**计算机体系结构实验报告**

组 长： 许诗瑶 20023105

组 员： 刘朝润 20023114

组 员： 刘晓航 20020070

国防科技大学

National University of Defense Technology

**目录**

[一、实验要求 4](#_Toc77432827)

[（一）ISA定义 4](#_Toc77432828)

[（二）5级流水线实现 4](#_Toc77432829)

[（三）程序测试 4](#_Toc77432830)

[二、实验环境 4](#_Toc77432831)

[三、实验内容 4](#_Toc77432832)

[（一）ISA定义 4](#_Toc77432833)

[1、RV32I + RVM 4](#_Toc77432834)

[2、扩展其他指令集 6](#_Toc77432835)

[（1）riscv-gnu-tool工具链编译 6](#_Toc77432836)

[（2）确定扩展指令 7](#_Toc77432837)

[3、小结 8](#_Toc77432838)

[（二）5级流水线实现 8](#_Toc77432839)

[1、基本5级流水线 8](#_Toc77432840)

[（1）取指IF 8](#_Toc77432841)

[（2）译码ID 9](#_Toc77432842)

[（3）执行EX 10](#_Toc77432843)

[（4）访存MEM 11](#_Toc77432844)

[（5）写回WB 11](#_Toc77432845)

[（6）其他 12](#_Toc77432846)

[2、扩展功能 12](#_Toc77432847)

[（1）浮点乘除 12](#_Toc77432848)

[（2）Trap 14](#_Toc77432849)

[3、小结 21](#_Toc77432850)

[（三）测试程序 21](#_Toc77432851)

[1、4个测试程序 21](#_Toc77432852)

[（1）Fibonacci by RV 21](#_Toc77432853)

[（2）Bubble Sort by RV 22](#_Toc77432854)

[（3）Fibonacci by C 22](#_Toc77432855)

[（4）Bubble Sort by C 23](#_Toc77432856)

[2、ELF可执行文件的转换 24](#_Toc77432857)

[3、模拟器执行结果 25](#_Toc77432858)

[（1）模拟器介绍 25](#_Toc77432859)

[（2）修改模拟器输出 26](#_Toc77432860)

[（3）模拟器执行结果 27](#_Toc77432861)

[4、小结 28](#_Toc77432862)

[四、实验结果 28](#_Toc77432863)

[（一）Fibonacci by RV 28](#_Toc77432864)

[（二）Bubble Sort by RV 29](#_Toc77432865)

[（三）Fibonacci by C 30](#_Toc77432866)

[（四）Bubble Sort by C 34](#_Toc77432867)

[（五）浮点测试 38](#_Toc77432868)

[五、实验总结 40](#_Toc77432869)

[（一）实验历程 40](#_Toc77432870)

[（二）实验分工 40](#_Toc77432871)

[（三）实验遇到的问题 40](#_Toc77432872)

[（四）附件说明 41](#_Toc77432873)

# 一、实验要求

## （一）ISA定义

根据应用场景、功能需求等，定义目标指令集。本实验以实现由含printf的C语言程序编译生成的RISC-V指令程序为目的，确定目标ISA。

## （二）5级流水线实现

使用硬件编程语言实现5级流水线，能满足ISA每条指令的功能需求并能成功通过测试程序产生正确结果。

## （三）程序测试

包含4个测试程序用来检测5级流水线功能，分别为：手写Fibonacci和Bubble Sort的RV程序、含有printf的Fibonacci和Bubble Sort的C程序编译生成的RV汇编指令程序。

# 二、实验环境

硬件编程语言：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

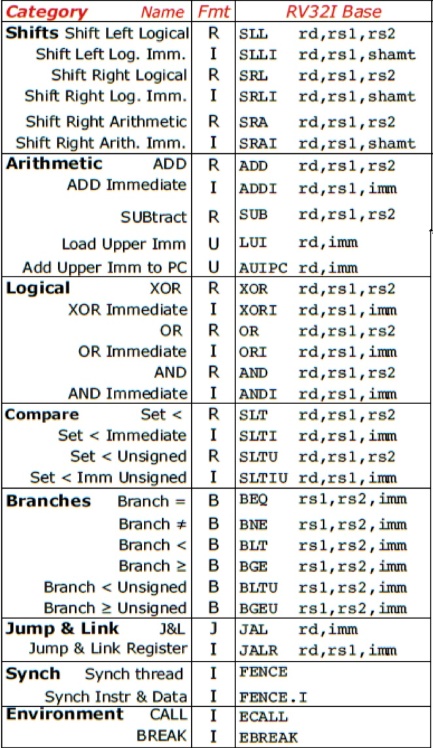
RISC-V ISA模拟器：Spike  
辅助转换工具：elf2hex

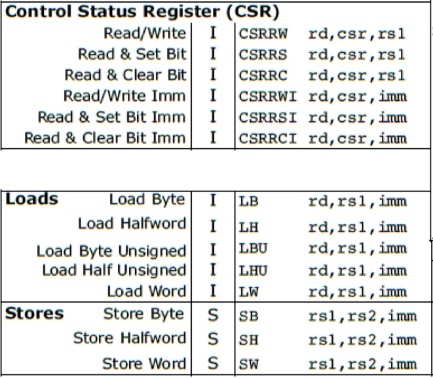
# 三、实验内容

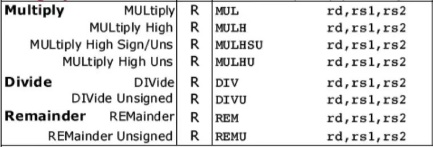
## （一）ISA定义

### 1、RV32I + RVM

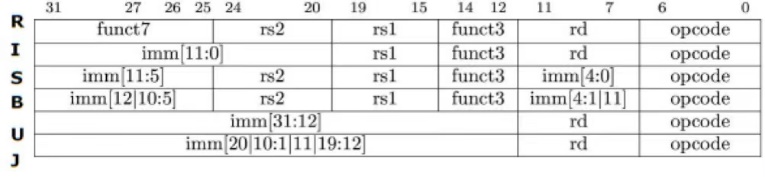
基本指令集包含RV32I和RVM两个RISC-V定义的指令子集，包含Shift移位、Arithmetic算术运算、Logical逻辑运算、Compare比较、Branch分支、Jump&Link跳转、Load取、Store存、CSR控制状态寄存器相关、Multiply乘、Divide除、Remainder取余，几大类指令，共51条，如下图所示：（暂不包含Synch同步和Environment环境相关类的指令）



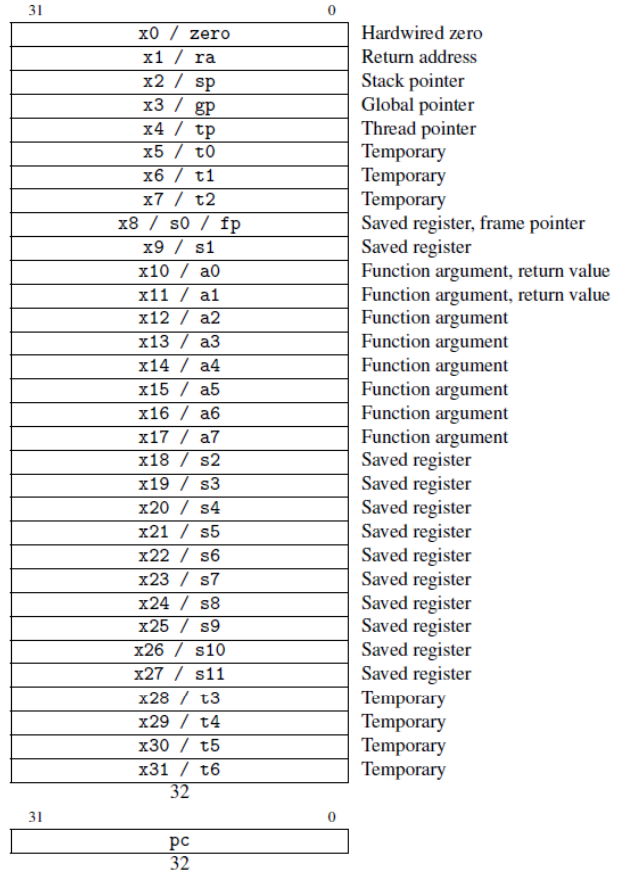




指令格式如下所示：



该指令集所需的整数寄存器如下所示：



### 2、扩展其他指令集

#### （1）riscv-gnu-tool工具链编译

对于手写RV汇编程序和C语言程序使用工具链辅助生成汇编指令及其对应的机器码。riscv-gnu-tool工具链的具体操作如下：

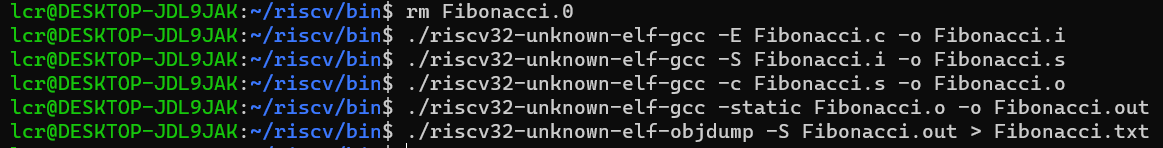
* 使用git clone工具链源码，因为使用了submodule所以可以使用—recursive在clone时同时clone submodule，故命令为git clone --recursive https://github.com/riscv/ riscv-gnu-toolchain。
* 由于使用的环境时win10下的wsl2（windows subsystem linux 2）的ubuntu20.04，故须西在相关的包，命令：sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev。
* 进入riscv-gnu-tool文件夹。
* 选择配置参数，开启了32i和fdm四个拓展，并使用了32bit hard float，并将生成的文件放在/home/lcr/riscv文件夹下，故配置参数为./configure --prefix=/home/lcr/riscv --with-arch=rv32imfd --with-abi=ilp32d。
* 编译：make，等待完成后进入/home/lcr/riscv即可使用riscv编译工具链。

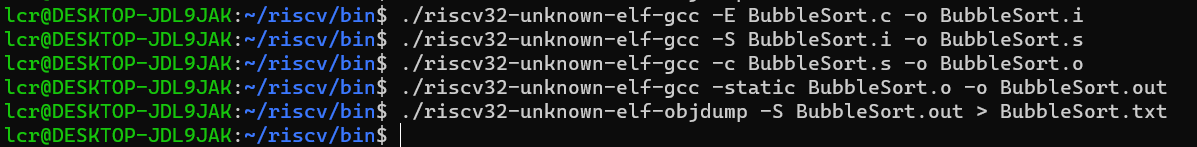
对于手写的RV汇编程序，编译后生成可执行文件，需最终反汇编获得最终带有机器码的汇编代码（生成文件Fibonacci\_ass.txt和BubbleSort\_ass.txt），如下所示：





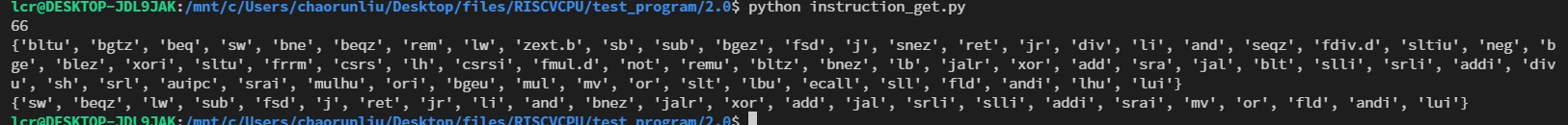
对于C语言的测试程序，由于使用printf函数，该函数依赖于静态库libc，故在第四步的时候使用-static参数静态链接libc，这样可以在反汇编代码中包含printf的机器码（文件Fibonacci.txt和BubbleSort.txt）：



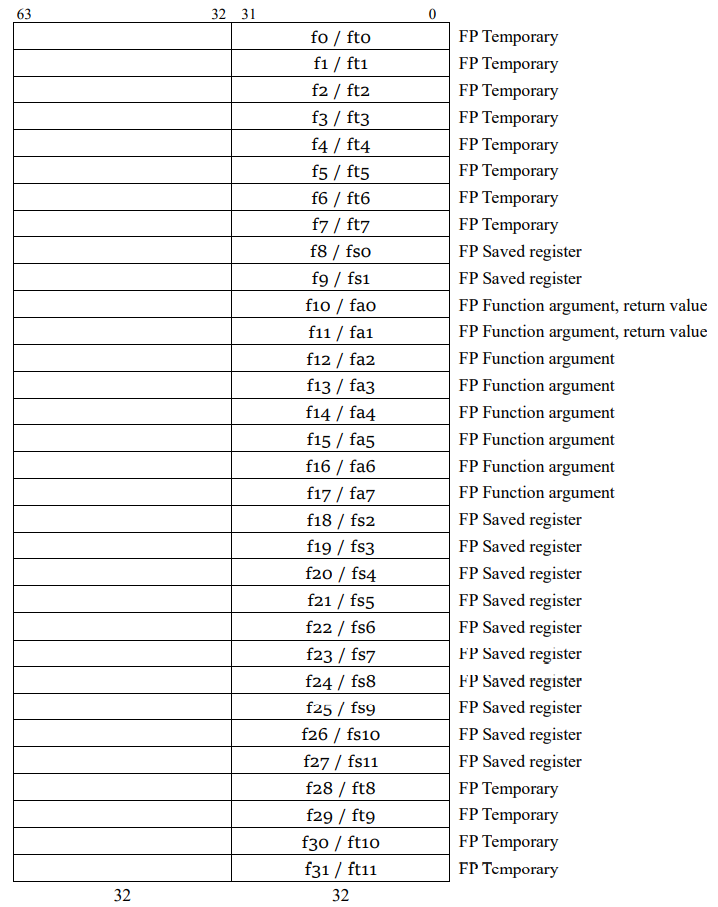


#### （2）确定扩展指令

利用生成Fibonacci.txt, Fibonacci\_ass.txt,BubbleSort.txt,BubbleSort\_ass.txt文件分析其中用到的指令，由于其中两个使用了printf语句的反汇编文件程度在21000行左右，我们编写了一个用于指令分析的python文件，它可以将这四个文件中用到的指令写入instruction.txt文件中，运行结果如下图：



整理测试程序涉及到的指令与此前确定的RV32I指令集比较，确定需要新增的指令为：FMUL.D、FDIV.S、FSD、FLD 和 ECALL，即浮点乘除和环境相关的指令。涉及到新增的浮点寄存器如下所示：



后续实现Trap功能时，中断处理程序需要恢复现场后返回，涉及到SRET指令，确定SRET为又一新增指令。

### 3、小结

ISA确定了5级流水线需要满足的基本功能，实验第一周确定了RV32I和RVM两大指令子集，指令详细分析可见附件ISA/1.0文件夹下指令和寄存器定义ISADefinition.xlsx和指令格式ISAFmt.xlsx文件。

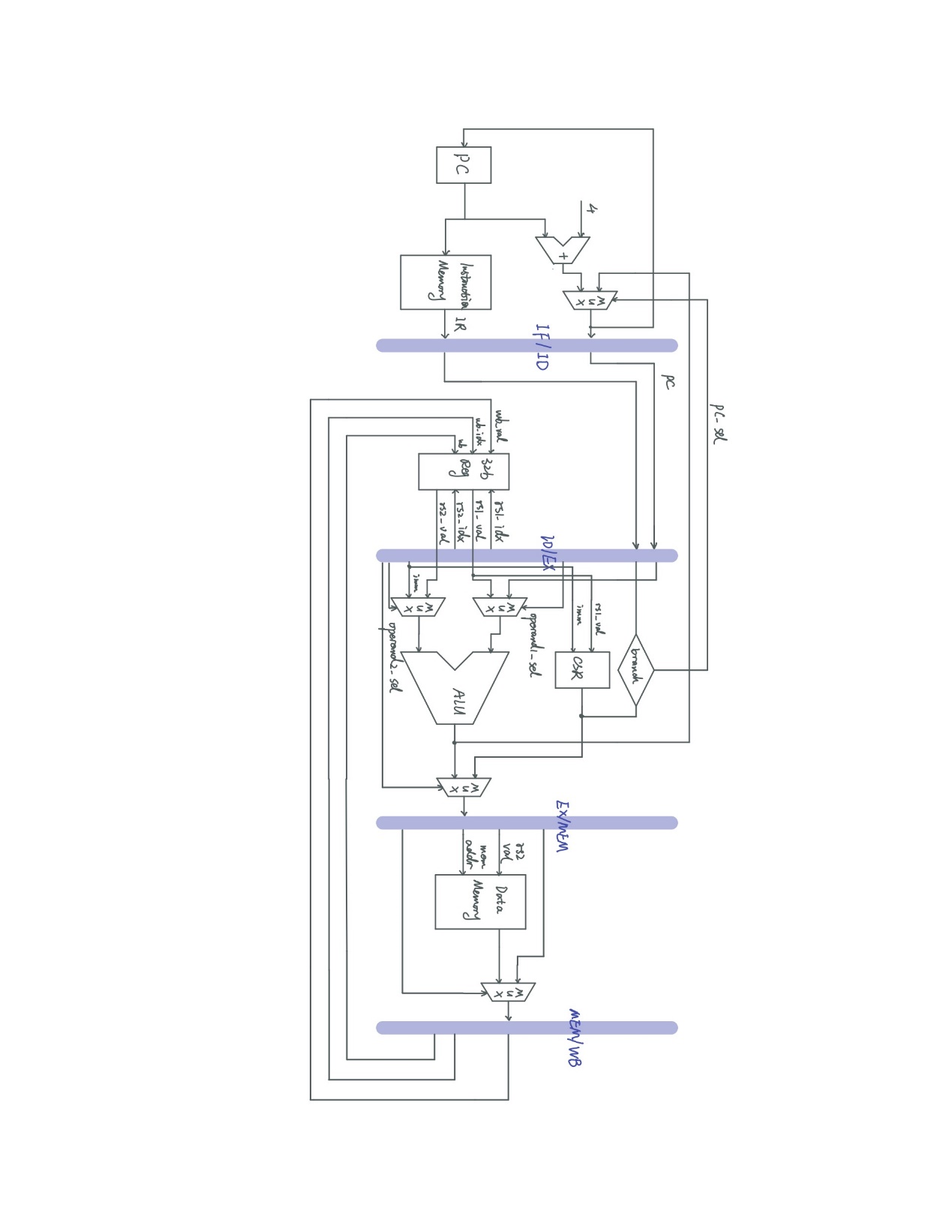
实验中期重新确定5级流水线的功能要求是通过printf的C语言程序测试，对ISA重新整理，与之前定义的ISA比较指令的增删情况，新增了浮点乘除和环境相关的指令，删除了程序没有用到的部分指令（如RVM中的无符号乘），指令详细分析可见附件ISA/2.0文件夹下指令和寄存器定义ISADefinition\_v2.xlsx文件。

实验后期实现Trap功能时涉及到中断程序的返回SRET指令，新增该指令，并将1.0版本中被2.0版本删除的指令继续加入ISA当中（因为此前5级流水线已实现这些指令的功能），最终确定5级流水线能实现的ISA可详见ISA/3.0文件夹下指令和寄存器定义ISADefinition\_v3.xlsx文件。

## （二）5级流水线实现

### 1、基本5级流水线

如下图所示为整体实现的基本架构，5级流水线为取指IF、译码ID、执行EX、访存MEM和写回WB，每级为一模块，每个模块固定包含时钟clk、重启rst、开始start三个输入信号和一个表示下一级是否可执行的next\_ena输出信号。最终由顶层文件TOP将各级模块互联。



#### （1）取指IF

根据当前PC从程序存储器中取出对应指令，并通过PC选择信号确定下一条指令的PC值为PC+4或是分支跳转指令后的PC。

|  |  |
| --- | --- |
| **信号名** | **功能** |
| **input** | |
| PC[31:0] | 当前需取指的PC值 |
| PC\_sel | PC选择信号，下一条指令选择PC+4或是由EX传来的分支指令 |
| PC\_jump[31:0] | EX传来的分支跳转后的指令PC值 |
| data\_conflic | 有无数据冲突 |
| flush | 是否需要冲刷本级操作 |
| **output** | |
| next\_PC[31:0] | 下一条指令的PC值 |
| instruction[31:0] | 以当前PC从程序存储器中取出的指令 |

#### （2）译码ID

译码阶段主要功能是实现对IF取得的指令译码其操作数及其操作，通过多级译码，对指令的不同字段逐步分析出指令的类型等，依次确定出指令的功能等，并保留相关的操作数与操作数选择信号等输出。

|  |  |
| --- | --- |
| **信号名** | **功能** |
| **input** | |
| next\_PC[31:0] | 由IF传来的下一条指令的PC值 |
| IR[31:0] | 指令寄存器，为IF取得的当前指令 |
| wb\_idx[4:0] | WB传来的需要写回整数寄存器的地址 |
| wb\_val[31:0] | WB传来的需要写回整数寄存器的值 |
| wb | WB传来的是否写回整数寄存器的选择信号 |
| wb\_float\_idx[4:0] | WB传来的写回浮点寄存器的地址 |
| wb\_float\_val[63:0] | WB传来的写回浮点寄存器的值 |
| wb\_float | WB传来的是否写回浮点寄存器的选择信号 |
| pc\_sel | 表示有无分支跳转需要冲刷本级 |
| **output** | |
| rs1\_val[31:0] | 对当前指令译码后并从寄存器文件中得到的寄存器rs1的值 |
| rs1\_float\_val[63:0] | 对当前指令译码后从浮点寄存器中得到的寄存器rs1的值 |
| PC[31:0] | 当前指令的PC值 |
| operand1\_sel | 操作数1的选择信号，为0选择rs1值，为1选择PC值 |
| rs2\_val[31:0] | 对当前指令译码后并从寄存器文件中得到的寄存器rs2的值 |
| rs2\_float\_val[63:0] | 对当前指令译码后从浮点寄存器中得到的寄存器rs2的值 |
| imm[31:0] | 经过指令要求的相关扩展后的32位立即数 |
| operand2\_sel | 操作数2的选择信号，为0选择rs2值，为1选择imm值 |
| rd\_idx[4:0] | 对当前指令译码出的写回寄存器rd的下标 |
| op\_type[4:0] | 对当前指令译码出的操作类型，可详见opType.vh头文件 |
| alu\_type[3:0] | 对当前指令译码出的alu操作类型，可详见opType.vh头文件 |
| float\_rm[2:0] | 浮点的舍入模式：RNE、RTZ、RDN、RUP或RMM |
| csr\_idx[11:0] | 对csr指令译码出的csr寄存器下标 |
| data\_conflict | 译码检测当前指令是否与之前已在执行的指令有数据冲突 |
| flush | 是否有数据冲突需要冲刷前两级操作 |

#### （3）执行EX

执行阶段主要是通过两个多路选择器对两个操作数进行选择，分别有rs1 op rs2、rs1 op imm、PC op imm、不做计算（lui）和对CSR寄存器的操作等，通过译码的alu\_type选择进行的计算操作类型，最后再通过op\_type对特殊的指令处理，如判断分支指令是否跳转等。

|  |  |
| --- | --- |
| **信号名** | **功能** |
| **input** | |
| rs1\_val[31:0] | 由ID传来的寄存器rs1的值 |
| rs1\_float\_val[63:0] | 由ID传来的浮点寄存器rs1的值 |
| PC[31:0] | 由ID传来的PC的值 |
| operand1\_sel | 由ID传来的操作数1的选择信号 |
| rs2\_val[31:0] | 由ID传来的寄存器rs2的值 |
| rs2\_float\_val[63:0] | 由ID传来的浮点寄存器rs2的值 |
| imm[31:0] | 由ID传来的imm的值 |
| operand2\_sel | 由ID传来的操作数2的选择信号 |
| rd\_idx[4:0] | 由ID传来的目标写回寄存器的下标 |
| op\_type[4:0] | 由ID传来的指令操作类型，可详见opType.vh头文件 |
| alu\_type[3:0] | 由ID传来的指令alu操作类型，可详见opType.vh头文件 |
| float\_rm[2:0] | 浮点的舍入模式 |
| csr\_idx[11:0] | 由ID传来的csr寄存器的下标 |
| **output** | |
| rs1\_val\_out[31:0] | rs1寄存器的值 |
| rs1\_float\_val\_out[63:0] | rs1浮点寄存器的值 |
| PC\_out[31:0] | PC值 |
| rs2\_val\_out[31:0] | rs2寄存器的值 |
| rs2\_float\_val\_out[63:0] | rs2浮点寄存器的值 |
| imm\_out[31:0] | 立即数imm的值 |
| rd\_idx\_out[4:0] | 目标写回寄存器rd的下标 |
| op\_type\_out[4:0] | 当前指令的操作类型 |
| csr\_idx\_out[11:0] | csr寄存器下标 |
| ex\_output[31:0] | alu执行结果 |
| ex\_float\_output[31:0] | 浮点运算执行结果 |
| new\_pc[31:0] | 对于跳转指令为目标分支的PC值 |
| pc\_sel | PC是否跳转的选择信号 |
| mask[3:0] | 对于store指令，通过掩码mask表示写回一字节或半字或一字 |
| store\_ena | 是否可以往数据存储器中存储数据 |

#### （4）访存MEM

MEM阶段主要执行对数据存储器的load和store操作。

|  |  |
| --- | --- |
| **信号名** | **功能** |
| **input** | |
| ex\_output[31:0] | 由EX传来的ALU计算出的值 |
| ex\_float\_output[63:0] | 浮点运算执行结果 |
| rd\_idx[4:0] | 目标写回寄存器的下标 |
| op\_type[4:0] | 当前指令操作类型 |
| rs1\_val[31:0] | 寄存器rs1的值 |
| rs1\_float\_val[63:0] | 浮点寄存器rs1的值 |
| rs2\_val[31:0] | 寄存器rs2的值 |
| rs2\_float\_val[63:0] | 浮点寄存器rs2的值 |
| imm[31:0] | 立即数的值 |
| PC[31:0] | 当前PC值 |
| csr\_idx[11:0] | csr相关寄存器的下标 |
| mask[3:0] | 用于辅助store指令功能的掩码 |
| store\_ena | 是否对数据存储器存储 |
| **output** | |
| rd\_val[31:0] | 确定最终写回寄存器的值（可能是alu执行出来的，可能是从存储器中load的值） |
| rd\_float\_val[63:0] | 需要写回浮点寄存器的值 |
| rd\_idx\_out[4:0] | 写回寄存器的下标 |
| op\_type\_out[4:0] | 指令操作类型 |
| rs1\_val\_out[31:0] | rs1的值 |
| imm\_out[31:0] | 立即数的值 |
| csr\_idx\_out[11:0] | csr寄存器下标 |

#### （5）写回WB

WB主要是对目标寄存器rd的写回，以及对csr指令写回普通数据寄存器相关的操作。

|  |  |
| --- | --- |
| **信号名** | **功能** |
| **input** | |
| rd\_val[31:0] | 需要写回rd寄存器的值 |
| rd\_float\_val[63:0] | 需要写回rd浮点寄存器的值 |
| rd\_idx[4:0] | 写回rd寄存器的下标地址 |
| op\_type[4:0] | 当前指令操作类型 |
| rs1\_val[31:0] | 寄存器rs1的值，用于csr相关操作 |
| imm[31:0] | 立即数的值，用于csr相关操作 |
| csr\_idx[11:0] | csr寄存器下标 |
| fflags\_accured\_exceptions[4:0] |  |
| **output** | |
| wb | 是否需要写回寄存器的选择信号 |
| wb\_idx[4:0] | 写回寄存器的下标 |
| wb\_val[31:0] | 写回寄存器的值 |
| wb\_float | 是否写回浮点寄存器的选择信号 |
| wb\_float\_idx[4:0] | 写回浮点寄存器的值 |
| wb\_float\_val[63:0] | 写回浮点寄存器的值 |

#### （6）其他

* 其他模块

opType头文件定义了op和alu操作类型和部分特殊寄存器及其对应地址；

prgROM为指令存储器，通过调用IP核实现，通过加载.coe汇编指令文件初始化；

RAM为数据存储器，可写入数据内容；

RegisterFiles为整数寄存器文件模块；

FloatRegisterFiles为浮点寄存器文件模块；

TOP将各模块互联形成5级流水线。

* 分支跳转

对于涉及到分支跳转的指令，如Branch类、J类和ECALL、SRET指令，需要在EX阶段计算出跳转PC地址后通过输出的pc\_sel信号传回IF阶段，执行正确的分支指令，但由于此时已IF已处理跳转指令后的两条指令，需要通过pc\_sel为1的信号传回IF和ID冲刷流水线。

对于条件判断的分支跳转，默认为不跳转，只有当在EX执行时通过条件判断确定跳转时，才如上述进行跳转和流水线的冲刷，否则如同正常指令继续通过MEM和WB阶段。

* 数据冲突

本实验实现顺序单发射5级流水线，数据冲突只有写后读一种，即下一指令需要读取上一指令写入的值，但此时该指令执行时，上一指令还未写入。我们使用最简单的方法，即直接停顿，消除这种数据冲突。

在ID译码阶段，通过维护一张寄存器读写表，来检测数据冲突的有无与消除与否。当译码指令对某些寄存器写时，寄存器写表相应寄存器置1，只有当指令执行完毕WB写回后，写回寄存器对应置0。当检测到写后读冲突时，通过ID发出flush冲刷信号，冲刷IF，产生停顿，保持该条指令在ID及之后阶段的执行，知道成功写回消除冲突，流水线恢复。

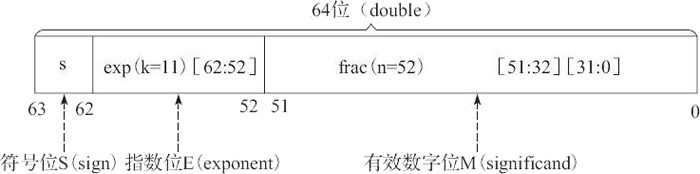
### 2、扩展功能

#### （1）浮点乘除

RISC-V使用的浮点数标准为IEEE754-2008标准，我们基于该标准和RISC-V官方文档中关于FD拓展的描述实现浮点指令FMUL.D、FDIV.S、FSD和FLD四条指令。该四条指令主要针对双精度浮点数进行处理。

* 浮点表示形式

使用IEEE754标准中的64位浮点数，其格式如下：



其中，frac[31:0]存放小数段的低32位（即第0位存放整个小数段的最低有效位LSB，第31位存放小数段低32位的最高有效位MSB）；frac[51:32]存放小数段的高20位（即第32位存放高20位的最低有效位LSB，第51位存放整个小数段的最高有效位MSB）；第52位存放指数段exp的最低有效位LSB，第62位存放指数段exp的最高有效位MSB；最高位，即第63位存放符号s。

指数可以为正数，也可以为负数。为了处理负指数的情况，实际的指数值按要求需要加上一个偏置（Bias）值作为保存在指数段中的值。因此，这种情况下的指数段被解释为以偏置形式表示的有符号整数。即指数的值为：E=e-Bias。其中，e是无符号数，其位表示为ek-1…e1e0。而Bias是一个等于2k-1-1（单精度是127，双精度是1023）的偏置值。由此产生指数的取值范围：单精度为-126 ~ +127，双精度-1022 ~ +1023。

对小数段frac，可解释为描述小数值f，其中0≤f<1，其二进制表示为0.fn-1…f1f0，也就是二进制小数点在最高有效位的左边。有效数字定义为M=1+f。有时候，这种方式也叫隐含的以1开头的表示法，因为我们可以把M看成一个二进制表达式为1.fn-1fn-2…f0的数字。能够调整指数 E，使得有效数字 M 的范围为 1≤M<2（假设没有溢出），这种表示方法是一种轻松获得一个额外精度位的技巧。同时，由于第一位总是等于 1，因此不需要显式地表示它。

特殊数值表示如下：

* 浮点乘法与除法的实现

浮点乘法的实现原理如下：两个数的符号位相乘位符号位，阶码相加再减去bias，小数位相乘选取前52位，再取后三位为舍入位。

浮点除法的实现原理如下：两个数的符号位相乘位符号位，阶码相减再加上bias，被除数扩展位数55位与除数相除得到结果取前52位，后三位为舍入位。

* 舍入与异常

IEEE754标准中共有五种舍入模式，分别如下表：

|  |  |
| --- | --- |
| 舍入模式 | 说明 |
| RNE | 向距离最近的偶数舍入 |
| RTZ | 向0的方向舍入 |
| RDN | 向负无穷大方向舍入 |
| RUP | 向正无穷大方向舍入 |
| RMM | 向距离最近的最大数量级数舍入 |

在指令执行过程中有3位rm码来确定舍入模式，其中3位rm码中有6种舍入模式，五种与上表相同，还有一种则是根据csr和fflags寄存器中的7位到5位设定的模式舍入。

浮点操作中共有五种异常，无效操作、除数位0，上溢、下溢和不精确。分别为csr与fflags寄存器中的后五位。

* + 无效操作，当乘法中出现NaN、、，除法中出现 NaN、等会导致无效操作异常。
  + 除数为0，当浮点除法中的除数为0。
  + 上溢与下溢

11位阶码：表示范围-1022~1023

0 0000 0000 0001 ~ 0 0011 1111 1110  -1022~-1

0 0100 0000 0001 ~ 0 0111 1111 1110  1~1023

下溢：

负数+负数

000 0000 0001 + 011 1111 1110 - 011 1111 1111 = 0 0000 0000 0000

000 0000 0001 + 011 1111 1101 - 011 1111 1111 = 1 1111 1111 1111

000 0000 0001 + 011 1111 1100 - 011 1111 1111 = 1 1111 1111 1110

000 0000 0001 + 011 1111 1011 - 011 1111 1111 = 1 1111 1111 1101

……

000 0000 0001 + 000 0000 0001 - 011 1111 1111 = 1 1100 0000 0011

负数-正数

000 0000 0001 - 100 0000 0001 + 011 1111 1111 = 1 1111 1111 1111

000 0000 0001 - 100 0000 0010 + 011 1111 1111 = 1 1111 1111 1110

000 0000 0001 - 100 0000 0011 + 011 1111 1111 = 1 1111 1111 1101

……

000 0000 0001 – 111 1111 1110 + 011 1111 1111 = 1 1100 0000 0010

上溢：

正数-负数

111 1111 1110 - 011 1111 1110 + 011 1111 1111 = 0 0111 1111 1111

111 1111 1110 - 011 1111 1101 + 011 1111 1111 = 0 1000 0000 0000

111 1111 1110 - 011 1111 1100 + 011 1111 1111 = 0 1000 0000 0001

……

111 1111 1110 - 000 0000 0001 + 011 1111 1111 = 0 1011 1111 1100

正数+ 正数

111 1111 1110 + 100 0000 0001 – 011 1111 1111 = 0 1000 0000 0000

111 1111 1110 + 100 0000 0010 – 011 1111 1111 = 0 1000 0000 0001

111 1111 1110 + 100 0000 0011 – 011 1111 1111 = 0 1000 0000 0010

……

111 1111 1110 + 111 1111 1110 – 011 1111 1111 = 0 1011 1111 1101

阶码使用无符号表示，将乘除结果的阶码位设为12位，

If [11:10] == 11 or [10:0] == 000 0000 0000是下溢

else if [11:10] == 10 or [10:0] == 111 1111 1111是上溢

#### （2）Trap

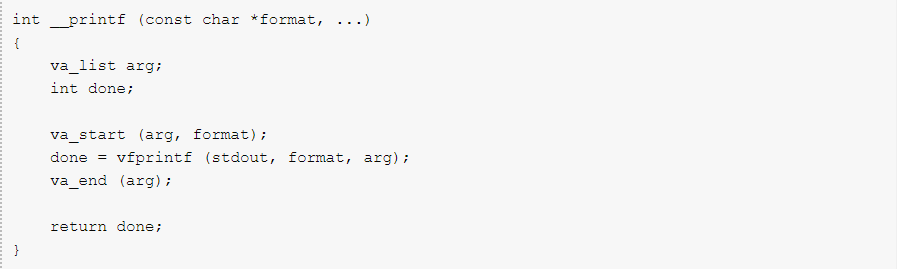
有3种事件会导致CPU搁置正常的指令执行并强制将控制权转移到处理该事件的特殊代码。一种情况是系统调用，当用户程序执行ecall指令要求内核为它做一些事情时。另一种情况是例外：指令（用户或内核）做了一些非法的事情，例如除以零或使用无效的虚拟地址。第三种情况是设备中断，当设备发出需要注意的信号时，例如当磁盘硬件完成读取或写入请求时。

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。

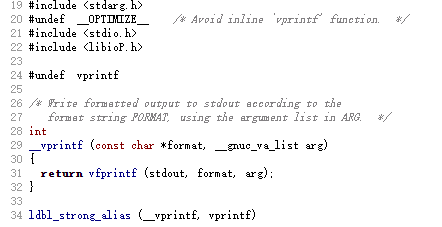
* printf执行过程
  + 从函数原型开始，函数原型在文件libc/libio/stdio.h中定义printf。



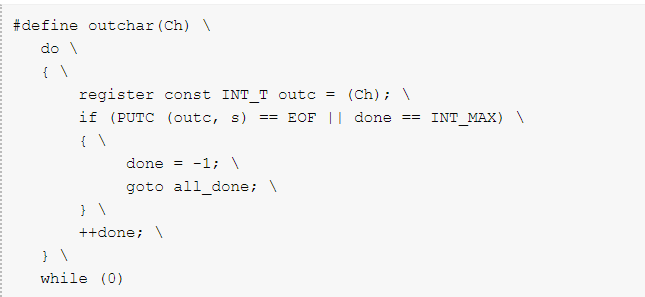
其函数实现如下：



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



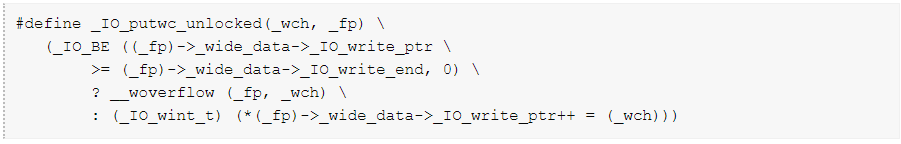
* + 在vfprintf定义文件中有个宏定义：



可以看到其依赖于PUTC也在同一文件中

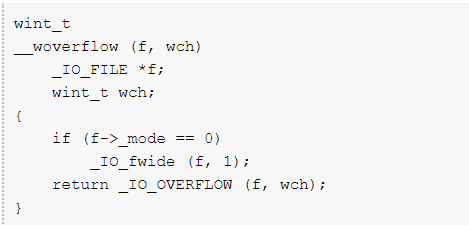


可以从libc/libio/libio.h获得IO\_putwc\_unlocked



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）。

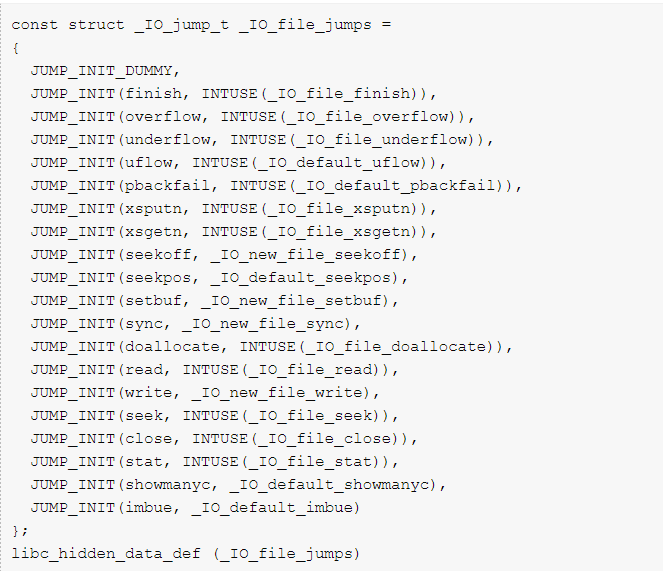
* + 查看 libc/libio/wgenops.c，会发现 :\_\_woverflow 的定义



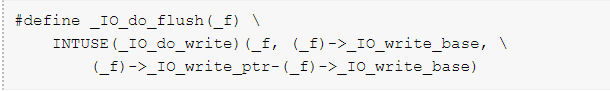
基本上，文件指针在GNU标准库中作为对象实现。它们具有数据成员和函数成员，可以使用宏的变体调用它们。在文件libc/libio/libioP.h中，会找到有关此技术的一些文档：JUMP。

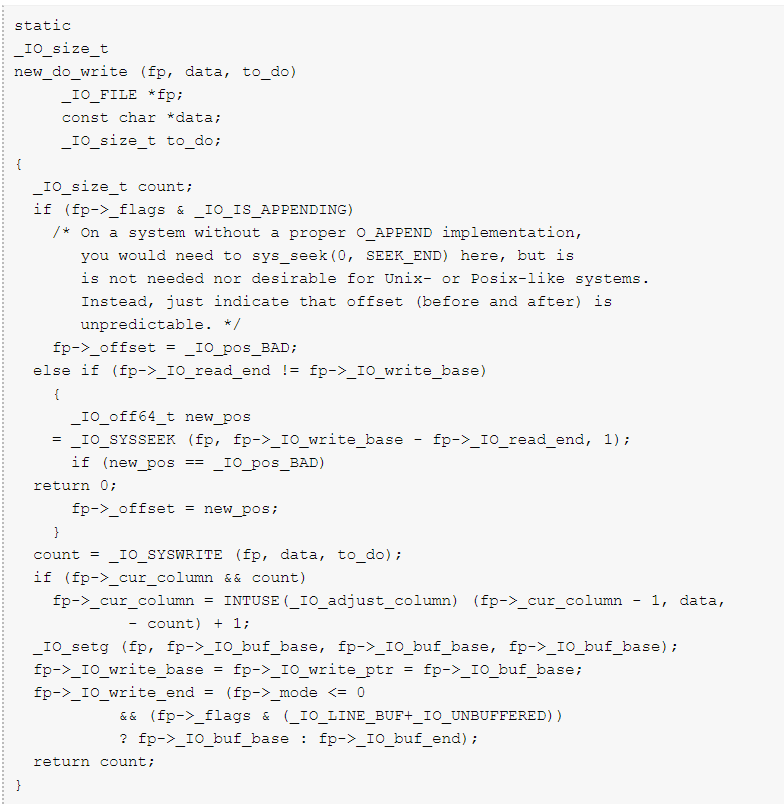
通过相关文件可以找到以下定义：



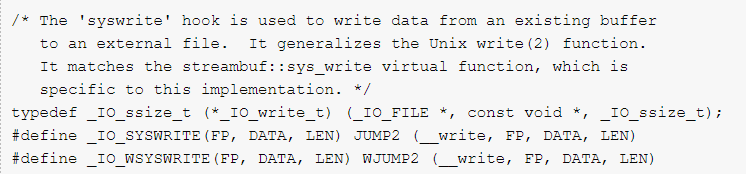


* + 源代码做了一堆更多的缓冲区操作，调用了\_IO\_do\_flush

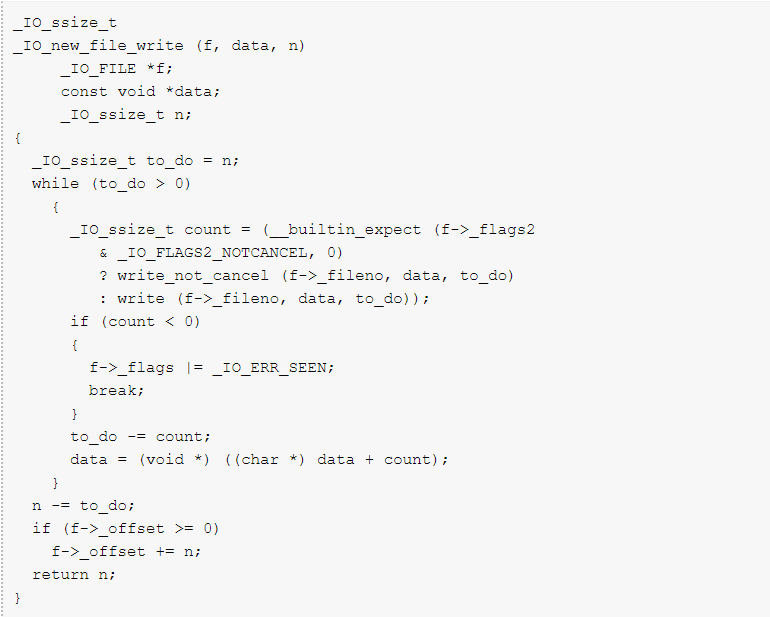




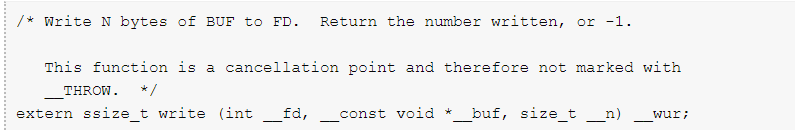
从中我们发现其调用了\_IO\_SYSWRITE



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



现在可以从libc/posix/unistd.h中找到write。



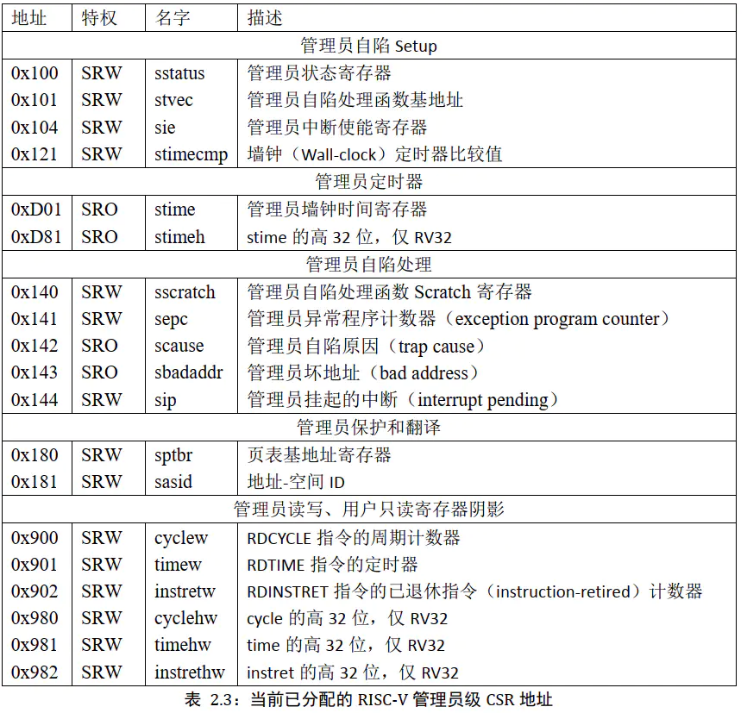
在GNU标准库中找到适用于Linux的write.c文件。相反，会发现以各种方式连接到操作系统写入功能的特定于平台的方法，所有这些都在libc/sysdeps/目录中。Write在 Linux如何做到这一点。有一个名为 sysdeps/unix/syscalls.list的文件，用于自动生成函数。表中相关数据为：write。这是一个系统调用的相关文件。

* 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



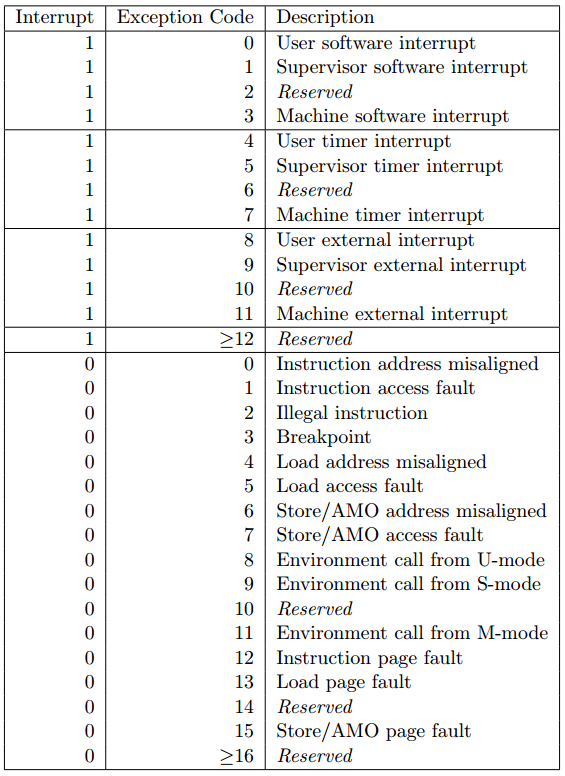
对上述部分ECALL与SRET涉及到的寄存器，提前在opType.vh头文件中定义地址，方便实现。

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。

* 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：

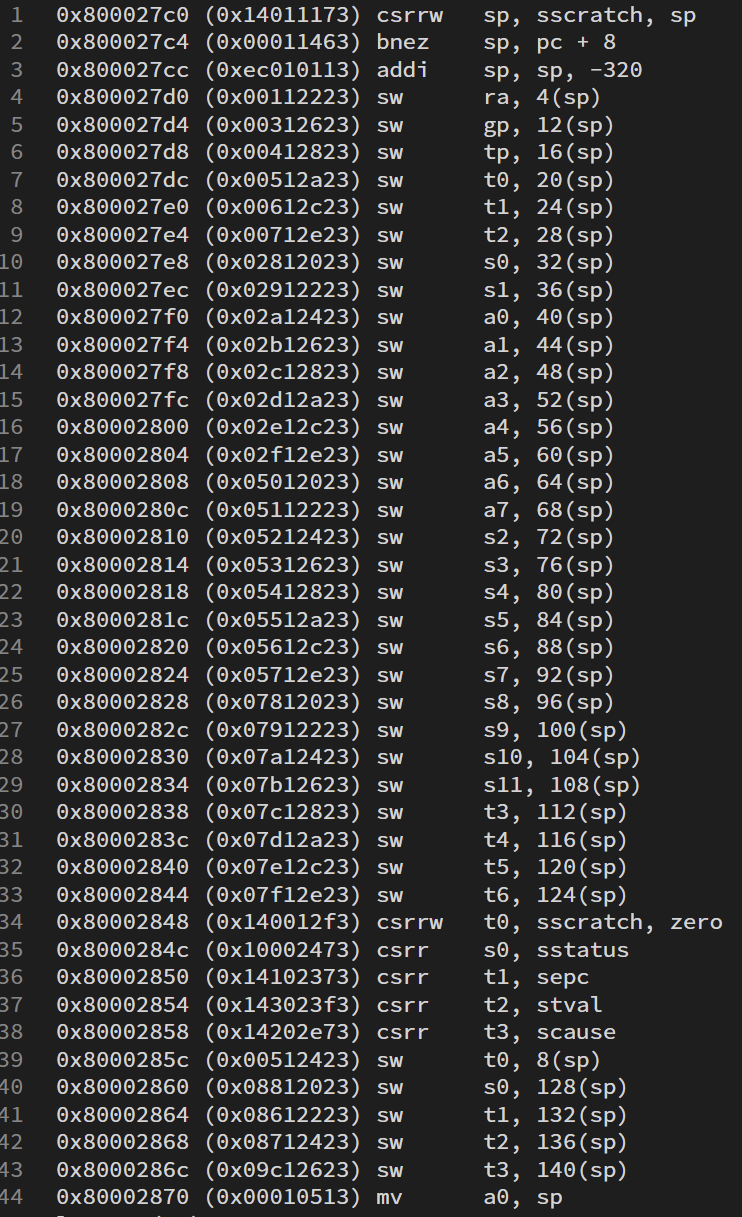
* + 更新mepc，mepc写入ECALL的下一条指令的地址。
  + 更新mcause，根据产生异常的类型更新mcause，异常类型的编码如下所示：



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

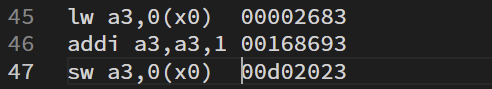
* 更新mtval，某些异常需要将异常相关的信息写入到mtval当中，我们假定没有相关的异常信息，即写入0即可。
* 更新mstatus，mstatus记录中断使能等信息，由于我们涉及到了从machine到supervisor级别的切换，需要将异常发生前的MIE保存到SPIE当中，将异常发生前所处的特权级（machine对应为1）保存到SPP中，SIE设为0。这意味着在硬件上，RISC-V是不支持嵌套中断的。若要实现嵌套中断，则只能通过软件的方式来实现。
* 跳转到mtvec中所定义的异常入口地址执行,初始时默认mtvec中的值已经指向了我们写的中断处理程序中。其中mtvec有两种模式，一种是直接模式，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。
* 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。



* 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。



* 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。

* 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：

* + 从mepc中定义的地址执行，恢复到异常发生前的程序流执行。
  + 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。

### 3、小结

5级流水线基于目标ISA，能实现各指令基本功能。除此之外，实验设计的流水线对写后读的数据冲突通过停顿和冲刷流水线来消除，对分支默认为不跳转的预测，分支跳转时冲刷流水线保证执行结果的正确性，对trap功能通过ECALL调用中断处理、切换特权级、保护现场、执行访存操作的“假”中断处理程序、恢复现场和SRET返回，模仿和实现了一般trap的流程。

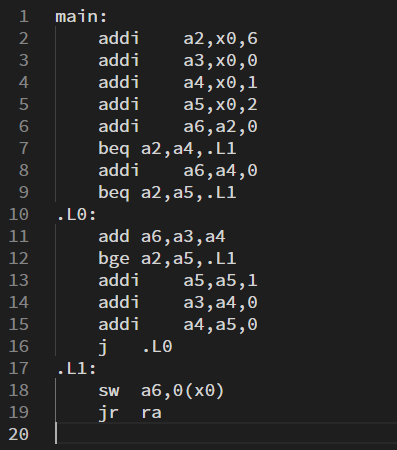
## （三）测试程序

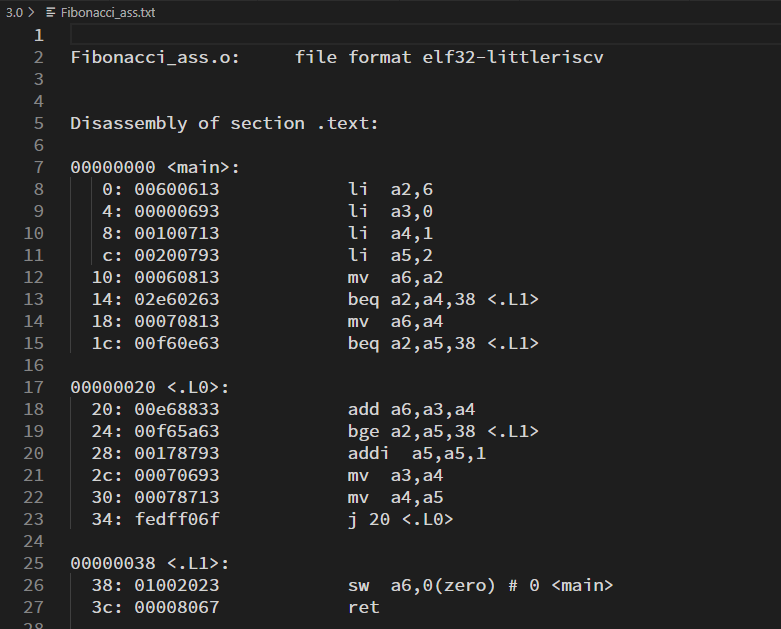
### 1、4个测试程序

测试程序共4个，分别为手写RV汇编和含printf的C语言的Fibbonaci和Bubble Sort程序。

#### （1）Fibonacci by RV

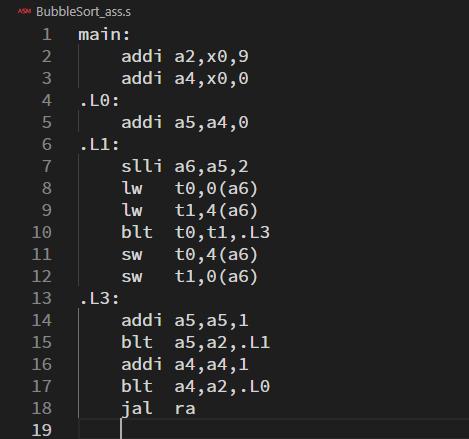
使用RV汇编人工编写的Fibonacci程序和通过工具链编译后反汇编得到的汇编指令：

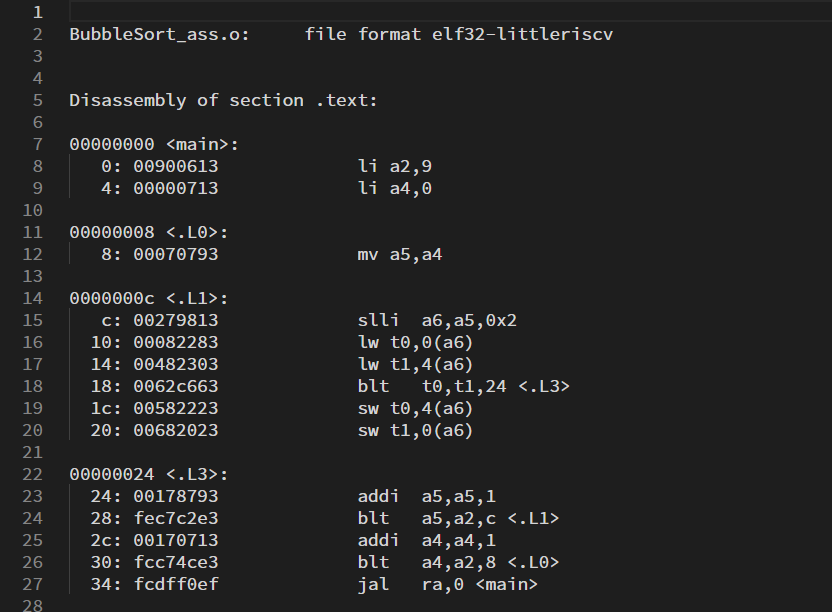




#### （2）Bubble Sort by RV

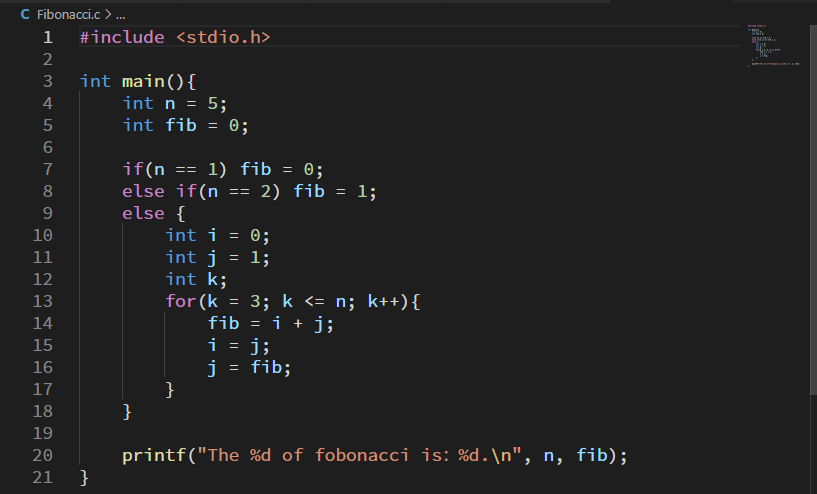
使用RV汇编人工编写的Bubble Sort程序和通过工具链编译后反汇编得到的汇编指令：

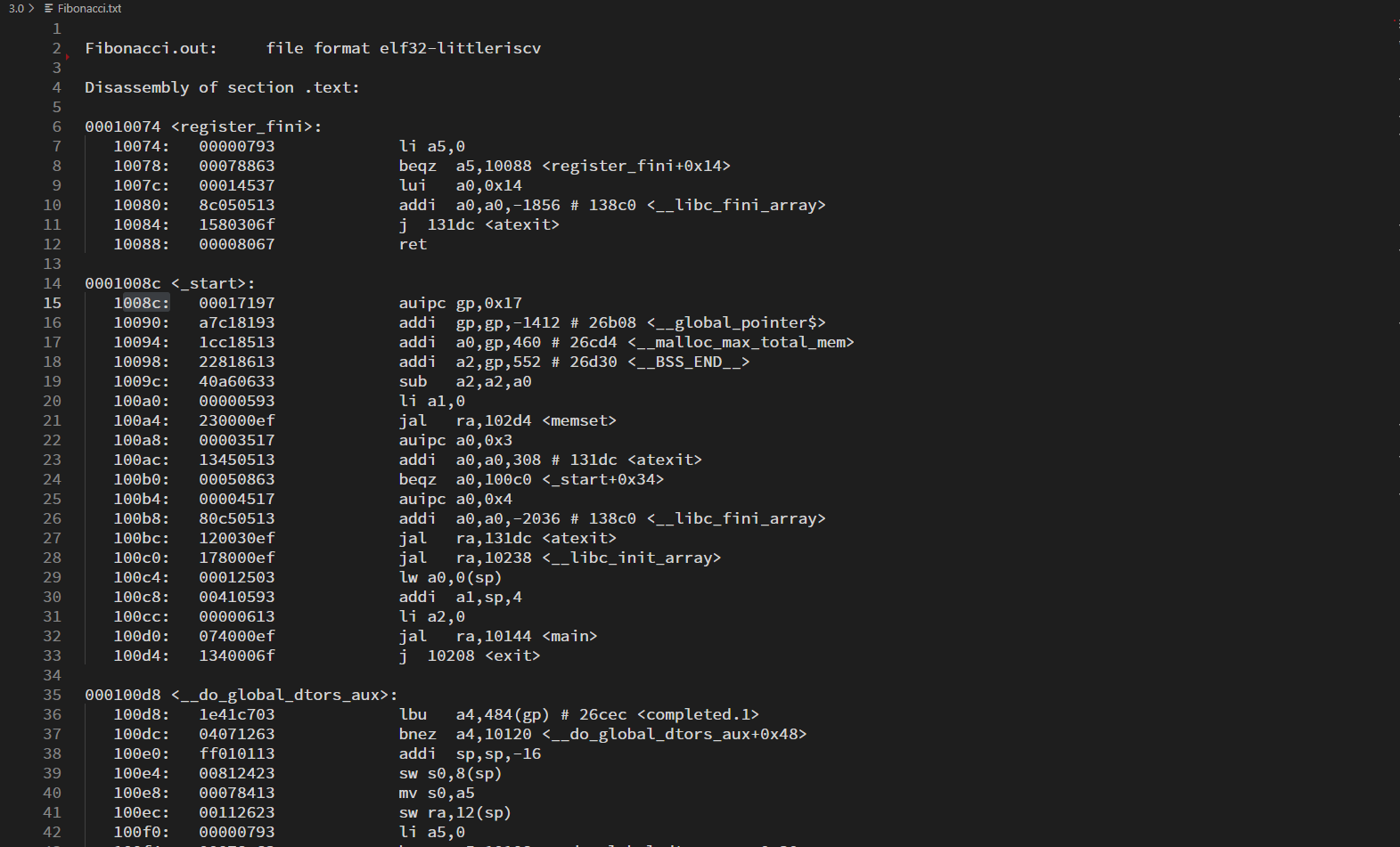




#### （3）Fibonacci by C

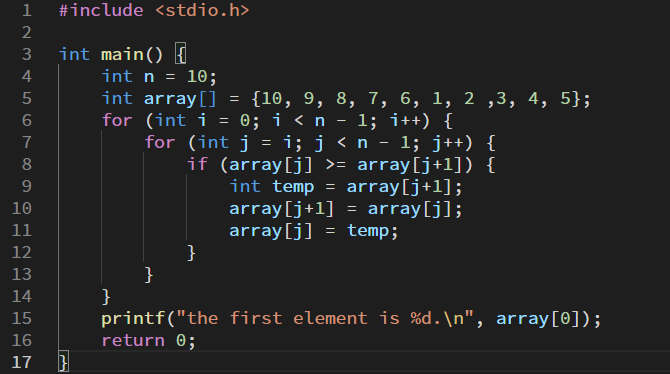
使用c语言编写的带printf语句的Fibonacci程序及其编译反汇编后的汇编程序：





#### （4）Bubble Sort by C

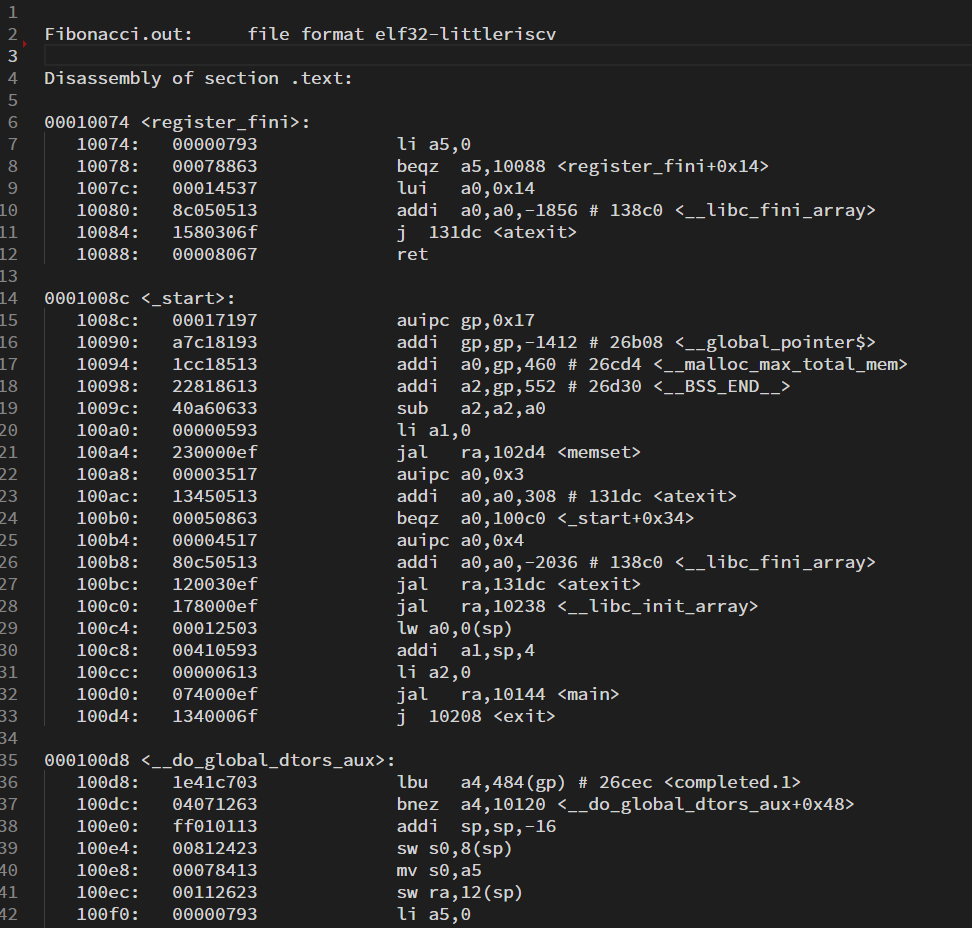
使用c语言编写的带printf语句的Bubble Sort程序及其编译反汇编后的汇编程序：





### 2、ELF可执行文件的转换

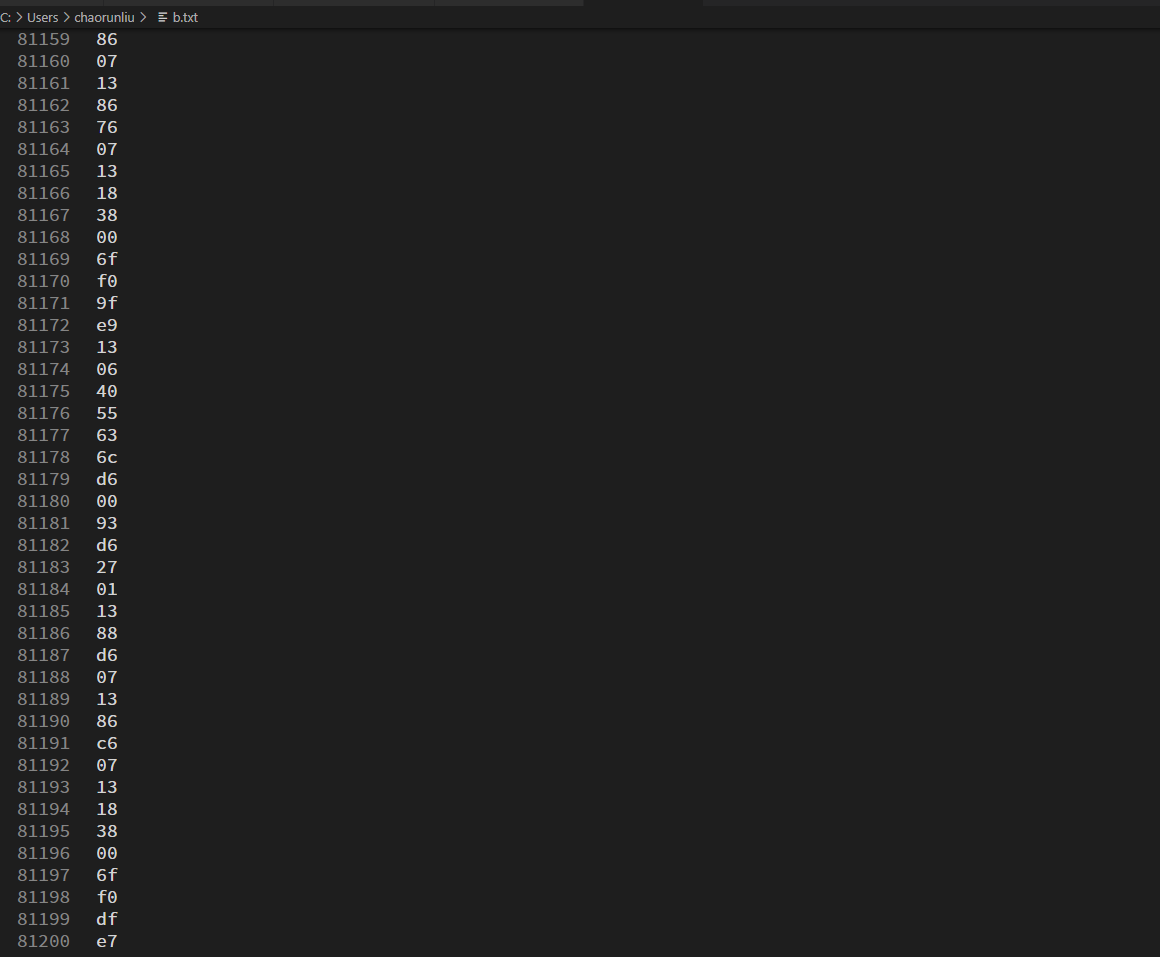
由于riscv32-unknown-elf-objdump仅可转换elf可执行文件为汇编代码，但是丢失了elf中数据段的内容，如下图。



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：



### 3、模拟器执行结果

#### （1）模拟器介绍

由于带有printf指令的反汇编代码共有2w多行，而我们实现的五级流水线并没有可以 printf的外设，因此如何验证最终结果中的寄存器数值正确是一个重要问题。我们可以使用riscv指令集模拟器执行相关代码并查看最终寄存器状态来确实最终执行结果是否正确，因此我们选择spike（https://github.com/riscv/riscv-isa-sim）作为riscv指令集同时由于我们仅需要模拟程序运行，因此需要使用riscv-pk（https://github.com/riscv/riscv-pk）作为代理内核。

模拟器安装：首先将riscv-isa-sim和riscv-pk从github上clone到本地

* 安装riscv-pk：

cd riscv-pk mkdir build cd build

../configure –prefix=/home/lcr/riscv --host=riscv32-unknown-elf –with- arch=rv32imfd –-with-abi=ilp32d

make

make install

* 安装 riscv-isa-sim：

apt-get install device-tree-compiler cd riscv- isa-sim

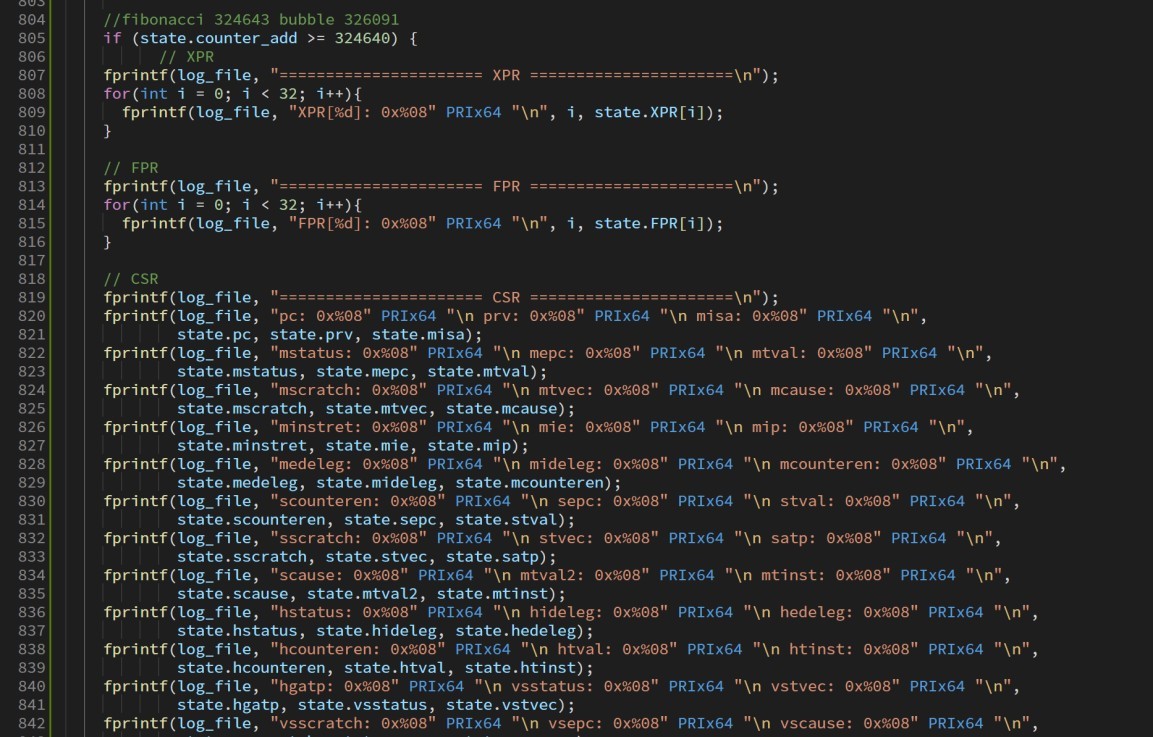
mkdir build cd build

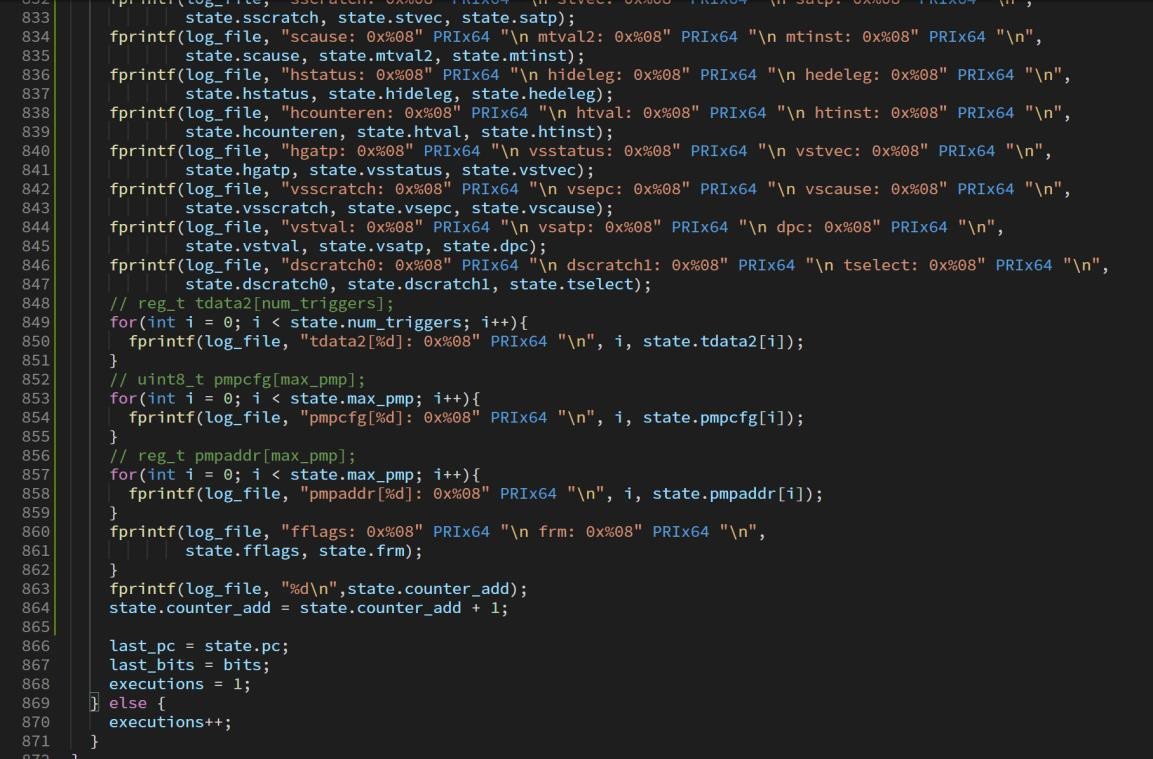
../configure –prefix=/home/lcr/riscv --with-isa=RV32IMFD make

make install

#### （2）修改模拟器输出

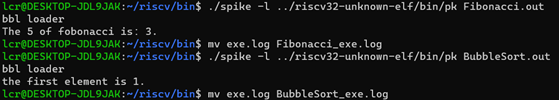
由于模拟器仅在交互模式下才可以查看整数和浮点寄存器，无法查看csr寄存器状态， 因此我们通过修改riscv-isa-sim的riscv文件下processor.cc和processor.h文件向日志文件中增加输出寄存器状态。核心代码如下：





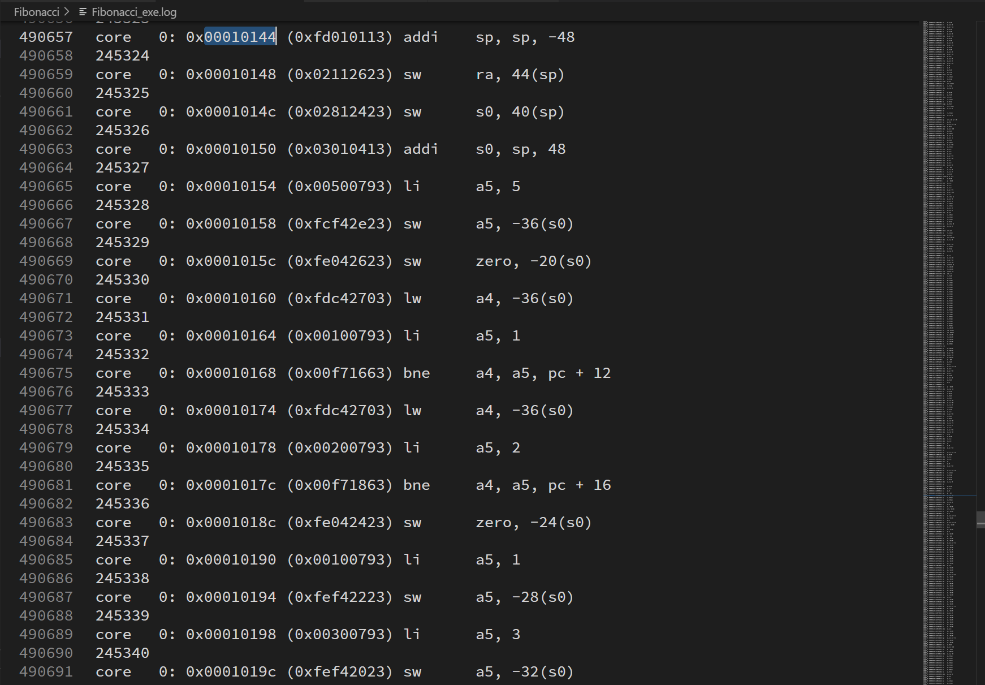
#### （3）模拟器执行结果

模拟器的执行结果用来对照5级流水线执行结果，使用模拟器运行带printf的Fibonacci和Bubble Sort的测试程序，执行结果如下所示：

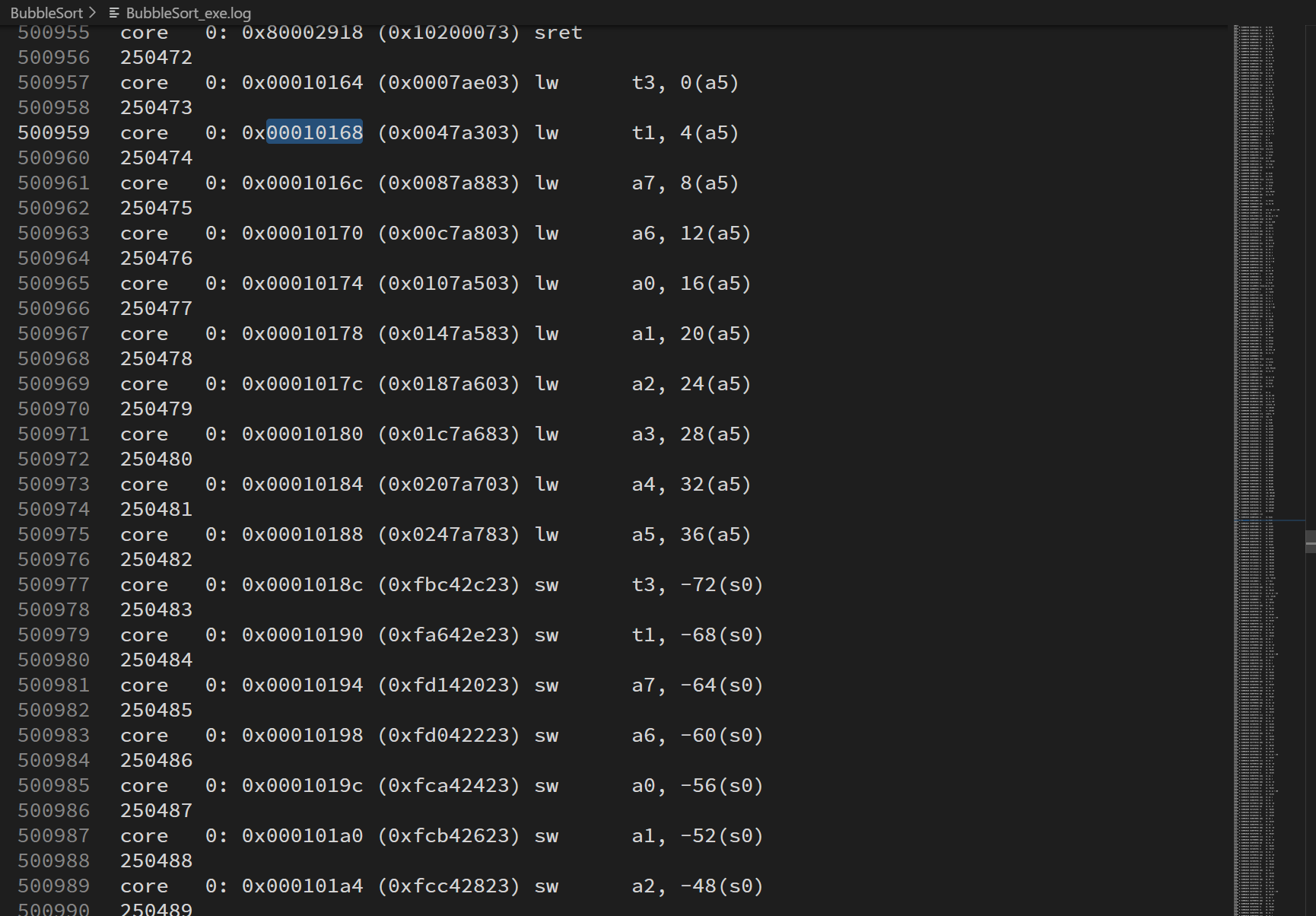


模拟器的输出日志能给出指令执行顺序以及指令执行后寄存器状态等。截取两个测试程序main函数部分日志结果，详见test\_program/Fibonacci文件夹下Fibonacci\_exe.log和test\_program/BubbleSort文件夹下BubbleSort\_exe.log文件。

Fibonacci程序



Bubble Sort程序



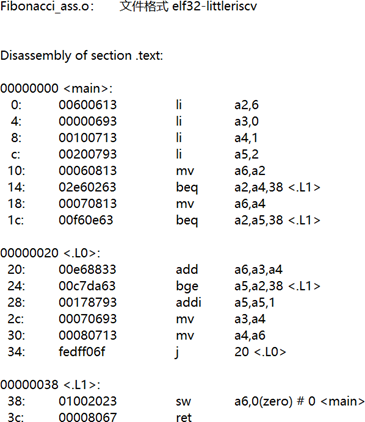
### 4、小结

实验通过执行4个测试程序检验5级流水线功能等，对于手写的RV汇编而言，代码规模小且易读，能检查每条指令执行结果。但对于带printf的由C语言编译而来的汇编程序，代码规模巨大，且涉及到验证结果方法的问题，因此选择通过spike模拟器执行C语言的汇编程序，通过与模拟器的执行结果对比，来验证5级流水线实验结果的正误。其中，为了便于查看模拟器每条指令执行结果及寄存器状态，修改了模拟器的输出代码等。

# 四、实验结果

## （一）Fibonacci by RV

使用人为手写RV汇编指令的Fibonacci测试程序，代码如下：

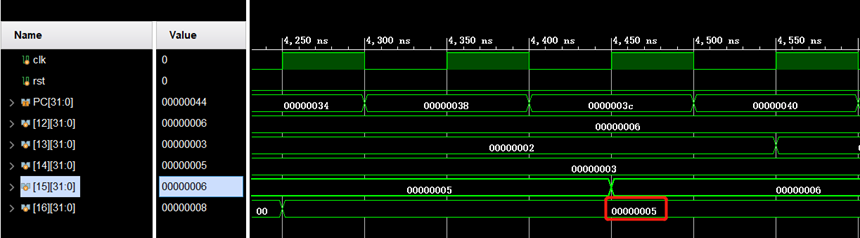


其中，寄存器a2存储需要取第几个Fibonacci值，a3、a4分别对应i-2和i-1的值，a5为计数器用于计数i，Fibonacci第i个值存在a6当中。

前4条指令为a2至a5寄存器赋值，中间结果如下：

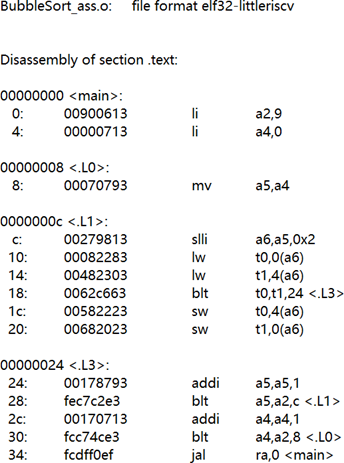


最终计算出来第6个Fibonacci值为5，a6中显示：



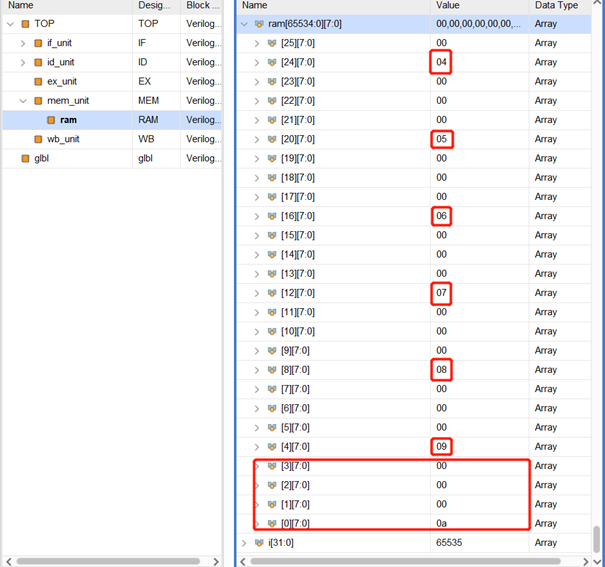
## （二）Bubble Sort by RV

使用人为手写 RV 汇编指令的 Bubble Sort 测试程序，代码如下：

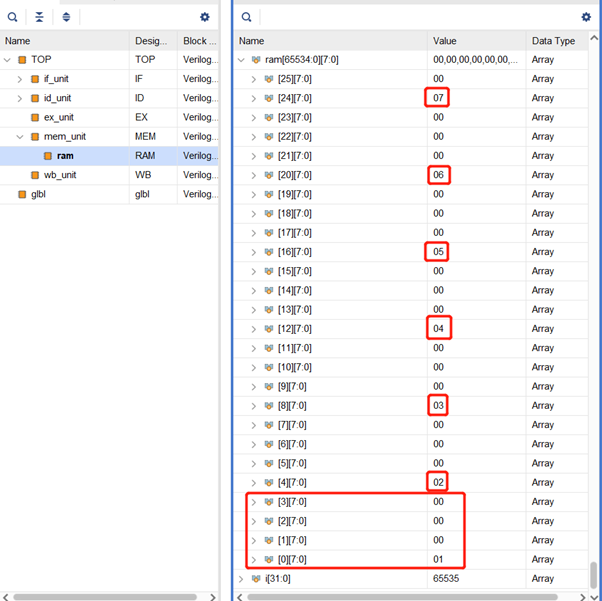


其中.L0为外层循环，.L1为内层循环。a4为外层循环变量，从0自增一，直至不小于数组大小（a2）。a5对应内层循环控制变量，初始为a4值，自增直至不小于a2。

初始时，数据存储器 RAM 中的值如下所示，按从小到大的地址分别对应从大到小的数组值：



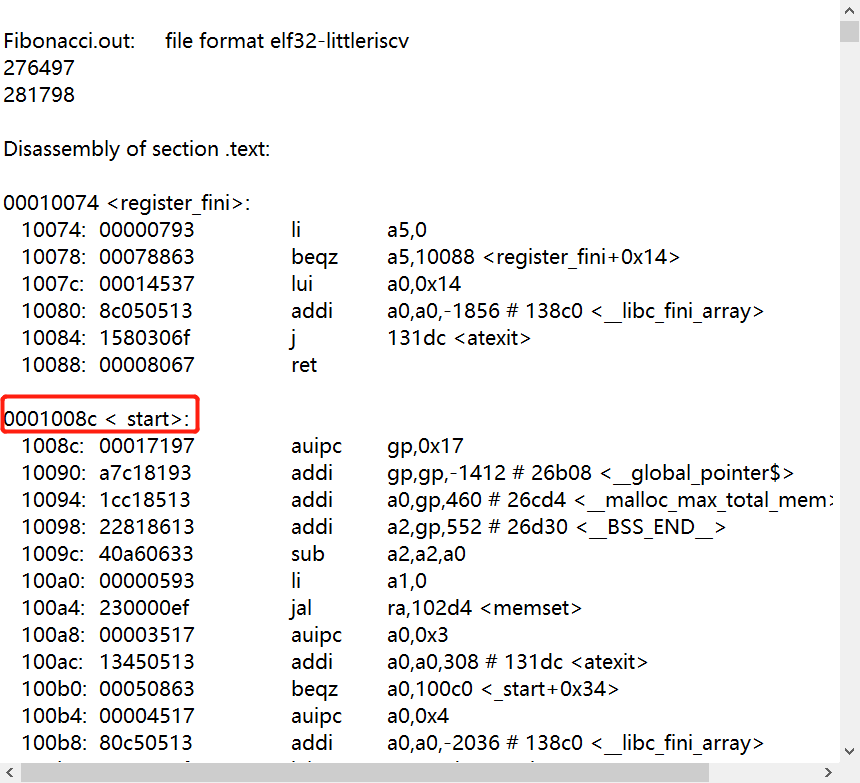
冒泡排序后，数据存储器RAM中的值如下所示，可发现经过排序后，数组被完全倒置：



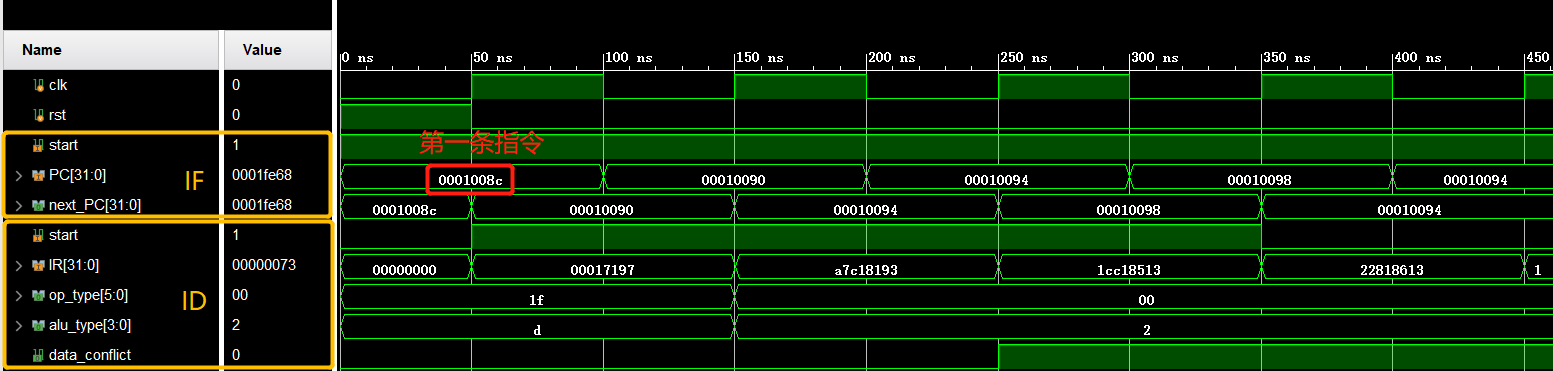
## （三）Fibonacci by C

**1、程序与数据加载**

将机器码Fibonacci.coe文件写入指令存储器prgROM中，将Fibonacci\_mem.txt读入数据存储器ram中。PC初始化为0x0001008c，如下图所示，<start>为程序执行入口：



5级流水线启动运行如下所示（左侧信号按不同流水段分类）：



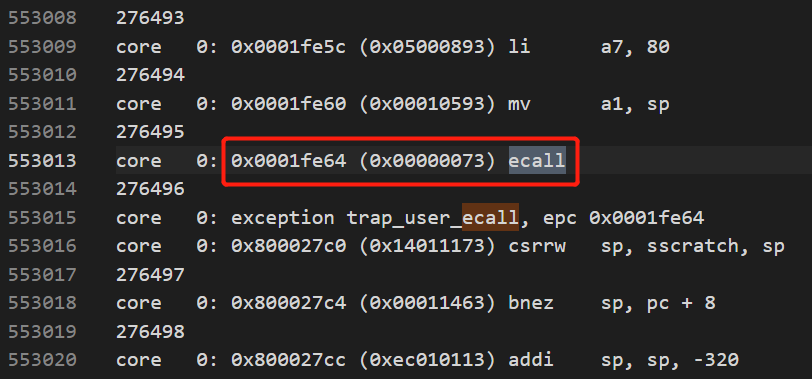
**2、运行前十个周期**

结果如下图所示，由于前3条指令存在对gp寄存器的数据冲突，需停顿等待，第10个周期时，PC此前已检测到第4条指令（PC为0x00010098）的数据冲突，冲刷掉IF对第5条指令（PC为0x0001009c）的取值，保持在第4条指令处

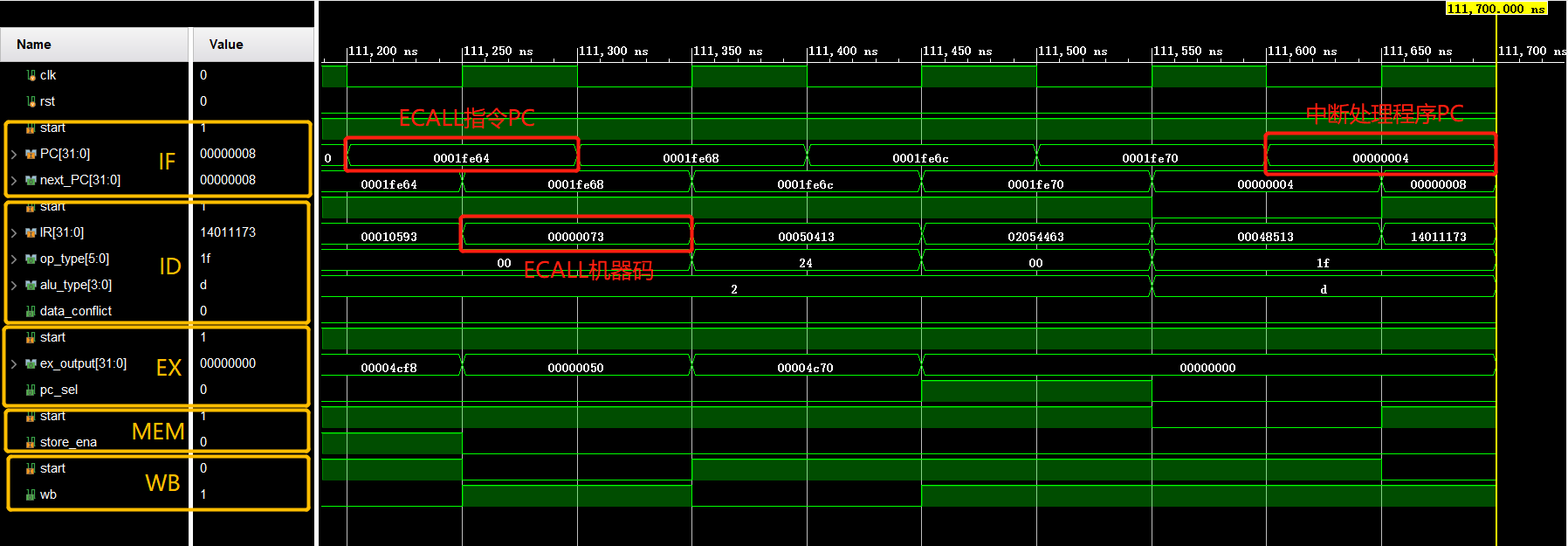


**3、运行至ECALL**

模拟器运行Fibonacci测试程序时第一次调用ECALL时指令PC（0x0001fe64）及指令（0x00000073）结果如下图所示：

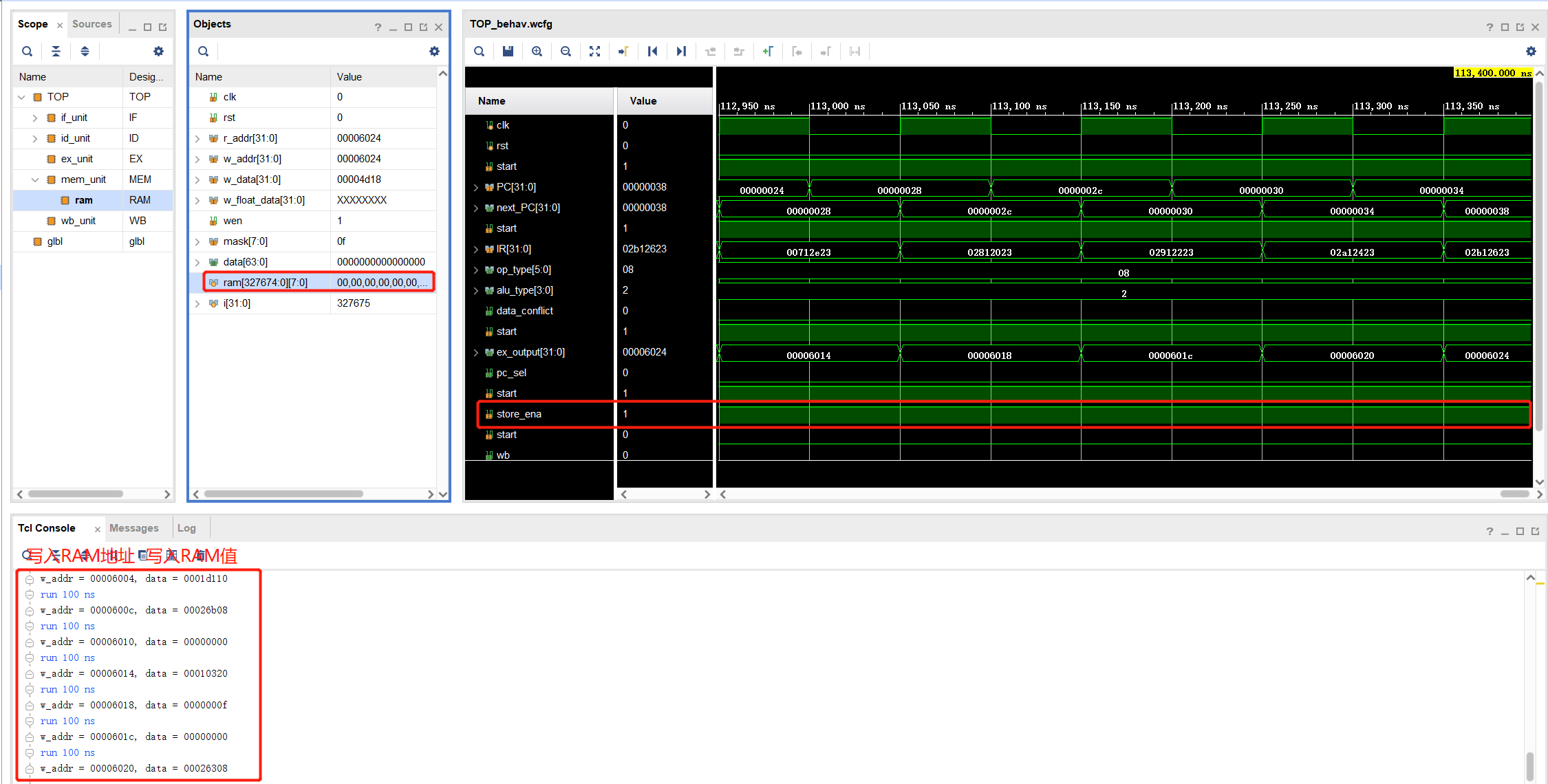


5级流水线运行至第一次系统调用ecall如下图所示：



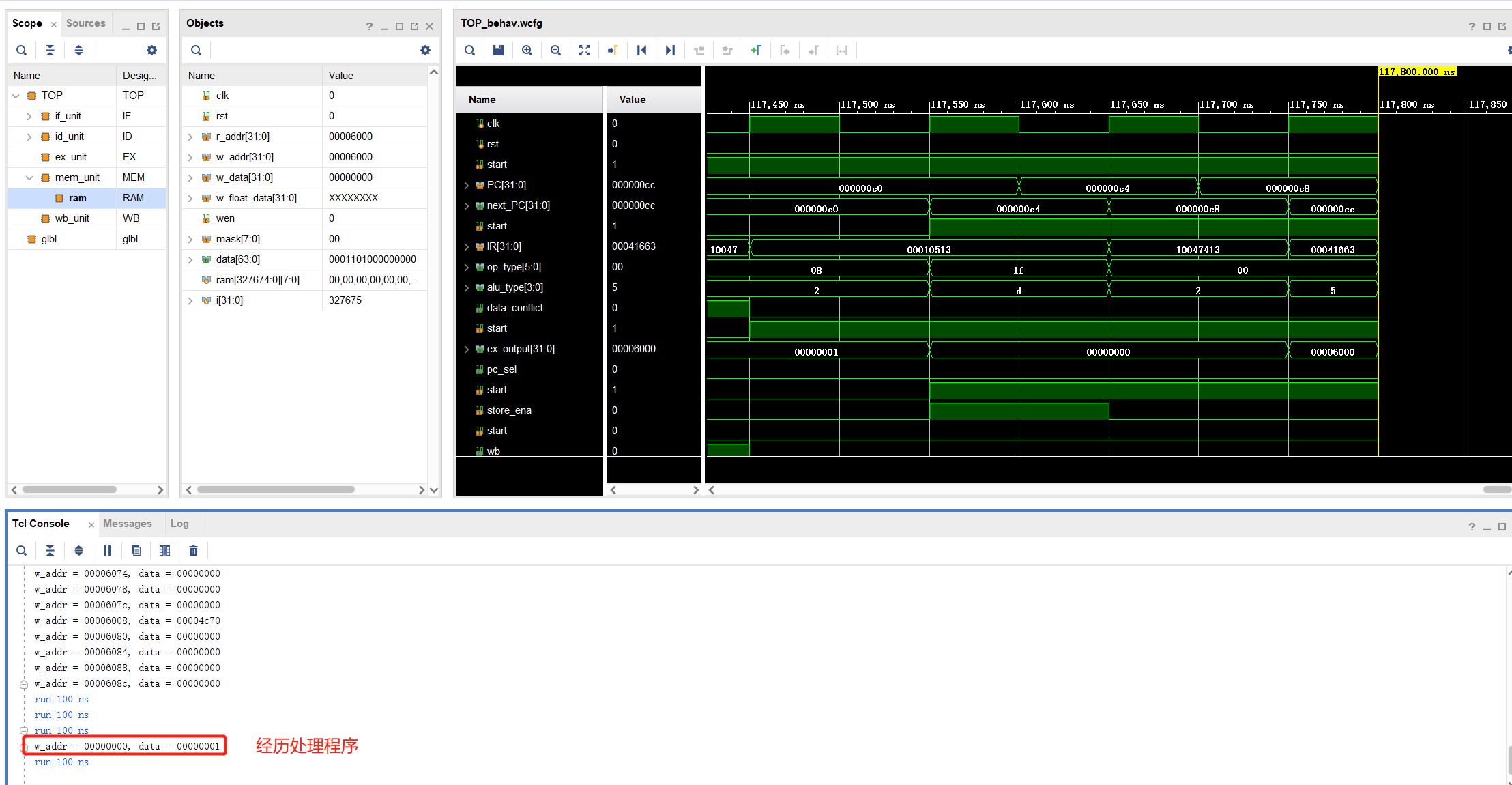
**4、trap程序保护现场**

由于RAM数组过大无法显示数据内容，通过display输出显示，如图所示为保护现场过程，不断向内存中写入要保护的寄存器数据



**5、Trap处理程序**

上文已提及过，由于真正的处理程序需要进行通过外设等其他处理过程，本实验设定“假”处理程序，即向地址为0的内存单元中写入数据1，以表示经历了该处理程序过程。如下图所示为经历该处理程序：



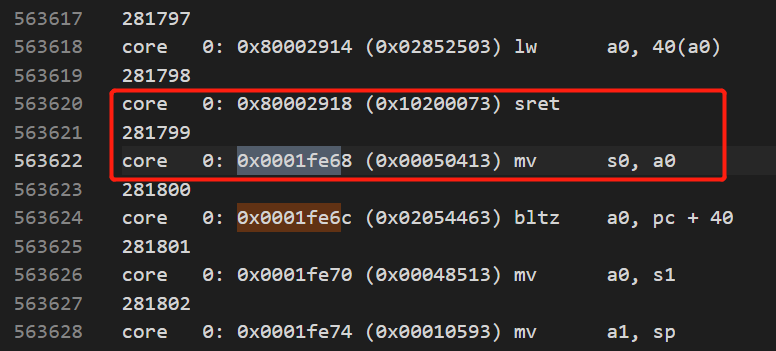
**6、恢复现场**

经过处理程序后恢复现场，即将保护现场时写入内存中的数据写回对应的寄存器。如下图所示，执行至恢复现场时，指令译码出的opType为5对应op为opLW，即LoadWord指令：

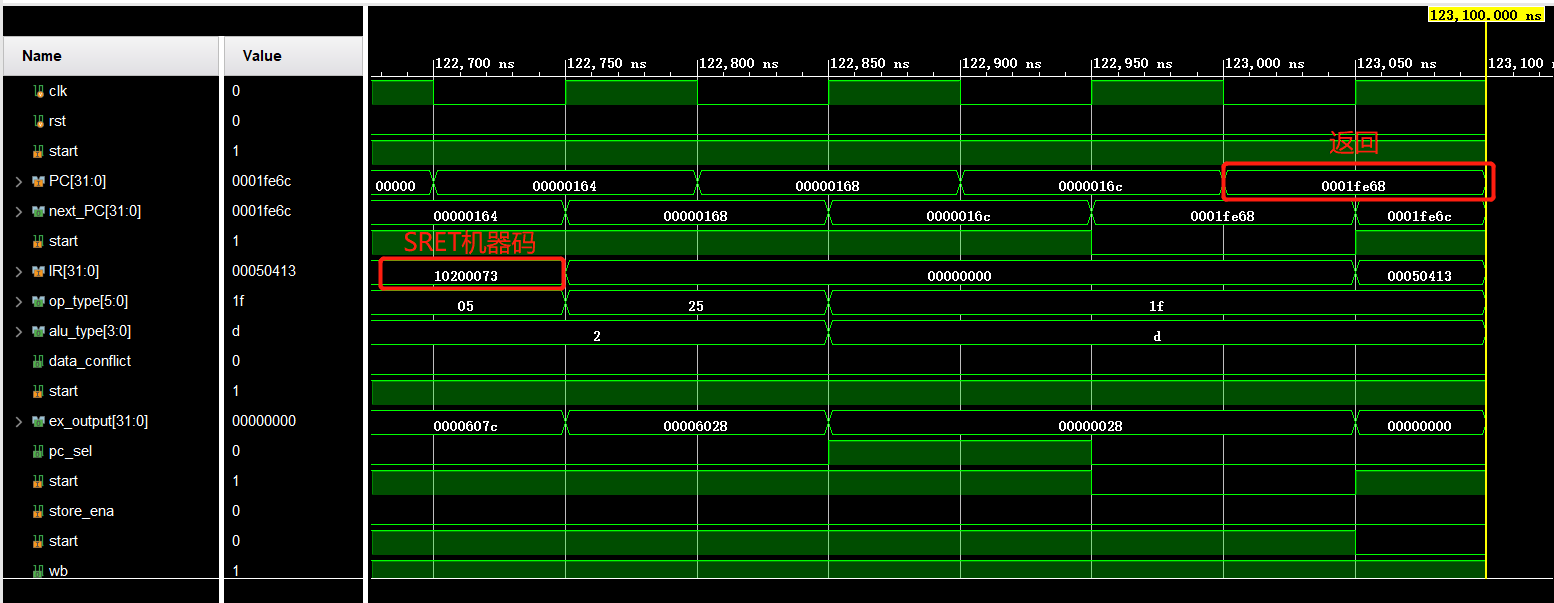


**7、返回进入trap前地址**

Trap处理结束后，返回调用ECALL指令PC的后一指令，模拟器执行结果如下图所示，sret指令结束后返回了原ECALL指令后的PC为0x0001fe68指令：

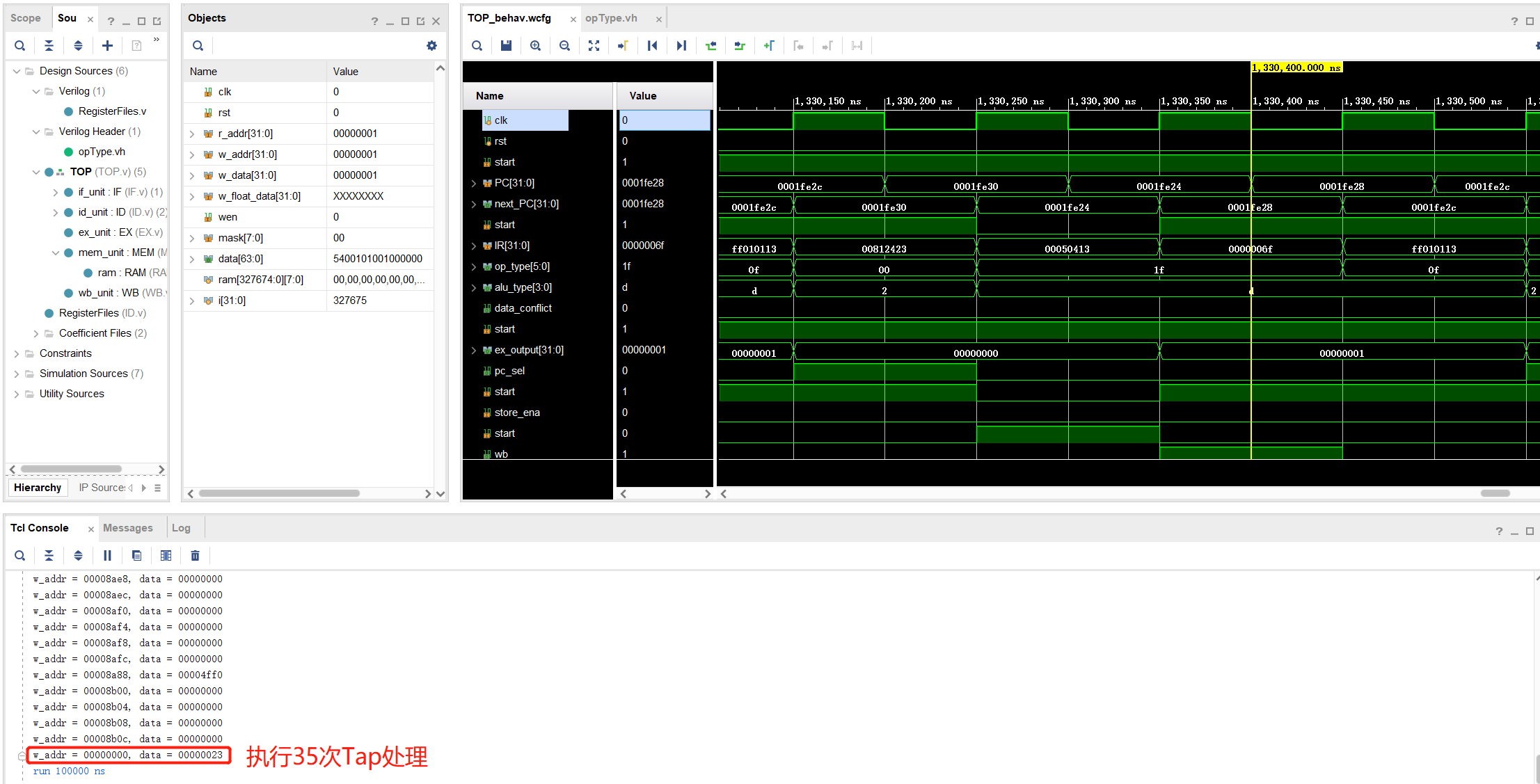


流水线执行结果如下图所示，经过SRET指令跳转后，PC指向了原ECALL指令的后一条指令，与模拟器结果相同：



**8、执行完成**

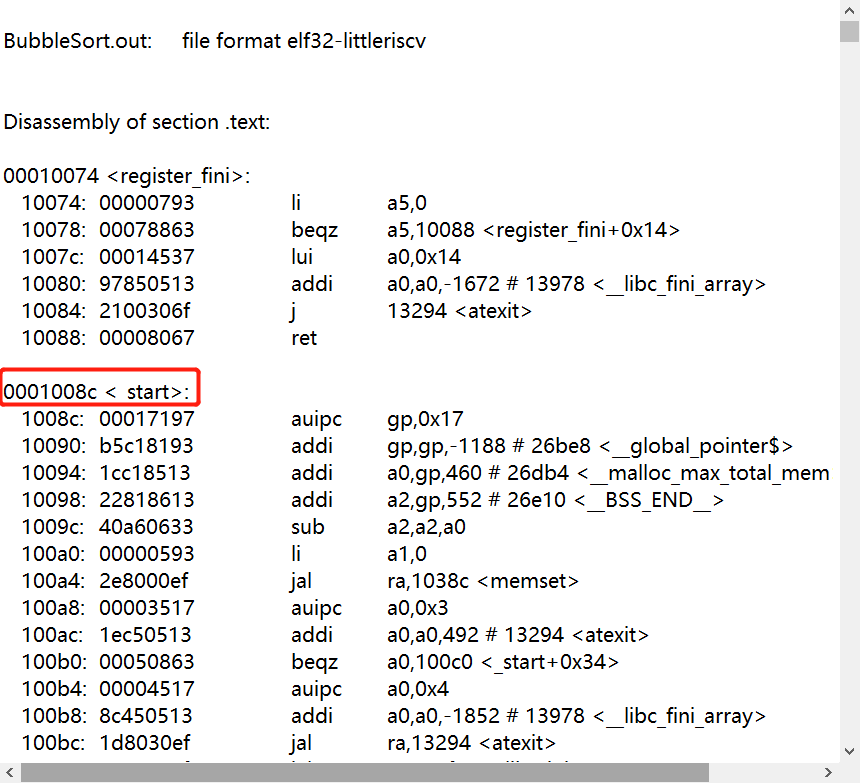
通过向地址为0的内存单元不断自加1，表明执行trap次数，最终内存单元值为35（16进制表示为23），表明整个程序经过35次trap处理。



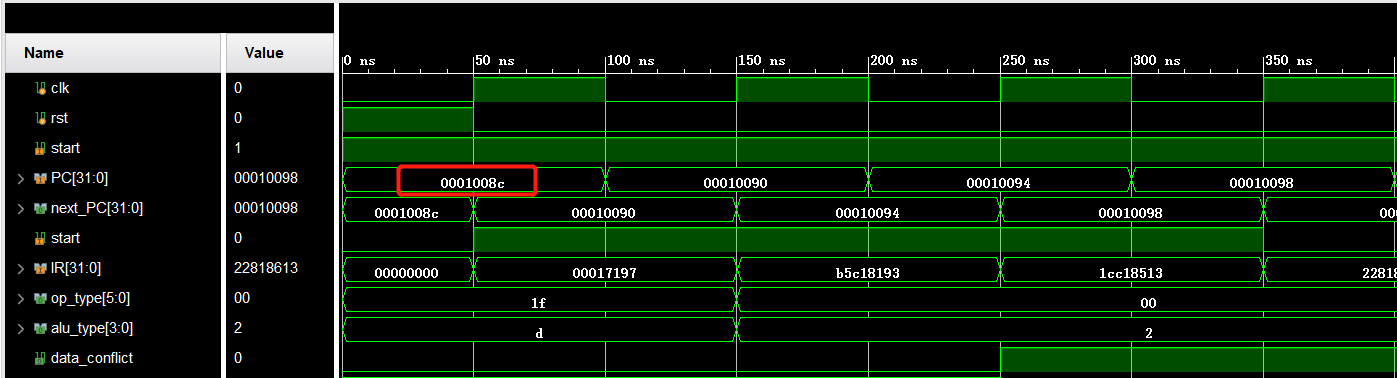
## （四）Bubble Sort by C

**1、程序与数据加载**

将机器码BubbleSort.coe文件写入指令存储器prgROM中，将BubbleSort\_mem.txt读入存储器ram中，PC设置为<start>地址作为程序执行入口地址（0x0001008c）



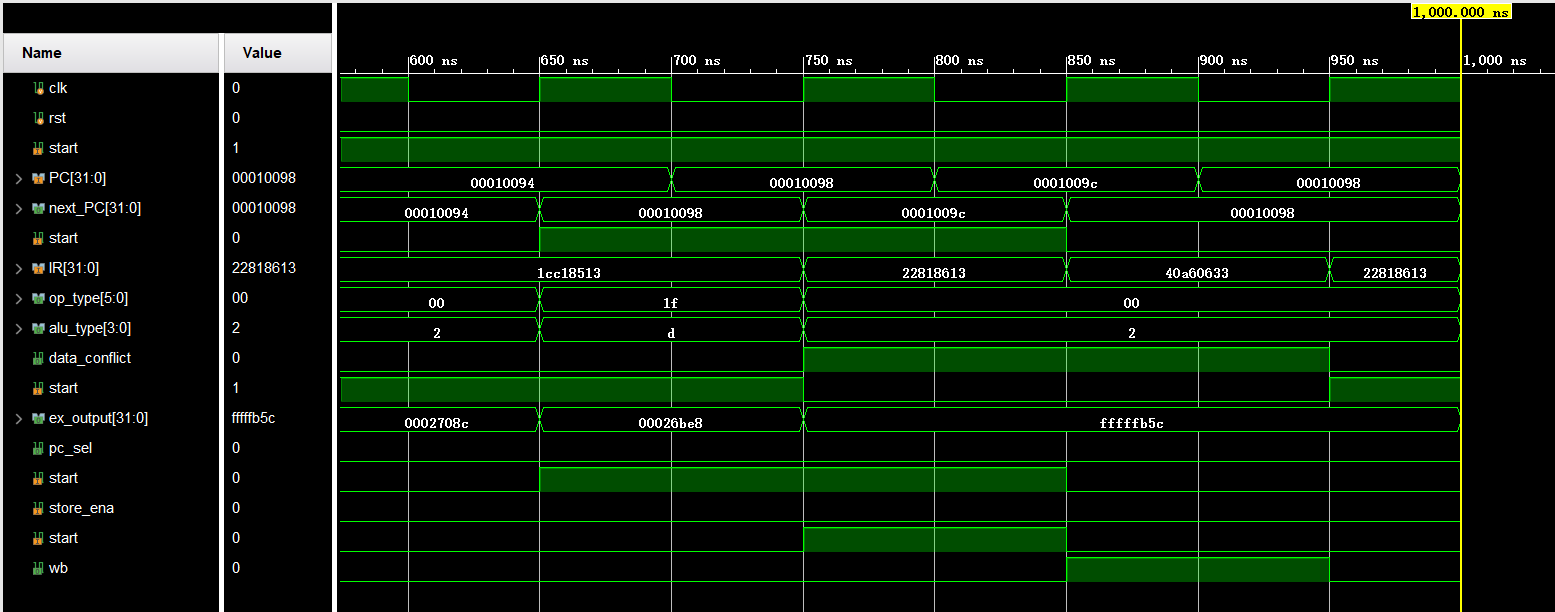
5级流水线程序执行启动如下图所示：



**2、运行前十个周期**

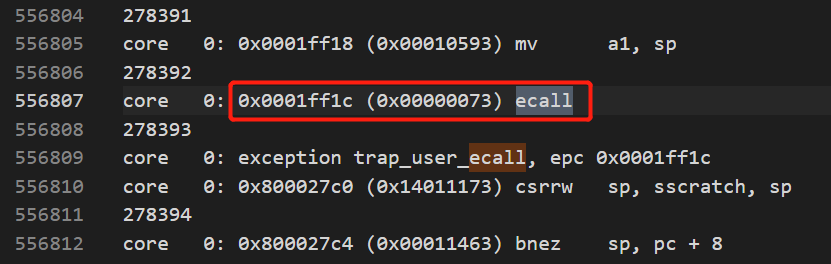
前10个周期运行结果如下图所示：

不难发现BubbleSort从<start>开始的指令及执行结果与Fibonacci相同，直到进入<main>，两者加载的指令是相同的。



**3、运行至ECALL**

模拟器运行BubbleSort测试程序时第一次调用ECALL时指令PC（0x0001ff1c）及指令（0x00000073）结果如下图所示：

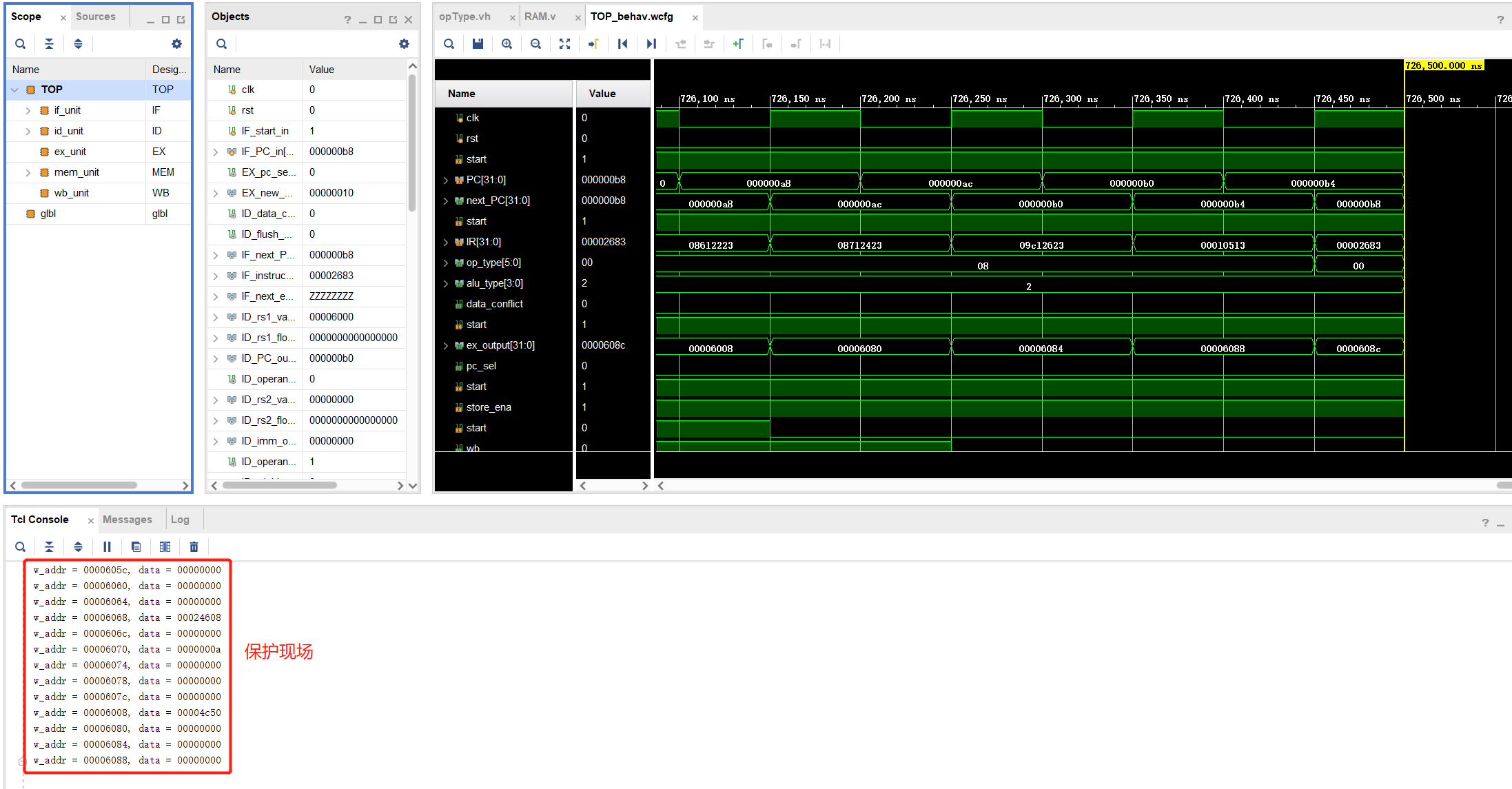


5级流水线运行至第一次系统调用ecall如下图所示：



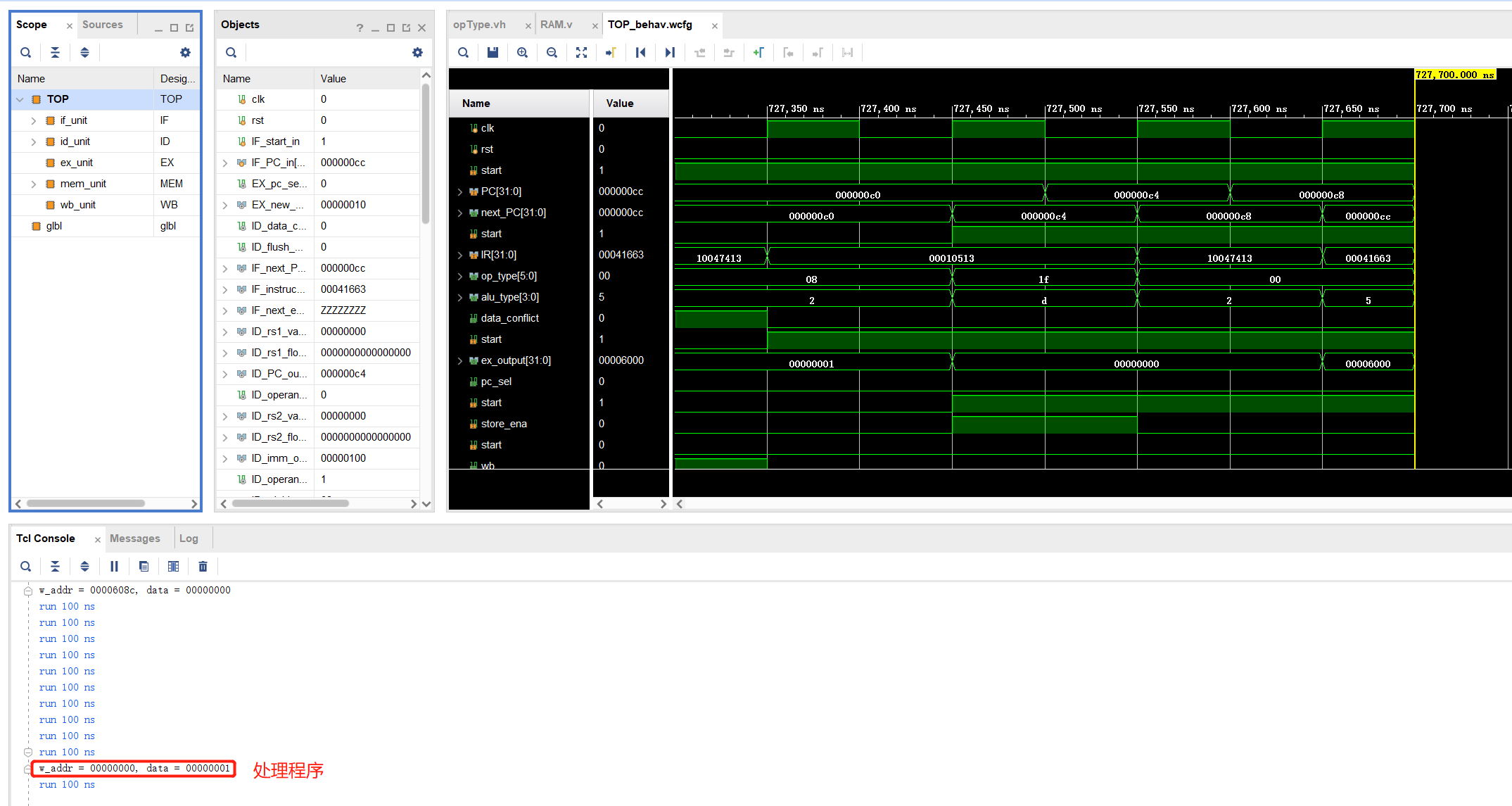
**4、trap程序保护现场**

通过display输出显示，如图所示为保护现场过程，不断向内存中写入要保护的寄存器数据



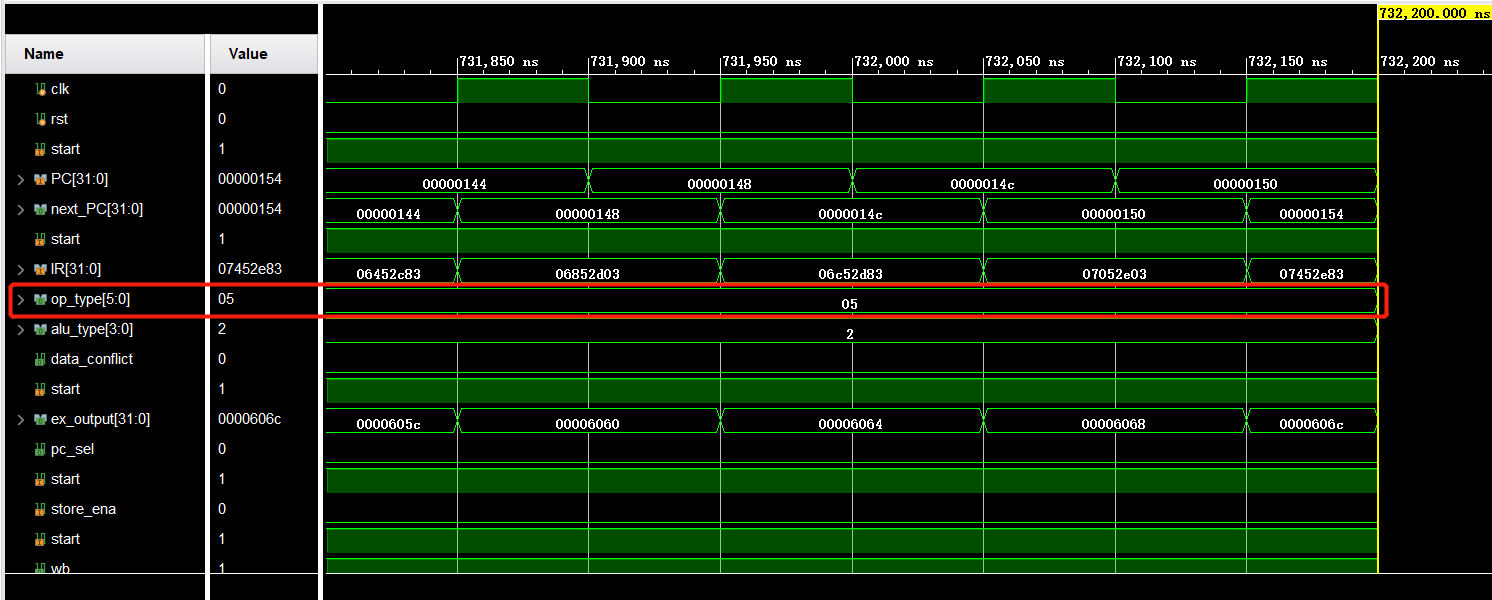
**5、Trap处理程序**

与Fibonacci处理程序一致，本实验设定“假”处理程序，即向地址为0的内存单元中写入数据1，以表示经历了该处理程序过程。如下图所示为经历该处理程序：



**6、恢复现场**

经过处理程序后恢复现场，即将保护现场时写入内存中的数据写回对应的寄存器。如下图所示，执行至恢复现场时，指令译码出的opType为5对应op为opLW，即LoadWord指令：

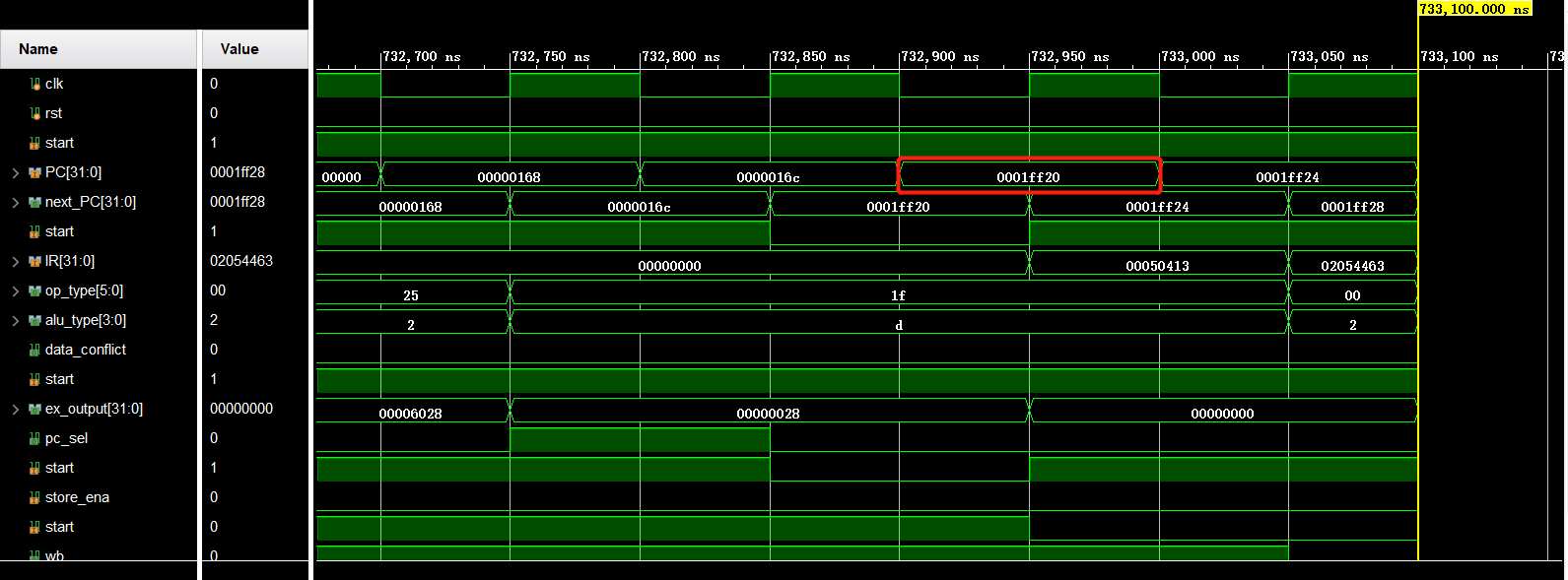


**7、返回进入trap前地址**

Trap处理结束后，返回调用ECALL指令PC的后一指令，模拟器执行结果如下图所示，sret指令结束后返回了原ECALL指令后的PC为0x0001ff20指令：

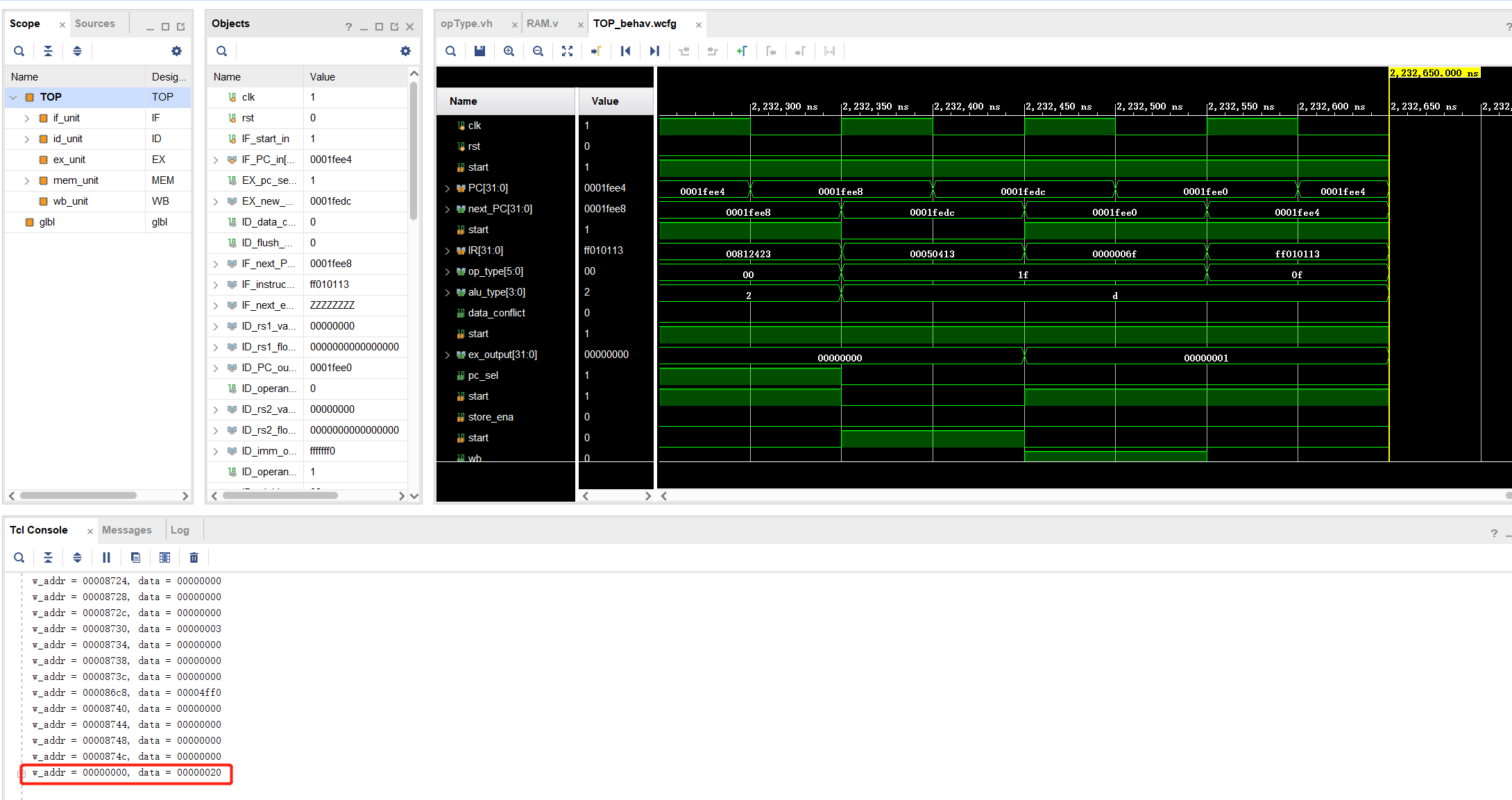


流水线执行结果如下图所示，经过SRET指令跳转后，PC指向了原ECALL指令的后一条指令，与模拟器结果相同：



**8、执行完成**

通过向地址为0的内存单元不断自加1，表明执行trap次数，最终内存单元值为32（16进制表示为20），表明整个程序经过32次trap处理。

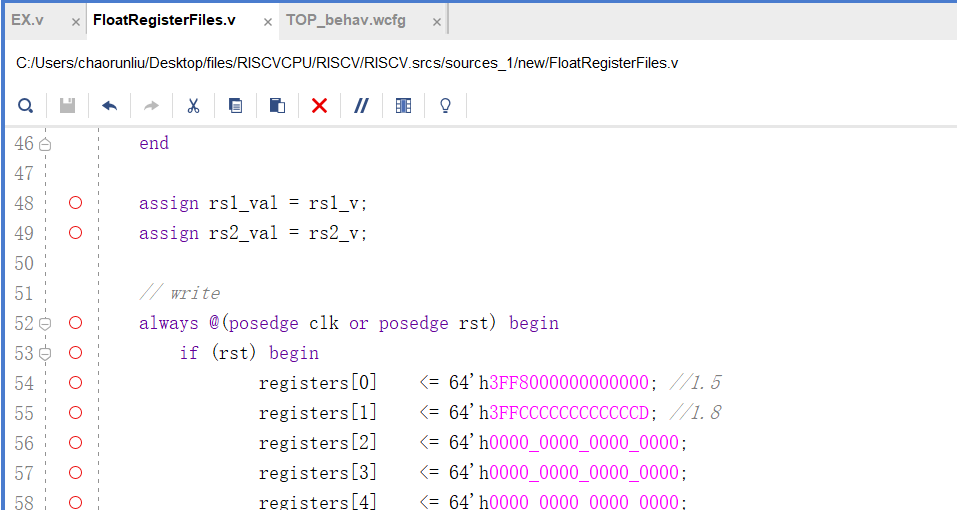


## （五）浮点测试

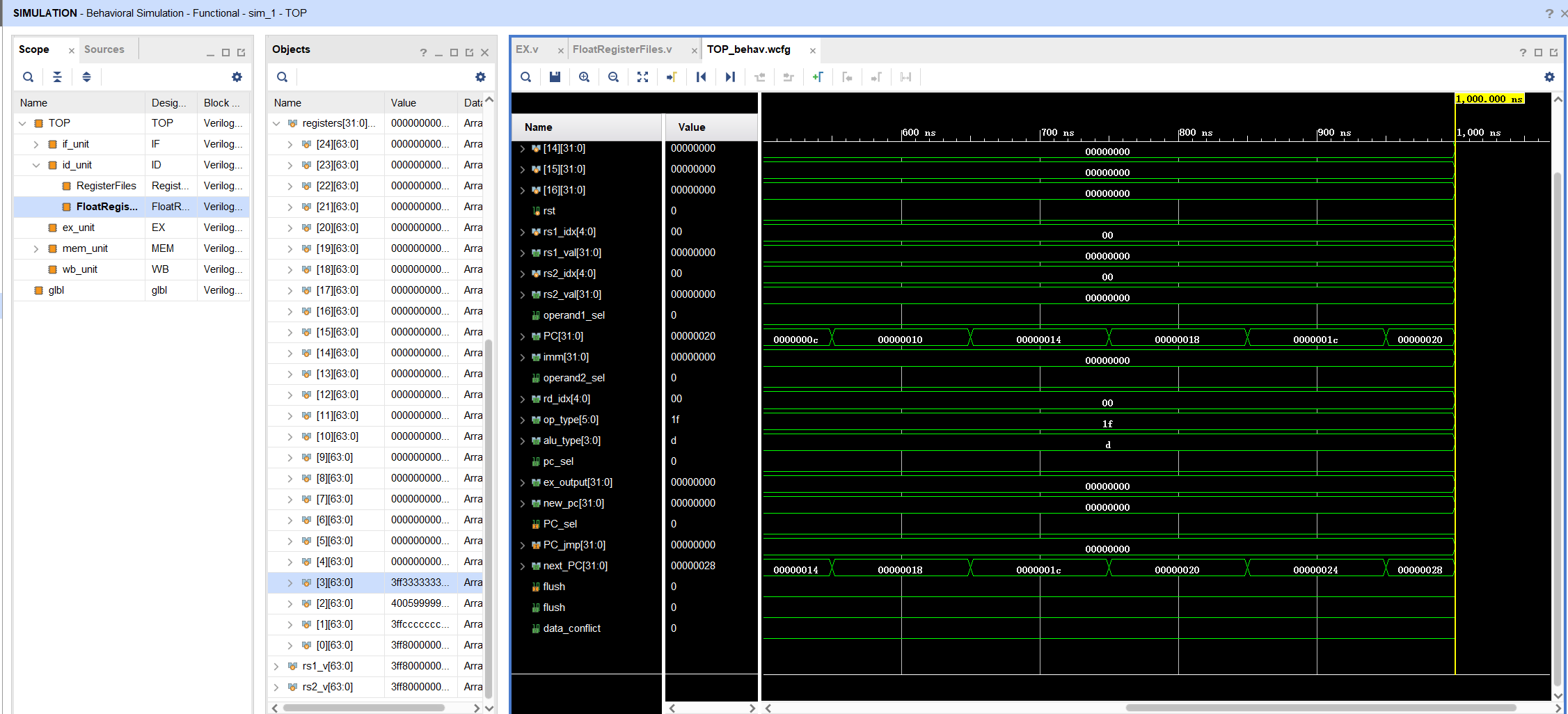
在对C语言测试程序静态链接编译后反汇编的指令筛选过程中，筛选出了浮点乘除指令。但在模拟器执行结果中并未发现有浮点乘除功能的指令，猜测其中原因可能是编译过程中调用了含浮点乘除指令的相关库，但在实际执行及分支跳转过程中，并未执行到浮点指令。

出于对5级流水线浮点乘除功能的测试考虑，额外使用浮点测试程序检验功能的正确与否。

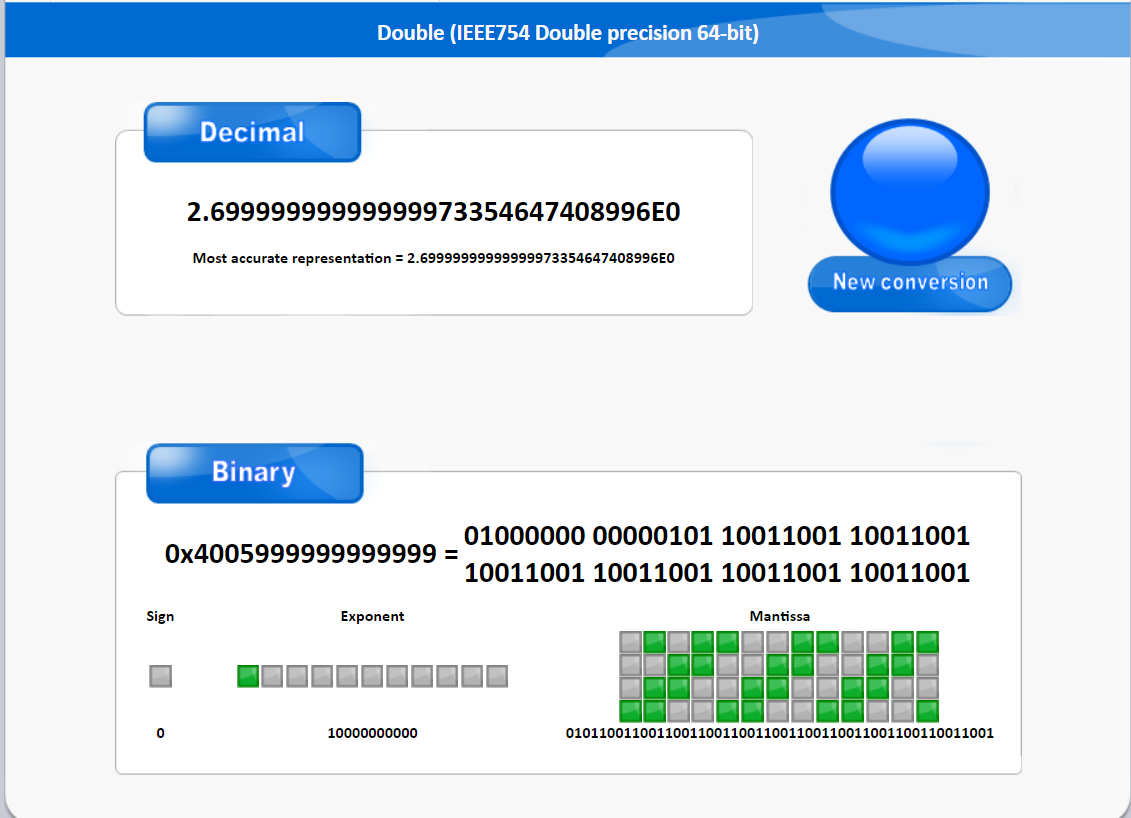
如图所示，将浮点寄存器中ft0和ft1置为1.5和1.8。

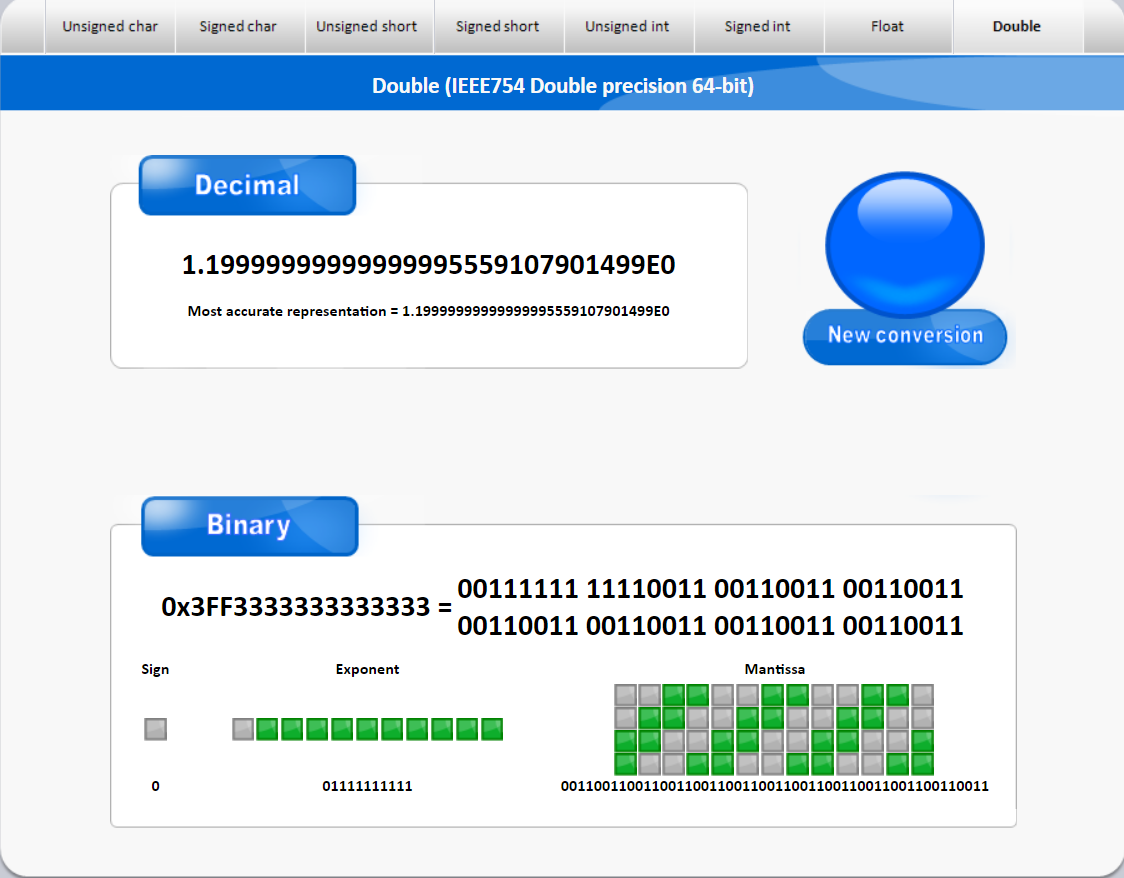


然后运行指令 fmul.d f2,f1,f0和fdiv.d f3,f1,f0。仿真结果如下：



f2和f3的值分别为0x4005999999999999和0x3FF3333333333333，转成10进制如下，结果正确。





# 五、实验总结

## （一）实验历程

第1周：确定初始ISA；

第2周：编写4个测试程序使用工具链编译并比较指令、实现基本5级流水线；

第3周：根据含printf的C语言测试程序汇编指令重新确定ISA、优化改进5级流水线、实现并测试手写RV汇编的2个测试程序和除浮点外的所有指令；

第4周：根据ISA扩增5级流水线的浮点乘除功能、5级流水线测试浮点乘除、Spike模拟器的修改并运行C语言编译程序；

第5周：扩增5级流水线的trap功能、5级流水线测试含printf的C语言测试程序。

## （二）实验分工

|  |  |  |  |
| --- | --- | --- | --- |
|  | 许诗瑶 | 刘朝润 | 刘晓航 |
| 第1周 | ISA初定义、指令格式整理 | | |
| 第2周 | IF、EX、MEM、Fibbonaci测试程序编写 | ID、WB、BubbleSort测试程序编写 | ALU、RAM、测试程序比较 |
| 第3周 | 5级流水线的优化改进代码的重新编写、5级流水线测试调试 | riscv-gnu-tool工具链的研究使用、测试程序指令集的筛选、5级流水线测试调试 | 指令集的调整比较、指令集细节整理 |
| 第4周 | riscv-isa-sim模拟器输出修改、printf程序调用分析 | 浮点乘除功能的实现和测试、模拟器的调试使用和输出修改、printf程序调用分析 | printf程序调用分析 |
| 第5周 | ECALL和SRET实现及相关CSR指令调整、printf测试程序调试 | trap处理程序实现、printf测试程序调试 | 文档整理 |

## （三）实验遇到的问题

**1、第2周**

（1）初步设计时将分支指令的执行提前到译码阶段，但在后续实现过程中发现与改设计冲突的各种逻辑问题，如不能将译码时期的分支判断控制在一周期内，数据冲突时PC值混乱等问题，最后还是决定将分支地址计算正常置于执行阶段。

（2）程序存储器与数据存储器初步设计为调用ip和实现，但实现Store指令时发现存入数据位宽又一字节、半字、一字的差别，无法通过固定的32位位宽数据传入8位数据，于是将数据存储器改用手动使用寄存器实现。

（3）数据冲突的解决，通过维护一张寄存器状态表检测数据冲突的有无，由于本周只实现了最基本的流水线结构，暂时没有对冲突使用其他方案解决，数据冲突存在时直接通过stall信号产生停顿。

（4）对分支跳转的处理，通过冲刷流水线，将正确分支前错误的取指和译码冲刷掉，返回正确的PC值等。

**2、第3周**

（1）译码阶段对指令分级译码需要提前将指令对应要求的信号一一整理出来后从大范围到小范围逐类区分，并且要保证每个指令在区分时有唯一辨识度，不会重复或错误译码

（2）拓展RVM指令集时，需要注意到与RV32I直接alu计算不同，需要先将计算结果的寄存器扩展至64位存储运算结果，再根据指令具体内容与要求取32位值。

**3、第4周**

（1）第4周工作集中于浮点乘除的实现和模拟器的调试使用，其中浮点乘除实现的困难在

于，浮点运算中的五个舍入模式和五个异常，上下溢出比较麻烦，花费了较多的时间和精力。

（2）带有printf的C语言程序经过静态链接编译后代码规模量大，指令行数高达2w+行，

但实际经过模拟器运行时，由于分支跳转等，真正运行的代码行数高达32w+行。修改模拟

器代码使之能输出结果后，由于每运行一行指令输出当前状态，最终输出文件大小有120+G。

于是再修改代码，手动加入计数变量，选择性地输出我们想要时的输出结果，并且方便获得

中间结果与5级流水线结果实时测试比较。

（3）模拟器调试使用时发现，模拟器运行的指令和反汇编的结果不一致，发现原因在于模拟器为用户模式，运行中对ecall之后新增了一些指令，用于保护寄存器等。

**4、第5周**

（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。

## （四）附件说明

RISCVCPU/homework文件夹：按周次顺序整理出的每周提交的实验报告和实验附件；

RISCVCPU/ISA文件夹：包含1.0、2.0和3.0共3个文件夹，记录ISA的调整和更新；

RISCVCPU/pics：为5级流水线框架图不断调整的3个版本；

RISCVCPU/RISCV：5级流水线工程项目源代码，工程结构为vivado环境下的项目结构；

RISCVCPU/test\_program：包含测试程序的3个版本文件，最终4个测试程序所需的数据和程序文件对应Fibonacci、FibonacciRV、BubbleSort、BubbleSortRV四个文件夹下内容，其他文件可见“使用前必读.txt”说明文档；

项目已开源至Github：https://github.com/liuchaorun/RISCVCPU