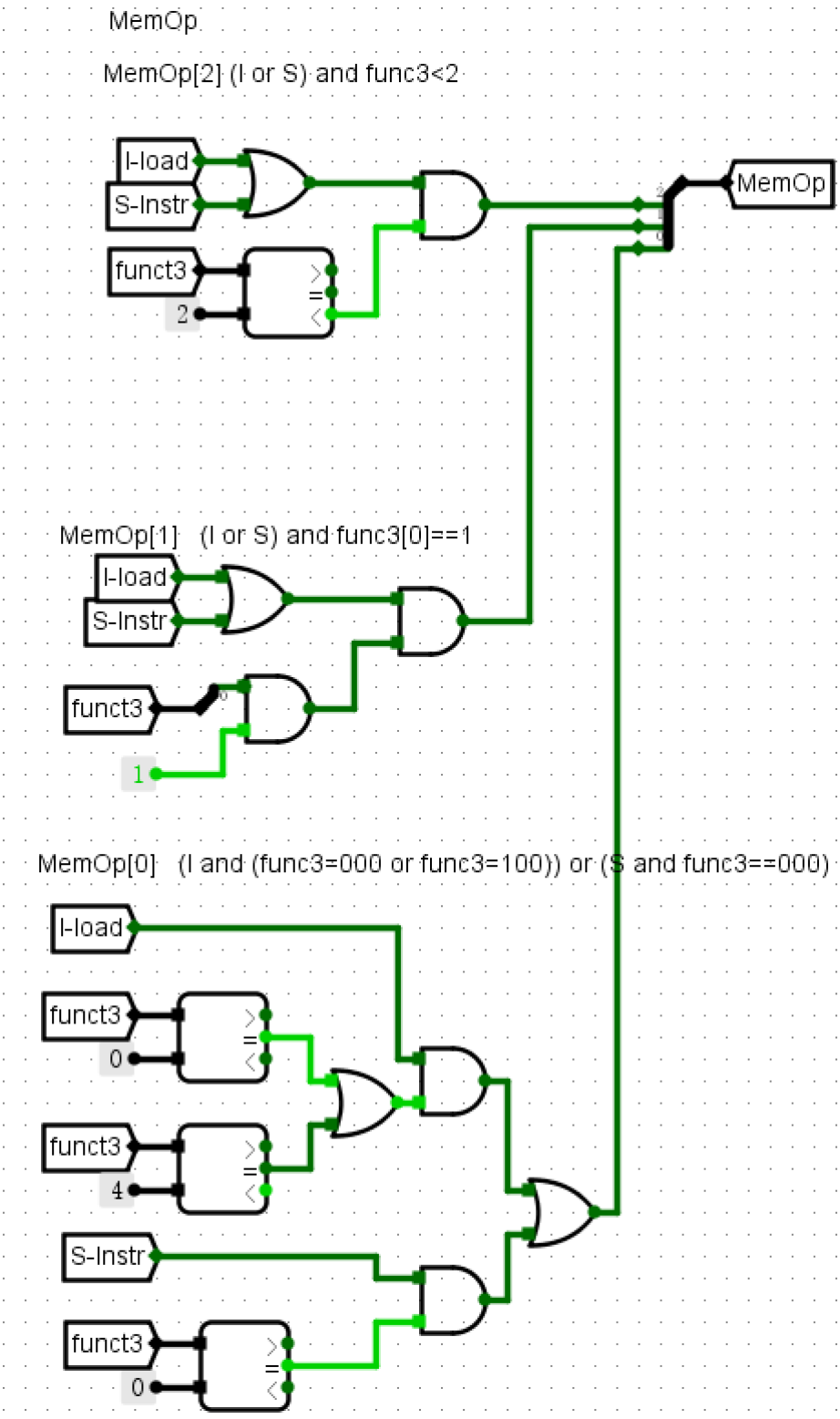

## MemToReg




## MemWr

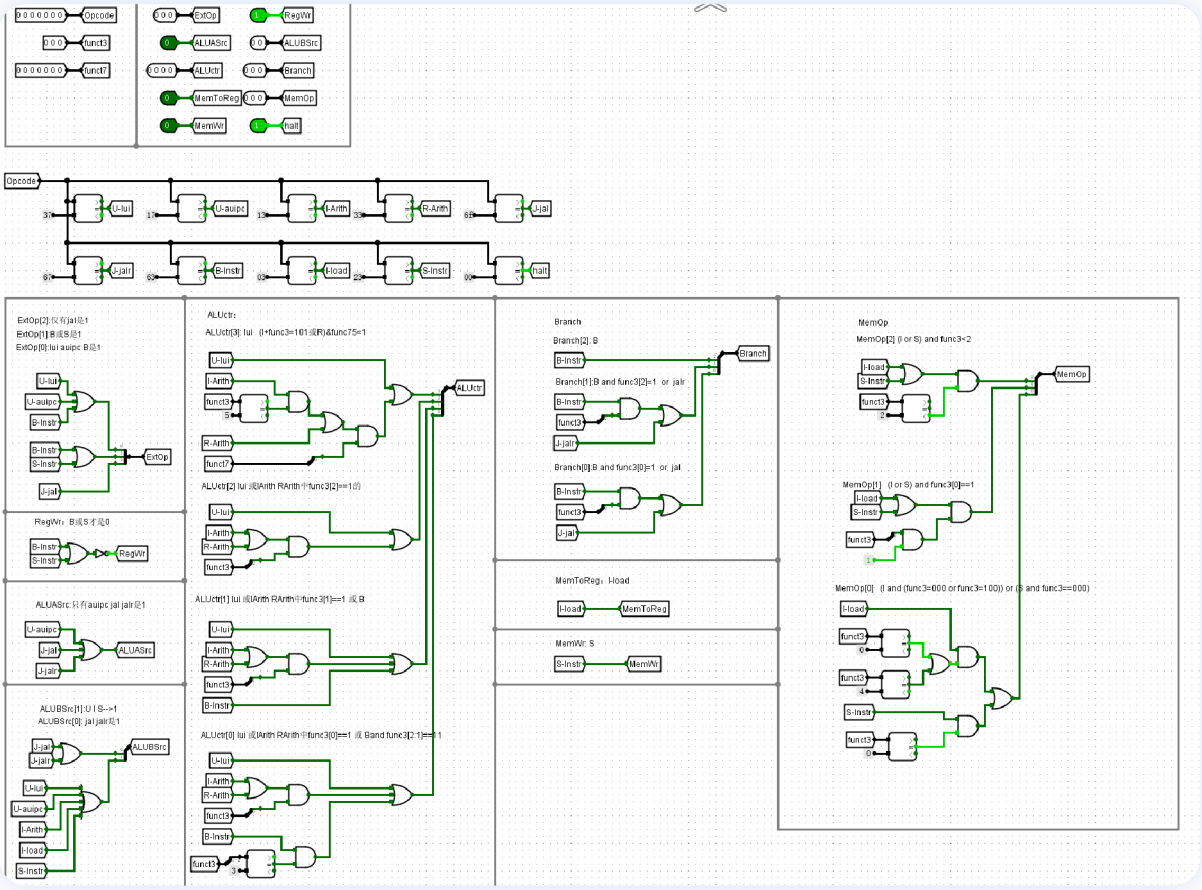


## MemOp

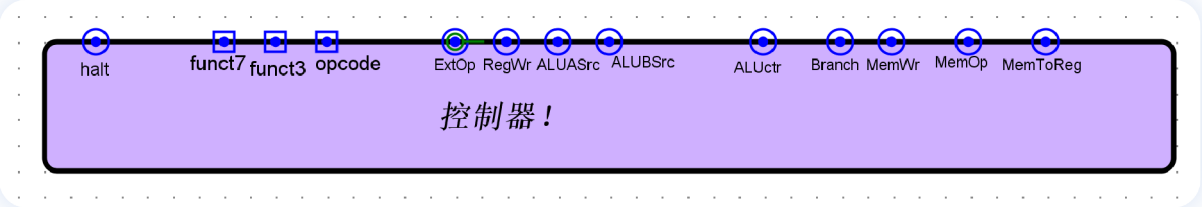



## 电路设计及运行结果

电路总体设计如下：




电路封装如下：



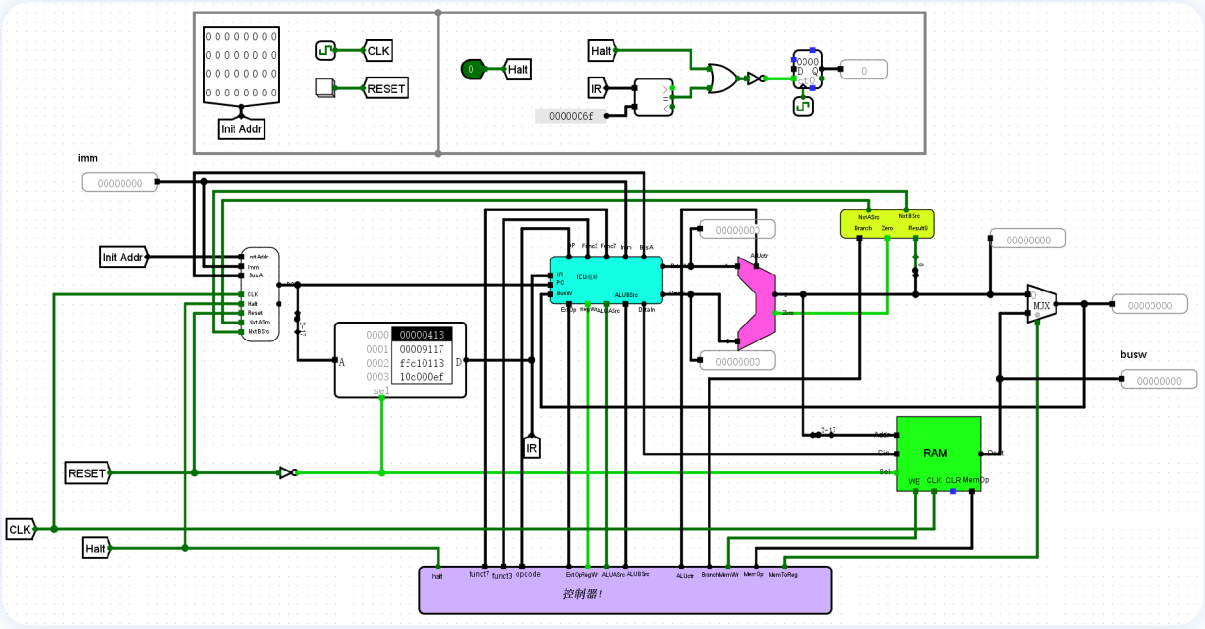

# 单周期CPU设计实验

## 实验整体方案设计

实验本身较为简单，就是在实验5的基础上，通过从指令取出opcode func3 func7输入控制器，在将相应的控制信号输出到正确位置，即可完成单周期CPU设计。

## 电路设计及运行结果

电路设计如下：




# 用累加和程序验证 CPU 设计

本题主要是的功能是实现累加器

## 编写汇编语言程序

这里不过赘述，实验要求中已经写好了

```
main:
    lw a0,0(x0)          # 从数据存储器地址 0x0000 单元中读取参数 n 到寄存器 a0；
    addi a2, x0,1        # 循环变量 i，存放在 a2，初值为 1
    add a3,x0,x0         # 累计和存放在 a3，初值为 0
loop:
    add a3, a3, a2       # 将 a3=a3+i
    beq a2, a0, finish   # 若 i=n，则跳出循环
    addi a2, a2, 1       #  i++
    jal x0, loop         # 无条件跳转到 loop 执行
finish:
    sw a3, 8(x0)         # 将累加结果保存到数据存储器 0x0008 单元
end:
    jal x0, end          # 无条件跳转到 end
```

写好并将其保存为asm文件

> ASM 文件是一种文本文件，其中包含了汇编语言的源代码。ASM 是 Assembly Language（汇编语言）的缩写。

## 将汇编语言程序转换成机器代码

主要流程如下：

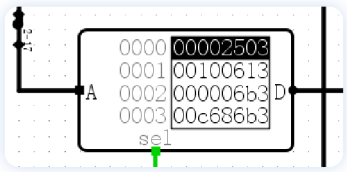
1. 打开sum.asm文件，并更改setting--Memory Configuration--Compact，Data at Address 0
2. assemble并将Value(+0)改为0x64（也就是100）
3. 运行程序，可以看见结果是13ba（也就是5050），证明了**代码正确性**

| Address | Value (+0) | Value (+4) | Value (+8) |
|---|---|---|---|
| 0x00000000 | 0x00000064 | 0x00000000 | 0x000013ba |
| 0x00000020 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x00000040 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x00000060 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x00000080 | 0x00000000 | 0x00000000 | 0x00000000 |

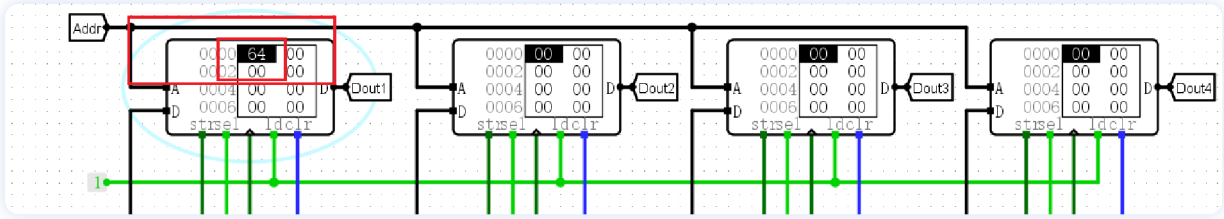
4. 导出机器代码为hex文件
5. 在hex文件用记事本打开，并添加第一行："v2.0 raw"

## 在设计的电路中验证程序

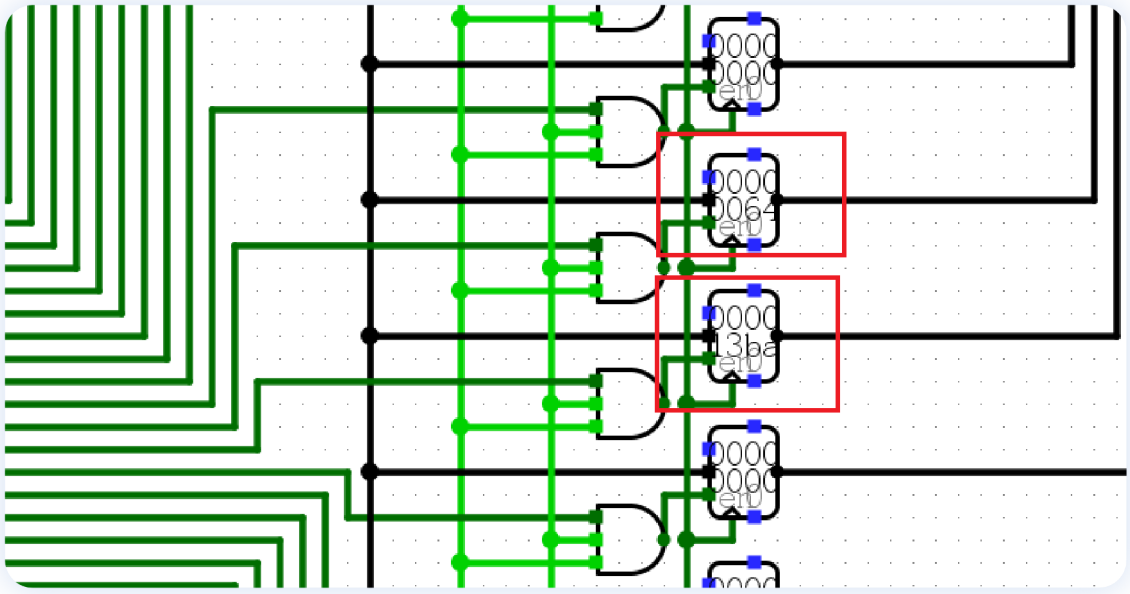
1. 在指令寄存器中加载指令



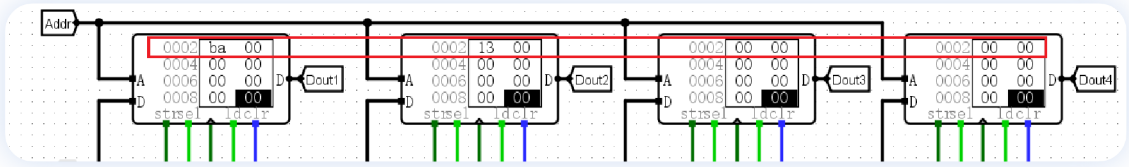

2. 在数据存储器的第一位输入64（100）




3. 运行程序
   - 可以看到，CPU在执行了402条指令后停止
   - 同时，在寄存器中可以看到，**a2存放了64，a3存放了13ba，分别是N和sum，可以说明实验的正确性**

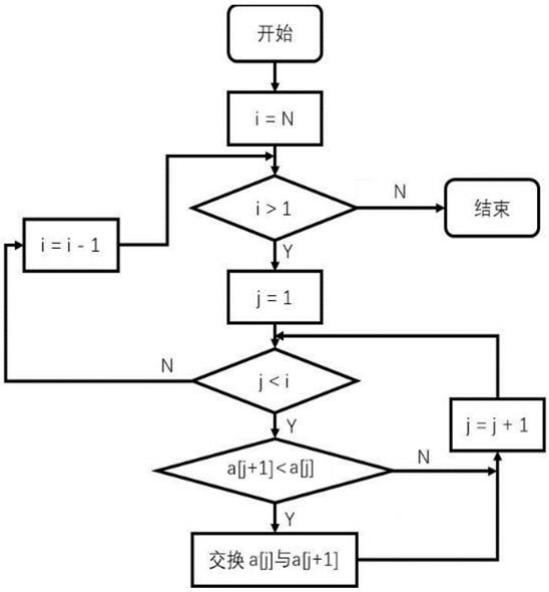



- 在**数据存储器**中也可以看到，在02地址（也就是0x0008）存放了实验结果




## 用冒泡排序程序进行 CPU 设计验证

本任务的主要任务就是采用**冒泡排序**对有限个数据按照从小到大的顺序排列，主要原理和代码如下图所示：




```
#冒泡排序算法 RV32I 汇编程序
    lw   a0,0(x0)   #a0,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,a0,x0   #a2，保存 i，初始值为 i=N
L1:
    add   a3,a1,x0 #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
    bgeu a7,a6,L4    #a[j]>=a[j+1] 跳转
    sw a7,0(a4)       #交换存储
    sw a6,4(a4)       #交换存储
L4:
    add a3,a3,a1      #j=j+1
    bltu   a3,a2,L2    # if j<i then 循环 读取两个元素比较
L3:
    sub a2,a2,a1       #i--
    bne   a2,a1,L1    #if i>1 then 循环 else 则结束
finish:
    jal x0, finish
```

## 编写汇编语言程序并转换成机器代码

与上一节的主要流程相同，将汇编语言程序转换成机器代码 `bubble.hex`，代码内容如下图所示：

```
v2.0 raw
00002503
00100593
00050633
000586b3
00269713
00072803
00472883
0108f663
01172023
01072223
00b686b3
fec6e2e3
40b60633
fcb61ce3
0000006f
```

## 编写待排序文件

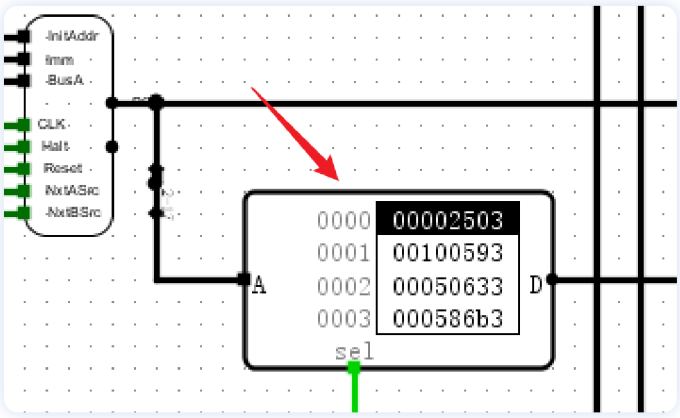
用记事本编辑待排序文件，写入

```
v2.0 raw
a 8 41 2 12 36 6 9 5 5b 7
```

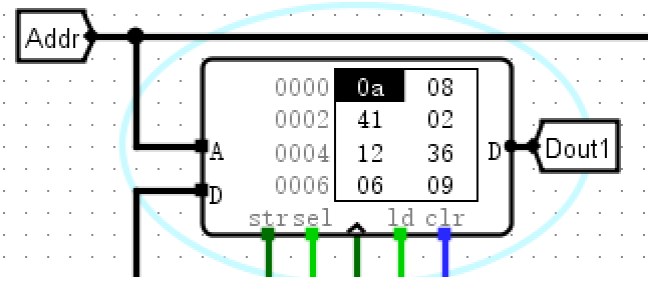
并将其转换成 `.bat` 文件

## 在设计的电路中验证程序

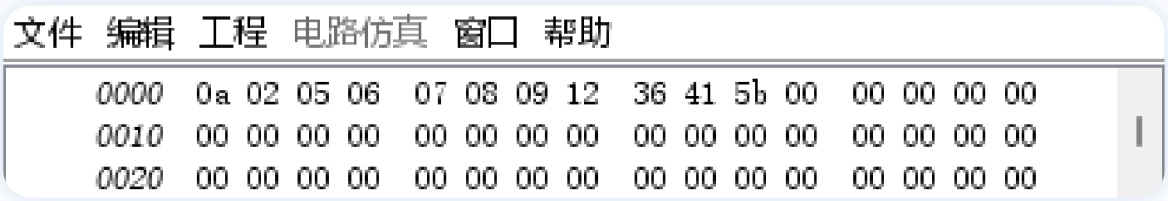
1. 在指令寄存器中加载 `bubble.hex`

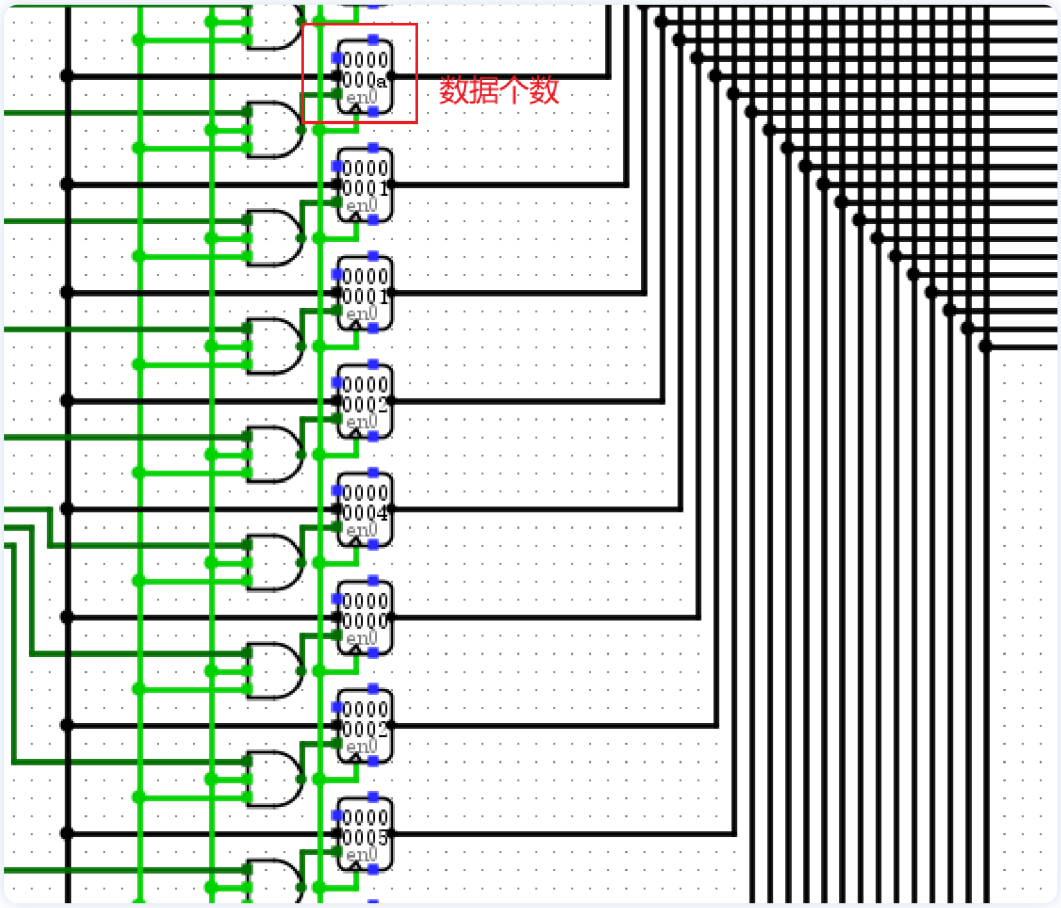



2. 在RAM中的最低位加载 `bubble.bat`




3. 运行程序，可以看到程序在运行346条指令后停止，在RAM中观察数据，可以看到，数据已经成功变成了从小到大排列，在寄存器中也有正确的数据

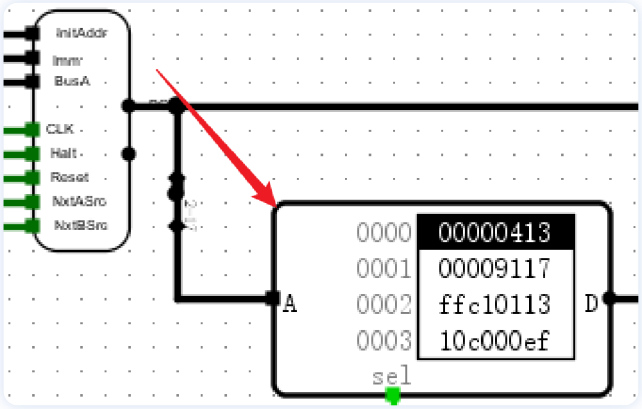




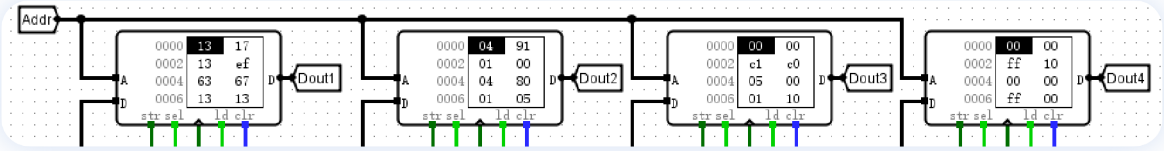

## C程序汇编测试

本任务是用的C Test目录下提供的文档，主要流程如下：

1. 将 `bubble-sort-riscv32-npc.bin-logisim-inst.txt` 文件加载到指令存储器

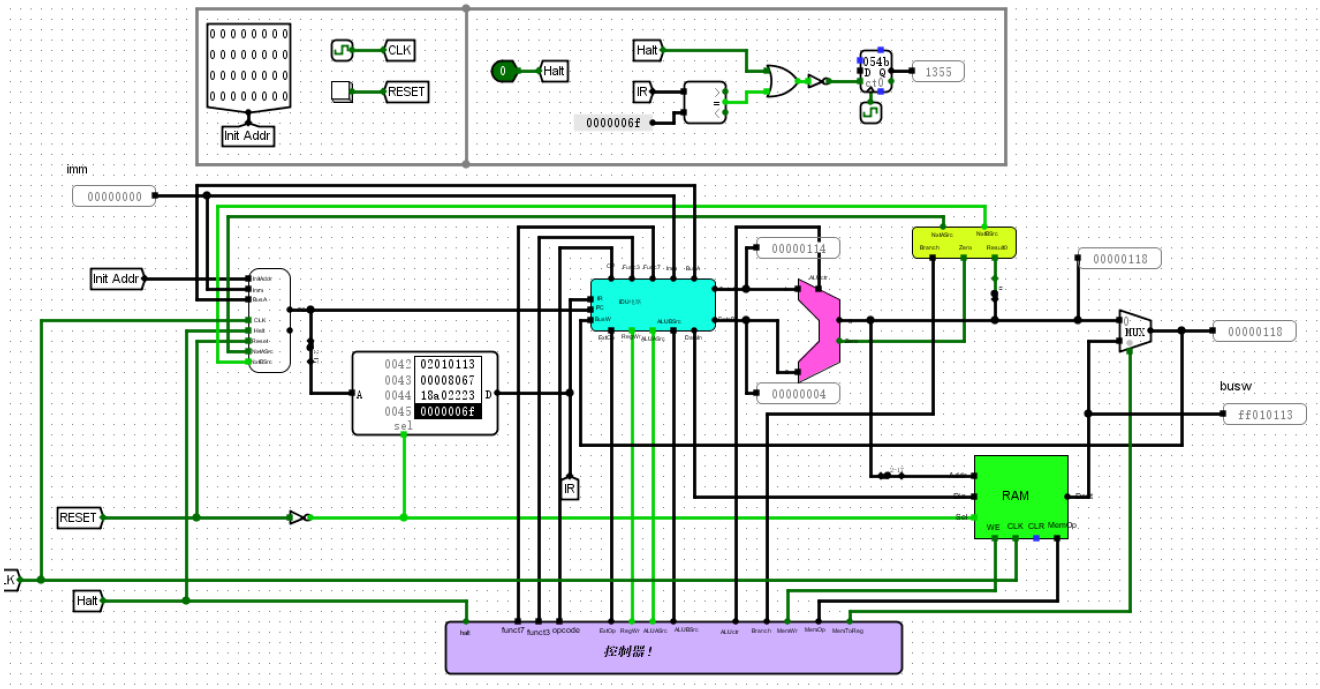



2. 将 `bubble-sort-riscv32-npc.bin-logisim-data0` 到 `bubble-sort-riscv32-npc.bin-logisim-data3` 依次加载到存储器的最低位到最高位

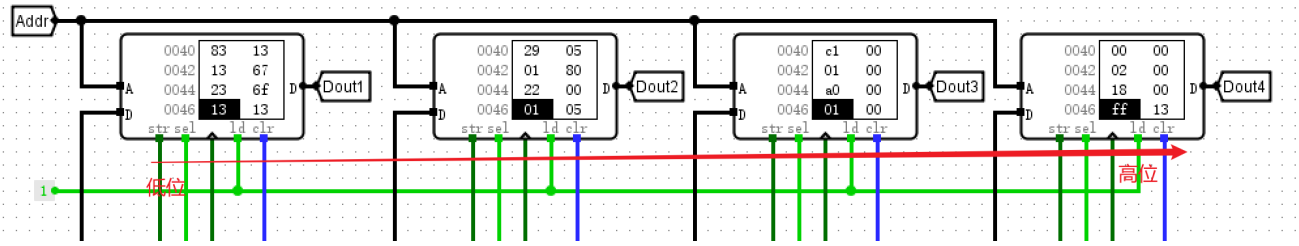


## 在设计的电路中验证程序

可以看到，程序在运行1355条指令后停止工作




数据存储器中的排序后的数据如下，可以看出，从上往下的数据在变大，证明了算法的正确性。




## 思考题

1. 在累加和计算程序中，添加溢出判读语句，并把最大的不溢出累计和以及累加序数保存到数据存储
   器中输出。
   - 在累加和的计算过程中，使用一个变量（例如 `sum` ）来保存累加和的值。
   - 在每次累加时，检查 `sum` 是否溢出。如果溢出，将溢出标志设置为1，并将当前累加和的值保存到数据存储器中的某个位置。
   - 同时，使用一个变量（例如 `max_sum` ）来保存当前的最大累计和值。
   - 在每次累加完成后，将 `sum` 与 `max_sum` 进行比较，如果 `sum` 大于 `max_sum` ，则更新 `max_sum` 的值，并将累加序数保存到数据存储器中的另一个位置
2. 如果分支跳转指令不在 ALU 内部使用减法运算来实现，而是在 ALU 外使用独立比较器来实现，说说单周期 CPU 的电路原理图中需要做哪些修改?

- 在指令译码阶段，需要增加一个比较器单元，用于比较两个操作数是否满足分支条件。
- 在分支跳转指令的执行阶段，将比较器的输出与分支控制逻辑进行连接，根据比较结果决定是否进行分支跳转。
- 修改控制单元，使其能够控制比较器的操作和分支跳转的控制信号。

3. 实现单周期 CPU 后，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。
   - 添加适当的输入输出接口电路，用于与键盘、TTY等外部设备进行通信。
   - 在计算机系统的总线上设置相应的输入输出地址空间。
   - 在单周期 CPU 的指令集中，增加特定的指令或者指令格式，用于进行输入输出操作。
   - 修改控制单元，使其能够识别并处理输入输出指令。
   - 在控制单元中，根据输入输出指令的执行时机，生成相应的输入输出控制信号。
   - 在输入输出接口电路中，根据输入输出指令的控制信号，进行数据的读取或写入操作。
4. 如果需要实现 5 级流水线 RV32I CPU，则如何在单周期 CPU 基础上进行修改?
   - 将单周期 CPU 的各个阶段（取指、译码、执行、访存、写回）细化为更小的流水线阶段，例如取指阶段细化为取指令、分支预测、分支决策等子阶段。
   - 为每个流水线阶段添加相应的寄存器，用于存储该阶段的中间结果。
   - 在每个流水线阶段之间添加适当的流水线寄存器，用于传递数据和控制信号。
   - 修改控制单元，使其能够控制流水线各个阶段的操作和流水线寄存器的读写。
   - 解决流水线冒险问题，如结构冒险、数据冒险和控制冒险，通过插入气泡（空操作周期）、数据旁路、分支预测等技术来解决冲突和提高流水线效率。
   - 在设计过程中要考虑流水线的时序和控制逻辑，并进行适当的优化，以提高流水线的性能和吞吐量。