**计算机体系结构实验 第五周实验报告**

组员：许诗瑶20023105、刘朝润20023114、刘晓航20020070

**目录**

[一、实验要求 2](#_Toc77153197)

[（一）ECALL和SRET功能实现 2](#_Toc77153198)

[（二）Trap实现 2](#_Toc77153199)

[（三）printf程序测试 2](#_Toc77153200)

[二、实验环境 2](#_Toc77153201)

[三、实验内容 2](#_Toc77153202)

[（一）ECALL和SRET功能实现 2](#_Toc77153203)

[1、相关CSR寄存器 2](#_Toc77153204)

[2、ECALL指令 4](#_Toc77153205)

[3、SRET指令 6](#_Toc77153206)

[（二）Trap实现 6](#_Toc77153207)

[1、保护现场 10](#_Toc77153208)

[2、处理程序 11](#_Toc77153209)

[3、恢复现场 11](#_Toc77153210)

[四、实验结果 12](#_Toc77153211)

[（一）ELF可执行文件的转换 12](#_Toc77153212)

[（二）printf功能C程序测试 13](#_Toc77153213)

[五、实验总结 16](#_Toc77153214)

[（一）实验分工 16](#_Toc77153215)

[（二）实验遇到的问题 16](#_Toc77153216)

[（三）附件文件说明 17](#_Toc77153217)

# 一、实验要求

## （一）ECALL和SRET功能实现

带有printf的C语言程序汇编为RISC-V指令后，需要有ECALL触发中断，进入中断处理流程，处理中断后通过SRET返回。ECALL和SRET需要处理CSR相关寄存器和PC的跳转等。

## （二）Trap实现

ECALL跳转至中断处理程序入口，中断处理程序需要分别进行：保留现场、中断处理、恢复现场等工作。

## （三）printf程序测试

使用带printf的测试程序，在5级流水线上测试，主要测试ECALL、中断处理、SRET的功能实现。

# 二、实验环境

编程语言：Verilog

IDE：Vivado 2018.3

编译工具链：riscv-gnu-tool

工程版本控制及代码托管：Github平台

RISC-V ISA模拟器：Spike  
**辅助转换工具：elf2hex**

# 实验内容

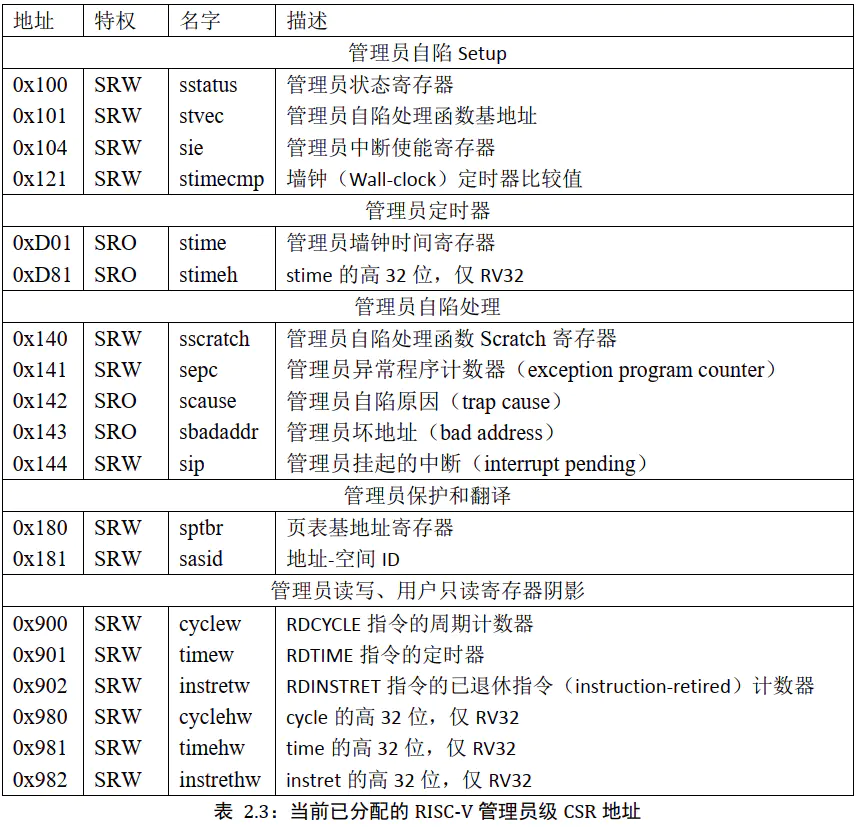
## （一）ECALL和SRET功能实现

### 1、相关CSR寄存器

标准RISC-V ISA设置了一个12位的编码空间（csr[11:0]）可用于4096个CSR。根据约定，CSR地址的高4位（csr[11:8]）用于编码 CSR 根据特权级读写的可访问性。最高2 位（csr[11:10]）指示这个寄存器是否是可以读/写（如只读是11）。后面2位（csr[9:8]）指示了能够访问这个 CSR 所需要的最低特权级。

RISC-V特权级可分为00用户级（user）、01特权级（supervisor）和11机器级（machine）。CSR寄存器根据不同特权级，有对应CSR用于表示和处理状态等，如我们设计的中断是由machine到supervisor级别的切换，该两个特权级涉及到如下一些相关的CSR寄存器。

supervisor级别



machine级别



对上述部分ECALL与SRET涉及到的寄存器，提前在opType.vh头文件中定义地址，方便实现。

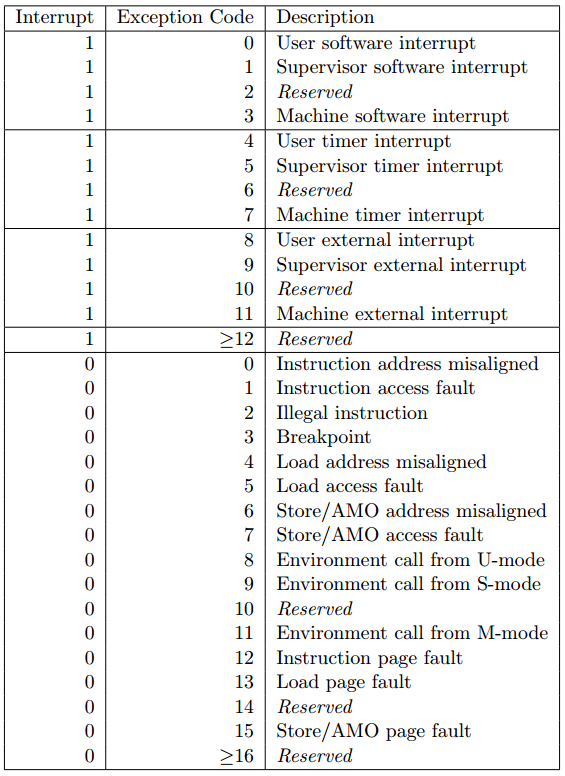
由于ECALL和SRET指令涉及到PC的跳转等，我们将其作为类似branch分支指令的处理，使之在EX阶段跳转。并在5级流水线上对CSR指令实现进行了调整，原本CSR的实现在WB阶段，但与CSR寄存器相关的读写操作提前，WB只余下对普通数据寄存器的写回。

### 2、ECALL指令

ECALL进入中断，需要对相关CSR寄存器更新。我们设定ECALL进入中断实现了从machine级至supervisor级的切换，因此需要执行如下行为：

（1）更新mepc，mepc写入ECALL的下一条指令的地址。

（2）更新mcause，根据产生异常的类型更新mcause，异常类型的编码如下所示：



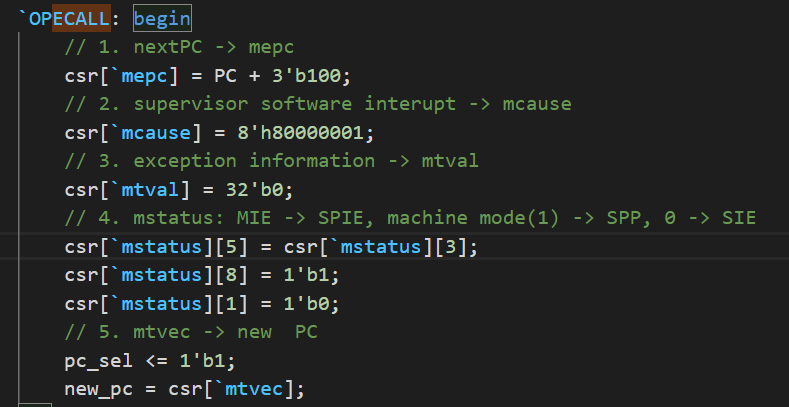
为简单实现，我们定义异常类型为32’h80000001，即对应Supervisor software interrupt。

（3）更新mtval，某些异常需要将异常相关的信息写入到mtval当中，我们假定没有相关的异常信息，即写入0即可。

（4）更新mstatus，mstatus记录中断使能等信息，由于我们涉及到了从machine到supervisor级别的切换，需要将异常发生前的MIE保存到SPIE当中，将异常发生前所处的特权级（machine对应为1）保存到SPP中，SIE设为0。这意味着在硬件上，RISC-V是不支持嵌套中断的。若要实现嵌套中断，则只能通过软件的方式来实现。

（5）跳转到mtvec中所定义的异常入口地址执行,初始时默认mtvec中的值已经指向了我们写的中断处理程序中。其中mtvec有两种模式，一种是直接模式，直接跳转到mtvec中的基地址执行。另一种是向量模式，根据mcause中的异常类型跳转到对应的异常处理程序首地址中执行。为简单实现，我们使用直接模式跳转到mtvec指向的地址执行中断处理。

实现的部分主要代码如下所示：



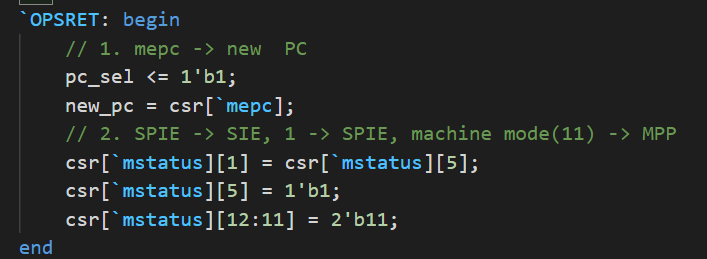
### 3、SRET指令

当中断处理程序执行完毕后，在程序最后会调用xRET指令来退出异常处理程序（machine下位mret，supervisor下为sret，user下为uret），我们设定中断处理为supervisor级别，从中断处理程序退出即为SRET指令，执行SRET指令硬件执行的行为如下：

（1）从mepc中定义的地址执行，恢复到异常发生前的程序流执行。

（2）更新mstatus，将异常发生前的mstatus的状态恢复，具体实现为SIE从SPIE中恢复，SPIE设置为1，特权模式设置为machine，即MPP设置为11。

部分主要代码如下所示：



## （二）Trap实现

有3种事件会导致CPU搁置正常的指令执行并强制将控制权转移到处理该事件的特殊代码。一种情况是系统调用，当用户程序执行ecall指令要求内核为它做一些事情时。另一种情况是例外：指令（用户或内核）做了一些非法的事情，例如除以零或使用无效的虚拟地址。第三种情况是设备中断，当设备发出需要注意的信号时，例如当磁盘硬件完成读取或写入请求时。

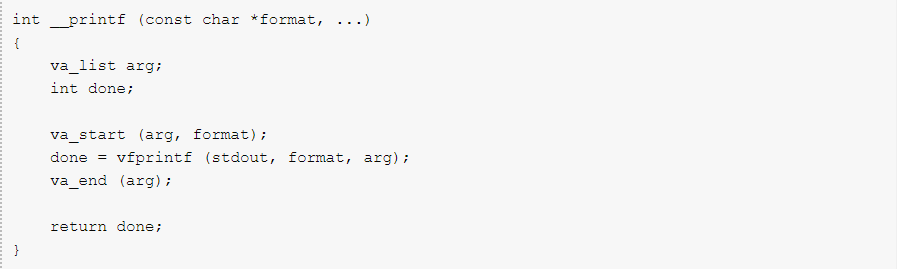
通常使用Trap（陷阱）作为这些情况的通用术语。通常，在陷阱发生时执行的任何代码稍后都需要恢复，并且不需要知道发生了任何特殊情况。也就是说，我们经常希望陷阱是透明的； 这对于中断尤其重要，被中断的代码通常不期望中断。通常的顺序是陷阱强制将控制权转移到内核中；内核保存寄存器和其他状态，以便可以恢复执行；内核执行适当的处理程序代码（例如，系统调用实现或设备驱动程序）；内核恢复保存的状态并从陷阱中返回；并且原始代码从它停止的地方恢复。

Printf执行过程：

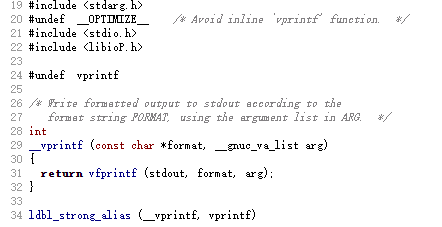
（1）从函数原型开始，函数原型在文件libc/libio/stdio.h中定义printf。



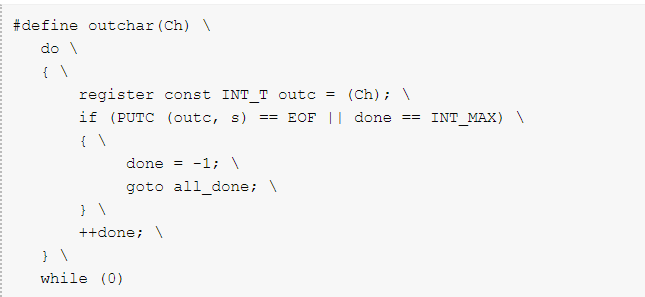
其函数实现如下：



可以在libc/stdio-common/vfprintf.c找到vfprintf的实现：



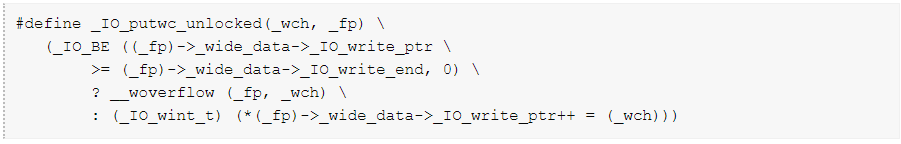
（2）在vfprintf定义文件中有个宏定义：



可以看到其依赖于PUTC也在同一文件中

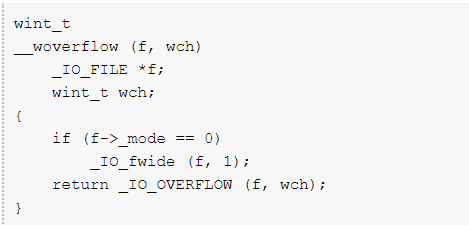


可以从libc/libio/libio.h获得IO\_putwc\_unlocked



它只是在做缓冲输出。如果文件指针的缓冲区中有足够的空间，那么它只会将字符插入其中，但如果没有，它会调用，由于缓冲区用完时的唯一选择是刷新到屏幕（或文件指针代表的任何设备）。

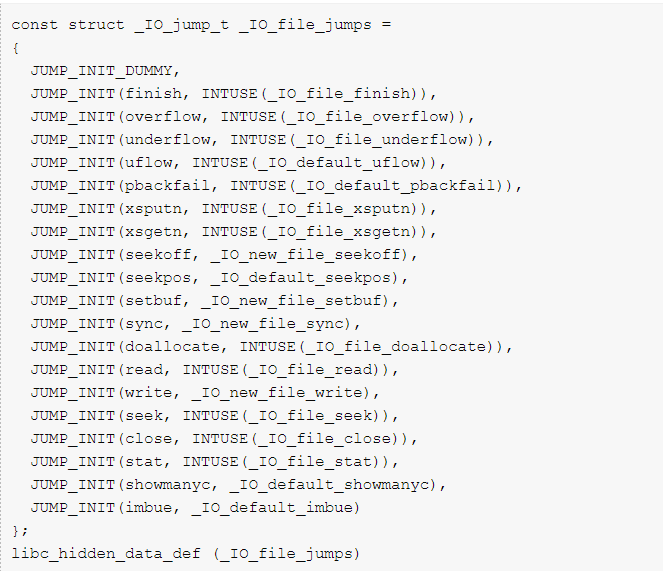
（3）查看 libc/libio/wgenops.c，会发现 :\_\_woverflow 的定义



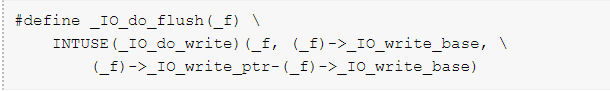
基本上，文件指针在GNU标准库中作为对象实现。它们具有数据成员和函数成员，可以使用宏的变体调用它们。在文件libc/libio/libioP.h中，会找到有关此技术的一些文档：JUMP。

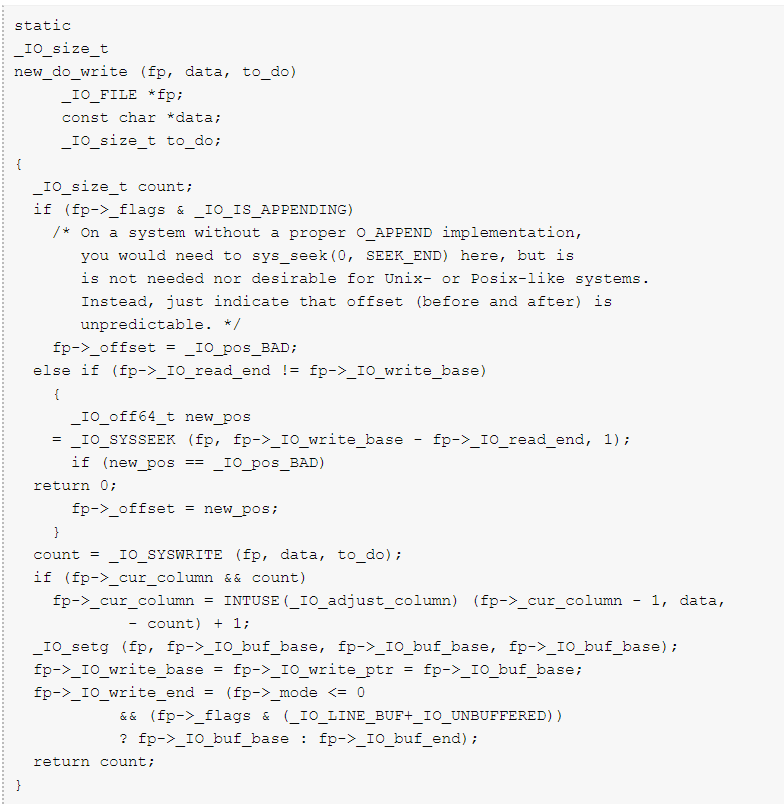
通过相关文件可以找到以下定义：



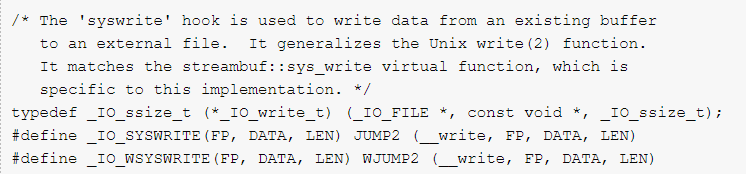


（4）源代码做了一堆更多的缓冲区操作，调用了\_IO\_do\_flush

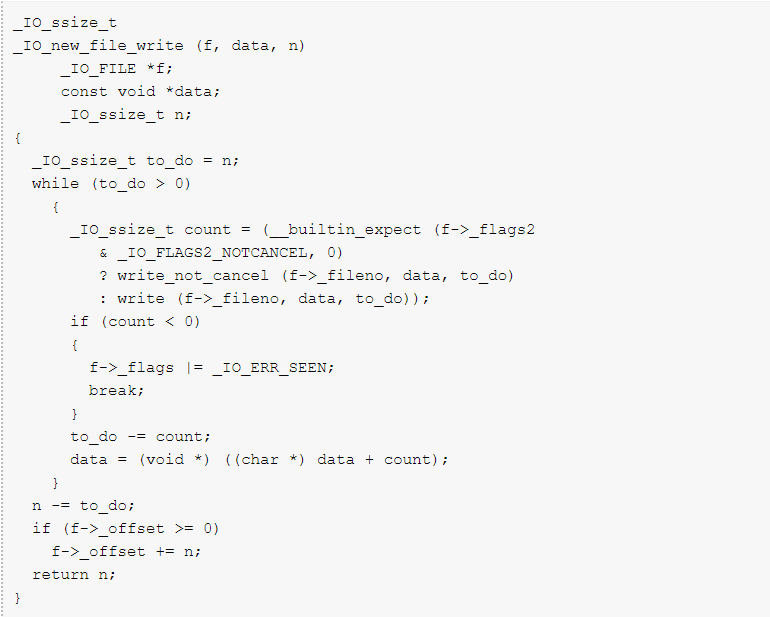




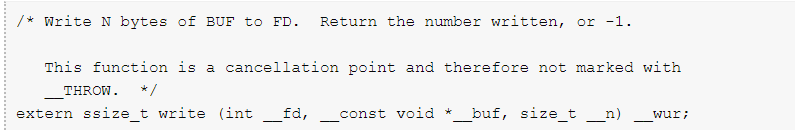
从中我们发现其调用了\_IO\_SYSWRITE。



所以在里面我们调用文件指针上的方法。我们从上面的跳转表中知道映射到\_IO\_new\_file\_write。



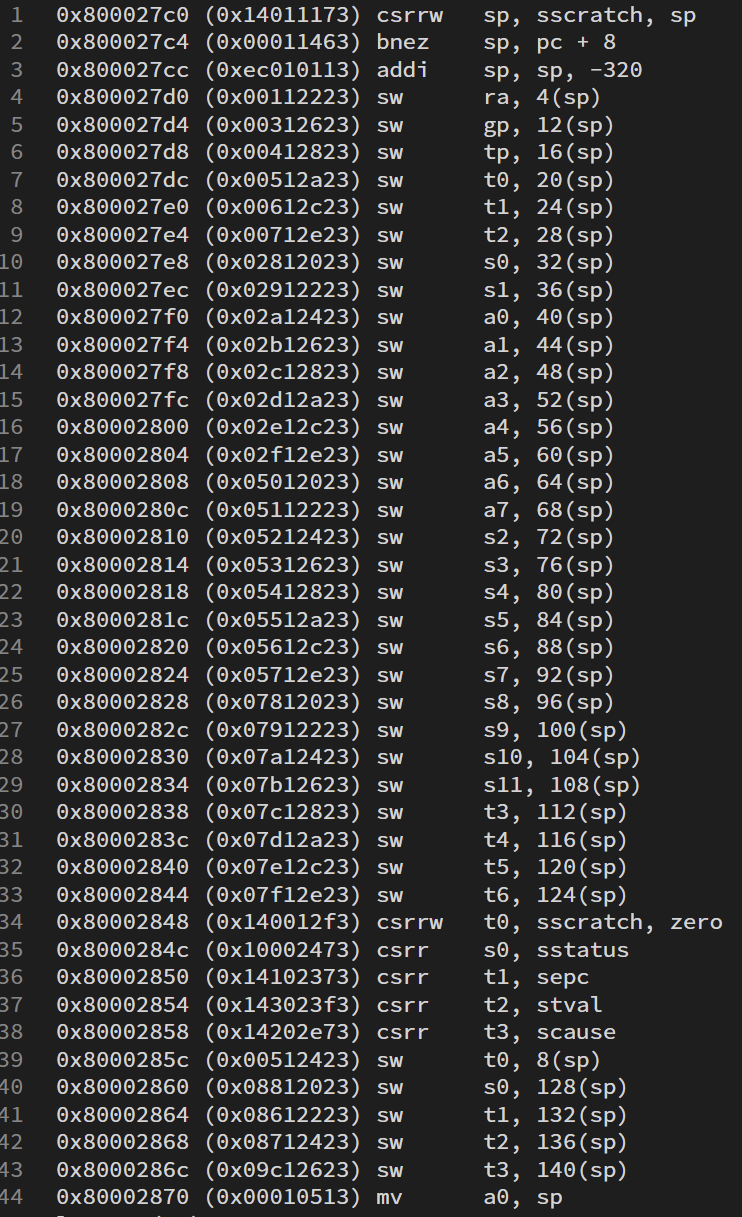
现在我们可以从libc/posix/unistd.h中找到write。



在GNU标准库中找到适用于Linux的write.c 文件。相反，您会发现以各种方式连接到操作系统写入功能的特定于平台的方法，所有这些都在libc/sysdeps/目录中。Write在 Linux如何做到这一点。有一个名为 sysdeps/unix/syscalls.list的文件，用于自动生成函数。表中相关数据为：write。这是一个系统调用的相关文件。

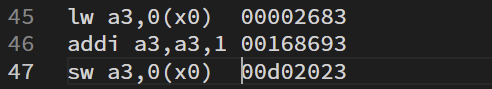
### 1、保护现场

进入trap之后首先程序需要保存进入trap之前的相关的寄存器和csr寄存器。



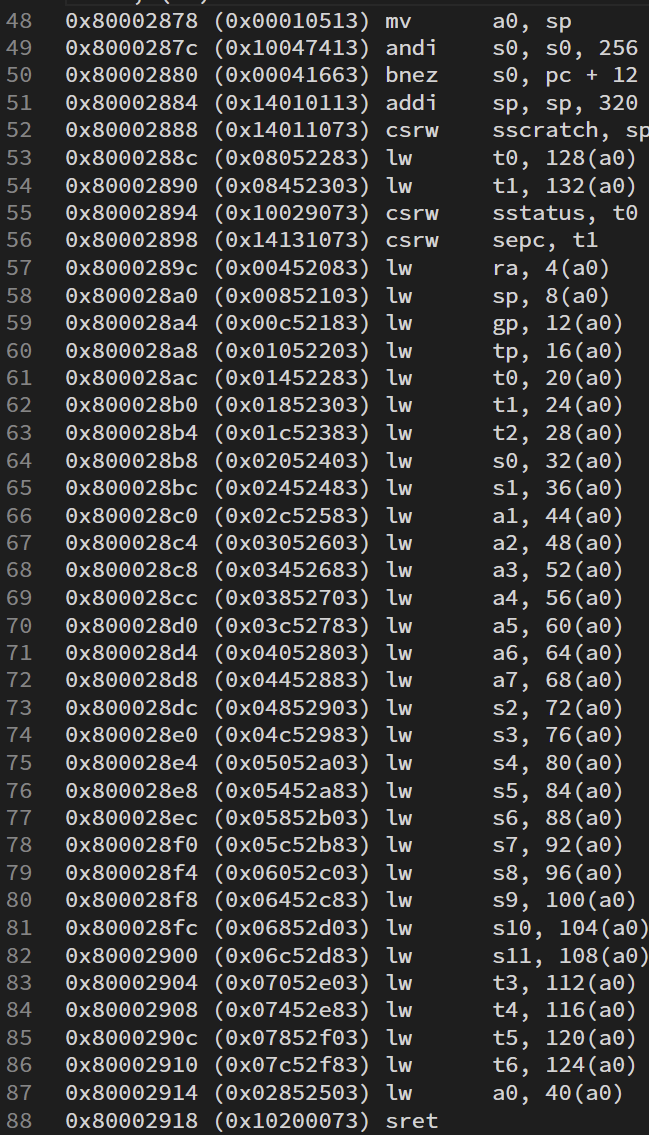
### 2、处理程序

针对于printf的trap处理程序而言，这步需要将相关输入写入系统缓冲区，并持续检测是否数据已经被相关I/O设备所接收，一直等I/O设备的信号，由于我们实现的RISCV流水线中并不具有该I/O设备，因此将该处换成读取32’h00000000初的值，并将其自增1后写回。



### 3、恢复现场

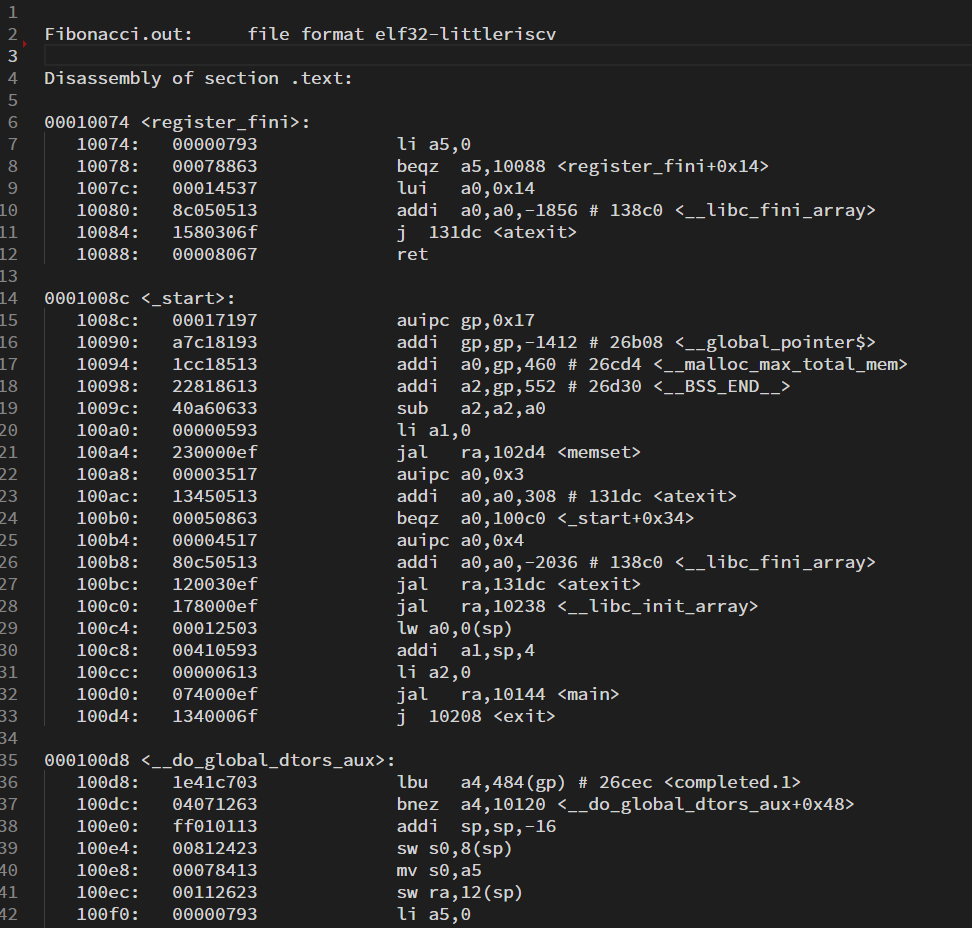
处理完成之后，根据保护现场所使用的栈指针将相关寄存器和csr寄存器恢复。通过SRET指令返回机器态。



# 四、实验结果

## （一）ELF可执行文件的转换

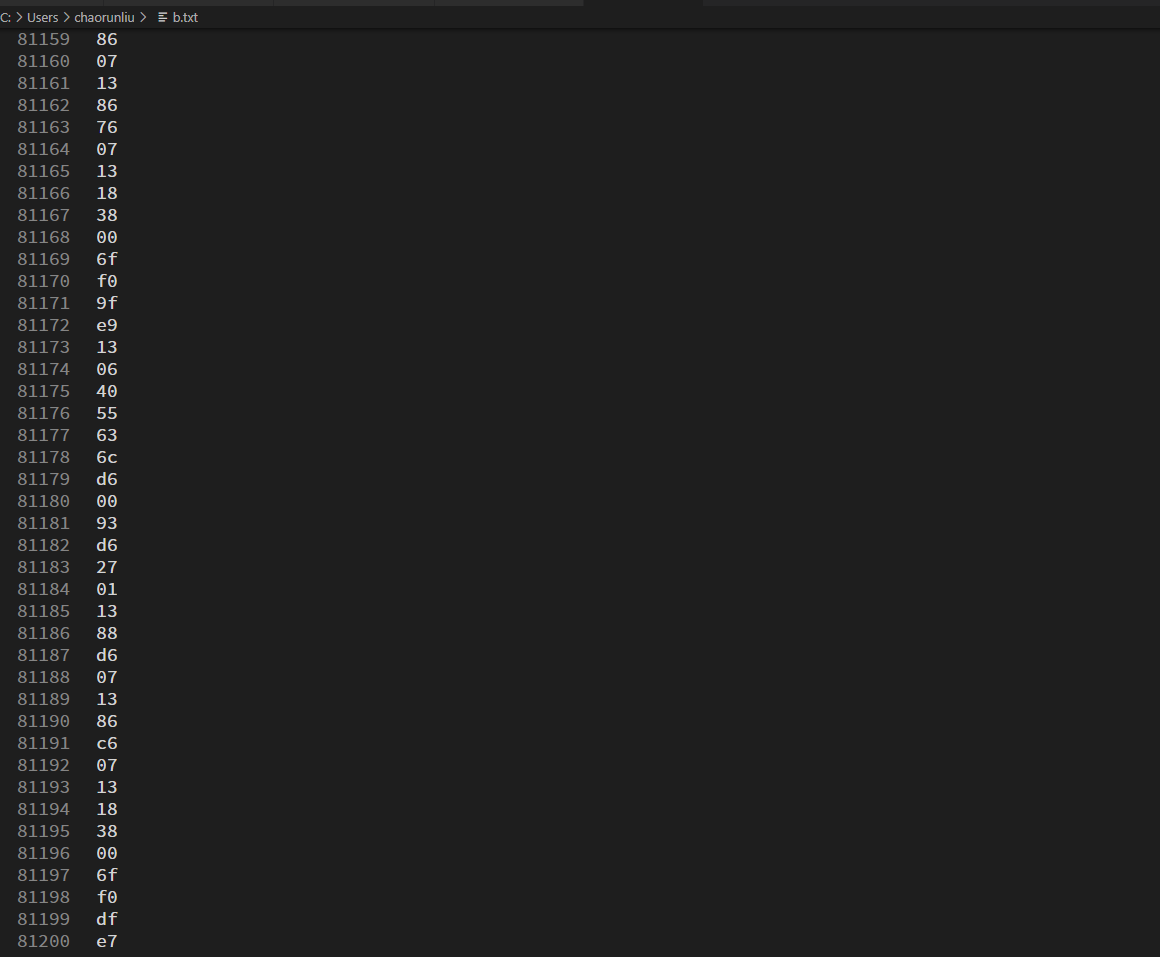
由于riscv32-unknown-elf-objdump仅可转换elf可执行文件为汇编代码，但是丢失了elf中数据段的内容，如下图。



因此需要将elf文件直接转换成16进制码文件，elf2hex可以帮助将elf可执行文件转换为16进制数。执行命令如下：

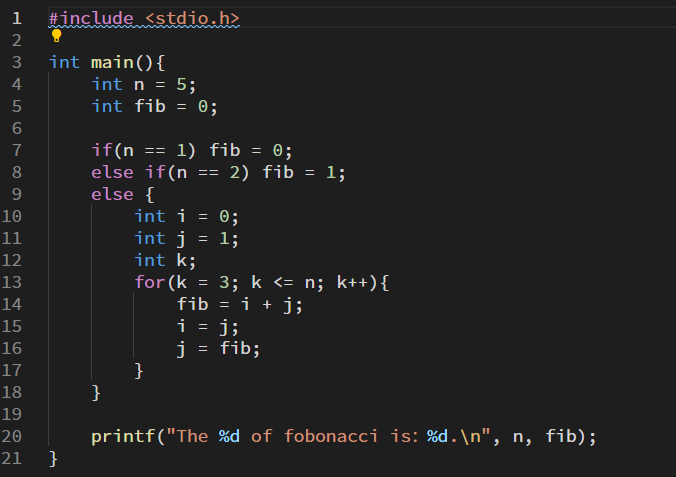


获得的内容部分如下：



## （二）printf功能C程序测试

使用带有printf的Fibonacci程序测试。C代码如下：



使用如下命令将其转换为elf可执行文件。

./riscv32-unknown-elf-gcc -E Fibonacci.c -o Fibonacci.i

./riscv32-unknown-elf-gcc -S Fibonacci.i -o Fibonacci.s

./riscv32-unknown-elf-gcc -c Fibonacci.s -o Fibonacci.o

./riscv32-unknown-elf-gcc -static Fibonacci.o -o Fibonacci.out

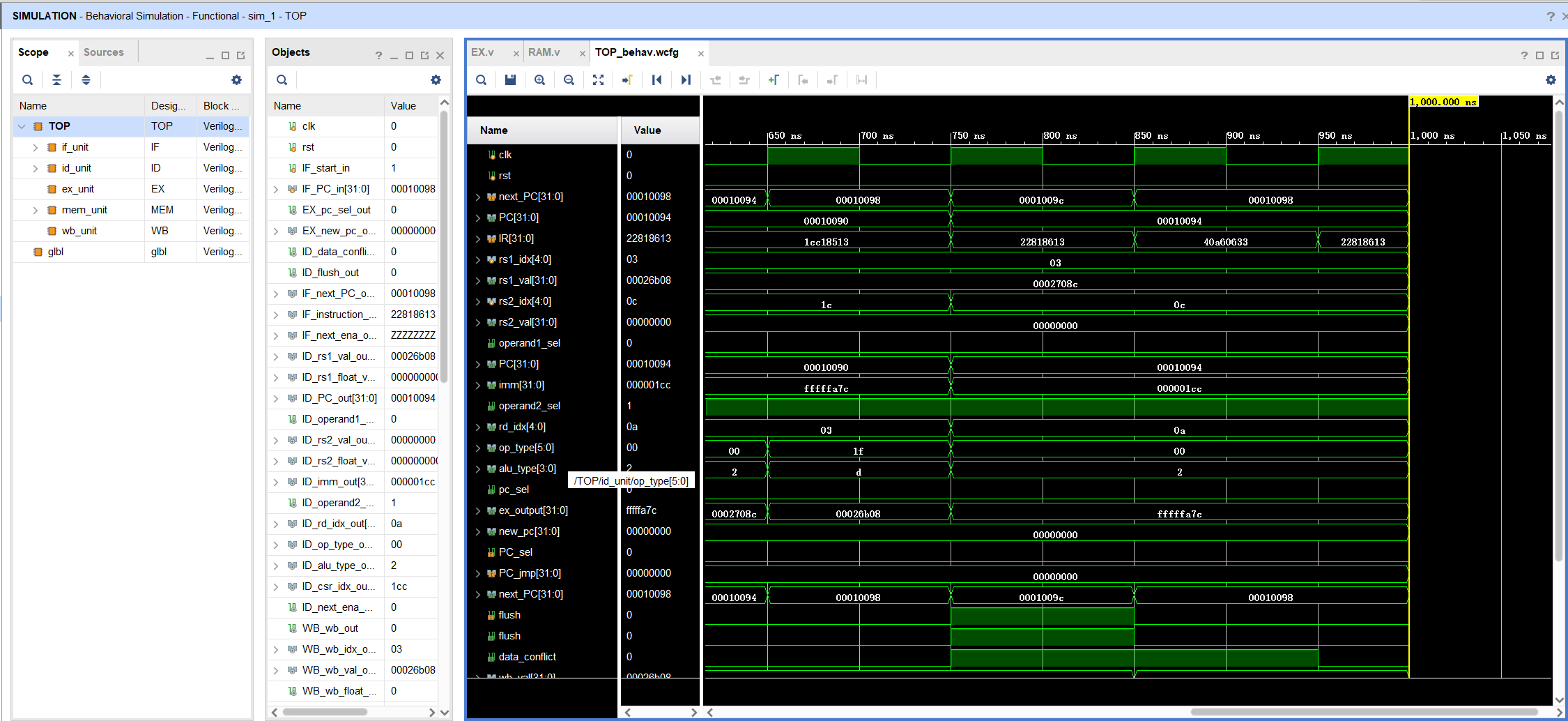
./riscv32-unknown-elf-objdump -S Fibonacci.out > Fibonacci.txt

将可执行文件转换为hex。

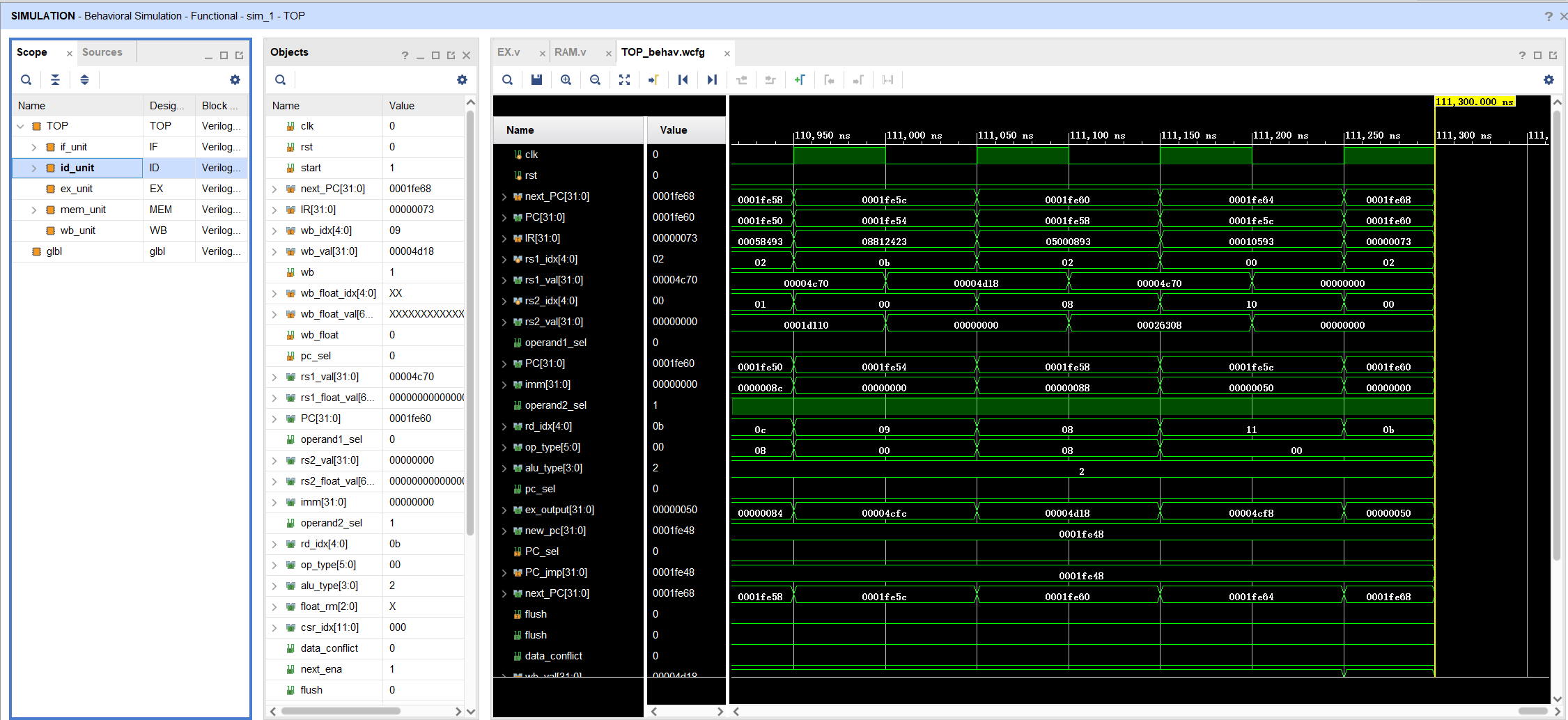
仿真过程：

1、将机器码Fibonacci.coe文件写入指令存储器，将f.txt读入存储器ram中，PC设置为0x0001008c。

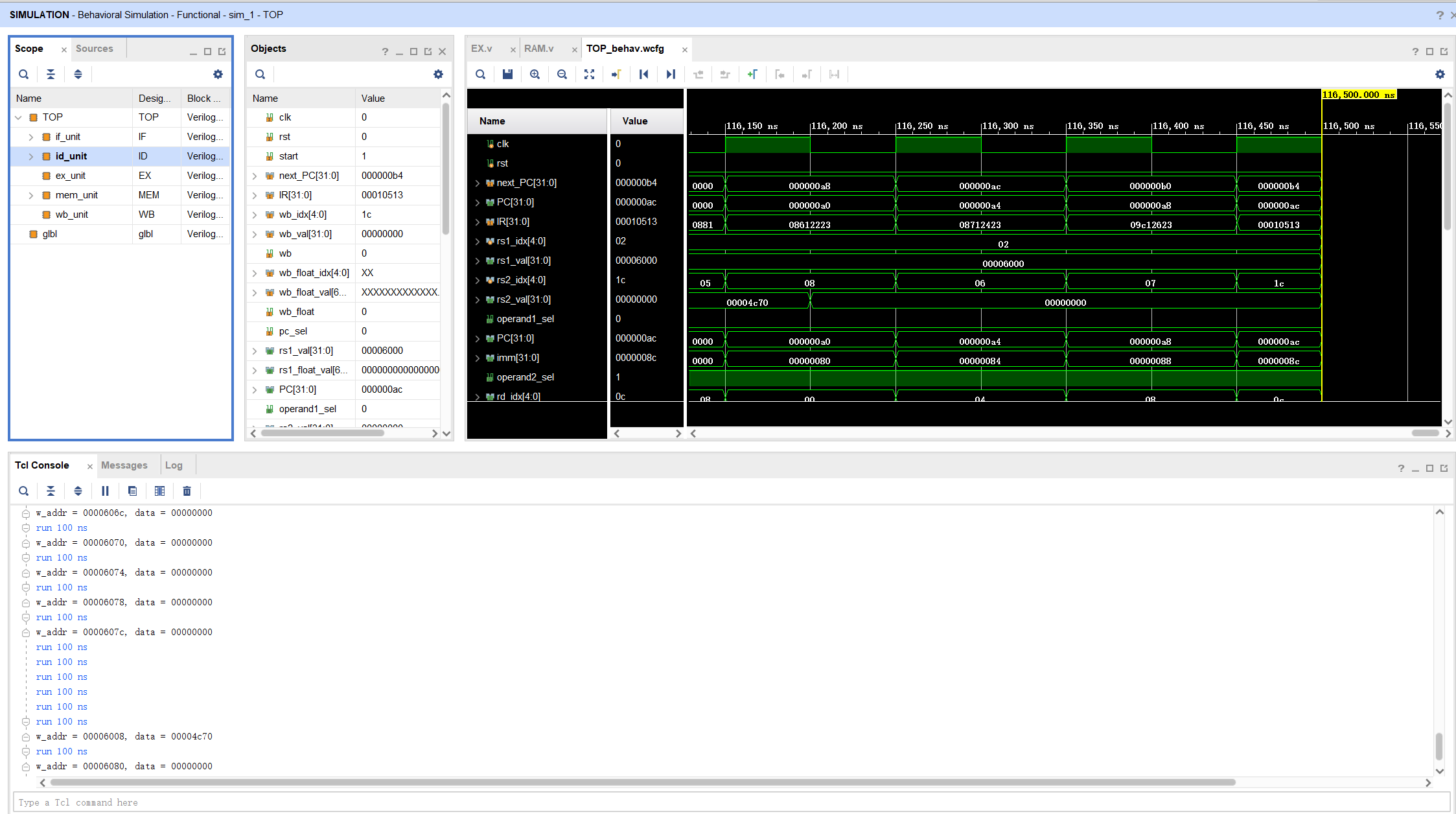
1. 运行前十个周期



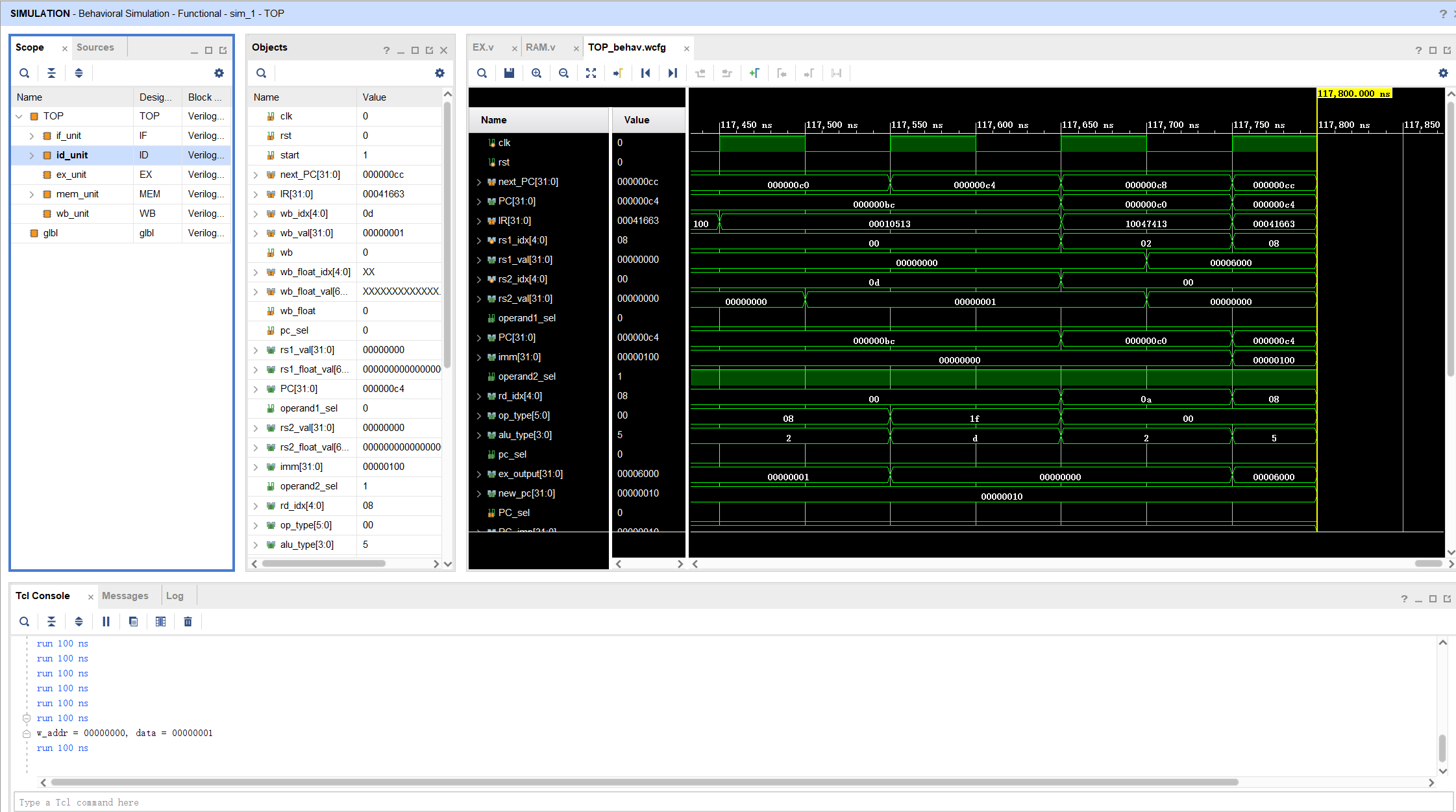
1. 运行至第一次系统调用ecall前



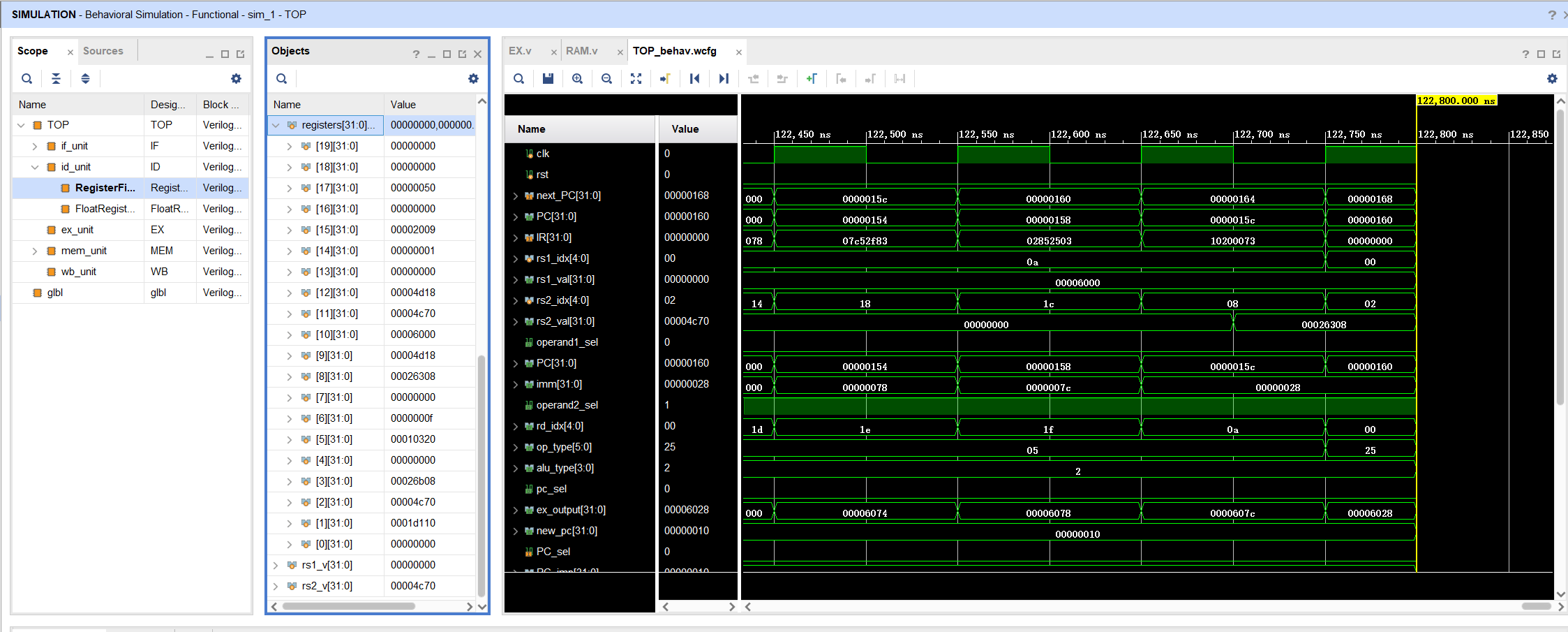
1. 进入trap程序保护现场



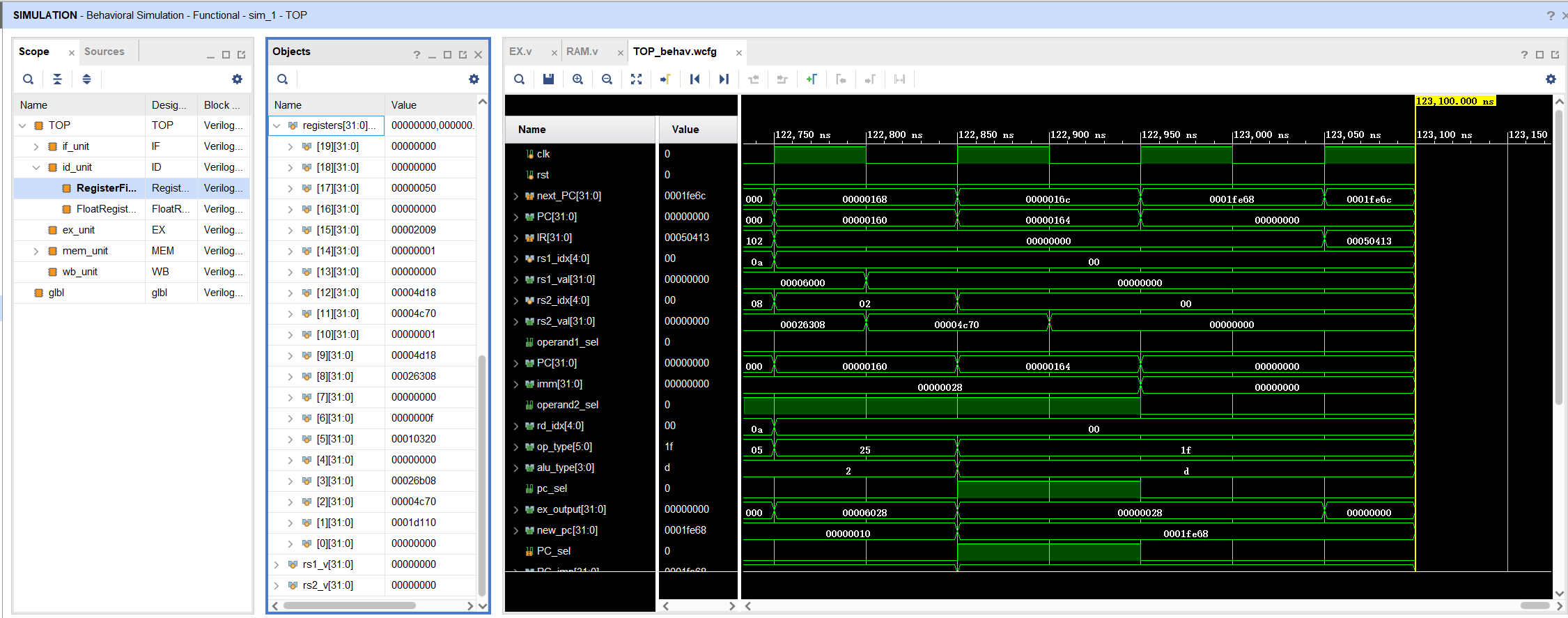
1. Trap处理程序



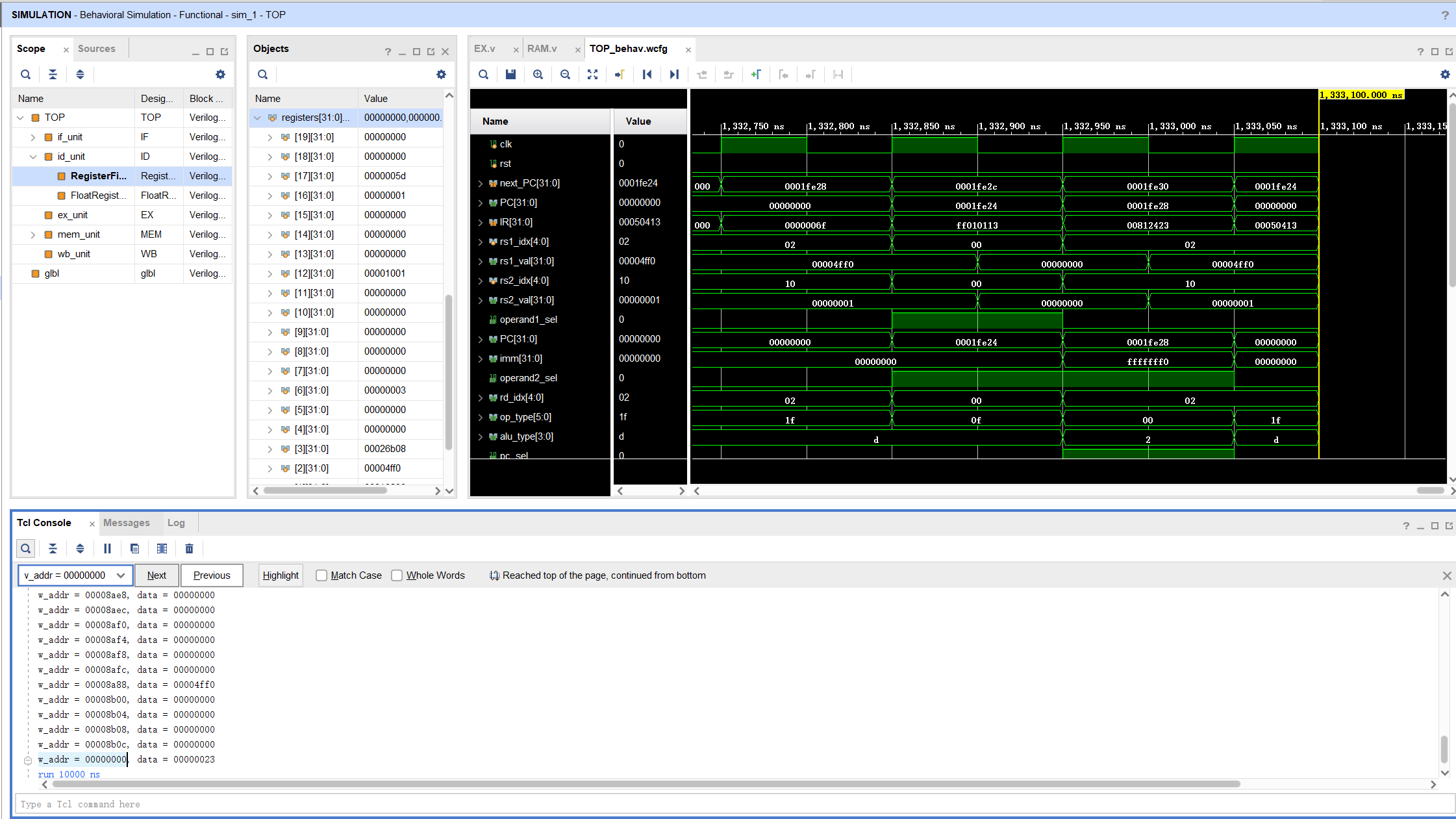
1. 恢复现场



1. 返回进入trap前地址



8、执行完成



# 五、实验总结

# （一）实验分工

许诗瑶：ECALL和SRET实现及相关CSR指令调整、测试程序调试

刘朝润：Trap实现、测试程序调试

# （二）实验遇到的问题

1、中断处理程序的实现时，此前一直在查阅资料或是从模拟器执行结果中试图找到能直接处理printf相关中断现有的程序，但查阅无果后才明白中断处理程序应由操作系统给出，我们因此只仿照了整个中断陷入过程，其中对中断的处理由访存加一的操作替换仅表示结果的执行。

2、栈空间大小的调整，调试过程中会由于不断地调用保留现场造成栈空间溢出问题，在调试中才逐渐探出适合的栈空间大小。

3、elf可执行文件使用原工具链转换会丢失elf中数据段的内容，使用elf2hex辅助工具转换为16进制获取原内容。

4、整个ecall、trap、ret过程涉及到特权级的转换、中断处理程序的跳转与执行、CSR寄存器更新、现场的保留与恢复等各个细节，资料查阅到的内容有限则结合研究模拟器执行结果，逐步完善与明确。

# （三）附件文件说明

RISCV文件夹：Vivado环境下项目工程代码

test\_program：测试程序的文件等