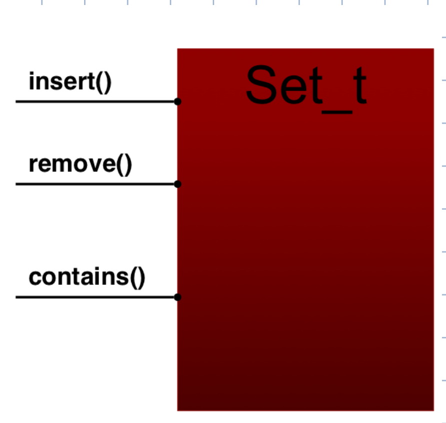
instruction set architecture ISA
used to talk to CPU

interface:

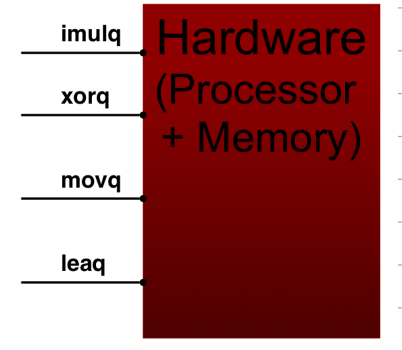


make up a data structure

state - data
methods - operations on data

methods are interface for data object

architecture is interface between software and hardware



state - contents of memory
methods - change memory

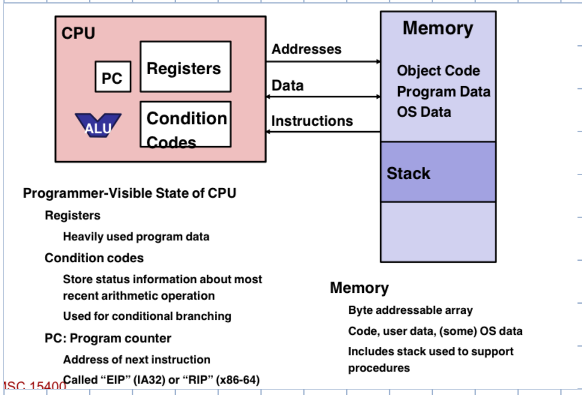
```
void swap( long *xp, long *yp)
    long t0 = *xp
    long t1 = *yp
   *xp = t1
   *yp = t0
```

compile to Assembly code

IBM 360

interface is tough: addresses compatability, but changing architecture is tuff

x86 - 64



Programmer-Visible State of CPU

- Registers
    - Heavily used program data
- Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
- PC: Program counter
    - Address of next instruction
    - Called "EIP" (IA32) or "RIP" (x86-64)

Memory

- Byte addressable array
- Code, user data, (some) OS data
- Includes stack used to support procedures

ISC 15400

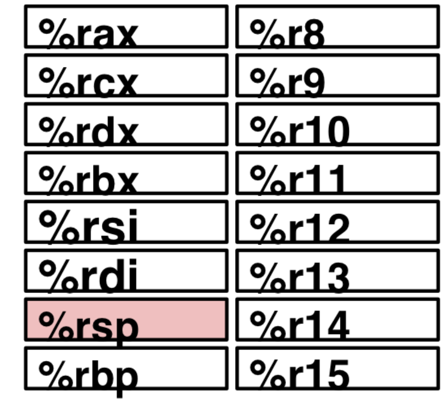
anything w/ % → register

registers hold highly used program data

16 registers (64bit-wide) in x86-64

↳ 8 bytes, 64bit wordsize

general purpose registers



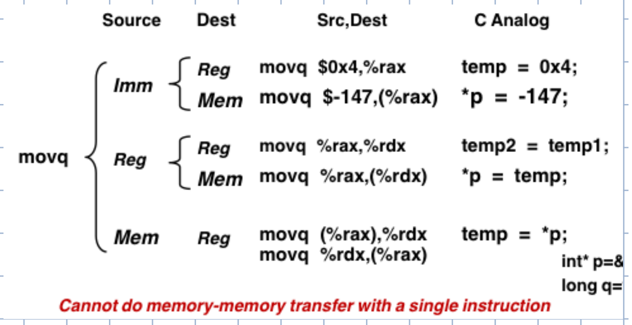
%rax	%r8
%rcx	%r9
%rdx	%r10
%rbx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

movq source, dest

immediate: constant integer data

register: one of 16 int. registers

memory: 8 bytes @ address given by register



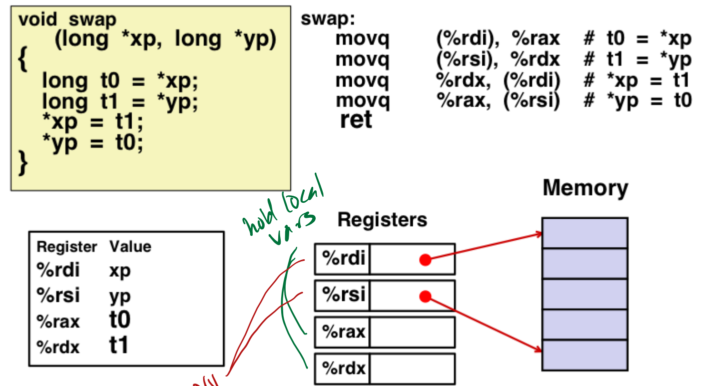
	Source	Dest	Src,Dest	C Analog
movq	Imm	Reg	movq $0x4,%rax	temp = 0x4;
		Mem	movq $-147,(%rax)	*p = -147;
	Reg	Reg	movq %rax,%rdx	temp2 = temp1;
		Mem	movq %rax,(%rdx)	*p = temp;
	Mem	Reg	movq (%rax),%rdx movq %rdx,(%rax)	temp = *p;

int* p=&
long q=

Cannot do memory-memory transfer with a single instruction

general rules

- must have 1 source & 1 dest.
- source before dest
- source not modified
- destination is modified
- dest. may be both operand & result
- at most 1 of source, dest can be memory



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

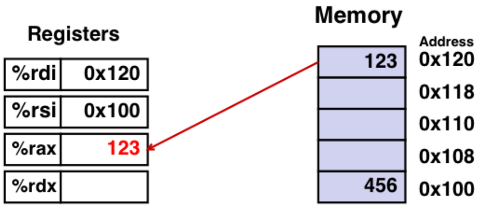
```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

hold local vars

not always 1 to 1

memory

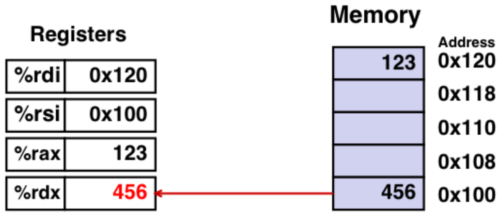


copy & save #0x120 & * 0x100

Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

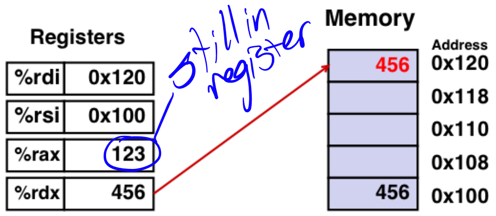
```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



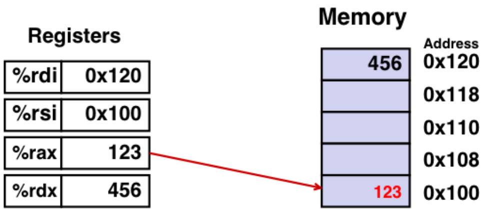
Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```



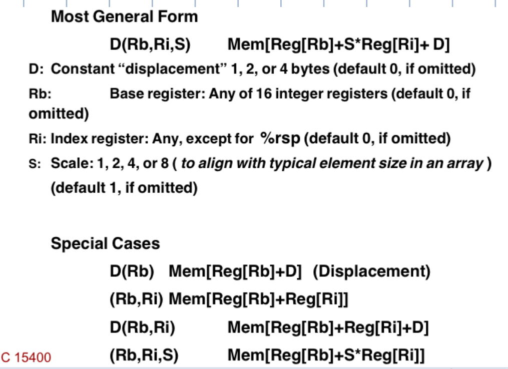
Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

ist helps w/other memory access mode →

normal: Mem[Reg[R]]

pointing dereferencing



Most General Form

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

D: Constant "displacement" 1, 2, or 4 bytes (default 0, if omitted)

Rb: Base register: Any of 16 integer registers (default 0, if omitted)

Ri: Index register: Any, except for %rsp (default 0, if omitted)

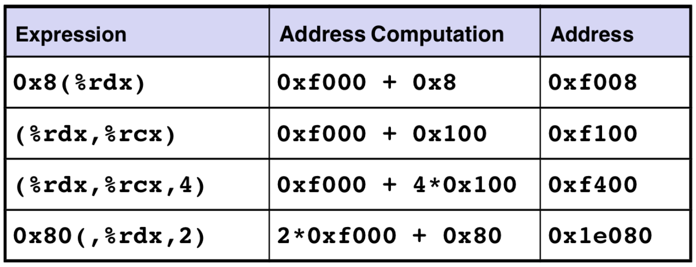
S: Scale: 1, 2, 4, or 8 (*to align with typical element size in an array*) (default 1, if omitted)

Special Cases

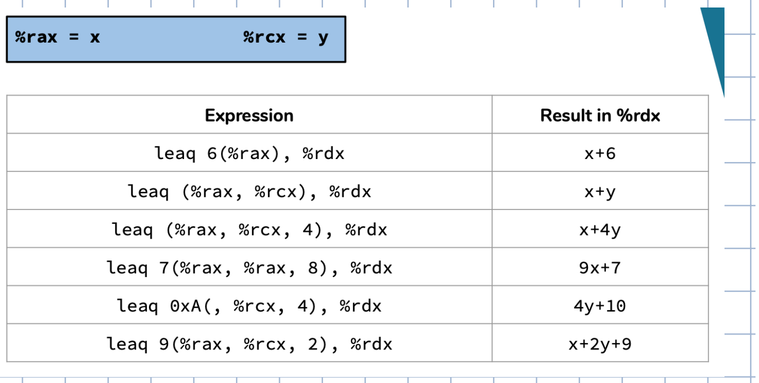
D(Rb)	Mem[Reg[Rb]+D] (Displacement)
(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]
D(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]+D]
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]

C 15400

%rdx = 0xf000
%rcx = 0x1000



Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`



`%rax = x` `%rcx = y`

Expression	Result in %rdx
`leaq 6(%rax), %rdx`	x+6
`leaq (%rax, %rcx), %rdx`	x+y
`leaq (%rax, %rcx, 4), %rdx`	x+4y
`leaq 7(%rax, %rax, 8), %rdx`	9x+7
`leaq 0xA(, %rcx, 4), %rdx`	4y+10
`leaq 9(%rax, %rcx, 2), %rdx`	x+2y+9

leaq   load effective address

leaq - arithmetic, or calculating ...

displacement   scale

leaq 4(%rdx , %rcx, 2) , %rax

base register   index register

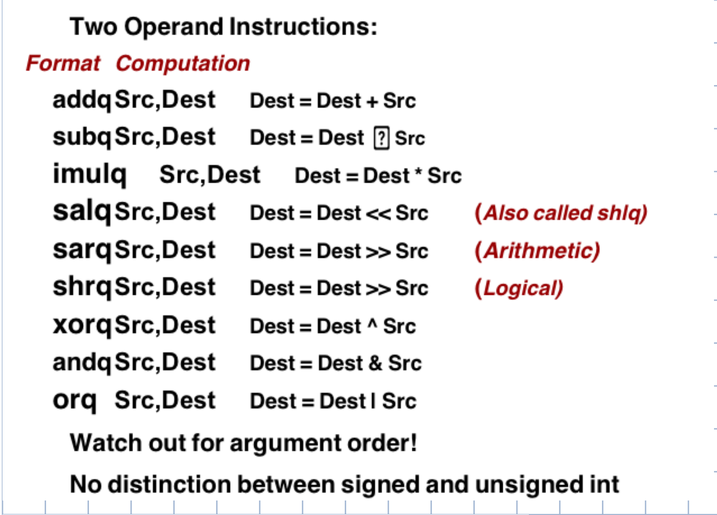
leaq (%rdx, %rdi, 2) , %rax
no memory

registers
%rdi 0x30
%rax 0x90

movq (%rdx, %rdi, 2) , %rax
fetches memory

%rdi 0x30
%rax 0x3F1

Memory
0x3F1   address: 0x90



Two Operand Instructions:

Format Computation

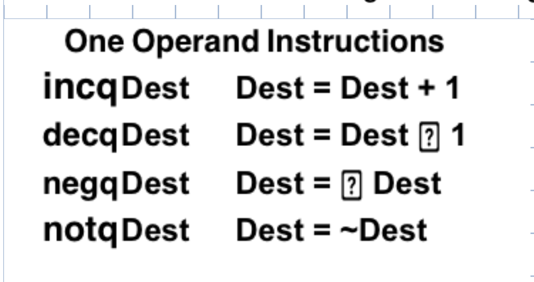
addq Src,Dest	Dest = Dest + Src	
subq Src,Dest	Dest = Dest ⍰ Src	
imulq Src,Dest	Dest = Dest * Src	
salq Src,Dest	Dest = Dest << Src	*(Also called shlq)*
sarq Src,Dest	Dest = Dest >> Src	*(Arithmetic)*
shrq Src,Dest	Dest = Dest >> Src	*(Logical)*
xorq Src,Dest	Dest = Dest ^ Src	
andq Src,Dest	Dest = Dest & Src	
orq Src,Dest	Dest = Dest I Src	

Watch out for argument order!

No distinction between signed and unsigned int

bytes to be shifted : src

left shift : multiplication



One Operand Instructions

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

q → 8 bytes

l → 4 bytes