1 中断和异常处理

1.1 异常和异常处理

处理器为了实现处理异常和中断，使用了一个数据结构，也就是中断描述符表，用于存放中断描述符。同时处理器为每个异常和中断条件都赋予了一个向量，用于作为中断描述符表的索引号。

中断可以从硬件和软件产生。外部中断通过 INTR 和 NMI 接收。NMI 接收的中断是不可屏蔽中断，其向量号为 2。INTR 接收的外部中断可屏蔽，通过设置 EFLAGS 中的 IF 位为 0 来屏蔽这些中断。这里的 IF 标志可以通过 STI 和 CLI 来设置或清零。只有当程序的 CPL(程序特权级) 小于 IOPL(I/O 特权级字段) 时，才可执行这两条指令。还有几种情况可以影响 IF 标志，比如 PUSHF 指令、IRET 指令等。当通过中断门处理一个中断时，IF 会被自动清零。

软件中断主要借助 INT 指令，在指令操作数中提供中断向量号。向量号 0 到 255 都可以作为 INT 指令的中断号，比如指令 INT 0x80 可以执行系统中断。EFLAGS 中的 IF 标志无法屏蔽软件中断。

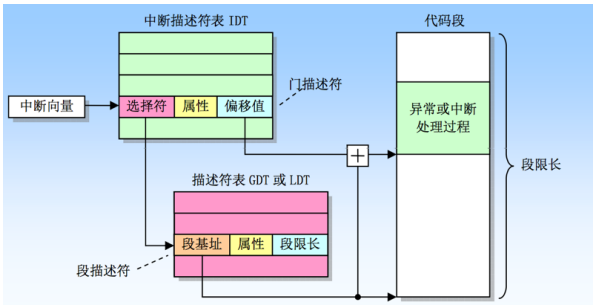
1.2 IDT 和 IDT 描述符

中断描述符表类似于全局描述符表，用于存放门描述符。处理器使用 IDTR 寄存器定位 IDT 表的位置，IDTR 有 48 位，高 32 位是 IDT 表的基地址，而低 16 位为 IDT 的长度值。使用 LIDT 指令可以将内存中的基地址和限长值加载到 IDTR 寄存器中，不过该指令只能由 CPL 为 0 的代码执行。

IDT 存放的门描述符有三类，为中断门描述符、陷阱门描述符和任务门描述符，它们都是 8 字节的。中断门描述符和陷阱门描述符中存放了段选择符和偏移值，用于对段的寻址，从而将程序执行权转移到代码段中异常或中断的处理过程中。而任务门描述符中含有段选择符，不过没有偏移值，因为在门中的偏移值没有意义。

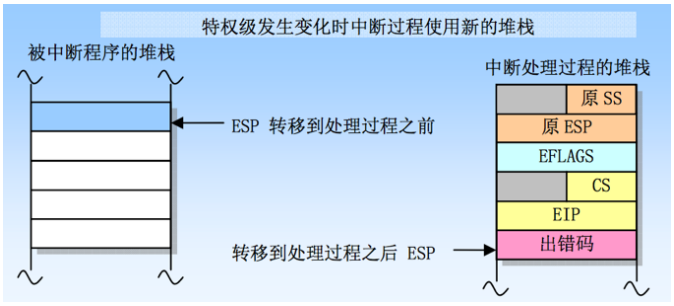
1.3 异常与中断处理

发生异常或中断时，处理器使用异常或中断的向量作为 IDT 表的索引，从而得到相应的门描述符。门描述符中的段选择符指向 GDT 或 IDT 中的段描述符，而段描述符给出相应代码段的段基址。门描述符中的偏移值作为段基址的偏移，从而指向执行异常或中断处理过程的代码段。我觉得下面这张图描述得很形象。



异常或中断处理分为两种情况。一种是在高特权级下执行，一种是在同一特权级下执行。

处理过程在高特权级上执行时，将会发生堆栈切换操作。此时处理器首先获得新栈的段选择符 SS 和栈指针 ESP，这里的段选择符和栈指针由当前任务的 TSS 段提供。然后把原栈选择符和栈指针压入新栈。随后将 EFLAGS、CS 和 EIP 当前值压入新栈。如果异常会产生一个错误号，该错误号也将被压入新栈。如下图所示。



处理过程在同一特权级上执行时，过程相对简单。处理器将 EFLAGS、CS 和 EIP 当前值压入堆栈。如果异常会产生一个错误号，该错误号也会被压入堆栈。这个过程如下图所示。

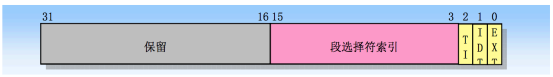
中断处理过程结束时，程序使用 IRET 指令从中断处理过程中返回。此时处理器会从堆栈中弹出代码段的选择符 CS 和指令指针 EIP。同时 IRET 会把保存的寄存器内容恢复到 EFLAGS 中。在特权级保护机制下，只有当 CPL 为 0 时，IOPL 字段才会被恢复。只

有 CPL 小等于 IOPL 时，IF 标志才会被改变。如果中断处理过程中发生了堆栈切换，那么 IRET 指令会切换回原来的堆栈。

需要注意的是，通过中断门访问异常或中断处理过程时，处理器会清零 IF 标志，随后再使用 IRET 指令恢复 IF 标志。而通过陷阱门访问处理过程则不会影响 IF 标志。

1.4 错误码

错误码类似于段选择符，用于寻址段。不过这里的段是与特定的异常条件有关的。段错误码的格式如下所示。



图中的低三位为 TI、IDT 和 EXT。EXT 为 0 时，表示执行程序以外的事件造成了异常。IDT 为 0 时，错误码的索引指向 GDT 或 LDT 的段描述符；当 IDT 为 1 时，错误码的索引指向 IDT 的一个门描述符。只有当 TI 为 0 时 TI 标志才有用。因为 TI 标志用于选择 GDT 表和 LDT 表。当 TI 为 1 时，错误码的索引部分指向 LDT 的段描述符；当 TI 为 0 时，错误码的索引部分指向 GDT 表中的描述符。

错误码有一个特例，也就是页故障异常的错误码。页故障错误码中没有段选择符索引，只有最低 3 位比特位有效，分别为 P、W/R 和 U/S。P=0 时，表示也不存在；P 为 1 时，表示违反页级保护权限。W/R=0 时，表示读操作引起了异常；W/R=1 时，表示异常由写操作引起。U/S=0 时，表示异常发生时 CPU 正在执行超级用户代码；U/S=1 时，表示异常发生时 CPU 正在执行一般用户代码。在第一次学习报告有提到过，CR2 控制寄存器中存放着引起页面故障异常的线性地址。

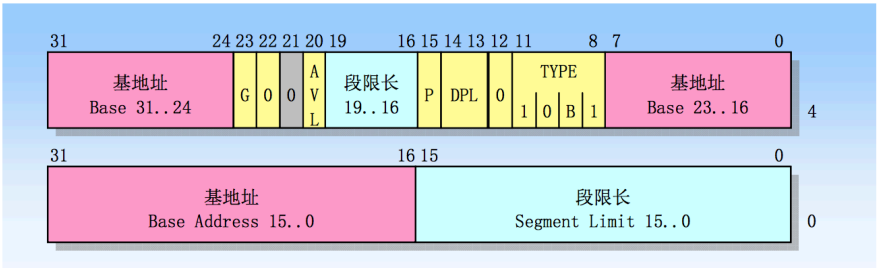
2 任务管理

2.1 用于任务管理的数据结构

为了进行任务管理，处理器定义了一些寄存器和数据结构，分别为任务状态段 TSS、TSS 描述符、任务寄存器 TR 和任务门描述符。

用于恢复一个任务执行的处理器状态信息被保存在 TSS 中。TSS 分为两类字段，一个是动态字段，还有一个是静态字段。当任务切换而被挂起时，处理器会更新动态字段的内容。而静态字段通常不会改变，它的字段内容在任务被创建时设置。

TSS 由任务状态段描述符来寻址和定义，以下是 TSS 段描述符的格式。

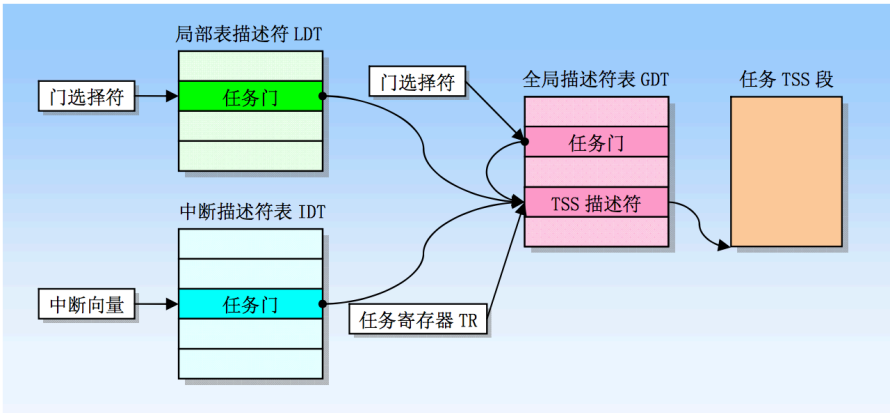


描述符中的 TYPE 字段中的 B 标志用于表示任务是否处于忙状态。B=1 时，表示任务正忙；B=0 时，表示任务处于非活动状态。描述符中的 G 标志是颗粒度，当 G=0 时，TSS 段的长度必须大等于 104 字节。描述符中的 DPL 标志用于特权级保护机制中。当发生任务切换时，访问 TSS 的程序的 CPL 必须小等于 TSS 中的 DPL。描述符中的段基址就是任务状态段的基址。

任务寄存器存放着段选择符和当前 TSS 段描述符。LTR 指令可以在系统初始化阶段给 TR 寄存器加载初值，之后 TR 的内容会在任务切换时自动地被改变。

除了使用段选择符直接访问 GDT 中的 TSS 描述符，还可以通过任务门描述符间接地访问 TSS 描述符，因为任务门描述符含有 TSS 选择符字段。任务门描述符中的 DPL 用于支持特权级保护机制。当程序通过任务门描述符调用程序时，程序的 CPL 以及指向任务门描述符的门选择符的 RPL 都必须小等于任务门描述符中的 DPL。

下图描述了调用任务的两种方式，一种是通过任务门描述符访问，一种是通过 TSS 段描述符访问。从图中可以看出，任务门描述符可以存放在 GDT、LDT 或 IDT 表中，程序通过任务门描述符间接访问到 TSS 描述符。



2.2 任务切换

处理器可以通过 4 种方式进行任务切换操作，分别是：1. 对 TSS 描述符执行 JMP 或 CALL 指令。2. 对任务门描述符执行 JMP 或 CALL 指令。3. 通过中断或异常向量指向任

务门描述符。4. 执行 IRET 指令。

以下是进行任务切换的过程:

- 首先获得新任务的 TSS 段选择符。段选择符的获取有三种途径: 1. 从 JMP 或 CALL 指令操作数中获取。2. 中断向量索引到 IDT 表中的任务门描述符，获取其中的 TSS 选择符。3. 执行 IRET 指令时，从当前 TSS 的前一任务链接字段中获取。

  这里说一下前一任务链接字段，它需要和 EFLAGS 中的 NT 标志配合使用。NT 标志为 1 时，表明当前任务嵌套在另一个任务中执行，并且当前任务的 TSS 段的前一任务链接字段中存放着高一层任务的 TSS 选择符。

- 经过特权级保护机制的检查。使用 JMP 或 CALL 调用程序时，当前任务的 CPL 和新任务的 TSS 段选择符的 RPL 必须小等于新任务 TSS 段描述符的 DPL。发生中断、异常或使用 IRET 指令时，则无视特权级保护机制。

- 对 B 标志和 NT 标志的设置。如果使用 JMP 进行任务切换，则将 B 标志清零。如果使用 IRET 指令进行任务切换，则将 B 标志和 NT 标志清零。

- 将当前任务状态保存到当前任务的 TSS 中，包括所有通用寄存器的值，段寄存器中的段选择符，标志寄存器 EFLAGS 以及指令指针 EIP。

- 加载新的 TSS 段描述符。如果任务切换由 CALL、JMP、异常或者中断产生，则对新 TSS 段描述符中的 B 标志置 1。

- 将新 TSS 的段选择符和描述符加载到任务寄存器中。同时设置 CR0 寄存器的任务已切换标志 TS 为 1。

- 将新 TSS 状态加载进处理器，包括所有通用寄存器、段选择符、标志寄存器 EFLAGS、LDTR 寄存器、CR3 寄存器以及 EIP。如果任务切换由 CALL、JMP、异常或者中断产生，则将 EFLAGS 中的 NT 位置一。

- 开始执行新任务。

2.3 任务地址空间

任务的地址空间由任务能够访问的段构成，这些段有代码段、数据段、堆栈段、TSS 中引用的系统段以及任务代码能够访问的任何其他段。

在任务之间共享数据有以下三个途径:

- 通过 GDT 共享数据。这个方法是很明显而且简单的。不同的段可以映射到相同的物理地址空间中，而每个段又有对应的段描述符。这些段描述符存放在 GDT 表中。于是任务通过 GDT 共享相同的物理地址空间。这种方法的缺点是所有任务都可以共享这些段中的代码和数据。

- 让任务共享相同的 LDT。让一些特定的任务的 TSS 中 LDT 字段指向同一个 LDT，从而访问到相同的物理地址空间。
- 让不同 LDT 中的某些段描述符映射到相同的物理地址。这样的段描述符通常被称为别名段。

3 编写保护模式

3.1 从实模式跳转到保护模式

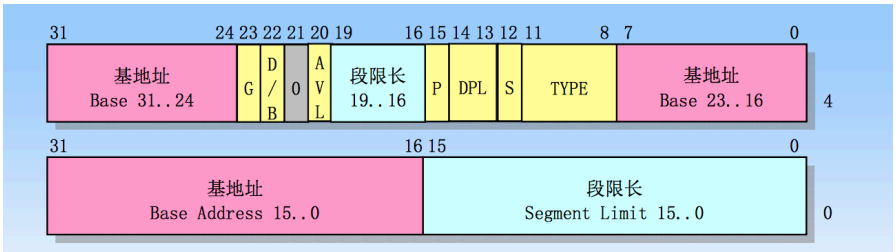
首先定义段描述符的数据结构。根据 X86 下的段描述符通用格式可知，段描述符都是 64 位的，也就是 8 字节。

```
%macro  Descriptor  3
        dw  %2 & 0FFFFh ; dw = define word 段限长1
        dw  %1 & 0FFFFh ; 段基址1
        db  (%1 >> 16) & 0FFh ; db = define byte 段基址2
        dw  ((%2 >> 8) & 0F00h) | (%3 & 0F0FFh) ; 属性1+段限长2+属性2
        db  (%1 >> 24) & 0FFh ; 段基址3
%endmacro
```

定义这个数据结构的时候，使用了多行宏。多行宏的格式如下:

```
%macro prologue 1 ; 这里的数字1定义了可以接收的参数个数
    push    ebp
    mov     ebp,esp
    sub     esp,%1 ; 用于引用宏调用中的第一个参数
%endmacro
```

现在回过头来看描述符的格式，之所以将它定义为这样的格式，是为了遵照 X86 中的段描述符通用格式，见下图。



接着我们需要写 gdt 的段。这个段定义 GDT 描述符表。正如上一份学习报告所写的，GDT 描述符表是一个存放段描述符的数组。

```
[SECTION .gdt]

```

```
3         ; GDT描述符表
4         ; 描述符第一个参数是段基址
5         LABEL_GDT:           Descriptor  0,       0,                    0
6         ; 由于32位代码段的段基址在此无法确定，所以先设为0，之后再初始化
7         LABEL_DESC_CODE32:   Descriptor  0,       SegCode32Len - 1,     DA_C + DA_32
8         ; 这个段描述符指向的是显存
9         LABEL_DESC_VIDEO:    Descriptor  0B8000h,0ffffh ,              DA_DRW
10
11        ; GDT表的长度，nasm中$代表当前行相对于段基址的偏移地址
12        GdtLen   equ $ - LABEL_GDT
13
14        ; GdtPtr的前2两个字节为GDT表的边界，后4个字节是GDT表的基地址
15        GdtPtr   dw  GdtLen - 1
16                 dd  0 ; define double-word
17
18        ; 以下是两个段选择符，用于索引GDT表中的段描述符
19        SelectorCode32   equ LABEL_DESC_CODE32 - LABEL_GDT
20        SelectorVideo    equ LABEL_DESC_VIDEO - LABEL_GDT
```

接下来我们要编写 [SECTION .s16] 的代码段，这是个 16 位代码段。我使用这个代码段跳转进入保护模式。

在给出代码之前，我想说一下为什么需要定义一个 16 位的代码段。首先要知道，在 16 位代码段中，跳转目标的偏移用 16 位表示。其次还要认识到，CPU 还处在实模式下。在实模式下，CPU 总是进行 16 位的跳转，也就是当他在解析跳转目标的时候，总是读取内存中的 16 位的值作为跳转目标。为此，汇编器要配合 CPU，产生这种用 16 位表示偏移量的代码。

下面就是我要写的 16 位代码段。

```
1         [SECTION .s16]
2         [BITS 16] ; 告诉编译器这个段需要按照16位进行编译
3         LABEL_BEGIN:
4             ; cs存放着cpu执行的代码段的基地址
5             ; 初始化寄存器中的值
6             mov ax,cs
7             mov ds,ax
8             mov es,ax
9             mov ss,ax
10            mov sp,0100h
11
12            ; 初始化LABEL_DESC_CODE32描述符
13            xor eax,eax ; 将eax中的值清零
14            mov ax,cs
15            ; 下面两句相当于cs*16+偏移地址
16            shl eax,4
17            add eax,LABEL_SEG_CODE32
18            ; 这样eax里就存放着32位代码段的物理地址
19            ; 接下来4条语句分别初始化了32位代码段描述符的基地址1、基地址2和基地址3
20            ; 要注意段描述符是8字节，这样会比较好理解代码
21            mov word [LABEL_DESC_CODE32 + 2], ax
22            shr eax,16
23            mov byte [LABEL_DESC_CODE32 + 4], al
24            mov byte [LABEL_DESC_CODE32 + 7], ah
```

```
25
26              ; 为加载GDTR作准备
27              xor eax,eax    ; 将eax中的值清零
28              mov ax,ds
29              ; 下面两句得到了GDT表在代码段中的物理地址
30              shl eax,4
31              add eax,LABEL_GDT
32              ; 初始化指向GDT表的指针，修改GdtPtr中的GDT基地址
33              mov dword [GdtPtr + 2], eax
34
35              ; 关闭中断
36              cli
37
38              ; 加载GdtPtr到寄存器gdtr
39              lgdt [GdtPtr]
40
41              ; 打开地址线A20
42              in   al,92h
43              or   al,00000010b
44              out 92h,al
45
46              ; 准备切换到保护模式
47              ; 将cr0的第0位设为1。cr0的第0位是PE位，PE=1时，CPU运行于保护模式。
48              mov eax,cr0
49              or   eax,1
50              mov cr0,eax
51
52              ; 真正进入到保护模式
53              jmp dword SelectorCode32:0
```

上述代码的注释虽然已经描述得很详细，但我觉得还是有必要完整得叙述一遍从实模式切换到保护模式的过程。

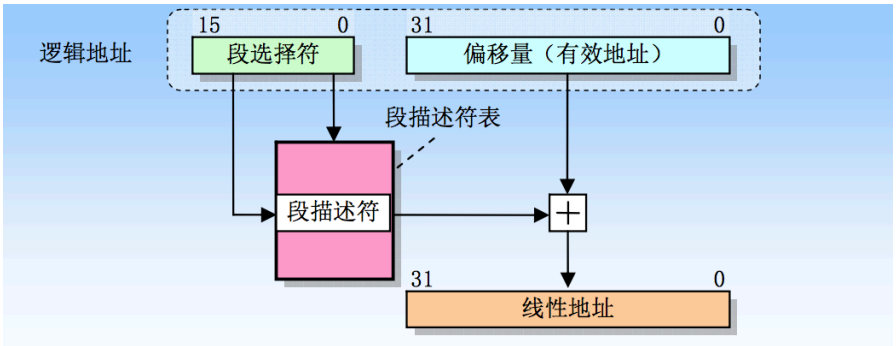
- 首先初始化一些起码的系统数据结构，比如 GDT 表和一个代码段。
- 禁止中断，这是为了确保在模式切换操作期间不产生异常和中断。
- 执行 LGDT 指令把 GDT 表的基地址加载进入 GDTR 寄存器。
- 将控制寄存器 CR0 中的 PE 标志置一。
- 进入保护模式。

接下来编写 32 位代码段。类似于 16 位代码段，在 32 位代码段中，跳转目标的偏移用 32 位表示。

```
1       [SECTION .s32]
2       [BITS 32] ; 告诉编译器这个段需要按照32位进行编译
3
4       LABEL_SEG_CODE32:
5           mov ax, SelectorVideo
6           ; gs是80386起增加的辅助段寄存器
7           ; gs目前存放了LABEL_DESC_VIDEO段描述符对应的段选择符
```

```
8               mov gs, ax
9               mov edi, (80 * 11 + 79) * 2
10              mov ah, 0Ch
11              mov al, 'P'
12              ; 逻辑地址gs:edi，经过段机制后转化为线性地址，为LABEL_DESC_VIDEO偏移edi
                    位
13              ; 将ax的值写入显存中偏移edi的位置
14              mov [gs:edi], ax
15              jmp $
16
17          SegCode32Len equ $ - LABEL_SEG_CODE32
```

虽然上一次学习报告已经提到过段机制，但是因为比较简略，所以在此我附上一张段式寻址方式的图加以说明。



3.2 从保护模式跳转到实模式

如果想要切换回实地址模式，需要按照下面的步骤:

- 首先禁止中断。
- 把程序的控制转移到长度为 64KB 的可读段中。这步操作使用实模式要求的段长度加载 CS 寄存器。
- 使用一个特殊定义的选择符来加载 SS、DS、ES、FS 和 GS 段寄存器，这个选择符指向的段描述符段限长为 64KB，颗粒度为字节，向上扩展，可写，且存在于内存中。
- 清除 CR0 中的 PE 标志。
- 执行一个远跳转指令跳转到实模式程序中。
- 加载实地址模式代码会使用的 SS、DS、ES、FS 和 GS 寄存器。

相应的代码如下，需要注意的是，跳转回实模式的代码需要在 16 位代码段中完成。

```
LABEL_GDT: Descriptor 0,0,0
LABEL_DESC_NORMAL: Descriptor 0,0ffffh,DA_DRW

SelectorNormal equ LABEL_DESC_NORMAL-LABEL_GDT

[SECTION .s16code]
[BITS 16]
LABEL_SEG_CODE16:
    cli
    ; 使用一个特殊定义的选择符来加载SS、DS、ES、FS和GS段寄存器
    mov ax,SelectorNormal
    mov ds,ax
    mov es,ax
    mov fs,ax
    mov gs,ax
    mov ss,ax

    mov eax,cr0
    and al,11111110b
    mov cr0,eax

    mov ax,cs
    mov es,ax
    mov ds,ax
    mov ss,ax

    ; 这里的SPValueInRealMode是在16位代码段初始化时赋值的
    mov sp,[SPValueInRealMode]

    in al,92h
    and al,11111101b
    out 92h,al

    sti
Code16Len equ $-LABEL_SEG_CODE16
```

3.3 完整的例子

我阅读了一个完整的例子，自己能理解它的意思，也把自己所理解的注释写在代码中了。看完这个例子，我有自信自己可以写出引导区代码是如何让操作系统从保护模式跳转到实模式，以及让操作系统从实模式跳转回保护模式。

在看代码之前，我觉得有必要先了解一些 X86 中的段寄存器

- x86 的段寄存器有 CS 寄存器、DS 寄存器、SS 寄存器、ES 寄存器、FS 以及 GS 寄存器。CS 寄存器存放着程序代码段的基地址。DS 寄存器存放着程序数据段的基地址。SS 寄存器堆栈段的基地址。ES、FS 和 GS 寄存器是扩展寄存器，用于内存寻址，常与变址寄存器配合使用。

- x86 的指针寄存器有 IP 寄存器、SP 寄存器和 BP 寄存器。IP 寄存器常与 CS 配合，

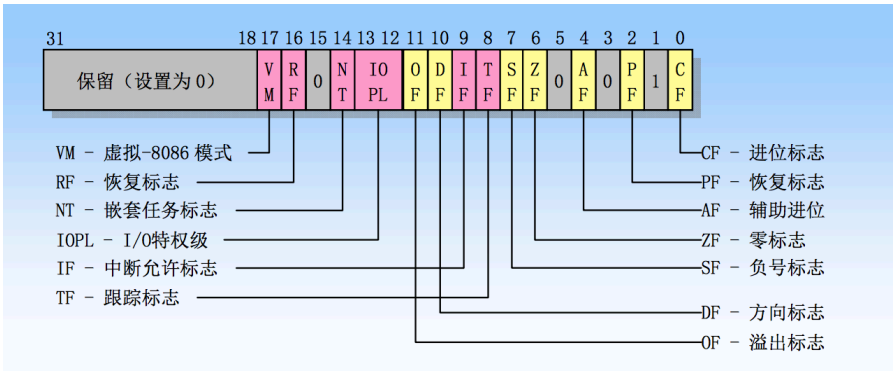
用于寻址下一条要执行的指令。SP 寄存器常与 SS 配合，用于寻址堆栈中的数据。BP 寄存器是一个扩展寄存器。

- x86 的通用寄存器有 AX 寄存器、BX 寄存器、CX 寄存器和 DX 寄存器。AX 寄存器用于输入输出以及大部分的算术运算。BX 寄存器是主要的基址寄存器，用于和变址寄存器配合。CX 寄存器是计数寄存器。DX 寄存器是数据寄存器。
- x86 的变址寄存器有 SI 寄存器和 DI 寄存器。SI 寄存器是源变址寄存器，DI 是目的变址寄存器。

为了更好的理解代码，我还去了解了 EFLAGS 标志寄存器中的各个标志位。

- 进位标志 CF。这个标志用于反映运算是否产生进位或借位。
- 奇偶标志 PF。用于反映运算结果中 1 的个数的奇偶性。
- 辅助进位标志 AF。在字操作时，发生低字节向高字节进位或借位时，AF=1。在字节操作时，发生低 4 位向高 4 位进位或借位时，AF=1。
- 零标志 ZF。用于反映运算结果是否为 0。
- 符号标志 SF。用于反映运算结果的符号位。
- 溢出标志 OF。用于反映有符号数加减所得结果是否溢出。
- 追踪标志 TF。TF=1 时，CPU 进入单步执行方式。
- 中断允许标志 IF。用于决定 CPU 是否响应 CPU 外部的可屏蔽中断发出的中断请求。
- 方向标志 DF。用于决定在串操作指令执行时有关指针寄存器发生调整的方向。
- IOPL 标志位，也就是 I/O 特权级字段。
- 嵌套任务标志 NT，这在之前有提到过。
- 重启动标志 RF，用于控制是否接收调试故障。
- 虚拟 8086 方式标志 VM。VM=1 时，表示处理机处于虚拟的 8086 方式下的工作状态。

下图是标志寄存器 EFLAGS。



下面就是我所读的代码。我在 linux 上调试了这段代码，不过由于是在另外一台电脑上，所以就不附上调试过程了。

```
%include "pm.inc"

org 0100h
    jmp LABEL_BEGIN

[SECTION .gdt]

LABEL_GDT: Descriptor 0,0,0
LABEL_DESC_NORMAL: Descriptor 0,0ffffh,DA_DRW
LABEL_DESC_CODE32: Descriptor 0,SegCode32Len-1,DA_C+DA_32
LABEL_DESC_CODE16: Descriptor 0,0ffffh,DA_C
LABEL_DESC_DATA: Descriptor 0,DataLen-1,DA_DRW
LABEL_DESC_STACK: Descriptor 0,TopOfStack,DA_DRWA+DA_32
LABEL_DESC_TEST: Descriptor 0500000h,0ffffh,DA_DRW
LABEL_DESC_VIDEO: Descriptor 0B8000h,0ffffh,DA_DRW

GdtLen equ $-LABEL_GDT
GdtPtr equ dw GdtLen-1
          dd 0

SelectorNormal equ LABEL_DESC_NORMAL-LABEL_GDT
SelectorCode32 equ LABEL_DESC_CODE32-LABEL_GDT
SelectorCode16 equ LABEL_DESC_CODE16-LABEL_GDT
SelectorData equ LABEL_DESC_DATA-LABEL_GDT
SelectorStack equ LABEL_DESC_STACK-LABEL_GDT
SelectorTest equ LABEL_DESC_TEST-LABEL_GDT
SelectorVideo equ LABEL_DESC_VIDEO-LABEL_GDT

[SECTION .data1]
ALIGN 32
[BITS 32]
LABEL_DATA:
    SPValueInRealMode dw 0
; 在NASM中，变量和标签是一样的
; 下面这句等价于 PMMessage db "In protect mode now",0
PMMessage: db "In protect mode now",0
    OffsetPMMessage equ PMMessage-$$
StrTest: db "ABCDEFGHJKLMNOPQRSTUVWXYZ",0
```

```
39      OffsetStrTest equ StrTest-$$
40  DataLen equ $-LABEL_DATA
41
42  [SECTION .gs]
43  ALIGN 32
44  [BITS 32]
45  LABEL_STACK:
46      times 512 db 0
47  TopOfStack equ $-LABEL_STACK-1
48
49  [SECTION .s16]
50  [BITS 16]
51  LABEL_BEGIN:
52      ; cs存放着代码段的段基址
53      mov ax,cs
54      mov es,ax
55      mov ds,ax
56      mov ss,ax
57      mov sp,0100h
58
59      mov [LABEL_GO_BACK_TO_REAL+3],ax
60      mov [SPValueInRealMode],sp
61
62      mov ax,cs
63      ; 将我们的源操作数取出来,然后置于目的操作数,目的操作数其余位用0填充
64      movzx eax,ax
65      ; 另一种实现方法
66      ; xor eax,eax
67      ; mov ax,cs
68      shl eax,4
69      add eax,LABEL_SEG_CODE16
70      mov word [LABEL_DESC_CODE16+2],ax
71      shr eax,16
72      mov byte [LABEL_DESC_CODE16+4],al
73      mov byte [LABEL_DESC_CODE16+7],ah
74
75      xor eax,eax
76      mov ax,cs
77      shl eax,4
78      add eax,LABEL_SEG_CODE32
79      mov word [LABEL_DESC_CODE32+2],ax
80      shr eax,16
81      mov byte [LABEL_DESC_CODE32+4],al
82      mov byte [LABEL_DESC_CODE32+7],ah
83
84      xor eax,eax
85      ; ds存放着数据段的段基址
86      mov ax,ds
87      shl eax,4
88      add eax,LABEL_DATA
89      mov word [LABEL_DESC_DATA+2],ax
90      shr eax,16
91      mov byte [LABEL_DESC_DATA+4],al
92      mov byte [LABEL_DESC_DATA+7],ah
93
94      xor eax,eax
```

```
95      mov ax,ds
96      shl eax,4
97      add eax,LABEL_STACK
98      mov word [LABEL_DESC_DATA+2],ax
99      shr eax,16
100     mov byte [LABEL_DESC_DATA+4],al
101     mov byte [LABEL_DESC_DATA+7],ah
102
103     ; 初始化指向GDT表的指针
104     xor eax,eax
105     mov ax,ds
106     shl eax,4
107     add eax,LABEL_GDT
108     mov dword [GdtPtr+2],eax
109
110     lgdt [GdtPtr]
111
112     cli
113
114     ; 打开A20地址线
115     in al,92h
116     or al,00000010b
117     out 92h,al
118
119     ; 修改CR0的PE位
120     mov eax,cr0
121     or eax,1
122     mov cr0,eax
123     sti
124
125     jmp dword SelectorCode32:0
126
127 LABEL_REAL_ENTRY:
128     mov ax,cs
129     mov ds,ax
130     mov es,ax
131     mov ss,ax
132
133     mov sp,[SPValueInRealMode]
134
135     in al,92h
136     and al,11111101b
137     out 92h,al
138     sti
139
140     mov ax,4c00h
141     int 21h ; 返回DOS
142
143 [SECTION .s32]
144 [BITS 32]
145 LABEL_SEG_CODE32:
146     ; 加载数据段描述符的选择符
147     mov ax,SelectorData
148     mov ds,ax
149     ; 加载测试段描述符的选择符
150     mov ax,SelectorTest
```

```
151     mov es,ax
152     ; 加载视频段描述符的选择符
153     mov ax,SelectorVideo
154     mov gs,ax
155     ; 加载堆栈段描述符的选择符
156     mov ax,SelectorStack
157     mov ss,ax
158 
159     ; 加载堆栈段的段指针，它是向下扩展的
160     mov esp,TopOfStack
161 
162     mov ah,0Ch
163     ; esi和edi是x86中的变址寄存器
164     xor esi,esi
165     xor edi,edi
166     mov esi,OffsetPMMessage
167     mov edi,(80 + 10 + 0) * 2
168     ; 设置从低地址向高地址传送
169     cld
170 .1:
171     ; 相当于 load ES:ESI to al, ESI将自动增加
172     lodsb
173     ; Performs a bit-wise logical AND of the two operands
174     ; The result of a bit-wise logical AND is 1 if the value of that bit in both
          operands is 1; otherwise, the result is 0.
175     ; The OF and CF flags are cleared; SF, ZF and PF flags are set according to the
          result.
176     test al,al
177     ; JZ，当ZF=1时跳转
178     ; 当ZF=1时，说明指向测试数据段已经到了尽头
179     ; 用来测试一方寄存器是否为空
180     jz .2
181     mov [gs:edi],ax
182     add edi,2
183     jmp .1
184 .2:
185     call DispReturn
186     call TestRead
187     call TestWrite
188     call TestRead
189 
190     jmp SelectorCode16:0
191 
192 TestRead:
193     xor esi,esi
194     mov ecx,8
195 .loop:
196     mov al,[es:esi]
197     call DipsAL
198     inc esi
199     ; loop decrements the count register
200     loop .loop
201     call DispReturn
202     ret
203 
204 TestWrite:
```

```
205     push esi
206     push edi
207     xor esi,esi
208     xor edi,edi
209     mov esi,OffsetStrTest
210     cld
211 .1:
212     lodsb
213     test al,al
214     jz .2
215     mov [es:edi],al
216     inc edi
217     jmp .1
218 .2:
219     pop edi
220     pop esi
221     ret
222
223 ; 用于显示al中的数字
224 DipsAL:
225     push ecx
226     push edx
227
228     mov ah,0Ch
229     mov dl,al
230     shr al,4
231     mov ecx,2
232 .begin:
233     and al,01111b
234     cmp al,9
235     ja .1
236     add al,'0'
237     jmp .2
238 .1:
239     sub al,0Ah
240     add al,'A'
241 .2:
242     mov [gs:edi],ax
243     add edi,2
244
245     mov al,dl
246     loop .begin
247     add edi,2
248
249     pop edx
250     pop ecx
251
252     ret
253
254 ; 模拟一个回车符的显示
255 DispReturn:
256     push eax
257     push ebx
258     mov eax,edi
259     mov bl,160
260     div bl
```

```
261     and eax,0FFh
262     inc eax
263     mov bl,160
264     mul bl
265     mov edi,eax
266     pop ebx
267     pop eax
268     ret
269
270 SegCode32Len equ $-LABEL_SEG_CODE32
271
272 [SECTION .s16code]
273 ALIGN 32
274 [BITS 16]
275 LABEL_SEG_CODE16:
276     cli
277
278     mov ax,SelectorNormal
279     mov ds,ax
280     mov es,ax
281     mov fs,ax
282     mov gs,ax
283     mov ss,ax
284
285     mov eax,cr0
286     and al,11111110b
287     mov cr0,eax
288
289     ; 在16位代码段初始化的过程中有 mov [LABEL_GO_BACK_TO_REAL+3],ax
290     ; LABEL_GO_BACK_TO_REAL是"jmp 0:LABEL_REAL_ENTRY"指令语句前的标号
291     ; LABEL_GO_BACK_TO_REAL后第四个和第五个字节就是段地址所在
292     ; 所以jmp指令中的段地址0就被改成了ax中的值
293 LABEL_GO_BACK_TO_REAL:
294     jmp 0:LABEL_REAL_ENTRY
```