实验 6：单周期 CPU 设计与测试

一、实验目的

1. 掌握 RV32I 控制器的设计方法。
2. 掌握单周期 CPU 中的时序设计。
3. 掌握 RISC-V 汇编语言程序的基本设计方法。
4. 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

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）。

整数运算指令：除了两条长立即数指令外，主要功能是对两个寄存器操作数，或一个寄存器一个立即数操作数进行计算后，结果送入目的寄存器。运算操作包括带符号数和无符号数的算术运算、移位、逻辑操作和比较后置位等。

控制转移指令：包括 6 条分支指令和 2 条无条件转移指令，分支指令根据寄存器内容关系运算的结果选择是否跳转。

存储器访问指令：包括 8 条指令，所有访存指令的寻址方式都是寄存器间接寻址方式，首先通过寄存器内容加上立即数计算出内存地址，再根据内存地址存取数据。读写时可按字节、半字和字三种格式访问存储器，在读取单个字节或半字时，按要求对内存数据进行符号扩展或无符号扩展后再写入寄

存器。

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。需要生成的控制信号包括：

ExtOp：宽度为 3 位，选择立即数产生器的输出类型，具体含义参见实验 5 图 5.12。

RegWr：宽度为 1 位，控制是否对寄存器 rd 进行写回，为 1 时写回寄存器。

ALUASsrc：宽度为 1 位，选择 ALU 输入端 A 的来源。为 0 时选择 BusA，为 1 时选择 PC。

ALUBSrc：宽度为 2 位，选择 ALU 输入端 B 的来源。为 00 时选择 BusB，为 01 时选择常数 4（用于跳转时计算返回地址 PC+4），为 10 时选择立即数 Imm(如果是立即数移位指令，只有低 5 位有效)。

ALUctr：宽度为 4 位，选择 ALU 执行的操作，具体含义参见实验 4 表 4.1。

Branch：宽度为 3 位，说明跳转指令的类型，用于生成最终的分支控制信号，具体含义参见表 6.1。

MemtoReg：宽度为 1 位，选择寄存器 rd 写回数据来源，为 0 时选择 ALU 输出，为 1 时选择数据存储器输出。

MemWr：宽度为 1 位，控制是否对数据存储器进行写入，为 1 时写回存储器。

MemOp：宽度为 3 位，控制数据存储器读写格式，具体含义参见实验 5 表 5.1。

RV32I 指令控制信号列表，如表 6.1 所示。

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

这些控制信号控制数据通路上的各个部件按指令的要求进行对应的操作,完成该指令的所有操作。根据指令的操作码、功能码来设计控制信号的逻辑表达式，生成 CPU 控制器。

通过分析 RV32I 指令编码可以发现，相同类型指令的操作码基本相同，通过功能码来区分，因此可以把相同操作码用某个标志位来表示。在 7 位功能码字段中，只有少量的第 5 位 func7[5]为 1，与其他 7 位功能码不同。

大部分控制信号直接可以用指令类型的逻辑表达式来表示，少量控制信号需要加上 3 位功能码来表示，极少数控制信号需要加上 7 位功能码的第 5 位来表示。

根据表 6.1 的定义，参照 6.1 所示的控制器电路示意图，列出每个控制信号的逻辑表达式。

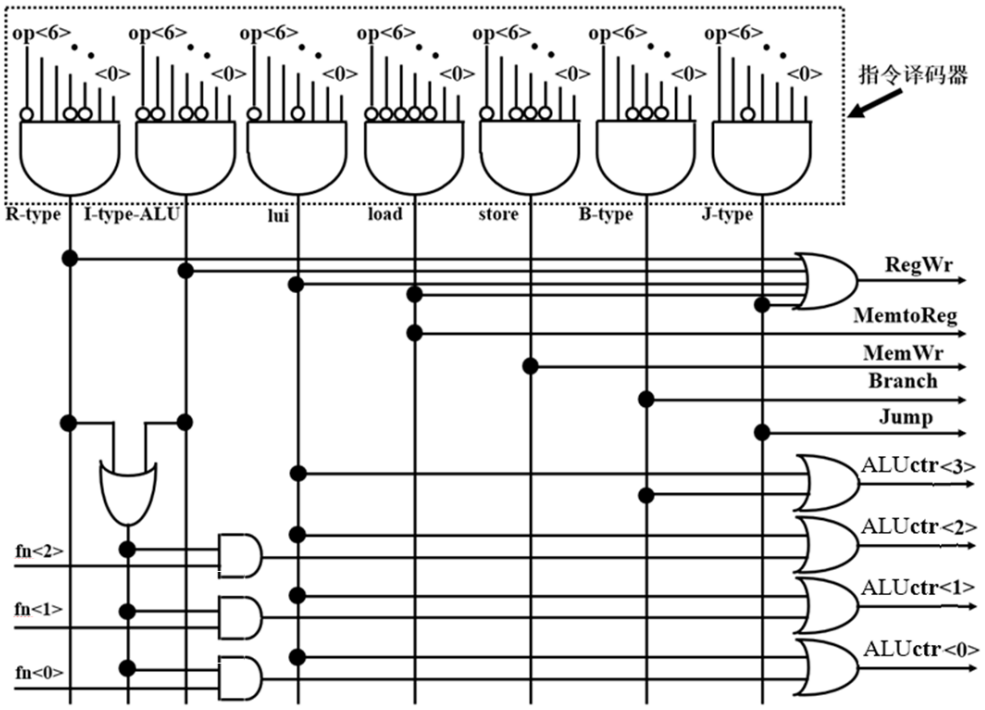

图 6.1 控制器电路示意图

实验步骤如下：

1） 创建子电路。在 Logisim 中添加一个名为“控制器”的子电路，双击该子电路名称，在右侧工作区中构建相应电路。

2） 设计子电路。根据图 6.2 所示的控制器电路引脚布局图，在工作区中添加所需的逻辑门、输入/输出引脚。根据逻辑表达式进行线路连接，添加标识符和电路功能描述信息，得到完整的控制器电路。为了方便程序的结束执行，定义了一个输出信号 halt，当指令操作码 opcode=0000000 时，赋值为 1，反相输出到 PC 寄存器的使能端，中止程序的执行。布局图还提示根据不同类型指令操作码进行比较输出 1 位标志位，实验时也可以通过与门来实现。

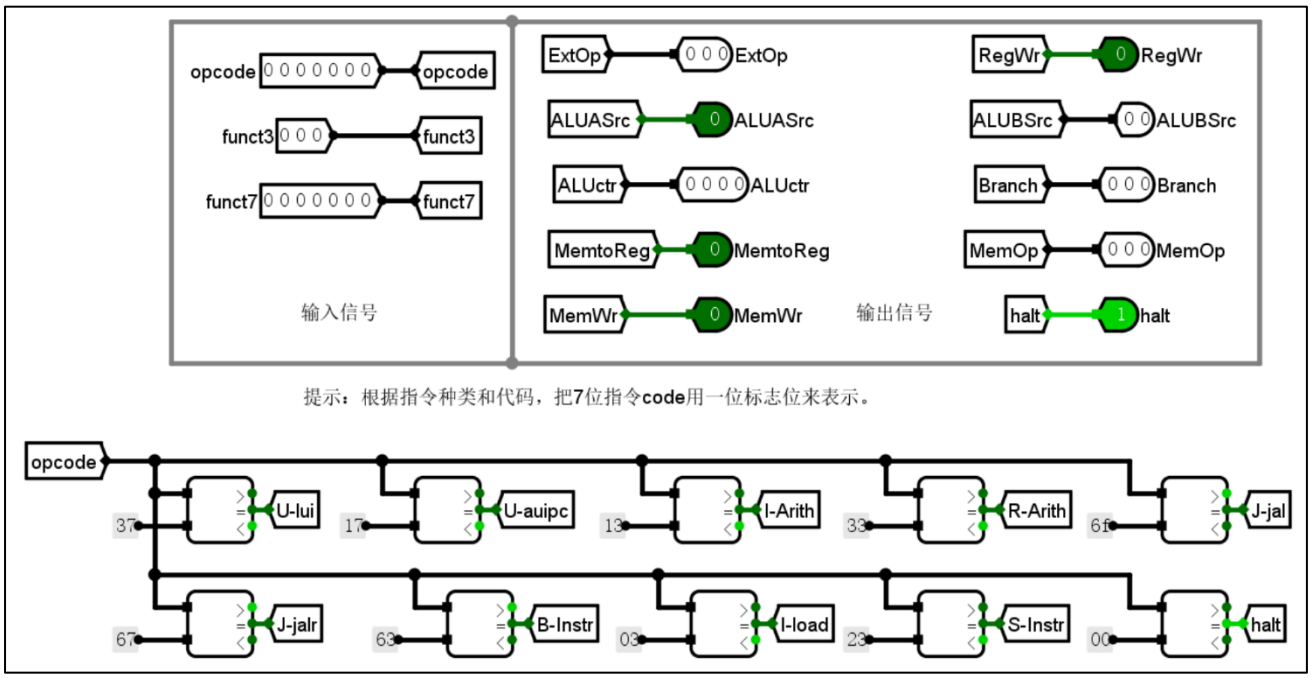

图 6.2 控制器电路引脚布局图

3） 仿真测试并封装子电路。根据表 6.1 中给出的 37 条目标指令的操作码和功能码，逐条输入，验证控制信号的正确性。将该子电路进行封装如图 6.3 所示，保存电路设计文件 lab6.circ。

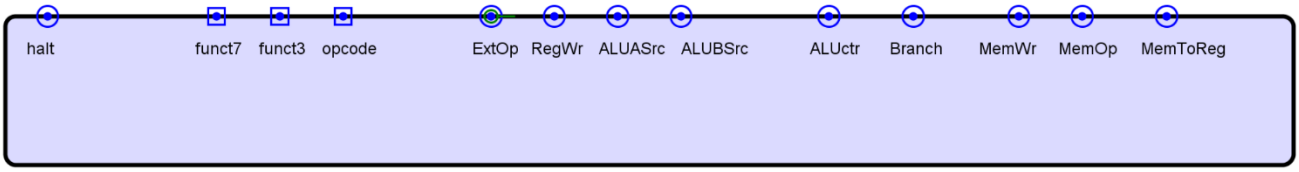

图 6.3 控制器电路封装图图

2. 单周期 CPU 设计实验

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，以时钟下降沿为每个时钟周期的开始，写入操作在时钟下降沿时同步实现，而读取操作，则是异步实现，只要输入有效地址后，立即输出对应数据。

单周期 CPU 需要考虑的时序信号，有复位信号 Reset、片选信号 Sel、中止信号 halt 等。

为了保证 CPU 每次重启时，都能从相同的初始状态开始执行程序，需要设计一个复位信号 Reset，用来初始化 CPU 中各个部件的控制信号，设置 PC 寄存器的起始地址。在单周期 CPU 中，复位信号可以设置为同步信号也可以设置为异步信号，一旦复位信号有效，则初始化系统状态。

指令存储器和数据存储器的片选信号 Sel，高电平有效。在单周期 CPU 中，每个时钟周期都能执行完成一条指令，所以，只要 CPU 的复位信号撤销，该信号设置恒为 1。数据存储器的清零信号可与复位信号相连，写使能信号 MemWr，高电平有效，在指令执行过程中赋值。

程序执行结束后，中止信号 halt 有效，PC 寄存器使能端无效，停止输出。

根据图 6.4 所示的单周期 CPU 原理图，CPU 由不同模块构成，分工协作完成功能。在前期的实验中已经完成全部模块的设计，在实验 3 中实现了寄存器堆模块，实验 4 中实现了 ALU 模块，实验 5 中实现了数据存储器、取指令部件和数据通路模块，本次实验完成了控制器模块。

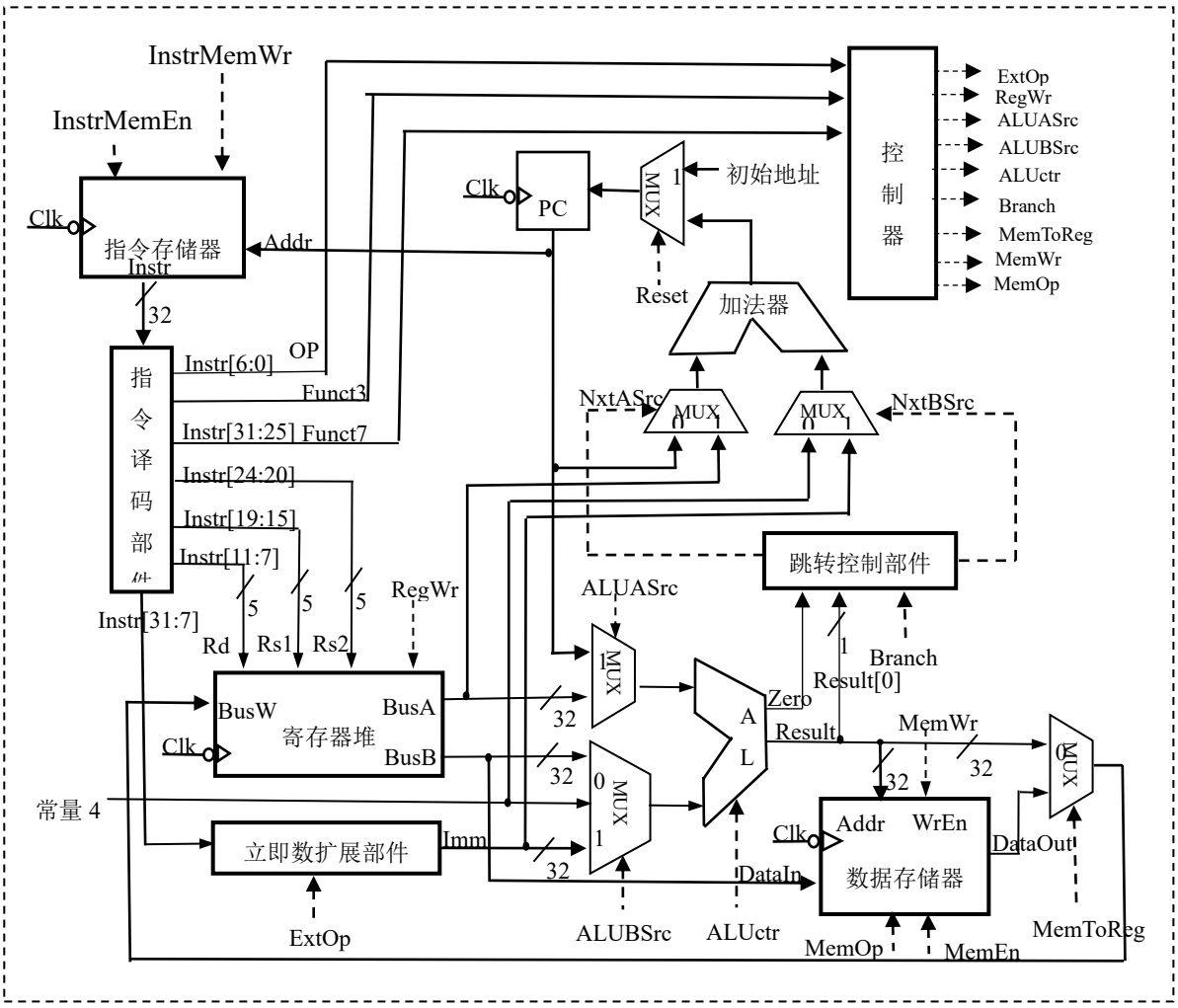

图 6.4 单周期 CPU 原理图

可利用前面实验已经完成的寄存器堆、取指令部件、ALU、数据通路、控制器等电路，连接生成单周期处理器，最终可在该处理器上运行 RV32I 的程序来验证处理器设计的正确性。

实验步骤如下。

1）创建子电路。在 Logisim 中添加一个名为“单周期 CPU”的子电路，双击该子电路名称，在右侧工作区中构建相应电路。

2）设计子电路。在 logisim 的 project 菜单下添加 logisim 库文件，把实验 3、实验 4 和实验 5 的设计文件添加到 lab6 的项目中。在工作区中放置取指令部件 IFU、指令存储器 ROM、数据通路 IDU、ALU、数据

存储器模块和控制器部件，以及输入输出引脚和隧道，如图 6.5 所示的布局引脚。根据模块中定义信号进行连接。设置指令存储器和数据存储器的属性，片选信号都定义为高电平有效。指令存储器的地址位宽设置为 16 位，数据位宽设置为 32 位。加载实验 5 提供的测试指令文件 lab5.o，时钟单步执行，验证实验结果，保存电路文件。

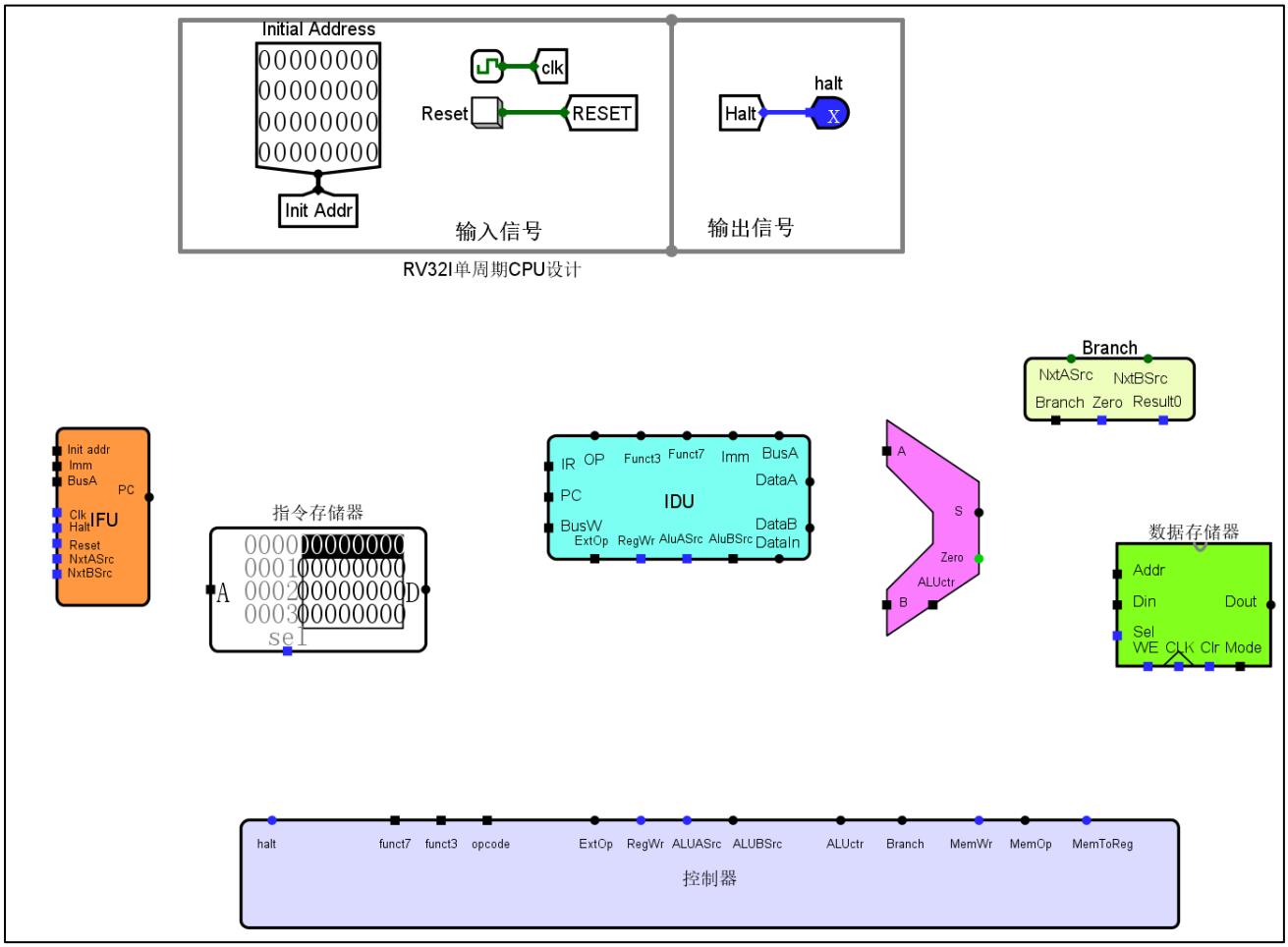

图 6.5 单周期 CPU 电路图

注意：RISC-V 架构定义 0 号寄存器的值始终为 0，需要检查寄存器堆中 0 号寄存器是否已经做了硬件设计处理。为了有足够的时间计算下一条指令的地址，可将 PC 的写入时钟信号设置为下降沿触发，寄存器堆和数据存储器的时钟触发也要设置为下降沿触发。

可添加一个时钟周期计数器，来记录不同程序执行的总周期数。

3. 用累加和程序验证 CPU 设计

首先，将计算累加和的 RV32I 汇编语言程序在 RARS 中调试通过，汇编成机器代码并导出。然后将机器代码载入到 CPU 的指令存储器部件中。设置参数 n，启动时钟信号，执行代码，程序执行结束后，观察寄存器和存储器指定存储单元中的结果，验证指令执行的正确性。

用累加和程序进行 CPU 设计验证的具体步骤如下。

1）编写汇编语言源程序

使用表 6.1 中的 37 条 RV32I 指令，编写一个计算 S=1+2+…+n 的累加和程序。进行累加和计算的高级语言伪代码如下：

```
S=0;
for (i=1; i<=n; i++) S=S+i;
```

假设入口参数 n 存放在存储器地址 0 单元，累加和 S 保存在存储器地址 4 单元中，程序运行过程中参数 n、循环变量 i、累加和 S 分别存放在寄存器 a0、a2、a3 中，对应的汇编语言源程序如下：

```
#计算累加和 RV32I 汇编程序
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
```

为便于观察程序执行结果，上述程序结束时执行了一条循环执行的无条件跳转指令。此外，程序中累加结果的保存存储器地址为 4 单元中。因为 Logisim 中按数据位宽编址，存储器地址 4 单元在 Logisim 中 RAM 组件中保存在地址 1 单元。

使用任意一种文本编辑器输入上述 RV32I 汇编程序，并保存为 sum.asm 文件。

2）将汇编语言源程序转换为机器代码。

在 RARS 中打开累加和汇编语言程序 sum.asm，如图 6.6 所示，可以在 RARS 的编辑窗口中编辑汇编程序，并进行保存。执行 Run 菜单下的 Aseemble（F3）命令，将 RV32I 源程序进行汇编处理，转换成机器代码。如果有语法错误，则汇编不通过，报告错误信息，则返回编辑窗口修改汇编程序。

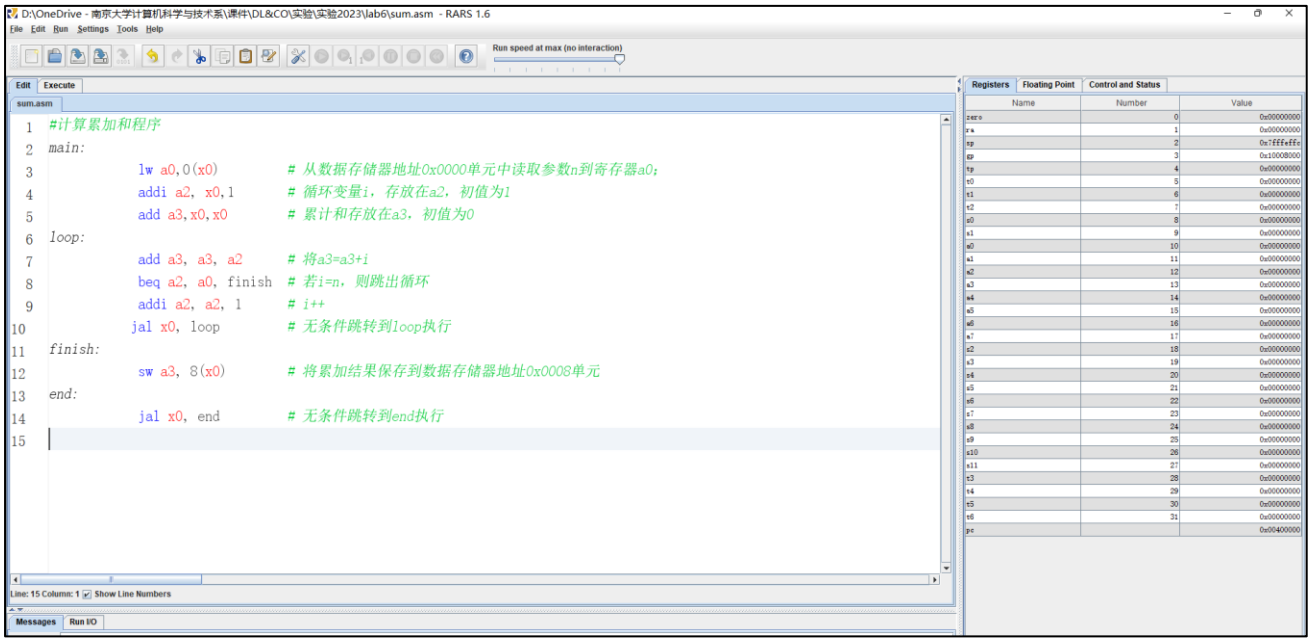

图 6.6 RARS 的编辑窗口

在我们的单周期 CPU 设计中，指令存储器和数据存储器分别实现独立编址，无操作系统环境。为了调试方便，设置初始地址从 0 开始。为了模拟单周期 CPU 的系统状态中，需配置 RARS 的内存分配模式，在菜单 Setting 中的 Memory Configuration 选项设置为“Compact，Data at Address 0”，如图 6.7 所示，这样数据段的起始地址就从 0 开始。

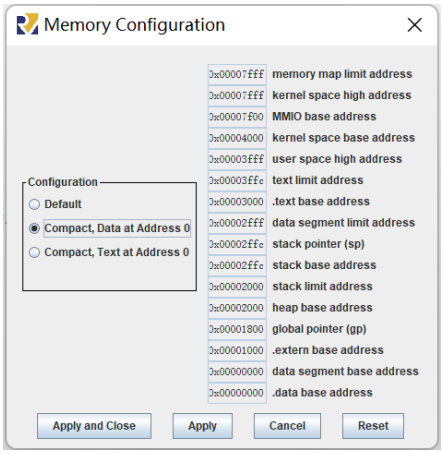

图 6.7 RARS 中 Memory Configuration 选项设置

3）对机器代码进行调试运行。

设置累计和计算参数 n 到数据存储器地址 0 单元中。如图 6.8 所示，在数据段（Data Segment）的起始地址 0x00000000 中，输入 16 进制数字 0x64（十进制数 100），按回车结束输入。

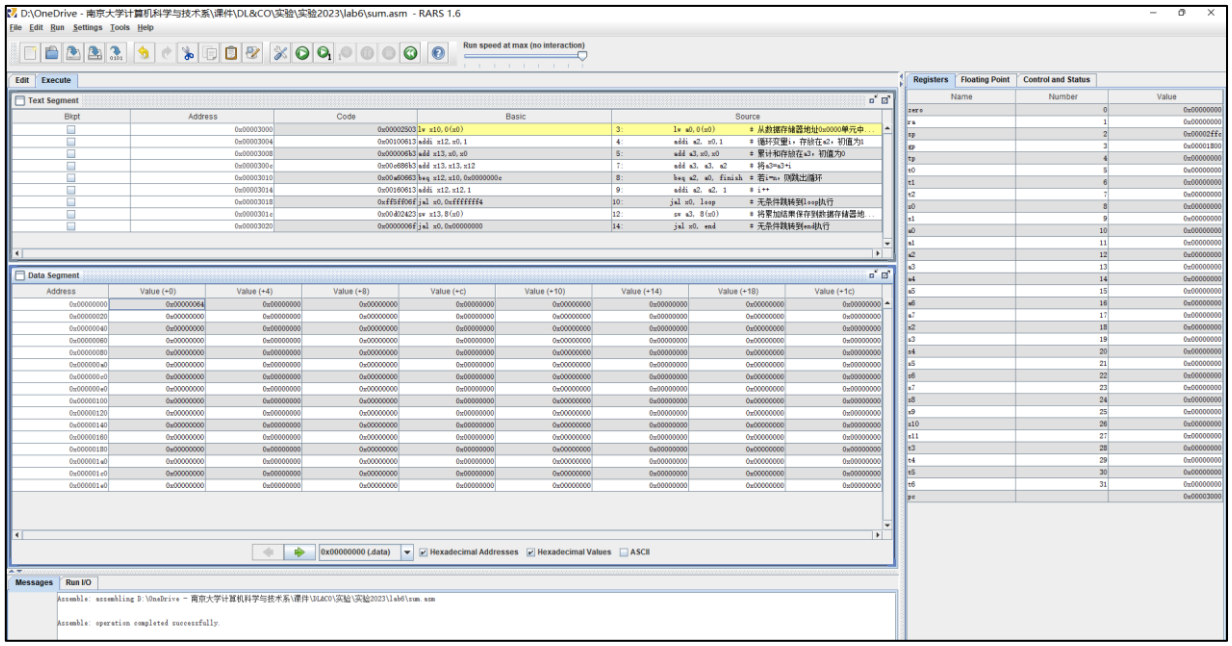

图 6.8　RARS 中程序仿真执行界面

在 Run 菜单下选择连续运行 Go（F5）方式，由于程序最后是一条循环执行的自跳转指令，因此需单击暂停按钮 Pause（F9）或停止按钮 Stop（F11）才能终止程序执行，此时可查看最终执行结果。如图 6.9 所示，在数据段（Data Segment）的地址 0x4（0x00000000+4）处，可以观察到累加和的结果为 0x000013ba，十进制数为 5050，同时可以查看寄存器 a0,a2,a3 里的数值，来验证程序执行正确性。

如果需要调试，则选择单步执行按钮 Step（F7）。每条指令执行后，可在右侧窗口中检查各寄存器内容的变化。也可以在代码段的列表中设置断点，然后连续运行至断点后，再检查寄存器和存储器里的数据。

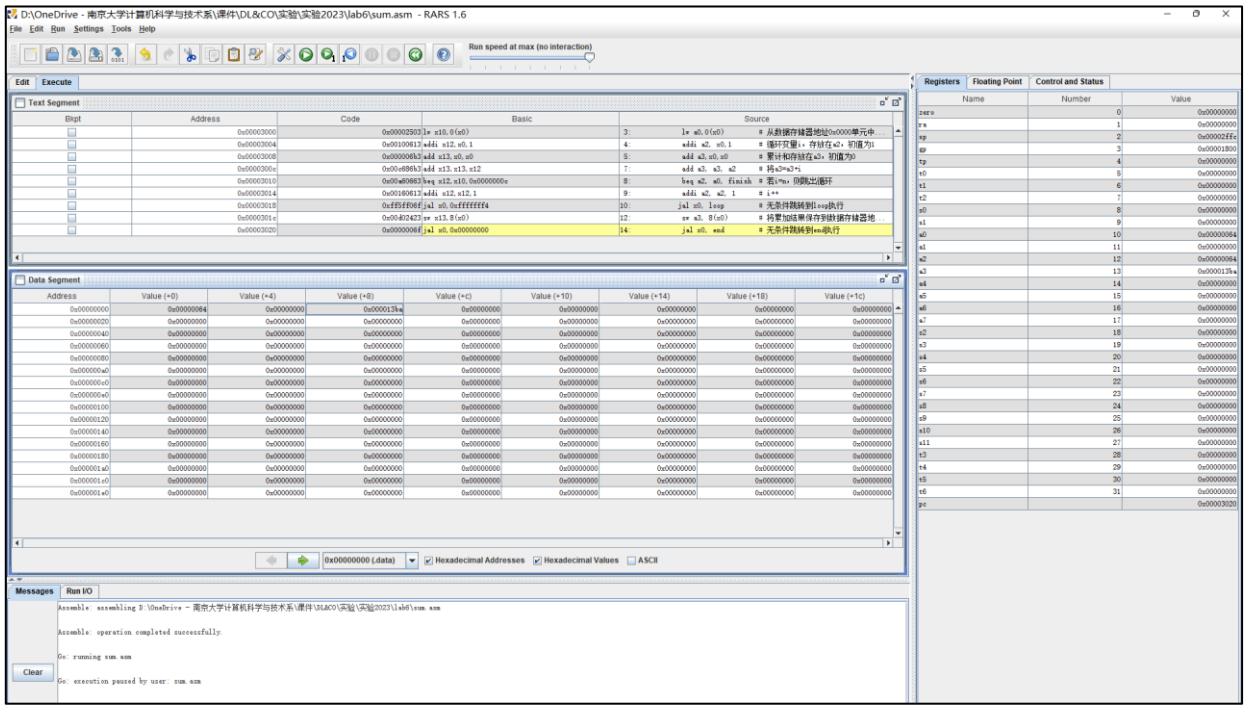

图 6.9 查看程序仿真运行结果

4）导出机器代码。

在 RARS 的 File 菜单中点击“Dump Memeory To File”按钮，可以将汇编程序对应的机器代码和数据段中的数据导出。如图 6.10 所示，首先选择内存段（Memory Segment）为代码段.text（0x00003000-0x00003024），选择导出格式（Dump Format）为 16 进制文本格式（Hexadecimal Text），点击 Dump to File 按钮，设置文件名称为 sum.hex，将程序可执行机器代码以十六进制数据文本格式导出到该文件。

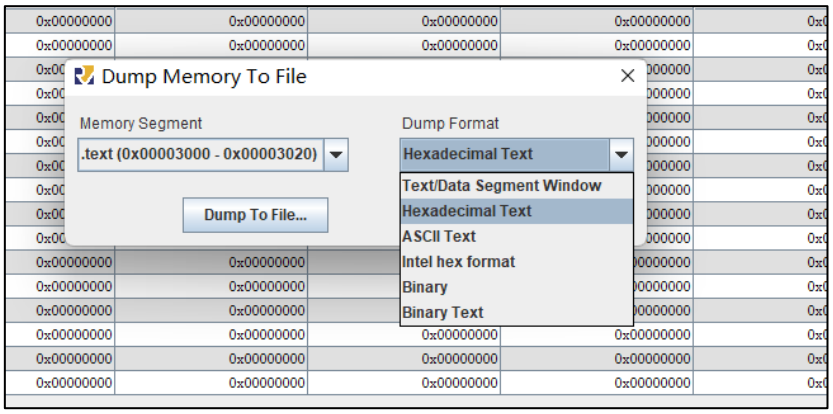

图 6.10　Dump Memeory To File 设置

5）在单周期 CPU 中运行测试程序。

在单周期 CPU 的指令存储器中装载可执行机器代码的数据镜像文件，需要在该文件头部插入一行文本“v2.0 raw”，并保存。利用文本编辑器打开 sum.hex 文件，插入首行后的文件内容如下所示：

```
v2.0 raw
00002503
00100613
000006b3
00c686b3
00a60663
00160613
ff5ff06f
00d02423
0000006f
```

在单周期 CPU 的指令存储器（ROM）中装载计算累加和可执行机器代码镜像文件 sum.hex，在数据存储器地址 0 单元中设置初始参数，然后启动时钟信号，执行程序，观察输出结果，验证 CPU 设计的正确性。具体步骤如下：

① 在 Logisim 中，打开单周期 CPU 电路图，用鼠标右键点击指令存储器（ROM 组件），选择装载镜像（load image）命令，将镜像文件 sum.hex 加载到指令存储器起始地址 0 单元中，可点击 Edit Content 打开 16 进制编辑器进行查看，如图 6.11 所示。

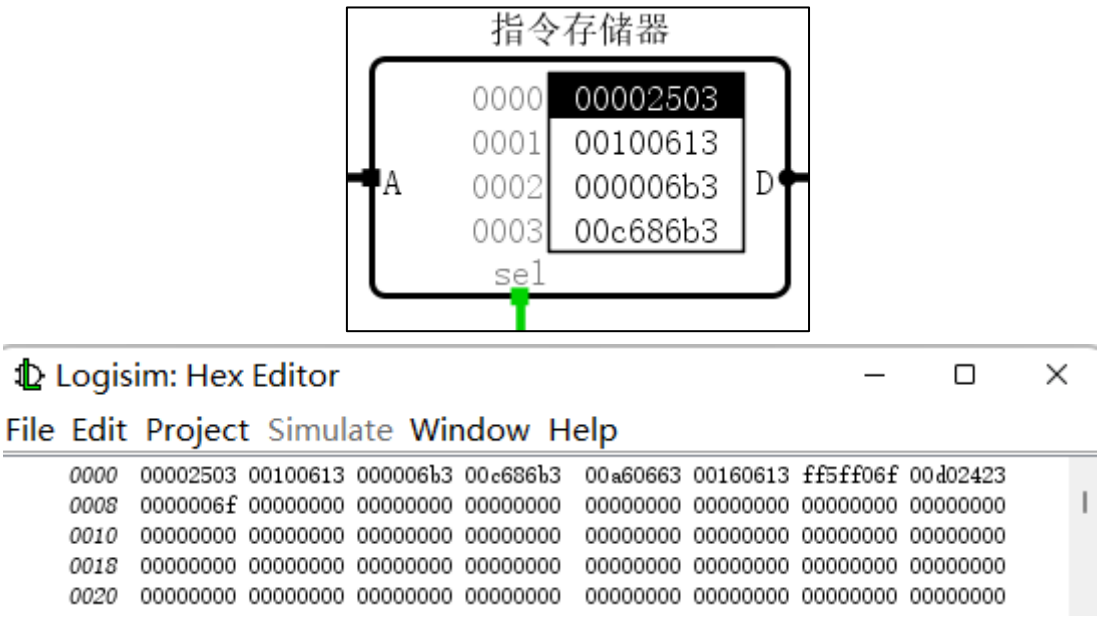

图 6.11 将代码装入指令存储器

在 CPU 电路图中选中数据存储器组件，进入子电路，在最低字节的 RAM 组件地址 0x0000 处，用鼠标左键点击选中，输入参数 0x64，如图 6.12 所示；或者在组件上点击鼠标右键，选择 Edit Content 菜单项，打开 16 进制编辑器在地址 0x0000 处输入参数 0x64。

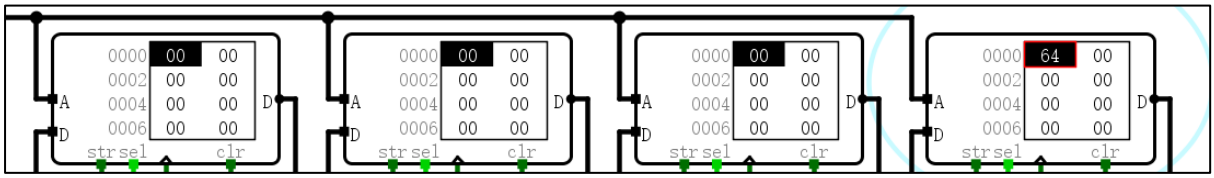

图 6.12 在数据存储器中设置参数

② 返回 CPU 设计主电路窗口。在 Logisim 的仿真菜单下，选择合适的时钟频率，如 1kHz，然后选中“时钟连续（Ticks Enabled）”（Ctrl+K），CPU 开始自动执行机器代码。为了便于观察，可在电路中添加探针，以查看每条指令的执行情况。当程序始终执行最后一行（第 8 行）指令时，说明程序执行已经结束，可中止执行，如图 6.13 所示。可按 Ctrl+K 取消“时钟连续（Ticks Enabled）”，以暂停/中止程序执行。

6.13 程序结束无限循环执行状态

程序执行中止后，选中数据存储器组件，进入数据存储器子电路中查看程序执行结果。可以看到，在低字节 ram 地址 0x2 处，显示数据 0xba；在次低字节 ram 地址 0x2 处显示数据 0x13，高 2 个字节的 ram

地址 0x2 处，显示数据 0x0，如图 6.14 所示。因此，累加和的计算机结果为 0x13ba（十进制数 5050），说明程序执行结果正确。如果不能直接在 ram 组件中观察到结果，则需要分别在不同字节的 ram 上点击 Edit Content，打开 16 进制编辑器来查看。

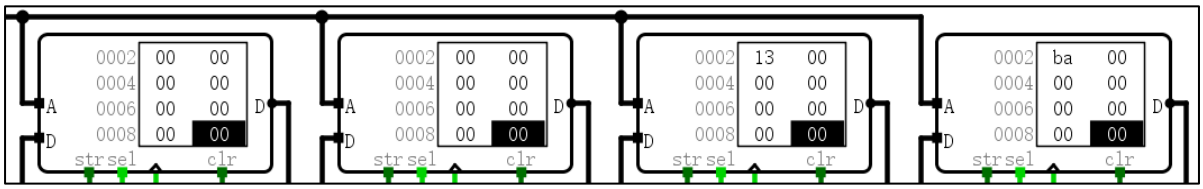

图 6.14 查看 n=100 时数据存储器中的执行结果

如果需要调试，则选择“时钟单步”（Ticks Once）（Ctrl+T）方式执行，可查看每一条指令的执行情况。

4. 用冒泡排序程序进行 CPU 设计验证

采用冒泡排序对有限个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。

假设数组 a 中存放的是关键字序列，对数组 a 的元素按照从小到大的顺序排序，其完整的冒泡排序算法流程如图 6.15 所示。

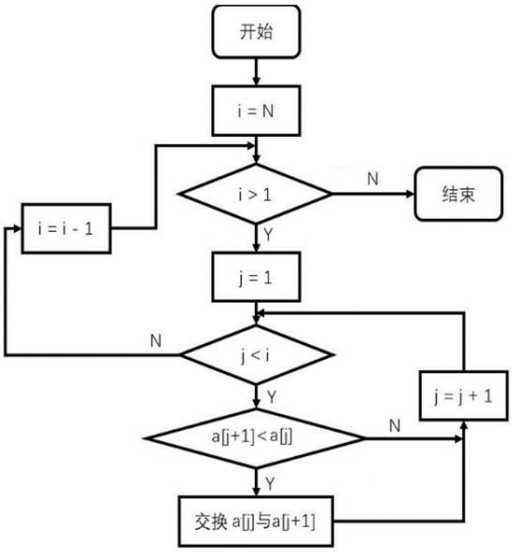

图 6.15 冒泡排序算法流程图

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;
        }
    }
}
```

假设所有入口参数存放在从 0x000000 开始的数据存储器中，首先存放的是待排序数据个数 *n*，接着存放的是 *n* 个待排序的数组元素。

汇编代码中将数据个数 n 读到寄存器 a0，常量 1 保存在寄存器 a1 中，外循环变量 i 保存在 a2，内循环变量 j 保存在 a3，第 j 个元素 a[j]的地址存放 a4。第 j 个元素 a[j]读入 a6，第 j+1 个元素 a[j+1]读入 a7。汇编语言程序参考代码如下：

```
#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
```

在文本编辑器中输入上述汇编程序，并保存文件为 bubble.asm。在 RARS 中打开 bubble.asm 汇编程序进行编辑保存，汇编通过后，配置测试数据进行测试运行，验证程序的正确性。以 16 进制数据格式导出汇编程序机器代码到文本文件 bubble.hex，然后添加首行字符串“v2.0 raw”后保存，作为数据镜像文件。

假设待排序的数据个数为 10（0xa），待排序的数据为单字节，，假设为 0x8、0x41、0x2、0x12、0x36、0x6、0x9、0x5、0x5b、0x7，则创建数据镜像文件 bubble.dat 的内容如下：

v2.0 raw

a 8 41 2 12 36 6 9 5 5b 7

在 Logisim 中打开单周期 CPU 电路图，在指令存储器中加载冒泡程序可执行机器代码数据镜像文件 bubble.hex，打开数据存储器子电路，在最低字节 RAM 中加载待排序数据镜像文件 bubble.dat。

在 Logisim 的仿真菜单下，选择合适的时钟频率，如 1kHz，然后选中“时钟连续”（Ticks Enabled）（Ctrl+K），CPU 开始自动执行机器代码。当指令存储器中的地址不再变化时，表示已经执行到最后一条指令，此时按 Ctrl+K 终止“时钟连续”执行方式。

选中最低字节数据存储器，通过 16 进制编辑器查看冒泡排序后的数据序列，如图 6.16 所示。

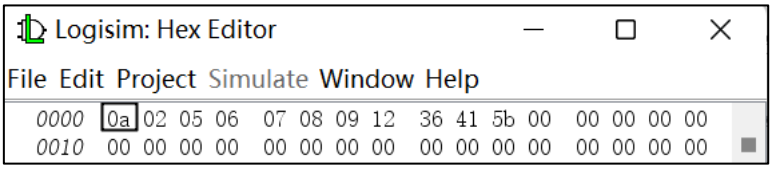

图 6.16 数据存储器中排序结果

将排序后的结果保存到镜像文件，可看到文件中的内容如下：

v2.0 raw

a 2 5 6 7 8 9 12 36 41 5b

5. C 程序汇编测试

C 程序汇编成 RV32I 程序，可以通过安装 risc-v gcc 工具链来实现。这里介绍在后续的计算机系统基础课程的 PA 项目提供了在 Ubuntu 下安装 risc-v gcc 工具链的方案，具体方法可以参考 PA2.2 中准备交叉编译环境。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

然后在 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
-# include <gnu/stubs-ilp32.h>
+//# include <gnu/stubs-ilp32.h>
 #endif
```

继续在 Ubuntu 下执行下列命令：

cd ~

export AM_HOME=`pwd`/abstract-machine

cd am-kernels/tests/cpu-tests，，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。然后在返回上一级目录，执行下列命令

make ARCH=riscv32-npc ALL=bubble-sort

可修改 ALL=后面的文件名 bubble-sort 为任何需要编译的测试用例，将生成可用于 Logisim 下 CPU 测试的指令镜像文件和 4 个按字节分开的数据镜像文件。

例如 bublle-sort.c 的源程序如下：

```
#include "trap.h"
#define N 20
int a[N] = {200, 1212, 3114, 76, 913, 5215, 8716, 910,30000, 18000, 7711, 190, 39, 21, 87, 2455, 400, 33, 800, 170};
void bubble_sort() {
    int i, j, t;
    for(j = 0; j < N; j ++) {
        for(i = 0; i < N - 1 - j; i ++) {
            if(a[i] > a[i + 1]) {
                t = a[i];
                a[i] = a[i + 1];
                a[i + 1] = t;
            }
        }
    }
}
int main() {
    bubble_sort();
    int i;
    for(i = 0; i < N; i ++) {
        check(a[i] == i);
    }
    check(i == N);
    bubble_sort();
    for(i = 0; i < N; i ++) {
        check(a[i] == i);
    }
    check(i == N);
    return 0;
}
```

执行 make 命令后生成 5 个文件（如果安装 risc-v gcc 工具链有困难，可直接使用实验讲义 lab6 压缩包中 C Test 目录下提供的文档）。将 bubble-sort-riscv32-npc.bin-logisim-inst.txt 文件加载到指令存储器中，将 bubble-sort-riscv32-npc.bin-logisim-data0.txt 加载到最低字节的数据存储器中，将 bubble-sort-riscv32-npc.bin-logisim-data1.txt 加载到次低字节的数据存储器，以此类推，加载 bubble-sort-riscv32-npc.bin-logisim-data2.txt、bubble-sort-riscv32-npc.bin-logisim-data3.txt 到指定的数据存储器中。选择连续时钟信号，启动程

序的执行，分析程序执行的结果。提示：在 bubble-sort-riscv32-npc.txt 文件中可查看 C 语言源程序的 RV32I 反汇编代码。

四、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。
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？
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。
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？