



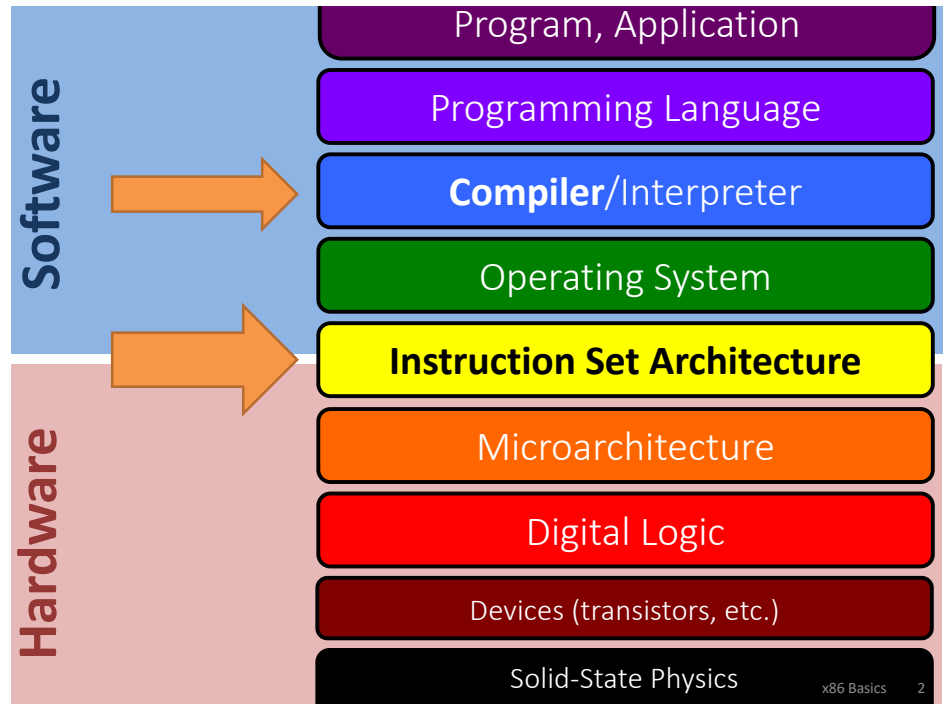
x86 Basics

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

linker

Executable: sum

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

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

```
0000000000400536 <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: 0x0000000000400536 <+0>: add %rdi,%rsi
0x48: 0x0000000000400539 <+3>: mov %rsi,(%rdx)
0x01: 0x000000000040053c <+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)
```

CISC vs. RISC

x86: real ISA, widespread

MW: toy, but based on real MIPS ISA

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.

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

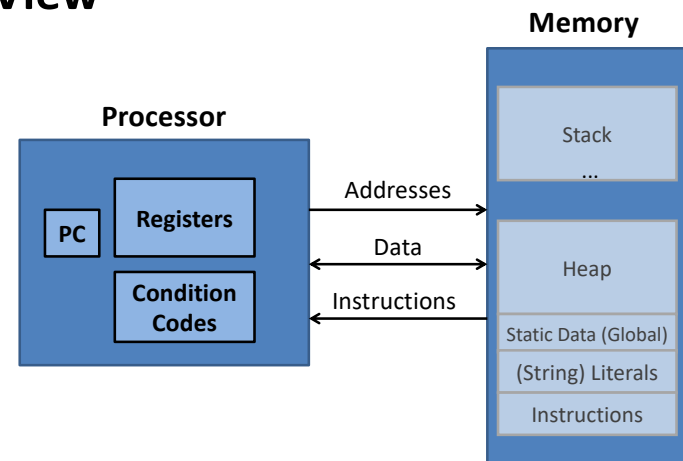
Usually many registers.

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.		

2016: most laptops, desktops, servers. →

ISA View

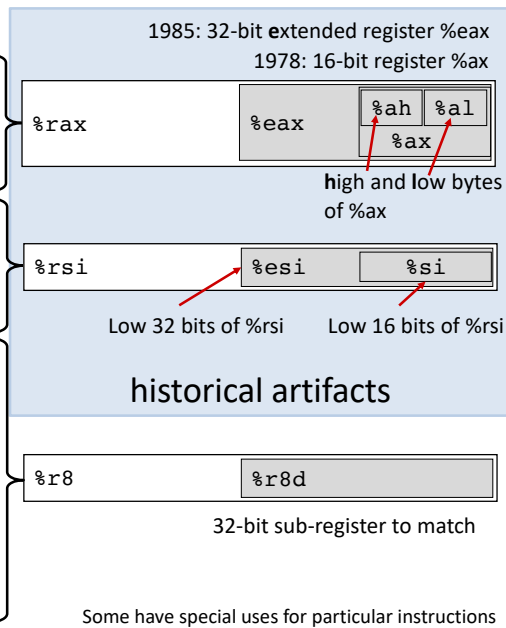


x86-64 registers

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

64-bits / 8 bytes

sub-registers



x86 Basics 9

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 12

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 13

mov Operand Combinations

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

Cannot do memory-memory transfer with a single instruction.

How would you do it?

x86 Basics 14

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

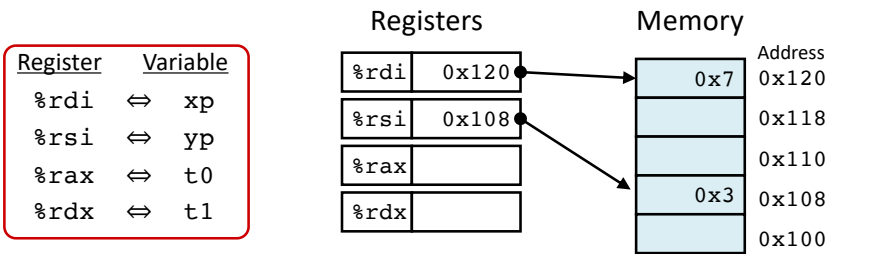
Ri: Index register: Any except %rsp

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;
}
```

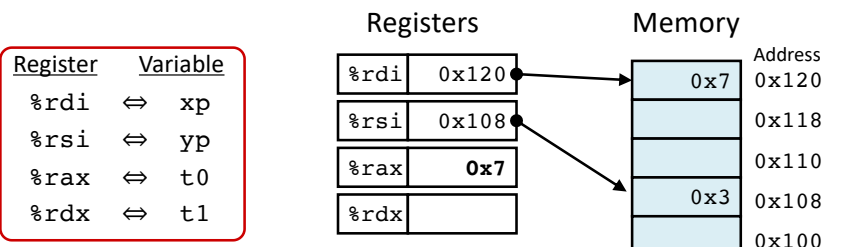
```
swap:
    movq (%rdi),%rax
    movq (%rsi),%rdx
    movq %rdx,(%rdi)
    movq %rax,(%rsi)
    retq
```



Pointers and Memory Addressing

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

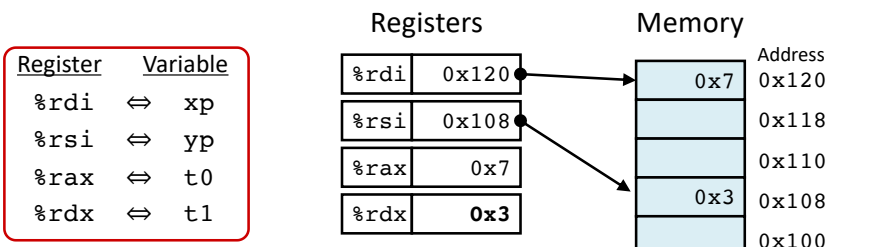
```
swap:
    movq (%rdi),%rax
    movq (%rsi),%rdx
    movq %rdx,(%rdi)
    movq %rax,(%rsi)
    retq
```



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
```

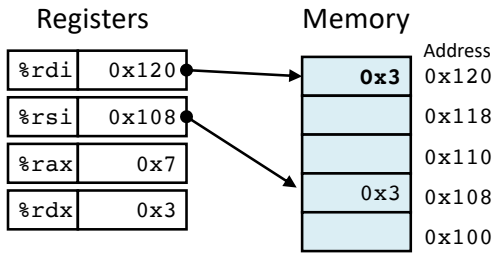


Pointers and Memory Addressing

```
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

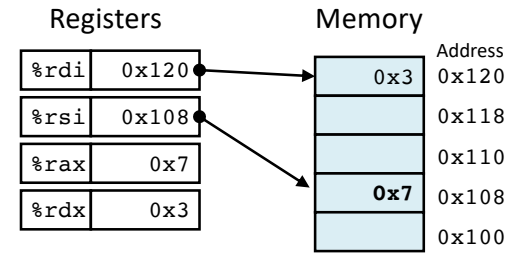


Pointers and Memory Addressing

```
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

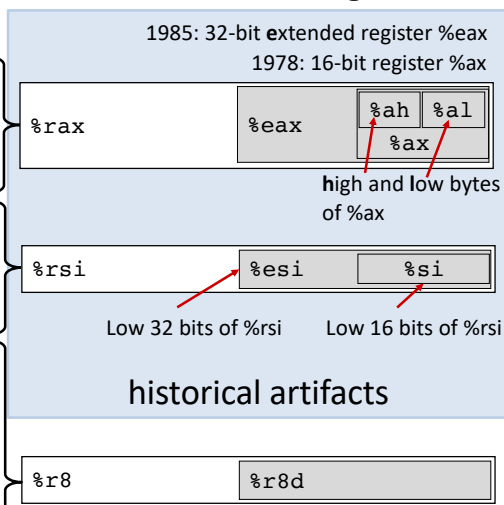


x86-64 registers

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

64-bits / 8 bytes

sub-registers



historical artifacts

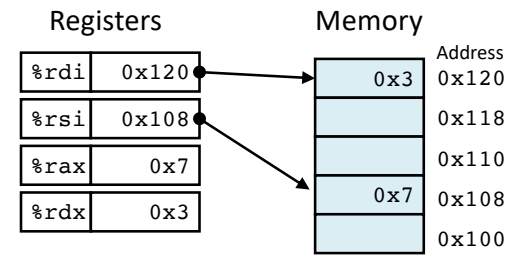
Some have special uses for particular instructions

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



Address Computation Examples



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:		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)		

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

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

Assembly Code

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

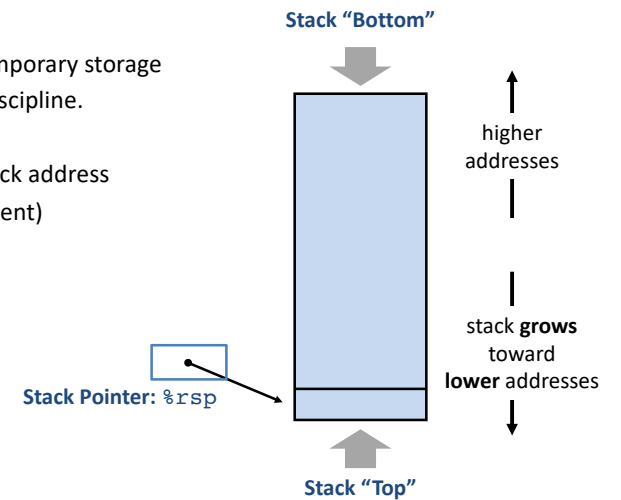
Memory address-space layout

Addr	Perm	Contents	Managed by	Initialized
2 ^N -1 ↑	RW	Procedure context	Compiler	Run time
Stack ↓				
↑	RW	Dynamic data structures	Programmer, malloc/free, new/GC	Run time
Heap ↑				
	RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
Statics				
	R	String literals	Compiler/ Assembler/Linker	Startup
Literals				
	X	Instructions	Compiler/ Assembler/Linker	Startup
Text				
0				

Call Stack

Memory region for temporary storage managed with stack discipline.

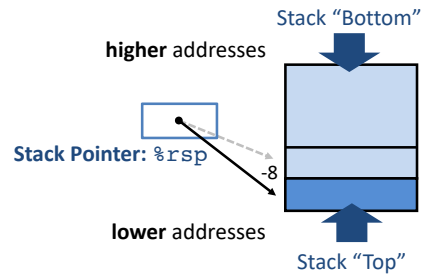
%rsp holds lowest stack address (address of "top" element)



Call Stack: Push, Pop

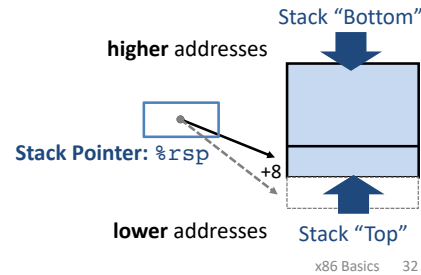
pushq Src

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



Procedure Preview (more soon)

call, ret, push, pop

Procedure arguments passed in 6 registers:

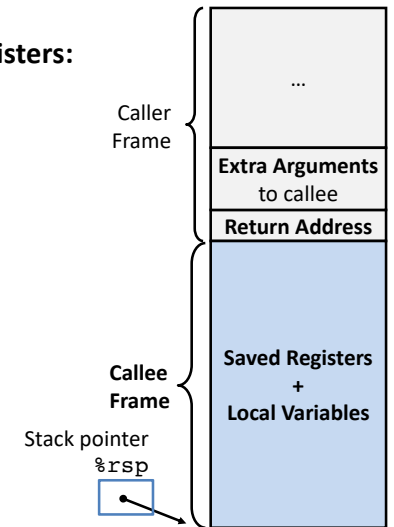
%rax	Return Value	%r8	Argument 5
%rbx		%r9	Argument 6
%rcx	Argument 4	%r10	
%rdx	Argument 3	%r11	
%rsi	Argument 2	%r12	
%rdi	Argument 1	%r13	
%rsp	Stack pointer	%r14	
%rbp		%r15	

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: 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

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 << Src
- Dest = Dest >> Src
- Dest = Dest >> Src
- Dest = Dest ^ Src
- Dest = Dest & Src
- Dest = Dest | Src

← argument order

a.k.a *salq*
Arithmetic
Logical

One-operand (unary) instructions

- incq Dest Dest = Dest + 1
- decq Dest Dest = Dest - 1
- negq Dest Dest = -Dest
- notq Dest Dest = ~Dest

- increment
- decrement
- negate
- bitwise complement

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

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
```

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
```

x86: Three Basic Kinds of Instructions

1. Data movement between memory and register

Load data from memory into register

$\%reg \leftarrow \text{Mem}[\text{address}]$

Store register data into memory

$\text{Mem}[\text{address}] \leftarrow \%reg$

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