

# 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;
```



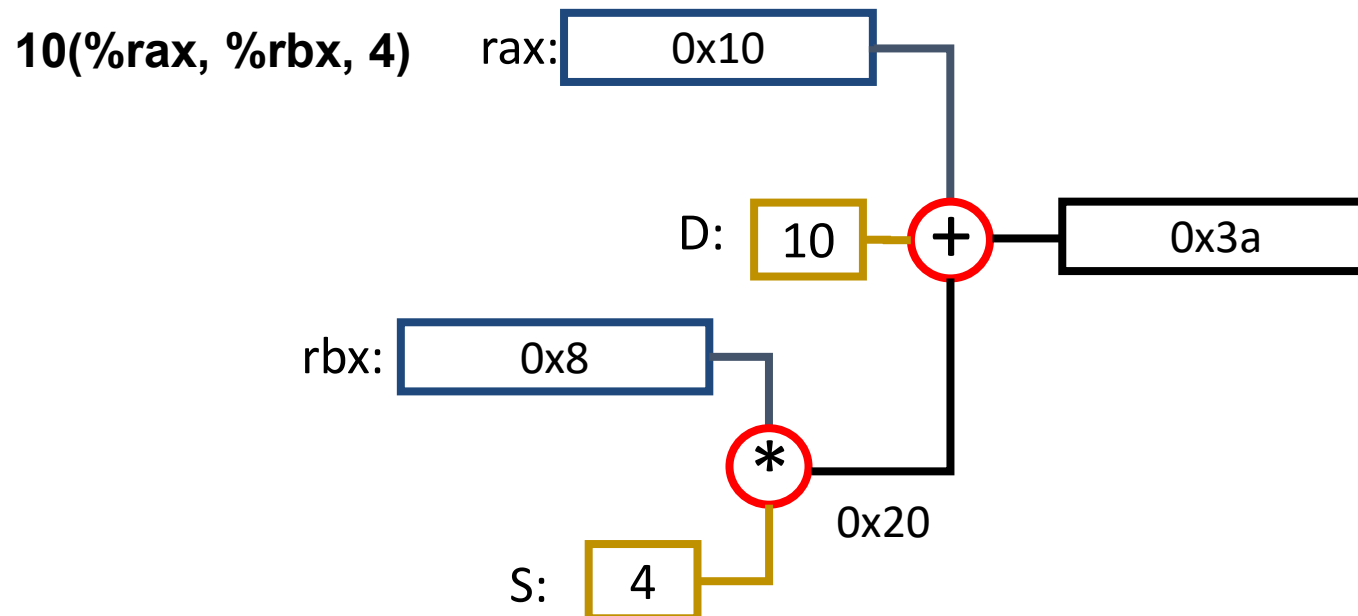
Calculate address of a[i], store it in a register (e.g. %rax)  
movq \$0, (%rax)

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

# 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

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

Expression	Address Computation	Address
<b>0x8 (%rdx)</b>		
<b>(%rdx, %rcx)</b>		
<b>(%rdx, %rcx, 4)</b>		
<b>0x80 (, %rdx, 2)</b>		

# Example

...

	0x00...0068	<b>addq</b> \$1, %rax
	0x00...0060	<b>movq</b> \$0, (%rdi, %rax, 8)
	0x00...0058	<b>addq</b> \$1, %rax
	0x00...0050	<b>movq</b> \$0, (%rdi, %rax, 8) ← <b>%rip</b>
	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

CPU

RIP: 0x00...0050

RAX: 0x00...0000

RBX:

RCX:

RDX:

RSI:

RDI: 0x00...0010

RSP:

RBP:

...

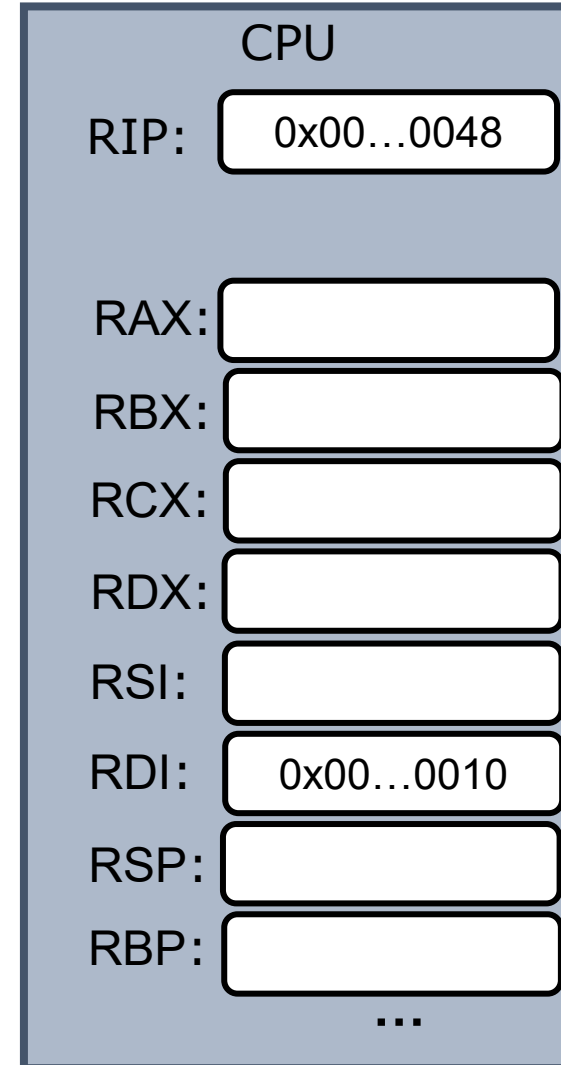
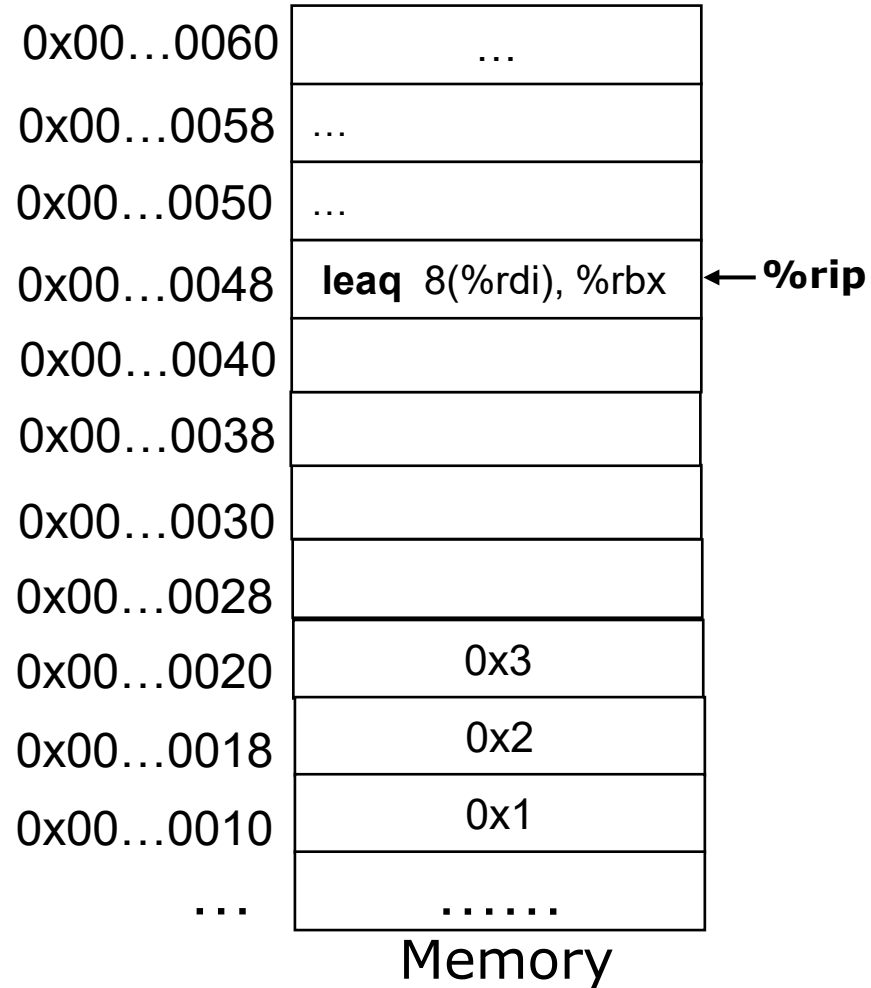


# lea instruction

## **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



# A common use case for leaq

Lea is used to compute certain simple arithmetic expression

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



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

**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       $\text{Dest} = \text{Dest} + \text{Src}$

**subq**      Src, Dest       $\text{Dest} = \text{Dest} - \text{Src}$

**imulq**     Src, Dest       $\text{Dest} = \text{Dest} * \text{Src}$

**incq**      Dest           $\text{Dest} = \text{Dest} + 1$


**decq**      Dest           $\text{Dest} = \text{Dest} - 1$

**negq**      Dest           $\text{Dest} = -\text{Dest}$

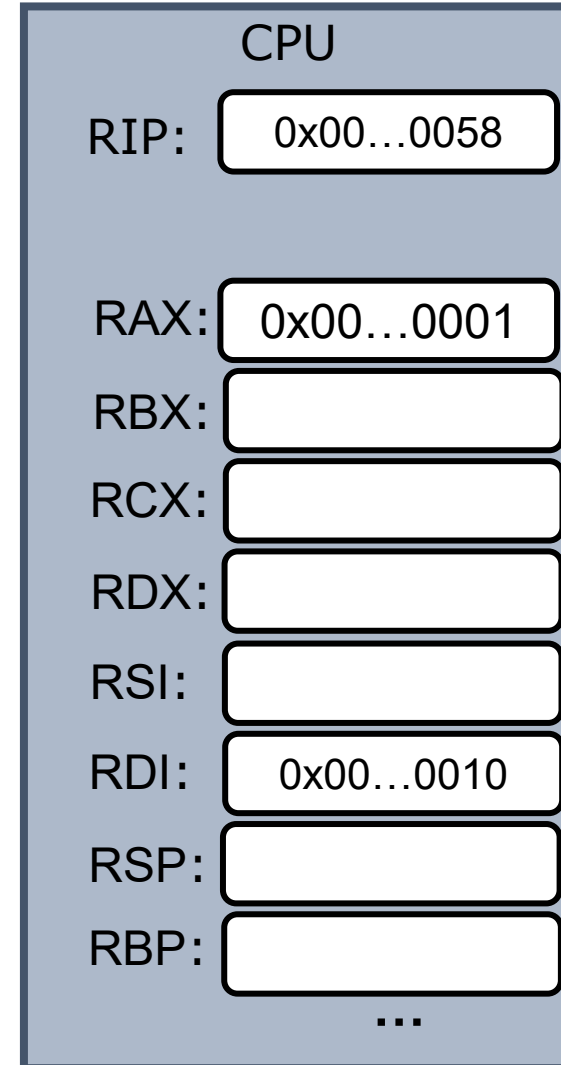
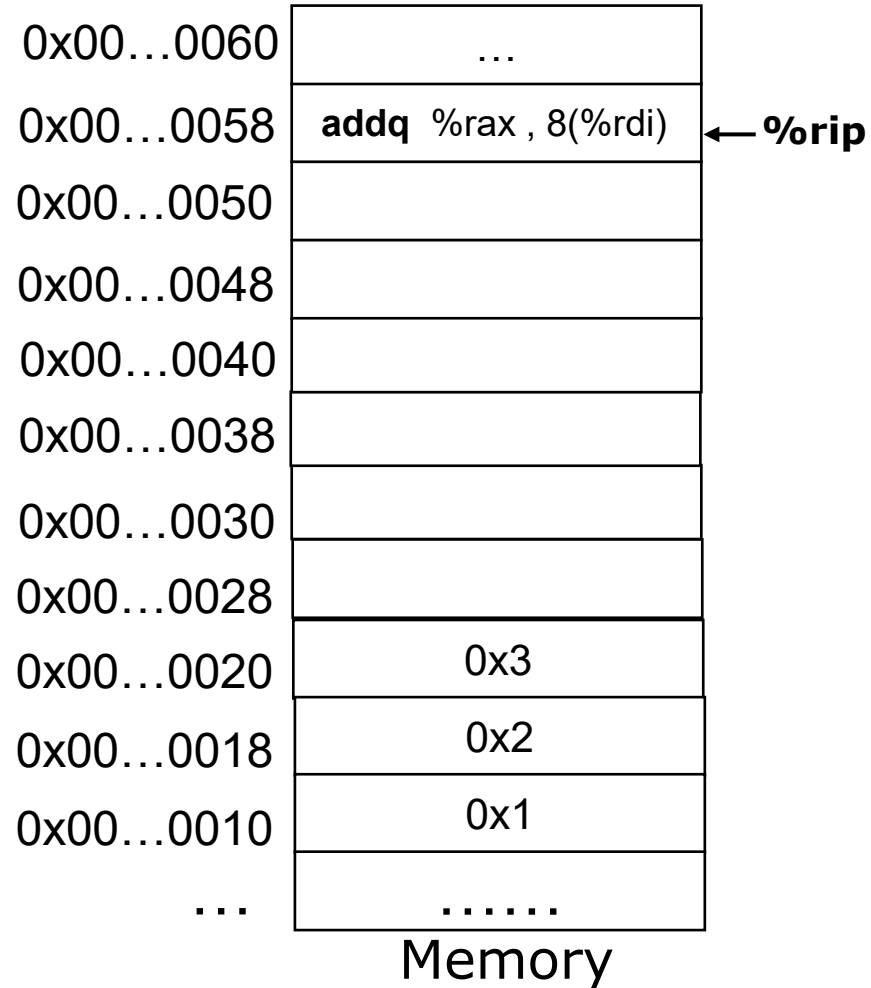
# Bitwise Operations

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

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



# Example





# 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**

