用 32 位 x86 汇编设计 8051 模拟器

同济大学 电子信息工程 黄轶纯

关键字：80x86，8051，32 位汇编，模拟器，程序设计

1.0 引言

Intel 80x86 是目前最普遍采用的桌面系统的中央处理器。自 80386 开始，Intel 开始引入了 32 位系统架构。随之而来的便是程序设计上的变革。32 位结构除了带来扩展了的寄存器以外，还给程序员带来了曾经只有大型机才具备的特性：保护模式，线性地址空间（Flat memory model）。后者为 x86 上的汇编语言设计带来了极大的便利，从此程序员可以告别繁琐的断式结构，而采用寻址空间可达 4GB 的线性地址结构，换句话说，所有的数据都位于同一段内。此外，Windows 操作系统的诞生，同样给 x86 上汇编语言设计带来了新的契机。Windows 提供了大约 2000 多个 Win32 API，这些 API 都可以在汇编程序内直接 CALL，这给编程效率带来了很大的提高。汇编编译器方面也有很多选择，最著名的是 MASM32，这是一个在 MASM6.14 的基础上开发的软件，它具有大量对 Win32 API 的引用文件（Include, Lib）和宏定义，可以用 invoke 调用 API，使用起来非常方便。本文中便使用 MASM32 进行开发。

8051 是 Intel 微处理器的又一个分支，它被广泛地应用在控制领域。8051 属于 8 位 CPU，具有价格低、实时性好、编程容易等优点。在大学中的电子信息专业都广泛开展了相关课程。然而，由于学生数量增多以及硬件设备数量有限的原因，在学习 8051 的过程中，学生们往往缺少上机编程实践的经验。而在 x86 上实现 8051 模拟，使大家在自己的 PC 上就可以联系 8051 的编程练习，就可以一定程度上缓解这个矛盾。此外，一个好的模拟器也被工程人员用来调试算法等。

2.0 具体实现

设计模拟器时，我把主要工作分成两个部分。一个是 8051 模拟算法的核心程序，另一个则是一个界面友好的调试器外壳。前者使用 32 位汇编设计，后者使用高级语言完成。本文只介绍前者，也就是模拟器核心程序。

2.1 总体设计

为了在高级语言中可以方便地使用核心程序，我把它设计成一个 DLL 文件，对外调用它的程序提供相应 API 接口，主要的接口有： SetACC（设置累加器）、GetRegValue（获取寄存器值）、 SetRegValue（设置寄存器值）、SetPC（设置 PC）、GetPC（获取 PC）、 FSRun（全速运行）、StepRun（单步运行）、GetDataMem（获取数据存储器值）、SetDataMem（设置数据存储器值）、GetCodeMem（获取指令存储器值）、SetCodeMem（设置指令存储器值）、 LoadCodeMem 、 SaveCodeMem 、 LoadDataMem 、 SaveDataMem（以上 4 个 API 为装入和保存存储器影像到磁盘）。

2.2 存储器模拟

2.2.1 内存寻址

内存寻址的基本原理是在 PC 的内存中建立若干连续区域，用来存放数据存储器和指令存储器的内容，寻址时采用基址加变址方式，基址为该区域的首地址，变址为模拟的 8051 的寻址地址。具体算法参照以下给出 GetCodeMem 和 SetDataMem 的代码：

```
DataMem         db   DataMemSize dup(?)      ;建立数据存储器空间
CodeMem         db   CodeMemSize dup(?)      ;建立指令存储器空间

GetCodeMem proc Address:DWORD
    push    esi
    lea     esi,CodeMem             ;取 CodeMem 首地  基址
    add     esi,Address             ;加上 8051 指令存储器地址  变址
    mov     al,[esi]                ;返回数据
    pop     esi
    ret
GetCodeMem endp

SetDataMem proc Address:BYTE,Data:BYTE
    push    esi
    push    edi
    mov     al,Data
    lea     esi,DataMem             ;取 CodeMem 首地  基址
    movzx   edi,Address             ;取 8051 数据存储器地址  变址，低 8 位有效
```

```
    mov     [esi+edi],al                ;存入数据
    pop     edi
    pop     esi
    ret
SetDataMem endp
```

2.2.2 寄存器寻址

由于 8051 的寄存器都位于数据存储器的某些区域，所以，寄存器寻址的算法基本和内存寻址相似。主要过程为从 PSW 获取寄存器区，然后和寄存器号（RegNo 参数）一起得出该寄存器在内存中的地址。以下是 SetRegVal 的代码：

```
ADDR_PSW  equ 208

SetRegValue     proc        RegNo:BYTE,Data:BYTE
        push    esi
        lea     esi,DataMem             ;取 DataMem 首址
        movzx   eax,byte ptr [esi+ADDR_PSW] ;取 PSW 的值到
        and     al,11000B               ;仅保留 PSW 的位 4 和位 3，其余位清零
        add     al,RegNo                ;加上寄存器号，得该寄存器的绝对地址
        add     esi,eax                 ;加上 DataMem 首址
        mov     al,Data
        mov     [esi],al
        pop     esi
        ret
SetRegValue endp
```

2.2.3 位寻址

位寻址是 8051 具有而 x86 不具有的一种寻址方式。因此在实现时需要一定的转换。为此我设计了一个宏，该宏的作用是根据 8051 的内存规划将位地址转换成字节地址，结果保存在传入参数里，字节的位偏移可以通过对位地址求 8 的模或屏蔽位 7 到位 3 获得。代码如下：

```
BitAddr MACRO arg
    .if arg>7FH
        and     arg,11111000B
    .else
```

```
            shr     arg,3
            add     arg,20H
        .endif
    endm
```

2.3 执行逻辑模拟

指令执行是模拟的关键。CPU 模拟的原理基本上可以分一下几类：（1）将目标机的指令编译成本机指令（2）解释目标机指令（3）设立虚拟机。第一种方法效率最高，但是每次执行目标机代码需要进行重新编译；第二种方法比较易于实现，但是要获得好的执行效率需要较好的算法；第三种方法我尚且不知道是否可行，因为虚拟机一般只能模拟本机指令，比如 386 以上处理器的 v86 模式，就是在一台机器上模拟多个 8086 系统。经过综合考虑后，我选择了第二种方法。接下来要考虑的就是实现方式了。起初我使用对指令字(Opcode)逐一比对的方法，虽然该方法比较简单而且容易扩充，但无疑效率低下，特别是一些排在比较队列后面的指令，执行延时将相当大。后来，我想到了使用跳转表的方法，经过多次尝试和优化，我得出了最终的方法。

2.3.1 指令解析

8051 的指令字都是单字节的，这为建立跳转表带来了方便。跳转表根据 8051 的指令字的顺序建立，每项的内容为相应指令的解释子程序的入口地址，由于是使用 32 位系统，每个入口地址为 32 位，所以跳转表的每项为一个双字(DW)。进行跳转时，由于 x86 支持带地址比例系数的跳转，因此很容易根据 8 位的 8051 指令字映射到 32 位的跳转表里。具体实现参见以下跳转表的定义（部分）和 StepRun 的代码：

```
OPTable dd   OP_NOP                 ;00H
        dd   OP_AJMP
        dd   OP_LJMP
        dd   OP_RR
        dd   OP_INC_A
        dd   OP_INC_Mem
        dd   2 DUP (OP_INC_Ri)
        dd   8 DUP (OP_INC_Rn)      ;08H
        dd   OP_JBC                 ;10H
```

```
        dd   OP_ACALL
        dd   OP_LCALL
        dd   OP_RRC
        dd   OP_DEC_A
        dd   OP_DEC_Mem
        dd   2 DUP (OP_DEC_Ri)      ;16H
        dd   8 DUP (OP_DEC_Rn)      ;18H
        dd   OP_JB                  ;20H
                .
                .
                .

StepRun proc
    push        esi
    push        edi                 ;esi,edi 在子程序中将被使用
    push        ebx
    GetCode     ebx                 ;获得当前的指令字
    call        [OPTable+4*ebx]     ;Call process by the OpCode
    pop         ebx
    pop         edi
    pop         esi
    ret
StepRun endp
```

以 OP_开头的子程序皆为指令解释子程序，由于在涉及寄存器的指令中，一个指令对不同寄存器的指令字都不同，所以用 DUP 定义多个跳转项更为简洁。