

Machine execution

Jinyang Li

Lesson Plan: last time

- Basic h/w execution model:
 - CPU fetch next instructions from memory according to %rip
 - Decode and execute instruction (e.g. mov instruction)
 - CPU updates %rip to point to next instruction
- ISA (instruction set architecture): x86, ARM, RISC-V
- X86 ISA
 - %rip, 16 general purpose registers
 - mov instruction

Lesson Plan: today

- mov
 - complete memory addressing
- lea
- arithmetic instructions
- How CPUs realize non-linear control flow

mov: limitation of direct addressing

Direct addressing: the address must be calculated and stored in the register before each memory access.

```
long a[3] = {1, 2, 3};  
for(int i = 0; i < 3; i++)
```

```
    a[i] = 0;
```



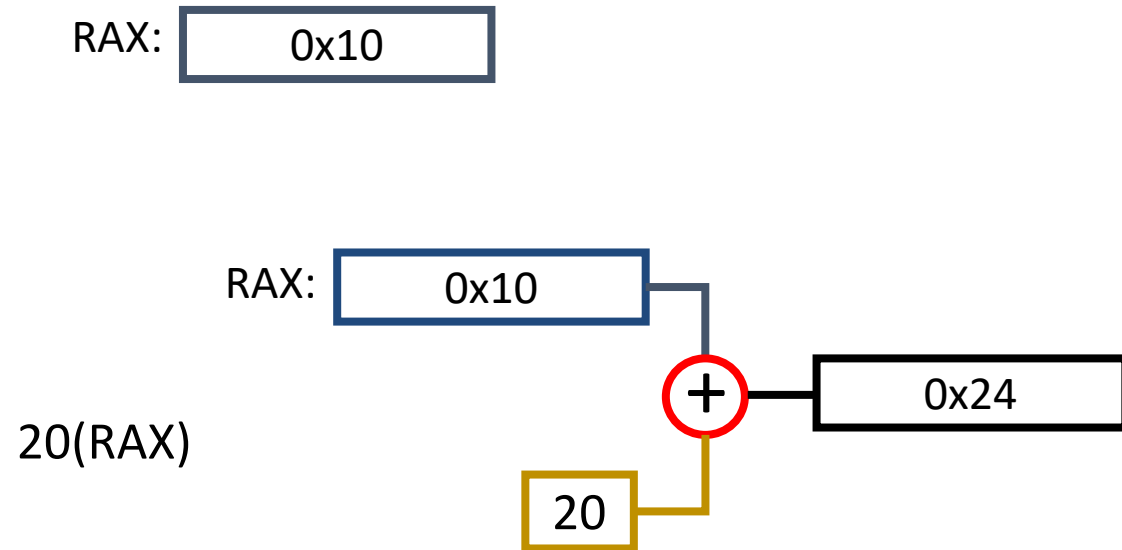
Calculate address of a[i], store it in a register (e.g. %rax)
movq \$0, (%rax)

$\&a[i] = \text{address of } a[0] + \text{stride} * i$

Address mode with displacement

$D(\text{Register}): \text{val}(\text{Register}) + D$

- Register specifies the start of the memory region
- Constant D specifies the offset



Address mode with displacement

$D(\text{Register}): \text{val}(\text{Register}) + D$

- Register specifies the start of the memory region
- Constant D specifies the offset

```
long a[] = {3, 2, 1};  
for(int i = 0; i < 3; i++) {  
    a[i] = i;  
}
```

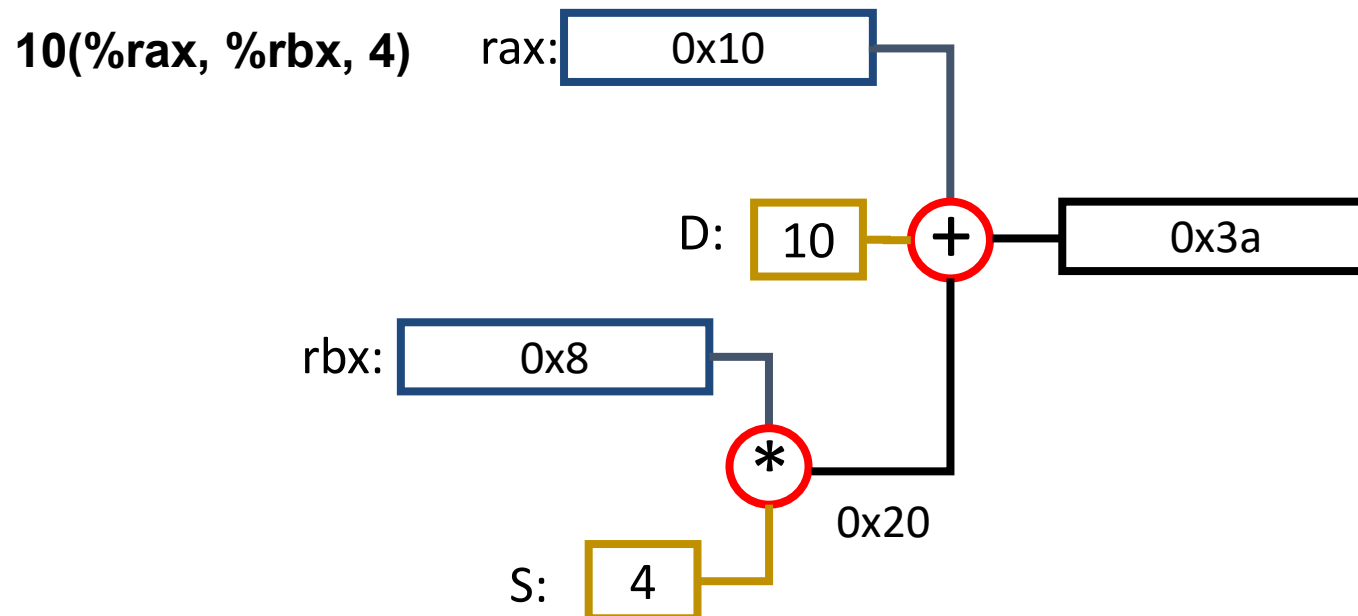


```
long a[] = {1, 2, 3};  
for(int i = 0; i < 3; i++) {  
    mov $i, D(reg); // D = i * 8, reg = &a[0]  
}
```

X86's Complete Memory Addressing Mode

$$D(Rb, Ri, S): \text{val}(Rb) + S * \text{val}(Ri) + D$$

- Rb: Base register
- D: Constant “displacement”
- Ri: Index register (not **%rsp**)
- S: Scale: 1, 2, 4, or 8



Complete Memory Addressing Mode

$D(Rb, Ri, S): val(Rb) + S * val(Ri) + D$

If S is 1 or D is 0, they can be omitted

- $(Rb, Ri): val(Rb) + val(Ri)$
- $D(Rb, Ri): val(Rb) + val(Ri) + D$
- $(Rb, Ri, S): val(Rb) + S * val(Ri)$

Address Computation Examples

%rdx	0xf000
%rcx	0x100

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

Address Computation Examples

%rdx	0xf000
%rcx	0x100

Expression	Address Computation	Address
0x8 (%rdx)	0xf000 + 0x8	0xf008
(%rdx, %rcx)		
(%rdx, %rcx, 4)		
0x80 (, %rdx, 2)		

Address Computation Examples

%rdx	0xf000
%rcx	0x100

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

Address Computation Examples

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

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

Address Computation Examples

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

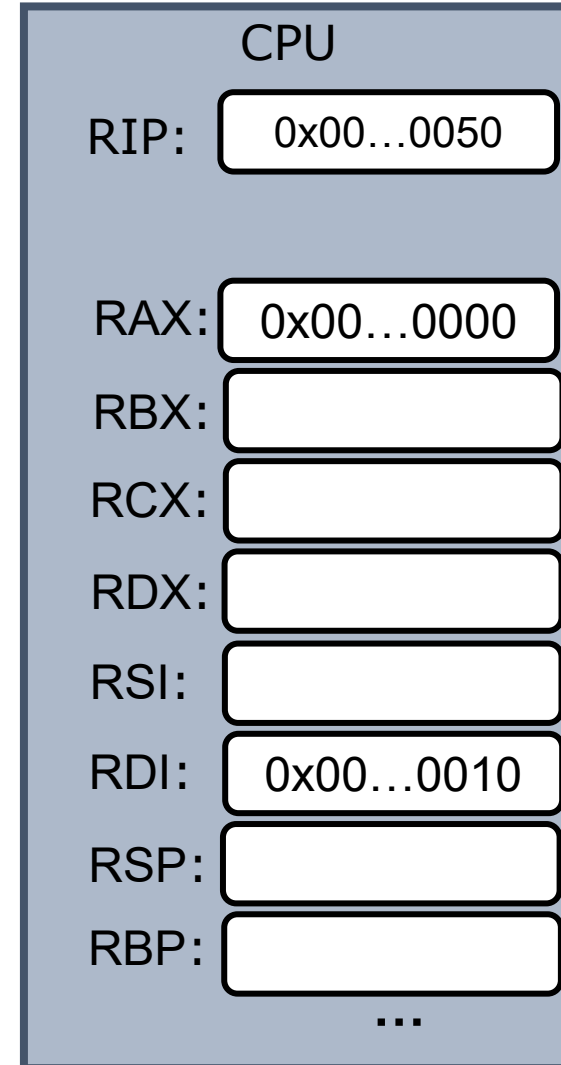
Expression	Address Computation	Address
<code>0x8(%rdx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%rdx,%rcx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%rdx,%rcx,4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80(,%rdx,2)</code>	<code>2*0xf000 +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

Example

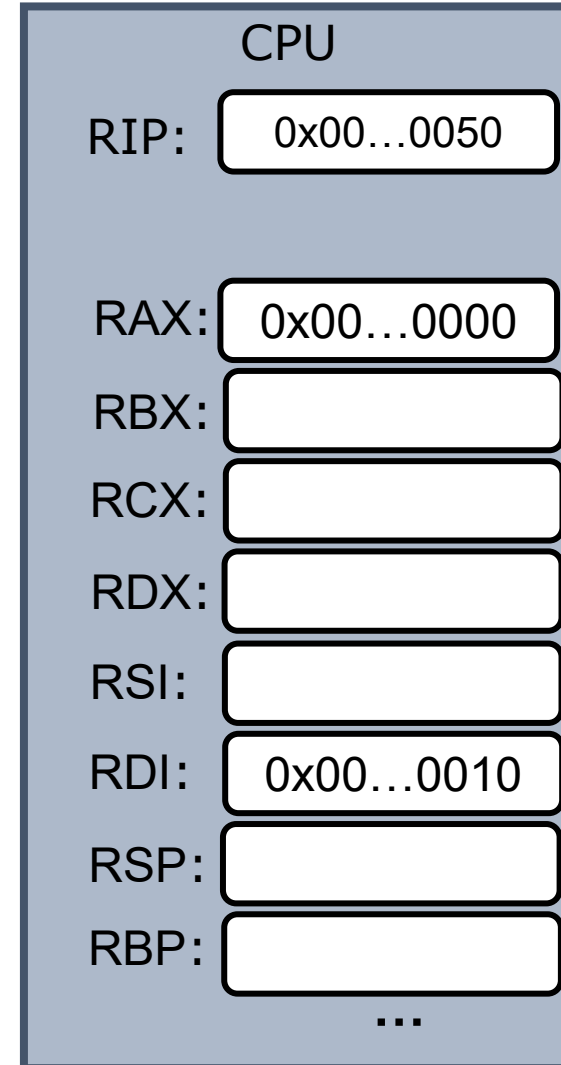
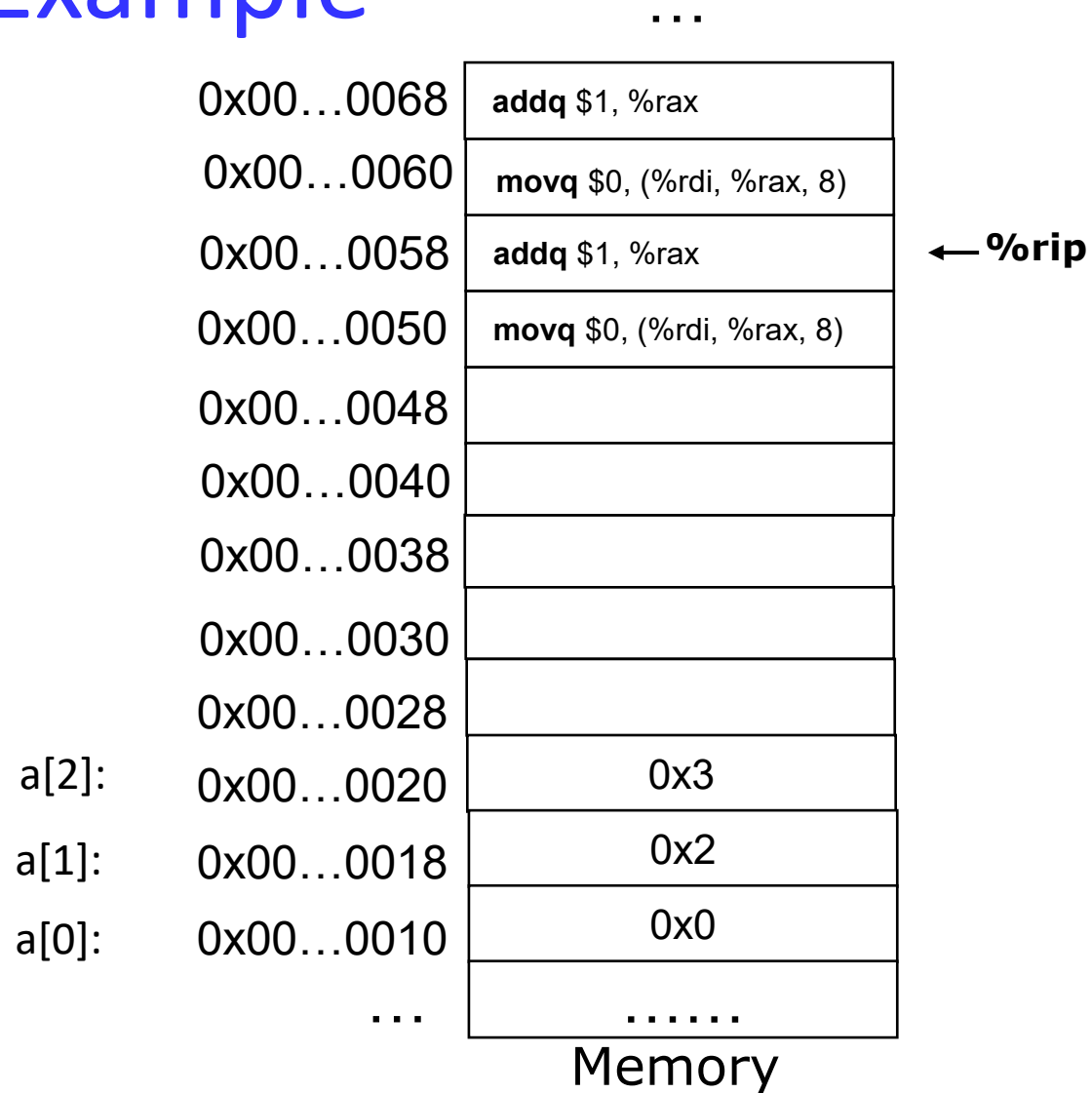
...

	0x00...0068	addq \$1, %rax
	0x00...0060	movq \$0, (%rdi, %rax, 8)
	0x00...0058	addq \$1, %rax
	0x00...0050	movq \$0, (%rdi, %rax, 8) ← %rip
	0x00...0048	
	0x00...0040	
	0x00...0038	
	0x00...0030	
	0x00...0028	
a[2]:	0x00...0020	0x3
a[1]:	0x00...0018	0x2
a[0]:	0x00...0010	0x1

Memory



Example



Example

...

	0x00...0068	addq \$1, %rax	
	0x00...0060	movq \$0, (%rdi, %rax, 8)	← %rip
	0x00...0058	addq \$1, %rax	
	0x00...0050	movq \$0, (%rdi, %rax, 8)	
	0x00...0048		
	0x00...0040		
	0x00...0038		
	0x00...0030		
	0x00...0028		
a[2]:	0x00...0020	0x3	
a[1]:	0x00...0018	0x2	
a[0]:	0x00...0010	0x0	
	

Memory

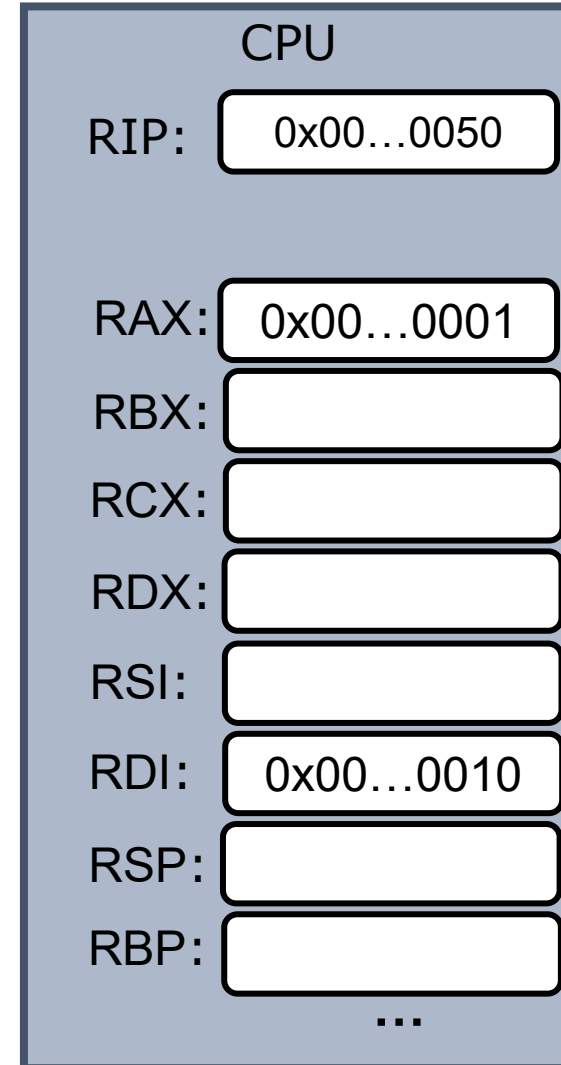
CPU	
RIP:	0x00...0050
RAX:	0x00...0001
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	0x00...0010
RSP:	
RBP:	
...	

Example

...

	0x00...0068	addq \$1, %rax	← %rip
	0x00...0060	movq \$0, (%rdi, %rax, 8)	
	0x00...0058	addq \$1, %rax	
	0x00...0050	movq \$0, (%rdi, %rax, 8)	
	0x00...0048		
	0x00...0040		
	0x00...0038		
	0x00...0030		
	0x00...0028		
a[2]:	0x00...0020	0x3	
a[1]:	0x00...0018	0x0	
a[0]:	0x00...0010	0x0	
	

Memory

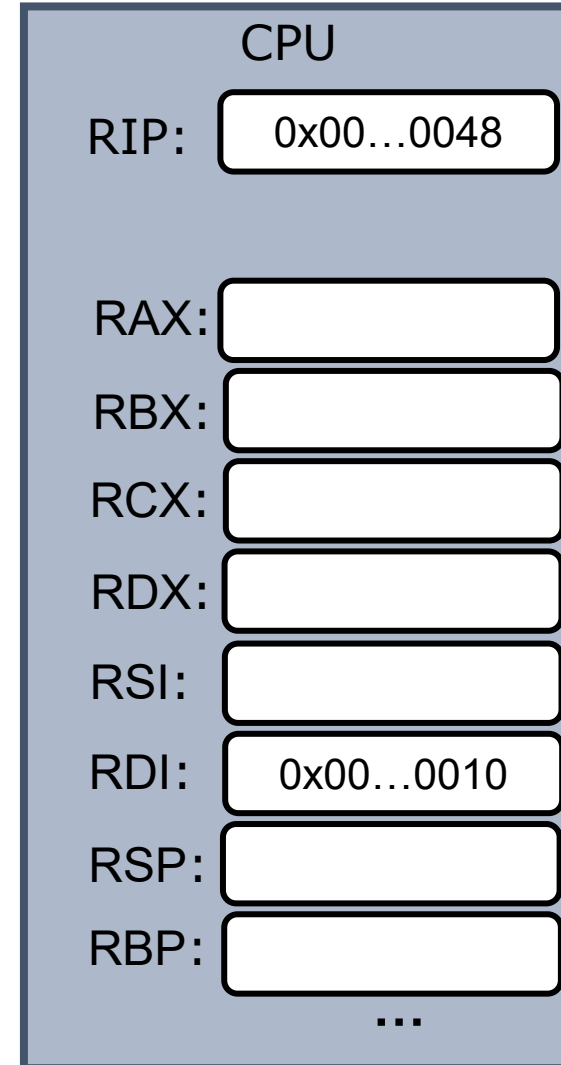
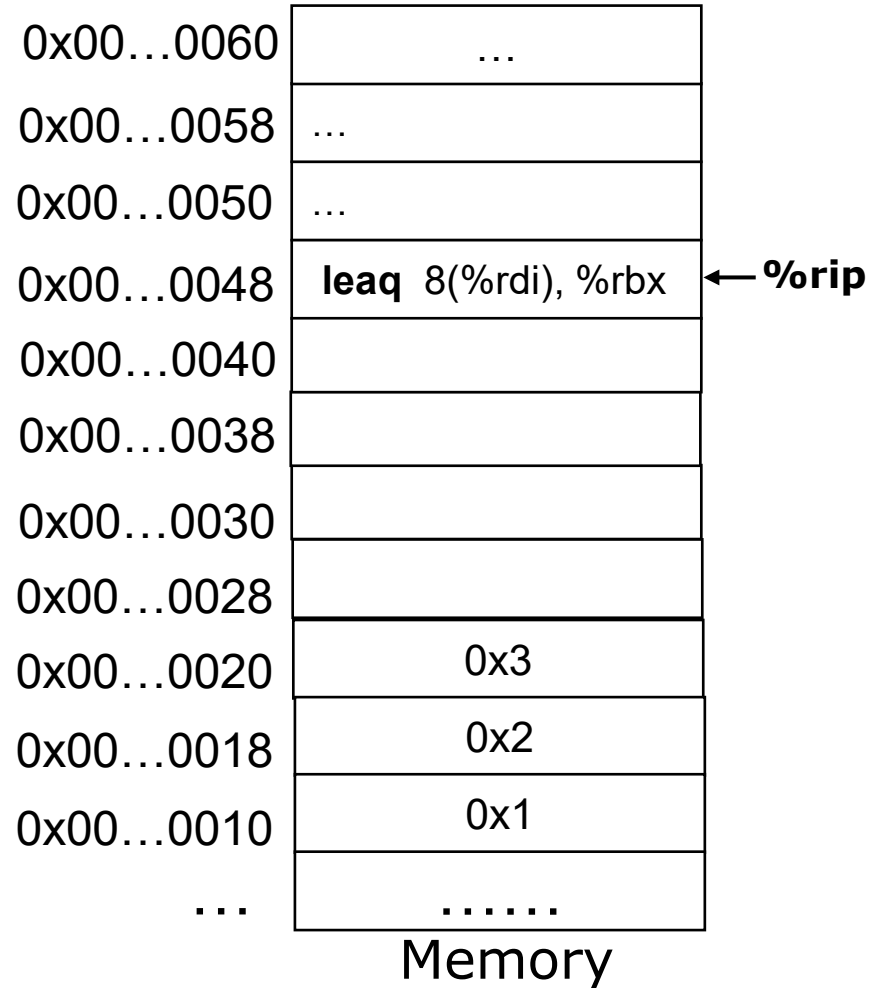


lea instruction

leaq *Source, Dest*

- Short for **L**oad **E**ffective **A**ddress
- Set *Dest* to the address denoted by *Source* address mode expression
- Performs address calculation only; no memory access!

Example



Example

0x00...0060	...	
0x00...0058	...	
0x00...0050	...	
0x00...0048	leaq 8(%rdi), %rbx	← %rip
0x00...0040		
0x00...0038		
0x00...0030		
0x00...0028		
0x00...0020	0x3	
0x00...0018	0x2	
0x00...0010	0x1	
...	

Memory

CPU	
RIP:	0x00...0048
RAX:	
RBX:	0x00...0018
RCX:	
RDX:	
RSI:	
RDI:	0x00...0010
RSP:	
RBP:	
...	

A common use case for leaq

Lea is used to compute certain simple arithmetic expression

```
long m3(long x)
{
    return x*3;
}
```



$\%rdi + \%rdi \times 2$

`leaq` `(%rdi, %rdi, 2)`, `%rax`

mov *or* *> seg fault*

i),

Assume `%rdi` has the value of `x`

Arithmetic Expression Puzzle

Suppose %rdi, %rsi, %rax contains variable x, y, s respectively

```
leaq    (%rdi,%rsi,2), %rax  
leaq    (%rax,%rax,4), %rax
```



```
long f(long x, long y)  
{  
    long s = ??;  
    return s;  
}
```

Arithmetic Expression Puzzle

Suppose %rdi, %rsi, %rax contains variable x, y, s respectively

```
leaq    (%rdi,%rsi,2), %rax  
leaq    (%rax,%rax,4), %rax
```



```
long f(long x, long y)  
{  
    long s = 5(x + 2y);  
    return s;  
}
```

Basic Arithmetic Operations

addq Src, Dest Dest = Dest + Src

subq Src, Dest Dest = Dest – Src

imulq Src, Dest Dest = Dest * Src

incq Dest Dest = Dest + 1


decq Dest Dest = Dest – 1

negq Dest Dest = – Dest

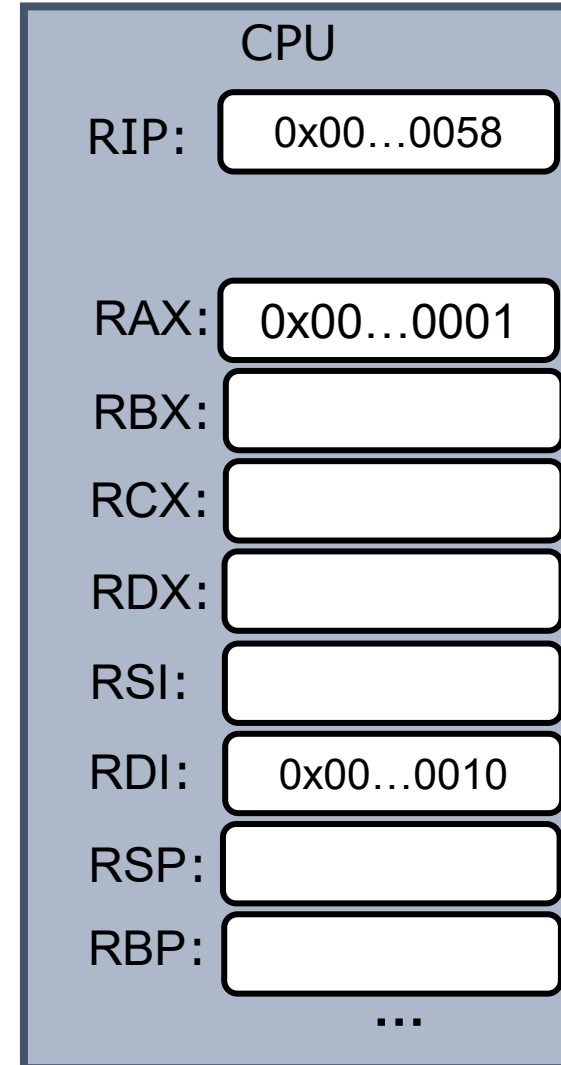
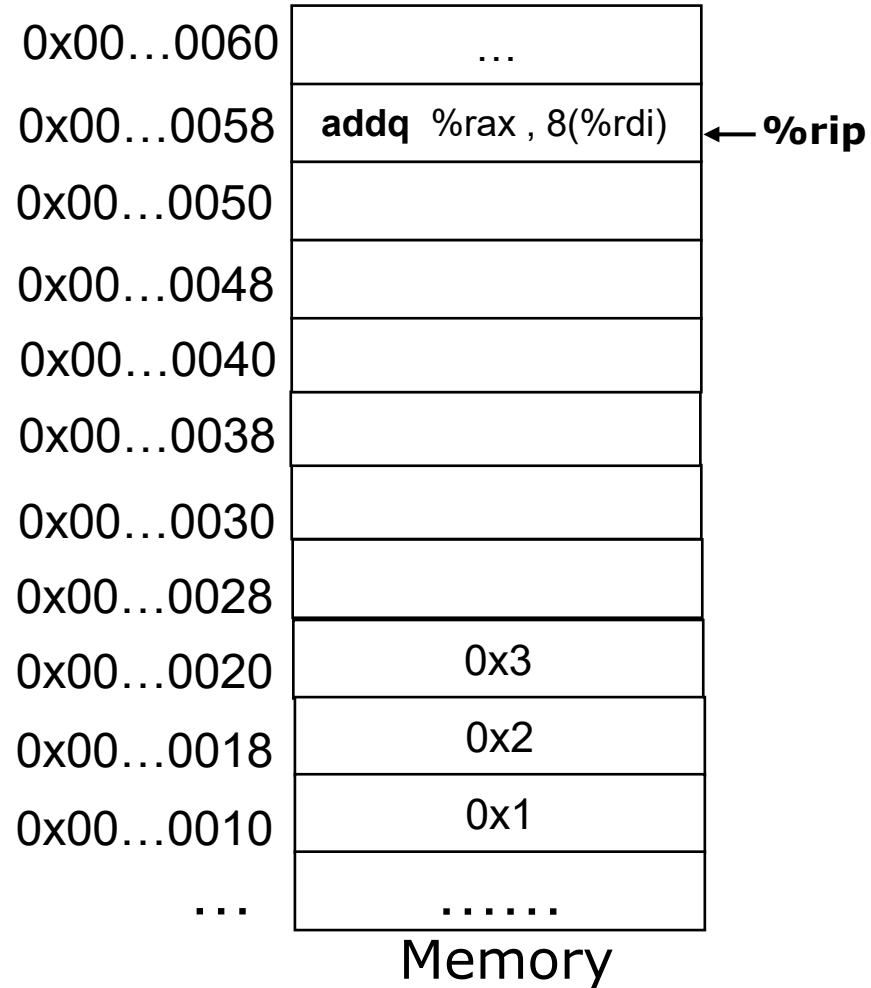
Bitwise Operations

salq	Src, Dest	Dest = Dest << Src
sarq	Src, Dest	Dest = Dest >> Src
shlq	Src, Dest	Dest = Dest << Src
shrq	Src, Dest	Dest = Dest >> Src
xorq	Src, Dest	Dest = Dest ^ Src
andq	Src, Dest	Dest = Dest & Src
orq	Src, Dest	Dest = Dest Src
notq	Dest	Dest = ~Dest

Arithmetic left shift
Arithmetic right shift
Logical left shift
Logical right shift



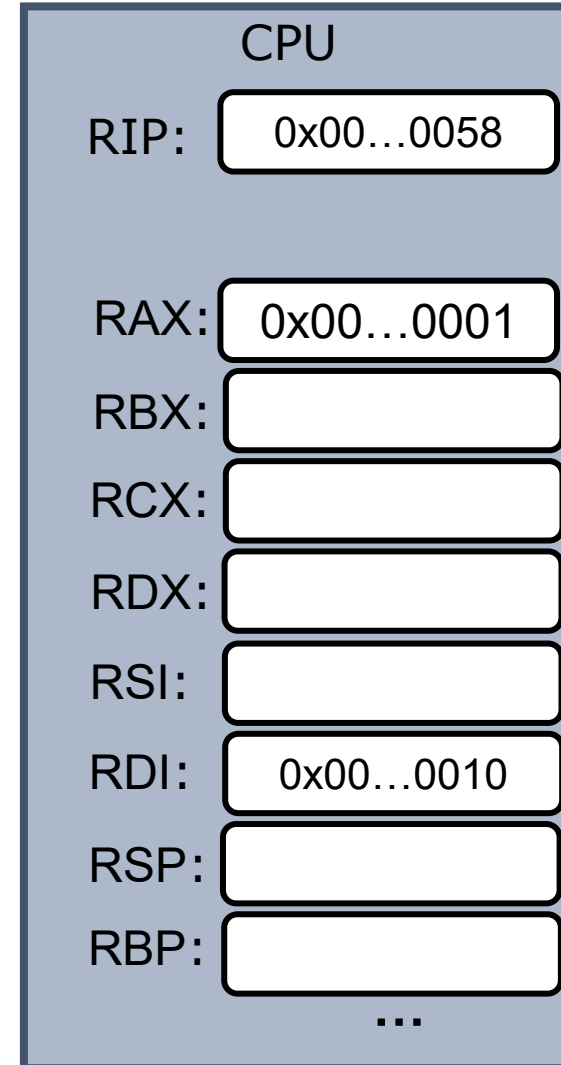
Example



Example

0x00...0060	...	← %rip
0x00...0058	addq %rax , 8(%rdi)	
0x00...0050		
0x00...0048		
0x00...0040		
0x00...0038		
0x00...0030		
0x00...0028		
0x00...0020	0x3	
0x00...0018	0x3	
0x00...0010	0x1	
...	

Memory



Lesson Plan: today

- mov
 - complete memory addressing
- lea
- arithmetic instructions
- How CPUs realize non-linear control flow

How is control flow realized?

if ... else



???

for loop

while loop

...

Control flow uses RFLAGS register

PC: Program counter

- Store memory address of next instruction
- Also called “RIP” in x86_64

IR: instruction register

- Store the fetched instruction

General purpose registers:

- Store operands and pointers used by program

Program status and control register:

- Contain status of the instruction executed
- All called “**RFLAGS**” in x86_64

How control flow uses RFLAGS register

- RFLAGS is a special purpose register
- Different bits represent different status flags
- Certain instructions set status flags
 - Regular arithmetic instructions
 - Special flag-setting instructions: **cmp**, **test**, **set**
- **jmp** instructions use flags to determine value of %rip

EFLAGS register: ZF

- ZF (Zero Flag):
 - Set if the result of the instruction is zero; cleared otherwise.

```
movq $2, %rax  
subq $2, %rax
```


EFLAGS register: SF

- SF (Sign Flag):
 - Set to be the most-significant bit of the result.

```
movq $2, %rax  
subq $10, %rax
```

EFLAGS register: CF

- CF (Carry Flag):
 - Set if adding/subtracting two numbers carries out of MSB
 - ➡ i.e. Set if overflow for unsigned integer arithmetic

```
movq $0xffffffffffffffff, %rax  
addq $2, %rax
```

```
movq $0, %rax  
subq $1, %rax
```

EFLAGS register: OF

- OF (Overflow Flag):
 - Set if there is carry-in but no carry-out of MSB
 - or, there is no carry-in but there's carry-out of MSB



Set if overflow for signed integer (2's complement) arithmetic.

```
movq $0x7fffffffffffffffff, %rax  
addq $1, %rax
```

```
movq $0x8000000000000000, %rax  
addq $0xffffffffffffffff, %rax
```

CF and OF are different flags

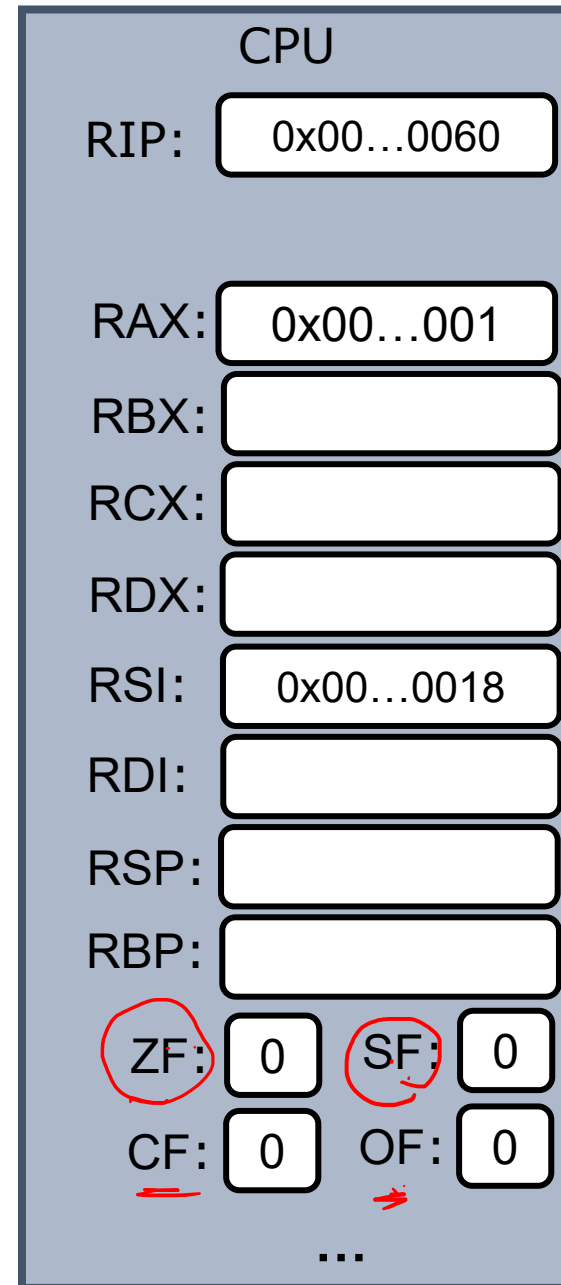
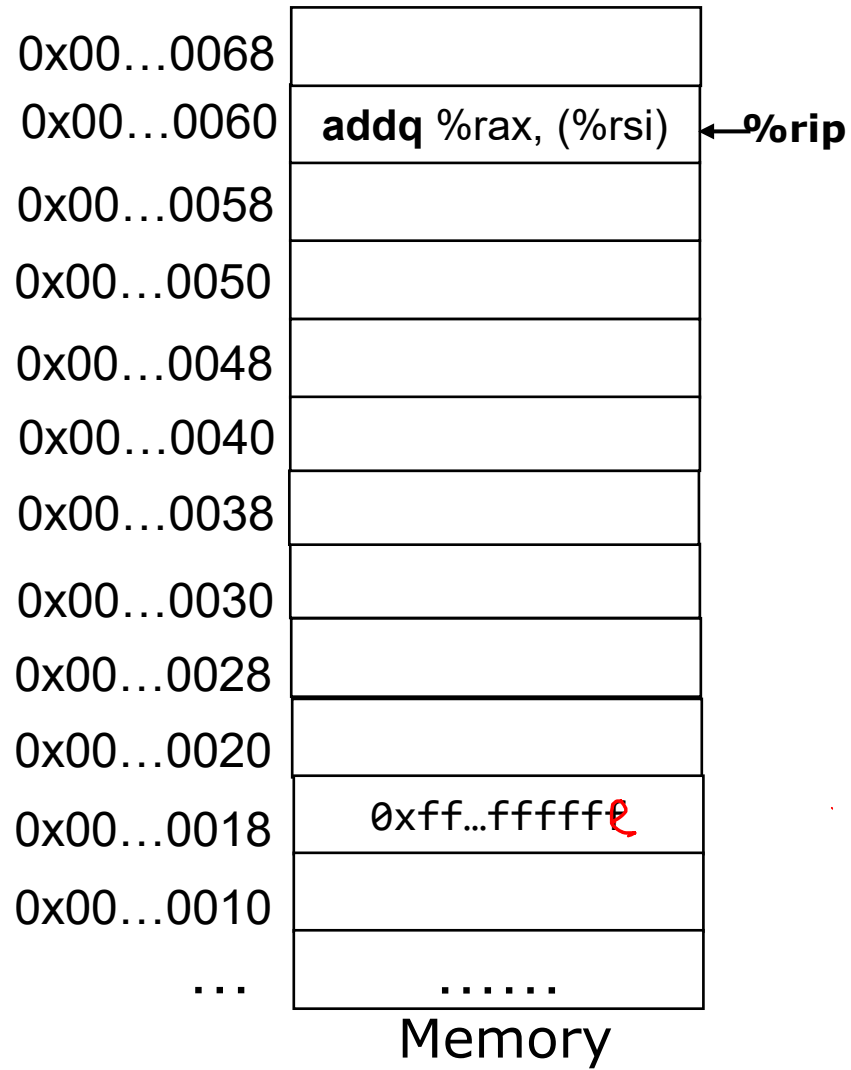
- CPU does care if data represents signed or unsigned integer:
 - Same underlying arithmetic hardware circuitry.
 - OF and CF flags are set by examining various carry bits
- Up to programmer/compiler to use either CF or OF flag

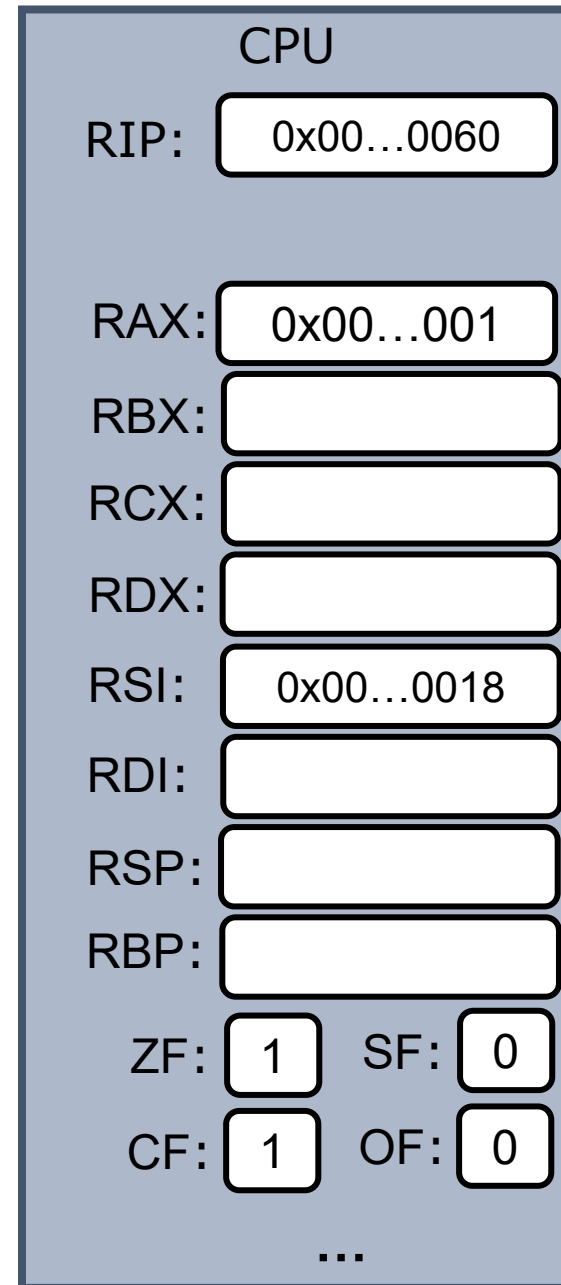
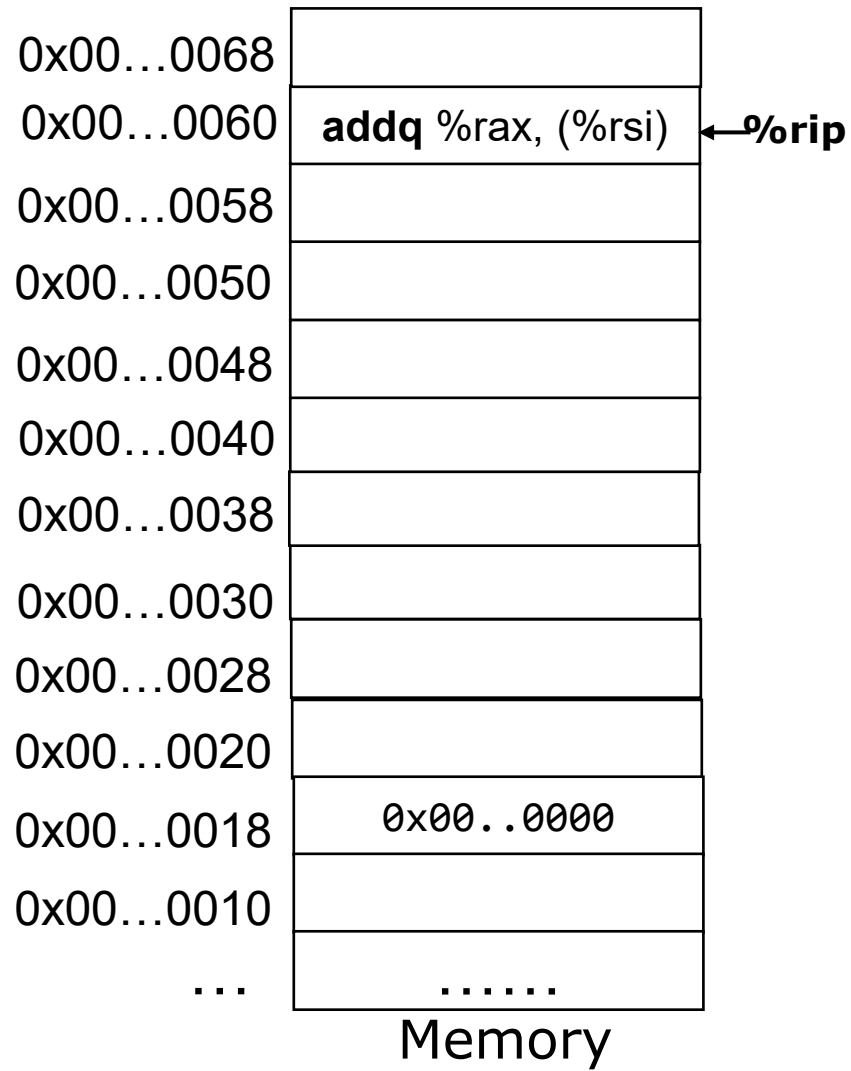
Status flags summary

flag	status
ZF (Zero Flag)	set if the result is zero.
SF (Sign Flag)	set if the result is negative.
CF (Carry Flag)	Overflow for unsigned-integer arithmetic
OF (Overflow Flag)	Overflow for signed-integer arithmetic

Set by arithmetic instructions, e.g. add, inc, and, sal

Not set by **lea**, **mov**





Summary

- X86 ISA
 - %rip, 16 general purpose registers
 - mov
 - Lea
 - Various arithmetic instructions
 - RFLAGS: ZF, SF, CF, OF