Brief Assembly Refresher

Learn AT&T syntax

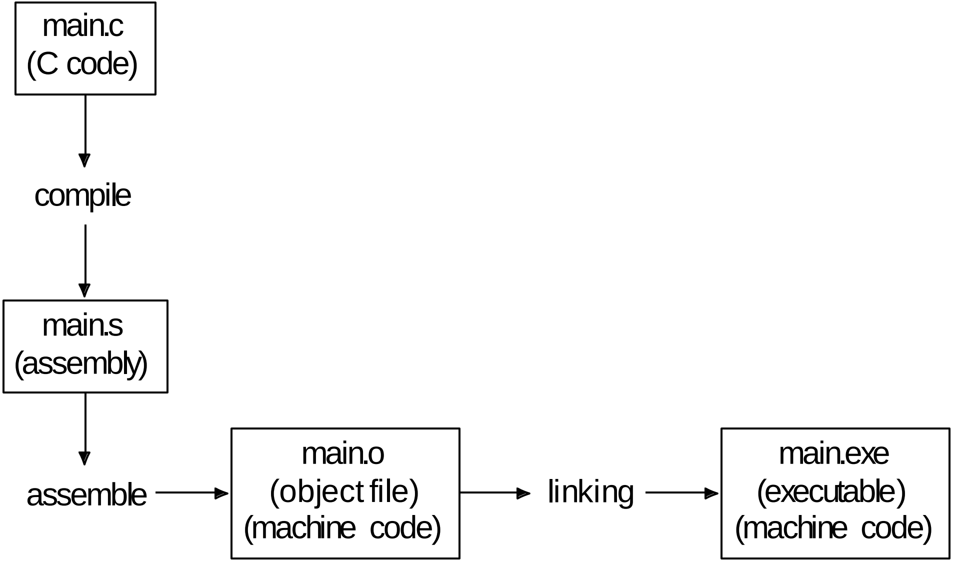
last time

- processors ↔ memory, I/O devices
  - processor: send addresses (or memory values)
  - memory: reply with stores value or retrieves at address.
- endianness:
  - little = least address is least significant
    - little endian: `0x1234` : **`0x34`** at address $x + 0$
    - big endian: `0x1234` : **`0x12`** at address $x + 0$
- object files and linking
  - relocations: "fill in the blank" with final addresses symbol table: location of labels within file like main
  - We will review in more detail.

Overview/ Learning Goals

- Generally understand the compilation pipeline
- Learn how to read and write AT&T syntax assembly
- Review x86 registers and condition codes .
- Be able to translate from C to AT&T syntax assembly

compilation pipeline



what's in those files?

hello.c

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

Assembly — dgg6b@portal02:~/public_html/ComputerNetworks/Website/CourseS...

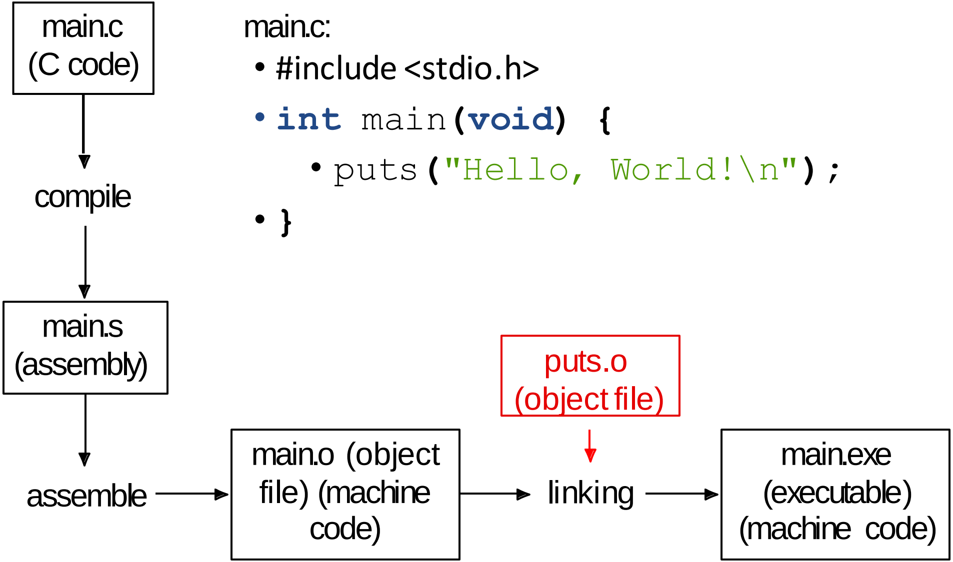
```
d-172-27-99-171:Assembly dgg6b$ cat file.c
#include <stdio.h>
int main(void) {
        printf("Hello, World!\n");
}

d-172-27-99-171:Assembly dgg6b$ 
```

compilation pipeline

main.c:

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



what's in those files?

hello.c

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

hello.s

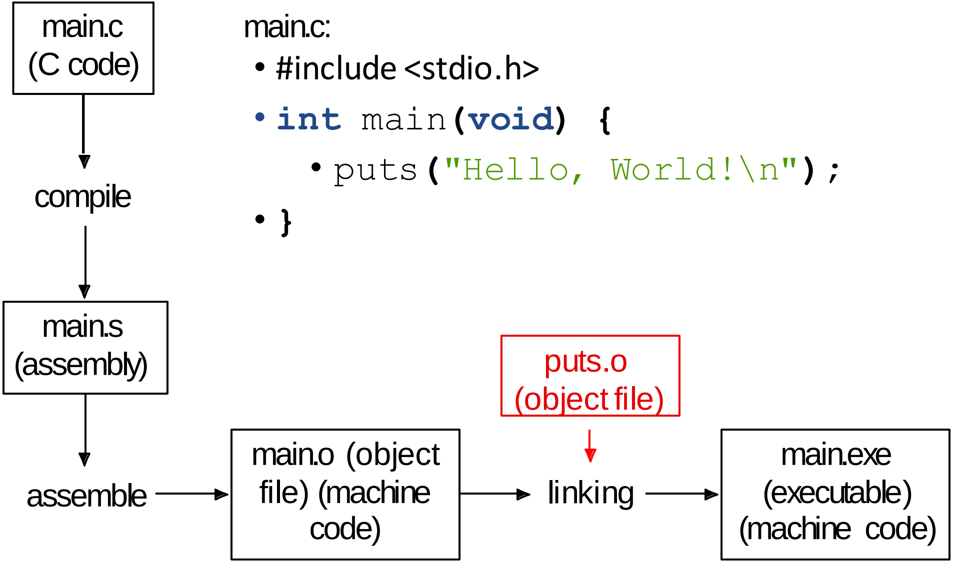
```
    .text
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret

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

Assembly — dgg6b@portal02:~/public_html/ComputerNetworks/Website/Cour...

```
d-172-27-99-171:Assembly dgg6b$ vim file.c
d-172-27-99-171:Assembly dgg6b$ gcc -Os -S file.c
d-172-27-99-171:Assembly dgg6b$ ls
file.c  file.s
d-172-27-99-171:Assembly dgg6b$ cat file.s
        .section        __TEXT,__text,regular,pure_instructions
        .macosx_version_min 10, 13
        .globl  _main                   ## -- Begin function main
_main:                                  ## @main
        .cfi_startproc
## BB#0:
        pushq   %rbp
Lcfi0:
        .cfi_def_cfa_offset 16
Lcfi1:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Lcfi2:
        .cfi_def_cfa_register %rbp
        leaq    L_str(%rip), %rdi
        callq   _puts
        xorl    %eax, %eax
        popq    %rbp
        retq
```

compilation pipeline



main.c:

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

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!"
```

+ stdio.o

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?

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

Assembly — dgg6b@portal02:~/public_html/ComputerNetworks/Website/CourseS...

```
d-172-27-99-171:Assembly dgg6b$ gcc -c file.s
d-172-27-99-171:Assembly dgg6b$ ls
file.c  file.o  file.s
d-172-27-99-171:Assembly dgg6b$ otool -t file.o
file.o:
Contents of (__TEXT,__text) section
0000000000000000        55 48 89 e5 48 8d 3d 09 00 00 00 e8 00 00 00 00
0000000000000010        31 c0 5d c3
d-172-27-99-171:Assembly dgg6b$ otool -r file.o
RELOCATION RECORDS FOR [__text]:
000000000000000c X86_64_RELOC_BRANCH _puts
0000000000000007 X86_64_RELOC_SIGNED __cstring

RELOCATION RECORDS FOR [__compact_unwind]:
0000000000000000 X86_64_RELOC_UNSIGNED __text

d-172-27-99-171:Assembly dgg6b$ nm file.o
0000000000000000 T _main
                 U _puts
d-172-27-99-171:Assembly dgg6b$
```

0xc = 12

Unwind section is for exception handling

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

\+ 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) …

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`
...

exercise (1) Visit Kahoot.it

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

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

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

Which files contain the **memory address** of `"Hello World"`?

A. main.s (assembly)

B. main.o (object)

C. main.exe (executable) *E.* something else

exercise (2). Kahoot.it

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. A, B and C

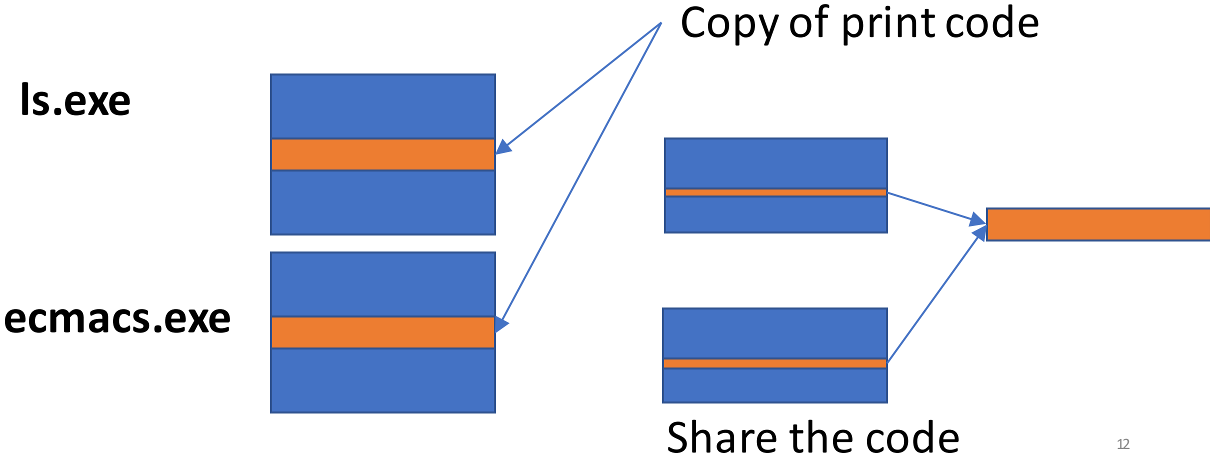
Relocation types

- machine code doesn't always use direct addresses
  - The address is sometime computed relative example relative to the program counter
  - "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`)



View a list of dynamic libraries that get loaded at run time

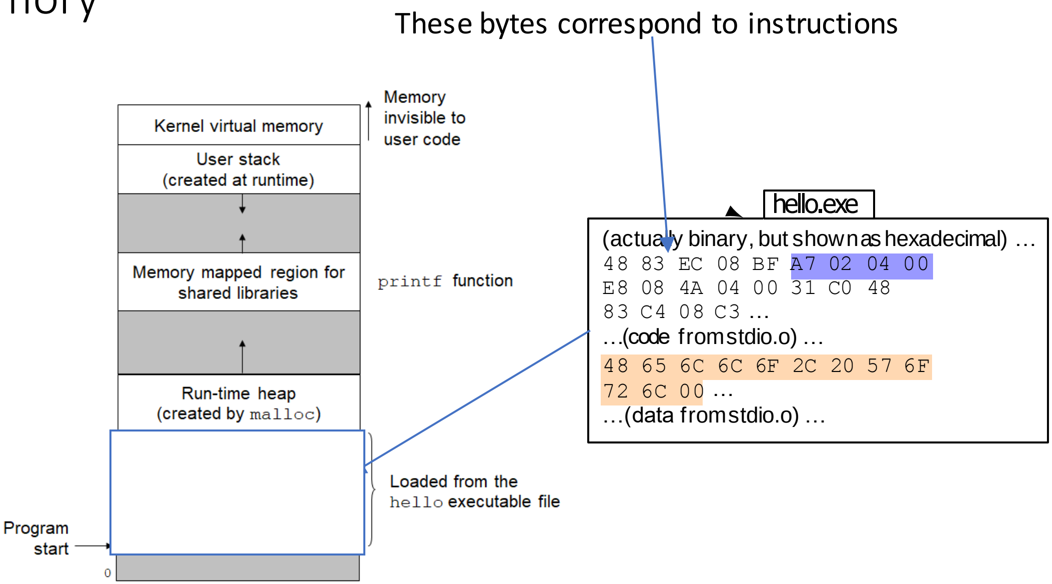
ldd /bin/ls. (linux)

```
$ 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)
```

Shared Object file

Great so now does the program get laid out in memory?

Memory


These bytes correspond to instructions
Kernel virtual memory
Memory invisible to user code
User stack (created at runtime)
Memory mapped region for shared libraries
printf function
Run-time heap (created by malloc)
Loaded from the hello executable file
Program start
0
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) …

Great I get how program get turned into binary. But I need a quick assembly refresh so that I can start reading assembly code again.

hello.s

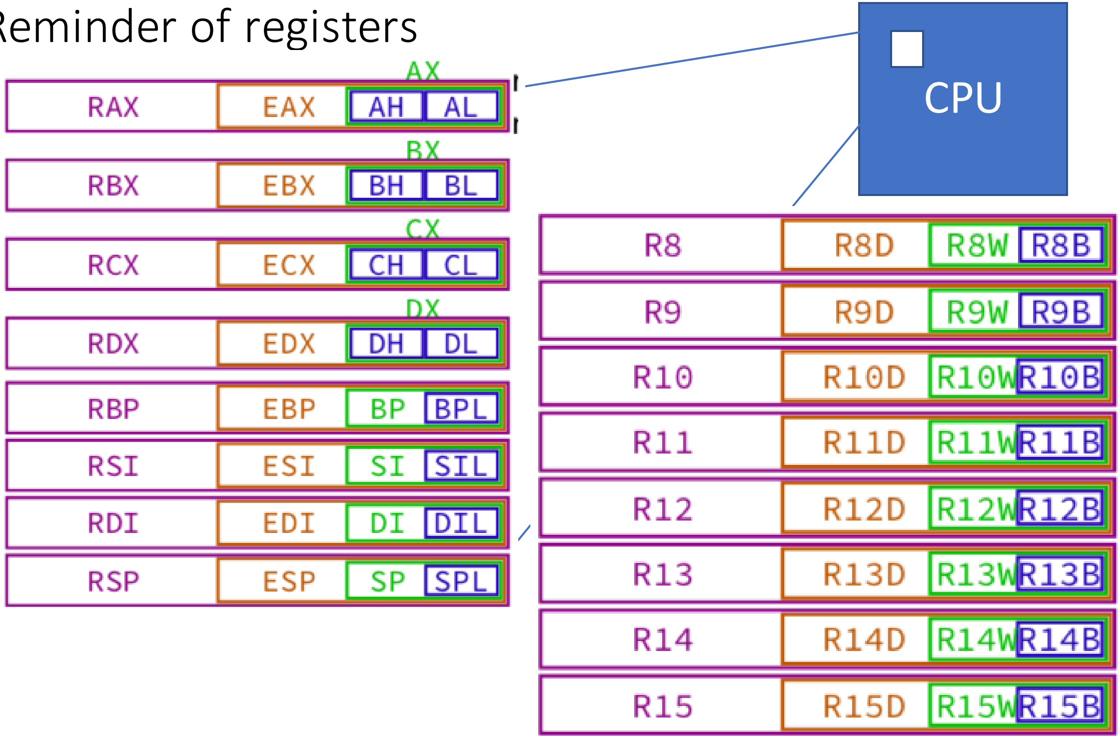
```
    .text
main:
    sub $8, %rsp
    mov $.Lstr, %rdi
    call puts
    xor %eax, %eax
    add $8, %rsp
    ret

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

Let's start by reviewing registers and the syntax

Does the RDI register represent

Reminder of registers

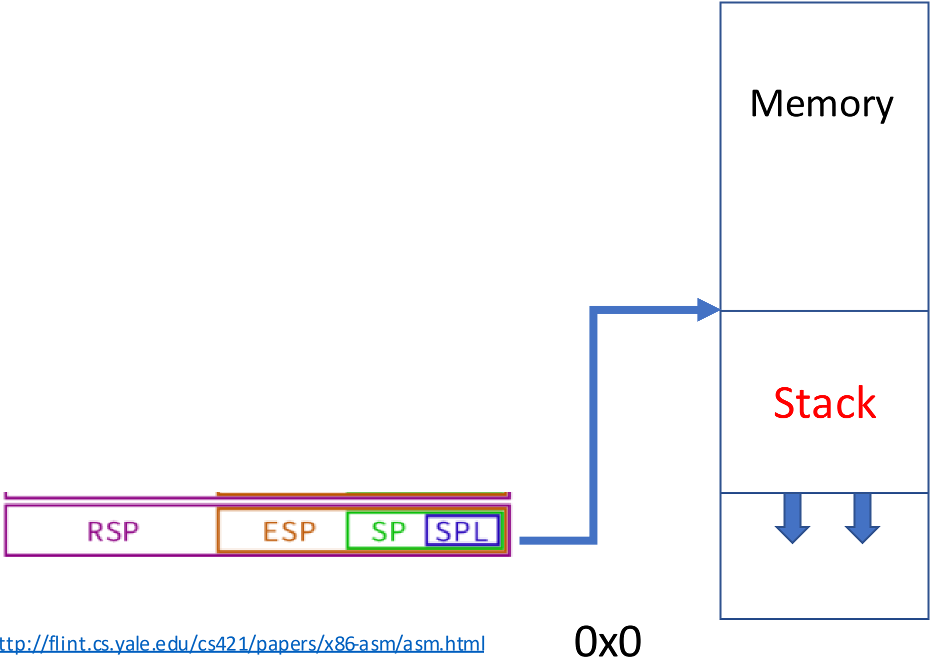


Key Registers Review

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

Callee-saved registers (AKAnon-volatile registers) are used to hold long-lived values that should be preserved across calls

Key Registers Review



http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html

AT&T syntax vs Intel Syntax

AT&T syntax	**Intel Syntax**
`movq $42, (%rbx)`	`mov QWORD PTR [rbx], 42`

We will be using AT&T syntax in this class

effect (pseudo-C):

`memory[rbx] <- 42`

destination last

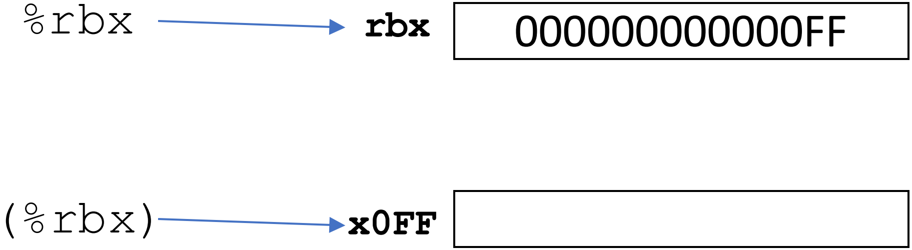
Key Points for AT&T syntax

- registers start with %

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

Key Points for AT&T syntax

- `()` s represent value in memory



Key Points for AT&T syntax

- constants start with $

\$42 → 000000000000002A

$16^1, 16^0$

$2*16 + 1*10(A) = 42$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

AT&T syntax example (1)

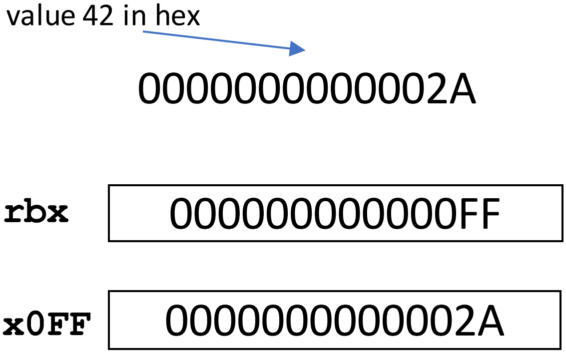
```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

`()`s represent value in memory

constants start with `$`

registers start with `%`



AT&T syntax example (1)

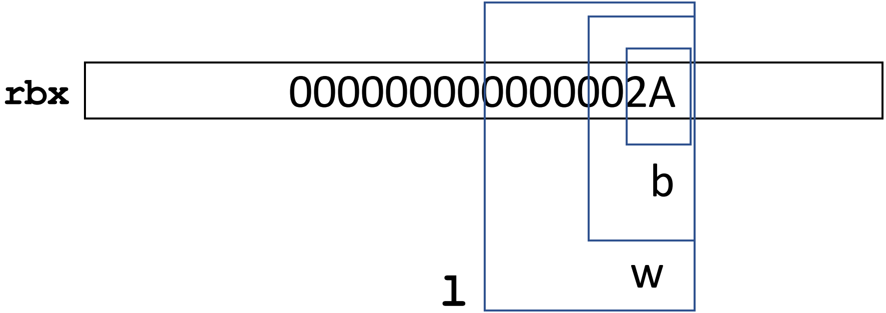
```
movq $42, (%rbx)
// memory[rbx] ← 42
```

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

l: 4; w: 2; b: 1

sometimes can be omitted

suffix	Meaning
b	"Byte": 1 byte
w	"Word": 2 bytes
l	"Long": 4 bytes
q	"Quad": 8 bytes (4 words)

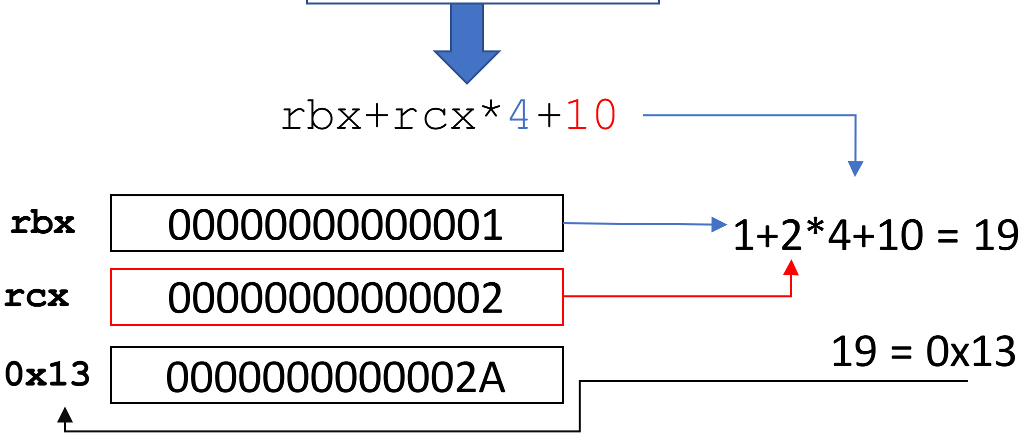


Other was to compute addresses

AT&T syntax:

```
movq $42, 10(%rbx,%rcx,4)
```

$42 = 0x2A



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 There are several variations

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

AT&T Syntax	Pseudo code
100(%rbx, %rcx, 4)	memory[rbx+rcx*4 + 100]
100(%rbx)	memory[rbx + 100]
100(%rbx,8)	memory[rbx * 8 + 100]
100(,%rbx,8):	memory[rbx * 8 + 100]
100(%rbx,%rcx):	memory[rbx+rcx+100]
100	memory[100]

Subtraction

```
r8 ← r8 - rax
```

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

same for cmpq %rax, %r8

Remember that is the destination

AT&T syntax: addresses

AT&T Syntax	Description
`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 $ —> memory address

AT&T syntax in one slide (Summary)

- destination last
- `()` means value in memory
- `disp(base, index, scale)` same as
- `memory[disp + base + index * scale]`
  - omit disp (defaults to `0`)
  - omit base (defaults to `0`)
  - scale (defualts to `1`)
- `$` means constant
- plain number/label means value in memory

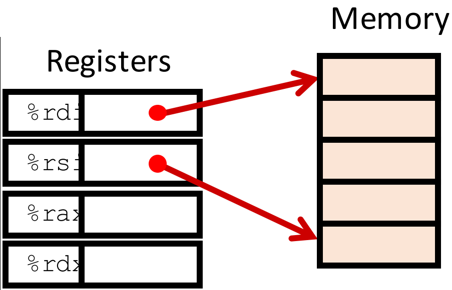
Example of Simple Addressing Modes

```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

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

Understanding Swap()

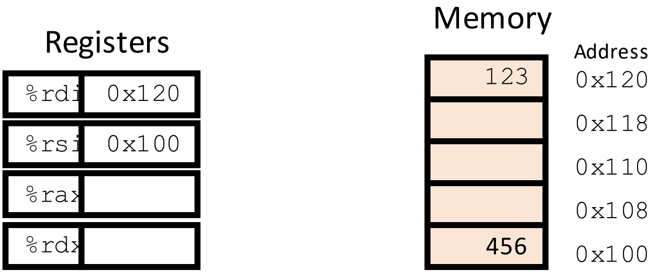
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

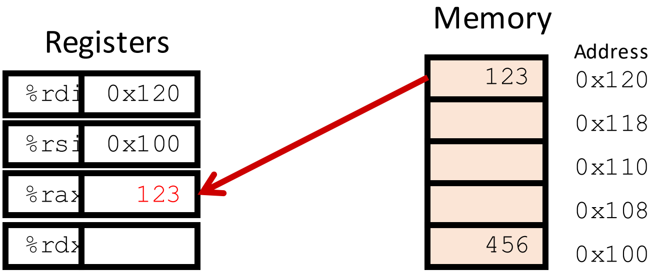
```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



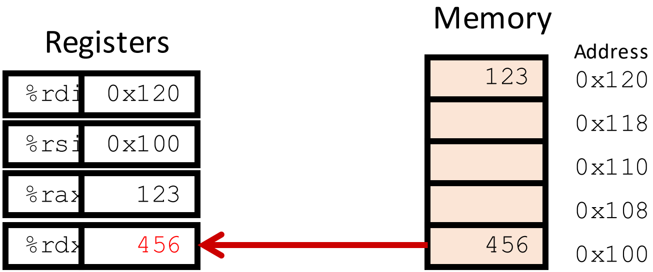

```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



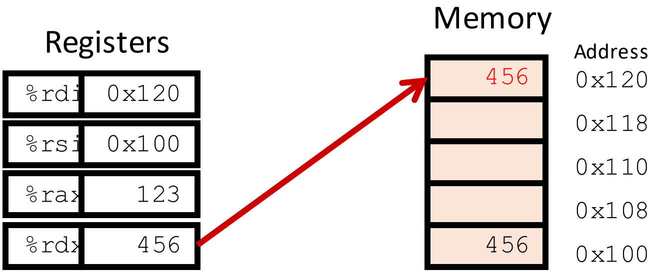

```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()



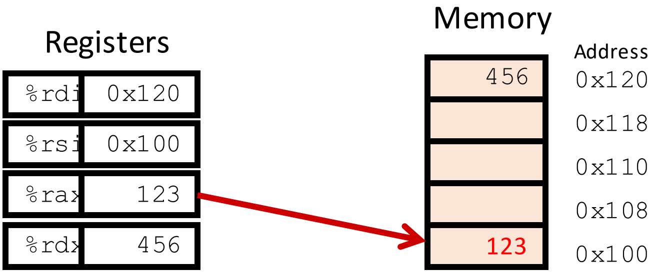

```
swap:
    movq     (%rdi), %rax    # t0 = *xp
    movq     (%rsi), %rdx    # t1 = *yp
    movq     %rdx, (%rdi)    # *xp = t1
    movq     %rax, (%rsi)    # *yp = t0
    ret
```

Understanding Swap()




```
swap:
    movq     (%rdi), %rax   # t0 = *xp
    movq     (%rsi), %rdx   # t1 = *yp
    movq     %rdx, (%rdi)   # *xp = t1
    movq     %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()




```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Let’s look at some more instructions

Assembly Continued

- Overlapping registers
- Lea (load effective address)
- Labels
- Condition codes
- Jmp (Computed Jumps)
- Translating from C to assembly

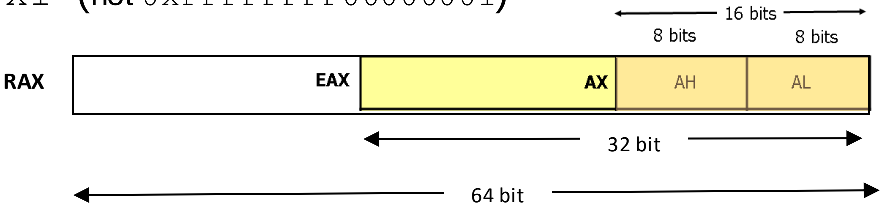
Overlapping Registers

overlapping registers (1)

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

```
movq $0xFFFFFFFFFFFFFFFF, %rax
movl $0x1, %eax
```

%rax is `0x1` (not `0xFFFFFFFF00000001`)

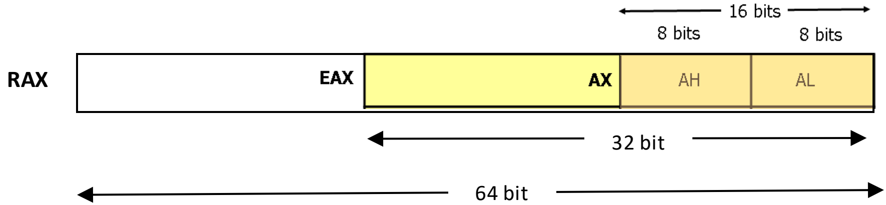


overlapping registers (2)

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

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

%rax is `0xFFFFFFFFFFFFFF01` not `0x01`



Labels

Labels represent addresses

labels

```
    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

What’s the different between lea and mov

leaq vs. movq example

Registers

%rax	
%rbx	
%rcx	0x4
%rdx	0x100
%rdi	
%rsi	

Memory

	Address
0x400	0x120
0xf	0x118
0x8	0x110
0x10	0x108
0x1	0x100

```
leaq  (%rdx,%rcx,4),  %rax
movq  (%rdx,%rcx,4),  %rbx
leaq  (%rdx), %rdi
movq  (%rdx), %rsi
```

leaq vs. movq example

Registers

Register	Value
%rax	0x110
%rbx	
%rcx	0x4
%rdx	0x100
%rdi	
%rsi	

Memory

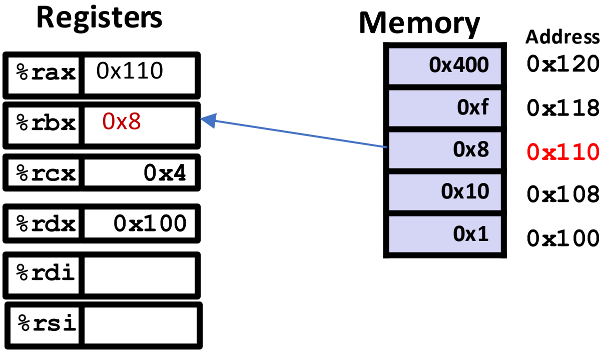
Memory	Address
0x400	0x120
0xf	0x118
0x8	0x110
0x10	0x108
0x1	0x100

```
leaq  (%rdx,%rcx,4),  %rax
movq  (%rdx,%rcx,4),  %rbx
leaq  (%rdx), %rdi
movq  (%rdx), %rsi
```

%rdx + %rcx * 4 -> %rax

0x100 + (0x4 * 4) = 0x110

leaq vs. movq example



Takes the value at address

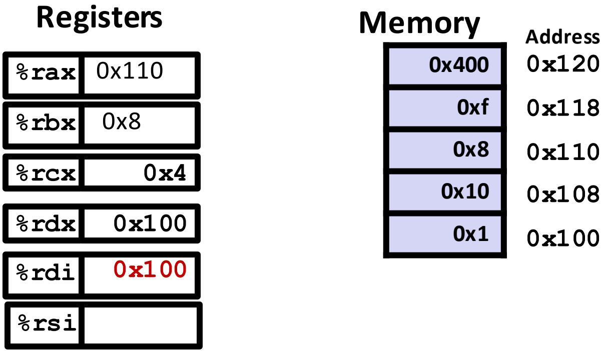
Memory value at rbx

```
Leaq  (%rdx,%rcx,4),  %rax
Movq  (%rdx,%rcx,4),  %rbx
leaq  (%rdx), %rdi
movq  (%rdx), %rsi
```

%rdx + %rcx * 4 -> %rbx

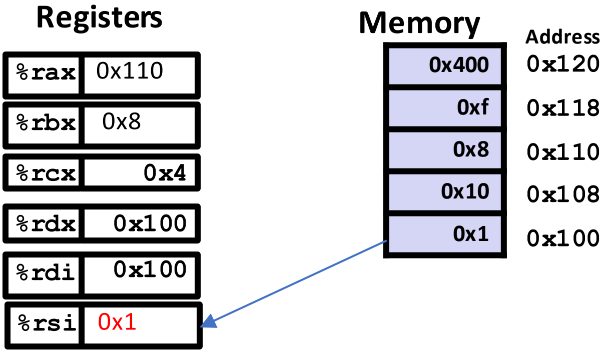
0x100 + (0x4 * 4) = 0x110

leaq vs. movq example




```
Leaq  (%rdx,%rcx,4),  %rax
Movq  (%rdx,%rcx,4),  %rbx
leaq  (%rdx), %rdi
movq  (%rdx), %rsi
```

leaq vs. movq example




```
Leaq  (%rdx,%rcx,4),  %rax
Movq  (%rdx,%rcx,4),  %rbx
leaq  (%rdx), %rdi
movq  (%rdx), %rsi
```

Address Computation Instruction

- **`leaq`** ***Src***, ***Dst***
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression

- Uses
  - Computing arithmetic expressions of the form
    - x + k*y
    - k = 1, 2, 4, or 8

LEA tricks

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

rax ← rax × 5

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

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

rdx ← rbx + 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. arg * 9

B. -arg * 9

C. none of these

D. arg * 8

https://create.kahoot.it/kahoots/my-kahoots

Condition Codes (Implicit Setting)

- Single bit registers
  - **CF** Carry Flag (for unsigned) **SF** Sign Flag (for signed)
  - **ZF** Zero Flag **OF** Overflow Flag (for signed)

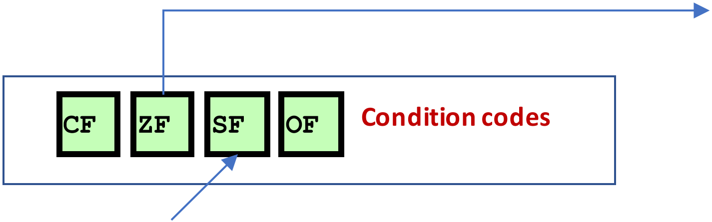
- Implicitly set (think of it as side effect) by arithmetic operations
  - Example: **`addq`** *Src,Dest* ↔ **`t = a+b`**
  - **CF set** if carry out from most significant bit (unsigned overflow)
  - **ZF set** if **`t == 0`**
  - **SF set** if **`t < 0`** (as signed)
  - **OF set** if two's-complement (signed) overflow
    `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

- Not set by **`leaq`** instruction

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

Set 1 if result was zero.



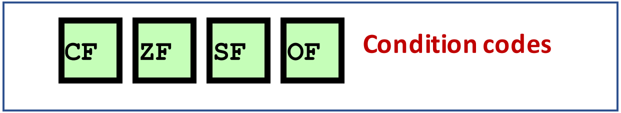
Set 1 if negative 0 if positive

JUMP instruction and their associated

[X86-guide]

Instruction	Description	Condition Code
jle	Jump if less or equal	(SF ^ OF) | ZF
jg	Jump if greater (signed)	~(SF ^ OF) & ~ZF
je	Jump if equal	ZF

Why set the overflow flag

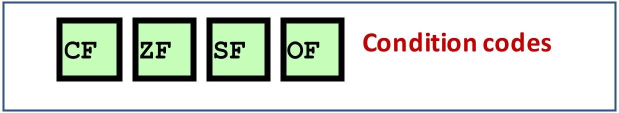


NOT XOR

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
```

<table>
<tr><td>jle</td><td>Jump if less or equal</td><td>(SF ^ OF) | ZF</td></tr>
</table>



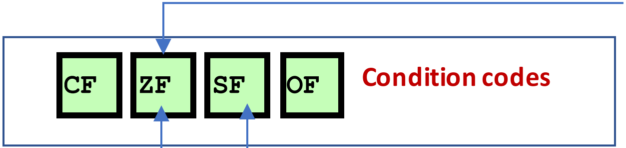
https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

condition codes and cmpq

`cmp` does subtraction (but doesn't store result)

`cmp %rax, %rdi -> rdi - rax`

Set zero flag if equal



0101 (decimal 5)
AND 0011 (decimal 3)
= 0001 (decimal 1)

similarly `test` does bitwise-and

`testq %rax, %rax` — result is %rax

Set zero flag if result of bitwise and is zero

Also sets the SF flag with most significant bit of the resutl

Computed Jumps

Computed jumps

Instruction	Description
jmpq *%rax	Intel syntax: jmp RAX goto address 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

From C to Assembly

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
     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:
```

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

00
00
00
00
01
00
00
00
00
00
00
00
02

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)

Appendix

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