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

（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

三、 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

地址	特权	名字	描述
管理员自陷 Setup			
0x100	SRW	sstatus	管理员状态寄存器
0x101	SRW	stvec	管理员自陷处理函数基地址
0x104	SRW	sie	管理员中断使能寄存器
0x121	SRW	stimecmp	墙钟（Wall-clock）定时器比较值
管理员定时器			
0xD01	SRO	stime	管理员墙钟时间寄存器
0xD81	SRO	stimeh	stime 的高 32 位，仅 RV32
管理员自陷处理			
0x140	SRW	sscratch	管理员自陷处理函数 Scratch 寄存器
0x141	SRW	sepc	管理员异常程序计数器（exception program counter）
0x142	SRO	scause	管理员自陷原因（trap cause）
0x143	SRO	sbadaddr	管理员坏地址（bad address）
0x144	SRW	sip	管理员挂起的中断（interrupt pending）
管理员保护和翻译			
0x180	SRW	sptbr	页表基地址寄存器
0x181	SRW	sasid	地址-空间 ID
管理员读写、用户只读寄存器阴影			
0x900	SRW	cyclew	RDCYCLE 指令的周期计数器
0x901	SRW	timew	RDTIME 指令的定时器
0x902	SRW	instretw	RDINSTRET 指令的已退休指令（instruction-retired）计数器
0x980	SRW	cyclehw	cycle 的高 32 位，仅 RV32
0x981	SRW	timehw	time 的高 32 位，仅 RV32
0x982	SRW	instrethw	instret 的高 32 位，仅 RV32

machine 级别

地址	特权	名字	描述
机器信息寄存器			
0xF00	MRO	mcpuid	CPU 描述
0xF01	MRO	mimpid	Vendor ID 和版本号
0xF10	MRO	mhartid	硬件线程 ID
机器自陷 Setup			
0x300	MRW	mstatus	机器状态寄存器
0x301	MRW	mtvec	机器自陷处理函数基地址
0x302	MRW	mtdeleg	机器自陷转移（delegation）寄存器
0x304	MRW	mie	机器中断使能寄存器
0x321	MRW	mtimecmp	机器墙钟（Wall-clock）定时器比较值
机器定时器和计数器			
0x701	MRW	mtime	机器墙钟时间寄存器
0x741	MRW	mtimeh	mtime 的高 32 位，仅 RV32
机器自陷处理			
0x340	MRW	mscratch	机器自陷处理函数 Scratch 寄存器
0x341	MRW	mepc	机器异常程序计数器（exception program counter）
0x342	MRW	mcause	机器自陷原因（trap cause）
0x343	MRW	mbadaddr	机器坏地址（bad address）
0x344	MRW	mip	机器挂起的中断（interrupt pending）
机器保护和翻译			
0x380	MRW	mbase	基本寄存器（base register）
0x381	MRW	mbound	绑定寄存器（bound register）
0x382	MRW	mibase	指令基本寄存器
0x383	MRW	mibound	指令绑定寄存器
0x384	MRW	mdbase	数据基本寄存器
0x385	MRW	mdbound	数据绑定寄存器
机器读写、Hypervisor 只读寄存器阴影			
0xB01	MRW	htimew	Hypervisor 墙钟定时器
0xB81	MRW	htimehw	Hypervisor 墙钟定时器高 32 位，仅 RV32
机器主机-目标机接口（非标准 Berkeley 扩展）			
0x780	MRW	mtohost	到主机去的输出寄存器
0x781	MRW	mfromhost	从主机来的输入寄存器

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：

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	≥12	*Reserved*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	≥16	*Reserved*

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：

```
`OPECALL: begin
    // 1. nextPC -> mepc
    csr[`mepc] = PC + 3'b100;
    // 2. supervisor software interupt -> mcause
    csr[`mcause] = 8'h80000001;
    // 3. exception information -> mtval
    csr[`mtval] = 32'b0;
    // 4. mstatus: MIE -> SPIE, machine mode(1) -> SPP, 0 -> SIE
    csr[`mstatus][5] = csr[`mstatus][3];
    csr[`mstatus][8] = 1'b1;
    csr[`mstatus][1] = 1'b0;
    // 5. mtvec -> new  PC
    pc_sel <= 1'b1;
    new_pc = csr[`mtvec];
```

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：

```
`OPSRET: begin
    // 1. mepc -> new  PC
    pc_sel <= 1'b1;
    new_pc = csr[`mepc];
    // 2. SPIE -> SIE, 1 -> SPIE, machine mode(11) -> MPP
    csr[`mstatus][1] = csr[`mstatus][5];
    csr[`mstatus][5] = 1'b1;
    csr[`mstatus][12:11] = 2'b11;
end
```

（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。

```
extern int printf (__const char *__restrict __format, ...);
```

其函数实现如下：

```
int __printf (const char *format, ...)
{
    va_list arg;
    int done;

    va_start (arg, format);
    done = vfprintf (stdout, format, arg);
    va_end (arg);

    return done;
}
```

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

```
#include <stdarg.h>
#undef  __OPTIMIZE__    /* Avoid inline `vprintf' function.  */
#include <stdio.h>
#include <libioP.h>

#undef  vprintf

/* Write formatted output to stdout according to the
   format string FORMAT, using the argument list in ARG.  */
int
__vprintf (const char *format, __gnuc_va_list arg)
{
  return vfprintf (stdout, format, arg);
}

ldbl_strong_alias (__vprintf, vprintf)
```

（2）在 vfprintf 定义文件中有个宏定义：

```
#define outchar(Ch) \
   do \
   { \
       register const INT_T outc = (Ch); \
       if (PUTC (outc, s) == EOF || done == INT_MAX) \
       { \
            done = -1; \
            goto all_done; \
       } \
       ++done; \
   } \
   while (0)
```

可以看到其依赖于 PUTC 也在同一文件中

```
#define PUTC(C, F) IO_putwc_unlocked (C, F)
```

可以从 libc/libio/libio.h 获得 IO_putwc_unlocked

```
#define _IO_putwc_unlocked(_wch, _fp) \
   (_IO_BE ((_fp)->_wide_data->_IO_write_ptr \
        >= (_fp)->_wide_data->_IO_write_end, 0) \
       ? __woverflow (_fp, _wch) \
       : (_IO_wint_t) (*(_fp)->_wide_data->_IO_write_ptr++ = (_wch)))
```

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

```
wint_t
__woverflow (f, wch)
    _IO_FILE *f;
    wint_t wch;
{
    if (f->_mode == 0)
        _IO_fwide (f, 1);
    return _IO_OVERFLOW (f, wch);
}
```

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：

```
#define IO_OVERFLOW(FP, CH) JUMP1 (__overflow, FP, CH)
```

```
const struct _IO_jump_t _IO_file_jumps =
{
  JUMP_INIT_DUMMY,
  JUMP_INIT(finish, INTUSE(_IO_file_finish)),
  JUMP_INIT(overflow, INTUSE(_IO_file_overflow)),
  JUMP_INIT(underflow, INTUSE(_IO_file_underflow)),
  JUMP_INIT(uflow, INTUSE(_IO_default_uflow)),
  JUMP_INIT(pbackfail, INTUSE(_IO_default_pbackfail)),
  JUMP_INIT(xsputn, INTUSE(_IO_file_xsputn)),
  JUMP_INIT(xsgetn, INTUSE(_IO_file_xsgetn)),
  JUMP_INIT(seekoff, _IO_new_file_seekoff),
  JUMP_INIT(seekpos, _IO_default_seekpos),
  JUMP_INIT(setbuf, _IO_new_file_setbuf),
  JUMP_INIT(sync, _IO_new_file_sync),
  JUMP_INIT(doallocate, INTUSE(_IO_file_doallocate)),
  JUMP_INIT(read, INTUSE(_IO_file_read)),
  JUMP_INIT(write, _IO_new_file_write),
  JUMP_INIT(seek, INTUSE(_IO_file_seek)),
  JUMP_INIT(close, INTUSE(_IO_file_close)),
  JUMP_INIT(stat, INTUSE(_IO_file_stat)),
  JUMP_INIT(showmanyc, _IO_default_showmanyc),
  JUMP_INIT(imbue, _IO_default_imbue)
};
libc_hidden_data_def (_IO_file_jumps)
```

（4）源代码做了一堆更多的缓冲区操作，调用了_IO_do_flush

```
#define _IO_do_flush(_f) \
    INTUSE(_IO_do_write)(_f, (_f)->_IO_write_base, \
        (_f)->_IO_write_ptr-(_f)->_IO_write_base)
```

```
static
_IO_size_t
new_do_write (fp, data, to_do)
     _IO_FILE *fp;
     const char *data;
     _IO_size_t to_do;
{
  _IO_size_t count;
  if (fp->_flags & _IO_IS_APPENDING)
    /* On a system without a proper O_APPEND implementation,
       you would need to sys_seek(0, SEEK_END) here, but is
       is not needed nor desirable for Unix- or Posix-like systems.
       Instead, just indicate that offset (before and after) is
       unpredictable. */
    fp->_offset = _IO_pos_BAD;
  else if (fp->_IO_read_end != fp->_IO_write_base)
    {
      _IO_off64_t new_pos
    = _IO_SYSSEEK (fp, fp->_IO_write_base - fp->_IO_read_end, 1);
      if (new_pos == _IO_pos_BAD)
  return 0;
      fp->_offset = new_pos;
    }
  count = _IO_SYSWRITE (fp, data, to_do);
  if (fp->_cur_column && count)
    fp->_cur_column = INTUSE(_IO_adjust_column) (fp->_cur_column - 1, data,
            - count) + 1;
  _IO_setg (fp, fp->_IO_buf_base, fp->_IO_buf_base, fp->_IO_buf_base);
  fp->_IO_write_base = fp->_IO_write_ptr = fp->_IO_buf_base;
  fp->_IO_write_end = (fp->_mode <= 0
           && (fp->_flags & (_IO_LINE_BUF+_IO_UNBUFFERED))
           ? fp->_IO_buf_base : fp->_IO_buf_end);
  return count;
}
```

从中我们发现其调用了_IO_SYSWRITE。

```
/* The 'syswrite' hook is used to write data from an existing buffer
   to an external file.  It generalizes the Unix write(2) function.
   It matches the streambuf::sys_write virtual function, which is
   specific to this implementation. */
typedef _IO_ssize_t (*_IO_write_t) (_IO_FILE *, const void *, _IO_ssize_t);
#define _IO_SYSWRITE(FP, DATA, LEN) JUMP2 (__write, FP, DATA, LEN)
#define _IO_WSYSWRITE(FP, DATA, LEN) WJUMP2 (__write, FP, DATA, LEN)
```

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。

```
_IO_ssize_t
_IO_new_file_write (f, data, n)
     _IO_FILE *f;
     const void *data;
     _IO_ssize_t n;
{
  _IO_ssize_t to_do = n;
  while (to_do > 0)
    {
      _IO_ssize_t count = (__builtin_expect (f->_flags2
         & _IO_FLAGS2_NOTCANCEL, 0)
         ? write_not_cancel (f->_fileno, data, to_do)
         : write (f->_fileno, data, to_do));
      if (count < 0)
      {
        f->_flags |= _IO_ERR_SEEN;
        break;
      }
      to_do -= count;
      data = (void *) ((char *) data + count);
    }
  n -= to_do;
  if (f->_offset >= 0)
    f->_offset += n;
  return n;
}
```

现在我们可以从 libc/posix/unistd.h 中找到 write。

```
/* Write N bytes of BUF to FD.  Return the number written, or -1.

   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern ssize_t write (int __fd, __const void *__buf, size_t __n) __wur;
```

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。

```
1   0x800027c0 (0x14011173) csrrw   sp, sscratch, sp
2   0x800027c4 (0x00011463) bnez    sp, pc + 8
3   0x800027cc (0xec010113) addi    sp, sp, -320
4   0x800027d0 (0x00112223) sw      ra, 4(sp)
5   0x800027d4 (0x00312623) sw      gp, 12(sp)
6   0x800027d8 (0x00412823) sw      tp, 16(sp)
7   0x800027dc (0x00512a23) sw      t0, 20(sp)
8   0x800027e0 (0x00612c23) sw      t1, 24(sp)
9   0x800027e4 (0x00712e23) sw      t2, 28(sp)
10  0x800027e8 (0x02812023) sw      s0, 32(sp)
11  0x800027ec (0x02912223) sw      s1, 36(sp)
12  0x800027f0 (0x02a12423) sw      a0, 40(sp)
13  0x800027f4 (0x02b12623) sw      a1, 44(sp)
14  0x800027f8 (0x02c12823) sw      a2, 48(sp)
15  0x800027fc (0x02d12a23) sw      a3, 52(sp)
16  0x80002800 (0x02e12c23) sw      a4, 56(sp)
17  0x80002804 (0x02f12e23) sw      a5, 60(sp)
18  0x80002808 (0x05012023) sw      a6, 64(sp)
19  0x8000280c (0x05112223) sw      a7, 68(sp)
20  0x80002810 (0x05212423) sw      s2, 72(sp)
21  0x80002814 (0x05312623) sw      s3, 76(sp)
22  0x80002818 (0x05412823) sw      s4, 80(sp)
23  0x8000281c (0x05512a23) sw      s5, 84(sp)
24  0x80002820 (0x05612c23) sw      s6, 88(sp)
25  0x80002824 (0x05712e23) sw      s7, 92(sp)
26  0x80002828 (0x07812023) sw      s8, 96(sp)
27  0x8000282c (0x07912223) sw      s9, 100(sp)
28  0x80002830 (0x07a12423) sw      s10, 104(sp)
29  0x80002834 (0x07b12623) sw      s11, 108(sp)
30  0x80002838 (0x07c12823) sw      t3, 112(sp)
31  0x8000283c (0x07d12a23) sw      t4, 116(sp)
32  0x80002840 (0x07e12c23) sw      t5, 120(sp)
33  0x80002844 (0x07f12e23) sw      t6, 124(sp)
34  0x80002848 (0x140012f3) csrrw   t0, sscratch, zero
35  0x8000284c (0x10002473) csrr    s0, sstatus
36  0x80002850 (0x14102373) csrr    t1, sepc
37  0x80002854 (0x143023f3) csrr    t2, stval
38  0x80002858 (0x14202e73) csrr    t3, scause
39  0x8000285c (0x00512423) sw      t0, 8(sp)
40  0x80002860 (0x08812023) sw      s0, 128(sp)
41  0x80002864 (0x08612223) sw      t1, 132(sp)
42  0x80002868 (0x08712423) sw      t2, 136(sp)
43  0x8000286c (0x09c12623) sw      t3, 140(sp)
44  0x80002870 (0x00010513) mv      a0, sp
```

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。

```
45  lw a3,0(x0)  00002683
46  addi a3,a3,1 00168693
47  sw a3,0(x0)  00d02023
```

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。

```
48  0x80002878 (0x00010513) mv      a0, sp
49  0x8000287c (0x10047413) andi    s0, s0, 256
50  0x80002880 (0x00041663) bnez    s0, pc + 12
51  0x80002884 (0x14010113) addi    sp, sp, 320
52  0x80002888 (0x14011073) csrw    sscratch, sp
53  0x8000288c (0x08052283) lw      t0, 128(a0)
54  0x80002890 (0x08452303) lw      t1, 132(a0)
55  0x80002894 (0x10029073) csrw    sstatus, t0
56  0x80002898 (0x14131073) csrw    sepc, t1
57  0x8000289c (0x00452083) lw      ra, 4(a0)
58  0x800028a0 (0x00852103) lw      sp, 8(a0)
59  0x800028a4 (0x00c52183) lw      gp, 12(a0)
60  0x800028a8 (0x01052203) lw      tp, 16(a0)
61  0x800028ac (0x01452283) lw      t0, 20(a0)
62  0x800028b0 (0x01852303) lw      t1, 24(a0)
63  0x800028b4 (0x01c52383) lw      t2, 28(a0)
64  0x800028b8 (0x02052403) lw      s0, 32(a0)
65  0x800028bc (0x02452483) lw      s1, 36(a0)
66  0x800028c0 (0x02c52583) lw      a1, 44(a0)
67  0x800028c4 (0x03052603) lw      a2, 48(a0)
68  0x800028c8 (0x03452683) lw      a3, 52(a0)
69  0x800028cc (0x03852703) lw      a4, 56(a0)
70  0x800028d0 (0x03c52783) lw      a5, 60(a0)
71  0x800028d4 (0x04052803) lw      a6, 64(a0)
72  0x800028d8 (0x04452883) lw      a7, 68(a0)
73  0x800028dc (0x04852903) lw      s2, 72(a0)
74  0x800028e0 (0x04c52983) lw      s3, 76(a0)
75  0x800028e4 (0x05052a03) lw      s4, 80(a0)
76  0x800028e8 (0x05452a83) lw      s5, 84(a0)
77  0x800028ec (0x05852b03) lw      s6, 88(a0)
78  0x800028f0 (0x05c52b83) lw      s7, 92(a0)
79  0x800028f4 (0x06052c03) lw      s8, 96(a0)
80  0x800028f8 (0x06452c83) lw      s9, 100(a0)
81  0x800028fc (0x06852d03) lw      s10, 104(a0)
82  0x80002900 (0x06c52d83) lw      s11, 108(a0)
83  0x80002904 (0x07052e03) lw      t3, 112(a0)
84  0x80002908 (0x07452e83) lw      t4, 116(a0)
85  0x8000290c (0x07852f03) lw      t5, 120(a0)
86  0x80002910 (0x07c52f83) lw      t6, 124(a0)
87  0x80002914 (0x02852503) lw      a0, 40(a0)
88  0x80002918 (0x10200073) sret
```

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。

```
Fibonacci.out:     file format elf32-littleriscv

Disassembly of section .text:

00010074 <register_fini>:
   10074:   00000793            li a5,0
   10078:   00078863            beqz  a5,10088 <register_fini+0x14>
   1007c:   00014537            lui   a0,0x14
   10080:   8c050513            addi  a0,a0,-1856 # 138c0 <__libc_fini_array>
   10084:   1580306f            j  131dc <atexit>
   10088:   00008067            ret

0001008c <_start>:
   1008c:   00017197            auipc gp,0x17
   10090:   a7c18193            addi  gp,gp,-1412 # 26b08 <__global_pointer$>
   10094:   1cc18513            addi  a0,gp,460 # 26cd4 <__malloc_max_total_mem>
   10098:   22818613            addi  a2,gp,552 # 26d30 <__BSS_END__>
   1009c:   40a60633            sub   a2,a2,a0
   100a0:   00000593            li a1,0
   100a4:   230000ef            jal   ra,102d4 <memset>
   100a8:   00003517            auipc a0,0x3
   100ac:   13450513            addi  a0,a0,308 # 131dc <atexit>
   100b0:   00050863            beqz  a0,100c0 <_start+0x34>
   100b4:   00004517            auipc a0,0x4
   100b8:   80c50513            addi  a0,a0,-2036 # 138c0 <__libc_fini_array>
   100bc:   120030ef            jal   ra,131dc <atexit>
   100c0:   178000ef            jal   ra,10238 <__libc_init_array>
   100c4:   00012503            lw a0,0(sp)
   100c8:   00410593            addi  a1,sp,4
   100cc:   00000613            li a2,0
   100d0:   074000ef            jal   ra,10144 <main>
   100d4:   1340006f            j  10208 <exit>

000100d8 <__do_global_dtors_aux>:
   100d8:   1e41c703            lbu   a4,484(gp) # 26cec <completed.1>
   100dc:   04071263            bnez  a4,10120 <__do_global_dtors_aux+0x48>
   100e0:   ff010113            addi  sp,sp,-16
   100e4:   00812423            sw s0,8(sp)
   100e8:   00078413            mv s0,a5
   100ec:   00112623            sw ra,12(sp)
   100f0:   00000793            li a5,0
```

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：

```
lcr@DESKTOP-JDL9JAK:~/riscv/bin$ ./elf2hex 1 262144 Fibonacci.out 0 > b.txt
lcr@DESKTOP-JDL9JAK:~/riscv/bin$
```

获得的内容部分如下：

C: > Users > chaorunliu > b.txt

```
81159  86
81160  07
81161  13
81162  86
81163  76
81164  07
81165  13
81166  18
81167  38
81168  00
81169  6f
81170  f0
81171  9f
81172  e9
81173  13
81174  06
81175  40
81176  55
81177  63
81178  6c
81179  d6
81180  00
81181  93
81182  d6
81183  27
81184  01
81185  13
81186  88
81187  d6
81188  07
81189  13
81190  86
81191  c6
81192  07
81193  13
81194  18
81195  38
81196  00
81197  6f
81198  f0
81199  df
81200  e7
```

（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：

```c
#include <stdio.h>

int main(){
    int n = 5;
    int fib = 0;

    if(n == 1) fib = 0;
    else if(n == 2) fib = 1;
    else {
        int i = 0;
        int j = 1;
        int k;
        for(k = 3; k <= n; k++){
            fib = i + j;
            i = j;
            j = fib;
        }
    }

    printf("The %d of fobonacci is: %d.\n", n, fib);
}
```

使用如下命令将其转换为 elf 可执行文件。

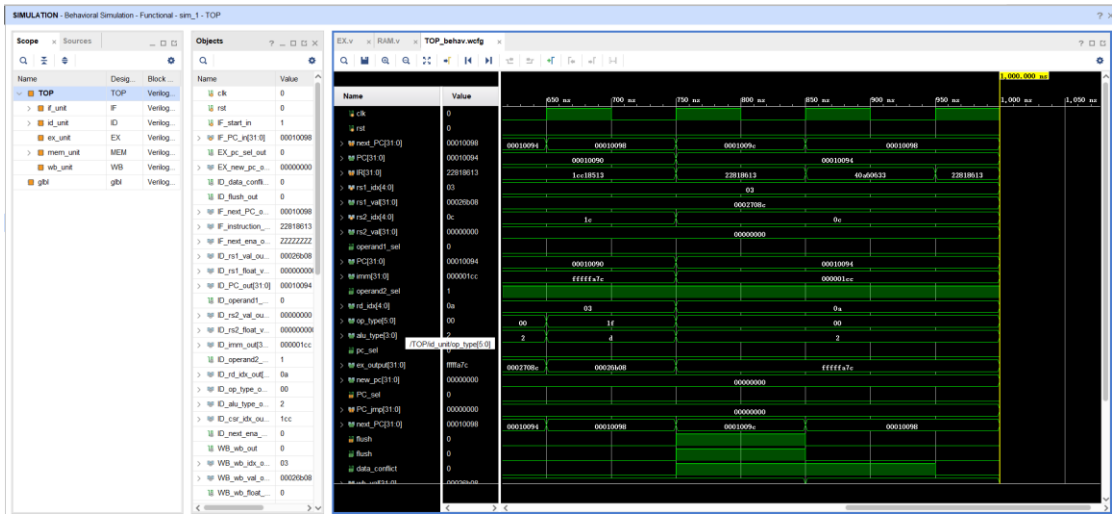
```
./riscv32-unknown-elf-gcc -E Fibonacci.c -o Fibonacci.i
./riscv32-unknown-elf-gcc -S Fibonacci.i -o Fibonacci.s
./riscv32-unknown-elf-gcc -c Fibonacci.s -o Fibonacci.o
./riscv32-unknown-elf-gcc -static Fibonacci.o -o Fibonacci.out
./riscv32-unknown-elf-objdump -S Fibonacci.out > Fibonacci.txt
```

将可执行文件转换为 hex。

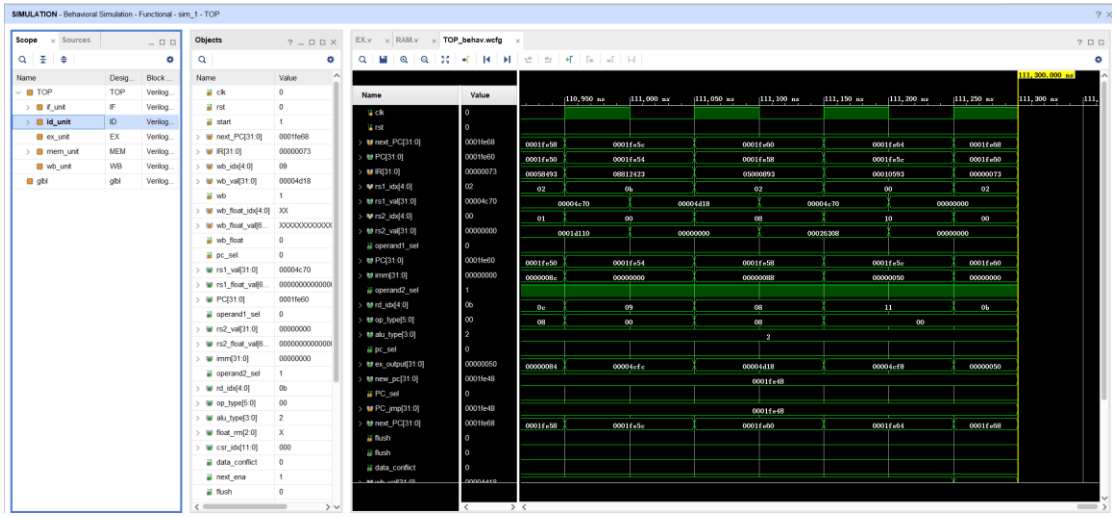
仿真过程：

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。

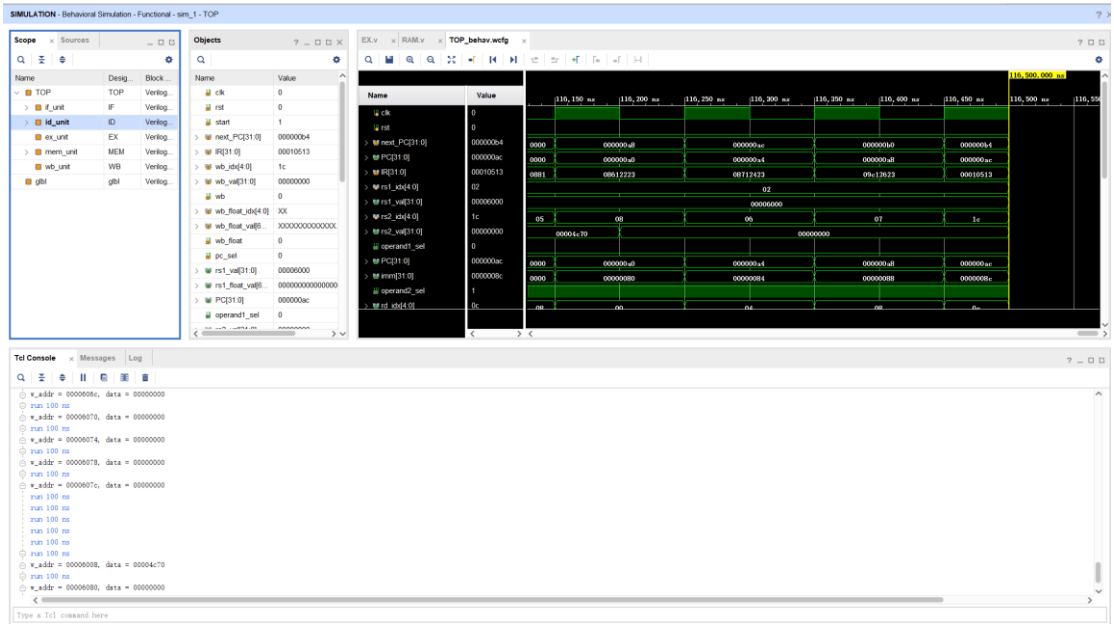
2、 运行前十个周期



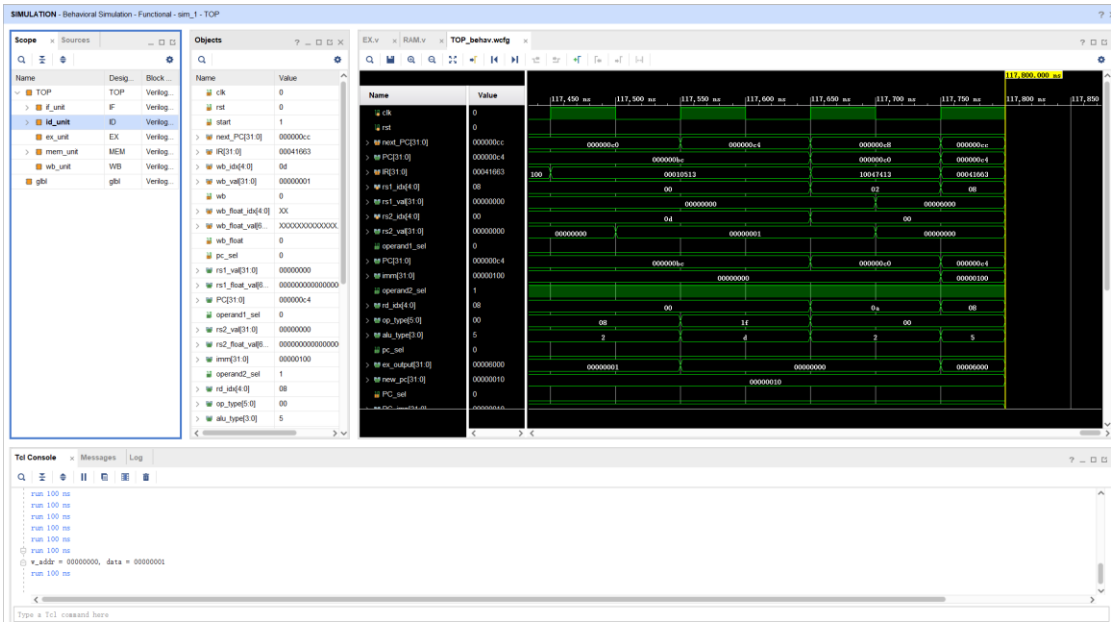
3、 运行至第一次系统调用 ecall 前



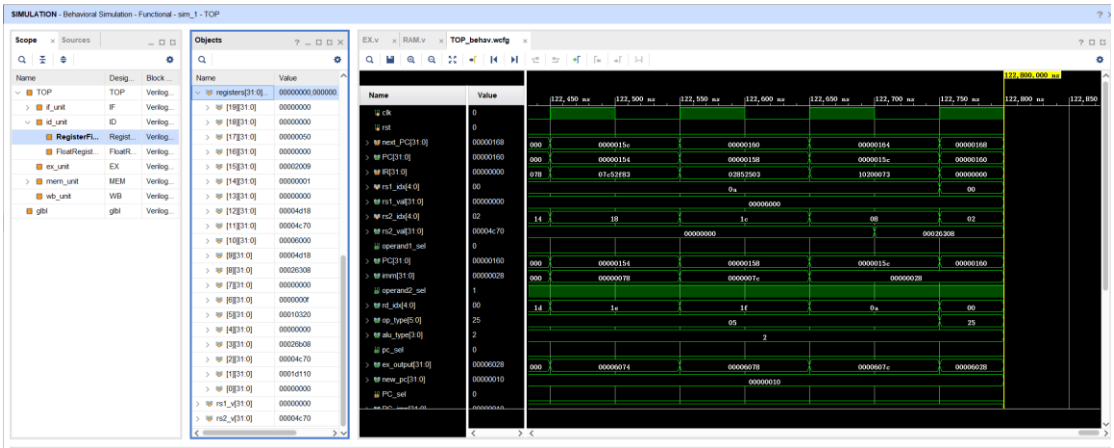
4、 进入 trap 程序保护现场



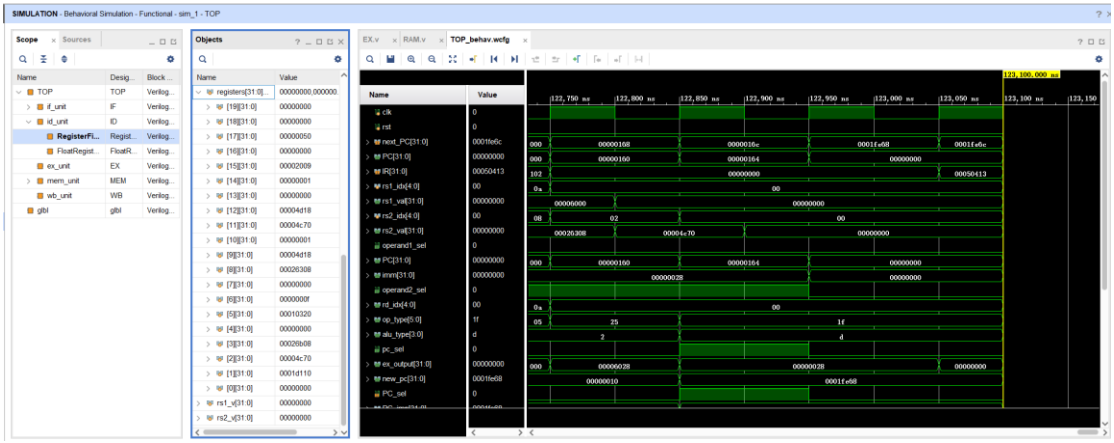
5、 Trap 处理程序



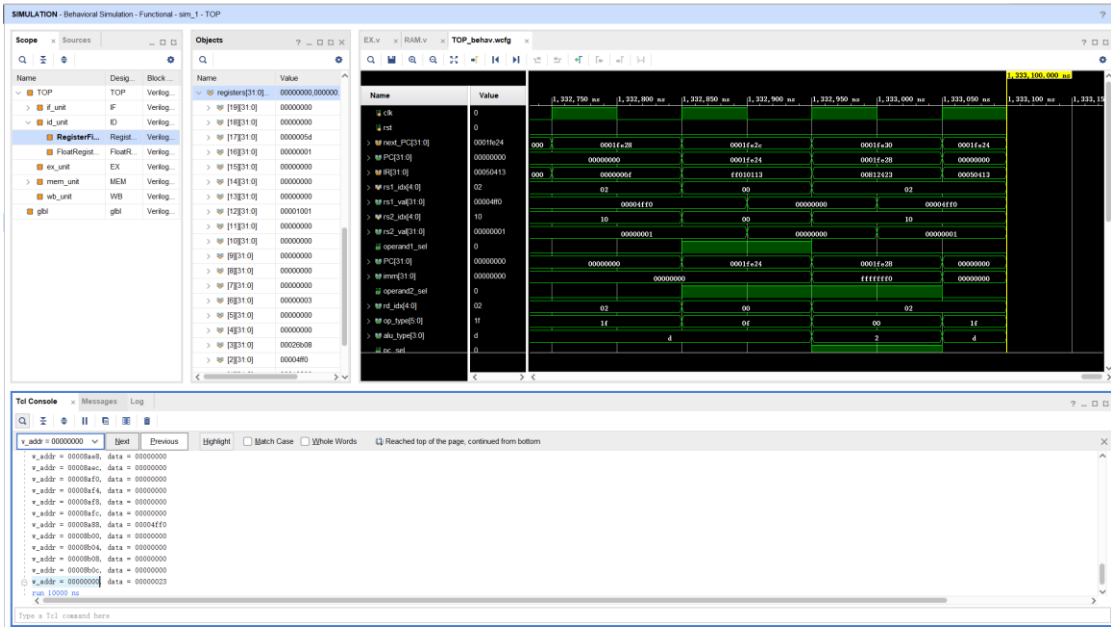
6、 恢复现场



7、 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