x86-64 (2)

Changelog

Corrections made in this version not in first posting:

28 Feb 2017: slide 55: REX prefix's first nibble is `0100`

VM assignment

please do it if you haven't

RE assignment

assembly reading practice

example manual page

INC—Increment by 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FE /0	INC *r/m8*	M	Valid	Valid	Increment *r/m* byte by 1.
REX + FE /0	INC *r/m8**	M	Valid	N.E.	Increment *r/m* byte by 1.
FF /0	INC *r/m16*	M	Valid	Valid	Increment *r/m* word by 1.
FF /0	INC *r/m32*	M	Valid	Valid	Increment *r/m* doubleword by 1.
REX.W + FF /0	INC *r/m64*	M	Valid	N.E.	Increment *r/m* quadword by 1.
40+ *rw***	INC *r16*	O	N.E.	Valid	Increment word register by 1.
40+ *rd*	INC *r32*	O	N.E.	Valid	Increment doubleword register by 1.

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

** 40H through 47H are REX prefixes in 64-bit mode.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
O	opcode + rd (r, w)	NA	NA	NA

Description

question: what was /0

- **/ digit** — A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.

"/ **digit** — A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The `reg` field contains the digit that provides an extension to the instruction's opcode."

huh? `ModR/M`? later today or Wednesday

LEA

like a `mov` — but stop at finding the memory address

never accesses memory

`lea` `(%rax), %rbx` is **`mov`** `%rax, %rbx`

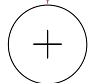
segmentation

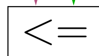
before virtual memory, there was segmentation

address

segment #: 0x1	offset: 0x23456

seg #	base	limit
0	0x14300	0x60000
1	0x50000	0x6F000
2	0x70000	0x30000





computed address

no segmentation fault?

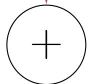
segmentation

before virtual memory, there was segmentation

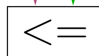
address

segment #: 0x1	offset: 0x23456

seg #	base	limit
0	0x0	0xFFFF FFFF FFFF FFFF
1	0x0	0xFFFF FFFF FFFF FFFF
2	0x0	0xFFFF FFFF FFFF FFFF



computed address



no segmentation fault?

x86 segmentation

addresses you've seen are the offsets

but every access uses a segment number!

segment numbers come from registers

- CS — code segment number (jump, call, etc.)
- SS — stack segment number (push, pop, etc.)
- DS — data segment number (mov, add, etc.)
- ES — addt'l data segment (string instructions)
- FS, GS — extra segments (never default)

instructions can have a segment override:

```
movq $42, %fs:100(%rsi)
    // move 42 to segment (# in FS),
    // offset 100 + RSI
```

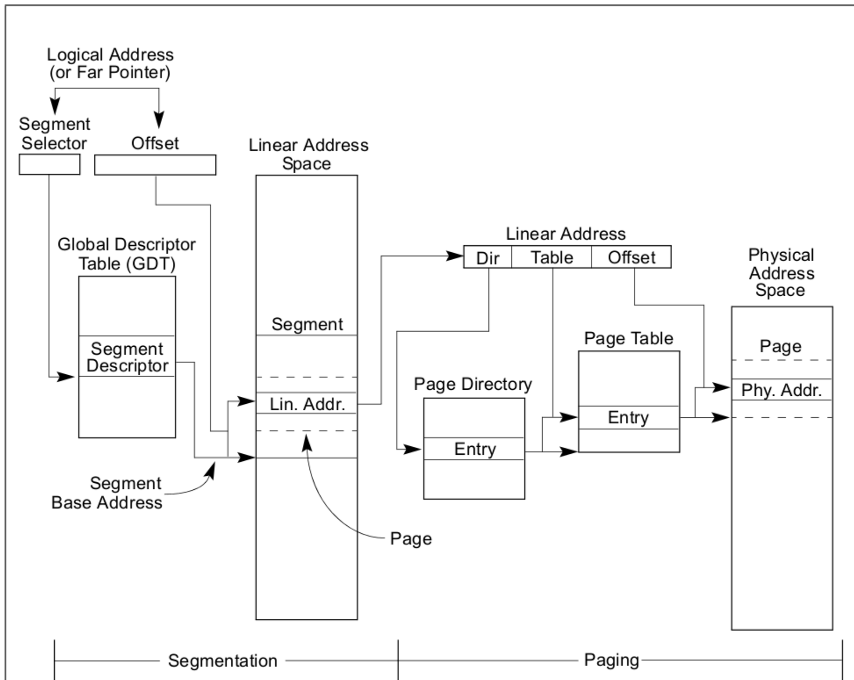


Figure 3-1. Segmentation and Paging

Figure: Intel manuals, Vol 3A

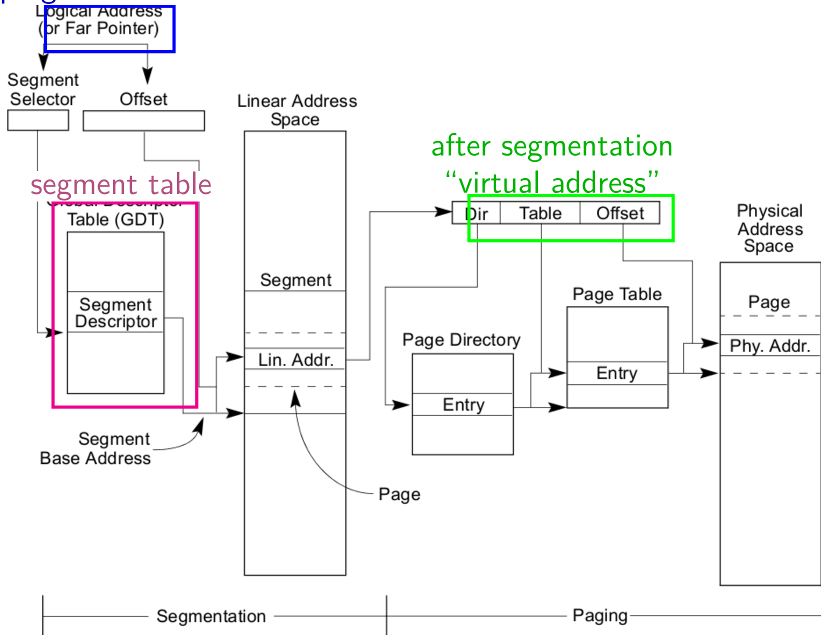


Figure: Intel manuals, Vol 3A

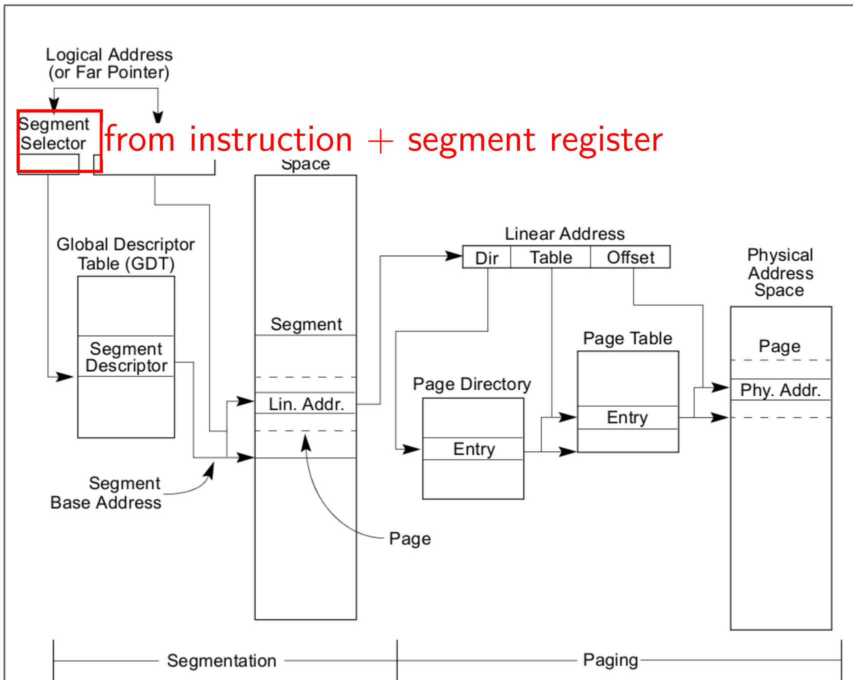
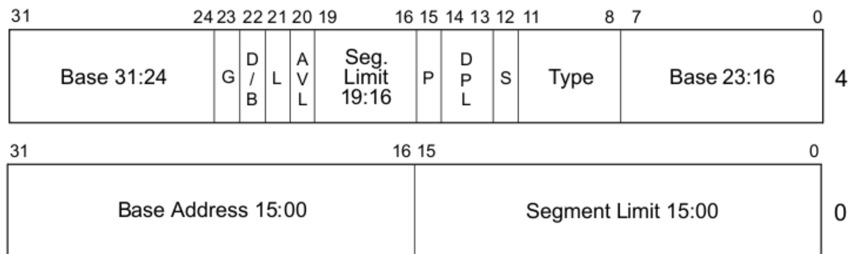


Figure: Intel manuals, Vol 3A

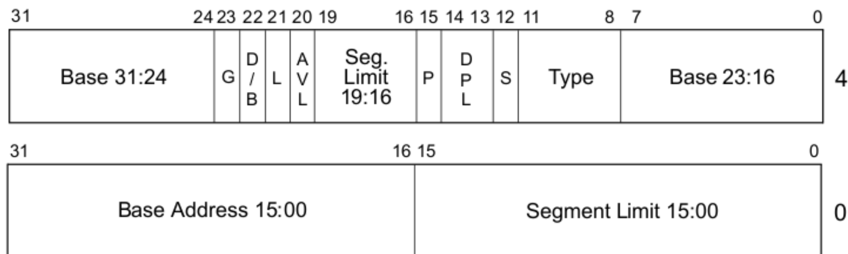
x86 segment descriptor



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure: Intel manuals, Volume 3A

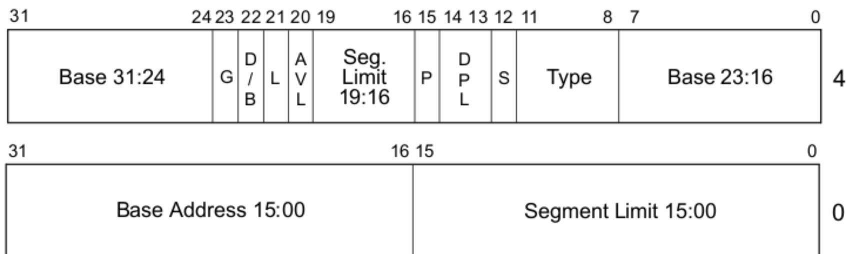
x86 segment descriptor



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level user or kernel mode? (if code)
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure: Intel manuals, Volume 3A

x86 segment descriptor



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level

64-bit or 32-bit or 16-bit mode? (if code)

LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure: Intel manuals, Volume 3A

64-bit segmentation

in 64-bit mode:

limits are ignored

base addresses are ignored

...except for `%fs`, `%gs`

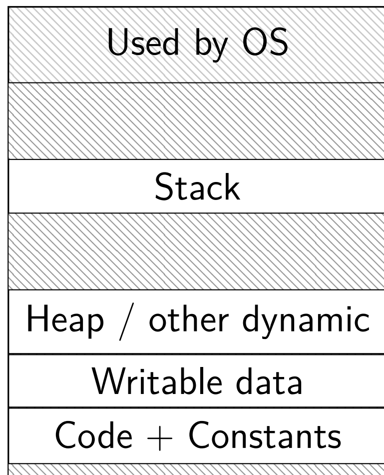
    when explicit segment override is used

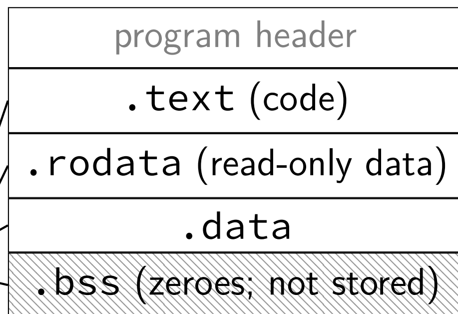
effectively: extra pointer register

segmentation and RE assignment

```
mov %fs:0x28, %rax
```

memory v. disk





ELF (executable and linking format)

Linux (and some others) executable/object file format

header: machine type, file type, etc.
program header: "segments" to load (also, some other information)
segment 1 data
segment 2 data

section header: list of "sections"(mostly for linker)

segments versus sections?

note: ELF terminology; may not be true elsewhere!

sections — object files (and usually executables), used by linker

- have information on intended purpose
- linkers combine these to create executables
- linkers might omit unneeded sections

segments — executables, used to actually load program

- program loader is dumb — doesn't know what segments are for

ELF example

`objdump -x /bin/busybox` (on my laptop)

`-x`: output all headers

```
/bin/busybox:     file format elf64-x86-64
/bin/busybox
architecture: i386:x86-64, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x0000000000401750

Program Header:
[...]

Sections:
[...]
```

ELF example

`objdump -x /bin/busybox` (on my laptop)

`-x`: output all headers

```
/bin/busybox:     file format elf64-x86-64
/bin/busybox
architecture: i386:x86-64, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x0000000000401750

Program Header:
[...]

Sections:
[...]
```

ELF example

`objdump -x /bin/busybox` (on my laptop)

`-x`: output all headers

```
/bin/busybox:     file format elf64-x86-64
/bin/busybox
architecture: i386:x86-64, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x0000000000401750

Program Header:
[...]

Sections:
[...]
```

a program header (1)

```
Program Header:
[...]
LOAD off    0x0000000 vaddr 0x0400000 paddr 0x0400000 align 2**21
     filesz 0x01db697 memsz 0x01db697 flags r-x
LOAD off    0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**21
     filesz 0x00021ee memsz 0x0007d18 flags rw-
[...]
```

load `0x1db697` bytes:

- from `0x0` bytes into the file
- to memory at `0x40000`
- readable and executable

load `0x21ee` bytes:

- from `0x1dbea8`
- to memory at `0x7dbea8`
- plus (`0x7d18`−`0x21ee`) bytes of zeroes
- readable and writable

a program header (1)

```
Program Header:
[...]
LOAD off    0x0000000 vaddr 0x0400000 paddr 0x0400000 align 2**21
     filesz 0x01db697 memsz 0x01db697 flags r-x
LOAD off    0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**21
     filesz 0x00021ee memsz 0x0007d18 flags rw-
[...]
```

load `0x1db697` bytes:

- from `0x0` bytes into the file
- to memory at `0x40000`
- readable and executable

load `0x21ee` bytes:

- from `0x1dbea8`
- to memory at `0x7dbea8`
- plus (`0x7d18`−`0x21ee`) bytes of zeroes
- readable and writable

a program header (1)

```
Program Header:
[...]
LOAD off    0x0000000 vaddr 0x0400000 paddr 0x0400000 align 2**21
     filesz 0x01db697 memsz 0x01db697 flags r-x
LOAD off    0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**21
     filesz 0x00021ee memsz 0x0007d18 flags rw-
[...]
```

load `0x1db697` bytes:

- from `0x0` bytes into the file
- to memory at `0x40000`
- readable and executable

load `0x21ee` bytes:

- from `0x1dbea8`
- to memory at `0x7dbea8`
- plus (`0x7d18`−`0x21ee`) bytes of zeroes
- readable and writable

a program header (1)

```
Program Header:
[...]
LOAD off    0x0000000 vaddr 0x0400000 paddr 0x0400000 align 2**21
     filesz 0x01db697 memsz 0x01db697 flags r-x
LOAD off    0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**21
     filesz 0x00021ee memsz 0x0007d18 flags rw-
[...]
```

load `0x1db697` bytes:

- from `0x0` bytes into the file
- to memory at `0x40000`
- readable and executable

load `0x21ee` bytes:

- from `0x1dbea8`
- to memory at `0x7dbea8`
- plus (`0x7d18`–`0x21ee`) bytes of zeroes
- readable and writable

a program header (2)

```
Program Header:
[...]
 NOTE off    0x0000190 vaddr 0x0400190 paddr 0x0400190 align 2**2
      filesz 0x0000044 memsz 0x0000044 flags r--
  TLS  off    0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**3
      filesz 0x0000030 memsz 0x000007a flags r--
STACK off    0x0000000 vaddr 0x0000000 paddr 0x0000000 align 2**4
      filesz 0x0000000 memsz 0x0000000 flags rw-
RELRO off    0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**0
      filesz 0x0000158 memsz 0x0000158 flags r--
[...]
```

NOTE — comment

TLS — thread-local storage region (used via `%fs`)

STACK — indicates stack is read/write

RELRO — make this read-only after runtime linking

section headers

```
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .note.ABI-tag 00000020  0000000000400190  0000000000400190  00000190  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.gnu.build-id 00000024  00000000004001b0  00000000004001b0  000001b0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .rela.plt     00000210  00000000004001d8  00000000004001d8  000001d8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .init         0000001a  00000000004003e8  00000000004003e8  000003e8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  4 .plt          00000160  0000000000400410  0000000000400410  00000410  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  5 .text         0017ff1d  0000000000400570  0000000000400570  00000570  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  6 __libc_freeres_fn 00002032  0000000000580490  0000000000580490  00180490  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  7 __libc_thread_freeres_fn 0000021b  00000000005824d0  00000000005824d0  001824d0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  8 .fini         00000009  00000000005826ec  00000000005826ec  001826ec  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  9 .rodata       00044ac8  0000000000582700  0000000000582700  00182700  2**6
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 __libc_subfreeres 000000c0  00000000005c71c8  00000000005c71c8  001c71c8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 11 .stapsdt.base 00000001  00000000005c7288  00000000005c7288  001c7288  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 12 __libc_atexit 00000008  00000000005c7290  00000000005c7290  001c7290  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 13 __libc_thread_subfreeres 00000018  00000000005c7298  00000000005c7298  001c7298  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .eh_frame     000141dc  00000000005c72b0  00000000005c72b0  001c72b0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 15 .gcc_except_table 0000020b  00000000005db48c  00000000005db48c  001db48c  2**0
```

sections

tons of “sections”

not actually needed/used to run program

size, file offset, flags (code/data/etc.)

- location in executable *and* in memory

some sections aren’t stored (no “CONTENTS” flag)

- just all zeroes

selected sections

`.text`	program code
`.bss`	initially zero data (block started by symbol)
`.data`	other writeable data
`.rodata`	read-only data
`.init`/`.fini`	global constructors/destructors
`.got`/`.plt`	linking related
`.eh_frame`	try/catch related

based on `http://people.redhat.com/mpolacek/src/devconf2012.pdf`

other executable formats

PE (Portable Executable) — Windows

Mach-O — MacOS X

broadly similar to ELF

differences:

- whether segment/section distinction exists
- how linking/debugging info represented
- how program start info represented

simple executable startup

copy segments into memory

jump to start address

executable startup code

Linux: executables don't start at `main`

why not?

- need to initialize `printf`, `cout`, `malloc`, etc. data structures
- `main` needs to return somewhere

compiler links in startup code

linking

`callq 0x458F0`

static v. dynamic linking

static linking — linking to create executable

dynamic linking — linking when executable is run

static v. dynamic linking

static linking — linking to create executable

dynamic linking — linking when executable is run

conceptually: no difference in how they work

reality — very different mechanisms

linking data structures

symbol table: `name` ⇒ (section, offset)

- example: `main:` in assembly adds symbol table entry for `main`

relocation table: offset ⇒ (name, kind)

- example: `call printf` adds relocation for name `printf`
- kind depends on how instruction encodes address

hello.s

```
.data
string: .asciz "Hello,␣World!"
.text
.globl main
main:
    movq $string, %rdi
    call puts
    ret
```

hello.o

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

hello.o

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

undefined symbol: look for `puts` elsewhere

hello.o

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

insert address of puts, format for `call`

hello.o

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

insert address of string, format for movq

hello.o

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

different ways to represent address

32S — signed 32-bit value

PC32 — 32-bit difference from current address

hello.o

```
SYMBOL TABLE:
0000000000000000 l     d  .text  0000000000000000 .text
0000000000000000 l     d  .data  0000000000000000 .data
0000000000000000 l     d  .bss   0000000000000000 .bss
0000000000000000 l        .data  0000000000000000 string
0000000000000000 g        .text  0000000000000000 main
0000000000000000          *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

g: global — used by other files
l: local

hello.o

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

.text segment beginning plus 0 bytes

interlude: strace

`strace` — system call tracer

- on Linux, some other Unices
- OS X approx. equivalent: `dtruss`
- Windows approx. equivalent: Process Monitor

indicates what system calls (operating system services) used by a program

statically linked hello.exe

`gcc -static -o hello-static.exe hello.s`

`strace ./hello-static.exe`:

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)                                  = 0x20a5000
brk(0x20a61c0)                             = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880)         = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) =
brk(0x20c71c0)                             = 0x20c71c0
brk(0x20c8000)                             = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)         = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)            = 14
exit_group(14)                             = ?
+++ exited with 14 +++
```

statically linked hello.exe

```
gcc -static -o hello-static.exe hello.s
```

`strace ./hello-static.exe`:

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)                                  = 0x20a5000
brk(0x20a61c0)                             = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880)         = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) =
brk(0x20c71c0)                             = 0x20c71c0
brk(0x20c8000)                             = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)         = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)            = 14
exit_group(14)                             = ?
+++ exited with 14 +++
```

standard library startup

statically linked hello.exe

```
gcc -static -o hello-static.exe hello.s
```

strace ./hello-static.exe:

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)                               = 0x20a5000
brk(0x20a61c0)                          = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880)      = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) =
brk(0x20c71c0)                          = 0x20c71c0
brk(0x20c8000)                          = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

memory allocation

statically linked hello.exe

`gcc -static -o hello-static.exe hello.s`

`strace ./hello-static.exe`:

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)                                 = 0x20a5000
brk(0x20a61c0)                            = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880)        = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) =
brk(0x20c71c0)                            = 0x20c71c0
brk(0x20c8000)                            = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)        = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)           = 14
exit_group(14)                            = ?
+++ exited with 14 +++
```

implementation of puts

statically linked hello.exe

```
gcc -static -o hello-static.exe hello.s
```

`strace ./hello-static.exe`:

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)                                  = 0x20a5000
brk(0x20a61c0)                             = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880)         = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) =
brk(0x20c71c0)                             = 0x20c71c0
brk(0x20c8000)                             = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)         = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)            = 14
exit_group(14)                             = ?
+++ exited with 14 +++
```

standard library shutdown

dynamically linked hello.exe

`gcc -o hello.exe hello.s`

`strace ./hello.exe`:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or direc
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, st_mode=S_IFREG|0644, st_size=137808, ...) = 0
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee91
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

dynamically linked hello.exe

```
gcc -o hello.exe hello.s
```

`strace ./hello.exe`:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or direc
open("/etc/
fstat(3, st
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee91
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

the standard C library (includes puts

dynamically linked hello.exe

`gcc -o hello.exe hello.s`

`strace ./hello.exe`:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or direc
open("/etc/
fstat(3, st
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee91
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

memory allocation (different method)

dynamically linked hello.exe

```
gcc -o hello.exe hello.s
```

`strace ./hello.exe`:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
access("/etc/ld.so.preload", R_OK)        = -1 ENOENT (No such file or direc
open("/etc/ld.so
fstat(3, st_mode
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee91
close(3)                                  = 0
...
write(1, "Hello, World!\n", 14)           = 14
exit_group(14)                            = ?
+++ exited with 14 +++
```

read standard C library header

dynamically linked hello.exe

`gcc -o hello.exe hello.s`

`strace ./hello.exe`:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or direc
open("/et
fstat(3,
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee91
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

load standard C library (3 = opened file)

dynamically linked hello.exe

`gcc -o hello.exe hello.s`

`strace ./hello.exe`:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or direc
open
fsta
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee91
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

allocate zero-initialized data segment for C library

dynamic linking

load and link (find address of `puts`) runtime

advantages:

- smaller executables
- easier upgrades
- less memory usage (load one copy of library for multiple programs)

disadvantages:

- library upgrades breaking programs
- programs less compatible between OS versions
- possibly slower

where's the linker

Where's the code that calls `open("...libc.so.6")`?

Could check `hello.exe` — it's not there!

where's the linker

Where's the code that calls `open("...libc.so.6")`?

Could check `hello.exe` — it's not there!

instead: "interpreter" `/lib64/ld-linux-x86-64.so.2`

on Linux: contains loading code instead of core OS

- OS loads it instead of program

objdump — the interpreter

excerpt from `objdump -sx hello.exe`:

```
Program Header:
...
  INTERP off    0x0000238 vaddr 0x0400238 paddr 0x0400238 align 2**0
         filesz 0x000001c memsz 0x000001c flags r--
...
Contents of section .interp:
 400238 2f6c6962 36342f6c 642d6c69 6e75782d  /lib64/ld-linux-
 400248 7838362d 36342e73 6f2e3200           x86-64.so.2.
```

dynamic linking information

symbol table — works the same, but in executable

could use same relocations — but these are expensive

rather just copy data from disk without changes

solutions: global lookup table!

dynamically linked puts

```
0000000000400400 <puts@plt>:
  400400:       ff 25 12 0c 20 00           jmpq   *0x200c12(%rip)
                   /* 0x200c12+RIP = _GLOBAL_OFFSET_TABLE_+0x18 */
... later in main: ...
  40052d:       e8 ce fe ff ff              callq  400400 <puts@plt>
                       /* instead of call puts */
```

replace `puts` with stub `puts@plt`

- plt = procedure linkage table

stub: jump to `*_GLOBAL_OFFSET_TABLE[3]`

dynamic linker changes table instead of code

- could change code — just would be less efficient

lazy binding

```
0000000000400400 <puts@plt>:
  400400:       ff 25 12 0c 20 00         jmpq   *0x200c12(%rip)
                   /* 0x200c12+RIP = _GLOBAL_OFFSET_TABLE_+0x18 */
  400406:       68 00 00 00 00            pushq  $0x0
  40040b:       e9 e0 ff ff ff            jmpq   4003f0 <_init+0x28>
```

could fill global offset table immediately

alternative: fill on demand

extra code (`pushq` then `jmpq`) runs "fixup code"

- reads symbol tables to find function
- edits global offset table
- jumps to function

called "lazy binding"

lazy binding pro/con

advantages:

- faster program loading
- no overhead for unused code (often a lot of stuff)

disadvantages:

- can move errors (missing functions, etc.) to runtime
- possibly more total overhead

x86 instruction encoding

in 2110, 3330 you learned a “teaching” machine code

Y86 (3330) is very like what x86 should be

...but it isn’t

why? history!

the 8086

1979 Intel processor

4 general purpose 16-bit registers: AX, BX, CX, DX

4 special 16-bit registers: SI, DI, BP, SP

8086 instruction encoding: simple

special cases: 1-byte instructions:

- anything with no arguments
- push ax, push bx, push cx, …(dedicated opcodes)
- pop ax, …

8086 instruction encoding: two-arg

1-byte opcode

sometimes `ModRM` byte:

- 2-bit "mod" and
- 3-bit register number (source or dest, depends on opcode) and
- 3-bit "r/m"

"mod" + "r/m" specify one of:

- `%reg` (mod = `11`)
- `(%bx/%bp, %si/%di)`
- `(%bx/%si/%di)`
- `offset(%bx/%bp/,%si/%di)` (8- or 16-byte offset)

non-intuitive table

8086 ModRM table

Effective Address	Mod	R/M
[BX+SI]	00	000
[BX+DI]		001
[BP+SI]		010
[BP+DI]		011
[SI]		100
[DI]		101
disp16[2]		110
[BX]		111
[BX+SI]+disp8[3]	01	000
[BX+DI]+disp8		001
[BP+SI]+disp8		010
[BP+DI]+disp8		011
[SI]+disp8		100
[DI]+disp8		101
[BP]+disp8		110
[BX]+disp8		111
[BX+SI]+disp16	10	000
[BX+DI]+disp16		001
[BP+SI]+disp16		010
[BP+DI]+disp16		011
[SI]+disp16		100
[DI]+disp16		101
[BP]+disp16		110
[BX]+disp16		111
EAX/AX/AL/MM0/XMM0	11	000
ECX/CX/CL/MM1/XMM1		001
EDX/DX/DL/MM2/XMM2		010
EBX/BX/BL/MM3/XMM3		011
ESP/SP/AHMM4/XMM4		100
EBP/BP/CH/MM5/XMM5		101
ESI/SI/DH/MM6/XMM6		110
EDI/DI/BH/MM7/XMM7		111

8086 evolution

Intel 8086 — 1979, 16-bit registers

Intel (80)386 — 1986, 32-bit registers

AMD K8 — 2003, 64-bit registers

x86 modes

x86 has multiple modes

maintains compatiblity

e.g.: modern x86 processor can work like 8086

called “real mode”

different mode for 32-bit/64-bit

same basic encoding; some sizes change

32-bit ModRM table

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =		
Effective Address	**Mod**	**R/M**
[EAX]	00	000
[ECX]		001
[EDX]		010
[EBX]		011
[--][--][1]		100
disp32[2]		101
[ESI]		110
[EDI]		111
[EAX]+disp8[3]	01	000
[ECX]+disp8		001
[EDX]+disp8		010
[EBX]+disp8		011
[--][--]+disp8		100
[EBP]+disp8		101
[ESI]+disp8		110
[EDI]+disp8		111
[EAX]+disp32	10	000
[ECX]+disp32		001
[EDX]+disp32		010
[EBX]+disp32		011
[--][--]+disp32		100
[EBP]+disp32		101
[ESI]+disp32		110
[EDI]+disp32		111
EAX/AX/AL/MM0/XMM0	11	000
ECX/CX/CL/MM/XMM1		001
EDX/DX/DL/MM2/XMM2		010
EBX/BX/BL/MM3/XMM3		011
ESP/SP/AH/MM4/XMM4		100
EBP/BP/CH/MM5/XMM5		101
ESI/SI/DH/MM6/XMM6		110
EDI/DI/BH/MM7/XMM7		111

32-bit addition: SIB bytes

8086 addressing modes made registers different

32-bit mode got rid of this (mostly)

problem: not enough spare bits in `ModRM` byte

solution: if "r/m" bits = `100`, extra "SIB" byte:

- 2 bit scale: `00` is 1, `01` is 2, `10` is 4, `11` is 8
- 3 bit index: index register number
- 3 bit base: base register number

`(%baseReg,%indexReg,scale)`

32-bit addition: SIB bytes

8086 addressing modes made registers different

32-bit mode got rid of this (mostly)

problem: not enough spare bits in `ModRM` byte

solution: if “r/m” bits = `100`, extra “SIB” byte:

- 2 bit scale: `00` is 1, `01` is 2, `10` is 4, `11` is 8
- 3 bit index: index register number
- 3 bit base: base register number

`(%baseReg,%indexReg,scale)`

32-bit addition: SIB bytes

8086 addressing modes made registers different

32-bit mode got rid of this (mostly)

problem: not enough spare bits in `ModRM` byte

solution: if "r/m" bits = `100`, extra "SIB" byte:

- 2 bit scale: `00` is 1, `01` is 2, `10` is 4, `11` is 8
- 3 bit index: index register number
- 3 bit base: base register number

```
(%baseReg,%indexReg,scale)
```

32-bit addition: SIB bytes

8086 addressing modes made registers different

32-bit mode got rid of this (mostly)

problem: not enough spare bits in `ModRM` byte

solution: if "r/m" bits = `100`, extra "SIB" byte:

- 2 bit scale: `00` is 1, `01` is 2, `10` is 4, `11` is 8
- 3 bit index: index register number
- 3 bit base: base register number

```
(%baseReg,%indexReg,scale)
```

intel manual: SIB table

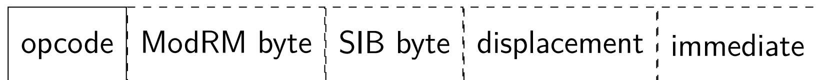
Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

NOTES:

1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the

basic 32-bit encoding



dashed: not always present

opcodes: 1-3 bytes

- some 5-bit opcodes, with 3-bit register field
- (alternate view: 8-bit opcode with fixed register)
- sometimes part of ModRM used as add'tl part of opcode

displacement, immediate: 1, 2, or 4 bytes

- or, rarely, 8 bytes

what about 64-bit?

adds 8 more registers — more bits for reg #?

didn't change encoding for existing instructions, so…

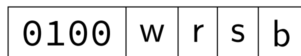
instruction prefix "REX"

- 32-bit x86 already had many prefixes

also selects 64-bit version of instruction

REX prefix

REX prefix byte



extra bit for ModRM r/m
or SIB base reg field

extra bit for SIB byte index reg field

extra bit for ModRM reg field

1 if 64-bit regs (%rax, etc.)
0 if 32-bit regs (%eax, etc.)

overall encoding

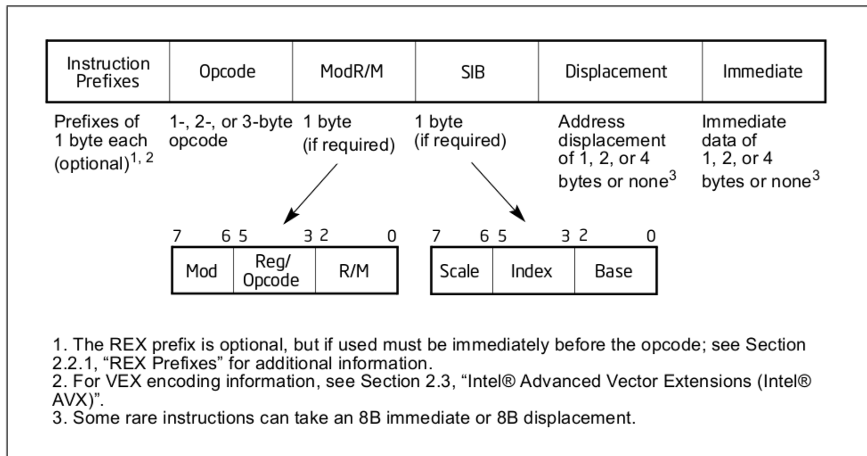


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

instruction prefixes

REX (64-bit and/or extra register bits)

VEX (SSE/AVX instructions; other new instrs.)

operand/address-size change (64/32 to 16 or vice-versa)

LOCK — synchronization between processors

REPNE/REPNZ/REP/REPE/REPZ — turns instruction into loop

segment overrides

x86 encoding example (1)

`pushq %rax` encoded as `50`

- 5-bit opcode `01010` plus 3-bit register number `000`

`pushq %r13` encoded as `41 55`

- `41`: REX prefix `0010` (constant), w:`0`, r:`0`, s:`0`, b:`1`
- w = `0` because push is never 32-bit in 64-bit mode
- `55`: 5-bit opcode `01010`; 3-bit reg # `101`
- 4-bit reg # `1101` = 13

x86 encoding example (2)

`addq 0x12345678(%rax,%rbx,2), %ecx`

`03`: opcode — add r/m32 to r/m32

`8c`: ModRM: mod = `10`; reg = `001`, r/m: `100`

- reg = `001` = `%ecx` (table)
- SIB byte + 32-bit displacement (table)

`58`: SIB: scale = `01`, index = `011`, base = `000`

- index `011` = `%rbx`; base `000` = `%rax`;

`78 56 32 12`: 32-bit constant `0x12345678`

x86 encoding example (3)

`addq 0x12345678(%r10,%r11,2), %rax`

`4b`: REX prefix `0100`+w:`1`, r:`0`, s:`1`, b:`1`

`03`: opcode — add r/m64 to r64 (with REX.w)

`84`: ModRM: mod = `10`; reg = `000`, r/m: `100`
reg = `0000` = %rax
SIB byte + 32-bit displacement (table)

`5a`: SIB: scale = `01`, index = `011`, base = `010`
with REX: index = `1011` (11), base = `1010` (10)

`78 56 32 12`: 32-bit constant `0x12345678`

x86 encoding example (3)

`addq 0x12345678(%r10,%r11,2), %rax`

`4b`: REX prefix `0100`+w:`1`, r:`0`, s:`1`, b:`1`

`03`: opcode — add r/m64 to r64 (with REX.w)

`84`: ModRM: mod = `10`; reg = `000`, r/m: `100`
- reg = `0000` = %rax
- SIB byte + 32-bit displacement (table)

`5a`: SIB: scale = `01`, index = `011`, base = `010`
- with REX: index = `1011` (11), base = `1010` (10)

`78 56 32 12`: 32-bit constant `0x12345678`

x86 encoding example (4)

`movq %fs:0x10,%r13`

64: FS segment override

48: REX: w: 1 (64-bit), r: 1, s: 0, b: 0

8b: opcode for MOV memory to register

2c: ModRM: mod = 00, reg = 101, r/m: 100
with REX: reg = **1**101 [%r12]; r/m = 100 (SIB follows)

25: SIB: scale = 00; index = 0100; base = 0101
no register/no register in table

10 00 00 00: 4-byte constant 0x10

x86-64 impossibilities

illegal: **movq** `0x12345678ab(%rax), %rax`

- maximum 32-bit displacement
- **movq** `0x12345678ab, %rax` okay
  - extra `mov` opcode for `%rax` only

illegal: **movq** `$0x12345678ab, %rbx`

- maximum 32-bit constant
- **movq** `$0x12345678ab, %rax` okay

illegal: **pushl** `%eax`

- no 32-bit push/pop in 64-bit mode
- but 16-bit allowed (operand size prefix byte 66)

illegal: **movq** `(%rax, %rsp), %rax`

- cannot use `%rsp` as index register
- **movq** `(%rsp, %rax), %rax` okay

instruction prefixes

REX (64-bit and/or extra register bits)

VEX (SSE/AVX instructions; other new instrs.)

operand/address-size change (64/32 to 16 or vice-versa)

LOCK — synchronization between processors

REPNE/REPNZ/REP/REPE/REPZ — turns instruction into loop

segment overrides

string instructions (1)

```
memcpy: // copy %rdx bytes from (%rsi) to (%rdi)
        cmpq %rdx, %rdx
        je done
        movsb
        subq $1, %rdx
        jmp memcpy
done:   ret
```

`movsb` (move data from string to string, byte)

mov one byte from `(%rsi)` to `(%rdi)`

increment %rsi and %rdi (*)

cannot specify other registers

string instructions (2)

```
memcpy: // copy %rdx bytes from (%rsi) to (%rdi)
     rep movsb
     ret
```

`rep` prefix byte

repeat instruction until `%rdx` is 0

decrement `%rdx` each time

cannot specify other registers

cannot use `rep` with all instructions

string instructions (3)

`lodsb`, `stosb` — load/store into string

`movsw`, `movsd` — word/dword versions

string comparison instructions

`rep movsb` is still recommended on modern Intel

- special-cased in processor?

exploring assembly

compiling little C programs looking at the assembly is nice:

`gcc -S -O`

- extra stuff like `.cfi` directives (for try/catch)

or disassemble:

`gcc -O -c file.c` (or make an executable)

`objdump -dr file.o` (or on an executable)

- d: disassemble
- r: show (non-dynamic) relocations

exploring assembly

compiling little C programs looking at the assembly is nice:

`gcc -S -O`

- extra stuff like `.cfi` directives (for try/catch)

or disassemble:

`gcc -O -c file.c` (or make an executable)

`objdump -dr file.o` (or on an executable)

- d: disassemble
- r: show (non-dynamic) relocations

assembly without optimizations

compilers do really silly things without optimizations:

```
int sum(int x, int y) { return x + y; }
```

```
sum:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret
```

instead of `gcc -O` version:

```
sum:
    leal (%rdi,%rsi), %eax
    ret
```

assembly reading advice

don't know what an instruction does: look it up!

remember calling conventions

function/variable names (if present) help

try to name values in registers, on stack

- based on context
- "input size" not "rax"

next time: looking at viruses

how/where to insert virus code?

how/where to copy self?