



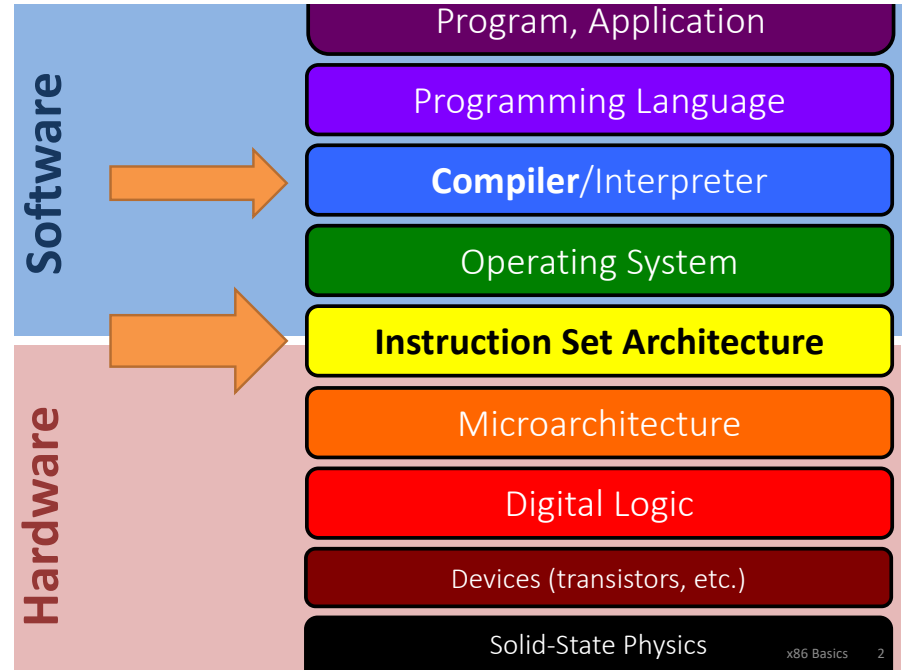
# x86 Basics

Translation tools: C -> assembly <-> machine code

x86 registers, data movement instructions,  
memory addressing, arithmetic instructions

CSAPP book is **highly useful** and well-aligned with class for the remainder of the course.

<https://cs.wellesley.edu/~cs240/>



## Turning C into Machine Code

C Code

```
void sumstore(long x, long y,
              long *dest) {
    long t = x + y;
    *dest = t;
}
```

sum.c

compiler (CS 301)

gcc -Og -S sum.c

Generated x86 Assembly Code

Human-readable language close to machine code.

```
sum:
    addq %rdi,%rsi
    movq %rsi,(%rdx)
    retq
```

sum.s

assembler

Object Code

```
01010101100010011110010110
00101101000101000011000000
00110100010100001000100010
01111011000101110111000011
```

sum.o

Executable: sum

Resolve references between object files,  
libraries, (re)locate data

linker

## Machine Instruction Example

```
*dest = t;
```

C Code

Store value t where indicated by dest

```
movq %rsi, (%rdx)
```

Assembly Code

Move 8-byte value to memory

t: Register %rsi  
 dest: Register %rdx  
 \*dest: Memory M[%rdx]

```
0x400539: 48 89 32
```

Object Code

3-byte instruction encoding  
Stored at address 0x400539

## Disassembling Object Code

Disassembled by `objdump -d sum`

```
000000000400536 <sumstore>:
 400536: 48 01 fe add %rdi,%rsi
 400539: 48 89 32 mov %rsi,(%rdx)
 40053c: c3      retq
```

```
001011010001010000110000
001101000101000010001000
011110110001011101110000
...
```

Disassembler

Object

Disassembled by GDB

```
0x00400536: 0x000000000400536 <+0>: add %rdi,%rsi
0x48      0x000000000400539 <+3>: mov %rsi,(%rdx)
0x01      0x00000000040053c <+6>: retq
0xfe      $ gdb sum
0x48      (gdb) disassemble sumstore
0x89      (disassemble function)
0x32      (gdb) x/7b sum
0xc3      (examine the 13 bytes starting at sum)
```

x86 Basics 5

## CISC vs. RISC

**x86:** real ISA, widespread

**CISC:** maximalism

Complex Instruction Set Computer  
Many instructions, specialized.  
Variable-size encoding,  
complex/slow decode.  
Gradual accumulation over time.

Original goal:

- humans program in assembly
- or simple compilers generate assembly by template
- hardware supports many patterns as single instructions
- fewer instructions per SLOC

Usually fewer registers.

We will stick to a small subset.

**HW:** toy, but based on real MIPS ISA

**RISC:** minimalism

Reduced Instruction Set Computer  
Few instructions, general.  
Regular encoding,  
simple/fast decode.

1980s+ reaction to bloated ISAs.

Original goal:

- humans use high-level languages
- smart compilers generate highly optimized assembly
- hardware supports fast basic instructions
- more instructions per SLOC

Usually many registers.

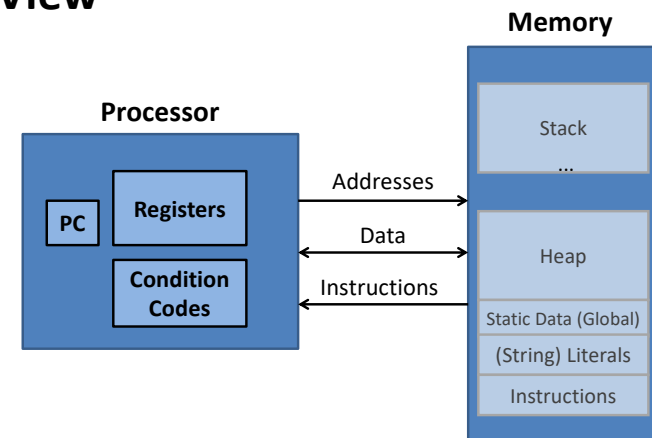
x86 Basics 6

## a brief history of x86

Word Size	ISA	First	Year
16	8086	Intel 8086	1978
	First 16-bit processor. Basis for IBM PC & DOS 1MB address space		
32	IA32	Intel 386	1985
	First 32-bit ISA. Flat addressing, improved OS support		
240 now: 64	x86-64	AMD Opteron	2003*
	Slow AMD/Intel conversion, slow adoption. *Not actually x86-64 until few years later. Mainstream only after ~10 years.		

x86 Basics 7

## ISA View



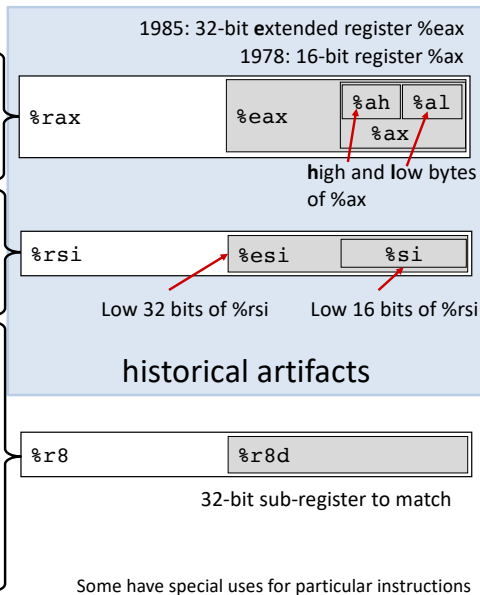
x86 Basics 8

## x86-64 registers

%rax	Return Value
%rbx	
%rcx	Argument 4
%rdx	Argument 3
%rsi	Argument 2
%rdi	Argument 1
%rsp	Special Purpose: Stack Pointer
%rbp	Special Purpose: Base Pointer
%r8	Argument 5
%r9	Argument 6
%r10	
%r11	
%r12	
%r13	
%r14	
%r15	

64-bits / 8 bytes

## sub-registers



x86 Basics 9

## x86-64 registers: function arguments and return value

%rax	Return Value
%rbx	
%rcx	Argument 4
%rdx	Argument 3
%rsi	Argument 2
%rdi	Argument 1
%rsp	Special Purpose: Stack Pointer
%rbp	Special Purpose: Base Pointer
%r8	Argument 5
%r9	Argument 6
%r10	
%r11	
%r12	
%r13	
%r14	
%r15	

64-bits / 8 bytes

## and return value

Mnemonic:

	register
Diana's	%rdi
silk	%rsi
dress	%rdx
costs	%rcx
\$89	%r8
	%9

Arguments 7 and above are passed via stack, not in registers.

x86 Basics 10

## x86: Three Basic Kinds of Instructions

### 1. Data movement between memory and register

**Load** data from memory into register

$\%reg \leftarrow Mem[address]$

**Store** register data into memory

$Mem[address] \leftarrow \%reg$

Memory is an array[] of bytes!

### 2. Arithmetic/logic on register or memory data

$c = a + b;$      $z = x \ll y;$      $i = h \& g;$

### 3. Comparisons and Control flow to choose next instruction

Unconditional jumps to/from procedures

Conditional branches

x86 Basics 11

## Data movement instructions

`mov _ Source, Dest`

data size \_ is one of {b, w, l, q}

`movq`: move 8-byte "quad word"

`movl`: move 4-byte "long word"

`movw`: move 2-byte "word"

`movb`: move 1-byte "byte"

Historical terms based on the 16-bit days, not the current machine word size (64 bits)

### Source/Dest operand types:

**Immediate**: Literal integer data

Examples:  $\$0x400$      $\$-533$

**Register**: One of 16 registers

Examples: `%rax`    `%rdx`

**Memory**: consecutive bytes in memory, at address held by register

Direct addressing:     $(\%rax)$

With displacement/offset:  $8(\%rsp)$

x86 Basics 12

## mov Operand Combinations

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	movq \$0x4,%rax	a = 0x4;
		Mem	movq \$-147,(%rax)	*p = -147;
	Reg	Reg	movq %rax,%rdx	d = a;
		Mem	movq %rax,(%rdx)	*q = a;
	Mem	Reg	movq (%rax),%rdx	d = *p;

**Cannot do memory-memory transfer with a single instruction.**  
How would you do it?

## Memory Addressing Modes

**Indirect** (R) Mem[Reg[R]]  
Register R specifies memory address: `movq (%rcx), %rax`

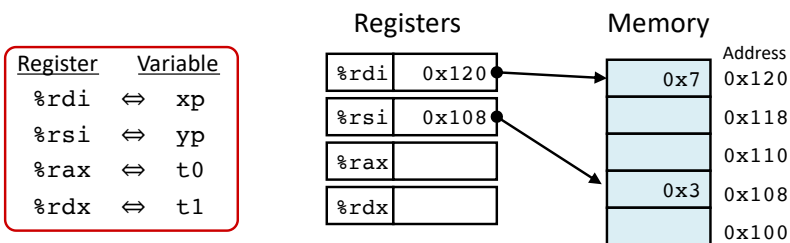
**Displacement** D(R) Mem[Reg[R]+D]  
**Register R** specifies **base** memory address (e.g. base of an object)  
**Displacement D** specifies literal **offset** (e.g. a field in the object)  
`movq %rdx, 8(%rsp)`

**General Form:** D(Rb,Ri,S) Mem[Reg[Rb] + S\*Reg[Ri] + D]  
D: Literal "displacement" value represented in 1, 2, or 4 bytes  
Rb: Base register: Any register  
Ri: Index register: Any except %rsp  
S: Scale: 1, 2, 4, or 8

## Pointers and Memory Addressing

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

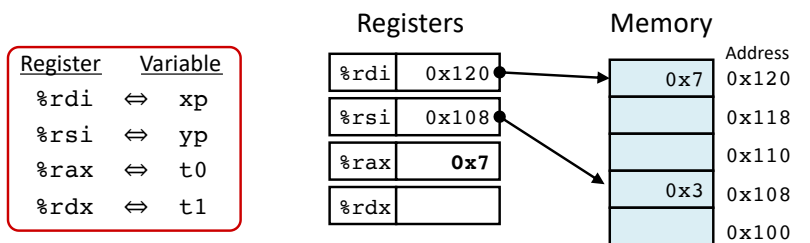
swap:
    movq (%rdi),%rax
    movq (%rsi),%rdx
    movq %rdx,(%rdi)
    movq %rax,(%rsi)
    retq
```



## Pointers and Memory Addressing

```
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)
    retq
```

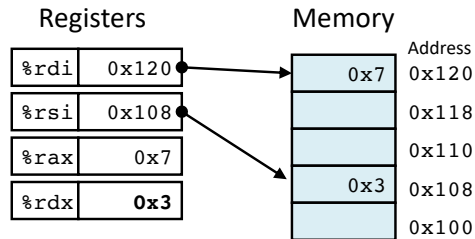


## Pointers and Memory Addressing

```
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)
    retq
```

Register	Variable
%rdi	↔ xp
%rsi	↔ yp
%rax	↔ t0
%rdx	↔ t1



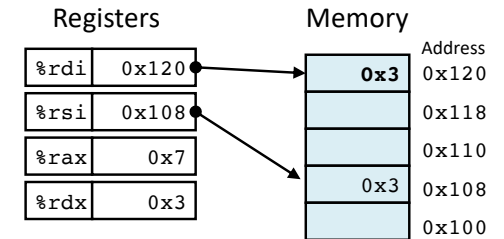
x86 Basics 17

## Pointers and Memory Addressing

```
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)
    retq
```

Register	Variable
%rdi	↔ xp
%rsi	↔ yp
%rax	↔ t0
%rdx	↔ t1



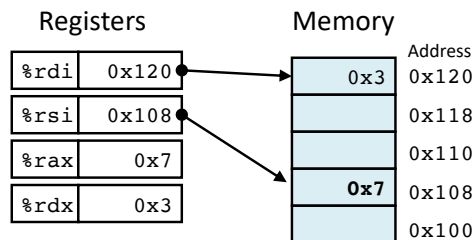
x86 Basics 18

## Pointers and Memory Addressing

```
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)
    retq
```

Register	Variable
%rdi	↔ xp
%rsi	↔ yp
%rax	↔ t0
%rdx	↔ t1



x86 Basics 19

## Address Computation Examples

ex

### General Addressing Modes

$$D(Rb,Ri,S) \text{ Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri] + D]$$

#### Register contents

%rdx	0xf000
%rcx	0x100

#### Special Cases:

Special Cases:	Implicitly:
(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]	(S=1,D=0)
D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D]	(S=1)
(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]	(D=0)

Address Expression	Address Computation	Address
0x8(%rdx)	0x8 + 0xf000	
(%rdx,%rcx)		
(%rdx,%rcx,4)		
0x80(,%rdx,2)		

x86 Basics 20

# Address Computation Examples



## General Addressing Modes

$$D(Rb, Ri, S) \text{ Mem}[\text{Reg}[Rb] + S * \text{Reg}[Ri] + D]$$

Register contents

<code>%rdx</code>	0xf000
<code>%rcx</code>	0x100

<b>Special Cases:</b>		<b>Implicitly:</b>
(Rb, Ri)	Mem[Reg[Rb]+Reg[Ri]]	(S=1, D=0)
D(Rb, Ri)	Mem[Reg[Rb]+Reg[Ri]+D]	(S=1)
(Rb, Ri, S)	Mem[Reg[Rb]+S*Reg[Ri]]	(D=0)

Address Expression	Address Computation	Address
<code>0x8(%rdx)</code>	$0x8 + 0xf000$	0xf008
<code>(%rdx,%rcx)</code>	$0xf000 + 0x100 * 1$	0xf100
<code>(%rdx,%rcx,4)</code>	$0xf000 + 0x100 * 4$	0xf400
<code>0x80(,%rdx,2)</code>	$0x80 + 0x0 + 0xf000 * 2$	0x1e080

Compute address given by this addressing mode expression and store it here.

# Load effective address

`leaq Src, Dest`

DOES NOT ACCESS MEMORY



Uses: "address of" "Lovely Efficient Arithmetic"

$$p = \&x[i]; \quad x + k * I, \text{ where } k = 1, 2, 4, \text{ or } 8$$

## leaq vs. movq

Registers	Memory	Address	Assembly Code
<code>%rax</code>	0x400	0x120	<code>leaq (%rdx,%rcx,4), %rax</code>
<code>%rbx</code>	0xf	0x118	<code>movq (%rdx,%rcx,4), %rbx</code>
<code>%rcx</code> 0x4	0x8	0x110	<code>leaq (%rdx), %rdi</code>
<code>%rdx</code> 0x100	0x10	0x108	<code>movq (%rdx), %rsi</code>
<code>%rdi</code>	0x1	0x100	
<code>%rsi</code>			

Compute address given by this addressing mode expression and store it here.

# Load effective address

`leaq Src, Dest`

DOES NOT ACCESS MEMORY



Uses: "address of" "Lovely Efficient Arithmetic"

$$p = \&x[i]; \quad x + k * I, \text{ where } k = 1, 2, 4, \text{ or } 8$$

## leaq vs. movq

Registers	Memory	Address	Assembly Code
<code>%rax</code> 0x110	0x400	0x120	<code>leaq (%rdx,%rcx,4), %rax</code>
<code>%rbx</code>	0xf	0x118	<code>movq (%rdx,%rcx,4), %rbx</code>
<code>%rcx</code> 0x4	0x8	0x110	<code>leaq (%rdx), %rdi</code>
<code>%rdx</code> 0x100	0x10	0x108	<code>movq (%rdx), %rsi</code>
<code>%rdi</code>	0x1	0x100	
<code>%rsi</code>			

Compute address given by this addressing mode expression and store it here.

# Load effective address

`leaq Src, Dest`

DOES NOT ACCESS MEMORY



Uses: "address of" "Lovely Efficient Arithmetic"

$$p = \&x[i]; \quad x + k * I, \text{ where } k = 1, 2, 4, \text{ or } 8$$

## leaq vs. movq

Registers	Memory	Address	Assembly Code
<code>%rax</code> 0x110	0x400	0x120	<code>leaq (%rdx,%rcx,4), %rax</code>
<code>%rbx</code> 0x8	0xf	0x118	<code>movq (%rdx,%rcx,4), %rbx</code>
<code>%rcx</code> 0x4	0x8	0x110	<code>leaq (%rdx), %rdi</code>
<code>%rdx</code> 0x100	0x10	0x108	<code>movq (%rdx), %rsi</code>
<code>%rdi</code>	0x1	0x100	
<code>%rsi</code>			

Compute address given by this addressing mode expression and store it here.

## Load effective address

`leaq Src, Dest`

DOES NOT ACCESS MEMORY



Uses: "address of" "Lovely Efficient Arithmetic"

`p = &x[i];`     `x + k*I`, where `k = 1, 2, 4, or 8`

### leaq vs. movq

Registers	Memory	Address	Assembly Code
<code>%rax</code> 0x110	0x400	0x120	<code>leaq (%rdx,%rcx,4), %rax</code>
<code>%rbx</code> 0x8	0xf	0x118	<code>movq (%rdx,%rcx,4), %rbx</code>
<code>%rcx</code> 0x4	0x8	0x110	<b><code>leaq (%rdx), %rdi</code></b>
<code>%rdx</code> 0x100	0x10	0x108	<code>movq (%rdx), %rsi</code>
<code>%rdi</code> 0x100	0x1	0x100	
<code>%rsi</code>			

Compute address given by this addressing mode expression and store it here.

## Load effective address

`leaq Src, Dest`

DOES NOT ACCESS MEMORY



Uses: "address of" "Lovely Efficient Arithmetic"

`p = &x[i];`     `x + k*I`, where `k = 1, 2, 4, or 8`

### leaq vs. movq

Registers	Memory	Address	Assembly Code
<code>%rax</code> 0x110	0x400	0x120	<code>leaq (%rdx,%rcx,4), %rax</code>
<code>%rbx</code> 0x8	0xf	0x118	<code>movq (%rdx,%rcx,4), %rbx</code>
<code>%rcx</code> 0x4	0x8	0x110	<code>leaq (%rdx), %rdi</code>
<code>%rdx</code> 0x100	0x10	0x108	<b><code>movq (%rdx), %rsi</code></b>
<code>%rdi</code> 0x100	0x1	0x100	
<code>%rsi</code> 0x1			

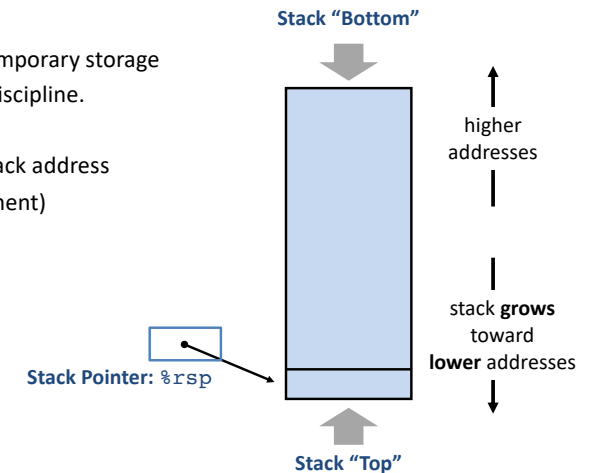
## Memory address-space layout

Addr	Perm	Contents	Managed by	Initialized
2 <sup>N</sup> -1	RW	Procedure context	Compiler	Run time
↑	RW	Dynamic data structures	Programmer, malloc/free, new/GC	Run time
↑	RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
↑	R	String literals	Compiler/ Assembler/Linker	Startup
↑	X	Instructions	Compiler/ Assembler/Linker	Startup
0				

## Call Stack

Memory region for temporary storage managed with stack discipline.

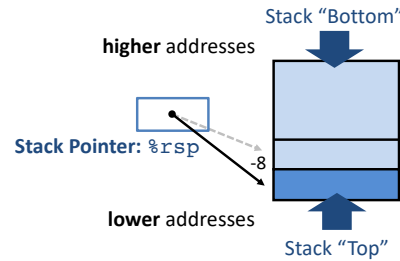
`%rsp` holds lowest stack address (address of "top" element)



## Call Stack: Push, Pop

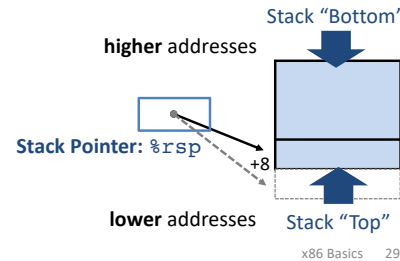
`pushq Src`

1. Fetch value from `Src`
2. Decrement `%rsp` by 8 (*why 8?*)
3. Store value at new address given by `%rsp`



`popq Dest`

1. Load value from address `%rsp`
2. Write value to `Dest`
3. Increment `%rsp` by 8



x86 Basics 29

## Procedure Preview (more soon)

**call, ret, push, pop**

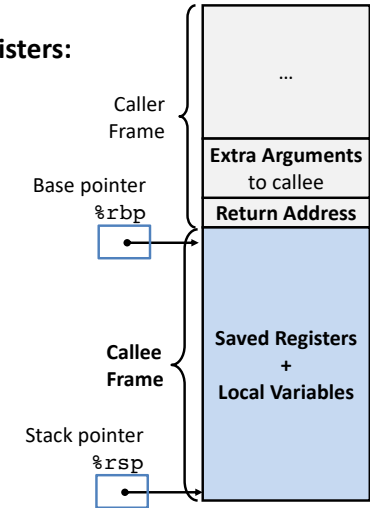
Procedure arguments passed in 6 registers:

<code>%rax</code>	Return Value	<code>%r8</code>	Argument 5
<code>%rbx</code>		<code>%r9</code>	Argument 6
<code>%rcx</code>	Argument 4	<code>%r10</code>	
<code>%rdx</code>	Argument 3	<code>%r11</code>	
<code>%rsi</code>	Argument 2	<code>%r12</code>	
<code>%rdi</code>	Argument 1	<code>%r13</code>	
<code>%rsp</code>	Stack pointer	<code>%r14</code>	
<code>%rbp</code>	Base pointer	<code>%r15</code>	

Return value in `%rax`.

Allocate/push new *stack frame* for each procedure call.

Some local variables, saved register values, extra arguments  
Deallocate/pop frame before return.



x86 Basics 30

## x86: Three Basic Kinds of Instructions

1. Data movement between memory and register

**Load** data from memory into register

$\%reg \leftarrow Mem[address]$

**Store** register data into memory

$Mem[address] \leftarrow \%reg$

Memory is an array[] of bytes!

2. Arithmetic/logic on register or memory data

$c = a + b; \quad z = x \ll y; \quad i = h \& g;$

3. Comparisons and Control flow to choose next instruction

Unconditional jumps to/from procedures

Conditional branches

x86 Basics 31

## Arithmetic Operations

Two-operand instructions:

**Format**

`addq Src, Dest`  
`subq Src, Dest`  
`imulq Src, Dest`  
`shlq Src, Dest`  
`sarq Src, Dest`  
`shrq Src, Dest`  
`xorq Src, Dest`  
`andq Src, Dest`  
`orq Src, Dest`

**Computation**

$Dest = Dest + Src$   
 $Dest = Dest - Src$   
 $Dest = Dest * Src$   
 $Dest = Dest \ll Src$   
 $Dest = Dest \gg Src$   
 $Dest = Dest \gg Src$   
 $Dest = Dest \wedge Src$   
 $Dest = Dest \& Src$   
 $Dest = Dest / Src$

← **argument order**

*a.k.a salq  
Arithmetic  
Logical*

One-operand (unary) instructions

`incq Dest`  
`decq Dest`  
`negq Dest`  
`notq Dest`

$Dest = Dest + 1$   
 $Dest = Dest - 1$   
 $Dest = -Dest$   
 $Dest = \sim Dest$

increment  
 decrement  
 negate  
 bitwise complement

See CSAPP 3.5.5 for: `mulq`, `cqto`, `idivq`, `divq`

x86 Basics 32



## leaq for arithmetic

```
long arith(long x, long y,
           long z){
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	
%rcx	

```
arith:
    leaq    (%rdi,%rsi), %rax
    addq   %rdx, %rax
    leaq   (%rsi,%rsi,2), %rdx
    salq   $4, %rdx
    leaq   4(%rdi,%rdx), %rcx
    imulq  %rcx, %rax
    ret
```

## leaq for arithmetic

```
long arith(long x, long y,
           long z){
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	t1
%rcx	

```
arith:
    leaq    (%rdi,%rsi), %rax
    addq   %rdx, %rax
    leaq   (%rsi,%rsi,2), %rdx
    salq   $4, %rdx
    leaq   4(%rdi,%rdx), %rcx
    imulq  %rcx, %rax
    ret
```

## leaq for arithmetic

```
long arith(long x, long y,
           long z){
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	<del>t1</del> t2
%rcx	

```
arith:
    leaq    (%rdi,%rsi), %rax
    addq   %rdx, %rax
    leaq   (%rsi,%rsi,2), %rdx
    salq   $4, %rdx
    leaq   4(%rdi,%rdx), %rcx
    imulq  %rcx, %rax
    ret
```

## leaq for arithmetic

```
long arith(long x, long y,
           long z){
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z 3*y
%rax	<del>t1</del> t2
%rcx	

```
arith:
    leaq    (%rdi,%rsi), %rax
    addq   %rdx, %rax
    leaq   (%rsi,%rsi,2), %rdx
    salq   $4, %rdx
    leaq   4(%rdi,%rdx), %rcx
    imulq  %rcx, %rax
    ret
```

## leaq for arithmetic

```
long arith(long x, long y,
           long z){
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z $3*y$ t4
%rax	<del>t1</del> t2
%rcx	

```
arith:
    leaq    (%rdi,%rsi), %rax
    addq   %rdx, %rax
    leaq   (%rsi,%rsi,2), %rdx
    salq  $4, %rdx
    leaq   4(%rdi,%rdx), %rcx
    imulq %rcx, %rax
    ret
```

## leaq for arithmetic

```
long arith(long x, long y,
           long z){
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z $3*y$ t4
%rax	<del>t1</del> t2
%rcx	4+x+t4 = t5

```
arith:
    leaq    (%rdi,%rsi), %rax
    addq   %rdx, %rax
    leaq   (%rsi,%rsi,2), %rdx
    salq  $4, %rdx
    leaq   4(%rdi,%rdx), %rcx
    imulq %rcx, %rax
    ret
```

## leaq for arithmetic

```
long arith(long x, long y,
           long z){
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z $3*y$ t4
%rax	<del>t1</del> <del>t2</del> rval
%rcx	4+x+t4 = t5

```
arith:
    leaq    (%rdi,%rsi), %rax
    addq   %rdx, %rax
    leaq   (%rsi,%rsi,2), %rdx
    salq  $4, %rdx
    leaq   4(%rdi,%rdx), %rcx
    imulq %rcx, %rax
    ret
```

## leaq for arithmetic

```
long arith(long x, long y,
           long z){
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z $3*y$ t4
%rax	<del>t1</del> <del>t2</del> rval
%rcx	4+x+t4 = t5

```
arith:
    leaq    (%rdi,%rsi), %rax
    addq   %rdx, %rax
    leaq   (%rsi,%rsi,2), %rdx
    salq  $4, %rdx
    leaq   4(%rdi,%rdx), %rcx
    imulq %rcx, %rax
    ret
```

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Same x86 code by compiling:  $(x+y+z)*(x+4+48*y)$

## Compiler optimization example

```
long logical(long x, long y){
    long t1 = x^y;
    long t2 = t1 >> 17;
    long mask = (1<<13) - 7;
    long rval = t2 & mask;
    return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	

```
logical:
    movq %rdi,%rax
    xorq %rsi,%rax
    sarq $17,%rax
    andq $8185,%rax
    retq
```

## Compiler optimization example

```
long logical(long x, long y){
    long t1 = x^y;
    long t2 = t1 >> 17;
    long mask = (1<<13) - 7;
    long rval = t2 & mask;
    return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	x

```
logical:
    movq %rdi,%rax
    xorq %rsi,%rax
    sarq $17,%rax
    andq $8185,%rax
    retq
```

## Compiler optimization example

```
long logical(long x, long y){
    long t1 = x^y;
    long t2 = t1 >> 17;
    long mask = (1<<13) - 7;
    long rval = t2 & mask;
    return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	* t1

```
logical:
    movq %rdi,%rax
    xorq %rsi,%rax
    sarq $17,%rax
    andq $8185,%rax
    retq
```

## Compiler optimization example

```
long logical(long x, long y){
    long t1 = x^y;
    long t2 = t1 >> 17;
    long mask = (1<<13) - 7;
    long rval = t2 & mask;
    return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	* <del>t1</del> t2

```
logical:
    movq %rdi,%rax
    xorq %rsi,%rax
    sarq $17,%rax
    andq $8185,%rax
    retq
```

## Compiler optimization example

```
long logical(long x, long y){
  long t1 = x^y;
  long t2 = t1 >> 17;
  long mask = (1<<13) - 7;
  long rval = t2 & mask;
  return rval;
}
```

```
logical:
  movq %rdi,%rax
  xorq %rsi,%rax
  sarq $17,%rax
  andq $8185,%rax
  retq
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	* <del>t1</del> <del>t2</del> rval

## x86: Three Basic Kinds of Instructions

### 1. Data movement between memory and register

**Load** data from memory into register

$\%reg \leftarrow Mem[address]$

**Store** register data into memory

$Mem[address] \leftarrow \%reg$

Memory is an  
array[] of bytes!

### 2. Arithmetic/logic on register or memory data

$c = a + b;$      $z = x \ll y;$      $i = h \& g;$

### Next Time:

### 3. Comparisons and Control flow to choose next instruction

Unconditional jumps to/from procedures

Conditional branches