



**《计算机组成原理》**

**课程实验报告**

姓名： 苏一涵

学院：信息学院

系：软件工程

专业： 软件工程

学号：36720232204041

2025年4月

**第4次实验 指令系统**

1. **实验目的**
2. **掌握MIPS汇编仿真器（MARS 4.5）、RISC-V汇编仿真器（RARS 1.5）、Intel x86汇编工具（masm32）、ARMv7在线汇编工具的使用方法。**
3. **在Logisim实现的单周期MIPS处理器、Logisim实现的单周期RISC-V处理器上运行程序。**
4. **分别使用MIPS指令、RISC-V指令、x86指令、ARMv7指令编写排序程序。**
5. **实验环境**

**（1）MARS 4.5、RARS 1.5、masm32、ARMv7在线汇编工具。**

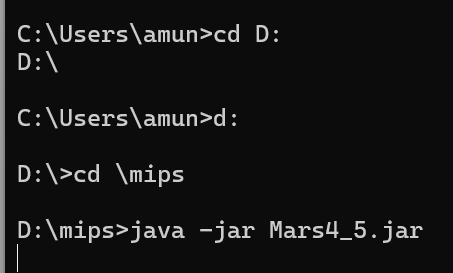
**（2）Windows系统下运行Logisim软件（需安装JDK）。**

1. **实验内容**
   1. **验证实验**
2. **MIPS汇编语言程序的运行**

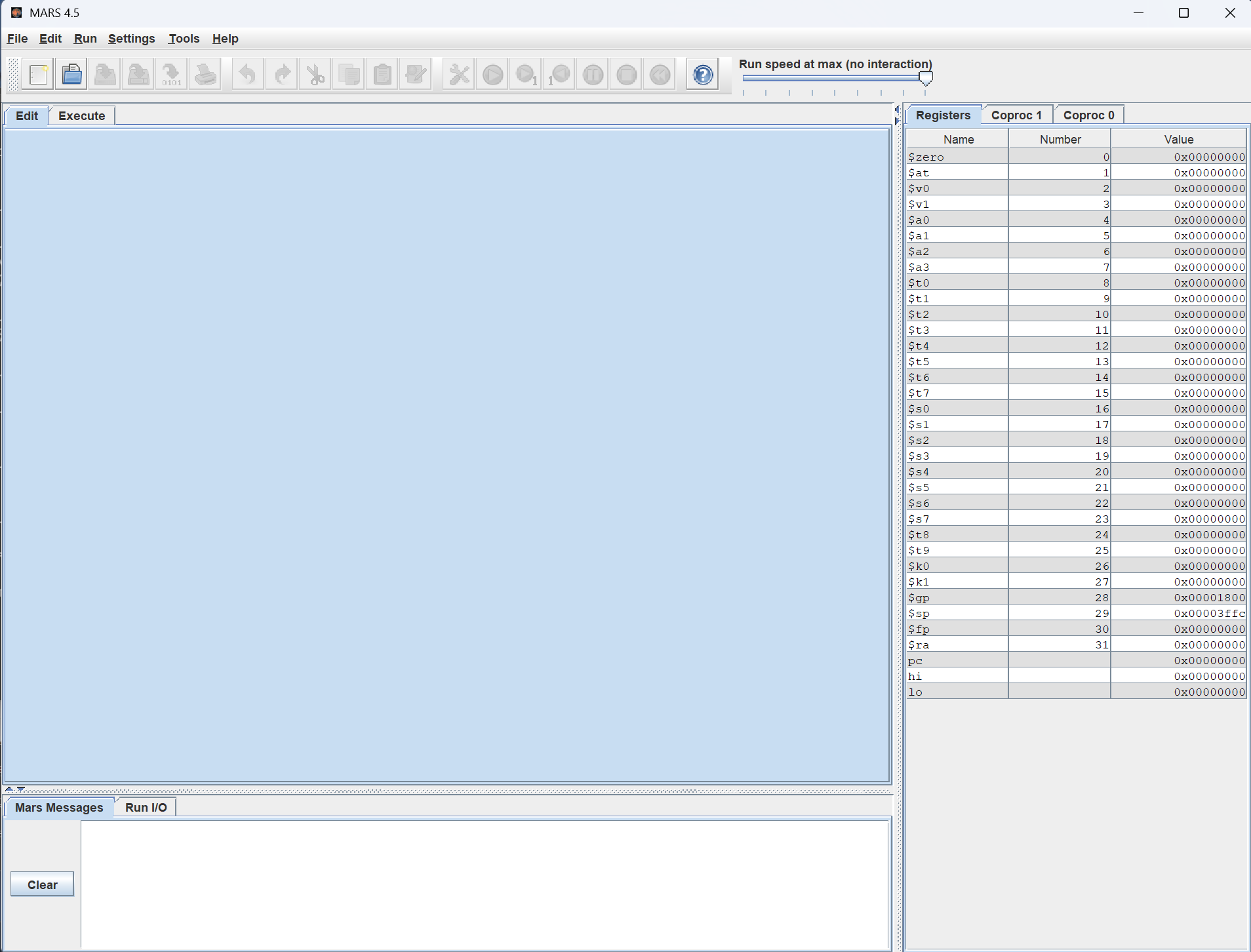
**Ⅰ：运行MIPS汇编仿真器**

**首先将给出的五个文件夹放入D盘根目录，然后win+r进入cmd界面**

**进行如下操作：**

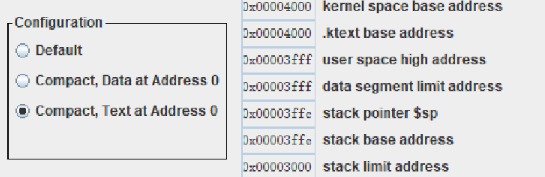


**运行成功**

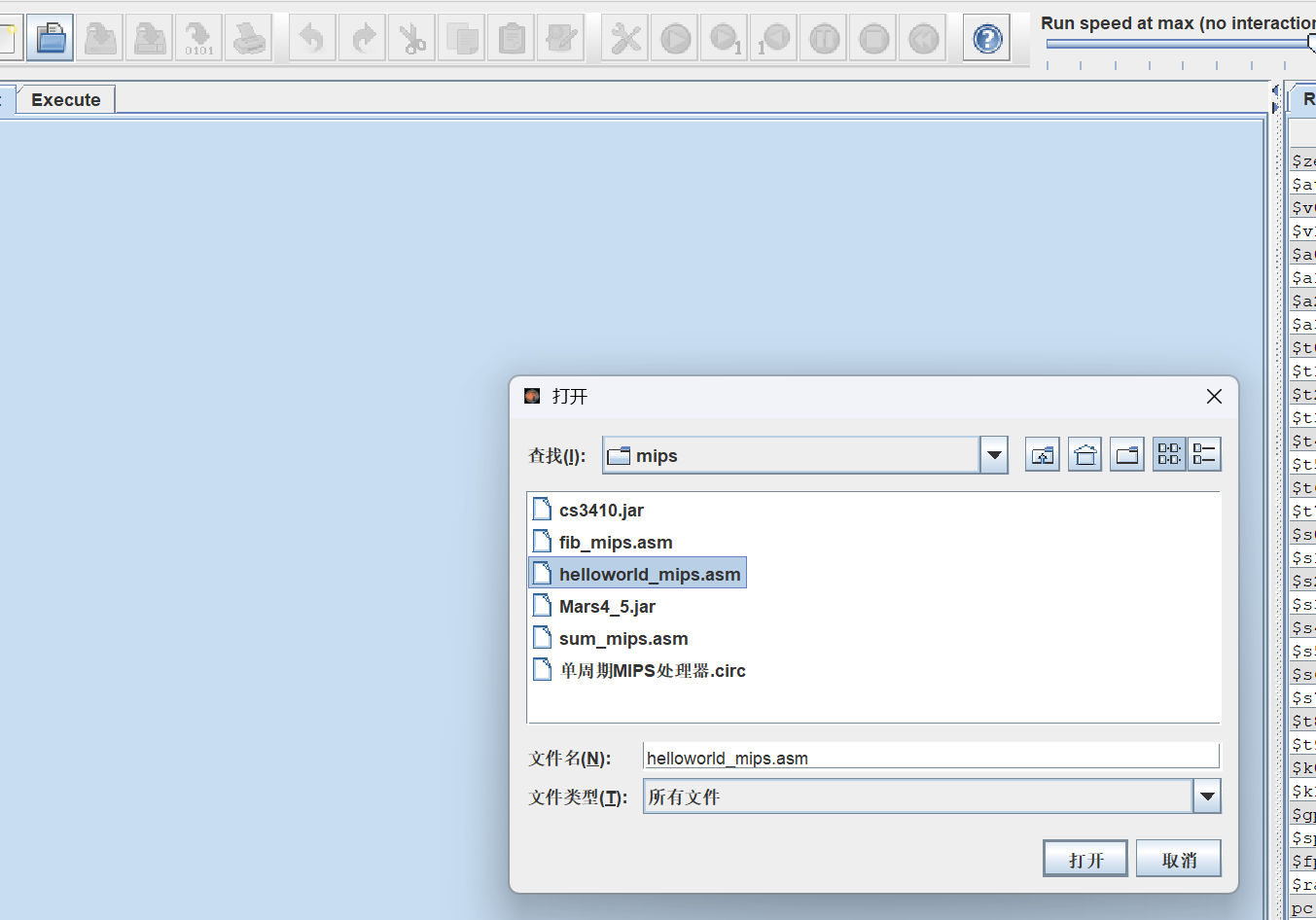


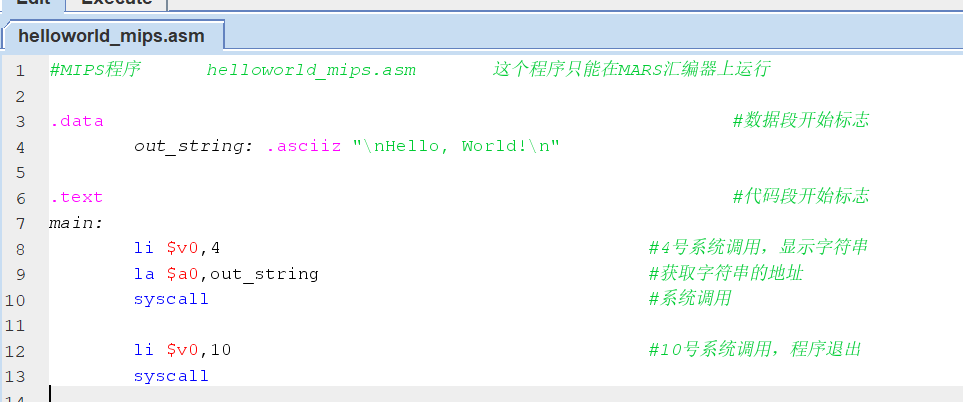
**Ⅱ：运行第一个MIPS汇编语言程序**

**先将 MARS 汇编仿真器的代码段的开始地址设置为 0**

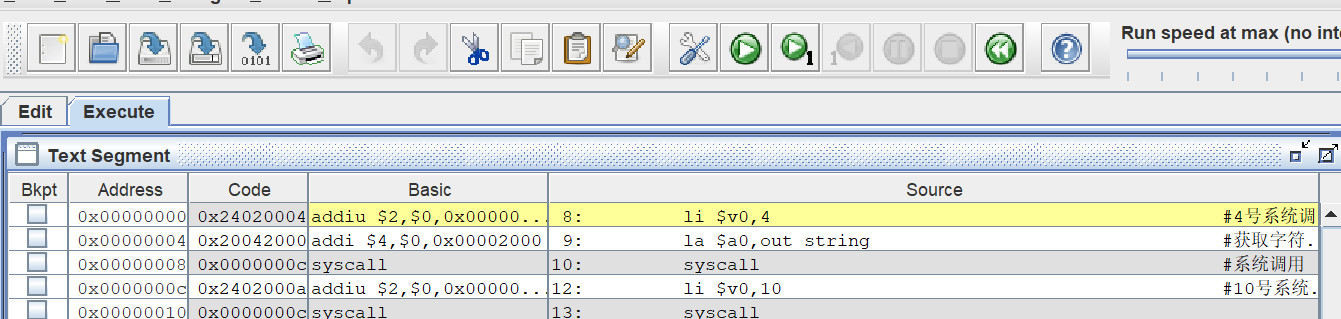


**然后在MARS汇编仿真器中打开helloworld\_mips.asm**

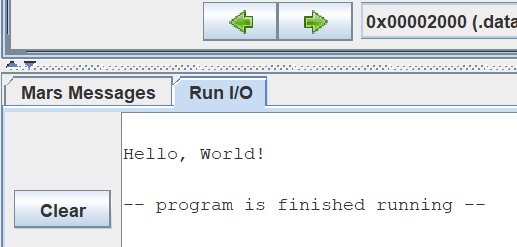




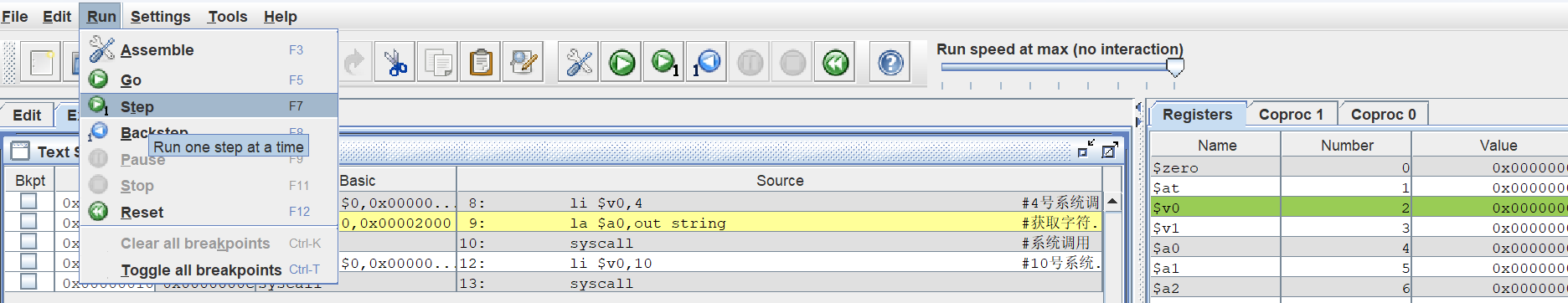
**再对helloworld\_mips.asm源程序进行汇编**

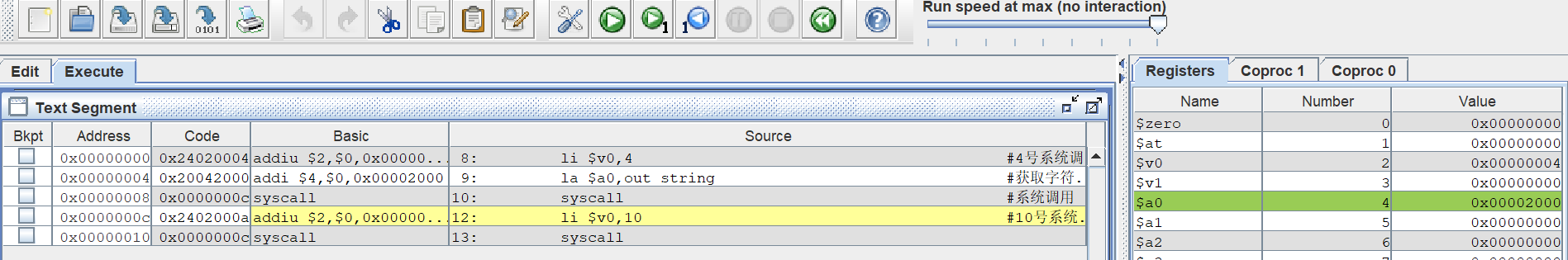


**运行汇编后的程序**



**逐步执行**

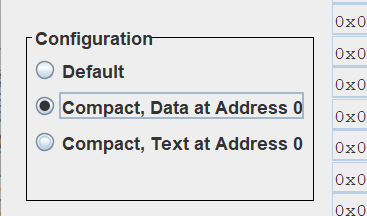






**Ⅲ：求累加和的MIPS汇编语言程序**

**首先将MARS汇编仿真器的数据段的开始地址设置为0**

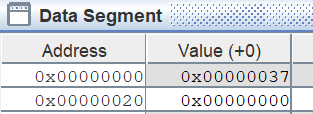


**在MARS汇编仿真器中打开sum\_mips.asm**

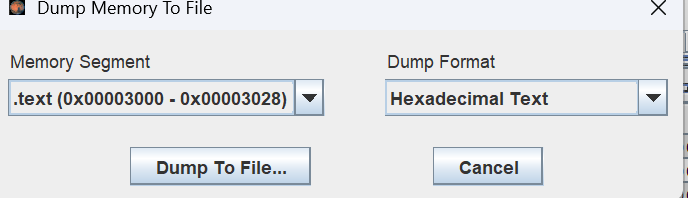
**对sum\_mips.asm源程序进行汇编**

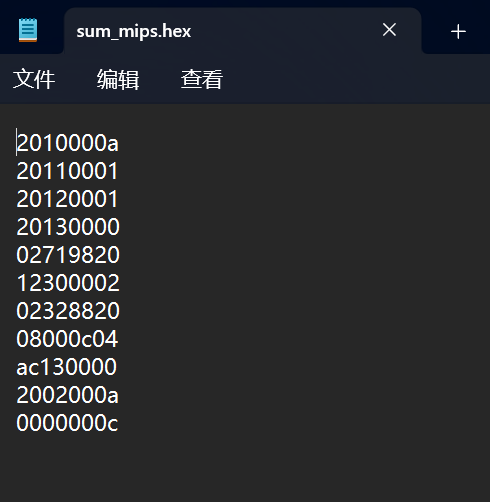
**运行汇编后的程序**

**观看数据段中的内容**



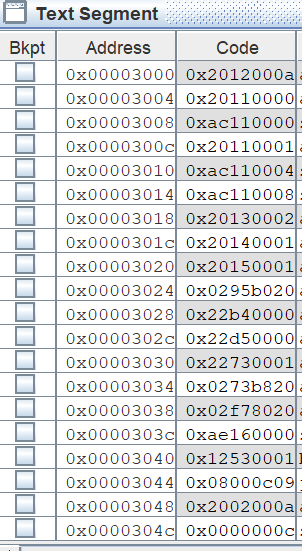
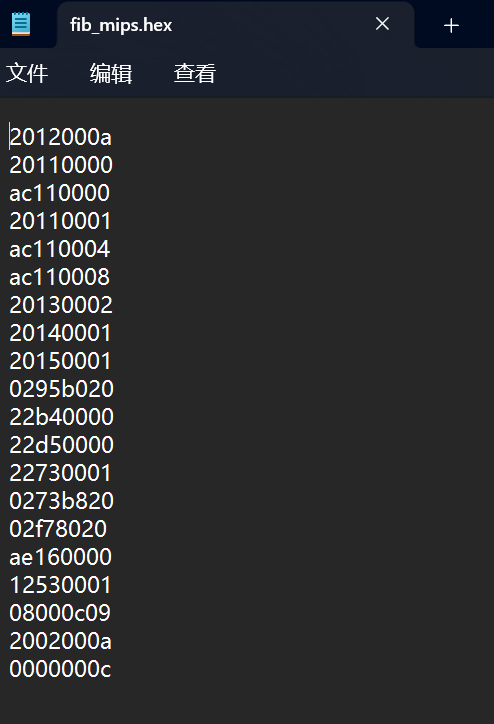
**再导出汇编后的机器码**





**Ⅳ：计算费波那契数列的MIPS汇编语言程序**

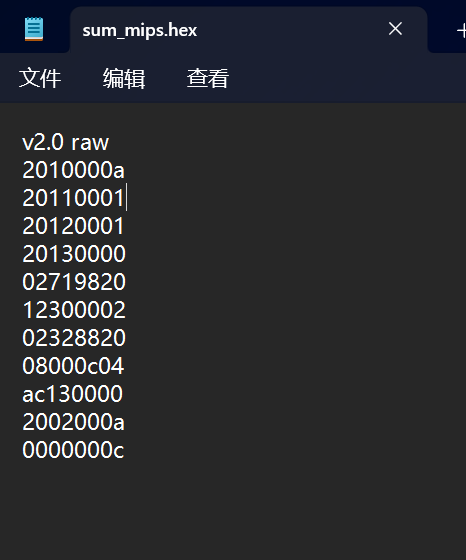
**在MARS 4.5汇编仿真器中运行该程序，并将该程序的机器码保存到fib\_mips.hex文件中**



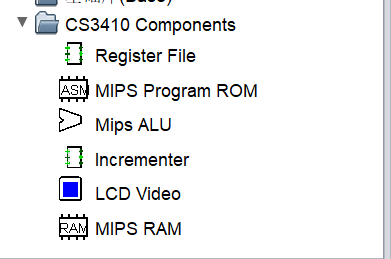
1. **在Logisim上运行MIPS程序**

**Ⅰ：在Logisim实现的单周期MIPS处理器上运行求累加和程序**

**第一步：修改前面导出的机器码文件sum\_mips.hex，增加：v2.0 raw**



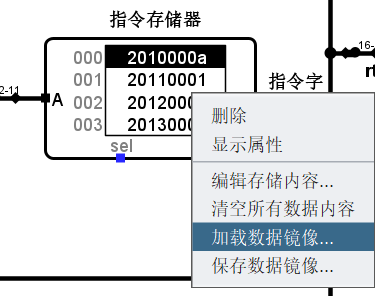
**第二步：运行Logisim，在Logisim中增加寄存器文件库“cs3410.jar”（位于\mips目录中）**



**第三步：在Logisim中打开设计文件“单周期MIPS处理器.circ”（位于\mips目录中）**



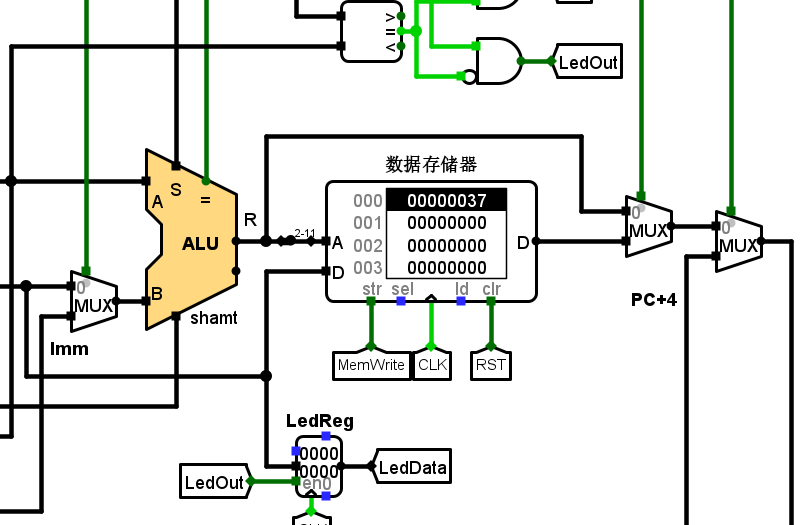
**第四步：在单周期MIPS处理器数据通路的指令存储器中装入机器码文件sum\_mips.hex**

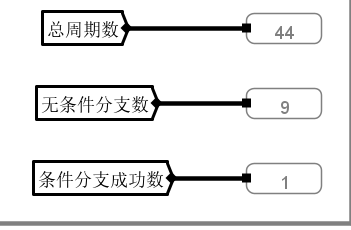


**第五步：设置时钟频率=1KHz，按Ctrl+K启动时钟，程序开始执行**

**第六步：按总复位，程序重新执行；指令存储器中的内容停止滚动后，表示程序执行结束；再按Ctrl+K停止时钟**

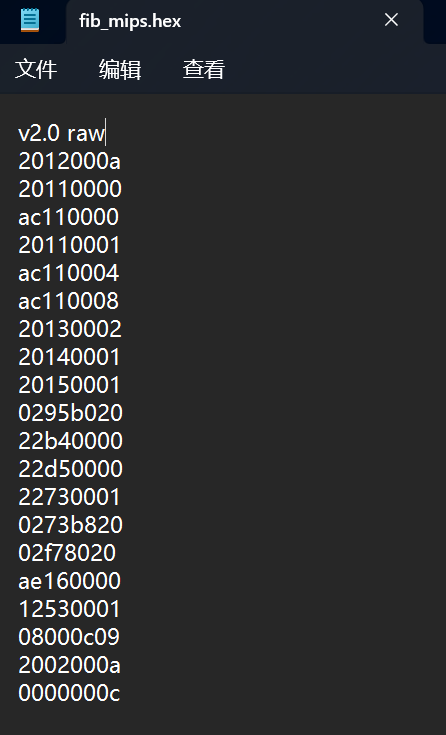
**第七步：此时观察数据存储器第0号单元的值（累加和，正确值为37h=55），以及程序运行的总周期数、无条件分支数、条件分支成功数**



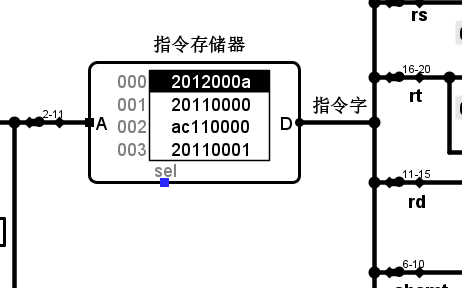


**Ⅱ：在Logisim实现的单周期MIPS处理器上运行计算费波那契数列程序**

**第一步：修改前面导出的机器码文件fib\_mips.hex，增加：v2.0 raw**



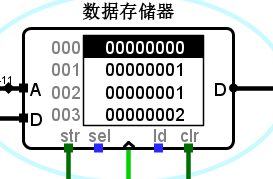
**第二步：在单周期MIPS处理器数据通路的指令存储器中装入机器码文件fib\_mips.hex**

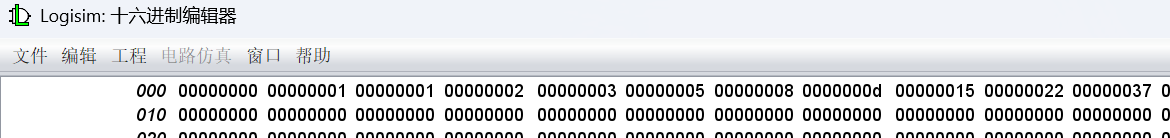


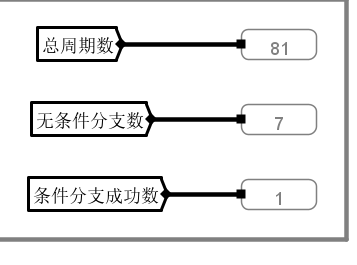
**第三步：设置时钟频率=1KHz，按Ctrl+K启动时钟，程序开始执行**

**第四步：按总复位，程序重新执行；指令存储器中的内容停止滚动后，表示程序执行结束；再按Ctrl+K停止时钟**

**第五步：此时观察数据存储器中的内容（0号单元开始的11个单元），以及程序运行的总周期数、无条件分支数、条件分支成功数**



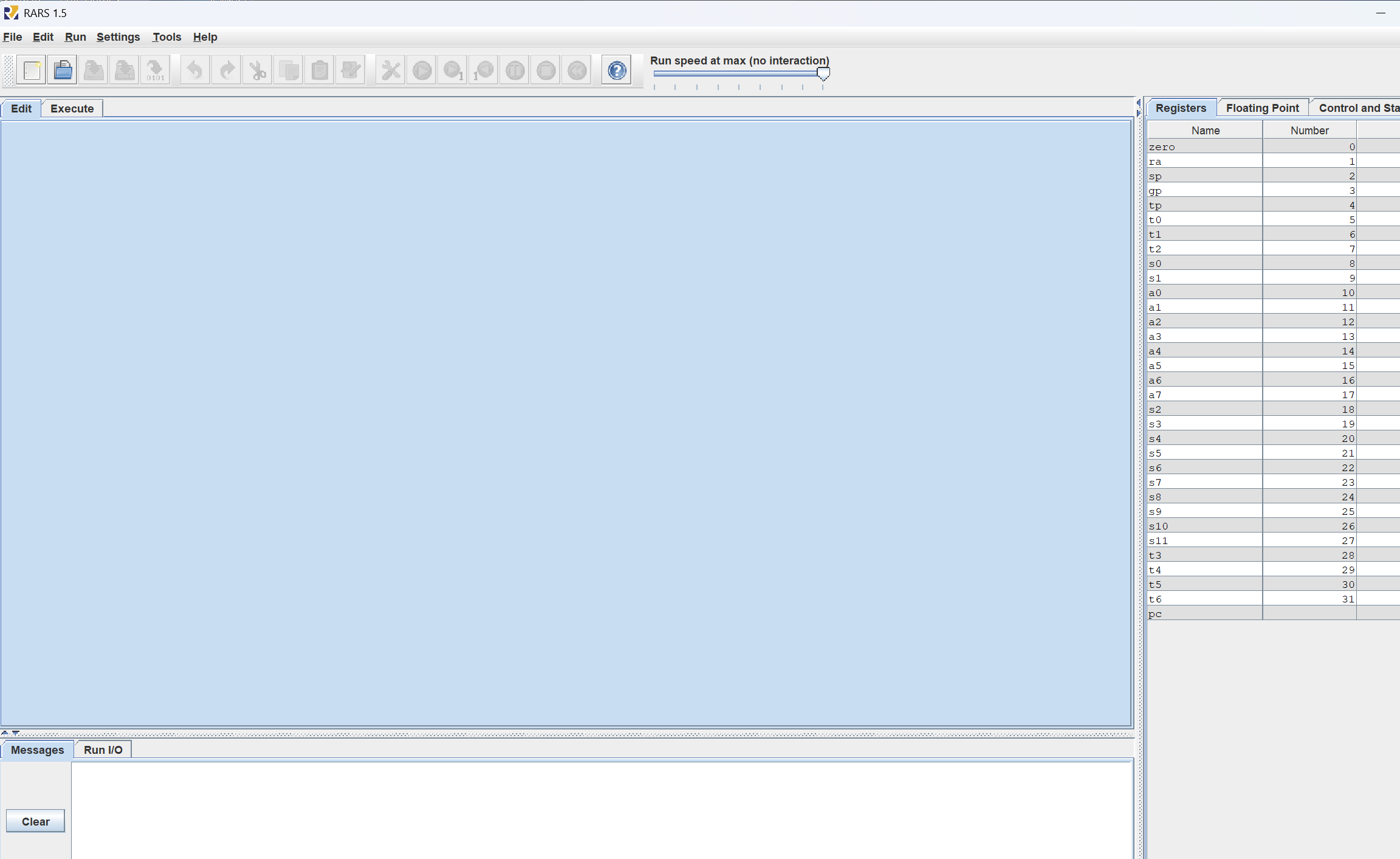




1. **RISC-V汇编语言程序的运行**

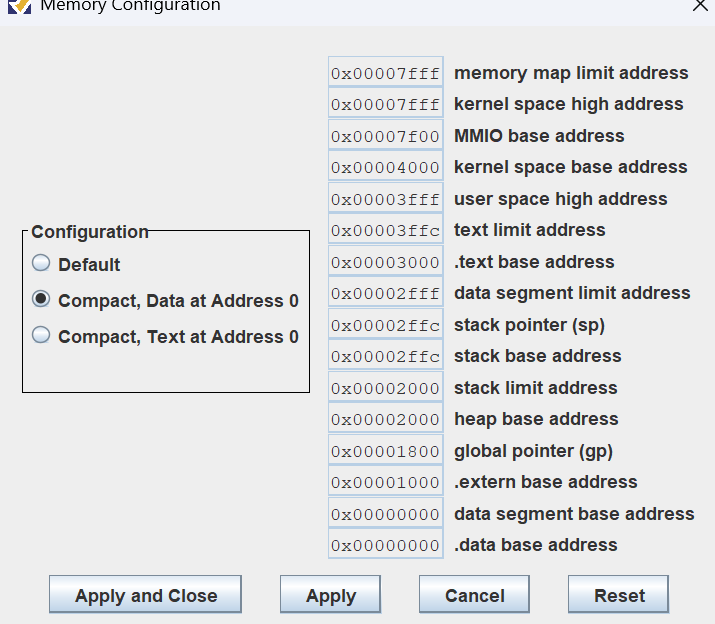
**Ⅰ：运行RISC-V汇编仿真器**



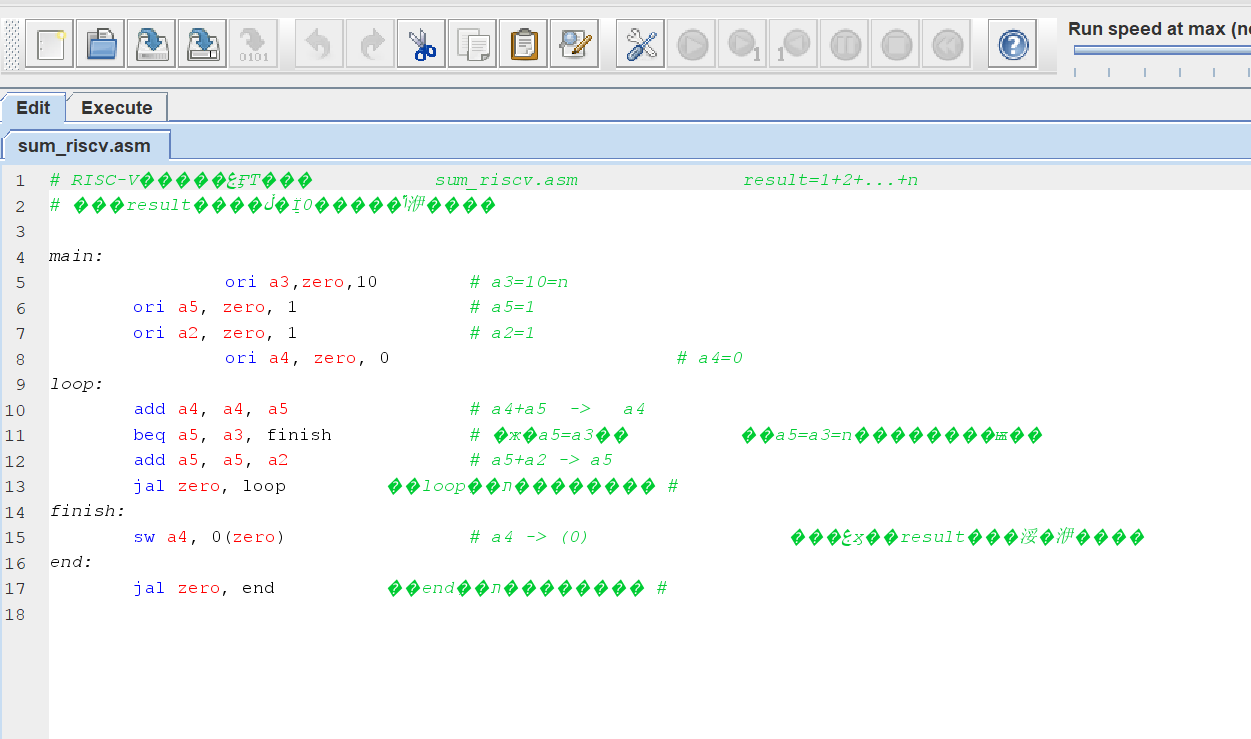


**Ⅱ：运行RISC-V求累加和程序**

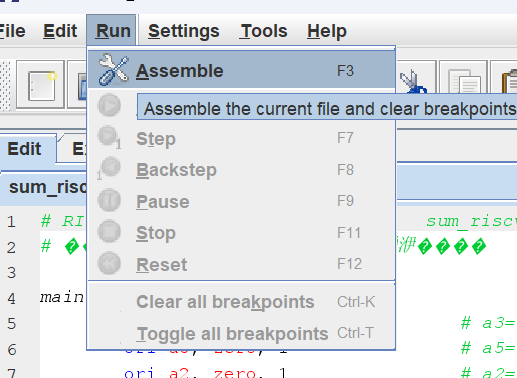
**第一步：将RARS汇编仿真器的数据段的开始地址设置为0**

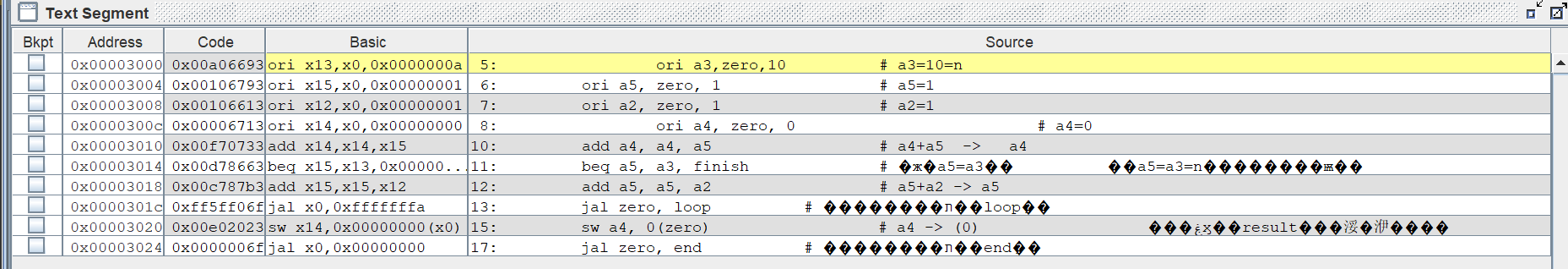


**第二步：在RARS中打开“sum\_riscv.asm”**

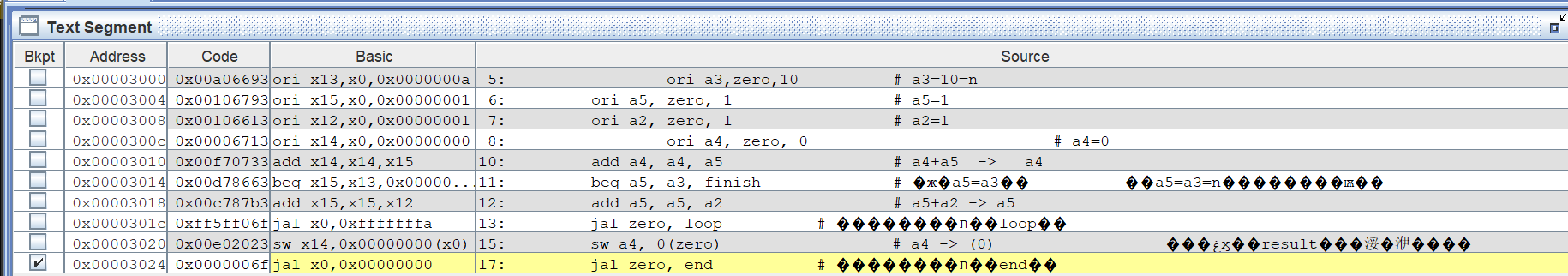


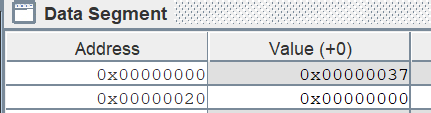
**第三步：在RARS中汇编“sum\_riscv.asm”**



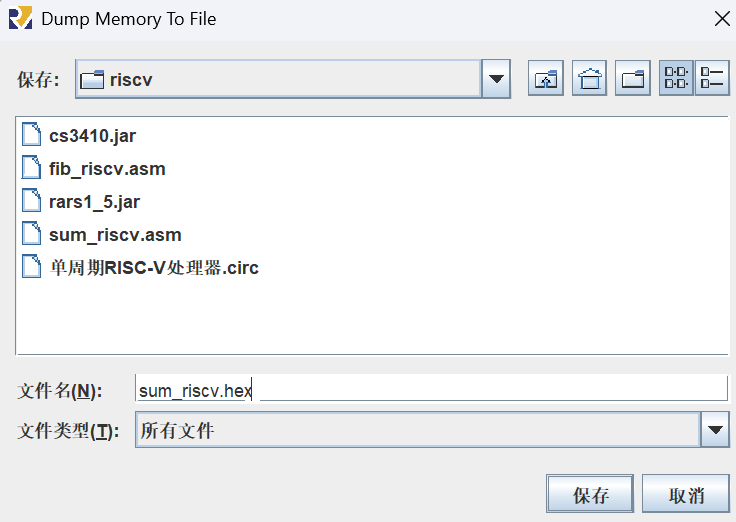


**第四步：在RARS中运行“sum\_riscv.asm”**



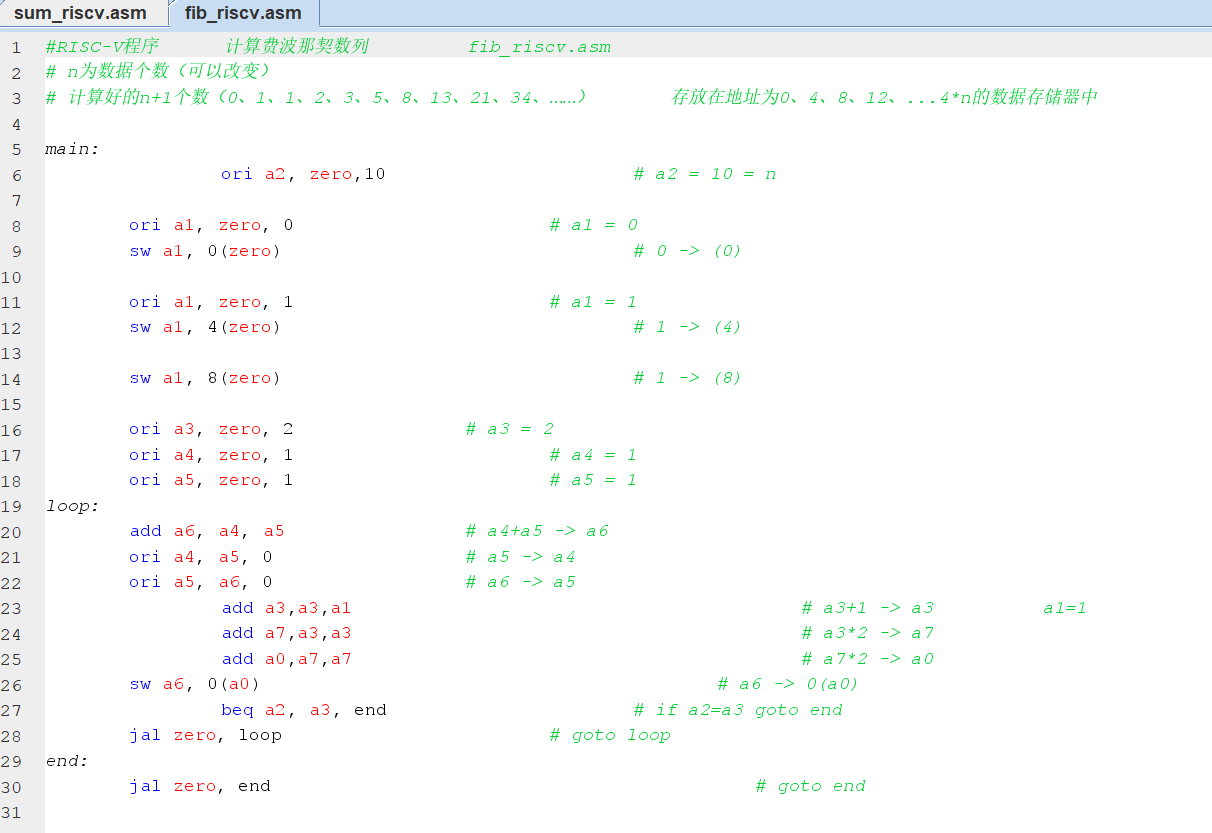
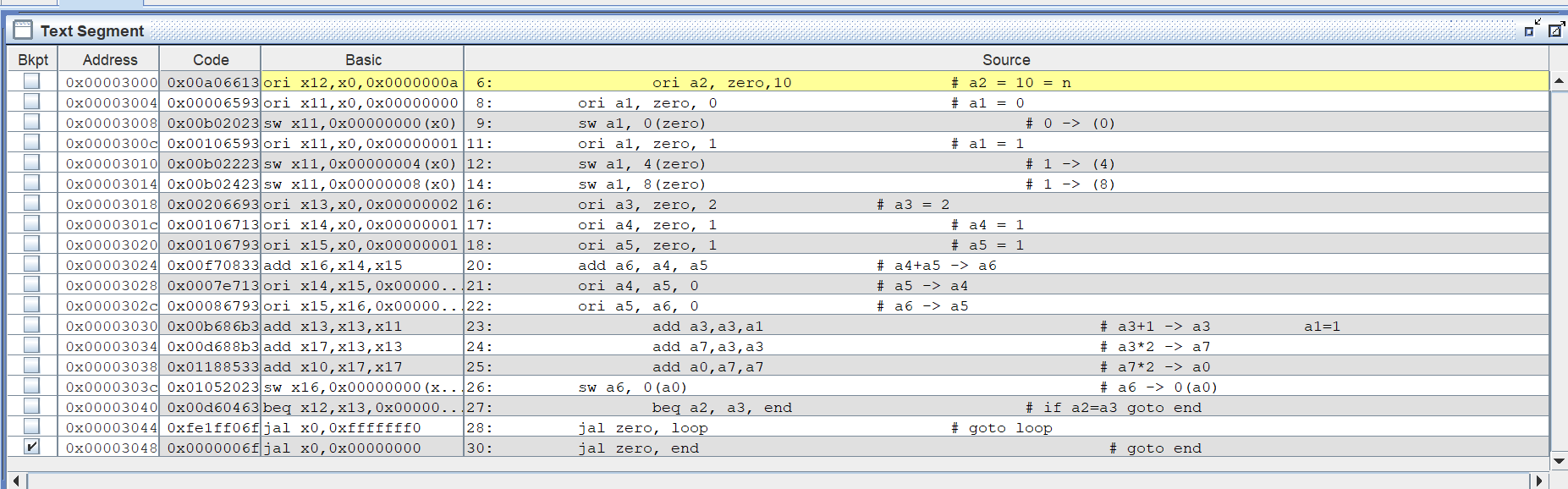


**第五步：将汇编后的机器码，导出到一个文本文件中**

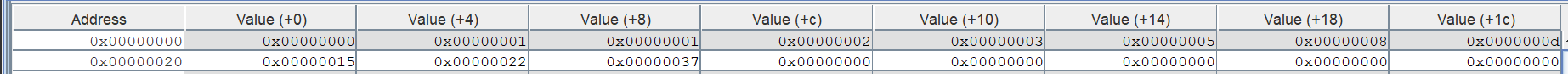


**Ⅲ：运行RISC-V计算费波那契数列的程序**

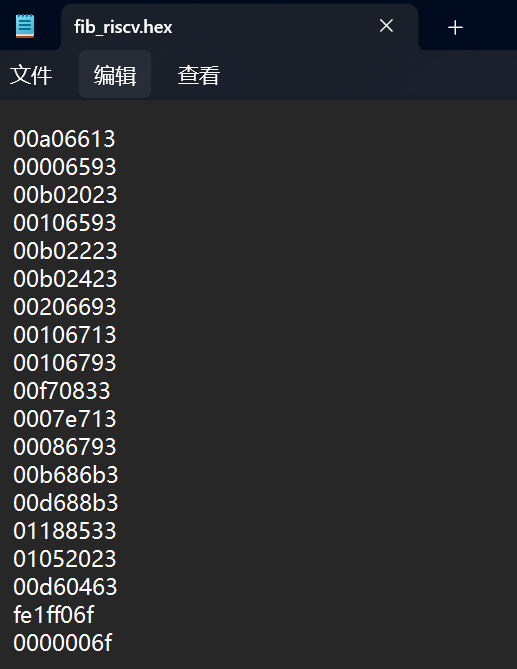
**在RARS 1.5汇编仿真器中运行该程序，并将该程序的机器码保存到fib\_riscv.hex文件中。**

结果如下



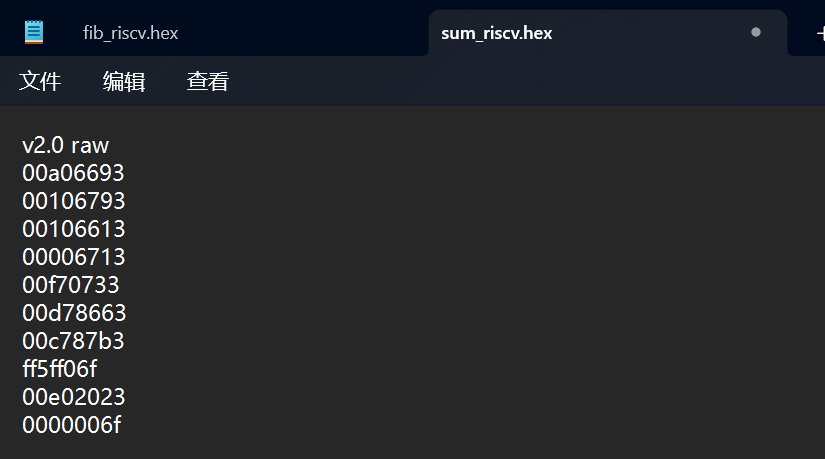
保存



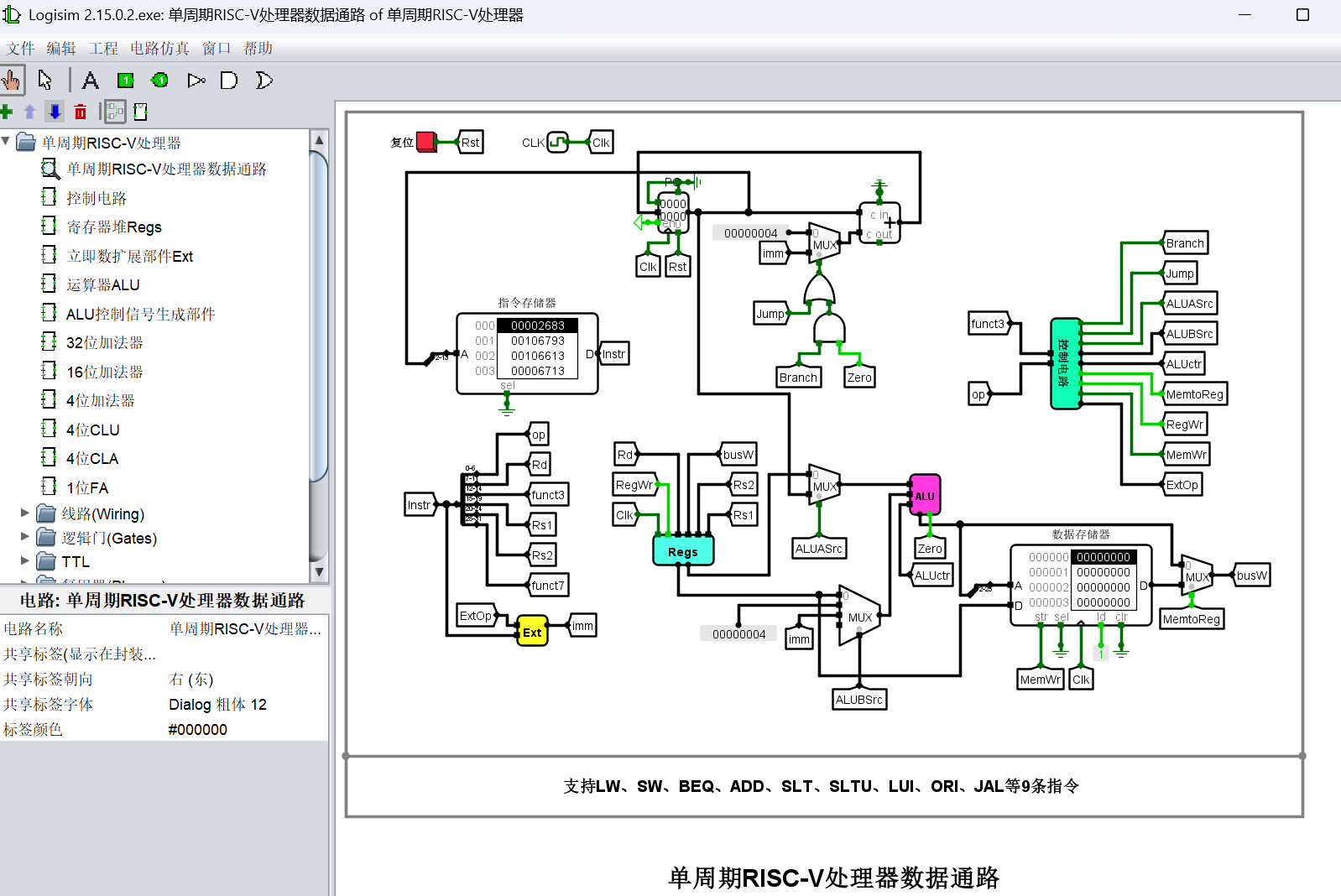
1. **在Logisim上运行RISC-V程序**

**Ⅰ：在Logisim实现的单周期RISC-V处理器上运行求累加和的程序**

**第一步：修改前面导出的机器码文件sum\_riscv.hex，增加：v2.0 raw**

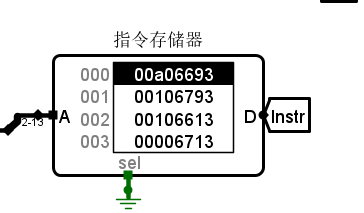


**第二步：在Logisim中打开设计文件“单周期RISC-V处理器.circ”（位于\riscv目录中）**

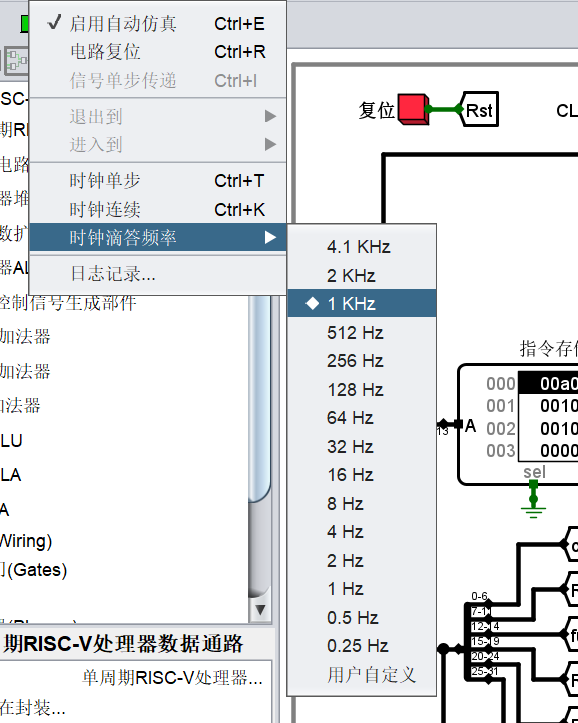


**第三步：按Ctrl+R**

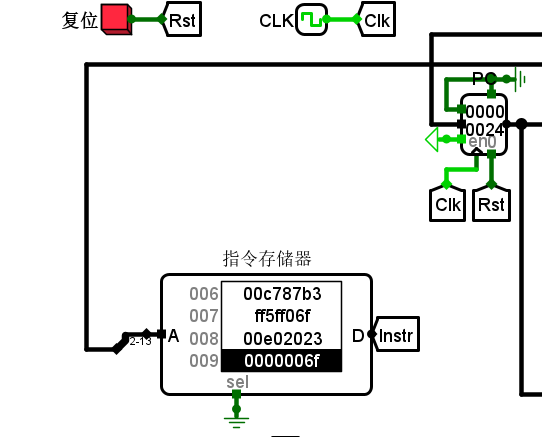
**第四步：在单周期RISC-V处理器数据通路的指令存储器中装入机器码文件sum\_riscv.hex**



**第五步：设置时钟频率=1KHz，按Ctrl+K启动时钟，程序开始执行**

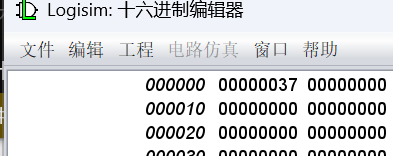


**第六步：指令存储器停止滚动后，表示程序执行结束**



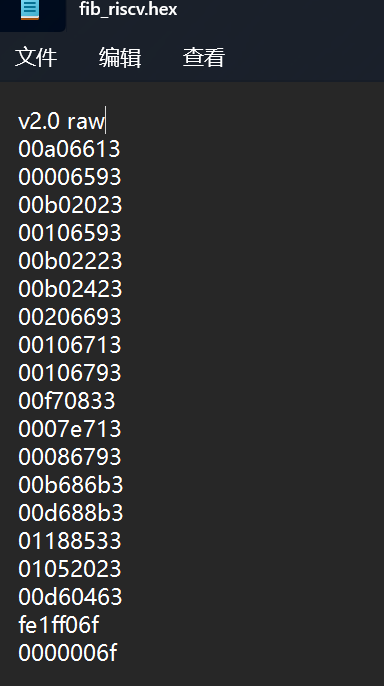
**第七步：再按Ctrl+K停止时钟**

**第八步：此时观察数据存储器第0号单元的值（累加和，正确值为37h=55）**



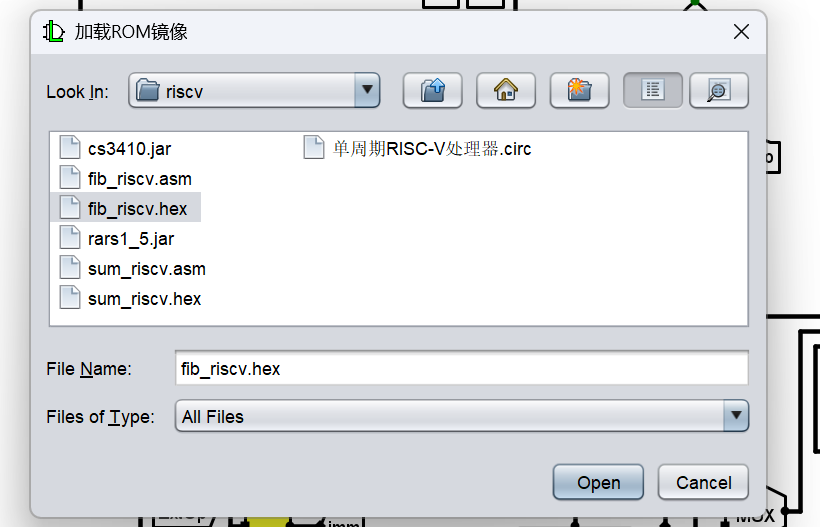
**Ⅱ：在Logisim实现的单周期RISC-V处理器上运行计算费波那契数列的程序**

**第一步：修改前面导出的机器码文件fib\_riscv.hex，增加：v2.0 raw**



**第二步：在Logisim中按Ctrl+R**

**第三步：在指令存储器中装入机器码文件fib\_riscv.hex**



**第四步：设置时钟频率=1KHz，按Ctrl+K启动时钟，程序开始执行**

**第五步：指令存储器停止滚动后，表示程序执行结束**

**第六步：再按Ctrl+K停止时钟**

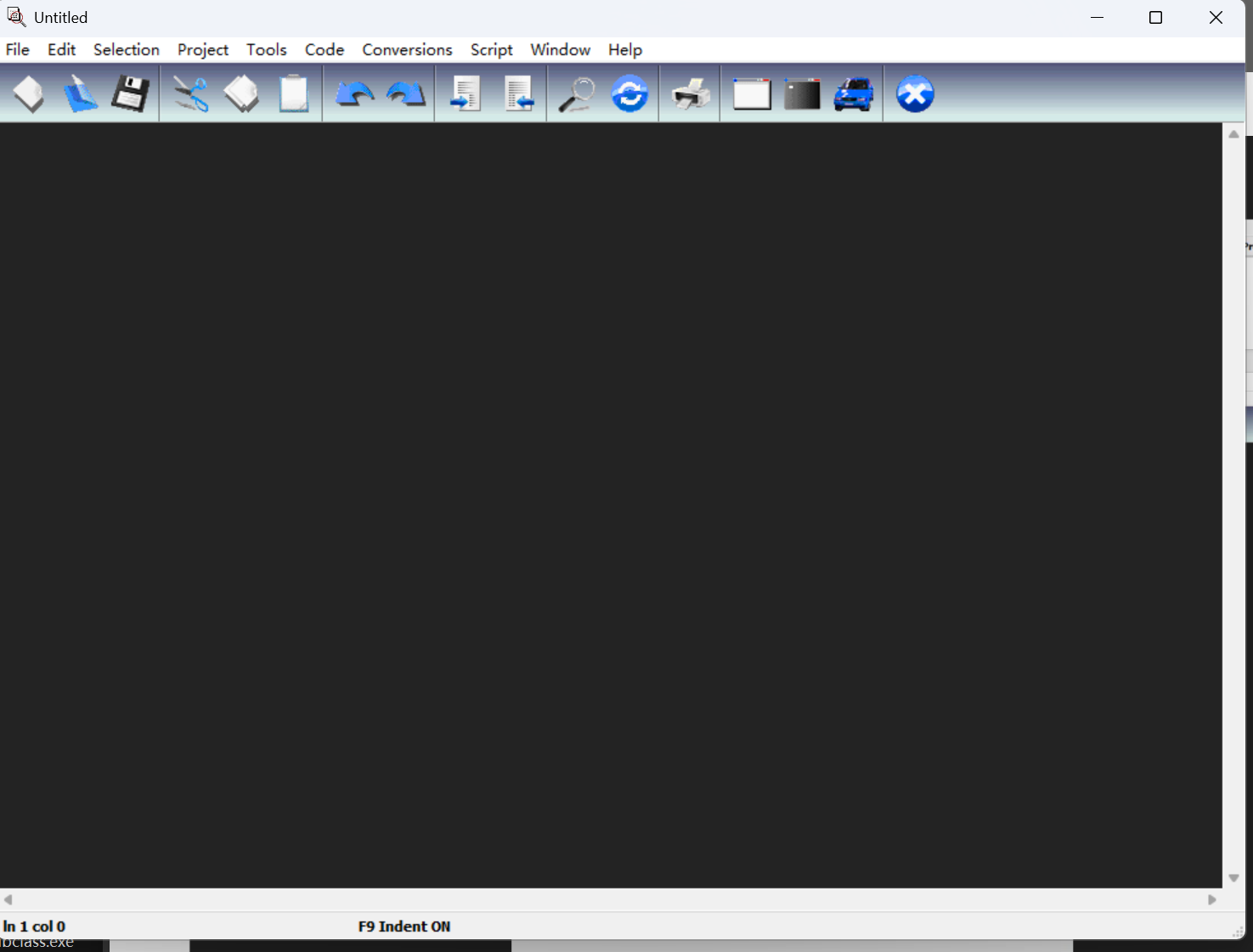
**第七步：此时观察数据存储器中的内容（0号单元开始）**



1. **Intel x86汇编语言程序的运行**

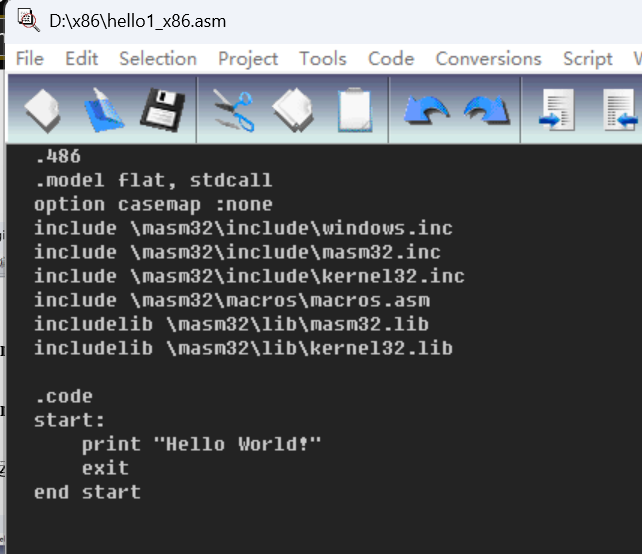
**Ⅰ：Intel x86汇编工具**

**点击运行\masm32目录中的“qeditor.exe”。**

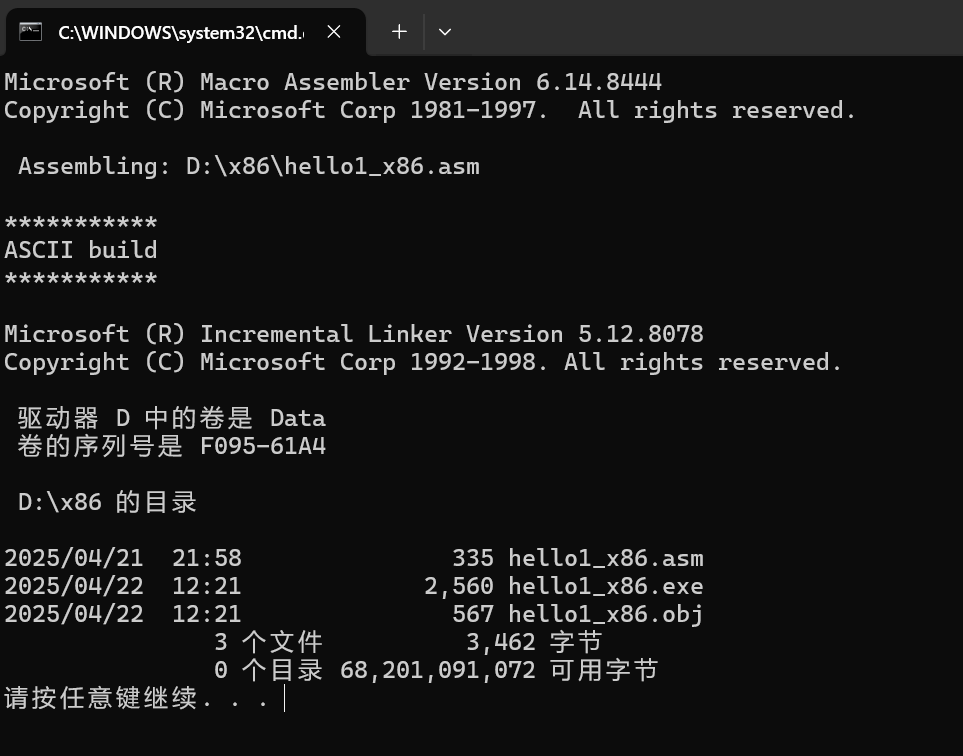


**Ⅱ：运行Intel x86汇编语言程序**

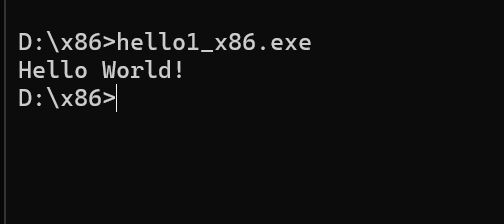
**第一步：在masm32汇编工具中打开源程序“hello1\_x86.asm”**



**第二步：汇编**



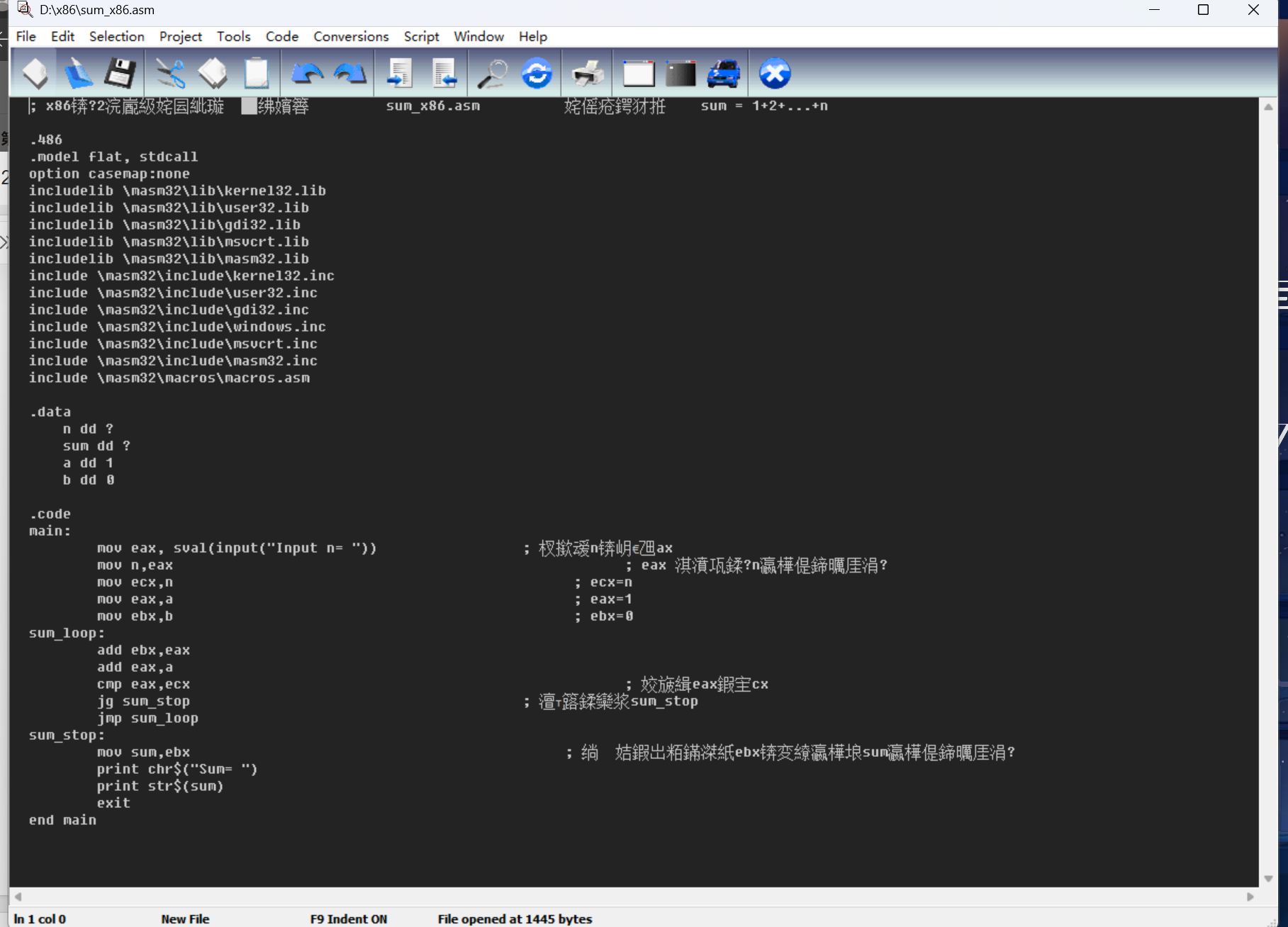
**第三步：运行“hello1\_x86.exe”**



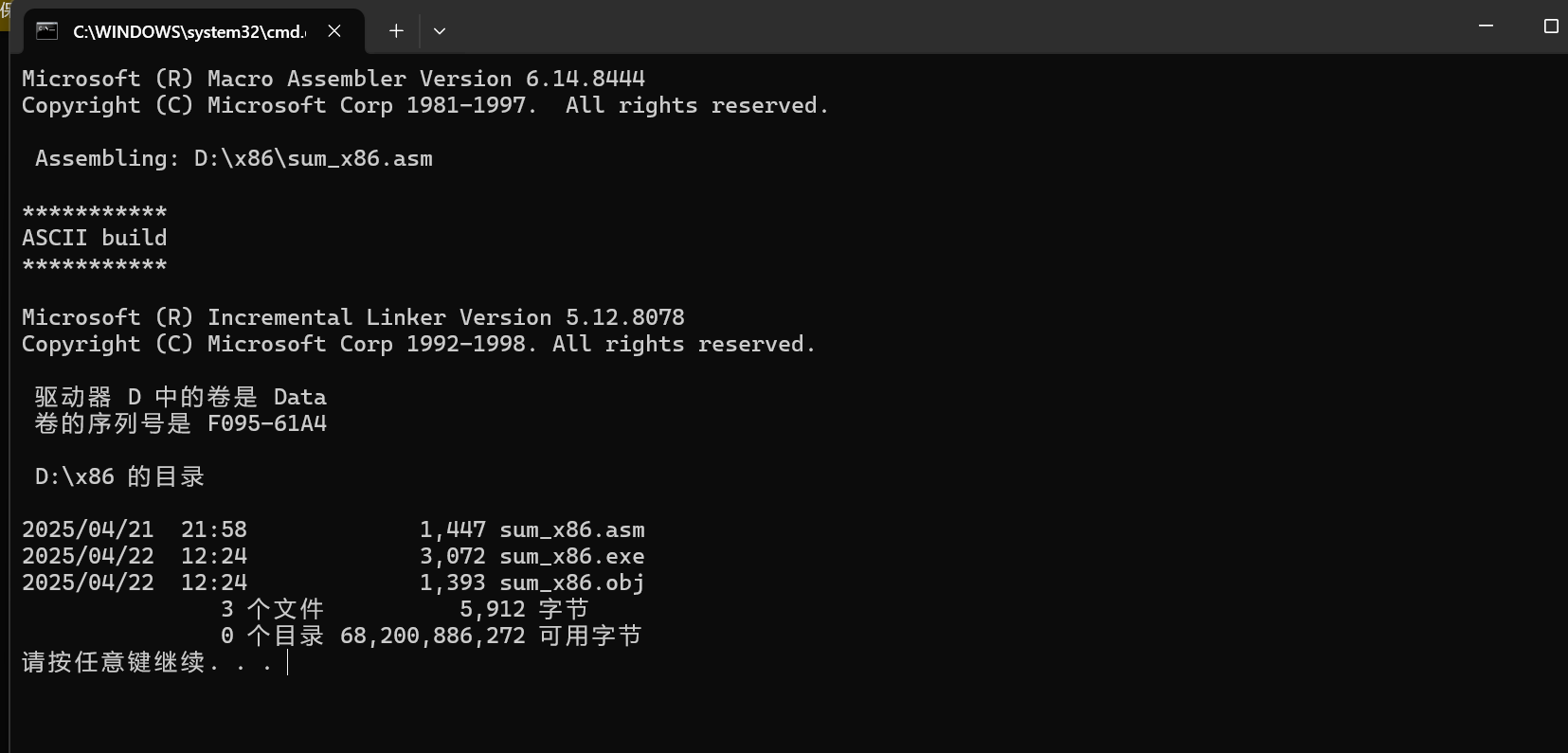
**其他的同理**

**Ⅲ：运行求累加和的Intel x86汇编语言程序**

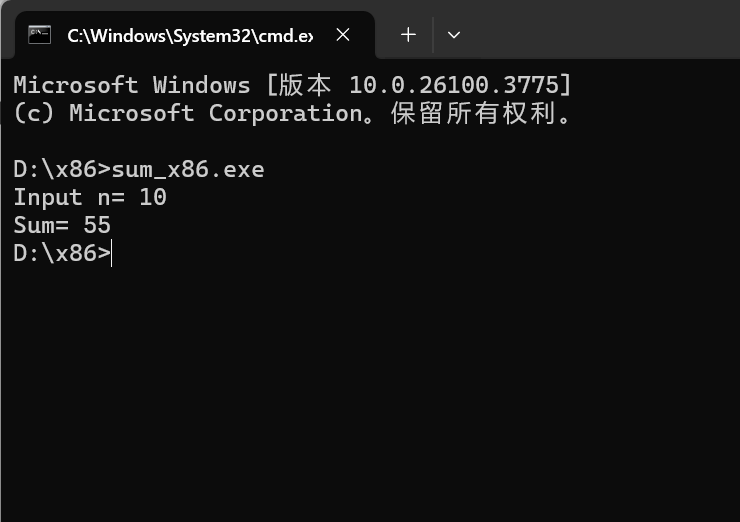
**第一步：在masm32汇编工具中打开源程序“sum\_x86.asm”**



**第二步：汇编**

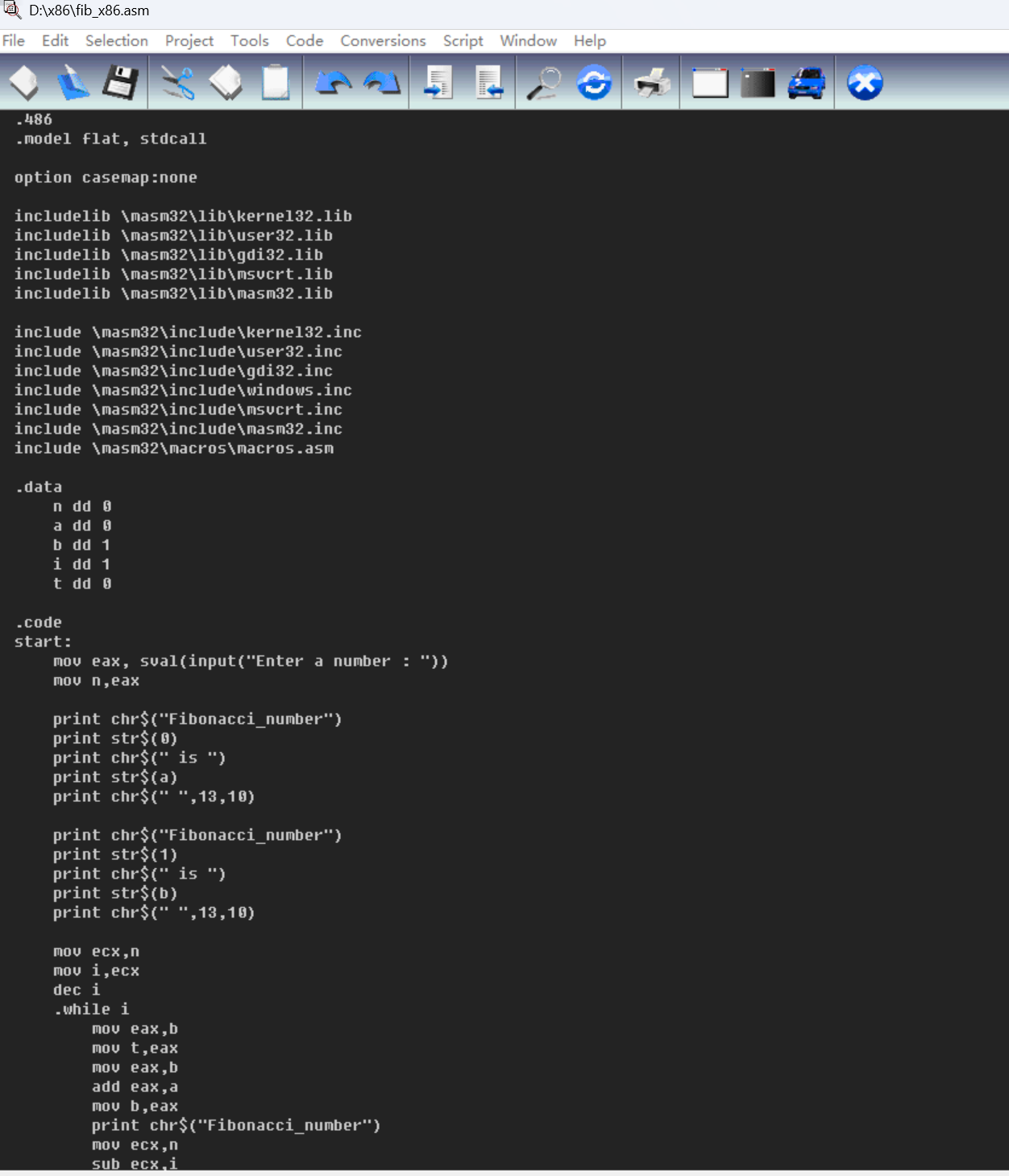


**第三步：运行“sum\_x86.exe”**

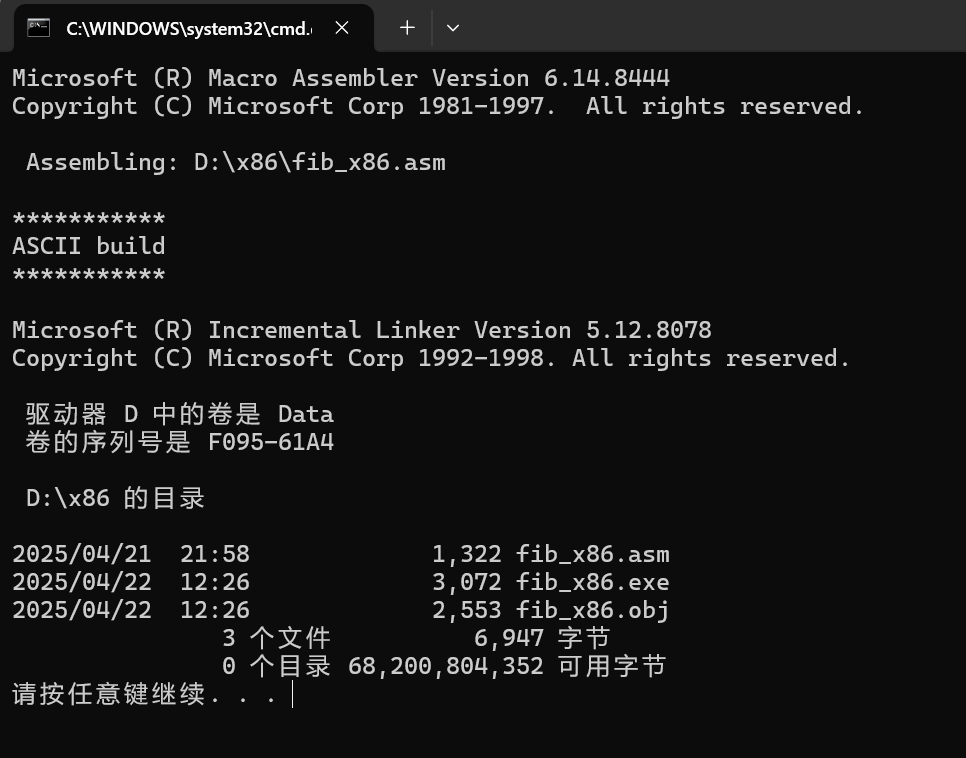


**Ⅳ：运行计算费波那契数列的Intel x86汇编语言程序**

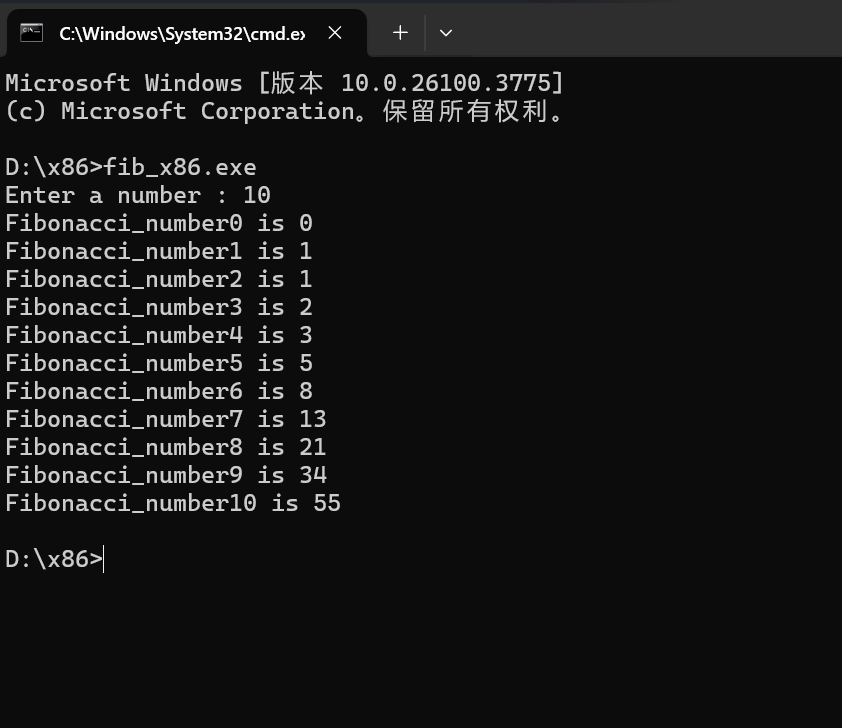
**第一步：在masm32汇编工具中打开源程序“fib\_x86.asm”**



**第二步：汇编**



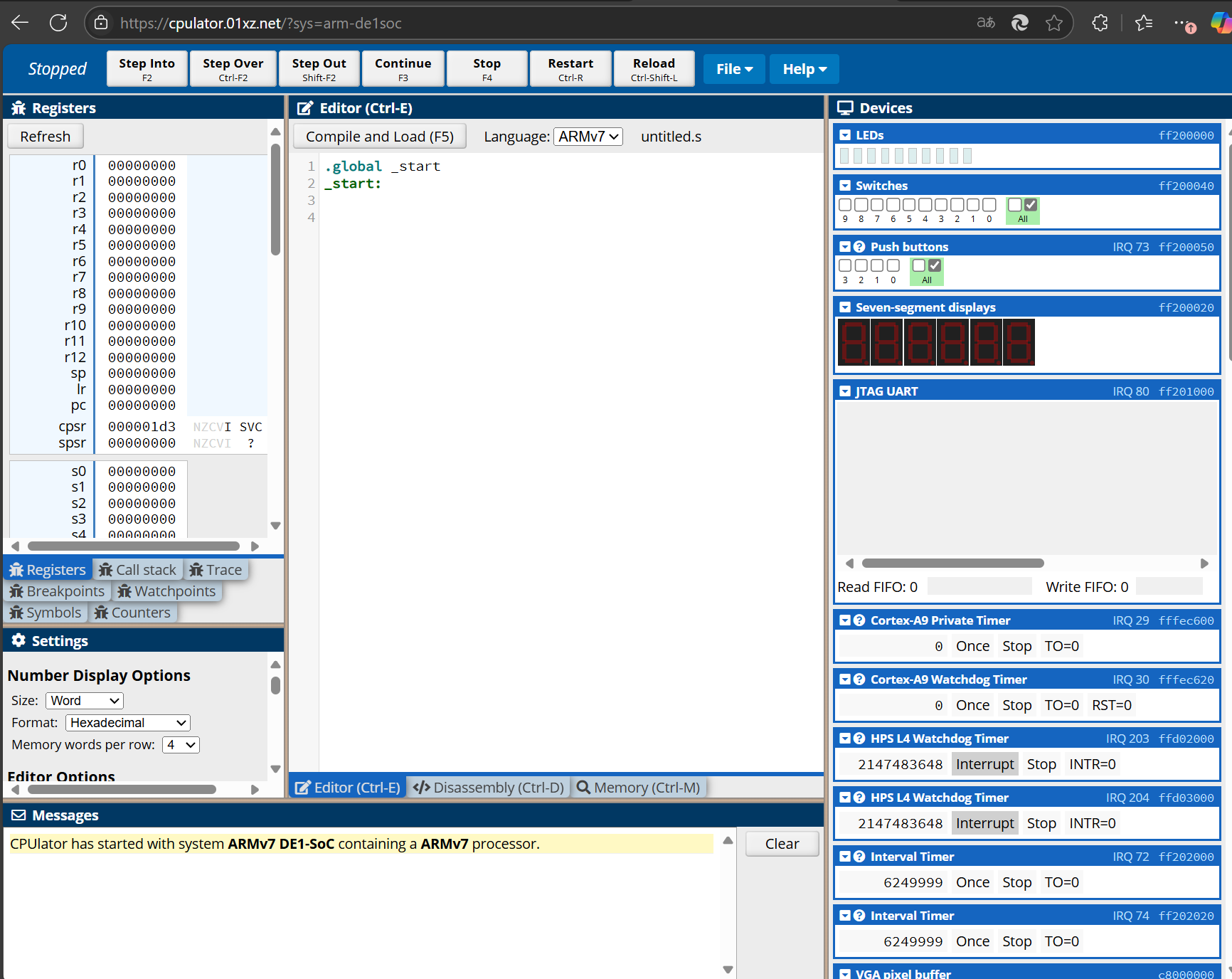
**第三步：运行“fib\_x86.exe”**



1. **ARMv7汇编语言程序的运行**

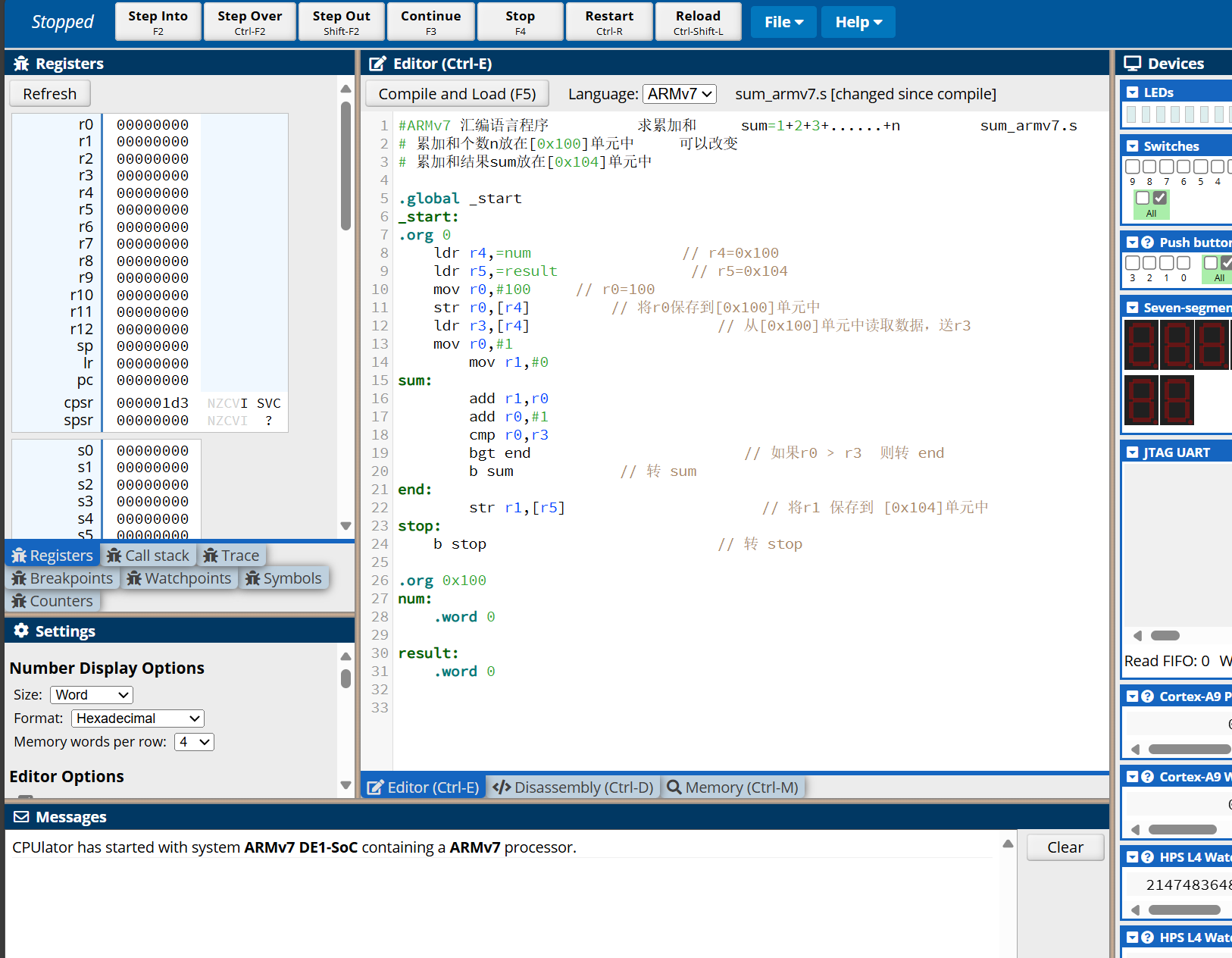
**Ⅰ：ARMv7汇编工具**

**在浏览器中打开ARMv7汇编工具**

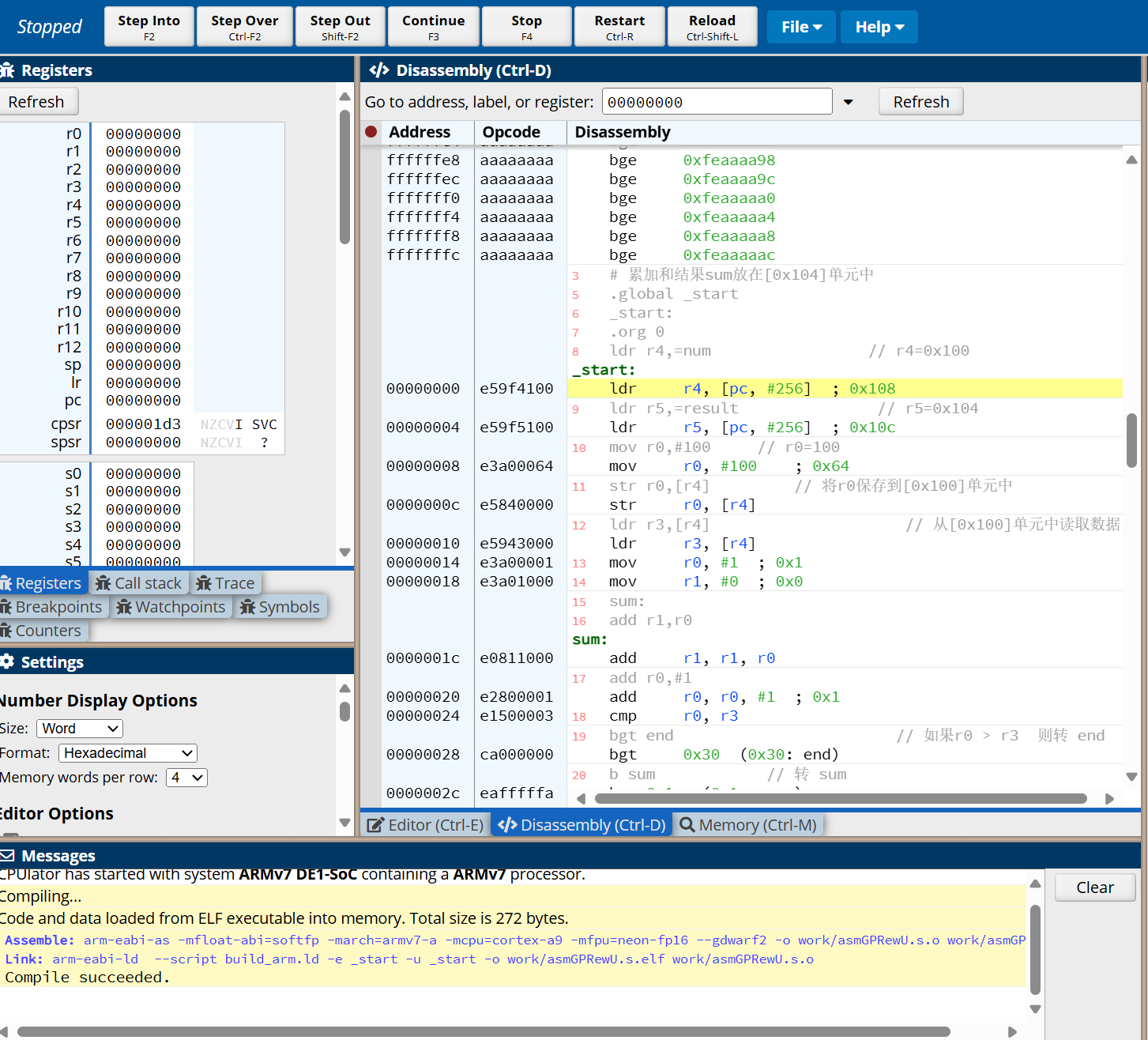


**Ⅱ：在ARMv7汇编工具中运行求累加和程序**

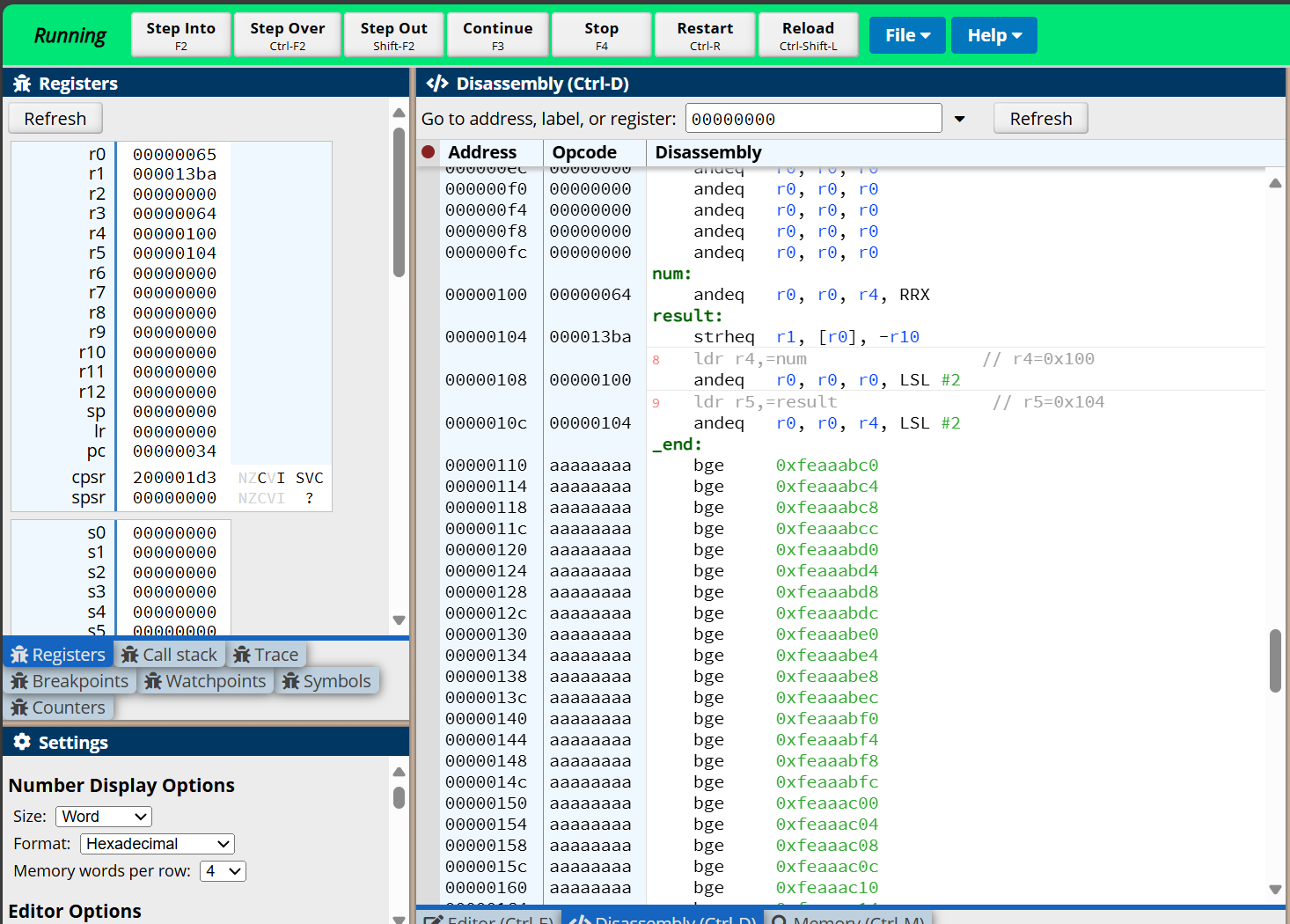
**第一步：打开源程序（sum\_armv7.s）**



**第二步：汇编**

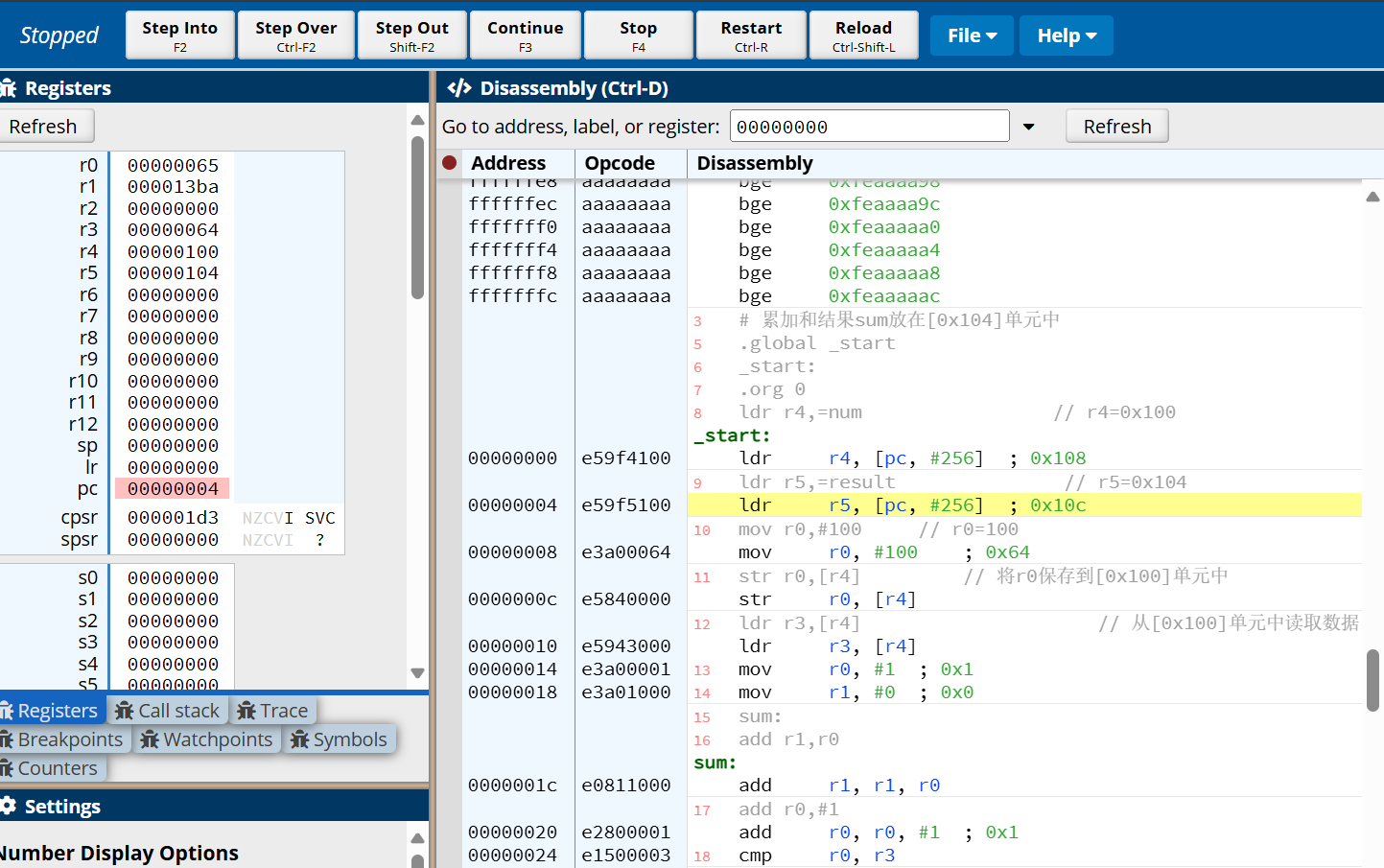


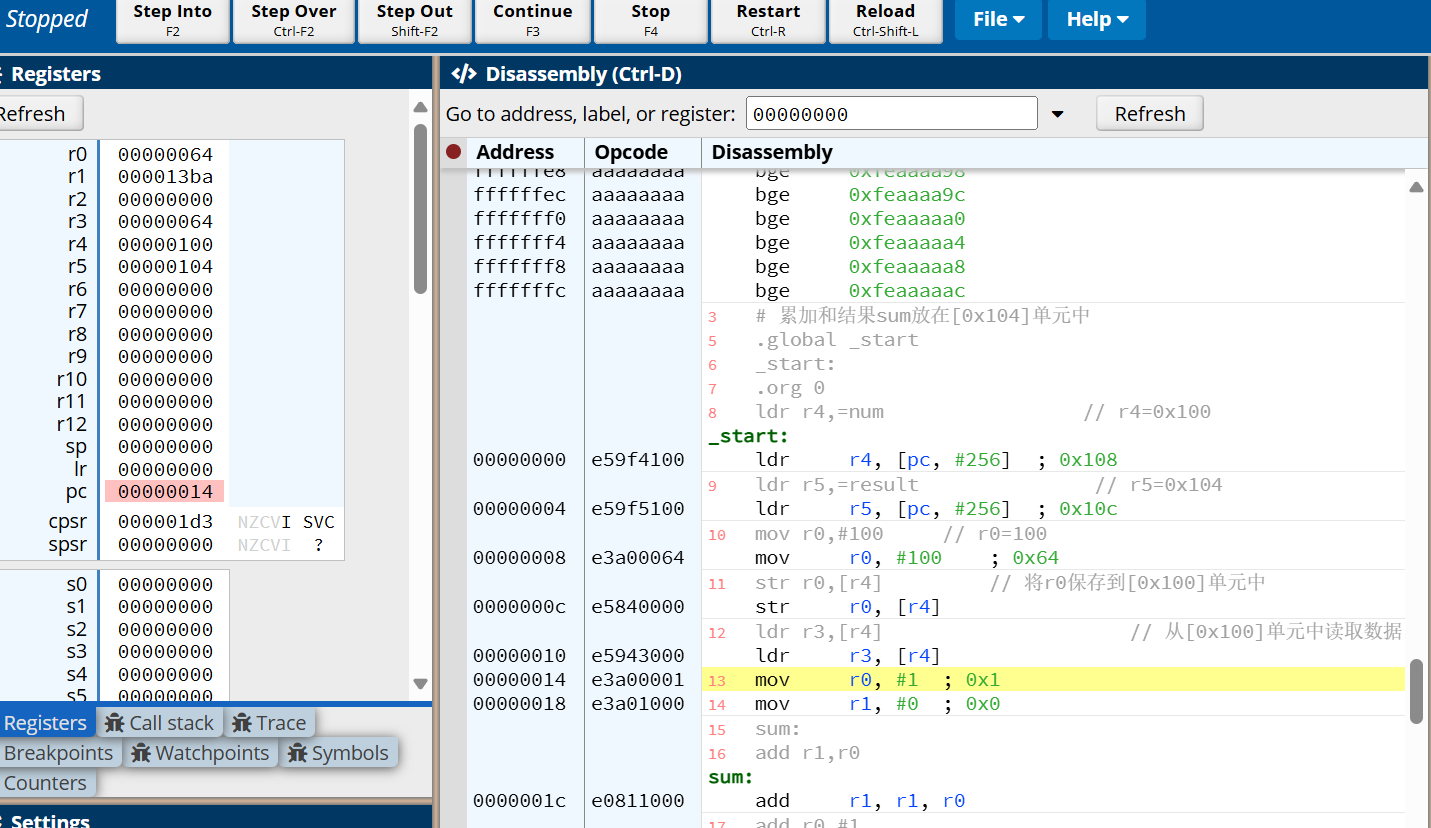
**第三步：连续执行程序**





第四步：单步执行程序



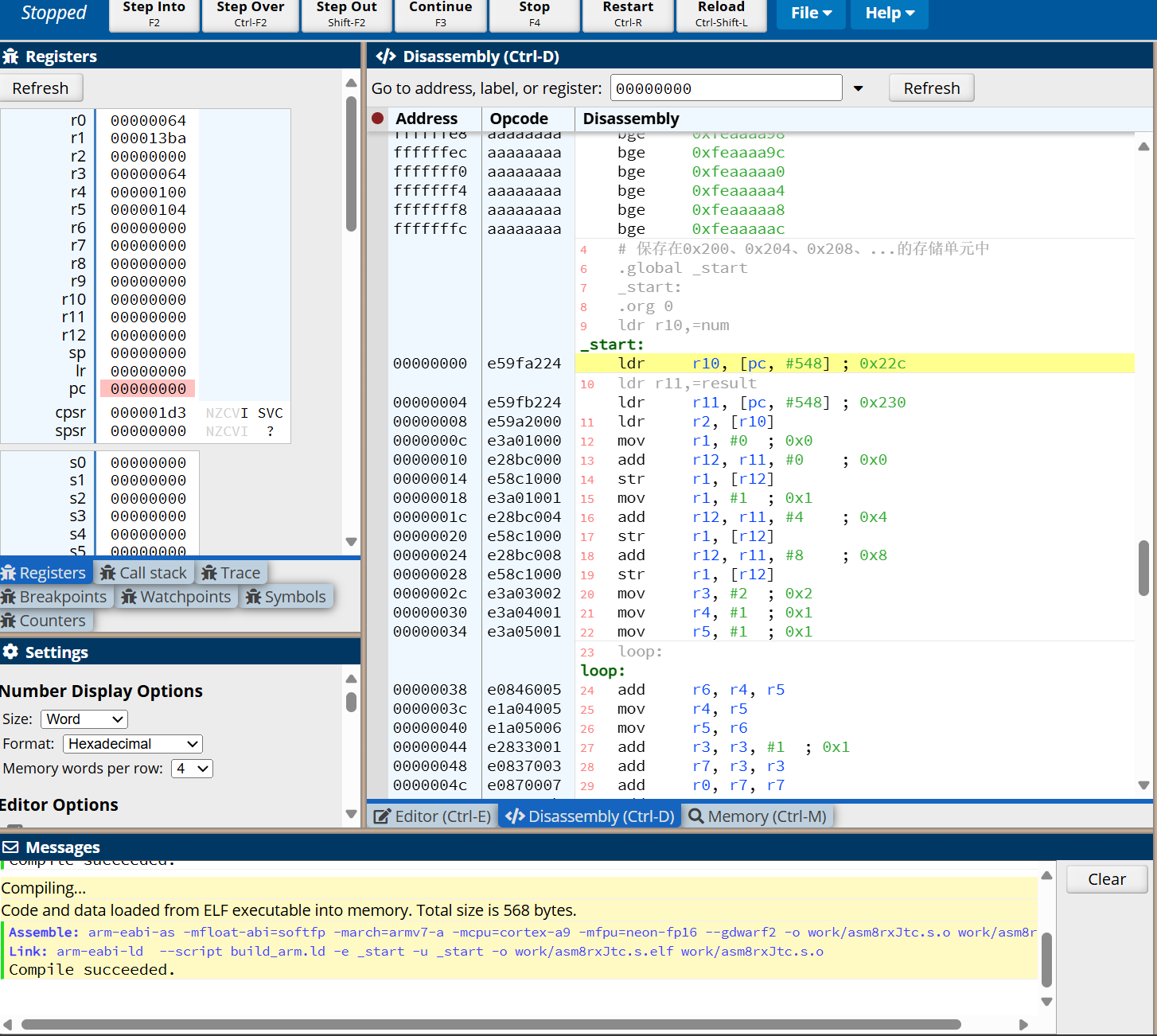


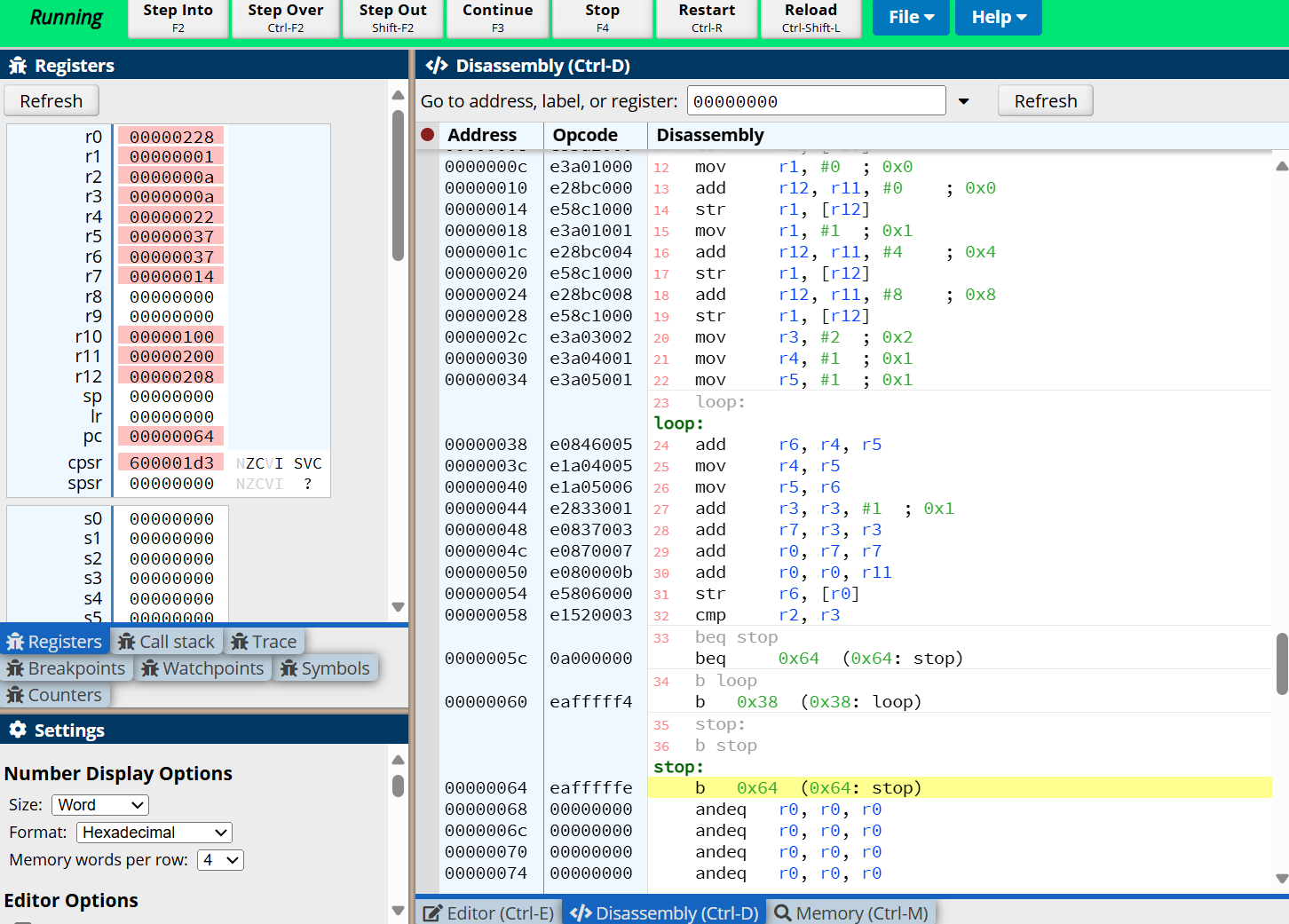
**Ⅲ：在ARMv7汇编工具中运行计算费波那契数列程序**

**第一步：打开源程序（fib\_armv7.s）**

**第二步：汇编**

**第三步：连续执行程序**



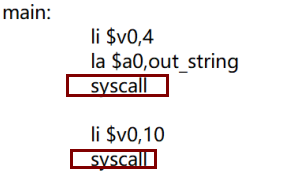


**程序执行结果：**



**回答问题：**

1. **Helloworld\_mips.asm源程序中，前后两个syscall语句的各自作用是什么？**

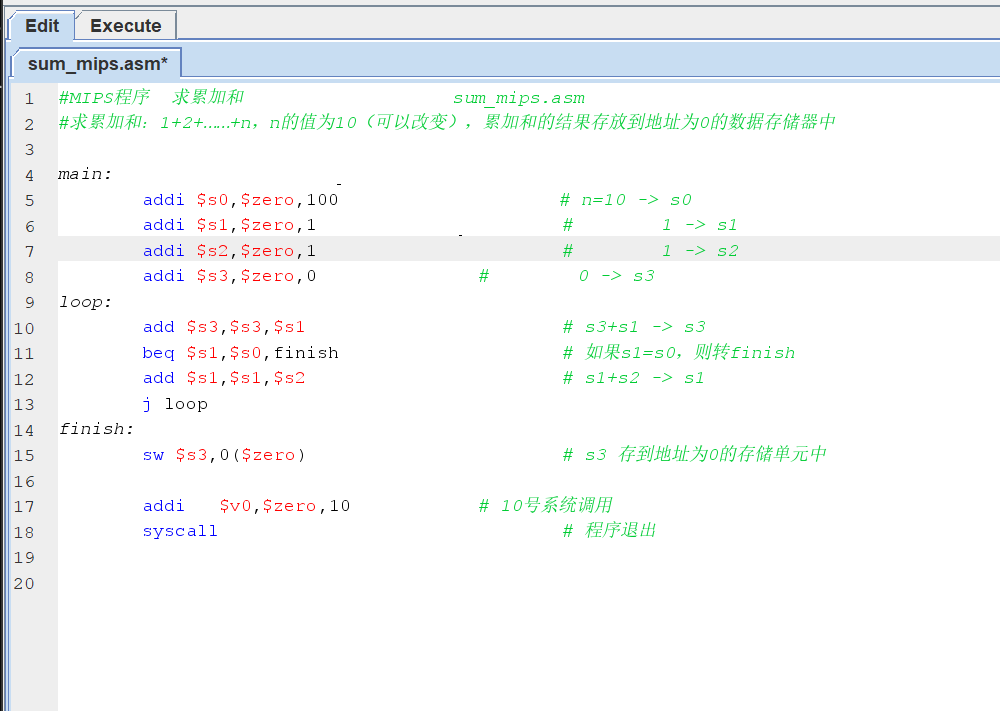


**答：第一个syscall前面的 li $v0, 4 设置系统调用号为 4，这是MIPS中用于显示字符串的系统调用。la $a0, out\_string 将待显示的字符串 out\_string（内容为 \nHello, World!\n）的地址加载到参数寄存器 $a0。此时执行 syscall，会触发系统调用，完成字符串的输出显示。**

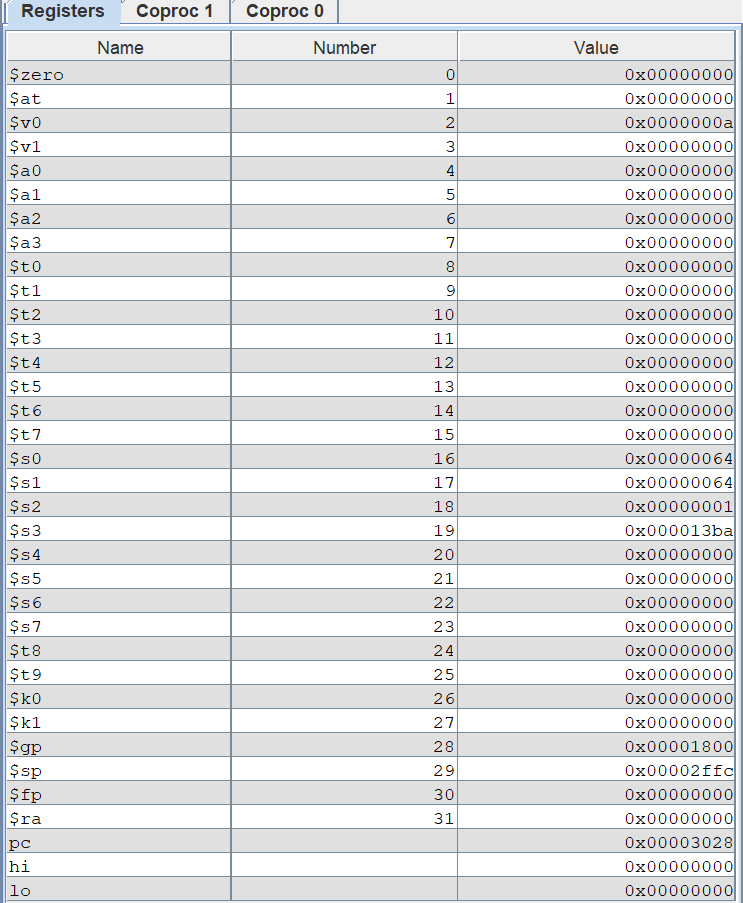
**第二个syscall，li $v0, 10 设置系统调用号为 10，在 MIPS 中，系统调用号 10 用于 程序终止。执行该 syscall 时，系统会结束当前程序的运行，释放资源并退出。**

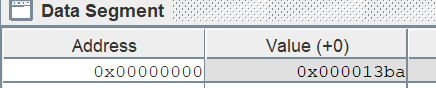
1. **修改sum\_mips.asm源代码,设n=100, 修改后程序命名为sum100\_mips.asm, 编译运行。要求展示所使用寄存器和内存的存储情况。若将其机器码导出，如何操作？ 导出文件命名为sum100\_mips.hex。**

**修改后代码：**

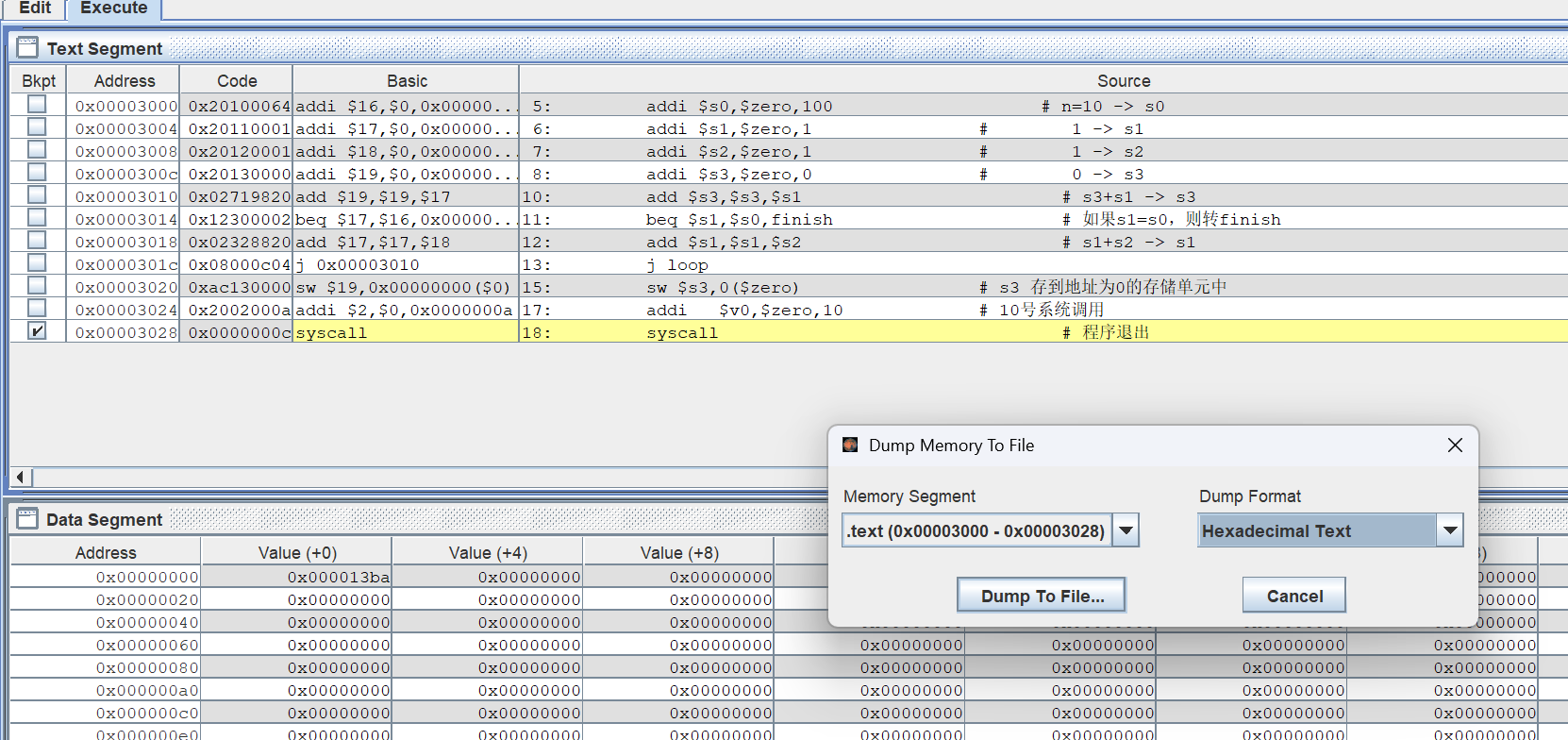


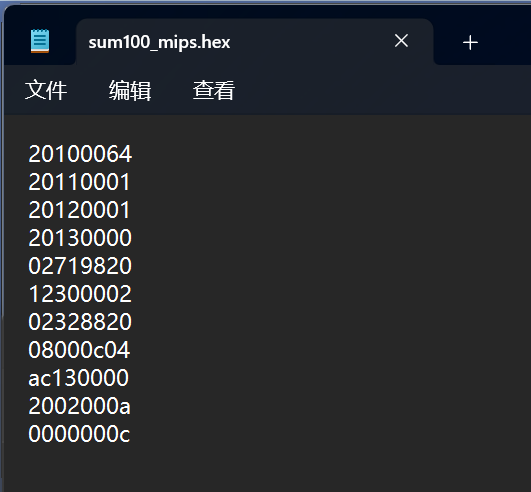
**运行结果如下：**



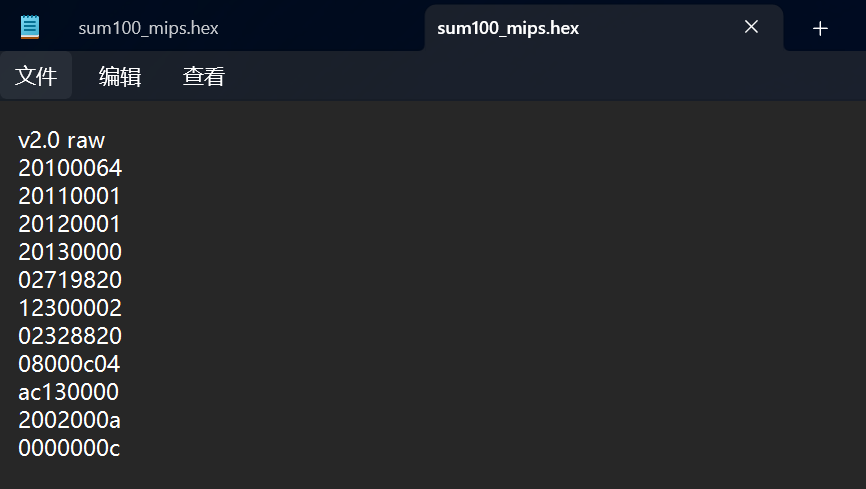


**结果导出：**

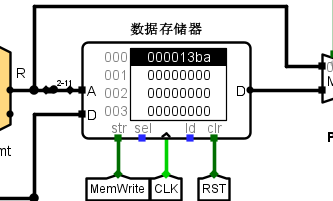
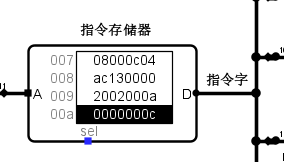




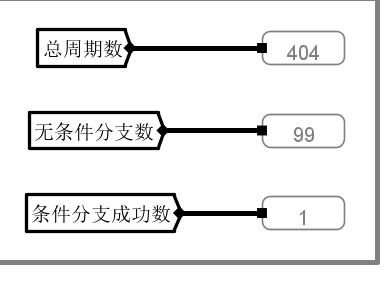
1. **在Logisim实现的单周期MIPS处理器上，运行计算1+2+3+……+100的机器语言程序（sum100\_mips.hex），观看数据存储器0号单元的值是不是13bah？程序运行的总周期数、无条件分支数、条件分支成功数分别是多少？**

**先给之前导出的sum100\_mips.hex文件添加v2.0raw头**

**然后导入指令寄存器，再运行程序，结果如下：**



**总周期数、无条件分支数、条件分支成功数分别是：**



1. **比较并分析sum\_riscv.asm与sum\_mips.asm两段代码在程序结束部分是如何处理的？**

**对于sum\_riscv.asm，用**

**finish:**

**sw a4, 0(zero) # 保存结果到内存地址0**

**end:**

**jal zero, end # 无限循环，程序挂起**

**这一段代码让程序无限循环从而挂起，并没有直接退出程序。**

**而sum\_mips.asm在结束部分**

**finish:**

**sw $s3, 0($zero) # 保存结果到内存地址0**

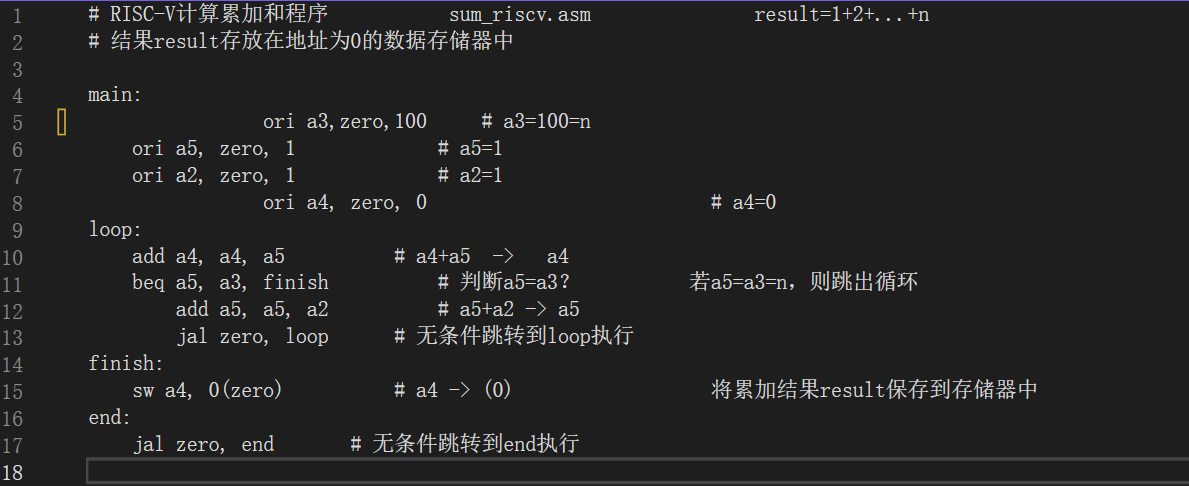
**addi $v0, $zero, 10 # 系统调用号10（退出程序）**

**syscall # 触发系统调用，终止程序**

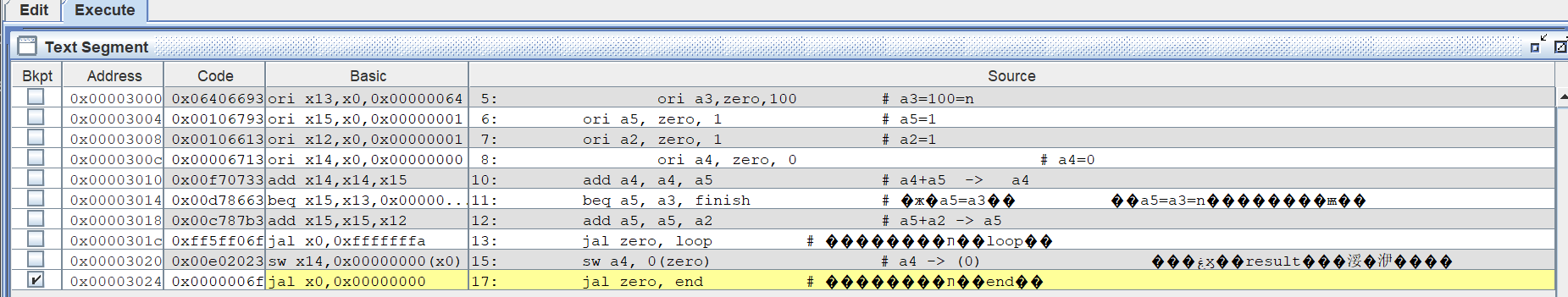
**通过调用10号程序直接退出程序，是正常退出**

1. **修改sum\_riscv.asm程序，计算1+2+3+……+100（修改后的程序命名为：sum100\_riscv.asm），然后在RARS 1.5汇编仿真器中运行，并将该程序的机器码保存到sum100\_riscv.hex文件中。**

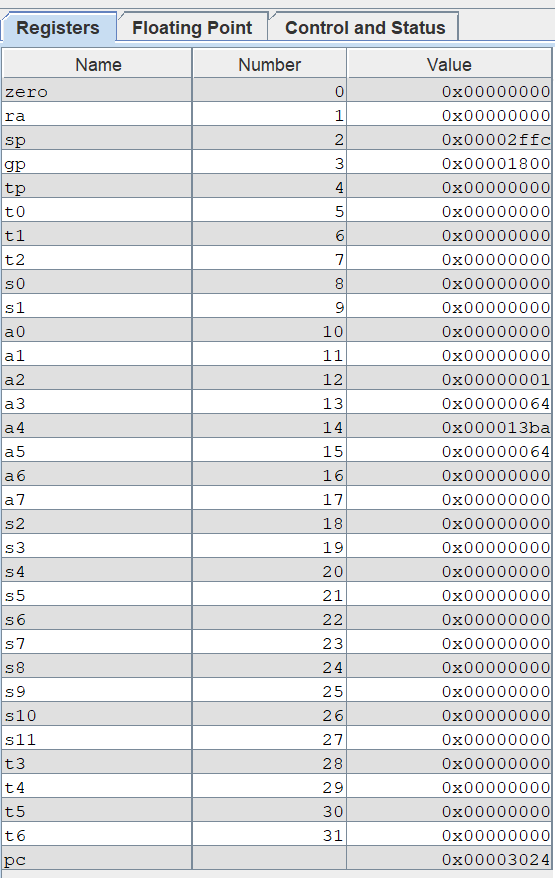
**程序修改如下：**



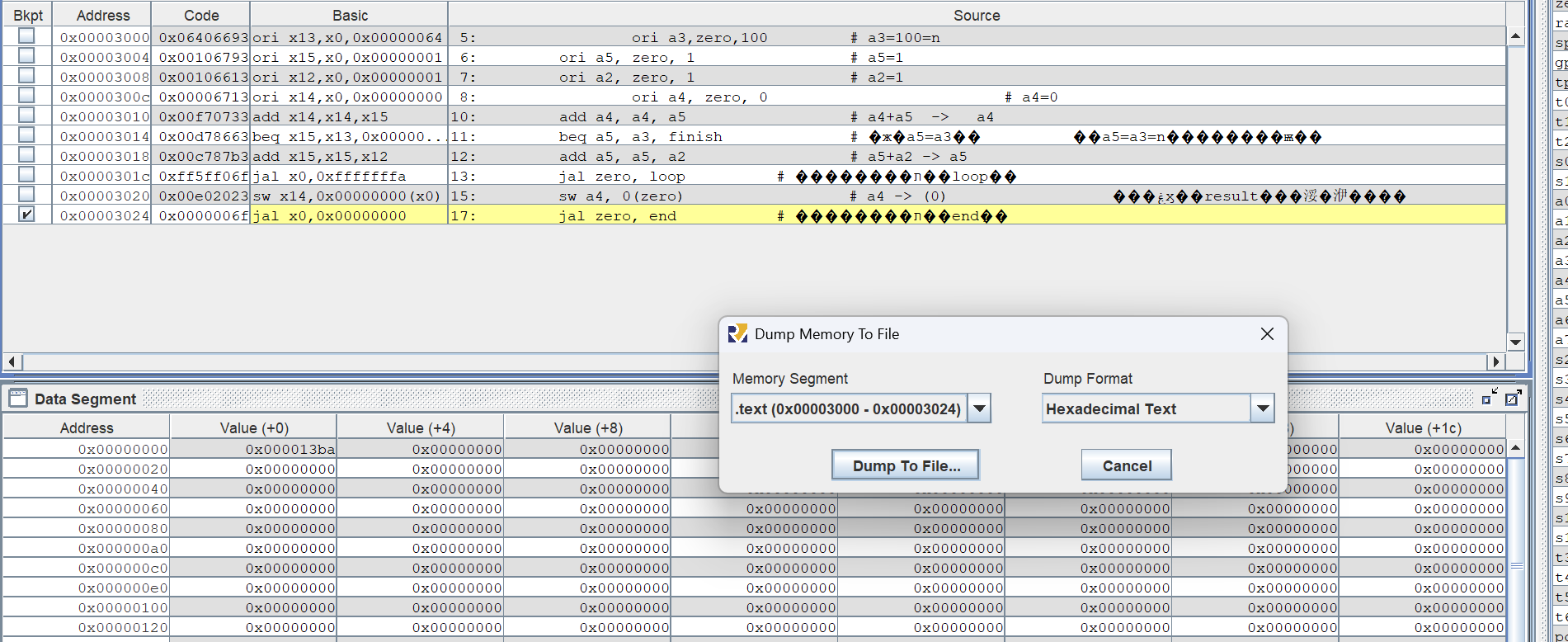
**运行结果如下：**

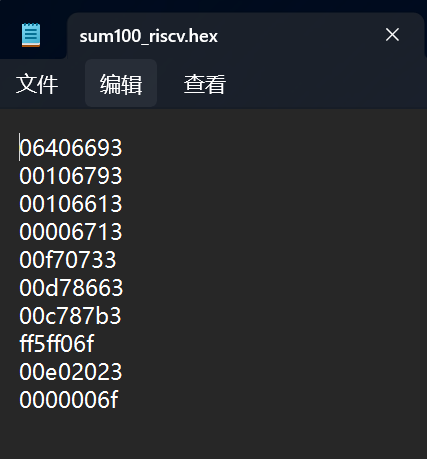






**保存结果：**





1. **针对在X86 ISA架构下，提供的5个helloworld.asm程序段，分析比较它们在语法、调用关系，展示等环节有何不同？**

**在语法上：**

|  |  |  |  |
| --- | --- | --- | --- |
| **程序** | **核心语法特点** | **数据段** | **典型语句** |
| **hello1** | **依赖 masm32 宏（print），极简语法，无显式 API 调用** | **无（宏内部处理字符串）** | **print "Hello World!"（单行完成输出）** |
| **hello2** | **启用 \_\_UNICODE\_\_，支持宽字符，使用 masm32rt.inc 控制台增强宏** | **无（字符串直接嵌入宏参数）** | **print "Hello World in UNICODE",13,10（含换行）** |
| **hello3** | **显式使用 chr$ 构造带 CR/LF 的字符串，冗余包含 gdi32.inc（未实际使用）** | **无（chr$ 生成临时字符序列）** | **print chr$("Hey...",13,10)（强制换行）** |
| **hello4** | **纯 Windows API 调用，invoke 语法显式传参，严格区分数据段与代码段** | **.data 定义标题和文本字符串** | invoke MessageBox, NULL, offset szText, ... |
| **hello5** | **兼容 C 语言语法，通过 crt\_printf 调用 C 库函数，数据段声明与 C 风格一致** | **.data 定义以 0 结尾的字符串** | **invoke crt\_printf, addr szText（类 C 输出）** |

**调用关系上：**

|  |  |  |  |
| --- | --- | --- | --- |
| **程序** | **调用层级** | **核心依赖** | **系统调用本质** |
| **hello1** | **宏 → WriteConsoleA（控制台 API）** | **masm32\macros\macros.asm 的 print 宏** | **直接调用 kernel32!WriteConsoleA** |
| **hello2** | **宏 → WriteConsoleW（UNICODE 控制台）** | **masm32rt.inc 的控制台封装函数** | **通过 masm32rt!output 间接调用 API** |
| **hello3** | **同 hello1（仅字符串格式不同）** | **同 hello1** | **同 hello1** |
| **hello4** | **user32!MessageBoxA → 内核渲染窗口** | **直接依赖 user32.dll 和 kernel32.dll** | **触发 GUI 线程，调用 NtUserMessageBox** |
| **hello5** | **C 库 → msvcrt!printf → WriteFile** | **msvcrt.lib 的 C 运行时函数** | **通过 C 库缓冲，最终调用 kernel32!WriteFile** |

**展示上：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **程序** | **输出类型** | **可视化形式** | **交互性** | **字符集** |
| **hello1** | **控制台文本** | **黑色控制台窗口，单行文本** | **无（直接退出）** | **ANSI** |
| **hello2** | **控制台文本** | **绿色控制台窗口，支持宽字符（如中文）** | **需按键退出（inkey）** | **UNICODE** |
| **hello3** | **控制台文本** | **黑色控制台窗口，文本后自动换行** | **无** | **ANSI** |
| **hello4** | GUI 弹窗 | **模态对话框（标题 + 文本 + 确定按钮）** | **必须点击 “确定” 关闭** | **ANSI** |
| **hello5** | **控制台文本** | **黑色控制台窗口，输出与 C 语言一致** | **无** | **ANSI** |

1. **对于sum\_x86.asm源程序，其中print语句的功能是怎样的？如果在8086ISA架构下，其可替换为什么语句？请在编译器中验证，并给出相应的截屏佐证。**

**Print语句的功能：print 是 MASM32 宏库提供的快捷指令，用于在控制台输出字符串或数值。**

**print chr$("Sum= ")：输出固定字符串 Sum= （chr$ 用于构造带格式的 ASCII 字符串）。**

**print str$(sum)：将内存单元 sum 中的数值（如累加和）转换为十进制字符串并输出。**

**在8086ISA架构下，其可替换为以下两种语句：**

**.data**

**msg db 'Sum= ', '$' ; 定义以 "$" 结尾的字符串**

**.code**

**mov dx, offset msg ; 字符串偏移地址送 DX**

**mov ah, 09h ; DOS 中断功能号（输出字符串）**

**int 21h ; 触发中断**

**替代print chr$("Sum= ")**

**.data**

**sum dw 5050 ; 假设累加和为 5050（16 位）**

**.code**

**mov ax, sum ; 数值送 AX**

**mov cx, 0 ; 计数器清零**

**; 分解数值为 ASCII 字符（逆向存储，如 5050 → '0','5','0','5'）**

**num\_to\_str:**

**mov dx, 0 ; 清 DX**

**mov bx, 10 ; 除数 10**

**div bx ; AX ÷ BX，商在 AL，余数在 AH**

**add ah, '0' ; 余数转 ASCII 字符**

**push ax ; 保存商（下次循环用）和字符（AH）**

**inc cx ; 字符计数**

**test al, al ; 商为 0 则结束**

**jnz num\_to\_str**

**; 输出字符（从高位到低位，需弹出栈）**

**print\_num:**

**pop dx ; 取出字符（在 AH 中，AX 已被压栈）**

**mov ah, 02h ; DOS 中断功能号（输出单个字符）**

**int 21h ; 触发中断**

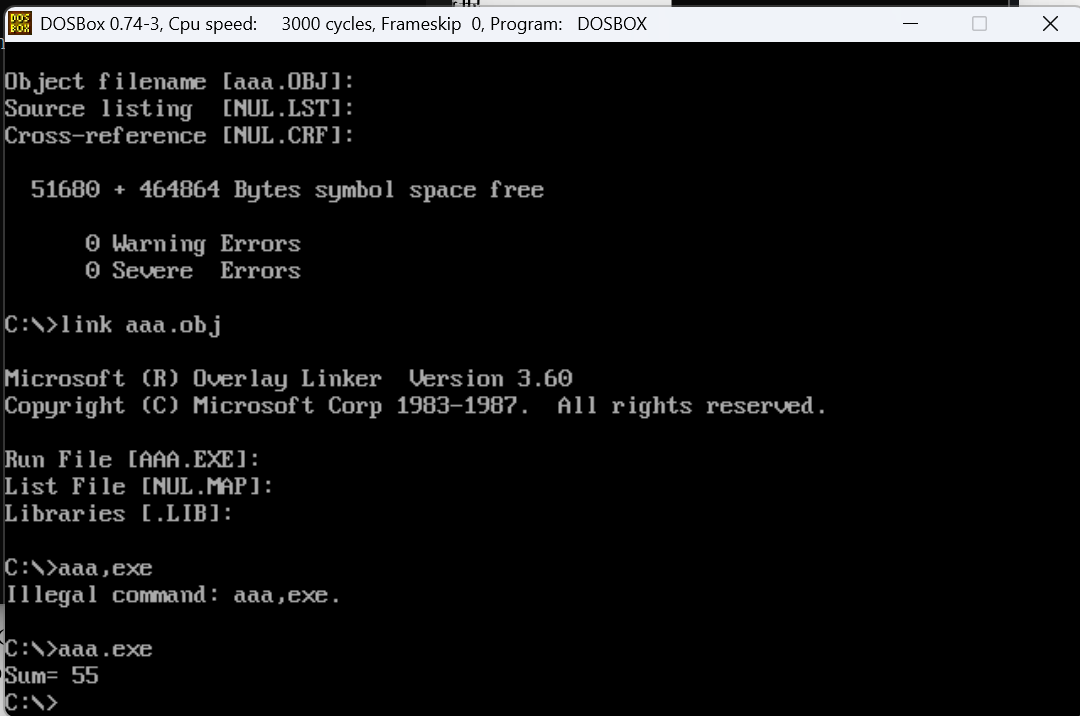
**loop print\_num**

**替代 print str$(sum)**

**完整代码如下：**



**当n=10时，在8086的运行结果如下：**



1. **比较下fib\_armv7.s与fib\_riscv.asm、fib\_mips.asm是如何完成输入数据和结果的存储的？**

**输入：n**

**输出：0、1、1、2、3、5、8、13、21、34、…保存在地址为0、4、8、12、...的数据存储器中**

**答：**

**输入数据存储：**

**fib\_armv7.s：输入数据 n 预先存储在内存地址 0x100 处（org 0x100 num:.word 10），通过 ldr r2,[r10] 指令将其加载到寄存器 r2，其中 r10 指向 num 的地址（ldr r10,=num）。这种方式通过内存读取输入值。**

**fib\_riscv.asm 与 fib\_mips.asm：直接在代码中通过指令设置输入数据 n。如 fib\_riscv.asm 中 ori a2, zero,10 将 a2 寄存器设为 10；fib\_mips.asm 中 addi $s2,$zero,10 将 $s2 寄存器设为 10。二者均未从内存特定地址读取，而是在寄存器中直接定义输入值。**

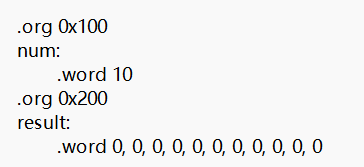
**结果存储：**

**fib\_armv7.s：结果存储在以 result（0x200）为起始地址的内存中。通过 str 指令存储，如 str r1,[r12]。地址计算通过 add 指令（如 add r0,r0,r11，r11 指向 result），将结果依次存入 0x200、0x204 等连续地址。**

**fib\_riscv.asm：用 sw 指令存储，如 sw a1,0(zero) 将数据存到地址 0，sw a6,0(a0)。地址计算通过 add a0,a7,a7 等指令（a7 = a3 \* 2，a3 是循环计数），实现 0、4、8 等间隔为 4 的连续地址存储。**

**fib\_mips.asm：同样用 sw 指令（如 sw $s1,0($zero) 存到地址 0，sw $s6,0($s0)）。地址计算通过 add $s0,$s7,$s7（$s7 = $s3 + $s3，$s3 是循环计数），将结果存到 0、4、8 等连续地址。**

1. **fib\_armv7.s中为何有数据段.word，**



**是否能改成与其他两套指令代码一样的形式？**

**答：.word 是 ARM 汇编中的伪指令，用于在内存中分配 字（4 字节） 大小的存储单元，并可对其进行初始化。**

**org 0x100 num:.word 10：**

**.org 0x100 表示将 num 变量的存储地址定位到 0x100。**

**.word 10 在此地址分配一个字的空间，并初始化为 10，作为输入的 n 值（斐波那契数列的项数）。**

**org 0x200 result:.word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0：**

**.org 0x200 将 result 数组的起始地址定位到 0x200。**

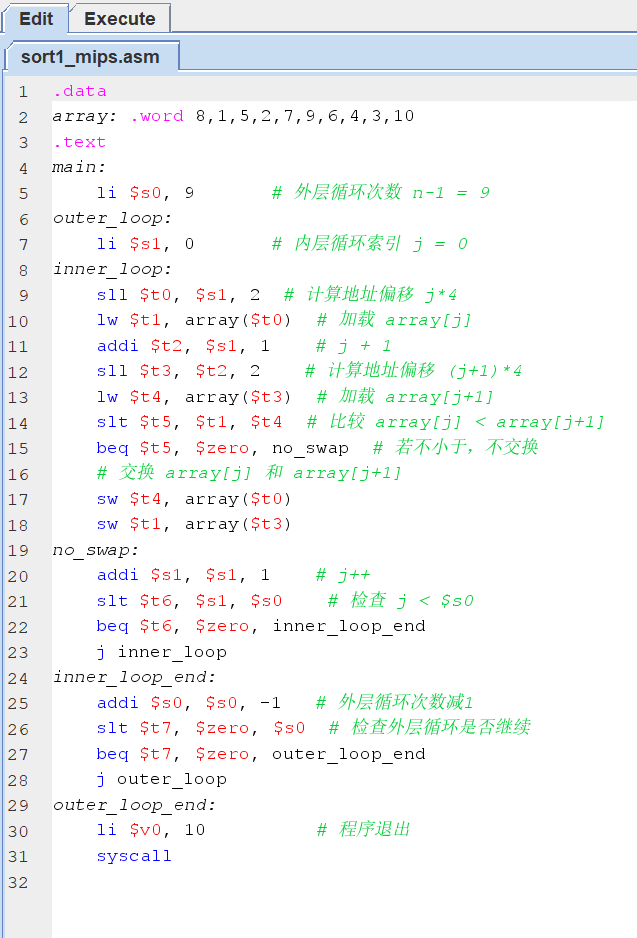
**.word 0, 0, … 分配 11 个字的空间并初始化为 0，用于存储斐波那契数列的计算结果（共 n + 1 项，n = 10 时共 11 项）。**

**能改成与其他两套指令代码一样的形式**

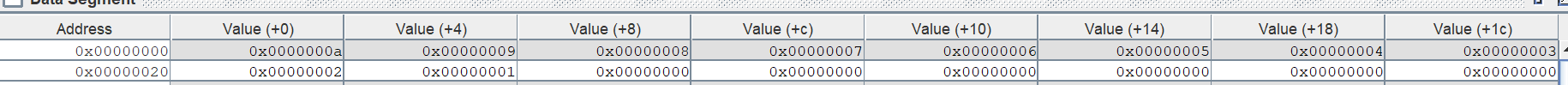
* 1. **设计实验**

1. **MIPS汇编语言程序设计（排序程序）——文件名：sort1\_mips.asm（降序排序）、sort2\_mips.asm（升序排序）。**

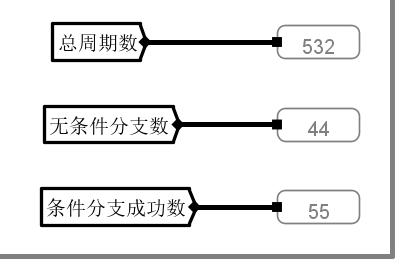
**sort1\_mips.asm（降序排序）代码设计如下：**



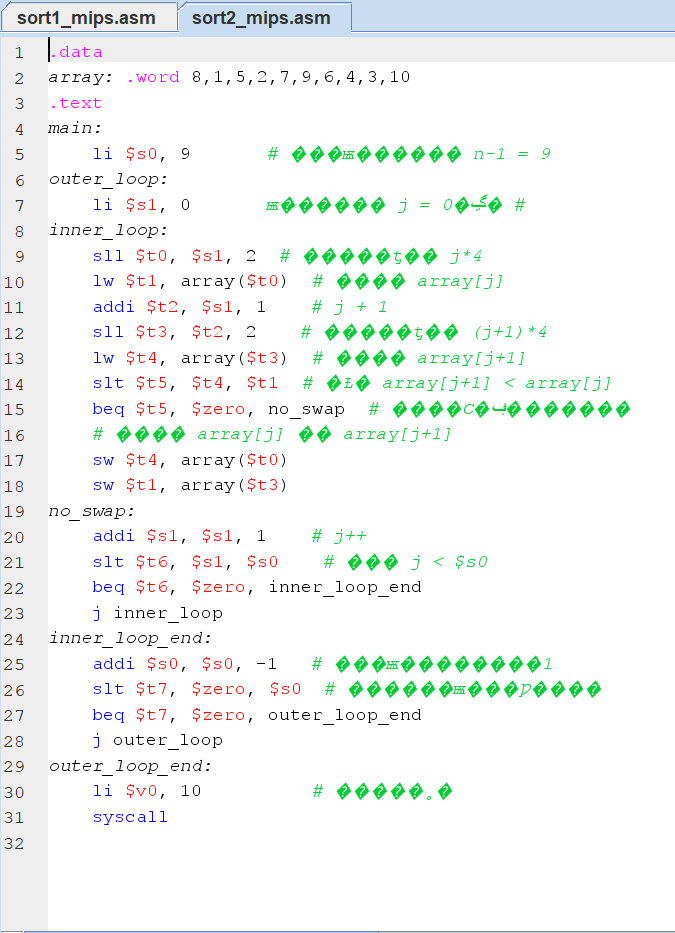
**运行结果如下：**



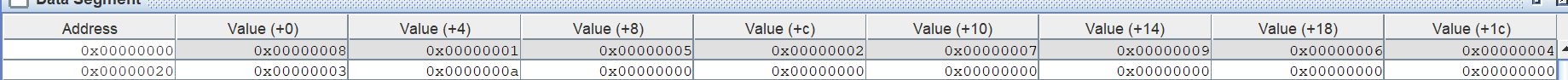
**导出为hex文件并在logisim上验证结果如下：**



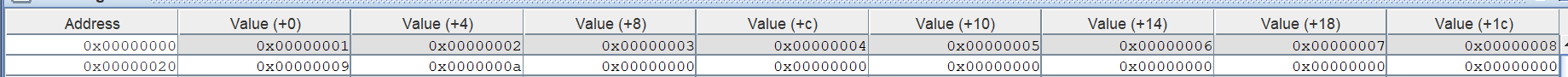
**sort2\_mips.asm（升序排序）代码如下：**



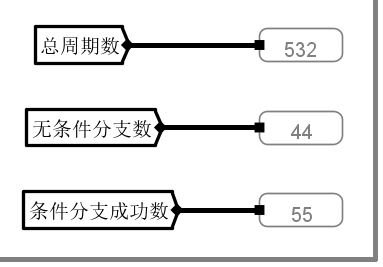
**运行前：**



**运行后：**

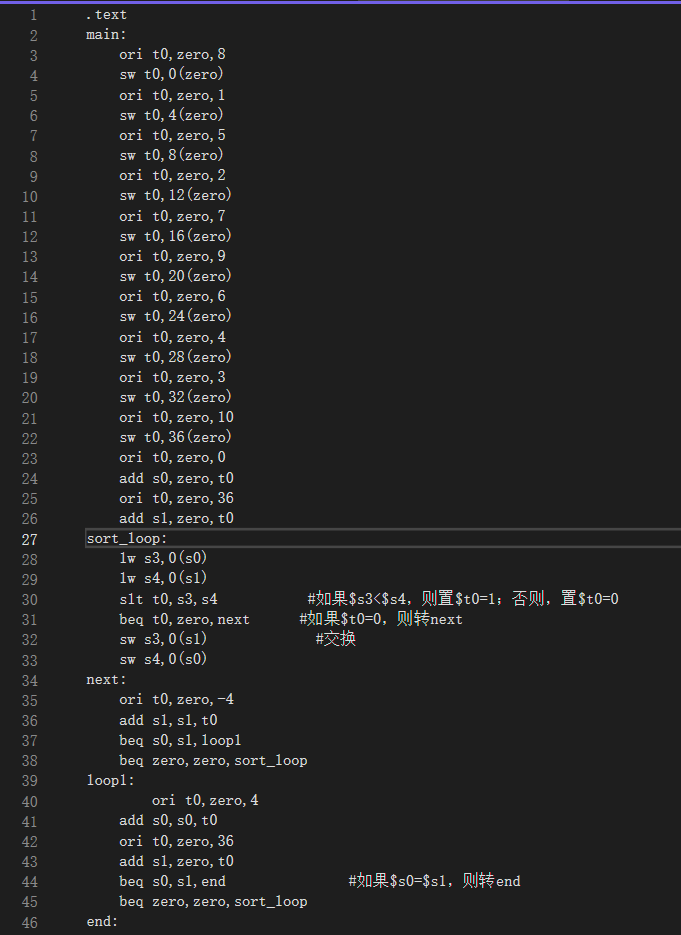


**Logisim验证如下：**

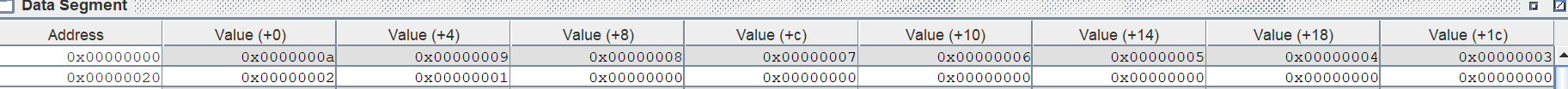


1. **RISC-V汇编语言程序设计（排序程序）——文件名：sort1\_riscv.asm（降序排序）、sort2\_riscv.asm（升序排序）。**

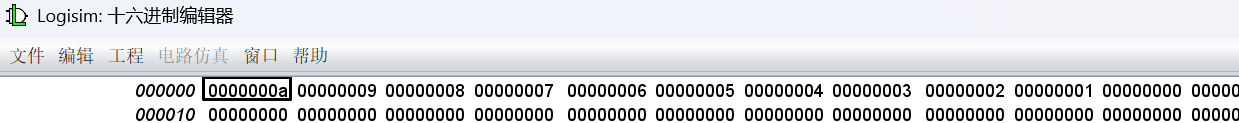
**Sort1\_riscv.asm（降序排序）代码如下：**



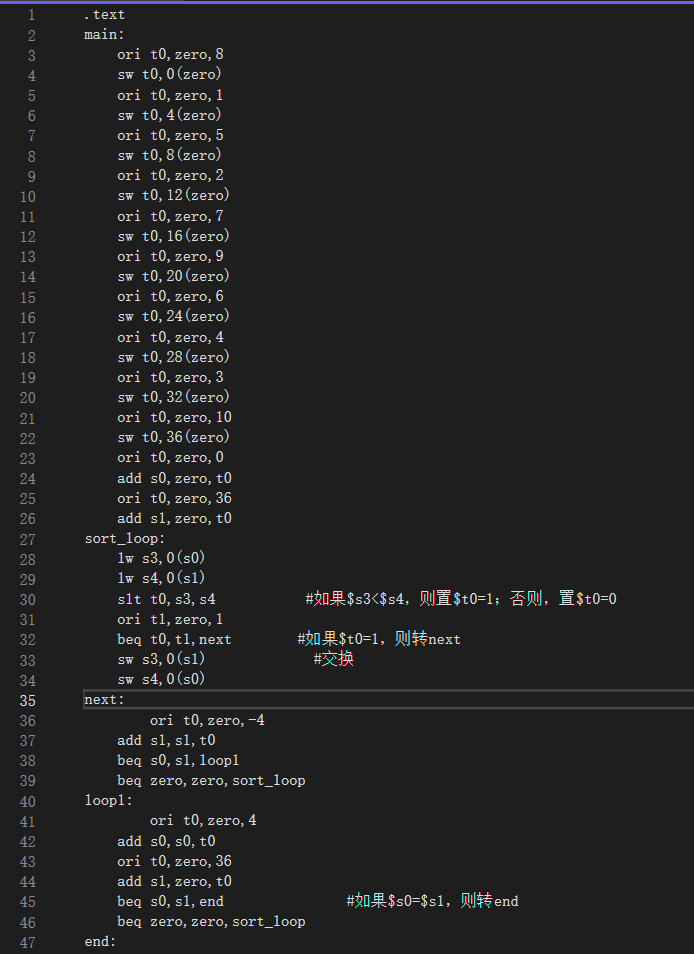
**运行结果如下：**



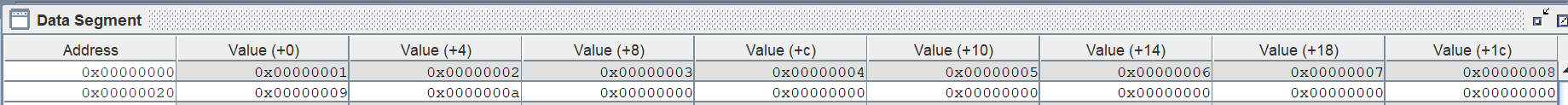
**Logisim验证结果如下：**



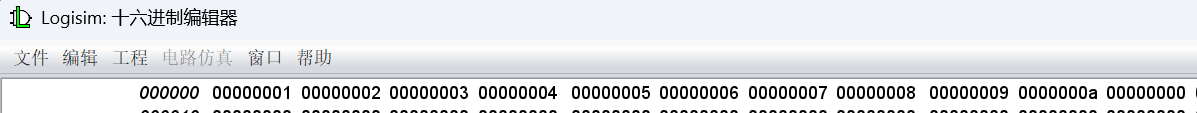
**Sort2\_riscv.asm（升序排序）代码如下：**



**验证结果如下：**



**Logisim验证如下：**

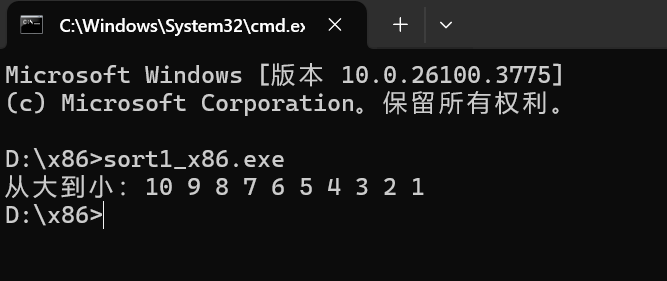


1. **Intel x86汇编语言程序设计（排序程序）——文件名：sort1\_x86.asm（降序排序）、sort2\_x86.asm（升序排序）。**

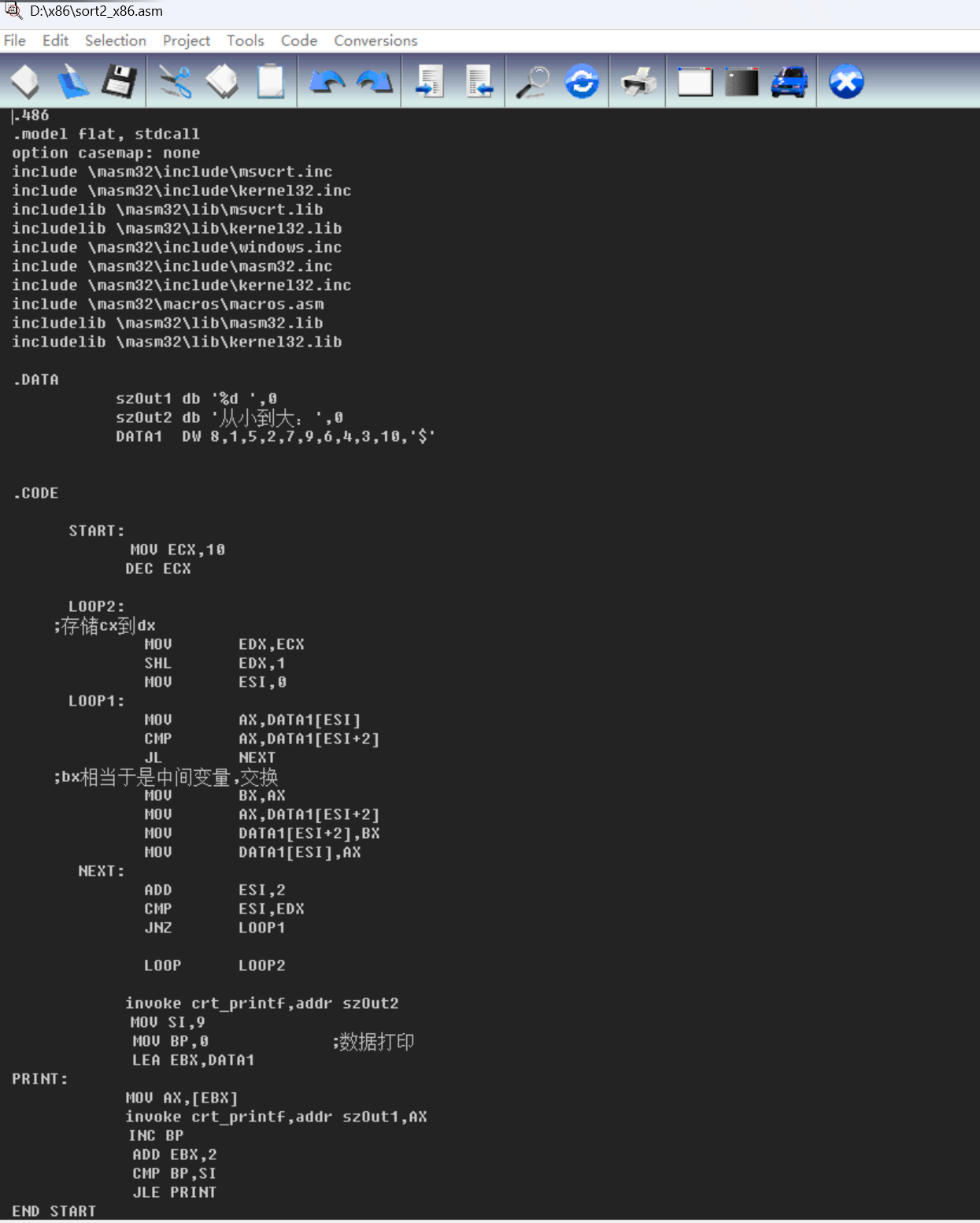
**sort1\_x86.asm（降序排序）代码如下：**



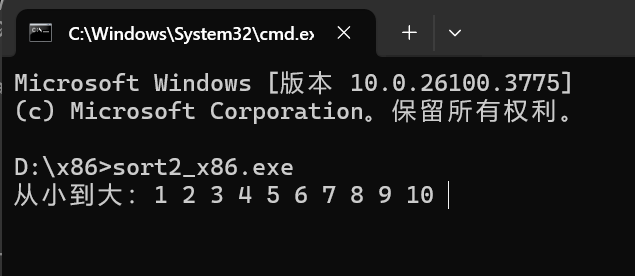
**运行结果如下：**



**sort2\_x86.asm（升序排序）代码如下：**



**运行结果如下：**



* 1. **挑战性实验**

**ARMv7汇编语言程序设计（排序程序）——文件名：sort1\_armv7.s（降序排序）、sort2\_armv7.s（升序排序）。**

1. **实验报告提交**
2. **实验报告命名为：学号+姓名+第4次实验报告.pdf。**
3. **将实验报告和设计文件打包为1个压缩文件，压缩文件命名为：学号+姓名+第4次实验.zip。**
4. **将压缩文件上传到数字化教学平台上，第4次实验报告提交截止时间（2周内）：2025年4月23日晚上24点。**