

# CS 24

## Introduction to Computing Systems

# x86-64 Basics

```
mov %edi, %eax
xor %ecx, %ecx
test %edi, %edi
setne %cl
mov $0xffffffff, %eax
cmovns %ecx, %eax
retq
```

- In 2021, almost all assembly is written by **compilers** or in the **operating system**, but who writes those?
- In 2021, the high-level language model occasionally breaks down and you have to read the assembly to understand the machine's behavior!
- In 2021, it's important to understand the types of optimizations a compiler is capable of making—and those it isn't!
- In 2021, software is generally distributed in binary form; if you want to **reverse engineer** or **security audit** software, it's going to be assembly!

```
blank@labradoodle:~$ cat identity.c
int identity(int x) {
    return x;
}
```

```
blank@labradoodle:~$ clang -S identity.c
blank@labradoodle:~$ cat identity.s
identity:
    movl %edi, %eax
    retq
```

```
blank@labradoodle:~$ cat identity.c
int identity(int x) {
    return x;
}
```

```
blank@labradoodle:~$ clang -c identity.c
blank@labradoodle:~$ objdump -d identity.o
```

```
simple.o:          file format elf64-x86-64
```

Disassembly of section .text:

```
0000000000000000 <identity>:
   0: 89 f8                mov     %edi,%eax
   2: c3                  retq
```

- **Immediates:** Constant integer data
  - Examples: `$0x400`, `$-533`
  - Like C literal, but prefixed with ‘\$’
  - Encoded with 1, 2, 4, or 8 bytes depending on the instruction
- **Registers:** behave like “global variables”, but hardwired in the processor
  - Examples: `%eax`, `%edi`
  - Some of them are reserved for special uses or have special meanings
- **Memory:** Consecutive bytes of memory

**Registers** are locations in the CPU that store a small amount of data, which can be accessed very quickly (once every clock cycle). They have names (e.g., `%rsi`)—not addresses. They are a precious commodity in all architectures, but especially x86-64.

(return)

`%eax`

`%ebx`

(arg 4)

`%ecx`

(arg 3)

`%edx`

(arg 1)

`%edi`

(arg 2)

`%esi`

`%esp`

`%ebp`

There are three major **classes** of things assembly instructions do:

- 1 Transfer data between memory and registers
  - Load data from memory into register: `%reg = mem[addr]`
  - Store register data into memory: `mem[addr] = %reg`
- 2 Perform arithmetic operation on register or memory data
  - e.g., `%eax += %ebx`
  - e.g., `%eax += mem[addr]`
- 3 Re-direct control flow (jumps and gotos)



## C Code

```
1 int identity(int x) {  
2     return x;  
3 }
```

## x86-64 Disassembly

```
0000000000400480 <identity>:  
400480: 89 f8                mov  %edi,%eax  
400482: c3                  retq
```

## C Code

```
1 int identity(int x) {  
2     return x;  
3 }
```

## x86-64 Disassembly

```
00000000000400480 <identity>:  
400480: 89 f8                mov  %edi,%eax  
400482: c3                  retq
```

## x86-64 Instruction: `mov`

### x86-64

`mov %src, %dst`

↔

### C Pseudocode

`%dst = %src`

## C Code

```
1 int identity(int x) {  
2     return x;  
3 }
```

## x86-64 Disassembly

```
00000000000400480 <identity>:  
400480: 89 f8                mov  %edi,%eax  
400482: c3                  retq
```

## x86-64 Instruction: mov

### x86-64

mov %src, %dst

↔

### C Pseudocode

%dst = %src

## Pseudocode Translation (so far)

```
1 identity:  
2     %eax = %edi  
3     retq
```

# Understanding Identity (`retq` and return value)

## C Code

```
1 int identity(int x) {  
2     return x;  
3 }
```

## x86-64 Disassembly

```
00000000000400480 <identity>:  
400480: 89 f8                mov  %edi,%eax  
400482: c3                  retq
```

## x86-64 Instruction: `mov`

x86-64  
`mov %src, %dst`

↔

C Pseudocode  
`%dst = %src`

## x86-64 Instruction: `retq`

x86-64  
`retq`

↔

C Pseudocode  
`return %eax`

## C Code

```
1 int identity(int x) {  
2     return x;  
3 }
```

## x86-64 Disassembly

```
00000000000400480 <identity>:  
400480: 89 f8                mov  %edi,%eax  
400482: c3                  retq
```

## x86-64 Instruction: `mov`

x86-64  
`mov %src, %dst`

↔

C Pseudocode  
`%dst = %src`

## x86-64 Instruction: `retq`

x86-64  
`retq`

↔

C Pseudocode  
`return %eax`

## Pseudocode Translation (so far)

```
1 identity:  
2     %eax = %edi  
3     return %eax
```

## C Code

```
1 int identity(int x) {  
2     return x;  
3 }
```

## x86-64 Disassembly

```
00000000000400480 <identity>:  
400480: 89 f8                mov  %edi,%eax  
400482: c3                  retq
```

## x86-64 Instruction: mov

### x86-64

mov %src, %dst

↔

### C Pseudocode

%dst = %src

## x86-64 Instruction: retq

### x86-64

retq

↔

### C Pseudocode

return %eax

## C Code

```
1 int identity(int x) {  
2     return x;  
3 }
```

## x86-64 Disassembly

```
00000000000400480 <identity>:  
400480: 89 f8                mov  %edi,%eax  
400482: c3                  retq
```

## x86-64 Instruction: mov

x86-64  
mov %src, %dst

↔

C Pseudocode  
%dst = %src

## x86-64 Instruction: retq

x86-64  
retq

↔

C Pseudocode  
return %eax

## Pseudocode Translation (so far)

```
1 identity(%edi):  
2     %eax = %edi  
3     return %eax
```

## C Code

```
1 int identity(int x) {  
2     return x;  
3 }
```

## x86-64 Disassembly

```
0000000000400480 <identity>:  
400480: 89 f8                mov  %edi,%eax  
400482: c3                  retq
```

## x86-64 Instruction: mov

### x86-64

mov %src, %dst

↔

### C Pseudocode

%dst = %src

## x86-64 Instruction: retq

### x86-64

retq

↔

### C Pseudocode

return %eax



## C Code

```
1 int identity(int x) {  
2     return x;  
3 }
```

## x86-64 Disassembly

```
0000000000400480 <identity>:  
400480: 89 f8                mov  %edi,%eax  
400482: c3                  retq
```

## x86-64 Instruction: mov

x86-64  
mov %src, %dst

↔

C Pseudocode  
%dst = %src

## x86-64 Instruction: retq

x86-64  
retq

↔

C Pseudocode  
return %eax

## Pseudocode Translation (so far)

```
1 identity(%edi):  
2     return %edi
```

## System V AMD64 ABI

- The value in `%eax` is automatically returned by `retq`.
- The first argument to a function is stored in `%edi`.

## Things to Notice About x86-64

- There are no types!!!
- The **conventions** define what the compiler is allowed to do.

```
000000000000000000 <arith>:  
 0: 83 c7 01          add  $0x1,%edi  
 3: 0f af ff          imul %edi,%edi  
 6: 89 f8             mov  %edi,%eax  
 8: c3                retq
```

```
0000000000000000 <arith>:  
 0: 83 c7 01          add  $0x1,%edi  
 3: 0f af ff          imul %edi,%edi  
 6: 89 f8             mov  %edi,%eax  
 8: c3                retq
```

## Pseudocode Translation (so far)

```
1 arith(%edi):  
2   add $0x1,%edi  
3   imul %edi,%edi  
4   %eax = %edi  
5   return %eax
```

```
000000000000000000 <arith>:
 0: 83 c7 01          add  $0x1,%edi
 3: 0f af ff          imul %edi,%edi
 6: 89 f8             mov  %edi,%eax
 8: c3                retq
```

## Pseudocode Translation (so far)

```
1 arith(%edi):
2   add $0x1,%edi
3   imul %edi,%edi
4   %eax = %edi
5   return %eax
```

## x86-64 Instruction: (Some) Arithmetic Operations

<u>x86-64</u>		<u>C Pseudocode</u>
<b>add</b> %src, %dst	↔	%dst += %src
<b>sub</b> %src, %dst	↔	%dst -= %src
<b>imul</b> %src, %dst	↔	%dst *= %src

```
000000000000000000 <arith>:
 0: 83 c7 01          add  $0x1,%edi
 3: 0f af ff          imul %edi,%edi
 6: 89 f8             mov  %edi,%eax
 8: c3               retq
```

## x86-64 Instruction: (Some) Arithmetic Operations

<u>x86-64</u>		<u>C Pseudocode</u>
<b>add</b> %src, %dst	↔	%dst += %src
<b>sub</b> %src, %dst	↔	%dst -= %src
<b>imul</b> %src, %dst	↔	%dst *= %src

## Pseudocode Translation (so far)

```
1 arith(%edi):
2   %edi += $0x1
3   %edi *= %edi
4   %eax = %edi
5   return %eax
```

```
000000000000000000 <arith>:
 0: 83 c7 01          add  $0x1,%edi
 3: 0f af ff          imul %edi,%edi
 6: 89 f8             mov  %edi,%eax
 8: c3                retq
```

## x86-64 Instruction: (Some) Arithmetic Operations

<u>x86-64</u>		<u>C Pseudocode</u>
<b>add</b> %src, %dst	↔	%dst += %src
<b>sub</b> %src, %dst	↔	%dst -= %src
<b>imul</b> %src, %dst	↔	%dst *= %src

## Pseudocode Translation (so far)

```
1 arith(%edi):
2   return (%edi + $0x1) * (%edi + $0x1)
```

## Things to Notice

- There are no types!!!
- The **ABI** defines what the compiler should do.

(return)	%eax
	%ebx
(arg 4)	%ecx
(arg 3)	%edx
(arg 1)	%edi
(arg 2)	%esi
	%esp
	%ebp



# Another Mystery Function (bit operations)

16

```
000000000000000000 <mystery>:  
0: 31 c9          xor    %ecx,%ecx  
2: 85 ff          test   %edi,%edi  
4: 0f 95 c1       setne  %cl  
7: b8 ff ff ff ff mov    $0xffffffff,%eax  
c: 0f 49 c1       cmovns %ecx,%eax  
f: c3            retq
```

(return)

%eax

%ebx

(arg 4)

%ecx

(arg 3)

%edx

(arg 1)

%edi

(arg 2)

%esi

%esp

%ebp

Pseudocode Translation (so far)

# Another Mystery Function (bit operations)

16

```
0000000000000000 <mystery>:  
0: 31 c9          xor    %ecx,%ecx  
2: 85 ff          test   %edi,%edi  
4: 0f 95 c1       setne %cl  
7: b8 ff ff ff ff mov    $0xffffffff,%eax  
c: 0f 49 c1       cmovns %ecx,%eax  
f: c3            retq
```

(return)

%eax

%ebx

(arg 4)

%ecx

(arg 3)

%edx

(arg 1)

%edi

(arg 2)

%esi

%esp

%ebp

## Pseudocode Translation (so far)

```
1 mystery(%edi):  
2   %ecx = 0  
3   ...
```

```
0000000000000000 <mystery>:  
0: 31 c9                xor  %ecx,%ecx  
2: 85 ff                test %edi,%edi  
4: 0f 95 c1            setne %cl  
7: b8 ff ff ff ff     mov  $0xffffffff,%eax  
c: 0f 49 c1            cmovns %ecx,%eax  
f: c3                retq
```

### x86-64 Instruction: test

The processor has a special register that contains “flags” which **test** sets.

```
test %r1, %r2
```

- ZF set to result of  $(\%r1 \& \%r2) == 0$
- SF set to result of  $(\%r1 \& \%r2) < 0$

```

0000000000000000 <mystery>:
 0: 31 c9          xor  %ecx,%ecx
 2: 85 ff          test %edi,%edi
 4: 0f 95 c1       setne %cl
 7: b8 ff ff ff ff mov  $0xffffffff,%eax
 c: 0f 49 c1       cmovns %ecx,%eax
 f: c3           retq
    
```

(return)	%eax
	%ebx
(arg 4)	%ecx
(arg 3)	%edx
(arg 1)	%edi
(arg 2)	%esi
	%esp
	%ebp
	CF ZF SF OF

## x86-64 Instruction: test

The processor has a special register that contains “flags” which **test** sets.

```
test %r1, %r2
```

- ZF set to result of  $(\%r1 \& \%r2) == 0$
- SF set to result of  $(\%r1 \& \%r2) < 0$

## Pseudocode Translation (so far)

```

1 mystery(%edi):
2   %ecx = 0
3   ZF = %edi == 0
4   SF = %edi < 0
5   ...
    
```

```

0000000000000000 <mystery>:
 0: 31 c9          xor  %ecx,%ecx
 2: 85 ff          test %edi,%edi
 4: 0f 95 c1      setne %cl
 7: b8 ff ff ff ff mov  $0xffffffff,%eax
 c: 0f 49 c1      cmovns %ecx,%eax
 f: c3          retq

```

## x86-64 Instruction: set??

<u>x86-64</u>		<u>C Pseudocode</u>
<b>sete</b> %r	↔	%r = ZF
<b>setne</b> %r	↔	%r = ~ZF
<b>sets</b> %r	↔	%r = SF
<b>setns</b> %r	↔	%r = ~SF

```

0000000000000000 <mystery>:
 0: 31 c9          xor  %ecx,%ecx
 2: 85 ff          test %edi,%edi
 4: 0f 95 c1      setne %cl
 7: b8 ff ff ff ff mov  $0xffffffff,%eax
 c: 0f 49 c1      cmovns %ecx,%eax
 f: c3          retq
    
```

(return)

%eax

%ebx

(arg 4)

%ecx

(arg 3)

%edx

(arg 1)

%edi

(arg 2)

%esi

%esp

%ebp

CF

ZF

SF

OF

## x86-64 Instruction: set??

### x86-64

### C Pseudocode

<code>sete %r</code>	$\leftrightarrow$	<code>%r = ZF</code>
<code>setne %r</code>	$\leftrightarrow$	<code>%r = ~ZF</code>
<code>sets %r</code>	$\leftrightarrow$	<code>%r = SF</code>
<code>setns %r</code>	$\leftrightarrow$	<code>%r = ~SF</code>

## Pseudocode Translation (so far)

```

1 mystery(%edi):
2   %ecx = 0
3   ZF = %edi == 0
4   SF = %edi < 0
5   %ecx = ~ZF
6   ...
    
```

```
0000000000000000 <mystery>:  
0: 31 c9          xor  %ecx,%ecx  
2: 85 ff          test %edi,%edi  
4: 0f 95 c1       setne %cl  
7: b8 ff ff ff ff  mov  $0xffffffff,%eax  
c: 0f 49 c1       cmovns %ecx,%eax  
f: c3           retq
```

```

0000000000000000 <mystery>:
 0: 31 c9          xor  %ecx,%ecx
 2: 85 ff          test %edi,%edi
 4: 0f 95 c1      setne %cl
 7: b8 ff ff ff ff  mov  $0xffffffff,%eax
 c: 0f 49 c1      cmovns %ecx,%eax
 f: c3          retq
    
```

## Pseudocode Translation (so far)

```

1 mystery(%edi):
2   %ecx = 0
3   ZF = %edi == 0
4   SF = %edi < 0
5   %ecx = ~ZF
6   %eax = -1
7   ...
    
```

(return)	%eax
	%ebx
(arg 4)	%ecx
(arg 3)	%edx
(arg 1)	%edi
(arg 2)	%esi
	%esp
	%ebp
	CF
	ZF
	SF
	OF



```

0000000000000000 <mystery>:
 0: 31 c9          xor  %ecx,%ecx
 2: 85 ff          test %edi,%edi
 4: 0f 95 c1       setne %cl
 7: b8 ff ff ff ff mov  $0xffffffff,%eax
 c: 0f 49 c1       cmovns %ecx,%eax
 f: c3           retq

```

### x86-64 Instruction: cmov??

#### x86-64

**cmove** %src, %dst

↔

%dst = %src **if ZF**

**cmovne** %src, %dst

↔

%dst = %src **if ~ZF**

**cmovs** %src, %dst

↔

%dst = %src **if SF**

**cmovns** %src, %dst

↔

%dst = %src **if ~SF**

#### C Pseudocode

```

0000000000000000 <mystery>:
 0: 31 c9          xor  %ecx,%ecx
 2: 85 ff          test %edi,%edi
 4: 0f 95 c1      setne %cl
 7: b8 ff ff ff ff mov  $0xffffffff,%eax
 c: 0f 49 c1      cmovns %ecx,%eax
 f: c3          retq
    
```

(return)

%eax

%ebx

(arg 4)

%ecx

(arg 3)

%edx

(arg 1)

%edi

(arg 2)

%esi

%esp

%ebp

CF

ZF

SF

OF

## x86-64 Instruction: cmov??

### x86-64

### C Pseudocode

<b>cmove</b> %src, %dst	↔	%dst = %src <b>if</b> ZF
<b>cmovne</b> %src, %dst	↔	%dst = %src <b>if</b> ~ZF
<b>cmovs</b> %src, %dst	↔	%dst = %src <b>if</b> SF
<b>cmovns</b> %src, %dst	↔	%dst = %src <b>if</b> ~SF

## Pseudocode Translation (so far)

```

1 mystery(%edi):
2   %ecx = 0
3   ZF = %edi == 0
4   SF = %edi < 0
5   %ecx = ~ZF
6   %eax = -1
7   if (~SF) { %eax = %ecx }
8   ...
    
```

```
0000000000000000 <mystery>:  
0: 31 c9          xor  %ecx,%ecx  
2: 85 ff          test %edi,%edi  
4: 0f 95 c1       setne %cl  
7: b8 ff ff ff ff mov  $0xffffffff,%eax  
c: 0f 49 c1       cmovns %ecx,%eax  
f: c3            retq
```

```

0000000000000000 <mystery>:
 0: 31 c9          xor  %ecx,%ecx
 2: 85 ff          test %edi,%edi
 4: 0f 95 c1       setne %cl
 7: b8 ff ff ff ff mov  $0xffffffff,%eax
 c: 0f 49 c1       cmovns %ecx,%eax
 f: c3           retq
    
```

(return)	%eax
	%ebx
(arg 4)	%ecx
(arg 3)	%edx
(arg 1)	%edi
(arg 2)	%esi
	%esp
	%ebp
	CF
	ZF
	SF
	OF

## Pseudocode Translation (so far)

```

1 mystery(%edi):
2     %ecx = 0
3     ZF = %edi == 0
4     SF = %edi < 0
5     %ecx = ~ZF
6     %eax = -1
7     if (~SF) { %eax = %ecx }
8     return %eax
    
```

# Another Mystery Function (simplifying)

22

```
0000000000000000 <mystery>:  
 0: 31 c9          xor  %ecx,%ecx  
 2: 85 ff          test %edi,%edi  
 4: 0f 95 c1       setne %cl  
 7: b8 ff ff ff ff mov  $0xffffffff,%eax  
c: 0f 49 c1       cmovns %ecx,%eax  
f: c3            retq
```

## Pseudocode Translation (so far)

```
1 mystery(%edi):  
2   %eax = -1  
3   if (%edi >= 0) { %eax = %edi != 0 }  
4   return %eax
```

(return) %eax

%ebx

(arg 4) %ecx

(arg 3) %edx

(arg 1) %edi

(arg 2) %esi

%esp

%ebp

CF ZF SF OF