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

C to Machine Code and x86 Basics

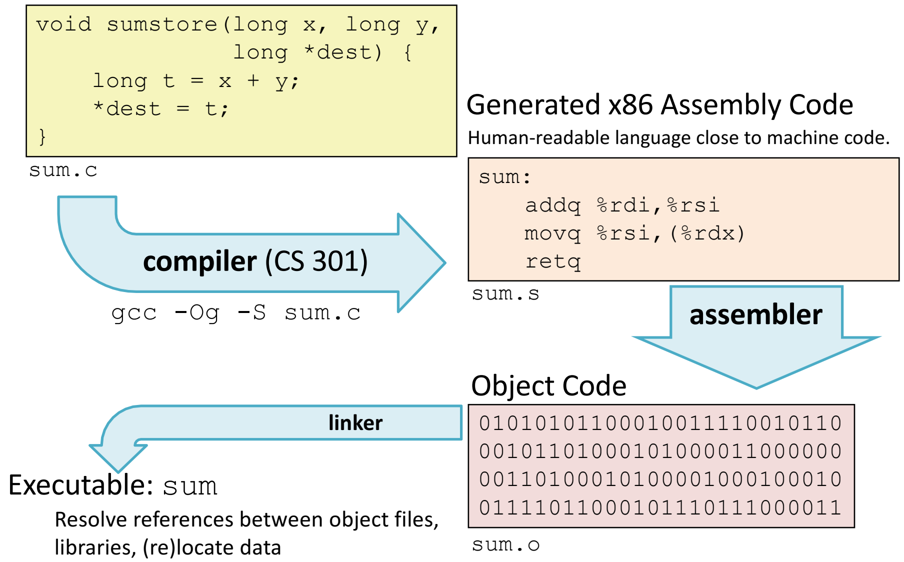
ISA context and x86 history

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

x86 Basics:

- Registers
- Data movement instructions
- Memory addressing modes
- Arithmetic instructions

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

```
0101010110001001111001011 0
```

```
01010101100010011110010110
00101101000101000011000000
00110100010100001000100010
01111011000101110111000011
```

sum.o

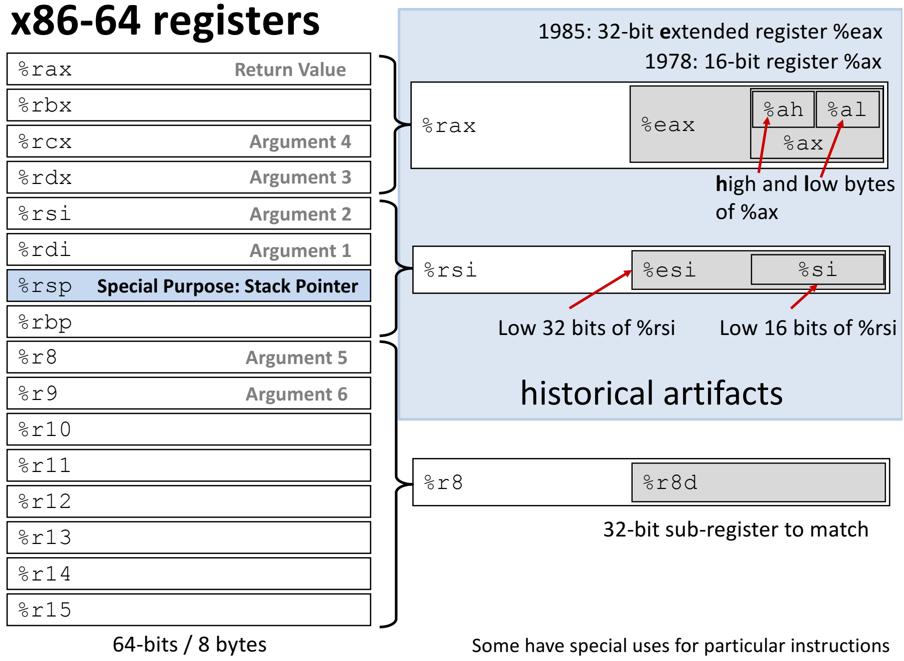
linker

Executable: `sum`

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

---

x86-64 registers



sub-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

1985: 32-bit extended register %eax

1978: 16-bit register %ax

high and low bytes of %ax

Low 32 bits of %rsi

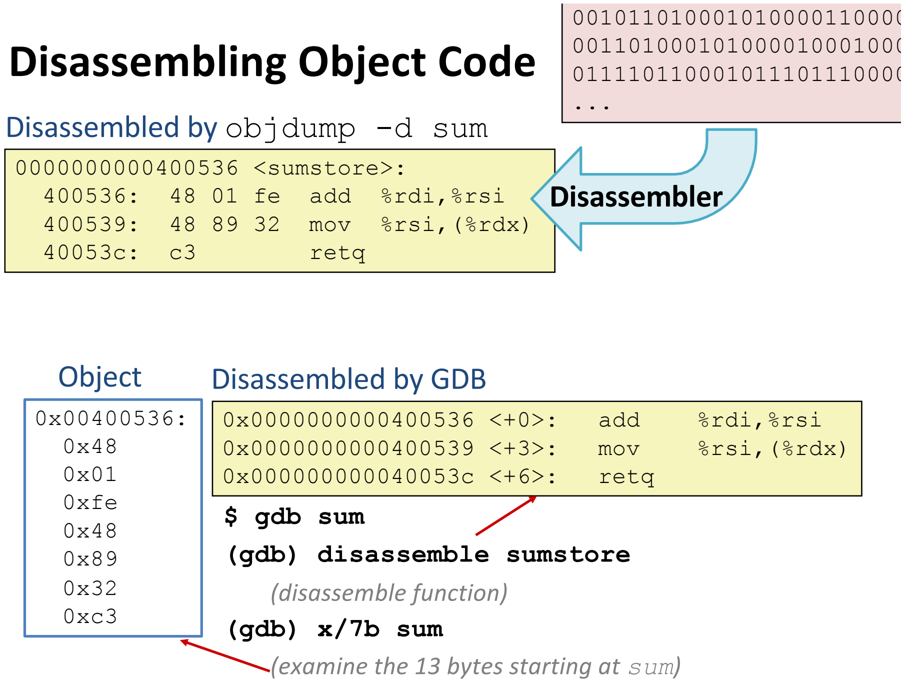
Low 16 bits of %rsi

historical artifacts

32-bit sub-register to match

Some have special uses for particular instructions

Disassembling Object Code



```
00101101000101000011000
00110100010100001000100
01111011000101110111000
...
```

Disassembled by `objdump -d sum`

```
0000000000400536 <sumstore>:
  400536:  48 01 fe   add  %rdi,%rsi
  400539:  48 89 32   mov  %rsi,(%rdx)
  40053c:  c3         retq
```

Disassembler

Object

```
0x00400536:
  0x48
  0x01
  0xfe
  0x48
  0x89
  0x32
  0xc3
```

Disassembled by GDB

```
0x0000000000400536 <+0>:   add    %rdi,%rsi
0x0000000000400539 <+3>:   mov    %rsi,(%rdx)
0x000000000040053c <+6>:   retq
```

```
$ gdb sum
(gdb) disassemble sumstore
```

(disassemble function)

```
(gdb) x/7b sum
```

(examine the 13 bytes starting at `sum`)

x86: Three Basic Kinds of Instructions

1. Data movement between memory and register

- ***Load*** data from memory into register
    - *%reg* ← Mem[*address*]
- ***Store*** register data into memory
    - Mem[*address*] ← *%reg*

Memory is an array[] of bytes!

2. Arithmetic/logic on register or memory data

c = a + b; z = x << y; i = h & g;

3. Comparisons and Control flow to choose next instruction

- Unconditional jumps to/from procedures
- Conditional branches

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)`**

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*]

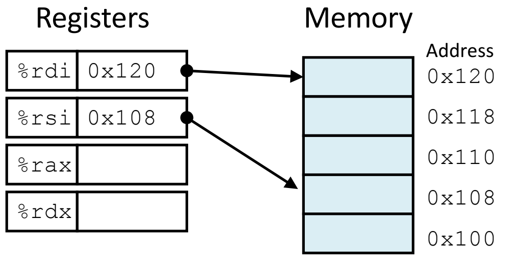
D:	Literal "displacement" value represented in 1, 2, or 4 bytes
Rb:	Base register: Any register
Ri:	Index register: Any except `%rsp`
S:	Scale: 1, 2, 4, or 8

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
```

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



Address Computation Examples

ex

Register contents

`%rdx`	`0xf000`
`%rcx`	`0x100`

General Addressing Modes

D(Rb,Ri,S) Mem[Reg[*Rb*]+*S**Reg[*Ri*] + *D*]

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)`		
`(%rdx,%rcx)`		
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		

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

`leaq` ***Src, Dest***

Load effective address

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

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

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

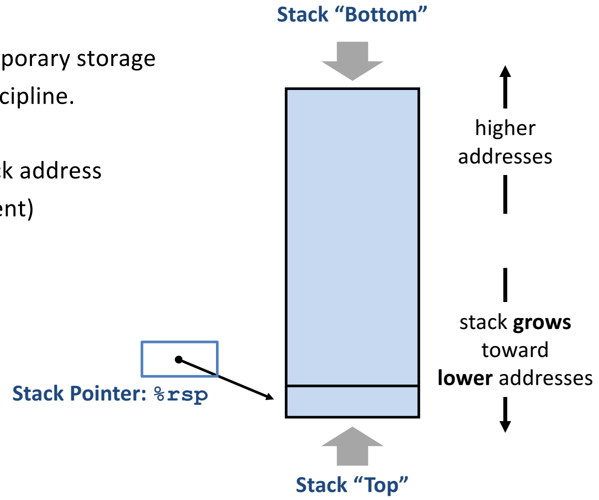
Assembly Code

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

Call **Stack**

Memory region for temporary storage managed with stack discipline.

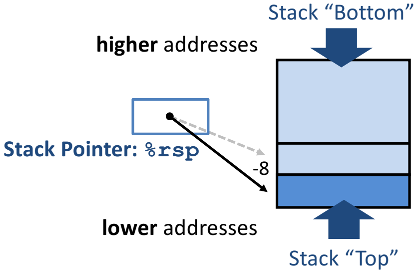
`%rsp` holds lowest stack address (address of "top" element)



Call Stack: **Push, Pop**

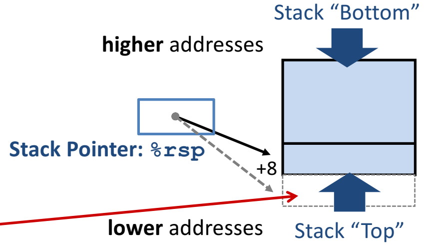
`pushq` ***Src***

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



Those bits are still there; we're just not using them.

Procedure Preview (more soon)

`call, ret, push, pop`

Procedure arguments passed in 6 registers:

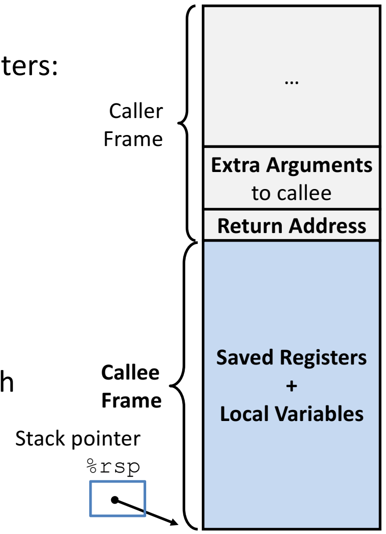
`%rax`	Return Value	`%r8`	Argument 5
`%rbx`		`%r9`	Argument 6
`%rcx`	Argument 4	`%r10`	
`%rdx`	Argument 3	`%r11`	
`%rsi`	Argument 2	`%r12`	
`%rdi`	Argument 1	`%r13`	
`%rsp`	Stack pointer	`%r14`	
`%rbp`		`%r15`	

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.



Arithmetic Operations

Two-operand instructions:

Format		*Computation*	
`addq`	*Src,Dest*	*Dest = Dest + Src*	
`subq`	***Src,Dest***	***Dest = Dest − Src***	***argument order***
`imulq`	*Src,Dest*	*Dest = Dest * Src*	
`shlq`	*Src,Dest*	*Dest = Dest << Src*	*a.k.a salq*
`sarq`	*Src,Dest*	*Dest = Dest >> Src*	*Arithmetic*
`shrq`	*Src,Dest*	*Dest = Dest >> Src*	*Logical*
`xorq`	*Src,Dest*	*Dest = Dest ^ Src*	
`andq`	*Src,Dest*	*Dest = Dest & Src*	
`orq`	*Src,Dest*	*Dest = Dest | Src*	

One-operand (unary) instructions

`incq`	*Dest*	*Dest = Dest + 1*	increment
`decq`	*Dest*	*Dest = Dest − 1*	decrement
`negq`	*Dest*	*Dest = −Dest*	*negate*
`notq`	*Dest*	*Dest = ~Dest*	*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
```

Another 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
```