EBook Gratis

APRENDIZAJE

Intel x86 Assembly Language & Microarchitecture

Free unaffiliated eBook created from
Stack Overflow contributors.

#x86

Tabla de contenido

Acerca de ... 1

Capítulo 1: Primeros pasos con Intel x86 Assembly Language & Microarchitecture ... 2

Observaciones ... 2

Examples ... 2

lenguaje ensamblador x86 ... 2

Linux x86 ejemplo de Hello World ... 3

Capítulo 2: Convenciones de llamadas ... 7

Observaciones ... 7

Recursos ... 7

Examples ... 7

Cdecl de 32 bits ... 7

Parámetros ... 7

Valor de retorno ... 8

Registros guardados y aplastados ... 8

Sistema V de 64 bits ... 8

Parámetros ... 8

Valor de retorno ... 8

Registros guardados y aplastados ... 9

Llamada de 32 bits ... 9

Parámetros ... 9

Valor de retorno ... 9

Registros guardados y aplastados ... 9

32 bits, cdecl - Tratar con enteros ... 9

Como parámetros (8, 16, 32 bits) ... 9

Como parámetros (64 bits) ... 10

Como valor de retorno ... 10

32 bits, cdecl - Tratar con punto flotante ... 11

Como parámetros (float, doble) ... 11

Como parámetros (doble largo) ... 11

Como valor de retorno 12
Windows de 64 bits 13
Parámetros 13
Valor de retorno 13
Registros guardados y aplastados 13
Alineación de la pila 13
32 bits, cdecl - Manejo de estructuras 14
Relleno 14
Como parámetros (pasar por referencia) 14
Como parámetros (pasar por valor) 14
Como valor de retorno 15
Capítulo 3: Convertir cadenas decimales en enteros 17
Observaciones 17
Examples 17
Asamblea IA-32, GAS, convención de cdecl 17
MS-DOS, función TASM / MASM para leer un entero sin signo de 16 bits 18
Lea un entero sin signo de 16 bits de la entrada. 18
Valores de retorno 19
Uso 19
Código 19
Portabilidad NASM 21
MS-DOS, función TASM / MASM para imprimir un número de 16 bits en binario, cuaternario, oc 21
Imprima un número en binario, cuaternario, octal, hexadecimal y una potencia general de do 21
Parámetros 22
Uso 22
Código 23
Datos 24
Portabilidad NASM 24
Extendiendo la función 24
MS-DOS, TASM / MASM, función para imprimir un número de 16 bits en decimal 25
Imprima un número sin firmar de 16 bits en decimal 25

Parámetros ... 25
Uso ... 26
Código ... 26
Portabilidad NASM ... 27
Capítulo 4: Ensambladores ... 28
Examples ... 28
Microsoft Assembler - MASM ... 28
Ensamblador de Intel ... 28
Ensamblador AT&T - como ... 29
Turbo Assembler de Borland - TASM ... 29
Ensamblador GNU - gas ... 30
Ensamblador Netwide - NASM ... 30
Sin embargo, otro ensamblador - YASM ... 31
Capítulo 5: Flujo de control ... 32
Examples ... 32
Saltos incondicionales ... 32
Relativo a saltos cercanos ... 32
Absolutos saltos indirectos cercanos ... 32
Saltos lejanos absolutos ... 32
Saltos lejanos indirectos absolutos ... 33
Saltos faltantes ... 33
Condiciones de prueba ... 33
Banderas ... 34
Ensayos no destructivos ... 34
Pruebas firmadas y no firmadas ... 35
Saltos condicionales ... 35
Sinónimos y terminología ... 35
Igualdad ... 36
Mas grande que ... 36
Menos que ... 37
Banderas especificas ... 37
Un salto más condicional (extra) ... 38

Probar relaciones aritméticas ... 38
Enteros sin firmar ... 38
Enteros firmados ... 39
a_label ... 40
Sinónimos ... 40
Códigos de acompañante firmados y sin firmar ... 40
Capítulo 6: Fundamentos del registro ... 41
Examples ... 41
Registros de 16 bits ... 41
Notas: ... 41
Registros de 32 bits ... 42
Registros de 8 bits ... 42
Registros de segmento ... 43
Segmentación ... 43
Registros del segmento original ... 43
Tamaño del segmento? ... 43
Más registros de segmento! ... 44
Registros de 64 bits ... 44
Registro de banderas ... 45
Códigos de condición ... 45
Accediendo a FLAGS directamente ... 46
Otras banderas ... 47
80286 Banderas ... 47
80386 Banderas ... 47
80486 Banderas ... 48
Banderas Pentium ... 48
Capítulo 7: Gestión multiprocesador ... 50
Parámetros ... 50
Observaciones ... 50
Examples ... 52
Despierta todos los procesadores ... 52

Capítulo 8: Manipulación de datos ... 60
Sintaxis ... 60
Observaciones ... 60
Examples ... 60
Usando MOV para manipular valores ... 60
Capítulo 9: Mecanismos de llamada al sistema ... 62
Examples ... 62
Llamadas del BIOS ... 62
Cómo interactuar con la BIOS ... 62
Usando llamadas del BIOS con función de selección ... 62
Ejemplos ... 62
Cómo escribir un carácter en la pantalla: ... 62
Cómo leer un carácter desde el teclado (bloqueo): ... 63
Cómo leer uno o más sectores desde una unidad externa (utilizando el direccionamiento CHS) ... 63
Cómo leer el sistema RTC (Real Time Clock): ... 63
Cómo leer la hora del sistema desde el RTC: ... 63
Cómo leer la fecha del sistema desde el RTC: ... 64
Cómo obtener el tamaño de memoria baja contigua: ... 64
Cómo reiniciar la computadora: ... 64
Manejo de errores ... 64
Referencias ... 64
Capítulo 10: Mejoramiento ... 66
Introducción ... 66
Observaciones ... 66
Examples ... 66
Poner a cero un registro ... 66
Mover la bandera de Carry a un registro ... 66
Fondo ... 66
Utilice 'sbb' ... 67
Pros ... 67
Contras ... 67
Prueba un registro para 0 ... 67

Fondo 67
test uso 67
Pros 68
Contras 68
Sistema Linux llama con menos hinchazón 68
Multiplica por 3 o 5 69
Fondo 69
Usar lea 69
Pros 69
Contras 69
Capítulo 11: Modos Real vs Protegido 71
Examples 71
Modo real 71
Modo protegido 72
Introducción 72
Diseño 72
Registro de segmento 72
Global / Local 72
Tabla Descriptora 73
Descriptor 73
¡Verdadera protección al fin! 73
Los errores 74
Cambio al modo protegido 74
Modo irreal 75
Capítulo 12: Paginación - Direccionamiento Virtual y Memoria 79
Examples 79
Introducción 79
Historia 79
Las primeras computadoras 79
Multiusuario, multiprocesamiento 79

Ejemplo 79
Sofisticación 79
Soluciones 79
Segmentación 80
Problemas 80
Paginacion 80
Direccionamiento virtual 80
Soporte de hardware y sistema operativo 80
Características de paginación 80
Multiprocesamiento 81
Datos escasos 81
Memoria virtual 81
Decisiones de paginación 82
¿Qué tan grande debe ser una página? 82
¿Cómo optimizar el uso de las tablas de páginas? 83
80386 Paginación 83
Diseño de alto nivel 83
Entrada de página 84
Page Directory Base de Registro (PDBR) 84
Fallas de la página 85
80486 Paginación 85
Paginación Pentium 86
Diseño de la dirección 86
Diseño de entrada de directorio 86
Extensión de dirección física (PAE) 86
Introducción 86
Más memoria RAM 87
Diseño 87
Extensión de tamaño de página (PSE) 88
PSE-32 (y PSE-40) 88
Creditos 90

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: intel-x86-assembly-language---microarchitecture

It is an unofficial and free Intel x86 Assembly Language & Microarchitecture ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Intel x86 Assembly Language & Microarchitecture.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Primeros pasos con Intel x86 Assembly Language & Microarchitecture

Observaciones

Esta sección proporciona una descripción general de qué es x86 y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema grande dentro de x86, y vincular a los temas relacionados. Dado que la Documentación para x86 es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

Examples

lenguaje ensamblador x86

La familia de lenguajes de ensamblaje x86 representa décadas de avances en la arquitectura original Intel 8086. Además de haber varios dialectos diferentes basados en el ensamblador utilizado, se han agregado instrucciones, registros y otras características adicionales del procesador a lo largo de los años, mientras que siguen siendo compatibles con el ensamblaje de 16 bits utilizado en la década de 1980.

El primer paso para trabajar con el ensamblaje x86 es determinar cuál es el objetivo. Si está buscando escribir código dentro de un sistema operativo, por ejemplo, también querrá determinar si elegirá usar un ensamblador independiente o funciones integradas de ensamblaje en línea de un lenguaje de nivel superior como C. Si Si desea codificar el código completo sin un sistema operativo, simplemente necesita instalar el ensamblador de su elección y comprender cómo crear un código binario que se puede convertir en memoria flash, imagen de arranque o cargar de otra manera en la memoria. Ubicación apropiada para comenzar la ejecución.

Un ensamblador muy popular que está bien soportado en varias plataformas es NASM (Netwide Assembler), que se puede obtener en http://nasm.us/ . En el sitio NASM, puede proceder a descargar la última versión de lanzamiento para su plataforma.

Windows

Tanto las versiones de 32 bits como las de 64 bits de NASM están disponibles para Windows. NASM viene con un instalador conveniente que se puede usar en su host de Windows para instalar el ensamblador automáticamente.

Linux

Bien puede ser que NASM ya esté instalado en su versión de Linux. Para verificar, ejecute:

```
nasm -v
```

Si no se encuentra el comando, deberá realizar una instalación. A menos que esté haciendo algo que requiera funciones de NASM de vanguardia, la mejor ruta es utilizar su herramienta de administración de paquetes incorporada para su distribución de Linux para instalar NASM. Por ejemplo, en sistemas derivados de Debian como Ubuntu y otros, ejecute lo siguiente desde un indicador de comando:

```
sudo apt-get install nasm
```

Para los sistemas basados en RPM, puede probar:

```
sudo yum install nasm
```

Mac OS X

Las versiones recientes de OS X (incluidas Yosemite y El Capitán) vienen con una versión anterior de NASM preinstalada. Por ejemplo, El Capitán tiene instalada la versión 0.98.40. Si bien esto probablemente funcionará para casi todos los propósitos normales, en realidad es bastante antiguo. En este momento, se lanza la versión 2.11 de NASM y la versión 2.12 tiene varios candidatos disponibles.

Puede obtener el código fuente de NASM desde el enlace anterior, pero a menos que tenga una necesidad específica para instalar desde la fuente, es mucho más sencillo descargar el paquete binario del directorio de la versión de OS X y descomprimirlo.

Una vez descomprimido, se recomienda encarecidamente que *no* sobrescriba la versión de NASM instalada en el sistema. En su lugar, puede instalarlo en / usr / local:

```
$ sudo su
<user's password entered to become root>
# cd /usr/local/bin
# cp <path/to/unzipped/nasm/files/nasm> ./
# exit
```

En este punto, NASM está en `/usr/local/bin` , pero no está en su ruta. Ahora debes agregar la siguiente línea al final de tu perfil:

```
$ echo 'export PATH=/usr/local/bin:$PATH' >> ~/.bash_profile
```

Esto añadirá `/usr/local/bin` a su ruta. La ejecución de `nasm -v` en el símbolo del sistema ahora debería mostrar la versión correcta y más nueva.

Linux x86 ejemplo de Hello World

Este es un programa básico de Hello World en el ensamblaje de NASM para Linux x86 de 32 bits, que utiliza llamadas del sistema directamente (sin ninguna función libc). Es mucho para asimilar, pero con el tiempo será comprensible. Las líneas que comienzan con un punto y coma (`;`) son comentarios.

Si aún no conoce la programación de sistemas Unix de bajo nivel, es posible que desee escribir funciones en asm y llamarlas desde programas C o C ++. Entonces puede preocuparse por aprender a manejar registros y memoria, sin aprender también la API de llamada de sistema POSIX y la ABI para usarla.

Esto hace dos llamadas al sistema: `write(2)` y `_exit(2)` (no la `exit(3)` envoltorio libc que vacía los buffers de stdio y así sucesivamente). (Técnicamente, `_exit()` llama a sys_exit_group, no a sys_exit, pero eso solo importa en un proceso de múltiples subprocesos). Consulte también `syscalls(2)` para obtener información sobre las llamadas al sistema en general, y la diferencia entre hacerlas directamente en lugar de usar libc. Funciones de envoltura.

En resumen, las llamadas al sistema se realizan colocando los argumentos en los registros apropiados y el número de llamada del sistema en `eax` , luego ejecutando una instrucción `int 0x80` . Consulte también ¿Cuáles son los valores de retorno de las llamadas del sistema en ensamblaje? para obtener una explicación más detallada de cómo se documenta la interfaz syscall de asm con la mayor parte de la sintaxis C.

Los números de llamada syscall para el ABI de 32 bits están en `/usr/include/i386-linux-gnu/asm/unistd_32.h` (el mismo contenido en `/usr/include/x86_64-linux-gnu/asm/unistd_32.h`).

`#include <sys/syscall.h>` finalmente incluirá el archivo correcto, por lo que podría ejecutar `echo '#include <sys/syscall.h>' | gcc -E - -dM | less` para ver las definiciones de macros (consulte esta respuesta para obtener más información sobre cómo encontrar constantes para asm en encabezados C)

```
section .text             ; Executable code goes in the .text section
global _start             ; The linker looks for this symbol to set the process entry point,
so execution start here
;;;a name followed by a colon defines a symbol.  The global _start directive modifies it so
it's a global symbol, not just one that we can CALL or JMP to from inside the asm.
;;; note that _start isn't really a "function".  You can't return from it, and the kernel
passes argc, argv, and env differently than main() would expect.
 _start:
    ;;; write(1, msg, len);
    ; Start by moving the arguments into registers, where the kernel will look for them
    mov     edx,len       ; 3rd arg goes in edx: buffer length
    mov     ecx,msg       ; 2nd arg goes in ecx: pointer to the buffer
    ;Set output to stdout (goes to your terminal, or wherever you redirect or pipe)
    mov     ebx,1         ; 1st arg goes in ebx: Unix file descriptor. 1 = stdout, which is
normally connected to the terminal.

    mov     eax,4         ; system call number (from SYS_write / __NR_write from unistd_32.h).
    int     0x80          ; generate an interrupt, activating the kernel's system-call
handling code.  64-bit code uses a different instruction, different registers, and different
call numbers.
    ;; eax = return value, all other registers unchanged.

    ;;;Second, exit the process.  There's nothing to return to, so we can't use a ret
instruction (like we could if this was main() or any function with a caller)
    ;;; If we don't exit, execution continues into whatever bytes are next in the memory page,
    ;;; typically leading to a segmentation fault because the padding 00 00 decodes to  add
[eax],al.
```

```
    ;;; _exit(0);
    xor     ebx,ebx       ; first arg = exit status = 0.  (will be truncated to 8 bits).
Zeroing registers is a special case on x86, and mov ebx,0 would be less efficient.
                      ;; leaving out the zeroing of ebx would mean we exit(1), i.e. with an
error status, since ebx still holds 1 from earlier.
    mov     eax,1         ; put __NR_exit into eax
    int     0x80          ;Execute the Linux function

section     .rodata       ; Section for read-only constants

             ;; msg is a label, and in this context doesn't need to be msg:.  It could be on a
separate line.
             ;; db = Data Bytes: assemble some literal bytes into the output file.
msg     db  'Hello, world!',0xa     ; ASCII string constant plus a newline (0x10)

             ;;  No terminating zero byte is needed, because we're using write(), which takes
a buffer + length instead of an implicit-length string.
             ;; To make this a C string that we could pass to puts or strlen, we'd need a
terminating 0 byte. (e.g. "...", 0x10, 0)

len     equ $ - msg       ; Define an assemble-time constant (not stored by itself in the
output file, but will appear as an immediate operand in insns that use it)
                          ; Calculate len = string length.  subtract the address of the start
                          ; of the string from the current position ($)
  ;; equivalently, we could have put a str_end: label after the string and done   len equ
str_end - str
```

En Linux, puedes guardar este archivo como `Hello.asm` y construir un ejecutable de 32 bits con estos comandos:

```
nasm -felf32 Hello.asm                  # assemble as 32-bit code.  Add -Worphan-labels -g -
Fdwarf  for debug symbols and warnings
gcc -nostdlib -m32 Hello.o -o Hello     # link without CRT startup code or libc, making a
static binary
```

Vea esta respuesta para obtener más detalles sobre cómo compilar ensamblados en ejecutables Linux de 32 o 64 bits estáticos o enlazados dinámicamente, para la sintaxis NASM / YASM o la sintaxis GNU AT&T con GNU `as` directivas. (Punto clave: asegúrese de usar `-m32` o equivalente al `-m32` código de 32 bits en un host de 64 bits, o tendrá problemas confusos en tiempo de ejecución).

Puede rastrear su ejecución con `strace` para ver las llamadas del sistema que hace:

```
$ strace ./Hello
execve("./Hello", ["./Hello"], [/* 72 vars */]) = 0
[ Process PID=4019 runs in 32 bit mode. ]
write(1, "Hello, world!\n", 14Hello, world!
)         = 14
_exit(0)                                = ?
+++ exited with 0 +++
```

La traza en stderr y la salida regular en stdout van al terminal aquí, por lo que interfieren en la línea con la llamada del sistema de `write` . Redirigir o rastrear a un archivo si te importa. Observe cómo esto nos permite ver fácilmente los valores de retorno de syscall sin tener que agregar

código para imprimirlos, y es incluso más fácil que usar un depurador regular (como gdb) para esto.

La versión x86-64 de este programa sería extremadamente similar, pasando los mismos argumentos a las mismas llamadas del sistema, solo en diferentes registros. Y usando la instrucción `syscall` lugar de `int 0x80` .

Lea Primeros pasos con Intel x86 Assembly Language & Microarchitecture en línea: https://riptutorial.com/es/x86/topic/1164/primeros-pasos-con-intel-x86-assembly-language--amp--microarchitecture

Capítulo 2: Convenciones de llamadas

Observaciones

Recursos

Resúmenes / comparaciones: la guía de convenciones de llamadas agradables de Agner Fog . Además, x86 ABIs (wikipedia) : convenciones de llamada para funciones, incluidas x86-64 Windows y System V (Linux).

- SystemV x86-64 ABI (estándar oficial) . Utilizado por todos los sistemas operativos excepto Windows. (Esta página wiki de github , actualizada por HJ Lu, tiene enlaces a 32 bits, 64 bits y x32. También contiene enlaces al foro oficial para mantenedores / contribuyentes de ABI). También tenga en cuenta que clang / gcc sign / zero extienden argumentos estrechos para 32 bits , aunque el ABI como está escrito no lo requiere. El código generado por el sonido depende de ello.
- SystemV 32bit (i386) ABI (estándar oficial) , utilizado por Linux y Unix. (versión antigua).
- OS X 32bit x86 convención de llamadas, con enlaces a los demás . La convención de llamadas de 64 bits es el Sistema V. El sitio de Apple solo se enlaza con un pdf de FreeBSD para eso.
- Windows x86-64 `__fastcall` llamando a la convención
- Windows `__vectorcall` : documenta las versiones de 32 bits y 64 bits
- Windows de 32 bits `__stdcall` : se utiliza para llamar a las funciones de la API de Win32. Esa página enlaza con los otros documentos de la convención de llamada (por ejemplo, `__cdecl`).
- ¿Por qué Windows64 utiliza una convención de llamada diferente de todos los demás sistemas operativos en x86-64? : alguna historia interesante, esp. para el SysV ABI donde los archivos de listas de correo son públicos y se remontan antes del lanzamiento del primer silicio de AMD.

Examples

Cdecl de 32 bits

cdecl es una convención de llamada de función de Windows de 32 bits que es *muy* similar a la convención de llamada utilizada en muchos sistemas operativos POSIX (documentado en el i386 System V ABI). Una de las diferencias está en devolver pequeñas estructuras.

Parámetros

Los parámetros se pasan a la pila, con el primer argumento en la dirección más baja de la pila en el momento de la llamada (empujado en último lugar, por lo que está justo arriba de la dirección de retorno al ingresar a la función). La persona que llama es responsable de quitar los parámetros de la pila después de la llamada.

Valor de retorno

Para los tipos de retorno escalar, el valor de retorno se coloca en EAX o EDX: EAX para enteros de 64 bits. Los tipos de punto flotante se devuelven en st0 (x87). La devolución de tipos más grandes como las estructuras se realiza por referencia, con un puntero pasado como primer parámetro implícito. (Este puntero se devuelve en EAX, por lo que la persona que llama no tiene que recordar lo que pasó).

Registros guardados y aplastados

EBee, EDI, ESI, EBP y ESP (y la configuración del modo de redondeo de FP / SSE) deben ser conservados por la persona que llama, de modo que la persona que llama pueda confiar en que esos registros no hayan sido modificados por una llamada.

Todos los demás registros (EAX, ECX, EDX, FLAGS (que no sean DF), x87 y registros vectoriales) pueden ser modificados libremente por el destinatario; Si una persona que llama desea conservar un valor antes y después de la llamada a la función, debe guardar el valor en otro lugar (como en uno de los registros guardados o en la pila).

Sistema V de 64 bits

Esta es la convención de llamada predeterminada para aplicaciones de 64 bits en muchos sistemas operativos POSIX.

Parámetros

Los primeros ocho parámetros escalares se pasan en (en orden) RDI, RSI, RDX, RCX, R8, R9, R10, R11. Los parámetros pasados los primeros ocho se colocan en la pila, con los parámetros anteriores más cerca de la parte superior de la pila. La persona que llama es responsable de quitar estos valores de la pila después de la llamada si ya no es necesario.

Valor de retorno

Para los tipos de retorno escalar, el valor de retorno se coloca en RAX. La devolución de tipos

más grandes, como las estructuras, se realiza cambiando conceptualmente la firma de la función para agregar un parámetro al principio de la lista de parámetros que es un puntero a una ubicación en la que colocar el valor de retorno.

Registros guardados y aplastados

El destinatario de la llamada conserva RBP, RBX y R12 – R15. Todos los demás registros pueden ser modificados por la persona que llama, y la persona que llama debe conservar el valor de un registro (por ejemplo, en la pila) si desea usar ese valor más adelante.

Llamada de 32 bits

stdcall se utiliza para llamadas de API de Windows de 32 bits.

Parámetros

Los parámetros se pasan a la pila, con el primer parámetro más cercano a la parte superior de la pila. La persona que llama sacará estos valores de la pila antes de regresar.

Valor de retorno

Los valores de retorno escalares se colocan en EAX.

Registros guardados y aplastados

EAX, ECX y EDX pueden ser modificados libremente por la persona que llama, y deben ser guardados por la persona que llama si lo desea. EBee, ESI, EDI y EBP deben ser guardados por el destinatario si se modifican y se restauran a sus valores originales en la devolución.

32 bits, cdecl - Tratar con enteros

Como parámetros (8, 16, 32 bits)

Los enteros de 8, 16, 32 bits siempre se pasan, en la pila, como valores de ancho completo de 32 bits [1] .
Ninguna extensión, firmada o puesta a cero, es necesaria.
La persona que llama solo usará la parte inferior de los valores de ancho completo.

```
//C prototype of the callee
void  __attribute__((cdecl)) foo(char a, short b, int c, long d);

foo(-1, 2, -3, 4);
```

```
;Call to foo in assembly

push DWORD 4              ;d, long is 32 bits, nothing special here
push DWORD 0ffffffdh      ;c, int is 32 bits, nothing special here
push DWORD 0badb0002h     ;b, short is 16 bits, higher WORD can be any value
push DWORD 0badbadffh     ;a, char is 8 bits, higher three bytes can be any value
call foo
add esp, 10h              ;Clean up the stack
```

Como parámetros (64 bits)

Los valores de 64 bits se pasan a la pila utilizando dos pulsaciones, respetando la convención littel endian [2] , empujando primero los 32 bits más altos y luego los más bajos.

```
//C prototype of the callee
void  __attribute__((cdecl)) foo(char a, short b, int c, long d);

foo(0x0123456789abcdefLL);

;Call to foo in assembly

push DWORD 89abcdefh        ;Higher DWORD of 0123456789abcdef
push DWORD 01234567h        ;Lower DWORD of 0123456789abcdef
call foo
add esp, 08h
```

Como valor de retorno

Los enteros de 8 bits se devuelven en `AL` , y finalmente se destruye todo el `eax` .
Los números enteros de 16 bits se devuelven en `AX` , eventualmente obstruyendo todo el `eax` .
Los enteros de 32 bits se devuelven en `EAX` .
Los números enteros de 64 bits se devuelven en `EDX:EAX` , donde `EAX` contiene los 32 bits inferiores y `EDX` los superiores.

```
//C
char foo() { return -1; }

;Assembly
mov al, 0ffh
ret

//C
unsigned short foo() { return 2; }

;Assembly
mov ax, 2
ret

//C
```

```
int foo() { return -3; }

;Assembly
mov eax, 0fffffffdh
ret

//C
int foo() { return 4; }

;Assembly
xor edx, edx            ;EDX = 0
mov eax, 4             ;EAX = 4
ret
```

---

[1] Esto mantiene la pila alineada en 4 bytes, el tamaño natural de la palabra. Además, una CPU x86 solo puede enviar 2 o 4 bytes cuando no está en modo largo.

[2] DWORD inferior en la dirección inferior

32 bits, cdecl - Tratar con punto flotante

Como parámetros (float, doble)

Los flotadores tienen un tamaño de 32 bits, se pasan naturalmente en la pila.
Los dobles tienen un tamaño de 64 bits, se pasan, en la pila, respetando la convención Little Endian [1] , empujando primero los 32 bits superiores y los inferiores.

```
//C prototype of callee
double foo(double a, float b);

foo(3.1457, 0.241);

;Assembly call

;3.1457 is 0x40092A64C2F837B5ULL
;0.241 is 0x3e76c8b4

push DWORD  3e76c8b4h        ;b, is 32 bits, nothing special here
push DWORD 0c2f837b5h        ;a, is 64 bits, Higher part of 3.1457
push DWORD  40092a64h        ;a, is 64 bits, Lower part of 3.1457
call foo
add esp, 0ch

;Call, using the FPU
;ST(0) = a, ST(1) = b
sub esp, 0ch
fstp QWORD PTR [esp]        ;Storing a as a QWORD on the stack
fstp DWORD PTR [esp+08h]    ;Storing b as a DWORD on the stack
call foo
add esp, 0ch
```

Como parámetros (doble largo)

Los dobles largos son 80 bits [2] de ancho, mientras que en la pila se puede almacenar un TBYTE con dos empujes de 32 bits y un empuje de 16 bits (para 4 + 4 + 2 = 10), para mantener la pila alineada en 4 bytes, termina ocupando 12 bytes, utilizando así tres empujes de 32 bits. Respetando la convención de Little Endian, los bits 79-64 son empujados primero [3] , luego los bits 63-32 seguidos por los bits 31-0.

```
//C prototype of the callee
void  __attribute__((cdecl)) foo(long double a);

foo(3.1457);

;Call to foo in assembly
;3.1457 is 0x4000c9532617c1bda800

push DWORD 4000h        ;Bits 79-64, as 32 bits push
push DWORD 0c9532617h        ;Bits 63-32
push DWORD 0c1bda800h        ;Bits 31-0
call foo
add esp, 0ch

;Call to foo, using the FPU
;ST(0) = a

sub esp, 0ch
fstp TBYTE PTR [esp]        ;Store a as ten byte on the stack
call foo
add esp, 0ch
```

Como valor de retorno

Los valores de punto flotante, cualquiera que sea su tamaño, se devuelven en `ST(0)` [4] .

```
//C
float one() { return 1; }

;Assembly
fld1            ;ST(0) = 1
ret

//C
double zero() { return 0; }

;Assembly
fldz            ;ST(0) = 0
ret

//C
long double pi() { return PI; }

;Assembly
fldpi            ;ST(0) = PI
ret
```

---

[1] DWORD inferior en la dirección inferior.

[2] Conocido como TBYTE, de Diez Bytes.

[3] Usando un empuje de ancho completo con cualquier extensión, no se usa la PALABRA más alta.

[4] Que es ancho TBYE, tenga en cuenta que, a diferencia de los enteros, los PF siempre se devuelven con la mayor precisión que se requiere.

Windows de 64 bits

Parámetros

Los primeros 4 parámetros se pasan en (en orden) RCX, RDX, R8 y R9. XMM0 a XMM3 se utilizan para pasar parámetros de punto flotante.

Cualquier otro parámetro se pasa en la pila.

Los parámetros mayores de 64 bits se pasan por dirección.

Espacio de derrame

Incluso si la función utiliza menos de 4 parámetros, la persona que llama siempre proporciona espacio para 4 parámetros de tamaño QWORD en la pila. El destinatario de la llamada es libre de usarlos para cualquier propósito, es común copiar los parámetros allí si otra llamada los derrama.

Valor de retorno

Para los tipos de retorno escalar, el valor de retorno se coloca en RAX. Si el tipo de retorno es mayor que 64 bits (por ejemplo, para estructuras), RAX es un puntero a eso.

Registros guardados y aplastados

Todos los registros utilizados en el paso de parámetros (RCX, RDX, R8, R9 y XMM0 a XMM3), RAX, R10, R11, XMM4 y XMM5 pueden ser derramados por el destinatario. Todos los demás registros deben ser conservados por la persona que llama (por ejemplo, en la pila).

Alineación de la pila

La pila debe mantenerse alineada de 16 bytes. Dado que la instrucción "llamada" empuja una dirección de retorno de 8 bytes, esto significa que cada función no hoja va a ajustar la pila en un valor de la forma 16n + 8 para restaurar la alineación de 16 bytes.
Es el trabajo de las personas que llaman limpiar la pila después de una llamada.

Fuente: La historia de las convenciones de llamadas, parte 5: amd64 Raymond Chen

32 bits, cdecl - Manejo de estructuras

Relleno

Recuerde, los miembros de una estructura generalmente se rellenan para asegurarse de que están alineados en su límite natural:

```
struct t
{
    int a, b, c, d;     // a is at offset 0, b at 4, c at 8, d at 0ch
    char e;             // e is at 10h
    short f;            // f is at 12h (naturally aligned)
    long g;             // g is at 14h
    char h;             // h is at 18h
    long i;             // i is at 1ch (naturally aligned)
};
```

Como parámetros (pasar por referencia)

Cuando se pasa por referencia, un puntero a la estructura en la memoria se pasa como el primer argumento en la pila. Esto es equivalente a pasar un valor entero de tamaño natural (32 bits); ver cdecl de 32 bits para más detalles.

Como parámetros (pasar por valor)

Cuando se pasa por valor, las estructuras se copian por completo en la pila, respetando el diseño de la memoria original (*es decir* , el primer miembro estará en la dirección inferior).

```
int __attribute__((cdecl)) foo(struct t a);

struct t s = {0, -1, 2, -3, -4, 5, -6, 7, -8};
foo(s);
```

```
; Assembly call

push DWORD 0ffffffh    ; i (-8)
push DWORD 0badbad07h    ; h (7), pushed as DWORD to naturally align i, upper bytes can be
garbage
push DWORD 0fffffffah    ; g (-6)
push WORD 5              ; f (5)
push WORD 033fch         ; e (-4), pushed as WORD to naturally align f, upper byte can be
garbage
push DWORD 0fffffffdh    ; d (-3)
push DWORD 2             ; c (2)
push DWORD 0ffffffffh    ; b (-1)
push DWORD 0             ; a (0)
call foo
```

```
add esp, 20h
```

Como valor de retorno

A menos que sean triviales [1] , las estructuras se copian en un búfer proporcionado por la persona que llama antes de regresar. Esto es equivalente a tener un primer parámetro oculto `struct S *retval` (donde `struct S` es el tipo de la estructura).

La función debe devolver con este puntero al valor de retorno en `eax` ; Se permite que la persona que llama dependa de que `eax` mantenga el puntero en el valor de retorno, que presionó justo antes de la `call` .

```
struct S
{
    unsigned char a, b, c;
};

struct S foo();         // compiled as struct S* foo(struct S* _out)
```

El parámetro oculto no se agrega al conteo de parámetros para la limpieza de la pila, ya que debe ser manejado por el destinatario.

```
sub esp, 04h        ; allocate space for the struct

; call to foo
push esp            ; pointer to the output buffer
call foo
add esp, 00h        ; still as no parameters have been passed
```

En el ejemplo anterior, la estructura se guardará en la parte superior de la pila.

```
struct S foo()
{
    struct S s;
    s.a = 1; s.b = -2; s.c = 3;
    return s;
}
```

```
; Assembly code
push ebx
mov eax, DWORD PTR [esp+08h]   ; access hidden parameter, it is a pointer to a buffer
mov ebx, 03fe01h               ; struct value, can be held in a register
mov DWORD [eax], ebx           ; copy the structure into the output buffer
pop ebx
ret 04h                        ; remove the hidden parameter from the stack
                               ; EAX = pointer to the output buffer
```

---

[1] Una estructura "trivial" es aquella que contiene solo un miembro de un tipo sin estructura, sin matriz (de hasta 32 bits de tamaño). Para tales estructuras, el valor de ese miembro simplemente se devuelve en el registro `eax` . (Este comportamiento se ha observado con GCC que apunta a

Linux)

La versión de cdecl para Windows es diferente de la convención de llamada ABI de System V: se permite que una estructura "trivial" contenga hasta dos miembros de un tipo sin estructura, sin matriz (de hasta 32 bits de tamaño). Estos valores se devuelven en `eax` y `edx` , como lo sería un entero de 64 bits. (Este comportamiento se ha observado para MSVC y Clang para Win32).

Lea Convenciones de llamadas en línea: https://riptutorial.com/es/x86/topic/3261/convenciones-de-llamadas

Capítulo 3: Convertir cadenas decimales en enteros

Observaciones

Convertir cadenas en números enteros es una de las tareas comunes.

Aquí mostraremos cómo convertir cadenas decimales en enteros.

El código de Psuedo para hacer esto es:

```
function string_to_integer(str):
    result = 0
    for (each characters in str, left to right):
        result = result * 10
        add ((code of the character) - (code of character 0)) to result
    return result
```

Tratar con cadenas hexadecimales es un poco más difícil porque los códigos de caracteres generalmente no son continuos cuando se trata de múltiples tipos de caracteres como dígitos (0-9) y alfabetos (af y AF). Los códigos de caracteres suelen ser continuos cuando se trata de un solo tipo de caracteres (trataremos con los dígitos aquí), por lo que trataremos solo con entornos en los que los códigos de caracteres para dígitos son continuos.

Examples

Asamblea IA-32, GAS, convención de cdecl

```
# make this routine available outside this translation unit
.globl string_to_integer

string_to_integer:
    # function prologue
    push %ebp
    mov %esp, %ebp
    push %esi

    # initialize result (%eax) to zero
    xor %eax, %eax
    # fetch pointer to the string
    mov 8(%ebp), %esi

    # clear high bits of %ecx to be used in addition
    xor %ecx, %ecx
    # do the conversion
string_to_integer_loop:
    # fetch a character
    mov (%esi), %cl
    # exit loop when hit to NUL character
    test %cl, %cl
```

```
    jz string_to_integer_loop_end
    # multiply the result by 10
    mov $10, %edx
    mul %edx
    # convert the character to number and add it
    sub $'0', %cl
    add %ecx, %eax
    # proceed to next character
    inc %esi
    jmp string_to_integer_loop
string_to_integer_loop_end:

    # function epilogue
    pop %esi
    leave
    ret
```

Este código de estilo GAS convertirá la cadena decimal dada como primer argumento, que se empuja en la pila antes de llamar a esta función, a entero y la devolverá a través de `%eax` . El valor de `%esi` se guarda porque es un registro de guardado de llamada y se utiliza.

Los caracteres de desbordamiento / ajuste y los caracteres no válidos no se verifican para simplificar el código.

En C, este código se puede usar de esta manera (suponiendo que el `unsigned int` y los punteros tengan una longitud de 4 bytes):

```
#include <stdio.h>

unsigned int string_to_integer(const char* str);

int main(void) {
    const char* testcases[] = {
        "0",
        "1",
        "10",
        "12345",
        "1234567890",
        NULL
    };
    const char** data;
    for (data = testcases; *data != NULL; data++) {
        printf("string_to_integer(%s) = %u\n", *data, string_to_integer(*data));
    }
    return 0;
}
```

Nota: en algunos entornos, dos `string_to_integer` en el código de ensamblaje se deben cambiar a `_string_to_integer` (agregar guión bajo) para que funcione con el código C.

MS-DOS, función TASM / MASM para leer un entero sin signo de 16 bits

Lea un entero sin signo de 16 bits de la entrada.

Esta función utiliza el servicio de interrupción Int 21 / AH = 0Ah para leer una cadena con búfer. El uso de una cadena de búfer permite al usuario revisar lo que ha escrito antes de pasarlo al programa para su procesamiento.
Se leen hasta seis dígitos (ya que 65535 = 2^{16} - 1 tiene seis dígitos).

Además de realizar la conversión estándar de *número* a *número,* esta función también detecta entradas y desbordamientos no válidos (número demasiado grande para encajar en 16 bits).

Valores de retorno

La función devuelve el número leído en `AX` . Las banderas `ZF` , `CF` , `OF` indican si la operación se completó con éxito o no y por qué.

Error	HACHA	ZF	CF	DE
Ninguna	El entero de 16 bits.	Conjunto	No establecido	No establecido
Entrada inválida	El número parcialmente convertido, hasta el último dígito válido encontrado	No establecido	Conjunto	No establecido
Rebosar	7fffh	No establecido	Conjunto	Conjunto

El `ZF` se puede usar para separar rápidamente entre entradas válidas y no válidas.

Uso

```
call read_uint16
jo _handle_overflow            ;Number too big (Optional, the test below will do)
jnz _handle_invalid            ;Number format is invalid

;Here AX is the number read
```

Código

```
;Returns:
  ;
  ;If the number is correctly converted:
  ;   ZF = 1, CF = 0, OF = 0
  ;   AX = number
  ;
  ;If the user input an invalid digit:
  ;   ZF = 0, CF = 1, OF = 0
  ;   AX = Partially converted number
  ;
  ;If the user input a number too big
  ;   ZF = 0, CF = 1, OF = 1
```

```
;   AX = 07fffh
;
;ZF/CF can be used to discriminate valid vs invalid inputs
;OF can be used to discrimate the invalid inputs (overflow vs invalid digit)
;
read_uint16:
  push bp
  mov bp, sp

  ;This code is an example in Stack Overflow Documentation project.
  ;x86/Converting Decimal strings to integers


  ;Create the buffer structure on the stack

  sub sp, 06h                ;Reserve 6 byte on the stack (5 + CR)
  push 0006h                 ;Header

  push ds
  push bx
  push cx
  push dx

  ;Set DS = SS

  mov ax, ss
  mov ds, ax


  ;Call Int 21/AH=0A

  lea dx, [bp-08h]            ;Address of the buffer structure
  mov ah, 0ah
  int 21h

  ;Start converting

  lea si, [bp-06h]
  xor ax, ax
  mov bx, 10
  xor cx, cx

 _r_ui16_convert:

  ;Get current char

  mov cl, BYTE PTR [si]
  inc si

  ;Check if end of string

  cmp cl, CR_CHAR
  je _r_ui16_end                     ;ZF = 1, CF = 0, OF = 0

  ;Convert char into digit and check

  sub cl, '0'
  jb _r_ui16_carry_end               ;ZF = 0, CF = 1, OF = X -> 0
  cmp cl, 9
  ja _r_ui16_carry_end               ;ZF = 0, CF = 0 -> 1, OF = X -> 0
```

```
 ;Update the partial result (taking care of overflow)

 ;AX = AX * 10
 mul bx

 ;DX:AX = DX:AX + CX
 add ax, cx
 adc dx, 0

 test dx, dx
jz _r_ui16_convert            ;No overflow

 ;set OF and CF
 mov ax, 8000h
 dec ax
 stc

jmp _r_ui16_end                      ;ZF = 0, CF = 1, OF = 1

_r_ui16_carry_end:

 or bl, 1                ;Clear OF and ZF
 stc                     ;Set carry

 ;ZF = 0, CF = 1, OF = 0

_r_ui16_end:
 ;Don't mess with flags hereafter!

 pop dx
 pop cx
 pop bx
 pop ds

 mov sp, bp

 pop bp
 ret

 CR_CHAR EQU 0dh
```

Portabilidad NASM

Para transferir el código a NASM, elimine la palabra clave `PTR` de los accesos a la memoria (por ejemplo, `mov cl, BYTE PTR [si]` convierte en `mov cl, BYTE [si]`)

MS-DOS, función TASM / MASM para imprimir un número de 16 bits en binario, cuaternario, octal y hexadecimal

Imprima un número en binario, cuaternario, octal, hexadecimal y una potencia general de dos

Todas las bases que son una potencia de dos, como las bases binaria (2^1), cuaternaria (2^2),

octal (2 [3]), hexadecimal (2 [4]), tienen un número entero de bits por dígito [1] .
Por lo tanto, para recuperar cada dígito [2] de un número, simplemente dividimos el número de grupos de introducción de *n* bits a partir de la LSb (derecha).
Por ejemplo, para la base cuaternaria, dividimos un número de 16 bits en grupos de dos bits. Hay 8 de tales grupos.
No toda la potencia de dos bases tiene un número integral de grupos que se ajusta a 16 bits; por ejemplo, la base octal tiene 5 grupos de 3 bits que representan 3 · 5 = 15 bits de 16, dejando un grupo parcial de 1 bit [3] .

El algoritmo es simple, aislamos cada grupo con un cambio seguido de una operación *AND* .
Este procedimiento funciona para todos los tamaños de los grupos o, en otras palabras, para cualquier potencia base de dos.

Para mostrar los dígitos en el orden correcto, la función comienza al aislar el grupo más significativo (el que está más a la izquierda), por lo que es importante saber: a) cuántos bits *D es* un grupo y b) la posición del bit *S* donde está más a la izquierda comienza el grupo.
Estos valores están precomputados y almacenados en constantes cuidadosamente elaboradas.

Parámetros

Los parámetros deben ser empujados en la pila.
Cada uno es de 16 bits de ancho.
Se muestran en orden de empuje.

Parámetro	Descripción
norte	El número a convertir
Base	La base a usar expresada usando las constantes `BASE2` , `BASE4` , `BASE8` y `BASE16`
Imprimir ceros iniciales	Si *cero* no se imprimen ceros no significativos, de lo contrario lo son. El número 0 se imprime como "0" aunque

Uso

```
push 241
push BASE16
push 0
call print_pow2              ;Prints f1

push 241
push BASE16
push 1
call print_pow2              ;Prints 00f1

push 241
push BASE2
push 0
```

```
call print_pow2             ;Prints 11110001
```

Nota para los usuarios de TASM : si coloca las constantes definidas con `EQU` después del código que las utiliza, habilite *el paso múltiple* con la *marca* `/m` de *TASM* o obtendrá una *anulación de las necesidades de Reenvío* .

Código

```
;Parameters (in order of push):
;
;number
;base (Use constants below)
;print leading zeros
print_pow2:
 push bp
 mov bp, sp

 push ax
 push bx
 push cx
 push dx
 push si
 push di

 ;Get parameters into the registers

 ;SI = Number (left) to convert
 ;CH = Amount of bits to shift for each digit (D)
 ;CL = Amount od bits to shift the number (S)
 ;BX = Bit mask for a digit

 mov si, WORD PTR [bp+08h]
 mov cx, WORD PTR [bp+06h]            ;CL = D, CH = S

 ;Computes BX = (1 << D)-1

 mov bx, 1
 shl bx, cl
 dec bx

 xchg cl, ch              ;CL = S, CH = D

_pp2_convert:
 mov di, si
 shr di, cl
 and di, bx         ;DI = Current digit

 or WORD PTR [bp+04h], di             ;If digit is non zero, [bp+04h] will become non zero
                     ;If [bp+04h] was non zero, result is non zero
 jnz _pp2_print                       ;Simply put, if the result is non zero, we must print
the digit


 ;Here we have a non significant zero
 ;We should skip it BUT only if it is not the last digit (0 should be printed as "0" not
 ;an empty string)

 test cl, cl
```

```
 jnz _pp_continue


_pp2_print:
 ;Convert digit to digital and print it


 mov dl, BYTE PTR [DIGITS + di]
 mov ah, 02h
 int 21h


_pp_continue:
 ;Remove digit from the number

 sub cl, ch
jnc _pp2_convert

 pop di
 pop si
 pop dx
 pop cx
 pop bx
 pop ax

 pop bp
 ret 06h
```

Datos

```
This data must be put in the data segment, the one reached by `DS`.

DIGITS    db    "0123456789abcdef"

;Format for each WORD is S D where S and D are bytes (S the higher one)
;D = Bits per digit  --> log2(BASE)
;S = Initial shift count --> D*[ceil(16/D)-1]

BASE2    EQU    0f01h
BASE4    EQU    0e02h
BASE8    EQU    0f03h
BASE16    EQU    0c04h
```

Portabilidad NASM

Para trasladar el código a NASM, elimine la palabra clave PTR de los accesos a la memoria (por ejemplo, `mov si, WORD PTR [bp+08h]` convierte en `mov si, WORD PTR [bp+08h]`)

Extendiendo la función

La función se puede extender fácilmente a cualquier base hasta 2^{255} , aunque cada base por encima de 2^{16} imprimirá el mismo número, ya que el número es solo de 16 bits.

Para añadir una base:

1. Defina una nueva constante `BASEx` donde *x* es 2^n .
   El byte inferior, llamado *D* , es *D* = *n* .
   El byte superior, llamado *S* , es la posición, en bits, del grupo superior. Se puede calcular como $S = n \cdot (\lceil 16 / n \rceil - 1)$.
2. Agregue los dígitos necesarios a la cadena `DIGITS` .

Ejemplo: añadiendo base 32

Tenemos *D* = 5 y *S* = 15, por lo que definimos `BASE32 EQU 0f05h` .
Luego agregamos dieciséis dígitos más: `DIGITS db "0123456789abcdefghijklmnopqrstuv"` .

Como debe quedar claro, los dígitos se pueden cambiar editando la cadena `DIGITS` .

---

[1] Si *B* es una base, entonces tiene *B* dígitos por definición. El número de bits por dígito es, por lo tanto, $log_2(B)$. Para la potencia de dos bases, esto se simplifica al $registro_2(2^n) = n$, que es un entero por definición.

[2] En este contexto, se supone implícitamente que la base en consideración es una potencia de dos base 2^n .

[3] Para que una base $B = 2^n$ tenga un número entero de grupos de bits, debe ser que $n \mid 16$ (*n* divide 16). Dado que el único factor en 16 es 2, debe ser que *n* sea en sí mismo una potencia de dos. Entonces *B* tiene la forma 2^{2^k} o, de manera equivalente, $log_2(log_2(B))$ debe ser un número entero.

MS-DOS, TASM / MASM, función para imprimir un número de 16 bits en decimal

Imprima un número sin firmar de 16 bits en decimal

El servicio de interrupción Int 21 / AH = 02h se utiliza para imprimir los dígitos.
La conversión estándar de *número* a *número* se realiza con la instrucción `div` , el dividendo es inicialmente la potencia más alta de diez bits de 16 bits (10^4) y se reduce a potencias más bajas en cada iteración.

Parámetros

Los parámetros se muestran en orden de empuje.
Cada uno es de 16 bits.

Parámetro	Descripción
número	El número sin signo de 16 bits para imprimir en decimal
mostrar ceros iniciales	Si 0 no se imprimen ceros no significativos, si no lo son. El número 0 siempre se imprime como "0"

Uso

```
push 241
push 0
call print_dec          ;prints 241

push 56
push 1
call print_dec          ;prints 00056

push 0
push 0
call print_dec          ;prints 0
```

Código

```
;Parameters (in order of push):
;
;number
;Show leading zeros
print_dec:
 push bp
 mov bp, sp

 push ax
 push bx
 push cx
 push dx

 ;Set up registers:
 ;AX = Number left to print
 ;BX = Power of ten to extract the current digit
 ;DX = Scratch/Needed for DIV
 ;CX = Scratch

 mov ax, WORD PTR [bp+06h]
 mov bx, 10000d
 xor dx, dx

_pd_convert:
 div bx                         ;DX = Number without highmost digit, AX = Highmost digit
 mov cx, dx                     ;Number left to print

 ;If digit is non zero or param for leading zeros is non zero
 ;print the digit
 or WORD PTR [bp+04h], ax
 jnz _pd_print

 ;If both are zeros, make sure to show at least one digit so that 0 prints as "0"
 cmp bx, 1
 jne _pd_continue

_pd_print:

 ;Print digit in AL

 mov dl, al
```

```
 add dl, '0'
 mov ah, 02h
 int 21h

_pd_continue:
 ;BX = BX/10
 ;DX = 0

 mov ax, bx
 xor dx, dx
 mov bx, 10d
 div bx
 mov bx, ax

 ;Put what's left of the number in AX again and repeat...
 mov ax, cx

 ;...Until the divisor is zero
 test bx, bx
jnz _pd_convert

 pop dx
 pop cx
 pop bx
 pop ax

 pop bp
 ret 04h
```

Portabilidad NASM

Para transferir el código a NASM, elimine la palabra clave `PTR` de los accesos a la memoria (por ejemplo, `mov ax, WORD PTR [bp+06h]` convierte en `mov ax, WORD [bp+06h]`)

Lea Convertir cadenas decimales en enteros en línea:
https://riptutorial.com/es/x86/topic/3273/convertir-cadenas-decimales-en-enteros

Capítulo 4: Ensambladores

Examples

Microsoft Assembler - MASM

Dado que el 8086/8088 se usó en la PC de IBM, y el sistema operativo de Microsoft fue el más utilizado, el ensamblador MASM de Microsoft fue el estándar de facto durante muchos años. Siguió de cerca la sintaxis de Intel, pero permitió una sintaxis conveniente pero "suelta" que (en retrospectiva) solo causó confusión y errores en el código.

Un ejemplo perfecto es el siguiente:

```
MaxSize      EQU     16          ; Define a constant
Symbol       DW      0x1234      ; Define a 16-bit WORD called Symbol to hold 0x1234

             MOV     AX, 10      ; AX now holds 10
             MOV     BX, MaxSize ; BX now holds 16
             MOV     CX, Symbol  ; ????
```

¿La última instrucción `MOV` pone el *contenido* de `Symbol` en `CX` , o la *dirección* de `Symbol` en `CX` ? ¿ `CX` termina con `0x1234` o `0x0102` (o lo que sea)? Resulta que `CX` termina con `0x1234` - si quieres la dirección, necesitas usar el especificador `OFFSET`

```
             MOV     AX, [Symbol]      ; Contents of Symbol
             MOV     CX, OFFSET Symbol ; Address of Symbol
```

Ensamblador de Intel

Intel escribió la especificación del lenguaje ensamblador 8086, un derivado de los procesadores 8080, 8008 y 4004 anteriores. Como tal, el ensamblador que escribieron siguió su propia sintaxis con precisión. Sin embargo, este ensamblador no se usó mucho.

Intel definió sus códigos de operación para tener cero, uno o dos operandos. Las instrucciones de dos operandos se definieron para estar en el `dest` , orden de `source` , que era diferente de otros ensambladores en ese momento. Pero algunas instrucciones utilizaron registros implícitos como operandos, solo tenía que saber qué eran. Intel también usó el concepto de "prefijo" códigos de operación: un código de operación afectaría la siguiente instrucción.

```
; Zero operand examples
NOP             ; No parameters
CBW             ; Convert byte in AL into word in AX
MOVSB           ; Move byte pointed to by DS:SI to byte pointed to by ES:DI
                ; SI and DI are incremented or decremented according to D bit

; Prefix examples
REP   MOVSB     ; Move number of bytes in CX from DS:SI to ES:DI
                ; SI and DI are incremented or decremented according to D bit
```

```
; One operand examples
NOT      AX     ; Replace AX with its one's complement
MUL      CX     ; Multiply AX by CX and put 32-bit result in DX:AX

; Two operand examples
MOV      AL, [0x1234] ; Copy the contents of memory location DS:0x1234 into AL register
```

Intel también rompió una convención utilizada por otros ensambladores: para cada código de operación, se inventó una mnemónica diferente. Esto requirió nombres sutil o claramente diferentes para operaciones similares: por ejemplo, `LDM` para "Cargar desde la memoria" y `LDI` para "Cargar inmediatamente". Intel usó el único `MOV` mnemotécnico, y esperaba que el ensamblador resolviera qué código de operación utilizar desde el contexto. Eso causó muchos escollos y errores para los programadores en el futuro cuando el ensamblador no pudo intuir lo que el programador realmente quería ...

Ensamblador AT&T - como

Aunque el 8086 fue el más utilizado en las PC de IBM junto con Microsoft, hubo una cantidad de otras computadoras y sistemas operativos que también lo utilizaron: en particular, Unix. Ese era un producto de AT&T, y ya tenía Unix ejecutándose en otras arquitecturas. Esas arquitecturas utilizaban una sintaxis de ensamblaje más convencional, especialmente que las instrucciones de dos operandos las especificaban en la `source` , orden de `dest` .

Así que las convenciones de ensambladores de AT&T anularon las convenciones dictadas por Intel, y se introdujo un dialecto completamente nuevo para el rango x86:

- Los nombres de los registros fueron prefijados por `%` :
  `%al` , `%bx` etc.
- Los valores inmediatos fueron preferidos por `$` :
  `$4`
- Los operandos estaban en `source` , orden de `dest`
- Opcodes incluyó sus tamaños de operandos:
  `movw $4, %ax ; Move word 4 into AX`

Turbo Assembler de Borland - TASM

Borland comenzó con un compilador de Pascal que llamaron "Turbo Pascal". Esto fue seguido por compiladores para otros idiomas: C / C ++, Prolog y Fortran. También produjeron un ensamblador llamado "Turbo Assembler", que, siguiendo la convención de nomenclatura de Microsoft, llamaron "TASM".

TASM intentó solucionar algunos de los problemas de escritura de código utilizando MASM (ver más arriba), al proporcionar una interpretación más estricta del código fuente en un modo `IDEAL` específico. Por defecto, asumió el modo `MASM` , por lo que pudo ensamblar la fuente MASM directamente, pero luego Borland descubrió que tenían que ser compatibles bug-for-bug con las idiosincrasias más "extravagantes" de MASM, así que también agregaron un modo `QUIRKS` .

Dado que TASM era (mucho) más barato que MASM, tenía una gran base de usuarios, pero no

mucha gente usaba el modo IDEAL, a pesar de sus ventajas promocionadas.

Ensamblador GNU - gas

Cuando el proyecto GNU necesitaba un ensamblador para la familia x86, utilizaban la versión de AT&T (y su sintaxis) asociada con Unix en lugar de la versión de Intel / Microsoft.

Ensamblador Netwide - NASM

NASM es, con diferencia, el ensamblador más adaptado para la arquitectura x86: está disponible para prácticamente todos los sistemas operativos basados en x86 (incluso se incluye con MacOS) y está disponible como ensamblador multiplataforma en otras plataformas.

Este ensamblador utiliza la sintaxis de Intel, pero es diferente de los demás porque se centra en gran medida en su propio lenguaje "macro", lo que permite al programador crear expresiones más complejas utilizando definiciones más simples, lo que permite crear nuevas "instrucciones".

Desafortunadamente, esta poderosa característica tiene un costo: el tipo de datos interfiere con las instrucciones generalizadas, por lo que no se aplica la tipificación de datos.

```
response:    db       'Y'     ; Character that user typed

             cmp      response, 'N' ; *** Error! Unknown size!
             cmp byte response, 'N' ; That's better!
             cmp      response, ax  ; No error!
```

Sin embargo, NASM introdujo una característica de la que otros carecían: nombres de símbolos en el ámbito. Cuando define un símbolo en otros ensambladores, ese nombre está disponible en el resto del código, pero eso "usa" ese nombre, "contaminando" el espacio de nombres global con símbolos.

Por ejemplo (usando la sintaxis NASM):

```
       STRUC     Point
X      resw      1
Y      resw      1
       ENDSTRUC
```

Después de esta definición, X e Y se definen para siempre. Para evitar "usar" los nombres `X` e `Y`, necesitaba usar nombres más definidos:

```
       STRUC     Point
Pt_X   resw      1
Pt_Y   resw      1
       ENDSTRUC
```

Pero NASM ofrece una alternativa. Al aprovechar su concepto de "variable local", puede definir campos de estructura que requieren que nombre la estructura contenedora en referencias futuras:

```
       STRUC      Point
```

```
.X      resw        1
.Y      resw        1
        ENDSTRUC

Cursor  ISTRUC      Point
        ENDISTRUC

        mov         ax,[Cursor+Point.X]
        mov         dx,[Cursor+Point.Y]
```

Desafortunadamente, debido a que NASM no realiza un seguimiento de los tipos, no puede usar la sintaxis más natural:

```
        mov         ax,[Cursor.X]
        mov         dx,[Cursor.Y]
```

Sin embargo, otro ensamblador - YASM

YASM es una reescritura completa de NASM, pero es compatible con las sintaxis de Intel y AT&T.

Lea Ensambladores en línea: https://riptutorial.com/es/x86/topic/2403/ensambladores

Capítulo 5: Flujo de control

Examples

Saltos incondicionales

```
jmp a_label                     ;Jump to a_label
jmp bx                          ;Jump to address in BX
jmp WORD [aPointer]             ;Jump to address in aPointer
jmp 7c0h:0000h                  ;Jump to segment 7c0h and offset 0000h
jmp FAR WORD [aFarPointer]      ;Jump to segment:offset in aFarPointer
```

Relativo a saltos cercanos.

`jmp a_label` es:

- **cerca**
  Solo especifica la parte de desplazamiento de la *dirección lógica* de destino. Se supone que el segmento es `CS` .
- **relativo**
  La instrucción semántica es jump *rel* bytes adelante [1] de la siguiente dirección de instrucción o `IP = IP + rel` .

La instrucción se codifica como `EB <rel8>` o `EB <rel16/32>` , y el ensamblador toma la forma más adecuada, por lo general prefiere una más corta.
La anulación por ensamblador es posible, por ejemplo con NASM `jmp SHORT a_label` , `jmp WORD a_label` y `jmp DWORD a_label` generan las tres formas posibles.

Absolutos saltos indirectos cercanos.

`jmp bx` y `jmp WORD [aPointer]` son:

- **cerca**
  Solo especifican la parte de desplazamiento de la dirección lógica de destino. Se supone que el segmento es `CS` .
- **absoluto indirecto**
  La semántica de las instrucciones es saltar a la dirección en *reg* o *mem* o `IP = reg` , `IP = mem` .

La instrucción se codifica como `FF /4` , para la memoria indirecta, el tamaño del operando se determina como para cualquier otro acceso a la memoria.

Saltos lejanos absolutos

`jmp 7c0h:0000h` es:

- **lejos**
  Especifica ambas partes de la dirección *lógica* : el segmento y el desplazamiento.
- **absoluto** La semántica de la instrucción es saltar al *segmento de* dirección *: desplazamiento* o `CS = segment, IP = offset` .

La instrucción se codifica como `EA <imm32/48>` según el tamaño del código.
Es posible elegir entre las dos formas en algún ensamblador, por ejemplo con NASM `jmp 7c0h: WORD 0000h` y `jmp 7c0h: DWORD 0000h` genera la primera y la segunda forma.

Saltos lejanos indirectos absolutos

`jmp FAR WORD [aFarPointer]` es:

- **Lejos** Especifica ambas partes de la dirección *lógica* : el segmento y el desplazamiento.
- **Absoluto indirecto** La semántica de la instrucción es saltar al *segmento: desplazamiento* almacenado en *mem* [2] o `CS = mem[23:16/32], IP = [15/31:0]` .

La instrucción se codifica como `FF /5` , el tamaño del operando puede ser controlador con los especificadores de tamaño.
En NASM, un poco no intuitivo, son `jmp FAR WORD [aFarPointer]` para un operando *16:16* y `jmp FAR DWORD [aFarPointer]` para un operando *16:32* .

Saltos faltantes

- **casi absoluto**
  Puede ser emulado con un salto casi indirecto.

```
mov bx, target            ;BX = absolute address of target
jmp bx
```

- **pariente lejano**
  No tiene sentido o es demasiado estrecho de uso de todos modos.

---

[1] Dos complementos se utilizan para especificar un desplazamiento firmado y, por lo tanto, saltar hacia atrás.
[2], que puede ser un *seg16: off16* o *seg16: off32* , de tamaños *16:16* y *16:32* .

Condiciones de prueba

Para utilizar un salto condicional se debe probar una condición. **La prueba de una condición** aquí se refiere solo al acto de verificar las banderas, el salto real se describe en Saltos condicionales .

x86 prueba las condiciones basándose en el registro EFLAGS, que contiene un conjunto de indicadores que cada instrucción puede establecer potencialmente.

Las instrucciones aritméticas, como `sub` o `add` , y las instrucciones lógicas, como `xor` y `and` , obviamente, "establecen las banderas". Esto significa que las banderas *CF* , *OF* , *SF* , *ZF* , *AF* , *PF* son modificadas por esas instrucciones. Sin embargo, cualquier instrucción puede modificar los indicadores, por ejemplo, `cmpxchg` modifica la *ZF* .

Siempre revise la referencia de la instrucción para saber qué indicadores son modificados por una instrucción específica.

x86 tiene un conjunto de *saltos condicionales* , referidos anteriormente, que saltan si y solo si algunos indicadores están establecidos o algunos son claros o ambos.

Banderas

Las operaciones aritméticas y lógicas son muy útiles para establecer las banderas. Por ejemplo, después de un `sub eax, ebx` , que ahora tiene valores **sin firmar** , tenemos:

Bandera	Cuando se establece	Cuando claro
ZF	Cuando el resultado es cero. *EAX - EBX = 0* $\Rightarrow$ EAX = EBX	Cuando el resultado **no** es cero. *EAX - EBX* $\neq 0 \Rightarrow$ EAX $\neq$ EBX
CF	Cuando el resultado fue necesario llevar para el MSb. *EAX - EBX <0* $\Rightarrow$ EAX <EBX	Cuando el resultado no **fue** necesario llevar para el MSb. *EAX - EBX* $\not< 0 \Rightarrow \mathrm{EAX} \not< \mathrm{EBX}$
SF	Cuando se establece el resultado MSb.	Cuando el resultado MSb **no se** establece.
DE	Cuando se produjo un desbordamiento firmado.	Cuando no **se** produjo un desbordamiento firmado.
PF	Cuando el número de bits establecido en el byte menos significativo del resultado es par.	Cuando el número de bits establecido en el byte menos significativo del resultado es impar.
AF	Cuando el dígito BCD inferior genera un carry. Es bit 4 carry.	Cuando el dígito BCD inferior **no** generó un acarreo. Es bit 4 carry.

Ensayos no destructivos.

El `sub` y `and` instrucciones que modifican su operando destino y requerirían dos copias adicionales

(guardar y restaurar) para mantener el destino sin modificar.

Para realizar una prueba no destructiva hay las instrucciones `cmp` y `test` . Son idénticos a sus homólogos destructivos, **excepto que el resultado de la operación se descarta y solo se guardan las banderas** .

Destructivo	No destructivo
sub	cmp
and	test

```
test eax, eax             ;and eax, eax
                          ;ZF = 1 iff EAX is zero

test eax, 03h             ;and eax, 03h
                          ;ZF = 1 if both bit[1:0] are clear
                          ;ZF = 0 if at least one of bit[1:0] is set

cmp eax, 241d             ;sub eax, 241d
                          ;ZF = 1 iff EAX is 241
                          ;CF = 1 iff EAX < 241
```

Pruebas firmadas y no firmadas.

La CPU no da un significado especial para registrar los valores [1] , signo es una construcción de programador. **No hay diferencia cuando se prueban valores firmados y no firmados.** El procesador calcula suficientes indicadores para probar las relaciones aritméticas habituales (igual, menor que, mayor que, etc.) tanto si los operandos se consideraran firmados como no firmados.

---

[1] Aunque tiene algunas instrucciones que solo tienen sentido con formatos específicos, como el complemento de dos. Esto es para hacer que el código sea más eficiente, ya que la implementación del algoritmo en el software requeriría una gran cantidad de código.

Saltos condicionales

Según el estado de las banderas, la CPU puede ejecutar o ignorar un salto. Una instrucción que realiza un salto basado en las banderas cae bajo el nombre genérico de *Jcc - Salta en el Código de condición* [1] .

Sinónimos y terminología

Para mejorar la legibilidad del código de ensamblaje, Intel definió varios sinónimos para el mismo código de condición. Por ejemplo, `jae` , `jnb` y `jnc` tienen el mismo código de condición *CF = 0* .

Si bien el nombre de la instrucción puede dar una idea muy clara de cuándo usarlo o no, el único enfoque significativo es reconocer los indicadores que deben probarse y **luego** elegir las instrucciones de manera adecuada.

Sin embargo, Intel dio los nombres de las instrucciones que tienen perfecto sentido cuando se usan después de una instrucción `cmp` . Para los propósitos de esta discusión, se asumirá que `cmp` ha establecido las banderas antes de un salto condicional.

Igualdad

El operando es igual si *ZF* se ha establecido, de lo contrario difieren. Para probar la igualdad necesitamos *ZF = 1* .

```
je a_label           ;Jump if operands are equal
jz a_label           ;Jump if zero (Synonym)

jne a_label          ;Jump if operands are NOT equal
jnz a_label          ;Jump if not zero (Synonym)
```

Instrucción	Banderas
`je` , `jz`	ZF = 1
`jne` `jnz`	ZF = 0

Mas grande que

Para los **operandos sin firmar** , el destino es mayor que el origen si no se necesita carry, es decir, si *CF = 0* . Cuando *CF = 0* es posible que los operandos fueran iguales, la prueba de *ZF* se desambiguará.

```
jae a_label      ;Jump if above or equal (>=)
jnc a_label      ;Jump if not carry (Synonym)
jnb a_label      ;Jump if not below (Synonym)

ja a_label       ;Jump if above  (>)
jnbe a_label     ;Jump if not below and not equal (Synonym)
```

Instrucción	Banderas
`jae` , `jnc` , `jnb`	CF = 0
`ja` , `jnbe`	CF = 0, ZF = 0

Para los **operandos firmados** debemos verificar que *SF = 0* , a menos que haya un desbordamiento firmado, en cuyo caso el *SF* resultante se invierte. Dado que *OF = 0* si no se produjo un desbordamiento firmado y 1 de lo contrario, debemos verificar que *SF = OF* .

ZF se puede utilizar para implementar una prueba estricta / no estricta.

```
jge a_label      ;Jump if greater or equal (>=)
jnl a_label      ;Jump if not less (Synonym)
```

```
jg a_label       ;Jump if greater (>)
jnle a_label     ;Jump if not less and not equal (Synonym)
```

Instrucción	Banderas
`jge` , `jnl`	SF = OF
`jg` , `jnle`	SF = OF, ZF = 0

Menos que

Estos utilizan las condiciones invertidas de arriba.

```
jbe a_label      ;Jump if below or equal (<=)
jna a_label      ;Jump if not above (Synonym)

jb a_label       ;Jump if below (<)
jc a_label       ;Jump if carry (Synonym)
jnae a_label     ;Jump if not above and not equal (Synonym)

;SIGNED

jle a_label      ;Jump if less or equal (<=)
jng a_label      ;Jump if not greater (Synonym)

jl a_label       ;Jump if less (<)
jnge a_label     ;Jump if not greater and not equal (Synonym)
```

Instrucción	Banderas
`jbe jna`	CF = 1 o ZF = 1
`jb jc jnae`	CF = 1
`jle` , `jng`	SF! = OF o ZF = 1
`jl` , `jnge`	SF! = OF

Banderas especificas

Cada bandera se puede probar individualmente con `j<flag_name>` donde nombre de *bandera* no contiene la *F* posterior (por ejemplo, $CF \rightarrow C$, $PF \rightarrow P$).

Los códigos restantes no cubiertos antes son:

Instrucción	Bandera
`js`	SF = 1

Instrucción	Bandera
`jns`	SF = 0
`jo`	OF = 1
`jno`	OF = 0
`jp` , `jpe` (e = par)	PF = 1
`jnp` , `jpo` (o = impar)	PF = 0

Un salto más condicional (extra)

Un salto condicional especial x86 no prueba la bandera. En su lugar, prueba el valor del registro `cx` o `ecx` (basado en que el modo de dirección de la CPU actual es de 16 o 32 bits), y el salto se ejecuta cuando el registro contiene cero.

Esta instrucción fue diseñada para la validación del registro de *contador* (`cx/ecx`) antes de las instrucciones similares a las `rep` , o antes de `loop` bucles de `loop` .

```
jcxz  a_label   ; jump if cx (16b mode) or ecx (32b mode) is zero
jecxz a_label   ; synonym of jcxz (recommended in source code for 32b target)
```

Instrucción	Registro (no bandera)
`jcxz` , `jecxz`	cx = 0 (modo 16b)
`jcxz` , `jecxz`	ecx = 0 (modo 32b)

---

[1] O algo así.

Probar relaciones aritméticas

Enteros sin firmar

Mas grande que

```
cmp eax, ebx
ja a_label
```

Mayor que o igual

```
cmp eax, ebx
jae a_label
```

Menos que

```
cmp eax, ebx
jb a_label
```

Menor o igual

```
cmp eax, ebx
jbe a_label
```

Igual

```
cmp eax, ebx
je a_label
```

No es igual

```
cmp eax, ebx
jne a_label
```

Enteros firmados

Mas grande que

```
cmp eax, ebx
jg a_label
```

Mayor que o igual

```
cmp eax, ebx
jge a_label
```

Menos que

```
cmp eax, ebx
jl a_label
```

Menor o igual

```
cmp eax, ebx
jle a_label
```

Igual

```
cmp eax, ebx
je a_label
```

No es igual

```
cmp eax, ebx
jne a_label
```

---

a_label

En los ejemplos anteriores, `a_label` es el destino de destino para la CPU cuando la condición probada es "verdadera". Cuando la condición probada es "falsa", la CPU continuará en la siguiente instrucción después del salto condicional.

Sinónimos

Hay sinónimos de instrucciones que pueden usarse para mejorar la legibilidad del código. Por ejemplo, `ja` y `jnbe` (Saltar no por debajo ni igual) son la misma instrucción.

Códigos de acompañante firmados y sin firmar.

Operación	No firmado	Firmado
>	`ja`	`jg`
> =	`jae`	`jge`
<	`jb`	`jl`
<=	`jbe`	`jle`
=	`je`	`je`
≠,! =, <>	`jne`	`jne`

Lea Flujo de control en línea: https://riptutorial.com/es/x86/topic/5808/flujo-de-control

Capítulo 6: Fundamentos del registro

Examples

Registros de 16 bits

Cuando Intel definió el 8086 original, era un procesador de 16 bits con un bus de direcciones de 20 bits (ver más abajo). Definieron 8 registros de propósito general de 16 bits, pero les dieron roles específicos para ciertas instrucciones:

- `AX` el registro del acumulador.
  Muchos opcodes asumieron este registro o fueron más rápidos si se especificaron.
- `DX` El registro de datos.
  Esto a veces se combinaba como el alto 16 bits de un valor de 32 bits con `AX` , por ejemplo, como resultado de una multiplicación.
- `CX` El registro de la cuenta.
  Esto se usó en varias instrucciones orientadas a bucles como contador implícito para esos bucles, por ejemplo, `LOOPNE` (bucle si no es igual) y `REP` (movimiento / comparación repetidos)
- `BX` El registro base.
  Esto podría usarse para indexar la base de una estructura en la memoria; ninguno de los registros anteriores se podría usar para indexar directamente en la memoria.
- `SI` El registro de índice de fuente.
  Este fue el índice de fuente implícito en la memoria para ciertas operaciones de movimiento y comparación.
- `DI` El registro del índice de destino.
  Este fue el índice de destino implícito en la memoria para ciertas operaciones de movimiento y comparación.
- `SP` El registro Stack Pointer.
  ¡Este es el registro de propósitos menos generales en el set! Señaló la posición actual en la pila, que se usó explícitamente para las operaciones `PUSH` y `POP` , implícitamente para `CALL` y `RET` con subrutinas, y MUY implícitamente durante las interrupciones. Como tal, usarlo para cualquier otra cosa era peligroso para su programa.
- `BP` El registro de puntero base.
  Cuando las subrutinas llaman a otras subrutinas, la pila contiene múltiples "marcos de pila". `BP` podría utilizarse para mantener el marco de pila actual, y luego, cuando se llamara una nueva subrutina, se podría guardar en la pila, se creó y utilizó el nuevo marco de pila y, al regresar de la subrutina interna, se pudo restaurar el antiguo valor del marco de pila .

Notas:

1. Los tres primeros registros no pueden usarse para indexar en la memoria.

2. `BX` , `SI` y `DI` por índice predeterminado en el segmento de datos actual (ver más abajo).

```
MOV    AX, [BX+5]     ; Point into Data Segment
MOV    AX, ES:[DI+5]  ; Override into Extra Segment
```

3. `DI` , cuando se usa en operaciones de memoria a memoria como `MOVS` y `CMPS` , solo usa el Segmento Extra (ver más abajo). Esto no puede ser anulado.

4. `SP` y `BP` usan el Segmento de Apilamiento (ver más abajo) de manera predeterminada.

Registros de 32 bits

Cuando Intel produjo el 80386, se actualizó de un procesador de 16 bits a uno de 32 bits. El procesamiento de 32 bits significa dos cosas: tanto los datos que se manipularon fueron de 32 bits, como las direcciones de memoria a las que se accedió fueron de 32 bits. Para hacer esto, pero aún siendo compatibles con sus procesadores anteriores, introdujeron modos completamente nuevos para el procesador. Se encontraba en modo de 16 bits o en modo de 32 bits, ¡pero se puede anular este modo de forma individualizada para los datos, el direccionamiento o ambos!

En primer lugar, tenían que definir registros de 32 bits. Lo hicieron simplemente extendiendo los ocho existentes de 16 bits a 32 bits y dándoles nombres "extendidos" con un prefijo `E` : `EAX` , `EBX` , `ECX` , `EDX` , `ESI` , `EDI` , `EBP` y `ESP` . Los 16 bits inferiores de estos registros fueron los mismos que antes, pero las mitades superiores de los registros estaban disponibles para operaciones de 32 bits como `ADD` y `CMP` . Las mitades superiores no eran accesibles por separado como lo habían hecho con los registros de 8 bits.

El procesador tenía que tener modos separados de 16 y 32 bits porque Intel usaba los mismos códigos de operación para muchas de las operaciones: `CMP AX,DX` en modo de 16 bits y `CMP EAX,EDX` en el modo de 32 bits tenía exactamente los mismos códigos de operación ! Esto significaba que el mismo código NO se podía ejecutar en ninguno de los modos:

> El código de operación para "Mover inmediatamente a `AX` " es `0xB8` , seguido de dos bytes del valor inmediato: `0xB8 0x12 0x34`
>
> El código de operación para "Mover inmediatamente a `EAX` " es `0xB8` , seguido de ***cuatro*** bytes del valor inmediato: `0xB8 0x12 0x34 0x56 0x78`

Por lo tanto, el miembro debe saber en qué modo está el procesador cuando ejecuta el código, para que sepa emitir el número correcto de bytes.

Registros de 8 bits

Los primeros cuatro registros de 16 bits podrían tener acceso a sus bytes de la mitad superior e inferior directamente como sus propios registros:

- `AH` y `AL` son las mitades altas y bajas del registro `AX` .
- `BH` y `BL` son las mitades Alta y Baja del registro `BX` .
- `CH` y `CL` son las mitades Alta y Baja del registro `CX` .
- `DH` y `DL` son las mitades Alta y Baja del registro `DX` .

¡Tenga en cuenta que esto significa que alterar `AH` o `AL` también alterará inmediatamente `AX` también! También tenga en cuenta que cualquier operación en un registro de 8 bits no podría afectar a su "socio", lo que incrementaría la `AL` tal manera que el desbordamiento de `0xFF` a `0x00` no alteraría `AH` .

Los registros de 64 bits también tienen versiones de 8 bits que representan sus bytes inferiores:

- `SIL` para `RSI`
- `DIL` para `RDI`
- `BPL` para `RBP`
- `SPL` para `RSP`

Lo mismo se aplica a los registros `R8` a `R15` : sus respectivas partes de byte inferior se denominan `R8B` - `R15B` .

Registros de segmento

Segmentación

Cuando Intel estaba diseñando el 8086 original, ya existían varios procesadores de 8 bits que tenían capacidades de 16 bits, pero querían producir un verdadero procesador de 16 bits. También querían producir algo mejor y más capaz que lo que ya existía, por lo que querían poder acceder a más del máximo de 65,536 bytes de memoria implícitos en los registros de direccionamiento de 16 bits.

Registros del segmento original

Así que implementaron la idea de "Segmentos", un bloque de memoria de 64 kilobytes indexado por los registros de direcciones de 16 bits, que se podrían basar para abordar diferentes áreas de la memoria total. Para mantener estas bases de segmento, se incluyeron Registros de Segmento:

- `CS` El registro del segmento de código.
  Esto contiene el segmento del código que se está ejecutando actualmente, indexado por el registro de `IP` (puntero de instrucción) implícito.
- `DS` El registro del segmento de datos.
  Esto contiene el segmento predeterminado para los datos que están siendo manipulados por el programa.
- `ES` El registro de Segmento Extra.
  Esto contiene un segundo segmento de datos, para operaciones de datos simultáneas en la memoria total.
- `SS` El registro del segmento de pila.
  Esto contiene el segmento de memoria que contiene la pila actual.

Tamaño del segmento?

Los registros de segmento pueden ser de cualquier tamaño, pero hacer que tengan 16 bits de ancho facilita la interoperación con los otros registros. La siguiente pregunta fue: ¿deberían superponerse los segmentos y, de ser así, cuánto? La respuesta a esa pregunta determinaría el tamaño de memoria total al que se podría acceder.

Si no hubiera ninguna superposición, entonces el espacio de direcciones sería de 32 bits (4 gigabytes), ¡un tamaño totalmente insólito en ese momento! Una superposición más "natural" de 8 bits produciría un espacio de direcciones de 24 bits, o 16 megabytes. Al final, Intel decidió guardar cuatro pines de dirección más en el procesador al hacer que el espacio de direcciones de 1 megabyte con una superposición de 12 bits, ¡consideraron que esto era lo suficientemente grande para el tiempo!

Más registros de segmento!

Cuando Intel estaba diseñando el 80386, reconocieron que la suite existente de 4 Registros de segmento no era suficiente para la complejidad de los programas que querían que fuera compatible. Así que agregaron dos más:

- `FS` El registro del segmento lejano
- `GS` El registro del segmento global

Estos nuevos registros de Segmento no tenían ningún uso forzado por el procesador: estaban simplemente disponibles para lo que el programador quisiera.

> Algunos dicen que los nombres fueron elegidos para simplemente continuar con el tema `C` , `D` , `E` del conjunto existente ...

Registros de 64 bits

AMD es un fabricante de procesadores que ha licenciado el diseño del 80386 de Intel para producir versiones compatibles, pero de la competencia. Hicieron cambios internos en el diseño para mejorar el rendimiento u otras mejoras al diseño, al mismo tiempo que podían ejecutar los mismos programas.

Para unir a Intel, crearon extensiones de 64 bits para el diseño de Intel de 32 bits y produjeron el primer chip de 64 bits que aún podía ejecutar código x86 de 32 bits. Intel terminó siguiendo el diseño de AMD en sus versiones de la arquitectura de 64 bits.

El diseño de 64 bits realizó una serie de cambios en el conjunto de registros, a la vez que sigue siendo compatible con versiones anteriores:

- Los registros de propósito general existentes se ampliaron a 64 bits y se nombraron con un prefijo `R` : `RAX` , `RBX` , `RCX` , `RDX` , `RSI` , `RDI` , `RBP` y `RSP` .

  > Nuevamente, las mitades inferiores de estos registros fueron los mismos registros de prefijo `E` que antes, y no se pudo acceder a las mitades superiores de forma independiente.

- Se agregaron 8 registros más de 64 bits, y no se nombraron sino que simplemente se numeraron: `R8` , `R9` , `R10` , `R11` , `R12` , `R13` , `R14` y `R15` .
    - La mitad baja de 32 bits de estos registros es de `R8D` a `R15D` (D para DWORD como es habitual).
    - Se puede acceder a los 16 bits más bajos de estos registros colocando una `W` en el nombre del registro: `R8W` a `R15W` .
- Ahora se puede acceder a los 8 bits más bajos de *los 16* registros:
    - Los tradicionales `AL` , `BL` , `CL` y `DL` ;
    - Los bytes bajos de los registros de puntero (tradicionalmente): `SIL` , `DIL` , `BPL` y `SPL` ;
    - Y los bytes bajos de los 8 nuevos registros: `R8B` a `R15B` .
    - Sin embargo, `AH` , `BH` , `CH` y `DH` son inaccesibles en las instrucciones que usan un prefijo REX (para el tamaño del operando de 64 bits, o para acceder a R8-R15, o para acceder a `SIL` , `DIL` , `BPL` o `SPL`). Con un prefijo REX, el patrón de bits del código de la máquina que solía significar `AH` significa `SPL` , y así sucesivamente. Consulte la Tabla 3-1 del manual de referencia de instrucciones de Intel (volumen 2).

Escribir en un registro de 32 bits siempre pone a cero los 32 bits superiores del registro de ancho completo, a diferencia de escribir en un registro de 8 o 16 bits (que se combina con el valor antiguo, que es una dependencia adicional para la ejecución fuera de orden)).

Registro de banderas

Cuando la unidad lógica aritmética (ALU) x86 realiza operaciones como `NOT` y `ADD` , marca los resultados de estas operaciones ("se convirtió en cero", "se desbordó", "se convirtió en negativo") en un registro `FLAGS` especial de 16 bits. Los procesadores de 32 bits lo actualizaron a 32 bits y lo llamaron `EFLAGS` , mientras que los procesadores de 64 bits lo actualizaron a 64 bits y lo llamaron `RFLAGS` .

Códigos de condición

Pero no importa el nombre, el registro no es accesible directamente (excepto por un par de instrucciones, consulte a continuación). En cambio, se hace referencia a indicadores individuales en ciertas instrucciones, como Salto condicional o Conjunto condicional, conocidos como `Jcc` y `SETcc` donde `cc` significa "código de condición" y hace referencia a la siguiente tabla:

Código de condición	Nombre	Definición
`E` , `Z`	Igual, cero	`ZF` == 1
`NE` , `NZ`	No es igual, no es cero	`ZF` == 0
`O`	Rebosar	`OF` == 1
`NO`	Sin desbordamiento	`OF` == 0
`S`	Firmado	`SF` == 1

Código de condición	Nombre	Definición
`NS`	No firmado	`SF` == 0
`P`	Paridad	`PF` == 1
`NP`	Sin paridad	`PF` == 0
----------	----	----------
`C` , `B` , `NAE`	Llevar, debajo, no arriba o igual	`CF` == 1
`NC` , `NB` , `AE`	No llevar, no debajo, arriba o igual	`CF` == 0
`A` , `NBE`	Arriba, no abajo o igual	`CF` == 0 y `ZF` == 0
`NA` , `BE`	No arriba, abajo o igual	`CF` == 1 o `ZF` == 1
----------	----	----------
`GE` , `NL`	Mayor o igual, no menos	`SF` == `OF`
`NGE` , `L`	No mayor o igual, menos	`SF` ! = `OF`
`G` , `NLE`	Mayor, no menos o igual	`ZF` == 0 y `SF` == `OF`
`NG` , `LE`	No mayor, menor o igual	`ZF` == 1 o `SF` ! = `OF`

En 16 bits, restar `1` de `0` es `65,535` o `-1` dependiendo de si se usa aritmética sin signo o con signo, pero el destino tiene `0xFFFF` cualquier manera. Es solo interpretando los códigos de condición que el significado es claro. Es aún más revelador si `1` se resta de `0x8000` : en aritmética sin signo, eso simplemente cambia `32,768` a `32,767` ; mientras que en la aritmética firmado cambia `-32,768` a `32,767` - un desbordamiento mucho más digno de mención!

Los códigos de condición se agrupan en tres bloques en la tabla: sin signo, sin signo y con signo. El nombre dentro de los últimos dos bloques usa "Arriba" y "Abajo" para los que no están firmados, y "Mayor" o "Menos" para los firmados. Así que `JB` sería "Saltar si está abajo" (sin firmar), mientras que `JL` sería "Saltar si menos" (firmado).

Accediendo a `FLAGS` directamente

Los códigos de condición anteriores son útiles para interpretar conceptos predefinidos, pero los bits de marca reales también están disponibles directamente con las siguientes dos instrucciones:

- `LAHF` Cargar registro `AH` con banderas
- `SAHF` Tienda `AH` registro en Banderas

Solo ciertas banderas se copian a través de estas instrucciones. Todo el `EFLAGS` `FLAGS` / `EFLAGS` / `RFLAGS` se puede guardar o restaurar en la pila:

- `PUSHF` / `POPF` Push / pop `FLAGS` 16 bits en / desde la pila
- `PUSHFD` / `POPFD` Empuje / `POPFD` `EFLAGS` 32 bits en / desde la pila
- `PUSHFQ` / `POPFQ` push / pop de 64 bits `RFLAGS` en / de la pila

Tenga en cuenta que las interrupciones guardan y restauran el registro actual de `[R/E]FLAGS` automáticamente.

Otras banderas

Además de los indicadores ALU descritos anteriormente, el registro `FLAGS` define otros indicadores de estado del sistema:

- `IF` la bandera de interrupción.
  Esto se configura con la instrucción `STI` para habilitar interrupciones globalmente, y se borra con la instrucción `CLI` para deshabilitar interrupciones globalmente.
- `DF` La bandera de dirección.
  Las operaciones de memoria a memoria como `CMPS` y `MOVS` (para comparar y moverse entre ubicaciones de memoria) incrementan o disminuyen automáticamente los registros de índice como parte de la instrucción. El indicador `DF` dicta cuál sucede: si se borra con la instrucción `CLD` , se incrementan; si se establece con la instrucción `STD` , se decrementan.
- `TF` La Bandera De La Trampa. Esta es una bandera de depuración. Al configurarlo, el procesador pasará al modo de "un solo paso": después de que se ejecute cada instrucción, llamará al "Controlador de interrupciones de un solo paso", que se espera sea manejado por un depurador. No hay instrucciones para establecer o borrar este indicador: debe manipular el bit mientras está en la memoria.

80286 Banderas

Para admitir las nuevas instalaciones de multitarea en el 80286, Intel agregó marcas adicionales al registro `FLAGS` :

- `IOPL` El nivel de privilegio de E / S.
  Para proteger el código de multitarea, algunas tareas necesitaban privilegios para acceder a los puertos de E / S, mientras que otras tenían que dejar de acceder a ellos. Intel introdujo una escala de privilegios de cuatro niveles, con 00_2 siendo los más privilegiados y 11_2 siendo los menos. Si `IOPL` era menor que el Nivel de privilegio actual, cualquier intento de acceder a los puertos de E / S, o habilitar o deshabilitar interrupciones, causaría un error de protección general en su lugar.
- Bandera de tarea anidada `NT` .
  Este indicador se estableció si una Tarea `CALL` otra Tarea, lo que provocó un cambio de contexto. El indicador establecido le dijo al procesador que hiciera un cambio de contexto cuando se ejecutó el `RET` .

80386 Banderas

El '386 necesitaba banderas adicionales para admitir características adicionales diseñadas en el procesador.

- `RF` La bandera del curriculum vitae.
  El `386 agregó registros de depuración, que podrían invocar al depurador en varios accesos de hardware como leer, escribir o ejecutar una determinada ubicación de memoria. Sin embargo, cuando el controlador de depuración regresó para ejecutar la instrucción, *el acceso volverá a invocar inmediatamente el controlador de depuración.* O al menos lo haría si no fuera por el indicador de reanudación, que se establece automáticamente en la entrada en el controlador de depuración, y se borra automáticamente después de cada instrucción. Si se establece el indicador de reanudación, el controlador de depuración no se invoca.
- `VM` The Virtual 8086 Flag.
  Para admitir el código de 16 bits más antiguo y el código de 32 bits más nuevo, el 80386 podría ejecutar tareas de 16 bits en un modo "Virtual 8086", con la ayuda de un ejecutivo de Virtual 8086. El indicador de `VM` indicó que esta tarea era una tarea 8086 virtual.

80486 Banderas

A medida que la arquitectura de Intel mejoró, se aceleró a través de tecnología como cachés y ejecución súper escalar. Eso tenía que optimizar el acceso al sistema haciendo suposiciones. Para controlar esos supuestos, se necesitaban más banderas:

- Indicador de comprobación de alineación de `AC` La arquitectura x86 siempre podría acceder a valores de memoria de múltiples bytes en cualquier límite de bytes, a diferencia de algunas arquitecturas que requerían que estuvieran alineadas con el tamaño (los valores de 4 bytes deben estar en los límites de 4 bytes). Sin embargo, fue menos eficiente hacerlo, ya que se necesitaban múltiples accesos de memoria para acceder a datos no alineados. Si se estableció el indicador `AC` , entonces un acceso no alineado provocaría una excepción en lugar de ejecutar el código. De esa manera, el código podría mejorarse durante el desarrollo con `AC` establecido, pero desactivado para el código de producción.

Banderas Pentium

El Pentium agregó más soporte para la virtualización, más el soporte para la instrucción `CPUID` :

- `VIF` La bandera de interrupción virtual.
  Esta es una copia virtual del `IF` de esta Tarea, ya sea que esta Tarea desee o no deshabilitar las interrupciones, sin afectar realmente a las Interrupciones globales.
- `VIP` La bandera pendiente de interrupción virtual.
  Esto indica que una interrupción fue prácticamente bloqueada por `VIF` , por lo que cuando la Tarea realiza una `STI` puede `STI` una interrupción virtual para ella.
- `ID` La `CPUID` -Bandera permitida.
  Si se permite o no que esta tarea ejecute la instrucción `CPUID` . Un monitor virtual podría no permitirlo y "mentir" a la tarea solicitante si ejecuta la instrucción.

Lea Fundamentos del registro en línea: https://riptutorial.com/es/x86/topic/2122/fundamentos-del-

registro

Capítulo 7: Gestión multiprocesador

Parámetros

Registro LAPIC	Dirección (Relativa a *APIC BASE*)
Registro local de IDIC APIC	+ 20h
Registro de vectores de interrupciones espurias	+ 0f0h
Registro de comando de interrupción (ICR); bits 0-31	+ 300h
Registro de comando de interrupción (ICR); bits 32-63	+ 310h

Observaciones

Para acceder a los registros LAPIC, un segmento debe poder alcanzar el rango de direcciones comenzando en *APIC Base* (en *IA32_APIC_BASE*).
Esta dirección es reubicable y, en teoría, se puede configurar para que apunte a algún lugar en la memoria inferior, lo que hace que el rango sea direccionable en modo real.

Sin embargo, los ciclos de lectura / escritura en el rango LAPIC **no se** propagan a la Unidad de Interfaz de Bus, lo que enmascara cualquier acceso a las direcciones "detrás".

Se supone que el lector está familiarizado con el modo irreal , ya que se utilizará en algún ejemplo.

También es necesario ser competente con:

- Manejando la diferencia entre direcciones *lógicas* y *físicas* [1]
- Segmentación en modo real .
- Alias de memoria, id. Es la capacidad de usar diferentes direcciones *lógicas* para la misma dirección *física*
- Absoluta, relativa, lejana, cercana a las llamadas y saltos.
- El ensamblador NASM , particularmente que la directiva `ORG` es global. La división del código en varios archivos simplifica en **gran medida** la codificación, ya que será posible dar diferentes *ORG a* diferentes *secciones* .

Finalmente, asumimos que la CPU tiene un *controlador de interrupción programable avanzado local* (*LAPIC*).
Si es ambiguo desde el contexto, APIC siempre significa LAPIC (e no IOAPIC, o xAPIC en general).

Referencias:

- Capítulo 8 y 10 de los manuales de Intel .

Campos de bits

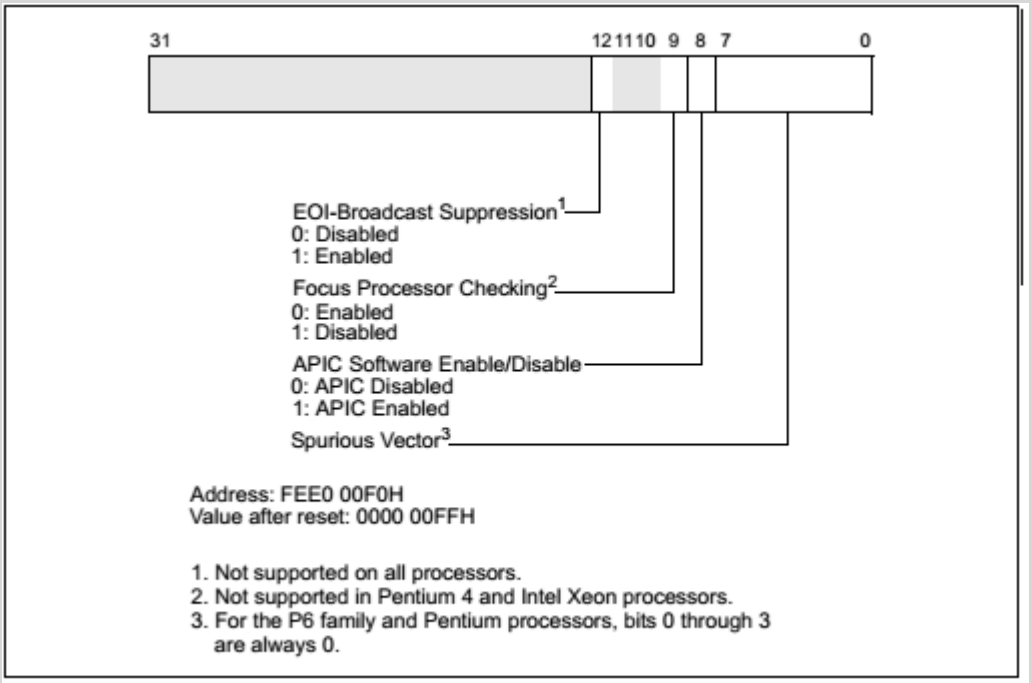


Figure 10-23. Spurious-Interrupt Vector Register (SVR)

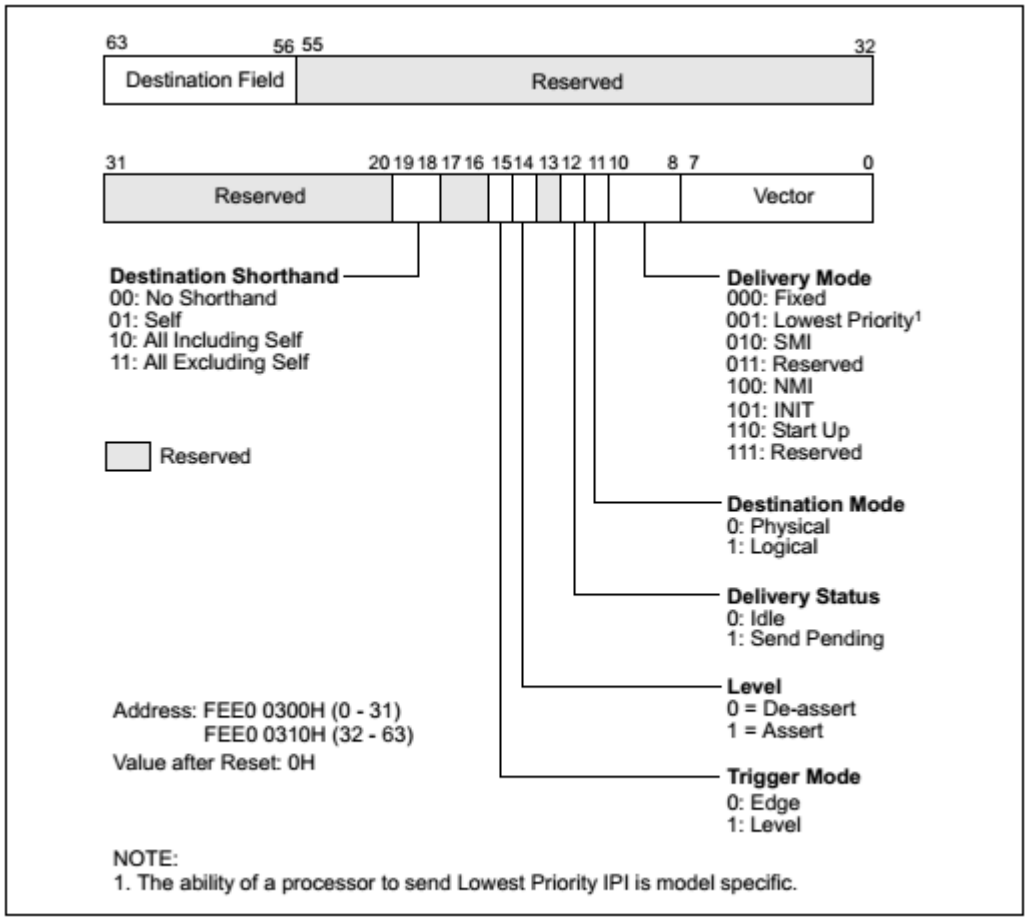


Figure 10-12. Interrupt Command Register (ICR)

Campos de bits

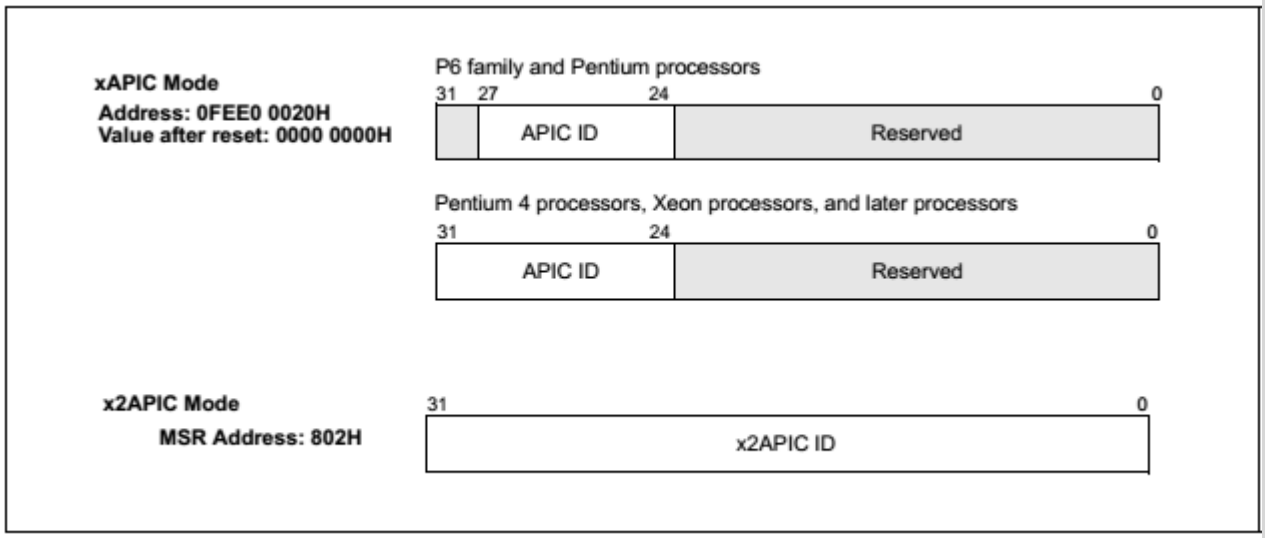


Figure 10-6. Local APIC ID Register

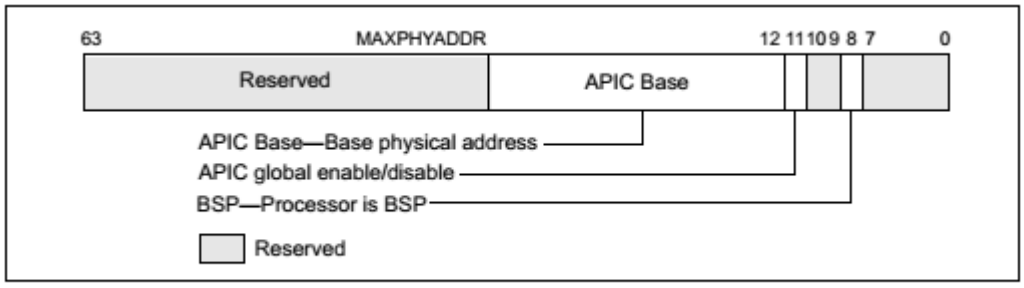


Figure 10-5. IA32_APIC_BASE MSR (APIC_BASE_MSR in P6 Family)

Nombre de MSR	Dirección
IA32_APIC_BASE	1bh

[1] Si se utilizará la paginación, *las* direcciones *virtuales* también entran en juego.

Examples

Despierta todos los procesadores

Este ejemplo activará todos los *Procesadores de aplicaciones* (AP) y los hará, junto con el *Procesador Bootstrap* (BSP), mostrar su ID de LAPIC.

```
; Assemble boot sector and insert it into a 1.44MiB floppy image
;
; nasm -f bin boot.asm -o boot.bin
; dd if=/dev/zero of=disk.img bs=512 count=2880
; dd if=boot.bin of=disk.img bs=512 conv=notrunc

BITS 16
; Bootloader starts at segment:offset 07c0h:0000h
section bootloader, vstart=0000h
jmp 7c0h:__START__
```

```
__START__:
 mov ax, cs
 mov ds, ax
 mov es, ax
 mov ss, ax
 xor sp, sp
 cld

 ;Clear screen
 mov ax, 03h
 int 10h

 ;Set limit of 4GiB and base 0 for FS and GS
 call 7c0h:unrealmode

 ;Enable the APIC
 call enable_lapic

 ;Move the payload to the expected address
 mov si, payload_start_abs
 mov cx, payload_end-payload + 1
 mov di, 400h                ;7c0h:400h = 8000h
 rep movsb


 ;Wakeup the other APs

 ;INIT
 call lapic_send_init
 mov cx, WAIT_10_ms
 call us_wait

 ;SIPI
 call lapic_send_sipi
 mov cx, WAIT_200_us
 call us_wait

 ;SIPI
 call lapic_send_sipi

 ;Jump to the payload
 jmp 0000h:8000h

 ;Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
 ;  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
 ;Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll

 ;CX = Wait (in ms) Max 65536 us (=0 on input)
 us_wait:
  mov dx, 80h               ;POST Diagnose port, 1us per IO
  xor si, si
  rep outsb

  ret

  WAIT_10_ms     EQU 10000
  WAIT_200_us    EQU 200

 ;Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
 ;  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
 ;Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
```

```
enable_lapic:

 ;Enable the APIC globally
 ;On P6 CPU once this flag is set to 0, it cannot be set back to 16
 ;Without an HARD RESET
 mov ecx, IA32_APIC_BASE_MSR
 rdmsr
 or ah, 08h        ;bit11: APIC GLOBAL Enable/Disable
 wrmsr

 ;Mask off lower 12 bits to get the APIC base address
 and ah, 0f0h
 mov DWORD [APIC_BASE], eax

 ;Newer processors enables the APIC through the Spurious Interrupt Vector register
 mov ecx, DWORD [fs: eax + APIC_REG_SIV]
 or ch, 01h                                ;bit8: APIC SOFTWARE enable/disable
 mov DWORD [fs: eax+APIC_REG_SIV], ecx

 ret

;Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
;  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
;Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll

lapic_send_sipi:
 mov eax, DWORD [APIC_BASE]

 ;Destination field is set to 0 has we will use a shorthand
 xor ebx, ebx
 mov DWORD [fs: eax+APIC_REG_ICR_HIGH], ebx

 ;Vector: 08h (Will make the CPU execute instruction ad address 08000h)
 ;Delivery mode: Startup
 ;Destination mode: ignored (0)
 ;Level: ignored (1)
 ;Trigger mode: ignored (0)
 ;Shorthand: All excluding self (3)
 mov ebx, 0c4608h
 mov DWORD [fs: eax+APIC_REG_ICR_LOW], ebx  ;Writing the low DWORD sent the IPI

 ret

 ;Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
;  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
;Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll

lapic_send_init:
 mov eax, DWORD [APIC_BASE]

 ;Destination field is set to 0 has we will use a shorthand
 xor ebx, ebx
 mov DWORD [fs: eax+APIC_REG_ICR_HIGH], ebx

 ;Vector: 00h
 ;Delivery mode: Startup
 ;Destination mode: ignored (0)
 ;Level: ignored (1)
 ;Trigger mode: ignored (0)
```

```
  ;Shorthand: All excluding self (3)
  mov ebx, 0c4500h
  mov DWORD [fs: eax+APIC_REG_ICR_LOW], ebx  ;Writing the low DWORD sent the IPI

  ret

 IA32_APIC_BASE_MSR    EQU    1bh

 APIC_REG_SIV        EQU    0f0h

 APIC_REG_ICR_LOW    EQU 300h
 APIC_REG_ICR_HIGH    EQU 310h

 APIC_REG_ID        EQU 20h

 ;Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
 ;  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
 ;Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll

 APIC_BASE            dd     00h

 ;Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
 ;  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
 ;Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll

unrealmode:
 lgdt [cs:GDT]

 cli

 mov eax, cr0
 or ax, 01h
 mov cr0, eax

 mov bx, 08h
 mov fs, bx
 mov gs, bx

 and ax, 0fffeh
 mov cr0, eax

 sti

 ;IMPORTAT: This call is FAR!
 ;So it can be called from everywhere
 retf

 GDT:
    dw 0fh
    dd GDT + 7c00h
    dw 00h

    dd 0000ffffh
    dd 00cf9200h

 ;Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
 ;  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll
 ;Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll  Ll

payload_start_abs:
; payload starts at segment:offset 0800h:0000h
```

```
section payload, vstart=0000h, align=1
 payload:

  ;IMPORTANT NOTE: Here we are in a "new" CPU every state we set before is no
  ;more present here (except for the BSP, but we handler every processor with
  ;the same code).
 jmp 800h: __RESTART__

 __RESTART__:
  mov ax, cs
  mov ds, ax
  xor sp, sp
  cld

  ;IMPORTANT: We can't use the stack yet. Every CPU is pointing to the same stack!

  ;Get an unique id
  mov ax, WORD [counter]
  .try:
    mov bx, ax
    inc bx
    lock cmpxchg WORD [counter], bx
   jnz .try

  mov cx, ax            ;Save this unique id

  ;Stack segment = CS + unique id * 1000
  shl ax, 12
  mov bx, cs
  add ax, bx
  mov ss, ax

  ;Text buffer
  push 0b800h
  pop es

  ;Set unreal mode again
  call 7c0h:unrealmode

  ;Use GS for old variables
  mov ax, 7c0h
  mov gs, ax

  ;Calculate text row
  mov ax, cx
  mov bx, 160d          ;80 * 2
  mul bx
  mov di, ax

  ;Get LAPIC id
  mov ebx, DWORD [gs:APIC_BASE]
  mov edx, DWORD [fs:ebx + APIC_REG_ID]
  shr edx, 24d
  call itoa8

  cli
  hlt

  ;DL = Number
  ;DI = ptr to text buffer
  itoa8:
```

```
    mov bx, dx
    shr bx, 0fh
    mov al, BYTE [bx +  digits]
    mov ah, 09h
    stosw

    mov bx, dx
    and bx, 0fh
    mov al, BYTE [bx +  digits]
    mov ah, 09h
    stosw

    ret

  digits db "0123456789abcdef"
  counter dw 0

 payload_end:



; Boot signature is at physical offset 01feh of
; the boot sector
section bootsig, start=01feh
 dw 0aa55h
```

Hay dos pasos principales para realizar:

1. Despertando los APs
Esto se logra mediante la inserción de una secuencia *INIT-SIPI-SIPI* (ISS) en todos los AP.

El BSP que enviará la secuencia ISS utilizando como destino la abreviatura *Todo excluido* , con lo que se dirigirá a todos los puntos de acceso.

Todas las CPU que están activadas al momento de su recepción ignoran un SIPI (Interrupción de Inter Interruptor de Inicio), por lo que el segundo SIPI se ignora si el primero es suficiente para activar los procesadores de destino. Es asesorado por Intel por razones de compatibilidad.

Un SIPI contiene un *vector* , es similar en significado, **pero absolutamente diferente en la práctica** , a un vector de interrupción (también conocido como número de interrupción).
El vector es un número de 8 bits, de valor *V* (representado como *vv* en la base 16), que hace que la CPU comience a ejecutar instrucciones en la dirección *física 0vv000h* .
Llamaremos a *0vv000h* la *dirección de Wake-up* (WA).
El WA se fuerza en un límite de 4 KB (o página).

Usaremos 08h como *V* , el WA es entonces *08000h* , 400h bytes después del cargador de arranque.

Esto le da control a los APs.

2. Inicializando y diferenciando los APs.
Es necesario tener un código ejecutable en el WA. El gestor de arranque está a *7c00h* , por lo que necesitamos reubicar algo de código en el límite de la página.

Lo primero que debe recordar al escribir la carga útil es que cualquier acceso a un recurso compartido debe estar protegido o diferenciado.
Un recurso compartido común es la pila , si inicializamos la pila de forma ingenua, ¡todos los AP terminarán usando la misma pila!

El primer paso es utilizar diferentes direcciones de pila, *diferenciando* así la pila.
Lo logramos asignando un número único, basado en cero, para cada CPU. Este número, lo llamaremos *índice* , se usa para diferenciar la pila y la línea donde la CPU escribirá su ID de APIC.

La dirección de pila para cada CPU es *800h: (índice * 1000h) que* proporciona a cada AP 64 KB de pila.
El número de línea para cada CPU es el *índice* , el puntero al búfer de texto es, por lo tanto, el *índice* 80 * 2 *.

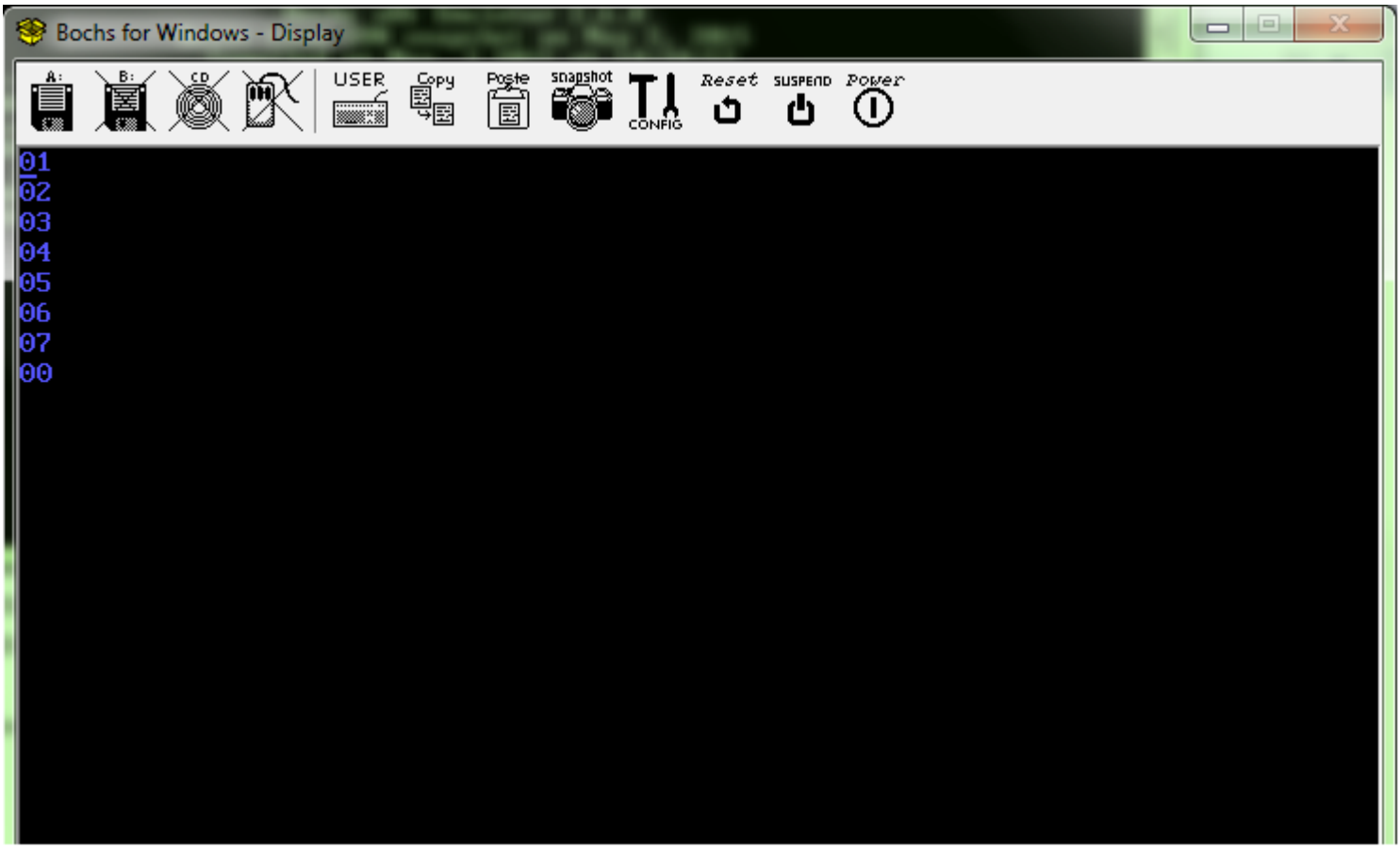
Para generar el índice, se utiliza un `lock cmpxchg` para incrementar y devolver una PALABRA atómicamente.

Notas finales

- Se utiliza una escritura en el puerto 80h para generar un retraso de 1 µs.
- `unrealmode` es una rutina lejana, por lo que también puede llamarse después del despertar.
- El BSP también salta a la WA.

Captura de pantalla

De Bochs con 8 procesadores.



Lea Gestión multiprocesador en línea: https://riptutorial.com/es/x86/topic/5809/gestion-multiprocesador

Capítulo 8: Manipulación de datos

Sintaxis

- ***.386*** : Le dice a **MASM** que compile para una versión mínima de chip x86 de 386.
- ***.model*** : establece el modelo de memoria a usar, vea .MODEL .
- ***.code*** : segmento de código, utilizado para procesos como el proceso principal.
- ***Proc*** : Declara proceso.
- ***ret*** : se utiliza para salir con éxito de las funciones, consulte Cómo trabajar con valores de retorno .
- ***endp*** : finaliza la declaración del proceso.
- ***público*** : pone el proceso a disposición de todos los segmentos del programa.
- ***end*** : Finaliza el programa, o si se usa con un proceso, como en " **end main main** ", hace que el proceso sea el método principal.
- ***llamada*** : llama al proceso y coloca su código de operación en la pila, vea Flujo de control .
- ***ecx*** : contador de registro, ver registros .
- ***ecx*** : contador de registro.
- ***mul*** : multiplica el valor por eax

Observaciones

mov se utiliza para transferir datos entre los registros .

Examples

Usando MOV para manipular valores

Descripción:

`mov` *copia los* valores de bits del argumento de origen al argumento de destino.

El origen / destino común son los registros , generalmente la forma más rápida de manipular valores con [en] CPU.

Otro grupo importante de valores source_of / destination_for es la memoria de la computadora.

Finalmente, algunos valores inmediatos pueden ser parte de la codificación de la instrucción `mov` , ahorrando tiempo de acceso a la memoria por separado leyendo el valor junto con la instrucción.

En la CPU x86 en modo de 32 y 64 bits hay muchas posibilidades de combinarlos, especialmente varios modos de direccionamiento de memoria. En general, la copia de memoria a memoria está fuera del límite (excepto las instrucciones especializadas como `MOVSB`), y dicha manipulación requiere primero el almacenamiento intermedio de valores en el registro [s].

Paso 1: configure su proyecto para usar **MASM** , consulte Ejecución del ensamblado x86 en

Visual Studio 2015

Paso 2: Escribe esto:

```
.386
.model small
.code

public main
main proc
    mov ecx, 16       ; Move immediate value 16 into ecx
    mov eax, ecx      ; Copy value of ecx into eax
    ret               ; return back to caller
        ; function return value is in eax (16)
main endp
end main
```

Paso 3: Compilar y depurar.

El programa debe devolver valor `16` .

Lea Manipulación de datos en línea: https://riptutorial.com/es/x86/topic/8030/manipulacion-de-datos

Capítulo 9: Mecanismos de llamada al sistema

Examples

Llamadas del BIOS

Cómo interactuar con la BIOS

El sistema básico de entrada / salida, o BIOS, es lo que controla la computadora antes de que se ejecute cualquier sistema operativo. Para acceder a los servicios proporcionados por el BIOS, el código de ensamblaje utiliza *interrupciones* . Una interrupción toma la forma de

```
int <interrupt> ; interrupt must be a literal number, not in a register or memory
```

El número de interrupción debe estar entre 0 y 255 (0x00 - 0xFF), inclusive.

La mayoría de las llamadas del BIOS usan el registro `AH` como un parámetro de "selección de función", y usan el registro `AL` como un parámetro de datos. La función seleccionada por `AH` depende de la interrupción llamada. Algunas llamadas de BIOS requieren un solo parámetro de 16 bits en `AX` , o no aceptan parámetros en absoluto, y son simplemente llamadas por la interrupción. Algunos tienen incluso más parámetros, que se pasan en otros registros.

Los registros utilizados para las llamadas del BIOS son fijos y no pueden intercambiarse con otros registros.

Usando llamadas del BIOS con función de selección

La sintaxis general para una interrupción de BIOS usando un parámetro de selección de función es:

```
mov ah, <function>
mov al, <data>
int <interrupt>
```

Ejemplos

Cómo escribir un carácter en la pantalla:

```
mov ah, 0x0E            ; Select 'Write character' function
mov al, <char>          ; Character to write
int 0x10                ; Video services interrupt
```

Cómo leer un carácter desde el teclado (bloqueo):

```
mov ah, 0x00             ; Select 'Blocking read character' function
int 0x16                 ; Keyboard services interrupt
mov <ascii_char>, al     ; AL contains the character read
mov <scan_code>, ah      ; AH contains the BIOS scan code
```

Cómo leer uno o más sectores desde una unidad externa (utilizando el direccionamiento CHS):

```
mov ah, 0x02             ; Select 'Drive read' function
mov bx, <destination>    ; Destination to write to, in ES:BX
mov al, <num_sectors>    ; Number of sectors to read at a time
mov dl, <drive_num>      ; The external drive's ID
mov cl, <start_sector>   ; The sector to start reading from
mov dh, <head>           ; The head to read from
mov ch, <cylinder>       ; The cylinder to read from
int 0x13                 ; Drive services interrupt
jc <error_handler>       ; Jump to error handler on CF set
```

Cómo leer el sistema RTC (Real Time Clock):

```
mov ah, 0x00             ; Select 'Read RTC' function
int 0x1A                 ; RTC services interrupt
shl ecx, 16              ; Clock ticks are split in the CX:DX pair, so shift ECX left by 16...
or cx, dx                ; and add in the low half of the pair
mov <new_day>, al        ; AL is non-zero if the last call to this function was before
midnight
                         ; Now ECX holds the clock ticks (approx. 18.2/sec) since midnight
                         ; and <new_day> is non-zero if we passed midnight since the last read
```

Cómo leer la hora del sistema desde el RTC:

```
mov ah, 0x02             ; Select 'Read system time' function
int 0x1A                 ; RTC services interrupt
                         ; Now CH contains hour, CL minutes, DH seconds, and DL the DST flag,
                         ; all encoded in BCD (DL is zero if in standard time)
                         ; Now we can decode them into a string (we'll ignore DST for now)

mov al, ch               ; Get hour
shr al, 4                ; Discard one's place for now
add al, 48               ; Add ASCII code of digit 0
mov [CLOCK_STRING+0], al ; Set ten's place of hour
mov al, ch               ; Get hour again
and al, 0x0F             ; Discard ten's place this time
add al, 48               ; Add ASCII code of digit 0 again
mov [CLOCK_STRING+1], al ; Set one's place of hour

mov al, cl               ; Get minute
shr al, 4                ; Discard one's place for now
add al, 48               ; Add ASCII code of digit 0
mov [CLOCK_STRING+3], al ; Set ten's place of minute
mov al, cl               ; Get minute again
and al, 0x0F             ; Discard ten's place this time
```

```
add al, 48               ; Add ASCII code of digit 0 again
mov [CLOCK_STRING+4], al ; Set one's place of minute

mov al, dh               ; Get second
shr al, 4                ; Discard one's place for now
add al, 48               ; Add ASCII code of digit 0
mov [CLOCK_STRING+6], al ; Set ten's place of second
mov al, dh               ; Get second again
and al, 0x0F             ; Discard ten's place this time
add al, 48               ; Add ASCII code of digit 0 again
mov [CLOCK_STRING+7], al ; Set one's place of second
...
db CLOCK_STRING "00:00:00", 0   ; Place in some separate (non-code) area
```

Cómo leer la fecha del sistema desde el RTC:

```
mov ah, 0x04             ; Select 'Read system date' function
int 0x1A                 ; RTC services interrupt
                         ; Now CH contains century, CL year, DH month, and DL day, all in BCD
                         ; Decoding to a string is similar to the RTC Time example above
```

Cómo obtener el tamaño de memoria baja contigua:

```
int 0x12                 ; Conventional memory interrupt (no function select parameter)
and eax, 0xFFFF          ; AX contains kilobytes of conventional memory; clear high bits of
EAX
shl eax, 10              ; Multiply by 1 kilobyte (1024 bytes = 2^10 bytes)
                         ; EAX contains the number of bytes available from address 0000:0000
```

Cómo reiniciar la computadora:

```
int 0x19                 ; That's it! One call. Just make sure nothing has overwritten the
                         ; interrupt vector table, since this call does NOT restore them to
the
                         ; default values of normal power-up. This means this call will not
                         ; work too well in an environment with an operating system loaded.
```

Manejo de errores

Es posible que algunas llamadas de BIOS no se implementen en todas las máquinas y no se garantiza que funcionen. A menudo, una interrupción no implementada devolverá `0x86` o `0x80` en el registro `AH` . **Casi todas las interrupciones establecerán el indicador de acarreo (CF) en una condición de error.** Esto facilita saltar a un controlador de errores con el salto condicional `jc` . (Ver saltos condicionales)

Referencias

Una lista bastante exhaustiva de llamadas del BIOS y otras interrupciones es la Lista de interrupciones de Ralf Brown . Una versión HTML se puede encontrar aquí .

Las interrupciones que a menudo se asume que están disponibles se encuentran en una lista en Wikipedia .

Puede encontrar una descripción más detallada de las interrupciones comúnmente disponibles en osdev.org

Lea Mecanismos de llamada al sistema en línea:
https://riptutorial.com/es/x86/topic/6946/mecanismos-de-llamada-al-sistema

Capítulo 10: Mejoramiento

Introducción

La familia x86 ha existido durante mucho tiempo, y como tal, hay muchos trucos y técnicas que se han descubierto y desarrollado que son de conocimiento público, o tal vez no tanto. La mayoría de estos trucos aprovechan el hecho de que muchas instrucciones hacen lo mismo de manera efectiva, pero las diferentes versiones son más rápidas, ahorran memoria o no afectan a los indicadores. Aquí hay una serie de trucos que se han descubierto. Cada uno tiene sus pros y contras, por lo que deben ser enumerados.

Observaciones

En caso de duda, siempre puede consultar el Manual de referencia de optimización de arquitecturas Intel 64 y IA-32 , que es un gran recurso de la compañía detrás de la arquitectura x86.

Examples

Poner a cero un registro

La forma obvia de poner a cero un registro es `MOV` en un `0` ejemplo:

```
B8 00 00 00 00    MOV eax, 0
```

Tenga en cuenta que esta es una instrucción de 5 bytes.

Si está dispuesto a pegar las banderas (`MOV` nunca afecta a las banderas), puede usar la instrucción `XOR` para hacer bitwise-XOR el registro consigo mismo:

```
33 C0             XOR eax, eax
```

Esta instrucción requiere solo 2 bytes y se ejecuta más rápido en todos los procesadores .

Mover la bandera de Carry a un registro

Fondo

Si el indicador de Carry (`C`) tiene un valor que desea poner en un registro, la forma más ingenua es hacer algo como esto:

```
    mov  al, 1
    jc   NotZero
```

```
    mov  al, 0
NotZero:
```

Utilice 'sbb'

Una forma más directa, evitando el salto, es usar "Restar con préstamo":

```
    sbb  al,al    ; Move Carry to al
```

Si `C` es cero, entonces `al` será cero. De lo contrario será `0xFF` (`-1`). Si necesita que sea `0x01` , agregue:

```
    and  al, 0x01 ; Mask down to 1 or 0
```

Pros

- Sobre el mismo tamaño
- Dos o una instrucciones menos.
- Ningún salto caro

Contras

- Es opaco para un lector que no está familiarizado con la técnica.
- Altera otras banderas.

Prueba un registro para 0

Fondo

Para averiguar si un registro contiene un cero, la técnica ingenua es hacer esto:

```
    cmp   eax, 0
```

Pero si miras el código de operación para esto, obtienes esto:

```
83 F8 00      cmp   eax, 0
```

`test` uso

```
    test   eax, eax      ; Equal to zero?
```

Examine el código de operación que obtiene:

```
85 c0         test   eax, eax
```

Pros

- Sólo dos bytes!

Contras

- Opaco a un lector no familiarizado con la técnica.

También puede consultar la pregunta de preguntas y respuestas sobre esta técnica .

Sistema Linux llama con menos hinchazón

En Linux de 32 bits, las llamadas al sistema generalmente se realizan mediante la instrucción sysenter (yo digo que generalmente porque los programas más antiguos usan el `int 0x80` ahora en desuso), sin embargo, esto puede ocupar mucho espacio en un programa, por lo que hay formas Puede cortar esquinas para acortar y acelerar las cosas.
Este suele ser el diseño de una llamada al sistema en Linux de 32 bits:

```
mov eax, <System call number>
mov ebx, <Argument 1> ;If applicable
mov ecx, <Argument 2> ;If applicable
mov edx, <Argument 3> ;If applicable
push <label to jump to after the syscall>
push ecx
push edx
push ebp
mov ebp, esp
sysenter
```

¡Eso es masivo! Pero hay algunos trucos que podemos tirar para evitar este lío.
El primero es establecer ebp al valor de esp disminuido por el tamaño de 3 registros de 32 bits, es decir, 12 bytes. Esto es genial siempre y cuando esté de acuerdo con sobrescribir ebp, edx y ecx con basura (como cuando moverá un valor a esos registros directamente después de todos modos), podemos hacerlo usando la instrucción LEA para que no sea necesario. Para afectar el valor de ESP en sí.

```
mov eax, <System call number>
mov ebx, <Argument 1>
mov ecx, <Argument 2>
mov edx, <Argument 3>
push <label to jump to after the syscall>
lea ebp, [esp-12]
sysenter
```

Sin embargo, no hemos terminado, si la llamada al sistema es sys_exit, podemos evitar que no

pongamos nada en absoluto en la pila.

```
mov eax, 1
xor ebx, ebx ;Set the exit status to 0
mov ebp, esp
sysenter
```

Multiplica por 3 o 5

Fondo

Para obtener el producto de un registro y una constante y almacenarlo en otro registro, la forma ingenua es hacer esto:

```
    imul ecx, 3      ; Set ecx to 5 times its previous value
    imul edx, eax, 5 ; Store 5 times the contend of eax in edx
```

Usar `lea`

Las multiplicaciones son operaciones caras. Es más rápido usar una combinación de turnos y adiciones. Para el caso particular de muliplying el contendiente de un registro de 32 o 64 bits que no es `esp` o `rsp` por 3 o 5, puede usar la instrucción lea. Esto utiliza el circuito de cálculo de dirección para calcular el producto rápidamente.

```
    lea ecx, [2*ecx+ecx] ; Load 2*ecx+ecx = 3*ecx into ecx
    lea edx, [4*edx+edx] ; Load 4*edx+edx = 5*edx into edx
```

Muchos ensambladores también entenderán

```
    lea ecx, [3*ecx]
    lea edx, [5*edx]
```

Para todos los multiplicandos posibles, otros `ebp` o `rbp` , la `ebp` instrucción resultante es la misma que con el uso de `imul` .

Pros

- Ejecuta mucho más rápido

Contras

- Si su multiplicando es `ebp` o `rbp` , toma un byte más usando `imul`
- Más para escribir si su ensamblador no admite los accesos directos
- Opaco a un lector no familiarizado con la técnica.

Lea Mejoramiento en línea: https://riptutorial.com/es/x86/topic/3215/mejoramiento

Capítulo 11: Modos Real vs Protegido

Examples

Modo real

Cuando Intel diseñó el x86 original, el 8086 (y el derivado de 8088), incluyeron la segmentación para permitir que el procesador de 16 bits acceda a más de 16 bits de direcciones. Hicieron esto haciendo que las direcciones de 16 bits fueran relativas a un registro de segmento de 16 bits dado, de los cuales definieron cuatro: segmento de código (`CS`), segmento de datos (`DS`), segmento extra (`ES`) y segmento de pila (`SS`) .

La mayoría de las instrucciones implicaban qué Registro de segmento utilizar: las instrucciones se incluyeron en el Segmento de código, `PUSH` y `POP` implicaron el Segmento de pila, y las referencias de datos simples implicaban el Segmento de datos, aunque esto podría anularse para acceder a la memoria en cualquiera de los otros Segmentos.

La implementación fue sencilla: para cada acceso a la memoria, la CPU tomaría el Registro de segmentos implícito (o explícito), lo desplazaría cuatro lugares a la izquierda y luego agregaría la dirección indicada:

```
        +-------------------+---------+
Segment | 16-bit value      | 0 0 0 0 |
        +-------------------+---------+
                   PLUS
        +---------+-------------------+
Address | 0 0 0 0 | 16-bit value      |
        +---------+-------------------+
                  EQUALS
        +-----------------------------+
Result  |  20-bit memory address      |
        +-----------------------------+
```

Esto permitió varias técnicas:

- Permitir que el Código, los Datos y la Pila sean de acceso mutuo (`CS` , `DS` y `SS` tenían el mismo valor);
- Mantener el Código, los Datos y la Pila completamente separados unos de otros (`CS` , `DS` y `SS` todos los 4K (o más) separados entre sí, recuerde que se multiplica por 16, por lo que es 64K).

¡También permitió superposiciones extrañas y todo tipo de cosas extrañas!

Cuando se inventó el 80286, admitió este modo heredado (ahora llamado "Modo Real"), pero agregó un nuevo modo llamado "Modo Protegido" (qv).

Lo importante a notar es que en Modo Real:

- Se pudo acceder a cualquier dirección de memoria, simplemente colocando el valor correcto

dentro de un registro de segmento y accediendo a la dirección de 16 bits;
- El alcance de la "protección" era permitir al programador separar diferentes áreas de la memoria para diferentes propósitos y hacer que sea *más difícil* escribir accidentalmente en los datos incorrectos, al mismo tiempo que es posible hacerlo.

En otras palabras ... ¡no muy protegido en absoluto!

Modo protegido

Introducción

Cuando se inventó el 80286, admitió la Segmentación 8086 heredada (ahora llamada "Modo Real"), y agregó un nuevo modo llamado "Modo Protegido". Este modo ha estado en todos los procesadores x86 desde entonces, aunque mejorado con varias mejoras, como el direccionamiento de 32 y 64 bits.

Diseño

En el modo protegido, el simple "Agregar dirección al registro de segmento cambiado" se eliminó por completo. Mantuvieron los registros de segmentos, pero en lugar de usarlos para calcular una dirección, los usaron para indexar en una tabla (en realidad, una de dos ...) que definía el segmento al que se debía acceder. Esta definición no solo describe dónde estaba en la memoria el Segmento (usando Base y Límite), sino también qué *tipo* de Segmento era (Código, Datos, Pila o incluso Sistema) y qué tipos de programas podrían acceder a él (Kernel del SO, programa normal). , Controlador del dispositivo, etc.).

Registro de segmento

Cada registro de segmento de 16 bits tomó la siguiente forma:

```
+------------+-----+------+
| Desc Index | G/L | Priv |
+------------+-----+------+
 Desc Index = 13-bit index into a Descriptor Table (described below)
 G/L        = 1-bit flag for which Descriptor Table to Index: Global or Local
 Priv       = 2-bit field defining the Privilege level for access
```

Global / Local

El bit Global / Local definió si el acceso se encontraba en una Tabla Global de descriptores (llamada, sin duda, la Tabla de Descriptor Global o GDT), o la Tabla de Descriptor Local (LDT). La idea para el LDT era que cada programa podría tener su propia tabla de descriptores: el sistema operativo definiría un conjunto global de segmentos, y cada programa tendría su propio conjunto de códigos locales, datos y segmentos de pila. El sistema operativo gestionaría la memoria entre

las diferentes tablas de descriptores.

Tabla Descriptora

Cada tabla de descriptores (global o local) era una matriz de 64K de 8.192 descriptores: cada uno de ellos era un registro de 8 bytes que definía múltiples aspectos del segmento que estaba describiendo. Los campos del Índice de Descriptor de los Registros de Segmentos permitieron 8,192 descriptores: ¡no es coincidencia!

Descriptor

Un Descriptor contenía la siguiente información: tenga en cuenta que el formato del Descriptor cambió cuando se lanzaron nuevos procesadores, pero se mantuvo el mismo tipo de información en cada uno:

- **Base**
  Esto definió la dirección de inicio del segmento de memoria.
- **Límite**
  Esto definió el tamaño del segmento de memoria - más o menos. Tuvieron que tomar una decisión: ¿un tamaño de `0x0000` significaría un tamaño de `0` , por lo que no sería accesible? ¿O tamaño máximo?
  En su lugar, eligieron una tercera opción: el campo Límite fue la última ubicación accesible dentro del Segmento. Eso significaba que se podía definir un segmento de un bye; o un tamaño máximo para el tamaño de la dirección.
- **Tipo**
  Hubo varios tipos de Segmentos: el Código, los Datos y la Pila tradicionales (ver más abajo), pero también se definieron otros Segmentos del Sistema:
  - Los segmentos de la tabla de descriptores locales definieron cuántos descriptores locales se podían acceder;
  - Los Segmentos de Estado de Tarea podrían usarse para el cambio de contexto administrado por hardware;
  - "Puertas de llamada" controladas que podrían permitir que los programas llamen al sistema operativo, pero solo a través de puntos de entrada cuidadosamente administrados.
- **Atributos**
  También se mantuvieron ciertos atributos del Segmento, donde fue relevante:
  - Sólo lectura frente a lectura-escritura;
  - Si el segmento estaba actualmente presente o no, lo que permite la administración de memoria a pedido;
  - Qué nivel de código (SO vs Driver vs programa) podría acceder a este segmento.

¡Verdadera protección al fin!

Si el sistema operativo conservaba las Tablas de Descriptor en Segmentos a las que no podían acceder los simples programas, entonces podría administrar con precisión qué Segmentos fueron definidos, y qué memoria fue asignada y accesible para cada uno. Un programa podría fabricar cualquier valor de Registro de segmento que le gustara, pero si tuviera la *audacia* de *cargarlo* realmente en un *Registro de Segmento* ... el hardware de la CPU reconocería que el valor del Descriptor propuesto rompió cualquiera de una gran cantidad de reglas, y en lugar de completar la solicitud, generaría una excepción en el sistema operativo para permitirle manejar el programa errante.

> Esta excepción fue generalmente la n.º 13, la excepción de protección general, que se hizo famosa en todo el mundo por Microsoft Windows ... (¿Alguien cree que un ingeniero de Intel era supersticioso?)

Los errores

Los tipos de errores que podrían ocurrir incluyen:

- Si el Índice de Descriptor propuesto era más grande que el tamaño de la tabla;
- Si el Descriptor propuesto era un Descriptor de Sistema en lugar de Código, Datos o Pila;
- Si el Descriptor propuesto era más privilegiado que el programa solicitante;
- Si el Descriptor propuesto se marcó como No legible (como un segmento de código), pero se intentó leer en lugar de ejecutarse;
- Si el Descriptor propuesto fue marcado No Presente.

> Tenga en cuenta que el último no puede ser un problema fatal para el programa: el sistema operativo podría observar el indicador, restablecer el segmento, marcarlo como ahora presente y permitir que la instrucción de fallas proceda con éxito.

O tal vez el Descriptor se cargó exitosamente en un Registro de Segmento, pero luego un acceso futuro con él rompió una de varias reglas:

- El registro de segmento se cargó con el índice de descriptor `0x0000` para el GDT. Esto fue reservado por el hardware como `NULL` ;
- Si el Descriptor cargado se marcó como Sólo lectura, se intentó una Escritura.
- Si alguna parte del acceso (1, 2, 4 o más bytes) estaba fuera del límite del segmento.

Cambio al modo protegido

Cambiar al modo protegido es fácil: solo necesita configurar un bit en un registro de control. Pero *mantenerse* en Modo Protegido, sin que la CPU levante las manos y se reinicie debido a que no sabe qué hacer a continuación, requiere mucha preparación.

En resumen, los pasos requeridos son los siguientes:

- Es necesario configurar un área de memoria para la tabla global de descriptores para definir un mínimo de tres descriptores:

    1. El zeroeth, `NULL` Descriptor;
    2. Otro descriptor para un segmento de código;
    3. Otro descriptor para un segmento de datos.

        Esto puede ser usado tanto para Datos como para Apilar.

- El Registro de la Tabla de Descriptor Global (`GDTR`) debe inicializarse para apuntar a esta área definida de la memoria;

```
GDT_Ptr    dw      SIZE GDT
           dd      OFFSET GDT

           ...

           lgdt    [GDT_Ptr]
```

- El bit `PM` en `CR0` necesita ser configurado:

```
     mov   eax, cr0      ; Get CR0 into register
     or    eax, 0x01     ; Set the Protected Mode bit
     mov   cr0, eax      ; We're now in Protected Mode!
```

- Los Registros de segmento deben cargarse desde el GDT para eliminar los valores actuales del Modo Real:

```
     jmp   0x0008:NowInPM  ; This is a FAR Jump. 0x0008 is the Code Descriptor

NowInPM:
     mov   ax, 0x0010      ; This is the Data Descriptor
     mov   ds, ax
     mov   es, ax
     mov   ss, ax
     mov   sp, 0x0000      ; Top of stack!
```

Tenga en cuenta que esto es lo ***mínimo,*** sólo para obtener la CPU en modo protegido. Para realmente tener todo el sistema listo puede requerir muchos más pasos. Por ejemplo:

- Es posible que las áreas de memoria superiores tengan que estar habilitadas: desactivar la puerta `A20` ;
- Definitivamente, las Interrupciones deberían estar deshabilitadas, pero tal vez los diversos Manejadores de fallas podrían configurarse antes de ingresar al Modo protegido, para permitir errores al inicio del proceso.

    El autor original de esta sección escribió un tutorial completo sobre cómo ingresar al Modo protegido y cómo trabajar con él.

Modo irreal

El *modo irreal* explota dos hechos sobre cómo los procesadores Intel y AMD cargan y guardan la información para describir un segmento.

1. El procesador almacena en caché la información del descriptor obtenida durante un *movimiento* en un registro de selección en modo protegido.
   Esta información se almacena en una parte invisible arquitectónica del registro de selector ellos mismos.

2. En el modo real, los registros de selección se denominan registros de segmento, pero, aparte de eso, designan el mismo conjunto de registros y, como tales, también tienen una parte invisible. Estas partes se llenan con valores fijos, pero para la base que se deriva del valor que se acaba de cargar.

En tal vista, el modo real es solo un caso especial de modo protegido: donde la información de un segmento, como la base y el límite, se obtiene sin un GDT / LDT pero aún se lee desde la parte oculta del registro de segmentos.

Cambiando en modo protegido y creando un GDT es posible crear un segmento con los atributos deseados, por ejemplo, una base de 0 y un límite de 4GiB.
A través de una carga sucesiva de un registro selector, dichos atributos se almacenan en caché, luego es posible volver a conmutar en modo real y tener un registro de segmento a través del cual acceder a todo el espacio de direcciones de 32 bits.

```
BITS 16

jmp 7c0h:__START__

__START__:
 push cs
 pop ds
 push ds
 pop ss
 xor sp, sp


 lgdt [GDT]            ;Set the GDTR register


 cli                ;We don't have an IDT set, we can't handle interrupts


 ;Entering protected mode

 mov eax, cr0
 or ax, 01h            ;Set bit PE (bit 0) of CR0
 mov cr0, eax            ;Apply

 ;We are now in Protected mode

 mov bx, 08h           ;Selector to use, RPL = 0, Table = 0 (GDT), Index = 1

 mov fs, bx            ;Load FS with descriptor 1 info
 mov gs, bx            ;Load GS with descriptor 1 info
```

```
;Exit protected mode

and ax, 0fffeh            ;Clear bit PE (bit0) of CR0
mov cr0, eax                   ;Apply

sti

;Back to real mode

;Do nothing
cli
hlt



GDT:
   ;First entry, number 0
   ;Null descriptor
   ;Used to store a m16&32 object that tells the GDT start and size

   dw 0fh                 ;Size in byte -1 of the GDT (2 descriptors = 16 bytes)
   dd GDT + 7c00h         ;Linear address of GDT start (24 bits)
   dw 00h                 ;Pad

   dd 0000ffffh           ;Base[15:00] = 0, Limit[15:00] = 0ffffh
   dd 00cf9200h           ;Base[31:24] = 0, G = 1, B = 1, Limit[19:16] = 0fh,
              ;P = 1, DPL = 0, E = 0, W = 1, A = 0, Base[23:16] = 00h


TIMES 510-($-$$) db 00h
dw 0aa55h
```

Consideraciones

- Tan pronto como se vuelve a cargar un registro de segmento, incluso con el mismo valor, el procesador vuelve a cargar los atributos ocultos de acuerdo con el modo actual. Esta es la razón por la que el código anterior utiliza `fs` y `gs` para contener los segmentos "extendidos": tales registros tienen menos probabilidades de ser utilizados / guardados / restaurados por los diversos servicios de 16 bits.
- La instrucción `lgdt` no carga un puntero lejano al GDT, en su lugar carga una *dirección lineal de* 24 bits (puede anularse a 32 bits). Esta no es una *dirección cercana* , es la *dirección física* (ya que la paginación debe estar deshabilitada). Esa es la razón de `GDT+7c00h` .
- El programa anterior es un cargador de arranque (para MBR, no tiene BPB) que establece `cs` / `ds` / `ss` tp *7c00h* e inicia el contador de ubicación a partir de 0. Entonces, un byte en el offset *X* en el archivo está en offset *X* en el segmento *7c00h* y En la dirección lineal *7c00h* + *X.*
- Las interrupciones deben estar deshabilitadas ya que un IDT no está configurado para el viaje de ida y vuelta corto en modo protegido.
- El código utiliza un truco para guardar 6 bytes de código. La estructura cargada por `lgdt` se guarda en el ... GDT mismo, en el descriptor nulo (el primer descriptor).

Para obtener una descripción de los descriptores GDT, consulte el Capítulo 3.4.3 de Intel Manual Volume 3A .

Lea Modos Real vs Protegido en línea: https://riptutorial.com/es/x86/topic/3679/modos-real-vs-protegido

Capítulo 12: Paginación - Direccionamiento Virtual y Memoria

Examples

Introducción

Historia

Las primeras computadoras

Las primeras computadoras tenían un bloque de memoria en el que el programador colocaba el código y los datos, y la CPU se ejecutaba en este entorno. Dado que las computadoras eran muy caras, fue desafortunado que hiciera un trabajo, detuviera y esperara a que se cargara el siguiente trabajo y luego procesara ese.

Multiusuario, multiprocesamiento

Así, las computadoras rápidamente se volvieron más sofisticadas y admitieron múltiples usuarios y / o programas simultáneamente, pero ahí fue cuando empezaron a surgir problemas con la simple idea de "un bloque de memoria". Si una computadora ejecutaba dos programas a la vez, o si ejecutaba el mismo programa para múltiples usuarios (lo que, por supuesto, habría requerido datos separados para cada usuario), entonces la administración de esa memoria se volvió crítica.

Ejemplo

Por ejemplo: si un programa fue escrito para funcionar en la dirección de memoria 1000, pero otro programa ya estaba cargado, entonces el nuevo programa no pudo cargarse. Una forma de resolver esto sería hacer que los programas funcionen con "direccionamiento relativo": no importaba dónde se cargaba el programa, simplemente hacía todo lo relacionado con la dirección de memoria en la que estaba cargado. Pero eso requería soporte de hardware.

Sofisticación

A medida que el hardware de la computadora se hizo más sofisticado, fue capaz de admitir bloques más grandes de memoria, permitiendo más programas simultáneos, y se volvió más complicado escribir programas que no interfirieran con lo que ya estaba cargado. Una referencia de memoria extraviada podría reducir no solo el programa actual, sino también cualquier otro programa en la memoria, ¡incluido el propio sistema operativo!

Soluciones

Lo que se necesitaba era un mecanismo que permitiera a los bloques de memoria tener direcciones *dinámicas* . De esa manera, se podría escribir un programa para trabajar con sus bloques de memorias en direcciones que reconoció, y no poder acceder a otros bloques para otros programas (a menos que alguna cooperación lo permitiera).

Segmentación

Un mecanismo que implementó esto fue la segmentación. Eso permitió que se definieran bloques de memoria de todos los tamaños diferentes, y el programa tendría que definir a qué segmento quería acceder todo el tiempo.

Problemas

Esta técnica era poderosa, pero su misma flexibilidad era un problema. Dado que los segmentos esencialmente subdividieron la memoria disponible en trozos de diferentes tamaños, entonces la administración de la memoria para esos segmentos fue un problema: asignación, desasignación, crecimiento, reducción, fragmentación: todo esto requería rutinas sofisticadas y, a veces, copias en masa para implementar.

Paginacion

Una técnica diferente dividió toda la memoria en bloques de igual tamaño, llamados "Páginas", que hicieron que las rutinas de asignación y desasignación fueran muy simples, y eliminaron el crecimiento, la reducción y la fragmentación (excepto la fragmentación interna, que es simplemente un problema de pérdida).

Direccionamiento virtual

Al dividir la memoria en estos bloques, se podrían asignar a diferentes programas según sea necesario con cualquier dirección que el programa lo necesite. Esta "asignación" entre la dirección física de la memoria y la dirección deseada del programa es muy poderosa y es la base de la gestión de memoria de todos los procesadores principales (Intel, ARM, MIPS, Power y otros) en la actualidad.

Soporte de hardware y sistema operativo

El hardware realizó la reasignación de forma automática y continua, pero requirió memoria para definir las tablas de qué hacer. Por supuesto, la limpieza asociada con esta reasignación tuvo que ser controlada por algo. El sistema operativo tendría que distribuir la memoria según sea necesario y administrar las tablas de datos requeridas por el hardware para admitir lo que los programas requerían.

Características de paginación

Una vez que el hardware pudo hacer esta reasignación, ¿qué permitió? El controlador principal era el multiprocesamiento: la capacidad de ejecutar varios programas, cada uno con su "propia" memoria, protegidos entre sí. Pero otras dos opciones incluían "datos dispersos" y "memoria virtual".

Multiprocesamiento

A cada programa se le asignó su propio "Espacio de direcciones" virtual, un rango de direcciones a las que se les podía asignar memoria física, en cualquier dirección deseada. Siempre que haya suficiente memoria física para circular (aunque vea "Memoria virtual" a continuación), se podrían admitir numerosos programas simultáneamente.

Además, esos programas ***no podían*** acceder a la memoria que no estaba asignada a su espacio de direcciones virtuales; la protección entre programas era automática. Si los programas necesitaban comunicarse, podrían pedir al sistema operativo que organice un bloque de memoria compartido, un bloque de memoria física que se asignó en dos espacios de direcciones de programas diferentes simultáneamente.

Datos escasos

Permitir un gran espacio de direcciones virtuales (4 GB es típico, para corresponder con los registros de 32 bits que típicamente tienen estos procesadores) no desperdicia memoria en sí misma, si grandes áreas de ese espacio de direcciones no se asignan. Esto permite la creación de enormes estructuras de datos donde solo algunas partes se asignan a la vez. Imagine una matriz tridimensional de 1,000 bytes en cada dirección: ¡eso tomaría normalmente mil millones de bytes! Pero un programa podría reservar un bloque de su espacio de direcciones virtuales para "retener" estos datos, pero solo asignar pequeñas secciones a medida que se rellenaban. Esto hace que la programación sea eficiente, mientras que no desperdicie memoria en datos que aún no son necesarios.

Memoria virtual

Anteriormente, utilicé el término "direccionamiento virtual" para describir el direccionamiento virtual a físico realizado por el hardware. A menudo, esto se denomina "memoria virtual", pero ese término corresponde más correctamente a la técnica de uso del direccionamiento virtual para brindar una ilusión de más memoria de la que realmente está disponible.

Funciona así:

- A medida que los programas se cargan y solicitan más memoria, el sistema operativo proporciona la memoria de lo que tiene disponible. Además de realizar un seguimiento de la memoria asignada, el sistema operativo también realiza un seguimiento de cuándo se utiliza realmente la memoria: el hardware admite el marcado de páginas usadas.

- Cuando el sistema operativo se queda sin memoria física, mira toda la memoria que ya ha entregado para la página que menos se usó, o que no se usó por más tiempo. Guarda el contenido de esa página en particular en el disco duro, recuerda dónde estaba, lo marca como "No presente" en el hardware para el propietario original, y luego pone a cero la página y se la entrega al nuevo propietario.
- Si el propietario original intenta acceder a esa página nuevamente, el hardware notifica al sistema operativo. El sistema operativo luego asigna una nueva página (¡quizás tenga que realizar el paso anterior nuevamente!), Carga el contenido de la antigua página y luego entrega la nueva página al programa original.

    > El punto importante a tener en cuenta es que, como cualquier página puede asignarse a cualquier dirección y cada página tiene el mismo tamaño, entonces una página es tan buena como cualquier otra, ¡siempre que el contenido siga siendo el mismo!

- Si un programa accede a una ubicación de memoria no asignada, el hardware notifica al sistema operativo como antes. Esta vez, el SO nota que no fue una página que se había guardado, por lo que la reconoce como un error en el programa, ¡y la termina!

    > Esto es realmente lo que sucede cuando su aplicación se desvanece misteriosamente en usted, tal vez con un MessageBox del sistema operativo. También es lo que (a menudo) causa una infame pantalla azul o Sad Mac: ¡el programa de buggy era en realidad un controlador de sistema operativo que accedía a la memoria y no debería!

Decisiones de paginación

Los arquitectos de hardware necesitaban tomar algunas decisiones importantes sobre la paginación, ya que el diseño afectaría directamente el diseño de la CPU. Un sistema muy flexible tendría una gran sobrecarga, que requeriría grandes cantidades de memoria solo para administrar la infraestructura de paginación en sí.

¿Qué tan grande debe ser una página?

En hardware, la implementación más sencilla de Paging sería tomar una dirección y dividirla en dos partes. La parte superior sería un indicador de a qué página acceder, mientras que la parte inferior sería el índice en la página para el byte requerido:

```
+-----------------+------------+
| Page index      | Byte index |
+-----------------+------------+
```

Sin embargo, rápidamente se hizo evidente que las páginas pequeñas requerirían vastos índices para cada programa: incluso la memoria que no estaba asignada necesitaría una entrada en la tabla que lo indicara.

Así que en su lugar se utiliza un índice de múltiples niveles. La dirección se divide en varias partes (tres se indican en el ejemplo a continuación), y la parte superior (comúnmente llamada "Directorio") se indexa en la siguiente parte y así sucesivamente hasta que se decodifique el índice de bytes final en la página final:

```
+-----------+------------+------------+
| Dir index | Page index | Byte index |
+-----------+------------+------------+
```

Eso significa que un índice de directorio puede indicar "no asignado" para una gran parte del espacio de direcciones, sin requerir numerosos índices de página.

¿Cómo optimizar el uso de las tablas de páginas?

Deberá asignarse cada acceso a la dirección que hará la CPU, por lo que el proceso virtual a físico debe ser lo más eficiente posible. Si se implementara el sistema de tres niveles descrito anteriormente, eso significaría que cada acceso a la memoria sería en realidad tres accesos: uno en el Directorio; uno en la tabla de páginas; Y luego, finalmente, los datos deseados en sí. Y si la CPU también necesita realizar tareas de limpieza, como indicar que esta página ya se ha accedido o escrito, entonces se necesitarían más accesos para actualizar los campos.

La memoria puede ser rápida, pero esto impondría una desaceleración triple en todos los accesos de memoria durante la paginación. Afortunadamente, la mayoría de los programas tienen una "localidad de alcance", es decir, si acceden a una ubicación en la memoria, es probable que los accesos futuros estén cerca. Y dado que las Páginas no son demasiado pequeñas, esa conversión de mapeo solo debería realizarse cuando se accedió a una Página nueva: no para absolutamente todos los accesos.

Pero incluso mejor sería implementar un caché de páginas de acceso reciente, no solo la más actual. El problema sería mantenerse al tanto de las páginas a las que se había accedido y de las que no: el hardware tendría que escanear a través de la caché en cada acceso para encontrar el valor almacenado en caché. Por lo tanto, la memoria caché se implementa como una memoria de contenido direccionable: en lugar de acceder a ella por dirección, se accede a ella mediante el contenido; si los datos solicitados están presentes, se ofrece, de lo contrario, se marca una ubicación vacía para que se complete. El caché maneja todo eso.

Este caché de contenido direccionable a menudo se denomina un búfer de traducción (TLB), y debe ser administrado por el sistema operativo como parte del subsistema de direccionamiento virtual. Cuando el sistema operativo modifica los directorios o las tablas de páginas, debe notificar a la TLB para actualizar sus entradas, o simplemente para invalidarlas.

80386 Paginación

Diseño de alto nivel

El 80386 es un procesador de 32 bits, con un espacio de memoria direccionable de 32 bits. Los

diseñadores del subsistema de paginación señalaron que un diseño de página 4K se asignó a esos 32 bits de forma bastante ordenada: 10 bits, 10 bits y 12 bits:

```
+-----------+------------+------------+
| Dir index | Page index | Byte index |
+-----------+------------+------------+
 3         2 2          1 1          0  Bit
 1         2 1          2 1          0  number
```

Eso significaba que el índice de bytes era de 12 bits de ancho, lo que se indexaría en una página 4K. Los índices del Directorio y de la Página eran de 10 bits, que se mapearían en una tabla de 1,024 entradas, y si esas entradas de la tabla fueran de 4 bytes, eso sería 4K por tabla: ¡también una Página!

Así que eso es lo que hicieron:

- Cada programa tendría su propio Directorio, una Página con 1.024 Entradas de Página que definían cada una de las siguientes páginas: si hubiera una.
- Si existiera, esa tabla de páginas tendría 1.024 entradas de página que cada una definía donde estaba la página del último nivel, si hubiera una.
- Si existiera, entonces esa Página podría tener su Byte directamente leído.

Entrada de página

Tanto el Directorio de nivel superior como la Tabla de páginas del siguiente nivel se componen de 1,024 Entradas de página. La parte más importante de estas entradas es la dirección de lo que está indexando: una tabla de páginas o una página real. Tenga en cuenta que esta dirección no necesita los 32 bits completos, ya que todo es una página, solo los 20 bits principales son significativos. Por lo tanto, los otros 12 bits en la Entrada de página se pueden usar para otras cosas: si el siguiente nivel está incluso presente; limpieza de si la página ha sido visitada o escrita; ¡E incluso si las escrituras deberían ser permitidas!

```
+--------------+----+------+-----+---+---+
| Page Address | OS | Used | Sup | W | P |
+--------------+----+------+-----+---+---+
Page Address = Top 20 bits of Page Table or Page address
OS           = Available for OS use
Used         = Whether this page has been accessed or written to
Sup          = Whether this page is Supervisory - only accessible by the OS
W            = Whether this page is allowed to be Written
P            = Whether this page is even Present
```

Tenga en cuenta que si el bit `P` es 0, entonces el resto de la entrada puede tener cualquier cosa que el sistema operativo quiera poner allí, como por ejemplo, donde el contenido de la página debería estar en el disco duro.

Page Directory Base de Registro (PDBR)

Si cada programa tiene su propio Directorio, ¿cómo sabe el hardware dónde comenzar el mapeo? Dado que la CPU solo ejecuta un programa a la vez, tiene un único Registro de control para mantener la dirección del Directorio del programa actual. Este es el Registro de Base de Directorio de Páginas (`CR3`) A medida que el SO cambia entre diferentes programas, actualiza el `PDBR` con el Directorio de páginas correspondiente al programa.

Fallas de la página

Cada vez que la CPU accede a la memoria, debe asignar la dirección virtual indicada a la dirección física apropiada. Este es un proceso de tres pasos:

1. Indexe los 10 bits principales de la dirección en la página indicada por el `PDBR` para obtener la dirección de la tabla de páginas apropiada;
2. Indexe los siguientes 10 bits de la dirección en la Página indicada por el Directorio para obtener la dirección de la Página correspondiente;
3. Indexe los últimos 12 bits de la dirección para obtener los datos de esa página.

Debido a que los pasos 1. y 2. anteriores utilizan entradas de página, cada entrada podría indicar un problema:

- El siguiente nivel puede estar marcado "No presente";
- El siguiente nivel puede marcarse como "Sólo lectura", y la operación es una escritura;
- El siguiente nivel puede estar marcado como "Supervisor", y es el programa que accede a la memoria, no al sistema operativo.

Cuando el hardware detecta un problema de este tipo, en lugar de completar el acceso, genera un error: Interrumpir n. ° 14, el error de página. También completa algunos Registros de control específicos con la información de por qué ocurrió la falla: la dirección a la que se hace referencia; si era un acceso de Supervisor; y si fue un intento de escritura.

Se espera que el sistema operativo atrape esa falla, decodifique los registros de control y decida qué hacer. Si se trata de un acceso no válido, puede terminar el programa de fallas. Sin embargo, si se trata de un acceso a la memoria virtual, el sistema operativo debería asignar una nueva página (¡es posible que deba desocupar una página que ya esté en uso!), Llenarla con el contenido requerido (todos los ceros o el contenido anterior cargado desde el disco)), asigne la nueva página en la tabla de páginas apropiada, márquela como presente y luego reanude la instrucción de fallas. Esta vez, el acceso avanzará con éxito, y el programa continuará sin saber que ocurrió algo especial (¡a menos que eche un vistazo al reloj!)

80486 Paginación

El subsistema de paginación 80486 era muy similar al de 80386. Era compatible con versiones anteriores, y las únicas características nuevas eran permitir el control de la memoria caché página por página; los diseñadores del sistema operativo podían marcar páginas específicas para que no se almacenaran en la caché, o usar diferentes escrituras o reescrituras Técnicas de almacenamiento en caché.

En todos los demás aspectos, es aplicable el ejemplo "80386 Paginación".

Paginación Pentium

Cuando se estaba desarrollando el Pentium, el tamaño de la memoria y los programas que se ejecutaban en ellos se hacían más grandes. El sistema operativo tenía que hacer cada vez más trabajo para mantener el subsistema de paginación solo en la cantidad de índices de páginas que debían actualizarse cuando se utilizaban grandes programas o conjuntos de datos.

Así que los diseñadores de Pentium agregaron un truco simple: pusieron un bit extra en las Entradas del Directorio de páginas que indicaban si el siguiente nivel era una Tabla de páginas (como antes), ¡o si iban directamente a una página de 4 MB! Al tener el concepto de 4 MB de páginas, el sistema operativo no tendría que crear una tabla de páginas y rellenarla con 1.024 entradas que básicamente eran direcciones de indexación 4K más altas que la anterior.

Diseño de la dirección

```
+-----------+----------------------+
| Dir Index | 4MB Byte Index       |
+-----------+----------------------+
 3         2 2                    0   Bit
 1         2 1                    0   number
```

Diseño de entrada de directorio

```
+-----------+----+---+------+-----+---+---+
| Page Addr | OS | S | Used | Sup | W | P |
+-----------+----+---+------+-----+---+---+
Page Addr = Top 20 bits of Page Table or Page address
OS        = Available for OS use
S         = Size of Next Level: 0 = Page Table, 1 = 4 MB Page
Used      = Whether this page has been accessed or written to
Sup       = Whether this page is Supervisory - onlly accessible by the OS
W         = Whether this page is allowed to be Written
P         = Whether this page is even Present
```

Por supuesto, eso tuvo algunas ramificaciones:

- La página de 4 MB tenía que comenzar en un límite de dirección de 4 MB, al igual que las páginas de 4K tenía que comenzar en un límite de dirección de 4K.
- Todos los 4 MB tenían que pertenecer a un solo programa, o ser compartidos por varios.

Esto era perfecto para el uso de periféricos de gran memoria, como adaptadores de gráficos, que tenían grandes ventanas de espacio de direcciones que necesitaban ser asignadas para que las usara el sistema operativo.

Extensión de dirección física (PAE)

Introducción

A medida que disminuían los precios de la memoria, las PC basadas en Intel podían tener más y más RAM de manera asequible, aliviando los problemas de muchos usuarios al ejecutar muchas de las aplicaciones cada vez más grandes que se producían simultáneamente. Mientras que la memoria virtual permitía que la memoria se "creara" virtualmente - el intercambio de los contenidos "antiguos" existentes en el disco duro para permitir que se almacenen los datos "nuevos" - esto ralentizó la ejecución de los programas, ya que la página "golpeando" se mantiene continuamente intercambiando datos Encendido y apagado del disco duro.

Más memoria RAM

Lo que se necesitaba era la capacidad de acceder a más RAM física, pero ya era un bus de direcciones de 32 bits, por lo que cualquier aumento requeriría registros de direcciones más grandes. ¿O sería? Al desarrollar el Pentium Pro (e incluso el Pentium M), como una interrupción hasta que se puedan producir procesadores de 64 bits, para agregar más bits de dirección física (lo que permite más memoria física) *sin* cambiar el número de bits de registro. Esto podría lograrse ya que las direcciones virtuales se asignaron a direcciones físicas de todos modos, todo lo que se necesitaba para cambiar era el sistema de mapeo.

Diseño

El sistema existente podría acceder a un máximo de 32 bits de direcciones físicas. Aumentar esto requiere un cambio completo de la estructura de Entrada de página, de 32 a 64 bits. Se decidió mantener la granularidad mínima en páginas 4K, por lo que la entrada de 64 bits tendría 52 bits de dirección y 12 bits de control (como la entrada anterior tenía 20 bits de dirección y 12 bits de control).

Tener una entrada de 64 bits, pero un tamaño de página de (aún) 4K, significaba que solo habría 512 entradas por tabla de página o directorio, en lugar de las 1.024 anteriores. Eso significaba que la dirección virtual de 32 bits se dividiría de manera diferente a antes:

```
+-----+-----------+------------+------------+
| DPI | Dir Index | Page Index | Byte Index |
+-----+-----------+------------+------------+
 3   3 2         2 2          1 1          0   Bit
 1   0 9         1 0          2 1          0   number

 DPI        = 2-bit index into Directory Pointer Table
 Dir Index  = 9-bit index into Directory
 Page Index = 9-bit index into Page Table
 Byte Index = 12-bit index into Page (as before)
```

Al cortar un bit del Índice de directorio y del Índice de página, se obtuvieron dos bits para un tercer nivel de asignación: llamaron a esto la Tabla de punteros del directorio de páginas (PDPT), una tabla de exactamente cuatro entradas de 64 bits que se dirigían a cuatro directorios en lugar de la

anterior. uno. El PDBR (`CR3`) ahora apuntaba al PDPT en su lugar, que, dado que `CR3` era solo de 32 bits, necesitaba ser almacenado en los primeros 4 GB de RAM para accesibilidad. Tenga en cuenta que, dado que los bits bajos de `CR3` se utilizan para el control, el PDPT debe comenzar en un límite de 32 bytes.

Extensión de tamaño de página (PSE)

Y, dado que las páginas de 4 MB anteriores eran una buena idea, querían poder admitir páginas grandes de nuevo. Sin embargo, esta vez, la eliminación de la última capa del sistema de niveles no produjo 10 + 12 bit 4MB Pages, sino 9 + 12 bit 2MB Pages.

PSE-32 (y PSE-40)

Dado que el modo de Extensión de Dirección Física (PAE) que se introdujo en el Pentium Pro (y Pentum M) fue un cambio en el subsistema de administración de memoria del Sistema Operativo, cuando Intel diseñó el Pentium II, decidieron mejorar el modo de página "normal" para admite los nuevos bits de dirección física del procesador dentro de las entradas de 32 bits definidas anteriormente.

Se dieron cuenta de que cuando se utilizaba una página de 4 MB, la entrada de directorio tenía este aspecto:

```
+-----------+------------+---------+
| Dir Index |  Unused    | Control |
+-----------+------------+---------+
```

Las áreas de Índice de Direccionamiento y Control de la Entrada fueron las mismas, pero el bloque de bits no utilizados entre ellos, que sería utilizado por el Índice de Página si existiera, se desperdició. ¡Así que decidieron usar esa área *para definir los bits de la dirección física superior a 31* !

```
+-----------+------+-----+---------+
| Dir Index |Unused|Upper| Control |
+-----------+------+-----+---------+
```

Esto permitió que los sistemas operativos que no adoptaron el modo PAE tuvieran acceso a una RAM superior a 4 GB; con un poco de lógica adicional, podrían proporcionar grandes cantidades de RAM extra al sistema, aunque no más que los 4 GB normales de cada programa. Al principio solo se agregaron 4 bits, lo que permite un direccionamiento físico de 36 bits, por lo que este modo se denominó Extensión de tamaño de página 36 (PSE-36). En realidad no cambió el tamaño de la página, solo el direccionamiento sin embargo.

Sin embargo, la limitación de esto fue que solo se podían definir páginas de 4 MB superiores a 4 GB, no se permitían las páginas 4K. La adopción de este modo no fue amplia : se informó que era más lenta que usar PAE, y Linux nunca terminó usándolo.

Sin embargo, en procesadores posteriores que tenían incluso más bits de dirección física, tanto

AMD como Intel ampliaron el área de PSE a 8 bits, lo que algunas personas denominaron "PSE-40"

Lea Paginación - Direccionamiento Virtual y Memoria en línea:
https://riptutorial.com/es/x86/topic/3218/paginacion---direccionamiento-virtual-y-memoria

Creditos

S. No	Capítulos	Contributors
1	Primeros pasos con Intel x86 Assembly Language & Microarchitecture	Community, David Hoelzer, Peter Cordes, Peter Mortensen, PyNEwbie, Runner
2	Convenciones de llamadas	Cody Gray, icktoofay, Margaret Bloom, Michael Petch, Peter Cordes, user45891, Zopesconk
3	Convertir cadenas decimales en enteros	Margaret Bloom, MikeCAT
4	Ensambladores	John Burger
5	Flujo de control	Margaret Bloom, owacoder, Ped7g, Zopesconk
6	Fundamentos del registro	hidefromkgb, John Burger, Ped7g, Peter Cordes
7	Gestión multiprocesador	Margaret Bloom, Michael Petch, RamenChef
8	Manipulación de datos	Ped7g, Zopesconk
9	Mecanismos de llamada al sistema	owacoder
10	Mejoramiento	Cody Gray, Downvoter, faissaloo, John Burger, sannaj, Stephen Leppik
11	Modos Real vs Protegido	John Burger, Margaret Bloom
12	Paginación - Direccionamiento Virtual y Memoria	John Burger