eBook Gratuit

APPRENEZ

Intel x86 Assembly Language & Microarchitecture

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#x86

Table des matières

À propos

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

Chapitre 1: Démarrer avec Intel x86 Assembly Language & Microarchitecture

Remarques

Cette section fournit une vue d'ensemble de ce qu'est x86 et pourquoi un développeur peut vouloir l'utiliser.

Il devrait également mentionner tous les grands sujets dans x86, et établir un lien avec les sujets connexes. La documentation de x86 étant nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Examples

x86 langage d'assemblage

La famille des langages d'assemblage x86 représente des décennies d'avancées sur l'architecture Intel 8086 d'origine. En plus de l'utilisation de plusieurs dialectes basés sur l'assembleur, des instructions, des registres et d'autres fonctionnalités supplémentaires ont été ajoutés au cours des années, tout en restant compatibles avec l'assemblage 16 bits utilisé dans les années 80.

La première étape pour travailler avec un assemblage x86 consiste à déterminer l'objectif. Si vous souhaitez écrire du code dans un système d'exploitation, par exemple, vous souhaiterez également déterminer si vous choisirez d'utiliser un assembleur autonome ou des fonctions d'assemblage intégré intégrées à un langage de niveau supérieur, tel que C. souhaitez coder sur le "bare metal" sans système d'exploitation, vous devez simplement installer l'assembleur de votre choix et comprendre comment créer du code binaire qui peut être transformé en mémoire flash, image de démarrage ou être chargé en mémoire au emplacement approprié pour commencer l'exécution.

Un assembleur très populaire qui est bien pris en charge sur un certain nombre de plates-formes est NASM (Netwide Assembler), qui peut être obtenu à partir de http://nasm.us/ . Sur le site NASM, vous pouvez télécharger la dernière version de votre plate-forme.

les fenêtres

Les versions 32 et 64 bits de NASM sont disponibles pour Windows. NASM est fourni avec un programme d'installation pratique qui peut être utilisé sur votre hôte Windows pour installer l'assembleur automatiquement.

Linux

Il se peut que NASM soit déjà installé sur votre version de Linux. Pour vérifier, exécutez:

```
nasm -v
```

Si la commande est introuvable, vous devrez effectuer une installation. À moins que vous ne fassiez quelque chose qui nécessite des fonctionnalités NASM de pointe, le meilleur moyen est d'utiliser votre outil de gestion de paquets intégré pour votre distribution Linux pour installer NASM. Par exemple, sous des systèmes dérivés de Debian tels qu'Ubuntu et d'autres, exécutez ce qui suit à partir d'une invite de commande:

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

Pour les systèmes basés sur RPM, vous pouvez essayer:

```
sudo yum install nasm
```

Mac OS X

Les versions récentes d'OS X (y compris Yosemite et El Capitan) sont livrées avec une ancienne version de NASM préinstallée. Par exemple, El Capitan a la version 0.98.40 installée. Bien que cela fonctionnera probablement pour presque toutes les utilisations normales, il est en réalité assez ancien. Au moment de la rédaction de cet article, la version NASM 2.11 est disponible et 2.12 dispose d'un certain nombre de candidats à la sortie.

Vous pouvez obtenir le code source NASM à partir du lien ci-dessus, mais à moins que vous ayez besoin de l'installer de manière spécifique, il est beaucoup plus simple de télécharger le paquet binaire à partir du répertoire de la version OS X et de le décompresser.

Une fois décompressé, il est fortement recommandé de *ne pas* écraser la version de NASM installée sur le système. Au lieu de cela, vous pouvez l'installer dans / usr / local:

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

À ce stade, NASM est dans `/usr/local/bin` , mais il ne se trouve pas dans votre chemin. Vous devez maintenant ajouter la ligne suivante à la fin de votre profil:

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

Cela ajoutera `/usr/local/bin` à votre chemin. L'exécution de `nasm -v` à l'invite de commande doit maintenant afficher la version la plus récente.

x86 Linux Hello World Exemple

Il s'agit d'un programme Hello World de base dans un assemblage NASM pour Linux x86 32 bits, utilisant directement les appels système (sans aucun appel de fonction libc). C'est beaucoup à prendre, mais avec le temps, cela deviendra compréhensible. Les lignes commençant par un point-virgule (`;`) sont des commentaires.

Si vous ne connaissez pas déjà la programmation de systèmes Unix de bas niveau, vous pouvez simplement écrire des fonctions dans asm et les appeler à partir de programmes C ou C ++. Ensuite, vous pouvez vous contenter d'apprendre à gérer les registres et la mémoire sans apprendre à utiliser l'API d'appel système POSIX et l'ABI pour l'utiliser.

Cela fait deux appels système: `write(2)` et `_exit(2)` (pas le wrapper libc `exit(3)` qui vide les tampons stdio, etc.). (Techniquement, `_exit()` appelle sys_exit_group, pas sys_exit, mais cela ne compte que dans un processus multi-thread .) Voir aussi `syscalls(2)` pour la documentation sur les appels système en général, et la différence entre les rendre directement et utiliser la libc fonctions d'emballage

En résumé, les appels système sont effectués en plaçant les arguments dans les registres appropriés et le numéro d'appel système dans `eax` , puis en exécutant une instruction `int 0x80` . Voir aussi Quelles sont les valeurs de retour des appels système dans Assembly? pour plus d'explications sur la façon dont l'interface asm syscall est documentée avec principalement la syntaxe C.

Les numéros d'appel syscall pour l'ABI 32 bits se trouvent dans `/usr/include/i386-linux-gnu/asm/unistd_32.h` (même contenu dans `/usr/include/x86_64-linux-gnu/asm/unistd_32.h`).

`#include <sys/syscall.h>` inclura le bon fichier pour que vous puissiez exécuter `echo '#include <sys/syscall.h>' | gcc -E - -dM | less` pour voir la macro defs (voir cette réponse pour en savoir plus sur la recherche de constantes pour asm dans les en-têtes 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
```

Sous Linux, vous pouvez enregistrer ce fichier sous le nom `Hello.asm` et en générer un exécutable 32 bits avec les commandes suivantes:

```
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
```

Voir cette réponse pour plus de détails sur la construction de l' assemblage en 32 ou 64 bits statiques ou liées de façon dynamique Linux executables, pour la syntaxe MSNA / yasm ou GNU AT & T avec la syntaxe GNU `as` directives. (Point clé: assurez-vous d'utiliser `-m32` ou équivalent lorsque vous construisez du code 32 bits sur un hôte 64 bits, ou vous rencontrerez des problèmes déroutants au moment de l'exécution.)

Vous pouvez suivre son exécution avec `strace` pour voir les appels système qu'il effectue:

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

La trace sur stderr et la sortie normale sur stdout vont toutes deux au terminal ici, elles interfèrent donc dans la ligne avec l'appel système en `write` . Rediriger ou tracer vers un fichier si vous le souhaitez. Notez que cela nous permet de voir facilement les valeurs de retour de syscall sans avoir à ajouter de code pour les imprimer, et est en fait encore plus facile que d'utiliser un débogueur standard (comme gdb) pour cela.

La version x86-64 de ce programme serait extrêmement similaire, en passant les mêmes arguments aux mêmes appels système, juste dans des registres différents. Et en utilisant l'instruction `syscall` au lieu de `int 0x80` .

Lire Démarrer avec Intel x86 Assembly Language & Microarchitecture en ligne: https://riptutorial.com/fr/x86/topic/1164/demarrer-avec-intel-x86-assembly-language--amp--microarchitecture

Chapitre 2: Assembleurs

Examples

Microsoft Assembler - MASM

Étant donné que le 8086/8088 était utilisé sur le PC IBM, et que le système d'exploitation était le plus souvent celui de Microsoft, l'assembleur MASM de Microsoft était la norme de facto pendant de nombreuses années. Il suivait de près la syntaxe d'Intel, mais autorisait une syntaxe pratique mais "souple" qui (avec le recul) ne provoquait que confusion et erreurs de code.

Un exemple parfait est le suivant:

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

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

La dernière instruction `MOV` met-elle le *contenu* de `Symbol` dans `CX` ou l' *adresse* de `Symbol` dans `CX` ? Est-ce que `CX` se retrouve avec `0x1234` ou `0x0102` (ou autre)? Il s'avère que `CX` se retrouve avec `0x1234` - si vous voulez l'adresse, vous devez utiliser le spécificateur `OFFSET`

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

Intel Assembler

Intel a écrit la spécification du langage d'assemblage 8086, dérivé des précédents processeurs 8080, 8008 et 4004. En tant que tel, l'assembleur qu'ils ont écrit a suivi sa propre syntaxe avec précision. Cependant, cet assembleur n'a pas été très utilisé.

Intel a défini ses opcodes pour avoir soit zéro, un ou deux opérandes. Les instructions ont été définies deux opérandes être dans la `dest` , `source` de l' ordre, ce qui était différent des autres assembleurs à l'époque. Mais certaines instructions utilisaient des registres implicites comme des opérandes - vous deviez juste savoir ce qu'elles étaient. Intel a également utilisé le concept d'opcodes "prefix" - un opcode affecterait la prochaine instruction.

```
; 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 a également enfreint une convention utilisée par d'autres assembleurs: pour chaque opcode, un mnémonique différent a été inventé. Cela nécessitait des noms différents ou subtilement différents pour des opérations similaires: par exemple, `LDM` pour "Load from Memory" et `LDI` pour "Load Immediate". Intel utilisait le seul `MOV` mnémonique - et attendait de l'assembleur qu'il détermine quel opcode utiliser à partir du contexte. Cela a causé beaucoup de pièges et d'erreurs aux programmeurs dans le futur, lorsque l'assembleur ne pouvait pas comprendre ce que le programmeur voulait réellement ...

AT & T Assembleur - comme

Bien que le 8086 ait été le plus utilisé sur les PC IBM avec Microsoft, plusieurs autres ordinateurs et systèmes d'exploitation l'utilisaient également, notamment Unix. C'était un produit d'AT & T, et Unix fonctionnait déjà sur plusieurs autres architectures. Ces architectures utilisaient une syntaxe d'assemblage plus conventionnelle - en particulier que les instructions à deux opérandes les spécifiaient dans l'ordre `source` et `dest` .

Les conventions d'assembleur AT & T ont donc supplanté les conventions dictées par Intel, et un tout nouveau dialecte a été introduit pour la gamme x86:

- Les noms de registre ont été préfixés par `%` :
  `%al` , `%bx` etc.
- Les valeurs immédiates ont été préférées par `$` :
  `$4`
- Les opérandes étaient à la `source` , ordre de `dest`
- Les opcodes incluaient leurs tailles d'opérandes:
  `movw $4, %ax ; Move word 4 into AX`

Borland's Turbo Assembler - TASM

Borland a démarré avec un compilateur Pascal appelé "Turbo Pascal". Cela a été suivi par des compilateurs pour d'autres langages: C / C ++, Prolog et Fortran. Ils ont également produit un assembleur appelé "Turbo Assembler", qui, selon la convention de dénomination de Microsoft, s'appelait "TASM".

TASM a tenté de résoudre certains problèmes d'écriture de code en utilisant MASM (voir ci-dessus), en fournissant une interprétation plus stricte du code source sous un mode `IDEAL` spécifié. Par défaut, il supposait le mode `MASM` , donc il pouvait assembler le source MASM directement - mais Borland a ensuite trouvé qu'il devait être compatible avec le bogue avec les particularités plus "bizarres" du MASM - ainsi ils ont ajouté un mode `QUIRKS` .

Étant donné que TASM était (beaucoup) moins cher que MASM, il disposait d'une large base

d’utilisateurs, mais peu de gens utilisaient le mode IDEAL, malgré ses avantages vantés.

Assembleur GNU - gaz

Lorsque le projet GNU avait besoin d'un assembleur pour la famille x86, ils utilisaient la version AT & T (et sa syntaxe) associée à Unix plutôt qu'à la version Intel / Microsoft.

Netwide Assembler - NASM

NASM est de loin l'assembleur le plus porté pour l'architecture x86 - il est disponible sur pratiquement tous les systèmes d'exploitation basés sur le x86 (même s'il est inclus avec MacOS) et est disponible en tant qu'assembleur multiplateforme sur d'autres plates-formes.

Cet assembleur utilise la syntaxe Intel, mais il est différent des autres car il se concentre fortement sur son propre langage "macro". Cela permet au programmeur de construire des expressions plus complexes en utilisant des définitions plus simples, permettant de créer de nouvelles "instructions".

Malheureusement, cette fonctionnalité puissante a un coût: le type de données gêne les instructions généralisées, de sorte que le typage des données n'est pas appliqué.

```
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!
```

Cependant, la NASM a introduit une fonctionnalité qui manquait à d'autres: des noms de symboles définis. Lorsque vous définissez un symbole dans d'autres assembleurs, ce nom est disponible dans le reste du code - mais il "utilise" ce nom, "polluant" l'espace de nom global avec des symboles.

Par exemple (en utilisant la syntaxe NASM):

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

Après cette définition, X et Y sont définis pour toujours. Pour éviter d’utiliser les noms `X` et `Y` , vous devez utiliser des noms plus précis:

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

Mais la NASM offre une alternative. En exploitant son concept de "variable locale", vous pouvez définir des champs de structure vous obligeant à nommer la structure contenant dans les futures références:

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

Cursor  ISTRUC      Point
        ENDISTRUC

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

Malheureusement, comme NASM ne garde pas trace des types, vous ne pouvez pas utiliser la syntaxe la plus naturelle:

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

Encore un autre assembleur - YASM

YASM est une réécriture complète de NASM, mais est compatible avec les syntaxes Intel et AT & T.

Lire Assembleurs en ligne: https://riptutorial.com/fr/x86/topic/2403/assembleurs

Chapitre 3: Conventions d'appel

Remarques

Ressources

Aperçus / comparaisons: le guide de convention d'appel agréable d'Agner Fog . Aussi, x86 ABIs (wikipedia) : conventions d'appel pour les fonctions, y compris x86-64 Windows et System V (Linux).

- SystemV x86-64 ABI (norme officielle) . Utilisé par tous les systèmes d'exploitation, sauf Windows. (Cette page wiki github , tenue à jour par HJ Lu, a des liens vers 32bit, 64bit et x32. Aussi des liens vers le forum officiel pour les mainteneurs / collaborateurs ABI.) Notez également que clang / signe gcc / zéro extension args étroites à 32bit , même si l'ABI tel qu'il est écrit n'en a pas besoin. Le code généré par Clang en dépend.
- SystemV 32bit (i386) ABI (norme officielle) , utilisé par Linux et Unix. (ancienne version).
- Convention d'appel OS X 32 bits x86, avec des liens vers les autres . La convention d'appel 64 bits est System V. Le site d'Apple ne contient que des liens vers un pdf FreeBSD.
- Convention d'appel de Windows x86-64 `__fastcall`
- Windows `__vectorcall` : documente les versions 32 bits et 64 bits
- Windows 32bit `__stdcall` : utilisé pour appeler les fonctions de l'API Win32. Cette page `__cdecl` aux autres documents de convention d'appel (par exemple, `__cdecl`).
- Pourquoi Windows64 utilise-t-il une convention d'appel différente de celle de tous les autres systèmes d'exploitation sur x86-64? : une histoire intéressante, esp. pour le SysV ABI où les archives de la liste de diffusion sont publiques et remontent avant la sortie du premier silicium d'AMD.

Examples

32 bits cdecl

cdecl est une convention d'appel de fonction Windows 32 bits *très* similaire à la convention d'appel utilisée sur de nombreux systèmes d'exploitation POSIX (documentée dans l' ABI i386 System V). L'une des différences réside dans le retour de petites structures.

Paramètres

Les paramètres sont passés sur la pile, avec le premier argument à l'adresse la plus basse de la pile au moment de l'appel (dernier appui, donc juste au-dessus de l'adresse de retour à l'entrée de la fonction). L'appelant est responsable de la suppression des paramètres de la pile après l'appel.

Valeur de retour

Pour les types de retour scalaires, la valeur de retour est placée dans EAX ou EDX: EAX pour les entiers 64 bits. Les types à virgule flottante sont renvoyés dans st0 (x87). Renvoyer des types plus grands comme les structures se fait par référence, avec un pointeur passé en tant que premier paramètre implicite. (Ce pointeur est retourné dans EAX, donc l'appelant n'a pas à se souvenir de ce qu'il a passé).

Registres enregistrés et obstrués

EBX, EDI, ESI, EBP et ESP (et les paramètres du mode d’arrondi FP / SSE) doivent être conservés par l’appelé de sorte que l’appelant puisse compter sur ces registres qui n’ont pas été modifiés par un appel.

Tous les autres registres (EAX, ECX, EDX, FLAGS (autres que DF), registres x87 et vectoriels) peuvent être librement modifiés par l'appelé; Si un appelant souhaite conserver une valeur avant et après l'appel de la fonction, il doit enregistrer la valeur ailleurs (comme dans l'un des registres enregistrés ou sur la pile).

Système V 64 bits

Ceci est la convention d'appel par défaut pour les applications 64 bits sur de nombreux systèmes d'exploitation POSIX.

Paramètres

Les huit premiers paramètres scalaires sont passés (dans l’ordre) RDI, RSI, RDX, RCX, R8, R9, R10, R11. Les paramètres au-delà des huit premiers sont placés sur la pile, les paramètres antérieurs étant plus proches du sommet de la pile. L'appelant est responsable de supprimer ces valeurs de la pile après l'appel si ce n'est plus nécessaire.

Valeur de retour

Pour les types de retour scalaires, la valeur de retour est placée dans RAX. Renvoyer des types plus grands comme les structures se fait en modifiant conceptuellement la signature de la fonction pour ajouter un paramètre au début de la liste de paramètres qui est un pointeur vers un emplacement dans lequel placer la valeur de retour.

Registres enregistrés et obstrués

RBP, RBX et R12 – R15 sont conservés par l'appelé. Tous les autres registres peuvent être modifiés par l'appelé et l'appelant doit conserver lui-même la valeur d'un registre (par exemple sur la pile) s'il souhaite utiliser cette valeur ultérieurement.

Appel de 32 bits

stdcall est utilisé pour les appels d'API Windows 32 bits.

Paramètres

Les paramètres sont transmis à la pile, le premier paramètre étant le plus proche du haut de la pile. L'appelé va faire sortir ces valeurs de la pile avant de revenir.

Valeur de retour

Les valeurs de retour scalaires sont placées dans EAX.

Registres enregistrés et obstrués

EAX, ECX et EDX peuvent être librement modifiés par l'appelé et doivent être sauvegardés par l'appelant si vous le souhaitez. EBX, ESI, EDI et EBP doivent être enregistrés par l'appelé s'ils ont été modifiés et restaurés à leurs valeurs d'origine lors du retour.

32 bits, cdecl - Gestion des nombres entiers

Comme paramètres (8, 16, 32 bits)

Les entiers de 8, 16, 32 bits sont toujours passés, sur la pile, en valeurs de 32 bits de largeur totale [1] .
Aucune extension, signée ou mise à zéro, n'est nécessaire.
Le destinataire utilisera simplement la partie inférieure des valeurs de largeur complète.

```
//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
```

Comme paramètres (64 bits)

Les valeurs de 64 bits sont passées sur la pile en utilisant deux poussées, en respectant la convention conventionnelle [2] , poussant d'abord les 32 bits supérieurs puis les bits inférieurs.

```
//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
```

Comme valeur de retour

8 bits entiers sont renvoyées dans `AL` , démolir finalement l'ensemble `eax` .
16 bits sont des nombres entiers de retournés dans `AX` , éventuellement démolir l'ensemble `eax` .
Les entiers 32 bits sont renvoyés dans `EAX` .
Les entiers 64 bits sont retournés dans `EDX:EAX` , où `EAX` contient les 32 bits inférieurs et `EDX` les bits supérieurs.

```
//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] Gardez la pile alignée sur 4 octets, la taille de mot naturelle. De plus, un processeur x86 ne peut pousser que 2 ou 4 octets lorsqu'il n'est pas en mode long.

[2] DWORD inférieur à l'adresse inférieure

32 bits, cdecl - Traitement des virgules flottantes

Comme paramètres (float, double)

Les flotteurs ont une taille de 32 bits, ils sont transmis naturellement sur la pile.
Les doublons ont une taille de 64 bits, ils sont passés, sur la pile, en respectant la convention Little Endian [1] , en poussant d'abord les 32 bits supérieurs et les inférieurs.

```
//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
```

Comme paramètres (double long)

Les doubles longs ont une largeur de 80 bits [2] , tandis que sur la pile, un TBYTE peut être stocké avec deux poussées de 32 bits et une poussée de 16 bits (pour 4 + 4 + 2 = 10). 12 octets, utilisant ainsi trois poussées de 32 bits.

En respectant la convention Little Endian, les bits 79 à 64 sont d'abord poussés [3] , puis les bits 63 à 32 suivis des bits 31 à 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
```

Comme valeur de retour

Une valeur à virgule flottante, quelle que soit sa taille, est renvoyée dans `ST(0)` [4] .

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

;Assembly
fld1            ;ST(0) = 1
ret

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

;Assembly
fldz            ;ST(0) = 0
ret

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

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

---

[1] DWORD inférieur à l'adresse inférieure.

[2] Connu sous le nom de TBYTE, à partir de dix octets.

[3] En utilisant une poussée pleine largeur avec une extension, le mot supérieur n'est pas utilisé.

[4] Ce qui est large TBYE, notez que contrairement aux entiers, FP sont toujours renvoyés avec plus de précision que nécessaire.

Windows 64 bits

Paramètres

Les 4 premiers paramètres sont passés dans (dans l'ordre) RCX, RDX, R8 et R9. XMM0 à XMM3 sont utilisés pour transmettre des paramètres à virgule flottante.

Tous les paramètres supplémentaires sont passés sur la pile.

Les paramètres supérieurs à 64 bits sont transmis par adresse.

Espace de déversement

Même si la fonction utilise moins de 4 paramètres, l'appelant fournit toujours de l'espace pour 4 paramètres de taille QWORD sur la pile. L'appelant est libre de les utiliser pour n'importe quel but, il est courant de copier les paramètres là-bas s'ils étaient renversés par un autre appel.

Valeur de retour

Pour les types de retour scalaires, la valeur de retour est placée dans RAX. Si le type de retour est supérieur à 64bits (par exemple pour les structures), RAX est un pointeur sur cette valeur.

Registres enregistrés et obstrués

Tous les registres utilisés dans le passage des paramètres (RCX, RDX, R8, R9 et XMM0 à XMM3), RAX, R10, R11, XMM4 et XMM5 peuvent être déversés par l'appelé. Tous les autres registres doivent être conservés par l'appelant (par exemple sur la pile).

Alignement de pile

La pile doit être alignée sur 16 octets. Puisque l'instruction "call" pousse une adresse de retour de 8 octets, cela signifie que chaque fonction non-feuille va ajuster la pile d'une valeur de la forme 16n + 8 afin de restaurer l'alignement sur 16 octets.
C'est le travail des appelants pour nettoyer la pile après un appel.

Source: L'histoire des conventions d'appel, partie 5: amd64 Raymond Chen

Rembourrage

Rappelez-vous que les membres d'une structure sont généralement remplis pour garantir qu'ils sont alignés sur leur limite naturelle:

```
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)
};
```

Comme paramètres (passe par référence)

Lorsqu'il est transmis par référence, un pointeur sur la structure en mémoire est transmis en tant que premier argument de la pile. Cela équivaut à transmettre une valeur entière de taille naturelle (32 bits); voir cdecl 32 bits pour plus de détails.

Comme paramètres (passe par valeur)

Si transmis par valeur, structs sont entièrement copiés sur la pile, en respectant la mise en page de mémoire d' origine (à *savoir,* le premier élément sera à l'adresse inférieure).

```
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
```

Comme valeur de retour

À moins qu'ils ne soient triviaux [1] , les structures sont copiées dans un tampon fourni par l'appelant avant de retourner. Cela équivaut à avoir un premier paramètre caché `struct S *retval` (où `struct S` est le type de la structure).

La fonction doit retourner avec ce pointeur à la valeur de retour dans `eax` ; L'appelant est autorisé à dépendre de `eax` maintenant le pointeur sur la valeur de retour, qu'il a enfoncée juste avant l' `call` .

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

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

Le paramètre masqué n'est pas ajouté au nombre de paramètres à des fins de nettoyage de pile, car il doit être géré par l'appelé.

```
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
```

Dans l'exemple ci-dessus, la structure sera enregistrée en haut de la pile.

```
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] Une structure "triviale" est une structure qui ne contient qu'un seul membre d'un type non struct, non-array (jusqu'à 32 bits). Pour de telles structures, la valeur de ce membre est simplement renvoyée dans le registre `eax` . (Ce comportement a été observé avec GCC ciblant Linux)

La version Windows de cdecl est différente de la convention d'appel de System V ABI: une

structure "triviale" est autorisée à contenir jusqu'à deux membres d'un type non struct, non-array (jusqu'à 32 bits). Ces valeurs sont renvoyées dans `eax` et `edx` , tout comme un entier 64 bits. (Ce comportement a été observé pour MSVC et Clang ciblant Win32.)

Lire Conventions d'appel en ligne: https://riptutorial.com/fr/x86/topic/3261/conventions-d-appel

Chapitre 4: Conversion de chaînes décimales en nombres entiers

Remarques

La conversion de chaînes en nombres entiers est l'une des tâches courantes.

Ici, nous allons montrer comment convertir des chaînes décimales en nombres entiers.

Le code Psuedo pour ce faire est:

```
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
```

Traiter les chaînes hexadécimales est un peu plus difficile car les codes de caractères ne sont généralement pas continus lorsque vous traitez plusieurs types de caractères tels que les chiffres (0-9) et les alphabets (af et AF). Les codes de caractères sont généralement continus lorsque vous traitez un seul type de caractères (nous traiterons ici des chiffres), nous ne traiterons donc que des environnements dans lesquels les codes de caractères pour les chiffres sont continus.

Examples

IA-32 assemblage, GAS, convention d'appel 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
```

Ce code de type GAS convertira la chaîne décimale donnée en premier argument, qui est insérée dans la pile avant d'appeler cette fonction, en entier et la renverra via `%eax` . La valeur de `%esi` est enregistrée car elle est appelée registre de sauvegarde et est utilisée.

Les débordements / enveloppements et les caractères non valides ne sont pas vérifiés pour simplifier le code.

Dans C, ce code peut être utilisé comme ceci (en supposant que `unsigned int` et les pointeurs ont une longueur de 4 octets):

```
#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;
}
```

Remarque: dans certains environnements, deux `string_to_integer` dans le code d'assemblage doivent être remplacés par `_string_to_integer` (ajouter un trait de soulignement) afin de le laisser fonctionner avec le code C.

Fonction MS-DOS, TASM / MASM pour lire un entier non signé 16 bits

Lire un entier non signé 16 bits à partir de l'entrée.

Cette fonction utilise le service d'interruption Int 21 / AH = 0Ah pour lire une chaîne mise en mémoire tampon.
L'utilisation d'une chaîne en mémoire tampon permet à l'utilisateur de revoir ce qu'il a tapé avant de le transmettre au programme pour traitement.
Jusqu'à six chiffres sont lus (65535 = 2^{16} - 1 à six chiffres).

En plus d' effectuer la conversion standard du *numérique* au *numéro* de cette fonction détecte également une entrée non valide et le débordement (nombre trop gros pour tenir 16 bits).

Valeurs de retour

La fonction renvoie le numéro lu dans `AX` . Les drapeaux `ZF` , `CF` , `OF` indiquent si l'opération s'est terminée avec succès ou non et pourquoi.

Erreur	HACHE	ZF	CF	DE
Aucun	L'entier 16 bits	Ensemble	Pas encore défini	Pas encore défini
Entrée invalide	Le nombre partiellement converti, jusqu'au dernier chiffre valide rencontré	Pas encore défini	Ensemble	Pas encore défini
Débordement	7fffh	Pas encore défini	Ensemble	Ensemble

Le `ZF` peut être utilisé pour distinguer rapidement les entrées valides des données invalides.

Usage

```
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
```

Code

```
;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
```

Portage NASM

Pour porter le code sur NASM, supprimez le mot-clé `PTR` des accès mémoire (par exemple `mov cl, BYTE PTR [si]` devient `mov cl, BYTE [si]`)

Fonction MS-DOS, TASM / MASM pour imprimer un nombre 16 bits en binaire, quaternaire, octal, hexadécimal

Imprimer un nombre en binaire, quaternaire, octal, hexadécimal et une puissance générale de deux

Toutes les bases qui ont une puissance de deux, comme les bases binaire (2^1), quaternaire (2^2), octale (2^3) et hexadécimale (2^4), ont un nombre entier de bits par chiffre [1].
Ainsi, pour récupérer chaque chiffre [2] d'un nombre, nous cassons simplement le groupe d'introduction numérique de *n* bits à partir du LSb (le droit).
Par exemple, pour la base quaternaire, nous divisons un nombre de 16 bits par groupes de deux bits. Il y a 8 de ces groupes.
Toutes les puissances de deux bases ne comportent pas un nombre entier de groupes correspondant à 16 bits; Par exemple, la base octale a 5 groupes de 3 bits qui représentent $3 \cdot 5 = 15$ bits sur 16, laissant un groupe partiel de 1 bit [3].

L'algorithme est simple, nous isolons chaque groupe avec un décalage suivi d'une opération *AND*.
Cette procédure fonctionne pour chaque taille de groupe ou, en d'autres termes, pour toute puissance de base de deux.

Afin de montrer les chiffres dans le bon ordre, la fonction commence par isoler le groupe le plus significatif (le plus à gauche), il est donc important de savoir: a) combien de bits *D* un groupe est et b) la position de bit *S* le groupe commence
Ces valeurs sont précalculées et stockées dans des constantes soigneusement conçues.

Paramètres

Les paramètres doivent être poussés sur la pile.
Chacun est large de 16 bits.
Ils sont affichés par ordre de poussée.

Paramètre	La description
N	Le nombre à convertir
Base	La base à utiliser exprimée en utilisant les constantes `BASE2`, `BASE4`, `BASE8` et `BASE16`
Imprimer des zéros en tête	Si *zéro,* aucun zéros non significatif n'est imprimé, sinon ils le sont. Le nombre 0 est imprimé comme "0" si

Usage

```
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
```

Note aux utilisateurs TASM: Si vous mettez les constantes définies avec `EQU` après le code qui les utilise, activez *plusieurs passes* avec le `/m` drapeau de *TASM* ou vous aurez *besoin de référence avant remplacement.*

Code

```
;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
```

Les données

```
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
```

Portage NASM

Pour porter le code sur NASM, supprimez le mot-clé PTR des accès mémoire (par exemple `mov si, WORD PTR [bp+08h]` devient `mov si, WORD PTR [bp+08h]`)

Extension de la fonction

La fonction peut être facilement étendue à n'importe quelle base jusqu'à 2^{255} , bien que chaque

base au-dessus de 2 [16] imprime le même nombre que le nombre est seulement 16 bits.

Ajouter une base:

1. Définissez une nouvelle constante `BASEx` où *x* est 2 [n] .
   L'octet inférieur, nommé *D* , est *D* = *n* .
   L'octet supérieur, nommé *S* , est la position, en bits, du groupe supérieur. Il peut être calculé comme *S* = *n* · (⌜16 / *n* 1 - 1).
2. Ajoutez les chiffres nécessaires à la chaîne `DIGITS` .

Exemple: ajout de la base 32

Nous avons *D* = 5 et *S* = 15, nous définissons donc `BASE32 EQU 0f05h` .
Nous ajoutons ensuite seize chiffres de plus: `DIGITS db "0123456789abcdefghijklmnopqrstuv"` .

Comme cela devrait être clair, les chiffres peuvent être modifiés en éditant la chaîne `DIGITS` .

---

[1] Si *B* est une base, alors il a *B* chiffres par définition. Le nombre de bits par chiffre est donc $log_2(B)$. Pour la puissance de deux bases, cela simplifie $log_2(2^n) = n$ qui est un entier par définition.

[2] Dans ce contexte, on suppose implicitement que la base considérée est une puissance de deux bases 2^n .

[3] Pour qu'une base $B = 2^n$ ait un nombre entier de groupes de bits, il faut que $n \mid 16$ (*n* divise 16). Puisque le seul facteur en 16 est 2, il faut que *n* soit lui-même une puissance de deux. Donc *B* a la forme 2^{2^k} ou, de manière équivalente, $log_2(log_2(B))$ doit être un entier.

MS-DOS, TASM / MASM, fonction pour imprimer un nombre 16 bits en décimal

Imprimer un nombre non signé 16 bits en décimal

Le service d'interruption Int 21 / AH = 02h est utilisé pour imprimer les chiffres.
La conversion standard à partir du *numéro* de *référence numérique* est réalisée avec la `div` instruction, le dividende est initialement la plus haute puissance de dix 16 bits d' ajustement (10 [4)] et il est réduit à des puissances inférieures à chaque itération.

Paramètres

Les paramètres sont affichés par ordre de poussée.
Chacun est de 16 bits.

Paramètre	La description
nombre	Le nombre non signé 16 bits à imprimer en décimal
afficher les zéros	Si 0 aucun zéros non significatif n'est imprimé, sinon ils le sont. Le

Paramètre	La description
en tête	nombre 0 est toujours imprimé comme "0"

Usage

```
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
```

Code

```
;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
```

Portage NASM

Pour porter le code sur NASM, supprimez le mot-clé `PTR` des accès mémoire (par exemple, `mov ax, WORD PTR [bp+06h]` devient `mov ax, WORD [bp+06h]`)

Lire Conversion de chaînes décimales en nombres entiers en ligne:
https://riptutorial.com/fr/x86/topic/3273/conversion-de-chaines-decimales-en-nombres-entiers

Chapitre 5: Flux de contrôle

Examples

Sauts inconditionnels

```
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
```

Relatif près des sauts

`jmp a_label` est:

- **près**
  Il spécifie uniquement la partie offset de l' *adresse logique* de destination. Le segment est supposé être `CS` .
- **relatif**
  La sémantique de l' instruction est saut *rel* octets avant [1] de la prochaine adresse d'instruction ou `IP = IP + rel` .

L'instruction est codée sous la forme `EB <rel8>` ou `EB <rel16/32>` , l'assembleur prenant la forme la plus appropriée, préférant généralement une forme plus courte.
La `jmp SHORT a_label` par assembleur est possible, par exemple avec NASM `jmp SHORT a_label` , `jmp WORD a_label` et `jmp DWORD a_label` génèrent les trois formes possibles.

Sauts indirects absolus

`jmp bx` et `jmp WORD [aPointer]` sont:

- **près**
  Ils spécifient uniquement la partie offset de l'adresse logique de destination. Le segment est supposé être `CS` .
- **indirect indirect**
  La sémantique des instructions est sautée à l'adresse dans *reg* ou *mem* ou `IP = reg` , `IP = mem` .

L'instruction est codée comme `FF /4` , pour la mémoire indirecte, la taille de l'opérande est déterminée comme pour tout autre accès mémoire.

Sauts absolus

`jmp 7c0h:0000h` est:

- **loin**
  Il spécifie les deux parties de l'adresse *logique* : le segment et le décalage.
- **absolute** La sémantique de l'instruction passe directement au *segment d'* adresse *: offset* ou `CS = segment, IP = offset` .

L'instruction est codée comme `EA <imm32/48>` fonction de la taille du code.
Il est possible de choisir entre les deux formes dans un assembleur, par exemple avec NASM `jmp 7c0h: WORD 0000h` et `jmp 7c0h: DWORD 0000h` génère la première et la seconde forme.

Sauts absolus indirects

`jmp FAR WORD [aFarPointer]` est:

- **far** Indique les deux parties de l'adresse *logique* : le segment et le décalage.
- **Absolue indirecte** La sémantique de l'instruction est sautée vers le *segment: offset* stocké dans *mem* [2] ou `CS = mem[23:16/32], IP = [15/31:0]` .

L'instruction est codée comme `FF /5` , la taille de l'opérande peut être contrôleur avec les spécificateurs de taille.
Dans NASM, un peu non intuitif, ils sont `jmp FAR WORD [aFarPointer]` pour un opérande *16:16* et `jmp FAR DWORD [aFarPointer]` pour un opérande *16:32* .

Sauts manquants

- **presque absolu**
  Peut être émulé avec un saut presque indirect.

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

- **loin relatif**
  Cela n'a aucun sens ou est trop étroit d'utilisation.

---

[1] Deux complément permet de spécifier un décalage signé et donc de revenir en arrière.
[2] Ce qui peut être un *seg16: off16* ou un *seg16: off32* , de tailles *16:16* et *16:32* .

Conditions de test

Pour utiliser un saut conditionnel, une condition doit être testée. **Tester une condition** ici ne concerne que l'acte de vérifier les drapeaux, le saut réel est décrit sous sauts conditionnels .

x86 teste les conditions en s'appuyant sur le registre EFLAGS, qui contient un ensemble

d'indicateurs que chaque instruction peut potentiellement définir.

Les instructions arithmétiques, comme `sub` ou `add` , et les instructions logiques, comme `xor` ou `and` , "définissent évidemment les drapeaux". Cela signifie que les drapeaux *CF* , *OF* , *SF* , *ZF* , *AF* , *PF* sont modifiés par ces instructions. Toute instruction est autorisée à modifier les indicateurs, par exemple `cmpxchg` modifie le *ZF* .

Toujours vérifier la référence de **l'instruction** pour savoir quels drapeaux sont modifiés par une instruction spécifique.

x86 possède un ensemble de *sauts conditionnels* , mentionnés précédemment, qui sautent si et seulement si certains indicateurs sont définis ou si certains sont clairs ou les deux.

Les drapeaux

Les opérations arithmétiques et logiques sont très utiles pour définir les indicateurs. Par exemple, après un `sub eax, ebx` , pour maintenant détenir **des** valeurs **non signées** , nous avons:

Drapeau	Quand réglé	Lorsque clair
ZF	Lorsque le résultat est zéro. *EAX - EBX = 0* $\Rightarrow$ EAX = EBX	Lorsque le résultat n'est **pas** nul. *EAX - EBX* $\neq$ *0* $\Rightarrow$ EAX $\neq$ EBX
CF	Quand le résultat a dû être porté pour le MSb. *EAX - EBX <0* $\Rightarrow$ EAX <EBX	Lorsque le résultat n'a **pas** besoin d'être porté pour le MSb. *EAX - EBX* $\not< 0 \Rightarrow \mathrm{EAX} \not< \mathrm{EBX}$
SF	Lorsque le résultat MSb est défini.	Lorsque le résultat MSb n'est **pas** défini.
DE	Quand un débordement signé s'est produit.	Lorsqu'un dépassement signé n'a **pas eu** lieu.
PF	Lorsque le nombre de bits défini dans l'octet de résultat le moins significatif est pair.	Lorsque le nombre de bits défini dans l'octet de résultat le moins significatif est impair.
UN F	Lorsque le chiffre inférieur BCD a généré un report. C'est le bit 4 transport.	Lorsque le chiffre inférieur BCD n'a **pas** généré de report. C'est le bit 4 transport.

Tests non destructifs

Le `sub` -programme `and` instructions modifient leur opérande de destination et nécessitent deux copies supplémentaires (sauvegarde et restauration) pour que la destination ne soit pas modifiée.

Pour effectuer un test non destructif, il y a les instructions `cmp` et `test` . Ils sont identiques à leur

homologue destructeur, **sauf que le résultat de l'opération est ignoré et que seuls les indicateurs sont enregistrés** .

Destructeur	Non destructif
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
```

Tests signés et non signés

Le processeur ne donne aucune signification particulière pour enregistrer les valeurs [1] , le signe est une construction de programmeur. **Il n'y a pas de différence lors du test des valeurs signées et non signées.** Le processeur calcule suffisamment de drapeaux pour tester les relations arithmétiques habituelles (égales, inférieures à, supérieures à, etc.) si les opérandes devaient être considérées comme signées et non signées.

---

[1] Bien que certaines instructions ne soient utiles que pour des formats spécifiques, comme le complément à deux. Cela permet de rendre le code plus efficace car l'implémentation de l'algorithme dans le logiciel nécessiterait beaucoup de code.

Sauts conditionnels

En fonction de l'état des indicateurs, le processeur peut soit exécuter soit ignorer un saut. Une instruction qui effectue un saut basé sur les indicateurs tombe sous le nom générique de *Jcc - Jump on Condition Code* [1] .

Synonymes et terminologie

Afin d'améliorer la lisibilité du code d'assemblage, Intel a défini plusieurs synonymes pour le même code de condition. Par exemple, `jae` , `jnb` et `jnc` ont tous le même code de condition *CF = 0* .

Bien que le nom de l'instruction puisse donner une idée très précise de son utilisation, la seule approche utile consiste à reconnaître les indicateurs à tester, **puis à** choisir les instructions appropriées.

Intel a cependant donné les noms d'instructions qui ont un sens parfait lorsqu'elles sont utilisées après une instruction `cmp` . Pour les besoins de cette discussion, `cmp` sera supposé avoir défini les drapeaux avant un saut conditionnel.

Égalité

L'opérande est égal si *ZF* a été défini, ils diffèrent autrement. Pour tester l'égalité, nous avons besoin de *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)
```

Instruction	Les drapeaux
`je` , `jz`	ZF = 1
`jne` , `jnz`	ZF = 0

Plus grand que

Pour les **opérandes non signés** , la destination est supérieure à la source si le transport n'est pas nécessaire, c'est-à-dire si *CF = 0* . Lorsque *CF = 0* il est possible que les opérandes soient égaux, le test de *ZF* va désambiguïser.

```
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)
```

Instruction	Les drapeaux
`jae` , `jnc` , `jnb`	CF = 0
`ja` , `jnbe`	CF = 0, ZF = 0

Pour les **opérandes signés,** nous devons vérifier que *SF = 0* , sauf en cas de dépassement signé, auquel cas le *SF* résultant est inversé. Puisque *OF = 0* si aucun débordement signé ne s'est produit et 1 sinon, nous devons vérifier que *SF = OF* .

ZF peut être utilisé pour implémenter un test strict / non strict.

```
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)
```

Instruction	Les drapeaux
`jge` , `jnl`	SF = OF
`jg` , `jnle`	SF = OF, ZF = 0

Moins que

Ceux-ci utilisent les conditions inversées ci-dessus.

```
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)
```

Instruction	Les drapeaux
`jbe` , `jna`	CF = 1 ou ZF = 1
`jb` , `jc` , `jnae`	CF = 1
`jle` , `jng`	SF! = OF ou ZF = 1
`jl` , `jnge`	SF! = De

Drapeaux spécifiques

Chaque indicateur peut être testé individuellement avec `j<flag_name>` où *flag_name* ne contient pas le *F final* (par exemple, $CF \rightarrow C$, $PF \rightarrow P$).

Les autres codes non couverts sont les suivants:

Instruction	Drapeau
`js`	SF = 1

Instruction	Drapeau
`jns`	SF = 0
`jo`	OF = 1
`jno`	OF = 0
`jp` , `jpe` (e = pair)	PF = 1
`jnp` , `jpo` (o = impair)	PF = 0

Un saut conditionnel supplémentaire (extra)

Un saut conditionnel x86 spécial ne teste pas le drapeau. Au lieu de cela, il teste la valeur du registre `cx` ou `ecx` (sur la base du mode d'adresse actuel du processeur, soit 16 ou 32 bits), et le saut est exécuté lorsque le registre contient zéro.

Cette instruction a été conçue pour la validation du registre de *compteur* (`cx/ecx`) avant les instructions de type `rep` , ou avant les `loop` de `loop` .

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

Instruction	Registre (pas de drapeau)
`jcxz` , `jecxz`	cx = 0 (mode 16b)
`jcxz` , `jecxz`	ecx = 0 (mode 32b)

---

[1] Ou quelque chose comme ça.

Tester les relations arithmétiques

Entiers non signés

Plus grand que

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

Meilleur que ou égal

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

Moins que

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

Inférieur ou égal

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

Égal

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

Inégal

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

Entiers signés

Plus grand que

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

Meilleur que ou égal

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

Moins que

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

Inférieur ou égal

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

Égal

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

Inégal

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

---

a_label

Dans les exemples ci-dessus, le `a_label` est la destination cible du processeur lorsque la condition testée est "true". Lorsque la condition testée est "false", le processeur continue sur l'instruction suivante après le saut conditionnel.

Des synonymes

Il existe des synonymes d'instructions qui peuvent être utilisés pour améliorer la lisibilité du code. Par exemple `ja` et `jnbe` (Jump non inférieur ni égal) sont la même instruction.

Codes compagnons non signés signés

Opération	Non signé	Signé
>	`ja`	`jg`
> =	`jae`	`jge`
<	`jb`	`jl`
<=	`jbe`	`jle`
=	`je`	`je`
≠,! =, <>	`jne`	`jne`

Lire Flux de contrôle en ligne: https://riptutorial.com/fr/x86/topic/5808/flux-de-controle

Chapitre 6: Gestion multiprocesseur

Paramètres

Registre LAPIC	Adresse (relative à *APIC BASE*)
Registre local APIC ID	+ 20h
Registre vectoriel d'interruption parasite	+ 0f0h
Registre de commande d'interruption (ICR); bits 0-31	+ 300h
Registre de commande d'interruption (ICR); bits 32-63	+ 310h

Remarques

Pour accéder aux registres LAPIC, un segment doit pouvoir atteindre la plage d'adresses commençant à *APIC Base* (dans *IA32_APIC_BASE*).
Cette adresse est relogeable et peut théoriquement être positionnée quelque part dans la mémoire inférieure, ce qui rend la plage adressable en mode réel.

Les cycles de lecture / écriture de la plage LAPIC **ne** sont cependant **pas** propagés à l'unité d'interface de bus, masquant ainsi tout accès aux adresses "derrière" celle-ci.

On suppose que le lecteur est familier avec le mode Unreal , car il sera utilisé dans certains exemples.

Il faut aussi maîtriser:

- Gérer la différence entre *les* adresses *logiques* et *physiques* [1]
- Segmentation en mode réel .
- Alias de mémoire, possibilité d'utiliser différentes adresses *logiques* pour la même adresse *physique*
- Absolu, relatif, loin, proche des appels et des sauts.
- Assembleur NASM , en particulier que la directive `ORG` est globale. Diviser le code en plusieurs fichiers simplifie **grandement** le codage car il sera possible de donner différents *ORG de* section différente.

Enfin, nous supposons que le CPU dispose d'un *contrôleur d'interruption programmable avancé local* (*LAPIC*).
S'il est ambigu du contexte, APIC signifie toujours LAPIC (et non IOAPIC ou xAPIC en général).

Les références:

- Chapitre 8 et 10 des manuels Intel .

Bitfields

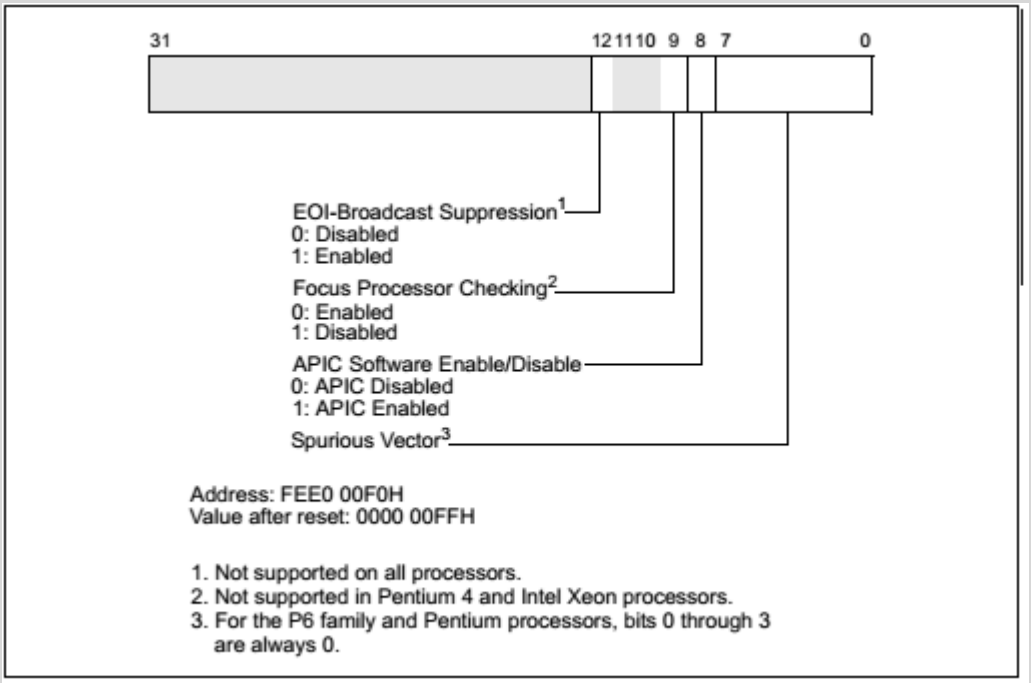


Figure 10-23. Spurious-Interrupt Vector Register (SVR)

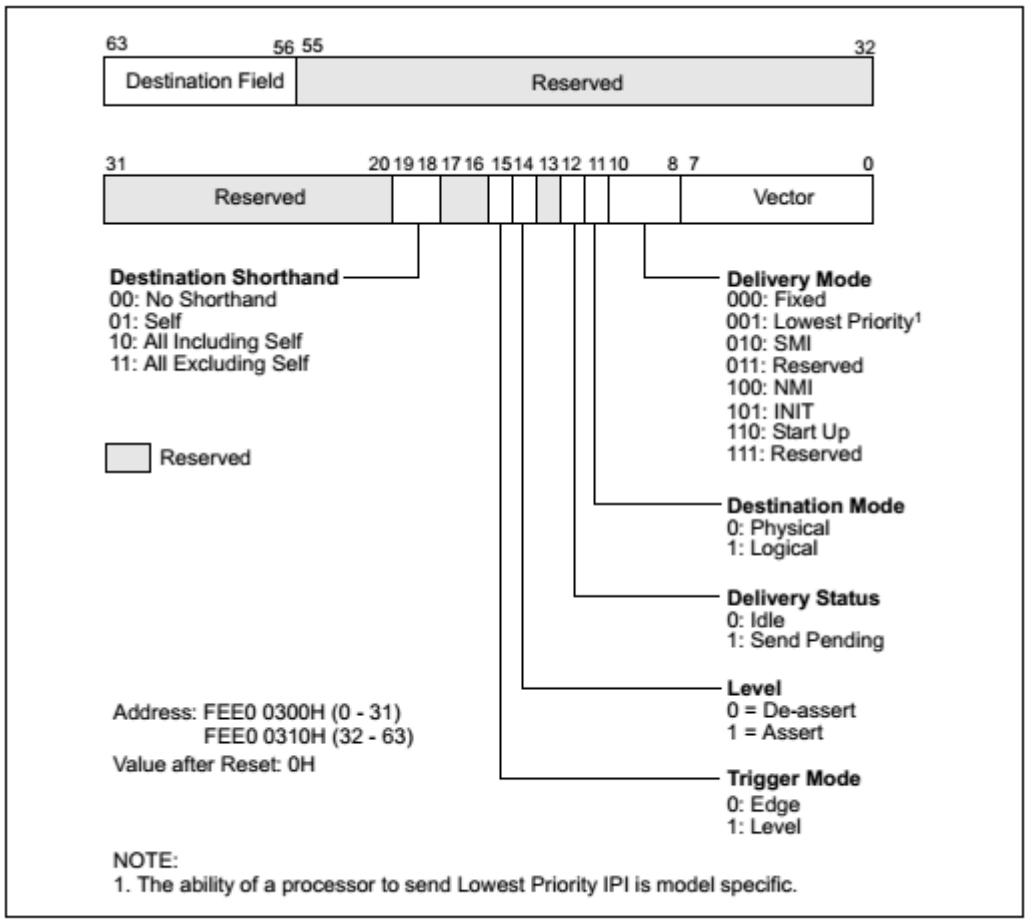


Figure 10-12. Interrupt Command Register (ICR)

Bitfields

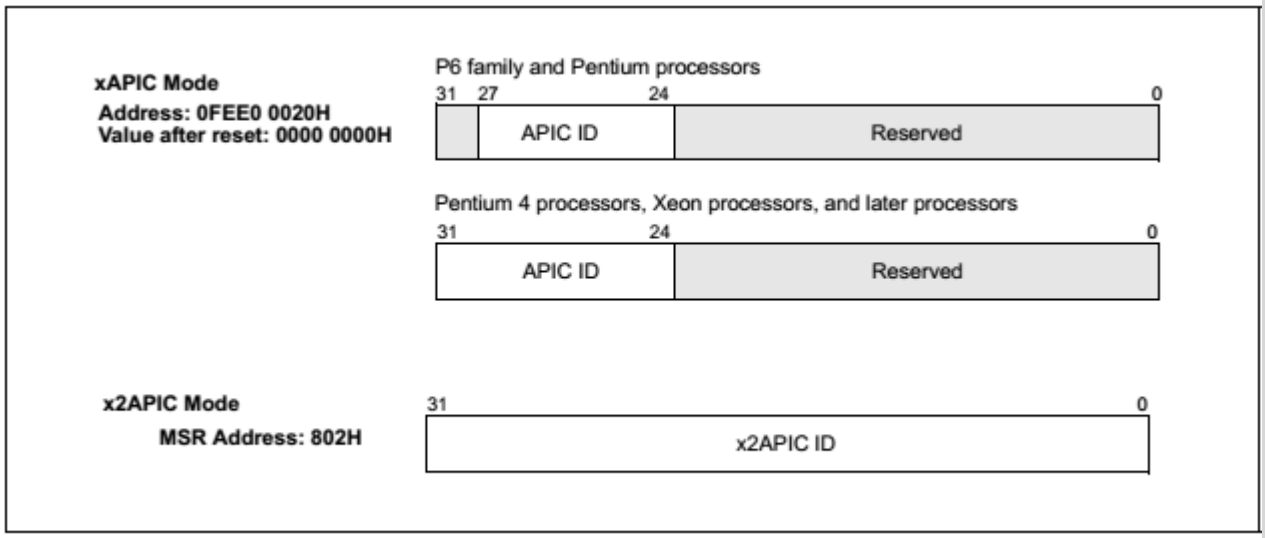


Figure 10-6. Local APIC ID Register

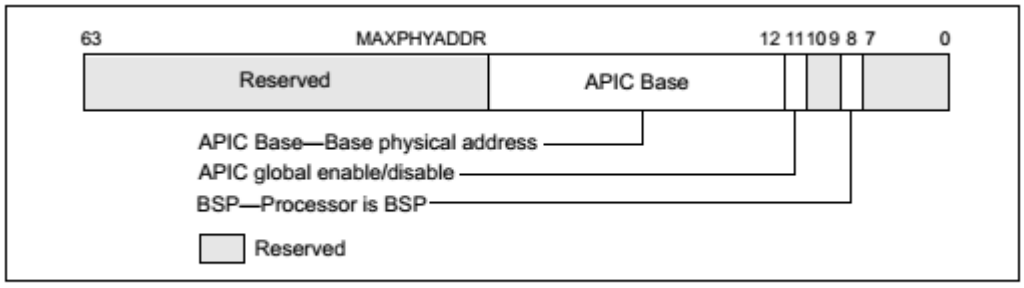


Figure 10-5. IA32_APIC_BASE MSR (APIC_BASE_MSR in P6 Family)

Nom MSR	Adresse
IA32_APIC_BASE	1bh

[1] Si la pagination est utilisée, *les* adresses *virtuelles* entrent également en jeu.

Examples

Réveillez tous les processeurs

Cet exemple réveillera tous les *processeurs d'application* (AP) et les affichera, avec le *processeur de* démarrage (BSP), pour afficher leur identifiant LAPIC.

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

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

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

 ;Clear screen
 mov ax, 03h
 int 10h

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

 ;Enable the APIC
 call enable_lapic

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


 ;Wakeup the other APs

 ;INIT
 call lapic_send_init
 mov cx, WAIT_10_ms
 call us_wait

 ;SIPI
 call lapic_send_sipi
 mov cx, WAIT_200_us
 call us_wait

 ;SIPI
 call lapic_send_sipi

 ;Jump to the payload
 jmp 0000h:8000h

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

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

  ret

  WAIT_10_ms     EQU 10000
  WAIT_200_us    EQU 200

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

```
enable_lapic:

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

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

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

 ret

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

lapic_send_sipi:
 mov eax, DWORD [APIC_BASE]

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

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

 ret

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

lapic_send_init:
 mov eax, DWORD [APIC_BASE]

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

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

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

  ret

 IA32_APIC_BASE_MSR    EQU    1bh

 APIC_REG_SIV        EQU    0f0h

 APIC_REG_ICR_LOW    EQU 300h
 APIC_REG_ICR_HIGH    EQU 310h

 APIC_REG_ID        EQU 20h

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

 APIC_BASE            dd     00h

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

unrealmode:
 lgdt [cs:GDT]

 cli

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

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

 and ax, 0fffeh
 mov cr0, eax

 sti

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

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

    dd 0000ffffh
    dd 00cf9200h

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

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

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

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

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

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

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

  mov cx, ax            ;Save this unique id

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

  ;Text buffer
  push 0b800h
  pop es

  ;Set unreal mode again
  call 7c0h:unrealmode

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

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

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

  cli
  hlt

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

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

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

    ret

  digits db "0123456789abcdef"
  counter dw 0

 payload_end:



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

Il y a deux étapes principales à réaliser:

1. Réveiller les AP
Ceci est réalisé en insérant une séquence *INIT-SIPI-SIPI* (ISS) à tous les *points d'accès* .

Le BSP qui enverra la séquence ISS en utilisant comme destination le raccourci *Tout excluant soi - même* , ciblant ainsi tous les points d'accès.

Un SIPI (Startup Inter Processor Interrupt) est ignoré par tous les processeurs qui sont désactivés au moment où ils le reçoivent, de sorte que le second SIPI est ignoré si le premier suffit à réveiller les processeurs cibles. Il est conseillé par Intel pour des raisons de compatibilité.

Un SIPI contient un *vecteur* , qui a un sens similaire, **mais absolument différent en pratique** , d'un vecteur d'interruption (alias numéro d'interruption).
Le vecteur est un nombre de 8 bits, de valeur *V* (représenté par *vv* en base 16), qui fait que le CPU commence à exécuter des instructions à l'adresse *physique 0vv000h* .
Nous appellerons *0vv000h* l' *adresse de réveil* (WA).
Le WA est forcé à une limite de 4 Ko (ou page).

Nous utiliserons 08h comme *V* , le WA sera alors *08000h* , 400h après le bootloader.

Cela donne le contrôle aux points d'accès.

2. Initialiser et différencier les points d'accès
Il est nécessaire d'avoir un code exécutable au WA. Le chargeur de démarrage est à *7c00h* , nous devons donc déplacer un peu de code à la limite de la page.

La première chose à retenir lors de l'écriture de la charge utile est que tout accès à une ressource

partagée doit être protégé ou différencié.
Une ressource partagée commune est la pile , si nous initialisons la pile naïvement, chaque AP finira par utiliser la même pile!

La première étape consiste alors à utiliser différentes adresses de pile, *différenciant* ainsi la pile. Nous accomplissons cela en attribuant un numéro unique, basé sur zéro, pour chaque CPU. Ce nombre, nous l'appellerons *index* , est utilisé pour différencier la pile et la ligne si le processeur écrit son identifiant APIC.

L'adresse de pile pour chaque CPU est *800h: (index * 1000h)* donnant à chaque AP 64KiB de pile.

Le numéro de ligne de chaque CPU est *index* , le pointeur dans le tampon de texte est donc 80 * 2 * *index* .

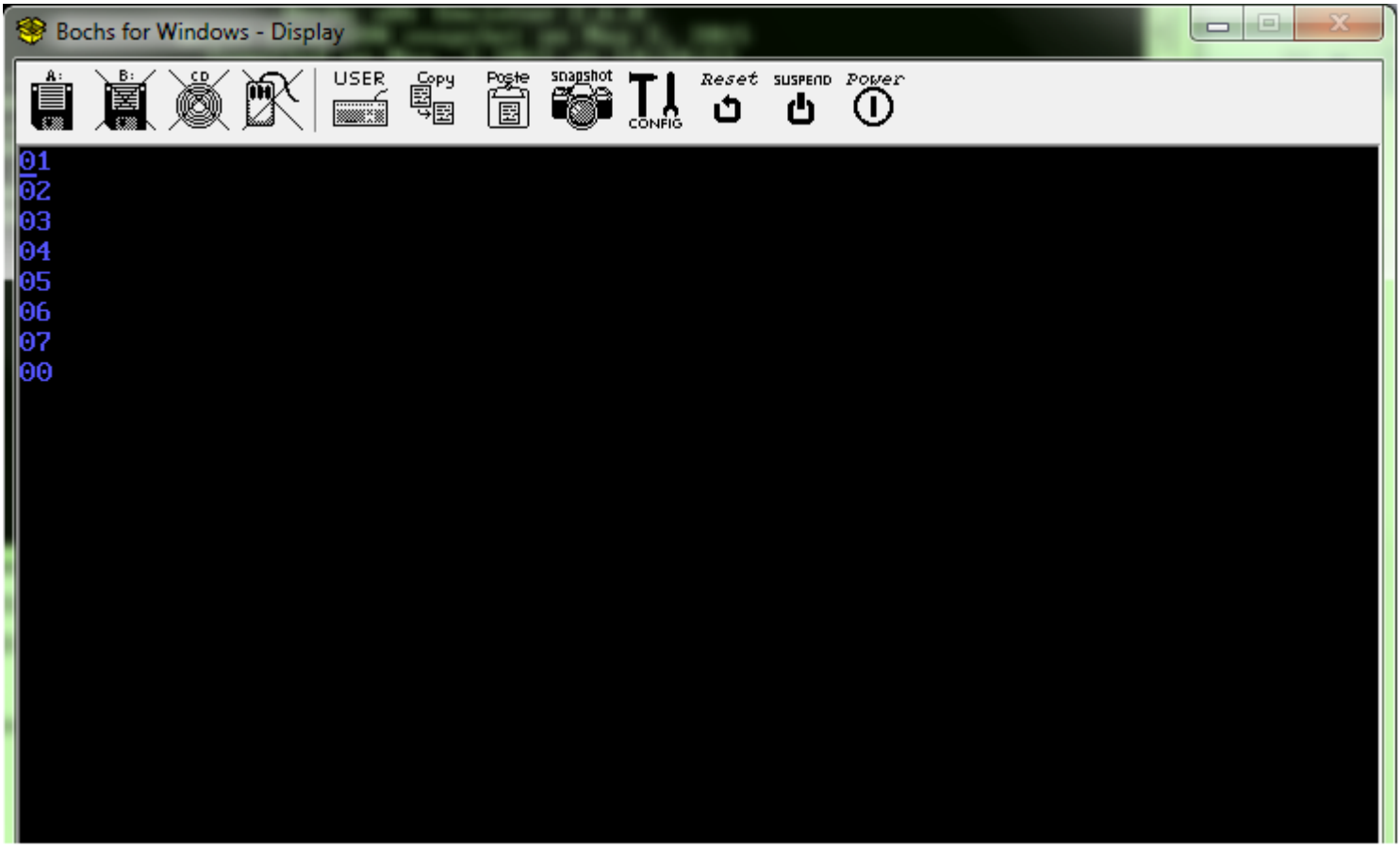
Pour générer l'index, un `lock cmpxchg` est utilisé pour incrémenter et retourner de manière atomique un mot.

Notes finales

- Une écriture sur le port 80h est utilisée pour générer un délai de 1 µs.
- `unrealmode` est une routine de loin, donc il peut être appelé après le réveil aussi.
- Le BSP passe également au WA.

Capture d'écran

De Bochs avec 8 processeurs



Lire Gestion multiprocesseur en ligne: https://riptutorial.com/fr/x86/topic/5809/gestion-multiprocesseur

Chapitre 7: Manipulation de données

Syntaxe

- ***.386*** : **demande** à **MASM** de compiler pour une version de puce x86 minimum de 386.
- ***.model*** : définit le modèle de mémoire à utiliser, voir .MODEL .
- ***.code*** : segment de code utilisé pour des processus tels que le processus principal.
- ***proc*** : déclare le processus.
- ***ret*** : utilisé pour sortir des fonctions avec succès, voir Travailler avec des valeurs de retour .
- ***endp*** : termine la déclaration de processus.
- ***public*** : met les processus à la disposition de tous les segments du programme.
- ***end*** : termine le programme ou, s'il est utilisé avec un processus, tel que " **end main** ", fait du processus la méthode principale.
- ***call*** : Appelle le processus et envoie son opcode sur la pile, voir Contrôle de flux .
- ***ecx*** : Contre-registre, voir registres .
- ***ecx*** : Contre registre.
- ***mul*** : multiplie la valeur par eax

Remarques

mov est utilisé pour transférer des données entre les registres .

Examples

Utiliser MOV pour manipuler les valeurs

La description:

`mov` *copie les* valeurs des bits de l'argument source vers l'argument de destination.

Les sources / destinations communes sont les registres , généralement le moyen le plus rapide de manipuler les valeurs avec [in] CPU.

Un autre groupe important de valeurs source_of / destination_for est la mémoire de l'ordinateur.

Enfin, certaines valeurs immédiates peuvent faire partie du codage des instructions `mov` lui-même, ce qui permet de gagner du temps lors de l'accès à la mémoire en lisant la valeur avec l'instruction.

Sur les processeurs x86 en mode 32 et 64 bits, il existe de nombreuses possibilités de les combiner, en particulier les différents modes d'adressage de la mémoire. En général, la copie mémoire-mémoire est hors limite (à l'exception des instructions spécialisées comme `MOVSB`), et une telle manipulation nécessite un stockage intermédiaire des valeurs dans le registre [s] en premier.

Étape 1: Configurez votre projet pour qu'il utilise **MASM** , voir Exécution d'un assembly x86 dans Visual Studio 2015
Étape 2: Tapez ceci:

```
.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
```

Étape 3: Compiler et déboguer.

Le programme doit retourner la valeur `16` .

Lire Manipulation de données en ligne: https://riptutorial.com/fr/x86/topic/8030/manipulation-de-donnees

Chapitre 8: Mécanismes d'appel système

Examples

Appels du BIOS

Comment interagir avec le BIOS

Le système d'entrée / sortie de base, ou BIOS, est ce qui contrôle l'ordinateur avant tout système d'exploitation. Pour accéder aux services fournis par le BIOS, le code de l'assemblage utilise des *interruptions* . Une interruption prend la forme de

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

Le numéro d'interruption doit être compris entre 0 et 255 (0x00 - 0xFF) inclus.

La plupart des appels du BIOS utilisent le registre `AH` comme paramètre de "sélection de fonction" et utilisent le registre `AL` comme paramètre de données. La fonction sélectionnée par `AH` dépend de l'interruption appelée. Certains appels du BIOS nécessitent un seul paramètre 16 bits dans `AX` , ou n'acceptent aucun paramètre, et sont simplement appelés par l'interruption. Certains ont encore plus de paramètres, qui sont passés dans d'autres registres.

Les registres utilisés pour les appels du BIOS sont fixes et ne peuvent pas être échangés avec d'autres registres.

Utiliser les appels du BIOS avec la fonction select

La syntaxe générale d'une interruption du BIOS utilisant un paramètre de sélection de fonction est la suivante:

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

Exemples

Comment écrire un caractère à l'écran:

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

Comment lire un personnage du clavier (blocage):

```
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
```

Comment lire un ou plusieurs secteurs à partir d'un disque externe (en utilisant l'adressage 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
```

Comment lire le système RTC (horloge temps réel):

```
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
```

Comment lire l'heure système du 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
```

Comment lire la date du système à partir du 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
```

Comment obtenir la taille de la mémoire basse contiguë:

```
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
```

Comment redémarrer l'ordinateur:

```
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.
```

La gestion des erreurs

Certains appels du BIOS peuvent ne pas être implémentés sur chaque machine et ne sont pas garantis pour fonctionner. Souvent, une interruption non `0x86` ou `0x80` dans le registre `AH` . **À peu près toutes les interruptions placeront l'indicateur de retenue (CF) sur une condition d'erreur.** Cela permet de passer facilement à un gestionnaire d'erreur avec le saut conditionnel `jc` . (Voir sauts conditionnels)

Les références

La liste d'interruptions de Ralf Brown contient une liste assez exhaustive d'appels du BIOS et d'autres interruptions. Une version HTML peut être trouvée ici .

Les interruptions souvent supposées disponibles se trouvent dans une liste sur Wikipedia .

Un aperçu plus approfondi des interruptions couramment disponibles est disponible sur osdev.org

Lire Mécanismes d'appel système en ligne: https://riptutorial.com/fr/x86/topic/6946/mecanismes-d-appel-systeme

Chapitre 9: Modes réel vs protégé

Examples

Mode réel

Lorsque Intel a conçu le x86 d'origine, le 8086 (et le dérivé 8088), ils incluaient la segmentation pour permettre au processeur 16 bits d'accéder à plus de 16 bits d'adresse. Pour ce faire, les adresses à 16 bits ont été associées à un registre de segments à 16 bits, dont elles ont défini quatre: segment de code (`CS`), segment de données (`DS`), segment supplémentaire (`ES`) et segment de pile (`SS`). .

La plupart des instructions impliquaient le registre de segment à utiliser: les instructions étaient extraites du segment de code, `PUSH` et `POP` impliquaient le segment de pile, et les références de données simples impliquaient le segment de données - bien que cela puisse être remplacé pour accéder à la mémoire des autres segments.

L'implémentation était simple: pour chaque accès mémoire, le CPU prendrait le registre de segments implicite (ou explicite), le décalerait de quatre places à gauche, puis ajouterait l'adresse indiquée:

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

Cela a permis diverses techniques:

- Autoriser le code, les données et la pile à tous être accessibles (`CS` , `DS` et `SS` avaient tous la même valeur);
- Garder le code, les données et la pile complètement séparés les uns des autres (`CS` , `DS` et `SS` tous les 4K (ou plus) sont séparés les uns des autres - rappelez-vous qu'il est multiplié par 16, soit 64K).

Il a également permis des chevauchements bizarres et toutes sortes de choses étranges!

Lorsque le 80286 a été inventé, il supportait ce mode hérité (maintenant appelé "Mode réel"), mais a ajouté un nouveau mode appelé "Mode protégé" (qv).

Les choses importantes à noter est que, en mode réel:

- Toute adresse mémoire était accessible, simplement en mettant la valeur correcte dans un

registre de segment et en accédant à l'adresse 16 bits;
- L'étendue de la "protection" consistait à permettre au programmeur de séparer différentes zones de la mémoire à des fins différentes et de rendre *plus difficile l'* écriture accidentelle dans les mauvaises données, tout en permettant de le faire.

En d'autres termes ... pas très protégé du tout!

Mode protégé

introduction

Lorsque le 80286 a été inventé, il supportait l'ancienne segmentation 8086 (désormais appelée «mode réel»), et a ajouté un nouveau mode appelé «mode protégé». Ce mode a été dans tous les processeurs x86 depuis, mais a été amélioré avec diverses améliorations telles que l'adressage 32 et 64 bits.

Conception

En mode protégé, la simple "Ajouter une adresse à la valeur du registre de segments décalés" a été complètement supprimée. Ils ont conservé les registres de segments, mais au lieu de les utiliser pour calculer une adresse, ils les ont utilisés pour indexer dans une table (en fait une parmi deux ...) qui définissait le segment auquel accéder. Cette définition ne décrivait pas seulement où se trouvait le segment (en utilisant Base et Limit), mais aussi quel *type* de segment c'était (Code, Data, Stack ou même System) et quels types de programmes pouvaient y accéder (noyau OS, programme normal) , Pilote de périphérique, etc.).

Registre de segment

Chaque registre de segment 16 bits prend la forme suivante:

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

Global / Local

Le bit Global / Local définit si l'accès se trouve dans une table globale de descripteurs (appelée sans surprise la table de descripteur globale ou GDT) ou une table de descripteur local (LDT). L'idée du LDT était que chaque programme puisse avoir sa propre table de descripteurs - le système d'exploitation définirait un ensemble global de segments et chaque programme aurait son propre ensemble de segments de code local, de données et de pile. Le système d'exploitation

gérerait la mémoire entre les différentes tables de descripteurs.

Table de descripteur

Chaque table de descripteurs (globale ou locale) était un tableau de 64 Ko composé de 8 192 descripteurs: chacun un enregistrement de 8 octets définissant plusieurs aspects du segment qu'il décrivait. Les champs Index des descripteurs des registres de segments permettaient 8 192 descripteurs: pas de coïncidence!

Descripteur

Un descripteur contenait les informations suivantes - notez que le format du descripteur a changé à mesure que de nouveaux processeurs étaient publiés, mais que le même type d'informations était conservé dans chacun d'eux:

- **Base**
  Ceci a défini l'adresse de départ du segment de mémoire.
- **Limite**
  Cela a défini la taille du segment de mémoire - en quelque sorte. Ils ont dû prendre une décision: une taille de `0x0000` signifierait-elle une taille de `0` , donc pas accessible? Ou taille maximale?
  Au lieu de cela, ils ont choisi une troisième option: le champ Limite était le dernier emplacement accessible dans le segment. Cela signifiait qu'un segment unique pouvait être défini; ou une taille maximale pour la taille de l'adresse.
- **Type**
  Il y avait plusieurs types de segments: le code, les données et la pile traditionnels (voir ci-dessous), mais d'autres segments de système ont également été définis:
  - Les segments de la table de descripteurs locaux définissaient le nombre de descripteurs locaux accessibles.
  - Les segments d'état de tâche pourraient être utilisés pour la commutation de contexte gérée par le matériel;
  - Des «portes d'appel» contrôlées qui pourraient permettre aux programmes d'appeler le système d'exploitation, mais uniquement par le biais de points d'entrée soigneusement gérés.
- **Les attributs**
  Certains attributs du segment ont également été conservés, le cas échéant:
  - Lecture seule vs lecture-écriture;
  - Si le segment était actuellement présent ou non - permettant une gestion de la mémoire à la demande;
  - Quel niveau de code (système d'exploitation vs pilote vs programme) pourrait accéder à ce segment.

La vraie protection enfin!

Si le système d'exploitation conservait les tables de descripteurs dans des segments auxquels il n'était pas possible d'accéder par de simples programmes, il pourrait alors gérer avec précision les segments définis et la mémoire attribuée et accessible à chacun. Un programme pourrait fabriquer n'importe quelle valeur de registre de segment qui lui plaisait - mais s'il avait l' *audace* de le *charger* réellement dans un *registre de segment* ! ..., le matériel CPU reconnaîtrait que la valeur de descripteur proposée enfreignait un grand nombre de règles. au lieu de compléter la demande, une exception au système d'exploitation serait levée pour lui permettre de gérer le programme errant.

> Cette exception était généralement n ° 13, l'exception de protection générale - rendue mondialement célèbre par Microsoft Windows ... (quelqu'un pense-t-il qu'un ingénieur Intel était superstitieux?)

les erreurs

Les types d'erreurs pouvant survenir sont les suivants:

- Si l'indice de descripteur proposé était plus grand que la taille de la table;
- Si le descripteur proposé était un descripteur de système plutôt qu'un code, une donnée ou une pile;
- Si le descripteur proposé était plus privilégié que le programme demandeur;
- Si le descripteur proposé était marqué comme étant illisible (tel qu'un segment de code), mais qu'il a été tenté de le lire plutôt que de l'exécuter;
- Si le descripteur proposé était marqué comme non présent.

> Notez que le dernier peut ne pas être un problème fatal pour le programme: le système d'exploitation pourrait noter le drapeau, rétablir le segment, le marquer comme étant présent, puis laisser l'instruction d'instruction se poursuivre avec succès.

Ou, peut-être le descripteur a-t-il été chargé avec succès dans un registre de segments, mais un futur accès à celui-ci enfreint l'une des nombreuses règles suivantes:

- Le registre de segment a été chargé avec l' `0x0000` descripteur `0x0000` pour le GDT. Cela a été réservé par le matériel comme `NULL` ;
- Si le descripteur chargé était marqué comme étant en lecture seule, une écriture y a été tentée.
- Si une partie de l'accès (1, 2, 4 octets ou plus) était en dehors de la limite du segment.

Passer en mode protégé

Passer en mode protégé est facile: il vous suffit de définir un seul bit dans un registre de contrôle. Mais *rester* en mode protégé, sans que le CPU ne lève les bras et se réinitialise en ne sachant

pas quoi faire, demande beaucoup de préparation.

En bref, les étapes requises sont les suivantes:

- Une zone de mémoire pour la table de descripteur globale doit être configurée pour définir un minimum de trois descripteurs:

    1. Le zeroeth, descripteur `NULL` ;
    2. Un autre descripteur pour un segment de code;
    3. Un autre descripteur pour un segment de données.

        Cela peut être utilisé à la fois pour Data et Stack.

- Le registre `GDTR` (Global Descriptor Table Register) doit être initialisé pour indiquer cette zone de mémoire définie.

```
GDT_Ptr    dw      SIZE GDT
           dd      OFFSET GDT

           ...

           lgdt    [GDT_Ptr]
```

- Le bit `PM` dans `CR0` doit être défini:

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

- Les registres de segment doivent être chargés depuis le GDT pour supprimer les valeurs actuelles du mode réel:

```
     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!
```

Notez que c'est le ***strict*** minimum, juste pour obtenir le CPU en mode protégé. Pour que l'ensemble du système soit prêt, plusieurs étapes peuvent être nécessaires. Par exemple:

- Les zones de mémoire supérieures doivent être activées - désactivation de la porte `A20` ;
- Les interruptions doivent définitivement être désactivées, mais les différents gestionnaires de pannes peuvent peut-être être configurés avant d'entrer en mode protégé, afin de permettre des erreurs au début du traitement.

    L'auteur original de cette section a écrit un didacticiel complet sur l'entrée du mode protégé et son utilisation.

Mode irréel

Le *mode irréel* exploite deux faits sur la façon dont les processeurs Intel et AMD chargent et enregistrent les informations pour décrire un segment.

1. Le processeur met en cache les informations de descripteur extraites lors d'un *déplacement* dans un registre de sélecteur en mode protégé.
   Ces informations sont stockées dans une partie architecturale invisible du registre de sélection elles-mêmes.

2. En mode réel, les registres de sélecteur sont appelés registres de segment mais, à part cela, ils désignent le même ensemble de registres et ont donc une partie invisible. Ces parties sont remplies avec des valeurs fixes, mais pour la base qui est dérivée de la valeur qui vient d'être chargée.

Dans une telle vue, le mode réel n'est qu'un cas particulier du mode protégé: où les informations d'un segment, telles que la base et la limite, sont extraites sans GDT / LDT mais toujours lues dans la partie cachée du registre de segment.

En commutant en mode protégé et en créant un GDT, il est possible de créer un segment avec les attributs souhaités, par exemple une base de 0 et une limite de 4 Go.
Grâce à un chargement successif d'un registre de sélecteur, de tels attributs sont mis en cache, il est alors possible de revenir en mode réel et d'avoir un registre de segment par lequel accède à l'espace d'adresse complet de 32 bits.

```
BITS 16

jmp 7c0h:__START__

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


 lgdt [GDT]            ;Set the GDTR register


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


 ;Entering protected mode

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

 ;We are now in Protected mode

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

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

;Exit protected mode

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

sti

;Back to real mode

;Do nothing
cli
hlt



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

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

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


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

Considérations

- Dès qu'un registre de segment est rechargé, même avec la même valeur, le processeur recharge les attributs masqués en fonction du mode en cours. C'est pourquoi le code ci-dessus utilise `fs` et `gs` pour contenir les segments "étendus": de tels registres sont moins susceptibles d'être utilisés / enregistrés / restaurés par les différents services 16 bits.
- L'instruction `lgdt` ne charge pas un pointeur éloigné sur le GDT, mais charge une *adresse linéaire de* 24 bits (pouvant être remplacée par 32 bits). Ce n'est pas une *adresse proche* , c'est l' *adresse physique* (car la pagination doit être désactivée). C'est la raison de `GDT+7c00h` .
- Le programme ci-dessus est un bootloader (pour MBR, il n’a pas de BPB) qui définit `cs` / `ds` / `ss` tp *7c00h* et lance le compteur d’emplacement à partir de 0. Un octet au décalage *X* dans le fichier est au décalage *X* dans le segment *7c00h* et à l'adresse linéaire *7c00h + X.*
- Les interruptions doivent être désactivées car un IDT n'est pas défini pour le court trajet aller-retour en mode protégé.
- Le code utilise un hack pour enregistrer 6 octets de code. La structure chargée par `lgdt` est enregistrée dans le ... GDT lui-même, dans le descripteur null (le premier descripteur).

Pour une description des descripteurs GDT, voir le chapitre 3.4.3 du manuel Intel Volume 3A .

Lire Modes réel vs protégé en ligne: https://riptutorial.com/fr/x86/topic/3679/modes-reel-vs-protege

Chapitre 10: Notions fondamentales sur les registres

Examples

Registres 16 bits

Lorsque Intel a défini le 8086 d'origine, il s'agissait d'un processeur 16 bits avec un bus d'adresse 20 bits (voir ci-dessous). Ils ont défini 8 registres 16 bits polyvalents - mais leur ont donné des rôles spécifiques pour certaines instructions:

- `AX` Le registre des accumulateurs.
  De nombreux opcodes ont pris ce registre ou ont été plus rapides s'ils ont été spécifiés.
- `DX` Le registre de données.
  Cela était parfois combiné avec les 16 bits supérieurs d'une valeur de 32 bits avec `AX` - par exemple, comme résultat d'une multiplication.
- `CX` Le registre Count.
  Cela a été utilisé dans un certain nombre d'instructions orientées boucle comme compteur implicite pour ces boucles - par exemple `LOOPNE` (boucle si elle n'est pas égale) et `REP` (mouvement répété / comparaison).
- `BX` Le registre de base.
  Cela pourrait être utilisé pour indexer la base d'une structure en mémoire - aucun des registres ci-dessus ne pourrait être utilisé pour indexer directement dans la mémoire.
- `SI` Le registre d'index source.
  C'était l'index source implicite en mémoire pour certaines opérations de déplacement et de comparaison.
- `DI` Le registre d'index de destination.
  C'était l'index de destination implicite dans la mémoire pour certaines opérations de déplacement et de comparaison.
- `SP` Le registre Stack Pointer.
  C'est le registre le moins polyvalent de l'ensemble! Il indiquait la position actuelle dans la pile, qui était utilisée explicitement pour les opérations `PUSH` et `POP` , implicitement pour `CALL` et `RET` avec des sous-routines, et TRÈS implicitement pendant les interruptions. En tant que tel, l'utiliser pour autre chose était dangereux pour votre programme!
- `BP` Le registre de pointeur de base.
  Lorsque les sous-routines appellent d'autres sous-routines, la pile contient plusieurs "cadres de pile". `BP` pourrait être utilisé pour contenir le frame de pile actuel, puis lorsqu'un nouveau sous-programme était appelé, il devrait être sauvegardé sur la pile, le nouveau frame de pile créé et utilisé, et après retour du sous-programme interne .

Remarques:

1. Les trois premiers registres ne peuvent pas être utilisés pour l'indexation en mémoire.

2. `BX` , `SI` et `DI` par défaut dans le segment de données actuel (voir ci-dessous).

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

3. `DI` , lorsqu'il est utilisé dans les opérations de mémoire à mémoire tels que `MOVS` et `CMPS` , utilise uniquement le segment supplémentaire (voir ci - dessous). Cela ne peut pas être remplacé.

4. `SP` et `BP` utilisent le segment de pile (voir ci-dessous) par défaut.

Registres 32 bits

Lorsque Intel a produit le 80386, ils sont passés d'un processeur 16 bits à un processeur 32 bits. Le traitement 32 bits signifie deux choses: à la fois les données manipulées étaient 32 bits et les adresses mémoire auxquelles on accédait étaient 32 bits. Pour ce faire, tout en restant compatibles avec leurs processeurs antérieurs, ils ont introduit de nouveaux modes pour le processeur. C'était soit en mode 16 bits, soit en mode 32 bits - mais vous pouvez remplacer ce mode instruction par instruction pour les données, l'adressage ou les deux!

Tout d'abord, ils devaient définir des registres 32 bits. Pour ce faire, ils ont simplement étendu les huit bits existants de 16 à 32 bits et leur ont donné des noms "étendus" avec un préfixe `E` : `EAX` , `EBX` , `ECX` , `EDX` , `ESI` , `EDI` , `EBP` et `ESP` . Les 16 bits inférieurs de ces registres étaient les mêmes que précédemment, mais les moitiés supérieures des registres étaient disponibles pour les opérations 32 bits telles que `ADD` et `CMP` . Les moitiés supérieures n'étaient pas accessibles séparément comme elles l'avaient fait avec les registres à 8 bits.

Le processeur devait disposer de modes 16 et 32 bits distincts car Intel utilisait les mêmes codes d'opération pour la plupart des opérations: `CMP AX,DX` en mode 16 bits et `CMP EAX,EDX` en mode 32 bits avaient exactement les mêmes codes d'opération. ! Cela signifiait que le même code ne pouvait pas être exécuté dans les deux modes:

> Le code opération de "Move immediate into `AX` " est `0xB8` , suivi de deux octets de la valeur immédiate: `0xB8 0x12 0x34`
>
> L'opcode pour "Move immediate into `EAX` " est `0xB8` , suivi de ***quatre*** octets de la valeur immédiate: `0xB8 0x12 0x34 0x56 0x78`

Ainsi, l'assembly doit savoir dans quel mode se trouve le processeur lors de l'exécution du code, afin qu'il sache émettre le nombre d'octets correct.

Registres 8 bits

Les quatre premiers registres à 16 bits peuvent avoir leurs propres octets supérieurs et inférieurs accessibles directement en tant que leurs propres registres:

- `AH` et `AL` sont les moitiés haute et basse du registre `AX` .
- `BH` et `BL` sont les moitiés haute et basse du registre `BX` .
- `CH`

et `CL` sont les moitiés haute et basse du registre `CX` .
- `DH` et `DL` sont les moitiés haute et basse du registre `DX` .

Notez que si vous modifiez `AH` ou `AL` , vous modifierez immédiatement `AX` ! Notez également que toute opération sur un registre à 8 bits ne pourrait pas affecter son "partenaire" - l'incrémentation de `AL` telle qu'elle déborde de `0xFF` à `0x00` ne modifierait pas `AH` .

Les registres 64 bits ont également des versions 8 bits représentant leurs octets inférieurs:

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

Il en va de même pour les registres `R8` à `R15` : leurs parties respectives d'octets inférieurs sont nommées `R8B` - `R15B` .

Registres de segments

Segmentation

Lorsque Intel concevait l'original 8086, il existait déjà un certain nombre de processeurs 8 bits dotés de capacités 16 bits, mais ils souhaitaient produire un véritable processeur 16 bits. Ils souhaitaient également produire quelque chose de meilleur et de plus performant que ce qui existait déjà. Ils souhaitaient donc pouvoir accéder à plus de 65 536 octets de mémoire maximum impliqués par les registres d'adressage 16 bits.

Registres de segments originaux

Ils ont donc implémenté l'idée de "Segments" - un bloc de mémoire de 64 kilo-octets indexé par les registres d'adresses à 16 bits - qui pourrait être recréé pour traiter différentes zones de la mémoire totale. Pour contenir ces bases de segment, ils comprenaient des registres de segments:

- `CS` Le registre de segment de code.
  Cela contient le segment du code en cours d'exécution, indexé par le registre implicite `IP` (Instruction Pointer).
- `DS` Le registre de segment de données.
  Cela contient le segment par défaut pour les données manipulées par le programme.
- `ES` Le registre Extra Segment.
  Cela contient un deuxième segment de données, pour des opérations de données simultanées sur toute la mémoire.
- `SS` Le registre de segment de pile.
  Cela contient le segment de mémoire qui contient la pile actuelle.

Taille du segment?

Les registres de segment peuvent avoir n'importe quelle taille, mais leur largeur de 16 bits facilite l'interopérabilité avec les autres registres. La question suivante était la suivante: les segments devraient-ils se chevaucher et, si oui, dans quelle mesure? La réponse à cette question dicterait la taille totale de la mémoire accessible.

S'il n'y avait pas de chevauchement du tout, l'espace d'adressage serait alors de 32 bits - 4 gigaoctets - une taille totalement inconnue à l'époque! Un chevauchement plus "naturel" de 8 bits produirait un espace d'adressage de 24 bits, soit 16 mégaoctets. En fin de compte, Intel a décidé de sauvegarder quatre autres adresses sur le processeur en rendant l'espace d'adresse de 1 mégaoctet avec un chevauchement de 12 bits.

Plus de registres de segments!

Lorsque Intel a conçu le 80386, ils ont reconnu que la suite existante de 4 registres de segments ne suffisait pas à la complexité des programmes qu'ils souhaitaient pouvoir prendre en charge. Ils ont donc ajouté deux autres:

- `FS` Le registre du segment lointain
- `GS` Le registre du segment global

Ces nouveaux registres de segment n'avaient aucune utilisation imposée par le processeur: ils étaient simplement disponibles pour tout ce que le programmeur voulait.

> Certains disent que les noms ont été choisis simplement pour continuer le thème `C` , `D` , `E` de l'ensemble existant ...

Registres 64 bits

AMD est un fabricant de processeurs qui avait autorisé la conception du 80386 d'Intel pour produire des versions compatibles, mais concurrentes. Ils ont apporté des modifications internes à la conception pour améliorer le débit ou d'autres améliorations apportées à la conception, tout en pouvant exécuter les mêmes programmes.

Pour un seul Intel, ils ont proposé des extensions 64 bits à la conception Intel 32 bits et ont produit la première puce 64 bits qui pouvait encore exécuter du code x86 32 bits. Intel a suivi la conception d'AMD dans ses versions de l'architecture 64 bits.

La conception 64 bits a apporté un certain nombre de modifications au jeu de registres, tout en restant compatible avec les versions antérieures:

- Les registres à usage général existants ont été étendus à 64 bits et nommés avec un préfixe `R` : `RAX` , `RBX` , `RCX` , `RDX` , `RSI` , `RDI` , `RBP` et `RSP` .

  > Encore une fois, les moitiés inférieures de ces registres étaient les mêmes que

les registres de préfixe `E` , et les moitiés supérieures ne pouvaient pas être consultées indépendamment.

- 8 registres 64 bits supplémentaires ont été ajoutés et non nommés mais simplement numérotés: `R8` , `R9` , `R10` , `R11` , `R12` , `R13` , `R14` et `R15` .
  - La moitié basse de 32 bits de ces registres est `R8D` à `R15D` (D pour DWORD comme d'habitude).
  - Les 16 bits les plus bas de ces registres sont accessibles en ajoutant un `W` au nom du registre: `R8W` à `R15W` .
- Les 8 bits les plus bas des *16* registres sont désormais accessibles:
  - Les traditionnels `AL` , `BL` , `CL` et `DL` ;
  - Les octets bas des registres de pointeurs (traditionnellement): `SIL` , `DIL` , `BPL` et `SPL` ;
  - Et les octets bas des 8 nouveaux registres: `R8B` à `R15B` .
  - Cependant, `AH` , `BH` , `CH` et `DH` sont inaccessibles dans les instructions qui utilisent un préfixe REX (pour une taille d'opérande de 64 bits, ou pour accéder à R8-R15, ou pour accéder à `SIL` , `DIL` , `BPL` ou `SPL`). Avec un préfixe REX, le modèle de bit de code machine qui signifiait `AH` au lieu de cela signifie `SPL` , et ainsi de suite. Voir le tableau 3-1 du manuel de référence des instructions d'Intel (volume 2).

L'écriture dans un registre 32 bits met toujours à zéro les 32 bits supérieurs du registre pleine largeur, contrairement à l'écriture dans un registre 8 ou 16 bits (qui fusionne avec l'ancienne valeur, ce qui constitue une dépendance supplémentaire pour l'exécution hors ordre)).

Registre des drapeaux

Lorsque l'unité ALU (Arithmetic Logic Unit) x86 effectue des opérations comme `NOT` et `ADD` , elle marque les résultats de ces opérations ("devenu zéro", "débordé", "devenu négatif") dans un registre `FLAGS` spécial de 16 bits. Les processeurs 32 bits l'ont mis à niveau à 32 bits et l'ont appelé `EFLAGS` , tandis que les processeurs 64 bits l'ont mis à 64 bits et l'ont appelé `RFLAGS` .

Codes de condition

Mais quel que soit le nom, le registre n'est pas directement accessible (sauf pour quelques instructions - voir ci-dessous). Au lieu de cela, les indicateurs individuels sont référencés dans certaines instructions, telles que le saut conditionnel ou le jeu conditionnel, appelés `Jcc` et `SETcc` où `cc` signifie «code de condition» et fait référence au tableau suivant:

Code de condition	prénom	Définition
`E` , `Z`	Égal, zéro	`ZF` == 1
`NE` , `NZ`	Pas égal, pas zéro	`ZF` == 0
`O`	Débordement	`OF` == 1
`NO`	Pas de débordement	`OF` == 0

Code de condition	prénom	Définition
`S`	Signé	`SF` == 1
`NS`	Pas signé	`SF` == 0
`P`	Parité	`PF` == 1
`NP`	Pas de parité	`PF` == 0
----------	----	----------
`C` , `B` , `NAE`	Porter, en dessous, pas au-dessus ou égal	`CF` == 1
`NC` , `NB` , `AE`	Pas de transport, pas au-dessous, au-dessus ou égal	`CF` == 0
`A` , `NBE`	Ci-dessus, non inférieur ou égal	`CF` == 0 et `ZF` == 0
`NA` , `BE`	Non supérieur, inférieur ou égal	`CF` == 1 ou `ZF` == 1
----------	----	----------
`GE` , `NL`	Plus grand ou égal, pas moins	`SF` == `OF`
`NGE` , `L`	Pas plus grand ou égal, moins	`SF` ! = `OF`
`G` , `NLE`	Plus grand, pas moins ou égal	`ZF` == 0 et `SF` == `OF`
`NG` , `LE`	Pas plus grand, moins ou égal	`ZF` == 1 ou `SF` ! = `OF`

En 16 bits, soustraire `1` de `0` est `65,535` ou `-1` selon que l'arithmétique non signée ou signée est utilisée - mais la destination contient `0xFFFF` deux cas. Ce n'est qu'en interprétant les codes de condition que la signification est claire. C'est encore plus révélateur si `1` est soustrait de `0x8000` : en arithmétique non signée, cela change simplement `32,768` en `32,767` ; tandis qu'en arithmétique signée, elle change de `-32,768` en `32,767` - un débordement beaucoup plus remarquable!

Les codes de condition sont regroupés en trois blocs dans le tableau: sign-unselevant, unsigned et signed. La désignation à l'intérieur des deux derniers blocs utilise "Above" et "Below" pour les non signés et "Greater" ou "Less" pour les signés. Donc, `JB` serait "Jump if Below" (non signé), alors que `JL` serait "Jump if Less" (signé).

Accéder à `FLAGS` directement

Les codes de condition ci-dessus sont utiles pour interpréter des concepts prédéfinis, mais les bits de drapeau réels sont également disponibles directement avec les deux instructions suivantes:

- `LAHF` Charge `AH` enregistre avec Flags
- `SAHF` Store `AH` s'inscrire dans les drapeaux

Seuls certains drapeaux sont copiés avec ces instructions. L'ensemble du `EFLAGS` `FLAGS` / `EFLAGS` / `RFLAGS` peut être sauvegardé ou restauré sur la pile:

- `PUSHF` / `POPF` Poussée / Pop `FLAGS` 16 bits sur / depuis la pile
- `PUSHFD` / `POPFD` Push / Pop 32-bit `EFLAGS` sur / depuis la pile
- `PUSHFQ` / `POPFQ` Push / Pop `RFLAGS` 64 bits sur / depuis la pile

Notez que les interruptions enregistrent et restaurent automatiquement le registre `[R/E]FLAGS` actuel.

Autres drapeaux

Outre les indicateurs ALU décrits ci-dessus, le registre `FLAGS` définit d'autres indicateurs d'état du système:

- `IF` le drapeau d'interruption.
  Ceci est défini avec l'instruction `STI` pour activer globalement les interruptions et effacé avec l'instruction `CLI` pour désactiver globalement les interruptions.
- `DF` Le drapeau de direction.
  Les opérations de mémoire à mémoire , tels que `CMPS` et `MOVS` (à comparer et à se déplacer entre des emplacements de mémoire) incrémenter automatiquement ou décrémenter les registres d'index en tant que partie de l'instruction. Le drapeau `DF` celui qui se produit: s'ils sont effacés avec l'instruction `CLD` , ils sont incrémentés; si elles sont définies avec l'instruction `STD` , elles sont décrémentées.
- `TF` Le drapeau de piège. Ceci est un indicateur de débogage. Le paramétrer mettra le processeur en mode "pas à pas": après chaque instruction, il appellera le "gestionnaire d'interruption en une seule étape", qui doit être géré par un débogueur. Il n'y a pas d'instructions pour définir ou effacer cet indicateur: vous devez manipuler le bit lorsqu'il est en mémoire.

80286 Drapeaux

Pour prendre en charge les nouvelles fonctionnalités multitâches du 80286, Intel a ajouté des indicateurs supplémentaires au registre `FLAGS` :

- `IOPL` Le niveau de privilège E / S.
  Pour protéger le code multitâche, certaines tâches nécessitaient des privilèges pour accéder aux ports d'E / S, tandis que d'autres devaient y accéder. Intel a introduit une échelle de privilège à quatre niveaux, 00_2 étant le plus privilégié et 11_2 le moins. Si `IOPL` était inférieur au niveau de privilège actuel, toute tentative d'accès aux ports d'E / S ou d'activation ou de désactivation des interruptions entraînerait une défaillance de protection générale.
- Indicateur de tâche imbriquée `NT` .
  Cet indicateur a été défini si une tâche `CALL` une autre tâche, ce qui a provoqué un

changement de contexte. Le drapeau défini a indiqué au processeur de faire un changement de contexte lorsque le `RET` été exécuté.

80386 Drapeaux

Le '386 avait besoin de drapeaux supplémentaires pour prendre en charge des fonctionnalités supplémentaires conçues dans le processeur.

- `RF` Le drapeau de reprise.
  Le `386 a ajouté des registres de débogage, qui pouvaient appeler le débogueur sur divers accès matériels, comme la lecture, l'écriture ou l'exécution d'un certain emplacement de mémoire. Cependant, lorsque le gestionnaire de débogage retournait pour exécuter l'instruction, *l'accès appelait immédiatement le gestionnaire de débogage!* Ou du moins si ce n'était pas pour l'indicateur de reprise, qui est automatiquement défini lors de l'entrée dans le gestionnaire de débogage, et effacé automatiquement après chaque instruction. Si l'indicateur de reprise est défini, le gestionnaire de débogage n'est pas appelé.
- `VM` The Virtual 8086 Flag.
  Pour prendre en charge le code 16 bits plus ancien ainsi que le code 32 bits plus récent, le 80386 pouvait exécuter des tâches 16 bits en mode "Virtual 8086", à l'aide d'un exécutif Virtual 8086. L'indicateur de `VM` a indiqué que cette tâche était une tâche virtuelle 8086.

80486 Drapeaux

Au fur et à mesure que l'architecture Intel s'améliorait, elle devenait plus rapide grâce à une technologie telle que les caches et l'exécution super-scalaire. Cela devait optimiser l'accès au système en faisant des hypothèses. Pour contrôler ces hypothèses, plus de drapeaux étaient nécessaires:

- Drapeau de vérification d'alignement `AC` L'architecture x86 peut toujours accéder à des valeurs de mémoire multi-octets sur n'importe quelle limite d'octets, contrairement à certaines architectures qui exigent qu'elles soient alignées (les valeurs de 4 octets doivent être sur 4 octets). Cependant, cela était moins efficace, car plusieurs accès mémoire étaient nécessaires pour accéder aux données non alignées. Si l'indicateur `AC` était défini, un accès non aligné déclencherait une exception plutôt que d'exécuter le code. De cette façon, le code pourrait être amélioré pendant le développement avec `AC` set, mais désactivé pour le code de production.

Drapeaux Pentium

Le Pentium a ajouté plus de support pour la virtualisation, plus le support de l'instruction `CPUID` :

- `VIF` L'indicateur d'interruption virtuelle.
  Ceci est une copie virtuelle de cette Tâche `IF` - si cette tâche veut ou non désactiver les interruptions, sans affecter réellement mondial Interruptions.
- `VIP` Le drapeau en attente d'interruption virtuelle.
  Cela indique qu'une interruption a été virtuellement bloquée par `VIF` , de sorte que lorsque la

tâche effectue une `STI` une interruption virtuelle peut être déclenchée.

- `ID` Le drapeau `CPUID` -allowed.
  Indique si cette tâche doit exécuter l'instruction `CPUID` . Un moniteur virtuel peut le refuser et "mentir" à la tâche demandeuse si elle exécute l'instruction.

Lire Notions fondamentales sur les registres en ligne:
https://riptutorial.com/fr/x86/topic/2122/notions-fondamentales-sur-les-registres

Chapitre 11: Optimisation

Introduction

La famille x86 existe depuis longtemps et, en tant que telle, de nombreuses astuces et techniques ont été découvertes et développées qui sont de notoriété publique - ou peut-être moins publiques. La plupart de ces astuces tirent parti du fait que de nombreuses instructions font effectivement la même chose, mais différentes versions sont plus rapides, permettent d'économiser de la mémoire ou n'affectent pas les indicateurs. Voici quelques astuces qui ont été découvertes. Chacun a ses avantages et ses inconvénients, donc devrait être répertorié.

Remarques

En cas de doute, vous pouvez toujours vous reporter au manuel de référence sur l'optimisation des architectures d'Intel 64 et IA-32 , qui est une excellente ressource de la société derrière l'architecture x86.

Examples

Remise à zéro d'un registre

La manière évidente de mettre un registre à zéro est de `MOV` dans un `0` - par exemple:

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

Notez qu'il s'agit d'une instruction de 5 octets.

Si vous souhaitez contourner les indicateurs (`MOV` n'affecte jamais les indicateurs), vous pouvez utiliser l'instruction `XOR` pour XOR-bit au niveau du registre avec lui-même:

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

Cette instruction ne nécessite que 2 octets et s'exécute plus rapidement sur tous les processeurs .

Déplacer le drapeau Carry dans un registre

Contexte

Si le drapeau Carry (`C`) contient une valeur que vous souhaitez mettre dans un registre, la méthode naïve consiste à faire quelque chose comme ceci:

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

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

Utilisez 'sbb'

Une manière plus directe, en évitant le saut, consiste à utiliser "Subtract with Borrow":

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

Si `C` est zéro, alors `al` sera zéro. Sinon, ce sera `0xFF` (`-1`). Si vous en avez besoin pour être `0x01` , ajoutez:

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

Avantages

- A peu près la même taille
- Deux ou un instructions en moins
- Pas de saut coûteux

Les inconvénients

- C'est opaque pour un lecteur peu familier avec la technique
- Il modifie les autres drapeaux

Tester un registre pour 0

Contexte

Pour savoir si un registre contient un zéro, la technique naïve consiste à le faire:

```
    cmp   eax, 0
```

Mais si vous regardez l'opcode pour cela, vous obtenez ceci:

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

Utiliser le `test`

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

Examinez l'opcode que vous obtenez:

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

Avantages

- Seulement deux octets!

Les inconvénients

- Opaque pour un lecteur peu familier avec la technique

Vous pouvez également jeter un œil à la question Q & R sur cette technique .

Appels système Linux avec moins de ballonnement

Dans Linux 32 bits, les appels système sont généralement effectués à l'aide de l'instruction sysenter (je dis généralement que les anciens programmes utilisent le maintenant `int 0x80` déconseillé `int 0x80`), mais cela peut prendre beaucoup de place dans un programme. peut raccourcir pour raccourcir et accélérer les choses.
C'est généralement la disposition d'un appel système sur Linux 32 bits:

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

C'est énorme! Mais il y a quelques astuces que nous pouvons tirer pour éviter ce désordre.
La première consiste à définir ebp à la valeur de esp diminuée de la taille de 3 registres 32 bits, soit 12 octets. C'est génial tant que vous ne voulez pas écraser ebp, edx et ecx avec des erreurs (par exemple lorsque vous déplacez une valeur dans ces registres de toute façon), nous pouvons le faire en utilisant l'instruction LEA pour ne pas avoir besoin pour affecter la valeur de ESP lui-même.

```
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
```

Cependant, nous n'avons pas fini, si l'appel système est sys_exit, nous pouvons éviter de pousser

n'importe quoi sur la pile!

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

Multipliez par 3 ou 5

Contexte

Pour obtenir le produit d'un registre et d'une constante et le stocker dans un autre registre, la manière naïve consiste à le faire:

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

Utiliser `lea`

Les multiplications sont des opérations coûteuses. Il est plus rapide d'utiliser une combinaison de changements et d'ajouts. Pour le cas particulier de multiplication du contenu d'un registre 32 ou 64 bits qui n'est pas `esp` ou `rsp` de 3 ou 5, vous pouvez utiliser l'instruction lea. Cela utilise le circuit de calcul d'adresse pour calculer le produit rapidement.

```
    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
```

De nombreux assembleurs comprendront aussi

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

Pour tous les multiplicables possibles autres que `ebp` ou `rbp` , la `rbp` l'instruction résultante est la même qu'avec l'utilisation d' `imul` .

Avantages

- Exécute beaucoup plus vite

Les inconvénients

- Si votre multiplicande est `ebp` ou `rbp` il faut un octet de plus en utilisant `imul`
- Plus à taper si votre assembleur ne prend pas en charge les raccourcis
- Opaque pour un lecteur peu familier avec la technique

Lire Optimisation en ligne: https://riptutorial.com/fr/x86/topic/3215/optimisation

Chapitre 12: Paging - Adressage virtuel et mémoire

Examples

introduction

Histoire

Les premiers ordinateurs

Les premiers ordinateurs avaient un bloc de mémoire dans lequel le programmeur mettait du code et des données, et le processeur était exécuté dans cet environnement. Étant donné que les ordinateurs étaient alors très coûteux, il était regrettable qu'ils fassent un travail, s'arrêtent et attendent que le prochain travail soit chargé, puis qu'ils le traitent.

Multi-utilisateurs, multi-traitement

Ainsi, les ordinateurs sont rapidement devenus plus sophistiqués et ont pris en charge plusieurs utilisateurs et / ou programmes simultanément - mais c'est à ce moment-là que les problèmes ont commencé à se poser avec l'idée simple d'un "bloc de mémoire". Si un ordinateur exécutait deux programmes simultanément ou exécutait le même programme pour plusieurs utilisateurs, ce qui aurait bien sûr exigé des données séparées pour chaque utilisateur, la gestion de cette mémoire devenait critique.

Exemple

Par exemple: si un programme a été écrit pour fonctionner à l'adresse mémoire 1000, mais qu'un autre programme y était déjà chargé, le nouveau programme n'a pas pu être chargé. Une façon de résoudre ce problème serait de faire fonctionner les programmes avec un "adressage relatif" - peu importe où le programme était chargé, il ne faisait que tout ce qui concernait l'adresse mémoire dans laquelle il était chargé.

Sophistication

Au fur et à mesure que le matériel informatique devenait plus sophistiqué, il était capable de prendre en charge des blocs de mémoire plus importants, autorisant davantage de programmes simultanés, et il devenait plus difficile d'écrire des programmes qui n'interféraient pas avec ce qui était déjà chargé. Une référence de mémoire parasite pourrait faire tomber non seulement le programme en cours, mais tout autre programme en mémoire, y compris le système d'exploitation lui-même!

Solutions

Ce qui était nécessaire était un mécanisme permettant à des blocs de mémoire d'avoir *des* adresses *dynamiques* . De cette façon, un programme pourrait être écrit pour fonctionner avec ses blocs de mémoire à des adresses reconnues - et ne pas pouvoir accéder à d'autres blocs pour d'autres programmes (à moins d'une coopération).

Segmentation

Un mécanisme implémenté était la segmentation. Cela permettait de définir des blocs de mémoire de différentes tailles et le programme devait définir le segment auquel il souhaitait accéder en permanence.

Problèmes

Cette technique était puissante, mais sa flexibilité même posait problème. Étant donné que les segments subdivisaient essentiellement la mémoire disponible en blocs de tailles différentes, la gestion de la mémoire pour ces segments posait un problème: allocation, désallocation, croissance, réduction, fragmentation.

Pagination

Une autre technique divisait toute la mémoire en blocs de même taille, appelés "Pages", ce qui rendait très simples les routines d'allocation et de désallocation et éliminait la croissance, la réduction et la fragmentation (à l'exception de la fragmentation interne gaspillage).

Adressage virtuel

En divisant la mémoire en ces blocs, ils pourraient être alloués à différents programmes selon les besoins, quelle que soit l'adresse à laquelle le programme en avait besoin. Ce "mapping" entre l'adresse physique de la mémoire et l'adresse souhaitée par le programme est très puissant et constitue la base de la gestion de la mémoire de tous les processeurs majeurs (Intel, ARM, MIPS, Power et autres).

Prise en charge du matériel et du système d'exploitation

Le matériel effectuait le remappage automatiquement et continuellement, mais nécessitait de la mémoire pour définir les tables de ce qu'il fallait faire. Bien sûr, le ménage associé à ce remappage devait être contrôlé par quelque chose. Le système d'exploitation devrait distribuer la mémoire selon les besoins et gérer les tables de données requises par le matériel pour prendre en charge les programmes requis.

Fonctions de pagination

Une fois que le matériel a pu faire ce remappage, qu'est-ce que cela permettait? Le principal moteur était le multitraitement - la possibilité d'exécuter plusieurs programmes, chacun avec sa propre mémoire, protégée l'un de l'autre. Mais deux autres options incluaient "données rares" et "mémoire virtuelle".

Multitraitement

Chaque programme recevait son propre «espace d'adressage» virtuel - une gamme d'adresses sur lesquelles il était possible de mapper la mémoire physique, à toutes les adresses souhaitées. Tant qu'il y avait suffisamment de mémoire physique pour se déplacer (voir "Mémoire virtuelle" ci-dessous), de nombreux programmes pouvaient être pris en charge simultanément.

De plus, ces programmes ***ne pouvaient pas*** accéder à la mémoire qui n'était pas mappée dans leur espace d'adressage virtuel - la protection entre les programmes était automatique. Si les programmes devaient communiquer, ils pouvaient demander au système d'exploitation d'organiser un bloc de mémoire partagé - un bloc de mémoire physique qui était mappé simultanément dans les espaces d'adressage de deux programmes différents.

Données rares

Autoriser un espace d'adressage virtuel énorme (4 Go est typique, pour correspondre aux registres 32 bits généralement utilisés par ces processeurs) ne gaspille pas en soi la mémoire si de grandes zones de cet espace d'adresses ne sont pas cartographiées. Cela permet la création d'énormes structures de données où seules certaines parties sont mappées à la fois. Imaginez un tableau tridimensionnel de 1 000 octets dans chaque direction: cela prend normalement un milliard d'octets! Mais un programme pourrait réserver un bloc de son espace d'adressage virtuel pour "conserver" ces données, mais uniquement pour cartographier les petites sections au fur et à mesure de leur remplissage. Cela permet une programmation efficace, sans gaspiller de mémoire pour les données qui ne sont pas encore nécessaires.

Mémoire virtuelle

Ci-dessus, j'ai utilisé le terme "adressage virtuel" pour décrire l'adressage virtuel-physique effectué par le matériel. Ceci est souvent appelé "mémoire virtuelle" - mais ce terme correspond plus correctement à la technique d'utilisation de l'adressage virtuel pour prendre en charge une illusion de mémoire plus importante que celle réellement disponible.

Cela fonctionne comme ceci:

- À mesure que les programmes sont chargés et demandent plus de mémoire, le système d'exploitation fournit la mémoire à partir des ressources disponibles. En plus de garder une trace de la mémoire qui a été mappée, le système d'exploitation garde également une trace du moment où la mémoire est réellement utilisée - le matériel prend en charge le marquage des pages utilisées.
- Lorsque le système d'exploitation est à court de mémoire physique, il examine toute la mémoire qu'il a déjà distribuée, quelle que soit la page la moins utilisée ou la plus longue. Il

enregistre le contenu de cette page particulière sur le disque dur, se souvient de l'endroit où il se trouvait, le marque comme "non présent" sur le matériel du propriétaire d'origine, puis met à zéro la page et la remet au nouveau propriétaire.
- Si le propriétaire d'origine tente à nouveau d'accéder à cette page, le matériel notifie le système d'exploitation. Le système d'exploitation alloue ensuite une nouvelle page (peut-être encore une fois à l'étape précédente!), Charge le contenu de l'ancienne page, puis transmet la nouvelle page au programme d'origine.

    > Le point important à noter est que, puisque n'importe quelle page peut être associée à n'importe quelle adresse et que chaque page a la même taille, une page est aussi bonne que n'importe quelle autre - tant que le contenu reste le même!

- Si un programme accède à un emplacement de mémoire non mappé, le matériel notifie le système d'exploitation comme avant. Cette fois, le système d'exploitation note que ce n'était pas une page qui avait été sauvegardée, donc le reconnaît comme un bogue dans le programme et le termine!

    > C'est effectivement ce qui se passe lorsque votre application disparaît mystérieusement sur vous - peut-être avec un MessageBox du système d'exploitation. C'est aussi ce qui arrive (souvent) à provoquer un tristement célèbre Blue Screen ou Sad Mac - le programme buggé était en fait un pilote de système d'exploitation qui avait accès à la mémoire qu'il ne devait pas!

Décisions de paging

Les architectes de matériel ont dû prendre de grandes décisions concernant Paging, car la conception affecterait directement la conception du processeur! Un système très flexible aurait un coût élevé, nécessitant de grandes quantités de mémoire uniquement pour gérer l'infrastructure de pagination elle-même.

Quelle taille doit avoir une page?

Dans le matériel, l'implémentation la plus simple de la pagination serait de prendre une adresse et de la diviser en deux parties. La partie supérieure serait un indicateur de la page à laquelle accéder, tandis que la partie inférieure serait l'index de la page pour l'octet requis:

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

Il est vite devenu évident que de petites pages nécessiteraient de vastes index pour chaque programme: même la mémoire qui n'était pas mappée aurait besoin d'une entrée dans la table pour indiquer cela.

Au lieu de cela, un index multi-niveaux est utilisé. L'adresse est divisée en plusieurs parties (trois

sont indiquées dans l'exemple ci-dessous) et la partie supérieure (communément appelée "Répertoire") indexe la partie suivante et ainsi de suite jusqu'à ce que l'index d'octet final de la page finale soit décodé:

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

Cela signifie qu'un index de répertoire peut indiquer "non mappé" pour une grande partie de l'espace d'adressage, sans nécessiter de nombreux index de page.

Comment optimiser l'utilisation des tables de pages?

Chaque accès d'adresse que fera le processeur devra être mappé - le processus virtuel vers physique doit donc être aussi efficace que possible. Si le système à trois niveaux décrit ci-dessus devait être implémenté, cela signifierait que chaque accès à la mémoire serait en réalité trois accès: un dans le répertoire; un dans la table des pages; et enfin les données souhaitées elles-mêmes. Et si le processeur devait également effectuer des tâches de maintenance, par exemple pour indiquer que cette page avait été accédée ou écrite, cela nécessiterait encore plus d'accès pour mettre à jour les champs.

La mémoire peut être rapide, mais cela imposerait un triple ralentissement sur tous les accès à la mémoire pendant la pagination! Heureusement, la plupart des programmes ont une "localité de portée". En d'autres termes, s'ils accèdent à un emplacement en mémoire, les accès futurs seront probablement proches. Et comme les pages ne sont pas trop petites, cette conversion de mappage n'a besoin d'être effectuée que lors de l'accès à une nouvelle page: pas pour tous les accès.

Mais mieux encore serait d'implémenter un cache de pages récemment accédées, pas seulement la plus récente. Le problème serait de suivre les pages auxquelles on avait accédé et ce qui ne l'était pas - le matériel devrait parcourir le cache à chaque accès pour trouver la valeur mise en cache. Ainsi, le cache est implémenté comme un cache adressable par le contenu: au lieu d'être accédé par adresse, il est accessible par contenu - si les données demandées sont présentes, elles sont proposées, sinon un emplacement vide est marqué. Le cache gère tout cela.

Ce cache adressable par contenu est souvent appelé TLB (Translation Lookaside Buffer) et doit être géré par le système d'exploitation dans le cadre du sous-système d'adressage virtuel. Lorsque les répertoires ou les tables de pages sont modifiés par le système d'exploitation, il doit notifier le TLB pour mettre à jour ses entrées - ou simplement les invalider.

80386 Paging

Conception de haut niveau

Le 80386 est un processeur 32 bits, avec un espace mémoire adressable de 32 bits. Les concepteurs du sous-système Paging ont noté qu'une conception de page 4K était mappée sur

ces 32 bits de manière très nette - 10 bits, 10 bits et 12 bits:

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

Cela signifiait que l'index d'octet était de 12 bits de large, ce qui indexerait dans une page 4K. Les index de répertoire et de page étaient de 10 bits, ce qui ferait en sorte que chaque carte dans une table de 1 024 entrées - et si ces entrées de table étaient chacune de 4 octets, ce serait 4K par table: également une page!

Alors c'est ce qu'ils ont fait:

- Chaque programme aurait son propre répertoire, une page avec 1 024 entrées de page, chacune définissant où se trouvait la table de pages suivante - s'il y en avait une.
- Si c'était le cas, cette table de pages comporterait 1 024 entrées de page, chacune définissant où se trouvait la dernière page - s'il y en avait une.
- S'il y en avait, alors son Byte pourrait être lu directement.

Entrée de la page

Le répertoire de niveau supérieur et la table de pages de niveau supérieur sont composés de 1 024 entrées de page. La partie la plus importante de ces entrées est l'adresse de ce qu'elle indexe: une table de pages ou une page réelle. Notez que cette adresse n'a pas besoin des 32 bits complets - puisque tout est une page, seuls les 20 bits supérieurs sont significatifs. Ainsi, les 12 autres bits de l'entrée de page peuvent être utilisés pour d'autres choses: si le niveau suivant est même présent; le ménage quant à savoir si la page a été accédée ou écrite; et même si les écritures devraient même être autorisées!

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

Notez que si le bit `P` est égal à 0, le reste de l'entrée peut contenir tout ce que le système d'exploitation souhaite y placer, par exemple le contenu de la page sur le disque dur!

Registre de base de répertoire de pages (PDBR)

Si chaque programme a son propre répertoire, comment le matériel sait-il où commencer le mappage? Étant donné que le processeur n'exécute qu'un seul programme à la fois, il dispose

d'un registre de contrôle unique pour contenir l'adresse du répertoire du programme en cours. C'est le registre de base du répertoire de pages (`CR3`). Lorsque le système d'exploitation bascule entre différents programmes, il met à jour le `PDBR` avec le répertoire de pages correspondant au programme.

Défauts de page

Chaque fois que le processeur accède à la mémoire, il doit mapper l'adresse virtuelle indiquée dans l'adresse physique appropriée. Ceci est un processus en trois étapes:

1. Indexer les 10 premiers bits de l'adresse dans la page indiquée par le `PDBR` pour obtenir l'adresse de la table de pages appropriée;
2. Indexer les 10 prochains bits de l'adresse dans la page indiquée par l'annuaire pour obtenir l'adresse de la page appropriée;
3. Indexez les 12 derniers bits de l'adresse pour extraire les données de cette page.

Parce que les deux étapes 1. et 2. ci-dessus utilisent les entrées de page, chaque entrée peut indiquer un problème:

- Le niveau suivant peut être marqué "Non présent";
- Le niveau suivant peut être marqué comme "Lecture seule" - et l'opération est une écriture;
- Le niveau suivant peut être marqué comme "Superviseur" - et c'est le programme qui accède à la mémoire, pas le système d'exploitation.

Lorsqu'un problème est détecté par le matériel, au lieu d'effectuer l'accès, il génère une erreur: Interruption # 14, le défaut de page. Il remplit également certains registres de contrôle spécifiques avec les informations expliquant pourquoi l'erreur s'est produite: l'adresse référencée; si c'était un accès de superviseur; et si c'était une tentative d'écriture.

Le système d'exploitation est censé piéger cette erreur, décoder les registres de contrôle et décider quoi faire. S'il s'agit d'un accès non valide, il peut mettre fin au programme défaillant. S'il s'agit d'un accès à la mémoire virtuelle, le système d'exploitation doit allouer une nouvelle page (qui peut devoir libérer une page déjà utilisée!), La remplir avec le contenu requis (soit tous les zéros, soit le contenu précédent chargé du disque).), mappez la nouvelle page dans la table de pages appropriée, marquez-la comme étant présente, puis reprenez l'instruction de défaillance. Cette fois, l'accès progressera avec succès et le programme se poursuivra sans savoir que quelque chose de spécial s'est produit (à moins de regarder l'horloge!)

80486 Paging

Le sous-système de pagination 80486 était très similaire au sous-système 80386. Il était rétrocompatible et les seules nouvelles fonctionnalités consistaient à autoriser le contrôle du cache mémoire sur une base de page par page - les concepteurs de système d'exploitation pouvaient marquer des pages spécifiques comme ne devant pas être mises en cache ou utiliser différentes écritures ou réécritures. techniques de mise en cache.

À tous autres égards, l'exemple "80386 Paging" est applicable.

Pentium Paging

Lors de la mise au point du Pentium, la taille de la mémoire et les programmes qui fonctionnaient étaient de plus en plus importants. Le système d'exploitation a dû faire de plus en plus de travail pour maintenir le sous-système de pagination juste dans le nombre même d'index de page devant être mis à jour lorsque des programmes ou des ensembles de données volumineux étaient utilisés.

Les concepteurs de Pentium ont donc ajouté un truc simple: ils ont ajouté un bit supplémentaire dans les entrées du répertoire de pages pour indiquer si le niveau suivant était un tableau de pages (comme auparavant) ou s'ils étaient directement sur une page de 4 Mo! En ayant le concept de 4 Mo Pages, le système d'exploitation n'aurait pas à créer une table de pages et à la remplir avec 1 024 entrées indexant essentiellement des adresses 4 Ko plus élevées que la précédente.

Disposition de l'adresse

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

Disposition d'entrée d'annuaire

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

Bien sûr, cela a eu des ramifications:

- La page 4 Mo devait commencer sur une limite d'adresse de 4 Mo, tout comme les pages 4K devaient commencer sur une limite d'adresse 4K.
- Tous les 4 Mo devaient appartenir à un seul programme - ou être partagés par plusieurs.

C'était parfait pour les périphériques à grande mémoire, tels que les cartes graphiques, qui avaient de grandes fenêtres d'espace d'adressage qui devaient être mappées pour que le système d'exploitation puisse les utiliser.

Extension d'adresse physique (PAE)

introduction

Au fur et à mesure que les prix de la mémoire chutaient, les PC à processeur Intel étaient en mesure d'avoir accès à de plus en plus de RAM, ce qui a permis d'atténuer les problèmes rencontrés par de nombreux utilisateurs. Alors que la mémoire virtuelle permettait de "créer" virtuellement la mémoire - en échangeant les "anciens" contenus de page existants sur le disque dur pour permettre le stockage de "nouvelles" données - cela ralentissait l'exécution des programmes. sur et hors du disque dur.

Plus de RAM

Ce qui était nécessaire était la possibilité d'accéder à plus de RAM physique - mais c'était déjà un bus d'adresse 32 bits, donc toute augmentation nécessiterait des registres d'adresses plus importants. Ou serait-ce? Lors du développement du Pentium Pro (et même du Pentium M), en tant que point d'arrêt jusqu'à ce que des processeurs 64 bits puissent être produits, ajouter davantage de bits d'adresse physique (permettant plus de mémoire physique) *sans* changer le nombre de bits du registre. Cela pourrait être réalisé puisque les adresses virtuelles étaient de toute façon mappées sur des adresses physiques - tout ce qui était nécessaire pour changer était le système de cartographie.

Conception

Le système existant pouvait accéder à un maximum de 32 bits d'adresses physiques. L'augmentation nécessitait un changement complet de la structure de saisie de page, de 32 à 64 bits. Il a été décidé de conserver la granularité minimale à 4K Pages, de sorte que l'entrée 64 bits aurait 52 bits d'adresse et 12 bits de contrôle (comme l'entrée précédente avait 20 bits d'adresse et 12 bits de contrôle).

Avoir une entrée de 64 bits, mais une taille de page (encore) de 4 Ko, signifiait qu'il n'y aurait que 512 entrées par table de pages ou répertoire, au lieu des 1 024 précédentes. Cela signifiait que l'adresse virtuelle 32 bits serait divisée différemment:

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

Couper un bit à la fois de l'index de répertoire et de l'index de page donnait deux bits pour un troisième niveau de mappage: ils appelaient cela la table PDPT (Table Directory Point Table), une table contenant exactement quatre entrées 64 bits au lieu de la précédente. un. Le PDBR (`CR3`)

pointe maintenant vers le PDPT à la place - qui, étant donné que le `CR3` n'a que 32 bits, doit être stocké dans les 4 premiers Go de RAM pour être accessible. Notez que puisque les bits bas de `CR3` sont utilisés pour le contrôle, le PDPT doit commencer sur une limite de 32 octets.

Extension de taille de page (PSE)

Et comme les pages précédentes de 4 Mo étaient une si bonne idée, elles souhaitaient pouvoir prendre en charge de grandes pages. Cette fois-ci, la suppression de la dernière couche du système de niveau n'a pas produit 10 pages de 12 Mo ou 4 Mo, mais 9 Mo de pages de 2 bits à la place.

PSE-32 (et PSE-40)

Étant donné que le mode d'extension d'adresse physique (PAE) introduit dans Pentium Pro (et Pentum M) était un tel changement du sous-système de gestion de la mémoire du système d'exploitation, Intel a conçu le mode de page «normal» pour prendre en charge les nouveaux bits d'adresse physique du processeur dans les entrées 32 bits définies précédemment.

Ils ont réalisé que quand une page de 4 Mo était utilisée, l'entrée de répertoire ressemblait à ceci:

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

Les zones Dir Index et Control de l'entrée étaient les mêmes, mais le bloc de bits inutilisés entre eux - qui serait utilisé par l'index de page s'il existait - était gaspillé. Ils ont donc décidé d'utiliser cette zone *pour définir les bits d'adresse physique supérieurs à 31* !

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

Cela permettait aux systèmes d'exploitation qui n'adoptaient pas le mode PAE d'accéder à une mémoire vive supérieure à 4 Go - avec un peu de logique supplémentaire, ils pouvaient fournir de grandes quantités de mémoire vive supplémentaire au système, sans dépasser les 4 Go normaux. Au début, seuls 4 bits ont été ajoutés, ce qui permet un adressage physique sur 36 bits. Ce mode s'appelait donc Extension de format de page 36 (PSE-36). En réalité, cela n'a pas modifié la taille de la page, mais uniquement l'adressage.

La limitation de ceci était que seulement 4MB Pages au-dessus de 4 Go étaient définissables - 4K Pages n'étaient pas autorisées. L'adoption de ce mode n'était pas très large - il était apparemment plus lent que d'utiliser PAE, et Linux ne l'a jamais utilisé.

Néanmoins, dans les processeurs ultérieurs qui avaient encore plus de bits d'adresse physique, AMD et Intel ont tous deux élargi la zone PSE à 8 bits, ce que certains appellent "PSE-40".

Lire Paging - Adressage virtuel et mémoire en ligne:

https://riptutorial.com/fr/x86/topic/3218/paging---adressage-virtuel-et-memoire

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Intel x86 Assembly Language & Microarchitecture	Community, David Hoelzer, Peter Cordes, Peter Mortensen, PyNEwbie, Runner
2	Assembleurs	John Burger
3	Conventions d'appel	Cody Gray, icktoofay, Margaret Bloom, Michael Petch, Peter Cordes, user45891, Zopesconk
4	Conversion de chaînes décimales en nombres entiers	Margaret Bloom, MikeCAT
5	Flux de contrôle	Margaret Bloom, owacoder, Ped7g, Zopesconk
6	Gestion multiprocesseur	Margaret Bloom, Michael Petch, RamenChef
7	Manipulation de données	Ped7g, Zopesconk
8	Mécanismes d'appel système	owacoder
9	Modes réel vs protégé	John Burger, Margaret Bloom
10	Notions fondamentales sur les registres	hidefromkgb, John Burger, Ped7g, Peter Cordes
11	Optimisation	Cody Gray, Downvoter, faissaloo, John Burger, sannaj, Stephen Leppik
12	Paging - Adressage virtuel et mémoire	John Burger