Brief Assembly Refresher

Changelog

Changes made in this version not seen in first lecture:

- 23 Jan 2018: if-to-assembly `if (...) goto` needed $b < 42$
- 23 Jan 2018: caller/callee-saved: correct comment about which register is callee-saved
- 23 Jan 2018: AT&T syntax: addresses: two more examples; correct $100+$ on last new one

last time

processors $\leftrightarrow$ memory, I/O devices

- processor: send addresses (+ values, sometimes)
- memory: reply with values (or store value)
- some addresses correspond to I/O devices
- type of value read: why did processor read it?

endianness: little = least address is least significant

- little endian: `0x12 34`: `0x34` at address $x + 0$
- big endian: `0x12 34`: `0x12` at address $x + 0$

object files and linking

- relocations: "fill in the blank" with final addresses
- symbol table: location of labels within file
- memory addresses not decided until final executable

a logistics note

on the waitlist? there are labs/lectures open

if you can't do the 2pm lecture, we can talk...

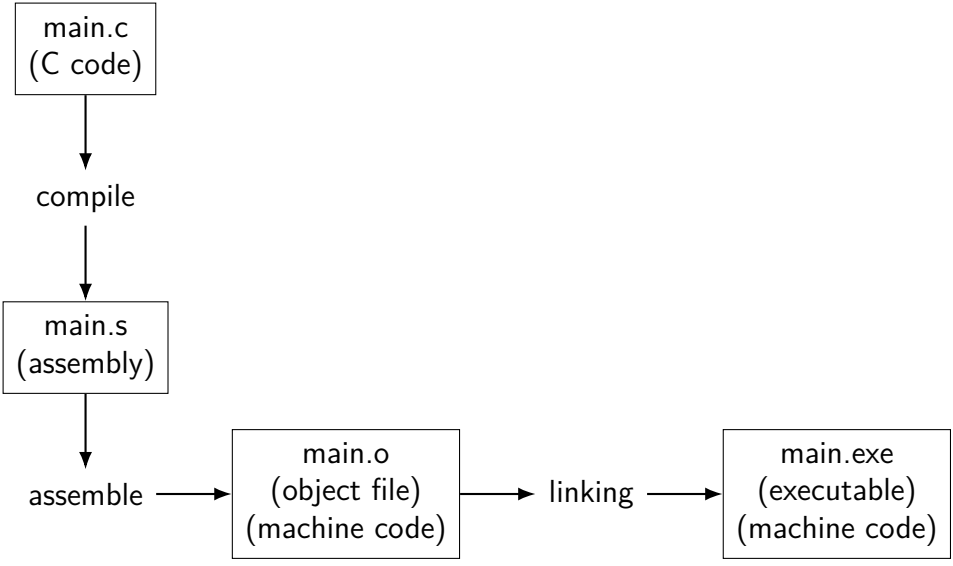
anonymous feedback

on Collab, I appreciate it — especially *specific* feedback:

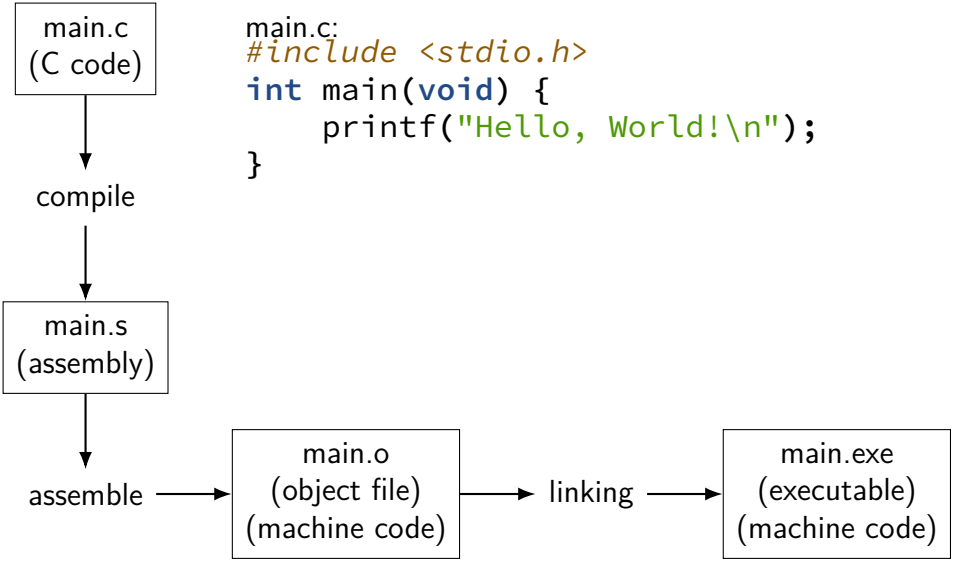
"The way that topics were introduced in class today was really confusing for me. When Prof. Reiss discussed the I/O bridge, he didn't first explain what each element was, making his explanation of the whole system extremely difficult to understand for those of us who didn't know what he was talking about. The explanation of Endianness also was confusing, and really would've been helped along by referencing a decimal number that makes more sense to the students rather than a hex number, which most people aren't nearly as comfortable with. Please consider changing your style of explanation to make the concepts more clear."

Hm — 2150 doesn't cover processors the way I thought...

compilation pipeline



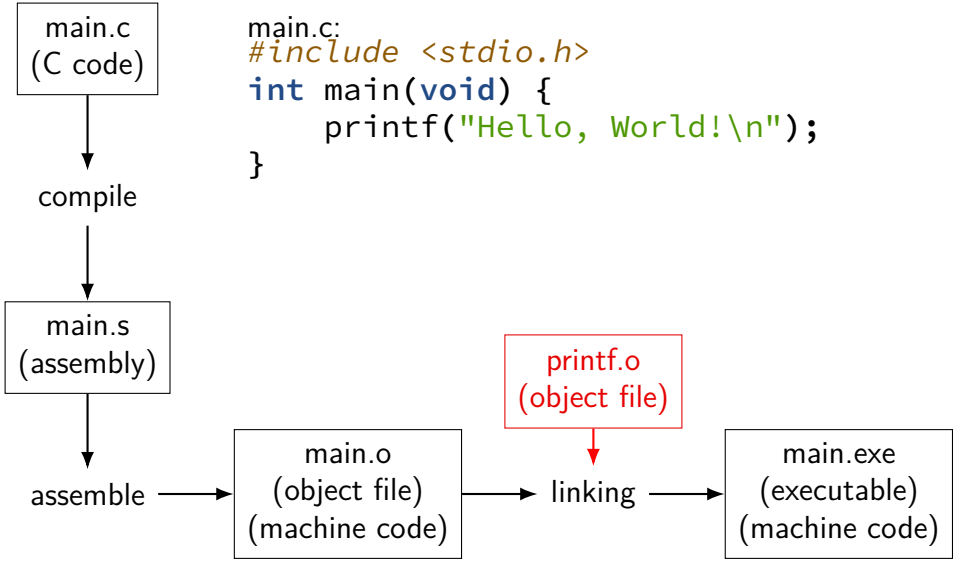
compilation pipeline



main.c:

```c
#include <stdio.h>
int main(void) {
    printf("Hello, World!\n");
}
```

compilation pipeline



main.c:

```c
#include <stdio.h>
int main(void) {
    printf("Hello, World!\n");
}
```

compilation commands

compile:	`gcc -S file.c`	⇒	`file.s` (assembly)
assemble:	`gcc -c file.s`	⇒	`file.o` (object file)
link:	`gcc -o file file.o`	⇒	`file` (executable)
c+a:	`gcc -c file.c`	⇒	`file.o`
c+a+l:	`gcc -o file file.c`	⇒	`file`

...

what's in those files?

hello.c

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

what's in those files?

hello.c

```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

```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

```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

```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

```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 10 ( )	address of puts

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
```

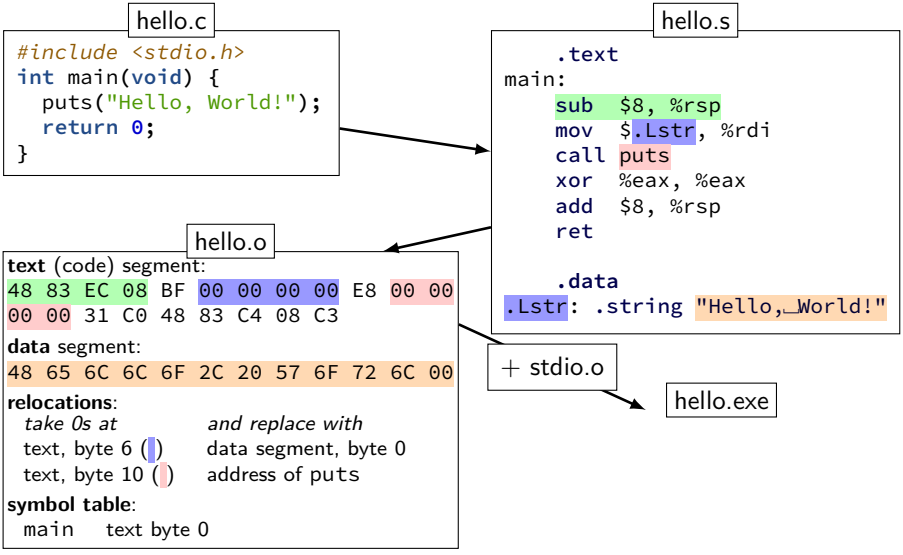
relocations:

take 0s at	*and replace with*
text, byte 6 ( )	data segment, byte 0
text, byte 10 ( )	address of puts

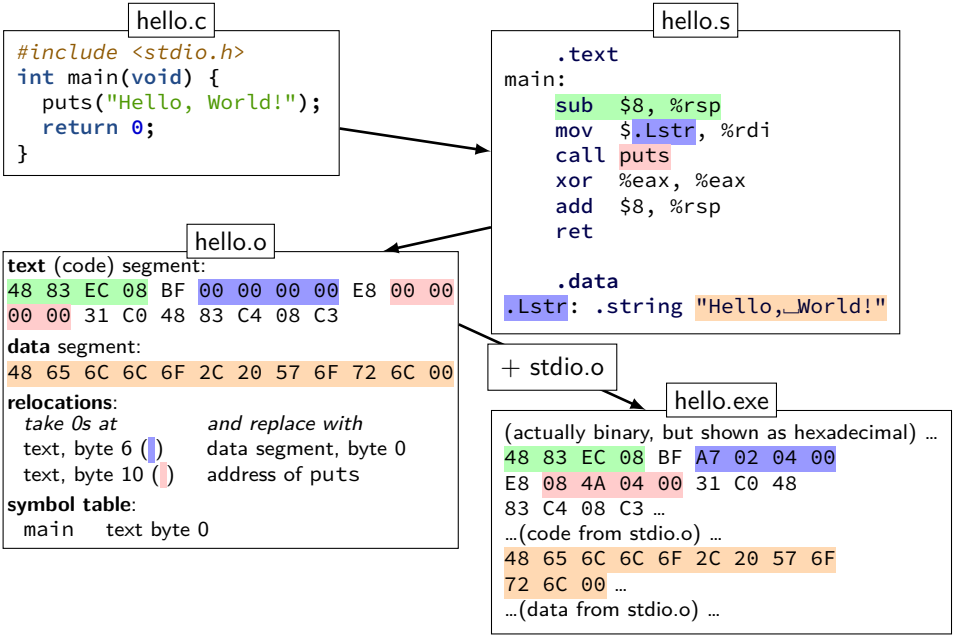
symbol table:

main	text byte 0

what's in those files?



what's in those files?



hello.s

```
        .section        .rodata.str1.1,"aMS",@progbi
.LC0:
        .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:

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

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

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

exercise (2)

main.c:

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

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

A. main.s (assembly)
B. main.o (object)
C. main.exe (executable)
D. B and C
E. A, B and C
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`

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

often extra indirection:

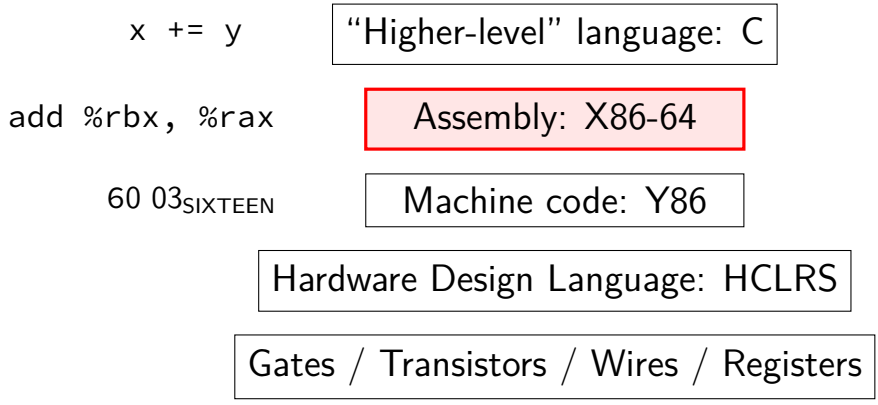
```
call functionTable[number_for_printf]
```

linker fills in `functionTable` instead of changing `call`s

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



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`

Intel syntax: `sub r8, rax`
AT&T syntax: `subq %rax, %r8`

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 (defualts 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 + 1
    // go to that address
```

recall: x86-64 general purpose registers

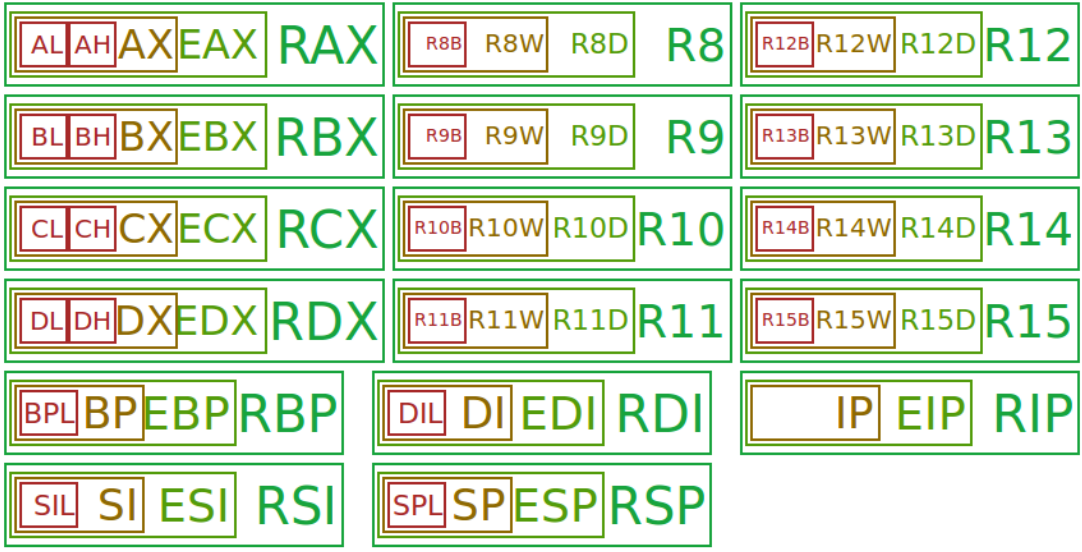


Figure: Immae via Wikipedia

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 `0xFFFFFFFFFFFFFF01`

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` ≈ `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` ≈ `addq $4, %rax`

"address of memory[rax + 4]" = rax + 4

LEA tricks

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

rax $\leftarrow$ rax $\times$ 5

rax $\leftarrow$ `address-of(memory[rax + rax * 4])`

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

rdx $\leftarrow$ rbx $+$ rcx

rdx $\leftarrow$`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. `arg * 9`
B. `-arg * 9`
C. `arg * 8`
D. `-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. `arg * 9`
B. `-arg * 9`
C. `arg * 8`
D. `-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 (**I**nstruction **P**ointer) = address of next instruction

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

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

on %rip

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

`movq 500(%rip), %rax`

rax ← memory[next instruction address + 500]

`label(%rip)` ≈ `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

authoriative source (1)

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

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

authoriative source (2)

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 Mitche

November 17, 2014

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
    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
```