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

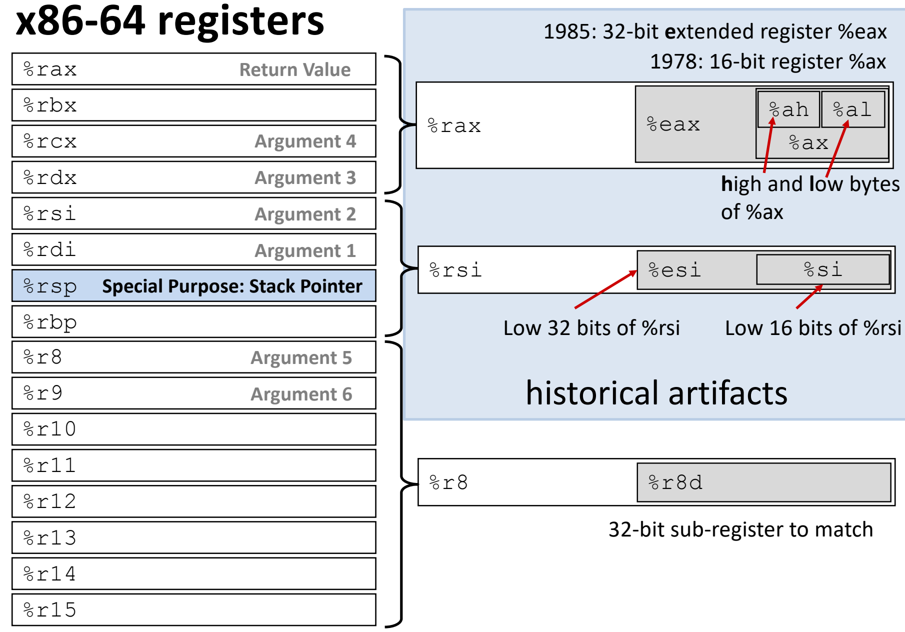
Translation tools: C --> assembly <--> machine code

x86 Basics:

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

x86-64 registers

sub-registers



64-bits / 8 bytes

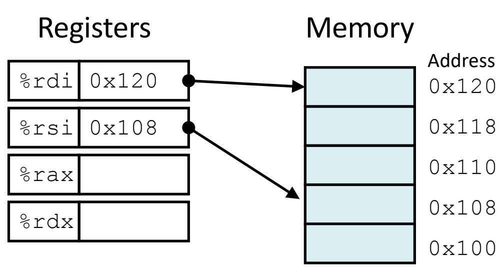
Some have special uses for particular instructions

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

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)		

Load effective address

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

`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

%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
```


Procedure Preview (more soon)

call, ret, push, pop

Procedure arguments passed in 6 registers:

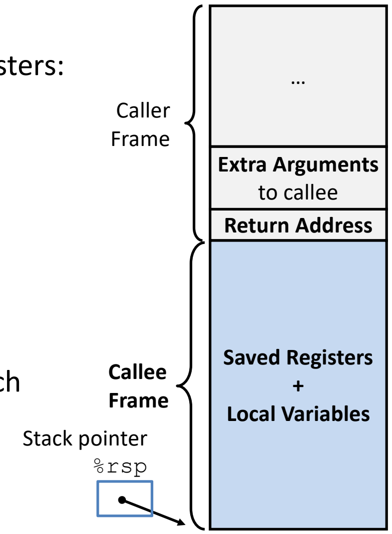
%rax	Return Value	%r8	Argument 5
%rbx		%r9	Argument 6
%rcx	Argument 4	%r10	
%rdx	Argument 3	%r11	
%rsi	Argument 2	%r12	
%rdi	Argument 1	%r13	
%rsp	Stack pointer	%r14	
%rbp		%r15	

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
```