实验三报告

231220088 陈翔宇

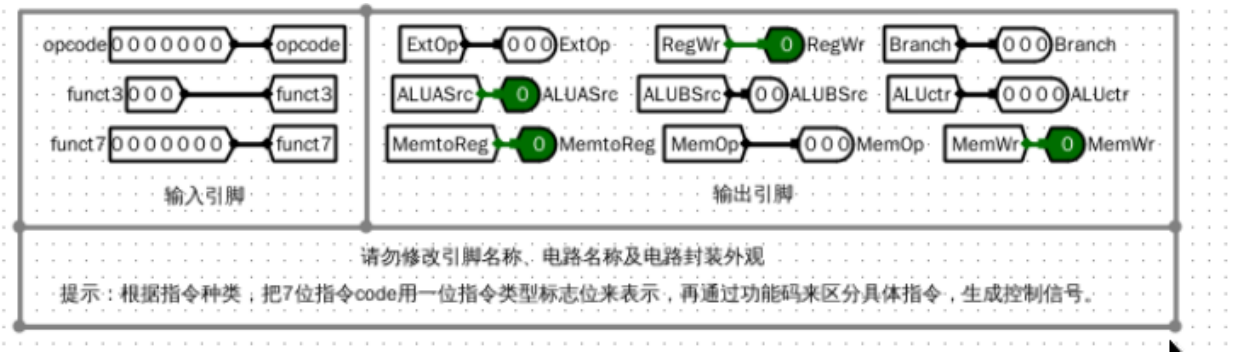
实验内容

一、*单周期cpu控制器*

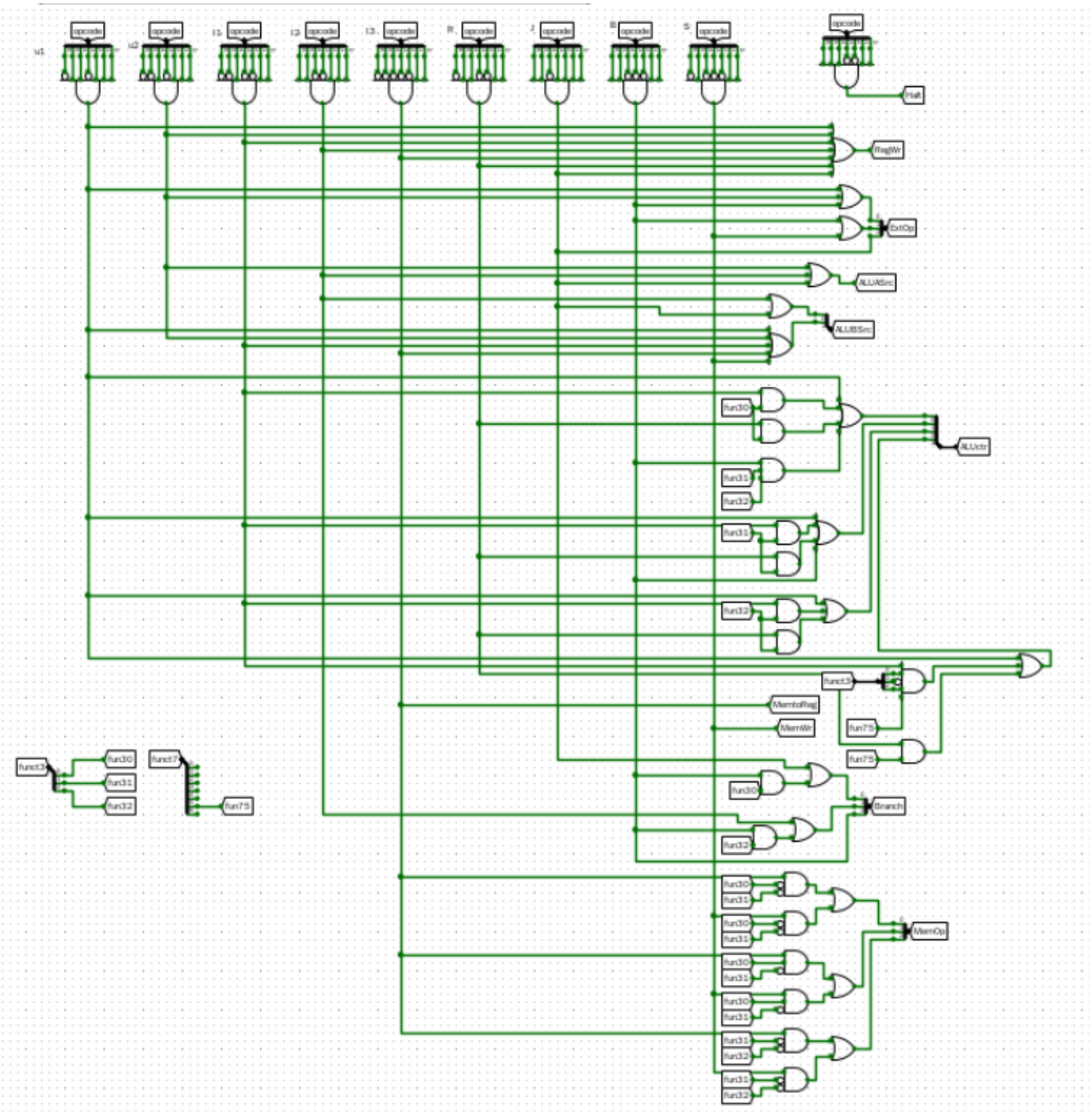
整体方案设计

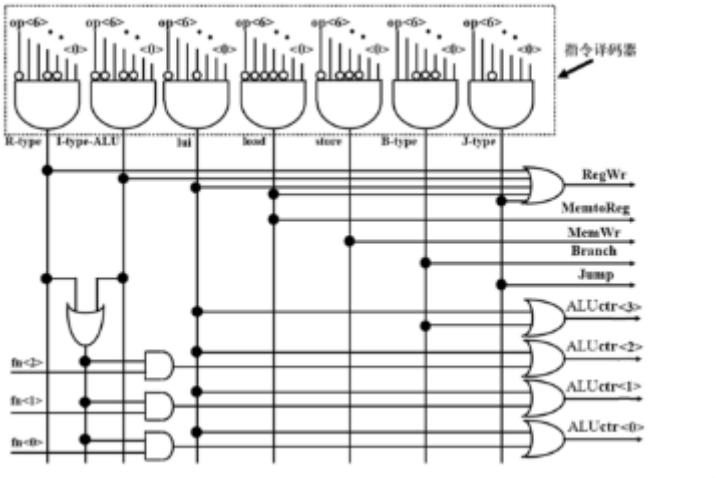
1. 输入输出引脚

   在确定具体指令后生成每个指令对应的控制信号，来控制数据通路部件进行对应的动作。控制信号生产部件根据指令代码中的操作码opcode、功能码func3和功能码func7来生成对应的控制信号的。

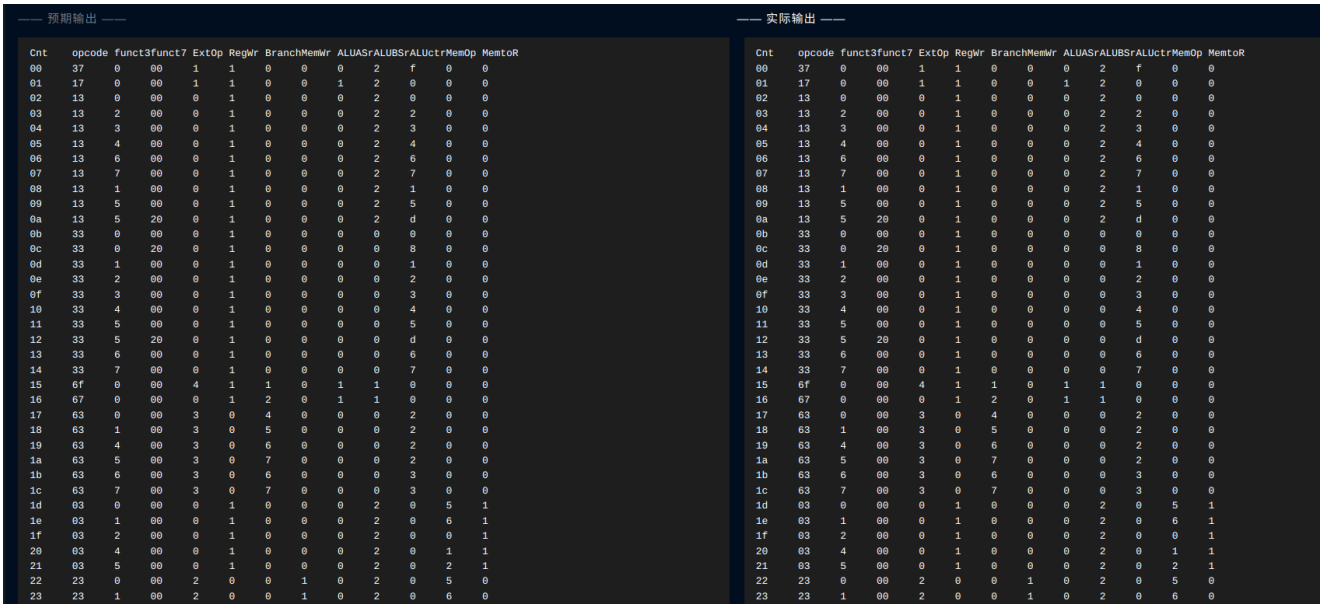


电路图 和 原理图


opcode
Halt
RegWr
ExtOp
ALUASrc
ALUBSrc
ALUctr
MemtoReg
MemWr
Branch
MemOp
funct3
fun7
fun30
fun31
fun32


op<6>
op<5>
op<4>
op<3>
op<2>
op<1>
op<0>
指令译码器
R-type
I-type-ALU
lui
load
store
B-type
J-type
RegWr
MemtoReg
MemWr
Branch
Jump
ALUctr<3>
ALUctr<2>
ALUctr<1>
ALUctr<0>
fn<2>
fn<1>
fn<0>

仿真测试



—— 预期输出 ——

Cnt	opcode	funct3	funct7	ExtOp	RegWr	BranchMemWr	ALUASrc	ALUBSrc	ALUctr	MemOp	MemtoR
00	37	0	00	1	1	0	0	0	2	f	0	0
01	17	0	00	1	1	0	0	1	2	0	0	0
02	13	0	00	0	1	0	0	0	2	0	0	0
03	13	2	00	0	1	0	0	0	2	2	0	0
04	13	3	00	0	1	0	0	0	2	3	0	0
05	13	4	00	0	1	0	0	0	2	4	0	0
06	13	6	00	0	1	0	0	0	2	6	0	0
07	13	7	00	0	1	0	0	0	2	7	0	0
08	13	1	00	0	1	0	0	0	2	1	0	0
09	13	5	00	0	1	0	0	0	2	5	0	0
0a	13	5	20	0	1	0	0	0	2	d	0	0
0b	33	0	00	0	1	0	0	0	0	0	0	0
0c	33	0	20	0	1	0	0	0	0	8	0	0
0d	33	1	00	0	1	0	0	0	0	1	0	0
0e	33	2	00	0	1	0	0	0	0	2	0	0
0f	33	3	00	0	1	0	0	0	0	3	0	0
10	33	4	00	0	1	0	0	0	0	4	0	0
11	33	5	00	0	1	0	0	0	0	5	0	0
12	33	5	20	0	1	0	0	0	0	d	0	0
13	33	6	00	0	1	0	0	0	0	6	0	0
14	33	7	00	0	1	0	0	0	0	7	0	0
15	6f	0	00	4	1	1	0	1	1	0	0	0
16	67	0	00	0	1	2	0	1	1	0	0	0
17	63	0	00	3	0	4	0	0	0	2	0	0
18	63	1	00	3	0	5	0	0	0	2	0	0
19	63	4	00	3	0	6	0	0	0	2	0	0
1a	63	5	00	3	0	7	0	0	0	2	0	0
1b	63	6	00	3	0	6	0	0	0	3	0	0
1c	63	7	00	3	0	7	0	0	0	3	0	0
1d	03	0	00	0	1	0	0	0	2	0	5	1
1e	03	1	00	0	1	0	0	0	2	0	6	1
1f	03	2	00	0	1	0	0	0	2	0	0	1
20	03	4	00	0	1	0	0	0	2	0	1	1
21	03	5	00	0	1	0	0	0	2	0	2	1
22	23	0	00	2	0	0	1	0	2	0	5	0
23	23	1	00	2	0	0	1	0	2	0	6	0

—— 实际输出 ——

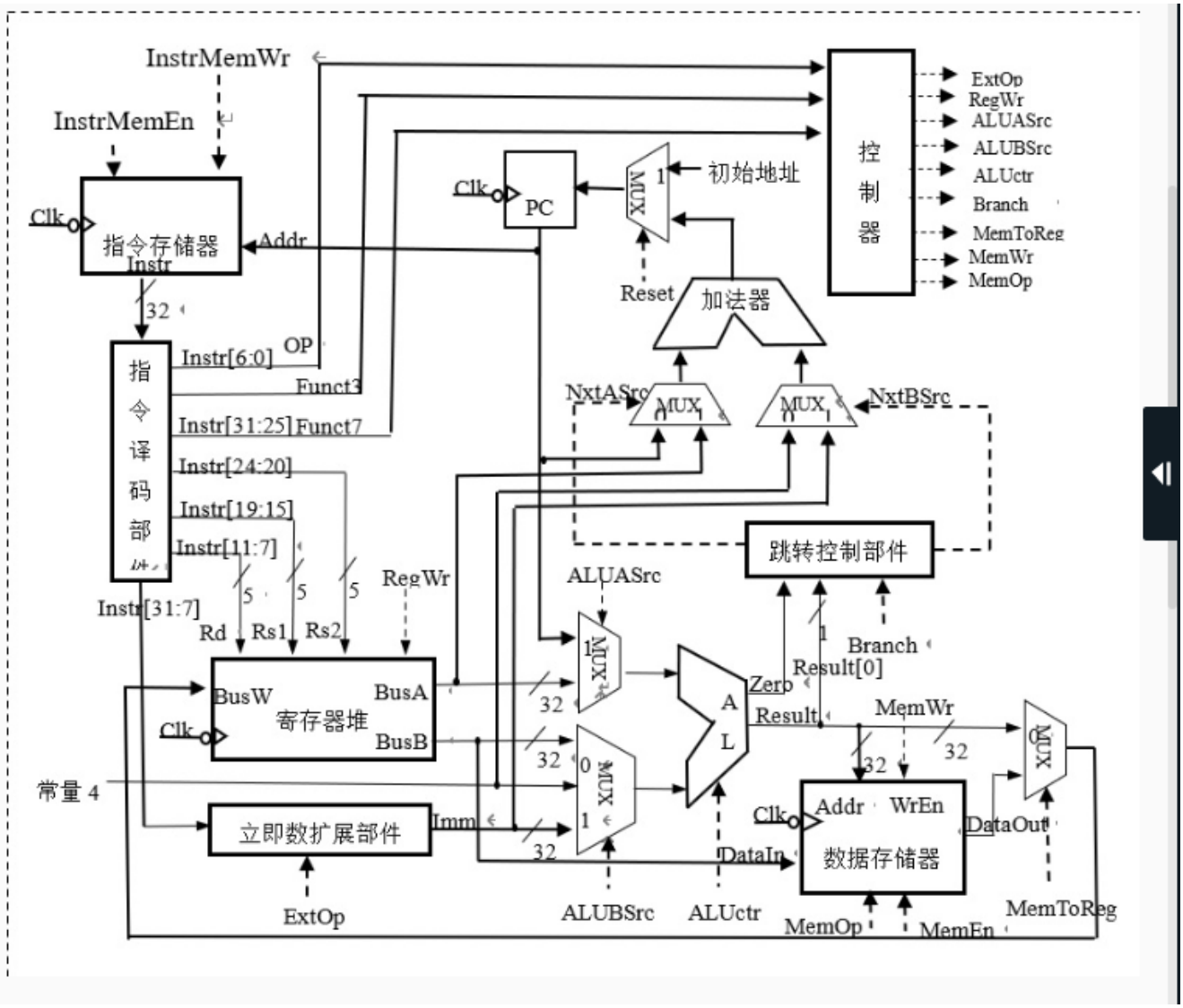
Cnt	opcode	funct3	funct7	ExtOp	RegWr	BranchMemWr	ALUASrc	ALUBSrc	ALUctr	MemOp	MemtoR
00	37	0	00	1	1	0	0	0	2	f	0	0
01	17	0	00	1	1	0	0	1	2	0	0	0
02	13	0	00	0	1	0	0	0	2	0	0	0
03	13	2	00	0	1	0	0	0	2	2	0	0
04	13	3	00	0	1	0	0	0	2	3	0	0
05	13	4	00	0	1	0	0	0	2	4	0	0
06	13	6	00	0	1	0	0	0	2	6	0	0
07	13	7	00	0	1	0	0	0	2	7	0	0
08	13	1	00	0	1	0	0	0	2	1	0	0
09	13	5	00	0	1	0	0	0	2	5	0	0
0a	13	5	20	0	1	0	0	0	2	d	0	0
0b	33	0	00	0	1	0	0	0	0	0	0	0
0c	33	0	20	0	1	0	0	0	0	8	0	0
0d	33	1	00	0	1	0	0	0	0	1	0	0
0e	33	2	00	0	1	0	0	0	0	2	0	0
0f	33	3	00	0	1	0	0	0	0	3	0	0
10	33	4	00	0	1	0	0	0	0	4	0	0
11	33	5	00	0	1	0	0	0	0	5	0	0
12	33	5	20	0	1	0	0	0	0	d	0	0
13	33	6	00	0	1	0	0	0	0	6	0	0
14	33	7	00	0	1	0	0	0	0	7	0	0
15	6f	0	00	4	1	1	0	1	1	0	0	0
16	67	0	00	0	1	2	0	1	1	0	0	0
17	63	0	00	3	0	4	0	0	0	2	0	0
18	63	1	00	3	0	5	0	0	0	2	0	0
19	63	4	00	3	0	6	0	0	0	2	0	0
1a	63	5	00	3	0	7	0	0	0	2	0	0
1b	63	6	00	3	0	6	0	0	0	3	0	0
1c	63	7	00	3	0	7	0	0	0	3	0	0
1d	03	0	00	0	1	0	0	0	2	0	5	1
1e	03	1	00	0	1	0	0	0	2	0	6	1
1f	03	2	00	0	1	0	0	0	2	0	0	1
20	03	4	00	0	1	0	0	0	2	0	1	1
21	03	5	00	0	1	0	0	0	2	0	2	1
22	23	0	00	2	0	0	1	0	2	0	5	0
23	23	1	00	2	0	0	1	0	2	0	6	0

错误现象及分析

在完成实验过程中没有遇到任何错误。

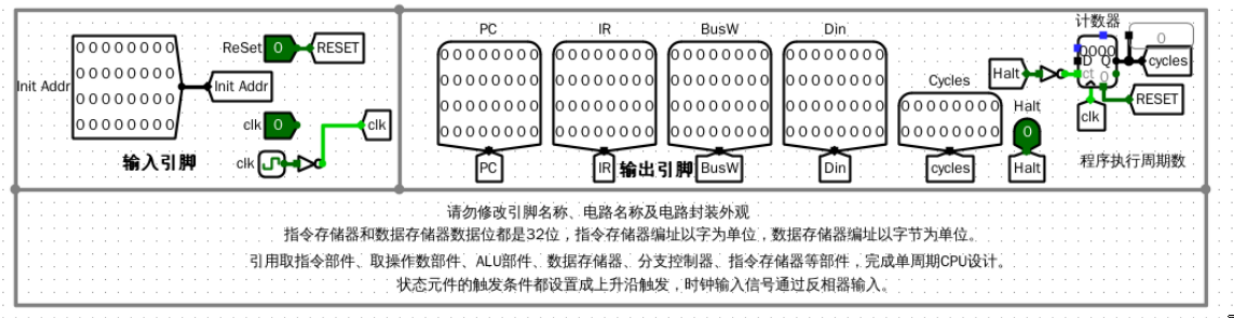
二、单周期cpu

基本原理

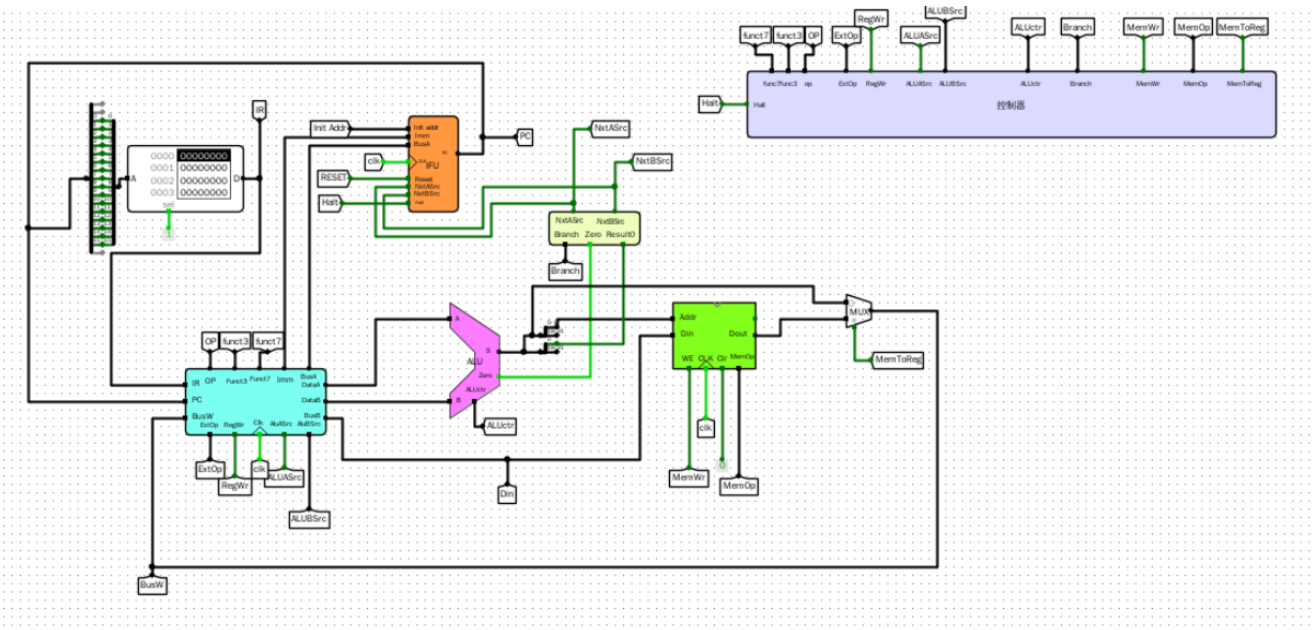


整体方案设计

1. 输入输出引脚



电路图



仿真测试

错误现象及分析

在完成实验过程中没有遇到任何错误。

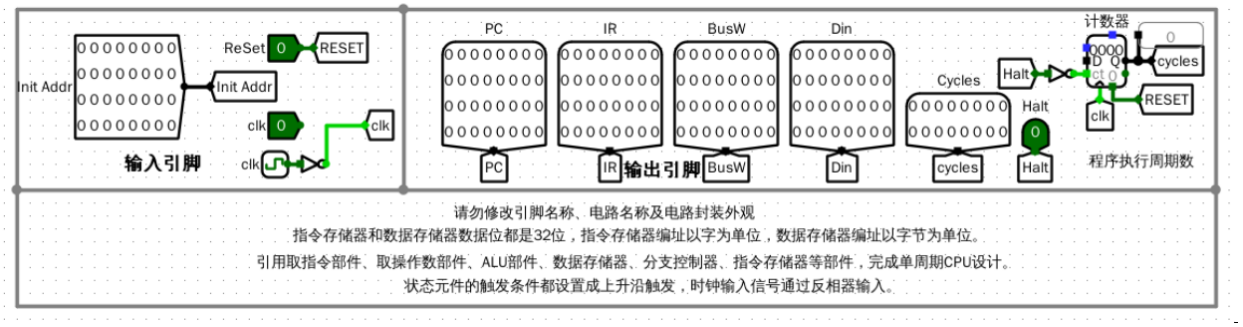
三、计算累加和程序

基本原理

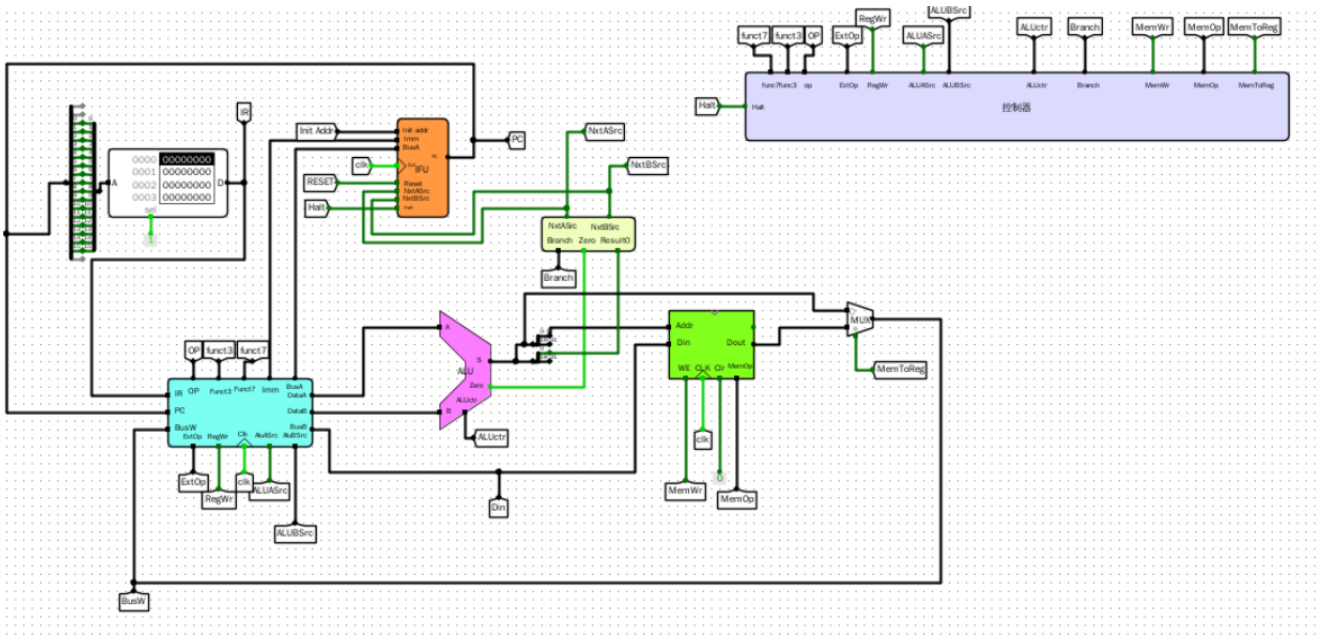
```
#计算累加和程序
.text
main:
  lw  a1,0(x0)       # R[a1]:n,R[a1]<- Mem[0], 读取参数n
  beq a1,x0,fail     # if n=0 goto fail
  ori a2,x0,1        # R[a2]:i,=1
  xor a3,a3,a3       # R[a3]:S,=0
 loop:
      add a3, a3, a2      # R[a3]=R[a3]+R[a2]
    beq a2, a1, finish   # if R[a2]=n goto finish
    addi a2, 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
```

整体方案设计

1. 输入输出引脚



电路图



仿真测试

错误现象及分析

在完成实验过程中没有遇到任何错误。

四、冒泡排序

基本原理

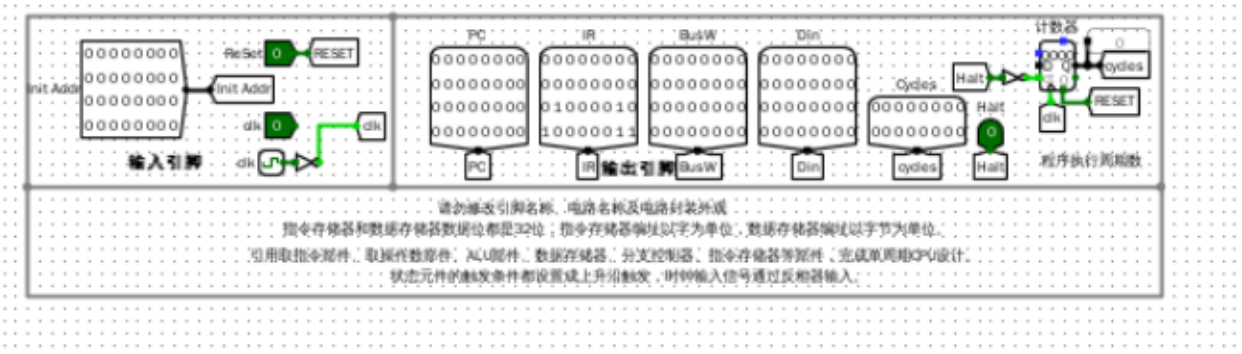
采用冒泡排序对有限数据按照从小到大的顺序排列。冒泡排序算法要点是：对所有相邻记录的关键字值进行比效，如果是逆序（a[j]>a[j+1]），则将其交换，最终达到有序化。其算法基本思想如下：首先，将整个待排序的记录序列划分成有序区和无序区，初始状态有序区为空，无序区包括所有待排序的记录。然后，对无序区从前向后依次将相邻记录的关键字进行比较，若逆序将其交换，从而使得关键字值小的记录向上“冒”（左移），关键字值大的记录向下“落”（右移）。 每经过一趟冒泡排序，都使无序区（左边区域）中关键字值最大的记录进入有序区（右边区域），对于由n个记录组成的记录序列，最多经过n-1趟冒泡排序，就可以将这n个记录按关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

列。

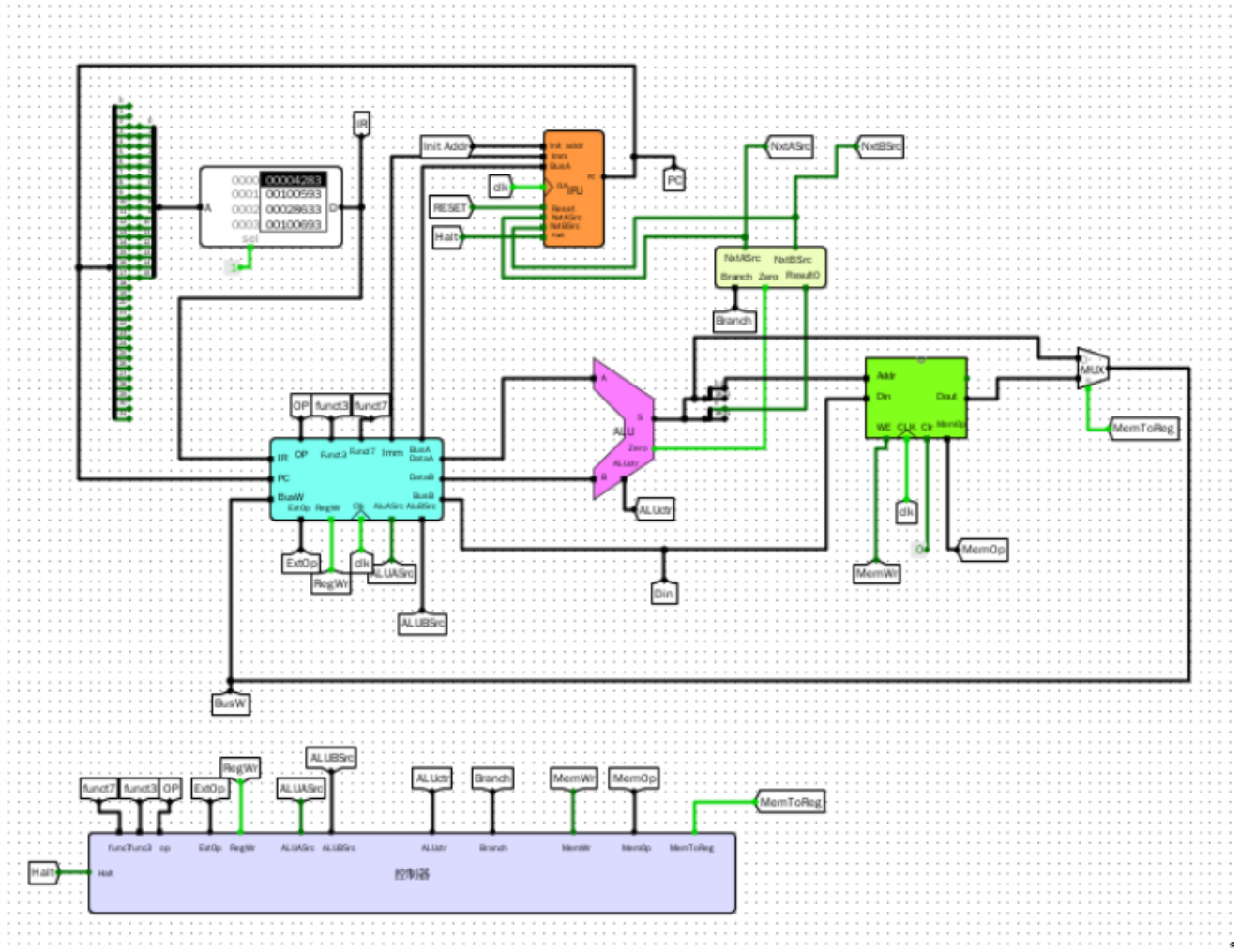
```
for (i=n; i>1; i--) {
    for (j=1; j<=i-1; j++) {
        if (a[j] >a[j+1]) {
            temp=a[j];
           a[j]=a[j+1];
            a[j+1]=temp;
          }
    }
  }
```

整体方案设计

1. 输入输出引脚



电路图



仿真测试

错误现象及分析

在完成实验过程中没有遇到任何错误。

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。

```
CPU 在不同任务之间快速切换，使得每个任务都获得一定的执行时间片，从而有一种同时运行多个任务的错觉。
将计算累加和与数据排序这两个程序分别作为两个任务。
定义一个调度器，它按固定的时间间隔（时间片）切换任务。
在每个任务运行一段时间后，保存任务的状态（如程序计数器、寄存器等）。
恢复另一个任务的状态，使其继续运行
```

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。

```
设置有优先权的中断机制
硬件中断机制用于处理来自输入输出设备的信号。例如，当按下键盘键时，键盘控制器会触发一个中断，通知 CPU 处理输入。
TTY 输出设备在需要显示信息时也可以触发中断，通知 CPU 数据已经传输完成，可以发送新的数据。
```

3. 如何在单周期 CPU 基础上实现多周期 CPU?

将单周期 CPU 转换为多周期 CPU 是一种常见的设计优化，用于提高处理器的效率和性能。在单周期 CPU 中，每条指令都在一个时钟周期内完成，这通常会导致较长的周期时间，以便满足所有指令的需求。相比之下，多周期 CPU 将指令的执行过程分解为多个步骤，每个步骤在一个时钟周期内完成，从而缩短每个时钟周期的长度，提高整体性能。以下是实现多周期 CPU 的基本步骤和思路：

首先，将每条指令的执行过程分解为 取指、 解码、 执行、 访存、 写回
每个步骤在一个时钟周期内完成。不同的指令类型（如 R 型、I 型、J 型指令）可能会有不同的执行步骤，但大致遵循上述步骤。

为了管理多周期执行，需要增加控制逻辑。控制逻辑负责在不同的时钟周期内控制数据通路和寄存器，以便正确地执行每个步骤。需要设计一个有限状态机（FSM, Finite State Machine）来控制各个步骤的顺序和执行。

在单周期 CPU 中，数据路径是为在一个周期内完成所有操作而设计的。转换为多周期 CPU 时，需要对数据路径进行修改，以便在多个周期内完成指令的执行。
寄存器组：需要额外的寄存器来存储中间结果，这些寄存器通常包括指令寄存器（IR, Instruction Register）、内存数据寄存器（MDR, Memory Data Register）、ALU 输出寄存器（ALUOut Register）等。
多路复用器：由于每个周期内的数据流可能不同，需要使用多路复用器（MUX）来选择正确的输入。
时钟周期控制：需要一个时钟信号来同步各个步骤的执行。