FREE eBook

LEARNING

Intel x86 Assembly Language & Microarchitecture

Free unaffiliated eBook created from
Stack Overflow contributors.

#x86

Table of Contents

About

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

Chapter 1: Getting started with Intel x86 Assembly Language & Microarchitecture

Remarks

This section provides an overview of what x86 is, and why a developer might want to use it.

It should also mention any large subjects within x86, and link out to the related topics. Since the Documentation for x86 is new, you may need to create initial versions of those related topics.

Examples

x86 Assembly Language

The family of x86 assembly languages represents decades of advances on the original Intel 8086 architecture. In addition to there being several different dialects based on the assembler used, additional processor instructions, registers and other features have been added over the years while still remaining backwards compatible to the 16-bit assembly used in the 1980s.

The first step to working with x86 assembly is to determine what the goal is. If you are seeking to write code within an operating system, for example, you will want to additionally determine whether you will choose to use a stand-alone assembler or built-in inline assembly features of a higher level language such as C. If you wish to code down on the "bare metal" without an operating system, you simply need to install the assembler of your choice and understand how to create binary code that can be turned into flash memory, bootable image or otherwise be loaded into memory at the appropriate location to begin execution.

A very popular assembler that is well supported on a number of platforms is NASM (Netwide Assembler), which can be obtained from http://nasm.us/. On the NASM site you can proceed to download the latest release build for your platform.

Windows

Both 32-bit and 64-bit versions of NASM are available for Windows. NASM comes with a convenient installer that can be used on your Windows host to install the assembler automatically.

Linux

It may well be that NASM is already installed on your version of Linux. To check, execute:

```
nasm -v
```

If the command is not found, you will need to perform an install. Unless you are doing something that requires bleeding edge NASM features, the best path is to use your built-in package management tool for your Linux distribution to install NASM. For example, under Debian-derived

systems such as Ubuntu and others, execute the following from a command prompt:

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

For RPM based systems, you might try:

```
sudo yum install nasm
```

Mac OS X

Recent versions of OS X (including Yosemite and El Capitan) come with an older version of NASM pre-installed. For example, El Capitan has version 0.98.40 installed. While this will likely work for almost all normal purposes, it is actually quite old. At this writing, NASM version 2.11 is released and 2.12 has a number of release candidates available.

You can obtain the NASM source code from the above link, but unless you have a specific need to install from source, it is far simpler to download the binary package from the OS X release directory and unzip it.

Once unzipped, it is strongly recommended that you *not* overwrite the system-installed version of NASM. Instead, you might install it into /usr/local:

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

At this point, NASM is in `/usr/local/bin`, but it is not in your path. You should now add the following line to the end of your profile:

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

This will prepend `/usr/local/bin` to your path. Executing `nasm -v` at the command prompt should now display the proper, newer, version.

x86 Linux Hello World Example

This is a basic Hello World program in NASM assembly for 32-bit x86 Linux, using system calls directly (without any libc function calls). It's a lot to take in, but over time it will become understandable. Lines starting with a semicolon(`;`) are comments.

If you don't already know low-level Unix systems programming, you might want to just write functions in asm and call them from C or C++ programs. Then you can just worry about learning how to handle registers and memory, without also learning the POSIX system-call API and the ABI for using it.

This makes two system calls: `write(2)` and `_exit(2)` (not the `exit(3)` libc wrapper that flushes stdio

buffers and so on). (Technically, `_exit()` calls sys_exit_group, not sys_exit, but that only matters in a multi-threaded process.) See also `syscalls(2)` for documentation about system calls in general, and the difference between making them directly vs. using the libc wrapper functions.

In summary, system calls are made by placing the args in the appropriate registers, and the system call number in `eax`, then running an `int 0x80` instruction. See also What are the return values of system calls in Assembly? for more explanation of how the asm syscall interface is documented with mostly C syntax.

The syscall call numbers for the 32-bit ABI are in `/usr/include/i386-linux-gnu/asm/unistd_32.h` (same contents in `/usr/include/x86_64-linux-gnu/asm/unistd_32.h`).

`#include <sys/syscall.h>` will ultimately include the right file, so you could run `echo '#include <sys/syscall.h>' | gcc -E - -dM | less` to see the macro defs (see this answer for more about finding constants for asm in C headers)

```
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
```

On Linux, you can save this file as `Hello.asm` and build a 32-bit executable from it with these commands:

```
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
```

See this answer for more details on building assembly into 32 or 64-bit static or dynamically linked Linux executables, for NASM/YASM syntax or GNU AT&T syntax with GNU `as` directives. (Key point: make sure to use `-m32` or equivalent when building 32-bit code on a 64-bit host, or you will have confusing problems at run-time.)

You can trace it's execution with `strace` to see the system calls it makes:

```
$ 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 +++
```

The trace on stderr and the regular output on stdout are both going to the terminal here, so they interfere in the line with the `write` system call. Redirect or trace to a file if you care. Notice how this lets us easily see the syscall return values without having to add code to print them, and is actually even easier than using a regular debugger (like gdb) for this.

The x86-64 version of this program would be extremely similar, passing the same args to the same system calls, just in different registers. And using the `syscall` instruction instead of `int 0x80`.

Read Getting started with Intel x86 Assembly Language & Microarchitecture online: https://riptutorial.com/x86/topic/1164/getting-started-with-intel-x86-assembly-language---microarchitecture

Chapter 2: Assemblers

Examples

Microsoft Assembler - MASM

Given that the 8086/8088 was used in the IBM PC, and the Operating System on that was most often from Microsoft, Microsoft's assembler MASM was the de facto standard for many years. It followed Intel's syntax closely, but permitted some convenient but "loose" syntax that (in hindsight) only caused confusion and errors in code.

A perfect example is as follows:

```
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  ; ????
```

Does the last `MOV` instruction put the *contents* of `Symbol` into `CX`, or the *address* of `Symbol` into `CX`? Does `CX` end up with `0x1234` or `0x0102` (or whatever)? It turns out that `CX` ends up with `0x1234` - if you want the address, you need to use the `OFFSET` specifier

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

Intel Assembler

Intel wrote the specification of the 8086 assembly language, a derivative of the earlier 8080, 8008 and 4004 processors. As such, the assembler they wrote followed their own syntax precisely. However, this assembler wasn't used very widely.

Intel defined their opcodes to have either zero, one or two operands. The two-operand instructions were defined to be in the `dest`, `source` order, which was different from other assemblers at the time. But some instructions used implicit registers as operands - you just had to know what they were. Intel also used the concept of "prefix" opcodes - one opcode would affect the next 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 also broke a convention used by other assemblers: for each opcode, a different mnemonic was invented. This required subtly- or distinctly-different names for similar operations: e.g. `LDM` for "Load from Memory" and `LDI` for "Load Immediate". Intel used the one mnemonic `MOV` - and expected the assembler to work out which opcode to use from context. That caused many pitfalls and errors for programmers in the future when the assembler couldn't intuit what the programmer actually wanted...

AT&T assembler - as

Although the 8086 was most used in IBM PCs along with Microsoft, there were a number of other computers and Operating Systems that used it too: most notably Unix. That was a product of AT&T, and it already had Unix running on a number of other architectures. Those architectures used more conventional assembly syntax - especially that two-operand instructions specified them in `source`, `dest` order.

So AT&T assembler conventions overrode the conventions dictated by Intel, and a whole new dialect was introduced for the x86 range:

- Register names were prefixed by `%`:
  `%al`, `%bx` etc.
- Immediate values were prefied by `$`:
  `$4`
- Operands were in `source`, `dest` order
- Opcodes included their operand sizes:
  `movw $4, %ax ; Move word 4 into AX`

Borland's Turbo Assembler - TASM

Borland started out with a Pascal compiler that they called "Turbo Pascal". This was followed by compilers for other languages: C/C++, Prolog and Fortran. They also produced an assembler called "Turbo Assembler", which, following Microsoft's naming convention, they called "TASM".

TASM tried to fix some of the problems of writing code using MASM (see above), by providing a more strict interpretation of the source code under a specified `IDEAL` mode. By default it assumed `MASM` mode, so it could assemble MASM source directly - but then Borland found that they had to be bug-for-bug compatible with MASM's more "quirky" idiosyncracies - so they also added a `QUIRKS` mode.

Since TASM was (much) cheaper than MASM, it had a large user base - but not many people used IDEAL mode, despite its touted advantages.

GNU assembler - gas

When the GNU project needed an assembler for the x86 family, they went with the AT&T version (and its syntax) that was associated with Unix rather than the Intel/Microsoft version.

Netwide Assembler - NASM

NASM is by far the most ported assembler for the x86 architecture - it's available for practically every Operating System based on the x86 (even being included with MacOS), and is available as a cross-platform assembler on other platforms.

This assembler uses Intel syntax, but it is different from others because it focuses heavily on its own "macro" language - this permits the programmer to build up more complex expressions using simpler definitions, allowing new "instructions" to be created.

Unfortunately this powerful feature comes at a cost: the type of the data gets in the way of generalised instructions, so data typing is not enforced.

```
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!
```

However, NASM introduced one feature that others lacked: scoped symbol names. When you define a symbol in other assemblers, that name is available throughout the rest of the code - but that "uses up" that name, "polluting" the global name space with symbols.

For example (using NASM syntax):

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

After this definition, X and Y are forevermore defined. To avoid "using up" the names `X` and `Y`, you needed to use more definite names:

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

But NASM offers an alternative. By leveraging its "local variable" concept, you can define structure fields that require you to nominate the containing structure in future references:

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

Cursor ISTRUC     Point
       ENDISTRUC
```

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

Unfortunately, because NASM doesn't keep track of types, you can't use the more natural syntax:

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

Yet Another Assembler - YASM

YASM is a complete rewrite of NASM, but is compatible with both Intel and AT&T syntaxes.

Read Assemblers online: https://riptutorial.com/x86/topic/2403/assemblers

Chapter 3: Calling Conventions

Remarks

Resources

Overviews/comparisons: Agner Fog's nice calling convention guide. Also, x86 ABIs (wikipedia): calling conventions for functions, including x86-64 Windows and System V (Linux).

- SystemV x86-64 ABI (official standard). Used by all OSes but Windows. (This github wiki page, kept up to date by H.J. Lu, has links to 32bit, 64bit, and x32. Also links to the official forum for ABI maintainers/contributors.) Also note that clang/gcc sign/zero extend narrow args to 32bit, even though the ABI as written doesn't require it. Clang-generated code depends on it.
- SystemV 32bit (i386) ABI (official standard) , used by Linux and Unix. (old version).
- OS X 32bit x86 calling convention, with links to the others. The 64bit calling convention is System V. Apple's site just links to a FreeBSD pdf for that.
- Windows x86-64 `__fastcall` calling convention
- Windows `__vectorcall`: documents the 32bit and 64bit versions
- Windows 32bit `__stdcall`: used used to call Win32 API functions. That page links to the other calling convention docs (e.g. `__cdecl`).
- Why does Windows64 use a different calling convention from all other OSes on x86-64?: some interesting history, esp. for the SysV ABI where the mailing list archives are public and go back before AMD's release of first silicon.

Examples

32-bit cdecl

cdecl is a Windows 32-bit function calling convention which is *very* similar to the calling convention used on many POSIX operating systems (documented in the i386 System V ABI). One of the differences is in returning small structs.

Parameters

Parameters are passed on the stack, with the first argument at the lowest address on the stack at the time of the call (pushed last, so it's just above the return address on entry to the function). The

caller is responsible for popping parameters back off the stack after the call.

Return Value

For scalar return types, the return value is placed in EAX, or EDX:EAX for 64bit integers. Floating-point types are returned in st0 (x87). Returning larger types like structures is done by reference, with a pointer passed as an implicit first parameter. (This pointer is returned in EAX, so the caller doesn't have to remember what it passed).

Saved and Clobbered Registers

EBX, EDI, ESI, EBP, and ESP (and FP / SSE rounding mode settings) must be preserved by the callee, such that the caller can rely on those registers not having been changed by a call.

All other registers (EAX, ECX, EDX, FLAGS (other than DF), x87 and vector registers) may be freely modified by the callee; if a caller wishes to preserve a value before and after the function call, it must save the value elsewhere (such as in one of the saved registers or on the stack).

64-bit System V

This is the default calling convention for 64-bit applications on many POSIX operating systems.

Parameters

The first eight scalar parameters are passed in (in order) RDI, RSI, RDX, RCX, R8, R9, R10, R11. Parameters past the first eight are placed on the stack, with earlier parameters closer to the top of the stack. The caller is responsible for popping these values off the stack after the call if no longer needed.

Return Value

For scalar return types, the return value is placed in RAX. Returning larger types like structures is done by conceptually changing the signature of the function to add a parameter at the beginning of the parameter list that is a pointer to a location in which to place the return value.

Saved and Clobbered Registers

RBP, RBX, and R12–R15 are preserved by the callee. All other registers may be modified by the callee, and the caller must preserve a register’s value itself (e.g. on the stack) if it wishes to use that value later.

32-bit stdcall

stdcall is used for 32-bit Windows API calls.

Parameters

Parameters are passed on the stack, with the first parameter closest to the top of the stack. The callee will pop these values off of the stack before returning.

Return Value

Scalar return values are placed in EAX.

Saved and Clobbered Registers

EAX, ECX, and EDX may be freely modified by the callee, and must be saved by the caller if desired. EBX, ESI, EDI, and EBP must be saved by the callee if modified and restored to their original values on return.

32-bit, cdecl — Dealing with Integers

As parameters (8, 16, 32 bits)

8, 16, 32 bits integers are always passed, on the stack, as full width 32 bits values[1].
No extension, signed or zeroed, is needed.
The callee will just use the lower part of the full width values.

```
//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
```

As parameters (64 bits)

64 bits values are passed on the stack using two pushes, respecting the littel endian convention[2], pushing first the higher 32 bits then the lower ones.

```
//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
```

As return value

8 bits integers are returned in `AL`, eventually clobbering the whole `eax`.
16 bits integers are returned in `AX`, eventually clobbering the whole `eax`.
32 bits integers are returned in `EAX`.
64 bits integers are returned in `EDX:EAX`, where `EAX` holds the lower 32 bits and `EDX` the upper ones.

```
//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] This keep the stack aligned on 4 bytes, the natural word size. Also an x86 CPU can only push 2 or 4 bytes when not in long mode.

[2] Lower DWORD at lower address

32-bit, cdecl — Dealing with Floating Point

As parameters (float, double)

Floats are 32 bits in size, they are passed naturally on the stack.
Doubles are 64 bits in size, they are passed, on the stack, respecting the Little Endian convention[1], pushing first the upper 32 bits and than the lower ones.

```
//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
```

As parameters (long double)

Long doubles are 80 bits[2] wide, while on the stack a TBYTE could be stored with two 32 bits pushes and one 16 bit push (for 4 + 4 + 2 = 10), to keep the stack aligned on 4 bytes, it ends occupying 12 bytes, thus using three 32 bits pushes.
Respecting Little Endian convention, bits 79-64 are pushed first[3], then bits 63-32 followed by bits 31-0.

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

foo(3.1457);

;Call to foo in assembly
;3.1457 is 0x4000c9532617c1bda800

push DWORD 4000h        ;Bits 79-64, as 32 bits push
```

```
push DWORD 0c9532617h        ;Bits 63-32
push DWORD 0c1bda800h        ;Bits 31-0
call foo
add esp, 0ch

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

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

As return value

A floating point values, whatever its size, is returned in `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] Lower DWORD at lower address.

[2] Known as TBYTE, from Ten Bytes.

[3] Using a full width push with any extension, higher WORD is not used.

[4] Which is TBYE wide, note that contrary to the integers, FP are always returned with more precision that it is required.

64-bit Windows

Parameters

The first 4 parameters are passed in (in order) RCX, RDX, R8 and R9. XMM0 to XMM3 are used to pass floating point parameters.

Any further parameters are passed on the stack.

Parameters larger than 64bit are passed by address.

Spill Space

Even if the function uses less than 4 parameters the caller always provides space for 4 QWORD sized parameters on the stack. The callee is free to use them for any purpose, it is common to copy the parameters there if they would be spilled by another call.

Return Value

For scalar return types, the return value is placed in RAX. If the return type is larger than 64bits (e.g. for structures) RAX is a pointer to that.

Saved and Clobbered Registers

All registers used in parameter passing (RCX, RDX, R8, R9 and XMM0 to XMM3), RAX, R10, R11, XMM4 and XMM5 can be spilled by the callee. All other registers need to be preserved by the caller (e.g. on the stack).

Stack alignment

The stack must be kept 16-byte aligned. Since the "call" instruction pushes an 8-byte return address, this means that every non-leaf function is going to adjust the stack by a value of the form 16n+8 in order to restore 16-byte alignment.
It is the callers job to clean the stack after a call.

Source: The history of calling conventions, part 5: amd64 Raymond Chen

32-bit, cdecl — Dealing with Structs

Padding

Remember, members of a struct are usually padded to ensure they are aligned on their natural boundary:

```
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)
};
```

As parameters (pass by reference)

When passed by reference, a pointer to the struct in memory is passed as the first argument on the stack. This is equivalent to passing a natural-sized (32-bit) integer value; see 32-bit cdecl for specifics.

As parameters (pass by value)

When passed by value, structs are entirely copied on the stack, respecting the original memory layout (*i.e.*, the first member will be at the lower address).

```
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
```

As return value

Unless they are trivial[1], structs are copied into a caller-supplied buffer before returning. This is equivalent to having an hidden first parameter `struct S *retval` (where `struct S` is the type of the struct).

The function must return with this pointer to the return value in `eax`; The caller is allowed to depend on `eax` holding the pointer to the return value, which it pushed right before the `call`.

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

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

The hidden parameter is not added to the parameter count for the purposes of stack clean-up, since it must be handled by the callee.

```
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
```

In the example above, the structure will be saved at the top of the stack.

```
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] A "trivial" struct is one that contains only one member of a non-struct, non-array type (up to 32 bits in size). For such structs, the value of that member is simply returned in the `eax` register. (This behavior has been observed with GCC targeting Linux)

The Windows version of cdecl is different from the System V ABI's calling convention: A "trivial" struct is allowed to contain up to two members of a non-struct, non-array type (up to 32 bits in size). These values are returned in `eax` and `edx`, just like a 64-bit integer would be. (This behavior has been observed for MSVC and Clang targeting Win32.)

Read Calling Conventions online: https://riptutorial.com/x86/topic/3261/calling-conventions

Chapter 4: Control Flow

Examples

Unconditional jumps

```
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
```

Relative near jumps

`jmp a_label` is:

- **near**
  It only specify the offset part of the *logical address* of destination. The segment is assumed to be `CS`.
- **relative**
  The instruction semantic is jump *rel* bytes forward[1] from next instruction address or `IP = IP + rel`.

The instruction is encoded as either `EB <rel8>` or `EB <rel16/32>`, the assembler picking up the most appropriate form, usually preferring a shorter one.
Per assembler overriding is possible, for example with NASM `jmp SHORT a_label`, `jmp WORD a_label` and `jmp DWORD a_label` generate the three possible forms.

Absolute indirect near jumps

`jmp bx` and `jmp WORD [aPointer]` are:

- **near**
  They only specify the offset part of the logical address of destination. The segment is assumed to be `CS`.
- **absolute indirect**
  The semantic of the instructions is jump to the address in *reg* or *mem* or `IP = reg`, `IP = mem`.

The instruction is encoded as `FF /4`, for memory indirect the size of the operand is determined as for every other memory access.

Absolute far jumps

`jmp 7c0h:0000h` is:

- **far**
  It specifies both parts of the *logical* address: the segment and the offset.
- **absolute** The semantic of the instruction is jump to the address *segment:offset* or `CS = segment, IP = offset.`

The instruction is encoded as `EA <imm32/48>` depending on the code size.
It is possible to choose between the two forms in some assembler, for example with NASM `jmp 7c0h: WORD 0000h` and `jmp 7c0h: DWORD 0000h` generate the first and second form.

Absolute indirect far jumps

`jmp FAR WORD [aFarPointer]` is:

- **far** It specifies both parts of the *logical* address: the segment and the offset.
- **Absolute indirect** The semantic of the instruction is jump to the *segment:offset* stored in *mem*[2] or `CS = mem[23:16/32], IP = [15/31:0].`

The instruction is encoded as `FF /5`, the size of the operand can be controller with the size specifiers.
In NASM, a little bit non intuitive, they are `jmp FAR WORD [aFarPointer]` for a *16:16* operand and `jmp FAR DWORD [aFarPointer]` for a *16:32* operand.

Missing jumps

- **near absolute**
  Can be emulated with a near indirect jump.

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

- **far relative**
  Make no sense or too narrow of use anyway.

---

[1] Two complement is used to specify a signed offset and thus jump backward.
[2] Which can be a *seg16:off16* or a *seg16:off32*, of sizes *16:16* and *16:32*.

Testing conditions

In order to use a conditional jump a condition must be tested. **Testing a condition** here refers only to the act of checking the flags, the actual jumping is described under Conditional jumps.

x86 tests conditions by relying on the EFLAGS register, which holds a set of flags that each instruction can potentially set.

Arithmetic instructions, like `sub` or `add`, and logical instructions, like `xor` or `and`, obviously "set the flags". This means that the flags *CF*, *OF*, *SF*, *ZF*, *AF*, *PF* are modified by those instructions. Any instruction is allowed to modify the flags though, for example `cmpxchg` modifies the *ZF*.

Always check the instruction reference to know which flags are modified by a specific instruction.

x86 has a set of *conditional jumps*, referred to earlier, that jump if and only if some flags are set or some are clear or both.

Flags

Arithmetic and logical operations are very useful in setting the flags. For example after a `sub eax, ebx`, for now holding **unsigned** values, we have:

Flag	When set	When clear
ZF	When result is zero. *EAX - EBX = 0* $\Rightarrow$ EAX = EBX	When result is **not** zero. *EAX - EBX* $\neq 0 \Rightarrow$ EAX $\neq$ EBX
CF	When result did need carry for the MSb. *EAX - EBX < 0* $\Rightarrow$ EAX < EBX	When result did **not** need carry for the MSb. *EAX - EBX* $\not< 0 \Rightarrow \mathrm{EAX} \not< \mathrm{EBX}$
SF	When result MSb is set.	When result MSb is **not** set.
OF	When a signed overflow occurred.	When a signed overflow did **not** occur.
PF	When the number of bits set in least significant byte of result is even.	When the number of bits set in least significant byte of result is odd.
AF	When the lower BCD digit generated a carry. It is bit 4 carry.	When the lower BCD digit did **not** generate a carry. It is bit 4 carry.

Non-destructive tests

The `sub` and `and` instructions modify their destination operand and would require two extra copies (save and restore) to keep the destination unmodified.

To perform a non-destructive test there are the instructions `cmp` and `test`. They are identical to their destructive counterpart **except the result of the operation is discarded, and only the flags are saved**.

Destructive	Non destructive
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
```

Signed and unsigned tests

The CPU gives no special meaning to register values[1], sign is a programmer construct. **There is no difference when testing signed and unsigned values.** The processor computes enough flags to test the usual arithmetic relationships (equal, less than, greater than, etc.) both if the operands were to be considered signed and unsigned.

---

[1] Though it has some instructions that make sense only with specific formats, like two's complement. This is to make the code more efficient as implementing the algorithm in software would require a lot of code.

Conditional jumps

Based on the state of the flags the CPU can either execute or ignore a jump. An instruction that performs a jump based on the flags falls under the generic name of *Jcc - Jump on Condition Code* [1].

Synonyms and terminology

In order to improve the readability of the assembly code, Intel defined several synonyms for the same condition code. For example, `jae`, `jnb` and `jnc` are all the same condition code *CF = 0*.

While the instruction name may give a very strong hint on when to use it or not, the only meaningful approach is to recognize the flags that need to be tested and **then** choose the instructions appropriately.
Intel however gave the instructions names that make perfect sense when used after a `cmp` instruction. For the purposes of this discussion, `cmp` will be assumed to have set the flags before a conditional jump.

Equality

The operand are equal iff *ZF* has been set, they differ otherwise. To test for equality we need *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	Flags
`je`, `jz`	ZF = 1
`jne`, `jnz`	ZF = 0

Greater than

For **unsigned operands**, the destination is greater than the source if carry was not needed, that is, if *CF = 0*. When *CF = 0* it is possible that the operands were equal, testing *ZF* will disambiguate.

```
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	Flags
`jae`, `jnc`, `jnb`	CF = 0
`ja`, `jnbe`	CF = 0, ZF = 0

For **signed operands** we need to check that *SF = 0*, unless there has been a signed overflow, in which case the resulting *SF* is reversed. Since *OF = 0* if no signed overflow occurred and 1 otherwise, we need to check that *SF = OF*.

ZF can be used to implement a strict/non strict test.

```
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	Flags
`jge`, `jnl`	SF = OF

Instruction	Flags
`jg`, `jnle`	SF = OF, ZF = 0

Less than

These use the inverted conditions of above.

```
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	Flags
`jbe`, `jna`	CF = 1 or ZF = 1
`jb`, `jc`, `jnae`	CF = 1
`jle`, `jng`	SF != OF or ZF = 1
`jl`, `jnge`	SF != OF

Specific flags

Each flag can be tested individually with `j<flag_name>` where *flag_name* does not contain the trailing *F* (for example $CF \rightarrow C$, $PF \rightarrow P$).

The remaining codes not covered before are:

Instruction	Flag
`js`	SF = 1
`jns`	SF = 0
`jo`	OF = 1
`jno`	OF = 0

Instruction	Flag
`jp`, `jpe` (e = even)	PF = 1
`jnp`, `jpo` (o = odd)	PF = 0

One more conditional jump (extra one)

One special x86 conditional jump doesn't test flag. Instead it does test value of `cx` or `ecx` register (based on current CPU address mode being 16 or 32 bit), and the jump is executed when the register contains zero.

This instruction was designed for validation of *counter* register (`cx/ecx`) ahead of `rep`-like instructions, or ahead of `loop` loops.

```
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	Register (not flag)
`jcxz`, `jecxz`	cx = 0 (16b mode)
`jcxz`, `jecxz`	ecx = 0 (32b mode)

---

[1] Or something like that.

Test arithmetic relations

Unsigned integers

Greater than

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

Greater than or equal

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

Less than

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

Less than or equal

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

Equal

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

Not equal

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

Signed integers

Greater than

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

Greater than or equal

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

Less than

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

Less than or equal

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

Equal

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

Not equal

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

---

a_label

In examples above the `a_label` is target destination for CPU when the tested condition is "true". When tested condition is "false", the CPU will continue on the next instruction following the conditional jump.

Synonyms

There are instruction synonyms that can be used to improve the readability of the code.
For example `ja` and `jnbe` (Jump non below nor equal) are the same instruction.

Signed unsigned companion codes

Operation	Unsigned	Signed
>	`ja`	`jg`
>=	`jae`	`jge`
<	`jb`	`jl`
<=	`jbe`	`jle`
=	`je`	`je`
≠, !=, <>	`jne`	`jne`

Read Control Flow online: https://riptutorial.com/x86/topic/5808/control-flow

Chapter 5: Converting decimal strings to integers

Remarks

Converting strings to integers is one of common tasks.

Here we'll show how to convert decimal strings to integers.

Psuedo code to do this is:

```
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
```

Dealing with hexadecimal strings is a bit more difficult because character codes are typically not continuous when dealing with multiple character types such as digits(0-9) and alphabets(a-f and A-F). Character codes are typically continuous when dealing with only one type of characters (we'll deal with digits here), so we'll deal with only environments in which character codes for digit are continuous.

Examples

IA-32 assembly, GAS, cdecl calling convention

```
# 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
```

This GAS-style code will convert decimal string given as first argument, which is pushed on the stack before calling this function, to integer and return it via `%eax`. The value of `%esi` is saved because it is callee-save register and is used.

Overflow/wrapping and invalid characters are not checked in order to make the code simple.

In C, this code can be used like this (assuming `unsigned int` and pointers are 4-byte long):

```
#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;
}
```

Note: in some environments, two `string_to_integer` in the assembly code have to be changed to `_string_to_integer` (add underscore) in order to let it work with C code.

MS-DOS, TASM/MASM function to read a 16-bit unsigned integer

Read a 16-bit unsigned integer from input.

This function uses the interrupt service Int 21/AH=0Ah for reading a buffered string.
The use of a buffered string let the user review what they had typed before passing it to the

program for processing.
Up to six digits are read (as 65535 = 2^{16} - 1 has six digits).

Besides performing the standard conversion from *numeral* to *number* this function also detects invalid input and overflow (number too big to fit 16 bits).

Return values

The function return the number read in `AX`. The flags `ZF`, `CF`, `OF` tell if the operation completed successfully or not and why.

Error	AX	ZF	CF	OF
None	The 16-bit integer	Set	Not Set	Not Set
Invalid input	The partially converted number, up to the last valid digit encountered	Not Set	Set	Not Set
Overflow	7fffh	Not Set	Set	Set

The `ZF` can be used to quickly tell valid vs invalid inputs apart.

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
```

NASM porting

To port the code to NASM remove the `PTR` keyword from memory accesses (e.g. `mov cl, BYTE PTR [si]` becomes `mov cl, BYTE [si]`)

MS-DOS, TASM/MASM function to print a 16-bit number in binary, quaternary, octal, hex

Print a number in binary, quaternary, octal, hexadecimal and a general power of two

All the bases that are a power of two, like the binary (2^1), quaternary (2^2), octal (2^3), hexadecimal (2^4) bases, have an integral number of bits per digit[1].
Thus to retrieve each digit[2] of a numeral we simply break the number intro group of *n* bits starting from the LSb (the right).

For example for the quaternary base, we break a 16-bit number in groups of two bits. There are 8 of such groups.
Not all power of two bases have an integral number of groups that fits 16 bits; for example, the octal base has 5 groups of 3 bits that account for 3·5 = 15 bits out of 16, leaving a partial group of 1 bit[3].

The algorithm is simple, we isolate each group with a shift followed by an *AND* operation.
This procedure works for every size of the groups or, in other words, for any base power of two.

In order to show the digits in the right order the function start by isolating the most significant group (the leftmost), thereby it is important to know: a) how many bits *D* a group is and b) the bit position *S* where the leftmost group starts.
These values are precomputed and stored in carefully crafted constants.

Parameters

The parameters must be pushed on the stack.
Each one is 16-bit wide.
They are shown in order of push.

Parameter	Description
N	The number to convert
Base	The base to use expressed using the constants `BASE2`, `BASE4`, `BASE8` and `BASE16`
Print leading zeros	If *zero* no non-significant zeros are print, otherwise they are. The number 0 is printed as "0" though

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 to TASM users: If you put the constants defined with `EQU` after the code that uses them, enable *multi-pass* with the `/m` flag of *TASM* or you'll get *Forward reference needs override*.

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
```

Data

```
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
```

NASM porting

To port the code to NASM remove the PTR keyword from memory accesses (e.g. `mov si, WORD PTR [bp+08h]` becomes `mov si, WORD PTR [bp+08h]`)

Extending the function

The function can be easily extended to any base up to 2^{255}, though each base above 2^{16} will print the same numeral as the number is only 16 bits.

To add a base:

1. Define a new constant `BASEx` where *x* is 2^n.
   The lower byte, named *D*, is $D = n$.
   The upper byte, named *S*, is the position, in bits, of the higher group. It can be calculated as $S = n \cdot (\lceil 16/n \rceil - 1)$.
2. Add the necessary digits to the string `DIGITS`.

Example: adding base 32

We have $D = 5$ and $S = 15$, so we define `BASE32 EQU 0f05h`.
We then add sixteen more digits: `DIGITS db "0123456789abcdefghijklmnopqrstuv"`.

As it should be clear, the digits can be changed by editing the `DIGITS` string.

---

[1] If B is a base, then it has B digits per definition. The number of bits per digit is thus $log_2(B)$. For power of two bases this simplifies to $log_2(2^n) = n$ which is an integer by definition.

[2] In this context it is assumed implicitly that the base under consideration is a power of two base 2^n.

[3] For a base $B = 2^n$ to have an integral number of bit groups it must be that $n \mid 16$ (n divides 16). Since the only factor in 16 is 2, it must be that n is itself a power of two. So B has the form 2^{2^k} or equivalently $log_2(log_2(B))$ must be an integer.

MS-DOS, TASM/MASM, function to print a 16-bit number in decimal

Print a 16-bit unsigned number in decimal

The interrupt service Int 21/AH=02h is used to print the digits.
The standard conversion from *number* to *numeral* is performed with the `div` instruction, the dividend is initially the highest power of ten fitting 16 bits (10^4) and it is reduced to lower powers at each iteration.

Parameters

The parameters are shown in order of push.
Each one is 16 bits.

Parameter	Description
number	The 16-bit unsigned number to print in decimal
show leading zeros	If 0 no non-significant zeros are printed, else they are. The number 0 is always printed as "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
```

NASM porting

To port the code to NASM remove the `PTR` keyword from memory accesses (e.g. `mov ax, WORD PTR [bp+06h]` becomes `mov ax, WORD [bp+06h]`)

Read Converting decimal strings to integers online:
https://riptutorial.com/x86/topic/3273/converting-decimal-strings-to-integers

Chapter 6: Data Manipulation

Syntax

- ***.386***: Tells **MASM** to compile for a minimum x86 chip version of 386.
- ***.model***: Sets memory model to use, see .MODEL.
- ***.code***: Code segment, used for processes such as the main process.
- ***proc***: Declares process.
- ***ret***: used for exiting functions successfully, see Working With Return Values.
- ***endp***: Ends process declaration.
- ***public***: Makes process available to all segments of the program.
- ***end***: Ends program, or if used with a process, such as in "**end main**", makes the process the main method.
- ***call***: Calls process and pushes its opcode onto the stack, see Control Flow.
- ***ecx***: Counter register, see registers.
- ***ecx***: Counter register.
- ***mul***: Multiplies value by eax

Remarks

mov is used to transfer data between the registers.

Examples

Using MOV to manipulate values

Description:

`mov` *copies* values of bits from source argument to destination argument.

Common source/destination are registers, usually the fastest way to manipulate values with[in] CPU.

Another important group of source_of/destination_for values is computer memory.

Finally some immediate values may be part of the `mov` instruction encoding itself, saving time of separate memory access by reading the value together with instruction.

On x86 CPU in 32 and 64 bit mode there are rich possibilities to combine these, especially various memory addressing modes. Generally memory-to-memory copying is out limit (except specialized instructions like `MOVSB`), and such manipulation requires intermediate storage of values into register[s] first.

Step 1: Set up your project to use **MASM**, see Executing x86 assembly in Visual Studio 2015
Step 2: Type in this:

```
.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
```

Step 3: Compile and debug.

The program should return value `16`.

Read Data Manipulation online: https://riptutorial.com/x86/topic/8030/data-manipulation

Chapter 7: Multiprocessor management

Parameters

LAPIC register	Address (Relative to *APIC BASE*)
Local APIC ID Register	+20h
Spurious Interrupt Vector Register	+0f0h
Interrupt Command Register (ICR); bits 0-31	+300h
Interrupt Command Register (ICR); bits 32-63	+310h

Remarks

In order to access the LAPIC registers a segment must be able to reach the address range starting at *APIC Base* (in *IA32_APIC_BASE*).
This address is relocatable and can theoretically be set to point somewhere in the lower memory, thus making the range addressable in real mode.

The read/write cycles to the LAPIC range are **not** however propagated to the Bus Interface Unit, thereby masking any access to the addresses "behind" it.

It is assumed that the reader is familiar with the Unreal mode, since it will be used in some example.

It is also necessary to be proficient with:

- Handling the difference between *logical* and *physical* addresses[1]
- Real mode segmentation.
- Memory aliasing, id est the ability to use different *logical* addresses for the same *physical* address
- Absolute, relative, far, near calls and jumps.
- NASM assembler, particularly that the `ORG` directive is global. Splitting the code into multiple files **greatly** simplify the coding as it will be possible to give different section different *ORGs*.

Finally, we assume the CPU has a *Local Advanced Programmable Interrupt Controller* (*LAPIC*).
If ambiguous from the context, APIC always means LAPIC (e not IOAPIC, or xAPIC in general).

References:

- Chapter 8 and 10 of Intel manuals.

Bitfields

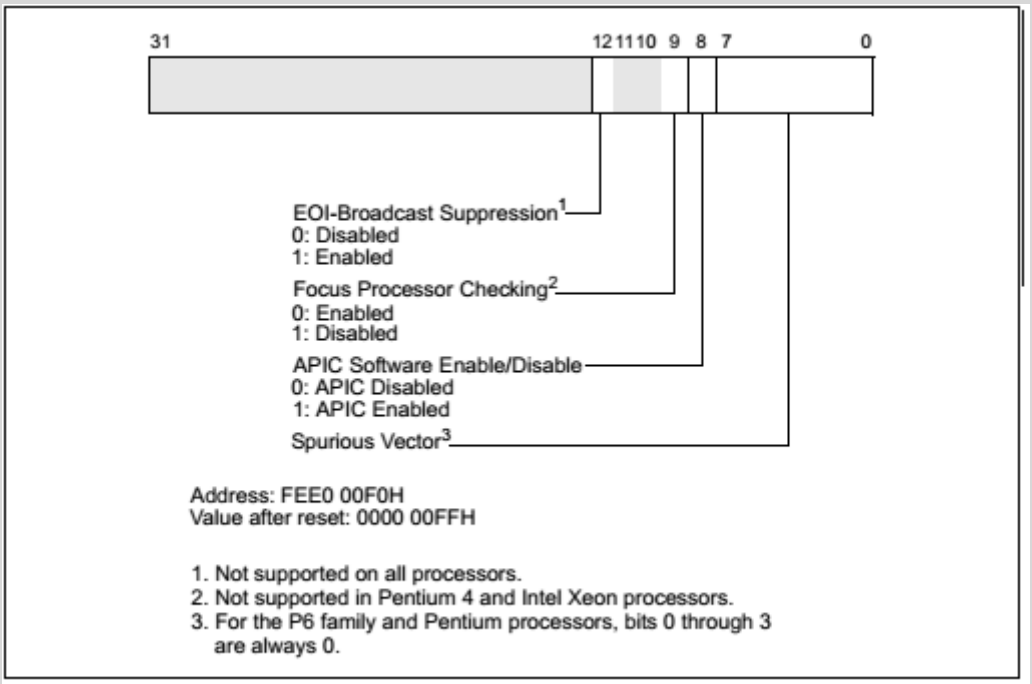


Figure 10-23. Spurious-Interrupt Vector Register (SVR)

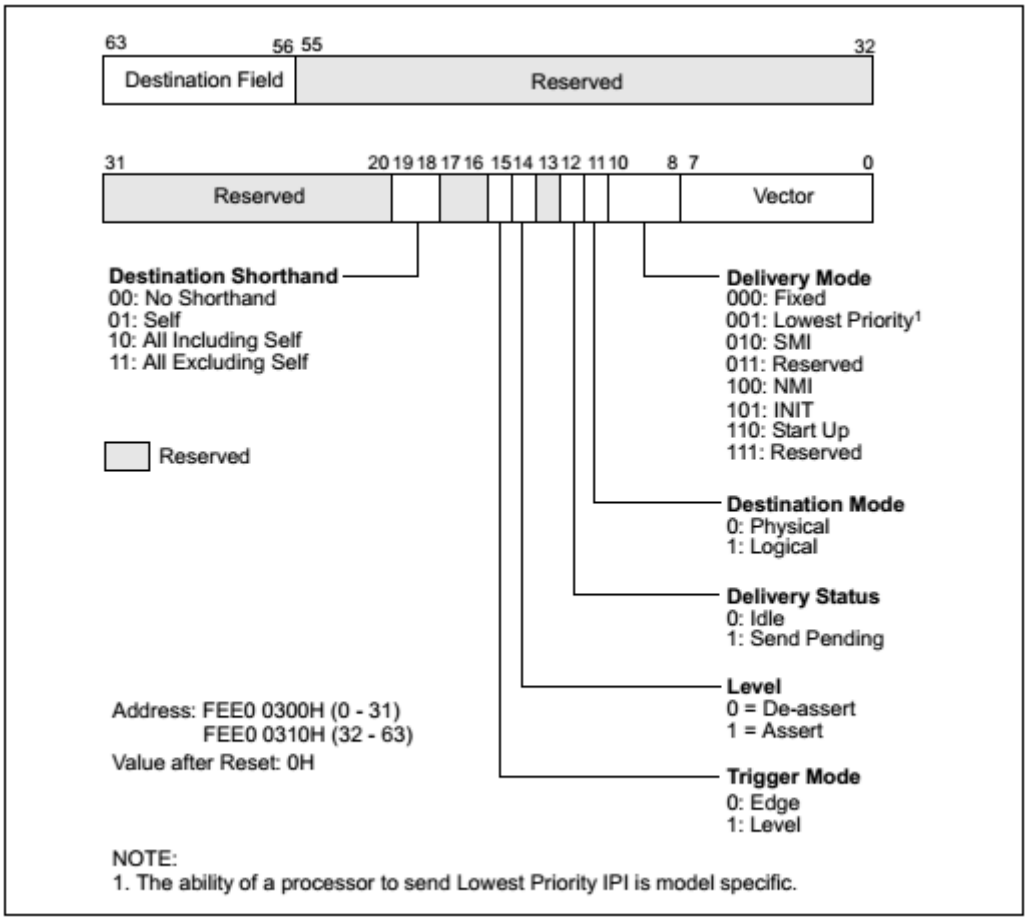


Figure 10-12. Interrupt Command Register (ICR)

Bitfields

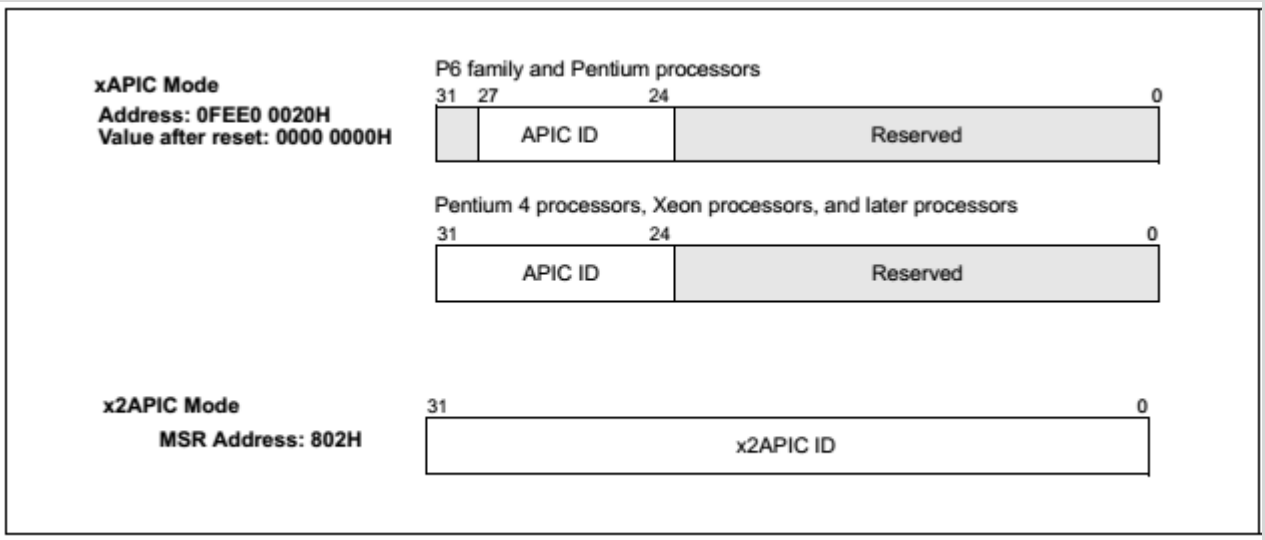


Figure 10-6. Local APIC ID Register

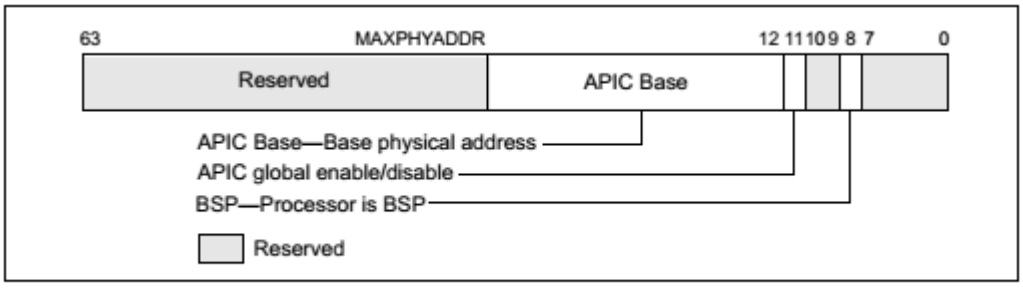


Figure 10-5. IA32_APIC_BASE MSR (APIC_BASE_MSR in P6 Family)

MSR name	Address
IA32_APIC_BASE	1bh

---

[1] If paging will be used, *virtual* addresses also come into play.

Examples

Wake up all the processors

This example will wake up every *Application Processor* (AP) and make them, along with the *Bootstrap Processor* (BSP), display their LAPIC ID.

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

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

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

 ;Clear screen
 mov ax, 03h
 int 10h

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

 ;Enable the APIC
 call enable_lapic

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


 ;Wakeup the other APs

 ;INIT
 call lapic_send_init
 mov cx, WAIT_10_ms
 call us_wait

 ;SIPI
 call lapic_send_sipi
 mov cx, WAIT_200_us
 call us_wait

 ;SIPI
 call lapic_send_sipi

 ;Jump to the payload
 jmp 0000h:8000h

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

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

  ret

  WAIT_10_ms     EQU 10000
  WAIT_200_us    EQU 200

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

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


enable_lapic:

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

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

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

 ret

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

lapic_send_sipi:
 mov eax, DWORD [APIC_BASE]

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

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

 ret

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

lapic_send_init:
 mov eax, DWORD [APIC_BASE]

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

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

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

  ret

 IA32_APIC_BASE_MSR    EQU    1bh

 APIC_REG_SIV        EQU    0f0h

 APIC_REG_ICR_LOW    EQU 300h
 APIC_REG_ICR_HIGH    EQU 310h

 APIC_REG_ID        EQU 20h

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

 APIC_BASE            dd     00h

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

unrealmode:
 lgdt [cs:GDT]

 cli

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

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

 and ax, 0fffeh
 mov cr0, eax

 sti

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

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

    dd 0000ffffh
    dd 00cf9200h

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

payload_start_abs:
```

```
; payload starts at segment:offset 0800h:0000h
section payload, vstart=0000h, align=1
 payload:

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

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

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

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

  mov cx, ax            ;Save this unique id

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

  ;Text buffer
  push 0b800h
  pop es

  ;Set unreal mode again
  call 7c0h:unrealmode

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

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

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

  cli
  hlt

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

```
  itoa8:
    mov bx, dx
    shr bx, 0fh
    mov al, BYTE [bx +  digits]
    mov ah, 09h
    stosw

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

    ret

  digits db "0123456789abcdef"
  counter dw 0

 payload_end:



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

There are two major steps to perform:

1. Waking the APs
This is achieved by inssuing a *INIT-SIPI-SIPI* (ISS) sequence to the all the APs.

The BSP that will send the ISS sequence using as destination the shorthand *All excluding self*, thereby targeting all the APs.

A SIPI (Startup Inter Processor Interrupt) is ignored by all the CPUs that are waked by the time they receive it, thus the second SIPI is ignored if the first one suffices to wake up the target processors. It is advised by Intel for compatibility reason.

A SIPI contains a *vector*, this is similar in meaning, **but absolutely different in practice**, to an interrupt vector (a.k.a. interrupt number).
The vector is an 8 bit number, of value *V* (represented as *vv* in base 16), that makes the CPU starts executing instructions at the *physical* address *0vv000h*.
We will call *0vv000h* the *Wake-up address* (WA).
The WA is forced at a 4KiB (or page) boundary.

We will use 08h as *V*, the WA is then *08000h*, 400h bytes after the bootloader.

This gives control to the APs.

2. Initializing and differentiating the APs
It is necessary to have an executable code at the WA. The bootloader is at *7c00h*, so we need to relocate some code at page boundary.

The first thing to remember when writing the payload is that any access to a shared resource must be protected or differentiated.
A common shared resource is the stack, if we initialize the stack naively, every APs will end up using the same stack!

The first step is then using different stack addresses, thus *differentiating* the stack.
We accomplish that by assigning an unique number, zero based, for each CPU. This number, we will call it *index*, is used for differentiating the stack and the line were the CPU will write its APIC ID.

The stack address for each CPU is *800h:(index * 1000h)* giving each AP 64KiB of stack.
The line number for each CPU is *index*, the pointer into the text buffer is thus 80 * 2 * *index*.

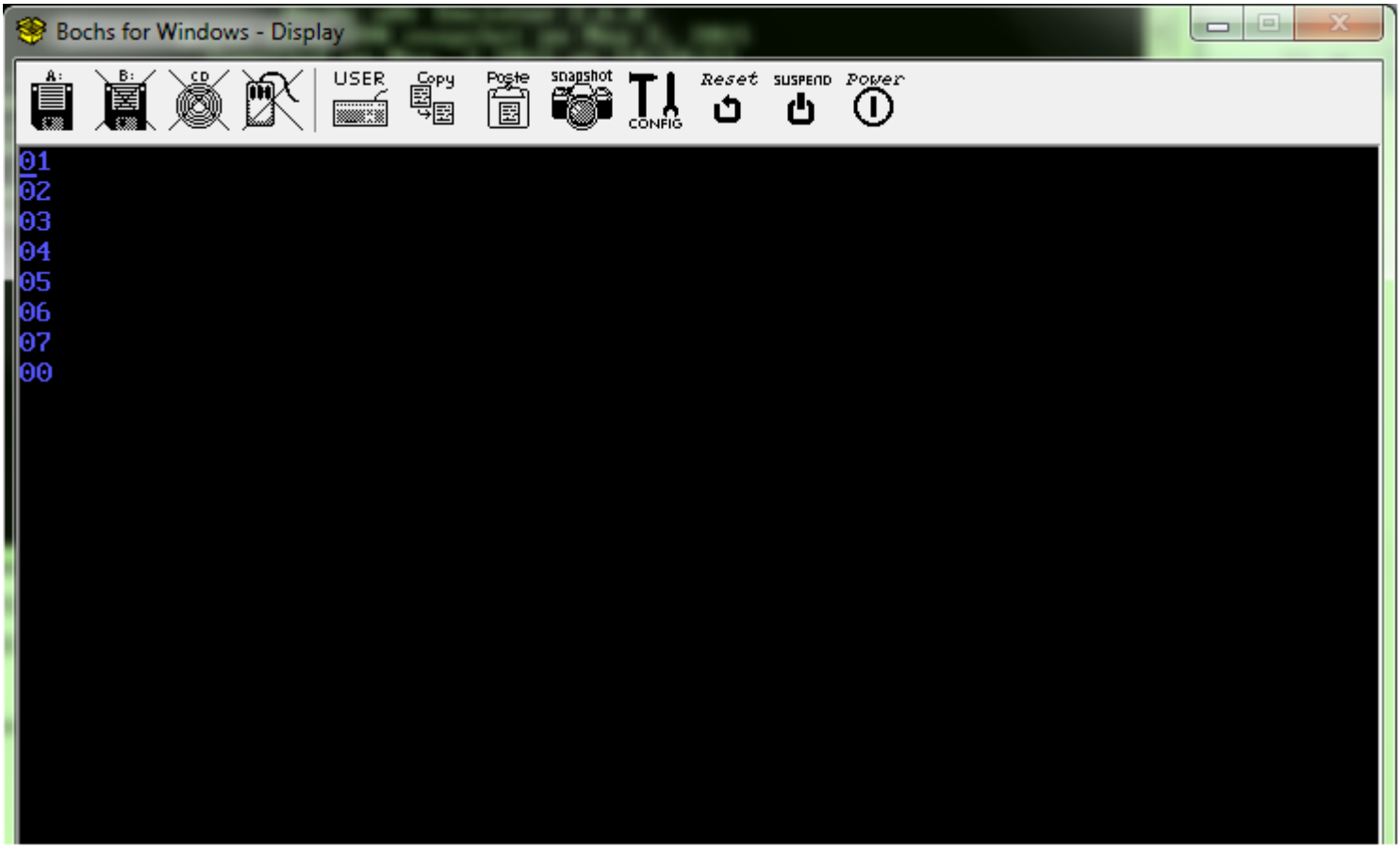
To generate the index a `lock cmpxchg` is used to atomically increment and return a WORD.

Final notes

- A write to port 80h is used to generate a delay of 1 µs.
- `unrealmode` is a far routine, so it can be called after the wake up too.
- The BSP also jump to the WA.

Screenshot

From Bochs with 8 processors



Read Multiprocessor management online: https://riptutorial.com/x86/topic/5809/multiprocessor-management

Chapter 8: Optimization

Introduction

The x86 family has been around for a long time, and as such there are many tricks and techniques that have been discovered and developed that are public knowledge - or maybe not so public. Most of these tricks take advantage of the fact that many instructions effectively do the same thing - but different versions are quicker, or save memory, or don't affect the Flags. Herein are a number of tricks that have been discovered. Each have their Pros and Cons, so should be listed.

Remarks

When in doubt, you can always refer to the pretty comprehensive Intel 64 and IA-32 Architectures Optimization Reference Manual, which is a great resource from the company behind the x86 architecture itsself.

Examples

Zeroing a register

The obvious way to zero a register is to `MOV` in a `0`—for example:

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

Notice that this is a 5-byte instruction.

If you are willing to clobber the flags (`MOV` never affects the flags), you can use the `XOR` instruction to bitwise-XOR the register with itself:

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

This instruction requires only 2 bytes and executes faster on all processors.

Moving Carry flag into a register

Background

If the Carry (`C`) flag holds a value that you want to put into a register, the naïve way is to do something like this:

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

```
NotZero:
```

Use 'sbb'

A more direct way, avoiding the jump, is to use "Subtract with Borrow":

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

If `C` is zero, then `al` will be zero. Otherwise it will be `0xFF` (`-1`). If you need it to be `0x01`, add:

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

Pros

- About the same size
- Two or one fewer instructions
- No expensive jump

Cons

- It's opaque to a reader unfamiliar with the technique
- It alters other Flags

Test a register for 0

Background

To find out if a register holds a zero, the naïve technique is to do this:

```
    cmp   eax, 0
```

But if you look at the opcode for this, you get this:

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

Use `test`

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

Examine the opcode you get:

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

Pros

- Only two bytes!

Cons

- Opaque to a reader unfamiliar with the technique

You can also have a look into the Q&A Question on this technique.

Linux system calls with less bloat

In 32-bit Linux, system calls are usually done by using the sysenter instruction (I say usually because older programs use the now deprecated `int 0x80`) however, this can take up quite alot of space in a program and so there are ways that one can cut corners in order to shorten and speed things up.
This is usually the layout of a system call on 32-bit Linux:

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

That's massive right! But there are a few tricks we can pull to avoid this mess.
The first is to set ebp to the value of esp decreased by the size of 3 32-bit registers, that is, 12 bytes. This is great so long as you are ok with overwriting ebp, edx and ecx with garbage (such as when you will be moving a value into those registers directly after anyway), we can do this using the LEA instruction so that we do not need to affect the value of ESP itself.

```
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
```

However, we're not done, if the system call is sys_exit we can get away with not pushing anything at all to the stack!

```
mov eax, 1
```

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

Multiply by 3 or 5

Background

To get the product of a register and a constant and store it in another register, the naïve way is to do this:

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

Use `lea`

Multiplications are expensive operations. It's faster to use a combination of shifts and adds. For the particular case of muliplying the contend of a 32 or 64 bit register that isn't `esp` or `rsp` by 3 or 5, you can use the lea instruction. This uses the address calculation circuit to calculate the product quickly.

```
    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
```

Many assemblers will also understand

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

For all possible multiplicands other them `ebp` or `rbp`, the resulting instruction lengh is the same as with using `imul`.

Pros

- Executes much faster

Cons

- If your multiplicand is `ebp` or `rbp` it takes one byte more them using `imul`
- More to type if your assembler dosn't support the shortcuts
- Opaque to a reader unfamiliar with the technique

Read Optimization online: https://riptutorial.com/x86/topic/3215/optimization

Chapter 9: Paging - Virtual Addressing and Memory

Examples

Introduction

History

The first computers

Early computers had a block of memory that the programmer put code and data into, and the CPU executed within this environment. Given that the computers then were very expensive, it was unfortunate that it would do one job, stop and wait for the next job to be loaded into it, and then process that one.

Multi-user, multi-processing

So computers quickly became more sophisticated and supported multiple users and/or programs simultaneously - but that's when problems started to arise with the simple "one block of memory" idea. If a computer was running two programs simultaneously, or running the same program for multiple users - whch of course would have required separate data for each user - then the management of that memory became critical.

Example

For example: if a program was written to work at memory address 1000, but another program was already loaded there, then the new program couldn't be loaded. One way of solving this would be to make programs work with "relative addressing" - it didn't matter where the program was loaded, it just did everything relative to the memory address that it was loaded in. But that required hardware support.

Sophistication

As computer hardware became more sophisticated, it was able to support larger blocks of memory, allowing for more simultaneous programs, and it became trickier to write programs that didn't interfere with what was already loaded. One stray memory reference could bring down not only the current program, but any other program in memory - including the Operating System itself!

Solutions

What was needed was a mechanism that allowed blocks of memory to have *dynamic* addresses. That way a program could be written to work with its blocks of memories at addresses that it recognised - and not be able to access other blocks for other programs (unless some cooperation allowed it to).

Segmentation

One mechanism that implemented this was Segmentation. That allowed blocks of memory to be defined of all different sizes, and the program would need to define which Segment it wanted to access all the time.

Problems

This technique was powerful - but its very flexibility was a problem. Since Segments essentially subdivided the available memory into different sized chunks, then the memory management for those Segments was an issue: allocation, deallocation, growing, shrinking, fragmentation - all required sophisticated routines and sometimes mass copying to implement.

Paging

A different technique divided all of the memory into equal-sized blocks, called "Pages", which made the allocation and deallocation routines very simple, and did away with growing, shrinking and fragmentation (except for internal fragmentation, which is merely a problem of wastage).

Virtual addressing

By dividing the memory into these blocks, they could be allocated to different programs as needed with whatever address the program needed it at. This "mapping" between the memory's physical address and the program's desired address is very powerful, and is the basis for every major processor's (Intel, ARM, MIPS, Power et. al.) memory management today.

Hardware and OS support

The hardware performed the remapping automatically and continually, but required memory to define the tables of what to do. Of course, the housekeeping associated with this remapping had to be controlled by something. The Operating System would have to dole out the memory as required, and manage the tables of data required by the hardware to support what the programs required.

Paging features

Once the hardware could do this remapping, what did it allow? The main driver was multiprocessing - the ability to run multiple programs, each with their "own" memory, protected from each other. But two other options included "sparse data", and "virtual memory".

Multiprocessing

Each program was given their own, virtual "Address Space" - a range of addresses that they could have physical memory mapped into, at whatever addresses were desired. As long as there was enough physical memory to go around (although see "Virtual Memory" below), numerous programs could be supported simultaneously.

What's more, those programs ***couldn't*** access memory that wasn't mapped into their virtual address space - protection between programs was automatic. If programs needed to communicate, they could ask the OS to arrange for a shared block of memory - a block of physical memory that was mapped into two different programs' address spaces simultaneously.

Sparse Data

Allowing a huge virtual address space (4 GB is typical, to correspond with the 32-bit registers these processors typically had) does not in and of itself waste memory, if large areas of that address space go unmapped. This allows for the creation of huge data structures where only certain parts are mapped at any one time. Imagine a 3-dimensional array of 1,000 bytes in each direction: that would normally take a billion bytes! But a program could reserve a block of its virtual address space to "hold" this data, but only map small sections as they were populated. This makes for efficient programming, while not wasting memory for data that isn't needed yet.

Virtual Memory

Above I used the term "Virtual Addressing" to describe the virtual-to-physical addressing performed by the hardware. This is often called "Virtual Memory" - but that term more correctly corresponds to the technique of using Virtual Addressing to support providing an illusion of more memory than is actually available.

It works like this:

- As programs are loaded and request more memory, the OS provides the memory from what it has available. As well as keeping track of what memory has been mapped, the OS also keeps track of when the memory is actually used - the hardware supports marking used pages.
- When the OS runs out of physical memory, it looks at all the memory that it has already handed out for whichever Page was used the least, or hadn't been used the longest. It saves that particular Page's contents to the hard disk, remembers where that was, marks it as "Not Present" to the hardware for the original owner, and then zeroes the Page and gives it to the new owner.
- If the original owner attempts to access that Page again, the hardware notifies the OS. The OS then allocates a new Page (perhaps having to do the previous step again!), loads up the

old Page's contents, then hands the new Page to the original program.

> The important point to notice is that since any Page can be mapped to any address, and each Page is the same size, then one Page is as good as any other - as long as the contents remain the same!

- If a program accesses an unmapped memory location, the hardware notifies the OS as before. This time, the OS notes that it wasn't a Page that had been saved away, so recognises it as a bug in the program, and terminates it!

> This is actually what happens when your app mysteriously vanishes on you - perhaps with a MessageBox from the OS. It's also what (often) happens to cause an infamous Blue Screen or Sad Mac - the buggy program was in fact an OS driver that accessed memory that it shouldn't!

Paging decisions

The hardware architects needed to make some big decisions about Paging, since the design would directly affect the design of the CPU! A very flexible system would have a high overhead, requiring large amounts of memory just to manage the Paging infrastructure itself.

How big should a Page be?

In hardware, the easiest implementation of Paging would be to take an Address and divide it into two parts. The upper part would be an indicator of which Page to access, while the lower part would be the index into the Page for the required byte:

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

It quickly became obvious though that small pages would require vast indexes for each program: even memory that wasn't mapped would need an entry in the table indicating this.

So instead a multi-tiered index is used. The address is broken into multiple parts (three are indicated in the below example), and the top part (commonly called a "Directory") indexes into the next part and so on until the final byte index into the final page is decoded:

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

That means that a Directory index can indicate "not mapped" for a vast chunk of the address space, without requiring numerous Page indexes.

How to optimise the usage of the Page Tables?

Every address access that the CPU will make will have to be mapped - the virtual-to-physical process must therefore be as efficient as possible. If the three-tier system described above were to be implemented, that would mean that every memory access would actually be three accesses: one into the Directory; one into the Page Table; and then finally the desired data itself. And if the CPU needed to perform housekeeping as well, such as indicating that this Page had now been accessed or written to, then that would require yet more accesses to update the fields.

Memory may be fast, but this would impose a triple-slowdown on all memory accesses during Paging! Luckily, most programs have a "locality of scope" - that is, if they access one location in memory, then future accesses will probably be nearby. And since Pages aren't too small, that mapping conversion would only need to be performed when a new Page was accessed: not for absolutely every access.

But even better would be to implement a cache of recently-accessed Pages, not just the most current one. The problem would be keeping up with what Pages had been accessed and what hadn't - the hardware would have to scan through the cache on every access to find the cached value. So the cache is implemented as a content-addressable cache: instead of being accessed by address, it is accessed by content - if the data requested is present, it is offered up, otherwise an empty location is flagged for filling in instead. The cache manages all of that.

This content-addressable cache is often called a Translation Lookaside Buffer (TLB), and is required to be managed by the OS as part of the Virtual Addressing subsystem. When the Directories or Page Tables are modified by the OS, it needs to notify the TLB to update its entries - or to simply invalidate them.

80386 Paging

High Level Design

The 80386 is a 32-bit processor, with a 32-bit addressable memory space. The designers of the Paging subsystem noted that a 4K page design mapped into those 32 bits in quite a neat way - 10 bits, 10 bits and 12 bits:

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

That meant that the Byte index was 12 bits wide, which would index into a 4K Page. The Directory and Page indexes were 10 bits, which would each map into a 1,024-entry table - and if those table entries were each 4 bytes, that would be 4K per table: also a Page!

So that's what they did:

- Each program would have its own Directory, a Page with 1,024 Page Entries that each defined where the next level Page Table was - if there was one.

- If there was, that Page Table would have 1,024 Page Entries that each defined where the last level Page was - if there was one.
- If there was, then that Page could have its Byte directly read out.

Page Entry

Both the top-level Directory and the next-level Page Table is comprised of 1,024 Page Entries. The most important part of these entries is the address of what it is indexing: a Page Table or an actual Page. Note that this address doesn't need the full 32 bits - since everything is a Page, only the top 20 bits are significant. Thus the other 12 bits in the Page Entry can be used for other things: whether the next level is even present; housekeeping as to whether the page has been accessed or written to; and even whether writes should even be allowed!

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

Note that if the `P` bit is 0, then the rest of the Entry is allowed to have anything that the OS wants to put in there - such as where the Page's contents mught be on the hard disk!

Page Directory Base Register (`PDBR`)

If each program has its own Directory, how does the hardware know where to start mapping? Since the CPU is only running one program at a time, it has a single Control Register to hold the address of the current program's Directory. This is the Page Directory Base Register (`CR3`). As the OS swaps between different programs, it updates the `PDBR` with the relevant Page Directory for the program.

Page Faults

Every time the CPU accesses memory, it has to map the indicated virtual address into the appropriate physical address. This is a three-step process:

1. Index the top 10 bits of the address into the Page indicated by the `PDBR` to get the address of the appropriate Page Table;
2. Index the next 10 bits of the address into the Page indicated by the Directory to get the address of the appropriate Page;
3. Index the last 12 bits of the address to get the data out of that Page.

Because both steps 1. and 2. above use Page Entries, each Entry could indicate a problem:

- The next level may be marked "Not Present";
- The next level may be marked as "Read Only" - and the operation is a Write;
- The next level may be marked as "Supervisor" - and it's the program accessing the memory, not the OS.

When such a problem is noted by the hardware, instead of completing the access it raises a Fault: Interrupt #14, the Page Fault. It also fills in some specific Control Registers with the information of why the Fault occurred: the address referenced; whether it was a Supervisor access; and whether it was a Write attempt.

The OS is expected to trap that Fault, decode the Control Registers, and decide what to do. If it's an invalid access, it can terminate the faulting program. If it's a Virtual Memory access though, the OS should allocate a new Page (which may need to vacate a Page that is already in use!), fill it with the required contents (either all zeroes, or the previous contents loaded back from disk), map the new Page into the appropriate Page Table, mark it as present, then resume the faulting instruction. This time the access will progress successfully, and the program will proceed with no knowledge that anything special happened (unless it takes a look at the clock!)

80486 Paging

The 80486 Paging Subsystem was very similar to the 80386 one. It was backward compatible, and the only new features were to allow for memory cache control on a Page-by-Page basis - the OS designers could mark specific pages as not to be cached, or to use different write-through or write-back caching techniques.

In all other respects, the "80386 Paging" example is applicable.

Pentium Paging

When the Pentium was being developed, memory sizes, and the programs that ran in them, were getting larger. The OS had to do more and more work to maintain the Paging Subsystem just in the sheer number of Page Indexes that needed to be updated when large programs or data sets were being used.

So the Pentium designers added a simple trick: they put an extra bit in the Entries of the Page Directory that indicated whether the next level was a Page Table (as before) - or went directly to a 4 MB Page! By having the concept of 4 MB Pages, the OS wouldn't have to create a Page Table and fill it with 1,024 Entries that were basically indexing addresses 4K higher than the previous one.

Address layout

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

```
 1         2 1                    0    number
```

Directory Entry layout

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

Of course, that had some ramifications:

- The 4 MB Page had to start on a 4 MB address boundary, just like the 4K Pages had to start on a 4K address boundary.
- All 4 MB had to belong to a single Program - or be shared by multiple ones.

This was perfect for use for large-memory peripherals, such as graphics adapters, that had large address space windows that needed to be mapped for the OS to use.

Physical Address Extension (PAE)

Introduction

As memory prices dropped, Intel-based PCs were able to have more and more RAM affordably, alleviating many users' problems with running many of the ever-larger applications that were being produced simultaneously. While virtual memory allowed memory to be virtually "created" - swapping existing "old" Page contents to the hard disk to allow "new" data to be stored - this slowed down the running of the programs as Page "thrashing" kept continually swapping data on and off the hard disk.

More RAM

What was needed was the ability to access more physical RAM - but it was already a 32-bit address bus, so any increase would require larger address registers. Or would it? When developing the Pentium Pro (and even the Pentium M), as a stop-gap until 64-bit processors could be produced, to add more Physical Address bits (allowing more Physical memory) *without* changing the number of register bits. This could be achieved since Virtual Addresses were mapped to Physical Addresses anyway - all that needed to change was the mapping system.

Design

The existing system could access a maximum of 32 bits of Physical Addresses. Increasing this required a complete change of the Page Entry structure, from 32 to 64 bits. It was decided to keep the minimum granularity at 4K Pages, so the 64-bit Entry would have 52 bits of Address and 12 bits of Control (like the previous Entry had 20 bits of Address and 12 bits of Control).

Having a 64-bit Entry, but a Page size of (still) 4K, meant that there would only be 512 Entries per Page Table or Directory, instead of the previous 1,024. That meant that the 32-bit Virtual Address would be divided differently than before:

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

Chopping one bit from both the Directory Index and Page Index gave two bits for a third tier of mapping: they called this the Page Directory Pointer Table (PDPT), a table of exactly four 64-bit Entries that addressed four Directories instead of the previous one. The PDBR (`CR3`) now pointed to the PDPT instead - which, since `CR3` was only 32 bits, needed to be stored in the first 4 GB of RAM for accessibility. Note that since the low bits of `CR3` are used for Control, the PDPT has to start on a 32-byte boundary.

Page Size Extension (PSE)

And, since the previous 4MB Pages were such a good idea, they wanted to be able to support large Pages again. This time though, removing the last layer of the tier system didn't produce 10+12 bit 4MB Pages, but 9+12 bit 2MB Pages instead.

PSE-32 (and PSE-40)

Since the Physical Address Extension (PAE) mode that was introduced in the Pentium Pro (and Pentum M) was such a change to the Operating System memory management subsystem, when Intel designed the Pentium II they decided to enhance the "normal" Page mode to support the new Physical Address bits of the processor within the previously-defined 32-bit Entries.

They realised that when a 4MB Page was used, the Directory Entry looked like this:

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

The Dir Index and Control areas of the Entry were the same, but the block of unused bits between them - which would be used by the Page Index if it existed - were wasted. So they decided to use

that area *to define the upper Physical Address bits above 31*!

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

This allowed RAM above 4 GB to be accessible to OSes that didn't adopt the PAE mode - with a little extra logic, they could provide large amounts of extra RAM to the system, albeit no more than the normal 4GB to each program. At first only 4 bits were added, allowing for 36-bit Physical Addressing, so this mode was called Page Size Extension 36 (PSE-36). It didn't actually change the Page size, only the Addressing however.

The limitation of this though was that only 4MB Pages above 4GB were definable - 4K Pages weren't allowed. Adoption of this mode wasn't wide - it was reportedly slower than using PAE, and Linux didn't end up ever using it.

Nevertheless, in later processors that had even more Physical Address bits, both AMD and Intel widened the PSE area to 8 bits, which some people dubbed "PSE-40"

Read Paging - Virtual Addressing and Memory online:
https://riptutorial.com/x86/topic/3218/paging---virtual-addressing-and-memory

Chapter 10: Real vs Protected modes

Examples

Real Mode

When Intel designed the original x86, the 8086 (and 8088 derivative), they included Segmentation to allow the 16-bit processor to access more than 16 bits worth of address. They did this by making the 16-bit addresses be relative to a given 16-bit Segment Register, of which they defined four: Code Segment (`CS`), Data Segment (`DS`), Extra Segment (`ES`) and Stack Segment (`SS`).

Most instructions implied which Segment Register to use: instructions were fecthed from the Code Segment, `PUSH` and `POP` implied the Stack Segment, and simple data references implied the Data Segment - although this could be overridden to access memory in any of the other Segments.

The implementation was simple: for every memory access, the CPU would take the implied (or explicit) Segment Register, shift it four places to the left, then add in the indicated address:

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

This allowed for various techniques:

- Allowing Code, Data and Stack to all be mutually accessable (`CS`, `DS` and `SS` all had the same value);
- Keeping Code, Data and Stack completely separate from each other (`CS`, `DS` and `SS` all 4K (or more) separate from each other - remember it gets multiplied by 16, so that's 64K).

It also allowed bizarre overlaps and all sorts of weird things!

When the 80286 was invented, it supported this legacy mode (now called "Real Mode"), but added a new mode called "Protected Mode" (q.v.).

The important things to notice is that in Real Mode:

- Any memory address was accessible, simply by putting the correct value inside a Segment Register and accessing the 16-bit address;
- The extent of "protection" was to allow the programmer to separate different areas of memory for different purposes, and make it *harder* to accidentally write to the wrong data - while still making it possible to do so.

In other words... not very protected at all!

Protected Mode

Introduction

When the 80286 was invented, it supported the legacy 8086 Segmentation (now called "Real Mode"), and added a new mode called "Protected Mode". This mode has been in every x86 processor since, albeit enhanced with various improvements such as 32- and 64-bit addressing.

Design

In Protected Mode, the simple "Add address to Shifted Segment Register value" was done away with completely. They kept the Segment Registers, but instead of using them to calculate an address, they used them to index into a table (actually, one of two...) which defined the Segment to be accessed. This definition not only described where in memory the Segment was (using Base and Limit), but also what *type* of Segment it was (Code, Data, Stack or even System) and what kinds of programs could access it (OS Kernel, normal program, Device Driver, etc.).

Segment Register

Each 16-bit Segment Register took on the following form:

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

The Global/Local bit defined whether the access was into a Global Table of descriptors (called unsurprisingly the Global Descriptor Table, or GDT), or a Local Descriptor Table (LDT). The idea for the LDT was that every program could have its own Descriptor Table - the OS woud define a Global set of Segments, and each program would have its own set of Local Code, Data and Stack Segments. The OS would manage the memory between the different Descriptor Tables.

Descriptor Table

Each Descriptor Table (Global or Local) was a 64K array of 8,192 Descriptors: each an 8-byte record that defined multiple aspects of the Segment that it was describing. The Segment

Registers' Descriptor Index fields allowed for 8,192 descriptors: no coincidence!

Descriptor

A Descriptor held the following information - note that the format of the Descriptor changed as new processors were released, but the same sort of information was kept in each:

- **Base**
  This defined the start address of the memory segment.
- **Limit**
  This defined the size of the memory segment - sort of. They had to make a decision: would a size of `0x0000` mean a size of `0`, so not accessible? Or maximum size?
  Instead they chose a third option: the Limit field was the last addressible location within the Segment. That meant that a one-bye Segment could be defined; or a maximum-sized one for the address size.
- **Type**
  There were multiple types of Segments: the traditional Code, Data and Stack (see below), but other System Segments were defined as well:
  - Local Descriptor Table Segments defined how many Local Descriptors could be accessed;
  - Task State Segments could be used for hardware-managed context switching;
  - Controlled "Call Gates" that could allow programs to call into the Operating System - but only through carefully managed entry points.
- **Attributes**
  Certain attributes of the Segment were also maintained, where relevant:
  - Read-Only vs Read-Write;
  - Whether the Segment was currently Present or not - allowing for on-demand memory management;
  - What level of code (OS vs Driver vs program) could access this Segment.

True protection at last!

If the OS kept the Descriptor Tables in Segments that couldn't be accessed by mere programs, then it could tightly manage which Segments were defined, and what memory was assigned and accessible to each. A program could manufacture whatever Segment Register value it liked - but if it had the *audaciousness* to actually *load* it into a *Segment Register*!... the CPU hardware would recognise that the proposed Descriptor value broke any one of a large number of rules, and instead of completing the request, it would raise an Exception to the Operating System to allow it to handle the errant program.

> This Exception was usually #13, the General Protection Exception - made world famous by Microsoft Windows... (Anyone think an Intel engineer was superstitious?)

Errors

The sorts of errors that could happen included:

- If the proposed Descriptor Index was larger than the size of the table;
- If the proposed Descriptor was a System Descriptor rather than Code, Data or Stack;
- If the proposed Descriptor was more privileged than the requesting program;
- If the proposed Descriptor was marked as Not Readable (such as a Code Segment), but it was attempted to be Read rather than Executed;
- If the proposed Descriptor was marked Not Present.

    Note that the last may not be a fatal problem for the program: the OS could note the flag, reinstate the Segment, mark it as now Present then allow the faulting instruction to proceed successfully.

Or, perhaps the Descriptor was successfully loaded into a Segment Register, but then a future access with it broke one of a number of rules:

- The Segment Register was loaded with the `0x0000` Descriptor Index for the GDT. This was reserved by the hardware as `NULL`;
- If the loaded Descriptor was marked Read-Only, but a Write was attempted to it.
- If any part of the access (1, 2, 4 or more bytes) was outside the Limit of the Segment.

Switching into Protected Mode

Switching into Protected Mode is easy: you just need to set a single bit in a Control Register. But *staying* in Protected Mode, without the CPU throwing up its hands and resetting itself due to not knowing what to do next, takes a lot of preparation.

In short, the steps required are as follows:

- An area of memory for the Global Descriptor Table needs to be set up to define a minimum of three Descriptors:

    1. The zeroeth, `NULL` Descriptor;
    2. Another Descriptor for a Code Segment;
    3. Another Descriptor for a Data Segment.

        This can be used for both Data and Stack.

- The Global Descriptor Table Register (`GDTR`) needs to be initialised to point to this defined area of memory;

```
GDT_Ptr    dw      SIZE GDT
```

```
            dd      OFFSET GDT

            ...

            lgdt    [GDT_Ptr]
```

- The `PM` bit in `CR0` needs to be set:

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

- The Segment Registers need to be loaded from the GDT to remove the current Real Mode values:

```
     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!
```

Note that this is the ***bare*** minimum, just to get the CPU into Protected Mode. To actually get the whole system ready may require many more steps. For example:

- The upper memory areas may have to be enabled - turning off the `A20` gate;
- The Interrupts should definitely be disabled - but perhaps the various Fault Handlers could be set up before entering Protected Mode, to allow for errors early on in the processing.

  The original author of this section wrote an entire tutorial on entering Protected Mode and working with it.

Unreal mode

The *unreal mode* exploits two facts on how both Intel and AMD processors load and save the information to describe a segment.

1. The processor caches the descriptor information fetched during a *move* in a selector register in protected mode.
   These information are stored in an architectural invisible part of the selector register themselves.

2. In real mode the selector registers are called segment registers but, other than that, they designate the same set of registers and as such they also have an invisible part. These parts are filled with fixed values, but for the base which is derived from the value just loaded.

In such view, real mode is just a special case of protected mode: where the information of a segment, suchlike the base and limit, are fetched without a GDT/LDT but still read from the segment register hidden part.

By switching in protected mode and crafting a GDT is possible to create a segment with the desired attributes, for example a base of 0 and a limit of 4GiB.
Through a successive loading of a selector register such attributes are cached, it is then possible to switch back in real mode and have a segment register through which access the whole 32 bit address space.

```
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
```

Considerations

- As soon as a segment register is reloaded, even with the same value, the processor reload the hidden attributes according to the current mode. This is why the code above use `fs` and `gs` to hold the "extended" segments: such registers are less likely to be used/saved/restored by the various 16 bit services.
- The `lgdt` instruction doesn't load a far pointer to the GDT, instead it loads a 24 bit (can be overridden to 32 bit) *linear address*. This is not a *near address*, it is the *physical address* (since paging must be disabled). That's the reason of `GDT+7c00h`.
- The program above is a bootloader (for MBR, it has no BPB) that set `cs`/`ds`/`ss` tp *7c00h* and start the location counter from 0. So a byte at offset *X* in the file is at offset *X* in the segment *7c00h* and at the linear address *7c00h* + *X*.
- Interrupts must be disabled as an IDT is not set for the short round trip in protected mode.
- The code use an hack to save 6 bytes of code. The structure loaded by `lgdt` is saved in the... GDT itself, in the null descriptor (the first descriptor).

For a description of the GDT descriptors see Chapter 3.4.3 of Intel Manual Volume 3A.

Read Real vs Protected modes online: https://riptutorial.com/x86/topic/3679/real-vs-protected-modes

Chapter 11: Register Fundamentals

Examples

16-bit Registers

When Intel defined the original 8086, it was a 16-bit processor with a 20-bit address bus (see below). They defined 8 general-purpose 16-bit registers - but gave them specific roles for certain instructions:

- `AX` The Accumulator register.
  Many opcodes either assumed this register, or were faster if it was specified.
- `DX` The Data register.
  This was sometimes combined as the high 16 bits of a 32-bit value with `AX` - for example, as the result of a multiply.
- `CX` The Count register.
  This was used in a number of loop-oriented instructions as the implicit counter for those loops - for example `LOOPNE` (loop if not equal) and `REP` (repeated move/compare)
- `BX` The Base register.
  This could be used to index the base of a structure in memory - none of the above registers could be used to directly index into memory.
- `SI` The Source Index register.
  This was the implicit source index into memory for certain move and compare operations.
- `DI` The Destination Index register.
  This was the implicit destination index into memory for certain move and compare operations.
- `SP` The Stack Pointer register.
  This is the least general-purpose register in the set! It pointed to the current position in the stack, which was used explicitly for `PUSH` and `POP` operations, implicitly for `CALL` and `RET` with subroutines, and VERY implicitly during interrupts. As such, using it for anything else was hazardous to your program!
- `BP` The Base Pointer register.
  When subroutines call other subroutines, the stack holds multiple "stack frames". `BP` could be used to hold the current stack frame, and then when a new subroutine was called it coould be saved on the stack, the new stack frame created and used, and on return from the inner subroutine the old stack frame value could be restored.

Notes:

1. The first three registers cannot be used for indexing into memory.

2. `BX`, `SI` and `DI` by default index into the current Data Segment (see below).

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

3. `DI`, when used in memory-to-memory operations such as `MOVS` and `CMPS`, solely uses the Extra Segment (see below). This cannot be overridden.

4. `SP` and `BP` use the Stack Segment (see below) by default.

32-bit registers

When Intel produced the 80386, they upgraded from a 16-bit processor to a 32-bit one. 32-bit processing means two things: both the data being manipulated was 32-bit, and the memory addresses being accessed were 32-bit. To do this, but still remain compatible with their earlier processors, they introduced whole new modes for the processor. It was either in 16-bit mode or 32-bit mode - but you could override this mode on an instruction-by-instruction basis for either data, addressing, or both!

First of all, they had to define 32-bit registers. They did this by simply extending the existing eight from 16 bits to 32 bits and giving them "extended" names with an `E` prefix: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP`, and `ESP`. The lower 16 bits of these registers were the same as before, but the upper halves of the registers were available for 32-bit operations such as `ADD` and `CMP`. The upper halves were not separately accessible as they'd done with the 8-bit registers.

The processor had to have separate 16-bit and 32-bit modes because Intel used the same opcodes for many of the operations: `CMP AX,DX` in 16-bit mode and `CMP EAX,EDX` in 32-bit mode had exactly the same opcodes! This meant that the same code could NOT be run in either mode:

> The opcode for "Move immediate into `AX`" is `0xB8`, followed by two bytes of the immediate value: `0xB8 0x12 0x34`
>
> The opcode for "Move immediate into `EAX`" is `0xB8`, followed by ***four*** bytes of the immediate value: `0xB8 0x12 0x34 0x56 0x78`

So the assember has to know what mode the processor is in when executing the code, so that it knows to emit the correct number of bytes.

8-bit Registers

The first four 16-bit registers could have their upper- and lower-half bytes accessed directly as their own registers:

- `AH` and `AL` are the High and Low halves of the `AX` register.
- `BH` and `BL` are the High and Low halves of the `BX` register.
- `CH` and `CL` are the High and Low halves of the `CX` register.
- `DH` and `DL` are the High and Low halves of the `DX` register.

Note that this means that altering `AH` or `AL` will immediately alter `AX` as well! Also note that any operation on an 8-bit register couldn't affect its "partner" - incrementing `AL` such that it overflowed from `0xFF` to `0x00` wouldn't alter `AH`.

64-bit registers also have 8-bit versions representing their lower bytes:

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

The same applies to registers `R8` through `R15`: their respective lower byte parts are named `R8B` – `R15B`.

Segment Registers

Segmentation

When Intel was designing the original 8086, there were already a number of 8-bit processors that had 16-bit capabilities - but they wanted to produce a true 16-bit processor. They also wanted to produce something better and more capable than what was already out there, so they wanted to be able to access more than the maximum of 65,536 bytes of memory implied by 16-bit addressing registers.

Original Segment Registers

So they implemented the idea of "Segments" - a 64 kilobyte block of memory indexed by the 16-bit address registers - that could be re-based to address different areas of the total memory. To hold these segment bases, they included Segment Registers:

- `CS` The Code Segment register.
  This holds the segment of the code that is currently being executed, indexed by the implicit `IP` (Instruction Pointer) register.
- `DS` The Data Segment register.
  This holds the default segment for data being manipulated by the program.
- `ES` The Extra Segment register.
  This holds a second data segment, for simultaneous data operations across the total memory.
- `SS` The Stack Segment register.
  This holds the segment of memory that holds the current stack.

Segment Size?

The segment registers could be any size, but making them 16 bits wide made it easy to interoperate with the other registers. The next question was: should the segments overlap, and if so, how much? The answer to that question would dictate the total memory size that could be accessed.

If there was no overlap at all, then the address space would be 32 bits - 4 gigabytes - a totally unheard-of size at the time! A more "natural" overlap of 8 bits would produce a 24-bit address

space, or 16 megabytes. In the end Intel decided to save four more address pins on the processor by making the address space 1 megabyte with a 12-bit overlap - they considered this sufficiently large for the time!

More Segment Registers!

When Intel was designing the 80386, they recognised that the existing suite of 4 Segment Registers wasn't enough for the complexity of programs that they wanted it to be able to support. So they added two more:

- `FS` The Far Segment register
- `GS` The Global Segment register

These new Segment registers didn't have any processor-enforced uses: they were merely available for whatever the programmer wanted.

> Some say that the names were chosen to simply continue the `C`, `D`, `E` theme of the existing set...

64-bit registers

AMD is a processor manufacturer that had licensed the design of the 80386 from Intel to produce compatible - but competing - versions. They made internal changes to the design to improve throughput or other enhancements to the design, while still being able to execute the same programs.

To one-up Intel, they came up with 64-bit extensions to the Intel 32-bit design and produced the first 64-bit chip that could still run 32-bit x86 code. Intel ended up following AMD's design in their versions of the 64-bit architecture.

The 64-bit design made a number of changes to the register set, while still being backward compatible:

- The existing general-purpose registers were extended to 64 bits, and named with an `R` prefix: `RAX`, `RBX`, `RCX`, `RDX`, `RSI`, `RDI`, `RBP`, and `RSP`.

  > Again, the bottom halves of these registers were the same `E`-prefix registers as before, and the top halves couldn't be independently accessed.

- 8 more 64-bit registers were added, and not named but merely numbered: `R8`, `R9`, `R10`, `R11`, `R12`, `R13`, `R14`, and `R15`.
  - The 32-bit low half of these registers are `R8D` through `R15D` (D for DWORD as usual).
  - The lowest 16 bits of these registers could be accessed by suffixing a `W` to the register name: `R8W` through `R15W`.
- The lowest 8 bits of *all 16* registers could now be accessed:
  - The traditional `AL`, `BL`, `CL`, and `DL`;
  - The low bytes of the (traditionally) pointer registers: `SIL`, `DIL`, `BPL`, and `SPL`;

- And the low bytes of the 8 new registers: `R8B` through `R15B`.
- However, `AH`, `BH`, `CH`, and `DH` are inaccessible in instructions that use a REX prefix (for 64bit operand size, or to access R8-R15, or to access `SIL`, `DIL`, `BPL`, or `SPL`). With a REX prefix, the machine-code bit-pattern that used to mean `AH` instead means `SPL`, and so on. See Table 3-1 of Intel's instruction reference manual (volume 2).

Writing to a 32-bit register always zeros the upper 32 bits of the full-width register, unlike writing to an 8 or 16-bit register (which merges with the old value, which is an extra dependency for out-of-order execution).

Flags register

When the x86 Arithmetic Logic Unit (ALU) performs operations like `NOT` and `ADD`, it flags the results of these operations ("became zero", "overflowed", "became negative") in a special 16-bit `FLAGS` register. 32-bit processors upgraded this to 32 bits and called it `EFLAGS`, while 64-bit processors upgraded this to 64 bits and called it `RFLAGS`.

Condition Codes

But no matter the name, the register is not directly accessible (except for a couple of instructions - see below). Instead, individual flags are referenced in certain instructions, such as conditional Jump or conditional Set, known as `Jcc` and `SETcc` where `cc` means "condition code" and references the following table:

Condition Code	Name	Definition
`E`, `Z`	Equal, Zero	`ZF` == 1
`NE`, `NZ`	Not Equal, Not Zero	`ZF` == 0
`O`	Overflow	`OF` == 1
`NO`	No Overflow	`OF` == 0
`S`	Signed	`SF` == 1
`NS`	Not Signed	`SF` == 0
`P`	Parity	`PF` == 1
`NP`	No Parity	`PF` == 0
---	---	---
`C`, `B`, `NAE`	Carry, Below, Not Above or Equal	`CF` == 1
`NC`, `NB`, `AE`	No Carry, Not Below, Above or Equal	`CF` == 0

Condition Code	Name	Definition
A, NBE	Above, Not Below or Equal	CF==0 and ZF==0
NA, BE	Not Above, Below or Equal	CF==1 or ZF==1
---	---	---
GE, NL	Greater or Equal, Not Less	SF==OF
NGE, L	Not Greater or Equal, Less	SF!=OF
G, NLE	Greater, Not Less or Equal	ZF==0 and SF==OF
NG, LE	Not Greater, Less or Equal	ZF==1 or SF!=OF

In 16 bits, subtracting `1` from `0` is either `65,535` or `-1` depending on whether unsigned or signed arithmetic is used - but the destination holds `0xFFFF` either way. It's only by interpreting the condition codes that the meaning is clear. It's even more telling if `1` is subtracted from `0x8000`: in unsigned arithmetic, that merely changes `32,768` into `32,767`; while in signed arithmetic it changes `-32,768` into `32,767` - a much more noteworthy overflow!

The condition codes are grouped into three blocks in the table: sign-irrelevant, unsigned, and signed. The naming inside the latter two blocks uses "Above" and "Below" for unsigned, and "Greater" or "Less" for signed. So `JB` would be "Jump if Below" (unsigned), while `JL` would be "Jump if Less" (signed).

Accessing `FLAGS` directly

The above condition codes are useful for interpreting predefined concepts, but the actual flag bits are also available directly with the following two instructions:

- `LAHF` Load `AH` register with Flags
- `SAHF` Store `AH` register into Flags

Only certain flags are copied across with these instructions. The whole `FLAGS` / `EFLAGS` / `RFLAGS` register can be saved or restored on the stack:

- `PUSHF` / `POPF` Push/pop 16-bit `FLAGS` onto/from the stack
- `PUSHFD` / `POPFD` Push/pop 32-bit `EFLAGS` onto/from the stack
- `PUSHFQ` / `POPFQ` Push/pop 64-bit `RFLAGS` onto/from the stack

Note that interrupts save and restore the current `[R/E]FLAGS` register automatically.

Other Flags

As well as the ALU flags described above, the `FLAGS` register defines other system-state flags:

- IF The Interrupt Flag.
  This is set with the STI instruction to globally enable interrupts, and cleared with the CLI instruction to globally disable interrupts.
- DF The Direction Flag.
  Memory-to-memory operations such as CMPS and MOVS (to compare and move between memory locations) automatically increment or decrement the index registers as part of the instruction. The DF flag dictates which one happens: if cleared with the CLD instruction, they're incremented; if set with the STD instruction, they're decremented.
- TF The Trap Flag. This is a debug flag. Setting it will put the processor into "single-step" mode: after each instruction is executed it will call the "Single Step Interrupt Handler", which is expected to be handled by a debugger. There are no instructions to set or clear this flag: you need to manipulate the bit while it is in memory.

80286 Flags

To support the new multitasking facilities in the 80286, Intel added extra flags to the FLAGS register:

- IOPL The I/O Privilege Level.
  To protect multitasking code, some tasks needed privileges to access I/O ports, while others had to be stopped from accessing them. Intel introduced a four-level Privilege scale, with 00_2 being most privileged and 11_2 being least. If IOPL was less than the current Privilege Level, any attempt to access I/O ports, or enable or disable interrups, would cause a General Protection Fault instead.
- NT Nested Task flag.
  This flag was set if one Task CALLed another Task, which caused a context switch. The set flag told the processor to do a context switch back when the RET was executed.

80386 Flags

The '386 needed extra flags to support extra features designed into the processor.

- RF The Resume Flag.
  The `386 added Debug registers, which could invoke the debugger on various hardware accesses like reading, writing or executing a certain memry location. However, when the debug handler returned to execute the instruction *the access would immediately re-invoke the debug handler!* Or at least it would if it wasn't for the Resume Flag, which is automatically set on entry into the debug handler, and automatically cleared after every instruction. If the Resume Flag is set, the Debug handler is not invoked.
- VM The Virtual 8086 Flag.
  To support older 16-bit code as well as newer 32-bit code, the 80386 could run 16-bit Tasks in a "Virtual 8086" mode, with the aid of a Virtual 8086 executive. The VM flag indicated that this Task was a Virtual 8086 Task.

80486 Flags

As the Intel architecture improved, it got faster through such technology as caches and super-

scalar execution. That had to optimise access to the system by making assumptions. To control those assumptions, more flags were needed:

- `AC` Alignment Check flag The x86 architecture could always access multi-byte memory values on any byte boundary, unlike some architectures which required them to be size-aligned (4-byte values needed to be on 4-byte boundaries). However, it was less efficient to do so, since multiple memory accesses were needed to access unaligned data. If the `AC` flag was set, then an unaligned access would raise an exception rather than execute the code. That way, code could be improved during development with `AC` set, but turned off for production code.

Pentium Flags

The Pentium added more support for virtualising, plus support for the `CPUID` instruction:

- `VIF` The Virtual Interrupt Flag.
  This is a virtual copy of this Task's `IF` - whether or not this Task wants to disable interrupts, without actually affecting Global Interrupts.
- `VIP` The Virtual Interrupt Pending Flag.
  This indicates that an interrupt was virtually blocked by `VIF`, so when the Task does an `STI` a virtual interrupt can be raised for it.
- `ID` The `CPUID`-allowed Flag.
  Whether or not to allow this Task to execute the `CPUID` instruction. A Virtual monitor could disallow it, and "lie" to the requesting Task if it executes the instruction.

Read Register Fundamentals online: https://riptutorial.com/x86/topic/2122/register-fundamentals

Chapter 12: System Call Mechanisms

Examples

BIOS calls

How to interact with the BIOS

The Basic Input/Output System, or BIOS, is what controls the computer before any operating system runs. To access services provided by the BIOS, assembly code uses *interrupts*. An interrupt takes the form of

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

The interrupt number must be between 0 and 255 (0x00 - 0xFF), inclusive.

Most BIOS calls use the `AH` register as a "function select" parameter, and use the `AL` register as a data parameter. The function selected by `AH` depends on the interrupt called. Some BIOS calls require a single 16-bit parameter in `AX`, or do not accept parameters at all, and are simply called by the interrupt. Some have even more parameters, that are passed in other registers.

The registers used for BIOS calls are fixed and cannot be interchanged with other registers.

Using BIOS calls with function select

The general syntax for a BIOS interrupt using a function select parameter is:

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

Examples

How to write a character to the display:

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

How to read a character from the keyboard (blocking):

```
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
```

How to read one or more sectors from an external drive (using CHS addressing):

```
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
```

How to read the system RTC (Real Time Clock):

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

How to read the system time from the 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
```

How to read the system date from the 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
```

How to get size of contiguous low memory:

```
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
```

How to reboot the computer:

```
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.
```

Error handling

Some BIOS calls may not be implemented on every machine, and are not guaranteed to work. Often an unimplemented interrupt will return either `0x86` or `0x80` in register `AH`. **Just about every interrupt will set the carry flag (CF) on an error condition.** This makes it easy to jump to an error handler with the `jc` conditional jump. (See Conditional Jumps)

References

A rather exhaustive list of BIOS calls and other interrupts is Ralf Brown's Interrupt List. An HTML version can be found here.

Interrupts often assumed to be available are found in a list on Wikipedia.

A more in-depth overview of commonly available interrupts can be found at osdev.org

Read System Call Mechanisms online: https://riptutorial.com/x86/topic/6946/system-call-mechanisms

Credits

S. No	Chapters	Contributors
1	Getting started with Intel x86 Assembly Language & Microarchitecture	Community, David Hoelzer, Peter Cordes, Peter Mortensen, PyNEwbie, Runner
2	Assemblers	John Burger
3	Calling Conventions	Cody Gray, icktoofay, Margaret Bloom, Michael Petch, Peter Cordes, user45891, Zopesconk
4	Control Flow	Margaret Bloom, owacoder, Ped7g, Zopesconk
5	Converting decimal strings to integers	Margaret Bloom, MikeCAT
6	Data Manipulation	Ped7g, Zopesconk
7	Multiprocessor management	Margaret Bloom, Michael Petch, RamenChef
8	Optimization	Cody Gray, Downvoter, faissaloo, John Burger, sannaj, Stephen Leppik
9	Paging - Virtual Addressing and Memory	John Burger
10	Real vs Protected modes	John Burger, Margaret Bloom
11	Register Fundamentals	hidefromkgb, John Burger, Ped7g, Peter Cordes
12	System Call Mechanisms	owacoder