assembly review/extras

last time

malware ("evil software") types

- viruses — embed/copy self into legit execs/docs
- worms — self-replicating programs
- trojan (horse)s — good-looking software that's evil
- …

malware motivations — profit and other

- fraud (fake antivirus; ad fraud; …)
- spying (financial info; nation-state espinage; stalking; …)
- ransomware
- adware
- taking CPU/network resources (DDoS; cryptocurrency mining; …)

quiz demo

x86-64 assembly

history: AMD constructed 64-bit extension to x86 first

- marketing term: AMD64

Intel first tried a new ISA (Itanium), which failed

Then Intel copied AMD64

- marketing term: EM64T
  - Extended Memory 64 Technology
- later marketing term: Intel 64

both Intel and AMD have manuals — definitive reference

Intel® 64 and IA-32 Architectures Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D

x86-64 manuals

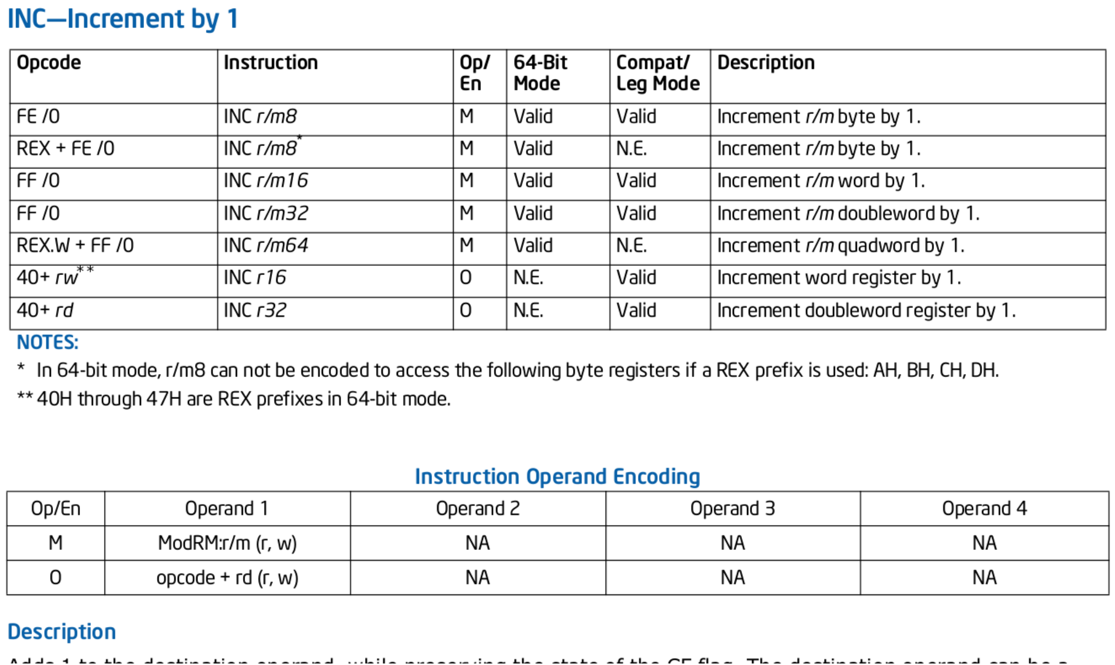
Intel manuals:

`https://software.intel.com/en-us/articles/intel-sdm`
24 MB, 4684 pages
Volume 2: instruction set reference (2190 pages)

AMD manuals:

`https://support.amd.com/en-us/search/tech-docs`
“AMD64 Architecture Programmer’s Manual”

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

Adds 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a

instruction listing parts (1)

opcode — first part of instruction encoding

> yes, variable length
> "REX"???
> more later (today or next week)

instruction — Intel assembly skeleton

`r/m32` — 32-bit memory or register value

64-bit mode — does instruction exist in 64-bit mode?

compat/leg mode — in 16-bit/32-bit modes?

instruction listing parts (2)

dscription + operation (later on page)

- text and pseudocode description

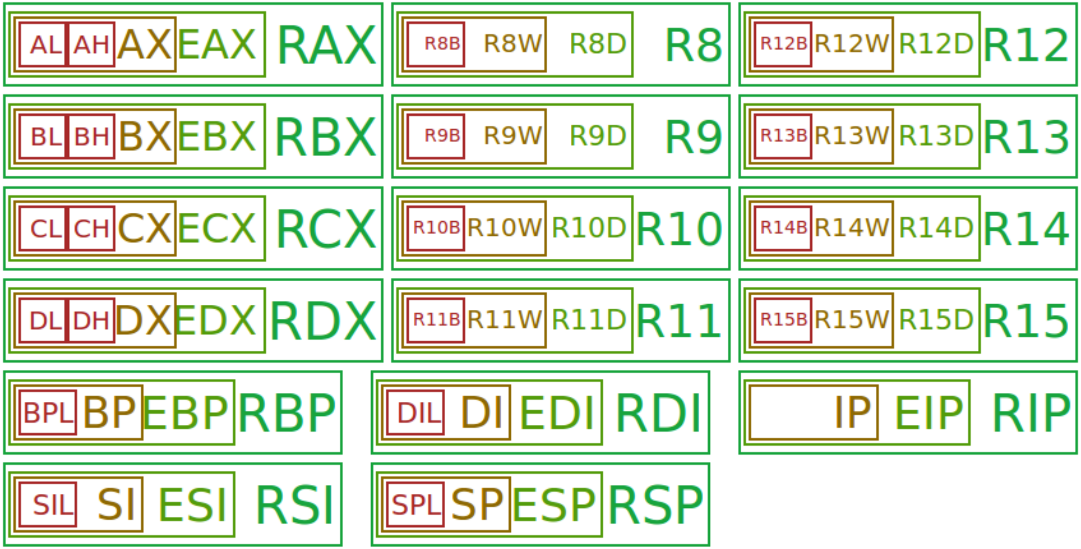
flags affected

- flags — used by `jne`, etc.

exceptions — how can OS be called from this?

- example: can invalid memory access happen?

recall: x86-64 general purpose registers



Immae via Wikipedia

overlapping registers (1)

setting 32-bit registers sets whole 64-bit register

extra bits are always zeroes

```
movq $0x123456789abcdef, %rax
xor %eax, %eax
// %rax is 0, not 0x1234567800000000
movl $−1, %ebx
// %rbx is 0xFFFFFFFF, not -1 (0xFFFFF...FFF)
```

overlapping registers (2)

setting 8/16-bit registers doesn't change rest of 64-bit register:

```
movq $0x12345789abcdef, %rax
movw $0xaaaa, %ax
// %rax is 0x123456789abaaaa
```

AT&T versus Intel syntax

AT&T syntax:

```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:

```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] <- 42
```

AT&T syntax (1)

```
movq $42, 100(%rbx,%rcx,4)
```

destination last

constants start with $

registers start with %

AT&T syntax (2)

`movq $42, 100(%rbx,%rcx,4)`

operand length: `q`

- `l` $= 4$; `w` $= 2$; `b` $= 1$

`100(%rbx,%rcx,4)`:
`memory[100 + rbx + rcx * 4]`

`sub %rax, %rbx`: `rbx` $\leftarrow$ `rbx - rax`

Intel syntax

destination first

`[...]` indicates location in memory

`QWORD PTR [...]` for 8 bytes in memory

- DWORD for 4
- WORD for 2
- BYTE for 1

On LEA

LEA = Load Effective Address

uses the syntax of a memory access, but...

just computes the address and uses it:

`leaq 4(%rax), %rax` same as `addq $4, %rax`

almost — doesn't set condition codes

LEA tricks

`leaq (%rax,%rax,4), %rax` multiplies `%rax` by 5

    address-of(memory[rax + rax * 4])

`leal (%rbx,%rcx), %eax` adds rbx + rcx into eax

    ignores top 64-bits

question

```
.data
string:
    .asciz "abcdefgh"
.text
    movq $string, %rax
    movq string, %rdx
    movb (%rax), %bl
    leal 1(%rbx), %ebx
    movb %bl, (%rax)
    movq %rdx, 4(%rax)
```

What is the final value of string?

a. `"abcdabcd"`
b. `"bbcdefgh"`
c. `"bbcdabcd"`
d. `"abcdefgh"`
e. something else / not enough info

reading objdump disassembly

often, we'll want to work from binaries to assembly

tool we'll use on Linux: `objdump`

example from *objdump –disassemble* of hello-wolrd program:

```
0000000000001060 <main>:
    1060:   f3 0f 1e fa             endbr64
    1064:   50                      push   %rax
    1065:   48 8d 3d 98 0f 00 00    lea    0xf98(%rip),%rdi # 2004 <_IO_stdin

    106c:   e8 df ff ff ff          callq  1050 <puts@plt>
    1071:   31 c0                   xor    %eax,%eax
    1073:   5a                      pop    %rdx
    1074:   c3                      retq
```

reading objdump disassembly

symbol main at address 0x1060

often, we'll want to work from binaries to assembly

tool we'll use on Linux: `objdump`

example from *objdump –disassemble* of hello-wolrd program:

```
0000000000001060 <main>:
    1060:   f3 0f 1e fa             endbr64
    1064:   50                      push   %rax
    1065:   48 8d 3d 98 0f 00 00    lea    0xf98(%rip),%rdi # 2004 <_IO_stdin

    106c:   e8 df ff ff ff          callq  1050 <puts@plt>
    1071:   31 c0                   xor    %eax,%eax
    1073:   5a                      pop    %rdx
    1074:   c3                      retq
```

reading objdump disassembly

often, we'll

tool we'll u

first column: instruction addresses in hexadecimal
(if executable/library has fixed address,
actual addresses they'll be loaded to)

example from *objdump –disassemble* of hello-wolrd program:

```
0000000000001060 <main>:
    1060:   f3 0f 1e fa             endbr64
    1064:   50                      push   %rax
    1065:   48 8d 3d 98 0f 00 00    lea    0xf98(%rip),%rdi # 2004 <_IO_stdin

    106c:   e8 df ff ff ff          callq  1050 <puts@plt>
    1071:   31 c0                   xor    %eax,%eax
    1073:   5a                      pop    %rdx
    1074:   c3                      retq
```

reading objdump disassembly

machine code as list of byte values in hexadecimal

often, we'll want to work from binaries to assembly

tool we'll use on Linux: `objdump`

example from *objdump –disassemble* of hello-wolrd program:

```
0000000000001060 <main>:
    1060:   f3 0f 1e fa             endbr64
    1064:   50                      push   %rax
    1065:   48 8d 3d 98 0f 00 00    lea    0xf98(%rip),%rdi # 2004 <_IO_stdin

    106c:   e8 df ff ff ff          callq  1050 <puts@plt>
    1071:   31 c0                   xor    %eax,%eax
    1073:   5a                      pop    %rdx
    1074:   c3                      retq
```

reading objdump disassembly

often, we'll want t

tool we'll use on L

```
callq 1050 <puts@plt>
```

call to address 0x1050

and `puts@plt` is at that address

example from *objdump –disassemble* of hello-wolrd program:

```
0000000000001060 <main>:
    1060:   f3 0f 1e fa             endbr64
    1064:   50                      push   %rax
    1065:   48 8d 3d 98 0f 00 00    lea    0xf98(%rip),%rdi # 2004 <_IO_stdin

    106c:   e8 df ff ff ff          callq  1050 <puts@plt>
    1071:   31 c0                   xor    %eax,%eax
    1073:   5a                      pop    %rdx
    1074:   c3                      retq
```

reading objdump disassembly

c

t

```
lea 0xf98(%rip),%rdi # 2004 <_IO_stdin_used+0x4>
```

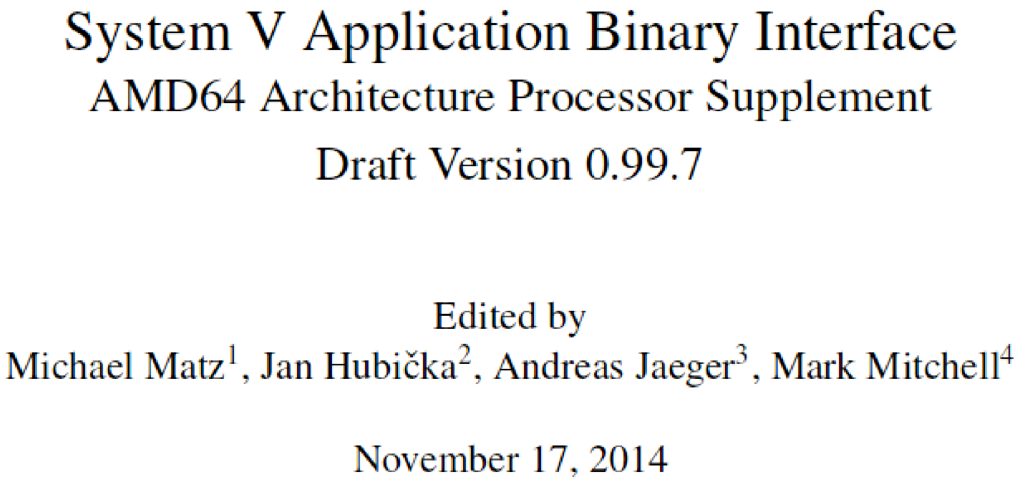
`0xf98(%rip)` computes address 0x2004

which is `0x4` bytes after `_IO_stdin_used`

example from *objdump –disassemble* of hello-wolrd program:

```
0000000000001060 <main>:
    1060:   f3 0f 1e fa             endbr64
    1064:   50                      push   %rax
    1065:   48 8d 3d 98 0f 00 00    lea    0xf98(%rip),%rdi # 2004 <_IO_stdin
             # 2004 <_IO_stdin_used+0x4>
    106c:   e8 df ff ff ff          callq  1050 <puts@plt>
    1071:   31 c0                   xor    %eax,%eax
    1073:   5a                      pop    %rdx
    1074:   c3                      retq
```

Linux x86-64 calling convention



System V Application Binary Interface

AMD64 Architecture Processor Supplement

Draft Version 0.99.7

Edited by

Michael Matz[1], Jan Hubička[2], Andreas Jaeger[3], Mark Mitchell[4]

November 17, 2014

Linux x86-64 calling summary

first 6 arguments: %rdi, %rsi, %rdx, %rcx, %r8, %r9

additional arguments: push on stack

return address: push on stack

- `call`, `ret` instructions assume this

return value: %rax

caller-saved registers

functions may freely trash these

return value register `%rax`

argument registers:
 `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`

%r11

MMX/SSE/AVX registers: `%xmm0-15`, etc.

floating point stack: `%st(0)-%st(7)`

condition codes (used by `jne`, etc.)

callee-saved registers

functions must preserve these

`%rsp` (stack pointer), `%rbp` (frame pointer, maybe)

`%r12`–`%r15`

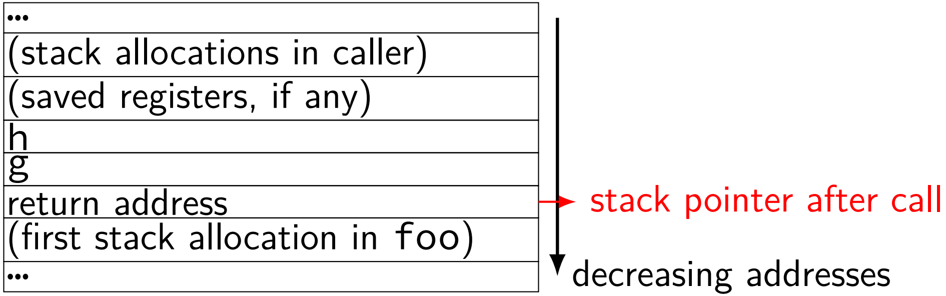
caller/callee-saved

```
foo:
    pushq %r12 // r12 is caller-saved
    ... use r12 ...
    popq %r12
    ret

...
other_function:
    pushq %r11 // r11 is caller-saved
    ...
    callq foo
    popq %r11
```

the call stack

```
foo(a,b,c,d,e,f,g,h);
```


...
(stack allocations in caller)
(saved registers, if any)
h
g
return address
(first stack allocation in foo)
...
stack pointer after call
decreasing addresses

calling convention example

```
int foo(int a, int b, int c, int d, int e, int f, int g, int h
...
foo(1, 2, 3, 4, 5, 6, 7, 8);
```

```
pushq    $8
pushq    $7
movl     $6, %r9d
movl     $5, %r8d
movl     $4, %ecx
movl     $3, %edx
movl     $2, %esi
movl     $1, %edi
call     foo
/* return value in %eax */
```

floating point operations

x86 has two ways to do floating point

method one — legacy: x87 floating point instructions
- still common in 32-bit x86

method two — SSE instructions
- work more like what you expect

x87 floating point stack

x87: 8 floating point registers

`%st(0)` through `%st(7)`

arranged as a stack of registers

example: `fld 0(%rbx)`

	before	after
`st(0)`	5.0	(value from memory at `%rbx`)
`st(1)`	6.0	5.0
`st(1)`	7.0	6.0
…	…	…
`st(6)`	10.0	9.0
`st(7)`	11.0	10.0

x87

not going to talk about x87 more in this course

essentially obsolete with 64-bit x86

SSE registers

SSE and SSE2 extensions brought vector instructions

```
numbers: .float 1 .float 2 .float 3. float 4
ones:    .float 1 .float 3 .float 5 .float 7
result:  .float 0 .float 0 .float 0 .float 0
...
movps numbers, %xmm0
movps ones, %xmm1
addps %xmm1, %xmm0
movps %xmm0, result
/* result contains: 1+1=2,2+3=5,3+5=8,4+7=11 */
```

SSE registers

SSE and SSE2 extensions brought vector instructions

```
numbers: .float 1 .float 2 .float 3. float 4
ones:    .float 1 .float 3 .float 5 .float 7
result:  .float 0 .float 0 .float 0 .float 0
...
movps numbers, %xmm0
movps ones, %xmm1
addps %xmm1, %xmm0
movps %xmm0, result
/* result contains: 1+1=2,2+3=5,3+5=8,4+7=11 */
```

array of 4 floats

SSE registers

SSE and SSE2 extensions brought vector instructions

```
numbers: .float 1 .float 2 .float 3. float 4
ones:    .float 1 .float 3 .float 5 .float 7
result:  .float 0 .float 0 .float 0 .float 0
...
movps numbers, %xmm0
movps ones, %xmm1
addps %xmm1, %xmm0
movps %xmm0, result
/* result contains: 1+1=2,2+3=5,3+5=8,4+7=11 */
```

move packed single
(single-precision float)

SSE registers

SSE and SSE2 extensions brought vector instructions

```
numbers: .float 1 .float 2 .float 3. float 4
ones:    .float 1 .float 3 .float 5 .float 7
result:  .float 0 .float 0 .float 0 .float 0
...
movps numbers, %xmm0
movps ones, %xmm1
addps %xmm1, %xmm0
movps %xmm0, result
/* result contains: 1+1=2,2+3=5,3+5=8,4+7=11 */
```

add packed single
(single-precision float)

XMM registers

`%xmm0` through `%xmm15` (`%xmm8` on 32-bit)

each holds 128-bits —

- 32-bit floating point values (`addps`, etc.)
- 64-bit floating point values (`addpd`, etc.)
- 64/32/16/8-bit integers (`paddq/d/w/b`, etc.)
- a 32-bit floating point value, 96 unused bits (`addss`, `movss`, etc.)
- a 64-bit floating point value, 64 unused bits (`addsd`, `movsd`, etc.)

more recently: `%ymm0` through `%ymm15` (256-bit, “AVX”)

- overlap with `%xmm`*X* registers

XMM registers

`%xmm0` through `%xmm15` (`%xmm8` on 32-bit)

each holds 128-bits —

- 32-bit floating point values (`addps`, etc.)
- 64-bit floating point values (`addpd`, etc.)
- 64/32/16/8-bit integers (`paddq/d/w/b`, etc.)
- a 32-bit floating point value, 96 unused bits (`addss`, `movss`, etc.)
- a 64-bit floating point value, 64 unused bits (`addsd`, `movsd`, etc.)

more recently: `%ymm0` through `%ymm15` (256-bit, “AVX”)

- overlap with `%xmm`*X* registers

FP example

```
multiplyEachElementOfArray:
/* %rsi = array, %rdi length,
   %xmm0 multiplier */
loop:   test %rdi, %rdi
        je done
        movss (%rsi), %xmm1
        mulss %xmm0, %xmm1
        movss %xmm1, (%rsi)
        subq $1, %rdi
        addq $4, %rsi
        jmp loop
done:   ret
```

addressing modes (1)

```
$constant

displacement(%base, %index, scale)
    displacement (absolute)
    displacement(%base)
    displacement(,%index, scale)
```

(= displacement + base + index $\times$ scale)

addressing modes (2)

`displacement(%rip)` (64-bit only)

```
thing: .quad 42
...
movq thing(%rip), %rax
```

- encoded as offset from address of next instruction
- (normally: label encoded as 32 or 64-bit address)
- helps relocatable code

addressing modes (3)

`jmp *%rax` (or `call`)

- Intel syntax: `jmp RAX`
- what you'd expect to be `jmp (%rax)`

`jmp *(%rax)`

- read value from memory at RAX
- PC becomes location in that value
- Intel syntax: `jmp [RAX]`

`jmp *(%rax,%rbx,8)`

where is the jump?

```
0xA0000: lea 0x1234(%rip), %rax   # 0xA123b
    (Intel syntax: LEA RAX, [RIP + 0x1234])
0xA0007: add %rbx, %rax
    (Intel syntax: ADD RAX, RBX)
0xA000A: jmp *(%rax)
    (Intel syntax: JMP [RAX])
...
0xA123B: 0x30000 (64-bit value)
....
0xB123B: 0x40000 (64-bit-value)
...
0xB124B: 0x50000 (64-bit value)
```

If %rax initially contains 0x40, then the instruction executed after the jump is at address _____.

ENDBR64?

partial output of *objdump –disassemble*:

```
0000000000001060 <main>:
    1060:       f3 0f 1e fa             endbr64
    1064:       50                      push   %rax
    1065:       48 8d 3d 98 0f 00 00    lea    0xf98(%rip),%rd
    106c:       e8 df ff ff ff          callq  1050 <puts@plt>
    1071:       31 c0                   xor    %eax,%eax
    1073:       5a                      pop    %rdx
    1074:       c3                      retq
```

endbr64: (currently) nop instruction that marks destination of branches

why? — we'll explain (much) later

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

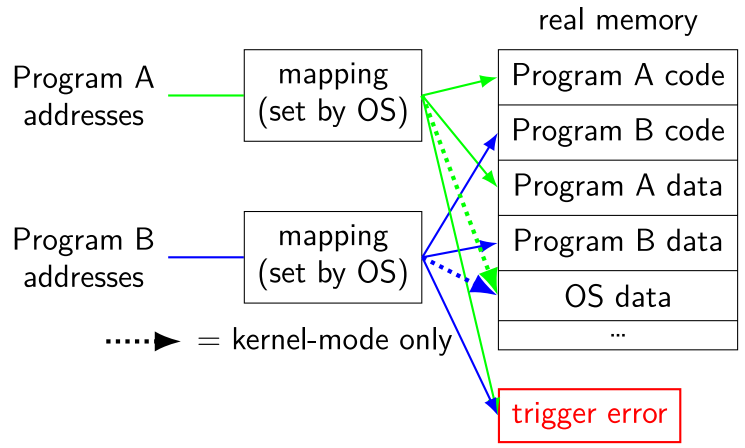
string comparison instructions

`rep movsb` is still recommended on modern Intel

- special-cased in processor?

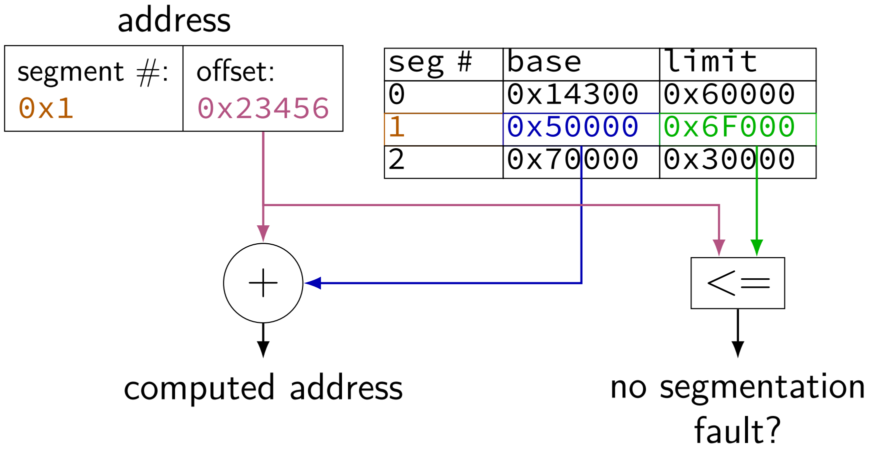
recall(?): virtual memory

illuision of dedicated memory



segmentation

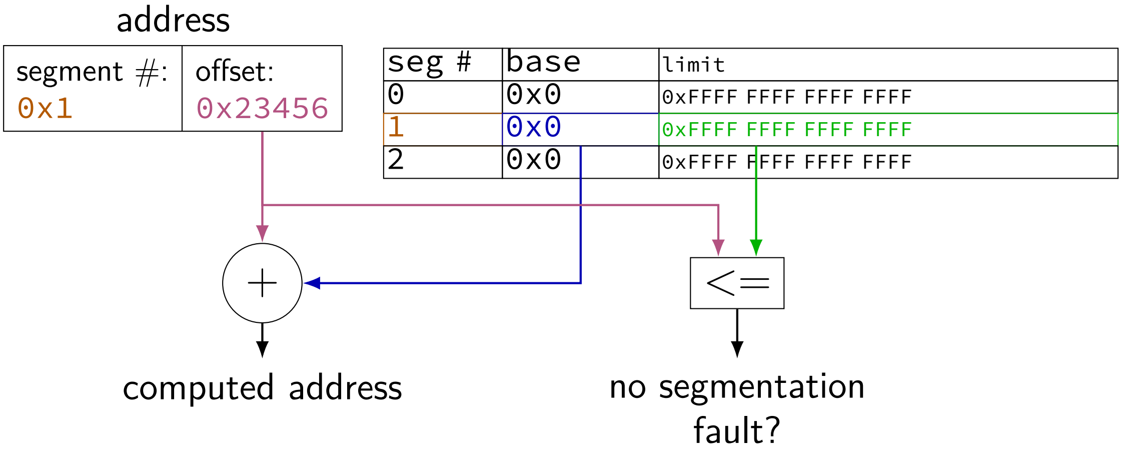
before virtual memory, there was segmentation



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

segmentation

before virtual memory, there was segmentation



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

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
```

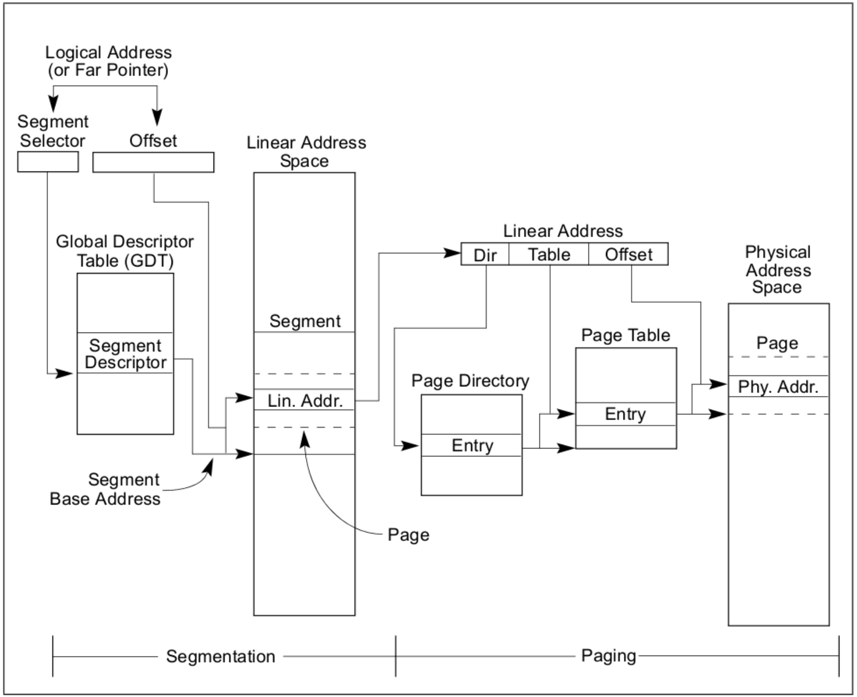


Figure 3-1. Segmentation and Paging

Figure: Intel manuals, Vol 3A

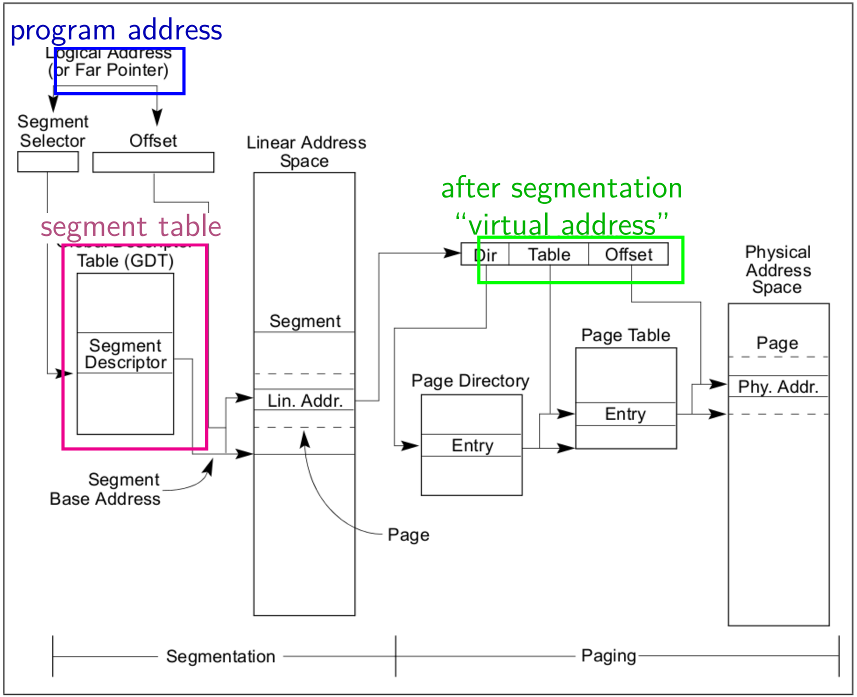


Figure 3-1. Segmentation and Paging

Figure: Intel manuals, Vol 3A

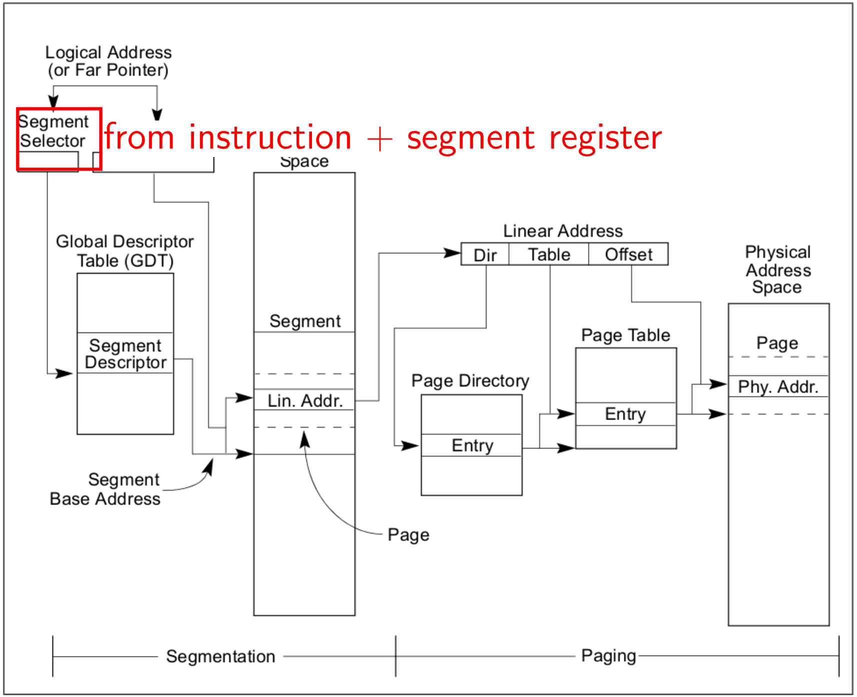
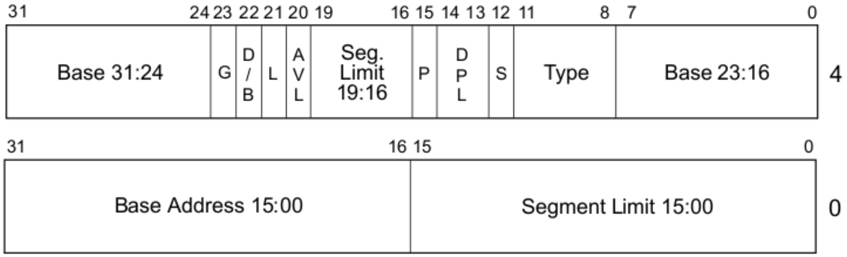


Figure 3-1. Segmentation and Paging

Figure: Intel manuals, Vol 3A

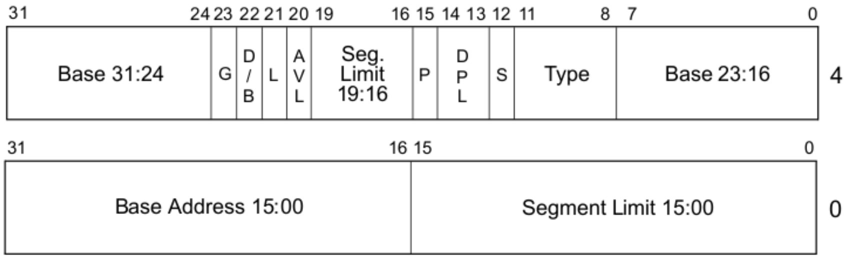
x86 segment descriptor



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

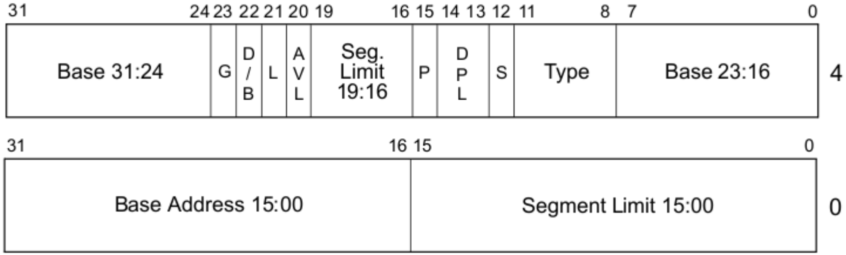
x86 segment descriptor



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)
$\boxed{\text{DPL — Descriptor privilege level}}$	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

thread-local storage

Linux, Windows use %fs, %gs for thread-local storage

variables that have different values in each thread

e.g. for program using multiple cores
to track different values for each core

TLS example (read) (C)

```
#include <threads.h>
thread_local int thread_local_value = 0;
int get_thread_local() {
    return thread_local_value;
}
```

```
0000000000001149 <get_thread_local>:
    1149:       f3 0f 1e fa
                    endbr64
    114d:       64 8b 04 25 fc ff ff ff
                    mov    %fs:0xfffffffffffffffc,%eax
    1155:       c3
                    retq
```

```
TLS off    0x0000000000002df0 vaddr 0x0000000000003df0 paddr 0x0000000000003df0 al
    filesz 0x0000000000000000 memsz 0x0000000000000004 flags r--
```

TLS example (write) (C)

```c
#include <threads.h>
thread_local int thread_local_value = 0;
void set_thread_local(int new_value) {
    thread_local_value = new_value;
}
```

```
0000000000001156 <set_thread_local>:
    1156:       f3 0f 1e fa
                    endbr64
    115a:       64 89 3c 25 fc ff ff ff
                    mov    %edi,%fs:0xfffffffffffffffc
    1162:       c3
                    retq
```

backup slides

floating point calling convention

use `%xmm` registers in order

note: variadic functions

variable number of arguments

- `printf`, `scanf`, …
- see `man stdarg`

same as usual

…but `%rax` contains number of `%xmm` used

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
```