体系实习 lab2

贺义鸣 1700012786

April 19, 2019

1 实验（开发）环境

项目	详细指标和参数
处理器型号及相关参数	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
内存	7879180 kB
外存	68.2G
操作系统及版本	ubuntu 16.04
编译器版本（及编译参数）	gcc 5.4.0
库函数及版本	GLIBC 2.23

2 设计概述

实现以下内容：

1. elf loader。没有使用开源代码，使用 <elf.h> 系统头文件。
2. memory 管理，在系统中使用开辟内存，支持任意 offset 的任意长度的读取与保存。
3. register pile ，我使用了一个类作为寄存器。
4. 指令的取值，解码，执行，访存，写回 5 个阶段，并且使用流水线管理（每次 loop 倒序执行 5 个阶段），处理了数据冒险和控制冒险。
5. gdb 单步调试，支持内存读取，寄存器堆读取。

3 具体设计和实现

3.1 可执行文件的装载、初始化和存储接口

代码位于 utils.cpp 文件的 loader_elf 函数，实现方式是将 elf 文件的 segmentation 复制到内存中对应的地方。

```
bool load_elf(const char * file_name){
    FILE* file=NULL;
    file=fopen(file_name,"rb");
    if(file==NULL){
        printf("open file error\n");
        return false;
    }
    unsigned long size=get_filesize(file_name);

    byte* elf_buffer=(byte*)malloc(sizeof(byte)*size);//把对应文件读入elf_buffer中
    if(fread(elf_buffer,sizeof(char),size,file)<size){
        printf("the programe is too big\n");
        return false;
    }else{
        if(GDB_MODE)
            printf(">");
        printf("read the programe successfully\n");
    }
    Elf64_Ehdr* elf_header;
    elf_header=(Elf64_Ehdr*)elf_buffer;
    memAddress program_entry_offset=(memAddress)(elf_header->e_entry);
    byte*  cur_p_mem=sim_mem.get_memory_point(program_entry_offset);
    Elf64_Half seg_num=elf_header->e_phnum;
    Elf64_Phdr* seg_header = (Elf64_Phdr*)((unsigned char*)elf_header + elf_header->e_phoff);
    Elf64_Half  seg_header_entry_size=elf_header->e_phentsize;
    cur_p_mem=sim_mem.get_memory_point(seg_header->p_vaddr);
    for(int cnt=0;cnt<seg_num;++cnt){
        unsigned char* seg_in_file=(unsigned char*)elf_header+seg_header->p_offset;
        Elf64_Xword seg_size_in_mem=seg_header->p_memsz;
        memcpy(cur_p_mem,seg_in_file,seg_size_in_mem);
        seg_header=(Elf64_Phdr*)((unsigned char*)seg_header+seg_header_entry_size);
        cur_p_mem=sim_mem.get_memory_point(seg_header->p_vaddr);
    }
    /*------------- end of segments copy---------------*/
    fclose(file);
    free(elf_buffer);

    /* ---- init regs ------*/
    sim_register_pile.setPC(program_entry_offset);
    sim_register_pile.writeReg(zero, 0);
    sim_register_pile.writeReg(sp, STACK_TOP);
    return true;
}
```

3.2 指令语义的解析和控制信号的处理

语义解析对应文件 instruction_decode.cpp，将读到的 01 串进行解析。控制信号采用很多全局变量进行控制，具体可以查看 main.cpp 最前的定义。

```
//execute 阶段
bool instruction_execute=false;
instruction instruction_to_execute;

//decode 阶段
bool instruction_decode=false;
ins instruction_to_decode;

//write back 阶段
bool write_reg=false;
int write_reg_type; //0 PC 1 int 2 float 3 double
regID write_reg_id;
struct write_reg_value reg_value;

//memory 阶段
bool use_memory=false;
int load_save_type; //0 int_load 1 int_save 2 float_load 3 float_save
memAddress address;
int get_length; //使用内存长度
bool memory_signed;
regID memory_reg;

//fetch 阶段
//frtch 阶段总是可以执行，除非遇到跳转指令
bool try_fetch=true;
```

3.3 系统调用和库函数接口的处理

实现了对于所有指令的处理（几乎完全）。

3.4 性能计数相关模块的处理

使用一个 map 统计具体指令执行数量，每个 loop 对周期计算 +1.

3.5 调试接口和其它接口等

实现了简易 gdb，支持单步调试，输出这一步 5 个阶段分别执行了什么。并且支持 memory 和 register pile 查看。

```
//gdb控制
if(GDB_MODE){
    char cmd[20];
    if(IS_FIRST_GDB){
        printf(">\n");
        printf("> select a mode you want to run with:\n");
        printf("> continue: continue running\n");
        printf("> step: step mode\n");
        printf("> memory: print memory content\n");
        printf("> register: print register file info\n");
        printf("> quit: quit gdb mode\n>\n>\n");
    }
    if(IS_FIRST_GDB||GDB_TYPE==1){
        entry: IS_FIRST_GDB=false;
        printf(">");
        scanf("%s",cmd);        //read command
        fflush(stdin);          //clean stdin buffer
        if(strcmp(cmd,"continue")==0){
            GDB_MODE=false;
            verbose=false;
        }else if(strcmp(cmd,"step")==0){
            GDB_TYPE=1;
            verbose=true;
        }else if(strcmp(cmd,"memory")==0){
            memAddress debug_mem=0;
            reg32 mem_content=0;
            printf("> set memory address\n");
            printf("> ");
            scanf("%lx",&debug_mem);
            for(int row=0;row<4;++row){
                printf("> ");
                for(int col=0;col<4;++col){
                    mem_content=sim_mem.get_memory_32(debug_mem);
                    printf("0x%08x",mem_content);
                    printf("    ");
                    debug_mem+=4;
                }
                printf("\n");
            }
            goto entry;
        }else if(strcmp(cmd,"register")==0){
            sim_register_pile.readReg();
            sim_register_pile.readFloatReg();
            goto entry;
        }else if(strcmp(cmd,"quit")==0){
            printf("gdb quit\n");
            return 0;
        }
    }

}
```

4 功能测试和性能评测

见程序输出。(默认跳转方式是不跳转)

5 其它需要说明的内容

1. 实现了完整指令集
2. 为了编程方便，我的 decode 阶段不会从 regster 读取数据，改为 execute 阶段执行，这样我可以不处理数据冒险。