

linking / assembly

last time

course intro / logistics / preview

layers of abstraction

C/etc. → asm → machine code → HDL → gates

powers of two

processor to memory bus

endianness

program memory layout

linking (started)

department account notes

have account from last semester — same password

no new email sent

password recovery:

<https://www.cs.virginia.edu/PasswordReset>

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```



hello.s

```
.text
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call   puts
    xor     %eax, %eax
    add    $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call   puts
    xor     %eax, %eax
    add    $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.s (Intel syntax)

```
.text
main:
    sub RSP, 8
    mov RDI, .Lstr
    call puts
    xor EAX, EAX
    add RSP, 8
    ret

.data
.Lstr: .string "Hello, World!"
```

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

Linux x86-64
calling convention:
stack addr. must be
multiple of 16

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call   puts
    xor     %eax, %eax
    add    $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

sets eax to 0
(shorter machine
code than mov)

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call   puts
    xor     %eax, %eax
    add     $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call   puts
    xor     %eax, %eax
    add    $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.o

```
text (code) segment:
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
data segment:
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call   puts
    xor    %eax, %eax
    add    $8, %rsp
    ret
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

```
.data
.Lstr: .string "Hello, World!"
```

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

relocations:

take 0s at and replace with
text, byte 6 () data segment, byte 0
text, byte 11 () address of puts

```
.data
.Lstr: .string "Hello, World!"
```

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

relocations:

take 0s at and replace with
text, byte 6 () data segment, byte 0
text, byte 11 () address of puts

symbol table:

main text byte 0

```
.data
.Lstr: .string "Hello, World!"
```

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

hello.s

```
.text
main:
    sub    $8, %rsp
    mov    $.Lstr, %rdi
    call   puts
    xor    %eax, %eax
    add    $8, %rsp
    ret

.data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

relocations:

take 0s at	and replace with
text, byte 6 ()	data segment, byte 0
text, byte 11 ()	address of puts

symbol table:

main	text byte 0
------	-------------

+ stdio.o

hello.exe

(actually binary, but shown as hexadecimal) ...

```
48 83 EC 08 BF A7 02 04 00
E8 08 4A 04 00 31 C0 48
83 C4 08 C3 ...
...(code from stdio.o) ...
48 65 6C 6C 6F 2C 20 57 6F
72 6C 00 ...
...(data from stdio.o) ...
```

hello.s

```
.LC0:      .section          .rodata.str1.1,"aMS",@progbt
           .string "Hello, World!"
           .text
           .globl  main

main:
           subq    $8, %rsp
           movl    $.LC0, %edi
           call    puts
           movl    $0, %eax
           addq    $8, %rsp
           ret
```

exercise (1)

main.c:

```
1  #include <stdio.h>
2  void sayHello(void) {
3      puts("Hello, World!");
4  }
5  int main(void) {
6      sayHello();
7  }
```

Which files contain the **memory address** of sayHello?

- | | |
|--------------------------|-------------------|
| A. main.s (assembly) | D. B and C |
| B. main.o (object) | E. A, B and C |
| C. main.exe (executable) | F. something else |

exercise (2)

main.c:

```
1  #include <stdio.h>
2  void sayHello(void) {
3      puts("Hello, World!");
4  }
5  int main(void) {
6      sayHello();
7  }
```

Which files contain the **literal ASCII string** of Hello, World!?

- | | |
|--------------------------|-------------------|
| A. main.s (assembly) | D. B and C |
| B. main.o (object) | E. A, B and C |
| C. main.exe (executable) | F. something else |

relocation types

machine code doesn't always use addresses as is

“call function 4303 bytes later”

linker needs to compute “4303”

extra field on relocation list

dynamic linking (very briefly)

dynamic linking — done **when application is loaded**

idea: don't have N copies of `printf` on disk

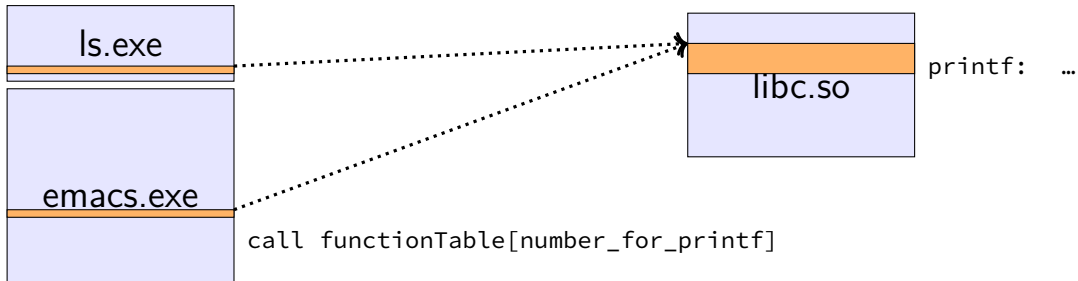
other type of linking: *static* (`gcc -static`)

load executable file + its libraries into memory when app starts

often extra indirection:

call `functionTable[number_for_printf]`

linker fills in `functionTable` instead of changing calls



ldd /bin/ls

```
$ ldd /bin/ls
linux-vdso.so.1 => (0x00007ffcca9d8000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1
(0x00007f851756f000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007f85171a5000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3
(0x00007f8516f35000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2
(0x00007f8516d31000)
/lib64/ld-linux-x86-64.so.2 (0x00007f8517791000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
(0x00007f8516b14000)
```

layers of abstraction

`x += y`

“Higher-level” language: C

`add %rbx, %rax`

Assembly: X86-64

`60 03SIXTEEN`

Machine code: Y86

Hardware Design Language: HCLRS

Gates / Transistors / Wires / Registers

AT&T versus Intel syntax by example

<code>movq \$42, (%rbx)</code>	
	<code>mov QWORD PTR [rbx], 42</code>
<code>subq %rax, %r8</code>	
	<code>sub r8, rax</code>
<code>movq \$42, 100(%rbx,%rcx,4)</code>	
	<code>mov QWORD PTR [rbx+rcx*4+100], 42</code>
<code>jmp *%rax</code>	
	<code>jmp rax</code>
<code>jmp *1000(%rax,%rbx,8)</code>	
	<code>jmp QWORD PTR [RAX+RBX*8+1000]</code>

AT&T versus Intel syntax (1)

AT&T syntax:

```
movq $42, (%rbx)
```

Intel syntax:

```
mov QWORD PTR [rbx], 42
```

effect (pseudo-C):

```
memory[rbx] <- 42
```

AT&T syntax example (1)

```
movq $42, (%rbx)  
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with \$

registers start with %

q ('quad') indicates length (8 bytes)

l: 4; w: 2; b: 1

sometimes can be omitted

AT&T syntax example (1)

```
movq $42, (%rbx)  
// memory[rbx] ← 42
```

destination last

()s represent value **in memory**

constants start with \$

registers start with %

q ('quad') indicates length (8 bytes)

l: 4; w: 2; b: 1

sometimes can be omitted

AT&T syntax example (1)

```
movq $42, (%rbx)  
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with \$

registers start with %

q ('quad') indicates length (8 bytes)

l: 4; w: 2; b: 1

sometimes can be omitted

AT&T syntax example (1)

```
movq $42, (%rbx)  
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with \$

registers start with %

q ('quad') indicates length (8 bytes)

l: 4; w: 2; b: 1

sometimes can be omitted

AT&T syntax example (1)

```
movq $42, (%rbx)  
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with \$

registers start with %

q ('quad') indicates **length** (8 bytes)

l: 4; w: 2; b: 1

sometimes can be omitted

AT&T versus Intel syntax (2)

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 versus Intel syntax (2)

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 versus Intel syntax (2)

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 versus Intel syntax (2)

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: addressing

100(%rbx): memory[rbx + 100]

100(%rbx,8): memory[rbx * 8 + 100]

100(,%rbx,8): memory[rbx * 8 + 100]

100(%rcx,%rbx,8):
memory[rcx + rbx * 8 + 100]

100:
memory[100]

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

AT&T versus Intel syntax (3)

$r8 \leftarrow r8 - rax$

AT&T syntax: **subq** %rax, %r8

Intel syntax: **sub** r8, rax

same for **cmpq**

AT&T syntax: addresses

```
addq 0x1000, %rax
```

```
// Intel syntax: add rax, QWORD PTR [0x1000]
```

```
// rax ← rax + memory[0x1000]
```

```
addq $0x1000, %rax
```

```
// Intel syntax: add rax, 0x1000
```

```
// rax ← rax + 0x1000
```

no \$ — probably memory address

AT&T syntax in one slide

destination **last**

() means value **in memory**

`disp(base, index, scale)` same as
`memory[disp + base + index * scale]`

omit `disp` (defaults to 0)

and/or omit `base` (defaults to 0)

and/or `scale` (defaults to 1)

\$ means constant

plain number/label means value **in memory**

extra detail: computed jumps

```
jmpq *%rax
```

```
// Intel syntax: jmp RAX
```

```
// goto RAX
```

```
jmpq *1000(%rax,%rbx,8)
```

```
// Intel syntax: jmp QWORD PTR[RAX+RBX*8+1000]
```

```
// read address from memory at RAX + RBX * 8 + 1000
```

```
// go to that address
```

AT&T versus Intel syntax by example

<code>movq \$42, (%rbx)</code>	
	<code>mov QWORD PTR [rbx], 42</code>
<code>subq %rax, %r8</code>	
	<code>sub r8, rax</code>
<code>movq \$42, 100(%rbx,%rcx,4)</code>	
	<code>mov QWORD PTR [rbx+rcx*4+100], 42</code>
<code>jmp *%rax</code>	
	<code>jmp rax</code>
<code>jmp *1000(%rax,%rbx,8)</code>	
	<code>jmp QWORD PTR [RAX+RBX*8+1000]</code>

swap

swap (AT&T syntax)

```
// swap(long *rdi,  
//      long *rsi)  
swap:  
    movq (%rdi), %rax  
    movq (%rsi), %rdx  
    movq %rdx, (%rdi)  
    movq %rax, (%rsi)  
    ret
```

swap

swap (AT&T syntax)

```
// swap(long *rdi,  
//      long *rsi)  
swap:  
    movq (%rdi), %rax  
    movq (%rsi), %rdx  
    movq %rdx, (%rdi)  
    movq %rax, (%rsi)  
    ret
```

swap (Intel syntax)

```
swap:  
    mov RAX, QWORD PTR [RDI]  
    mov RDX, QWORD PTR [RSI]  
    mov QWORD PTR [RDI], RDX  
    mov QWORD PTR [RSI], RAX  
    ret
```

swap

swap (AT&T syntax)

```
// swap(long *rdi,  
//      long *rsi)  
swap:  
    movq (%rdi), %rax  
    movq (%rsi), %rdx  
    movq %rdx, (%rdi)  
    movq %rax, (%rsi)  
    ret
```

as pseudocode

```
swap:  
    RAX ← memory[RDI (arg 1)]  
    RDX ← memory[RSI (arg 2)]  
    memory[RDI (arg 1)] ← RDX  
    memory[RSI (arg 2)] ← RAX  
    return
```

swap

swap (AT&T syntax)

```
// swap(long *rdi,  
//      long *rsi)  
swap:  
    movq (%rdi), %rax  
    movq (%rsi), %rdx  
    movq %rdx, (%rdi)  
    movq %rax, (%rsi)  
    ret
```

registers

%rax	???
%rdx	???
%rdi	0x04000
%rsi	0x04030
%rsp	0xEFFF8
...	...

memory

address	value
0x00000	0xFFFF3
0x00008	0x32123
...	...
0x04000	0x99999
0x04008	0x00002
...	...
0x04028	0x00090
0x04030	0x77777
0x04038	0x00078
...	...

swap

swap (AT&T syntax)

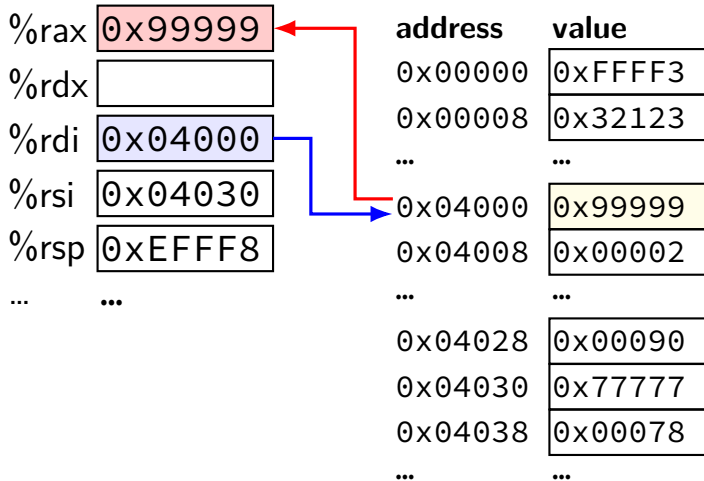
```
// swap(long *rdi,  
//      long *rsi)  
swap:  
    movq (%rdi), %rax  
    movq (%rsi), %rdx  
    movq %rdx, (%rdi)  
    movq %rax, (%rsi)  
    ret
```

registers

%rax	0x99999
%rdx	
%rdi	0x04000
%rsi	0x04030
%rsp	0xEFF8
...	...

memory

address	value
0x00000	0xFFFF3
0x00008	0x32123
...	...
0x04000	0x99999
0x04008	0x00002
...	...
0x04028	0x00090
0x04030	0x77777
0x04038	0x00078
...	...



swap

swap (AT&T syntax)

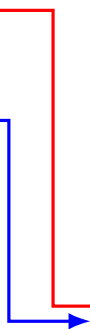
```
// swap(long *rdi,  
//      long *rsi)  
swap:  
    movq (%rdi), %rax  
    movq (%rsi), %rdx  
    movq %rdx, (%rdi)  
    movq %rax, (%rsi)  
    ret
```

registers

%rax	0x99999
%rdx	0x77777
%rdi	0x04000
%rsi	0x04030
%rsp	0xEFF8
...	...

memory

address	value
0x00000	0xFFFF3
0x00008	0x32123
...	...
0x04000	0x99999
0x04008	0x00002
...	...
0x04028	0x00090
0x04030	0x77777
0x04038	0x00078
...	...



swap

swap (AT&T syntax)

```
// swap(long *rdi,  
//      long *rsi)  
swap:  
    movq (%rdi), %rax  
    movq (%rsi), %rdx  
    movq %rdx, (%rdi)  
    movq %rax, (%rsi)  
    ret
```

registers

%rax	0x999999
%rdx	0x777777
%rdi	0x04000
%rsi	0x04030
%rsp	0xEFFF8
...	...

memory

address	value
0x00000	0xFFFF3
0x00008	0x32123
...	...
0x04000	0x999990x77
0x04008	0x00002
...	...
0x04028	0x00090
0x04030	
0x04038	0x00078
...	...

swap

swap (AT&T syntax)

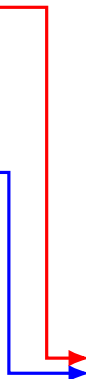
```
// swap(long *rdi,  
//      long *rsi)  
swap:  
    movq (%rdi), %rax  
    movq (%rsi), %rdx  
    movq %rdx, (%rdi)  
    movq %rax, (%rsi)  
    ret
```

registers

%rax	0x99999
%rdx	0x77777
%rdi	0x04000
%rsi	0x04030
%rsp	0xEFF8
...	...

memory

address	value
0x00000	0xFFFF3
0x00008	0x32123
...	...
0x04000	0x999990x77
0x04008	0x00002
...	...
0x04028	0x00090
0x04030	0x99999
0x04038	0x00078
...	...



swap

swap (AT&T syntax)

```
// swap(long *rdi,  
//      long *rsi)  
swap:  
    movq (%rdi), %rax  
    movq (%rsi), %rdx  
    movq %rdx, (%rdi)  
    movq %rax, (%rsi)  
    ret
```

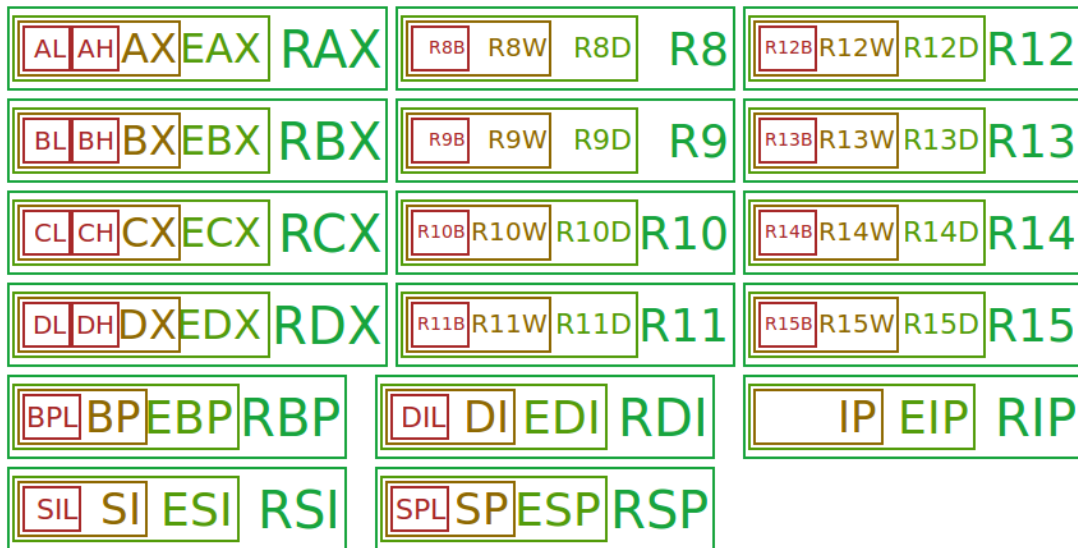
registers

%rax	0x99999
%rdx	0x77777
%rdi	0x04000
%rsi	0x04030
%rsp	0xEFF8
...	...

memory

address	value
0x00000	0xFFFF3
0x00008	0x32123
...	...
0x04000	0x77777
0x04008	0x00002
...	...
0x04028	0x00090
0x04030	0x99999
0x04038	0x00078
...	...

recall: x86-64 general purpose registers



overlapping registers (1)

setting 32-bit registers — clears corresponding 64-bit register

```
movq $0xFFFFFFFFFFFFFFFF, %rax
```

```
movl $0x1, %eax
```

%rax is 0x1 (not 0xFFFFFFFF00000001)

overlapping registers (2)

setting 8/16-bit registers: don't clear 64-bit register

```
movq $0xFFFFFFFFFFFFFFFF, %rax  
movb $0x1, %al
```

%rax is 0xFFFFFFFFFFFFFFFF01

labels (1)

labels represent addresses

labels (2)

```
addq string, %rax
// intel syntax: add rax, QWORD PTR [label]
// rax ← rax + memory[address of "a string"]
addq $string, %rax
// intel syntax: add rax, OFFSET label
// rax ← rax + address of "a string"
string: .ascii "a_string"
```

addq label: read value at the address

addq \$label: use address as an integer constant

on LEA

LEA = **L**oad **E**ffective **A**ddress

effective address = computed address for memory access

syntax looks like a **mov** from memory, but...

skips the memory access — just uses the address
(sort of like & operator in C?)

`leaq 4(%rax), %rax` $\approx$ `addq $4, %rax`

on LEA

LEA = **L**oad **E**ffective **A**ddress

effective address = computed address for memory access

syntax looks like a **mov** from memory, but...

skips the memory access — just uses the address
(sort of like & operator in C?)

`leaq 4(%rax), %rax` $\approx$ `addq $4, %rax`

“address of memory[$\text{rax} + 4$]” = $\text{rax} + 4$

LEA tricks

```
leaq (%rax,%rax,4), %rax
```

$\text{rax} \leftarrow \text{rax} \times 5$

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

```
leaq (%rbx,%rcx), %rdx
```

$\text{rdx} \leftarrow \text{rbx} + \text{rcx}$

```
rdx ← address-of(memory[rbx + rcx])
```

exercise: what is this function?

mystery:

```
    leal 0(,%rdi,8), %eax  
    subl %edi, %eax  
    ret
```

```
int mystery(int arg) { return ...; }
```

- A. $\text{arg} * 9$
- B. $-\text{arg} * 9$
- C. $\text{arg} * 8$
- D. $-\text{arg} * 7$
- E. none of these
- F. it has a different prototype

exercise: what is this function?

mystery:

```
    leal 0(,%rdi,8), %eax
    subl %edi, %eax
    ret
```

```
int mystery(int arg) { return ...; }
```

- A. $\text{arg} * 9$
- B. $-\text{arg} * 9$
- C. $\text{arg} * 8$
- D. $-\text{arg} * 7$
- E. none of these
- F. it has a different prototype

Linux x86-64 calling convention

registers for first 6 arguments:

%rdi (or %edi or %di, etc.), then
%rsi (or %esi or %si, etc.), then
%rdx (or %edx or %dx, etc.), then
%rcx (or %ecx or %cx, etc.), then
%r8 (or %r8d or %r8w, etc.), then
%r9 (or %r9d or %r9w, etc.)

rest on stack

return value in %rax

don't memorize: Figure 3.28 in book

x86-64 calling convention example

```
int foo(int x, int y, int z) { return 42; }  
...  
    foo(1, 2, 3);  
...  
...  
    // foo(1, 2, 3)  
    movl $1, %edi  
    movl $2, %esi  
    movl $3, %edx  
    call foo // call pushes address of next instruction  
             // then jumps to foo  
...  
foo:  
    movl $42, %eax  
    ret
```

call/ret

call:

push address of **next instruction** on the stack

ret:

pop address from stack; jump

callee-saved registers

functions **must preserve** these

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

`%r12-%r15`

caller/callee-saved

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

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

question

```
pushq $0x1  
pushq $0x2  
addq $0x3, 8(%rsp)  
popq %rax  
popq %rbx
```

What is value of %rax and %rbx after this?

- a. %rax = 0x2, %rbx = 0x4
- b. %rax = 0x5, %rbx = 0x1
- c. %rax = 0x2, %rbx = 0x1
- d. the snippet has invalid syntax or will crash
- e. more information is needed
- f. something else?

on %rip

%rip (Instruction **P**ointer) = address of next instruction

```
movq 500(%rip), %rax
```

$\text{rax} \leftarrow \text{memory}[\text{next instruction address} + 500]$

on %rip

%rip (Instruction **P**ointer) = address of next instruction

movq 500(%rip), %rax

rax $\leftarrow$ memory[next instruction address + 500]

label(%rip) $\approx$ label

different ways of writing address of label in machine code
(with %rip — relative to next instruction)

things we won't cover (today)

floating point; vector operations (multiple values at once)

special registers: %xmm0 through %xmm15

segmentation (special registers: %ds, %fs, %gs, ...)

lots and lots of instructions

goto

```
for (...) {  
    for (...) {  
        if (thingAt(i, j)) {  
            goto found;  
        }  
    }  
}  
printf("not found!\n");  
return;  
found:  
printf("found!\n");
```

goto

```
for (...) {  
    for (...) {  
        if (thingAt(i, j)) {  
            goto found;  
        }  
    }  
}  
printf("not found!\n");  
return;  
found:  
printf("found!\n");
```

assembly:
jmp found

assembly:
found:

if-to-assembly (1)

```
if (b >= 42) {  
    a += 10;  
} else {  
    a *= b;  
}
```

if-to-assembly (1)

```
if (b >= 42) {  
    a += 10;  
} else {  
    a *= b;  
}
```

```
        if (b < 42) goto after_then;  
        a += 10;  
        goto after_else;  
after_then: a *= b;  
after_else:
```

if-to-assembly (2)

```
if (b >= 42) {  
    a += 10;  
} else {  
    a *= b;  
}
```

```
// a is in %rax, b is in %rbx  
    cmpq $42, %rbx    // computes rbx - 42 to 0  
                        // i.e compare rbx to 42  
    jl  after_then    // jump if rbx - 42 < 0  
                        // AKA rbx < 42  
    addq $10, %rax    // a += 1  
    jmp after_else  
after_then:  
    imulq %rbx, %rax  // rax = rax * rbx  
after_else:
```

do-while-to-assembly (1)

```
int x = 99;  
do {  
    foo()  
    x--;  
} while (x >= 0);
```

do-while-to-assembly (1)

```
int x = 99;  
do {  
    foo()  
    x--;  
} while (x >= 0);
```

```
    int x = 99;  
start_loop:  
    foo()  
    x--;  
    if (x >= 0) goto start_loop;
```

do-while-to-assembly (2)

```
int x = 99;  
do {  
    foo()  
    x--;  
} while (x >= 0);
```

```
    movq $99, %r12 // register for x  
start_loop:  
    call foo  
    subq $1, %r12  
    cmpq $0, %r12  
    // computes r12 - 0 = r12  
    jge start_loop // jump if r12 - 0 >= 0
```

condition codes

x86 has **condition codes**

set by (almost) all arithmetic instructions
addq, subq, imulq, etc.

store info about **last arithmetic result**
was it zero? was it negative? etc.

condition codes and jumps

`jg`, `jle`, etc. read condition codes

named based on interpreting **result of subtraction**

0: equal; negative: less than; positive: greater than

condition codes example (1)

```
movq $-10, %rax
movq $20, %rbx
subq %rax, %rbx // %rbx - %rax = 30
// result > 0: %rbx was > %rax
jle foo // not taken; 30 > 0
```

condition codes and cmpq

“last arithmetic result”???

then what is cmp, etc.?

cmp does subtraction (but doesn't store result)

similar test does bitwise-and

testq %rax, %rax — result is %rax

condition codes example (2)

```
movq $-10, %rax  
movq $20, %rbx  
cmpq %rax, %rbx  
jle foo // not taken; %rbx - %rax > 0
```

omitting the cmp

```
    movq $99, %r12 // register for x
start_loop:
    call foo
    subq $1, %r12
    cmpq $0, %r12
    // compute r12 - 0 + sets cond. codes
    jge  start_loop // r12 >= 0?
                     // or result >= 0?
```

```
    movq $99, %r12 // register for x
start_loop:
    call foo
    subq $1, %r12
    // new r12 = old r12 - 1 + sets cond. codes
    jge  start_loop // old r12 >= 1?
                     // or result >= 0?
```

condition codes example (3)

```
movq $-10, %rax
movq $20, %rbx
subq %rax, %rbx
jle  foo // not taken, %rbx - %rax > 0 -> %rbx
```

```
movq $20, %rbx
addq $-20, %rbx
je   foo // taken, result is 0
      // x - y = 0 -> x = y
```

what sets condition codes

most instructions that compute something **set condition codes**

some instructions **only** set condition codes:

cmp ~ **sub**

test ~ **and** (bitwise and — later)

testq %rax, %rax — result is %rax

some instructions don't change condition codes:

lea, **mov**

control flow: **jmp**, **call**, **ret**, **jle**, etc.

condition codes examples (4)

```
movq $20, %rbx  
addq $-20, %rbx // result is 0  
movq $1, %rax // irrelevant  
je   foo // taken, result is 0
```


backup slides

authoritative source (1)



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

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

authoritative source (2)

System V Application Binary Interface

AMD64 Architecture Processor Supplement

Draft Version 0.99.7

Edited by
Michael Matz¹, Jan Hubička², Andreas Jaeger³, Mark Mitchener

November 17, 2014

objdump -sx test.o (Linux) (1)

```
test.o:      file format elf64-x86-64
test.o
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000
```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000000	0000000000000000	0000000000000000	00000040	2**0
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
1	.data	00000000	0000000000000000	0000000000000000	00000040	2**0
	CONTENTS, ALLOC, LOAD, DATA					
2	.bss	00000000	0000000000000000	0000000000000000	00000040	2**0
	ALLOC					
3	.rodata.str1.1	0000000e	0000000000000000	0000000000000000	00000040	2**0
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
4	.text.startup	00000014	0000000000000000	0000000000000000	0000004e	2**0
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE					
5	.comment	0000002b	0000000000000000	0000000000000000	00000062	2**0
	CONTENTS, READONLY					
6	.note.GNU-stack	00000000	0000000000000000	0000000000000000	0000008d	2**0
	CONTENTS, READONLY					
7	.eh_frame	00000030	0000000000000000	0000000000000000	00000090	2**3
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA					

objdump -sx test.o (Linux) (2)

SYMBOL TABLE:

```
0000000000000000 l    df *ABS*  0000000000000000 test.c
0000000000000000 l    d  .text 0000000000000000 .text
0000000000000000 l    d  .data 0000000000000000 .data
0000000000000000 l    d  .bss  0000000000000000 .bss
0000000000000000 l    d  .rodata.str1.1 0000000000000000 .rodata.str1.1
0000000000000000 l    d  .text.startup 0000000000000000 .text.startup
0000000000000000 l    d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame 0000000000000000 .eh_frame
0000000000000000 l    .rodata.str1.1 0000000000000000 .LC0
0000000000000000 l    d  .comment 0000000000000000 .comment
0000000000000000 g    F  .text.startup 0000000000000014 main
0000000000000000      *UND* 0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000      *UND* 0000000000000000 puts
```

columns:

memory address (not yet assigned, so 0)

flags: l=local, g=global, F=function, ...

section (.text, .data, .bss, ...)

offset in section

name of symbol

objdump -sx test.o (Linux) (3)

RELOCATION RECORDS FOR [.text.startup]:

OFFSET	TYPE	VALUE
0000000000000003	R_X86_64_PC32	.LC0-0x0000000000000004
000000000000000c	R_X86_64_PLT32	puts-0x0000000000000004

RELOCATION RECORDS FOR [.eh_frame]:

OFFSET	TYPE	VALUE
0000000000000020	R_X86_64_PC32	.text.startup

Contents of section .rodata.str1.1:

0000 48656c6c 6f2c2057 6f726c64 2100	Hello, World!.
--------------------------------------	----------------

Contents of section .text.startup:

0000 488d3d00 00000048 83ec08e8 00000000	H.=....H.....
0010 31c05ac3	1.Z.

Contents of section .comment:

0000 00474343 3a202855 62756e74 7520372e	.GCC: (Ubuntu 7.
0010 332e302d 32377562 756e7475 317e3138	3.0-27ubuntu1~18
0020 2e303429 20372e33 2e3000	.04) 7.3.0.

Contents of section .eh_frame:

0000 14000000 00000000 017a5200 01781001	zR..x..
0010 1b0c0708 90010000 14000000 1c000000	
0020 00000000 14000000 004b0e10 480e0800	K..H...