# Neko's X86 Virtual Machine (NXVM)

# IBM-PC/AT 虚拟机设计文档

cshaxu@gmail.com

## 序言

IBM-PC/AT 在上世纪 90 年代曾经广泛使用，也是现在个人计算机的基础。对于任何有兴趣探究计算机组成原理、运行过程的人而言，理解 IBM-PC/AT 应该是最基础的一课。

然而，透彻理解计算机的运作过程并非如此简单。尽管到处都能找到诸如《计算机组成原理》和《微机原理与接口技术》这样的教材，但通读它们并不是一件令人愉快的事情，因为纸面上的知识终究没有落到实处，读起来将是很枯燥的。实践出真知，最好的学习方法不是单纯的阅读，而是通过实际应用来掌握。在实践中出现问题、查阅资料，才能把所学知识天然地联系到一起。

因此，一个可行的方案是亲自动手编写虚拟机软件。尽管对于个人而言，工作量略显庞大——例如 NXVM 源代码本身就超过 37,000 行——但依然力所能及。本文提供了一个虚拟机的设计思路，有助于读者理解 IBM-PC/AT 的运行原理和虚拟机的工作原理。同时也提供了相应的代码导读，为读者自己动手编写虚拟机作翔实的参考。

本文适合对汇编、操作系统、PC 和 MSDOS 感兴趣的读者。

**关键词：**8086，虚拟机，PC，汇编，计算机原理

# 目录

# 第一章 概述

本章描述 NXVM 虚拟机的设计目标、开发环境和预备资料。

虚拟机泛指由软件编写的软件容器，它们通过虚拟出完整和独立的计算机硬件环境来支持运行于这一环境中的软件。NXVM 是一款系统虚拟机，它模拟了一个可以运行完整操作系统(MS-DOS 6.22) 的计算机系统平台。对于 NXVM 而言，更加合适的术语是“模拟器”，因为它通过软件编程来模拟所有的虚拟硬件设备，而不借助于真实的硬件本身。

## 1.1 设计目标

NXVM 是一款小型 PC 模拟器。麻雀虽小，五脏俱全，NXVM 的开发需符合如下设计原则。

完备性：NXVM 必须能够提供一个较为完整的计算机硬件模拟环境。所有运行 MSDOS 操作系统所需的硬件设备都应得到模拟。

可验证性：整个系统可以进行自顶向下的逐层分解，易于检查、测试和审阅，保证系统的正确性。

一致性：整个系统的各模块都使用一致的概念、符号，如数据类型和接口。

模块化和局部化：每个逻辑相对独立的部分都是独立的编程单元。模块内部有较高的聚合度，模块之间有较低的耦合度。

可移植性：虚拟机的硬件虚拟部分可以划分为平台无关代码和平台相关代码。例如，键盘、显示器的模拟，就依赖于宿主(Host)操作系统的特性，而 CPU 和各硬件控制器就与宿主无关。NXVM 的目标宿主操作系统是 Windows 2000 及以上版本 NT 操作系统，或者 Linux 操作系统(ncurses 库和 pthread 库)。

可用性：虚拟机的配置、启动、调试功能应易于使用，能可靠、高效地完成使用目的。

## 1.2 开发环境

NXVM 主要是在 Windows 7 + Visual Studio 2008 上开发的。Linux 平台相关部分是在 Ubuntu 10.04 下开发的，需调用 ncurses 和 pthread 库。

### 1.2.1 Windows 编译

将章节 2.2 中所列的所有源代码文件加入 Visual C++的工程中，使用 x64 Release 选项编译即可。

### 1.2.2 Linux 编译

虚拟机源码目录中提供了 makefile，可以直接在 Linux 使用 make 命令来编译。

## 1.3 预备资料

工欲善其事，必先利其器。在开发 NXVM 虚拟机之前，以下知识有助于把握整个开发

过程，同时列出了关键资料，在开发过程中重点参考。

(a) x86 CPU 指令集知识，包括编码解码过程、指令执行、内存访问
IA-32 Intel Architecture Software Developer's Manual (Volume 1,2,3)
Intel 80386 Manual
http://www.mouseos.com

(b) PC 架构知识，包括 IO 端口功能、BIOS 中断
PC 技术内幕
PC 硬件工程师手册
The 80x86 IBM PC and Compatible Computers (Volume 1,2)
PC 中断大全 BIOS，DOS 及第三方调用的程序员参考资料

(c) 各硬件设备的功能、命令和寄存器
中断控制器, Programmable Interrupt Controller, Intel 8259A
直接内存访问, Direct Memory Access Controller, Intel 8237A
软驱控制器, Floppy Drive Controller, Intel 8272A
键盘控制器, Keyboard Controller
显示适配器, Video Adapter, VGA Standard
CMOS 和时钟, Programmable Internal Timer, 参阅《PC 技术内幕》

(d) 平台相关知识，包括 Win32/Linux 下的键盘输入处理、屏幕显示、多线程

(e) 其他开源虚拟机
Bochs，C++编写的 PC 模拟器，完整模拟了 PC 各设备的功能，可作为最权威的参考来源。NXVM CPU 调试过程中利用 Bochs 的 CPU 来进行比较核对，以修正错误。
easyVM，简单虚拟机，其架构非常轻便。NXVM 的初始架构设计有一部分参考了 easyVM (I/O 端口和指令分派)。
LightMachine，轻型虚拟机，其中的文档值得参考。

# 第二章 虚拟机结构设计

本章描述 NXVM 虚拟机的结构设计思路。

NXVM 的框架决定了虚拟机的运行流程、编码和扩展方式，以及调试方法。通过合理的框架设计，可以循序渐进地编写每个代码模块并执行相应的单元测试和整体测试。

## 2.1 结构划分

NXVM 分为两层，控制台和模拟器。

### 2.1.1 控制台

控制台提供了一个操作界面，通过文本界面接受用户输入的命令，对虚拟机进行配置和控制，例如设定磁盘镜像、内存大小，启动、复位、关闭虚拟机，或者转储指令记录和调试虚拟机中的客户(Guest)操作系统。控制台是用户和模拟器之间的控制接口。

### 2.1.2 模拟器

模拟器是 NXVM 的主体部分，定义了各数据结构，实现了虚拟机的执行逻辑。模拟器提供了一系列外部接口供控制台调用，以控制虚拟机的运行。

模拟器分为 4 个部分，硬件逻辑模拟、平台相关实现、BIOS 程序和调试工具。

#### 2.1.2.1 硬件逻辑模拟

硬件逻辑模拟属于虚拟机的“硬件”部分，包含了虚拟机的核心硬件模块，包括：

端口模拟、内存模拟、中央处理器、中断控制器、内部时钟、CMOS、DMA 控制器、软驱控制器、软盘驱动器、硬盘驱动器。此外还有未实现的键盘控制器和显示适配器。

#### 2.1.2.2 平台相关实现

平台相关实现部分属于虚拟机的“硬件”部分，包含了 Win32 平台实现和 Linux 平台实现，它主要提供了对虚拟机键盘和显示器的模拟，是用户和模拟器之间的操作接口。此外，这部分也提供了一些平台相关的函数接口，例如 Sleep 函数。

为了保证模块化和可移植性，硬件逻辑模拟和平台相关实现这两个部分的耦合度被尽量降低。它们之间只能通过在 vapi 模块中定义接口和回调函数来实现联络。其中，硬件逻辑模拟部分调用平台相关部分的接口函数较少，主要负责启动虚拟机的多线程、设置显示器和磁盘镜像处理；而平台相关部分调用硬件模拟逻辑部分的接口函数较多，以回调函数为名，即 vapiCallBack。

#### 2.1.2.3 BIOS 程序*

BIOS 程序属于虚拟机的“软件”部分，通过汇编来实现基本输入输出功能，符合 IBM PC/AT 的规范标准。这些中断处理程序通过对虚拟机的 I/O 端口功能的调用来实现其功能。NXVM 有两类 BIOS 程序。第一种是 POST 程序，用途是加载磁盘镜像中的客户操作系统（POST 程序）。第二种是中断处理程序，供客户操作系统调用，来实现其底层功能。

由于 NXVM 虚拟机中模拟的某些硬件设备由于缺乏文档资料，I/O 端口功能没有完整实现，导致有一部分本该通过 I/O 端口执行硬件操作的中断处理程序改用 NXVM 的特殊 QDX 指令来运作。例如，10 号 BIOS 中断处理程序的内容仅仅是 QDX 10 和 IRET，而不是

通过 I/O 端口对显卡进行访问。当 NXVM 的 CPU 执行 QDX 10 指令时，虚拟机会调用预先用 C 代码写好的程序来完成相关功能，而不是执行 BIOS 汇编代码。换句话说，本来应该是执行很多条汇编指令才能完成的功能，现在由一条 QDX 特殊指令就完成了。值得注意的是，这并不符合 IBM PC 的实际情况，也是将来版本中需要继续改进、最终消除的。

#### 2.1.2.4 调试工具

调试工具是虚拟机开发过程中的重要帮手，也是在虚拟机成功运行后调试其中客户操作系统的手段。它包括 3 个模块：调试器，汇编器/反汇编器和记录器。调试器具有 16 位调试和 32 位调试的功能，它可以观测和修改 CPU 寄存器，对内存进行查看、修改和搜索。此外，调试器还是进行断点执行、单步执行的入口。NXVM 的汇编器可以对 x86 指令进行汇编，支持 16 位和 32 位指令。在启动虚拟机、加载 BIOS 程序时，BIOS 程序的汇编指令就是通过汇编器转换成机器码，加载到内存中的。NXVM 的反汇编器用于解码和查看内存中的指令，方便调试。当转储 CPU 指令序列时，会调用反汇编器来打印每一条指令，供调试者参考。记录器用于记录每一条执行过的指令和当时的状态，供转储分析。

## 2.2 代码文件

### 2.2.1 程序入口

| | |
|---|---|
| main.c | 虚拟机版本控制和函数入口 |

### 2.2.2 控制台

| | |
|---|---|
| console.ch | 虚拟机的控制台层 |

### 2.2.3 模拟器

模拟器的所有代码在 vmachine 目录中。

#### 2.2.3.1 硬件逻辑模拟

| | |
|---|---|
| vcmos.ch | CMOS |
| vcpu.ch | 中央处理器数据结构 |
| vcpuins.ch | 中央处理器指令集 |
| vdma.ch | 直接内存访问控制器 |
| vfdc.ch | 软驱控制器 |
| vfdd.ch | 软盘驱动器 |
| vglobal.h | 类型、宏定义和编译选项 |
| vhdd.ch | 硬盘驱动器 |
| vkbc.ch | 键盘控制器（未实现） |
| vmachine.ch | 虚拟机组装和控制接口 |
| vpic.ch | 中断控制器 |
| vpit.ch | 内部时钟 |
| vport.ch | 端口模拟 |
| vram.ch | 内存模拟 |
| vvadp.ch | 显示适配器（未实现） |

#### 2.2.3.2 平台相关实现

平台相关实现的所有代码在 vmachine/system 目录中。

| | |
|---|---|
| linux.ch | Linux 下的多线程、键盘输入和屏幕输出 |
| w32adisp.ch | Win32 窗口下的屏幕输出 |
| w32cdisp.ch | Win32 控制台下的屏幕输出 |
| win32.ch | Win32 下的函数接口以及键盘输入 |
| win32app.ch | Win32 窗口下的函数接口和多线程 |
| win32con.ch | Win32 控制台下的函数接口和多线程 |

#### 2.2.3.3 BIOS 程序*

BIOS 程序的所有代码在 vmachine/bios 目录中。

| | |
|---|---|
| post.h | POST 汇编程序 |
| qdbios.ch | BIOS 加载程序和特殊指令分派 |
| qdcga.ch | Quick and Dirty 显示功能 BIOS 中断调用 |
| qddisk.ch | 软盘功能 BIOS 中断调用 |
| | Quick and Dirty 硬盘功能 BIOS 中断调用 |
| qdkeyb.ch | Quick and Dirty 键盘功能 BIOS 中断调用 |
| qdmisc.ch | 机器检测等杂项 BIOS 中断调用 |
| qdrtc.h | 实时时钟 BIOS 中断调用 |

#### 2.2.3.4 调试工具

调试工具的所有代码在 vmachine/debug 目录中。

| | |
|---|---|
| aasm32.ch | x86 汇编器 |
| dasm32.ch | x86 反汇编器 |
| debug.ch | 调试器 |
| record.ch | 记录器 |

## 2.3 运行流程

本节描述 NXVM 的执行流程。

### 2.3.1 程序初始化

虚拟机程序启动（不同于**虚拟机启动**）的调用顺序如下：

main.c: main()

console.c: console()

console.c: init()

vmachine.c: vmachineInit()

vmachineInit 函数进一步调用各模拟硬件设备的 Init 函数，执行内存分配、数据初始化和/或设置函数指针。程序启动完成之后，进入虚拟机控制台等待用户输入命令。

### 2.3.2 虚拟机启动

常规启动的流程如下：

虚拟机控制台中输入 start

console.c: console()

vmachine.c: vmachineStart()

vmachine.c: vmachineReset()

vmachine.c: vmachineResume()

vapi.c: vapiStartMachine()

vmachineReset 函数进一步调用各模拟硬件设备的 Reset 函数，执行硬件复位和加载 BIOS 程序。

调试启动的流程如下：

虚拟机控制台中输入 debug

console.c: console()

debug.c: debug()

调试器提示符中输入 g/t/xg/xt 等命令

debug.c: debug()

debug.c: g()/t()/xg()/xt()

vmachine.c: vmachineResume()

vapi.c: vapiStartMachine()

调试启动之前，必须先在虚拟机控制台中执行 reset 命令，以调用 vmachineReset 函数来进行硬件复位和 BIOS 程序加载。因为程序启动时并不会自动做这些。

从上述流程中，我们可以看到，除去命令解析和初始化硬件以外，真正启动虚拟机的函数其实是 vmachineResume 和 vapiStartMachine。前者将 vmachine.flagrun 置位，表示虚拟机处于运行状态，然后调用后者。后者是一个平台相关的函数，因为接下来就要初始化键盘、显示器等平台相关组件，并创建相应的线程。

我们以 Win32 窗口下的虚拟机代码为例，Linux 代码同理：

vapi.c: vapiStartMachine()

win32.c: win32StartMachine()

win32app.c: win32appStartMachine()

```
CreateThread(NULL, 0, ThreadDisplay, NULL, 0, &ThreadIdDisplay);
CreateThread(NULL, 0, ThreadKernel, NULL, 0, &ThreadIdKernel);
```

在 win32appStartMachine 中，分别为显示器和虚拟机核心创建了一个线程。显示器的线程实际上是 Win32 窗口线程，负责定时打印屏幕到窗口，以及接收键盘消息。至于当前的线程，则返回到 console 函数，接受用户的控制命令输入。但是，在 Win32 控制台模式和 Linux 终端模式下，虚拟机所创建的显示器线程仅仅负责打印屏幕，当前线程则负责接收键盘消息而不是返回到 console 函数。因此，当虚拟机启动并运行在 Win32 控制台模式或 Linux 终端模式时，不存在虚拟机控制台，也不能输入虚拟机控制命令，要停止虚拟机的执行，只能使用快捷键 F9。

### 2.3.3 虚拟机运行

我们接着看虚拟机的核心线程：

win32app.c: ThreadKernel()

vmachine.c: vapiCallBackMachineRun()

vapiCallBackMachineRun 是虚拟机的引擎，其中只有一个 while 循环，不断执行 vmachineRefresh 函数。它调用各模拟硬件设备的 Refresh 函数来推动模拟器的运转。在其中，最关键的就是 vcpuRefresh 及其调用的 vcpuinsRefresh 函数。每当 Refresh 循环一次，就有一条 x86 指令被读取、解码和执行。只有当 vmachine.flagrun 标志被置零时，才退出该循环，从而关闭虚拟机核心的线程，停止运行。

### 2.3.4 虚拟机停止

NXVM 虚拟机中的 vmachine.flagrun 标志位十分重要，它控制着虚拟机核心线程(vapiCallBackMachineRun)的运转。将该标志置零有好几种方法：在虚拟机控制台中使用 stop 命令，在虚拟机显示窗口中按 F9 快捷键，在 Win32 窗口模式下关闭虚拟机显示窗口，或者虚拟机中的软件执行了"QDX 00"指令。

让我们看看 vmachine.flagrun 清零时，正在运行的虚拟机会发生什么：

在 Win32 窗口模式，核心线程将退出，显示线程则保持原样，而虚拟机控制台作为主线程始终存在；在另外两种模式下，显示线程和核心线程将退出，同时主线程将结束对键盘消息的监听，返回 console，出现虚拟机控制台。

### 2.3.5 虚拟机恢复运行

当虚拟机被停止，我们可以使用调试器来查看当前状态、修改寄存器或使用虚拟机控制台的 record 命令打印指令流。完成调试后，可以在虚拟机控制台中使用 resume 命令来恢复运行，或者在调试器中再次使用 g/t/xg/xt 命令。这些命令都调用同一个函数 vmachineResume 来恢复运行。vmachineResume 将 vmachine.flagrun 置位，然后调用 vapiStartMachine 函数启动虚拟机。

## 2.4 编码规范

本节描述 NXVM 虚拟机的编码规范。NXVM 虚拟机使用 C 语言编写。

### 2.4.1 全局类型

在 vglobal.h 中，虚拟机对数据类型进行了重新封装，定义了普通数据、指针和解引用。所有数据类型均应使用 vglobal.h 中的定义。

整数类型为 t_nubit；

布尔类型为 t_bool；

字符串类型为 t_string，长度为 256；字符串指针为 t_strptr；

浮点类型为 t_float；

数据指针为 t_vaddrcc，函数指针为 t_faddrcc。

以上均为类型定义。相应类型的指针以 p_开头，解引用以 d_开头。

函数指针的解引用（即函数调用）方法是使用 ExecFun()宏。

### 2.4.2 编译选项

在 vglobal.h 的开头部分，用户可以指定编译选项。

对于 64 位编译，应当将宏变量 VGLOBAL_SIZE_INTEGER 的值设置为 64；对于 32 位编译，则设置为 32 或者 64。

在 Win32 下编译时，应保证宏变量 VGLOBAL_PLATFORM 的值等于 VGLOBAL_VAR_WIN32，否则 VGLOBAL_VAR_LINUX。这个值一般不会出错，因为我们在代码中预先检查了宏变量_WIN32。

通常而言，保证宏变量 VGLOBAL_SIZE_INTEGER 的值设置为 64，不需要再额外关注编译选项。

### 2.4.3 函数和宏函数的命名规则

NXVM 中，各模块的内部函数（static）的命名只要能表达其功能即可。

对于硬件逻辑模拟部分自身的模块间的接口函数，应当命名为“模块名+功能”，例如

vcpuReset、vramRealByte、linuxStartMachine。平台相关实现部分同理。

所有硬件逻辑模拟部分的模块都有 Init、Reset、Refresh 和 Final 函数，这四个函数分别用于虚拟机程序启动时的内存分配、虚拟机运行前的硬件复位和数据初始化、虚拟机运转时的执行和虚拟机程序退出时的内存析构。大部分硬件模块还有 IO_Read 和 IO_Write 类型的函数，它们定义了各自硬件的 I/O 端口功能。

对于硬件模拟部分调用平台相关部分的函数，应命名为“vapi+模块名+功能”，例如 vapiFloppyInsert。在编译时，vapi.c 中平台相关函数会检测当前编译选项，并选择合适的平台功能函数。

对于平台相关部分调用硬件模拟部分的函数，应命名为“vapiCallBack+模块名+功能”，例如 vapiCallBackMachineStop。

虚拟机控制台、虚拟机模拟器的 BIOS 程序和调试工具中的接口函数较少，可直接参阅相关头文件。

### 2.4.4 宏定义的命名规则

各模块的宏定义应命名为“模块名+类型+功能”，例如 VCPU_EFLAGS_VM、VGLOBAL_VAR_WIN32 等。

# 第三章 虚拟机控制台

本章描述 NXVM 虚拟机的控制台功能设计和实现，对应的源代码文件是 console.ch。

## 3.1 功能设计

打开虚拟机后，键入'HELP'可看到如下菜单：

```
NXVM Console Commands
=====================
HELP      Show help info
EXIT      Quit the console
INFO      List all NXVM info

DEBUG     Launch NXVM hardware debugger
RECORD    Record cpu status for each instruction

SET       Change BIOS settings
DEVICE    Change hardware parts
NXVM      Change virtual machine status
```

本节具体描述以上命令的功能。

### 3.1.1 快速命令

| | |
|---|---|
| mode | 切换显示模式：Win32 窗口或 Win32 控制台 |
| start | 启动虚拟机（加载 BIOS，复位硬件并启动） |
| reset | 复位虚拟机（加载 BIOS，复位硬件） |
| stop | 停止虚拟机 |
| resume | 恢复虚拟机运行（仅启动） |

这类命令属于常用命令。可以发现，start 命令实际上等于是联用 reset 和 resume 命令。

### 3.1.2 帮助

单独使用 HELP 命令可打印出命令菜单；

使用 HELP [command] 可打印具体命令的用法。

### 3.1.3 退出

键入 EXIT 退出虚拟机。只有在虚拟机停止运行的情况下才可以退出。

### 3.1.4 打印状态

键入 INFO 打印虚拟机的状态，包括硬件配置、EXIT 设置和调试状态。

### 3.1.5 调试

键入 DEBUG 进入虚拟机的调试器后，使用"?"命令可以打印 16 位调试命令菜单。大部分情况下，虚拟机调试器的 16 位调试命令用法类似于一个 MSDOS 的调试器(debug.exe)。使用"x?"命令可以打印 32 位调试命令菜单。调试器具体用法请参阅调试器章节。

### 3.1.6 记录

记录器的状态可以使用 INFO 命令查看。

记录器有两种使用方法，记录当前所有指令并同步转储（即时记录），或者记录最后 65535 条指令并再转储（事后记录）。

对于前者，可以使用“RECORD now 文件名”指定转储文件。当虚拟机启动后，所有执行过的指令都将直接写入转储文件，直到虚拟机停止。

对于后者，在启动虚拟机之前，使用“RECORD on”命令打开记录器，使用“RECORD off”可以关闭记录器。当虚拟机停止（触发断点、F9 快捷键或使用 stop 命令等）后，记录器会自动关闭。此时使用“RECORD dump 文件名”来转储虚拟机最后执行的 65535 条指令。

### 3.1.7 BIOS 设置

BIOS 设置命令的格式为：SET <item> <value>，例如“set boot fdd”。

当前支持的设置项如下：

| ITEM | VALUE | 描述 |
|---|---|---|
| boot | fdd | 从软盘镜像启动 |
| | hdd | 从硬盘镜像启动 |

### 3.1.8 设备配置

DEVICE 设置命令的格式较为多变，以下分点描述。

#### 3.1.8.1 设置内存

命令格式：

| DEVICE | ram | <size> | 分配 size 大小的内存，单位是 KB。默认值是 16384。 |
|---|---|---|---|

例如，分配 4MB 的内存，就使用命令“DEVICE ram 4096”。

#### 3.1.8.2 设置显示器

命令格式：

| DEVICE | display | console | 默认值。使用 Win32 控制台作为虚拟机输入输出。 |
|---|---|---|---|
| | | window | 额外创建 Win32 窗口作为输入输出，保留虚拟机控制台。 |

在 Linux 下，该命令无效。

在 Win32 下，命令“DEVICE display console”使用 Win32 控制台作为输入输出，虚拟机控制台被覆盖；命令“DEVICE display window”则额外创建一个窗口作为输入输出。

该命令的快捷方式为 MODE 命令。

#### 3.1.8.3 设置软盘镜像

命令格式：

| DEVICE | fdd | create | 创建空白的软盘镜像供虚拟机使用 |
|---|---|---|---|
| | | remove | 移除软盘镜像 |
| | | insert <file> | 读入镜像文件供虚拟机使用 |
| | | remove <file> | 移除软盘镜像，并写入到文件中 |

#### 3.1.8.4 设置硬盘镜像

命令格式：

| DEVICE | hdd | create | 创建空白的硬盘镜像供虚拟机使用，20 个磁道 |
|---|---|---|---|
| | | create cyl <num> | 创建空白的硬盘镜像供虚拟机使用，num 个磁道 |
| | | connect <file> | 读入镜像文件供虚拟机使用 |

| | | | |
|---|---|---|---|
| | | disconnect | 移除硬盘镜像 |
| | | disconnect <file> | 移除硬盘镜像，并写入到文件中 |

### 3.1.9 运行命令

命令格式：

| | | |
|---|---|---|
| NXVM | start | 启动虚拟机（加载 BIOS，复位硬件并启动） |
| | reset | 复位虚拟机（加载 BIOS，复位硬件） |
| | stop | 停止虚拟机 |
| | resume | 恢复虚拟机运行（仅启动） |

以上命令都有各自的快捷方式，因此去掉 NXVM 也执行同样功能。

## 3.2 代码实现

虚拟机控制台的源代码较为简单，通过设置各种标志和调用相关接口函数实现功能。

主要函数为 console。该函数是一个循环，用户输入命令执行相应功能后返回该循环等待下一条命令，直到退出。

# 第四章 调试工具

本章描述 NXVM 虚拟机的调试工具的功能定义、数据结构和代码实现，对应的源代码文件在 vmachine/debug。

## 4.1 调试器

调试器是 NXVM 虚拟机的重要工具，从功能和用法上看，也是虚拟机控制台的一部分，通过键入 DEBUG 命令开启。它是用户控制和调试虚拟机的接口，可以查看、修改 CPU 寄存器，查看、修改、搜索模拟内存，访问模拟硬件的 I/O 端口，断点或单步执行 CPU 指令，监视内存访问和指令执行。调试器的源代码文件是 vmachine/debug/debug.ch。

调试器中输入的数值均为 16 进制数值。

### 4.1.1 数据结构

该模块定义的全局变量是 t_debug vdebug，结构定义：

```
typedef struct {
    t_bool      flagbreak;                  /* breakpoint set (1) or not (0) */
    t_bool      flagbreakx;
    t_nubitcc breakcnt;
    t_nubit16 breakcs, breakip;
    t_nubit32 breaklinear;
    t_nubitcc tracecnt;
} t_debug;
```

该结构中，flagbreak 控制 16 位断点。当它置位时，虚拟机核心线程的每个循环都会检测 breakcs 和 breakip，符合条件时就停止运行。flagbreakx 控制 32 位断点。当它置位时，虚拟机核心线程的每个循环都会检测 breaklinear，符合条件时就停止运行。breakcnt 是断点计数器，它记录着断点前执行的指令数。

tracecnt 控制虚拟机核心线程执行多少次循环，为 1 则是单步执行。

### 4.1.2 16 位调试器

在 16 位调试器中，有效的地址格式可以是偏移量或逻辑地址。对于偏移量地址，调试器会自动采用默认的段，例如，对于 a、g、u、t 等操作指令的命令，默认段是 cs，对于 c、d、e、f、s 等操作内存数据的命令，默认段是 ds。对于逻辑地址，可以直接采用段地址：偏移量的格式，如 0100:0200，也可以采用助记符，如 ds:1234。

进入调试器后，输入“?”可以打印 16 位调试器命令：

```
assemble        A [address]
compare         C range address
dump            D [range]
enter           E address [list]
fill            F range list
go              G [[address] breakpoint]
hex             H value1 value2
input           I port
```

load L [address]
move M range address
name N pathname
output O port byte
quit Q
register R [register]
search S range list
trace T [[address] value]
unassemble U [range]
verbal V
write W [address]
debug32 X?

以下表格描述了这些命令。

| 命令 | 参数 | 示例 | 描述 |
|---|---|---|---|
| ? | | ? | 打印帮助菜单 |
| a | [address] | a | 从上次汇编结束处继续汇编 8086 指令 |
| | | a 100 | 在内存的 cs:0100 处汇编 8086 指令 |
| c | range address | c 100 200 300 | 比较两块内存，第一块从 ds:0100 到 ds:0200，第二块起始地址是 ds:0300 |
| d | range | d | 从上次显示内存内容结尾处开始显示 80 字节内存 |
| | | d 100 | 显示内存内容，从 ds:0100 开始，共 80 字节 |
| | | d 100 200 | 显示内存内容，从 ds:0100 到 ds:0200 |
| e | address [list] | e 100 | 显示 ds:0100 处的 1 个字节，并提示输入新值 |
| | | e 100 a b c d | 将 0x0a, 0x0b, 0x0c, 0x0d 依次写入 ds:0100 |
| f | range list | f 100 200 a b c | 将 0x0a, 0x0b, 0x0c 依次重复地从 ds:0100 开始填充到 ds:0200 为止 |
| g | [[address] breakpoint ] | g | 从当前 CPU 指令指针指向处开始执行 |
| | | g 100 | 从当前 CPU 指令指针指向处开始执行，到 cs:0100 为止 |
| | | g 100 200 | 从 cs:0100 执行到 cs:0200 为止 |
| h | value1 value2 | h 100 200 | 显示 value1 和 value2 的和、差 |
| i | port | i 70 | 从 70h 端口读入 1 个字节并打印 |
| l | [address] | l | 将 n 命令指定的文件读取到内存的默认 cs:0100 处 |
| | | l 200 | 将 n 命令指定的文件读取到内存的 cs:0200 处 |
| m | range address | m 100 200 300 | 移动内存，源数据从 ds:0100 到 ds:0200，目的地的起始地址为 ds:0300 |
| n | pathname | n d:\test.com | 指定文件名，供读取或写入 |
| o | port byte | o 71 ff | 将数据 0xff 写入端口 71h |
| q | | q | 退出调试器，返回虚拟机控制台 |
| r | [register] | r | 显示 CPU 的 16 位寄存器内容和当前指令 |

| | | | |
|---|---|---|---|
| | | r ax | 显示 ax 寄存器的内容，并提示输入新值 |
| s | range list | s 100 200 a b c | 搜索字节序列 0x0a,0x0b,0x0c，<br>搜索范围是 ds:0100 到 ds:0200 |
| t | [[address] value] | t | 从当前 CPU 指令指针指向处开始，执行 1 条指令 |
| | | t 10 | 从当前 CPU 指令指针指向处开始，执行 0x10 条指令 |
| | | t 100 10 | 从 cs:0100 处开始执行 0x10 条指令 |
| u | [range] | u | 从上次反汇编结束处开始，解码 32 个字节 |
| | | u 100 | 从 cs:0100 处开始，解码 32 个字节 |
| | | u 100 200 | 从 cs:0100 处开始解码，直到 cs:0200 |
| v | | v | 输入字符串，显示对应的 ASCII 码 |
| x | | x? | 扩展调试命令，即 32 位调试命令 |

### 4.1.3 32 位调试器

在 32 位调试器中，地址格式是 32 位线性地址。

进入调试器后，输入“x?”可以打印 32 位调试器命令：

assemble XA [address]
compare XC addr1 addr2 count_byte
dump XD [address [count_byte]]
enter XE address [byte_list]
fill XF address count_byte byte_list
go XG [address [count_instr]]
move XM addr1 addr2 count_byte
register XR [register]
  regular XREG
  segment XSREG
  control XCREG
search XS address count_byte byte_list
trace XT [count_instr]
unassemble XU [address [count_instr]]
watch XW r/w/e address

以下表格描述了这些命令。

| 命令 | 参数 | 示例 | 描述 |
|---|---|---|---|
| x? | | x? | 打印 32 位调试器帮助菜单 |
| xa | [address] | xa | 从上次汇编结束处继续汇编 80386 指令 |
| | | xa 100 | 在内存的 0x00000100 处汇编 80386 指令 |
| xc | addr1 addr2 count_byte | xc 100 200 10 | 比较两块内存，第一块在 0x00000100，第二块在 0x00000200，比较 0x10 个字节 |
| xd | [address [count_byte] | xd | 从上次显示内存内容结尾处开始显示 80 字节内存 |
| | | xd 100 | 显示内存内容，从 0x00000100 开始，默认 80 字节 |
| | | xd 100 20 | 显示内存内容，从 0x00000100 开始的 0x20 个字节 |
| xe | address [byte_list] | xe 100 | 显示 0x00000100 处的 1 个字节，并提示输入新值 |

| | | | |
|---|---|---|---|
| | | xe 100 a b c d | 将 0x0a, 0x0b, 0x0c, 0x0d 依次写入 0x00000100 |
| xf | address count_byte byte_list | xf 100 20 a b c | 将 0x0a, 0x0b, 0x0c 依次重复地从 0x00000100 开始填充，一共填充 0x20 个字节 |
| xg | [address [count_instr] | xg | 从当前 CPU 指令指针指向处开始执行 |
| | | xg 100 | 从当前 CPU 指令指针指向处开始执行，到 0x00000100 为止 |
| | | xg 100 10 | 从当前 CPU 指令指针指向处开始执行，到 0x00000100 为止，为第 1 次循环。之后再从 0x00000100 执行到 0x00000100。一共循环 0x10 次 |
| xm | addr1 addr2 count_byte | xm 100 200 10 | 移动内存，源地址 0x00000100，目的地 0x00000200，一共复制 0x10 个字节 |
| xr | [register] | xr | 显示 CPU 的 32 位通用寄存器内容和当前指令 |
| | | xr eax | 显示 CPU 寄存器 eax 的内容，并提示输入新值 |
| xreg | | xreg | 显示 CPU 的 32 位通用寄存器内容和当前指令 |
| xsreg | | xsreg | 显示 CPU 的段寄存器内容 |
| xcreg | | xcreg | 显示 CPU 的控制寄存器内容 |
| xs | address count_byte byte_list | xs 100 200 a b c | 搜索字节序列 0x0a,0x0b,0x0c，搜索范围是 0x00000100 开始的 0x200 个字节 |
| xt | [count_instr] | xt | 从当前 CPU 指令指针指向处开始执行 1 条指令 |
| | | xt 10 | 从当前 CPU 指令指针指向处开始执行 0x10 条指令 |
| xu | [address [count_instr]] | xu | 从上次反汇编结束处开始，解码 16 条指令 |
| | | xu 100 | 从 0x00000100 处开始，解码 16 条指令 |
| | | xu 100 20 | 从 0x00000100 处开始，解码 0x20 条指令 |
| xw | r/w/e address | xw r 123 | 监视 0x00000123 处的读取操作并打印 |
| | | xw w 456 | 监视 0x00000456 处的写入操作并打印 |
| | | xw e 789 | 监视 0x00000789 处的指令执行并打印 |

## 4.2 记录器

记录器用于记录和转储虚拟机 CPU 所执行过的所有指令和当时的状态。用户通过虚拟机控制台来操作记录和转储功能。记录器的功能定义参阅虚拟机控制台一章。记录器的源代码是 vmachine/debug/record.ch。

### 4.2.1 数据结构

该模块定义的全局变量是 t_record vrecord，结构定义：

```
#define RECORD_SIZE          0xffff              /* maximum record number */
typedef struct {
    t_cpu   recpu[RECORD_SIZE];
```

```
        t_nubitcc start, size;
        t_bool flagrecord; /* recorder is turned on */
        t_bool flagready;
        t_bool flagnow; /* record now! */
        char fn[0x100];
        FILE *fp;
    } t_record;
```

该结构中，recpu 是一个长度为 65535 的 CPU 数组，是记录器的主要部分。记录器是一个基于数组的 FIFO 队列，start 和 size 用于标记队列的起始位置和长度。flagrecord 控制着记录器是否打开。flagready 表明记录器是否完成初始化，如果没有，则初始化之。flagnow 则表明记录器是否在进行即时转储。fn 和 fp 保存了转储的文件名和文件指针。

### 4.2.2 代码实现

该模块有如下接口：

void recordNow(const t_strptr fname); 打开记录文件供写入；设置 flagnow

void recordDump(const t_strptr fname); 将队列中的记录写入文件

void recordInit(); 初始化记录器

void recordExec(t_cpu *rcpu); 将 rcpu 指向的 CPU 内容进行记录；假如是即时记录，就写入文件，否则将这条记录加入队列

void recordRefresh(); 将 vcpu 的内容进行记录，调用 recordExec 完成

void recordFinal(); 记录器释放内存

## 4.3 汇编器

NXVM 的汇编器是一个微型编译器，用户输入的合法汇编指令以字符串形式传入 aasm32 或 aasm32x 函数，将编译后的机器码放入数组 rcode，并返回机器码的长度（aasm32 是 1~15，aasm32x 是 1~FFFFFFFFh）。假如返回 0，则表示汇编出错。汇编器的源代码是 vmachine/debug/aasm.ch。

### 4.3.1 功能定义

汇编器有 2 个不同的接口供使用。第一个是单指令汇编，第二个是批量汇编。

#### 4.3.1.1 单指令汇编

单指令汇编对一条汇编指令进行编译。它主要用于调试器的汇编功能，同时也是批量汇编的基础。

单指令汇编的函数原型：

```
t_nubit8 aasm32(const t_strptr stmt, t_vaddrcc rcode);
```

NXVM 汇编语句大致遵循 Intel 的格式，包括指令、操作数和注释，例如：

"CMP DWORD PTR DS:[EAX+EBX*2+abcd], 12345678  ; compare 32-bit numbers"

"CALL FAR PTR DWORD PTR CS:[BX+DI]  ; call far pointer at bx+di"

#### 4.3.1.2 批量汇编

批量汇编对多条汇编指令（段落）进行编译。段落存放于一个字符串中，各条指令用‘\n’分隔。它主要用于 BIOS 程序的加载。查看 vmachine/bios 目录中的几个头文件，会发现 BIOS 程序被定义为若干个长字符串，每个字符串都是由汇编指令组成的段落。

批量汇编的函数原型：

t_nubit32 aasm32x(const t_strptr stmt, t_vaddrcc rcode);

批量汇编除了可一次性编译多条指令外，还支持跳转标签功能。用户可在段落的任意位置定义字符串，例如："$(LABEL_EXAMPE):"。在标签定义的前面或后面，用户可以使用 jmp 或 call 指令来实现跳转，例如："JNZ $(LABEL_EXAMPLE)"。

跳转分为 3 种，短跳转 SHORT、近跳转 NEAR 和远跳转 FAR。短跳转可以跳转到当前指令的前 127 字节或后 128 字节；近跳转可以跳转到当前段内的任意位置；远跳转可以跨段跳转到内存中的任意位置。在 NXVM 汇编器的批量汇编中，不能使用远跳转 FAR 来跳转到一个标签，因为批量汇编产生的机器码必须放在同一个段中。

对于循环指令（LOOPcc）和 JCXZ 指令，可以使用限定词 SHORT，默认为 SHORT。

对于条件跳转指令(Jcc)，可以使用限定词 SHORT 或 NEAR，默认为 SHORT。

对于无条件跳转指令（JMP），可以使用限定词 SHORT 或 NEAR，默认为 NEAR。

对于函数调用指令（CALL），可以使用限定词 NEAR，默认为 NEAR。

### 4.3.2 代码实现

汇编器包括词法分析、语法分析和代码生成。因为汇编相对简单，不需要语法树。

词法分析器是一个确定性有限状态自动机（DFA），函数是 gettoken。该函数读取每个字符，并产生相应的状态，直至获取一个合法的标记（token）。每调用一次 gettoken，就按字符串中的顺序返回一个新的标记。

语法分析器是 parsearg 函数。它调用 gettoken 获取标记，进而根据标记和标记的组合方式获取汇编语句的真实含义，记录到 t_aasm_oprinfo 结构体中，供代码生成器使用。

代码生成器是 exec 函数。它首先判断汇编指令，然后选择合适的指令编码函数产生机器码。

aasm32 是一个 32 位的汇编器，对 16 位和 32 位的指令一起进行汇编。我们知道，指令编码和译码过程中，需要根据代码段的属性（D/B 位）和前缀指令（66h 和 67h）判断操作数的位数和寻址的位数。在 aasm32 中，假如代码段的位数和指令的操作数位数不同，则加上 66h 前缀；假如代码段的位数和指令的寻址位数不同，则加上 67h 前缀。

## 4.4 反汇编器

NXVM 反汇编器将机器码解译成可读的字符串。它主要用于调试器的反汇编功能和记录器的转储功能，方便用户对虚拟机进行调试。

### 4.4.1 功能定义

反汇编器每次解码一条指令。函数原型：

t_nubit8 dasm32(const t_strptr stmt, t_vaddrcc rcode);

它读取 rcode 中的机器码，根据 CPU 的译码方式，打印成汇编语句字符串，放入 stmt 中。返回的数值为该指令的编码长度（1~15），如果为 0，表示译码失败。

### 4.4.2 代码实现

反汇编器源代码 dasm32.ch 是根据 vcpuins.ch 修改编写的，它的执行流程和 vcpuins.ch 中的解码过程完全一致，只是把指令执行替换成了指令打印输出。因此代码实现部分可以参阅中央处理器一章。

# 第五章 虚拟机模拟器接口

本章描述 NXVM 虚拟机的模拟器接口的功能设计、数据结构和代码实现，对应的源代码文件是 vmachine/vmachine.ch。

## 5.1 功能设计

虚拟模拟器接口模块的功能是提供 vmachine 全局变量，用于标志虚拟机的运行状态。它同时包含一系列用于控制虚拟机的接口函数。

## 5.2 数据结构

该模块定义的全局变量是 t_machine vmachine，结构定义：

```
typedef struct {
    t_bool    flagrun;        /* vmachine is running (1) or not running (0) */
    t_bool    flagmode;       /* mode flag: console (0) or window (1) */
    t_bool    flagboot;       /* boot from floppy (0) or hard disk (1) */
    t_bool    flagreset;
} t_machine;
```

该结构中，flagrun 控制虚拟机的运行状态，置位时表明正在运行。虚拟机运行时，显示器、键盘和核心线程均会监视该标志，一旦置零则停止运行。

flagmode 控制虚拟机的显示模式，置零时运行于 Win32 控制台，否则运行于 Win32 窗口。默认为零。

flagboot 控制虚拟机的启动镜像，置零时从软盘启动，否则从硬盘启动。默认为零。

flagreset 控制虚拟机的复位。当键入 reset 命令或虚拟机中的软件执行了 QDX 01 特殊指令，该标志置位。当虚拟机正在运行时，核心循环函数 vmachineRefresh 会执行 vmachineReset 进行复位，然后继续循环。当虚拟机不运行时，该位不被使用。这是因为当虚拟机不运行时，可以直接调用 vmachineReset 进行复位。而当虚拟机正在运行时，核心线程以外的线程（如控制台线程）直接调用 vmachineReset 复位会造成访问冲突。我们必须保证由核心线程来执行复位。

## 5.3 代码实现

该模块有如下接口：

```
void vmachineStart();
```

启动虚拟机：复位虚拟机并启动

```
void vmachineReset();
```

复位虚拟机

```
void vmachineStop();
```

停止虚拟机：置零 vmachine.flagrun

```
void vmachineResume();
```

恢复虚拟机运行：执行平台相关部分的启动

void vmachineRefresh();

虚拟机核心循环函数，

void vmachineInit();

虚拟机程序启动时初始化（分配内存、设置函数指针）

void vmachineFinal();

虚拟机程序关闭时析构内存

void vapiCallBackMachineRun();

虚拟机引擎，用 while 循环调用 vmachineRefresh。供平台相关部分创建虚拟机核心线程时调用。

t_bool vapiCallBackMachineGetFlagRun();

获取运行状态。供平台相关部分调用。为了保证隔离，硬件逻辑模拟部分可以直接检测 vmachine.flagrun，而平台相关部分不能。

void vapiCallBackMachineReset();

复位虚拟机。供 vcpuins.c 调用，因为 vcpuins.c 不能包含 vmachine.h，只能通过 vapi.h 调用。

void vapiCallBackMachineStop();

停止虚拟机。当按下 F9 快捷键或关闭 Win32 显示器窗口时，该函数被平台相关部分调用。当执行 QDX 00 特殊指令，vcpuins.c 调用该函数。

# 第六章 I/O 端口模拟

本章描述虚拟机硬件逻辑模拟部分 I/O 端口模块的功能设计、数据结构和接口函数，对应的源代码文件是 vmachine/vport.ch。

## 6.1 功能设计

I/O 端口提供了 CPU 和外部设备的接口。I/O 端口模块的功能是提供 vport 全局变量。所有硬件逻辑模拟的模块都分别实现各自的 I/O 端口功能函数，然后连接到 vport 的函数指针数组。I/O 端口的数据交换都通过 vport 中的 iodword 来进行。具体地，IO_Read 类型的函数会修改 vport.iodword，供 CPU 或者调试器读取；IO_Write 类型的函数会使用 vport.iodword 的数值进行操作，而该数值必须先由 CPU 或调试器赋值。

## 6.2 数据结构

该模块定义的全局变量是 t_port vport，结构定义：

```
typedef struct {
    t_faddrcc in[0x10000];
    t_faddrcc out[0x10000];
    union {
        t_nubit8    iobyte;
        t_nubit16 ioword;
        t_nubit32 iodword;
    };
} t_port;
```

该结构中，in 和 out 分别是 IO_Read 和 IO_Write 函数的指针数组。数组的索引值即端口号。NXVM 程序启动时，I/O 端口模块首先将所有的 in 函数指针指向 IO_Read_VOID，out 函数指针指向 IO_Write_VOID，表明此时各端口未连接。接着各硬件逻辑模拟模块的 Init 函数会将各自的 IO 函数连接到这两个指针数组，完成虚拟机的“硬件”连接。

iodword 结构体则表示输入、输出的数据值。当其他硬件模块执行 IO_Read 函数时，将根据操作数的大小写入 iobyte/ioword/iodword 供 CPU 或调试器读取。而当 CPU 或调试器要对端口写数据时，会先写入 iobyte/ioword/iodword，然后调用相应端口的 IO_Write 函数读取该操作数，并执行其他操作。

## 6.3 接口函数

该模块有如下接口：

```
void IO_Read_VOID();
```

空函数，未定义的输入端口

```
void IO_Write_VOID();
```

空函数，未定义的输出端口

void vportInit();

虚拟机程序启动初始化函数，将所有端口连接到未定义函数

void vportReset();

虚拟机复位函数，将 iodword 清零

void vportRefresh();

虚拟机执行函数，空

void vportFinal();

虚拟机退出函数，空

# 第七章 内存模拟

本章描述虚拟机硬件逻辑模拟部分内存模块的功能设计、数据结构和接口函数，对应的源代码文件是 vmachine/vram.ch。

为了避免混淆，虚拟机中运行的软件所访问的物理地址称为**客户地址**，虚拟机模拟内存块在宿主操作系统中的地址称为**宿主地址**，C 代码中的变量取址得到的地址称为**引用地址**。宿主地址和引用地址都是虚拟机应用程序在宿主操作系统中的线性地址。

## 7.1 功能设计

内存模块负责模拟虚拟机的物理内存。我们在宿主操作系统中简单地动态分配一块字节数组作为虚拟机的物理内存。该模块提供了 vram 全局变量和一些宏函数，使得虚拟机的各硬件逻辑模拟模块可以直接按照物理地址访问。注意，CPU 的内存分段机制和内存分页机制下的内存访问由 CPU 模块的内存访问单元（MMU）负责，与内存模块无关。

## 7.2 数据结构

该模块定义的全局变量是 t_ram vram，结构定义：

```
typedef struct {
    t_bool      flaga20;                    /* 0 = disable, 1 = enable */
    t_vaddrcc base;                         /* memory base address is 20 bit */
    t_nubitcc size;                         /* memory size in byte */
} t_ram;
```

在这个结构体中，flaga20 控制着 A20 地址线的开启与关闭。在 16 位 IBM-PC 系统中，地址线有 A0~A19，共 20 位，可以寻址 1MB 的内存空间。当地址超过 1MB 时，超过 20 位的部分被截断，形成了环绕返转（Warp Around）。在 32 位 PC 中，为了保证和 16 位系统的兼容性，IBM 的 PC 设计者们设置了 A20 开关。当 A20 关闭时，超过 20 位的地址被截断为 20 位，和 16 位情况下一致；当 A20 开启时，就可以访问 1MB 以上的地址空间了。实际情况中，A20 地址线的开关由键盘控制器负责，但在 NXVM 中为了方便，放在了内存模块中，参阅 7.3 节。

base 是一个变量指针，指向虚拟机物理内存块在宿主操作系统中的起始地址。size 则表明有多少字节被分配作虚拟机物理内存。

## 7.3 接口函数

### 7.3.1 宏定义

内存模块定义了如下宏变量和宏函数：

```
#define vramSize vram.size
```

获取分配的内存大小

```
#define vramWrapA20(offset) ((offset) & ((vram.flaga20) ? 0xffffffff : 0xffefffff))
```

根据 A20 开关决定客户地址（参数 offset 应改为 physical）

```
#define vramAddr(physical) (vram.base + (t_vaddrcc)(vramWrapA20(physical)))
```

将客户地址转换为宿主地址，这样可以直接地通过 C 代码“解引用”来访问。

```
#define vramByte(physical)    (d_nubit8(vramAddr(physical)))
```

访问客户物理地址的 1 个字节（BYTE）

```
#define vramWord(physical)    (d_nubit16(vramAddr(physical)))
```

访问客户物理地址的 2 个字节（1 个字，WORD）

```
#define vramDWord(physical) (d_nubit32(vramAddr(physical)))
```

访问客户物理地址的 4 个字节（1 个双字，DWORD）

```
#define vramQWord(physical) (d_nubit64(vramAddr(physical)))
```

访问客户物理地址的 8 个字节（1 个四字，QWORD）

```
/* macros below are designed for real-addressing mode */
#define vramIsAddrInMem(ref) \
    (((t_vaddrcc)(ref) >= vram.base) && ((t_vaddrcc)(ref) < (vram.base + vram.size)))
```

判断引用地址是否指向内存块宿主地址之中

```
#define vramGetRealAddr(segment, offset)    (vram.base + \
    (((((t_nubit16)(segment) << 4) + (t_nubit16)(offset)) & \
      (vram.flaga20 ? 0xffffffff : 0xffefffff)) % vram.size))
```

将实模式下的逻辑地址转换为物理地址

```
#define vramRealByte(segment, offset)    (d_nubit8(vramGetRealAddr(segment, offset)))
```

通过实模式下的逻辑地址直接访问 1 个字节（BYTE）

```
#define vramRealWord(segment, offset)    (d_nubit16(vramGetRealAddr(segment, offset)))
```

通过实模式下的逻辑地址直接访问 2 个字节（1 个字，WORD）

```
#define vramRealDWord(segment, offset) (d_nubit32(vramGetRealAddr(segment, offset)))
```

通过实模式下的逻辑地址直接访问 4 个字节（1 个双字，DWORD）

```
#define vramRealQWord(segment, offset) (d_nubit64(vramGetRealAddr(segment, offset)))
```

通过实模式下的逻辑地址直接访问 8 个字节（1 个四字，QWORD）

### 7.3.2 函数定义

```
void IO_Read_0092();
```

读取 92h 键盘控制器端口，实际上用于判断 A20 的开关状态

```
void IO_Write_0092();
```

写入 92h 键盘控制器端口，实际上用于写入 A20 的开关状态

```
void vramAlloc(t_nubitcc newsize);
```

重新分配内存空间，供“DEVICE ram <size>”命令修改模拟内存的大小

```
void vramInit();
```

虚拟机程序启动时硬件连接，设置函数指针

```
void vramReset();
```

虚拟机硬件复位，清空内存，关闭 A20

```
void vramRefresh();
```

虚拟机执行函数，空

```
void vramFinal();
```

程序终止函数，析构所分配的内存空间

# 第八章 中央处理器

中央处理器是虚拟机硬件逻辑模拟诸模块中最核心和最复杂的部分，超过 10,000 行。本章节将首先简单介绍 Intel 80386 CPU 的背景知识，然后描述虚拟 CPU 的功能设计、数据结构和接口函数。相关代码文件为 vmachine/vcpu.ch 和 vmachine/vcpuins.ch。

## 8.1 Intel 80386

本节介绍 Intel 80386 CPU 的背景知识。由于 CPU 结构复杂、功能多样，因此本节只从虚拟机软件的角度来介绍。

<TODO: i386 微架构>

### 8.1.1 CPU 的组成

#### 8.1.1.1 寄存器

8 个 32 位通用寄存器： EAX，ECX，EDX，EBX，ESP，EBP，ESI 和 EDI

1 个 32 位标志寄存器：EFLAGS

<TODO: 标志位>

1 个 32 位指令指针寄存器：EIP

段寄存器有三类。它们包含 16 位的程序可见部分（选择子）和 64 位程序不可见部分（段描述符）。对于 16 位情形，段基址等于(选择子 * 16)，对于 32 位情形，段基址从描述符中取得。

代码段寄存器：CS

堆栈段寄存器：SS

数据段寄存器：ES，DS，FS，GS

全局描述符表（GDT）寄存器：GDTR，是 48 位寄存器，存储 GDT 的基址和限长。

<TODO:GDTR>

中断描述符表（IDT）寄存器：IDTR，是 48 位寄存器，存储 IDT 的基址和限长。

<TODO:IDTR>

局部描述符表（LDT）寄存器：LDTR，包含 16 位 LDT 选择子和 64 位 LDT 描述符

任务寄存器：TR，包含 16 位任务状态段（TSS）选择子和 64 位 TSS 描述符。

#### 8.1.1.2 算术逻辑单元（ALU）

算术逻辑单元相关函数执行数值运算，如 ADD、SUB、CMP、MUL、DIV 等，并修改 EFLAGS 中相应的算术标志位：OF，SF，ZF，AF，CF，PF。

#### 8.1.1.3 内存管理单元（MMU）

内存管理单元负责管理逻辑地址、线性地址、物理地址的转换，以及数据的读取和写入。因为特权级的存在，执行内存访问前需要通过分段保护机制和分页保护机制。

#### 8.1.1.4 指令预取和解析

指令预取有助于减少内存访问次数以提高执行效率；但 NXVM 虚拟机不进行指令预取，以减少复杂性。

NXVM 虚拟机的指令执行函数 vcpuinsExecIns 读取操作码并进行指令分派，这一过程通过函数指针数组来完成。每个指令有各自的解析和执行函数。

## 8.1.2 指令解析

本节内容摘选自 IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference 的 Chapter 2: Instruction Format。

所有 IA-32 指令的编码都遵循下图所示的指令格式。指令包含：可选的指令前缀；1 或 2 个操作码字节；可能有 1 个 ModR/M 寻址字节，有时再加上 1 个 SIB 字节；可能有 1 个偏移量；可能有 1 个立即数。

<TODO:INSTR FORMAT FIGURE 2-1>

### 8.1.2.1 前缀码（Prefix）

指令前缀码的编码顺序可变。前缀码分为 4 组，每组内的指令前缀码只能出现一次，否则会造成不可预料的行为。

#### 8.1.2.1.1 锁和重复前缀

F0h 是 LOCK 前缀码，锁前缀。它保证在多处理器环境中，原子操作（atomic operation）能够独占共享内存区域。因为 NXVM 虚拟机是单处理器，LOCK 前缀没有实际作用。LOCK 前缀码的有效操作是：ADC，ADD，AND，BTC，BTR，BTS，DEC，INC，NEG，NOT，OR，SBB，SUB，XOR 和 XCHG。其中，XCHG 指令总是假设存在 LOCK 前缀，无论是否有 F0h 编码。

F2h 和 F3h 都是重复前缀，只对串操作指令有效，对其他指令可能引起不可预料的行为。有效操作是：MOVS，CMPS，SCAS，LODS，STOS，INS 和 OUTS。当执行一次串操作后 CX 或 ECX 非零，则根据 ZF 的值重复下一次操作。

其中，F2h 是 REPNE/REPNZ 前缀码。当执行一次串操作后，若 ZF 非零，则重复下一次操作，它用于 CMPS 和 SCAS 指令。

F3h 是 REP/REPE/REPZ 前缀码。当指令是 CMPS 和 SCAS 时，它表示 REPE/REPZ，当执行一次操作后 ZF 为零则重复下一次操作。对于其他串操作指令，它表示 REP，即重复下一次操作。

#### 8.1.2.1.2 段超越前缀

2Eh——CS

36h——SS

3Eh——DS

26h——ES

64h——FS

65h——GS

当使用段超越前缀后，默认的寻址数据访问段将变为段超越前缀指定的段。例如，MOV EAX, [SI+1234]的默认情形为 MOV EAX, DS:[SI+1234]。当使用 ES 前缀后，该指令被解析为 MOV EAX, ES:[SI+1234]。

#### 8.1.2.1.3 操作数长度超越前缀

66h 是操作数长度超越前缀。它允许软件选择 16 位或 32 位操作数。当 CPU 的 CS 段为 16 位时，默认操作数长度是 16 位；CS 段为 32 位时，默认操作数长度为 32 位。使用该前缀后，操作数长度变为非默认值。

#### 8.1.2.1.4 寻址长度超越前缀

67h 是寻址长度超越前缀。它允许软件选择 16 位或 32 位寻址方式。当 CPU 的 CS 段为 16 位时，默认寻址长度是 16 位；CS 段为 32 位时，默认寻址长度为 32 位。使用该前缀后，

寻址长度变为非默认值。

#### 8.1.2.2 操作码（Opcode)

操作码指定了 CPU 要执行的指令及其解码方式。大多数操作码为 1 个字节；0Fh 前缀的操作码为 2 个字节。对于某些操作码而言，ModR/M 中的 Reg 域也决定了具体操作。NXVM 虚拟机的 CPU 指令模拟代码中，每个操作码都由一个对应的函数来解析和执行。

#### 8.1.2.3 ModR/M 和 SIB

大多数对内存执行访问的指令都有 ModR/M 寻址字节，该字节紧随操作码之后。ModR/M 字节包含 3 个域，如图<TODO: FIGURE ABOVE>所示：

mod 域和 r/m 域一起组成了 32 种不同的组合：24 种内存地址组合（mod 为 0，1，2）和 8 种寄存器组合（mod 为 3）。

r/m 域选择 8 种寄存器组合中的某一个，或者 24 种内存地址组合中的某一个。

reg 域可能选择 8 种寄存器中某个，或者指定某个操作码的 8 种可能指令中的某一个（例如，当操作码为 80，ModR/M 的 reg 域为 0，则表示 ADD 指令，若 reg 为 1，则表示 OR 指令等）。

16 位寻址时，没有 SIB 字节。32 位寻址时，如果内存地址形式包含 base+index，或者是 scale+index，就存在 SIB 字节。SIB 字节也有 3 个域，如图<TODO: FIGURE ABOVE>所示：

scale 域定义了比例因子，0x00 代表 1 倍，0x01 代表 2 倍，0x02 代表 4 倍，0x03 代表 8 倍。

index 域定义了 index 所用的寄存器，例如 0x00 代表 EAX。但是，0x04 不代表 ESP，ESP 不能作为 index。

base 域定义了基址所用的寄存器。但是当 base 为 0x05 且 ModR/M 的 mod 域为 0x00 时，base 为空，而不是 EBP。

例如，在 32 位寻址模式下采用 SIB 时（r/m = 0x04, mod != 0x03），当 base = 0x02，index = 0x03，scale = 0x01 时，所代表的寻址方式便是[EDX+EBX*2]。

ModR/M 和 SIB 的所有寻址方法可以通过查表来得到，参阅 8.1.2.5 和 8.1.2.6。

#### 8.1.2.4 偏移量和立即数

偏移量（Displacement）是 ModR/M 和 SIB 寻址的一部分，一般紧随 ModR/M 字节之后。如果存在 SIB 字节，则紧随 SIB 字节之后。在指令编码中，它可以是 1、2、4 个字节长度的数值。偏移量在寻址中的作用可以参阅 8.1.2.5 和 8.1.2.6。

立即数在指令编码中也是 1、2、4 个字节长度的数值，但它是指令的操作数。立即数紧随偏移量之后。

#### 8.1.2.5 16 位寻址

16 位寻址方式根据 ModR/M 查表可得。

ModR/M 字节可以同时表示 2 个操作数，一个是寄存器，由 reg 域决定；另一个是内存地址或寄存器，由 mod 域和 r/m 域同时决定。具体细节可以参阅下图，以及 vcpuins.ch 中的 _kdf_modrm 函数。

这两个操作数（寄存器-寄存器，或内存地址-寄存器）的长度由操作码和 CPU 代码段的默认长度属性决定。例如，有的操作码的操作数只能是 8 位长度，而另一些操作码的操作数长度是 16 位或 32 位。

<TODO: FIGURE>

#### 8.1.2.6 32 位寻址

32 位寻址方式根据 ModR/M 和 SIB 查表可得。

<TODO: FIGURE>

### 8.1.3 内存管理

本节部分内容摘选自 IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide 的 Chapter 3: Protected-Mode Memory Management。

#### 8.1.3.1 实地址模式和虚拟 86 模式

实地址模式和虚拟 86 模式下，内存访问采用分段机制。段的基址可以由选择子直接计算得到，即 16 位选择子的值左移 4 位（即乘以 16）。段基址加偏移量后形成了一个可以直接访问的物理地址。每个段的限长为 FFFFh，即 64KB。数据段、堆栈段和代码段中的内容都是可读写的。当加载段寄存器时，段基址会被更新为 16 位选择子的 16 倍。

#### 8.1.3.2 保护模式

通过将 CPU 的 CR0 寄存器的 PE 标志置位可以开启保护模式。

保护模式下，内存地址变得复杂了，不仅分段机制发生了改变，还出现了分页机制。由此产生了逻辑地址、线性地址和物理地址的区别。

我们先来了解一下逻辑地址、线性地址和物理地址的概念。

逻辑地址就是段：偏移量，例如，0123:456789ab。实模式和虚拟 86 模式下，段的基地址就是(0x0124 << 4)，该地址对应的线性地址是基地址+偏移量，即(0x0123 << 4) + 0x456789ab。而保护模式下，0123 是一个选择子。这个选择子相当于一个索引值，可以在全局描述符表或局部描述符表中查询到一个 64 位长度的描述符。描述符里面包含了段的基址、限长和访问权限等属性。因此，此时该逻辑地址对应的线性地址是：选择子——查询——描述符，描述符——解析——基地址，基地址+偏移量。假设 0123 对应的描述符里面的基地址是 0x10000000，那么线性地址就是 0x556789ab。

线性地址就是段基址+偏移量。在实模式和虚拟 86 模式下，它就是物理地址。在保护模式下，当内存不分页，它也等于物理地址；内存分页时，线性地址被解析成 3 个部分，然后再经过转换形成物理地址，参阅 8.1.3.2.1。在保护模式下，线性地址最大可以是 FFFFFFFFh，即 4GB。

物理地址就是物理内存中的绝对地址，它的大小不应当超过物理内存的大小。在虚拟机源代码中动态分配的内存空间就是虚拟机的物理内存。物理地址称为客户地址。参阅内存模块一章。

线性地址是可能大于物理地址的，但经过分页机制的转换，线性地址会指向物理内存中的某个位置，而不会超出，除非页表有错误。

<TODO: FIGURE:SELECTOR, SEG DESCRIPTOR, MEMORY MANAGEMENT>

##### 8.1.3.2.1 分段机制

不论处于保护模式还是实模式、虚拟 86 模式，分段机制始终存在。

保护模式下，当加载段寄存器时，CPU 将解析选择子的数值，选择全局描述符表或者局部描述符表，然后根据选择子的索引值查表得到一个段描述符。段描述符包含了段基址、限长、访问权限和其他属性。16 位的选择子被加载到段寄存器的软件可见部分，64 位的描述符被加载到段寄存器的软件不可见部分。

选择子中的特权级称为 Requestor's Privilege Level（RPL），描述符中的特权级称为 Descriptor Privilege Level（DPL），当前特权级称为 Current Privilege Level（CPL）。其中，CPL 就是 CPU 的 CS 寄存器不可见部分的描述符的 DPL。

当加载段寄存器时，CPU 将检查 CPL、DPL 和 RPL，满足权限保护的要求。CPU 还检查段描述符的属性，确保 CS 段寄存器加载的段是可执行段，SS 段寄存器加载的是可写的数据段，ES、DS、FS、GS 段寄存器加载的是数据段或者可读的可执行段。可执行段都是不可写的；数据段都是可读的。

在保护模式下，段的最大限长可以是 FFFFFFFFh，即 4GB。当访问的地址超过数据段的限长，将产生常规保护错误#GP；当访问的地址超过堆栈段的限长，将产生堆栈错误#SS。也就是说，假如出现跨段访问，CPU 将产生异常，所有的跨段访问都是非法的。

只有在保护模式下加载段寄存器，才能改变段的限长、属性等。实模式和虚拟 86 模式下加载段寄存器只能改变段基址。这一特性很有意思，例如，进入保护模式将段限长改为 FFFFFFFFh 再返回实模式后，就可以直接进行 32 位寻址，访问地址高于 1MB 的内存数据了。

分段机制的代码实现可参阅 vcpuins.c 的_kma_linear_logical 函数。

#### 8.1.3.2.2 分页机制

当关闭分页模式时，线性地址就是物理地址。在保护模式下，通过将 CPU 的 CR0 寄存器的 PG 标志置位，可以开启分页模式。

分页模式下，1 个内存页的大小是 4KB。我们需要使用表格来保存每个内存页的物理地址供访问。由于最大线性地址是 4GB，假如用 1 个页表来表示所有的页面，那么这个表中将包含 1M 个项，用于指向各自的页：4KB * 1M = 4GB。每个页表项就是对应页面的物理地址，而物理地址的长度是 32 位，4 个字节，因此整个页表的大小将是 4B * 1M = 4MB。

用 4MB 来保存所有的页面显得过于昂贵，因此 Intel 的设计者们采用了二级页表的方法，节省内存空间：1 个页目录表，包含了 1K 个页表的物理地址；1K 个页表，每个页表包含了 1K 个内存页的物理地址。页目录表总是在内存中；但这 1K 个页表则并不全部载入内存，仅当需要的时候才从硬盘读取，就像普通的内存页一样。页目录表自身的物理地址被保存在页目录表寄存器（Page Directory Base Register，PDBR）中，也就是 CPU 的 CR3 寄存器。

分页模式下，线性地址被分成如下 3 个部分：

directory 域，用于在页目录表（Page Directory Table，PDT）中找到页表（Page Table，PT）的入口（Page Directory Entry，PDE），该入口保存了页表在物理内存中的地址。这是第一级查询。

table 域，用于在页表（PT）中找到内存页在物理内存中的起始地址（PTE）。这是第二级查询。

offset 域，表示要访问的数据在内存页里的偏移量。

<TODO: FIGURE 3-13>

80386 的分页机制采用了二级页表，虽然复杂，但节省了内存空间。

PDE 和 PTE 都是 32 位长度的数据结构，如下图所示：

<TODO: PDE/PTE 3-14>

假如 PDE 指示的页表不存在于内存中，或 PTE 指示的内存页不在内存中，就产生页错误#PF。客户操作系统的页错误异常处理程序将会把相应的内存页从其他地方（例如磁盘）加载到内存中，把页存在标志（P 位）置位，然后重新执行刚才引起错误的指令。

分页机制的代码实现可参阅 vcpuins.c 的_kma_physical_linear 函数。

#### 8.1.3.2.3 跨页访问

由于内存页的大小是 4KB，假如要访问的内存单元在页的末边界上，例如访问 00000FFEh 上的 4 个字节，将出现跨页访问。因此，每当进行内存访问，需要预先检测访问的地址是否在页边界上，访问的数据长度是否会超出页边界。

在 Bochs 虚拟机和 NXVM 虚拟机中，假如内存访问超过页边界，则将数据分成两部分来访问。在上面的例子中，虚拟 CPU 会从 00000000h 内存页中读取该页的最后两个字节（线

性地址是 00000FFEh 和 00000FFFh），然后从 00001000h 内存页（下一个内存页）中读取该页的开头两个字节（线性地址是 00001000h 和 00001001h）。

NXVM 虚拟机的这一特性对其虚拟 CPU 的所有内存访问都适用，不论是实模式、虚拟 86 模式还是保护模式，不论是否开启分页，皆如此。该特性的实现可参考_kma_read_linear 函数和_kma_write_linear 函数。

### 8.1.4 中断处理

CPU 的中断处理分为两大类，软件中断和硬件中断。软件中断指 CPU 执行 INT 指令或发生异常，是不会被屏蔽的；硬件中断分为可屏蔽中断和不可屏蔽中断（NMI），通过 CPU 的 INTR 和 NMI 引脚输入信号。EFLAGS 寄存器的 IF 标志位控制着可屏蔽中断被屏蔽或被响应。假如硬件中断被响应，则向虚拟机中断控制器发送响应信号，使得当前中断从中断控制器的“中断请求寄存器（IRR）”移除，加入到“中断服务寄存器（ISR）”。可屏蔽中断的处理程序的结束处应当向中断控制器发送中断结束命令（EOI）表明中断结束。关于中断控制器（PIC）请参阅相关章节。

在 x86 系统中，一共有 256 个中断号，每个中断号对应一个中断处理程序。在实模式下，中断处理程序的入口放在物理内存的 0000:0000 到 0000:03FF。每个中断处理程序的入口有 4 个字节，低 2 字节是偏移量，高 2 字节是段选择子值，这 4 个字节称为中断向量，中断向量所在的内存空间称为中断向量表。在保护模式和虚拟 86 模式下，中断处理程序的入口为 64 位的“中断门描述符”，存放在中断描述符表（IDT）中。中断描述符表的起始地址和限长存放在 IDTR 寄存器中。

关于 Intel 8088 中断处理的知识请参考微机原理教材。

## 8.2 数据结构

本节分别描述 CPU 的数据结构和 CPU 指令执行模块的数据结构。

### 8.2.1 CPU 数据结构

该模块定义的全局变量是 t_cpu vcpu。有若干个子结构：

```
typedef enum {
    SREG_DATA,
    SREG_STACK,
    SREG_CODE,
    SREG_LDTR,
    SREG_TR,
    SREG_GDTR,
    SREG_IDTR
} t_cpu_sreg_type;
```

结构体 t_cpu_sreg 定义了段寄存器的枚举类型。在保护模式分段机制函数_kma_linear_logical 和_ksa_load_sreg 中，会对段寄存器的类型进行判断，然后进行相应的处理，比如检查特权级、检查段描述符属性和加载相应寄存器等。

```
typedef struct {
    t_bool flagwrite;
    t_nubit32 byte;
    t_nubit32 linear;
    t_nubit64 data;
```

```
} t_cpu_memory;
```

结构体t_cpu_memory定义了一次内存访问记录。flagwrite表明该内存方式是读还是写；byte 表明数据的长度；linear 表明数据的线性地址；data 为所读取或写入的数据。

```
typedef struct {
    t_bool flagvalid;
    t_nubit16 selector;
    /* invisible portion/descriptor part */
    t_cpu_sreg_type sregtype;
    t_nubit32 base;
    t_nubit32 limit;
    t_nubit4    dpl; /* if segment is cs, this is cpl */
    union {
        struct {
            t_bool executable;
            t_bool accessed;
            union {
                struct {
                    t_bool defsize; /* 16-bit (0) or 32-bit (1) */
                    t_bool conform;
                    t_bool readable;
                } exec;
                struct {
                    t_bool big;
                    t_bool expdown;
                    t_bool writable;
                } data;
            };
        } seg;
        struct {
            t_nubit4 type;
        } sys;
    };
} t_cpu_sreg;
```

结构体 t_cpu_sreg 定义了段寄存器。该结构体已经将段描述符的属性解析拆分成独立的属性变量。flagvalid 表明在保护模式下段寄存器是否有效（例如，数据段是空描述符时为无效）；selector 即 16 位选择子；base 即段基址；limit 即段限长；dpl 为段描述符的 DPL，当段寄存器是 CS 时，该值就是 CPL。结构体中含有一个 union 子结构，包括 seg 子结构和 sys 子结构，用于存储段的属性。使用哪个子结构是通过 sregtype 来区分的。当段寄存器是代码段、堆栈段、数据段时，使用 seg 子结构；否则使用 sys 子结构。seg 子结构中又用 union 分为 exec 和 data 子结构，供可执行段和数据段使用，通过 executable 来区分。sys 子结构包含了 type，用于确定具体系统段描述符的属性。

```
typedef struct {
    /* general registers */
    union {
```

```
        union {
            struct {t_nubit8 al,ah;};
            t_nubit16 ax;
        };
        t_nubit32 eax;
    };
    union {
        union {
            struct {t_nubit8 bl,bh;};
            t_nubit16 bx;
        };
        t_nubit32 ebx;
    };
    union {
        union {
            struct {t_nubit8 cl,ch;};
            t_nubit16 cx;
        };
        t_nubit32 ecx;
    };
    union {
        union {
            struct {t_nubit8 dl,dh;};
            t_nubit16 dx;
        };
        t_nubit32 edx;
    };
    union {
        t_nubit16 sp;
        t_nubit32 esp;
    };
    union {
        t_nubit16 bp;
        t_nubit32 ebp;
    };
    union {
        t_nubit16 si;
        t_nubit32 esi;
    };
    union {
        t_nubit16 di;
        t_nubit32 edi;
    };
    union {
```

```
        t_nubit16 ip;
        t_nubit32 eip;
    };
    union {
        t_nubit16 flags;
        t_nubit32 eflags;
    };
    /* segment registers */
    t_cpu_sreg es, cs, ss, ds, fs, gs;
    t_cpu_sreg ldtr, tr, gdtr, idtr;
    /* control registers */
    t_nubit32 cr0, cr1, cr2, cr3, cr4, cr5, cr6, cr7;
    t_nubit32 dr0, dr1, dr2, dr3, dr4, dr5, dr6, dr7;
    t_nubit32 tr0, tr1, tr2, tr3, tr4, tr5, tr6, tr7;
    /* control flags */
    t_bool flagmasknmi, flagnmi, flaghalt;

    /* cpu recorder */
    t_bool flagignore;
    t_cpu_memory mem[0x20];
    t_nubit8 msize;
    t_nubit8 oplen;
    t_nubit8 opcodes[15];
    t_nubit8 svcextl;
    t_nubit16 reccs;
    t_nubit32 receip;
    t_nubit32 linear;
    t_string stmt;
} t_cpu;
```

结构体 t_cpu 中利用 union 结构体定义了 8 个 32 位通用寄存器、指令指针寄存器、标志寄存器，以及 8 个控制寄存器，8 个调试寄存器和 8 个测试寄存器。

flagmasknmi 控制 NMI 不可屏蔽中断是否允许；flagnmi 表明当前外部是否传入一个 NMI 信号；flaghalt 表明当前 CPU 是否在正常执行指令或仅仅是等待中断的出现来激活 CPU。

其他数据成员用于 CPU 状态记录：flagignore 表示当前指令不用于同步比较调试；mem 数组存储最多 32 个内存访问记录；msize 表明有多少次内存访问；oplen 表明当前执行的指令长度；opcodes 数组存储了当前执行的指令的机器码；svcextl 用于标记 BIOS 中断处理程序的调用深度；reccs、receip 和 linear 记录了当前指令执行前的段选择子、指令指针和线性地址；stmt 记录了当前执行的指令的反汇编源码。

### 8.2.2 CPU 指令执行模块的数据结构

该模块定义的变量是 t_cpuins vcpuins，结构定义如下：

```
typedef struct {
    /* prefixes */
    t_cpuins_prefix_rep    prefix_rep;
    t_cpuins_prefix        prefix_oprsize;
```

```
    t_cpuins_prefix         prefix_addrsize;
    t_cpu_sreg *roverds, *roverss, *rmovsreg;

    /* execution control */
    t_cpu oldcpu;
    t_bool flaginsloop;
    t_bool flagmaskint; /* if int is disabled once */

    /* memory management */
    t_cpuins_logical mrm;
    t_vaddrcc rrm, rr;
    t_nubit64 crm, cr, cimm;
    t_bool flagmem; /* if rm is in memory */
    t_bool flaglock;

    /* arithmetic operands */
    t_nubit64 opr1, opr2, result;
    t_nubit32 bit;
    t_cpuins_arithtype type;
    t_nubit32 udf; /* undefined eflags bits */

    /* exception handler */
    t_nubit32 except, excode;

    /* debugger */
    t_bool flagwr, flagww, flagwe;
    t_nubit32 wrlin, wwlin, welin;
} t_cpuins;
```

该结构体用于记录在指令解码和执行中所用到的所有属性和变量。

第一部分是前缀属性：prefix_rep 表示重复前缀的类型，可能是不存在，或者是 REP/REPE，或者是 REPNE；prefix_oprsize 表示是否存在操作数长度超越前缀，prefix_addrsize 表示是否存在寻址长度超越前缀；roverds、roverss 指针分别指向当前选定的访问数据段，假如不存在段超越前缀，则默认访问段为 DS、SS，否则均指向段超越前缀所指定的段寄存器；rmovsreg 指针指向 MOV 指令所要访问的段寄存器。

第二部分是内存寻址数据：当 ModR/M 的 mod 域小于 3 时，flagmem 置位，表示此时 mod 和 r/m 域指示访问内存，并且 mrm 为进行内存寻址计算后得到的逻辑地址；当 ModR/M 的 mod 域为 3 时，flagmem 置零，表示此时 mod 和 r/m 域指示访问寄存器，并且 rrm 为 mod 域和 r/m 域所指示的寄存器的引用地址；rr 表示 ModR/M 中由 reg 域所指示的寄存器的引用地址；crm 用于读取或者写入 mrm 或 rrm 所指向的地址的值，相关函数是_m_read_rm 和 _m_write_rm；cr 用于读取 rr 所指向的地址的值；cimm 表示指令的机器码中的立即数；flaglock 表示当前指令是否存在 LOCK 前缀。

第三部分是算术和逻辑计算数据：opr1、opr2 表示操作数；result 表示计算结果；bit 表明操作数的位数，8、16、32 表明两个操作数都是 8 位、16 位或 32 位，12 表明操作数分别

是 16 位和 8 位，20 表明两个操作数分别是 32 位和 8 位；type 表明当前的计算指令；udf 表明执行计算指令后，EFLAGS 寄存器中的未定义标志。

第四部分是异常处理数据：except 有 32 位长度，每 1 位表示发生了 1 个异常，每位的含义可参阅 vcpuins.h 中的宏定义（如 VCPUINS_EXCEPT_GP）；excode 表示保护模式下发生异常时的错误码。

第五部分是调试数据：flagwr，flagww 和 flagwe 分别表示是否监视内存读取、内存写入和指令执行；wrlin，wwlin 和 welin 分别表示要监视的读取的内存地址、写入的内存地址和执行的内存地址。

## 8.3 接口函数

本节描述了 CPU 模块和 CPU 指令执行模块的接口函数。

### 8.3.1 CPU 接口函数

虚拟机 CPU 提供了如下接口函数：

void vcpuInit();

虚拟机程序启动初始化函数，负责调用 vcpuinsInit

void vcpuReset();

虚拟机硬件复位函数，重置 vcpu 各寄存器、各标志的值，清空 CPU 运行记录

void vcpuRefresh();

虚拟机运行函数，负责调用 vcpuinsRefresh 处理指令

void vcpuFinal();

虚拟机程序退出函数，空

### 8.3.2 CPU 指令执行接口函数

虚拟机 CPU 指令执行模块提供了如下接口函数：

t_bool vcpuinsLoadSreg(t_cpu_sreg *rsreg, t_nubit16 selector);

加载段选择子到指定的段寄存器，供调试器调用

t_bool vcpuinsReadLinear(t_nubit32 linear, t_vaddrcc rdata, t_nubit8 byte);

读取线性地址中的数据，供调试器调用

t_bool vcpuinsWriteLinear(t_nubit32 linear, t_vaddrcc rdata, t_nubit8 byte);

将数据写入线性地址，供调试器调用

void vcpuinsInit();

虚拟机程序启动初始化函数，负责注册具体指令函数。即：设置指令函数指针表，普通操作码和 0Fh 操作码各 256 个指针；未定义的操作码均指向函数 UndefinedOpcode；

void vcpuinsReset();

虚拟机硬件复位函数，将 svcextl 外部中断处理程序调用深度计数器清零。

void vcpuinsRefresh();

虚拟机执行函数，调用 ExecIns 函数处理具体指令，参阅 8.3.4

void vcpuinsFinal();

虚拟机程序退出函数，空

## 8.4 运行流程

本节描述虚拟机 CPU 的指令执行流程，有助于理解源代码的逻辑结构。

<TODO: FIGURE>

1) vmachine.c: vmacineRefresh()
   虚拟机核心执行函数
2) vcpu.c: vcpuRefresh()
   CPU 执行函数，负责调用 vcpuinsRefresh
3) vcpuins.c: vcpuinsRefresh()
   假如 CPU 处于 HALT 状态（先前执行过 HALT 指令且未被硬件中断激活），就 Sleep；否则执行 ExecIns
4) vcpuins.c: ExecIns()
   该函数是主控函数，负责执行一条指令，以下分点解释：
   a) ExecInit()
      初始化执行环境，例如，将 roverds 和 roverss 指向 DS 和 SS 段寄存器，清空上一次的计算结果等
   b) While Loop
      该循环用于处理指令前缀和执行指令的主体。
      i. _s_read_cs()
         读取操作码，指令指针加 1
      ii. ExecFun(table[opcode])
         执行操作码对应的具体指令函数，指令指针移向该操作码对应的指令之后
      iii. If (opcode == prefix), goto (i); else goto (c).
         假如是前缀码，就继续解码该指令；假如当前执行的不是前缀码，说明整条指令都已执行完毕。
   c) Check for watch point of execution
      假如当前执行的指令符合监视点，就触发断点：打印当前状态并停止虚拟机。
   d) ExecFinal()
      该函数负责结束指令的执行。如果执行了串操作指令、并且该串操作指令还会被再次执行（例如，存在 REP 前缀且 CX 不等于 0），便会将 CPU 的指令指针回溯到该指令执行前的位置，供下次 ExecIns 执行。假如指令执行时发生了异常，ExecFinal 还负责调用异常处理程序_e_except，并打印异常发生时的状态。
5) vcpuins.c: ExecInt()
   该函数负责检测硬件中断。
   a) If (NMI)，存在不可屏蔽中断，则调用 Int 02h
   b) If (EFLAGS_IF)，调用 vpicScanINTR()，查看是否存在可屏蔽中断。如果不是，跳到(c)；否则跳到(i)
      i. vpicGetINTR()从中断控制器获取中断号
      ii. 调用中断执行函数_e_intr_n()
   c) If(ELFAGS_TF)，存在单步调试陷阱，则调用 Int 01h

## 8.5 代码实现

### 8.5.1 CPU 相关的宏定义

虚拟机 CPU 定义了很多宏变量和宏函数，用于定义、读取和修改 CPU 的诸多数据结构：EFLAGS、CR0、CR3 寄存器的各标志位，选择子和描述符。此外，还有宏函数用于获取当前状态，如保护模式、实模式、虚拟 86 模式，当前特权级等。

虚拟机 CPU 指令执行模块中定义了各异常标志位和设置异常的宏函数，以及检测当前实际操作数长度、当前实际寻址长度、当前堆栈地址长度（由 CPU 的 SS 寄存器中段描述符的 D/B 位决定）的宏。

### 8.5.2 CPU 指令执行模块内部函数

虚拟机 CPU 指令执行模块内部的函数分为如下组别。

#### 8.5.2.1 内存访问函数

核心内存访问函数的命名以_kma_开始，包括对数据的引用、物理地址、线性地址和逻辑地址之间进行相互转换、读取和写入。这类函数实现了分段访问机制、分页访问机制和对物理内存的读写。核心内存访问还允许进行超越特权级限制的数据访问（当 force 置位时），供指令读取等 CPU 核心功能调用。核心内存访问函数只能被普通内存访问函数、主控函数、核心解码函数和抽象执行函数调用，任何其他函数都不应当调用核心内存访问函数。

普通内存访问的命名以_m_开头，是核心内存访问函数的封装（Wrapper），提供了常规的内存访问功能。它们在调用核心内存访问函数时，将 force 置零，并传入 CPL 的值作为访问的特权级。所有其他普通函数和具体指令函数都只能调用普通内存访问函数，受到特权级 CPL、DPL 和 RPL 的限制，并且必须以逻辑地址的形式访问虚拟机的物理内存。假如不符合特权级要求，将产生异常。

#### 8.5.2.2 段式访问函数

核心段式访问函数的命名以_ksa_开始，包括加载段寄存器、读写各描述符表。这类函数供普通段式访问函数、核心解码函数和抽象执行函数调用，任何其他函数都不应当调用核心内存访问函数。

普通段式访问函数的命名以_s_开始，是核心段式访问函数或核心内存访问函数的封装，提供了常规的逻辑地址访问功能。

#### 8.5.2.3 端口访问函数

核心端口访问函数的命名以_kpa_开始，用于保护模式下的 I/O 权限检查和 I/O 位图检查，供普通端口访问函数调用。

普通端口访问函数的命名以_p_开始，用于端口的 I/O 操作。它们会通过 vport 中的函数指针数组和端口号，执行相应的 I/O 端口函数。普通端口访问函数供抽象执行函数或具体指令函数调用。

#### 8.5.2.4 解码函数

核心解码函数的命名以_kdf_开始，用于检查前缀码、读取机器码、解析 ModR/M 和 SIB 寻址字节，供普通解码函数使用。

普通解码函数的命名以_d_开始，是核心解码函数的封装，除了核心解码函数提供的功能外，还针对具体指令函数所需的解码功能进行了优化。普通解码函数用于读取前缀码、读取操作码、解析 ModR/M 和 SIB、读取偏移量、读取立即数。

#### 8.5.2.5 抽象执行函数

核心抽象执行函数的命名以_kec_开始，实现了堆栈操作、任务切换、跳转、调用、中断处理等涉及 CPU 指令执行流程的功能，供分支抽象执行函数和普通抽象执行函数调用。

分支抽象执行函数的命名以_ser_开始，是普通抽象执行函数的分支延伸，仅供普通抽象执行函数调用。

普通抽象执行函数的命名以_e_开始，提供堆栈操作、跳转、调用、中断处理的功能，供具体指令函数调用。

#### 8.5.2.6 抽象算术逻辑函数

抽象算术逻辑函数的命名以_kaf_或_a_开始，提供了算术计算、逻辑计算和标志位计算等核心运算功能，供具体指令函数调用。

#### 8.5.2.7 具体指令函数

具体指令函数为数众多，每一个函数代表了一条可解析的指令，其函数命名相似于对应的汇编指令。具体指令函数通过调用普通解码函数来完成解码功能，调用抽象执行函数或抽象算术逻辑函数来执行指令功能。

#### 8.5.2.8 主控函数

主控函数为 ExecInit，ExecFinal，ExecIns 和 ExecInt。

ExecInit 将当前指令数据初始化，ExecFinal 则检查是否出现异常。

ExecIns 首先调用 ExecInit 初始化，然后读取操作码，并解码执行。假如该操作码是前缀码，就再次读取、解码和执行下一个操作码。解决所有前缀码后，就处理指令的主体部分，然后调用 ExecFinal 结束指令执行。

ExecInt 负责处理硬件中断和单步调试。它依次处理不可屏蔽中断、可屏蔽中断和单步调试标志。假如存在要处理的中断，它会在调用_e_int 前使用 ExecInit 初始化执行环境，在_e_int 执行之后使用 ExecFinal 检查异常，就如同调用了 INT 指令一样。这保持了代码和运行流程的一致性。

## 8.6 Quick and Dirty：虚拟机特殊指令*

NXVM 虚拟机为了方便编写和调试，尽快满足完整性的设计要求，将某些本该由汇编编写的 BIOS 中断处理程序改用 C 语言编写，这样这些 BIOS 中断处理程序只需要调用这样一条特殊指令就可以达到设计要求。

NXVM 设计的特殊指令的助记符是 QDX，指令格式是 QDX imm8，操作码是 F1h。实际上，F1h 操作码是 Intel 保留的操作码，但在 IA-32 手册中没有描述，所以对于 i386 模拟器而言，可以安全地使用。

在客户操作系统中的软件假如存在 QDX 指令，便可实现特殊功能。

以下表格描述了 QDX 指令功能：

| | |
|---|---|
| QDX 00 | 调用 vmachineStop 停止虚拟机，相当于 Bochs 的 Magic Break |
| QDX 01 | 调用 vmachineReset 复位虚拟机 |
| QDX 02 | 供 BIOS 中断处理程序进入时调用，计算调用深度 |
| QDX 03 | 供 BIOS 中断处理程序退出时调用，计算调用深度 |
| QDX 10 | BIOS 中断 INT 10h，显示功能，见 qdcga.c |
| QDX A2 | BIOS 中断 INT 13h 的 2 号子功能，读硬盘扇区，见 qddisk.c |

| | |
|---|---|
| QDX A3 | BIOS 中断 INT 13h 的 3 号子功能，写硬盘扇区，见 qddisk.c |
| QDX 09 | 硬件中断 INT 09h，处理键盘中断，见 qdkeyb.c |
| QDX 16 | BIOS 中断 INT 16h，键盘功能，见 qdkeyb.c |

其中 QDX 00，QDX 01，QDX 02，QDX 03 的代码实现在 vcpuins.c 的 QDX 函数中，用户可以修改 vcpuins.c 的 QDX 函数，实现更多的功能。

除此之外的所有其他 QDX 特殊功能均由 vmachine/bios 部分实现。

在虚拟机源代码文件夹中，特殊指令的例子是 stop.com 和 reset.com。将这 2 个文件放在磁盘镜像中载入虚拟机后，在客户操作系统 MS-DOS 提示符中运行它们，就能执行相应的操作。

Stop.com 的源代码：

QDX 00

INT 20

Reset.com 的源代码：

QDX 01

INT 20