北京郵電大學

Beijing University of Posts and Telecommunications

计算机组成原理课程设计报告

课题：基于高级语言的硬件虚拟机仿真设计

学院：计算机学院

姓名：XXX

班级：XXX

学号：XXX

二〇一八年五月

目录

一、 实验概述.....2
（一） 学科背景.....2
（二） 实验目的.....2
（三） 实验环境.....2
（四） 开发工具.....2
二、 虚拟硬件结构.....3
（一） 冯·诺依曼计算机体系.....3
（二） Simple-VM 数据通路.....4
（三） Simple-VM 模块化设计.....4
（1） 中央处理器.....5
（2） 主存储器.....6
（3） 输入/输出设备.....7
（4） 中断处理器.....7
三、 指令集架构.....8
（一） 操作数与寻址模式.....8
（二） 指令格式.....8
（三） 指令集说明.....9
（1） 指令运行框架.....9
（2） 一般传送指令.....10
（3） 堆栈操作指令.....10
（4） 算术运算指令.....11
（5） 逻辑运算指令.....11
（6） 转移指令.....12
（7） 特殊指令.....12
（四） 软件设计.....13
（1） 总体框架.....13
（2） 数据结构与信号量.....14
（3） 启动窗口与设置窗口.....15
A. 界面设计.....15
B. 功能设计.....15
（4） 运行窗口.....17
A. 界面设计.....17
B. 功能设计.....17
（5） 软硬件映射关系总结.....20
（五） 软件测试.....20
（1）指令完备性测试.....20
A. 测试用例.....20
B. 结果分析.....22
（2） 中断处理测试.....23
（3） 异常检测测试.....25
（六） 用户手册.....25
（1） 安装说明.....25
（2） 加载程序.....26
（3） 查看虚拟机状态.....26
（4） 功能按钮.....27
（5） 输入框.....27
（6） 个性化配置.....27
（七） 实验总结与心得.....28
（八） 参考文献.....28
（九） 附录.....29

一、实验概述

（一） 学科背景

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对全方位计算机专业人才的需求日益迫切，软件专精或硬件专精已无法紧跟时代的潮流。计算机组成原理课程致力于培养学生计算机系统结构的知识，建立学生基本的整机概念，而虚拟机的开发则极好地帮助学生将所学知识融会贯通。本实验着眼于建立基本的虚拟机模型，以小见大，从而巩固计算机系统结构知识。

（二） 实验目的

通过高级语言以软件的方式模拟实体计算机的基本运作方式，实现冯诺依曼体系计算机系统，模拟计算机系统整机工作袁立，直观展现硬件运作过程，将所学的软件开发知识与硬件基础知识进行综合，从而锻炼学生的计算机系统综合能力。

（三） 实验环境

操作系统	Linux Ubuntu 64bits (version 4.13.0-41-generic) By VMware
处理器	Intel(R) Core(TM) i5-5300U 2.30GHz
显卡	Intel(R) HD Graphics 5500
编译器版本	GCC 7.2.0

（四） 开发工具

开发语言	Python 3.6.3
GUI 开发库	PyGtk 3.24.1
第三方工具	Pango（风格样式）、Glade（GUI Designer）

二、虚拟硬件结构

（一） 冯·诺依曼计算机体系

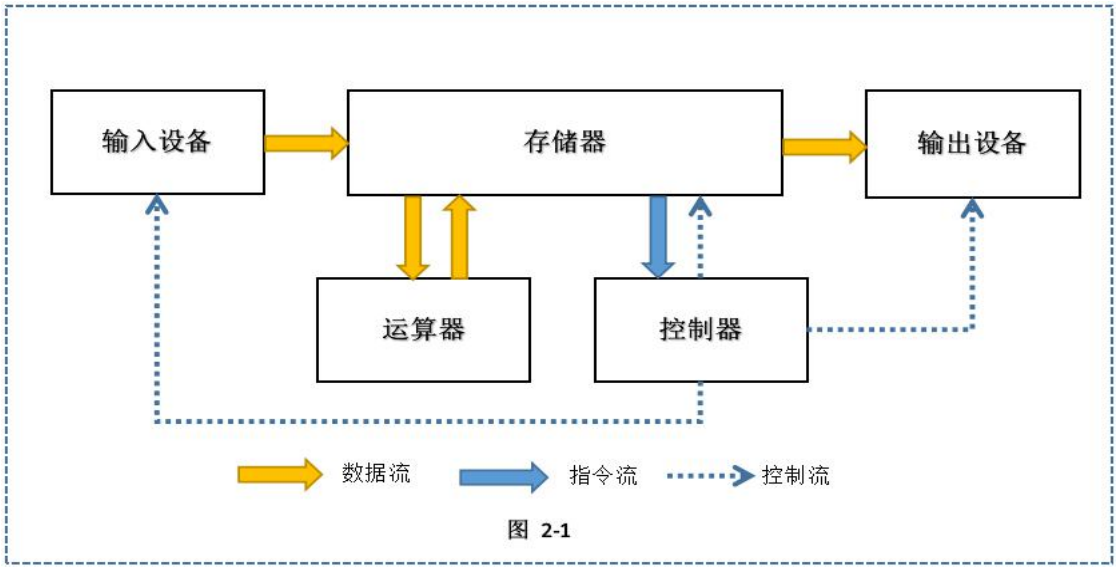


图 2-1

在详述 Simple-VM 的虚拟硬件结构之前，简要概括一下冯·诺依曼计算机体系，Simple-VM 将遵循冯·诺依曼计算机系统结构，并利用高级语言的特性完成一些仿真层面上的简化。

众所周知，冯·诺依曼体系结构是现代计算机的基础，如图 2-1 所示，其由五大模块构成，分别包括输入设备、输出设备、存储器、运算器、控制器，Simple-VM 将利用高级语言实现这些部件的仿真。

另外，在冯·诺依曼体系中有一些重要的观点[4]，它们也分别在 Simple-VM 中体现。

①采用存储程序方式，指令和数据不加区别混合存储在同一个存储器中。
②存储器是按地址访问的线性编址的一维结构，每个单元的位数是固定的。
③指令由操作码和地址组成。操作码指明本指令的操作类型，地址码指明操作数和地址。
④通过执行指令直接发出控制信号控制计算机的操作。
⑤以运算器为中心，I/O 设备与存储器间的数据传送都要经过运算器。

（二） Simple-VM 数据通路

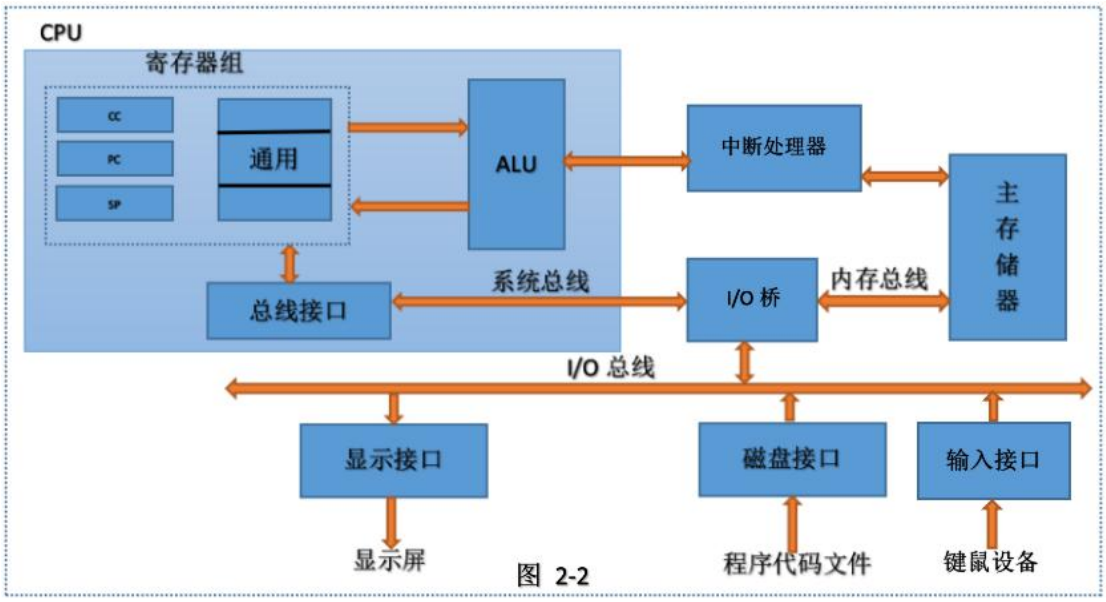


图 2-2

Simple-VM 的设计延续了冯·诺依曼计算机体系结构，图 2-2 是 Simple-VM 的数据通路，五大模块在其中都有体现。整体上，Simple-VM 由 CPU、主存储器、输入/输出设备（I/O）、中断处理器构成，其中 CPU 又由寄存器组和运算器组成，Simple-VM 的 I/O 设备主要包含磁盘、显示设备、键盘、鼠标，分别由接口程序负责与虚拟机主体交互。

（三） Simple-VM 模块化设计

我们的 Simple-VM 作为虚拟机，在宿主机内存中占有特定一部分，将其作为虚拟机的真实内存，调用系统内存如同虚拟机在使用真正的内存，如图 2-3 所示。

Simple-VM 的设计采用了模块化的理念，下面将分块介绍 Simple-VM 的中央处理器模块（包括寄存器组和运算器）、主存储器模块（包括内存分区方案）、输入/输出设备模块以及中断处理器（包括中断响应、中断处理等细节）模块。

图 2-3

（1） 中央处理器

1. Simple-VM 的中央处理器模块包括寄存器组（Registers）、逻辑算术运算单元（ALU）以及控制器（Controller）。
2. Simple-VM 的控制器是虚拟机的主体，负责执行指令、信号处理、中断处理等根本功能，基本功能说明见下表，其具体运行流程详见附录流程图 1。

功能	信号有效	信号无效
判断暂停	暂停虚拟机	继续执行
判断单步	进入单步模式	继续执行
执行指令	更新 PC	
判断中断	处理中断	继续执行
判断停机	关闭虚拟机	继续执行

3. 寄存器组包含至少 16 个寄存器（支持用户自定义），由于 Simple-VM 是 64 位虚拟机，因此寄存器组中的寄存器均能存储 64 位数据。寄存器组中含有 3 个特殊寄存器以及至少 13 个通用寄存器，其分布如下（以 16 个为例）：

编号	名称	描述
0x0	R1	通用寄存器
0x1	R2	通用寄存器
0x2	R3	通用寄存器
0x3	R4	通用寄存器
0x4	R5	通用寄存器
0x5	R6	通用寄存器
0x6	R7	通用寄存器
0x7	R8	通用寄存器
0x8	R9	通用寄存器
0x9	R10	通用寄存器
0xA	R11	通用寄存器
0xB	R12	通用寄存器
0xC	R13	通用寄存器
0xD	CC	条件码寄存器
0xE	PC	程序计数器
0xF	SP	栈指针寄存器

4. Simple-VM 拥有 3 个特殊寄存器：条件码寄存器、程序计数器、栈指针寄存器，其功能在整个虚拟机工作中弥足重要。

5. 特殊寄存器组

CC（Condition Code Register）	条件码寄存器，维护 ZC（零标志）、LC（小于标志）、GC（大于标志），供条件转移指令使用。
PC（Program Counter）	程序计数器，维护当前执行程序指令的地址。
SP（Stack Pointer）	栈指针，维护栈区的栈顶地址。

6. 运算器在 Simple-VM 是一个抽象的功能集合，其提供了执行指令的接口，能够根据内存序列以及控制器的请求完成对应指令的执行，具体细节在后文软件架构中给出。

7. 虚拟机的 CPU 为 64 位，因此将寄存器设置为能够存储 64 位数据的类型，尽管 CC 只需要 3 位就可以满足功能，但是考虑到扩展性和方便统一处理，将 CC 也定义成 64 位，其存储结构如下:

Other	Other	Other	...	...	Other	ZC	LC	GC
63	62	61	...	...	3	2	1	0
0	0	0	...	...	0	0/1	0/1	0/1

CC 低三位分别维护 ZC、LC、GC 三个条件码，用 0/1 表示状态，在具体实现中，这样的存储结构十分容易扩展到更多的条件码。

8. 关于条件码的使用，这里给出一个简单的范例，如图 2-4，利用条件码以及条件转移指令，我们可以实现诸如循环、分支语句的功能，指令具体细节请参考后文指令集架构部分。

```
0x1: IRMOV 0 9
0x2: IRMOV 0 10 // 初始化

0x3: Instructions    // Your function
0x4: INC 10 // R10 = R10 + 1
0x5: CMP 9 10 // Set CC
0x6: JNE 3
```

```
I = 0;

While (i < 10) {
        //Your function
}
```

图 2-4

（2） 主存储器

1. Simple-VM 的主存储器能够同时保存指令和数据，大小为 1MB，即 2^{20}*8 位。
2. Simple-VM 的存储单元为 64 位，即主存储器在整体上由 2^{17} 个存储单元构成，即内存地址由 0 到 2^{17}-1。
3. Simple-VM 的主存储器采用分区方式，将主存储器分为代码区（包含用户代码区、中断代码区、用户数据区、用户堆栈区、待扩展区）。下表给出分区的一些具体细节：

分区名称	功能	地址范围	备注
数据区	提供程序需要的数据读写	0x00000~0x07FFF	可读/可写
栈区	程序栈	0x08000~0x0EA60	可读/可写
待扩展区	为虚拟机提供扩展空间	0x0EA61~0x10000	由扩展功能决定
用户代码区	存储用户的代码指令序列	0x10001~0x17FFE	可读/可写
中断代码区	存储中断程序指令序列	0x17FFF~0x1FFFF	只读

4. Simple-VM 的主存储器结构如图 2-5 所示，事实上，有能力的用户可以在设置界面进行一些虚拟机的个性化配置，不过强烈建议不加修改。

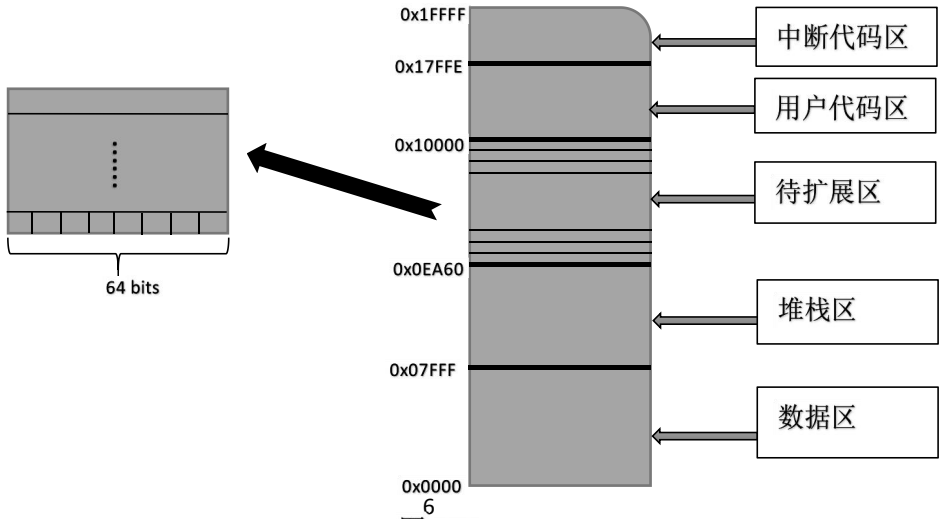


图 2-5

（3） 输入/输出设备

1. Simple-VM 的 I/O 设备主要为显示设备、磁盘、键盘和鼠标。
2. 显示设备提供结果显示、设备监控两大功能，统一由显示接口协调处理。
①显示接口：在高级语言实现中体现为函数，能够打包程序运行结果与寄存器等设备的运行时数值并交由显示设备输出。
②设备监控：为用户提供监视寄存器、查看内存的功能。
③结果显示：必要时用户代码中特定指令能够输出程序的数据。
3. 磁盘设备主要负责存储用户源代码，磁盘控制器能够将用户源代码转换成虚拟机可识别的数据结构并由 I/O 桥加载到内存中，最终由 CPU 调取执行。
4. 鼠标设备主要用于虚拟机运行时能够调控运行状态，设置单步/多步、暂停、模拟中断等。
5. 键盘设备主要用于输入指令执行时向内存输入一定数据。

（4） 中断处理器

1. Simple-VM 将中断处理独立出来，交由中断处理器负责中断的一些事项，其主要功能为根据中断信号以及 INT 中断指令、中断向量去执行预置的中断代码。
2. Simple-VM 采用单级中断方式[1]，中断信号分为中断出现信号以及中断允许信号，当设备向虚拟机发出中断信号（这里用户可通过点击中断按钮选取中断程序来模拟）时，并且中断允许信号有效时，中断出现信号置为有效且将中断允许信号置无效；当中断程序处理完毕后，中断出现信号置为无效，中断允许信号置为有效，从而保证中断时不应答另一个中断信号。在执行中断程序前后，分别需要进行保存现场以及恢复现场操作，Simple-VM 在保存现场时将寄存器信息全部压入用户堆栈，在恢复现场时再将这些寄存器信息出栈，返回对应的寄存器中。
3. 关于中断处理器的大致结构如图 2-6。

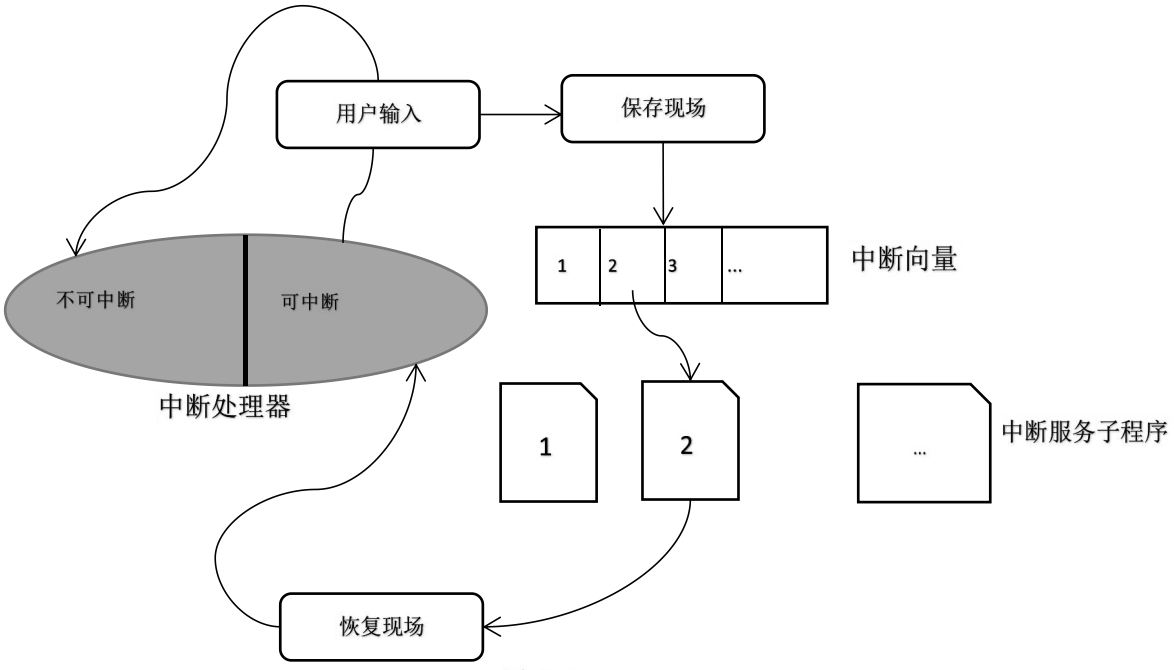


图 2-6

三、指令集架构

（一）操作数与寻址模式

1. 操作数[2]：Simple-VM 的操作数一共有三种，分别是立即数、寄存器编号、内存地址。

立即数	64 位数据，用 I 表示
寄存器	寄存器编号，用 R 表示
内存	内存地址，用 M 表示

2. 操作数组合类型：
   ①零操作数，如 HLT。
   ②单操作数，如 INC、DEC。
   ③双操作数：

RR	寄存器到寄存器
RM	寄存器到内存
MR	内存到寄存器
IR	立即数到寄存器
IM	立即数到内存

3. 寻址模式：虚拟机采用间接寻址，例如 RM 类型双操作数指令，M（内存地址）是通过索引寄存器数据来获得的。

（二）指令格式

1. Simple-VM 的指令在用户层面是不定长指令（即用户编写时不需要针对零操作数指令、单操作数指令进行无意义的填充），但是在虚拟机层面仍然是定长指令（虚拟机预处理模块会自动扩展指令到双操作数指令）。
2. Simple-VM 的大部分指令是单字长指令，即占 64 位，分别由指令名（insName）、源操作数（S）、目的操作数（D）构成。

insName	**op1：S**	**op2：D**
63-58	**57-29**	**28-0**

3. Simple-VM 指令集中涉及到立即数的指令，例如 IRMOV 指令，则是双字长指令，因为 Simple-VM 是 64 位机器，用户可以进行 64 位数据运算，单字长指令无法提供足够的加载空间。

InsName	**op2：D**
127-122	**121-64**
op1：S	
63-0	

4. 值得注意的是，Simple-VM 充分利用了高级语言的特性，并未对用户输入指令进行二进制解码译码操作，这是因为在高级语言实现中，Simple-VM 采用了结构体数组而非字节数组的方式存储指令代码。然而这并不意味 Simple-VM 忽略了实验原本的目的性，因为 Simple-VM 指令集不超过 26 个指令，指令名在其二进制表示中不超过 6 位，操作数设置为 6 位是合理的，尽管我们忽略了字符串形式的指令到二进制位形式的存储方式转化，但是在观点上，Simple-VM 仍然遵循了冯·诺依曼体系中的内存结构。

（三）指令集说明

（1）指令运行框架

Simple-VM 执行指令严格延续冯·诺依曼体系结构，根据程序计数器从内存中调用指令[3]，并针对不同的指令采取不同的执行流程，大体框架如图 3-1，具体到指令的执行流程在后文也有详述。

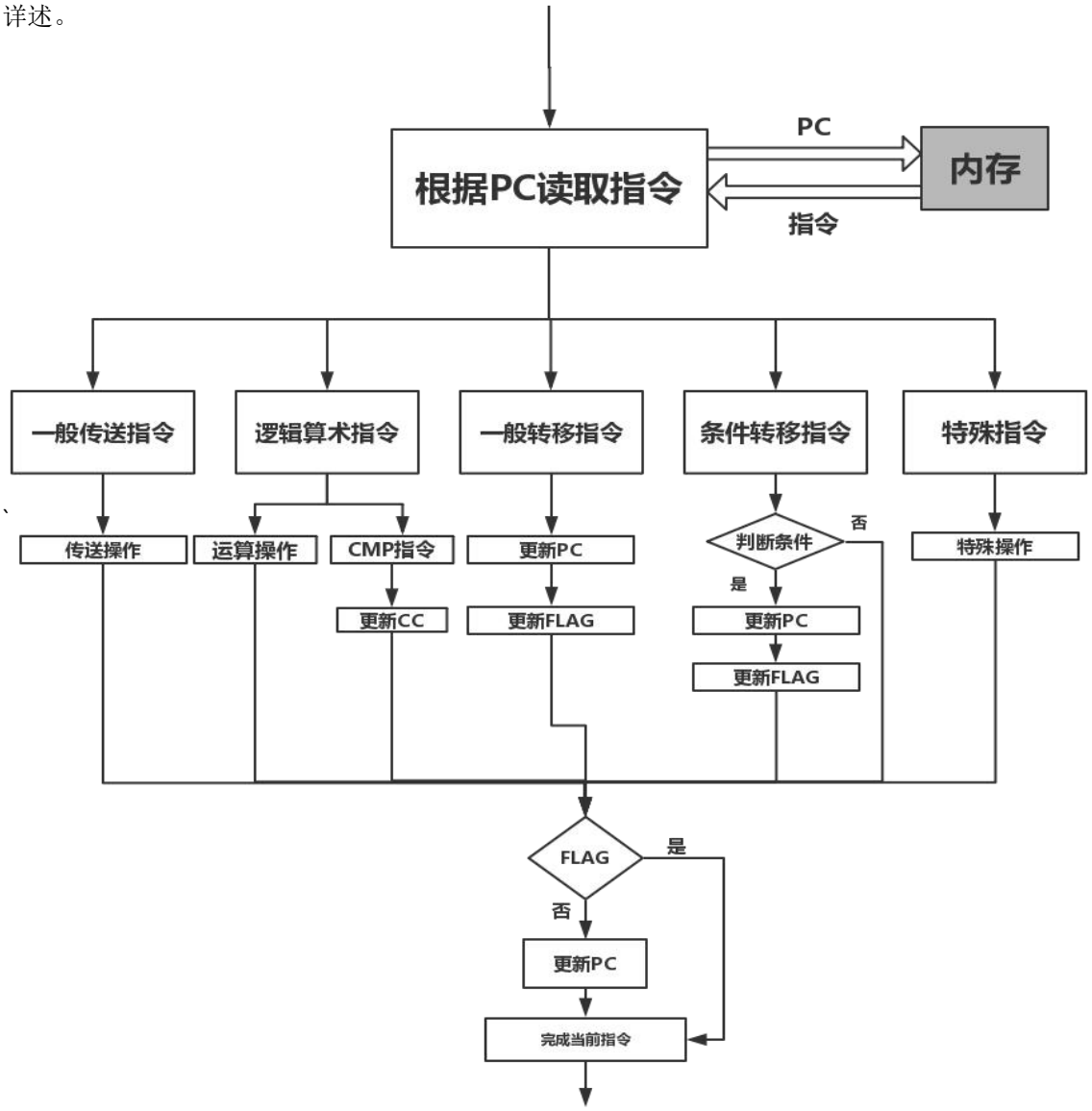
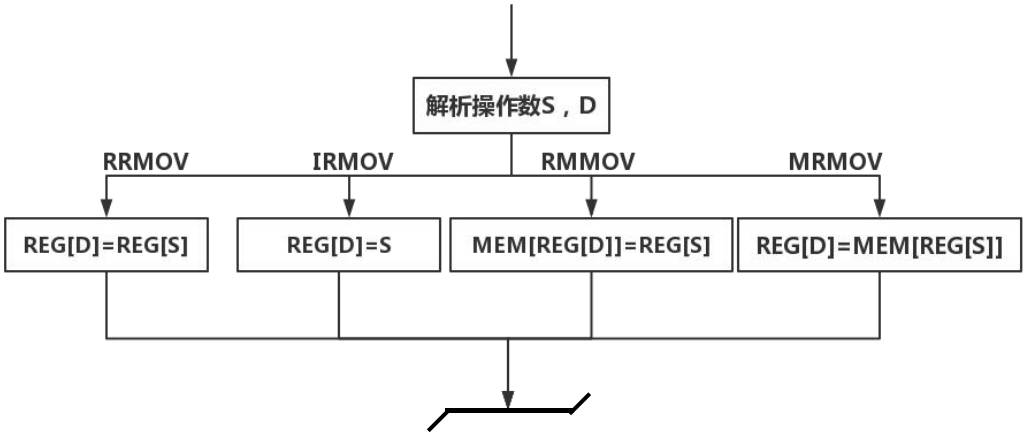


图 3-1

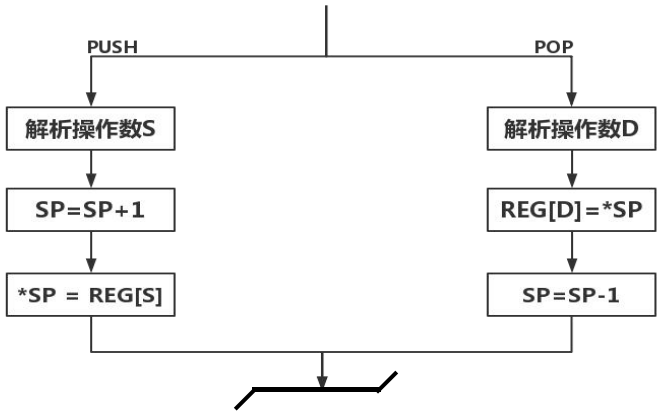
（2）一般传送指令

指令名称	操作数类别	功能说明
RRMOV S,D	RR	寄存器到寄存器一般传送，D<---S
IRMOV S,D	IR	立即数到寄存器一般传送，D<---S
RMMOV S,D	RM	寄存器到内存一般传送，D<---S
MRMOV S,D	MR	内存到寄存器一般传送，D<---S



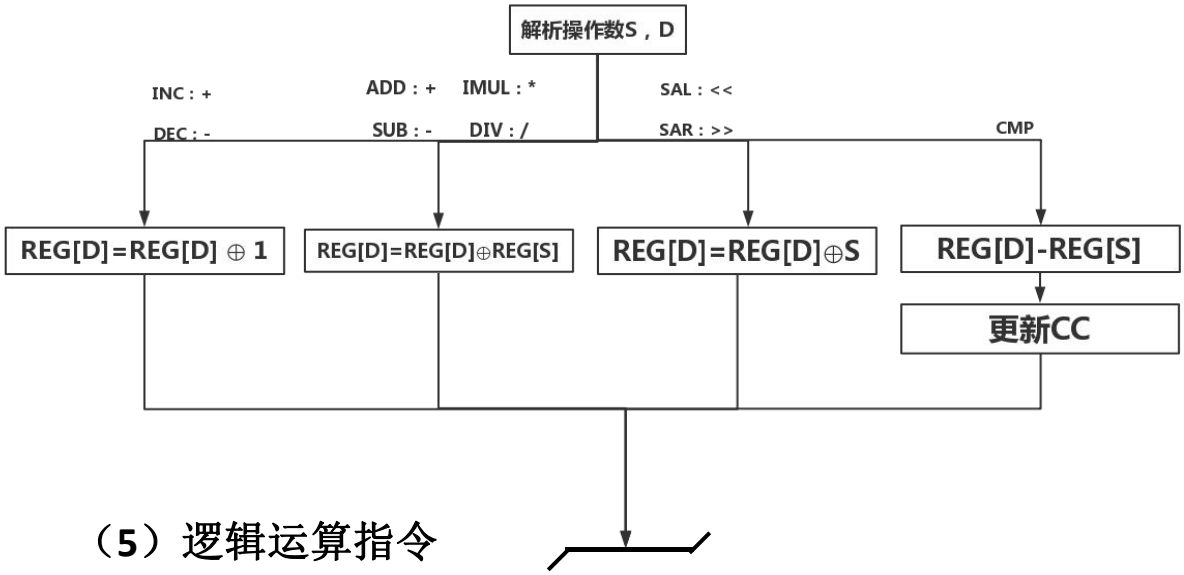
（3）堆栈操作指令

指令名称	操作数类别	功能说明
PUSH S	R	入栈，SP++,*SP<---S
POP D	R	出栈，D<---*SP,SP--



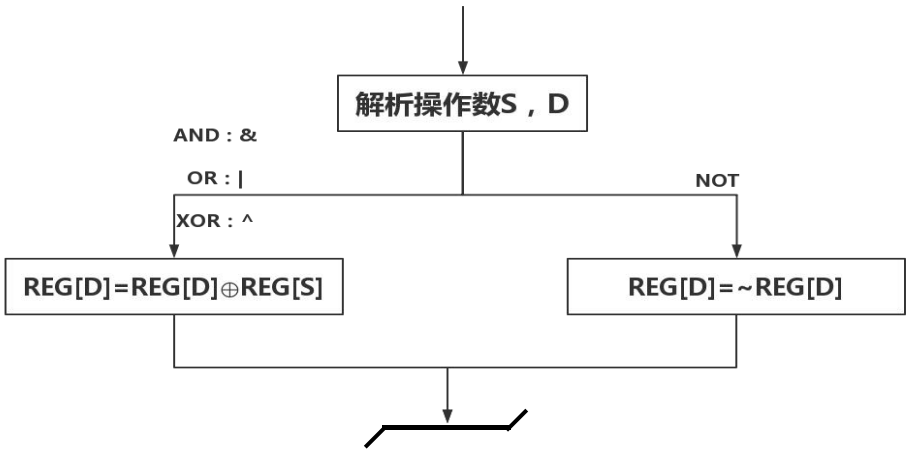
（4）算术运算指令

指令名称	操作数类别	功能说明
INC D	R	自增，D<---D+1
DEC D	R	自减，D<---D-1
ADD S,D	RR	加，D<---D+S
SUB S,D	RR	减，D<---D-S
IMUL S,D	RR	乘，D<---D*S
DIV S,D	RR	除，D<---D/S
CMP S1,S2	RR	比较，S2-S1
SAL k,R	IR	左移，D<---D << k
SAR k,R	IR	右移，D<---D >> k



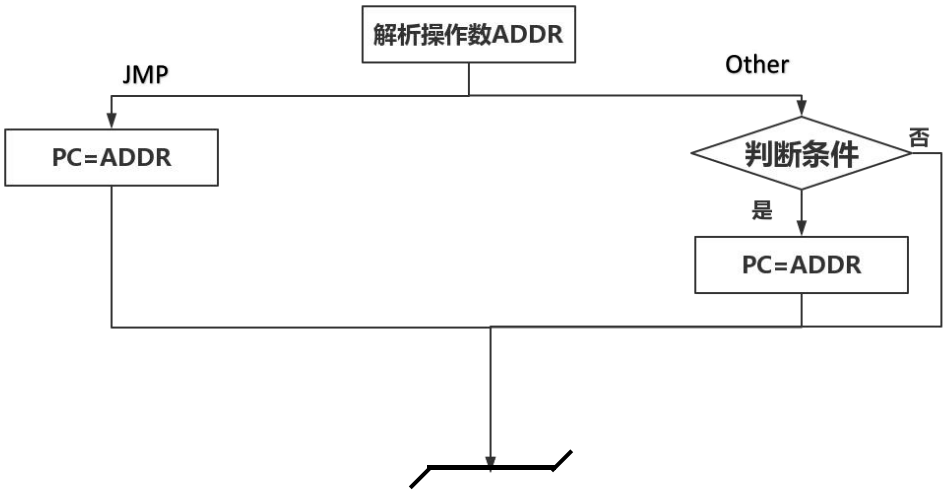
（5）逻辑运算指令

指令名称	操作数类型	功能说明
NOT D	R	逻辑非，D<---!D
XOR S,D	RR	逻辑异或，D<---D^S
OR S,D	RR	逻辑或，D<---D|S
AND S,D	RR	逻辑与，D<---D&S



（6）转移指令

指令名称	操作数类型	功能说明
JMP ADDR	I	无条件转移，PC<---ADDR
JE ADDR	I	条件转移，当 ZC=1 时，PC<---ADDR
JNE ADDR	I	条件转移，当 ZC=0 时，PC<---ADDR
JLE ADDR	I	条件转移，当 ZC=1 或 LC=0 时，PC<---ADDR
JL ADDR	I	条件转移，当 LC=0 时，PC<---ADDR
JGE ADDR	I	条件转移，当 GC=0 或 ZC=0 时，PC<---ADDR
JG ADDR	I	条件转移，当 GC=0 时，PC<---ADDR



（7）特殊指令

指令名称	操作数类型	功能说明
HLT	NULL	停机指令
ROUT S	R	寄存器输出指令，显示设备<---S
INPUT D	R	内存输入指令，[D]<---用户输入
IRET	NULL	中断结束指令
INT S	I	调用中断服务子程序（由用户调用）
NOP	NULL	空指令

Simple-VM 中的特殊指令较为简单，其中 IRET 指令是中断结束指令，负责向虚拟机发送恢复现场的信号；输入输出指令分别能够输入数据到内存以及输出寄存器的数字。

（四）软件设计

（1） 总体框架

本实验的核心在于使用高级语言完成这些硬件上的功能，Simple-VM 采用 PyGtk[5]作为开发语言，简洁而不失完整性地实现了虚拟机所需要的基本功能。

在开发过程中，Simple-VM 以 GUI 窗口为导向，在窗口部件的事件函数中加入虚拟机的逻辑处理部分，在窗口设计中彰显 Python 的极简主义，界面小而功能完备，力求用最小的窗口展现更多的信息。

Simple-VM 采取了多线程开发技术，将图形界面和逻辑处理界面分离开来，独自运行，并通过接口将逻辑处理的结果交由图形界面显示。首先，让我们来看一下 Simple-VM 的设计框架图，如图 4-1。

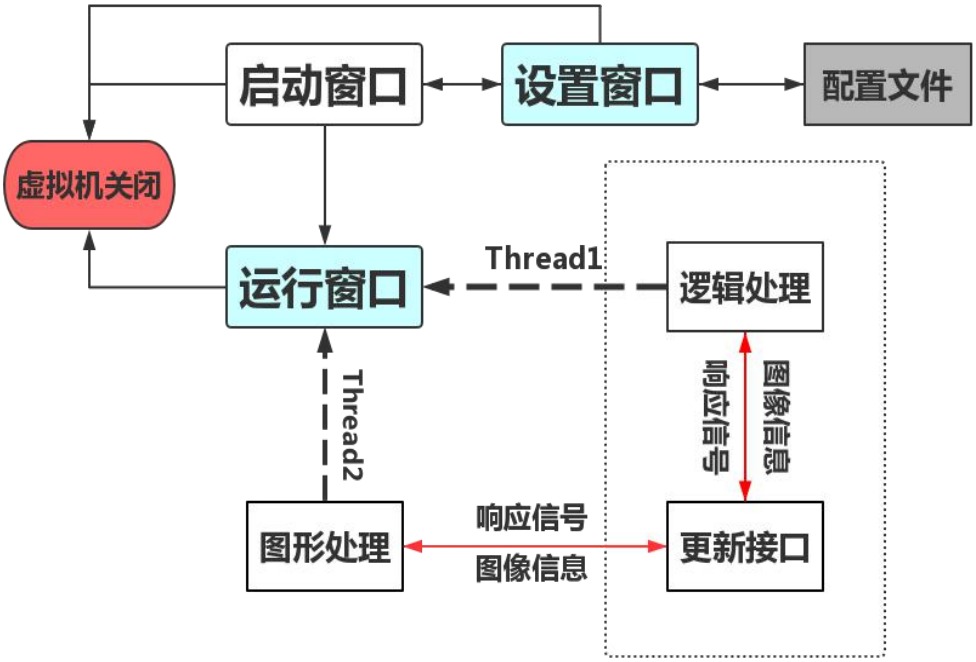


图 4-1

从框架图中，我们可以看到，Simple-VM 分别由三大窗口构成：启动窗口、设置窗口、运行窗口，其中运行窗口是 Simple-VM 的核心。在启动窗口中，用户能够点击设置按钮进入设置窗口或者点击加载按钮选择指令流执行；在设置窗口，用户能够选择一定的设置选项进行一些个性化配置，这些个性化配置由配置文件维护；在运行界面，用户能够观察虚拟机的运行状态以及进行一些虚拟机功能的选择。逻辑处理和图形处理分别是 Simple-VM 的两大线程，逻辑处理主要负责虚拟机主体的运行、指令执行、接收信号并响应等，两者由更新接口负责通信。接下来将对窗口功能分别进行详述。

（2） 数据结构与信号量

I. 寄存器：Simple-VM 中，寄存器是一个 List，其容量与特殊寄存器位置由一组常量维护，这些常量由配置文件给出。一些相关变量见下表：

NUM_REG	寄存器数量
SP_REG	栈指针寄存器位置
PC_REG	程序计数器寄存器位置
CC_REG	条件码寄存器位置

II. 内存：Simple-VM 充分利用了 Python 语言强大的字符处理功能，忽略了字符指令与二进制位之间的解码译码过程，而是利用了结构体（哈希表）直接将指令信息存在内存之中，因此 Simple-VM 的内存需要分为代码区和数据区。

代码区由 Python 内置的字典类型数组构成，数组元素是一条指令字典，例如“RRMOV 1 2”指令在 Simple-VM 中是**{'insName':'RRMOV','op1':'1','op2':'2'}**。

数据区由普通数据构成的 List 构成，数组元素在设计上是 64 位数据，但是由于 Python 动态变量的机制，在 Python 编译器的机制上无法给出这种信息，不过 Simple-VM 在设计上、测试程序的编写上是绝对不会突破设计方案的限制的。

尽管代码区和数据区在空间上并非连续，但是在 Simple-VM 的显示区中，我们仍然将其按照连续地址的方式展示，具体实现方式是在代码区的源地址上加上一个数据区大小的偏移量，从而实现这个显示上的连续。

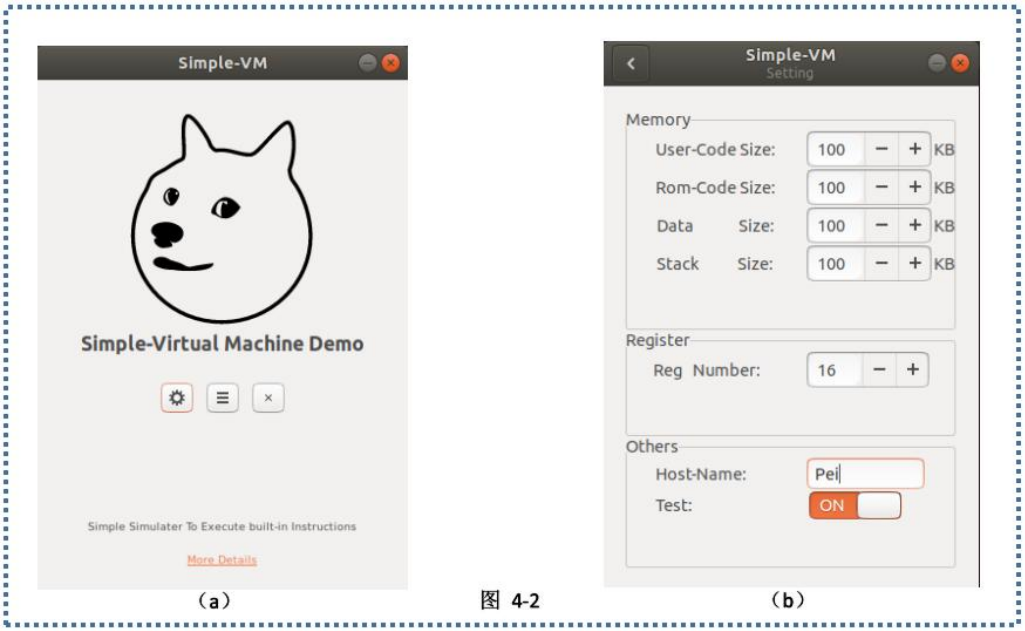
关于内存，也有一些常量与之相关，在此列出：

LEN_MEM_DATA	数据区长度（包含堆栈区）
LEN_MEM_CODE	代码区长度（包含用户代码和中断代码）
POS_INT_START	中断代码起始位置
POS_INT_END	中断代码结束位置
POS_STACK	堆栈区位置

III. 信号量：Simple-VM 运行过程中各部件之间通过各种通信信号进行交互以完成虚拟机的不同功能，这些信号量列在下表中：

signal_run	虚拟机是否运行信号
signal_pause	虚拟机是否暂停信号
signal_onestep	虚拟机是否单步执行信号
signal_int	虚拟机中断出现信号
mutex_one	虚拟机单步执行控制信号
frequency	虚拟机运行频率
in_int	虚拟机中断进行信号
in_end	虚拟机是否停机信号
error_xxx	虚拟机异常出现信号
INT_VECTOR	虚拟机中断向量
FLAG_JMP	虚拟机指令跳转信号

（3） 启动窗口与设置窗口



（a） 图 4-2 （b）

A. 界面设计

Simple-VM 的启动界面简洁大方，如图 4-2（a）所示，主要功能按钮有三个，分别作为“设置键”、“加载键”、“关机键”，图标表意明显。下方对 Simple-VM 的概述“Simple Simulater To Execute built-in instructions”（一个运行内置指令的模拟器）也表明了 Simple-VM 的功能，界面的底端拥有一个超链接按钮，供用户查看更详细的用户手册。

Simple-VM 的设置界面同样不过多缀饰，采用 SpinButton 作为接收用户设置信息的控件，用户能够进行内存分配、寄存器数量的个性化配置，以及虚拟机名等一些边缘配置（无实际作用），但我们建议不加修改，而采取默认配置。

B. 功能设计

Simple-VM 的启动界面功能设计主要是图形窗口类的切换显示，这里采取了在某窗口类的事件响应成员函数中添加当前窗口的关闭与新窗口的启动功能来完成，其核心伪代码如下：

```
class StartWindow(Gtk.Window):
    # 窗口布局信息
    # 某功能按钮负责切换窗口，例如设置界面的返回键，启动界面的加载键
    buttonX.connect("clicked",self.on_buttonX_clicked)
    self.show_all()

    def on_buttonX_clicked:
        self.destroy() # 关闭当前窗口
        newW = newWindow() # 建立新的窗口
        Gtk.main() # 进入图形界面主循环
```

而设置界面的主要功能在于获取用户设置的信息然后与配置信息交互，是基本的文件读取操作，不加赘述。

关于加载程序是虚拟机较为重要的一步，Simple-VM 选择了图形界面，能够可视化地供用户选择对应文件，在软件实现中，用户选择文件这一步实际上是向虚拟机加载子函数传递一个文件名参数，然后 Simple-VM 能够利用文件名链接到该文件并进行预处理、加载至内存等操作。

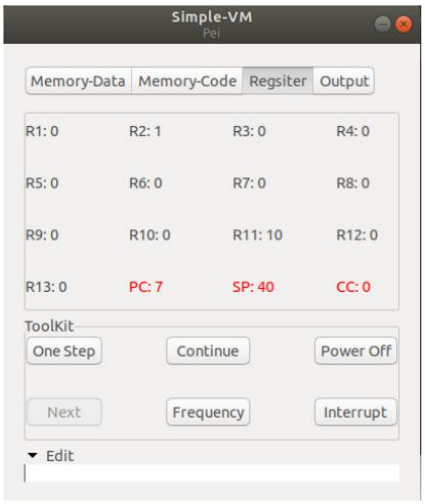
I. **预处理程序**，Simple-VM 由于能够支持用户对代码进行注释以及用户层的不定长指令，所以预处理程序的主要功能是将用户注释信息删除、自动扩展零操作数、单操作数指令到双操作数指令（通过补充 NULL 字段）。其核心代码如下：

```
def VM_preprocess(self, src_filename,new_filename): # 分别是源文件以及预处理之后的文件
    file = open(src_filename,"r")
    newF = open(new_filename,"w")
    new_file = ''
    for line in file.readlines():
        line = line.strip()
        # 处理用户注释信息：1. 获取注释符位置； 2：删除该位置之后的内容
        pos_comment = line.find("//")
        if (pos_comment != -1):
            line = line[0:pos_comment - 1]
        ins = re.compile('\s+').split(line)
        # 扩展指令：1.判断操作数个数； 2. 根据操作数个数补充NULL字段
        if (len(ins) == 1):
            ins.append('NULL')
            ins.append('NULL')
        elif (len(ins) == 2):
            ins.append('NULL')
        new_file = new_file + " ".join(ins) + '\n'
    newF.write(new_file)
    file.close()
    newF.close()
```

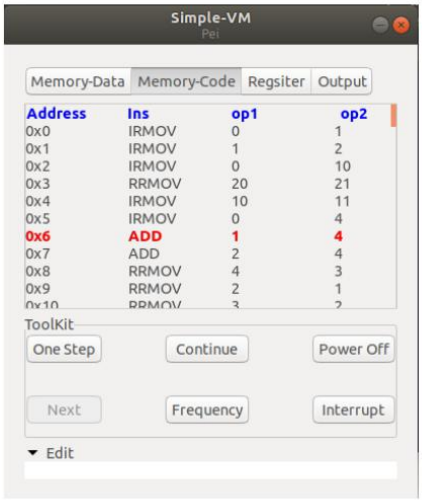
II. **加载程序**，Simple-VM 遵循冯诺依曼体系结构，首先需要将用户汇编代码转换成虚拟机可以执行的格式并保存至内存中，然后虚拟机按照一定的序列取出该代码段并执行。Simple-VM 充分利用了高级语言的特性，忽略了字符串与二进制位的解码译码工作，在内存中，代码用结构体存储，结构体定义在上文已有提及，加载程序的主要流程就是读取文本信息、转换至字典信息、保存至内存，其核心代码如下：

```
def VM_load(filename):
    global MEM_CODE    # MEM_CODE是内存中的代码区，本质上是哈希数组
    file = open(filename,"r")
    insNum = 0
    for line in file.readlines():
        # 将文本信息分割并保存至List中
        temp_ins = line.strip().split(' ')
        # 构造新的结构体（哈希表），存储当前指令信息
        ins = {'insName':None,'op1':None,'op2':None}
        ins['insName'] = temp_ins[0].upper()
        ins['op1'] = temp_ins[1]
        ins['op2'] = temp_ins[2]
        # 将该指令加载到内存中
        MEM_CODE[insNum] = ins
        insNum = insNum + 1
```

（4） 运行窗口



(a)



(b)

图 4-3

A. 界面设计

Simple-VM 的显示区分别能显示内存、寄存器、输出信息（来自程序或虚拟机），其中内存包含数据区/堆栈区、代码区两大部分，采取不同的方式显示（数字/字符串）。

Simple-VM 的主运行界面充分体现了设计的简洁感，利用 Stack 页式界面在小窗口中显示更多的信息，图 4-3(a)和图 4-3(b)分别展示了 Simple-VM 寄存器页和内存代码区的情况，Simple-VM 在显示区设计了不同的细节以展示不同的信息，红色高亮了特殊寄存器以及当前运行的指令，中断代码区则用了截然不同的颜色表示，这样用户能更好的区分信息。

Simple-VM 运行窗口提供了一组控制按钮，其功能见下表：

One Step	点击则进入单步执行模式，此时 Frequency 按钮无效。
Next	多步执行模式下无效，单步执行模式下点击则控制下一条代码执行。
Continue/Pause	切换虚拟机暂停/继续状态。
Frequency	调节虚拟机运行频率，利用 Pop Over 按钮进行二级调节。
Power Off	关闭虚拟机。
Interrupt	用户主动触发中断，利用 Pop Over 按钮进行二级选择。

Simple-VM 运行窗口的底端是 Edit Entry 控件，当代码需要输入数据时，用户可以在其中输入合法的数据供代码使用，由于装在 Expander 类中，当用户不使用时，该控件能够隐藏，从而再次提升 Simple-VM 的简洁性。

B. 功能设计

如图 4-1 所示，运行窗口是 Simple-VM 的核心部分，它由逻辑处理部分进行驱动，并交由图像处理部分。逻辑处理部分主要是虚拟机的运行主体（控制器）运作中产生的一系列数值变换，图像处理部分捕获这些变换并将其显示在窗口中。

I. **初始化程序**：本虚拟机启动伊始，会进行设备初始化，例如寄存器置零等，其核心代码如下：

```
def VM_init():
    global LEN_MEM_DATA,LEN_MEM_CODE,MEM_DATA,MEM_CODE,REG,PC_REG,SP_REG,FLAG_JMP
    # 初始化数据区为0；
    for i in range(LEN_MEM_DATA):
        MEM_DATA.append(0)
    # 初始化用户代码区为None；
    for i in range(LEN_MEM_CODE):
        MEM_CODE.append({'insName':None,'op1':None,'op2':None})
    # 预置中断代码区为intFile的内容；
    intFile = open("intFile.txt","r").readlines()
    for i in range(POS_INT_START,POS_INT_END + 1):
        temp = intFile[i - POS_INT_START]
        MEM_CODE.append({'insName':temp[0],'op1':temp[1],'op2':temp[2]})
    # 初始化寄存器为0，栈指针寄存器为堆栈初始位置-1；
    for i in range(NUM_REG):
        REG.append(0)
    REG[PC_REG] = 0
    REG[SP_REG] = POS_STACK - 1;
    #初始化跳转信号为False；
    FLAG_JMP = False
```

II. **运行主体程序**：在初始化完毕之后，虚拟机逻辑处理部分的主循环就会进行执行指令的步骤，附录中流程图 1 有详细的流程说明，其核心代码如下：

```
def VM_run():
    currentIns = MEM_CODE[REG[PC_REG]]
    while (currentIns['insName'] != 'HLT' and not is_end):
        GLib.idle_add(update) # 阻塞更新接口线程
        while (signal_onestep and not mutex_one): # 处理单步执行信号
            print("")
            if (is_end): # 处理中途关闭虚拟机
                break
        mutex_one = False
        while (signal_pause): # 处理暂停信号
            print("")
            if (is_end): # 处理中途关闭虚拟机
                break
        if (not signal_onestep and not signal_pause): # 处理延时（频次）
            time.sleep(frequency)
        ins_exec(currentIns) # 执行指令
        if (not FLAG_JMP): # 根据FLAG_JMP更新PC并取指
            REG[PC_REG] = REG[PC_REG] + 1
        else:
            ins_exec(MEM_CODE[REG[PC_REG] - 1])
        currentIns = MEM_CODE[REG[PC_REG]]; FLAG_JMP = False
        if (signal_int and not in_int): # 判断中断信号，若为真，则进行中断处理
            save_scene() # 保存现场，恢复现场由中断子程序结尾的IRET执行
            # 重置中断现场：通用寄存器、CC置零，SP置上一次的SP+REG_NUM+1位置，PC置中断代码起始位置
            for i in range(len(REG)):
                REG[i] = 0
            REG[SP_REG] = sp + len(REG) + 1; REG[PC_REG] = INT_VECTOR[INT_NUMBER]
```

III. **指令执行程序**：Simple-VM 的指令执行程序模拟了冯·诺依曼计算机体系中的运算器部分，由于是基于高级语言的虚拟机，并未完全按照传统的 ALU 芯片那样处理数据，而是直接利

用高级语言的运算方式进行运算，由于指令过多，指令执行程序较为庞大，这里抽取其中部分指令作为核心代码展示，了解 Simple-VM 的指令执行模式即可：

```
def ins_exec(Ins):
    global REG,MEM_DATA,SP_REG,PC_REG,FLAG_JMP,CC_REG,error_over
    # 解析指令参数
    insName = Ins['insName']
    op1 = Ins['op1']
    op2 = Ins['op2']
    # 依照if-elif的方式寻找特定指令执行，若无对应指令，则抛出“指令名错误”异常
    if (insName == 'RRMOV'):
        [s,d] = [int(op1), int(op2)]
        # 若寻址超出寄存器数量或内存范围，则抛出“寻址错误”异常
        if (d > len(REG) or s > len(REG)):
            error_over = True
        else:
            REG[d - 1] = REG[s - 1]
    elif (insName == "NOP"):
        pass
```

IV. **保存现场程序：**Simple-VM 在处理中断程序之前需要进行保存现场，即将寄存器信息与上一次栈指针压入栈，其核心代码如下：

```
def save_scene():
    global MEM_DATA,REG,SP_HISTORY,SP_REG,in_int
    # 执行PUSH指令，将寄存器信息一一压入堆栈
    for i in range(len(REG)):
        ins = {};
        ins['insName'] = "PUSH"; ins["op1"] = i + 1; ins["op2"] = "NULL"
        ins_exec(ins)
    # 设定中断进行信号为有效
    in_int = True
```

V. **恢复现场程序：**Simple-VM 在中断程序结束之后由 IRET 调用恢复现场程序，将寄存器值恢复到中断之前，其核心代码如下：

```
def recover_scene():
    global MEM_DATA,REG,SP_HISTORY,SP_REG,signal_int,in_int,last_sp
    # 设定中断出现信号无效
    signal_int = False
    # 设置中断前栈顶指针，因为Simple-VM采取单极中断，因此只需要一个变量即可
    sp = last_sp
    # 将寄存器值恢复到之前的值
    for i in range(len(REG)):
        REG[i] = MEM_DATA[sp+i]
    # 设定中断进行信号为无效
    in_int = False
```

VI. **更新接口程序：**Simple-VM 的更新接口程序包含在逻辑处理部分，接口程序中存有运行窗口类中的控件，当这些控件值发生变化时，接口程序将这信息反馈给图形处理程序，从而显示出来，其核心代码如下（控件数量较多，这里用伪代码展示其主要过程）：

```
def update():
    regs = running_window.regs
    mem_data = running_window.mem_data # 窗口类data区控件
    mem_code = running_window.mem_code # 窗口类code区控件
    global REG,MEM_DATA,MEM_CODE,PC_REG,SP_REG,CC_REG,INT_START,INT_END,error_over
    if (error_XXX):
        # 若异常信息有效，在显示区输出对应异常信息
    # 更新寄存器区
    for i in range(len(regs)):
        # 更新寄存器信息
    # 更新数据区
    for i in range(1,len(MEM_DATA)+1):
        # 判断mem_data[i]是否修改，若修改则进行更新
    # 更新代码区
    for i in range(1,len(MEM_CODE)+1):
        # 判断mem_code[i]是否修改，若修改则进行更新
```

（5） 软硬件映射关系总结

至此，Simple-VM 设计方面的一些事宜已经交待完毕，在这一节，我们总结一下 Simple-VM 中一些软硬件映射关系，以帮助用户更好地理解 Simple-VM。

硬件结构	软件结构
键盘	未设置缓冲区，需要特定指令配合进行数据输入
鼠标	图形化界面按钮的操作等
控制器	VM_run()函数，负责虚拟机整体调控
运算器	Ins_exec()函数，负责模拟各种运算
寄存器	REG[] List
主存储器	哈希表数组 MEM_CODE，以及普通数组 MEM_DATA

（五）软件测试

（1）指令完备性测试

A. 测试用例

Simple-VM 的指令集可分为传送指令、运算指令、无条件转移指令、条件转移指令、特殊指令，指令完备性测试旨在编写能够包含这五大部分的指令的测试代码并顺利让 Simple-VM 通过测试。

本测试用例的功能是根据用户输入的上限进行斐波那契数列的计算，并保存到数组中，最后在屏幕输出数组内容，涉及到的指令及其功能列在下表：

指令类别	指令名	指令功能
传送指令	**IRMOV**	向寄存器写立即数
	RRMOV	寄存器转移值
	MRMOV	内存向寄存器传送
	RMMOV	寄存器向内存传送
运算指令	**INC**	自增
	DEC	自减法
	ADD	加法
	CMP	比较指令
条件转移指令	**JL**	条件转移，LT 为判断位
输入指令	**INPUT**	输入指令
输出指令	**OUTPUT**	输出指令
堆栈指令	**PUSH**	压栈操作
	POP	出栈操作

```
0: IRMOV 10 5
1: PUSH 5
2: POP 11
3: IRMOV 2 8 //R[5] = &a[0]; R[8] = &n
4: IRMOV 2 7 //R[7] = i = 2
5: IRMOV 0 1
6: IRMOV 1 2 // R[1] = 0; R[2] = 1;
7: INPUT 8
8: MRMOV 8 6 // R[6] = n
9: RMMOV 1 5
10: INC 5
11: RMMOV 2 5
12: DEC 5    // a[0] = r1; a[1] = r2
13: IRMOV 0 4
14: ADD 1 4
15: ADD 2 4
16: RRMOV 4 3    //r3 = r1 + r2
17: RRMOV 2 1    // r1 = r2
18: RRMOV 3 2    // r2 = r3
19: IRMOV 0 9
20: ADD 5 9
21: ADD 7 9
22: RMMOV 3 9 //a[base + i] = r[3]
23: INC 7
24: CMP 6 7
25: JL 13
26: IRMOV 0 7 // i = 0
27: IRMOV 0 9
28: ADD 5 9
29: ADD 7 9
30: MOUT 9
31: INC 7
32: CMP 6 7
33: JL 27
34: HLT
```

用于测试 Simple-VM 指令完备性的测试程序如左图所示，当删除行号后，Simple-VM 可以直接加载运行。

正如上表所述，这段测试程序功能虽然简单，但涉及了 Simple-VM 指令集各个部分，可以作为指令完备性测试的例程。

测试代码的主要功能是根据用户输入的 n，生成长度为 n 的斐波那契数列数组存于内存之中并在程序末尾打印出来，在这段程序伊始，测试程序进行了堆栈操作，测试入栈和出栈的指令有效性，分别是第 1 行和第 2 行。

让我们看一下对应的 C 语言伪代码：

```c
int *a,n,i=2;
int r1 = 0,r2 = 1,r3;
scanf("%d",&n);
a[0]=r1; a[1]=r2;
while(i < n) {
    r3 = r1 + r2;
    r1 = r2;
    r2 = r3;
    a[i] = r3;
    i = i + 1;
}
while(i < n) {
    printf("%d",a+i);
    i = i + 1;
}
```

根据左图的的注释，代码正确性没有问题。可以预见，当代码执行完毕，Simple-VM 的内存区存在 n 长度的斐波那契数列，而 Simple-VM 的显示区同样会输出 n 长度的斐波那契数列数组。

B. 结果分析

指令	指令功能	运行结果
IRMOV 10 5	将立即数存入寄存器	R5 数值置为 10
PUSH 5	将寄存器数压栈	栈顶元素为 5，栈指针加一
POP 11	出栈至寄存器	R11 数值置为栈顶元素 5，栈指针减一
IRMOV 2 8	将立即数存入寄存器	R8 数值置为 2
IRMOV 2 7	将立即数存入寄存器	R7 数值置为 2
IRMOV 0 1	将立即数存入寄存器	R1 数值置为 2
IRMOV 1 2	将立即数存入寄存器	R2 数值置为 1
INPUT 8	用户输入数值至内存	用户输入的 n 存入 R8 数值对应的内存地址
MRMOV 8 6	将内存数存入寄存器	R6 数值置为内存 2 号位置的数
RMMOV 1 5	将寄存器数存入内存	R5 数据对应的内存地址存入 R1 数值
INC 5	寄存器数自增	R5 自增
RMMOV 2 5	将寄存器值存入内存	R5 数据对应的内存地址存入 R2 数值
DEC 5	寄存器数自减	R5 自减
IRMOV 0 4	将立即数存入寄存器	R4 数值置为 0
ADD 1 4	寄存器数相加	R4=R4+R1
ADD 2 4	寄存器数相加	R4=R4+R2
RRMOV 4 3	寄存器数交换	R3=R4
RRMOV 2 1	寄存器数交换	R1=R2
RRMOV 3 2	寄存器数交换	R2=R3
IRMOV 0 9	将立即数存入寄存器	R9 数值置为 0
ADD 5 9	寄存器数相加	R9=R9+R5
ADD 7 9	寄存器数相加	R9=R9+R7
RMMOV 3 9	将寄存器数存入内存	R9 数据对应内存地址存入 R3 数值
INC 7	寄存器数自增	R7 自增
CMP 6 7	比较寄存器数	比较 R6 和 R7，更新 CC
JL 13	条件转移（小于）	若 CC 为 010，则跳转，模拟循环
IRMOV 0 7	将立即数存入寄存器	R7 数值置为 0
IRMOV 0 9	将立即数存入寄存器	R9 数值置为 0
ADD 5 9	寄存器数相加	R9=R9+R5
ADD 7 9	寄存器数相加	R9=R9+R7
MOUT 9	将内存数输出	输出 R9 数值对应的内存位置数据
INC 7	寄存器数自减	R7 自增
CMP 6 7	比较寄存器数	比较 R6 和 R7，更新 CC
JL 27	条件转移	若 CC 为 010，则跳转，模拟循环
HLT	停机指令	关闭虚拟机

测试代码在Simple-VM上的运行结果如上表所列，可见Simple-VM能够顺利完成指令的功能，在图形化上 Simple-VM 也能保持正确性，如图 5-1，(a)(b)分别是运行结束之后的内存状态和输出状态。

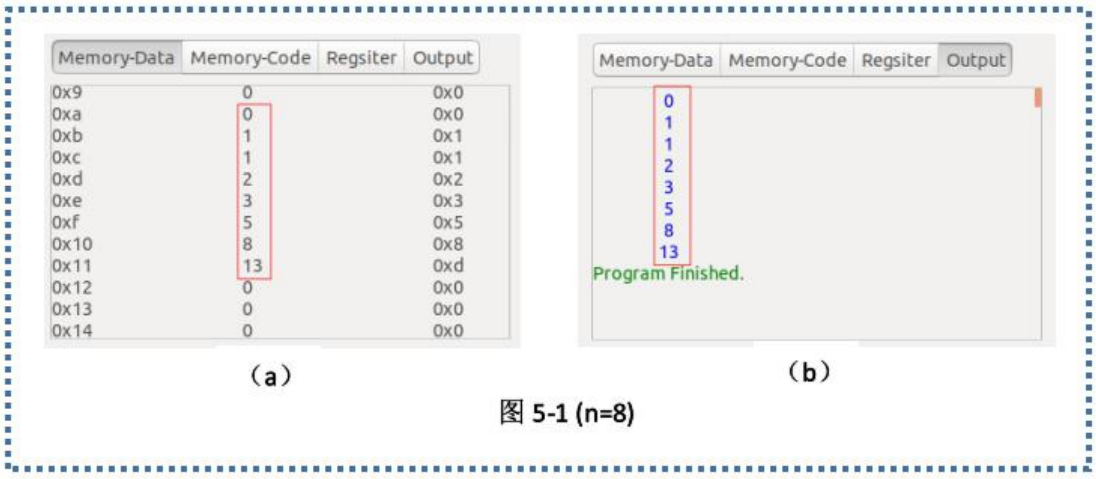


图 5-1 (n=8)

（2）中断处理测试

Simple-VM 支持单级中断，能够让用户（模拟外部设备）在虚拟机运行当前指令时跳转到新的中断服务子程序中，同时 Simple-VM 能够在中断前将寄存器信息记录，即保存现场，在中断程序运行完毕之后，虚拟机负责恢复现场。注意的是，中断服务子程序是 Simple-VM 事先预置在中断代码区的，Simple-VM 目前预置了两个中断服务子程序，其首地址分别存储在中断向量中。两个中断服务子程序较为简单，分别如下：

INT1	INT2	对应的中断向量：[0x186A0, 0x186AA]
0: IRMOV 1 1 1: IRMOV 2 2 2: IRMOV 3 3 3: ADD 1 2 4: ADD 2 3 5: IRET //简单的寄存器信息修改	0: IRMOV 2 1 1: INPUT 1 2: MRMOV 2 1 3: IRET //向用户读取数据并放在 1 号寄存器内	

本次测试对于中断程序的功能不做要求，仅选择其一作为测试，中断处理测试目标在于：

1. Simple-VM 能够响应用户模拟的中断信号，进入中断；
2. Simple-VM 在已响应当前中断信号后，不再响应其他中断信号，并作相关提示，直到当前中断结束；
3. Simple-VM 在处理中断程序时，能够保存现场以及恢复现场，即前后寄存器信息保持不变。

测试结果如下表：

操作	操作目的	操作结果
加载并查看中断代码区	判断中断程序是否预置	已预置并用高亮表示
点击中断按钮选择 INT1	模拟中断信号	虚拟机进入中断程序
查看中断程序寄存器信息	判断中断程序是否运行	虚拟机在执行中断代码
记录中断前后寄存器信息	判断是否保存/恢复现场	虚拟机成功恢复现场
在中断时		

图 5-2 展示了测试中的一些细节，(a)是 Simple-VM 加载中断代码之后的结果，我用绿色高亮表示，与用户代码区分开，指令的地址与中断向量是符合的。而(b)图是中断代码正在执行，红色表明虚拟机正在执行的指令，可见 Simple-VM 正在处理中断代码，而(c)图则是测试在中断代码执行时是否会对外来中断信号做出响应，在中断代码测试时，再次点击中断按钮，Simple-VM 给出提示“Interrupt has happened”，表明不再响应。

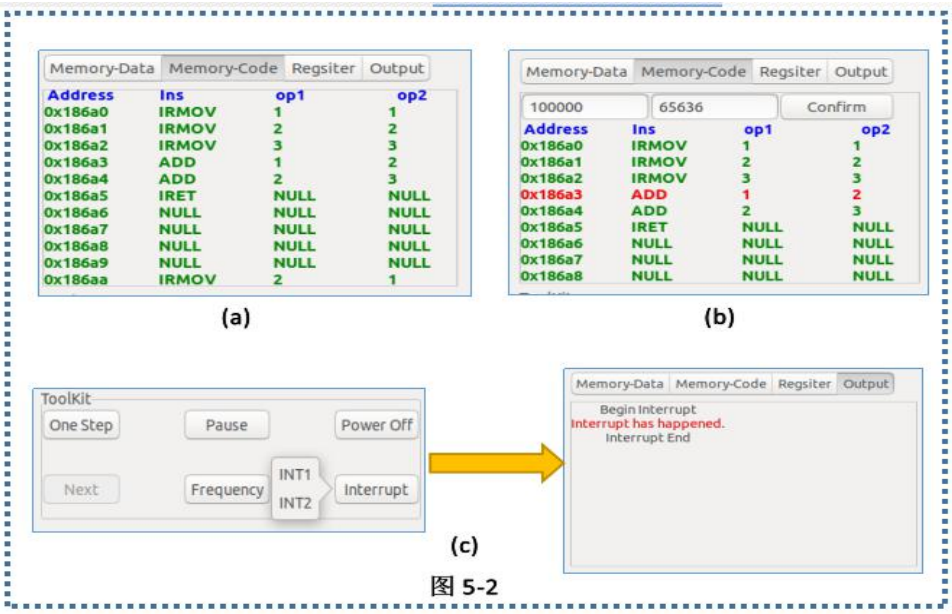


图 5-2

为了更好地帮助用户明白 Simple-VM 中断处理的细节，在本次测试中，我特别增加了命令行辅助信息，将过程量输出，见图 5-3。首先值得注意的是，PC 中存储的值是指令结构体数组的下标，其在内存中地址是该数字加上数据区大小的偏移量。从图中，我们看到在 PC=18 时，Simple-VM 进行了中断，并且记录了中断前的寄存器信息（现场），用红色表示；在中断代码运行中，我将中断处理时的寄存器信息也记录，见绿色方框，指令执行正确，并且在中断处理结束后，寄存器信息恢复到了之前的状态（恢复现场），并且开始 PC=19 的指令执行。由此可见，Simple-VM 能够进行单级中断。

```
Currentins:  {'insName': 'RRMOV', 'op1': '3', 'op2': '2'}
PC: 18
Before INT [1, 2, 2, 2, 10, 8, 3, 2, 12, 0, 10, 0, 0, 19, 32767, 10]
INT runnning: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 34462, 32784, 0]
Currentins:  {'insName': 'IRMOV', 'op1': '1', 'op2': '1'}
PC: 34462
INT runnning: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 34463, 32784, 0]
Currentins:  {'insName': 'IRMOV', 'op1': '2', 'op2': '2'}
PC: 34463
INT runnning: [1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 34464, 32784, 0]
Currentins:  {'insName': 'IRMOV', 'op1': '3', 'op2': '3'}
PC: 34464
INT runnning: [1, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 34465, 32784, 0]
Currentins:  {'insName': 'ADD', 'op1': '1', 'op2': '2'}
PC: 34465
INT runnning: [1, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 34466, 32784, 0]
Currentins:  {'insName': 'ADD', 'op1': '2', 'op2': '3'}
PC: 34466
INT runnning: [1, 3, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 34467, 32784, 0]
Currentins:  {'insName': 'IRET', 'op1': 'NULL', 'op2': 'NULL'}
PC: 34467
After INT [1, 2, 2, 2, 10, 8, 3, 2, 12, 0, 10, 0, 0, 19, 32767, 10]
Currentins:  {'insName': 'IRMOV', 'op1': '0', 'op2': '9'}
PC: 19
```

图 5-3

（3）异常检测测试

Simple-VM 提供了有限的异常检测能力，在用户代码编写不妥当时能够返回一些错误信息，帮助用户更好地调试代码（但是异常检测功能有限，不能覆盖所有的错误情况，用户不可过多依赖）。下表是 Simple-VM 的异常类型：

错误类型	错误信息	举例
寻址错误	Address Overflow	读取超出内存界限的数值
栈溢出	Stack Overflow	过多 PUSH、POP，超出堆栈边界
指令错误	InsName Error	指令名错误

针对异常检测测试，我编写了不同的错误代码，让 Simple-VM 去运行，观察它是否抛出异常信息，测试代码及结果如下：

测试代码	测试结果
RRMOV 1 18 //不存在 18 号寄存器 HLT	Address Overflow Program Finished.
PUSH 1 POP 2 POP 2　　// 栈下溢 HLT	Stack Overflow Program Finished.
China 1　//不存在 China 指令 HLT	insName Error:CHINA Program Finished.

可见，Simple-VM 拥有一定的异常检测能力。

（六）用户手册

（1） 安装说明

Simple-VM 使用 Python3.6 和 PyGtk 开发，开发系统是 Linux，用户应该实现配置相似的环境，下面给出 PyGtk 在 Linux 下的安装，打开 terminal，输入以下命令：

```
sudo apt-get install -y python-gtk2
```

当环境准备完成后，将 PVM.py 文件放在目录下，执行：

```
python3 PVM.py
```

即可进入 PVM 的启动界面。

（2） 加载程序

Simple-VM 采用图形化加载方式，在启动界面可以点击加载按钮，选择特定的用户代码，Simple-VM 会自动进行加载，将代码转换成虚拟机可识别的数据结构并存入内存。如图 6-1，值得注意的是，Simple-VM 未对用户选择的文件进行内容筛选，即用户应保证输入的文件是你想执行的 txt 文件，而不是其他格式的文件。

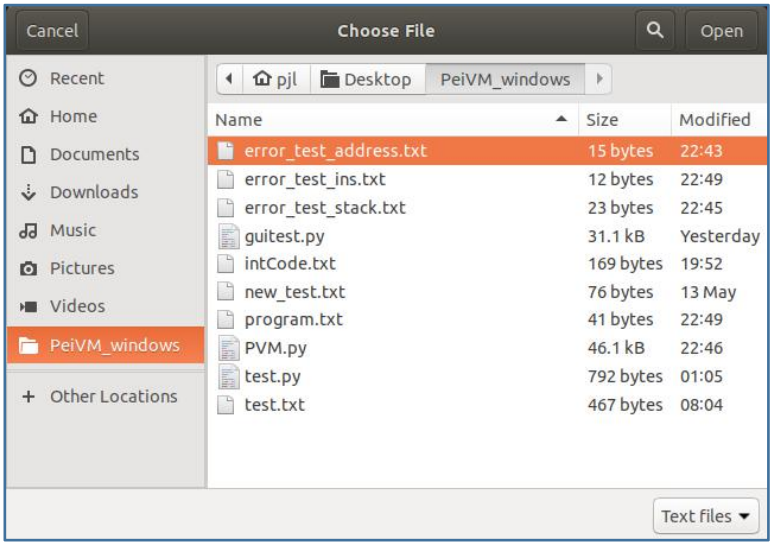


图 6-1

（3） 查看虚拟机状态

当代码加载完毕，虚拟机会进入运行状态。用户可以在显示栏选择不同的的虚拟机状态信息进行查看，数据区可以查看用户数据区和堆栈区的内容，代码区可以查看用户代码和中断代码，Simple-VM 专门将堆栈和中断代码区用绿色高亮表示，以便用户区分，特殊寄存器也作了相似的处理，用红色高亮表示，显示区则输出虚拟机或代码需要输出的信息（输出结果、异常信息等）。Simple-VM 为了方便用户查看状态，使用了区间查看的方式（因为内存空间很大，即使时滚动布局，用户也很难找到特定的数据），用户可以输入起始位置和结束位置从而查看对应区段的内容，见图 6-2。

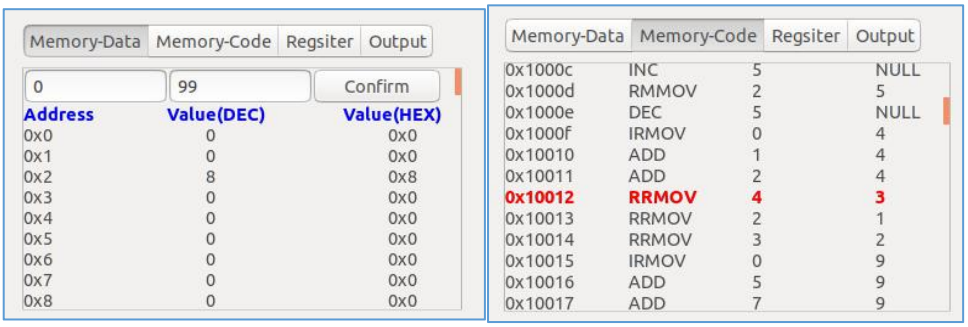


图 6-2

（4） 功能按钮

Simple-VM 提供了一组功能按钮供用户调试使用，分别为单步执行模式按钮、暂停按钮、关机按钮、单步执行控制按钮、运行频率按钮、中断按钮。按钮功能如其名，其中单步执行控制按钮只有在单步执行模式下才有效，当点击时会进行下一步操作（画面刷新、指令执行等），见图 6-3。

图 6-3

（5） 输入框

Simple-VM 提供了一个输入框，方便输入指令的执行，当虚拟机执行 INPUT 指令时，虚拟机会读取输入框的内容，用户可以在 INPUT 指令之前输入（否则 INPUT 会等待用户输入），但是用户需要明白输入框需要用”#”结尾，即”123#”表示输入数字 123，另外 Simple-VM 的输入框采取 Expander 的方式，能够随时隐藏，如图 6-4。

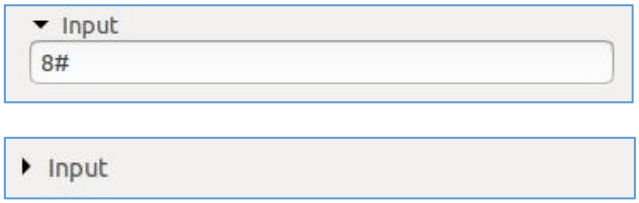


图 6-4

（6） 个性化配置

Simple-VM 提供了一定的配置功能，在启动界面可以选择进入设置界面从而进行一些虚拟机的配置，可以更改数据区大小、代码区大小、堆栈区起始位置、寄存器数量、虚拟机名称，所谓虚拟机名称是指在运行界面子标题的字符串，没有实际功能。

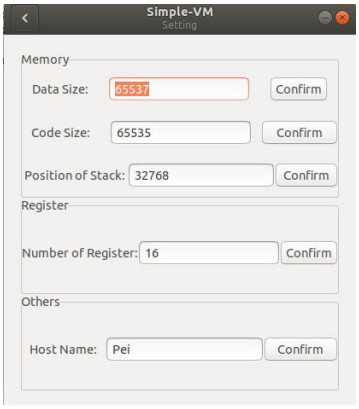


图 6-5

（七）实验总结与心得

本次计算机组成原理课程设计到此已落下帷幕，途中多次遭遇棘手的问题，在老师的讲解、同学的帮助以及自己的主动探索下，我完成了基于高级语言硬件虚拟机仿真的实现，最终收益颇丰，建立了更加完整的计算机系统结构的知识体系和整体知识框架，养成了良好的编程工程能力，从而奠定了后续学习操作系统、编译、计算机体系结构等专业课程的基础。

失败乃成功之母，本次实验的成功落幕归功于难题的出现与解决。实验初期，我对虚拟机的概念不甚了解，对任务内容的界定十分模糊，不知从何下手，幸而能力较强的同学在实验初期就展示了自己的成品，让我明确了前进的方向，再参考助教对 CCSP 赛题的实现，我脑海中出现了具体实施方案的雏形——指令模拟器，这是一个很有趣的概念，在实验开始之前我对虚拟机的概念都是 VMware 等大型虚拟机，可以跑一个成熟的操作系统，甚至对实验产生了畏惧，认为自己无法胜任这项庞大的工作。我曾经读过《编码》一书，对其中机器仅用二进制开关就能实现各种丰富多彩的功能感到惊奇，现在的虚拟机也是这样，我不能畏缩于 VMware 那些大型虚拟机展现出来的强大功能，我所需要做的是定义一个自己的指令集以及能够执行这些指令的模拟器，至于模拟器能给用户展现出怎样强大的功能，是由使用这些指令编码的人的工作（当然虚拟机的实现细节决定了最终功能的上限），这样一来，我完全可以以小见大，定义较小的指令集，实现较简易的虚拟机结构，从而加深自己对计算机结构的了解。

这样，具体到做法上的实验目标就是搭建一个框架与指令集，能够让用户抽取部分指令在框架上运行，同时框架可以实时展示运行状态。然而，倘若把这个视作字符串处理程序，那么核心算法就是冯诺依曼体系结构，我查阅了一些资料，例如《深入了解计算机系统》等，深入了解了指令在机器底层的执行细节，从我的理解上，当指令加载到内存后，程序计数器就会记录当前执行指令的位置，虚拟机所做的便是根据程序计数器执行对应的指令，本质上是一个自动机的模型。至于其他细节，报告中已有涉及。

在对虚拟机的概念以及计算机体系结构的深入了解后，我便迈上了代码实现的过程，我选择了 Python 作为图形化开发的语言，因为 Python 语法简单、功能强大，能够让我忽略高级语言层面的很多繁文缛节，从而更加聚焦于虚拟机原理的探讨。同学们一一上台讲解自己的设计报告以及实现细节，再一次指明了我的方向；关于图形化工具，我选择了 PyGtk，其融合了 Python 的简易性深得我的青睐，虽然途中被很多代码问题纠缠过，不过最终还是幸运地冲出困境。

国内有些院校有些专业在计算机方面的教育上，更加侧重软件编写，忽略了底层细节，在经过这次实验的锤炼后，我深深为其感到遗憾，甚至觉得他们错过了计算机之美。其实，评判计算机学得如何不简单是你硬件用的多么纯熟、代码写得多好，本科生教育应当立足培养学生完整的计算机体系结构，从门电路到应用软件，其间一层比一层抽象，本次实验注重芯片到汇编语言这些层，虚拟机完成了汇编器以及机器代码执行流程的仿真，如果说对本次课程设计的展望，我认为可以保留到学习编译原理之后，汇编语言之上是高级语言，高级语言转换成汇编语言需要编译器的支持，当我们对编译相关知识有更加深入的了解之后，我认为 Simple-VM 可以继续拓展，添加编译器的支持，定义属于自己的高级语言。

总而言之，本次课程设计细化了我对计算机体系的见解，明白了硬件流的细节，为今后的学习打下了坚实的基础！

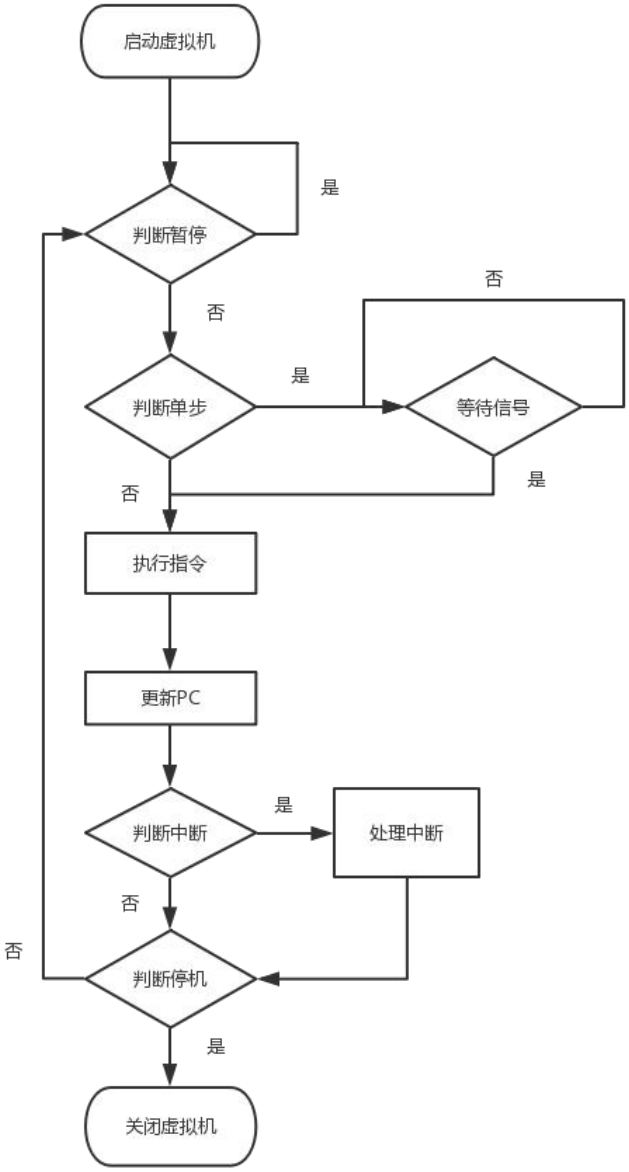
（八）参考文献

[1] 白中英，计算机组成原理（第五版，立体化教材），北京：科学出版社，2013
[2] 兰德尔·布莱恩特，大卫·奥哈拉伦，深入理解计算机系统，机械工业出版社，2017
[3] Peter Baer Galvin, Operating System Concepts, 高等教育出版社，2007

[4] https://en.wikipedia.org/wiki/Von_Neumann_architecture
[5] https://python-gtk-3-tutorial.readthedocs.io/en/latest/

（九）附录

（1）部分流程图



（2）源代码

```
#########################################################
#                                                       #
#                     Simple-VM                         #
#      Simple Simulater to execute instructions.        #
#                                                       #
#                                            by  Pei    #
#########################################################

# 加载相关库
import gi
import re
import time
import threading
gi.require_version('Gtk','3.0')
from gi.repository import Gtk, Pango, Gdk, GObject, GLib

# 线程初始化
GObject.threads_init()


#-------------------------------------全局常量与相关信号量定义
-----------------------------------------------
filename = "program.txt"                # 预处理用户代码后保存的位置，即虚拟机每次执行
的代码在磁盘存放位置
INT_VECTOR = [0x186A0,0x186AA]          # 中断向量，Simple-VM 预设了两组中断程序，其代
    码初始位置由中断向量给出
label_output = [""]                     # 存储显示区信息的结构
REG = []                                # 寄存器 List 数据结构，用来存储通用寄存器、特殊
    寄存器数值
MEM_DATA = []                           # 内存代码区 Dictionary  List 数据结构，用来存
    储中断代码以及用户代码
MEM_CODE = []                           # 内存数据区 List 数据结构，用来存储用户数据和堆
    栈数据
NUM_REG = 16                            # 默认的寄存器数量
PC_REG = 13                             # PC 在寄存器 List 中的位置
SP_REG = 14                             # SP 在寄存器 List 中的位置
CC_REG = 15                             # CC 在寄存器 List 中的位置
LEN_MEM_DATA = 65537                    # 内存数据区长度
LEN_MEM_CODE = 65535                    # 内存代码区长度，在 GUI 中，代码区地址需要加上
    数据区长度的偏移量
INT_START = 98302                       # 中断代码区起始位置
INT_END = 131071                        # 中断代码区结束位置
POS_STACK = 32768                       # 内存数据区的堆栈起始位置
POS_EXTEND = 60001                      # 内存待扩展区起始位置  内存数据区按地址顺序分
    别为数据区、堆栈区、待扩展区
HOST_NAME = "Pei"                       # 个性化配置中的虚拟机名

INT_NUMBER = 0                          # 中断序列号，用户中断时会记录对应的中断代码序
    号
last_sp = 0                             # 上一次栈指针位置，单级中断只需要单个变量保
    存
signal_run = False                      # True：虚拟机运行状态  False：虚拟机不运行
signal_pause = False                    # 虚拟机暂停信号
signal_onestep = False                  # 虚拟机单步执行信号
signal_int_happen = False               # 虚拟机中断出现信号
signal_int_run = False                  # 虚拟机中断正处理信号
signal_jmp = False                      # 虚拟机指令流中出现跳转信号
mutex_one = False                       # 虚拟机单步执行模式中的控制前进信号
```

```
frequency = 1                           # 虚拟机运行频率（用指令间延时估
    计）
is_end = False                          # 虚拟机结束信号
error_happen = False                    # 虚拟机异常信号
#-------------------------------------------------------------------------
    ---------------------------

##########################################
#                                        #
#                                        #
#         虚拟机图形主体：窗口类          #
#                                        #
#                                        #
##########################################

#---------------------启动窗口：继承自 Gtk.Window，功能：窗口切换、选择用户代码、用户代
    码预处理-------------------
class StartWindow(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self, title="Simple-VM")
        self.set_border_width(20)
        self.set_resizable(False)
        self.set_size_request(400,550)

        box = Gtk.VBox()

        # 图标设置
        grand = Gtk.Image.new_from_file("dog.png")
        box.pack_start(grand, False, False, 0)


        # 按钮设置
        box_option = Gtk.Box(spacing=10)

        # 设置键
        button_setting = Gtk.Button.new_from_icon_name("emblem-system-symbolic",
     Gtk.IconSize.MENU)
        button_setting.connect("clicked",self.on_setting_clicked)

        # 启动键
        button_start = Gtk.Button.new_from_icon_name("open-menu-symbolic", Gtk.I
    conSize.MENU)
        button_start.connect("clicked",self.on_start_clicked)

        # 退出键
        button_quit = Gtk.Button.new_from_icon_name("window-close-symbolic", Gtk.
    IconSize.MENU)
        button_quit.connect("clicked",self.on_quit_clicked)

        alignment = Gtk.Alignment.new(0.5,0.2,0,0)
        alignment.add(box_option)
        box_option.pack_start(button_setting,False,True,0)
        box_option.pack_start(button_start,False,True,0)
        box_option.pack_start(button_quit,False,True,0)


        # 附加信息


        none = Gtk.Label() # 空行标签

        # 软件名、软件功能简洁、软件用户手册链接
        label1 = Gtk.Label()
        label1.set_markup("<big><b>Simple-Virtual Machine</b></big>")
```

```
        label2 = Gtk.Label("")
        label2.set_markup("Simple Simulater To Execute built-in Instructions")
        font = Pango.FontDescription("Sans 7")
        label2.modify_font(font)
        label3 = Gtk.Label()
        label3.set_markup("<a href=\"silicat.info\" title=\"Click it\">More Deta
    ils</a>")
        label3.modify_font(font)

        # 容器装入
        box.pack_start(label1,False,True,0)
        box.pack_start(alignment,True,True,0)
        box.pack_start(label2,False,True,0)
        box.pack_start(none,False,True,0)
        box.pack_start(label3,False,True,0)

        self.add(box)

    # 预处理程序
    def VM_preprocess(self, src_filename,new_filename):
        file = open(src_filename,"r")
        newF = open(new_filename,"w")
        new_file = ''
        for line in file.readlines():
            line = line.strip()

            # 处理空行
            if (len(line) == 0):
                continue

            # 处理用户注释
            pos_comment = line.find("//")
            if (pos_comment != -1):
                line = line[0:pos_comment - 1]

            # 读取用户指令进行双操作数扩展，不足的用 NULL 字段填充
            ins = re.compile('\s+').split(line)
            if (ins[-1]==""):
                ins = ins[0:-1]
            if (len(ins) == 1):
                ins.append('NULL')
                ins.append('NULL')
            elif (len(ins) == 2):
                ins.append('NULL')

            # 虚拟机主循环需要，设计问题，与实验无关，需要在 HLT 前加 NOP 指令
            if (ins[0] == "HLT"):
                new_file = new_file + "NOP NULL NULL\n"
            new_file = new_file + " ".join(ins) + '\n'

        # 将处理之后的代码存入新的文件，供虚拟机加载
        newF.write(new_file)
        file.close()
        newF.close()

    # 启动键响应事件函数，利用 FileChooser 控件供用户进行代码文件选择
    def on_start_clicked(self, widget):
        dialog = Gtk.FileChooserDialog("Choose File", self,Gtk.FileChooserAction.
    OPEN,(Gtk.STOCK_CANCEL, Gtk.ResponseType.CANCEL,Gtk.STOCK_OPEN, Gtk.ResponseT
    ype.OK))
        self.add_filters(dialog)

        response = dialog.run()
        if response == Gtk.ResponseType.OK:
            self.VM_preprocess(dialog.get_filename(),filename)
```

```
            dialog.destroy()

            # 成功选择用户代码之后，将虚拟机运行信号置有效并退出当前窗口
            global signal_run
            signal_run = True

            self.destroy()
            Gtk.main_quit()

    # 配合 FileChooser 的文件筛选器
    def add_filters(self, dialog):
        filter_text = Gtk.FileFilter()
        filter_text.set_name("Text files")
        filter_text.add_mime_type("text/plain")
        dialog.add_filter(filter_text)

        filter_any = Gtk.FileFilter()
        filter_any.set_name("Any files")
        filter_any.add_pattern("*")
        dialog.add_filter(filter_any)

    # 设置键事件函数，退出当前窗口并进入设置界面
    def on_setting_clicked(self, button):
        self.destroy()
        Gtk.main_quit()
        setting_window = SettingWindow()
        setting_window.connect("delete-event",Gtk.main_quit)
        setting_window.show_all()
        Gtk.main()

    # 退出键事件函数，直接退出虚拟机
    def on_quit_clicked(self, button):
        Gtk.main_quit()

#-------------------------------------------------------------------------------
    ----------------------------

#--------------------设置窗口：继承自 Gtk.Window，功能：提供个性化配置，建议不要修改，
    用户手册给出推荐配置----------
class SettingWindow(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self, title="Simple-VM")
        self.set_border_width(20)
        self.set_resizable(False)
        self.set_size_request(400,550)

        # 标题栏设置，将返回键设置在标题栏，体现了 Simple-VM 的简洁性
        hb = Gtk.HeaderBar()
        hb.set_show_close_button(True)
        hb.props.title = "Simple-VM"
        hb.set_subtitle("Setting")
        self.set_titlebar(hb)

        # 返回按钮，箭形图标
        box = Gtk.Box(orientation=Gtk.Orientation.HORIZONTAL)
        button=Gtk.Button()
        button.add(Gtk.Arrow(arrow_type=Gtk.ArrowType.LEFT,shadow_type=Gtk.Shado
    wType.NONE))
        button.connect("clicked",self.on_clicked)
        box.add(button)
        hb.pack_start(box)

        # 设置信息：内存设置
        frame_memory = Gtk.Frame.new("Memory")
        frame_memory.set_shadow_type(Gtk.ShadowType.ETCHED_OUT)
```

```
        vbox = Gtk.VBox()
        label = Gtk.Label("Data Size:"); self.entry1 = Gtk.Entry(); self.entry1.
    set_text(str(LEN_MEM_DATA))
        button1 = Gtk.Button(label="Confirm"); button1.connect("clicked",self.on
    _confirm_data)
        hbox = Gtk.HBox(); hbox.pack_start(label,True,False,0); hbox.pack_start(
    self.entry1,True,False,0); hbox.pack_start(button1,True,False,0)
        vbox.pack_start(hbox,True,False,0)
        label = Gtk.Label("Code Size:"); self.entry2 = Gtk.Entry(); self.entry2.
    set_text(str(LEN_MEM_CODE))
        button1 = Gtk.Button(label="Confirm"); button1.connect("clicked",self.on
    _confirm_code)
        hbox = Gtk.HBox(); hbox.pack_start(label,True,False,0); hbox.pack_start(
    self.entry2,True,False,0); hbox.pack_start(button1,True,True,0)
        vbox.pack_start(hbox,True,False,0)
        label = Gtk.Label("Position of Stack:"); self.entry3 = Gtk.Entry(); self.
    entry3.set_text(str(POS_STACK))
        button1 = Gtk.Button(label="Confirm"); button1.connect("clicked",self.on
    _confirm_stack)
        hbox = Gtk.HBox(); hbox.pack_start(label,True,True,0); hbox.pack_start(s
    elf.entry3,True,True,0); hbox.pack_start(button1,True,True,0)
        vbox.pack_start(hbox,True,False,0)

        frame_memory.add(vbox)

        # 设置信息：寄存器设置
        none = Gtk.Label("       ")
        frame_reg = Gtk.Frame.new("Register")
        frame_reg.set_shadow_type(Gtk.ShadowType.ETCHED_OUT)
        vbox = Gtk.VBox()
        label = Gtk.Label("Number of Register:"); self.entry4 = Gtk.Entry(); sel
    f.entry4.set_text(str(NUM_REG))
        button1 = Gtk.Button(label="Confirm"); button1.connect("clicked",self.on
    _confirm_reg)
        hbox = Gtk.HBox(); hbox.pack_start(label,True,True,0); hbox.pack_start(s
    elf.entry4,True,True,0); hbox.pack_start(button1,True,True,0)
        vbox.pack_start(hbox,True,False,0)
        frame_reg.add(vbox)

        # 设置信息：  其他设置（虚拟机名，可显示在运行窗口的子标题中；开机后单步执行模式
    开关）
        frame_others = Gtk.Frame.new("Others")
        frame_others.set_shadow_type(Gtk.ShadowType.ETCHED_OUT)
        vbox=Gtk.VBox()
        label = Gtk.Label("Host Name:"); self.entry5 = Gtk.Entry(); self.entry5.
    set_text(str(HOST_NAME))
        button1 = Gtk.Button(label="Confirm"); button1.connect("clicked",self.on
    _confirm_name)
        hbox = Gtk.HBox(); hbox.pack_start(label,True,True,0); hbox.pack_start(s
    elf.entry5,True,True,0); hbox.pack_start(button1,True,True,0)
        vbox.pack_start(hbox,True,False,0)
        frame_others.add(vbox)

        box = Gtk.VBox()
        box.pack_start(frame_memory,True,True,0)
        box.pack_start(frame_reg,True,True,0)
        box.pack_start(frame_others,True,True,0)
        self.add(box)

    def on_confirm_data(self,button):
        global LEN_MEM_DATA
        LEN_MEM_DATA = int(self.entry1.get_text())

    def on_confirm_code(self,button):
        global LEN_MEM_CODE
```

```
        LEN_MEM_CODE = int(self.entry2.get_text())

    def on_confirm_stack(self,button):
        global POS_STACK
        POS_STACK = int(self.entry3.get_text())

    def on_confirm_reg(self,button):
        global NUM_REG
        NUM_REG = int(self.entry4.get_text())


    def on_confirm_name(self,button):
        global HOST_NAME
        HOST_NAME = self.entry5.get_text()


    # 返回键事件函数：关闭当前窗口，返回启动界面
    def on_clicked(self, button):
        self.destroy()
        Gtk.main_quit()
        start_window = StartWindow()
        start_window.connect("delete-event",Gtk.main_quit)
        start_window.show_all()
        Gtk.main()

#--------------------------------------------------------------------------------
    ------------------------------

#--------------------运行窗口：继承自 Gtk.Window，功能：提供机器状态展示、虚拟机控制功
    能等-------------------------
class RunningWindow(Gtk.Window):
    # 一些重要的成员变量

    regs = []   # 寄存器控件组
    mem_data = [] # 内存数据区控件组
    mem_code = [] # 内存代码区控件组
    output_labels = [] # 显示区信息标签控件组
    entry = None # 输入框
    data_start = 0
    data_end = 99
    code_start = 0 + LEN_MEM_DATA
    code_end = 99 + LEN_MEM_DATA

    def __init__(self):
        Gtk.Window.__init__(self, title="Simple-VM")
        self.set_border_width(20)
        self.set_resizable(False)
        self.set_size_request(400,550)
        # 标题栏设置，增加子标题为虚拟机名
        hb = Gtk.HeaderBar()
        hb.set_show_close_button(True)
        hb.props.title = "Simple-VM"
        global HOST_NAME
        hb.set_subtitle(HOST_NAME)
        self.set_titlebar(hb)


        vbox = Gtk.VBox(spacing=6)
        self.add(vbox)


        # 机器状态显示区：分成四页，分别是数据区、代码区、寄存器区、虚拟机及程序输出信息
    区
        frame_device = Gtk.Frame.new("")
```

```
        frame_device.set_shadow_type(Gtk.ShadowType.ETCHED_OUT)
        device_view = Gtk.Stack()
        device_view.set_transition_type(Gtk.StackTransitionType.SLIDE_LEFT_RIGHT)

        device_view.set_transition_duration(1000)

        # 数据区视图,采用滚动布局
        md_box = Gtk.VBox()
        alignment = Gtk.Alignment.new(0,0,0,0)
        alignment.add(md_box)
        sw_data = Gtk.ScrolledWindow(); sw_data.set_size_request(200,200)
        sw_data.set_shadow_type(Gtk.ShadowType.ETCHED_IN)
        sw_data.set_policy(Gtk.PolicyType.AUTOMATIC, Gtk.PolicyType.AUTOMATIC)
        sw_data.add(alignment)

        # 新增内存数据区查看范围设置
        self.data_entry1 = Gtk.Entry(); self.data_entry2 = Gtk.Entry()
        self.data_entry1.set_width_chars(8); self.data_entry2.set_width_chars(8)

        self.data_entry1.set_text(str(self.data_start)); self.data_entry2.set_te
    xt(str(self.data_end))
        data_scale_button = Gtk.Button("Confirm"); data_scale_button.connect("cl
    icked",self.data_scale)
        temp_box = Gtk.HBox(); temp_box.pack_start(self.data_entry1,True,True,0);
     temp_box.pack_start(self.data_entry2,True,True,0)
        temp_box.pack_start(data_scale_button,True,True,0); md_box.pack_start(te
    mp_box,True,True,0)

        # 数据区布局
        md = Gtk.Label("Address"+"  "*10+"Value(DEC)"+"  "*10+"Value(HEX)") # 数
    据区第一行信息栏
        self.mem_data.append(md)
        md_box.pack_start(md,False,False,0)
        global MEM_DATA,LEN_MEM_DATA
        for i in range(100):
            md1 = Gtk.Label(hex(i)); md2 = Gtk.Label("0"); md3 = Gtk.Label(hex(0)
    )

            self.mem_data.append([md1,md2,md3])
            # 固定布局，确认显示对齐
            MD = Gtk.Fixed()
            md1.set_xalign(0); md2.set_xalign(0); md3.set_xalign(0)
            MD.put(md1,0,0); MD.put(md2,150,0);MD.put(md3,300,0)
            md_box.pack_start(MD,True,True,0)

        device_view.add_titled(sw_data,"data","Memory-Data")


        # 代码区视图：采取滚动布局
        mc_box = Gtk.VBox()
        alignment = Gtk.Alignment.new(0,0,0,0)
        alignment.add(mc_box); sw_code = Gtk.ScrolledWindow()
        sw_code.set_shadow_type(Gtk.ShadowType.ETCHED_IN)
        sw_code.set_policy(Gtk.PolicyType.AUTOMATIC, Gtk.PolicyType.AUTOMATIC)
        sw_code.add(alignment)

        # 新增内存代码区查看范围设置
        self.code_entry1 = Gtk.Entry(); self.code_entry2 = Gtk.Entry()
        self.code_entry1.set_width_chars(8); self.code_entry2.set_width_chars(8)

        self.code_entry1.set_text(str(self.code_start)); self.code_entry2.set_te
    xt(str(self.code_end))
        code_scale_button = Gtk.Button("Confirm"); code_scale_button.connect("cl
    icked",self.code_scale)
```

```
        temp_box = Gtk.HBox(); temp_box.pack_start(self.code_entry1,True,True,0);
 temp_box.pack_start(self.code_entry2,True,True,0)
        temp_box.pack_start(code_scale_button,True,True,0); mc_box.pack_start(te
mp_box,True,True,0)

        # 代码区布局
        mc = Gtk.Label("Address"+"  "*10+"Ins"+"  "*10+"op1"+"  "*10+"op2") # 代
码区第一行为信息栏
        self.mem_code.append(mc)
        mc_box.pack_start(mc,True,True,0)
        global MEM_CODE,LEN_MEM_CODE
        for i in range(100):
            mc0 = Gtk.Label(hex(i+LEN_MEM_DATA))
            mc1 = Gtk.Label(str(MEM_CODE[i-1]['insName'])); mc2 = Gtk.Label(str(
MEM_CODE[i-1]['op1'])); mc3 = Gtk.Label(str(MEM_CODE[i-1]['op2']))
            self.mem_code.append([mc0,mc1,mc2,mc3])
            # 采用固定布局，保证显示对齐
            MC = Gtk.Fixed()
            mc0.set_xalign(0); mc1.set_xalign(0); mc2.set_xalign(0); mc3.set_xal
ign(0)
            MC.put(mc0,0,0); MC.put(mc1,100,0); MC.put(mc2,200,0);MC.put(mc3,300,
0)
            mc_box.pack_start(MC,True,True,0)

        device_view.add_titled(sw_code,"code","Memory-Code")

        # 寄存器视图：采用嵌套 Box 布局和 Fixed 布局
        label_reg = Gtk.Fixed()
        frame_reg = Gtk.Frame.new("")
        frame_reg.set_shadow_type(Gtk.ShadowType.ETCHED_OUT)
        frame_reg.add(label_reg)
        global NUM_REG,PC_REG,SP_REG,CC_REG
        for i in range(NUM_REG):
            reg_row = Gtk.Box()
            for j in range(4):
                # 对于特殊寄存器，Simple-VM 高亮显示
                if (len(self.regs) == PC_REG):
                    reg = Gtk.Label(); reg.set_markup("<span color=\"red\">PC: <
/span>")
                elif (len(self.regs) == SP_REG):
                    reg = Gtk.Label(); reg.set_markup("<span color=\"red\">SP: <
/span>")
                elif (len(self.regs) == CC_REG):
                    reg = Gtk.Label(); reg.set_markup("<span color=\"red\">CC: <
/span>")
                else:
                    reg = Gtk.Label("R"+str(len(self.regs) + 1))
                self.regs.append(reg)
                label_reg.put(reg,j * 100, i * 50)
                if (len(self.regs) == NUM_REG):
                    break
            if (len(self.regs) == NUM_REG):
                break

        device_view.add_titled(frame_reg, "reg", "Regsiter")

        # 显示区视图：采用滚动布局，对于不同的信息来源采用不同的颜色高亮
        mo_box = Gtk.VBox()
        alignment = Gtk.Alignment.new(0,0,0,0)
        alignment.add(mo_box); sw_code = Gtk.ScrolledWindow()
        sw_code.set_shadow_type(Gtk.ShadowType.ETCHED_IN)
        sw_code.set_policy(Gtk.PolicyType.AUTOMATIC, Gtk.PolicyType.AUTOMATIC)
        sw_code.add(alignment)
        mo = Gtk.Label("Output Information")
        self.output_labels.append(mo)
```

```
        for i in range(1000):
            mo = Gtk.Label("")
            self.output_labels.append(mo)
            mo_box.pack_start(mo,False,False,0)
        device_view.add_titled(sw_code,"output","Output")

        stack_switcher = Gtk.StackSwitcher()
        stack_switcher.set_stack(device_view)
        vbox.pack_start(stack_switcher,False,True,1)
        vbox.pack_start(device_view,False,True,0)

        # 功能按钮：单步多步切换、暂停继续切换、关机键、单步控制、运行频率、模拟中断按钮
        frame_buttons = Gtk.Frame.new("ToolKit")
        frame_buttons.set_shadow_type(Gtk.ShadowType.ETCHED_OUT)
        buttons = Gtk.Grid()
        frame_buttons.add(buttons)
        buttons.set_row_spacing(30)
        buttons.set_column_spacing(60)

        # 单步按钮
        button1 = Gtk.ToggleButton(label="One Step")
        button1.connect("toggled",self.step_shift)

        # 关机按钮
        button2 = Gtk.Button(label="Power Off")
        button2.connect("clicked",self.on_quit_clicked)

        # 暂停按钮
        button3 = Gtk.Button(label="Pause")
        button3.connect("clicked",self.on_pause_clicked)

        # 频率按钮：PopOver 控件作为二级按钮
        self.button4 = Gtk.Button(label="Frequency")
        self.button4.connect("clicked",self.on_frequency_clicked)
        vbox4 = Gtk.VBox()
        fre_0_1 = Gtk.ModelButton("0.1"); vbox4.pack_start(fre_0_1,False,True,0);
    fre_0_1.connect("clicked",self.on_fre_0_1_clicked)
        fre_1_0 = Gtk.ModelButton("1.0"); vbox4.pack_start(fre_1_0,False,True,0);
    fre_1_0.connect("clicked",self.on_fre_1_0_clicked)
        fre_2_0 = Gtk.ModelButton("2.0"); vbox4.pack_start(fre_2_0,False,True,0);
    fre_2_0.connect("clicked",self.on_fre_2_0_clicked)
        fre_5_0 = Gtk.ModelButton("5.0"); vbox4.pack_start(fre_5_0,False,True,0);
    fre_5_0.connect("clicked",self.on_fre_5_0_clicked)
        self.popover = Gtk.Popover()
        self.popover.add(vbox4)
        self.popover.set_position(Gtk.PositionType.LEFT)

        # 模拟中断按钮：采用 PopOver 控件作为二级按钮
        self.button5 = Gtk.Button(label="Interrupt")
        self.button5.connect("clicked",self.on_interrupt_clicked)
        vbox5 = Gtk.VBox()
        int1 = Gtk.ModelButton("INT1"); vbox5.pack_start(int1,False,True,0); int
    1.connect("clicked",self.on_int1_clicked)
        int2 = Gtk.ModelButton("INT2"); vbox5.pack_start(int2,False,True,0); int
    2.connect("clicked",self.on_int2_clicked)
        self.popover5 = Gtk.Popover()
        self.popover5.add(vbox5)
        self.popover5.set_position(Gtk.PositionType.LEFT)

        # 中断前进控制按钮
        self.button6 = Gtk.Button(label="Next")
        self.button6.set_sensitive(False)
        self.button6.connect("clicked",self.on_next_clicked)

        # 按钮布局，Grid 布局
```

```
        buttons.attach(button1,0,0,1,1)
        buttons.attach(button3,1,0,1,1)
        buttons.attach(button2,2,0,1,1)
        buttons.attach(self.button6,0,1,1,1)
        buttons.attach(self.button4,1,1,1,1)
        buttons.attach(self.button5,2,1,1,1)
        buttons.set_size_request(100,100)
        vbox.pack_start(frame_buttons, True, False, 0)

        # 输入框，用 Expander 存储，可随时隐藏
        self.entry = Gtk.Entry()
        expand = Gtk.Expander.new("Input")
        expand.set_expanded(False)
        expand.add(self.entry)
        vbox.pack_start(expand,False,False,0)

        # 判断虚拟机是否运行信号，若信号有效，显示窗口
        global signal_run

        if (signal_run):
            self.show_all()

    # 两组范围确认按钮事件函数
    def data_scale(self, data_scale_button):
        self.data_start = int(self.data_entry1.get_text())
        self.data_end = int(self.data_entry2.get_text())

    def code_scale(self, code_scale_button):
        self.code_start = int(self.code_entry1.get_text())
        self.code_end = int(self.code_entry2.get_text())


    # 关机键按钮事件函数：设置虚拟机状态为关闭，关闭窗口
    def on_quit_clicked(self,widget):
        global is_end
        is_end = True
        Gtk.main_quit()

    # 运行频率按钮事件函数：将二级按钮显示
    def on_frequency_clicked(self,widget):
        self.popover.set_relative_to(self.button4)
        self.popover.show_all()
        self.popover.popup()

    # 频率二级按钮事件函数，改变虚拟机运行频率（指令间延时）
    def on_fre_0_1_clicked(self,widget):
        global frequency
        frequency = 0.1
    def on_fre_1_0_clicked(self,widget):
        global frequency
        frequency = 1.0
    def on_fre_2_0_clicked(self,widget):
        global frequency
        frequency = 2.0
    def on_fre_5_0_clicked(self,widget):
        global frequency
        frequency = 5.0

    # 模拟中断按钮事件函数，调用二级按钮
    def on_interrupt_clicked(self,widget):
        self.popover5.set_relative_to(self.button5)
        self.popover5.show_all()
        self.popover5.popup()

    # 用户选择中断序列，Simple-VM 提供了两个预置中断代码，用户可以自行选择
```

```
    def on_int1_clicked(self,widget):
        global INT_NUMBER
        INT_NUMBER = 0
        global signal_int_happen,signal_int_run
        if (signal_int_run):
            # 如果当前已有中断信号，则抛出信息
            global label_output; label_output.append("<span color=\"red\">Interr
    upt has happened.</span>")
        else:
            # 否则将中断信号置为有效
            signal_int_happen = True

    def on_int2_clicked(self,widget):
        global INT_NUMBER
        INT_NUMBER = 1
        global signal_int_happen,signal_int_run
        if (signal_int_run):
            # 如果当前已有中断信号，则抛出信息
            global label_output; label_output.append("<span color=\"red\">Interr
    upt has happened.</span>")
        else:
            # 否则将中断信号置为有效
            signal_int_happen = True

    # 单步控制前进按钮事件函数：点击一下，虚拟机继续跑一次指令
    def on_next_clicked(self,widget):
        global mutex_one,signal_onestep
        if (signal_onestep):
                mutex_one = True

    # 暂停事件函数：改变虚拟机相关信号
    def on_pause_clicked(self, button3):
        global signal_pause
        # 根据虚拟机当前状态选择不同处理，若已暂停则继续，若正在运行则暂停
        if (button3.get_label() == "Pause"):
            button3.set_label("Continue")
            signal_pause = True
        else:
            button3.set_label("Pause")
            signal_pause = False

    # 单步按钮事件函数：改变虚拟机相关信号
    def step_shift(self,button1):
        global signal_onestep
        if (button1.get_active()):
            signal_onestep = True
            self.button6.set_sensitive(True)
            self.button4.set_sensitive(False)
        else:
            signal_onestep = False
            self.button6.set_sensitive(False)
            self.button4.set_sensitive(True)

#-------------------------------------------------------------------------------
    ------------------------------

##########################################
#                                        #
#                                        #
#         虚拟机逻辑主体:逻辑函数          #
#                                        #
#                                        #
##########################################
```

```
# 虚拟机初始化函数
def VM_init():
    global LEN_MEM_DATA,LEN_MEM_CODE,MEM_DATA,MEM_CODE,REG,PC_REG,SP_REG,signal_
    jmp,INT_START,INT_END,INT_VECTOR

    # 初始化数据区为 0
    for i in range(LEN_MEM_DATA):
        MEM_DATA.append(0)

    # 初始化代码区为 None
    for i in range(LEN_MEM_CODE):
        MEM_CODE.append({'insName':None,'op1':None,'op2':None})

    # 预置中断代码区
    intFile = open("intCode.txt","r").readlines()
    for i in range(INT_VECTOR[0],INT_VECTOR[0]+len(intFile)):
        temp = intFile[i - INT_VECTOR[0]].split()
        MEM_CODE[i-LEN_MEM_DATA-1] = {'insName':temp[0],'op1':temp[1],'op2':temp
    [2]}


    # 清空寄存器，设置 SP 为初始栈位置-1
    for i in range(NUM_REG):
        REG.append(0)
    REG[PC_REG] = 0
    REG[SP_REG] = POS_STACK - 1

    # 设置跳转信号为无效
    signal_jmp = False

# 虚拟机加载函数
def VM_load(filename):
    global MEM_CODE,REG
    file = open(filename,"r")
    insNum = 0
    for line in file.readlines():
        # 解析字符串
        temp_ins = line.strip().split(' ')
        # 构造临时指令结构
        ins = {'insName':None,'op1':None,'op2':None}
        ins['insName'] = temp_ins[0].upper()
        ins['op1'] = temp_ins[1]
        ins['op2'] = temp_ins[2]
        # 加载该指令到内存
        MEM_CODE[insNum] = ins
        insNum = insNum + 1


running_window = None
signal_iret = False

# 软件核心主函数
def app_main():
    running_window = RunningWindow()
    running_window.connect("delete-event",Gtk.main_quit)
    # 保存现场
    def save_scene():
        global MEM_DATA,REG,last_sp,SP_REG,signal_int_run,SP_REG
        # 记录保存现场的起点，以便恢复现场
        last_sp = REG[SP_REG]
        #print("Before INT",REG)
        # 调用 PUSH 指令，将寄存器信息压入堆栈
        for i in range(len(REG)):
            ins = {};
```

```
            ins['insName'] = "PUSH"; ins["op1"] = i + 1; ins["op2"] = "NULL"
            ins_exec(ins)
        MEM_DATA[last_sp+SP_REG+1] = last_sp
        # 开始执行中断程序，设置中断代码运行信号有效
        signal_int_run = True

    # 恢复现场
    def recover_scene():
        global MEM_DATA,REG,last_sp,SP_REG,signal_int_happen,signal_int_run
        # 将中断出现信号置无效
        signal_int_happen = False
        # 将寄存器信息重新返回
        sp = last_sp
        for i in range(len(REG)):
            REG[i] = MEM_DATA[sp+i + 1]
           # print(sp,sp+i,MEM_DATA[sp+i])
        # 关闭中断程序，设置中断运行信号无效
        #print("After INT",REG)
        signal_int_run = False
        output("Interrupt End")

    # 封装了一个在显示区输出字符串的函数
    def output(str):
        label_output.append(str)

    # 控制器
    def VM_run():
        global MEM_CODE,REG,PC_REG,signal_jmp,signal_pause,signal_onestep,mutex_
    one,frequency,signal_int_run,last_sp
        global SP_REG,MEM_DATA,INT_VECTOR,INT_NUMBER,CC_REG,is_end,signal_iret
        global error_happen,signal_int_run

        currentIns = MEM_CODE[REG[PC_REG]]

        while (currentIns['insName'] != 'HLT' and not is_end):
            # 阻塞更新接口线程
            GLib.idle_add(update)
            #if (signal_int_run):
            #    print("INT runnning:",REG)
            # 处理单步执行信号，并保证中途退出能够关闭界面
            while (signal_onestep and not mutex_one):
                print("")
                if (is_end):
                    break
            mutex_one = False

            # 处理暂停信号，并保证中途退出能够关闭界面
            while (signal_pause):
                print("")
                if (is_end):
                    break

            # 在多步执行模式下，调节延时，以估计虚拟机运行频率
            if (not signal_onestep and not signal_pause):
                time.sleep(frequency)


            #if (True):
            #    print("Currentins: ",currentIns)
            #    print("PC:",REG[PC_REG])


            # 执行指令
            ins_exec(currentIns)
```

```
            # 根据跳转信号，更新 PC
            if (not signal_jmp):
                REG[PC_REG] = REG[PC_REG] + 1
            if (signal_iret):
                REG[PC_REG] = REG[PC_REG] - 1
                signal_iret = False
            signal_jmp = False

            # 读取下一条指令
            currentIns = MEM_CODE[REG[PC_REG]]


            if (signal_int_happen and not signal_int_run):
                output("Begin Interrupt")
                sp = REG[SP_REG]
                save_scene()
                # 初始化中断代码的寄存器信息
                for i in range(len(REG)):
                    REG[i] = 0
                REG[SP_REG] = sp + len(REG) + 1
                REG[PC_REG] = INT_VECTOR[INT_NUMBER] - LEN_MEM_DATA - 1
                currentIns = MEM_CODE[REG[PC_REG]]


        output("<span color=\"green\">Program Finished.</span>")
        update()
        #Gtk.main_quit()

    # 解析条件码寄存器，由于 Simple-VM 只有三个条件码，因此直接用十进制作为模拟，110 表示前两个条件码有效，以此类推
    def extractCC(N):
        return [((N//100)%10),((N//10)%10),N%10]

    # 运算器，负责执行单条指令
    def ins_exec(Ins):
        global REG,MEM_DATA,SP_REG,PC_REG,signal_jmp,CC_REG,error_happen,LEN_MEM_DATA,LEN_MEM_CODE,POS_STACK

        # 解析指令，获取指令名与操作数
        insName = Ins['insName']
        op1 = Ins['op1']
        op2 = Ins['op2']

       # 执行指令
        if (insName == 'RRMOV'):
            [s,d] = [int(op1), int(op2)]
            if (d > len(REG) or s > len(REG) or d < 1 or s < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = REG[s - 1]
        elif (insName == "NOP"):
            pass
        elif (insName == 'IRMOV'):
            [s,d] = [int(op1), int(op2)]
            if (d > len(REG) or d < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = s
        elif (insName == 'RMMOV'):
            [s,d] = [int(op1), REG[int(op2) - 1]]
            if (s > len(REG) or d > LEN_MEM_DATA or d < 0 or s < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                MEM_DATA[d] = REG[s - 1]
```

```
        elif (insName == 'MRMOV'):
            [s,d] = [REG[int(op1) - 1], int(op2)]
            if (d > len(REG) or s > LEN_MEM_DATA or s < 0 or d < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = MEM_DATA[s]
        elif (insName == 'PUSH'):
            s = int(op1)
            if (s > len(REG) or s < 1):
                output("<span color=\"red\">Address Overflow</span>")
            elif (REG[SP_REG] >= LEN_MEM_DATA or REG[SP_REG] < POS_STACK - 1):
                output("<span color=\"red\">Stack Overflow</span>")
            else:
                REG[SP_REG] = REG[SP_REG] + 1
                MEM_DATA[REG[SP_REG]] = REG[s - 1]
        elif (insName == 'POP'):
            d = int(op1)
            if (d > len(REG) or d < 1):
                output("<span color=\"red\">Address Overflow</span>")
            elif (REG[SP_REG] >= LEN_MEM_DATA or REG[SP_REG] < POS_STACK):
                output("<span color=\"red\">Stack Overflow</span>")
            else:
                REG[d - 1] = MEM_DATA[REG[SP_REG]]
                REG[SP_REG] = REG[SP_REG] - 1
        elif (insName == 'INC'):
            d = int(op1)
            if (d > len(REG) or d < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = REG[d - 1] + 1
        elif (insName == 'DEC'):
            d = int(op1)
            if (d > len(REG) or d < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = REG[d - 1] - 1
        elif (insName == 'ADD'):
            [s,d] = [int(op1), int(op2)]
            if (d > len(REG) or s > len(REG) or d < 1 or s < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = REG[d - 1] + REG[s - 1]
        elif (insName == 'SUB'):
            [s,d] = [int(op1), int(op2)]
            if (d > len(REG) or s > len(REG) or d < 1 or s < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = REG[d - 1] - REG[s - 1]
        elif (insName == 'IMUL'):
            [s,d] = [int(op1), int(op2)]
            if (d > len(REG) or s > len(REG) or d < 1 or s < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = REG[d - 1] * REG[s - 1]
        elif (insName == 'DIV'):
            [s,d] = [int(op1), int(op2)]
            if (d > len(REG) or s > len(REG) or d < 1 or s < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = REG[d - 1] / REG[s - 1]
        elif (insName == "NOT"):
            d = int(op1)
            if (d > len(REG) or d < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = ~REG[d - 1]
```

```
        elif (insName == "XOR"):
            [s,d] = [int(op1), int(op2)]
            if (d > len(REG) or s > len(REG) or d < 1 or s < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = REG[d - 1] ^ REG[s - 1]
        elif (insName == "OR"):
            [s,d] = [int(op1), int(op2)]
            if (d > len(REG) or s > len(REG) or d < 1 or s < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = REG[d - 1]|REG[s - 1]
        elif (insName == "AND"):
            [s,d] = [int(op1), int(op2)]
            if (d > len(REG) or s > len(REG) or d < 1 or s < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = REG[d - 1]®[s - 1]
        elif (insName == "SAL"):
            [s,d] = [int(op1), int(op2)]
            if (d > len(REG) or s > len(REG) or d < 1 or s < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = REG[d - 1] << s
        elif (insName == "SAR"):
            [k,d] = [int(op1), int(op2)]
            if (d > len(REG) or s > len(REG) or d < 1 or s < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[d - 1] = REG[d - 1] >> k
        elif (insName == "JMP"):
            addr = int(op1)
            if (addr < 0 or addr > LEN_MEM_CODE):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                REG[PC_REG] = addr
                signal_jmp = True
        elif (insName == "JE"):
            addr = int(op1)
            CZ,LZ,GZ = extractCC(REG[CC_REG])
            if (CZ == 1):
                if (addr < 0 or addr > LEN_MEM_CODE):
                    output("<span color=\"red\">Address Overflow</span>")
                else:
                    REG[PC_REG] = addr
                    signal_jmp = True
        elif (insName == "JNE"):
            addr = int(op1)
            CZ,LZ,GZ = extractCC(REG[CC_REG])
            if (CZ == 0):
                if (addr < 0 or addr > LEN_MEM_CODE):
                    output("<span color=\"red\">Address Overflow</span>")
                else:
                    REG[PC_REG] = addr
                    signal_jmp = True
        elif (insName == "JL"):
            addr = int(op1)
            CZ,LZ,GZ = extractCC(REG[CC_REG])
            if (LZ == 1):
                if (addr < 0 or addr > LEN_MEM_CODE):
                    output("<span color=\"red\">Address Overflow</span>")
                else:
                    REG[PC_REG] = addr
                    signal_jmp = True
        elif (insName == "JLE"):
            addr = int(op1)
```

```
            CZ,LZ,GZ = extractCC(REG[CC_REG])
            if (LZ == 1 or CZ == 1):
                if (addr < 0 or addr > LEN_MEM_CODE):
                    output("<span color=\"red\">Address Overflow</span>")
                else:
                    REG[PC_REG] = addr
                    signal_jmp = True
        elif (insName == "JG"):
            addr = int(op1)
            CZ,LZ,GZ = extractCC(REG[CC_REG])
            if (GZ == 1):
                if (addr < 0 or addr > LEN_MEM_CODE):
                    output("<span color=\"red\">Address Overflow</span>")
                else:
                    REG[PC_REG] = addr
                    signal_jmp = True
        elif (insName == "JGE"):
            addr = int(op1)
            CZ,LZ,GZ = extractCC(REG[CC_REG])
            if (GZ == 1 or CZ == 1):
                if (addr < 0 or addr > LEN_MEM_CODE):
                    output("<span color=\"red\">Address Overflow</span>")
                else:
                    REG[PC_REG] = addr
                    signal_jmp = True
        elif (insName == "CMP"):
            [s,d] = [int(op1), int(op2)]
            if (d > len(REG) or s > len(REG) or d < 1 or s < 1):
                output("<span color=\"red\">Address Overflow</span>")
            else:
                if (REG[d-1] - REG[s-1] == 0):
                    REG[CC_REG] = 100
                elif (REG[d-1] - REG[s-1] < 0):
                    REG[CC_REG] = 10
                elif (REG[d-1] - REG[s-1] > 0):
                    REG[CC_REG] = 1
        elif (insName == "MOUT"):
            d = int(op1)
            if (d > len(REG) or d < 1 or REG[d-1] > LEN_MEM_DATA or REG[d-1] < 0)
    :
                output("<span color=\"red\">Address Overflow</span>")
            else:
                output("<span color=\"blue\">"+str(MEM_DATA[REG[d-1]])+"</span>")

        elif (insName == "INPUT"):
            d = int(op1)
            if (d > len(REG) or d < 1 or REG[d-1] > LEN_MEM_DATA or REG[d-1] < 0)
    :
                output("<span color=\"red\">Address Overflow</span>")
            else:
                s = ""
                while (len(s) <= 0 or s[-1] != '#'):
                    s = running_window.entry.get_text()
                MEM_DATA[REG[d-1]] = int(s[0:-1])
        elif (insName == "IRET"):
            recover_scene()
            global signal_iret
            signal_iret = True
        else:
            output("<span color=\"red\">insName Error:"+str(insName)+"</span>")


    # 更新接口程序，负责更新窗口信息
    def update():
        regs = running_window.regs
        mem_data = running_window.mem_data
```

```
        mem_code = running_window.mem_code
        output_labels= running_window.output_labels
        global REG,MEM_DATA,MEM_CODE,PC_REG,SP_REG,CC_REG,INT_START,INT_END,erro
    r_over

        # 更新显示区信息
        for i in range(1,len(label_output)):
            output_labels[i].set_markup(label_output[i])

        # 更新寄存器信息
        for i in range(len(regs)):
            if (i == PC_REG):
                s = "<span color=\"red\">PC: " + str(hex(REG[i] + LEN_MEM_DATA))
    +"</span>"
            elif (i == SP_REG):
                s = "<span color=\"red\">SP: " + str(REG[i])+"</span>"
            elif (i == CC_REG):
                s = "<span color=\"red\">CC: " + str(REG[i])+"</span>"
            else:
                s = "R"+str(i + 1)+": "+str(REG[i])
            regs[i].set_markup(s)

        # 更新数据区信息
        mem_data[0].set_markup("<b><span color=\"blue\">Address</span>"+" "*15+"
    <span color=\"blue\">Value(DEC)</span>"+"  "*8+"<span color=\"blue\">Value(HE
    X)</span></b>")
        start = running_window.data_start; end = running_window.data_end; end =
    start + 99
        for i in range(start,end+1):
            s0 = hex(i); s1 = str(MEM_DATA[i]); s2 = hex(MEM_DATA[i])
            if (i >= POS_STACK and i < POS_EXTEND):
                s0 = "<b><span color=\"green\">"+s0+"</span></b>"
                s1 = "<b><span color=\"green\">"+s1+"</span></b>"
                s2 = "<b><span color=\"green\">"+s2+"</span></b>"
            if (i >= POS_EXTEND):
                s0 = "<b><span color=\"red\">"+s0+"</span></b>"
                s1 = "<b><span color=\"red\">"+s1+"</span></b>"
                s2 = "<b><span color=\"red\">"+s2+"</span></b>"
            mem_data[i-start+1][0].set_markup(s0)
            mem_data[i-start+1][0].set_xalign(0)
            mem_data[i-start+1][1].set_markup(s1)
            mem_data[i-start+1][1].set_xalign(0)
            mem_data[i-start+1][2].set_markup(s2)
            mem_data[i-start+1][2].set_xalign(0)

        # 更新代码区信息
        mem_code[0].set_markup("<b><span color=\"blue\">Address</span>"+" "*10+"
    <span color=\"blue\">Ins</span>"+"  "*10+"<span color=\"blue\">op1</span>"+"
     "*10+"<span color=\"blue\">op2</span></b>")
        start = running_window.code_start; end = running_window.code_end; end =
    start + 99
        for i in range(start,end+1):
            s0 = hex(i); s1 = str(MEM_CODE[i-LEN_MEM_DATA-1]['insName']); s2 = s
    tr(MEM_CODE[i-LEN_MEM_DATA-1]['op1']); s3 = str(MEM_CODE[i-LEN_MEM_DATA-1]['o
    p2'])

            if (i >= INT_START and i <= INT_END and REG[PC_REG] + LEN_MEM_DATA !
    = i):
                s0 = "<b><span color=\"green\">"+s0+"</span></b>"
                s1 = "<b><span color=\"green\">"+s1+"</span></b>"
                s2 = "<b><span color=\"green\">"+s2+"</span></b>"
                s3 = "<b><span color=\"green\">"+s3+"</span></b>"
            if (REG[PC_REG]+LEN_MEM_DATA == i):
                s0 = "<b><span color=\"red\">"+s0+"</span></b>"
                s1 = "<b><span color=\"red\">"+s1+"</span></b>"
```

```
                s2 = "<b><span color=\"red\">"+s2+"</span></b>"
                s3 = "<b><span color=\"red\">"+s3+"</span></b>"

            mem_code[i-start+1][0].set_markup(s0)
            mem_code[i-start+1][0].set_xalign(0)
            mem_code[i-start+1][1].set_markup(s1)
            mem_code[i-start+1][1].set_xalign(0)
            mem_code[i-start+1][2].set_markup(s2)
            mem_code[i-start+1][2].set_xalign(0)
            mem_code[i-start+1][3].set_markup(s3)
            mem_code[i-start+1][3].set_xalign(0)


    running_window.show_all()
    thread = threading.Thread(target=VM_run)
    thread.daemon = True
    thread.start()


def main():
    GObject.threads_init()
    start_window = StartWindow()
    start_window.connect("delete-event",Gtk.main_quit)
    start_window.show_all()
    Gtk.main()
    if (signal_run):
        VM_init()
        VM_load(filename)
        app_main()
        Gtk.main()

if __name__ == "__main__":
    main()
```