

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

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。

ALUASrc：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 5.4。

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，并按照**最小风险法**来化简逻辑表达式，不考虑无关项。

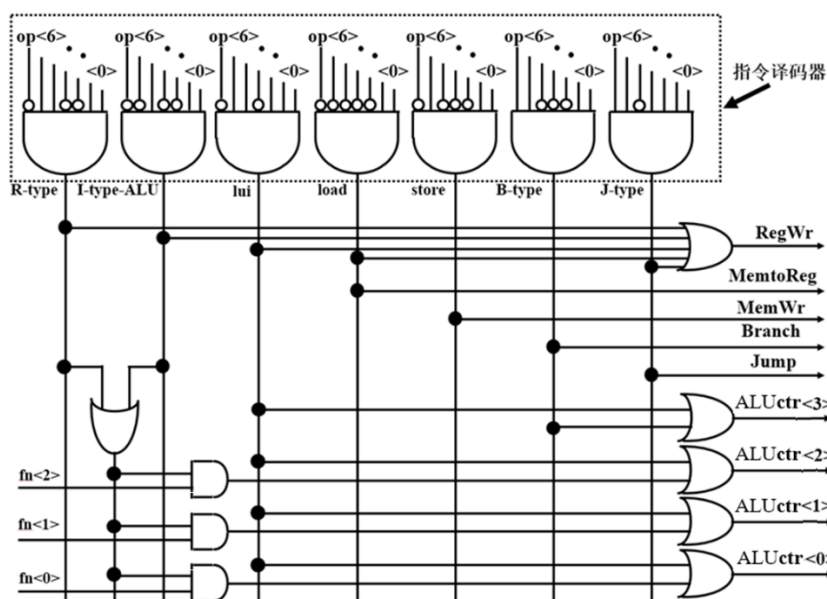


图 6.1 控制器电路示意图

实验步骤如下：

- 1) 创建电路。在 Logisim 中修改主电路名称为“控制器”，在右侧工作区中构建相应电路。
- 2) 设计控制器电路。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。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。

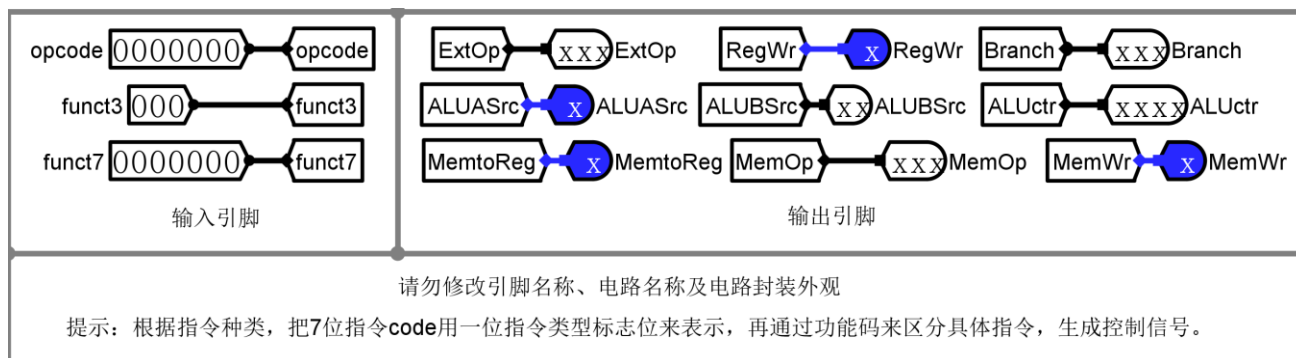


图 6.2 控制器电路引脚布局图

- 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。不要改变该子电路封装外观，保存电路设计文件 lab6.1.circ。

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。

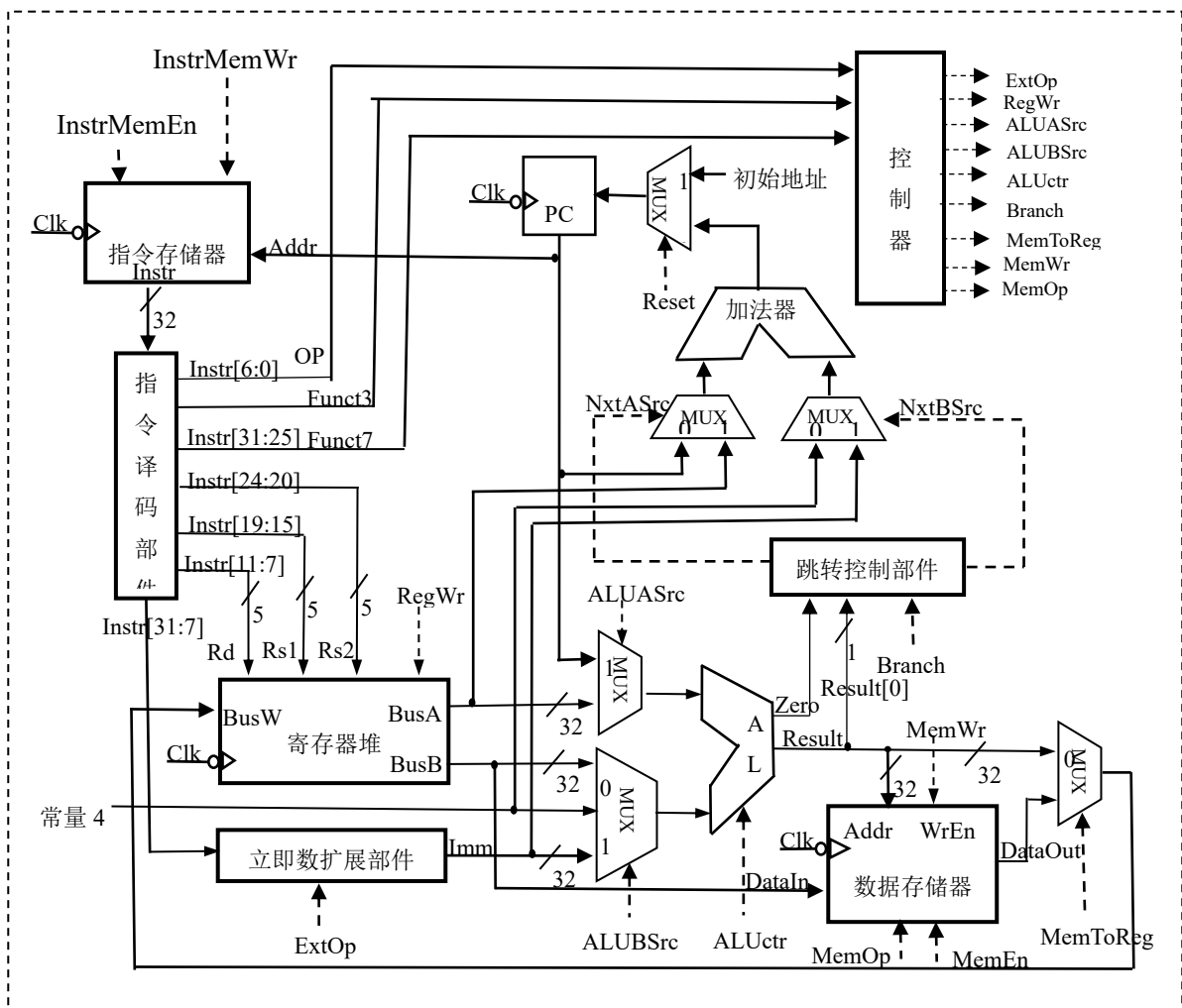


图 6.4 单周期 CPU 原理图

在实验五数据通路部件的基础上，加上控制器部件就能连接成单周期处理器。加载 RV32I 测试程序，观测实验结果，验证单周期处理器的功能。

实验步骤如下。

1) 修改 IFU 子电路。在项目中添加一个名为“IFU”的子电路，添加 lab5.3 库文件，将库文件中的取指令组件的电路设计和封装外观复制到 IFU 子电路中，然后卸载库文件 lab5.3。按照实验要求，添加控制 PC 寄存器时钟信号是否有效的输入端口 halt，如果 halt 输入有效，则 PC 寄存器停止输出。

2) 修改控制器子电路。在项目中添加一个名为“控制器”的子电路，添加 lab6.1 库文件，将库文件中的取指令组件的电路设计和封装外观复制到 IFU 子电路中，然后卸载库文件 lab6.1。在子电路中添加输出引脚中止信号 Halt，在电路中，添加生成中止信号的逻辑设计，当指令操作码 opcode 等于 0x73 时，Halt 信号赋值为 1。

3) 设计单周期处理器。在项目中添加 lab4.5.circ、lab5.2.circ、lab5.4.circ 库文件。根据图 6.4 原理图设计，在主电路工作区中放置取指令部件 IFU 子电路、指令存储器 ROM、数据通路 IDU、ALU、数据存储器

DataRAM 和控制器子电路等组件以及输入输出引脚和隧道等。如图 6.5 所示的布局引脚，将控制器的输入输出信号连接到其他子电路对应的引脚上。设置指令存储器属性，片选信号定义为高电平有效，指令存储器容量设置位 256KB，地址位宽为 16 位，数据位宽为 32 位。PC 寄存器、寄存器堆和数据存储器的触发方式设置为相同的时钟边沿触发。添加一个程序执行周期计数器，来记录不同程序执行的总时钟周期数，复位信号 ReSet 有效时清零，停止信号 Halt 有效时停止计数。为了便于观察程序执行状态，将 PC 寄存器、指令 IR、寄存器堆输入数据 BusW、数据存储器输入数据 Din 和中止信号 Halt 作为输出信号。（提示：为了方便在线评测，可将每个库文件中的主电路设计和封装外观，复制到当前项目的子电路中，然后卸载库文件。）

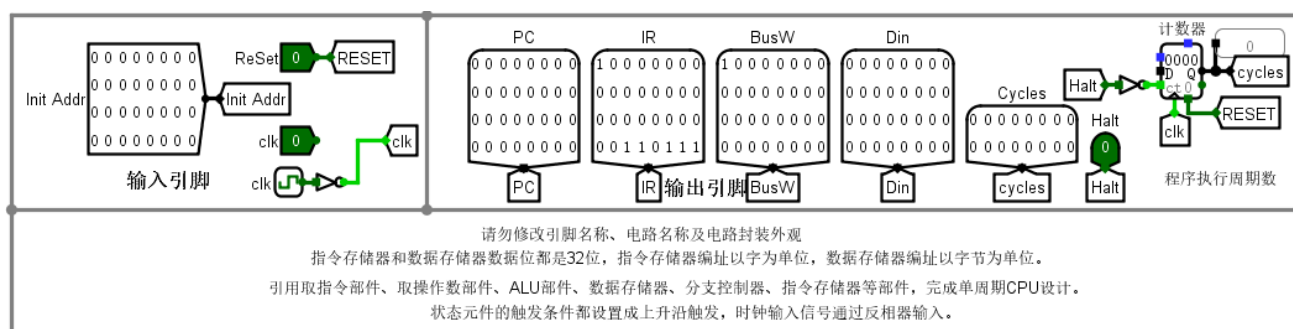


图 6.5 单周期 CPU 引脚图

4) 加载的指令测试文件。为了测试系统整体功能和主要指令的执行情况，本次实验提供了一个指令程序 lab6.2.asm，用来检查加减运算、移位、访存、跳转等指令的功能，如果指令执行结果验证通过，则在 10 号寄存器中写入 “00c0ffee”；否则在 10 号寄存器写入 “deaddead”，并在 3 号寄存器中写入当前测试指令的序号。

```
.text
main:
test_1: #x0 register,addi,xor,lui,bne test
    lui    x0, 0x80000
    addi   x0, x0, 0x7ff
    xor    x1, x1, x1
    addi   x3, x0, 1    #R[3]=1
    bne    x0, x1, fail
test_2:    #add test
    lui    x1, 0x80000
    lui    x2, 0xffff8
    add    x14, x1, x2
    lui    x7, 0x7fff8
    addi   x3, x0, 2    #R[3]=2
    bne    x14, x7, fail
test_3:    #and test
    lui    x1, 0xff01
    addi   x1, x1, -16 # 0ff00ff0
```



```

lui      x2,0xf0f0f
addi     x2,x2,240 # f0f0f0f0
and      x15,x1,x2
lui      x7,0xf00
addi     x7,x7,0xf0
addi     x3,x0,3    #R[3]=3
bne      x15,x7, fail
test_4:   #sll test
addi     x1,x0,1
addi     x2,x0,63
sll      x16,x1,x2
lui      x7,0x80000
addi     x3,x0,4    #R[3]=4
bne      x16,x7, fail
test_5:   #sra test
lui      x1,0x80000
addi     x2,x0,14
sra      x17,x1,x2
lui      x7,0xfffe0
addi     x3,x0,5    #R[3]=5
bne      x17,x7, fail
test_6:   #slti test
lui      x1,0x80000
slti     x18,x1,2047
addi     x7,x0,1
addi     x3,x0,6    #R[3]=1
bne      x18,x7, fail
test_7:   #sltiu test
lui      x1,0x80000
sltiu    x19,x1,2047
addi     x7,x0,0
addi     x3,x0,7    #R[3]=7
bne      x19,x7, fail
test_8:   #load,store test
auipc    x1, 0
addi     x1,x1,204 #buffer
addi     x2,x0,0xee
sb       x2,0(x1)
addi     x2,x0,0xff
sb       x2,1(x1)
addi     x2,x0,0xc0
sb       x2,2(x1)
addi     x2,x0,0x00
sb       x2,3(x1)

```

```

lb    x20,1(x1)
    addi x7,x0,-1
    addi x3,x0,8    #R[3]=8
    bne    x20,x7, fail
lhu x21,0(x1)
    lui    x7,16    #10
    addi x7,x7,-18    #0x0ffee
    addi x3,x0,9    #R[3]=9
    bne    x21,x7, fail
sh    x21,4(x1)
lh    x22,2(x1)
    addi x7,x0,0xc0
    addi x3,x0,10    #R[3]=10
    bne    x22,x7, fail
sh    x22,6(x1)
lw    x23,0(x1)
        lui    x7,0xc10
        addi x7,x7,-18 # c0ffee
    addi x3,x0,11    #R[3]=11
    bne    x23,x7, fail
lw    x24,4(x1)
        lui    x7,0xc10
        addi x7,x7,-18 # c0ffee
    addi x3,x0,12    #R[3]=12
    bne    x24,x7, fail
test_9:    #jalr test
    auipc  x25, 0
    jalr   x25, x25,24    # test_9_2
test_9_1:
    add x0, x0,x0
    jal x0, fail
test_9_2:
    add x0,x0,x0
    jal x0, fail
auipc x7,0
addi x7,x7,-21    #test_9_1
addi x3,x0,13    #R[3]=13
bne x25,x7,fail
pass:
    lui    x10,0xc10
    addi    x10,x10,-18    # R[10]=0X00c0ffee
    ecall                #程序中中止执行
fail:
    lui    x10,0xdeade

```

```

addi    x10,x10,-339  # R[10]=0Xdeaddead
ecall                                #程序中止执行

```

在指令寄存器中加载 lab6.2.asm 已经生成的机器代码数据文件 lab6.2.hex，时钟单步执行，观测实验结果，写出寄存器堆和存储器里的数据，验证电路功能，保存电路文件为 lab6.2.circ。

3. 用累加和程序验证 CPU 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。

本次实验要求是先将计算累加和的 RV32I 汇编语言程序在 RARS 中调试通过，然后导出机器代码并在首行添加“V2.0 raw”语句。在单周期 CPU 的指令存储器中加载该计算累加和的机器代码数据镜像文件。在数据存储器的 0 号单元中设置参数 n。设置仿真使能，启动时钟信号，执行机器代码。程序执行结束后，在数据存储器的地址 4 号单元中观测累加和的计算结果，验证 CPU 设计和测试程序的正确性。

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 $S=1+2+\dots+n$ 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 号单元中，累加和 S 保存在存储器地址 4 号单元中，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 a1、a2、a3 中，对应的汇编语言源程序如下：

```

#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

```

为了便于在数据存储器中加载测试数据文件，程序中读取数据存储器的指令使用的是按字节读取。为了中止程序的执行，程序结束时执行 `ecall` 指令。使用任意一种文本编辑器输入上述 RV32I 汇编程序，并保存为 `lab6.3.asm`。

2) 将汇编语言源程序转换为机器代码。

RARS 是一款开源的 RISC-V 汇编程序编译仿真工具，需要安装 java 运行环境，可跨平台使用。在线评测环境中在命令行输入“`java -jar rars1_6.jar`”打开 rars 程序。在 RARS 中打开累加和汇编语言程序 `lab6.3.asm`，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 `Assemble (F3)` 命令或点击汇编图标，将汇编语言程序进行汇编处理，转换成机器代码，如图 6.6 所示。如果有语法错误，则汇编不通过，在信息窗口报告错误信息，并返回编辑窗口修改汇编程序。

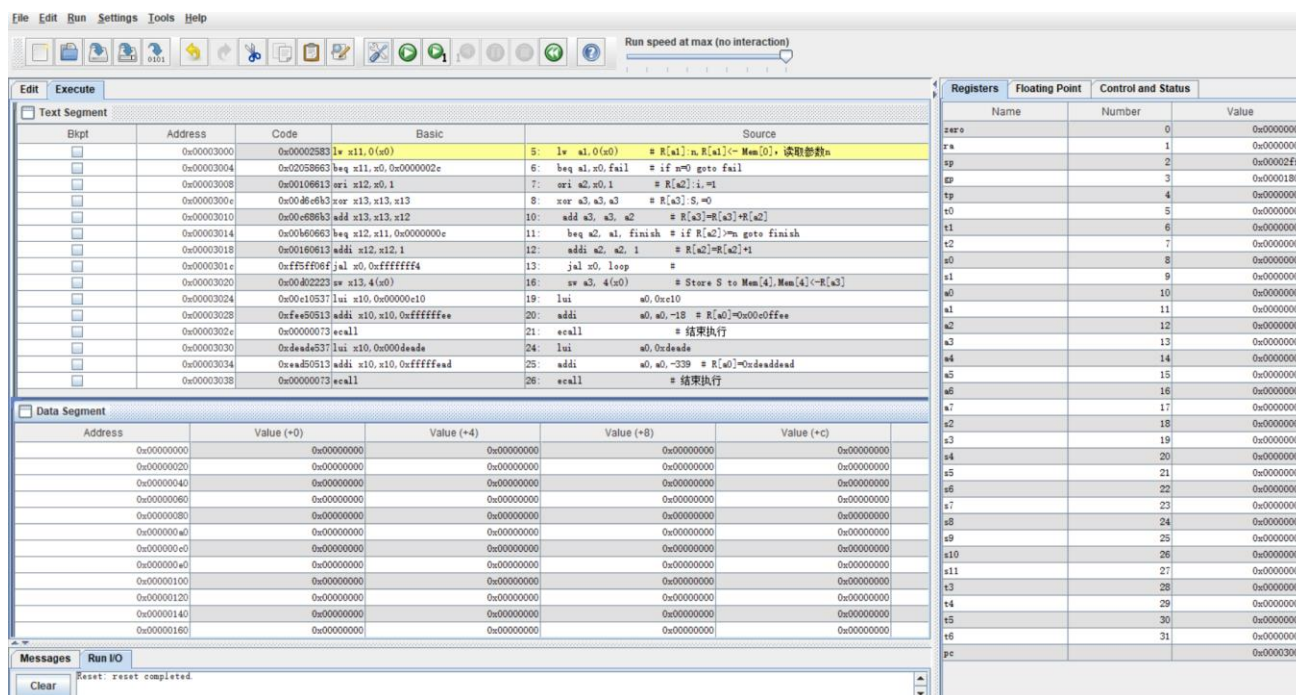


图 6.6 RARS 的执行窗口

为了调试的方便，需要在 `setting` 菜单下勾选“`Self-modifying code`”，允许直接修改代码段的内容。

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。

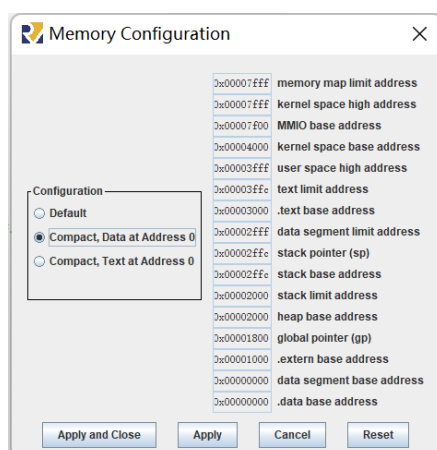


图 6.7 RARS 中 Memory Configuration 选项设置

3) 对机器代码进行调试运行。

在数据存储器地址 0 号单元中设置计算参数 n。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。

Data Segment		
Address	Value (+0)	Value (+4)
0x00000000	0x00000064	0x00000000
0x00000020	0x00000000	0x00000000
0x00000040	0x00000000	0x00000000
0x00000060	0x00000000	0x00000000
0x00000080	0x00000000	0x00000000

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，执行 `ecall` 语句后终止程序执行，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。在数据段的地址单元 0x00000000 处，分别输入参数：0x0f,0x0ff,0x0f00,0x0ff00，执行后，记录运算结果。

查看寄存器堆，如果在 `a0 (x10)` 寄存器中存放数据是“0x00c0ffee”，表示程序执行成功。如果是“0xdeaddead”，则表示运算失败，需要进行调试，可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。

Edit Execute					Registers Floating Point Control and Status		
Text Segment					Name	Number	Value
<input type="checkbox"/>	Address	Code	Basic	Source	zero	0	0x00000000
<input type="checkbox"/>	0x00003000	0x00002583	lw x11,0(x0)	5: lw a1,0(x0) # R[a1]=Mem[0], 读取参数n	a*	1	0x00000000
<input type="checkbox"/>	0x00003004	0x00002586	beq x11,x0,0x00000002e	6: beq a1,x0,fail # if n=0 goto fail	sp	2	0x00002ff0
<input type="checkbox"/>	0x00003008	0x00106613	ori x12,x0,1	7: ori a2,x0,1 # R[a2]=1	bp	3	0x00001800
<input type="checkbox"/>	0x0000300c	0x0040e643	xor x13,x13,x13	8: xor a3,a3,a3 # R[a3]=0	fp	4	0x00000000
<input type="checkbox"/>	0x00003010	0x00e88643	add x13,x13,x12	10: add a3,a3,a2 # R[a3]=R[a3]+R[a2]	t0	5	0x00000000
<input type="checkbox"/>	0x00003014	0x0040e643	beq x12,x11,0x00000000e	11: beq a2,a1,finish # if R[a2]=n goto finish	t1	6	0x00000000
<input type="checkbox"/>	0x00003018	0x00106613	addi x12,x12,1	12: addi a2,a2,1 # R[a2]=R[a2]+1	t2	7	0x00000000
<input type="checkbox"/>	0x0000301c	0x005ff06f	jal x0,0xfffffff4	13: jal x0,loop	s0	8	0x00000000
<input type="checkbox"/>	0x00003020	0x0040e223	sw x13,4(x0)	16: sw a3,4(x0) # Store S to Mem[4],Mem[4]<-R[a3]	s1	9	0x00000000
<input type="checkbox"/>	0x00003024	0x00c10537	lui x10,0x000000c10	19: lui a0,0xc10	s2	10	0x00c0ffff
<input type="checkbox"/>	0x00003028	0x00e50513	addi x10,x10,0xfffffff4	20: addi a0,a0,-18 # R[a0]=0xc0c0ffff	s3	11	0x00000004
<input type="checkbox"/>	0x0000302c	0x00000077	ecall	21: ecall # 结束执行	a2	12	0x00000004
<input type="checkbox"/>	0x00003030	0x0040e537	lui x10,0x00000040e	24: lui a0,0x40e	a3	13	0x00001300
<input type="checkbox"/>	0x00003034	0x00e50513	addi x10,x10,0xfffffff4	25: addi a0,a0,-339 # R[a0]=0x40e40e40	a4	14	0x00000000
<input type="checkbox"/>	0x00003038	0x00000077	ecall	26: ecall # 结束执行	a5	15	0x00000000
<input type="checkbox"/>					a6	16	0x00000000
<input type="checkbox"/>					a7	17	0x00000000
<input type="checkbox"/>					s2	18	0x00000000
<input type="checkbox"/>					s3	19	0x00000000
<input type="checkbox"/>					s4	20	0x00000000
<input type="checkbox"/>					s5	21	0x00000000
<input type="checkbox"/>					s6	22	0x00000000
<input type="checkbox"/>					s7	23	0x00000000

Data Segment				
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x00000000	0x00000004	0x00001300	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000

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 Memory To File”按钮，可以将汇编语言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 lab6.3.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。

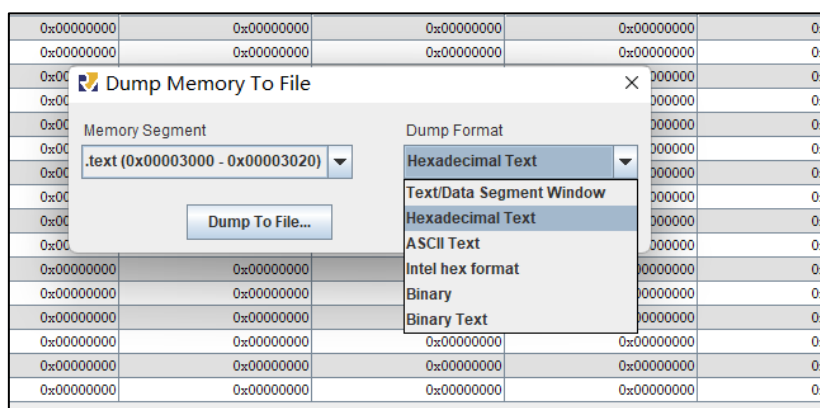


图 6.10 Dump Memory To File 设置

5) 在单周期 CPU 中运行测试程序。

利用文本编辑器打开 lab6.3.hex 文件，在该文件首行前插入文本“v2.0 raw”后保存。在单周期 CPU 的指令存储器（ROM）中该文件 lab6.3.hex，然后在数据存储器 DataRAM 地址单元 0 中设置初始参数，启动时钟脉冲，执行程序，观察输出结果。具体步骤如下：

① 加载累加和测试程序机器代码并设置参数。打开单周期 CPU 电路设计图，在指令存储器 ROM 组件中，加载镜像文件 lab6.3.hex 到指令存储器起始地址 0 单元中，如图 6.11 所示。可通过编辑内容菜单 Edit Content 打开 16 进制编辑器进行查看。

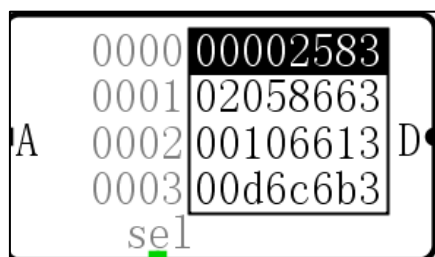


图 6.11 将代码装入指令存储器

在选中数据存储器组件 DataRAM，进入子电路，在最低字节的 RAM0 组件地址 0x0000 处，用鼠标左键点击选中，输入参数 n，假设 n 为十进制数 100，则输入 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。

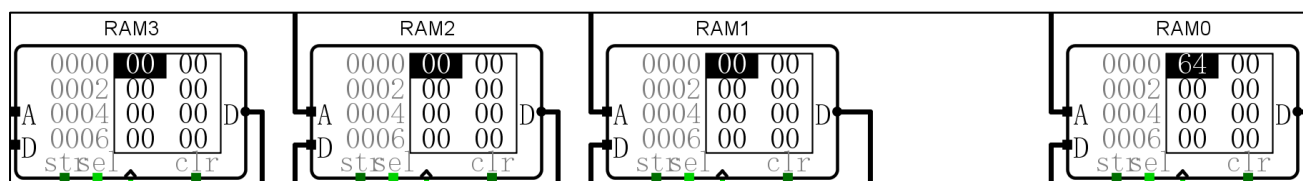


图 6.12 在数据存储器中设置参数

② 测试验证。在 Logisim 的仿真菜单下，首先选中“启用自动仿真 (Ctrl+E)”，然后可以选择“时钟单步 (Ctrl+T)”或选择“时钟连续 (Ctrl+K)”，CPU 就开始单步或连续执行机器代码。当执行最后一条机器代码 0x0073 时，设计电路检测到该指令为 ecall 指令，将设置停止执行信号 Halt 为高电平，PC 寄存器将停止输出，CPU 停止执行，如图 6.13 所示。在程序执行的过程中和程序执行结束后，可通过组合键 Ctrl+K 取消时钟连续中止程序执行，为了提高程序执行的速度，可以在时钟频率中选择合适的频率。



6.13 程序停止执行状态

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地址单元 0x0001 的数据，验证程序执行结果。数据存储器中的数据是按照小端模式存储，地址单元 0x0001 的数据为 0x000013ba，最低字节存储器 RAM0 中存储数据 0xba；在次低字节 RAM1 中存储数据 0x13，如图 6.14 所示。因此累加和测试程序的执行结果为 0x1003ba，等于十进制数 5050，符合 1 到 100 的累计和的计算结果，说明在单周期 CPU 上通过了累加和的测试程序。

分别在数据存储器的 RAM0 和 RAM1 的地址单元 0 中输入 0x0f, 0x0f0, 0x0f00, 0x0ff00 等参数，观测记录测试程序执行结果，并与 Rars 中模拟执行结果进行比对。

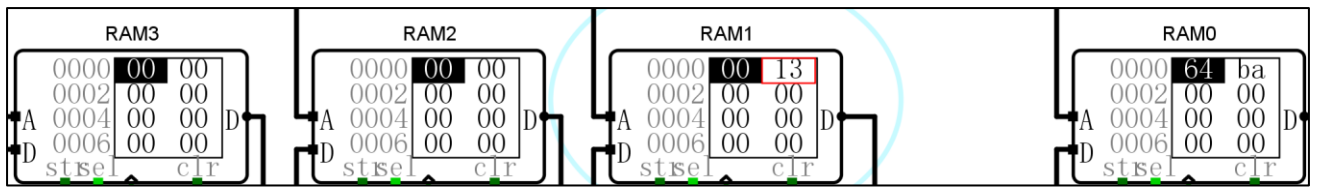


图 6.14 查看 $n=100$ 时数据存储器中的执行结果

如果验证结果不正确，则通过选择“时钟单步 (Ctrl+T)”方式，单步执行每一条指令，检测每条指令的执行状态，查找和分析原因，修改汇编语言程序或电路设计图。

③ 保存电路设计文件。累加和测试程序检测通过后，指令存储器 ROM 加载了测试代码文件 lab6.3-tst.hex，保存电路设计文件为 lab6.3.circ，并在线实验平台上进行评测。

4. 用冒泡排序程序进行 CPU 设计验证

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较，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。

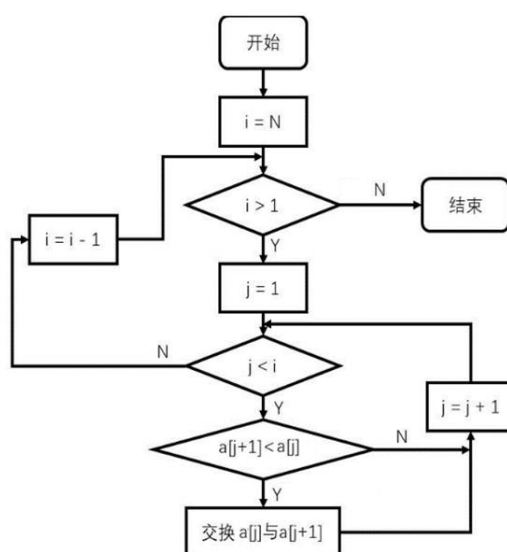


图 6.15 冒泡排序算法流程图

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数据存储器中，从地址单元 0x0000 开始，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 $t0$ ，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
.text
main:
    lw    t0,0(x0)  # R[t0]<-Mem[0],t0 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,t0,x0  # a2, 保存 i, 初始值为 i=n
L1:
    addi  a3,x0,1   # 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
    bge   a7,a6,L4  # a[j]>=a[j+1] 跳转,按照带符号数比较
    sw    a7,0(a4)  # 交换存储
    sw    a6,4(a4)  # 交换存储
L4:
    addi  a3,a3,1   # j=j+1
    bltu  a3,a2,L2  # if j<i then 循环, 序号按照无符号数比较
L3:
    sub   a2,a2,a1  # i--
    bne   a2,a1,L1  # if i>1 then 循环 else 则结束
    ecall           # 结束执行
```

在文本编辑器中输入上述汇编程序，并保存文件为 lab6.4.asm。在 RARS 中打开 lab6.4.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。

在数据段的地址单元 0x0000000 处依次输入：0x0a,0x005678,0x07001234,0xa0020002,0x00a012,0x000340a3,0x8000e756,0x00800adb,0x00d00205,0xff009149,0x007000c7，仿真执行后，记录运算结果，如图 6.16 所示。

Text Segment				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00003000	0x00002283	lw x5, 0(x0)	4: lw t0, 0(x0) = R[t0] < Mem[0], t0保存排序数量n, 待排序的数字个数n存在0x00处
<input type="checkbox"/>	0x00003004	0x00100593	addi x11, x0, 1	5: addi a1, x0, 1 = a1, 保存常量1
<input type="checkbox"/>	0x00003008	0x00028633	add x12, x5, x0	6: add a2, t0, x0 = a2, 保存i, 初始值为i=n
<input type="checkbox"/>	0x0000300c	0x00100693	addi x13, x0, 1	8: addi a3, x0, 1 = a3, 保存j, 初始值为j=1
<input type="checkbox"/>	0x00003010	0x00269713	slli x14, x13, 2	10: slli a4, a3, 2 = a4保存a[j]地址
<input type="checkbox"/>	0x00003014	0x00072803	lw x16, 0(x14)	11: lw a6, 0(a4) = 读取第j个元素
<input type="checkbox"/>	0x00003018	0x00472883	lw x17, 4(x14)	12: lw a7, 4(a4) = 读取第j+1个元素
<input type="checkbox"/>	0x0000301c	0x0108d663	bge x17, x16, 0x0000000c	13: bge a7, a6, L4 = a[j] >= a[j+1] 跳转, 按照带符号数比较
<input type="checkbox"/>	0x00003020	0x01172023	sw x17, 0(x14)	14: sw a7, 0(a4) = 交换存储
<input type="checkbox"/>	0x00003024	0x01072223	sw x16, 4(x14)	15: sw a6, 4(a4) = 交换存储
<input type="checkbox"/>	0x00003028	0x00168693	addi x13, x13, 1	17: addi a3, a3, 1 = j=j+1
<input type="checkbox"/>	0x0000302c	0xfce6e2a3	bltu x13, x12, 0xffffffffe4	18: bltu a3, a2, L2 = if j < i then 循环, 序号按照无符号数比较
<input type="checkbox"/>	0x00003030	0x40b60633	sub x12, x12, x11	20: sub a2, a2, a1 = i--
<input type="checkbox"/>	0x00003034	0xfcb61ce3	bne x12, x11, 0xffffffffd8	21: bne a2, a1, L1 = if i > 1 then 循环 else 则结束
<input type="checkbox"/>	0x00003038	0x00000073	ecall	22: ecall = 结束执行

(a)

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x0000000a	0x00000567	0x07001234	0xa0020002	0x0000a012	0x000340a3	0x0800e756	0x00800adb
0x00000020	0x00400205	0xff009149	0x007000c7	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

(b)

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0xa0020002	0xff009149	0x00000567	0x0000a012	0x000340a3	0x007000c7	0x00800adb	
0x00000020	0x00400205	0x07001234	0x0800e756	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

图 6.16 (a) 仿真执行窗口及待排序数据 (b) 冒泡排序编译仿真执行结果

排序结果验证通过后，导出汇编程序机器代码到文本文件 lab6.4.hex，然后添加首行字符串“v2.0 raw”后保存，作为测试程序可执行代码镜像文件。

在单周期 CPU 中，数据存储器使用了 4 个单字节 RAM 来实现，因此需要使用文件编辑器将待排序的数据分解到 4 个数据镜像文件 lab6.4_d.hex0~lab6.4_d.hex3，并在首行加入“v2.0 raw”。

在 Logisim 中打开单周期 CPU 电路图 lab6.2.circ，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 lab6.4.hex，在数据存储器 RAM0~RAM3 中分别加载待排序数据镜像文件 lab6.4_d.hex0~lab6.4_d.hex3。在 Logisim 的仿真菜单下，选中时钟连续，CPU 开始自动执行机器代码，直到执行 ecall 指令，此时按 Ctrl+K 终止“时钟连续”执行方式。查看数据存储器中内容，验证执行冒泡测试程序后的数据排列结果，如图 6.17 所示。

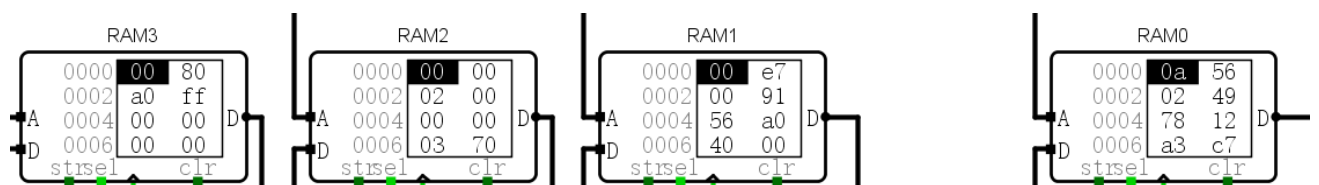


图 6.17 数据存储器中排序结果

把数据存储器中将排序后的结果保存到镜像文件，验证排序后的数据次序。

当冒泡测试程序检测通过后，在指令存储器 ROM 加载了测试代码文件 lab6.4-tst.hex，保存电路设计文件为 lab6.4.circ，并在线实验平台上进行评测。

5. 官方测试集测试

RISC-V 社区开发了官方测试集,可以在 github 上直接下载 <https://github.com/riscv-software-src/riscv-tests>。

如果需要使用官方测试集,可以运行下列命令:

- 1) `$ git clone https://github.com/riscv/riscv-tests`
- 2) `$ cd riscv-tests`
- 3) `$ git submodule update --init --recursive`
- 4) `$ autoconf`
- 5) `$./configure --with-xlen=32`
- 6) `$ make isa`

前 3 句是从 github 上下载源代码,第 4-5 句是生成 makefile,最后是 make 测试集。

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,u 表示是用户态,i 表示是整数基本指令集。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。所以,主要关注 rv32ui-p 开头的测试即可。

官方测试集需要使用 risc-v gcc 工具链来编译,在官方测试集的代码中使用了系统调用指令,需要对官方测试的代码进行一定的修改才生成用于本次实验的测试文件。有关要求超出了本课程的范围,因此本次实验提供已经编译通过后测试文件,在 testcase 文件夹中。每条 RV32I 的指令通常有 3 个文件,一个二进制可执行文件,一个根据二进制可执行文件反汇编生成的.dump 文件,还有一个是用于加载到 CPU 的指令存储器的测试代码.hex 文件。在 Load 和 Store 指令测试程序中,还需要在数据存储器中加载测试数据,不同字节位置的数据分别使用 hex0~hex3 来表示,在测试时加载到数据存储器的 RAM0~RAM3 中。

加载指令测试代码后,选择连续时钟信号,执行程序。如果指令测试通过,则在 a0 寄存器中的数据为 0x00c0ffee,如果测试不通过,则 a0 寄存器中的数据为 0xdeaddead。

依次加载所有的测试程序,通过官方测试集的验证。

6. 计算机系统基础 PA 程序测试

除了通过官方汇编语言的测试集验证,还可以验证 C 程序。比如,可以验证后续《计算机系统基础》课程中 PA 项目中的测试程序。具体方法如下:

1、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`

2、修改文件权限。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
```

3、设置环境参数。在 Ubuntu 下执行下列命令：

`cd ~`，进入用户目录，显示 `abstract-machine` 路径。

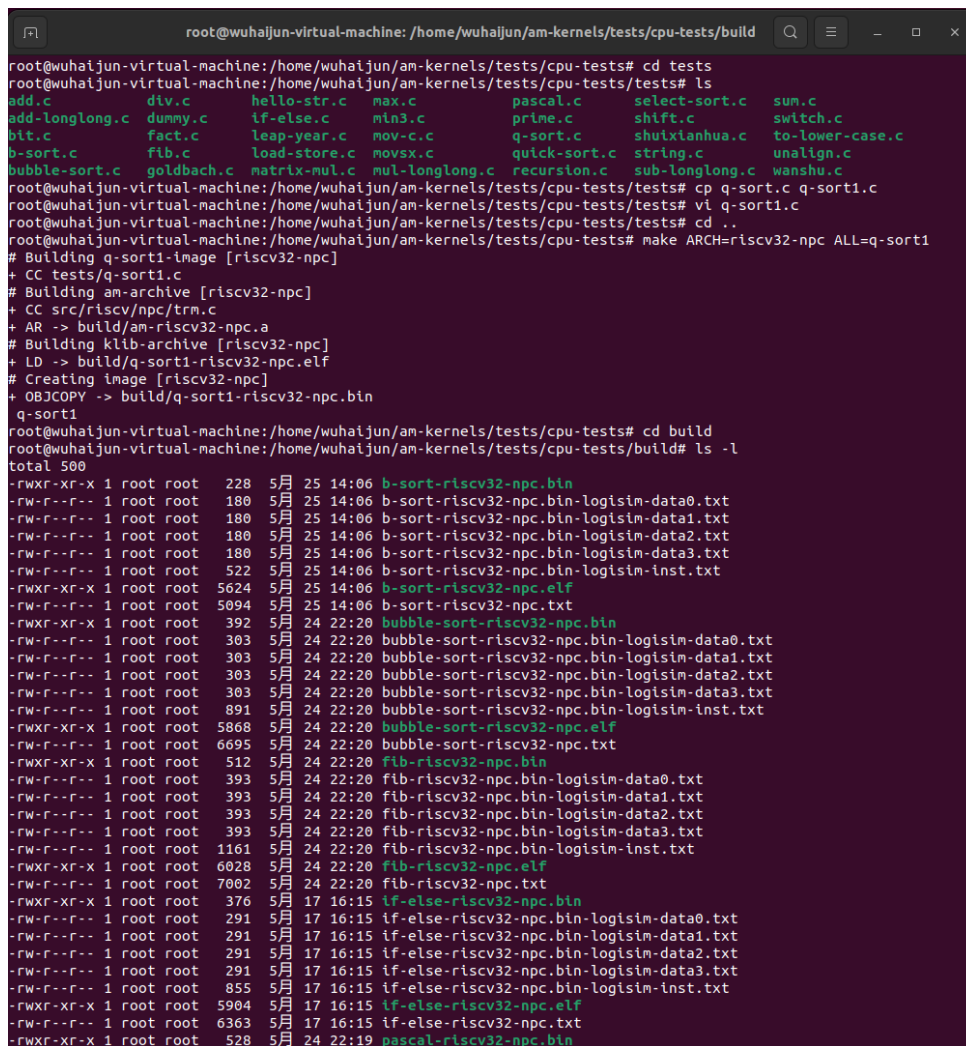
`export AM_HOME=`pwd`/abstract-machine`，使得 `AM_HOME` 和 `abstract-machine` 目录的路径一致。

`cd am-kernels/tests/cpu-tests`，在 `tests` 子目录中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。

4、生成测试文件。在 `am-kernels/tests/cpu-tests` 目录下执行下列命令：

`make ARCH=riscv32-npc ALL=bubble-sort`

则生成可用于 Logisim 下 CPU 测试的指令镜像文件 `bubble-sort-riscv32-npc.bin-logisim-inst.txt` 和 4 个按字节分开的数据镜像文件 `bubble-sort-riscv32-npc.bin-logisim-data0.txt~bubble-sort-riscv32-npc.bin-logisim-data3.txt`。



例如 `bubble-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。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~bubble-sort-riscv32-npc.bin-logisim-data3.txt 依次加载到数据存储器的 RAM0~RAM3 中。选择连续时钟信号，启动程序执行。执行结束后，查看数据存储器，分析程序执行的结果。如果遇到问题，可根据 bubble-sort-riscv32-npc.txt 文件中的 RV32I 反汇编程序代码进行检测。

修改 ALL=后面的文件名“bubble-sort”为其他 C 程序文件名，则生成该 C 程序的测试文件。

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 目录下提供的测试文件进行验证。

四、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。
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。
3. 如何在单周期 CPU 基础上实现多周期 CPU？