项目结题报告

项目名称：

1） RISC-V 指令级模拟器设计与实现

2） RISC-V 指令模拟器中对系统调用的支持

1.项目描述

项目内容：实现 RISC-V 指令级模拟器设计，同时该模拟器可支持系统调用。输入是 RISC-V 工具链生成的可执行文件，输出是该可执行文件的执行结果，并统计输出执行程序的指令数。

项目思路：首先对使用 RISC-V 工具链生成的 ELF 文件进行解析， ELF 文件的最初 52 个字节记录了所有的 Header Table 在文件中的位置，在 ELF 文件头中放有程序的虚拟地址入口点。然后进行模拟器初始化即根据读取的 ELF 文件信息将数据段和代码段载入到虚拟内存中，并初始化 SP、GP、PC。最后对指令进行取址、译码、执行操作，同时通过添加 scall 指令实现系统调用。

2.ELF 格式解析

ELF 文件中的信息可以划分为两种：Header 和 Section。Header 描述 Section 的信息，如 Section 的用途，长度，以及如何在整个文件中找到。读取 ELF 文件的过程，就是根据 Header 找到 Section 的过程。ELF 文件的最初 52 个字节记录了所有的 Header Table 在文件中的位置。Elf Header 的定义如下：

```
typedef struct {
    unsigned char  e_ident[16];
    Elf32_Half     e_type;
    Elf32_Half     e_machine;
    Elf32_Word     e_version;
    Elf32_Addr     e_entry;
    Elf32_Off      e_phoff;
    Elf32_Off      e_shoff;
    Elf32_Word     e_flags;
    Elf32_Half     e_ehsize;
    Elf32_Half     e_phentsize;
    Elf32_Half     e_phnum;
    Elf32_Half     e_shentsize;
    Elf32_Half     e_shnum;
    Elf32_Half     e_shstrndx;
} Elf32_Ehdr;
```

其中 e_entry 定义了程序入口地址，根据此信息来设置 PC 寄存器；e_phoff 定义了 program header table 在整个文件中的位置；e_phentsize 定义了 program header table 中每一项的大小；e_phnum 定义了 program header table 中有多少项。

Program Header 定义如下：

```
typedef struct {
    Elf32_Word    p_type;
    Elf32_Off     p_offset;
    Elf32_Addr    p_vaddr;
    Elf32_Addr    p_paddr;
    Elf32_Word    p_filesz;
    Elf32_Word    p_memsz;
    Elf32_Word    p_flags;
    Elf32_Word    p_align;
} Elf32_Phdr;
```

其中 p_offset 定义了程序段在整个文件中的位置；p_vaddr 定义了程序段放在内存中时，首个字节的虚拟地址；p_memsz 定义了程序段在内存中的大小。可以根据这三个内容将文件中起始位置为 p_offset 的程序段载入到起始地址为 P_vaddr，大小为 p_memsz 的虚拟内存块中，并且用符号表中保存的 GP 的值设置 GP 寄存器。

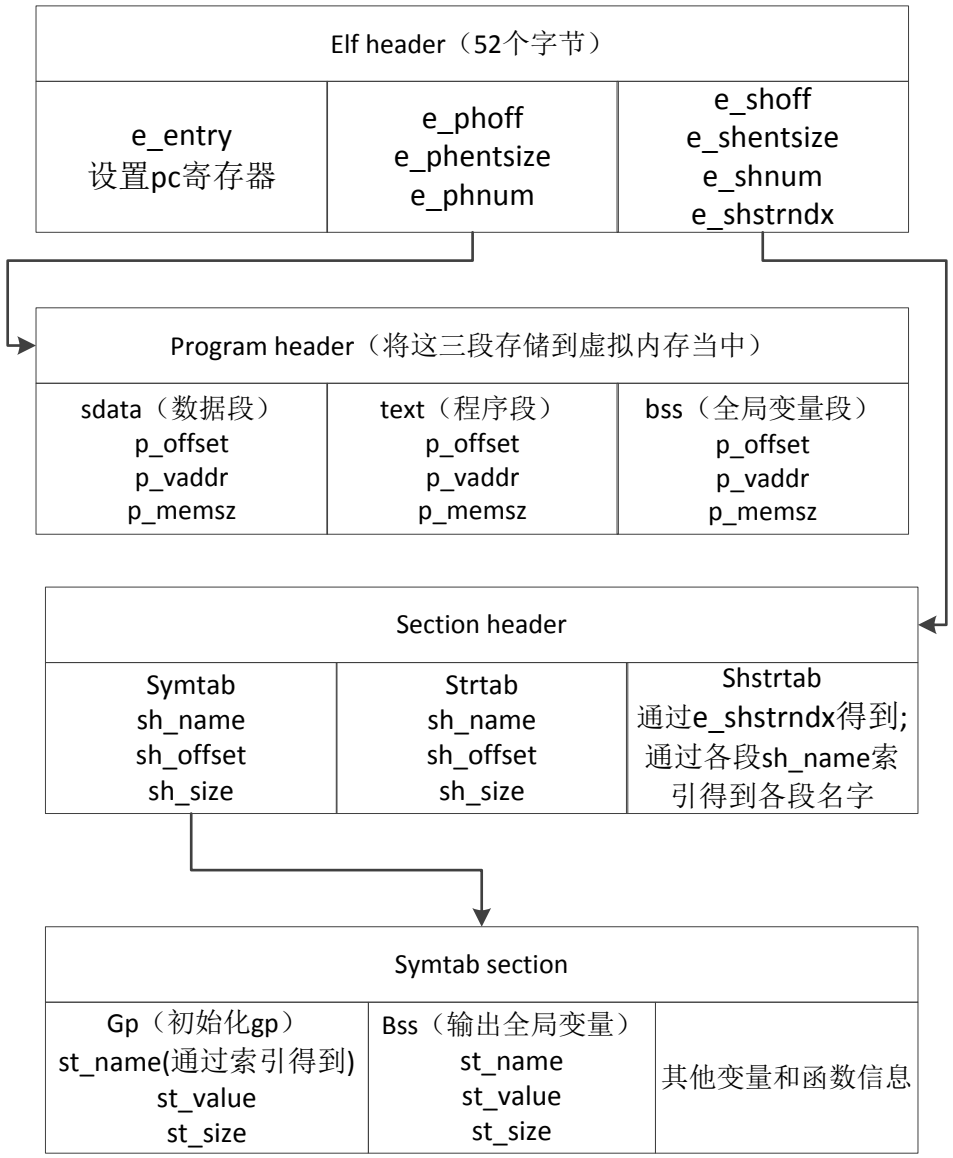

图 3 - 1 ELF 文件结构图

3. 指令模拟器结构及工作流程

1） 指令模拟器的组织结构用函数结构图来表示，如图 3-1 所示

Main 函数调用 ELF_parser()函数对文件进行解析，然后再调用 sim_init()函数，对模拟器进行初始化；最后调用 sim_exec()函数，进行取址、译码、执行过程。

Elf文件解析函数ELF_parser()：调用header、section各部分解析函数，将得到的这些信息存储到相应的结构中，然后将这些信息输出，以便调试。

模拟器初始化函数 sim_init()：主要实现①将数据段和代码段存入内存；②初始化 GP、PC、SP 等；③插入开始和结束指令。其中会调用内存读写函数 sim_mem_read()和 sim_mem_write()

模拟器执行函数 sim_exec()：实现指令的取址、译码、执行，并通过实现 scall 指令实现系统调用。

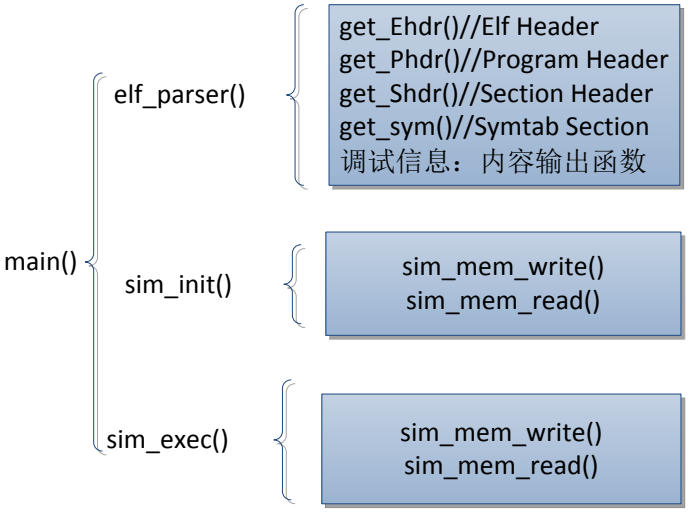

图 3 - 2 指令模拟器函数结构图

2） 指令模拟器的工作流程如图 3-2 所示：

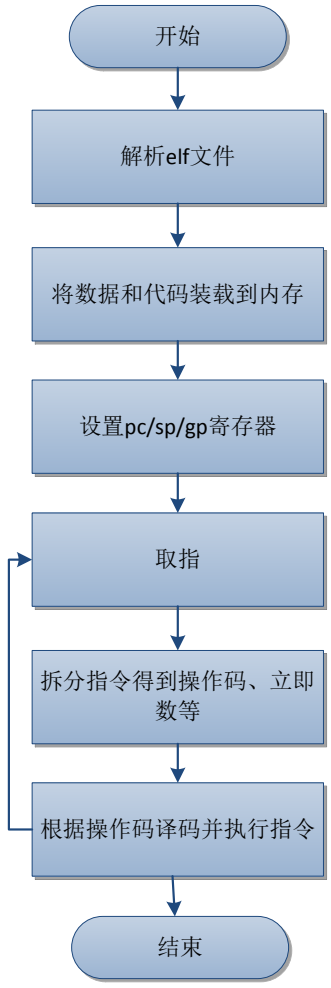

图 3 - 3 指令模拟器工作流程

4.基本功能测试

以下是三个简单测试程序的运行结果截图以及统计的指令数：

（1） HanNo.c 测试结果

编译选项：riscv64-unknown-linux-gnu-gcc -o HanNo.out -m32 -static -e main HanNo.c

指令数为：1264

（2） quickSort.c 测试结果

编译选项：riscv64-unknown-linux-gnu-gcc -o quickSort.out -m32 -static -e main quickSort.c

指令数为：1603

（3） mat.c 测试结果

编译选项：riscv64-unknown-linux-gnu-gcc -o mat.out -m32 -static -e main mat.c

指令数为：606

5.系统调用的实现

5.1 系统调用原理

系统调用是用户程序与内核交互的一个接口。

RISC-V 中系统调用号存储在寄存器 a[7]即 r[17]中，通过 a[7]的值就可以判断调用的系统调用。

对于参数传递，通过寄存器完成的。最多允许向系统调用传递 7 个参数，分别依次由 a[0]、a[1]、a[2]、a[3]、a[4]、a[5]、a[6]（也就是 r[10]、r[11]、r[12]、r[13]、r[14]、r[15]、r[16]）这个 7 个寄存器完成。

一个系统调用函数可能被多个函数进行调用。

5.2 系统调用实现流程

（1）添加系统调用指令 scall，scall 指令的指令格式为 0x00000073；（2）系统调用号存储在寄存器 r[17]中，通过 r[17]中的值判断将要进行哪个系统调用；（3）添加需要的系统调用。这里我们添加的系统调用主要有一下四个：

Write：write 要传递三个参数，一个文件描述符，一个内存区的地址（该缓冲区包含传送的数据的存放位置），以及一个数 count（指定应该传送多少字节），这些参数的相关内容存储在寄存器 r[10]、r[11]、r[12]（分别表示文件描述符、内存偏移地址、传入的长度），其返回值是所成功传送的字节数到 r[10]寄存器中。

```
case RISCV_write:
    x[10] = write(x[10], (const void*)(x[11]+MEM), x[12]);
    break;
```

read：read 的实现与 write 相似，在此不再赘述

time：time传递一个参数为内存所指向的地址，内存偏移地址存储在寄存器r[10]当中，成功返回秒数。

```
case RISCV_time:
    x[10] = time((time_t*)(x[10]+MEM));
    break;
```

exit：最后还要添加退出指令 exit 其系统调用号为 93，识别到 93 时，直接退出。

以 printf 函数为例简述函数调用过程：进入 printf 函数后，经过一系列的函数调用，进入 write 函数，执行到其中的 scall 指令后，判断 r[17]值为 63，即要调用 write 系统调用，然后进行参数传递进入内核执行 write 系统调用，最后将返回值存入 r[10]中，系统调用结束。

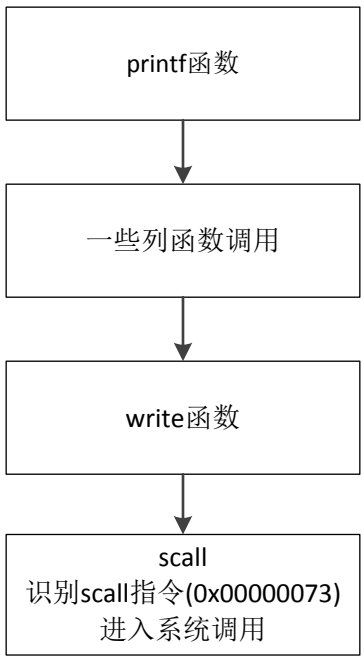

图 5 - 1 printf 为例系统调用过程

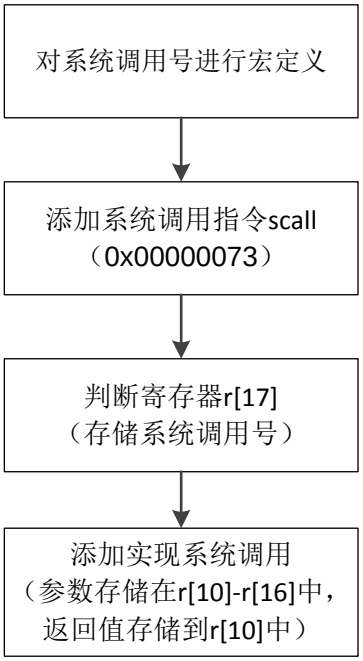

图 5 - 2 系统调用实现

6.Dhrystone 的运行

6.1 Dhrystone 的动态执行指令数，并和 Spike 执行结果比较

Dhrystone编译：用Makefile进行编译
编译工具链为：riscv64-unknown-elf-gcc
编译选项为：-m32 -march=RV32I
总指令数为 71663702，循环执行次数为 100000 次，所以动态执行指令数约为：716 条；执行时间为 2521668μm；程序运行速度为 28419166 条/秒

6.2 优化方案

（1）指令计数从大循环处开始。

（2）尝试不同的编译选项，对结果进行比较。

（3）实现 64 位指令模拟器。