Бесплатная электронная книга

УЧУСЬ

Intel x86 Assembly Language & Microarchitecture

Free unaffiliated eBook created from
Stack Overflow contributors.

#x86

1
1: Intel x86 2
2
Examples 2
x86 2
x86 Linux , 4
2: 7
7
7
Examples 7
MOV 7
3: 9
Examples 9
BIOS 9
BIOS 9
BIOS 9
9
: 9
(): 9
(CHS): 10
RTC ( ): 10
RTC: 10
RTC: 11
: 11
: 11
11
11
4: 13
13
13
Examples 15

15

5: 23

Examples 23

Microsoft Assembler - MASM 23

Intel 23

AT & T - 24

Borland - TASM 24

GNU - 25

Netwide Assembler - NASM 25

- YASM 26

6: 27

27

27

Examples 27

27

27

27

'sbb' 28

Pros 28

Cons 28

0 28

28

test 28

Pros 29

Cons 29

Linux 29

3 5 30

30

lea 30

Pros 30

Cons 31

7: 32
Examples 32
16- 32
: 33
32- 33
8- 34
34
34
34
?35
!35
64- 35
36
36
FLAGS 38
38
80286 39
80386 39
80486 40
Pentium 40
8: - 41
Examples 41
41
41
41
41
41
41
42
42
42

Paging 42
42
42
43
43
43
44
44
? 45
? 45
80386 46
46
47
PDBR (PDBR) 47
47
80486 48
Pentium 48
49
49
(PAE) 49
49
50
50
(PSE) 51
PSE-32 (PSE-40) 51
9: 53
Examples 53
53
53
53
53

54
54
54
55
56
56
56
56
57
57
, 58
58
() 59
59
59
60
a_label 61
61
61
10: 62
62
Examples 62
IA-32, GAS, cdecl 62
MS-DOS, TASM / MASM 16- 64
16- 64
64
64
65
NASM 66
MS-DOS, TASM / MASM 16- , , , 67
, , , 67

67

68

68

69

NASM 70

70

MS-DOS, TASM / MASM, 16- 71

16- 71

71

71

71

NASM 73

11: **74**

Examples 74

74

75

75

75

75

/ 76

76

76

! **77**

77

78

79

12: **82**

82

82

Examples 82

32- cdecl 82

83
83
83
64- V 83
83
84
84
32- stdcall 84
84
84
84
32-, cdecl - 85
(8, 16, 32) 85
(64) 85
85
32-, cdecl - 86
(float, double) 86
( ) 87
87
64- Windows 88
88
89
89
89
32-, cdecl - 89
89
( ) 90
( ) 90
90
92

Около

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

глава 1: Начало работы с Intel x86 Сборочный язык и микроархитектура

замечания

В этом разделе представлен обзор того, что такое x86, и почему разработчик может захотеть его использовать.

Следует также упомянуть любые крупные темы в x86 и ссылки на связанные темы. Поскольку документация для x86 является новой, вам может потребоваться создать начальные версии этих связанных тем.

Examples

x86 Язык ассемблера

Семейство языков ассемблера x86 представляет собой десятилетия достижений в оригинальной архитектуре Intel 8086. В дополнение к наличию нескольких разных диалектов на основе используемого ассемблера, на протяжении многих лет были добавлены дополнительные инструкции процессора, регистры и другие функции, в то же время оставаясь обратно совместимыми с 16-разрядной сборкой, используемой в 1980-х годах.

Первым шагом к работе с сборкой x86 является определение цели. Например, если вы хотите написать код в операционной системе, вы также захотите дополнительно определить, будете ли вы использовать автономный ассемблер или встроенные встроенные функции сборки языка более высокого уровня, например C. Если вы хотите скомпоновать «голый металл» без операционной системы, вам просто нужно установить ассемблер по вашему выбору и понять, как создать двоичный код, который можно превратить в флеш-память, загрузочное изображение или иначе загрузить в память на подходящее место для начала выполнения.

Очень популярным ассемблером, который хорошо поддерживается на нескольких платформах, является NASM (Netwide Assembler), который можно получить на http://nasm.us/ . На сайте NASM вы можете приступить к загрузке последней версии для вашей платформы.

Windows

Для Windows доступны 32-разрядные и 64-разрядные версии NASM. NASM поставляется с удобным установщиком, который можно использовать на вашем хосте Windows для установки ассемблера автоматически.

Linux

Вполне возможно, что NASM уже установлен в вашей версии Linux. Чтобы проверить, выполните:

```
nasm -v
```

Если команда не найдена, вам необходимо выполнить установку. Если вы не делаете что-то, что требует кратковременных функций NASM, лучший путь - использовать встроенный инструмент управления пакетами для вашего дистрибутива Linux для установки NASM. Например, в Debian-производных системах, таких как Ubuntu и другие, выполните следующее из командной строки:

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

Для систем на основе RPM вы можете попробовать:

```
sudo yum install nasm
```

Mac OS X

Последние версии OS X (включая Yosemite и El Capitan) поставляются со старой версией NASM, предварительно установленной. Например, El Capitan имеет версию 0.98.40. Хотя это, вероятно, будет работать практически во всех нормальных целях, оно на самом деле довольно старое. На этом написании выпущена версия 2.11 NASM, а в версии 2.12 имеется несколько доступных релизов.

Вы можете получить исходный код NASM из приведенной выше ссылки, но если у вас нет конкретной потребности в установке из исходного кода, гораздо проще загрузить двоичный пакет из каталога выпуска OS X и разархивировать его.

После разархивирования настоятельно рекомендуется *не* перезаписывать системную версию NASM. Вместо этого вы можете установить его в / usr / local:

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

На данный момент NASM находится в `/usr/local/bin` , но это не на вашем пути. Теперь вы должны добавить следующую строку в конец своего профиля:

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

Это добавит `/usr/local/bin` на ваш путь. Выполнение `nasm -v` в командной строке должно

теперь отображать правильную, более новую версию.

x86 Linux Привет, мир

Это базовая программа Hello World в сборке NASM для 32-разрядного x86 Linux, используя системные вызовы напрямую (без каких-либо вызовов функций libc). Это очень много, но со временем это станет понятным. Строки, начинающиеся с точки с запятой (`;`), являются комментариями.

Если вы еще не знаете низкоуровневое системное программирование Unix, вы можете просто написать функции в asm и вызвать их из программ C или C ++. Тогда вы можете просто беспокоиться о том, как обращаться с регистрами и памятью, не изучая также API-интерфейс POSIX и API ABI для его использования.

Это делает два системных вызова: `write(2)` и `_exit(2)` (а не оболочка libc `exit(3)` которая сбрасывает буферы stdio и т. Д.). (Технически, `_exit()` вызывает sys_exit_group, а не sys_exit, но это имеет значение только в многопоточном процессе .) См. Также `syscalls(2)` для документации о системных вызовах вообще и разницу между их непосредственным использованием или использованием libc обертки.

Таким образом, системные вызовы производятся путем размещения аргументов в соответствующих регистрах и номера системного вызова в `eax` , а затем выполнения команды `int 0x80` . См. Также Каковы возвращаемые значения системных вызовов в Assembly? для более подробного объяснения того, как интерфейс asm syscall документирован в основном синтаксисе Си.

Номера вызовов syscall для 32-разрядного ABI находятся в `/usr/include/i386-linux-gnu/asm/unistd_32.h` (то же самое содержимое в `/usr/include/x86_64-linux-gnu/asm/unistd_32.h`).

`#include <sys/syscall.h>` конечном итоге включит нужный файл, поэтому вы можете запустить `echo '#include <sys/syscall.h>' | gcc -E - -dM | less` чтобы увидеть макрос defs (см. этот ответ для получения дополнительной информации о поиске констант для asm в заголовках 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
```

В Linux вы можете сохранить этот файл как `Hello.asm` и создать из него 32-битный исполняемый файл с помощью следующих команд:

```
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
```

См. Этот ответ для получения дополнительной информации о сборке в 32 или 64-разрядных статических или динамически связанных Linux-исполняемых файлах, для синтаксиса NASM / YASM или синтаксиса GNU AT & T с GNU в `as` директив. (Ключевой момент: обязательно используйте `-m32` или эквивалент при построении 32-битного кода на 64-битном хосте или у вас будут проблемы с запуском во время выполнения.)

Вы можете проследить его выполнение с помощью `strace` чтобы увидеть, как он вызывает системные вызовы:

```
$ 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 +++
```

След на STDERR и регулярный выход на стандартном выводе оба собираются терминал здесь, так как они мешают в соответствии с `write` системного вызова. При необходимости перенаправляйте или трассируйте файл. Обратите внимание на то, как это позволяет нам легко увидеть значения возвратов syscall без необходимости добавлять код для их печати, и на самом деле даже проще, чем использовать обычный отладчик (например, gdb).

Версия этой программы x86-64 была бы очень похожей, передавая одни и те же аргументы на одни и те же системные вызовы, только в разных регистрах. И используя команду `syscall` вместо `int 0x80` .

Прочитайте Начало работы с Intel x86 Сборочный язык и микроархитектура онлайн: https://riptutorial.com/ru/x86/topic/1164/начало-работы-с-intel-x86-сборочный-язык-и-микроархитектура

глава 2: Манипуляция данными

Синтаксис

- ***.386*** : Сообщает **MASM** о компиляции для минимальной версии чипа x86 386.
- ***.model*** : Устанавливает модель памяти для использования, см. .MODEL .
- ***.code*** : сегмент кода, используемый для процессов, таких как основной процесс.
- ***proc*** : Объявляет процесс.
- ***ret*** : используется для успешного завершения функций, см. « Работа с возвращаемыми значениями» .
- ***endp*** : ***завершает*** декларацию процесса.
- ***public*** : делает процесс доступным для всех сегментов программы.
- ***end*** : Ends program или если используется с процессом, например, в « **end main** », делает процесс основным методом.
- ***call*** : ***вызывает*** процесс и выталкивает свой код операции в стек, см. « Управление потоком» .
- ***ecx*** : ***регистр*** счетчика, см. регистры .
- ***ecx*** : ***регистр*** счетчика.
- ***mul*** : умножает значение на eax

замечания

mov используется для передачи данных между регистрами .

Examples

Использование MOV для управления значениями

Описание:

`mov` *копирует* значения бит из аргумента источника в целевой аргумент.

Общий источник / назначение - это регистры , как правило, самый быстрый способ управления значениями с [in] CPU.

Еще одна важная группа значений source_of / destination_for - это память компьютера.

Наконец, некоторые непосредственные значения могут быть частью самой кодировки команды `mov` , экономя время отдельного доступа к памяти, считывая значение вместе с инструкцией.

На процессоре x86 в режиме 32 и 64 бит есть богатые возможности для их объединения,

особенно в различные режимы адресации памяти. Обычно копирование памяти на память выходит за пределы (за исключением специализированных инструкций, таких как `MOVSB`), и такая манипуляция требует промежуточного хранения значений в регистре [s].

Шаг 1. Настройте проект на использование **MASM** , см. « Выполнение сборки x86 в Visual Studio 2015».
Шаг 2: Введите следующее:

```
.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
```

Шаг 3: Скомпилируйте и отлаживайте.

Программа должна вернуть значение `16` .

Прочитайте Манипуляция данными онлайн: https://riptutorial.com/ru/x86/topic/8030/манипуляция-данными

глава 3: Механизмы системного вызова

Examples

Вызов BIOS

Как взаимодействовать с BIOS

Базовая система ввода / вывода или BIOS - это то, что контролирует компьютер до запуска любой операционной системы. Для доступа к услугам, предоставляемым BIOS, код сборки использует *прерывания* . Прерывание принимает форму

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

Номер прерывания должен быть между 0 и 255 (0x00 - 0xFF) включительно.

Большинство вызовов BIOS используют регистр `AH` в качестве параметра «выбор функции» и используют регистр `AL` в качестве параметра данных. Функция, выбранная `AH` зависит от вызываемого прерывания. Некоторые вызовы BIOS требуют одного 16-битного параметра в `AX` или вообще не принимают параметры и просто вызываются прерыванием. У некоторых есть еще больше параметров, которые передаются в других регистрах.

Регистры, используемые для вызовов BIOS, являются фиксированными и не могут быть взаимозаменяемы с другими регистрами.

Использование вызовов BIOS с функцией выбора

Общий синтаксис прерывания BIOS с использованием параметра выбора функции:

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

Примеры

Как написать символ на дисплей:

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

Как читать символ с клавиатуры (блокировка):

```
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
```

Как читать один или несколько секторов с внешнего диска (с использованием адресации 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
```

Как читать систему RTC (часы реального времени):

```
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
```

Как прочитать системное время из 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
```

Как читать системную дату из 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
```

Как получить размер смежной низкой памяти:

```
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
```

Как перезагрузить компьютер:

```
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.
```

Обработка ошибок

Некоторые вызовы BIOS не могут быть реализованы на каждом компьютере и не гарантируются. Часто нереализованное прерывание возвращает `0x86` или `0x80` в регистр `AH` . **Почти каждое прерывание установит флаг переноса (CF) на условие ошибки.** Это позволяет легко перейти к обработчику ошибок с помощью `jc` условного перехода. (См. Условные прыжки)

Рекомендации

Весьма исчерпывающий список вызовов BIOS и других прерываний - это список прерываний Ralf Brown . HTML-версию можно найти здесь .

Прерывания, которые часто считаются доступными, находятся в списке в Википедии .

Более подробный обзор общедоступных прерываний можно найти на сайте osdev.org

Прочитайте Механизмы системного вызова онлайн: https://riptutorial.com/ru/x86/topic/6946/механизмы-системного-вызова

глава 4: Многопроцессорное управление

параметры

Регистр LAPIC	Адрес (относительно *базы APIC*)
Местный идентификатор идентификатора APIC	+ 20h
Векторный векторный регистр прерываний	+ 0f0h
Регистр команд прерывания (ICR); бит 0-31	+ 300h
Регистр команд прерывания (ICR); бит 32-63	+ 310h

замечания

Чтобы получить доступ к LAPIC-регистрам, сегмент должен иметь возможность достичь диапазона адресов, начиная с *базы APIC* (в *IA32_APIC_BASE*).
Этот адрес является перемещаемым и теоретически может быть установлен в точке где-то в нижней памяти, что делает диапазон доступным в реальном режиме.

Циклы чтения / записи в диапазон LAPIC, однако, **не** распространяются на модуль интерфейса шины, тем самым маскируя любой доступ к адресам «позади».

Предполагается, что читатель знаком с режимом Unreal , так как он будет использоваться в некотором примере.

Также необходимо обладать навыками:

- Обработка разницы между *логическими* и *физическими* адресами [1]
- Сегментация в реальном режиме .
- Память aliasing, идентификация способности использовать разные *логические* адреса для одного и того же *физического* адреса
- Абсолютные, относительные, дальние, ближние вызовы и прыжки.
- NASM , особенно, что директива `ORG` является глобальной. Разделение кода на несколько файлов **значительно** упрощает кодирование, поскольку можно будет предоставить разные разделы разных *ORG* .

Наконец, мы предполагаем, что у процессора есть *локальный расширенный программируемый контроллер прерываний* (*LAPIC*).
Если это неоднозначно из контекста, APIC всегда означает LAPIC (е не IOAPIC, или xAPIC в целом).

Рекомендации:

- Глава 8 и 10 руководств Intel .

Битовые поля

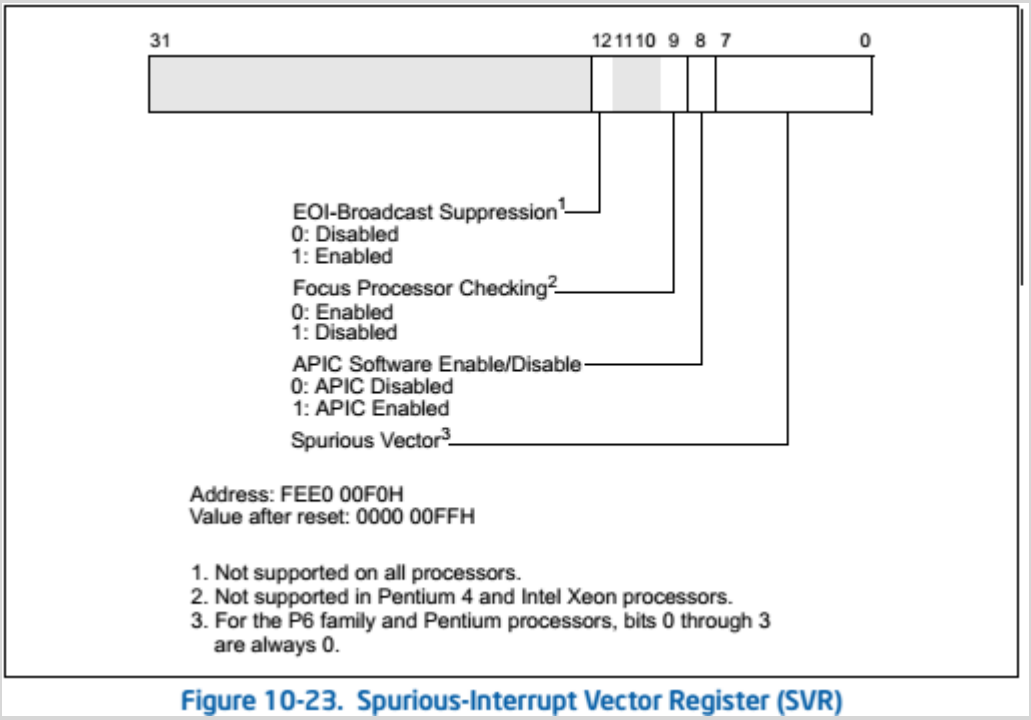


Figure 10-23. Spurious-Interrupt Vector Register (SVR)

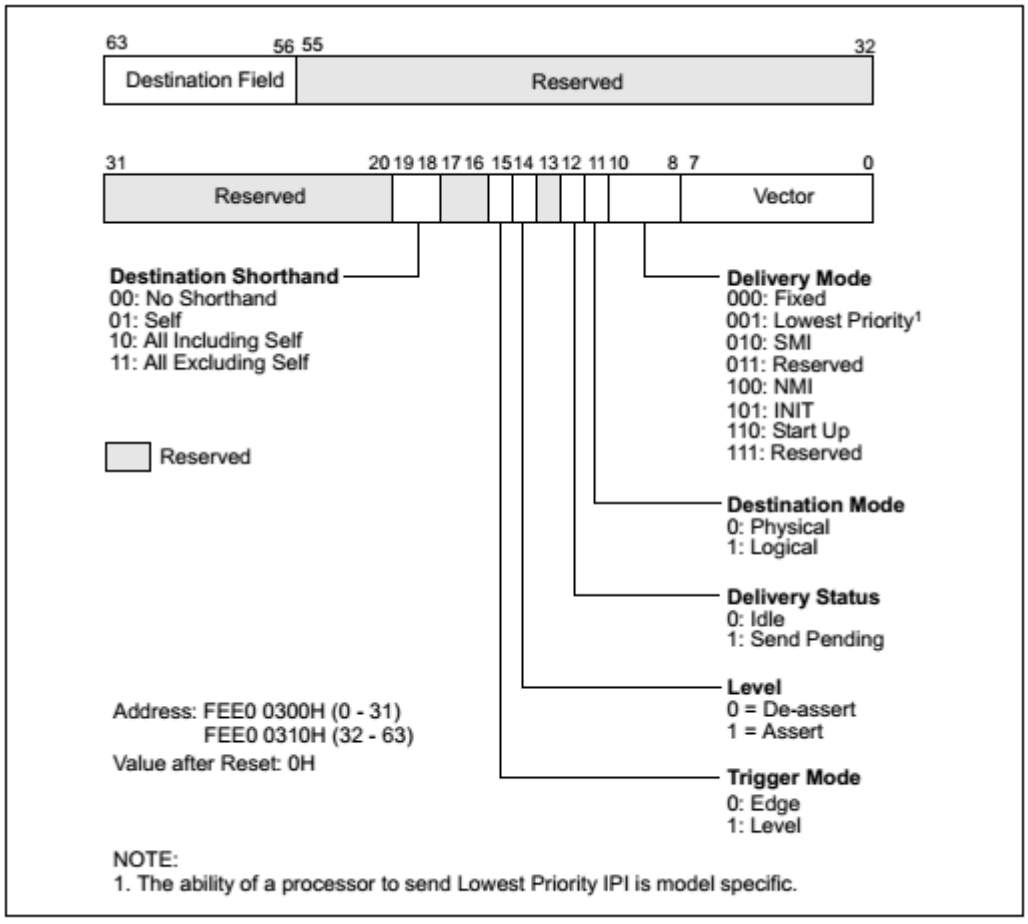


Figure 10-12. Interrupt Command Register (ICR)

Битовые поля

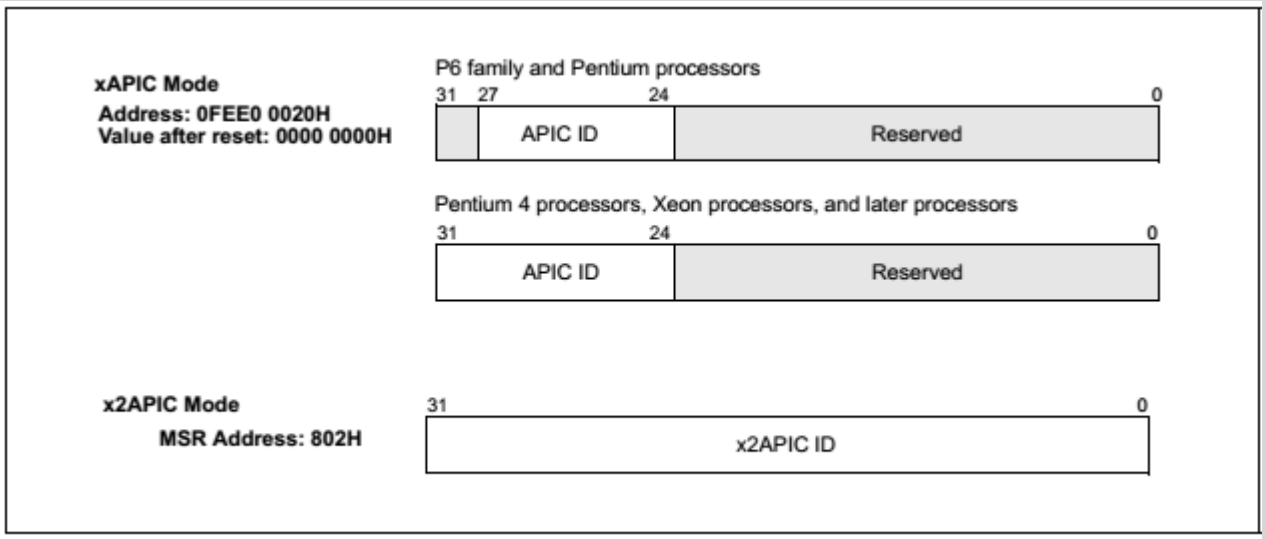


Figure 10-6. Local APIC ID Register

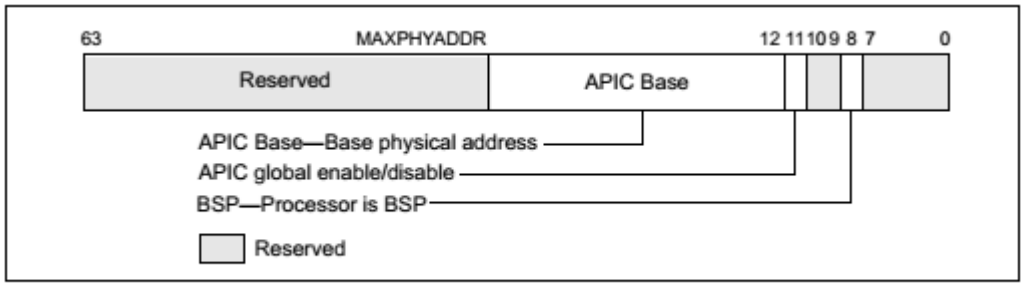


Figure 10-5. IA32_APIC_BASE MSR (APIC_BASE_MSR in P6 Family)

Название MSR	Адрес
IA32_APIC_BASE	1Bh

---

[1] Если пейджинг будет использоваться, *виртуальные* адреса также вступят в игру.

Examples

Проснись все процессоры

Этот пример пробудит каждый *процессор приложений* (AP) и сделает их вместе с *Bootstrap Processor* (BSP) отобразит их LAPIC ID.

```
; 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
```

Есть два основных шага:

1. Пробуждение точек доступа
Это достигается путем установки последовательности *INIT-SIPI-SIPI* (ISS) для всех точек доступа.

BSP, который отправит последовательность MKC, используя в качестве адресата стенографию *Все, исключая «я»* , тем самым нацеливая все точки доступа.

SIPI (Interpt Interrupt Interrupt Interrupt) игнорируется всеми процессорами, которые просыпаются к моменту их получения, поэтому второй SIPI игнорируется, если первый из них достаточно для пробуждения целевых процессоров. Intel рекомендует Intel по соображениям совместимости.

SIPI содержит *вектор* , это по смыслу, **но совершенно иное на практике** , вектору прерывания (ака номер прерывания).
Вектор представляет собой 8-битное число, значение *V* (представленное как *vv* в базе 16), что заставляет процессор *запускать* инструкции по *физическому* адресу *0vv000h* .
Мы будем называть *0vv000h адресом Wake-up* (WA).
WA принудительно на границе 4KiB (или страницы).

Мы будем использовать 08h как *V* , тогда WA будет *08000h* , 400h байт после загрузчика.

Это дает контроль над AP.

2. Инициализация и дифференциация точек доступа
В WA необходимо иметь исполняемый код. Загрузчик находится в состоянии *7c00h* , поэтому нам нужно переместить код на границе страницы.

Первое, что нужно помнить при написании полезной нагрузки, - это то, что любой доступ к общему ресурсу должен быть защищен или дифференцирован.
Общий общий ресурс - это стек , если мы инициализируем стек наивно, все точки доступа будут в конечном итоге использовать один и тот же стек!

На первом этапе используются разные адреса стека, что *отличает* стек.
Мы достигаем этого, назначая уникальное число, основанное на нуле, для каждого CPU. Это число, которое мы будем называть его *индексом* , используется для дифференциации стека, а строка - это то, что CPU будет писать свой APIC ID.

Адрес стека для каждого процессора равен *800h: (индекс * 1000h),* каждый стек AP 64KiB. Номер строки для каждого процессора является *индексом,* указатель в текстовый буфер, таким образом , 80 * 2 * *индекс.*

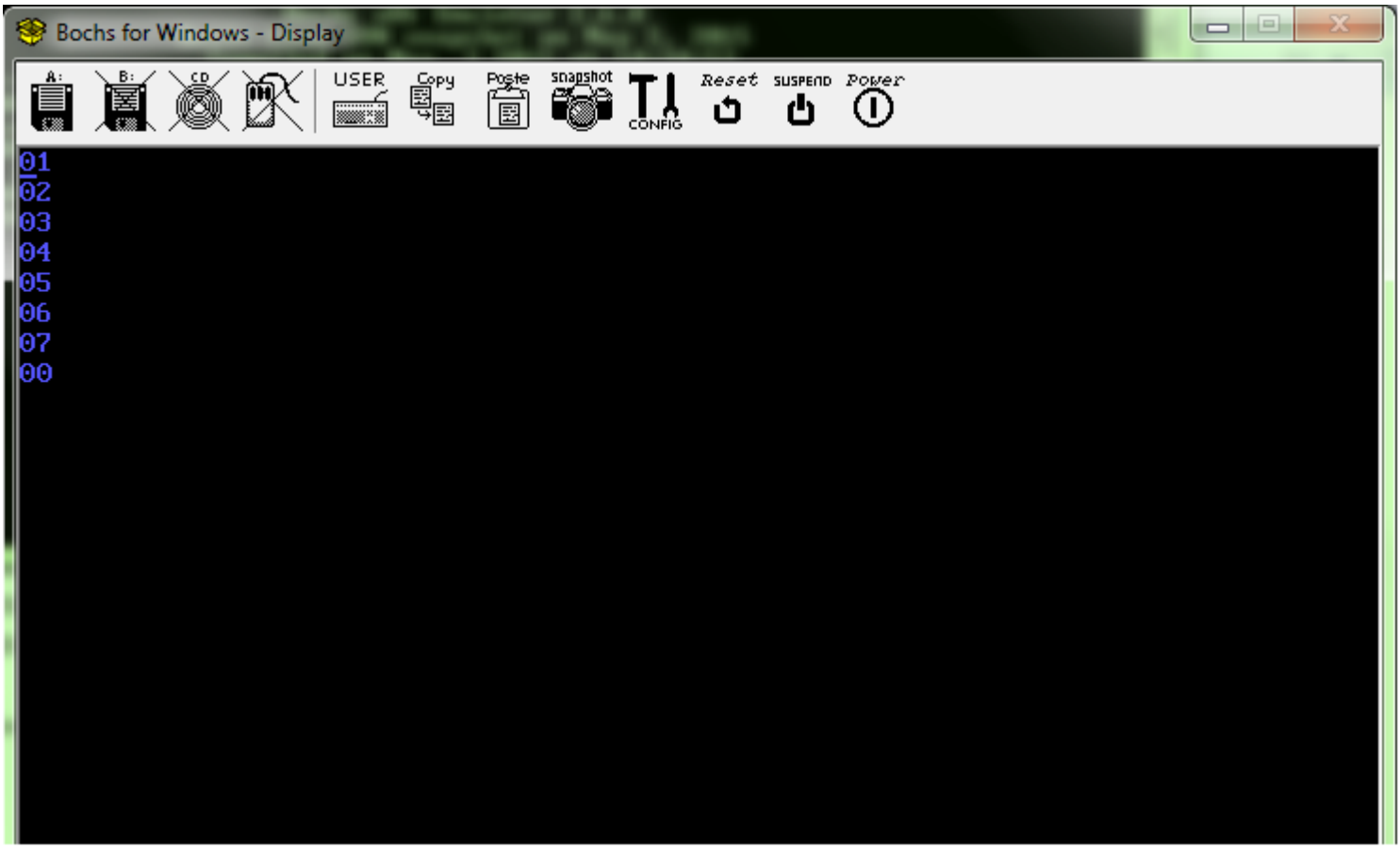
Для генерации индекса `lock cmpxchg` используется для атомарного увеличения и возврата WORD.

Итоговые заметки

- Запись на порт 80h используется для генерации задержки 1 мкс.
- `unrealmode` - это далеко `unrealmode` процедура, поэтому ее можно вызвать и после пробуждения.
- BSP также переходит к WA.

Скриншот

От Bochs с 8 процессорами



Прочитайте Многопроцессорное управление онлайн: https://riptutorial.com/ru/x86/topic/5809/многопроцессорное-управление

глава 5: Монтажники

Examples

Microsoft Assembler - MASM

Учитывая, что 8086/8088 использовался на ПК IBM, а операционная система, которая чаще всего принадлежала Microsoft, ассемблер MASM от Microsoft был стандартом де-факто в течение многих лет. Он внимательно следил за синтаксисом Intel, но допускал некоторый удобный, но «свободный» синтаксис, который (в ретроспективе) вызывал только путаницу и ошибки в коде.

Идеальный пример:

```
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  ; ????
```

Включает ли последняя инструкция `MOV` *содержимое* `Symbol` в `CX` или *адрес* `Symbol` в `CX` ? `0x1234` ли `CX` `0x1234` или `0x0102` (или что-то еще)? Оказывается, `CX` заканчивается `0x1234` - если вам нужен адрес, вам нужно использовать спецификатор `OFFSET`

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

Ассемблер Intel

Intel написала спецификацию языка ассемблера 8086, производную от предыдущих 8080, 8008 и 4004 процессоров. Таким образом, ассемблер, который они писали, точно следовал их синтаксису. Однако этот ассемблер не использовался очень широко.

Intel определила свои коды операций, чтобы иметь либо нулевой, либо один, либо два операнда. Инструкции два операнда были определены , чтобы быть в `dest` , `source` порядок, который отличается от других монтажников в то время. Но некоторые инструкции использовали неявные регистры в качестве операндов - вам просто нужно было знать, что они собой представляют. Intel также использовала концепцию опкодов «префикс» - один код операции повлиял бы на следующую инструкцию.

```
; 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 также нарушила соглашение, используемое другими ассемблерами: для каждого кода операции была изобретена другая мнемоника. Для этого требовались тонко или явно разные имена для подобных операций: например, `LDM` для «Load from Memory» и `LDI` для « Load Immediate». Intel использовала один мнемонический `MOV` - и ожидала, что ассемблер сможет определить, какой код операции использовать из контекста. Это вызвало много ошибок и ошибок для программистов в будущем, когда ассемблер не смог понять, что действительно хотел программист ...

AT & T - как

Несмотря на то, что 8086 был наиболее употребительным в IBM PC вместе с Microsoft, в нем также использовался ряд других компьютеров и операционных систем: в первую очередь Unix. Это был продукт AT & T, и у него уже был Unix, работающий на ряде других архитектур. В этих архитектурах использовался более традиционный синтаксис сборки - особенно, что инструкции из двух операндов указывали их в `source` , `dest` порядке.

Таким образом, AT & T ассемблерные соглашения перечеркнули соглашения, продиктованные Intel, и был введен целый новый диалект для диапазона x86:

- Названия регистра были префиксом `%` :
  `%al` , `%bx` и т. д.
- Непосредственные значения были получены в `$` :
  `$4`
- Операнды были в `source` , `dest` порядке
- Коды операций включали в себя размеры операндов:
  `movw $4, %ax ; Move word 4 into AX`

Турбокомпрессор Borland - TASM

Borland начал с компилятора Pascal, который они назвали «Turbo Pascal». За этим следовали компиляторы для других языков: C / C ++, Prolog и Fortran. Они также создали ассемблер под названием «Turbo Assembler», который, следуя соглашению об именах Microsoft, называется «TASM».

TASM попытался исправить некоторые проблемы написания кода с помощью MASM (см.

Выше), предоставив более строгую интерпретацию исходного кода в указанном режиме `IDEAL` . По умолчанию он принял режим `MASM` , поэтому он мог напрямую собрать источник MASM, но затем Borland обнаружил, что они должны быть ошибкой для ошибок, совместимыми с более «причудливыми» идиосинкратиями MASM, поэтому они также добавили режим `QUIRKS` .

Поскольку TASM был (намного) дешевле, чем MASM, у него была большая пользовательская база, но не многие люди использовали режим IDEAL, несмотря на его преувеличенные преимущества.

Ассемблер GNU - газ

Когда проекту GNU нужен ассемблер для семейства x86, они пошли с версией AT & T (и ее синтаксисом), которая была связана с Unix, а не с версией Intel / Microsoft.

Netwide Assembler - NASM

NASM на сегодняшний день является самым портированным ассемблером архитектуры x86 - он доступен практически для каждой операционной системы на базе x86 (даже будучи включенным в MacOS) и доступен в качестве кросс-платформенного ассемблера на других платформах.

Этот ассемблер использует синтаксис Intel, но он отличается от других, поскольку он в значительной степени фокусируется на собственном «макро» языке - это позволяет программисту создавать более сложные выражения с использованием более простых определений, позволяя создавать новые «инструкции».

К сожалению, эта мощная функция стоит дорого: тип данных мешает обобщенным инструкциям, поэтому ввод данных не выполняется.

```
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!
```

Тем не менее, NASM представила одну из функций, которой не хватало другим: имена символов в скобках. Когда вы определяете символ в других ассемблерах, это имя доступно во всей остальной части кода, но это «использует» это имя, «загрязняя» глобальное пространство имен символами.

Например (используя синтаксис NASM):

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

После этого определения X и Y навсегда определены. Чтобы избежать «использования» имен `X` и `Y` , вам нужно было использовать более определенные имена:

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

Но NASM предлагает альтернативу. Используя концепцию «локальная переменная», вы можете определить поля структуры, которые требуют, чтобы вы назначили содержащую структуру в будущих ссылках:

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

Cursor  ISTRUC      Point
        ENDISTRUC

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

К сожалению, поскольку NASM не отслеживает типы, вы не можете использовать более естественный синтаксис:

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

Еще один ассемблер - YASM

YASM - это полная перезапись NASM, но совместима с синтаксисами Intel и AT & T.

Прочитайте Монтажники онлайн: https://riptutorial.com/ru/x86/topic/2403/монтажники

глава 6: оптимизация

Вступление

Семейство x86 существует уже давно, и поэтому существует множество трюков и приемов, которые были обнаружены и разработаны, которые являются общедоступными, или, может быть, не столь публичными. Большинство этих трюков используют тот факт, что многие инструкции эффективно выполняют одно и то же, но разные версии быстрее или сохраняют память или не влияют на флаги. Вот несколько трюков, которые были обнаружены. У каждого есть свои «за» и «против», поэтому они должны быть перечислены.

замечания

Если вы сомневаетесь, вы всегда можете обратиться к довольно обширному Справочному руководству по оптимизации архитектуры Intel 64 и IA-32 , которое является отличным ресурсом от компании, стоящей за архитектурой x86.

Examples

Обнуление регистра

Очевидным способом нулевого регистра является `MOV` в `0` например:

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

Обратите внимание, что это 5-байтная инструкция.

Если вы захотите сфотографировать флаги (`MOV` никогда не влияет на флаги), вы можете использовать инструкцию `XOR` для бит-XOR для регистрации:

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

Эта инструкция требует всего 2 байта и выполняется быстрее для всех процессоров .

Перемещение флага переноса в регистр

Фон

Если флаг Carry (`C`) содержит значение, которое вы хотите поместить в регистр, наивный способ - сделать что-то вроде этого:

```
    mov  al, 1
    jc   NotZero
    mov  al, 0
NotZero:
```

Используйте 'sbb'

Более прямой путь, избегая прыжка, заключается в использовании «Вычесть с заимствованием»:

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

Если `C` равно нулю, то `al` будет равным нулю. В противном случае это будет `0xFF` (`-1`). Если вам нужно, чтобы он был `0x01` , добавьте:

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

Pros

- Примерно того же размера
- Два или несколько инструкций
- Нет дорогих прыжков

Cons

- Это непрозрачно для читателя, незнакомого с техникой
- Он изменяет другие флаги

Проверить регистр на 0

Фон

Чтобы узнать, имеет ли регистр нуль, наивная техника заключается в следующем:

```
    cmp   eax, 0
```

Но если вы посмотрите на код операции для этого, вы получите следующее:

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

Использовать `test`

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

Изучите код операции:

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

Pros

- Только два байта!

Cons

- Непрозрачный читатель, незнакомый с техникой

Вы также можете изучить Q & A Question по этой методике .

Системные вызовы Linux с меньшей раздутой

В 32-битном Linux системные вызовы обычно выполняются с помощью команды sysenter (обычно я говорю, потому что старые программы используют устаревший `int 0x80`), однако это может занимать довольно много места в программе, и поэтому есть способы, может сократить углы, чтобы сократить и ускорить работу.
Обычно это компоновка системного вызова на 32-битном Linux:

```
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
```

Это массивно! Но есть несколько трюков, которые мы можем сделать, чтобы избежать этого беспорядка.
Первое - установить значение ebp на значение esp, уменьшенное на 3 32-битных регистра, то есть 12 байтов. Это здорово, пока вы в порядке с перезаписыванием ebp, edx и ecx с помощью мусора (например, когда вы будете перемещать значение в эти регистры сразу же после этого), мы можем сделать это, используя инструкцию LEA, чтобы нам не понадобилось чтобы повлиять на значение самого ESP.

```
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
```

Однако мы не закончили, если системный вызов sys_exit, мы можем уйти, не нажимая ничего на стек!

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

Умножать на 3 или 5

Фон

Чтобы получить продукт регистра и константы и сохранить его в другом регистре, наивный способ заключается в следующем:

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

Использовать `lea`

Умножения - это дорогостоящие операции. Быстрее использовать комбинацию сдвигов и добавлений. Для конкретного случая muliplying contend 32 или 64-битного регистра, который не является `esp` или `rsp` на 3 или 5, вы можете использовать инструкцию lea. Это использует схему расчета адресов для быстрого вычисления продукта.

```
    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
```

Многие ассемблеры также поймут

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

Для всех возможных мультипликаций, отличных от `ebp` или `rbp` , полученная команда lengh такая же, как с использованием `imul` .

Pros

- Выполняется намного быстрее

Cons

- Если ваш multipicand равен `ebp` или `rbp` он принимает по одному байту больше, используя `imul`
- Подробнее введите, если ваш ассемблер не поддерживает ярлыки
- Непрозрачный читатель, незнакомый с техникой

Прочитайте оптимизация онлайн: https://riptutorial.com/ru/x86/topic/3215/оптимизация

глава 7: Основы регистрации

Examples

16-разрядные регистры

Когда Intel определила оригинальный 8086, это был 16-разрядный процессор с 20-битной адресной шиной (см. Ниже). Они определили 8 16-разрядных регистров общего назначения, но дали определенные роли для определенных инструкций:

- `AX` Аккумуляторный регистр.
  Многие коды операций либо принимали этот регистр, либо быстрее, если он был указан.
- `DX` Регистр данных.
  Иногда это сочеталось как 16 бит 32-битного значения с `AX` - например, в результате умножения.
- `CX` Регистр счетчика.
  Это использовалось в ряде ориентированных на цикл инструкций в качестве неявного счетчика для этих циклов - например, `LOOPNE` (цикл, если не равный) и `REP` (повторное перемещение / сравнение)
- `BX` Базовый регистр.
  Это можно использовать для индексации базы структуры в памяти - ни один из вышеперечисленных регистров не может использоваться для непосредственного индексации в память.
- `SI` Регистр исходного индекса.
  Это был неявный исходный индекс в памяти для определенных операций перемещения и сравнения.
- `DI` Регистр индекса назначения.
  Это был неявный индекс назначения в память для определенных операций перемещения и сравнения.
- `SP` Регистр указателя стека.
  Это наименее универсальный регистр в наборе! Он указал на текущую позицию в стеке, которая была явно использована для операций `PUSH` и `POP` , неявно для `CALL` и `RET` с подпрограммами, и ОЧЕНЬ неявно во время прерываний. Таким образом, использование его для чего-либо другого было опасным для вашей программы!
- `BP` Регистр базового указателя.
  Когда подпрограммы вызывают другие подпрограммы, стек содержит несколько «кадров стека». `BP` может использоваться для хранения текущего кадра стека, а затем, когда была вызвана новая подпрограмма, она может быть сохранена в стеке, новый стек стека создан и используется, а при возврате из внутренней подпрограммы можно восстановить значение старого фрейма стека ,

Заметки:

1. Первые три регистра не могут использоваться для индексирования в память.
2. `BX` , `SI` и `DI` по умолчанию в текущий сегмент данных (см. Ниже).

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

3. `DI` , при использовании в операциях памяти в оперативную память , такие как `MOVS` и `CMPS` , исключительно использует Добавочный сегмент (см ниже). Это нельзя переопределить.
4. `SP` и `BP` используют сегмент стека (см. Ниже) по умолчанию.

32-разрядные регистры

Когда Intel выпустила 80386, они обновились с 16-разрядного процессора до 32-разрядного. 32-битная обработка означает две вещи: обе манипулируемые данные были 32-битными, а адреса доступа к памяти были 32-разрядными. Чтобы сделать это, но по-прежнему совместимы с более ранними процессорами, они представили совершенно новые режимы для процессора. Это было либо в 16-битном режиме, либо в 32-битном режиме, но вы можете переопределить этот режим по принципу «инструкция за инструкцией» для данных, адресации или обоих!

Прежде всего, они должны были определить 32-битные регистры. Они сделали это, просто расширив существующие восемь из 16 бит до 32 бит и предоставив им «расширенные» имена с префиксом `E` : `EAX` , `EBX` , `ECX` , `EDX` , `ESI` , `EDI` , `EBP` и `ESP` . Нижние 16 бит этих регистров были такими же, как и раньше, но верхние половины регистров были доступны для 32-битных операций, таких как `ADD` и `CMP` . Верхние половинки не были доступны отдельно, как это было сделано с 8-битными регистрами.

Процессор должен был иметь отдельные 16-битные и 32-битные режимы, потому что Intel использовал одни и те же коды операций для многих операций: `CMP AX,DX` в 16-битном режиме и `CMP EAX,EDX` в 32-битном режиме имели точно такие же коды операций ! Это означало, что один и тот же код НЕ МОЖЕТ запускаться в любом режиме:

> Код операции для «Переместить немедленно в `AX` » равен `0xB8` , за которым следуют два байта `0xB8 0x12 0x34` значения: `0xB8 0x12 0x34`
>
> Код операции для «Переместить немедленно в `EAX` » равен `0xB8` , за которым следуют ***четыре*** байта непосредственного значения: `0xB8 0x12 0x34 0x56 0x78`

Таким образом, ассемблер должен знать, в каком режиме находится процессор при выполнении кода, чтобы он знал, что испускает правильное количество байтов.

8-битные регистры

Первые четыре 16-битных регистра могли бы иметь их верхнюю и нижнюю половину байт, которые были доступны непосредственно в качестве собственных регистров:

- `AH` и `AL` - это верхние и нижние половинки регистра `AX` .
- `BH` и `BL` - это верхние и нижние половинки регистра `BX` .
- `CH` и `CL` - это верхние и нижние половинки регистра `CX` .
- `DH` и `DL` - это верхние и `DX` регистры регистра `DX` .

Обратите внимание, что это означает, что изменение `AH` или `AL` немедленно изменит `AX` ! Также обратите внимание, что любая операция в 8-битном регистре не может повлиять на его «партнер» - приращение `AL` такое, что оно переполнено с `0xFF` на `0x00` , не изменит `AH` .

64-разрядные регистры также имеют 8-битные версии, представляющие их нижние байты:

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

То же самое относится к регистрам `R8` через `R15` : их соответствующие нижние части байта называются `R8B` - `R15B` .

Сегментные регистры

сегментация

Когда Intel разрабатывала оригинальные 8086, уже было множество 8-разрядных процессоров с 16-разрядными возможностями, но они хотели создать настоящий 16-разрядный процессор. Они также хотели создать что-то лучшее и более способное, чем то, что уже было там, поэтому они хотели иметь доступ к максимальному объему памяти 65 536 байт, что подразумевалось 16-разрядными регистрами адресации.

Регистры исходного сегмента

Поэтому они внедрили идею «Сегменты» - блок памяти объемом 64 килобайта, индексированный 16-разрядными регистрами адресов, которые могут быть повторно основаны на адресах различных областей общей памяти. Чтобы удерживать эти сегментные базы, они включали регистры сегментов:

- `CS` Регистр сегмента кода.
  Это содержит сегмент кода, который в настоящее время выполняется, индексируется

неявным регистром `IP` (указателем инструкций).
- `DS` Регистр сегмента данных.
  Это содержит сегмент по умолчанию для данных, которыми управляет программа.
- `ES` Регистр дополнительных сегментов.
  Это содержит второй сегмент данных для одновременных операций с данными по всей памяти.
- `SS` Регистр сегмента стека.
  Это удерживает сегмент памяти, который содержит текущий стек.

Размер сегмента?

Регистры сегментов могут быть любого размера, но их ширина 16 бит облегчает взаимодействие с другими регистрами. Следующий вопрос: должны ли сегменты перекрываться, и если да, то сколько? Ответ на этот вопрос будет определять общий объем памяти, к которому можно получить доступ.

Если бы не было перекрытия вообще, тогда адресное пространство было бы 32 бита - 4 гигабайта - совершенно неслыханный размер в то время! Более «естественное» перекрытие 8 бит создаст 24-битное адресное пространство или 16 мегабайт. В конце концов Intel решила сохранить еще четыре адресных контакта на процессоре, сделав адресное пространство 1 мегабайт с 12-битным перекрытием - они считали это достаточно большим на время!

Больше регистров сегментов!

Когда Intel разрабатывала 80386, они признали, что существующего набора из 4 регистров сегмента недостаточно для сложности программ, которые они хотели, чтобы они могли поддерживать. Поэтому они добавили еще два:

- `FS` Регистр дальнего сегмента
- `GS` Регистр глобального сегмента

Эти новые регистры сегментов не имели применений, которые использовались в процессорах: они были просто доступны для любого программиста.

> Некоторые говорят, что имена были выбраны просто для продолжения темы `C` , `D` , `E` существующего набора ...

64-разрядные регистры

AMD - производитель процессоров, который лицензировал дизайн 80386 от Intel для производства совместимых, но конкурирующих версий. Они внесли внутренние изменения в

дизайн, чтобы улучшить пропускную способность или другие улучшения в дизайне, при этом все еще можно выполнять одни и те же программы.

Для однонаправленного Intel они придумали 64-битные расширения для 32-битного дизайна Intel и выпустили первый 64-битный чип, который все еще мог бы работать с 32-разрядным кодом x86. Intel закончила дизайн AMD в своих версиях 64-битной архитектуры.

64-битная конструкция сделала ряд изменений в наборе регистров, сохраняя обратную совместимость:

- Существующие регистры общего назначения были расширены до 64 бит и названы с префиксом `R` : `RAX` , `RBX` , `RCX` , `RDX` , `RSI` , `RDI` , `RBP` и `RSP` .

  > Опять же, нижние половины этих регистров были теми же `E` prefix-регистрами, что и раньше, и верхние половины не могли быть независимо доступны.

- Добавлено еще 8 64-битных регистров и не названы, а просто пронумерованы: `R8` , `R9` , `R10` , `R11` , `R12` , `R13` , `R14` и `R15` .
  - 32-разрядная `R8D` половина этих регистров составляет от `R8D` до `R15D` (D для DWORD, как обычно).
  - К младшим 16 битам этих регистров можно было получить суффикс `W` до имени регистра: `R8W` `R15W` .
- Теперь можно получить доступ к самым низким 8 битам из *всех 16* регистров:
  - Традиционные `AL` , `BL` , `CL` и `DL` ;
  - Низкие байты (традиционно) указателя регистрируются: `SIL` , `DIL` , `BPL` и `SPL` ;
  - А низкие байты 8 новых регистров: `R8B` через `R15B` .
  - Однако `AH` , `BH` , `CH` и `DH` недоступны в инструкциях, в которых используется префикс REX (для 64-разрядного размера операнда или для доступа к R8-R15 или для доступа к `SIL` , `DIL` , `BPL` или `SPL`). С префиксом REX бит-шаблон машинного кода, который раньше означал `AH` означает `SPL` и т. Д. См. Таблицу 3-1 руководства по эксплуатации инструкции Intel (том 2).

Запись в 32-разрядный регистр всегда нулирует верхние 32 бита регистра полной ширины, в отличие от записи в 8 или 16-разрядный регистр (который сливается со старым значением, что является дополнительной зависимостью для исполнения вне порядка).

Регистрация флагов

Когда арифметический логический блок x86 (ALU) выполняет операции типа `NOT` и `ADD` , он помещает результаты этих операций («стал нулевым», «переполнен», «стал отрицательным») в специальном 16-битном регистре `FLAGS` . 32-разрядные процессоры обновили это до 32 бит и назвали его `EFLAGS` , в то время как 64-разрядные процессоры обновили его до 64 бит и назвали его `RFLAGS` .

Коды условий

Но независимо от имени, регистр не доступен напрямую (за исключением нескольких инструкций - см. Ниже). Вместо этого отдельные флаги ссылаются в определенных инструкциях, таких как условный переход или условный набор, известный как `Jcc` и `SETcc` где `cc` означает «код условия» и ссылается на следующую таблицу:

Код условия	название	Определение
`E , Z`	Равно, ноль	`ZF` == 1
`NE , NZ`	Не равно, но не ноль	`ZF` == 0
`O`	перелив	`OF` == 1
`NO`	Нет переполнения	`OF` == 0
`S`	подписанный	`SF` == 1
`NS`	Не подписано	`SF` == 0
`P`	паритет	`PF` == 1
`NP`	Нет четности	`PF` == 0
----------	----	----------
`C , B , NAE`	Нести, ниже, не выше или равно	`CF` == 1
`NC , NB , AE`	Не переносите, не ниже, выше или равно	`CF` == 0
`A , NBE`	Выше, не ниже или равно	`CF` == 0 и `ZF` == 0
`NA , BE`	Не выше, ниже или равно	`CF` == 1 или `ZF` == 1
----------	----	----------
`GE , NL`	Больше или равно, не меньше	`SF` == `OF`
`NGE , L`	Не больше или равно, меньше	`SF` ! = `OF`
`G , NLE`	Больше, не меньше или равно	`ZF` == 0 и `SF` == `OF`
`NG , LE`	Не больше, меньше или равно	`ZF` == 1 или `SF` ! = `OF`

В 16 бит вычитание `1` из `0` равно либо `65,535` либо `-1` зависимости от того, используется ли

беззнаковая или подписанная арифметика, но пункт назначения имеет значение `0xFFFF` любом случае. Только интерпретируя коды условий, значение ясно. Еще более важно, если `1` вычитается из `0x8000` : в беззнаковой арифметике, которая просто меняет `32,768` на `32,767` ; в то время как в знаковой арифметике он меняет `-32,768` в `32,767` - гораздо более примечательное переполнение!

Коды условий сгруппированы в три блока в таблице: знак-нерелевантный, неподписанный и подписанный. Именование внутри последних двух блоков использует «Выше» и «Ниже» для неподписанных, а «Большой» или «Меньше» для подписания. Таким образом, `JB` будет «Jump if Below» (без знака), тогда как `JL` будет «Jump if Less» (подписано).

Доступ к `FLAGS` напрямую

Вышеупомянутые коды условий полезны для интерпретации предопределенных понятий, но фактические биты флага также доступны непосредственно со следующими двумя инструкциями:

- `LAHF` Загрузить регистр `AH` с помощью флагов
- `SAHF` Store `AH` регистрируется во флагах

С этими инструкциями копируются только определенные флаги. Весь регистр `FLAGS` / `EFLAGS` / `RFLAGS` можно сохранить или восстановить в стеке:

- `PUSHF` / `POPF` Push / pop 16-бит `FLAGS` в / из стека
- `PUSHFD` / `POPFD` Push / pop 32-разрядные `EFLAGS` в / из стека
- `PUSHFQ` / `POPFQ` Push / pop 64-разрядные `RFLAGS` на / из стека

Обратите внимание, что прерывания сохраняют и восстанавливают текущий регистр `[R/E]FLAGS` автоматически.

Другие флаги

Помимо флагов ALU, описанных выше, регистр `FLAGS` определяет другие флагов системного состояния:

- `IF` Флаг прерывания.
  Это задано с инструкцией `STI` чтобы глобально разрешить прерывания и очиститься с помощью команды `CLI` чтобы глобально отключить прерывания.
- `DF` Флаг направления.
  Память-память операции , такие как `CMPS` и `MOVS` (для сравнения и перемещения между ячейками памяти) автоматически увеличивать или уменьшать регистры индекса в качестве части инструкции. Флаг `DF` определяет, что происходит: если он очищен командой `CLD` , они увеличиваются; если они установлены с инструкцией `STD` , они

уменьшаются.
- `TF` Флаг ловушки. Это флаг отладки. Установка этого параметра превратит процессор в режим «одноступенчатый»: после выполнения каждой команды он вызывается «Обработчик прерываний с одним шагом», который, как ожидается, будет обрабатываться отладчиком. Нет инструкций по установке или очистке этого флага: вам нужно манипулировать битом, пока он находится в памяти.

Флаги 80286

Чтобы поддерживать новые многозадачные объекты в 80286, Intel добавила дополнительные флаги в регистр `FLAGS` :

- `IOPL` Уровень привилегий ввода-вывода.
  Для защиты многозадачного кода некоторые задачи требовали привилегий для доступа к портам ввода-вывода, в то время как другим пришлось прекратить доступ к ним. Intel представила четырехуровневую шкалу Privilege, причем 00_2 является наиболее привилегированным, а 11_2 - наименее. Если `IOPL` меньше текущего уровня привилегий, любая попытка доступа к портам ввода-вывода, а также включение или отключение прерываний приведет к сбою общей защиты.
- `NT` Вложенная задача.
  Этот флаг был установлен, если одна задача `CALL` изменила другую задачу, которая вызвала контекстный переключатель. Флаг установки сказал процессору, чтобы сделать контекстный переключатель обратно, когда `RET` был выполнен.

80386 Флаги

«386 нужны дополнительные флаги для поддержки дополнительных функций, разработанных в процессоре.

- `RF` Флаг возобновления.
  В «386» добавлены регистры Debug, которые могут вызывать отладчик при различных аппаратных доступах, таких как чтение, запись или выполнение определенного местоположения memry. Однако, когда обработчик отладки вернулся для выполнения команды, *доступ немедленно повторно вызовет обработчик отладки!* Или, по крайней мере, это было бы, если бы не флаг Возобновления, который автоматически устанавливается при входе в обработчик отладки и автоматически очищается после каждой инструкции. Если флажок Resume установлен, обработчик Debug не вызывается.
- `VM` Виртуальный 8086 Флаг.
  Для поддержки старого 16-битного кода, а также более нового 32-разрядного кода 80386 может запускать 16-разрядные задачи в режиме «Виртуальный 8086» с помощью администратора Virtual 8086. Флаг `VM` указывал, что эта задача представляет собой задачу Virtual 8086.

80486 Флаги

По мере совершенствования архитектуры Intel она стала быстрее благодаря такой технологии, как кэширование и суперскалярное выполнение. Это должно было оптимизировать доступ к системе, сделав предположения. Чтобы контролировать эти предположения, требуется больше флагов:

- Флаг проверки соответствия `AC` Архитектура x86 всегда может обращаться к значениям многобайтовой памяти на любой границе байта, в отличие от некоторых архитектур, которые требовали, чтобы они были выровнены по размеру (4-байтовые значения должны были быть на 4-байтных границах). Тем не менее, это было менее эффективно, так как для доступа к неуравновешенным данным необходимы множественные обращения к памяти. Если был установлен флаг `AC` , то неприглаженный доступ приведет к возникновению исключения, а не к выполнению кода. Таким образом, код может быть улучшен во время разработки с помощью набора `AC` , но отключен для производственного кода.

Флаги Pentium

Pentium добавил больше поддержки для виртуализации, а также поддержку инструкции `CPUID` :

- `VIF` Виртуальный флаг прерывания.
  Это виртуальная копия `IF` этой задачи - должна ли эта задача отключать прерывания, фактически не влияя на глобальные прерывания.
- `VIP` Ожидающий флаг виртуального прерывания.
  Это указывает на то, что прерывание было фактически заблокировано `VIF` , поэтому, когда Task выполняет `STI` для него может быть поднято виртуальное прерывание.
- `ID` . Идентификационный флаг `CPUID` .
  Разрешить или не разрешать этой Задаче выполнять инструкцию `CPUID` . Виртуальный монитор может запретить его и «лгать» запрашивающей Задаче, если он выполняет инструкцию.

Прочитайте Основы регистрации онлайн: https://riptutorial.com/ru/x86/topic/2122/основы-регистрации

глава 8: Пейджинг - виртуальная адресация и память

Examples

Вступление

история

Первые компьютеры

Ранние компьютеры имели блок памяти, в который программист помещал код и данные, а ЦП выполнялся в этой среде. Учитывая, что компьютеры тогда были очень дорогими, было жаль, что он выполнит одно задание, остановится и дождитесь загрузки следующей работы в него, а затем обработает его.

Многопользовательская многопроцессорная обработка

Таким образом, компьютеры быстро стали более сложными и одновременно поддерживали несколько пользователей и / или программ, но при этом возникли проблемы с простой идеей «один блок памяти». Если компьютер одновременно запускал две программы или запускал одну и ту же программу для нескольких пользователей - конечно, для каждого пользователя требовались отдельные данные - тогда управление этой памятью стало критическим.

пример

Например: если программа была написана для работы с адресом памяти 1000, но там была загружена другая программа, тогда новая программа не может быть загружена. Одним из способов решения этой проблемы было бы заставить программы работать с «относительной адресацией» - неважно, где была загружена программа, она просто сделала все относительно адреса памяти, в который она была загружена. Но для этого требовалась аппаратная поддержка.

утонченность

По мере усложнения компьютерного оборудования он смог поддерживать более крупные блоки памяти, что позволяло использовать более одновременные программы, и стало

сложнее писать программы, которые не мешали тому, что уже было загружено. Одна обратная ссылка на память может снизить не только текущую программу, но и любую другую программу в памяти, включая операционную систему!

Решения

Нужен был механизм, который позволял блокам памяти иметь *динамические* адреса. Таким образом, программа может быть написана для работы со своими блоками памяти на адресах, которые она распознала, и не сможет получить доступ к другим блокам для других программ (если только это не позволило кому-то сотрудничать).

сегментация

Одним из механизмов, который реализовал это, была Сегментация. Это позволило определить блоки памяти всех разных размеров, и программа должна будет определить, какой сегмент он хочет получить доступ все время.

Проблемы

Этот метод был мощным, но его гибкость была проблемой. Поскольку сегменты по существу подразделяли доступную память на куски разного размера, тогда управление памятью для этих сегментов было проблемой: распределение, освобождение, рост, сжатие, фрагментация - все требовали сложных процедур и иногда массового копирования для реализации.

Paging

Другой метод разделил всю память на блоки равного размера, называемые «Страницы», что упростило процедуры распределения и освобождения, и ушло с ростом, сокращением и фрагментацией (за исключением внутренней фрагментации, которая является просто проблемой отходы).

Виртуальная адресация

Разделив память на эти блоки, они могут быть отнесены к различным программам по мере необходимости с любым адресом, в котором она нуждалась. Это «сопоставление» между физическим адресом памяти и желаемым адресом программы является очень мощным и является основой для управления памятью каждого основного процессора (Intel, ARM, MIPS, Power et al.).

Поддержка оборудования и ОС

Аппаратное обеспечение выполняло переназначение автоматически и постоянно, но требуемую память определяло таблицы того, что делать. Конечно, домашнее хозяйство, связанное с этим переназначением, должно контролироваться чем-то. Операционная система должна будет использовать память по мере необходимости и управлять таблицами данных, требуемыми аппаратным обеспечением, для поддержки необходимых программ.

Функции пейджинга

Как только аппаратное обеспечение может сделать это переназначение, что это позволило? Основным драйвером была многопроцессорность - возможность запускать несколько программ, каждая со своей «собственной» памятью, защищенных друг от друга. Но два других варианта включали «разреженные данные» и «виртуальную память».

многопроцессорная обработка

Каждой программе было предоставлено собственное виртуальное «адресное пространство» - целый ряд адресов, в которые они могли бы иметь физическую память, отображаемую по любым адресам. До тех пор, пока существует достаточно физической памяти (хотя см. «Виртуальная память» ниже), многочисленные программы могут поддерживаться одновременно.

Более того, эти программы ***не могли*** получить доступ к памяти, которая не была отображена в их виртуальное адресное пространство. Защита между программами была автоматической. Если программам необходимо было общаться, они могли бы попросить ОС организовать общий блок памяти - блок физической памяти, который одновременно отображался в адресные пространства двух разных программ.

Редкие данные

Разрешение огромного виртуального адресного пространства (4 ГБ типично, чтобы соответствовать 32-разрядным регистрам, которые обычно имели эти процессоры) само по себе не является ненужной памятью, если большие области этого адресного пространства остаются без изменений. Это позволяет создавать огромные структуры данных, где только определенные части отображаются в любой момент времени. Представьте себе 3-мерный массив из 1000 байт в каждом направлении: это обычно занимает миллиард байт! Но программа может зарезервировать блок своего виртуального адресного пространства для «удержания» этих данных, но только отображать небольшие разделы по мере их заполнения. Это позволяет эффективно программировать, не теряя память для данных, которые еще не нужны.

Виртуальная память

Выше я использовал термин «Виртуальная адресация» для описания виртуальной-физической адресации, выполняемой аппаратным обеспечением. Это часто называют «виртуальной памятью», но этот термин более правильно соответствует методике использования виртуального адресации для поддержки иллюзии большего объема памяти, чем на самом деле.

Он работает следующим образом:

- Когда программы загружаются и запрашивают больше памяти, ОС обеспечивает память из того, что она имеет. Помимо отслеживания того, какая память была сопоставлена, ОС также отслеживает, когда фактически используется память - аппаратное обеспечение позволяет маркировать используемые страницы.
- Когда у ОС заканчивается физическая память, она просматривает всю память, которую она уже выдала для того, какая страница использовалась наименее или не использовалась самой длинной. Он сохраняет содержимое конкретной страницы на жестком диске, запоминает, где это было, отмечает это как «Не настоящее» для аппаратного обеспечения для первоначального владельца, а затем обнуляет страницу и передает ее новому владельцу.
- Если первоначальный владелец пытается снова получить доступ к этой странице, аппаратное обеспечение уведомляет ОС. Затем ОС выделяет новую страницу (возможно, придется повторить предыдущий шаг!), Загружает содержимое старой страницы, а затем передает новую страницу в исходную программу.

    > Важно отметить, что поскольку любая страница может быть сопоставлена с любым адресом, а каждая страница имеет тот же размер, то одна страница будет такой же хорошей, как и любая другая, - пока содержимое остается неизменным!

- Если программа обращается к немаркированной ячейке памяти, аппаратное обеспечение уведомляет ОС по-прежнему. На этот раз ОС отмечает, что это была не Страница, которая была сохранена, поэтому распознает ее как ошибку в программе и завершает ее!

    > На самом деле это происходит, когда ваше приложение загадочно исчезает на вас - возможно, с MessageBox от ОС. Это также то, что (часто) приводит к печально известному Blue Screen или Sad Mac - багги-программа была фактически драйвером ОС, который получал доступ к памяти, которой он не должен!

Пейджинговые решения

Архитекторам аппаратного обеспечения необходимо было принять некоторые важные решения в отношении пейджинга, поскольку дизайн напрямую повлиял бы на дизайн процессора! Очень гибкая система будет иметь большие накладные расходы, требуя больших объемов памяти для управления самой инфраструктурой пейджинга.

Насколько велика должна быть страница?

В аппаратном обеспечении самой простой реализацией пейджинга было бы принять адрес и разделить его на две части. Верхняя часть будет индикатором доступа к странице, а нижняя часть будет индексом на странице для требуемого байта:

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

Это быстро стало очевидным, хотя для небольших страниц потребовались огромные индексы для каждой программы: даже для памяти, которая не была отображена, нужна запись в таблице, указывающая это.

Поэтому вместо этого используется многоуровневый индекс. Адрес разбит на несколько частей (три указаны в приведенном ниже примере), а верхняя часть (обычно называемая « Directory») индексируется в следующую часть и так далее до тех пор, пока окончательный индекс байта на финальную страницу не будет декодирован:

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

Это означает, что индекс Directory может указывать «не отображаемый» для огромного фрагмента адресного пространства, не требуя многочисленных индексов страниц.

Как оптимизировать использование таблиц страниц?

Каждый адресный доступ, который будет делать ЦП, должен быть сопоставлен - процесс виртуальный-физический должен быть настолько эффективным, насколько это возможно. Если трехуровневая система, описанная выше, должна быть реализована, это будет означать, что каждый доступ к памяти фактически будет иметь три доступа: один в каталог; один в таблицу страниц; а затем, наконец, желаемые данные. И если ЦП также должен был выполнять домашнее хозяйство, например, указав, что эта страница была обращена к ней или была записана, тогда для этого потребуется еще больше доступа для обновления полей.

Память может быть быстрой, но это приведет к тройному замедлению всех обращений к памяти во время пейджинга! К счастью, большинство программ имеют «локальность масштаба» - то есть, если они обращаются к одному месту в памяти, тогда будущие

обращения, вероятно, будут рядом. И поскольку страницы не слишком малы, преобразование отображения необходимо будет выполнять только при обращении к новой странице: не для абсолютно каждого доступа.

Но даже лучше было бы реализовать кэш недавно просмотренных страниц, а не только самый последний. Проблема будет идти в ногу с тем, что страницы были доступны, а что нет - аппаратное обеспечение должно было бы сканировать через кеш при каждом доступе, чтобы найти кешированное значение. Таким образом, кеш реализован в виде кэша, адресуемого контентом: вместо доступа к нему по адресу он получает доступ к содержимому - если запрашиваемые данные присутствуют, он предлагается вверх, в противном случае вместо этого помещается пустое место для заполнения. Кэш управляет всем этим.

Этот контент-адресуемый кэш часто называют буферным буфером перевода (TLB), и он должен управляться ОС как часть подсистемы виртуального адресации. Когда каталоги или таблицы страниц модифицируются ОС, необходимо уведомить TLB о необходимости обновить свои записи или просто аннулировать их.

80386 Пейджинг

Дизайн высокого уровня

80386 - это 32-разрядный процессор с 32-разрядным адресным пространством памяти. Дизайнеры подсистемы пейджинга отметили, что дизайн страницы 4 КБ сопоставлен с этими 32 битами довольно аккуратным способом - 10 бит, 10 бит и 12 бит:

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

Это означало, что индекс байта был 12 бит в ширину, который бы индексировался на страницу 4K. Индексы Directory и страницы составляли 10 бит, каждая из которых была бы отображена в таблицу с 1024 байтами, и если эти записи в таблице составляли 4 байта, это было бы 4 КБ за таблицу: также страница!

Вот что они сделали:

- Каждая программа будет иметь свой собственный каталог, страницу с 1024 страницами, каждая из которых определяет, где следующая таблица страниц уровня - если она есть.
- Если бы это было так, то на этой странице Таблица было бы 1024 Записи страниц, каждая из которых определяла бы, где была страница последнего уровня, - если

таковая была.
- Если бы это было так, то эта страница могла бы непосредственно считывать свой байт.

Вступление страницы

Оба каталога верхнего уровня и таблица страниц следующего уровня содержат 1024 записи страниц. Наиболее важной частью этих записей является адрес того, что он индексирует: Таблица страниц или фактическая страница. Обратите внимание, что для этого адреса не нужны полные 32 бита - поскольку все является страницей, важны только 20 лучших битов. Таким образом, остальные 12 бит в записи страницы могут использоваться для других вещей: присутствует ли следующий уровень; ведение домашнего хозяйства в отношении того, была ли страница доступна или написана; и даже нужно ли писать даже!

```
+--------------+----+------+-----+---+---+
| 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
```

Обратите внимание, что если бит `P` равен 0, остальная часть записи имеет право иметь все, что ОС хочет разместить там - например, где содержимое страницы должно находиться на жестком диске!

Базовый регистр `PDBR` (`PDBR`)

Если каждая программа имеет свой собственный каталог, как аппаратное обеспечение знает, с чего начать картографирование? Поскольку процессор работает только по одной программе за раз, он имеет один регистр управления для хранения адреса каталога текущей программы. Это регистр базы данных страниц (`CR3`). По мере `PDBR` ОС между различными программами он обновляет `PDBR` с помощью соответствующего каталога страниц для программы.

Ошибки страницы

Каждый раз, когда процессор обращается к памяти, он должен сопоставить указанный виртуальный адрес с соответствующим физическим адресом. Это трехэтапный процесс:

1. Индексируйте верхние 10 бит адреса на страницу, указанную `PDBR` чтобы получить адрес соответствующей таблицы страниц;
2. Индексируйте следующие 10 бит адреса на страницу, указанную Каталогом, чтобы получить адрес соответствующей страницы;
3. Индекс последних 12 бит адреса, чтобы получить данные из этой страницы.

Поскольку в обоих шагах 1 и 2. выше используются записи страниц, каждая запись может указывать на проблему:

- Следующий уровень может быть отмечен «Not Present»;
- Следующий уровень может быть отмечен как «Только для чтения» - и операция - запись;
- Следующий уровень может быть отмечен как «Супервизор» - и это программа, обращающаяся к памяти, а не к ОС.

Когда такая проблема отмечена аппаратным обеспечением, вместо завершения доступа возникает ошибка Fault: Прерывание # 14, ошибка страницы. Он также заполняет некоторые конкретные контрольные регистры информацией о том, почему произошел сбой: адрес, на который делается ссылка; будь то доступ к супервизору; и была ли это попыткой записи.

Ожидается, что ОС поймает этот Fault, расшифрует контрольные регистры и решит, что делать. Если это недопустимый доступ, он может завершить программу сбоя. Если это доступ к виртуальной памяти, ОС должна выделить новую страницу (которой может понадобиться освободить страницу, которая уже используется!), Заполнить ее необходимым содержимым (либо все нули, либо предыдущее содержимое, загруженное с диска), сопоставьте новую страницу в соответствующей Таблице страниц, пометьте ее как присутствующую, затем возобновите инструкцию по сбою. На этот раз доступ будет успешно развиваться, и программа не будет знать, что что-то особенное произошло (если только не взглянуть на часы!)

80486 Пейджинг

Подсистема подкачки 80486 была очень похожа на 80386. Он был совместим с обратной связью, и единственными новыми функциями были возможность управления кешем памяти на странице по-странице - разработчики ОС могли отмечать определенные страницы, которые нельзя кэшировать, или использовать разные записи или записи методы кеширования.

Во всех других отношениях применим пример «80386 Пейджинг».

Пейджинг Pentium

Когда разрабатывался Pentium, размеры памяти и программы, которые запускались в них, становились все больше. ОС должна была все больше и больше работать, чтобы

поддерживать подсистему подкачки только в большом количестве индексов страниц, которые необходимо было обновить, когда использовались большие программы или наборы данных.

Поэтому дизайнеры Pentium добавили простой трюк: они добавили дополнительный бит в записи каталога страниц, указав, был ли следующий уровень таблицей страниц (как и раньше) - или перешел непосредственно на страницу 4 МБ! Имея концепцию 4 МБ страниц, ОС не нужно было бы создавать таблицу страниц и заполнять ее 1024 байта, которые в основном индексировали адреса на 4 K выше предыдущего.

Макет адреса

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

Схема размещения каталога

```
+-----------+----+---+------+-----+---+---+
| 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
```

Конечно, это имело некоторые последствия:

- Страница 4 МБ должна была начинаться с границы адреса 4 МБ, так же как 4K-страницы должны были начинаться с границы адреса 4K.
- Все 4 МБ должны принадлежать одной программе - или быть разделены несколькими.

Это было идеально для использования для периферийных устройств большой памяти, таких как графические адаптеры, которые имели большие окна адресного пространства, которые необходимо было сопоставить для использования ОС.

Расширение физического адреса (PAE)

Вступление

По мере снижения цен на память компьютеры на базе Intel могли получать все больше и больше оперативной памяти, что облегчало многие проблемы пользователей при запуске многих из когда-либо больших приложений, которые выпускались одновременно. В то время как виртуальная память позволяла виртуально «создавать» память - заменять существующее «старое» содержимое страницы на жесткий диск, чтобы можно было хранить «новые» данные - это замедляло работу программ, поскольку «перерыв» страницы продолжал постоянно меняться на жестком диске и вне его.

Больше ОЗУ

Необходима была возможность доступа к большей физической памяти - но это была уже 32-разрядная адресная шина, поэтому для любого увеличения потребовались бы более крупные регистры адресов. Или это? При разработке Pentium Pro (и даже Pentium M) в качестве стоп-кадра до 64-битных процессоров можно было бы добавить больше физических битов адреса (позволяющих больше физической памяти) *без* изменения количества бит регистра. Это может быть достигнуто, так как виртуальные адреса были сопоставлены с физическими адресами в любом случае - все, что необходимо было изменить, было системой сопоставления.

дизайн

Существующая система может получить доступ к 32 битам физических адресов. Для этого потребовалось полное изменение структуры страницы, от 32 до 64 бит. Было решено сохранить минимальную детализацию на страницах 4K, поэтому в 64-битной записи будет 52 бита адреса и 12 бит элемента управления (например, предыдущая запись имела 20 бит адреса и 12 бит элемента управления).

Наличие 64-битной записи, но размер страницы (все еще) 4K означал, что вместо прежних 1024 будет только 512 записей на таблицу страниц или каталог. Это означало, что 32-битный виртуальный адрес будет разделен по-разному, чем раньше:

```
+-----+-----------+------------+------------+
| 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)
```

Измельчение одного бита из индекса каталога и индекса страницы дало два бита для третьего уровня сопоставления: они назвали эту таблицу указателей страниц (PDPT) таблицей ровно четырьмя 64-битными Записями, которые адресовали четыре Каталога, а

не предыдущие один. PDBR (`CR3`) теперь указал на PDPT вместо этого, который, поскольку `CR3` был всего 32 бита, необходимо было сохранить в первых 4 ГБ ОЗУ для обеспечения доступности. Обратите внимание, что поскольку младшие бит `CR3` используются для Control, PDPT должен начинаться с 32-байтной границы.

Расширение размера страницы (PSE)

И, поскольку предыдущие страницы 4 МБ были такой хорошей идеей, они хотели снова поддерживать крупные страницы. На этот раз, хотя удаление последнего слоя системы уровня не создало 10 + 12 бит 4 МБ страниц, но вместо 9 + 12 бит 2 МБ страниц.

PSE-32 (и PSE-40)

Поскольку режим Physical Address Extension (PAE), который был представлен в Pentium Pro (и Pentum M), был таким изменением в подсистеме управления памятью операционной системы, когда Intel разработала Pentium II, они решили повысить «нормальный» режим страницы до поддерживать новые бит физического адреса процессора в ранее определенных 32-битных Записях.

Они поняли, что, когда использовалась страница 4 МБ, запись в каталоге выглядела так:

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

Индексы Dir Index и Control в Entry были одинаковыми, но блок неиспользуемых битов между ними, который будет использоваться индексом страницы, если он существовал, был потрачен впустую. Поэтому они решили использовать эту область *для определения верхних битов физического адреса выше 31* !

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

Это позволило RAM свыше 4 ГБ быть доступным для ОС, которые не применяли режим PAE - с небольшой дополнительной логикой они могли бы обеспечить большое количество дополнительной ОЗУ для системы, хотя и не превышало нормальную 4 ГБ для каждой программы. Сначала было добавлено только 4 бита, что обеспечило 36-битную физическую адресацию, поэтому этот режим назывался расширением размера страницы 36 (PSE-36). На самом деле это не изменило размер страницы, но только Адресация.

Ограничением этого было то, что только 4 МБ страниц выше 4 ГБ были определены - 4 тыс. Страниц не разрешалось. Принятие этого режима было невеликим - он, как сообщается, был медленнее, чем использование PAE, и Linux в конечном итоге не использовал его.

Тем не менее, в более поздних процессорах, у которых было еще больше бит физического адреса, как AMD, так и Intel расширили область PSE до 8 бит, что некоторые люди называли «PSE-40»,

Прочитайте Пейджинг - виртуальная адресация и память онлайн:
https://riptutorial.com/ru/x86/topic/3218/пейджинг---виртуальная-адресация-и-память

глава 9: Поток управления

Examples

Безусловные прыжки

```
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
```

Относительные близкие прыжки

`jmp a_label` :

- **возле**
  Он указывает только смещение части *логического адреса* адресата. Сегмент считается `CS` .
- **родственник**
  Семантика команды - это переход с *перекрестными* байтами вперед [1] из следующего адреса инструкции или `IP = IP + rel` .

Инструкция кодируется как `EB <rel8>` или `EB <rel16/32>` , сборщик `EB <rel16/32>` наиболее подходящую форму, обычно предпочитая более короткую.
Возможно `jmp SHORT a_label` ассемблер, например, с помощью NASM `jmp SHORT a_label` , `jmp WORD a_label` и `jmp DWORD a_label` генерируют три возможные формы.

Абсолютное косвенное приближение

`jmp bx` и `jmp WORD [aPointer]` :

- **возле**
  Они указывают только смещенную часть логического адреса адресата. Сегмент считается `CS` .
- **абсолютное**
  Семантика инструкций переходит на адрес в *регистре* или *mem* или `IP = reg` , `IP = mem` .

Инструкция кодируется как `FF /4` , для косвенной памяти размер операнда определяется как для любого другого доступа к памяти.

Абсолютные далеко прыжки

`jmp 7c0h:0000h` :

- **далеко**
  Он определяет обе части *логического* адреса: сегмент и смещение.
- **absolute** Семантика команды - переход к *сегменту* адреса *: смещение* или `CS = segment, IP = offset` .

Инструкция кодируется как `EA <imm32/48>` зависимости от размера кода.
В некоторых ассемблерах можно выбирать между двумя формами, например, с помощью NASM `jmp 7c0h: WORD 0000h` и `jmp 7c0h: DWORD 0000h` сгенерировать первую и вторую формы.

Абсолютные косвенные дальние прыжки

`jmp FAR WORD [aFarPointer]` :

- **far** Он определяет обе части *логического* адреса: сегмент и смещение.
- **Абсолютная косвенная** семантика команды - переход к *сегменту: смещение,* сохраненное в *mem* [2] или `CS = mem[23:16/32], IP = [15/31:0]` .

Инструкция кодируется как `FF /5` , размер операнда может быть контроллером с спецификаторами размера.
В NASM, немного не интуитивно понятные, это `jmp FAR WORD [aFarPointer]` для операнда *16:16* и `jmp FAR DWORD [aFarPointer]` для операнда *16:32* .

Отсутствие прыжков

- **вблизи абсолютного**
  Можно эмулировать с почти косвенным прыжком.

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

- **далеко относительный**
  В любом случае, никакого смысла или слишком узкого использования.

---

[1] Два дополнения используются для указания смещенного смещения и, следовательно, перехода назад.
[2] Который может быть *seg16: off16* или *seg16: off32* , размеров *16:16* и *16:32* .

Условия тестирования

Для использования условного перехода необходимо проверить состояние. **Тестирование**

условия здесь относится только к действию проверки флагов, фактический прыжок описывается в разделе « Условные прыжки» .

x86 проверяет условия, полагаясь на регистр EFLAGS, который содержит набор флагов, которые каждая команда может установить.

Арифметические инструкции, такие как `sub` или `add` , и логические инструкции, такие как `xor` или, `and` , очевидно, «устанавливают флаги». Это означает, что флаги *CF* , *OF* , *SF* , *ZF* , *AF* , *PF* модифицируются этими инструкциями. Любая инструкция может изменять флаги, например, `cmpxchg` модифицирует *ZF* .

Всегда проверяйте ссылку на команду, чтобы знать, какие флаги изменены определенной инструкцией.

x86 имеет набор *условных переходов* , упомянутых ранее, которые скапливаются тогда и только тогда, когда установлены некоторые флаги, или некоторые из них ясны или оба.

Флаги

Арифметические и логические операции очень полезны при установке флагов. Например, после `sub eax, ebx` , для хранения значений **без знака** , мы имеем:

Флаг	Когда установлено	Когда
ZF	Когда результат равен нулю. *EAX - EBX = 0* ⇒ EAX = EBX	Когда результат **не** равен нулю. *EAX - EBX ≠ 0* ⇒ EAX ≠ EBX
CF	Когда результат понадобился переносить для MSb. *EAX - EBX <0* ⇒ EAX <EBX	Когда результат **не** нужно переносить для MSb. *EAX - EBX* $\not< 0 \Rightarrow \mathrm{EAX} \not< \mathrm{EBX}$
SF	Когда задан результат MSb.	Когда результат MSb **не** установлен.
O	Когда произошло переполнение подписей.	Когда подписанное переполнение **не** произошло.
PF	Когда количество бит, заданное в младшем значении, равно четному результату.	Когда число бит, заданное в младшем значении байта результата, является нечетным.
AF	Когда нижняя цифра BCD генерирует перенос. Это бит 4.	Когда нижняя цифра BCD **не** генерирует перенос. Это бит 4.

Неразрушающие испытания

`sub` и `and` инструкции изменяют свой операнд назначения и требуют двух дополнительных копий (сохранение и восстановление), чтобы сохранить цель немодифицированной.

Для выполнения неразрушающего теста есть инструкции `cmp` и `test` . Они идентичны их деструктивным аналогам, **за исключением того, что результат операции отбрасывается, и сохраняются только флаги** .

разрушительный	Неразрушающий
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
```

Подписанные и неподписанные тесты

ЦП не дает особого значения для регистрации значений [1] , знак - это программа-программист. **Нет никакой разницы при тестировании подписанных и неподписанных значений.** Процессор вычисляет достаточно флагов для проверки обычных арифметических отношений (равных, меньше, больше и т. Д.), Как если бы операнды считались подписанными и неподписанными.

---

[1] Хотя в нем есть некоторые инструкции, которые имеют смысл только в определенных форматах, например, в дополнении. Это должно сделать код более эффективным, так как реализация алгоритма в программном обеспечении потребует большого количества кода.

Условные прыжки

На основе состояния флагов ЦП может выполнять или игнорировать скачок. Инструкция, выполняющая прыжок на основе флагов, подпадает под общее название *Jcc- Jump on Condition Code* [1] .

Синонимы и термины

Чтобы улучшить читаемость кода сборки, Intel определила несколько синонимов для одного и того же кода условий. Например, `jae` , `jnb` и `jnc` - одинаковый код условия *CF = 0* .

Хотя имя команды может дать очень сильный намек на то, когда ее использовать или нет, единственным значимым подходом является распознавание флагов, которые необходимо протестировать, а **затем** правильно выбрать инструкции.
Intel, однако, давала имена инструкций, которые имеют прекрасный смысл при использовании после команды `cmp` . Для целей этого обсуждения `cmp` будет считаться установленным флажком перед условным прыжком.

равенство

Операнд равен, если *ZF* установлен, они отличаются друг от друга. Для проверки равенства нам понадобится *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)
```

инструкция	Флаги
`je` , `jz`	ZF = 1
`jne` , `jnz`	ZF = 0

Лучше чем

Для **неподписанных операндов** назначение больше, чем источник, если перенос не нужен, то есть, если *CF = 0* . Когда *CF = 0,* возможно, что операнды были равны, тестирование *ZF* будет неоднозначно.

```
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)
```

инструкция	Флаги
`jae` , `jnc` , `jnb`	CF = 0
`ja` , `jnbe`	CF = 0, ZF = 0

Для **подписанных операндов** нам нужно проверить, что *SF = 0* , если не было подписанного переполнения, и в этом случае результирующий *SF* будет отменен. Поскольку *OF = 0,* если не было подписанного переполнения и 1 в противном случае, нам нужно проверить, что *SF = OF* .

ZF может использоваться для реализации строгого / нестандартного теста.

```
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)
```

инструкция	Флаги
`jge` , `jnl`	SF = OF
`jg` , `jnle`	SF = OF, ZF = 0

Меньше, чем

Они используют перевернутые условия выше.

```
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)
```

инструкция	Флаги
`jbe` , `jna`	CF = 1 или ZF = 1
`jb` , `jc` , `jnae`	CF = 1
`jle` , `jng`	SF! = OF или ZF = 1
`jl` , `jnge`	SF! = OF

Специальные флаги

Каждый флаг может быть протестирован индивидуально с помощью `j<flag_name>` где *flag_name* не содержит *конечную F* (например, *CF* → *C* , *PF* → *P*).

Остальные коды, которые не были рассмотрены ранее:

инструкция	Флаг
`js`	SF = 1
`jns`	SF = 0
`jo`	OF = 1
`jno`	OF = 0
`jp` , `jpe` (e = четный)	PF = 1
`jnp` , `jpo` (o = нечетный)	PF = 0

Еще один условный переход (дополнительный)

Один специальный x86 условный переход не проверяет флаг. Вместо этого он проверяет значение регистра `cx` или `ecx` (на основе текущего режима адреса процессора 16 или 32 бит), и скачок выполняется, когда регистр содержит ноль.

Эта инструкция была разработана для проверки регистра *счетчика* (`cx/ecx`) перед инструкциями, `rep` описанию, или перед циклами `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)
```

инструкция	Регистрация (не флаг)
`jcxz` , `jecxz`	cx = 0 (режим 16b)
`jcxz` , `jecxz`	ecx = 0 (режим 32b)

---

[1] Или что-то в этом роде.

Тестовые арифметические отношения

Неподписанные целые числа

Лучше чем

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

Больше или равно

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

Меньше, чем

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

Меньше или равно

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

равных

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

Не равный

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

Подписанные целые числа

Лучше чем

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

Больше или равно

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

Меньше, чем

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

Меньше или равно

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

равных

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

Не равный

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

`a_label`

В приведенных выше примерах `a_label` является целевым назначением для CPU, когда проверенное состояние является «истинным». Когда проверенное состояние «ложно», CPU продолжит следующую команду после условного перехода.

Синонимы

Существуют синонимы инструкций, которые могут использоваться для улучшения читаемости кода.
Например, `ja` и `jnbe` (Jump не ниже и не равно) являются одной и той же инструкцией.

Подписанные неподписанные сопутствующие коды

операция	неподписанный	подписанный
>	`ja`	`jg`
> =	`jae`	`jge`
<	`jb`	`jl`
<=	`jbe`	`jle`
знак равно	`je`	`je`
≠,! =, <>	`jne`	`jne`

Прочитайте Поток управления онлайн: https://riptutorial.com/ru/x86/topic/5808/поток-управления

глава 10: Преобразование десятичных строк в целые числа

замечания

Преобразование строк в целые числа является одной из общих задач.

Здесь мы покажем, как преобразовать десятичные строки в целые числа.

Код Psuedo для этого:

```
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
```

Работа с шестнадцатеричными строками немного сложнее, потому что коды символов обычно не являются непрерывными при работе с несколькими типами символов, такими как цифры (0-9) и алфавиты (af и AF). Коды символов обычно непрерывны при работе только с одним типом символов (мы будем иметь дело с цифрами здесь), поэтому мы будем иметь дело только с средами, в которых символьные коды для цифры являются непрерывными.

Examples

Сборка IA-32, GAS, соглашение о вызове 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
```

Этот код в стиле GAS преобразует десятичную строку, указанную как первый аргумент, который помещается в стек перед вызовом этой функции, в целое число и возвращает его через `%eax` . Значение `%esi` сохраняется, так как оно является регистром регистрации и используется.

Переполнение / обертка и недопустимые символы не проверяются, чтобы сделать код простым.

В C этот код может быть использован как это (при условии, что `unsigned int` и указатели имеют длину 4 байта):

```
#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;
}
```

Примечание: в некоторых средах два `string_to_integer` в коде сборки должны быть изменены на `_string_to_integer` (добавить символ подчеркивания), чтобы позволить ему работать с кодом C.

Прочитайте 16-разрядное целое число без знака от ввода.

Эта функция использует службу прерываний Int 21 / AH = 0Ah для чтения буферизованной строки.
Использование буферизованной строки позволяет пользователю просмотреть то, что они набрали, прежде чем передать его в программу для обработки.
Чтение до шести цифр (как 65535 = 2^{16} - 1 имеет шесть цифр).

Помимо выполнения стандартного преобразования от *числа* к *числу* эта функция также обнаруживает недопустимый ввод и переполнение (число слишком велико, чтобы соответствовать 16 битам).

Возвращаемые значения

Функция возвращает число, считанное в `AX` . Флаги `ZF` , `CF` , `OF` указывают, успешно ли выполнена операция или нет, и почему.

ошибка	AX	ZF	CF	O
Никто	16-разрядное целое число	Задавать	Не установлен	Не установлен
Неправильный ввод	Частично преобразованное число, до последней действительной цифры	Не установлен	Задавать	Не установлен
перелив	7FFFH	Не установлен	Задавать	Задавать

`ZF` можно использовать для быстрого указания допустимых и недействительных входов.

использование

```
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
```

Код

```
;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
```

Портирование NASM

Чтобы портировать код в NASM, удалите ключевое слово `PTR` из доступа к памяти (например, `mov cl, BYTE PTR [si]` становится `mov cl, BYTE [si]`)

MS-DOS, TASM / MASM для печати 16-разрядного числа в двоичном, четвертичном, восьмеричном, шестнадцатеричном

Распечатайте число в двоичном, четвертичном, восьмеричном, шестнадцатеричном и общей мощности двух

Все основания, которые являются степенью двух, как двоичные (2^1), четвертичные (2^2), восьмеричные (2^3), шестнадцатеричные (2^4) основания, имеют целое число бит на цифру ^{1}.
Таким образом, чтобы получить каждую цифру ^{2} цифры, мы просто разбиваем число intro group из *n* бит, начиная с LSb (справа).
Например, для четвертичной базы мы разбиваем 16-битное число в группах по два бита. Есть 8 таких групп.
Не все мощности двух оснований имеют целое число групп, которые соответствуют 16 бит; например, восьмеричная база имеет 5 групп из 3 бит, которые составляют $3 \cdot 5 = 15$ бит из 16, оставляя частичную группу из 1 бит ^{3}.

Алгоритм прост, мы изолировать каждую группу со сдвигом с последующей операции *И.* Эта процедура работает для каждого размера групп или, другими словами, для любой базовой мощности двух.

Чтобы показать цифры в правильном порядке, функция начинается с выделения наиболее значимой группы (самой левой), поэтому важно знать: a) сколько бит *D* является группой и b) битную позицию *S,* где самая левая начинается группа.
Эти значения предварительно вычисляются и хранятся в тщательно обработанных константах.

параметры

Параметры должны быть сдвинуты в стек.
Каждый из них имеет ширину 16 бит.
Они показаны в порядке толчка.

параметр	Описание
N	Число конвертируемых
База	База для использования, выраженная с использованием констант

параметр	Описание
	`BASE2` , `BASE4` , `BASE8` И `BASE16`
Печатать ведущие нули	Если в *нуле* нет неосновных нулей, в противном случае они будут. Число 0 печатается как «0», хотя

использование

```
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
```

Примечание для пользователей TASM : если вы поместите константы, определенные с `EQU` после кода, который их использует, включите *многопроходный* с флагом `/m` *TASM* или вы получите *переопределение ссылок на Forward* .

Код

```
;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
```

Данные

```
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
```

Портирование NASM

Чтобы портировать код в NASM, удалите ключевое слово PTR из доступа к памяти (например, `mov si, WORD PTR [bp+08h]` становится `mov si, WORD PTR [bp+08h]`).

Расширение функции

Функция может быть легко расширена до любой базы до 2^{255} , хотя каждое основание выше 2^{16} будет печатать тот же номер, что и число, всего 16 бит.

Чтобы добавить базу:

1. Определите новую константу `BASEx` где *x* равно 2^n .
   Нижний байт, названный *D* , равен *D* = *n* .
   Верхний байт, названный *S* , представляет собой позицию в битах более высокой группы. Его можно рассчитать как $S = n \cdot (\lceil 16 / n \rceil - 1)$.
2. Добавьте необходимые цифры в строку `DIGITS` .

Пример: добавление базы 32

Мы имеем *D* = 5 и *S* = 15, поэтому мы определяем `BASE32 EQU 0f05h` .
Затем мы добавим еще шестнадцать цифр: `DIGITS db "0123456789abcdefghijklmnopqrstuv"` .

Как и должно быть ясно, цифры можно изменить, отредактировав строку `DIGITS` .

---

[1] Если *B* является базой, то у него есть *B* цифр для каждого определения. Таким образом, количество бит на цифру составляет $log_2(B)$. Для мощности двух баз это упрощает $log_2(2^n) = n$, который по определению является целым числом.

[2] В этом контексте подразумевается, что рассматриваемая база является степенью двух базовых 2^n .

[3] Для того чтобы база $B = 2^n$ имела целое число бит-групп, должно быть, что $n \mid 16$ (*n* делит 16). Поскольку единственным фактором в 16 является 2, должно быть, что *n* само является степенью двух. Таким образом, *B* имеет вид 2^{2^k} или, что то же самое, $log_2(log_2$

(*B*)) должен быть целым числом.

MS-DOS, TASM / MASM, функция для печати 16-разрядного числа в десятичном формате

Печать 16-разрядного беззнакового числа в десятичном формате

Служба прерываний Int 21 / AH = 02h используется для печати цифр.
Стандартное преобразование из *числа* в *цифру* выполняется с помощью команды `div` , дивиденд изначально является наивысшей степенью десяти фитингов 16 бит (10^4), и на каждой итерации он уменьшается до более низких мощностей.

параметры

Параметры отображаются в порядке нажатия.
Каждый из них имеет 16 бит.

параметр	Описание
число	16-разрядное число без знака для печати в десятичной форме
показать ведущие нули	Если 0 нет неосновных нулей, иначе они есть. Число 0 всегда печатается как «0»,

использование

```
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
```

Код

```
;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
```

Портирование NASM

Чтобы портировать код в NASM, удалите ключевое слово `PTR` из доступа к памяти (например, `mov ax, WORD PTR [bp+06h]` станет `mov ax, WORD [bp+06h]`).

Прочитайте Преобразование десятичных строк в целые числа онлайн:
https://riptutorial.com/ru/x86/topic/3273/преобразование-десятичных-строк-в-целые-числа

глава 11: Реальные против защищенных режимов

Examples

Реальный режим

Когда Intel разработала исходный x86, 8086 (и 8088 производных), они включили Сегментацию, чтобы позволить 16-разрядному процессору получить доступ к адресу, превышающему 16 бит. Они сделали это, сделав 16-разрядные адреса относительно заданного 16-битного регистра сегментов, из которых они определили четыре: сегмент кода (`CS`), сегмент данных (`DS`), дополнительный сегмент (`ES`) и сегмент стека (`SS`) ,

Большинство инструкций подразумевают, какой сегментный регистр использовать: инструкции были отклонены от сегмента кода, `PUSH` и `POP` подразумевал сегмент стека, а простые ссылки на данные подразумевали сегмент данных - хотя это можно было бы переопределить для доступа к памяти в любом из других сегментов.

Реализация была простой: для каждого доступа к памяти CPU принимает подразумеваемый (или явный) регистр сегментов, сдвигает его на четыре места влево, а затем добавляет указанный адрес:

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

Это позволило использовать различные методы:

- Разрешение кодов, данных и стека для всех быть взаимно доступным (`CS` , `DS` и `SS` имеют одинаковое значение);
- Сохранение кода, данных и стека полностью отделено друг от друга (`CS` , `DS` и `SS` все 4 K (или более), отделенные друг от друга - помните, что он умножается на 16, так что это 64K).

Это также позволило причудливые совпадения и всевозможные странные вещи!

Когда 80286 был изобретен, он поддерживал этот устаревший режим (теперь он

называется «Реальный режим»), но добавил новый режим «Защищенный режим» (qv).

Важно отметить, что в реальном режиме:

- Любой адрес памяти был доступен, просто поместив правильное значение в регистр сегментов и получив доступ к 16-битовому адресу;
- Степень «защиты» заключалась в том, чтобы позволить программисту разделить разные области памяти для разных целей и *затруднить* случайную запись неверных данных - при этом все же можно сделать это.

Другими словами ... не очень защищен вообще!

Защищенный режим

Вступление

Когда 80286 был изобретен, он поддерживал наследственную сегментацию 8086 (теперь называемую «Реальный режим») и добавил новый режим «Защищенный режим». Этот режим был в каждом процессоре x86 с тех пор, хотя и расширен с различными улучшениями, такими как 32- и 64-разрядная адресация.

дизайн

В защищенном режиме простой «Добавить адрес в значение регистра сдвинутого сегмента» был полностью удален. Они хранили регистры сегментов, но вместо того, чтобы использовать их для вычисления адреса, они использовали их для индексации в таблицу (фактически, одну из двух ...), которая определяла сегмент, к которому нужно получить доступ. Это определение не только описывало, где в памяти сегмент был (с использованием Base и Limit), но и каким *типом* сегмента он был (код, данные, стек или даже система) и какие программы могли получить к нему доступ (ядро ОС, обычная программа , Драйвер устройства и т. Д.).

Регистр сегментов

Каждый 16-битный регистр сегментов принял следующий вид:

```
+------------+-----+------+
| 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
```

Глобальный / местный

Глобальный / локальный бит определял, был ли доступ включен в глобальную таблицу дескрипторов (называемую неудивительно глобальной таблицей дескриптора или GDT) или локальной таблицей дескриптора (LDT). Идея для LDT заключалась в том, что каждая программа может иметь свою собственную таблицу дескрипторов - ОС woud определяет глобальный набор сегментов, и каждая программа будет иметь свой собственный набор локальных кодов, данных и сегментов стека. ОС будет управлять памятью между различными таблицами дескрипторов.

Таблица дескрипторов

Каждая таблица дескрипторов (глобальная или локальная) представляла собой массив из 64 тыс. Дескрипторов из 8 192 дескрипторов: каждая из 8-байтных записей определяла несколько аспектов сегмента, который он описывал. Поля индекса дескриптора регистров сегментов разрешены для 8 192 дескрипторов: не совпадение!

дескриптор

Дескриптор содержал следующую информацию: обратите внимание, что формат дескриптора изменился по мере выпуска новых процессоров, но в каждом из них хранилась одна и та же информация:

- **База**
  Это определило начальный адрес сегмента памяти.
- **предел**
  Это определило размер сегмента памяти - своего рода. Они должны были принять решение: будет ли размер `0x0000` иметь размер `0` , поэтому недоступен? Или максимальный размер?
  Вместо этого они выбрали третий вариант: поле «Лимит» было последним адресным местом в сегменте. Это означало, что можно было бы определить одноразовый сегмент; или максимальный размер для размера адреса.
- **Тип**
  Было несколько типов сегментов: традиционный код, данные и стеки (см. Ниже), но также были определены другие сегменты системы:
  - Сегменты локальных дескрипторов таблицы определили, сколько локальных дескрипторов можно было получить;
  - Сегменты состояния задачи могут использоваться для коммутации аппаратного управления;
  - Контролируемые «Call Gate», которые могут позволить программам звонить в Операционную систему, - но только через тщательно управляемые точки входа.

- **Атрибуты**
  Определенные атрибуты Сегмента также поддерживались, если это необходимо:
  - Только чтение и чтение-запись;
  - Был ли данный сегмент представлен или нет, - позволяет управлять памятью по требованию;
  - Какой уровень кода (OS vs Driver vs program) может получить доступ к этому сегменту.

Истинная защита наконец!

Если ОС хранила таблицы дескрипторов в сегментах, к которым не могли быть доступны простые программы, тогда она могла бы жестко управлять тем, какие сегменты были определены, и какая память была назначена и доступна для каждого. Программа могла бы изготовить любое значение регистра Сегмента, которое ему понравилось, но если бы у него была *смелость* фактически *загрузить* его в *Регистр сегментов* ! ... аппаратное обеспечение ЦП распознало бы, что предлагаемое значение дескриптора нарушило любое из большого числа правил и вместо того, чтобы завершать запрос, он поднимет Исключение к операционной системе, чтобы позволить ему обрабатывать ошибочную программу.

> Это исключение обычно было # 13, исключение общей защиты - сделало мир знаменитым Microsoft Windows ... (Кто-нибудь думает, что инженер Intel был суеверным?)

ошибки

Ошибки, которые могут произойти, включали:

- Если предлагаемый индекс дескриптора был больше, чем размер таблицы;
- Если предлагаемый дескриптор был системным дескриптором, а не кодом, данными или стеком;
- Если предлагаемый дескриптор был более привилегированным, чем запрашивающая программа;
- Если предложенный дескриптор был помечен как нечитаемый (например, сегмент кода), но он был попытален читать, а не выполняться;
- Если предложенный дескриптор отмечен Not Present.

> Обратите внимание, что последнее не может быть фатальной проблемой для программы: ОС может отметить флаг, восстановить Segment, пометить

его как сейчас Present, а затем позволить инструкции сбоя продолжить успешно.

Или, возможно, дескриптор был успешно загружен в регистр сегментов, но затем доступ к нему в будущем нарушил один из нескольких правил:

- Регистр сегментов был загружен индексом дескриптора `0x0000` для GDT. Это было зарезервировано аппаратным обеспечением как `NULL` ;
- Если загруженный дескриптор был помечен как «Только для чтения», но попытка записи была предпринята.
- Если какая-либо часть доступа (1, 2, 4 или более байтов) находилась вне пределов сегмента.

Включение в защищенный режим

Переключение в защищенный режим прост: вам просто нужно установить один бит в регистре управления. Но, *находясь* в защищенном режиме, процессор не поднимает руки и не перезагружается из-за того, что не знает, что делать дальше, требует большой подготовки.

Короче говоря, требуются следующие шаги:

- Область памяти для Глобальной таблицы дескрипторов должна быть настроена для определения как минимум трех дескрипторов:

    1. Нулевой, `NULL` дескриптор;
    2. Другой дескриптор для сегмента кода;
    3. Другой дескриптор для сегмента данных.

        Это может использоваться как для данных, так и для стека.

- Регистр глобальной дескрипторной таблицы (`GDTR`) необходимо инициализировать, чтобы указать на эту определенную область памяти;

```
GDT_Ptr    dw      SIZE GDT
           dd      OFFSET GDT

           ...

           lgdt    [GDT_Ptr]
```

- Бит `PM` в `CR0` должен быть установлен:

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

- Регистры сегментов необходимо загрузить из GDT, чтобы удалить текущие значения

Real Mode:

```
     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!
```

Обратите внимание , что это ***абсолютный*** минимум, только чтобы получить процессор в защищенный режим. Чтобы получить всю готовую систему, может потребоваться еще много шагов. Например:

- Возможно, необходимо включить верхние области памяти - выключение затвора `A20` ;
- Прерывания должны быть обязательно отключены - но, возможно, различные обработчики ошибок могут быть настроены до входа в защищенный режим, чтобы допускать ошибки на ранней стадии обработки.

  Оригинальный автор этого раздела написал полное руководство по входу защищенного режима и работе с ним.

Нереальный режим

В *нереальном режиме* используются два факта о том, как и процессоры Intel, и AMD загружают и сохраняют информацию для описания сегмента.

1. Процессор кэширует информацию дескриптора, извлеченную во время *перемещения* в регистре селектора в защищенном режиме.
   Эти данные хранятся в архитектурной невидимой части регистра селектора.

2. В реальном режиме регистры селектора называются сегментными регистрами, но, кроме этого, они обозначают один и тот же набор регистров, и поэтому они также имеют невидимую часть. Эти части заполнены фиксированными значениями, но для базы, которая получена из только что загруженного значения.

В таком представлении реальный режим является лишь особым случаем защищенного режима: где информация сегмента, такая как база и предел, извлекается без GDT / LDT, но все еще считывается из скрытой части регистра сегмента.

При переключении в защищенном режиме и при создании GDT можно создать сегмент с желаемыми атрибутами, например базой 0 и лимитом 4GiB.
При последовательной загрузке регистра селектора такие атрибуты кэшируются, тогда можно вернуться в реальном режиме и иметь сегментный регистр, через который можно получить доступ ко всему 32-битовому адресному пространству.

```
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
```

Соображения

- Как только перезагружается сегментный регистр, даже с тем же значением, процессор перезагружает скрытые атрибуты в соответствии с текущим режимом. Вот почему приведенный выше код использует `fs` и `gs` для хранения «расширенных» сегментов: такие регистры с меньшей вероятностью будут использоваться / сохранены / восстановлены различными 16-битными службами.
- Команда `lgdt` не загружает дальний указатель на GDT, вместо этого загружает 24-разрядный (может быть переопределен до 32-разрядного) *линейный адрес* . Это не самый *близкий адрес* , это *физический адрес* (поскольку пейджинг должен быть отключен). Вот почему `GDT+7c00h` .
- Программа выше - это загрузчик (для MBR, у него нет BPB), который устанавливает `cs` / `ds` / `ss` tp *7c00h* и запускает счетчик местоположения от 0. Таким образом, байт со смещением *X* в файле находится со смещением *X* в сегменте *7c00h* и на линейном адресе *7c00h + X.*
- Прерывания должны быть отключены, так как IDT не установлен для короткого раунда в защищенном режиме.
- В коде используется хак для сохранения 6 байтов кода. Структура, загруженная `lgdt` , сохраняется в ... самом GDT, в нулевом дескрипторе (первый дескриптор).

Описание дескрипторов GDT см. В главе 3.4.3 руководства Intel Volume 3A .

Прочитайте Реальные против защищенных режимов онлайн:
https://riptutorial.com/ru/x86/topic/3679/реальные-против-защищенных-режимов

глава 12: Условные обозначения

замечания

Ресурсы

Обзор / сопоставления: справочник по условному названию Agner Fog . Кроме того, x86 ABIs (wikipedia) : вызов соглашений для функций, включая x86-64 Windows и System V (Linux).

- SystemV x86-64 ABI (официальный стандарт) . Используется всеми ОС, но Windows. (Эта страница github wiki , обновленная HJ Lu, имеет ссылки на 32bit, 64bit и x32. Также ссылки на официальный форум для разработчиков / вкладчиков ABI.) Также обратите внимание, что clang / gcc sign / zero расширяет узкие аргументы до 32bit , хотя ABI, как написано, не требует этого. От него зависит код, генерируемый Кланом.
- SystemV 32bit (i386) ABI (официальный стандарт) , используемый Linux и Unix. (старая версия).
- OS X 32bit x86, вызывающая конвенцию, со ссылками на другие . 64-битное соглашение о вызове - это система V. Сайт Apple просто ссылается на файл FreeBSD для этого.
- `__fastcall` вызове Windows x86-64 `__fastcall`
- Windows `__vectorcall` : документирует 32-битную и 64-битную версии
- Windows 32bit `__stdcall` : используется для вызова функций API Win32. Эта страница ссылается на другие документы соглашения о вызовах (например, `__cdecl`).
- Почему Windows64 использует другое соглашение о вызове от всех других ОС на x86-64? : некоторая интересная история, особенно. для SysV ABI, где архивы списков рассылки являются общедоступными и возвращаются до выпуска AMD первого кремния.

Examples

32-битный cdecl

cdecl - это соглашение о назначении 32-битной функции Windows, которое *очень* похоже на соглашение о вызове, используемое во многих операционных системах POSIX (

задокументировано в i386 System V ABI). Одно из отличий заключается в возвращении небольших структур.

параметры

Параметры передаются в стеке, причем первый аргумент имеет наименьший адрес в стеке во время вызова (нажата последняя, поэтому она находится чуть выше адреса возврата при входе в функцию). Вызывающий отвечает за сброс параметров после стека после вызова.

Возвращаемое значение

Для скалярных типов возврата возвращаемое значение помещается в EAX или EDX: EAX для 64-битных целых чисел. Тип плавающей точки возвращается в st0 (x87). Возвращение больших типов, таких как структуры, выполняется по ссылке, причем указатель передается как неявный первый параметр. (Этот указатель возвращается в EAX, поэтому вызывающему абоненту не нужно помнить, что он передал).

Сохраненные и сгруппированные регистры

EBK, EDI, ESI, EBP и ESP (и настройки режима округления FP / SSE) должны быть сохранены вызываемым абонентом, так что вызывающий может полагаться на те регистры, которые не были изменены вызовом.

Все остальные регистры (EAX, ECX, EDX, FLAGS (кроме DF), x87 и векторные регистры) могут быть свободно изменены вызываемым пользователем; если вызывающий абонент хочет сохранить значение до и после вызова функции, он должен сохранить значение в другом месте (например, в одном из сохраненных регистров или в стеке).

64-битная система V

Это соглашение о вызове по умолчанию для 64-разрядных приложений во многих операционных системах POSIX.

параметры

Первые восемь скалярных параметров передаются (по порядку) RDI, RSI, RDX, RCX, R8,

R9, R10, R11. Параметры, прошедшие первые восемь, помещаются в стек, причем более ранние параметры ближе к вершине стека. Вызывающий отвечает за выведение этих значений из стека после вызова, если он больше не нужен.

Возвращаемое значение

Для скалярных типов возврата возвращаемое значение помещается в RAX. Возвращение более крупных типов, таких как структуры, осуществляется путем концептуального изменения сигнатуры функции для добавления параметра в начале списка параметров, который является указателем на местоположение, в которое нужно поместить возвращаемое значение.

Сохраненные и сгруппированные регистры

RBP, RBX и R12-R15 сохраняются вызываемым. Все остальные регистры могут быть изменены вызываемым абонентом, и вызывающий должен сохранить значение регистра (например, в стеке), если он захочет использовать это значение позже.

32-битный stdcall

stdcall используется для 32-битных вызовов Windows API.

параметры

Параметры передаются в стек, причем первый параметр находится ближе всего к вершине стека. Вызывающий выталкивает эти значения из стека перед возвратом.

Возвращаемое значение

Скалярные значения возвращаются в EAX.

Сохраненные и сгруппированные регистры

EAX, ECX и EDX могут быть свободно изменены вызываемым абонентом и, если

необходимо, должны быть сохранены вызывающим абонентом. EBX, ESI, EDI и EBP должны быть сохранены вызываемым пользователем, если они будут восстановлены и возвращены к исходным значениям.

32-бит, cdecl - Работа с целыми

Поскольку параметры (8, 16, 32 бита)

8, 16, 32 бита всегда передаются в стеке как значения ширины 32 бита [1] .
Не требуется расширение, подпись или обнуление.
Вызов будет использовать только нижнюю часть значений полной ширины.

```
//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
```

Поскольку параметры (64 бит)

Значения 64 бит передаются в стеке с использованием двух нажатий, в соответствии с условным соглашением littel [2] , нажимая сначала более высокие 32 бита, а затем нижние.

```
//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
```

В качестве возвращаемого значения

8-битные целые числа возвращаются в `AL` , в конечном итоге сбивая весь `eax` .
16-битные целые числа возвращаются в `AX` , в конечном итоге сбивая весь `eax` .
32-битные целые числа возвращаются в `EAX` .
В `EDX:EAX` возвращаются 64-битные целые числа `EDX:EAX` , где `EAX` содержит более низкие 32 бита, а `EDX` - верхние.

```
//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] Это приведет к выравниванию стека на 4 байта, размер натурального слова. Также процессор x86 может вызывать только 2 или 4 байта, если не в длинном режиме.

[2] Нижний DWORD по нижнему адресу

32-бит, cdecl - Работа с плавающей точкой

В качестве параметров (float, double)

Поплавки имеют размер 32 бита, они естественно передаются в стек.
Удваивается 64-битный размер, они передаются в стеке, соблюдая соглашение Little Endian [1] , нажимая сначала верхние 32 бита и нижние.

```
//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
```

В качестве параметров (длинный двойной)

Длинные удваивания - 80 бит [2] в ширину, в то время как в стеке TBYTE может быть сохранен с двумя 32-битными нажатиями и одним 16-битным нажатием (для 4 + 4 + 2 = 10), чтобы поддерживать выравнивание стека на 4 байта, он заканчивается занятием 12 байт, таким образом, используя три 32 бита.
В соответствии с соглашением Little Endian биты 79-64 толкаются сначала [3] , затем бит 63-32, за которым следуют биты 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
```

В качестве возвращаемого значения

Значения с плавающей запятой, независимо от их размера, возвращаются в `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] Нижний DWORD с нижним адресом.

[2] Известен как TBYTE, от десяти байтов.

[3] Используя толчок толщины с любым расширением, более высокое значение WORD не используется.

[4] Что является TBYE широким, обратите внимание, что вопреки целым числам FP всегда возвращаются с большей точностью, что требуется.

64-разрядная версия Windows

параметры

Первые 4 параметра передаются (в порядке) RCX, RDX, R8 и R9. XMM0-XMM3 используются для передачи параметров с плавающей запятой.

Любые дополнительные параметры передаются в стек.

Параметры, превышающие 64 бит, передаются по адресу.

Проливные пространства

Даже если функция использует менее 4 параметров, вызывающий всегда предоставляет пространство для 4 параметров QWORD в стеке. Вызов может использовать их для любых целей, обычно копировать параметры там, если они будут разлиты другим вызовом.

Возвращаемое значение

Для скалярных типов возврата возвращаемое значение помещается в RAX. Если тип возврата больше 64 бит (например, для структур), то RAX является указателем на это.

Сохраненные и сгруппированные регистры

Все регистры, используемые при передаче параметров (RCX, RDX, R8, R9 и XMM0 до XMM3), RAX, R10, R11, XMM4 и XMM5 могут разливаться вызываемым абонентом. Все остальные регистры должны быть сохранены вызывающим абонентом (например, в стеке).

Выравнивание стека

Стек должен храниться в 16 байт. Так как команда «вызов» подталкивает 8-байтовый обратный адрес, это означает, что каждая нелистная функция собирается настроить стек на значение формы 16n + 8, чтобы восстановить выравнивание по 16 байт.
Это задание вызывающих абонентов для очистки стека после вызова.

Источник: история призвания конвенций, часть 5: amd64 Раймонд Чен

32-бит, cdecl - Работа с структурами

набивка

Помните, что элементы структуры обычно дополняются, чтобы обеспечить их выравнивание по их естественной границе:

```
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)
```

```
};
```

В качестве параметров (проход по ссылке)

При передаче по ссылке указатель на структуру в памяти передается как первый аргумент в стеке. Это эквивалентно передаче натурального размера (32-разрядного) целочисленного значения; см. 32-разрядную спецификацию cdecl .

В качестве параметров (перейдите по значению)

Когда они передаются по значению, структуры полностью копируются в стек, соблюдая исходный формат памяти (*т. Е.* Первый член будет на нижнем адресе).

```
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 0ffffff8h    ; 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
```

В качестве возвращаемого значения

Если они не являются тривиальными [1] , структуры возвращаются в буфер, предоставляемый вызывающим абонентом. Это эквивалентно наличию скрытого первого параметра `struct S *retval` (где `struct S` - тип структуры).

Функция должна возвращать с этим указателем на возвращаемое значение в `eax` ; Вызывающему абоненту разрешено зависеть от `eax` удерживающего указатель на возвращаемом значении, которое оно нажало прямо перед `call` .

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

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

Скрытый параметр не добавляется к счету параметров для очистки стека, так как он должен обрабатываться вызываемым пользователем.

```
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
```

В приведенном выше примере структура будет сохранена в верхней части стека.

```
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] «Тривиальная» структура - это та, которая содержит только один элемент неструктурного типа без массива (до 32 бит). Для таких структур значение этого элемента просто возвращается в регистре `eax` . (Такое поведение наблюдается при настройке GCC для Linux)

Версия cdecl для Windows отличается от соглашения о вызове System V ABI: « Тривиальная» структура допускает содержать до двух членов неструктурного типа без массива (до 32 бит). Эти значения возвращаются в `eax` и `edx` , как и 64-битное целое число. (Это поведение наблюдается для MSVC и Clang, предназначенных для Win32.)

Прочитайте Условные обозначения онлайн: https://riptutorial.com/ru/x86/topic/3261/условные-обозначения

кредиты

S. No	Главы	Contributors
1	Начало работы с Intel x86 Сборочный язык и микроархитектура	Community, David Hoelzer, Peter Cordes, Peter Mortensen, PyNEwbie, Runner
2	Манипуляция данными	Ped7g, Zopesconk
3	Механизмы системного вызова	owacoder
4	Многопроцессорное управление	Margaret Bloom, Michael Petch, RamenChef
5	Монтажники	John Burger
6	оптимизация	Cody Gray, Downvoter, faissaloo, John Burger, sannaj, Stephen Leppik
7	Основы регистрации	hidefromkgb, John Burger, Ped7g, Peter Cordes
8	Пейджинг - виртуальная адресация и память	John Burger
9	Поток управления	Margaret Bloom, owacoder, Ped7g, Zopesconk
10	Преобразование десятичных строк в целые числа	Margaret Bloom, MikeCAT
11	Реальные против защищенных режимов	John Burger, Margaret Bloom
12	Условные обозначения	Cody Gray, icktoofay, Margaret Bloom, Michael Petch, Peter Cordes, user45891, Zopesconk