

# Machine Program: Basics

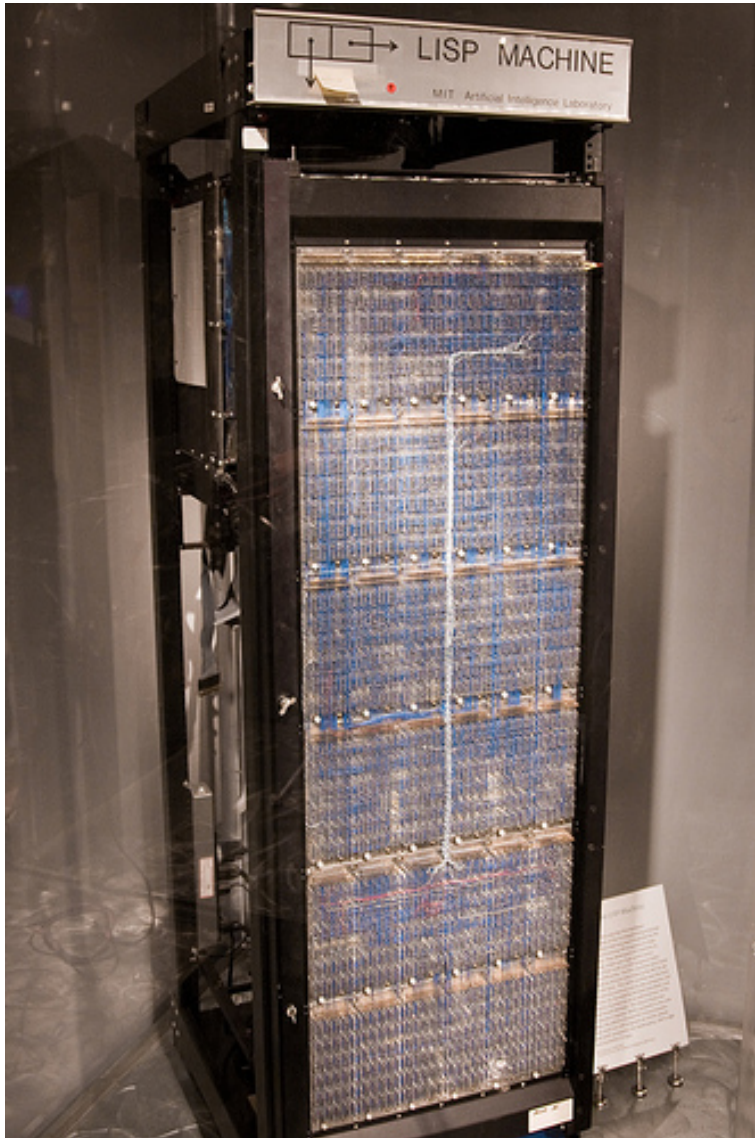
Jinyang Li

based on Tiger Wang's slides

# What we've learnt so far

- Programming in C
- Question: can we build a CPU to execute C program directly?

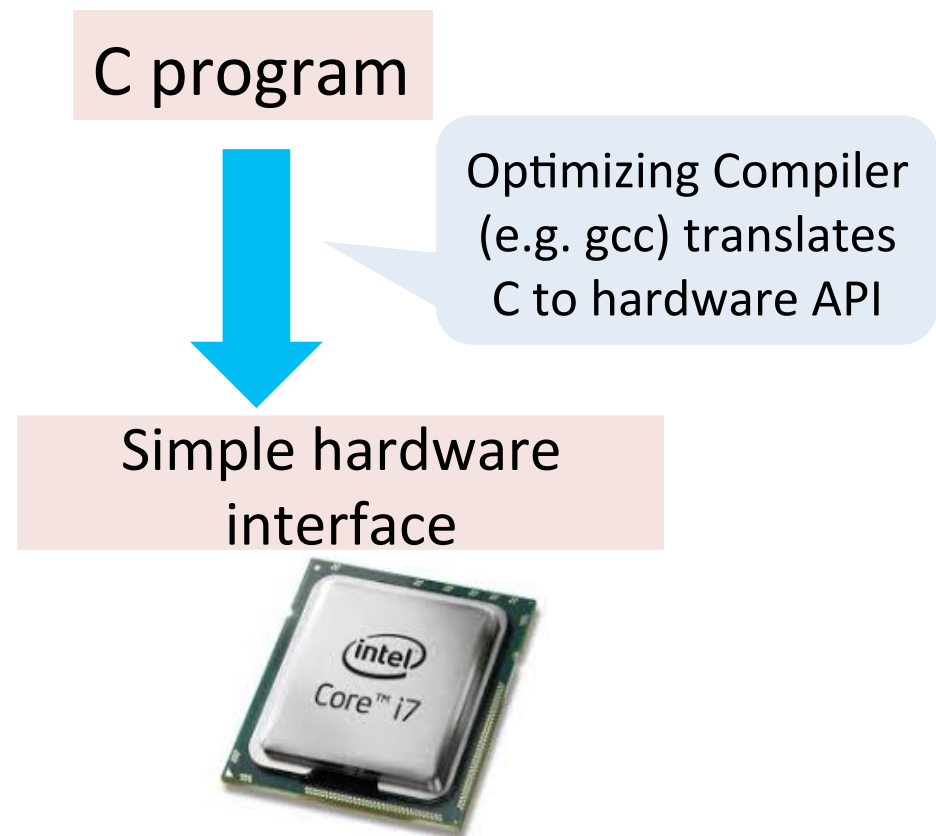
# a CPU to execute C directly?



- Historical precedents:
  - LISP machine (80s)
  - Intel iAPX 432 (Ada)

# Why not build a CPU that directly executes C?


- Leads to very complex hardware design
  - Complex → Hard to implement w/ high performance
- A better approach:

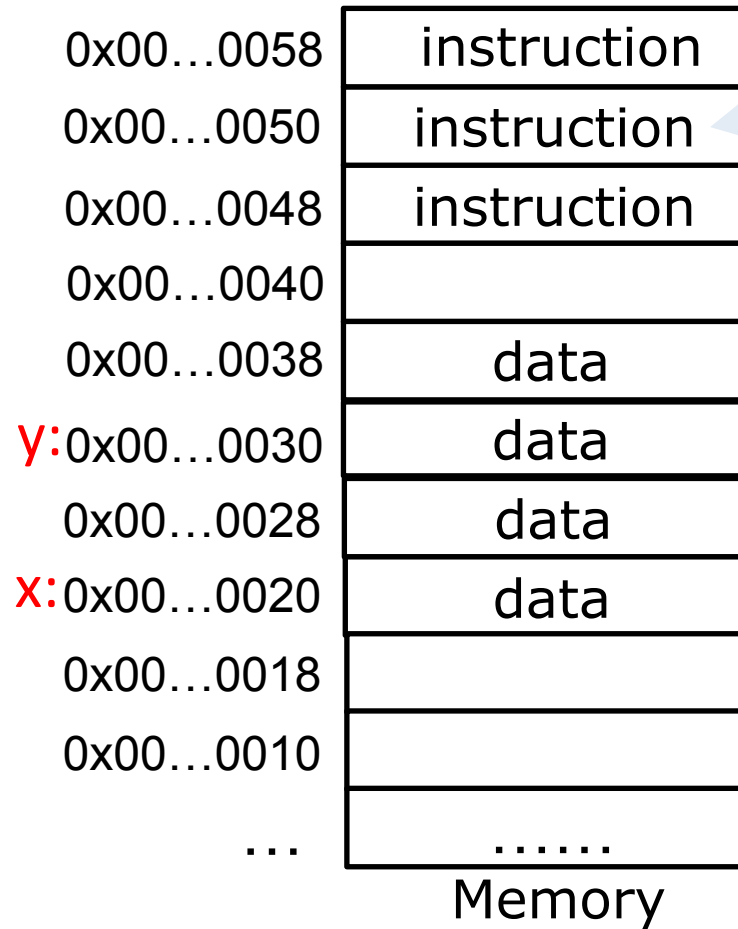


# C vs. machine code

```
long x;  
long y;
```

```
y = x;  
y = 2*y;
```

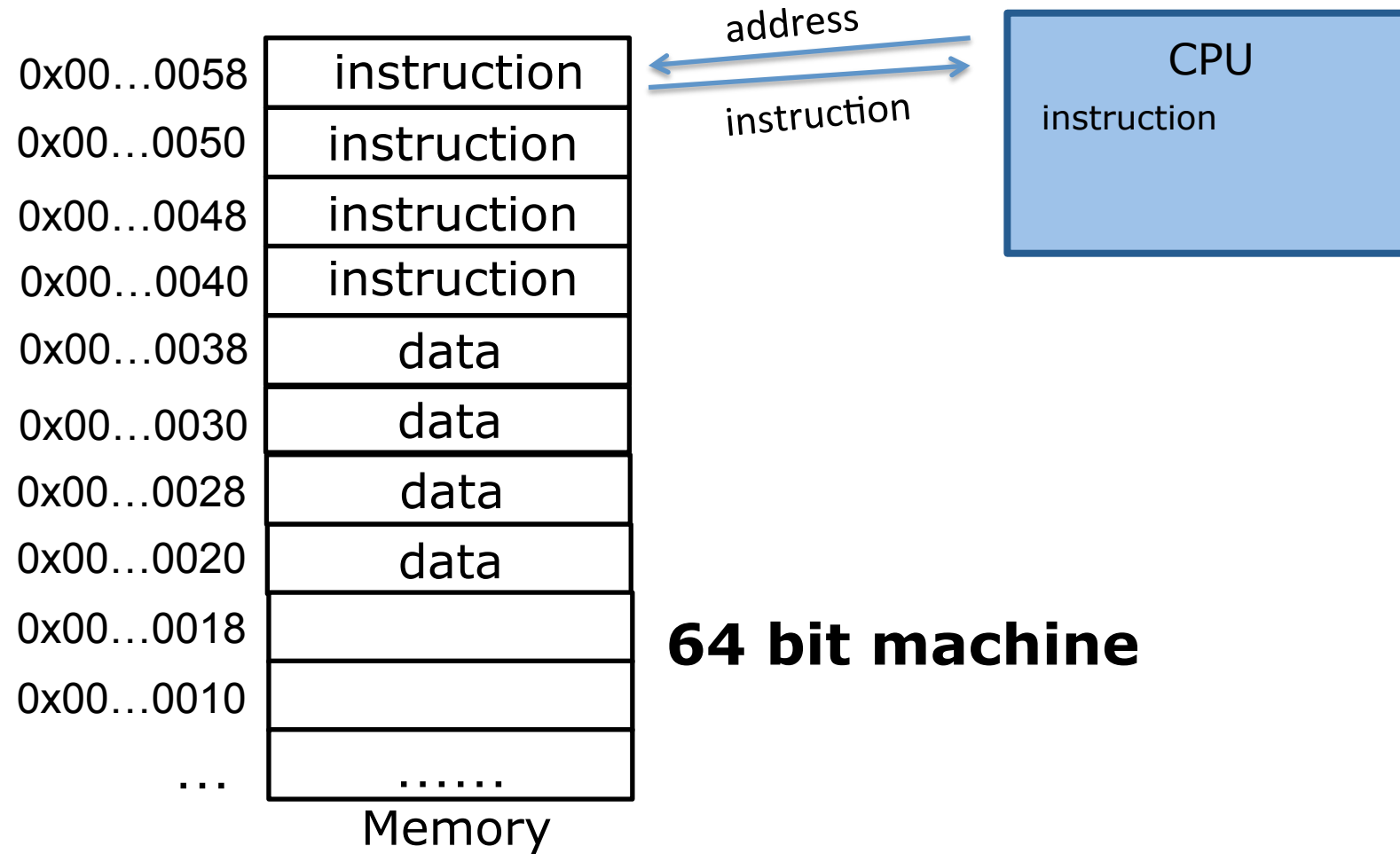
  
compile to  
x86 executable



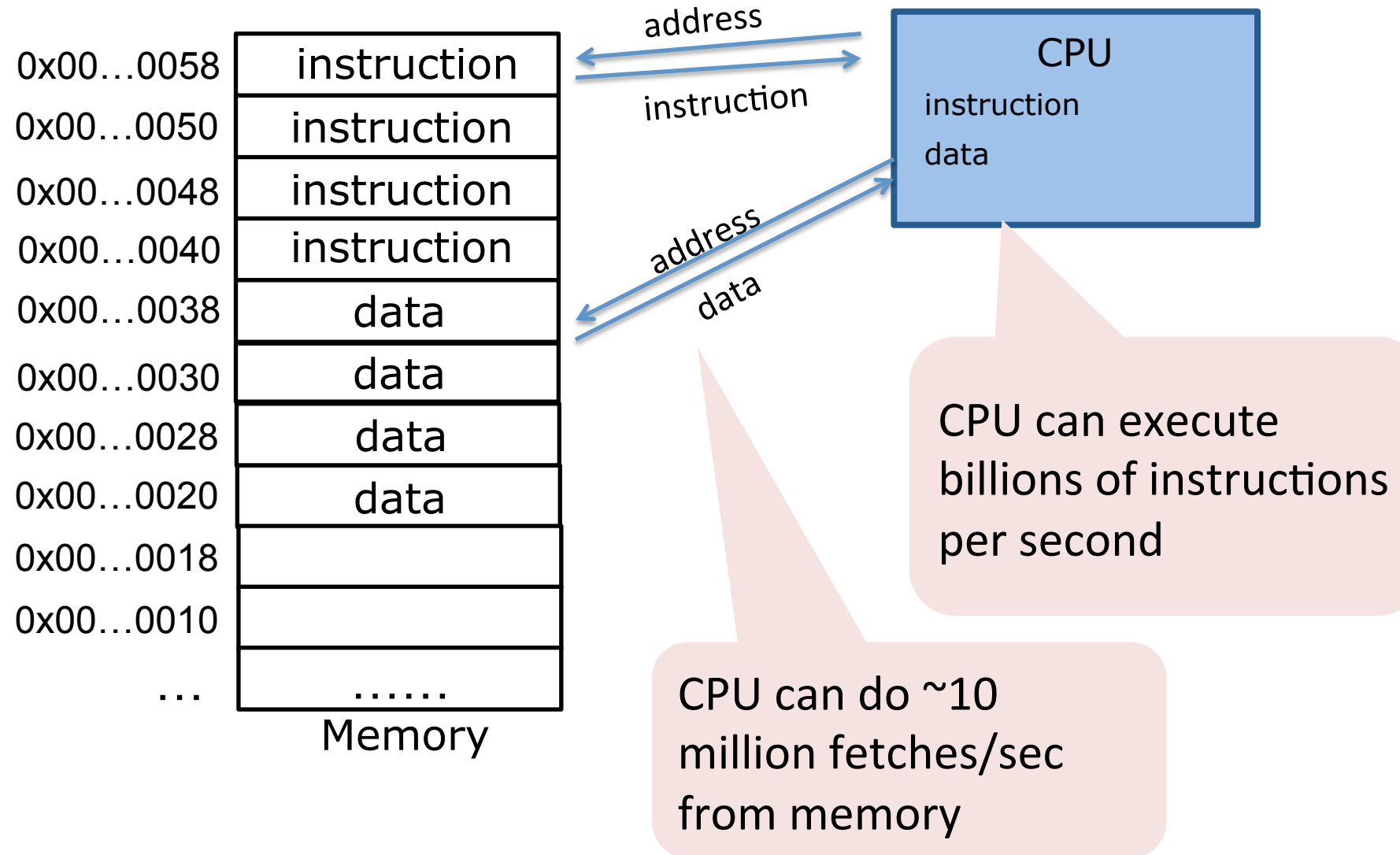
E.g. move data  
from one  
memory location  
to another

E.g. multiply the  
number at some  
memory location  
by a constant

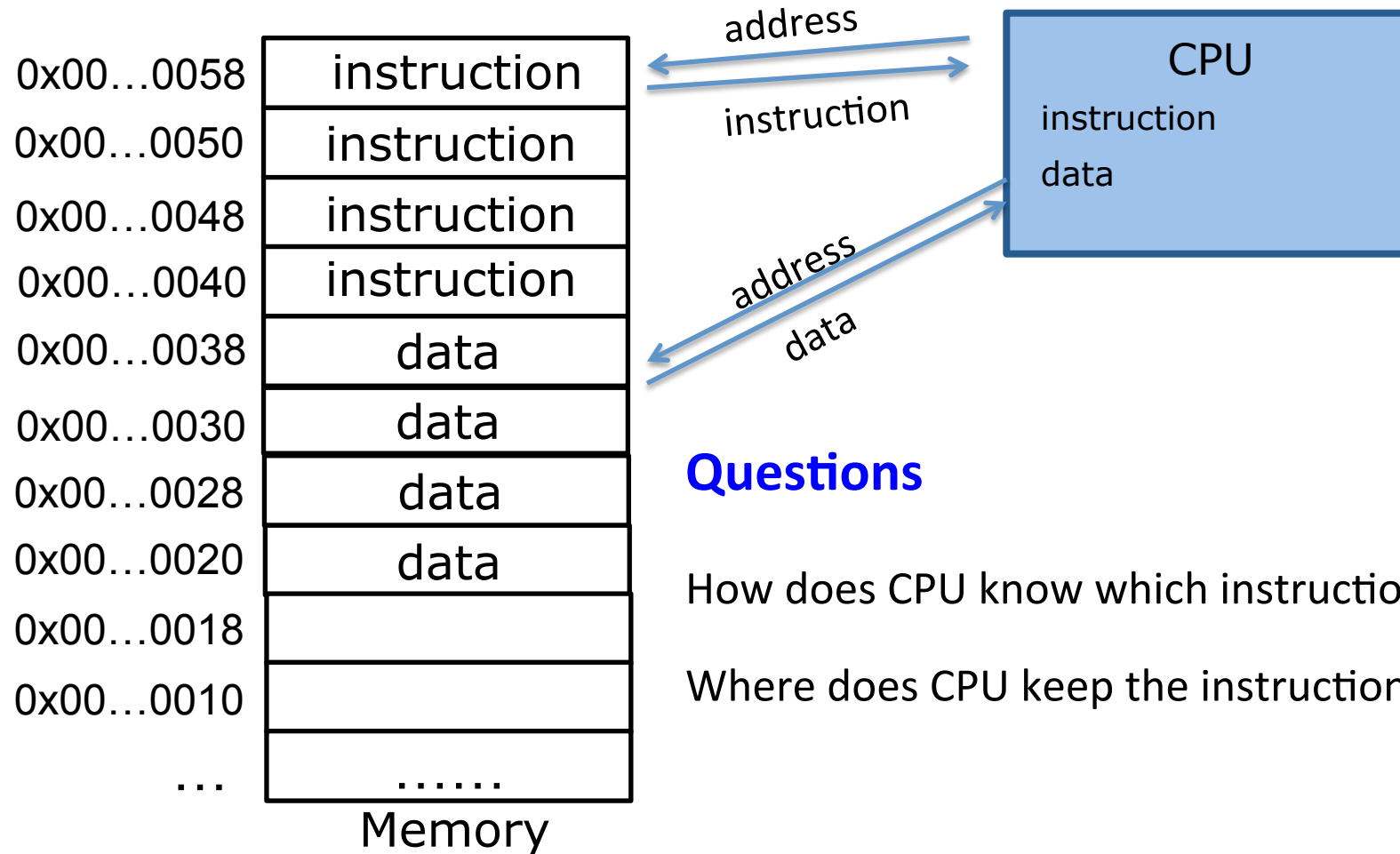
# How CPU executes a program



# How CPU executes a program



# How CPU executes a program



## Questions

How does CPU know which instruction to fetch?

Where does CPU keep the instruction and data?



# Register – temporary storage area built into a CPU

PC: Program counter, also called instruction pointer (IP).

- Store memory address of next instruction
- Called “RIP” in x86\_64

IR: instruction register

- Store the fetched instruction

General purpose registers:

- Store data and address used by program

Program status and control register:

- Status of the program being executed
- Called “EFLAGS” in x86\_64

# Register – temporary storage area built into a CPU

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:

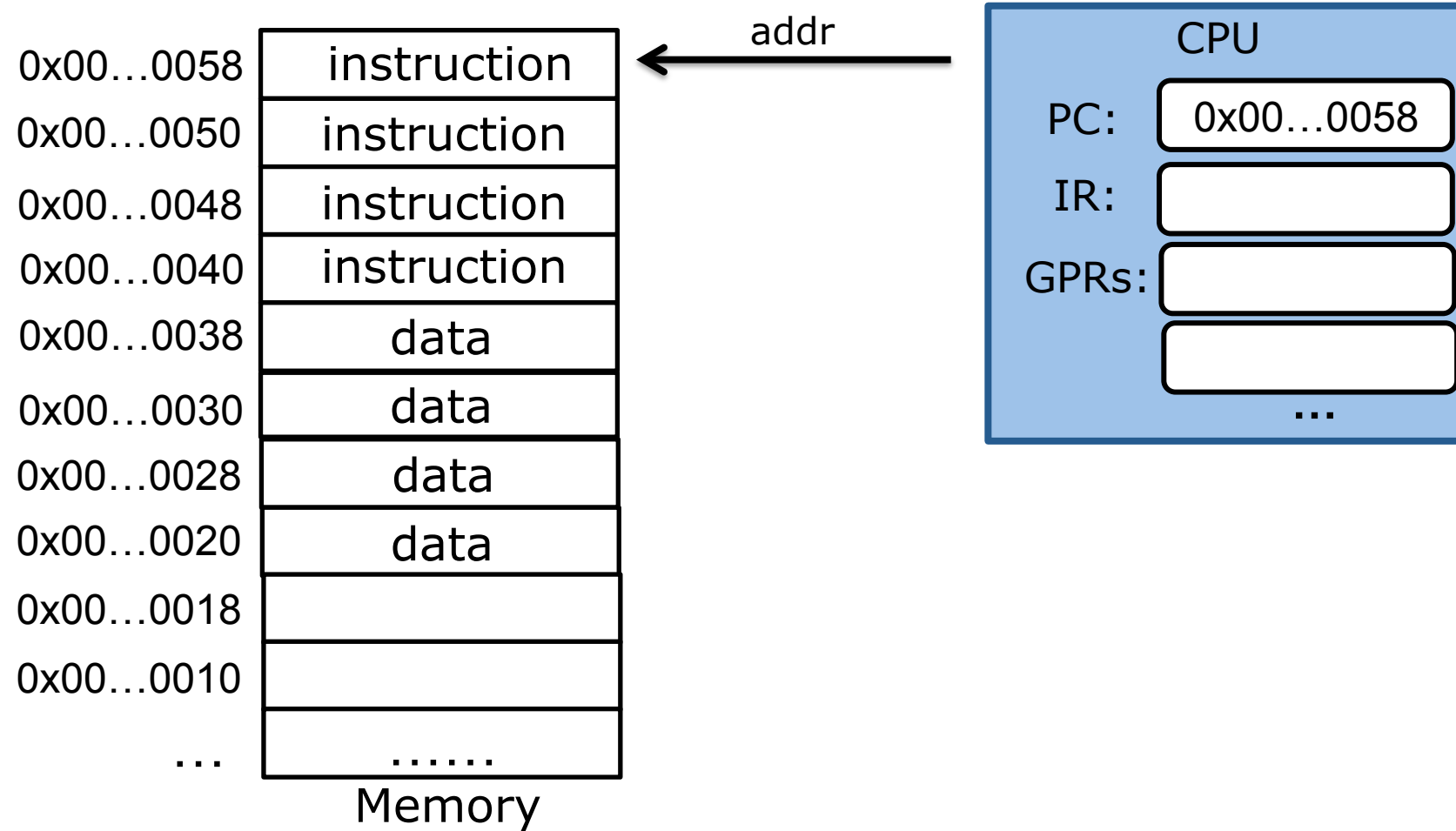
- Store operands and pointers used by program

Program status and control register:

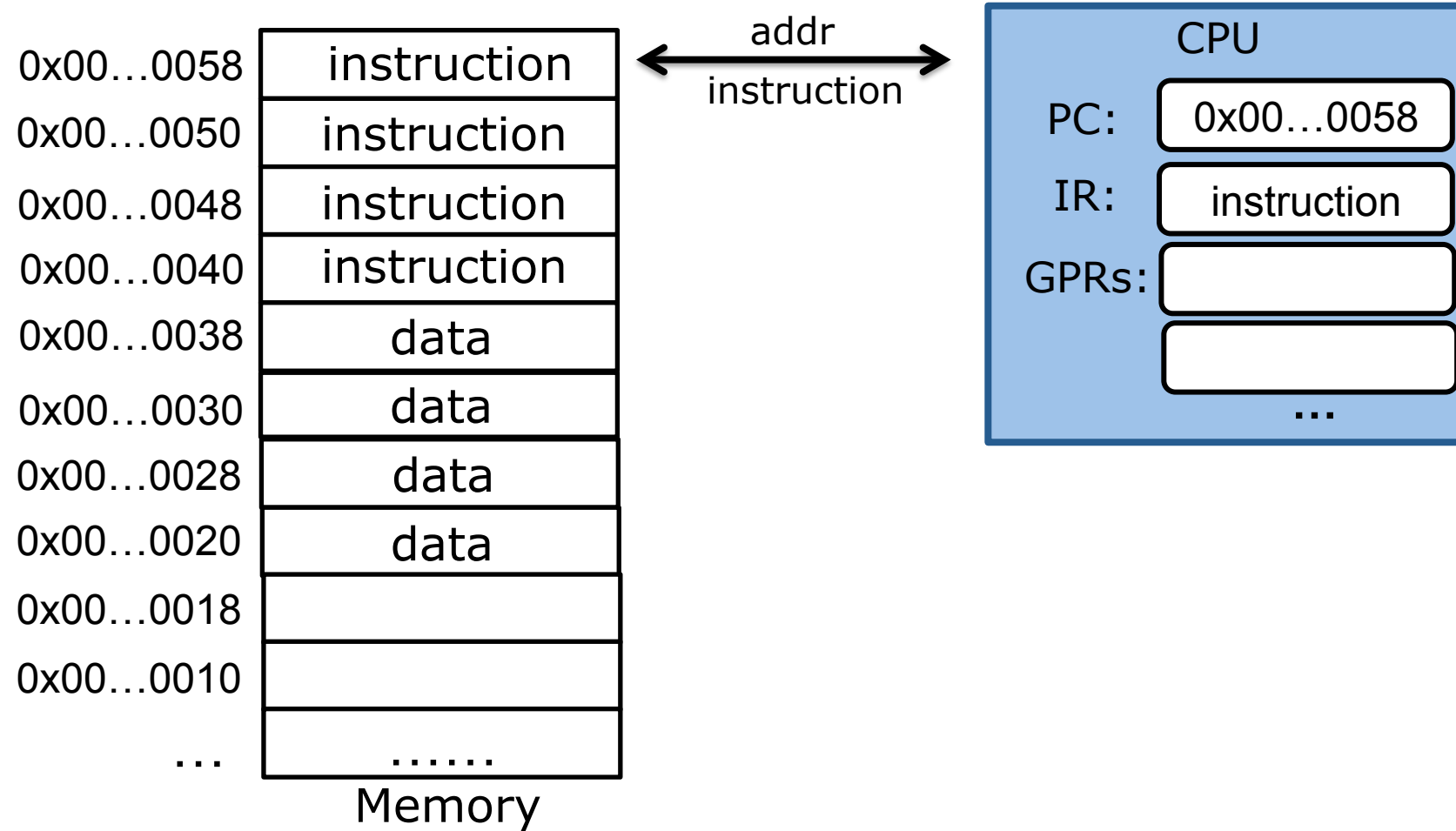
- Status of the program being executed
- All called “EFLAGS” in x86\_64

**Visible to programmers  
(aka part of hardware interface)**

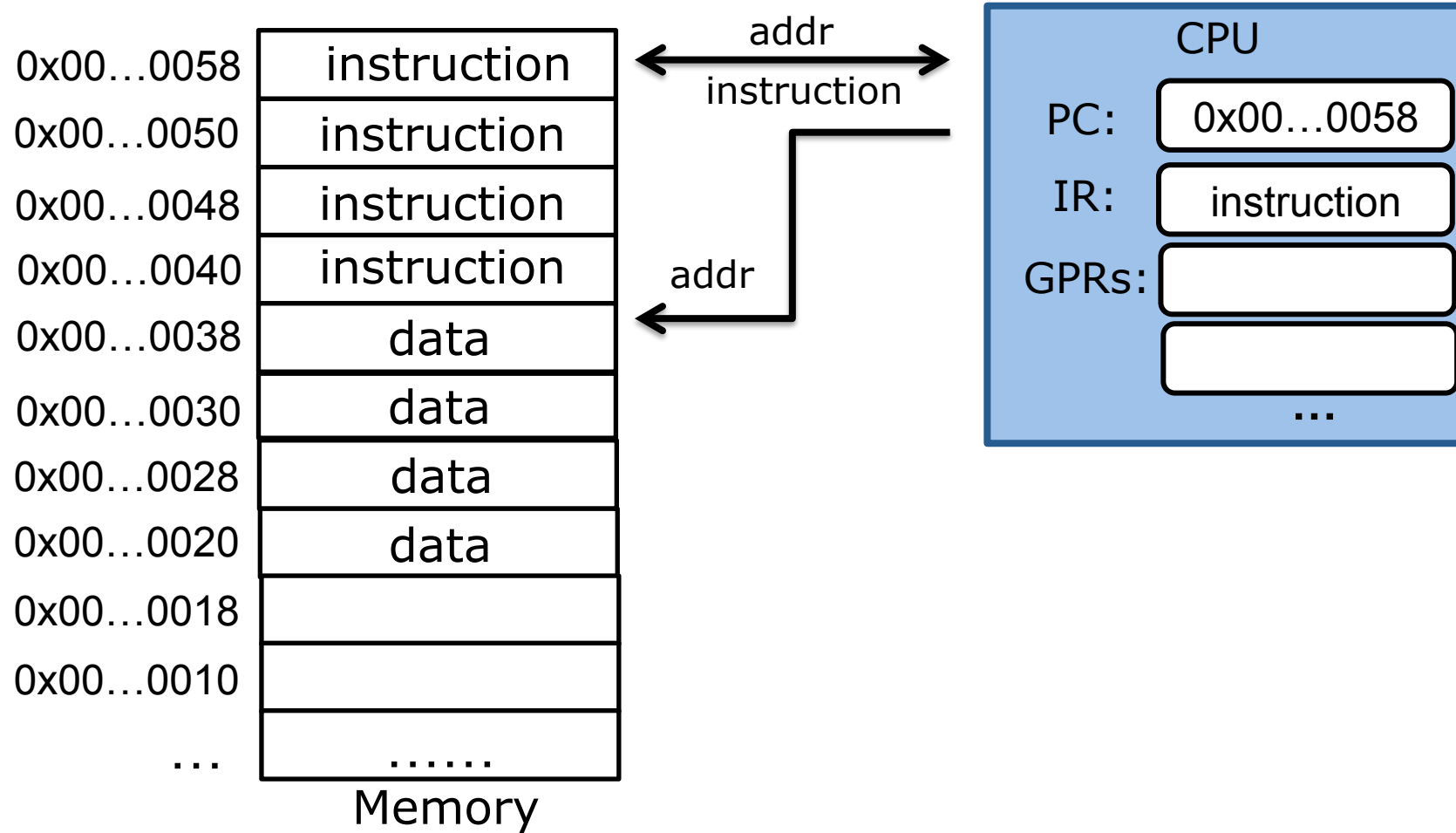
# CPU execution



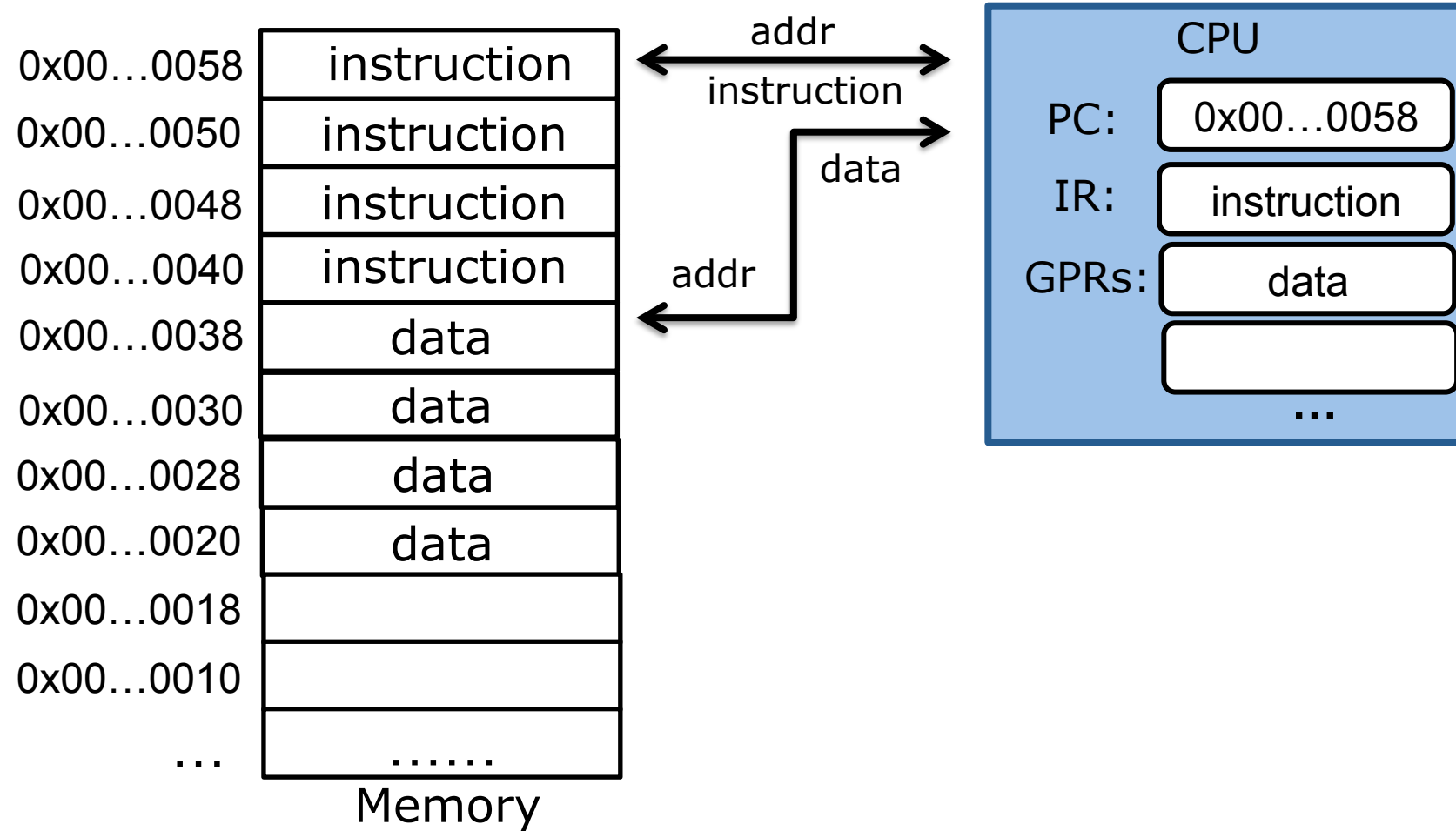
# CPU execution



# CPU execution



# CPU execution



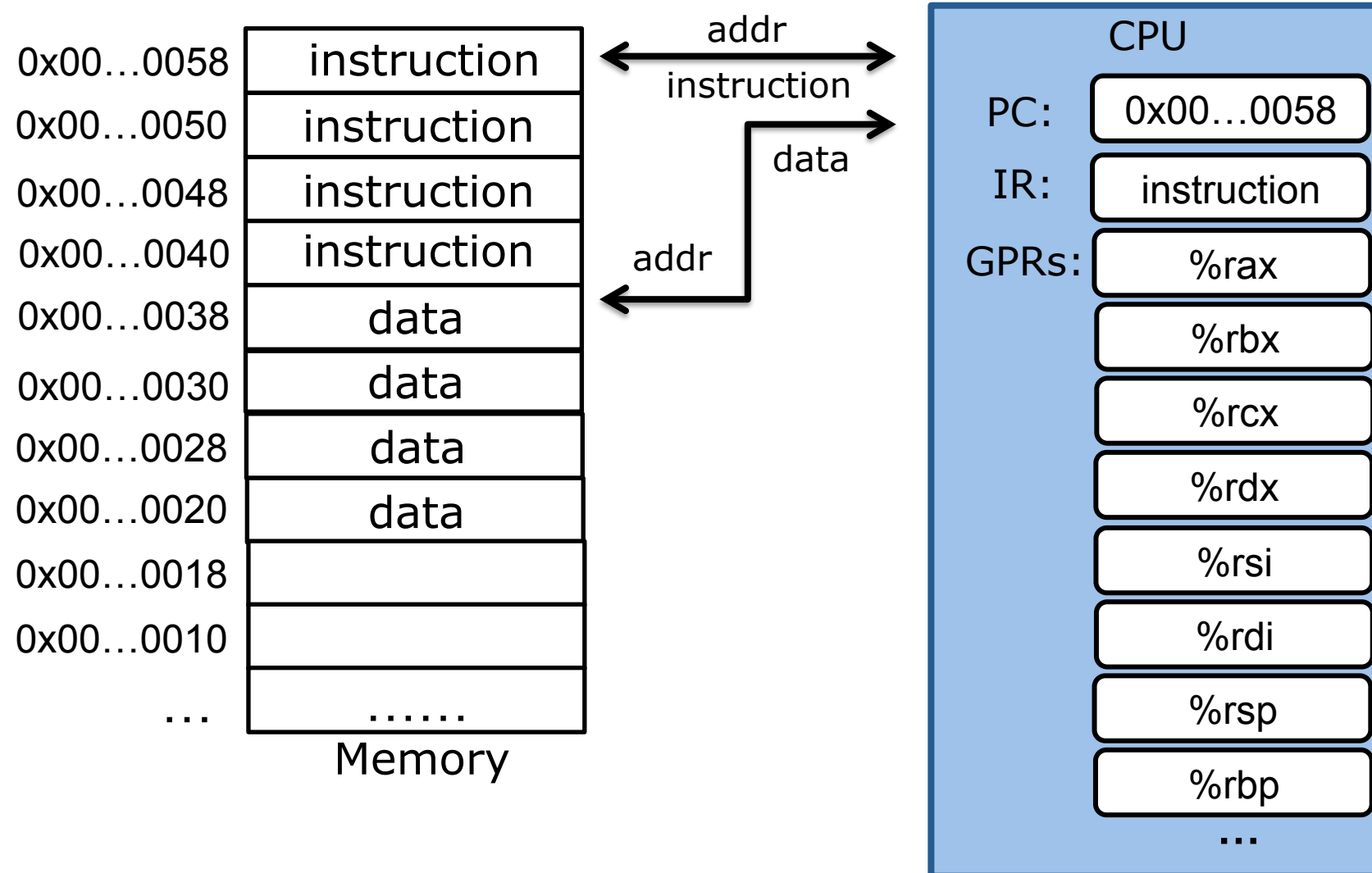
# General Purpose Registers (intel x86-64)

<b>%rax</b>
<b>%rbx</b>
<b>%rcx</b>
<b>%rdx</b>
<b>%rsi</b>
<b>%rdi</b>
<b>%rsp</b>
<b>%rbp</b>

**8 bytes**

<b>%r8</b>
<b>%r9</b>
<b>%r10</b>
<b>%r11</b>
<b>%r12</b>
<b>%r13</b>
<b>%r14</b>
<b>%r15</b>

# CPU execution





# General Purpose Registers (intel x86-64)

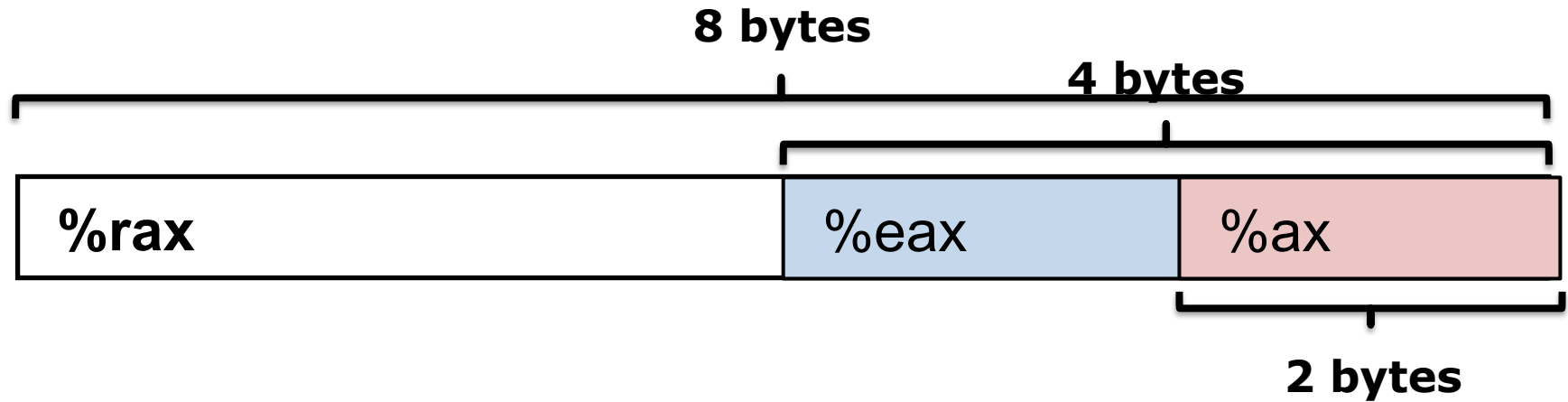
<b>%rax</b>	<b>%eax</b>
<b>%rbx</b>	<b>%ebx</b>
<b>%rcx</b>	<b>%ecx</b>
<b>%rdx</b>	<b>%edx</b>
<b>%rsi</b>	<b>%esi</b>
<b>%rdi</b>	<b>%edi</b>
<b>%rsp</b>	<b>%esp</b>
<b>%rbp</b>	<b>%ebp</b>

**8 bytes**

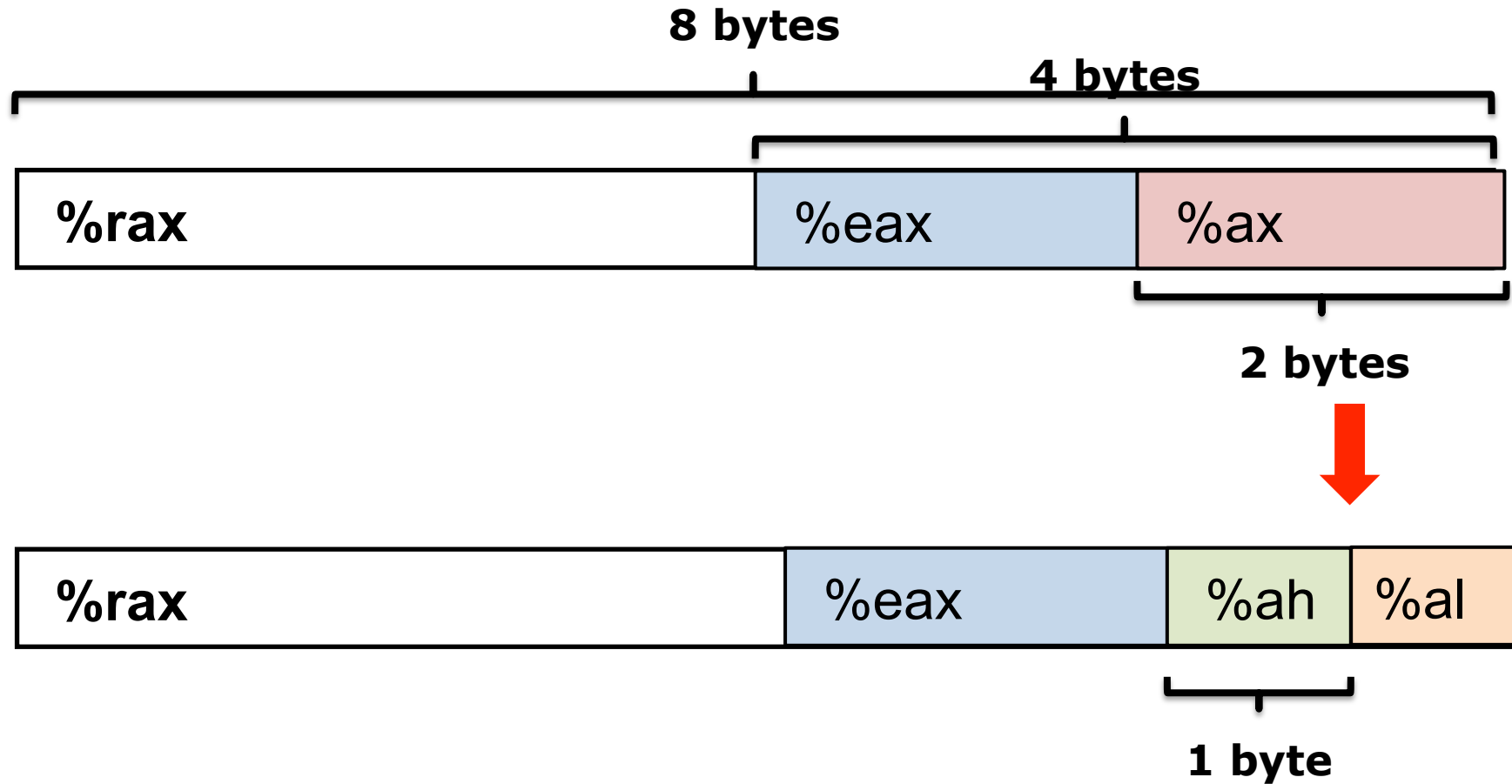
<b>%r8</b>	<b>%r8d</b>
<b>%r9</b>	<b>%r9d</b>
<b>%r10</b>	<b>%r10d</b>
<b>%r11</b>	<b>%r11d</b>
<b>%r12</b>	<b>%r12d</b>
<b>%r13</b>	<b>%r13d</b>
<b>%r14</b>	<b>%r14d</b>
<b>%r15</b>	<b>%r15d</b>

**4 bytes**

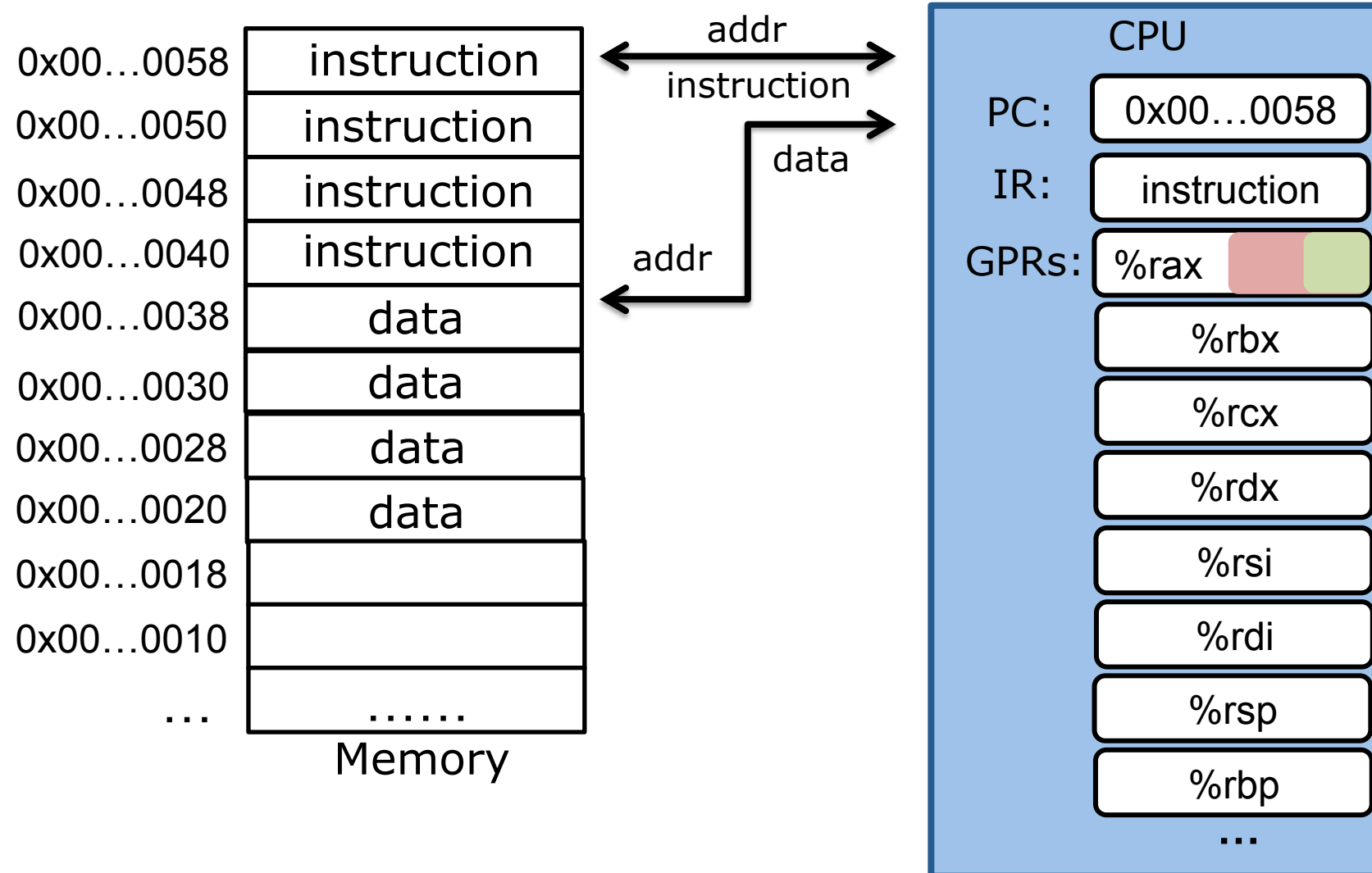
# Use %rax as an example



# Use %rax as an example



# CPU execution (intel x86-64)



# Steps of execution

1. PC contains the instruction's address
2. Fetch the instruction into IR
3. Execute the instruction

# Instruction Set Architecture (ISA)

The interface exposed by hardware to software writers

X86\_64 is the ISA implemented by Intel/AMD CPUs

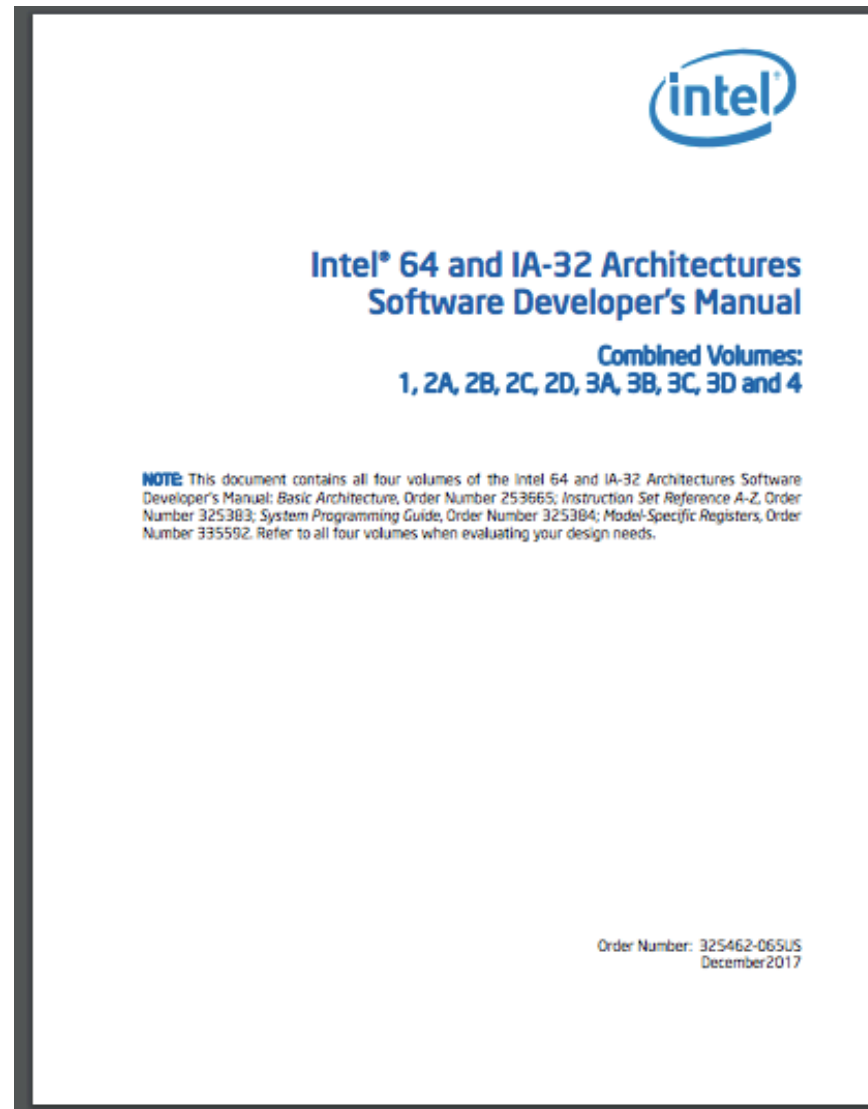
- 64-bit version of x86

← taught by CSO

ARM is another common ISA

- Phones, tablets, Raspberry Pi

# X86 ISA



A must-read for  
compiler and OS writers

<https://software.intel.com/en-us/articles/intel-sdm#combined>

# Moving data

**movq** *Source, Dest*

- Copy a quadword (64 bits) from the source operand (first operand) to the destination operand (second operand).



# Moving data

**mov** *Source, Dest*

- Copy a quadword (64 bits) from the source operand (first operand) to the destination operand (second operand).

Suffix	Name	Size (byte)
B	Byte	1
W	Word	2
L	Long	4
Q	Quadword	8

# Why using a size suffix?

**movq** *Source, Dest*

- Copy a quadword (64 bits) from the source operand (first operand) to the destination operand (second operand).
- In the Intel x86 world , a word = 16 bits.
  - 8086 uses 16 bits as a word
  - Support **full backward compatibility**
    - New processor can run the same binary file compiled for older processors

# Moving data

**movq** *Source, Dest*

## Operand Types

- **Immediate:** Constant integer data
  - Prefixed with \$
  - Example: \$0x400, \$-533
- **Register:** One of general purpose registers
  - Example: %rax, %rsi
- **Memory:** 8 consecutive bytes of memory
  - Indexed by register with various “address modes”
  - Simplest example: (%rax)

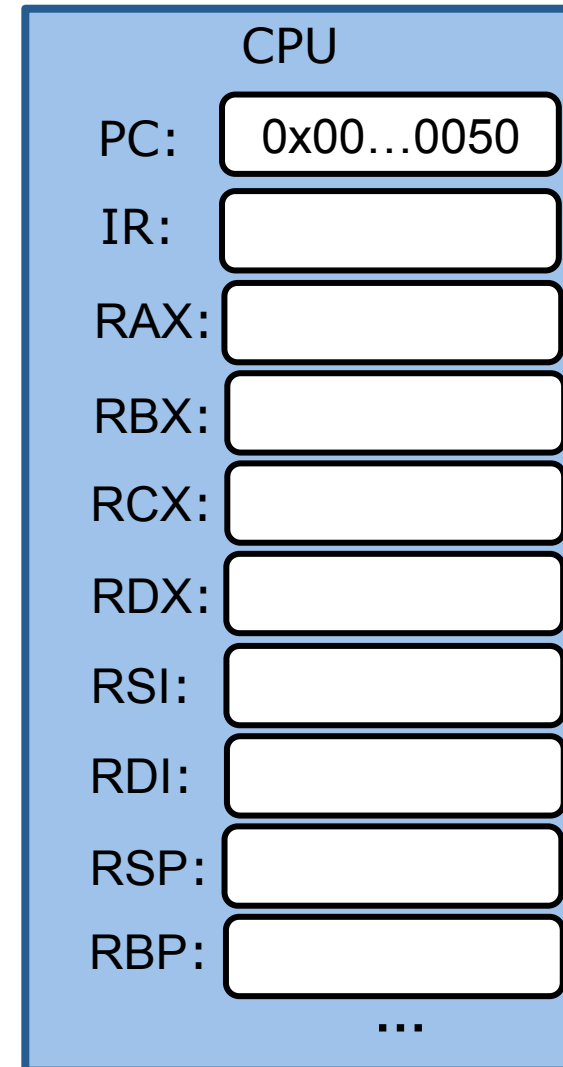
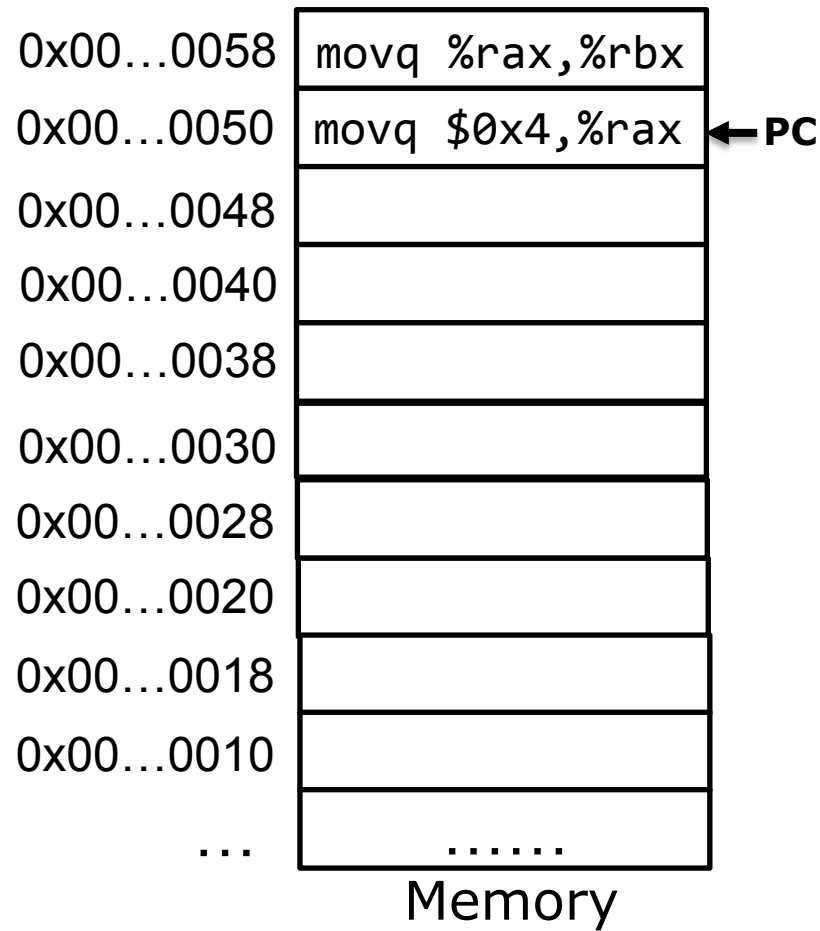
# movq Operand combinations

	Source	Dest	Source, Dest
movq	Imm	Reg	movq \$0x4,%rax
		Mem	movq \$0x4, (%rax)
	Reg	Reg	movq %rax,%rdx
		Mem	movq %rax, (%rdx)
	Mem	Reg	movq (%rax),%rdx

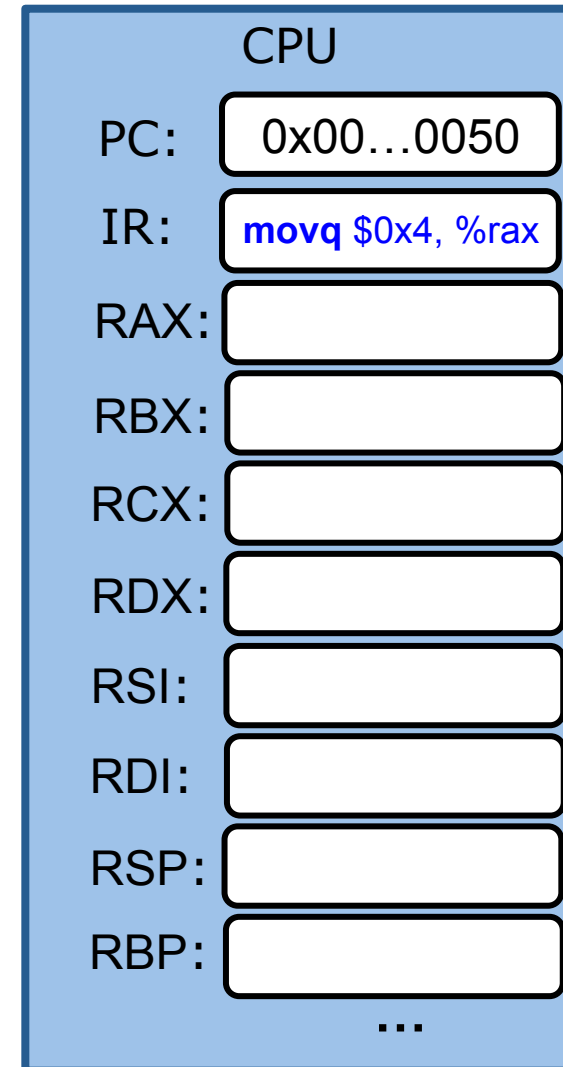
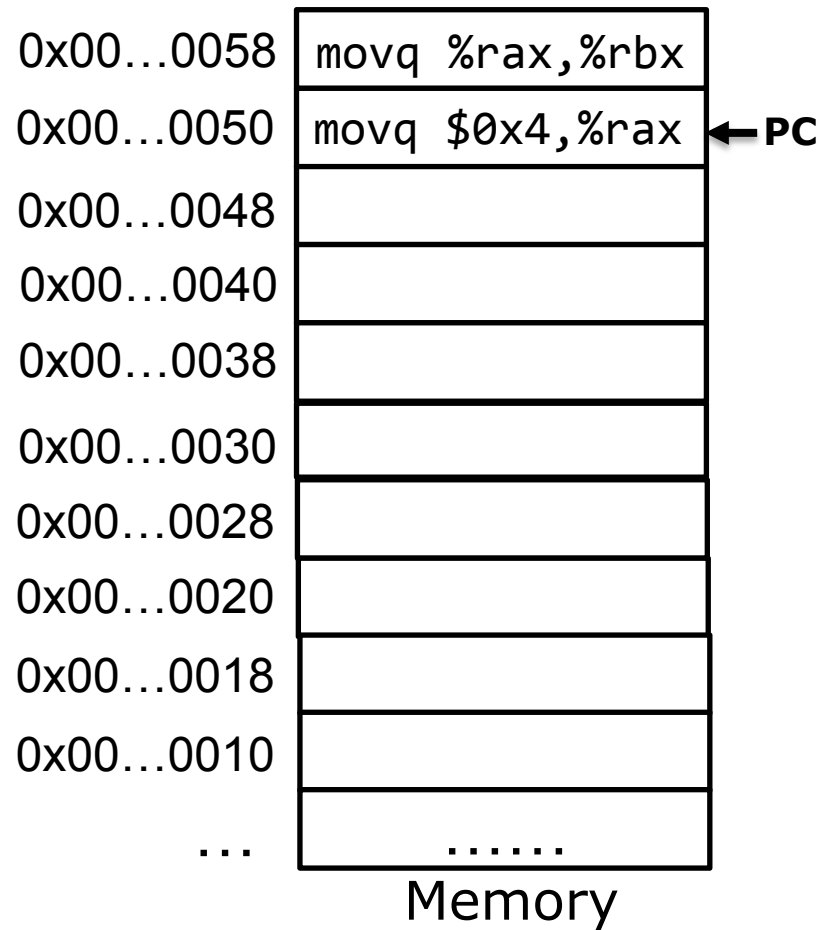
1. Immediate can only be *Source*

2. No memory-memory mov

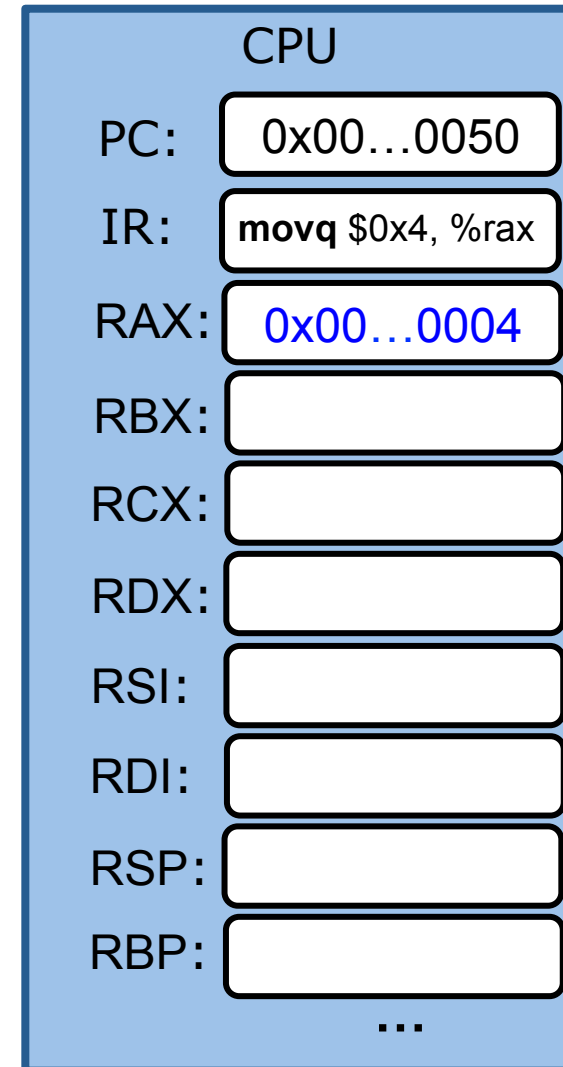
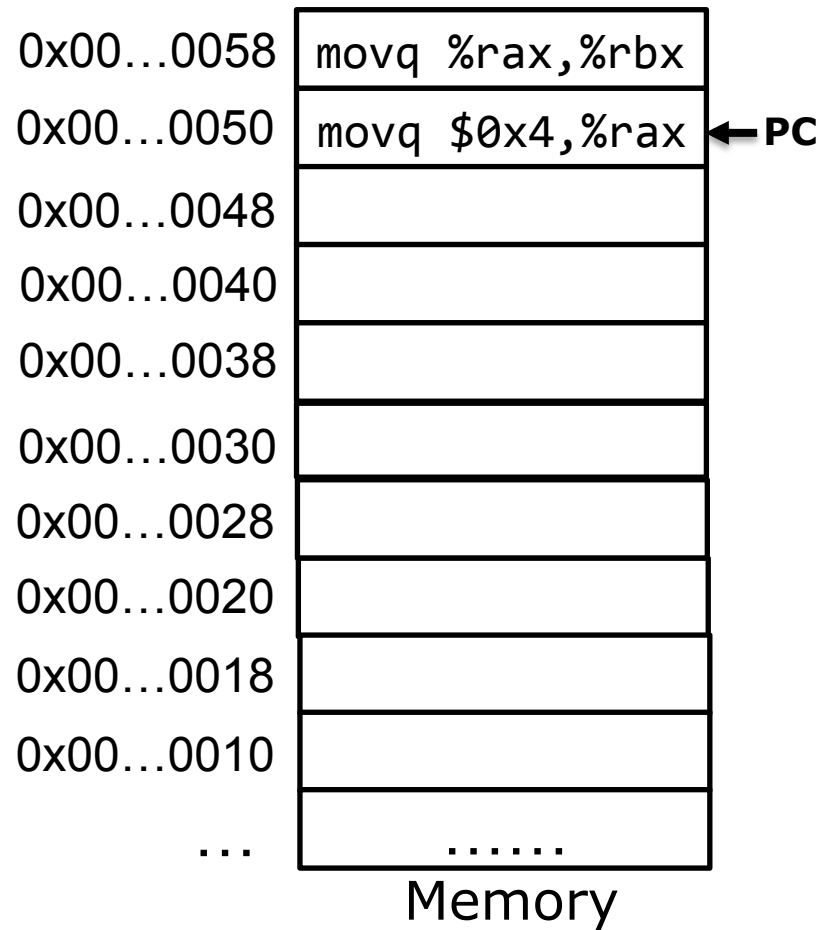
# movq Imm, Reg



# movq Imm, Reg



# movq Imm, Reg

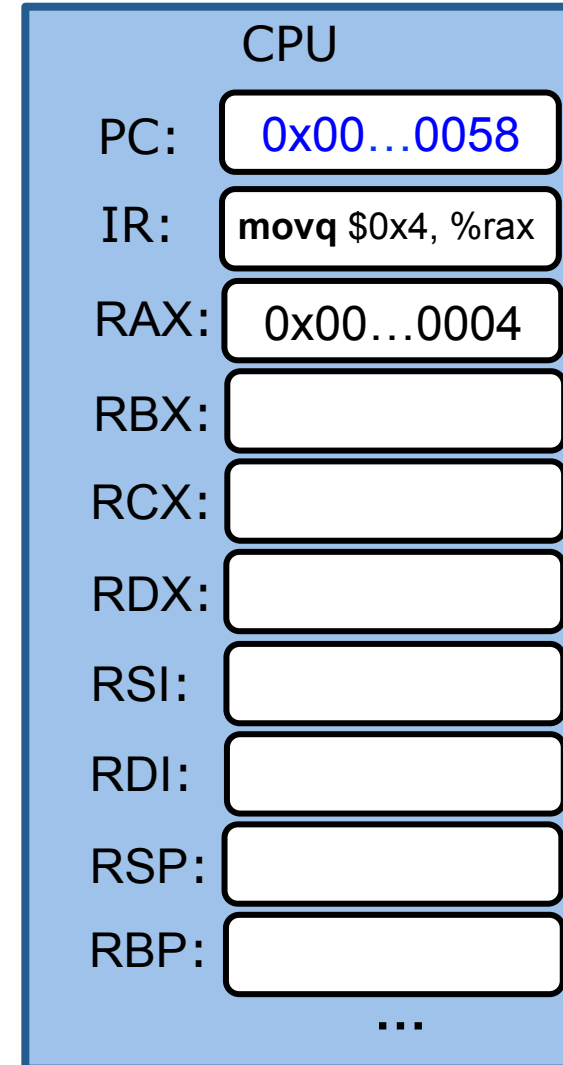
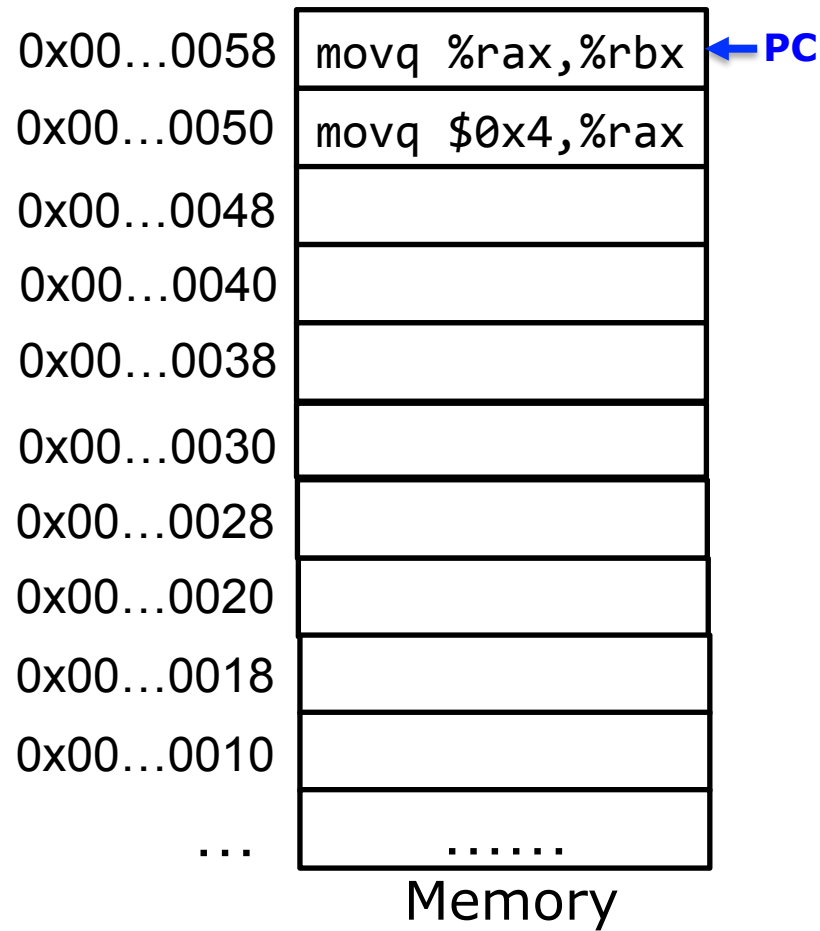


# Steps

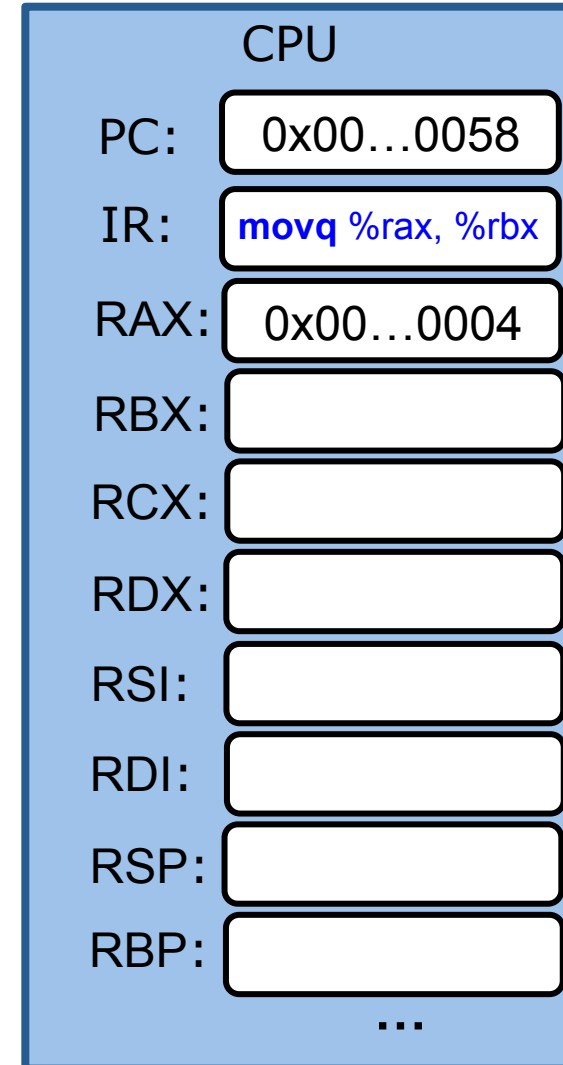
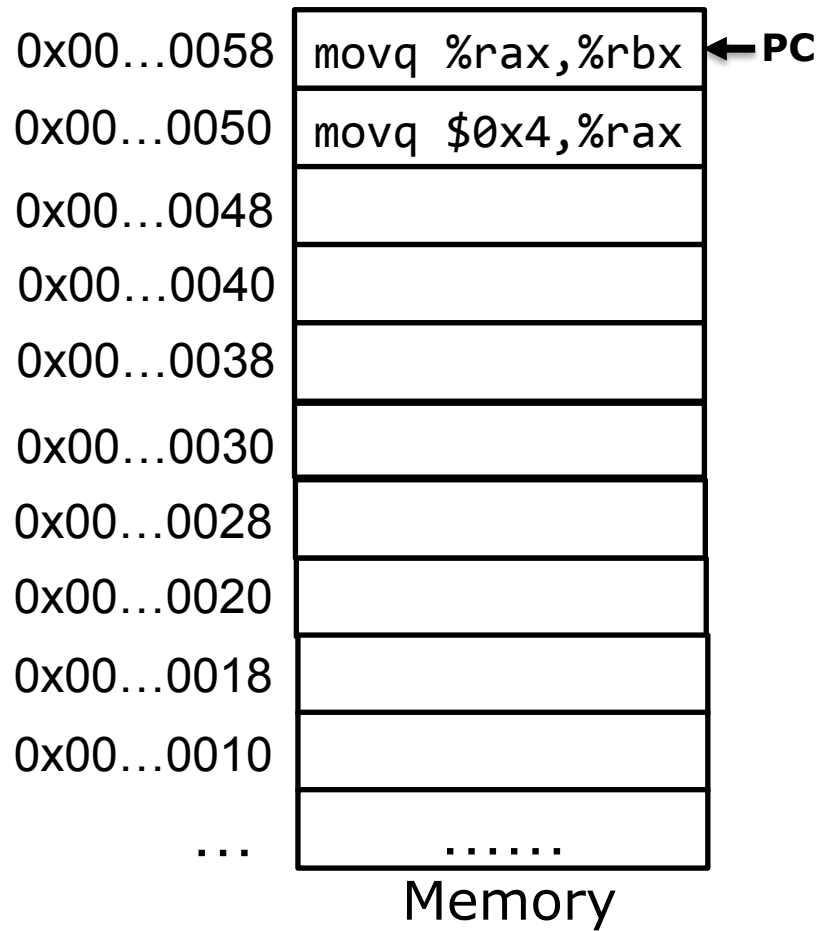
1. PC contains the instruction's address
2. Load the instruction into IR
3. Execute the instruction
4. CPU automatically updates PC after current instruction finishes (is retired).



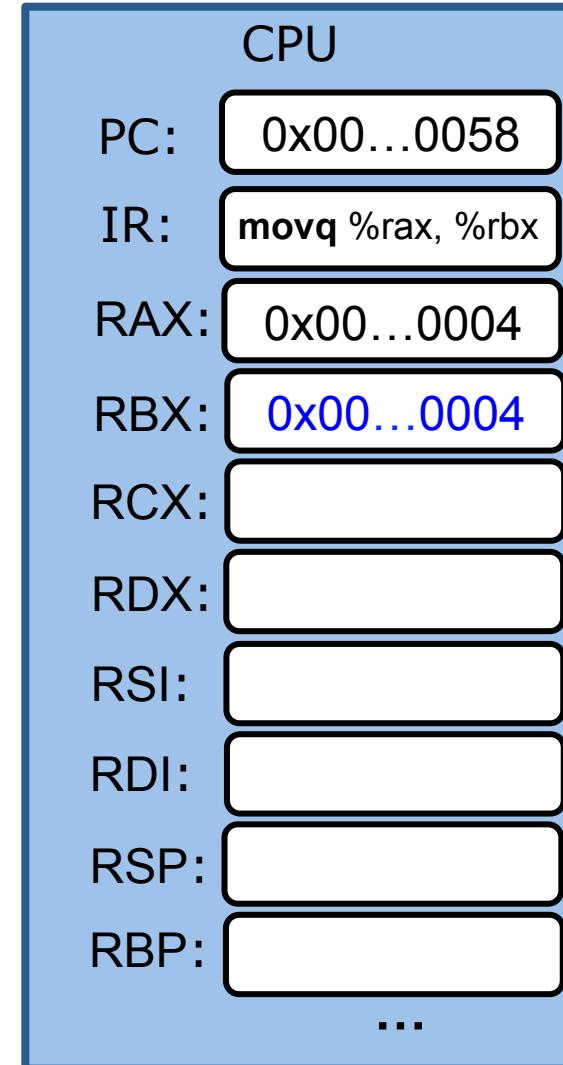
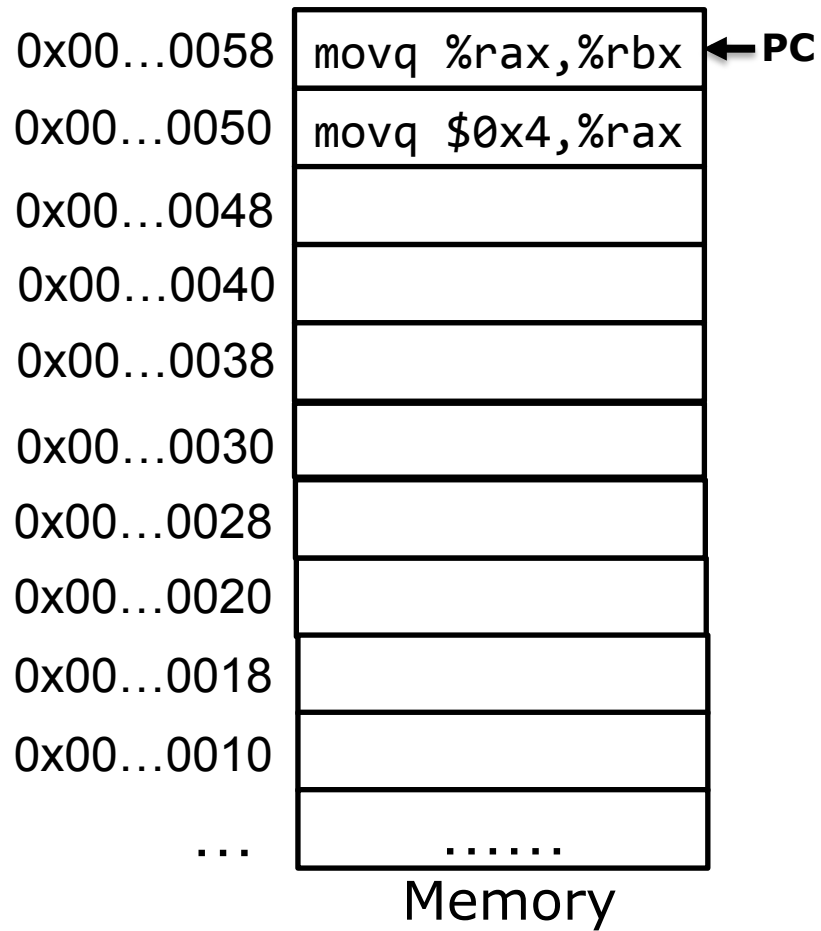
# movq *Reg, Reg*



# movq *Reg, Reg*



# movq *Reg, Reg*



**movq** *Mem, Reg*

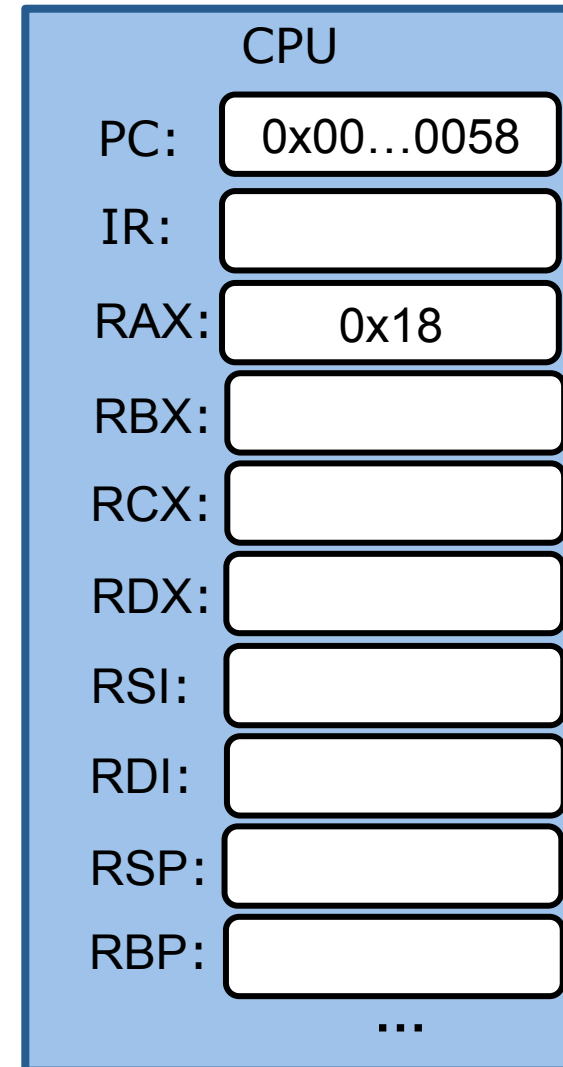
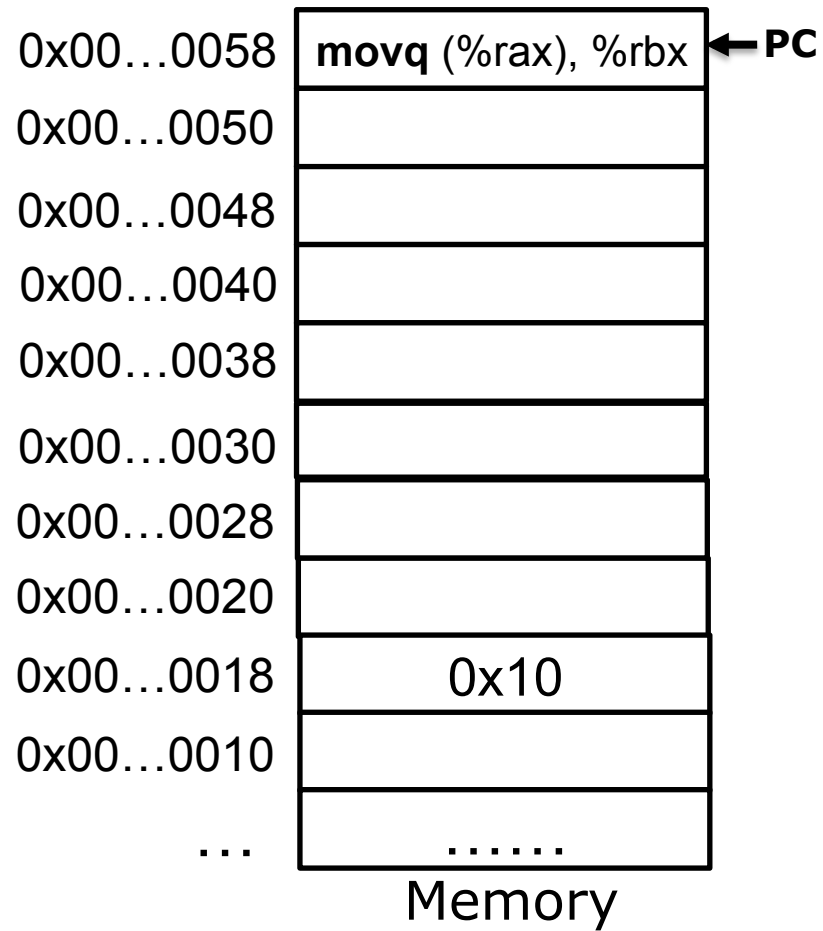
How to represent a “memory” operand?

# Direct addressing: use registers to index the memory

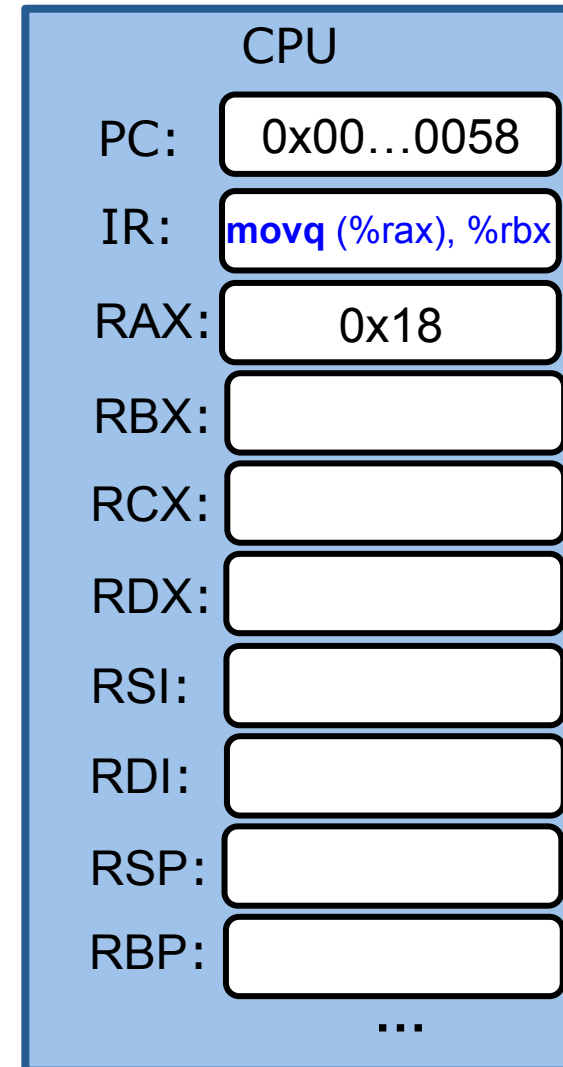
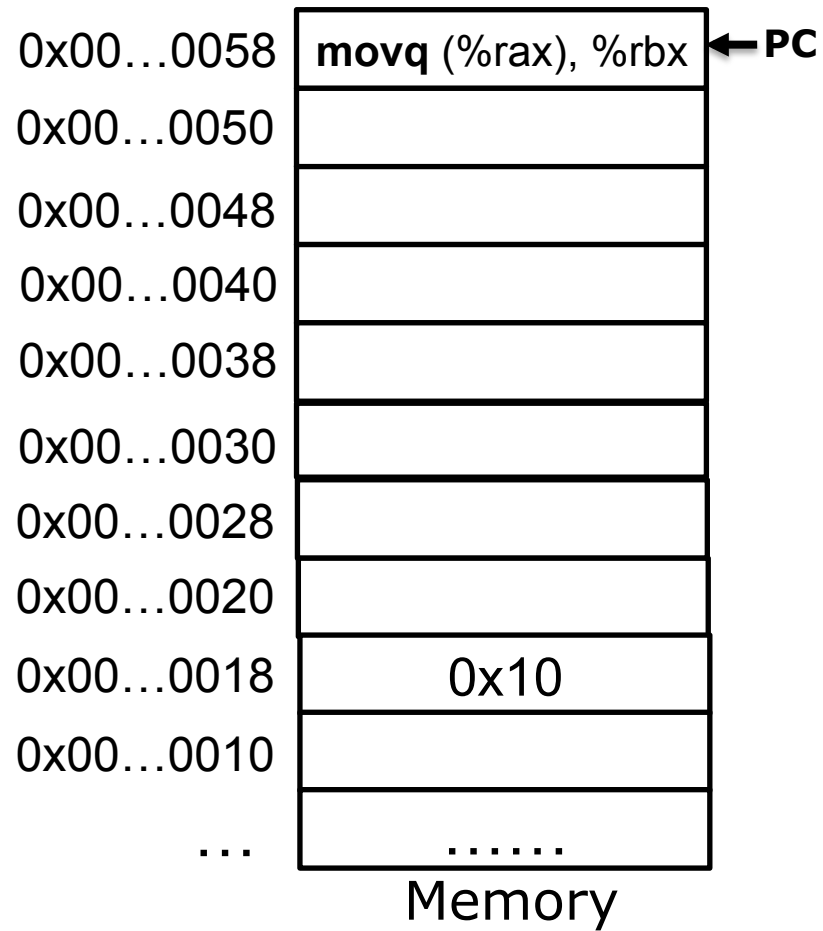
(Register)

- The content of the register specifies memory address
- `movq (%rax), %rbx`

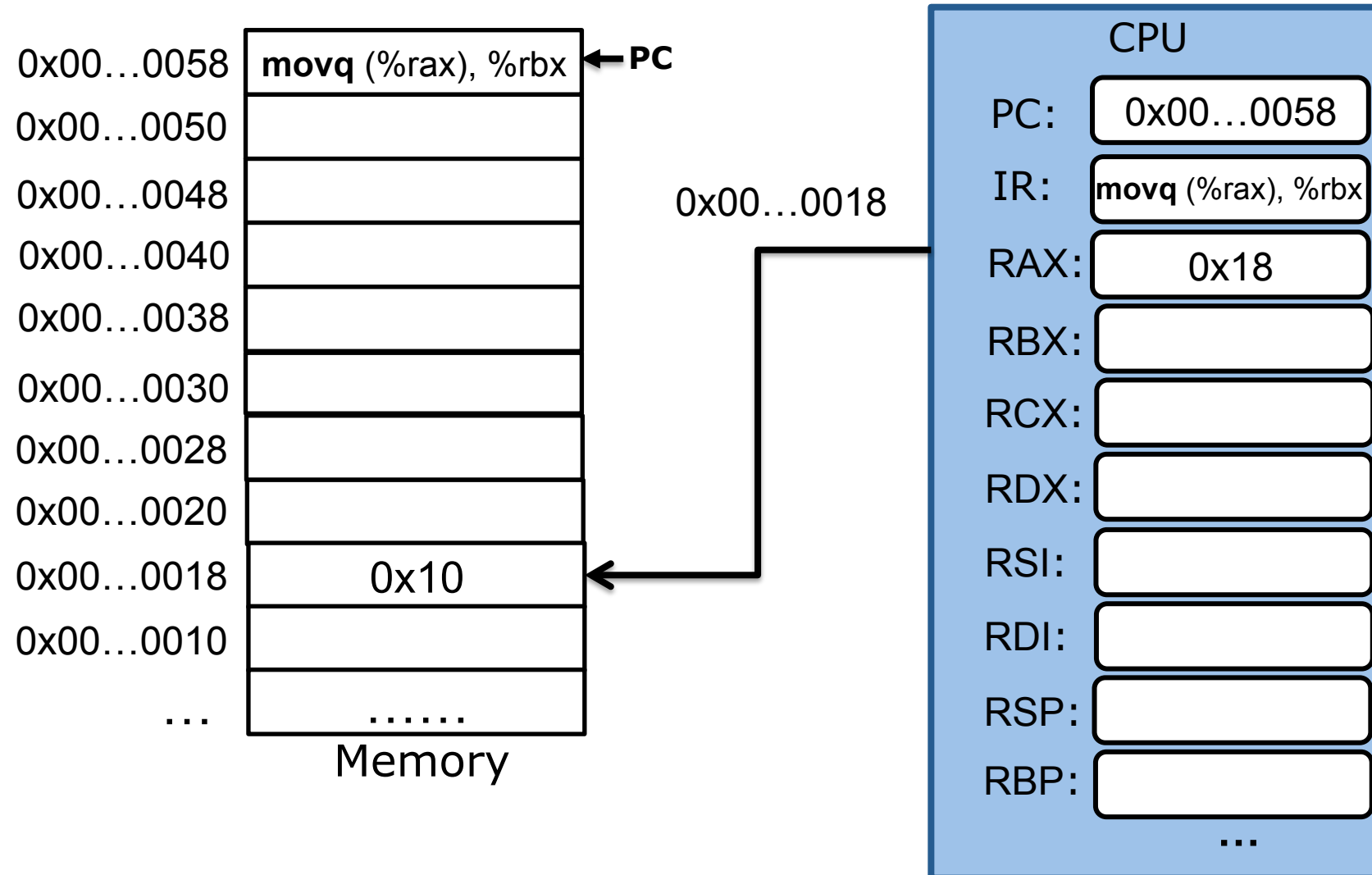
# movq (%rax), %rbx



# movq (%rax), %rbx

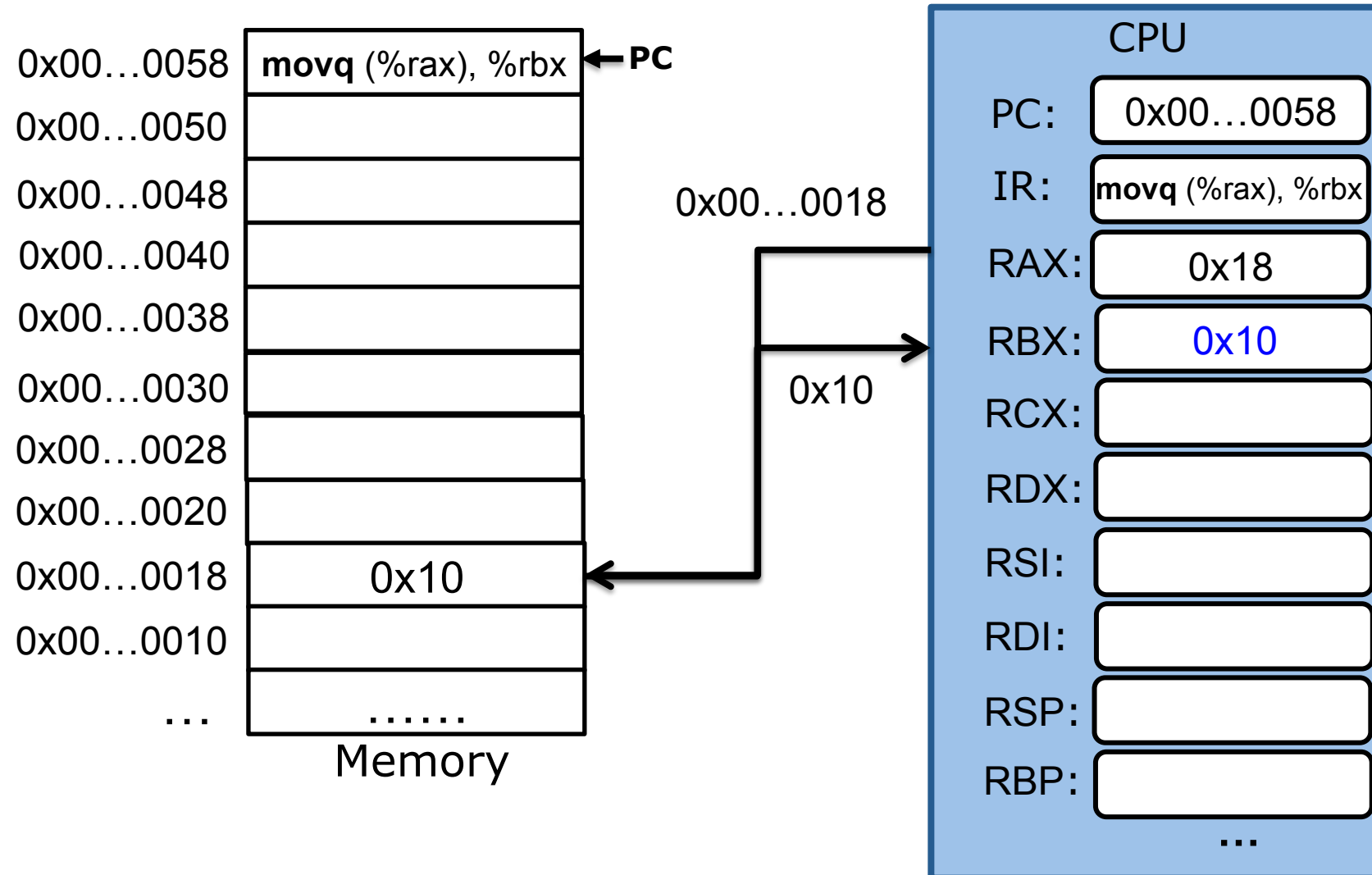


# movq (%rax), %rbx






# movq (%rax), %rbx



# swap function

```
void  
swap(long *a, long* b) {  
    long tmp = *a;  
    *a = *b;  
    *b = tmp;  
}
```

**swap:**



gcc -S -O3 swap.c

# swap function

```
void  
swap(long *a, long* b) {  
    long tmp = *a;  
    *a = *b;  
    *b = tmp;  
}
```

gcc -S -O3 swap.c

swap:

```
movq    (%rdi), %rax  
movq    (%rsi), %rdx  
movq    %rdx, (%rdi)  
movq    %rax, (%rsi)
```

%rdi stores a

%rsi stores b

%rax is local variable tmp

# swap function

```
void  
swap(long *a, long* b) {  
  
    long tmp = *a;  
    *a = *b;  
    *b = tmp;  
  
}
```

gcc -S -O3 swap.c

swap:

```
movq    (%rdi), %rax  
movq    (%rsi), %rdx  
movq    %rdx, (%rdi)  
movq    %rax, (%rsi)
```

Use two instructions and %rdx to perform  
memory to memory move

# swap function

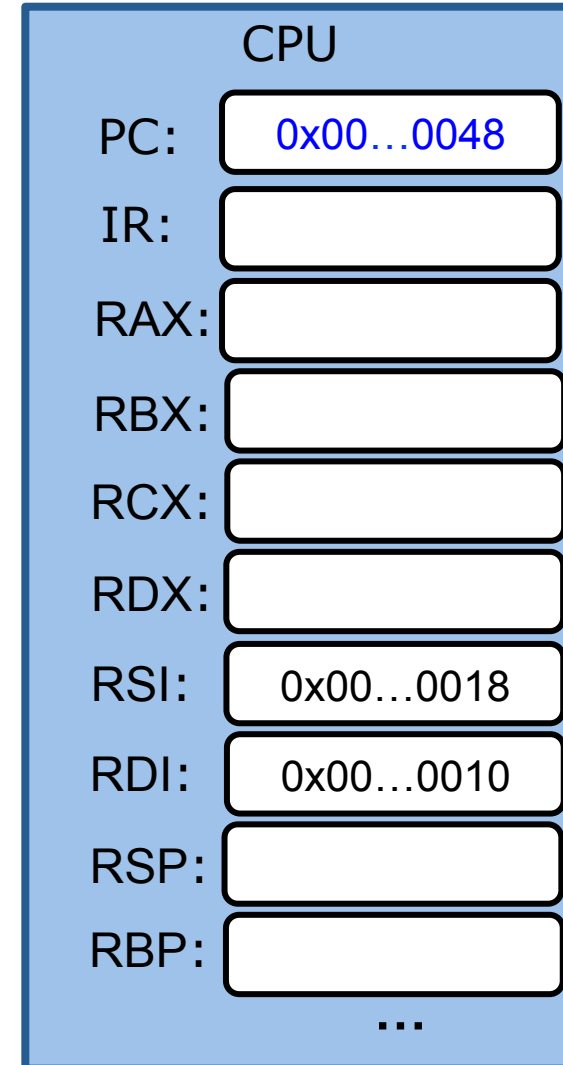
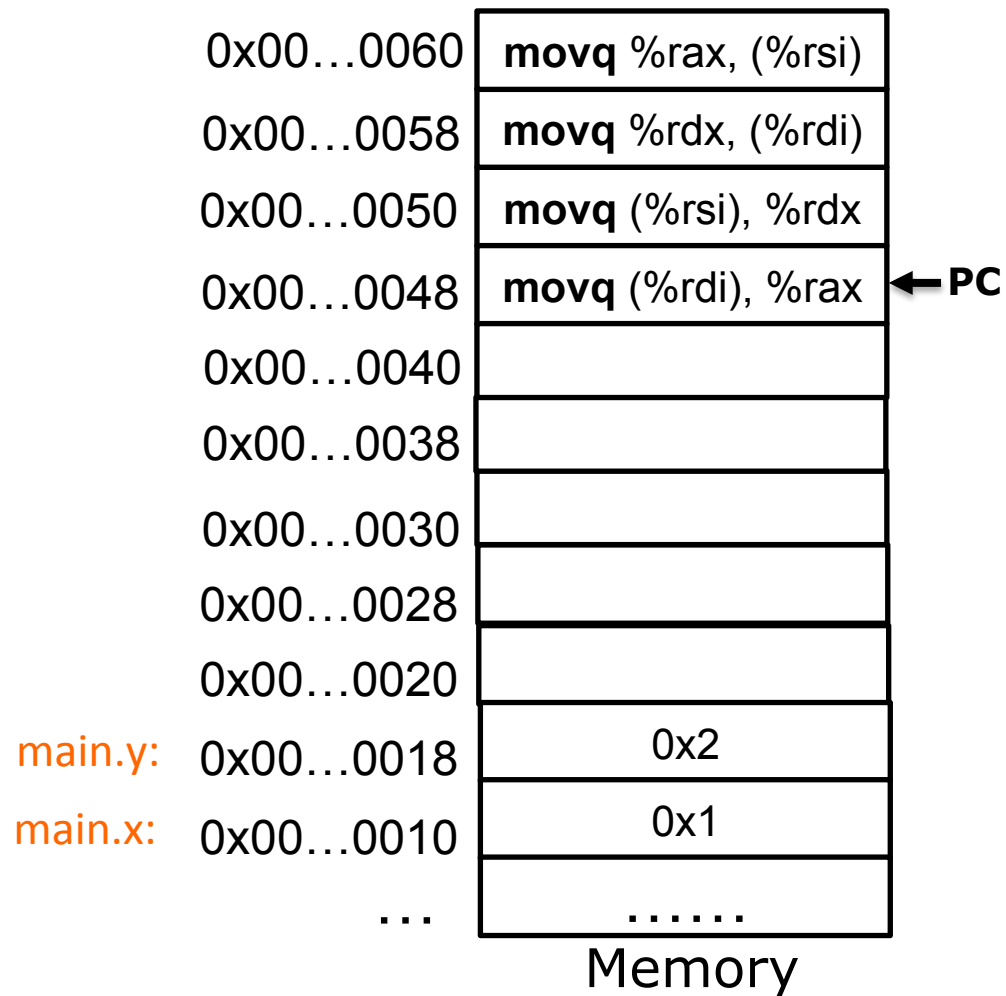
```
void  
swap(long *a, long* b) {  
  
    long tmp = *a;  
    *a = *b;  
    *b = tmp;  
  
}
```



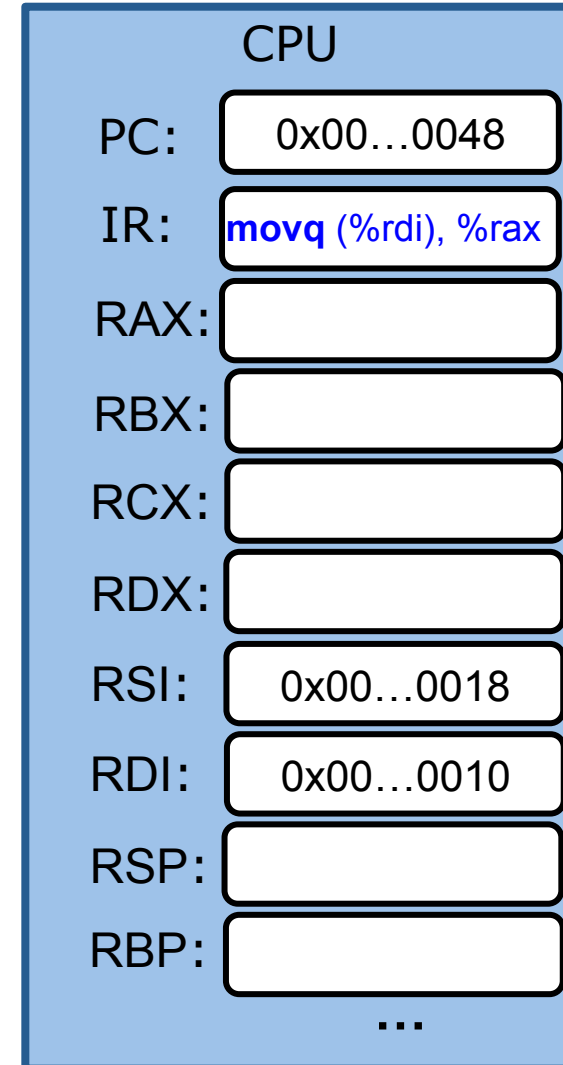
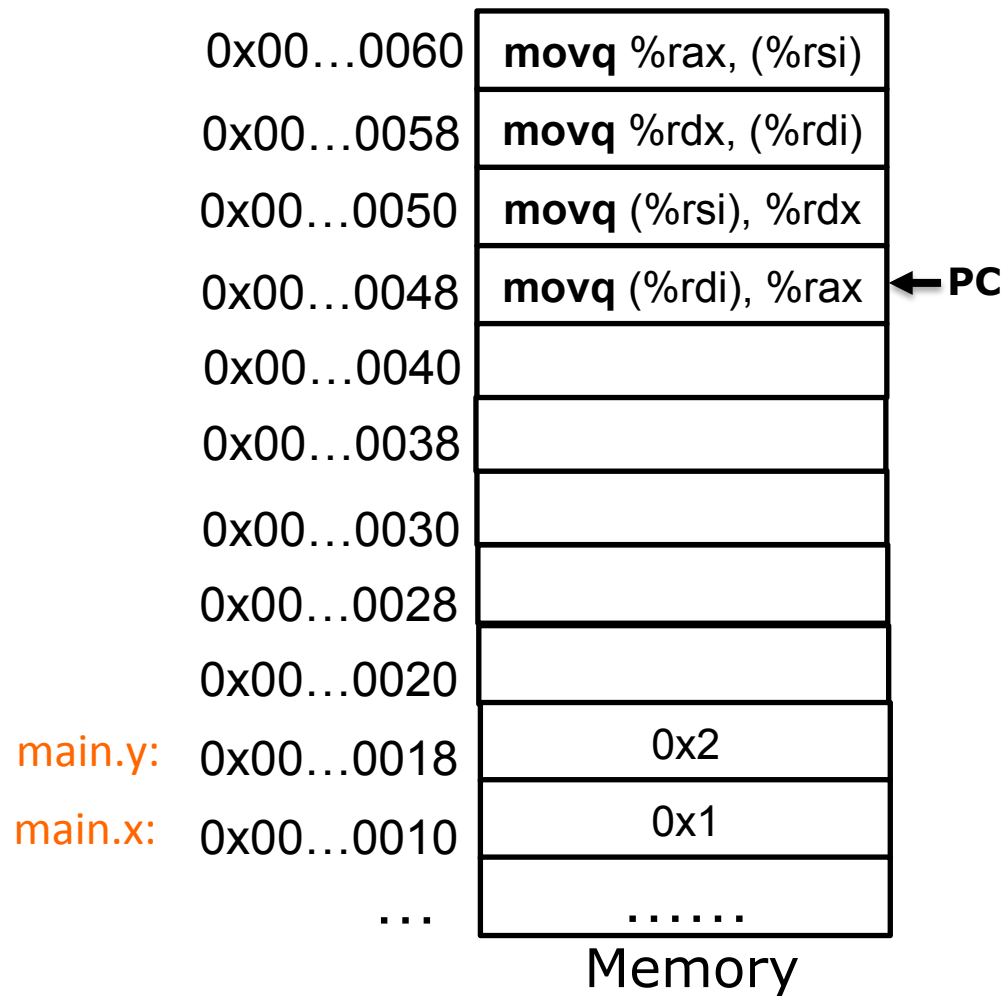
gcc -S -O3 swap.c

```
swap:  
    movq    (%rdi), %rax  
    movq    (%rsi), %rdx  
    movq    %rdx, (%rdi)  
    movq    %rax, (%rsi)
```

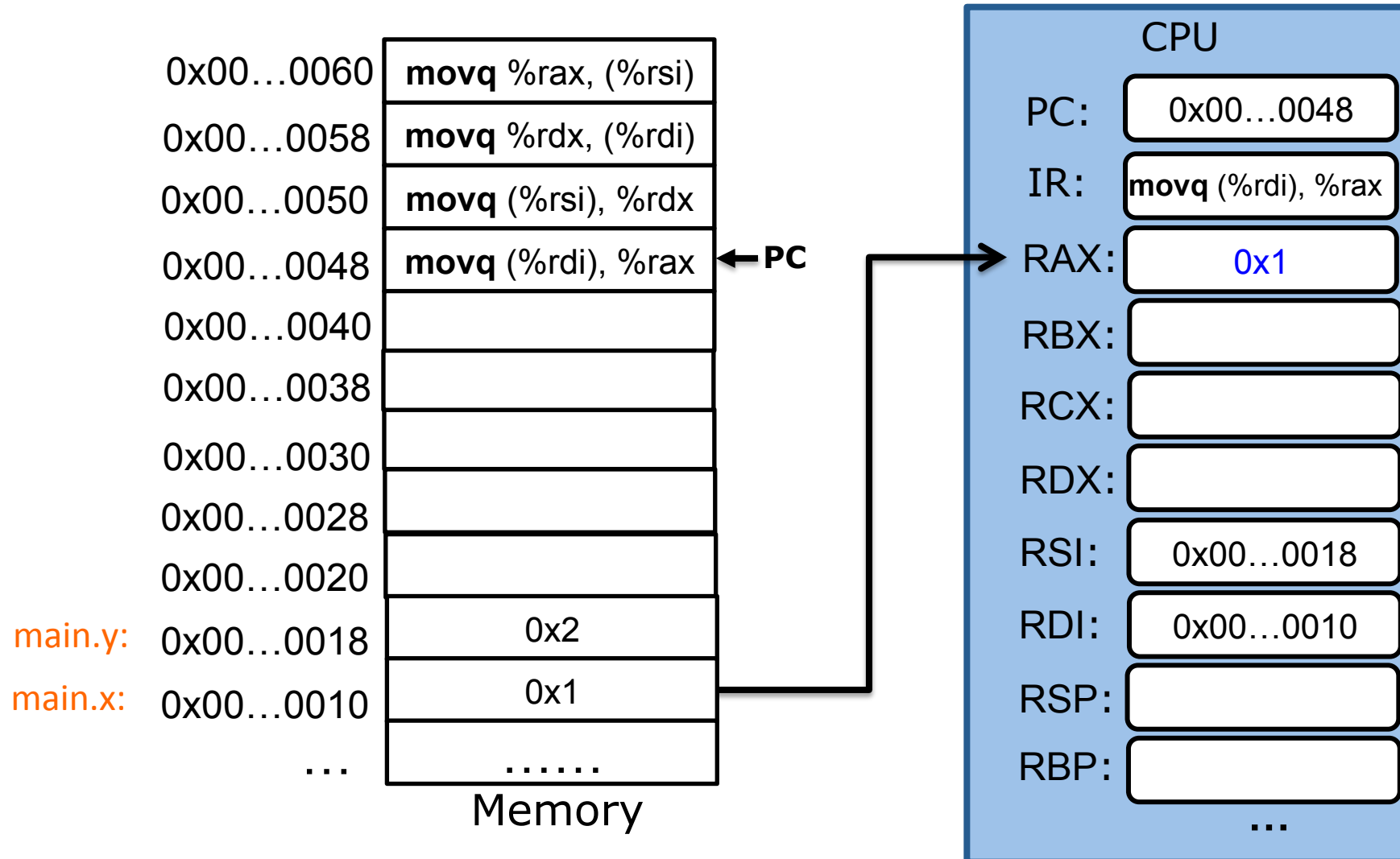
# swap func



# swap func

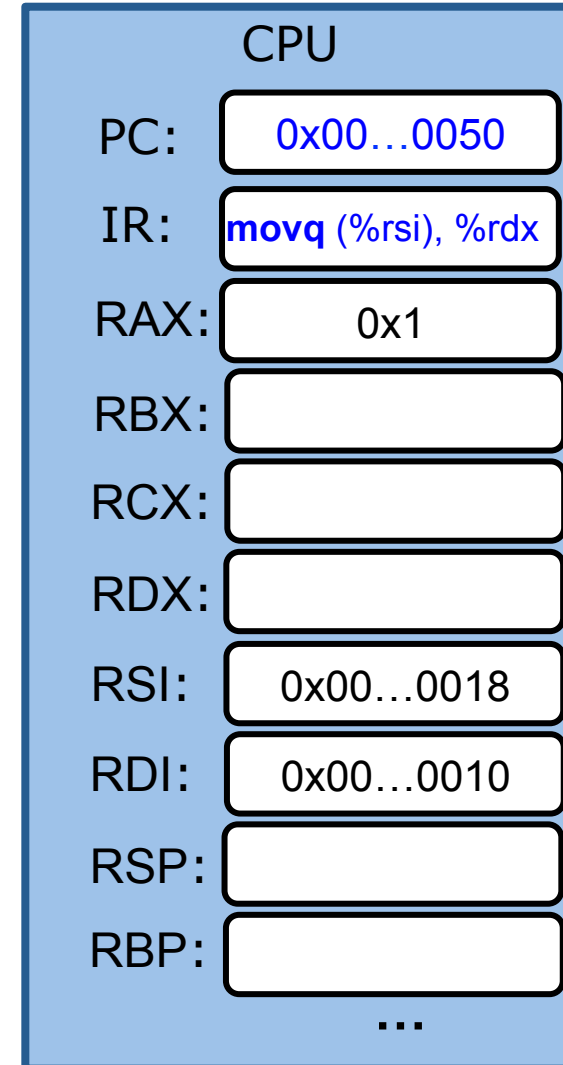
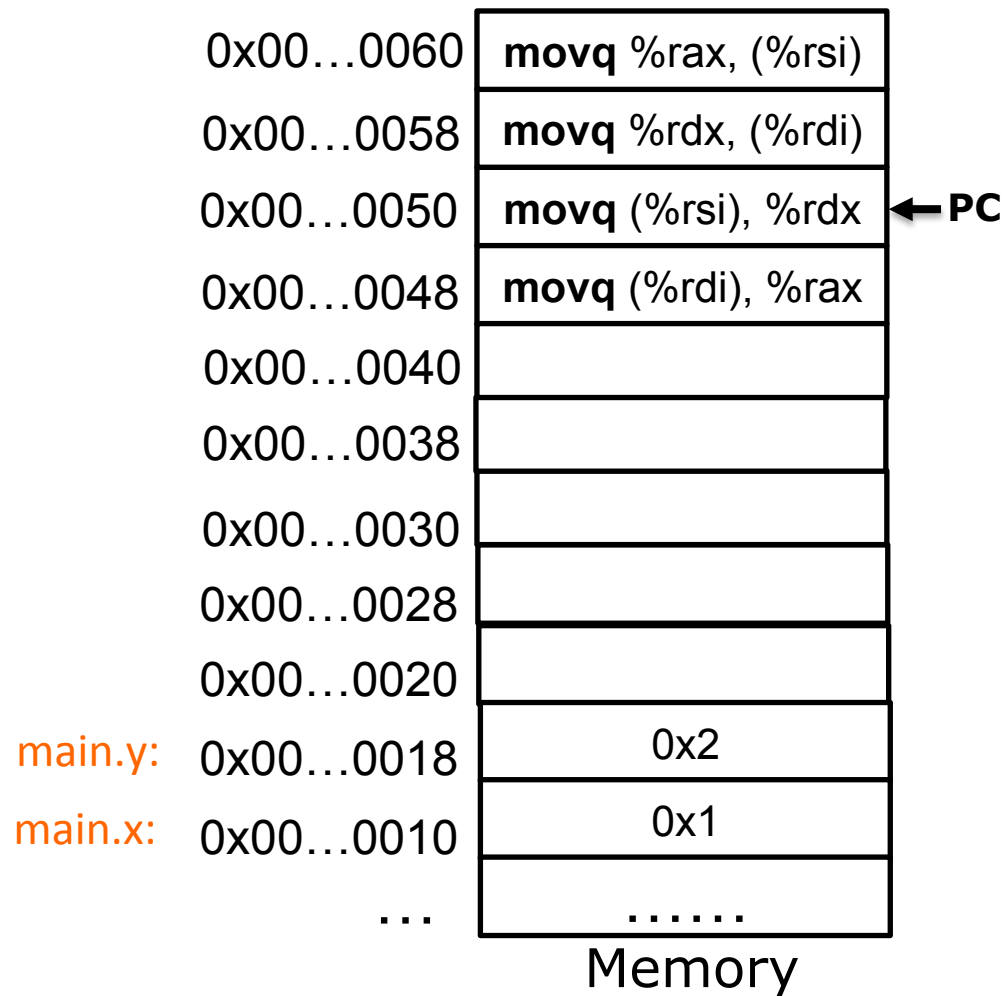


# swap func

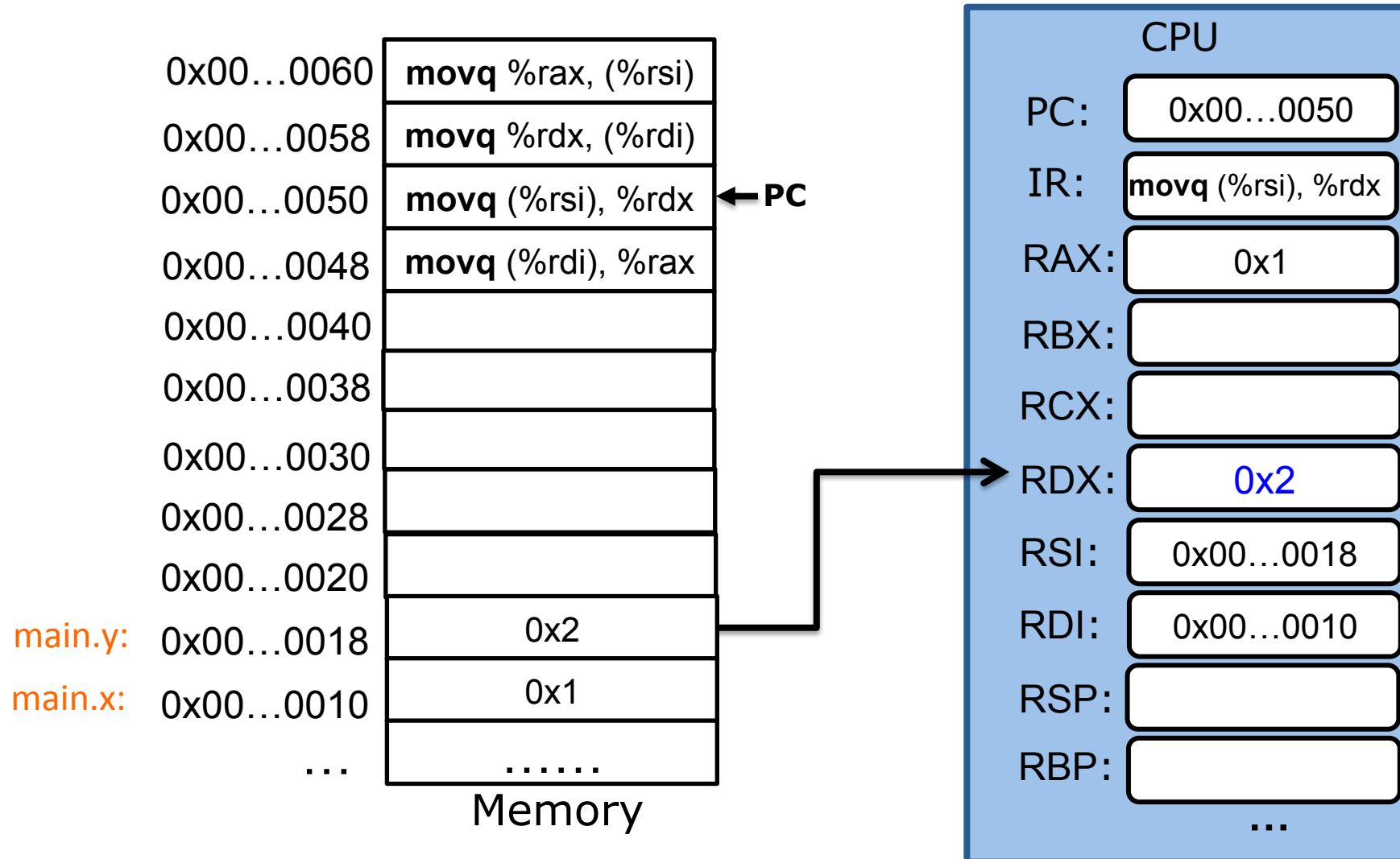




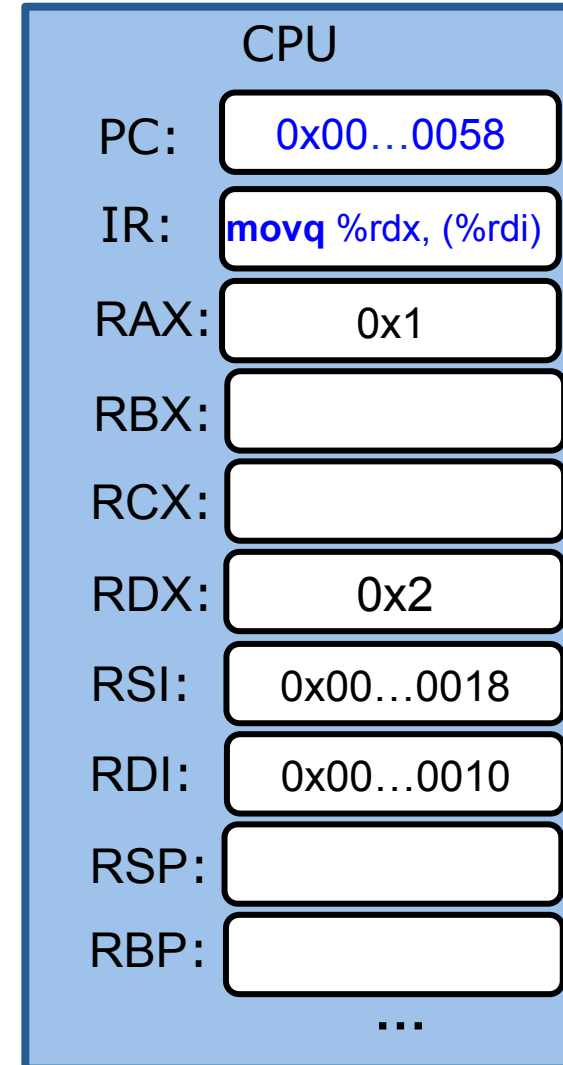
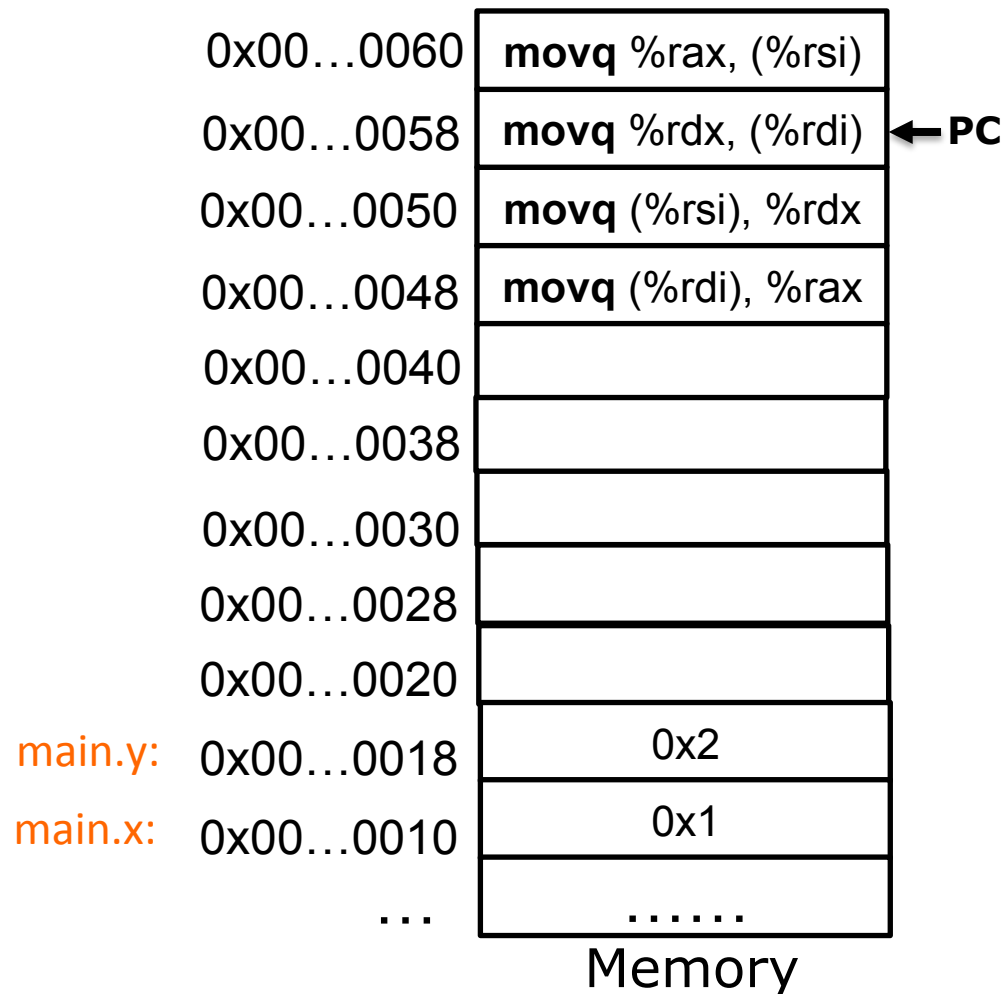
# swap func



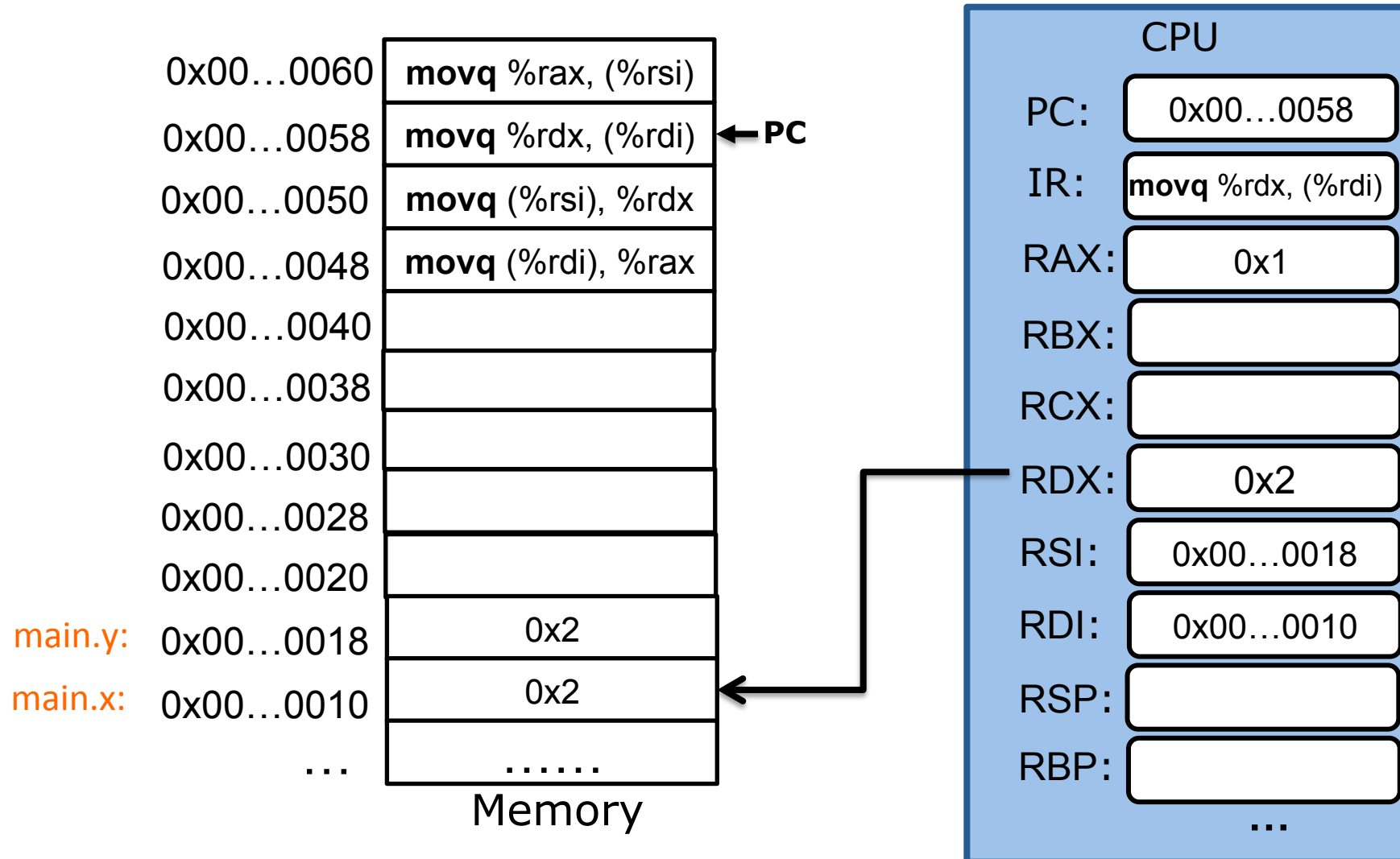
# swap func



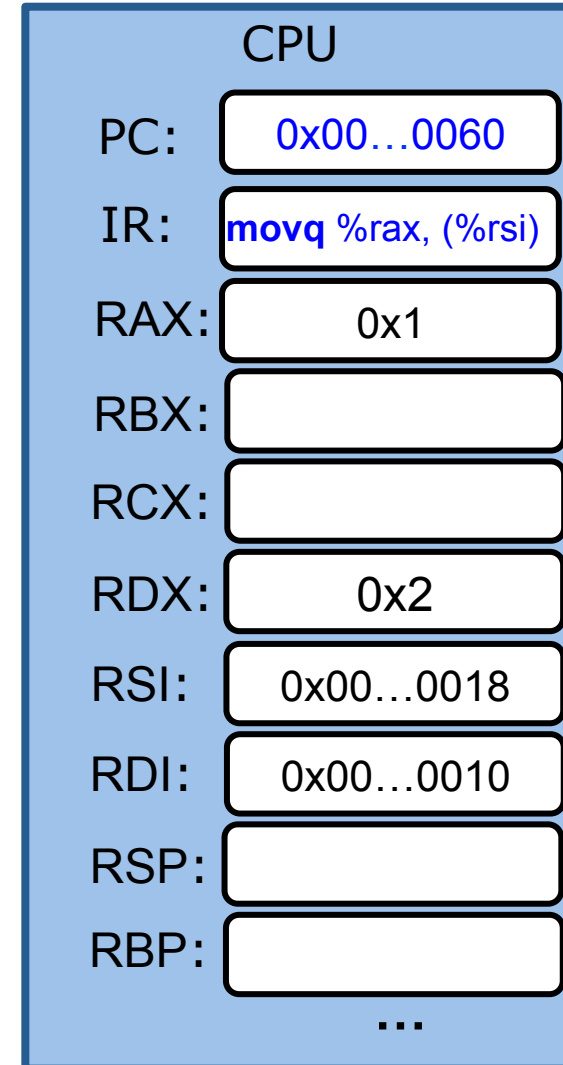
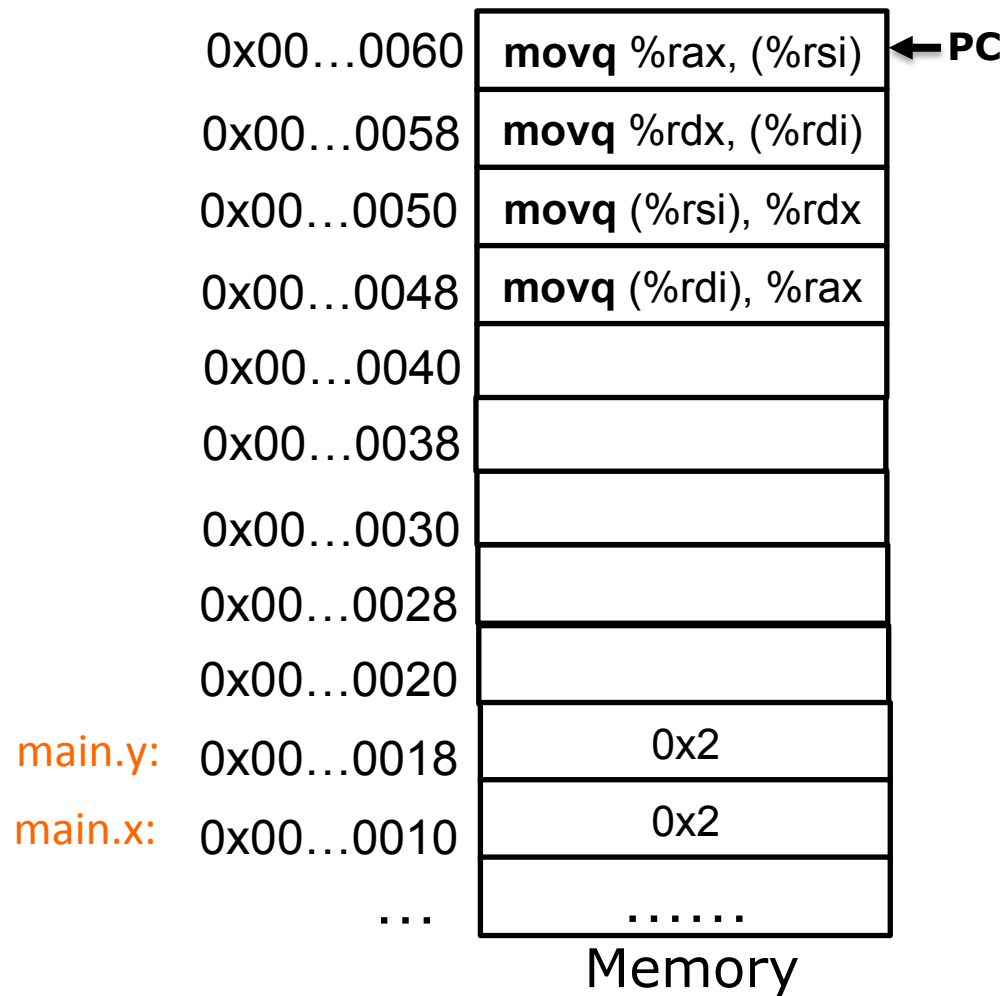
# swap func



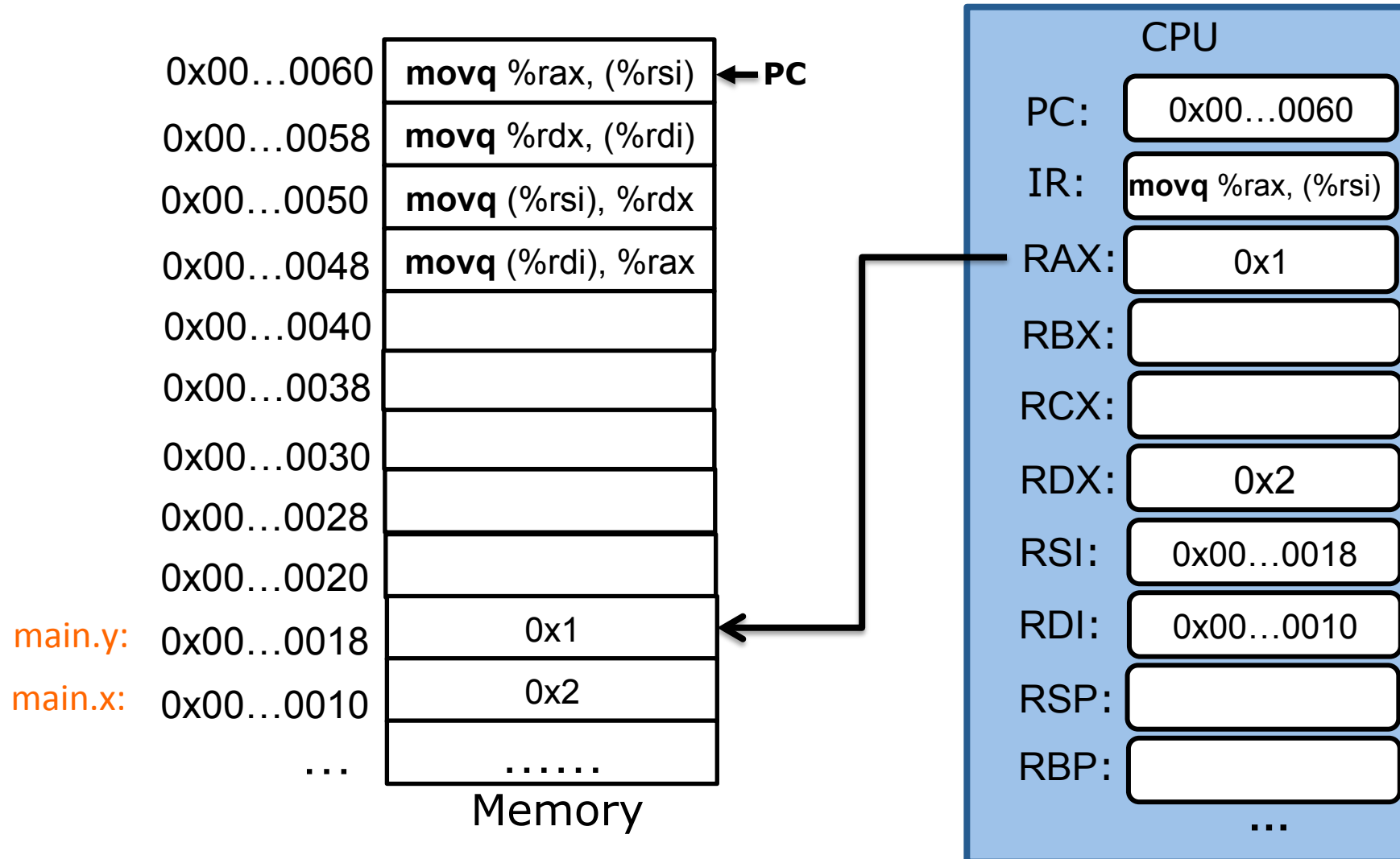
# swap func



# swap func



# swap func



# Limitation of direct addressing

**Issue:** the address must be calculated and stored in the register before each memory access.

# Issue

**Issue:** the address must be calculated and stored in the register before each memory access.

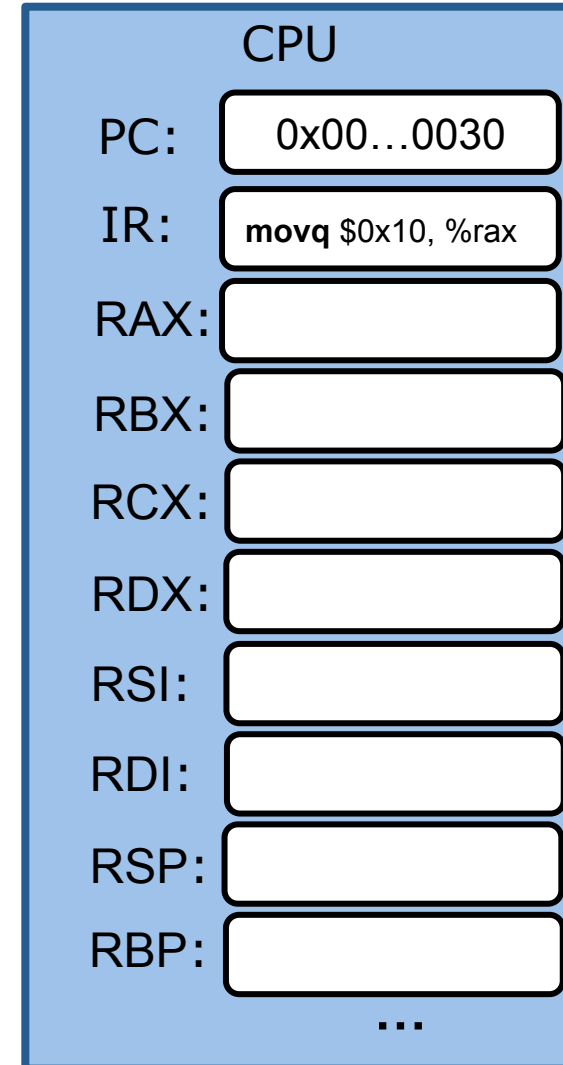
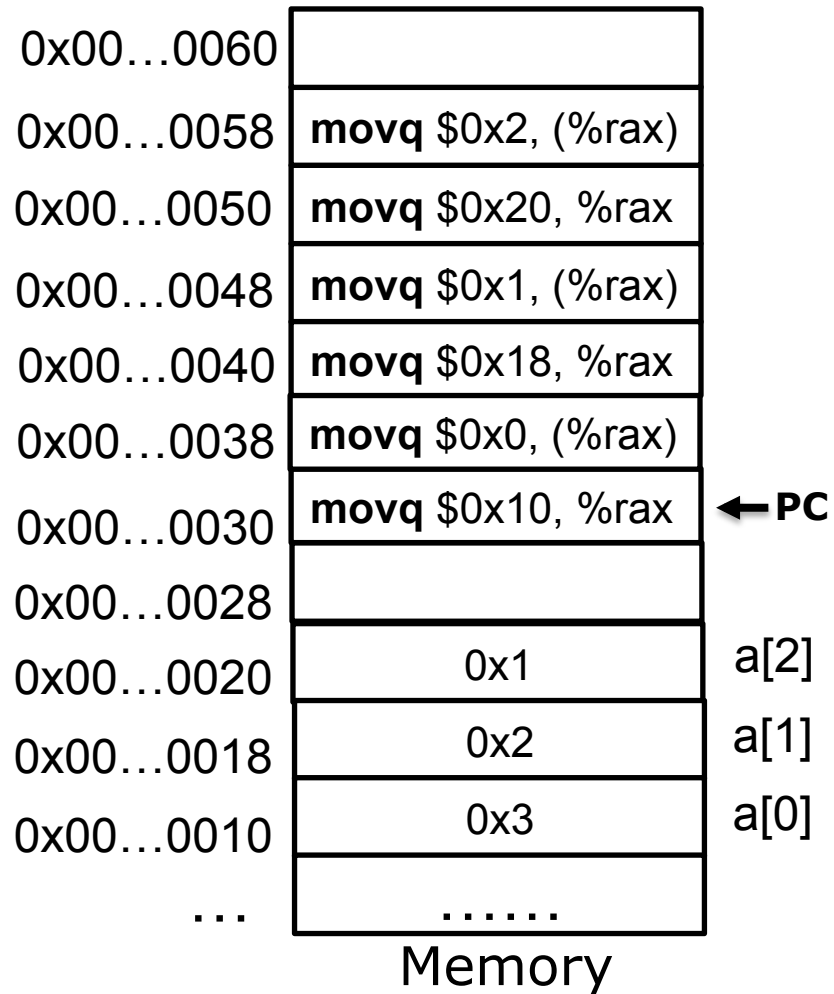
```
long a[] = {3, 2, 1};  
for(int i = 0; i < 3; i++) {  
    a[i] = i;  
}
```



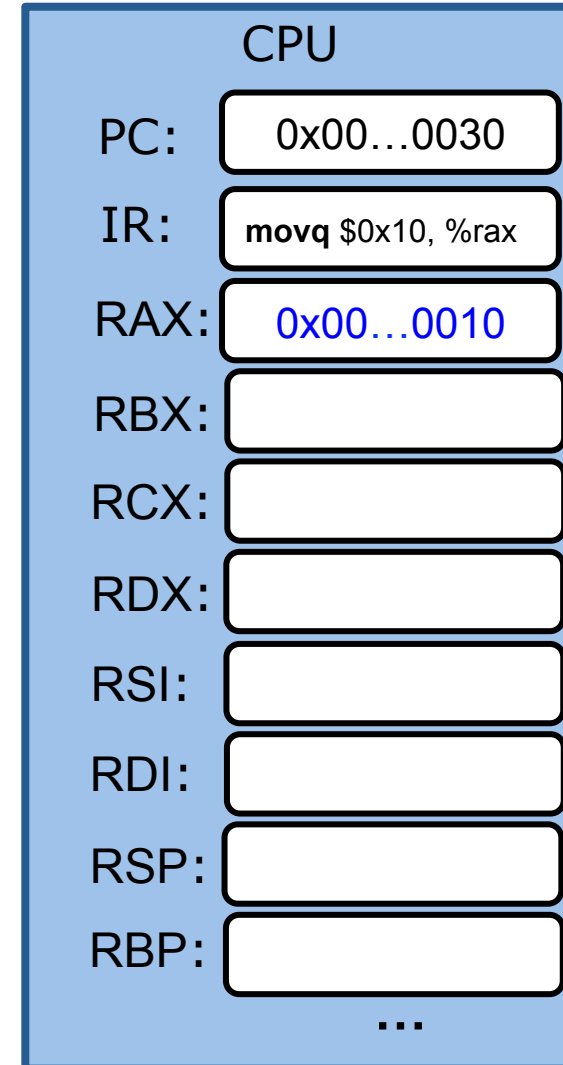
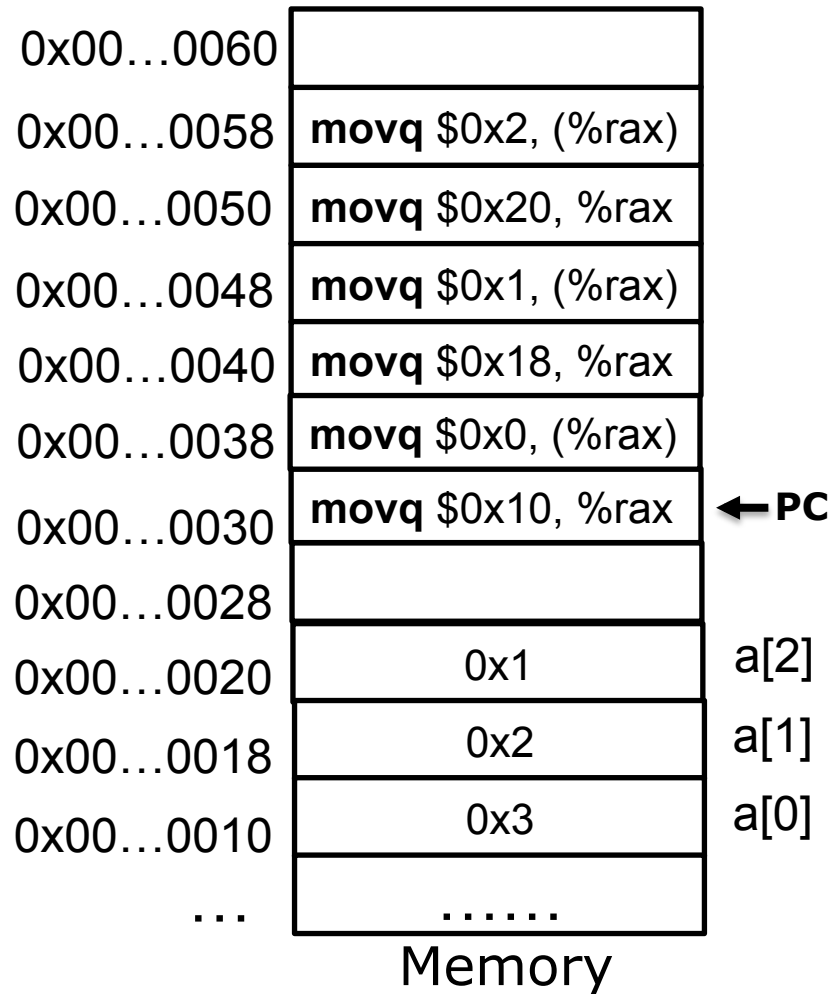
```
long a[] = {1, 2, 3};  
for(int i = 0; i < 3; i++) {  
    1. calculate &a[i] and put result in reg  
    2. mov $i, (reg)  
}
```



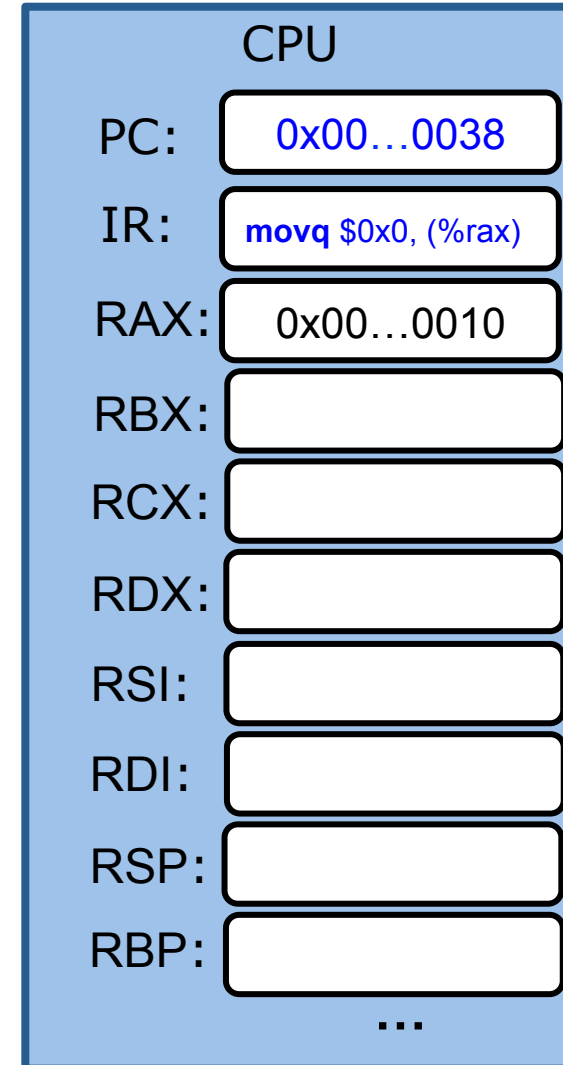
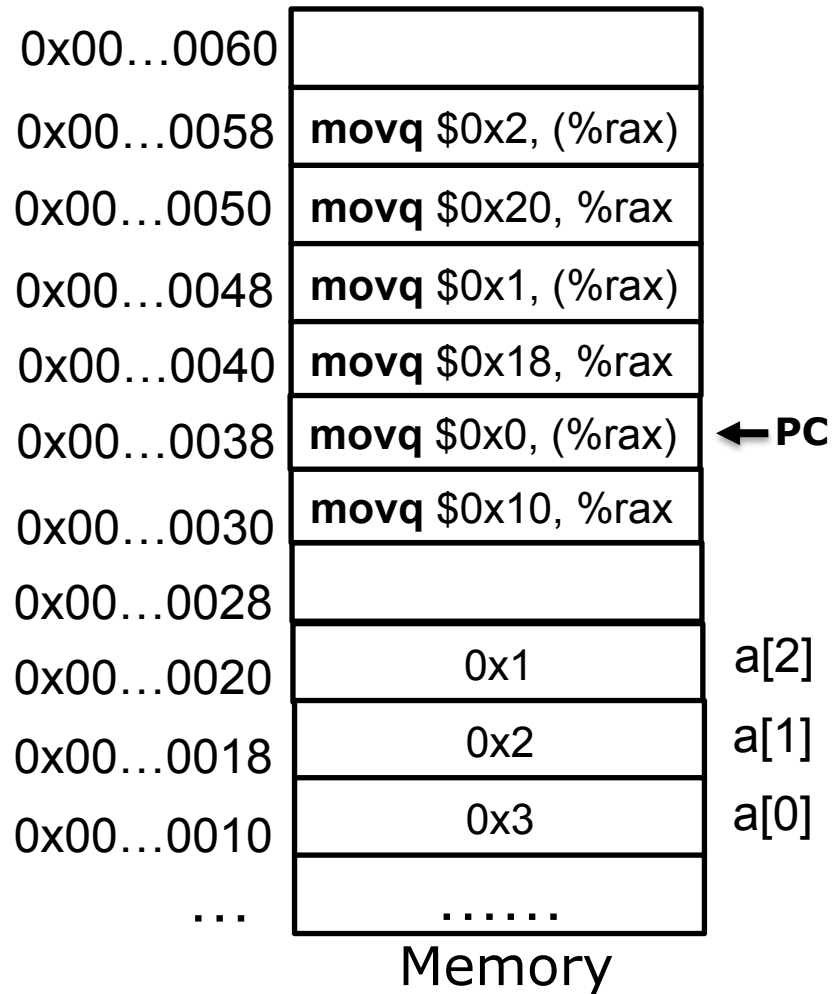
# Example



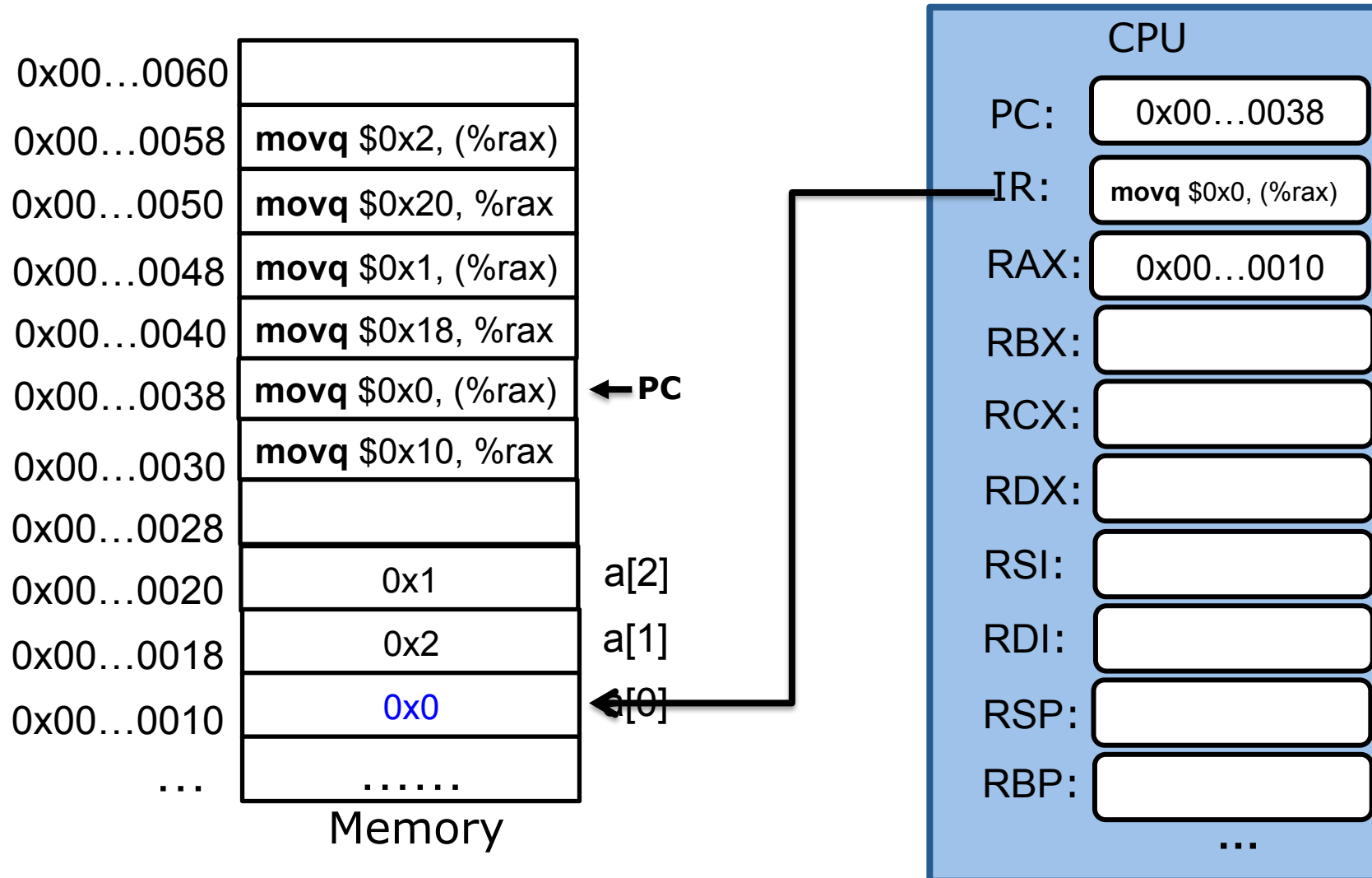
# Example



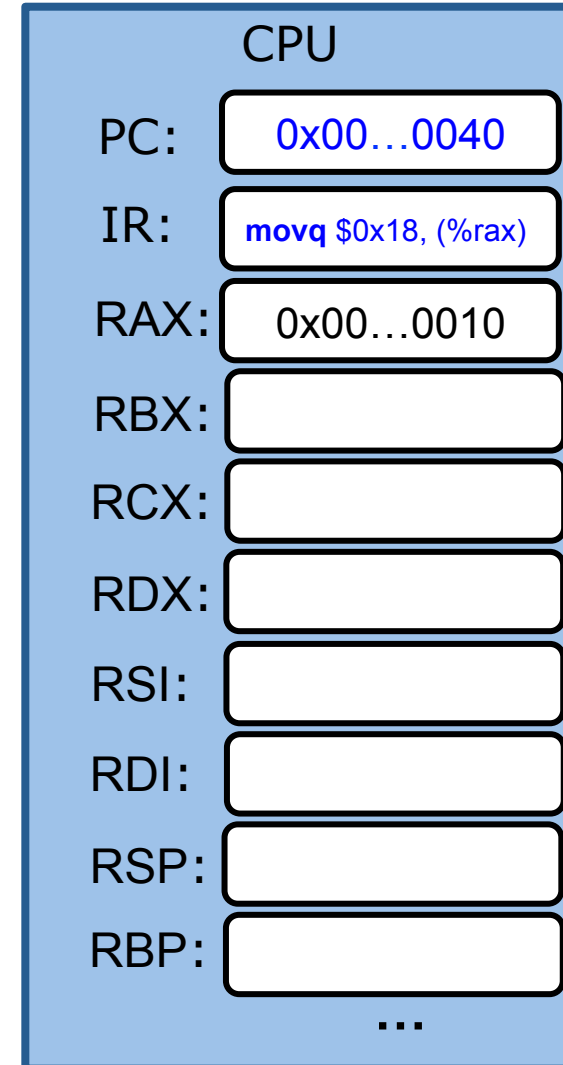
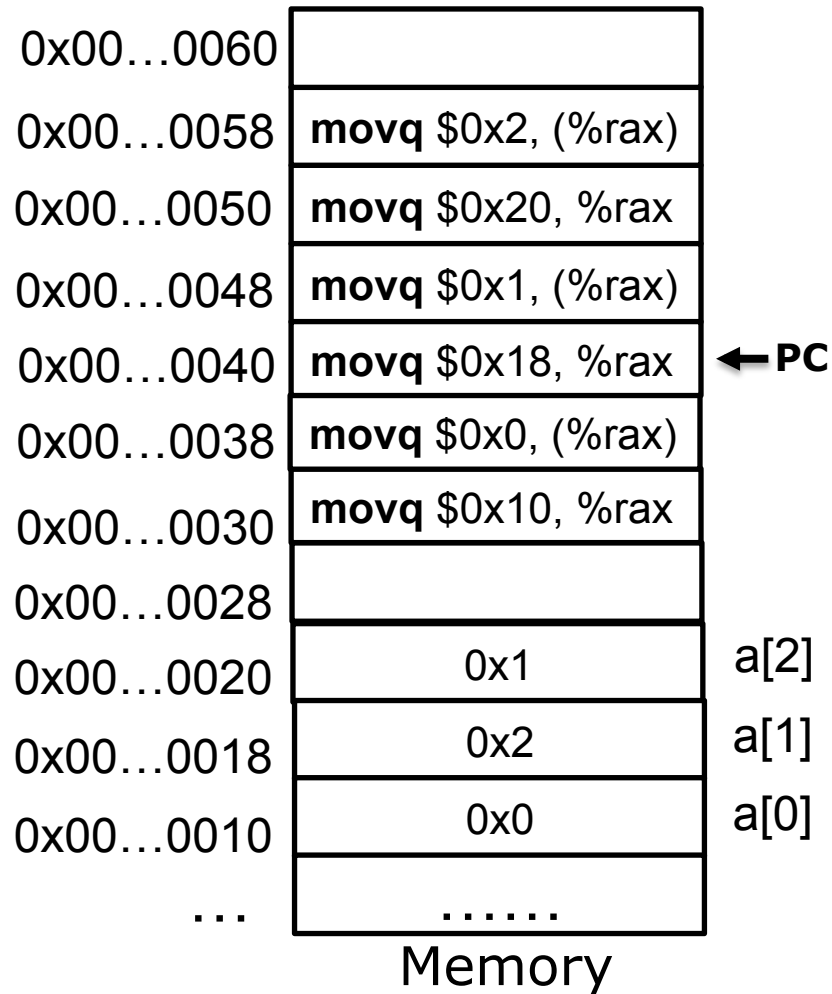
# Example



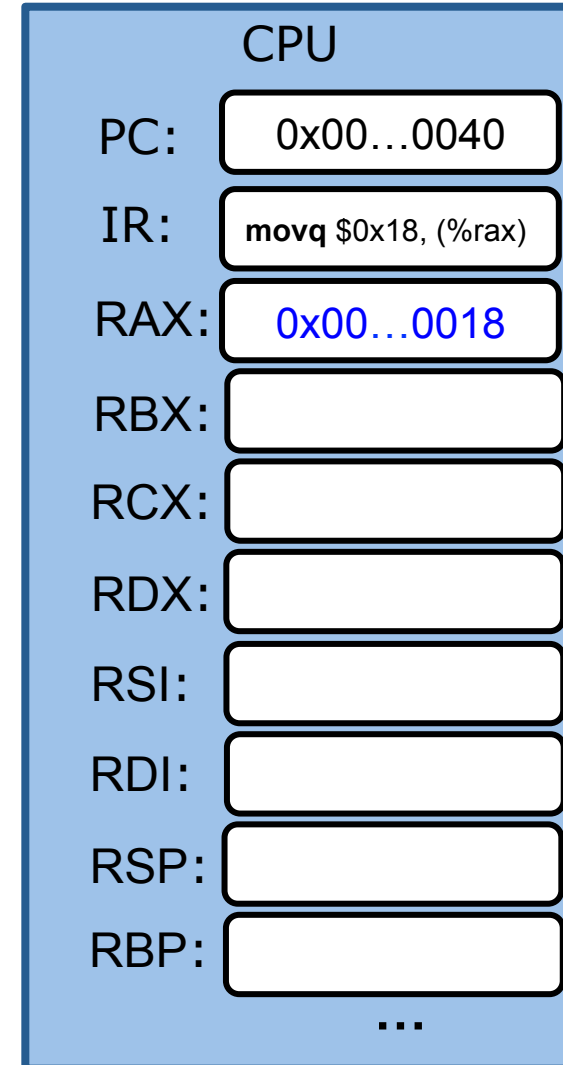
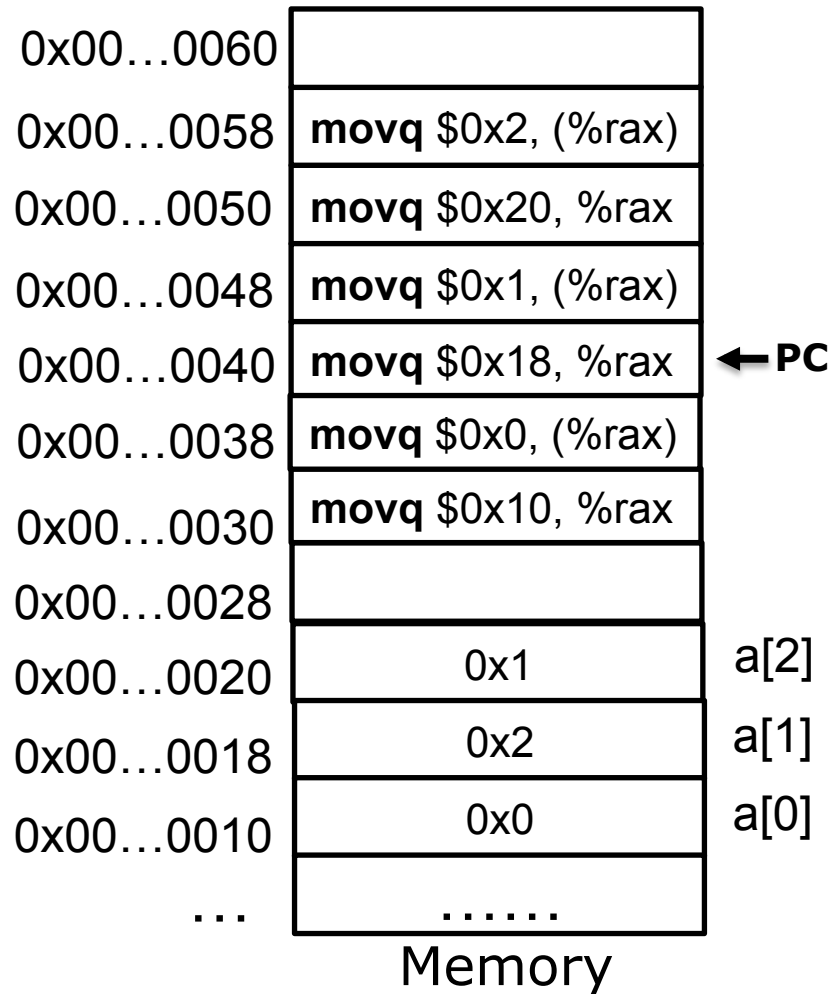
# Example



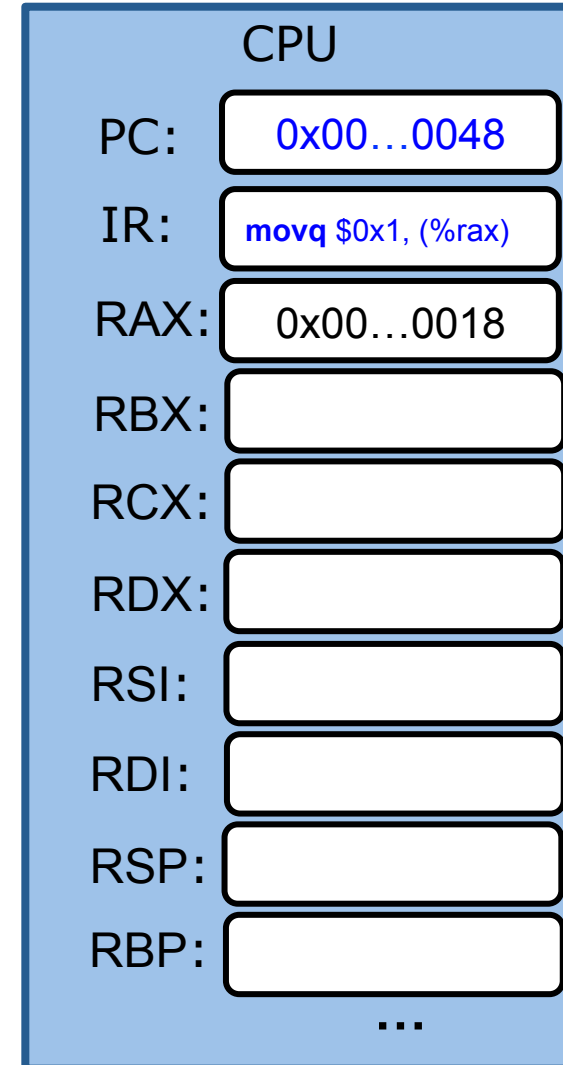
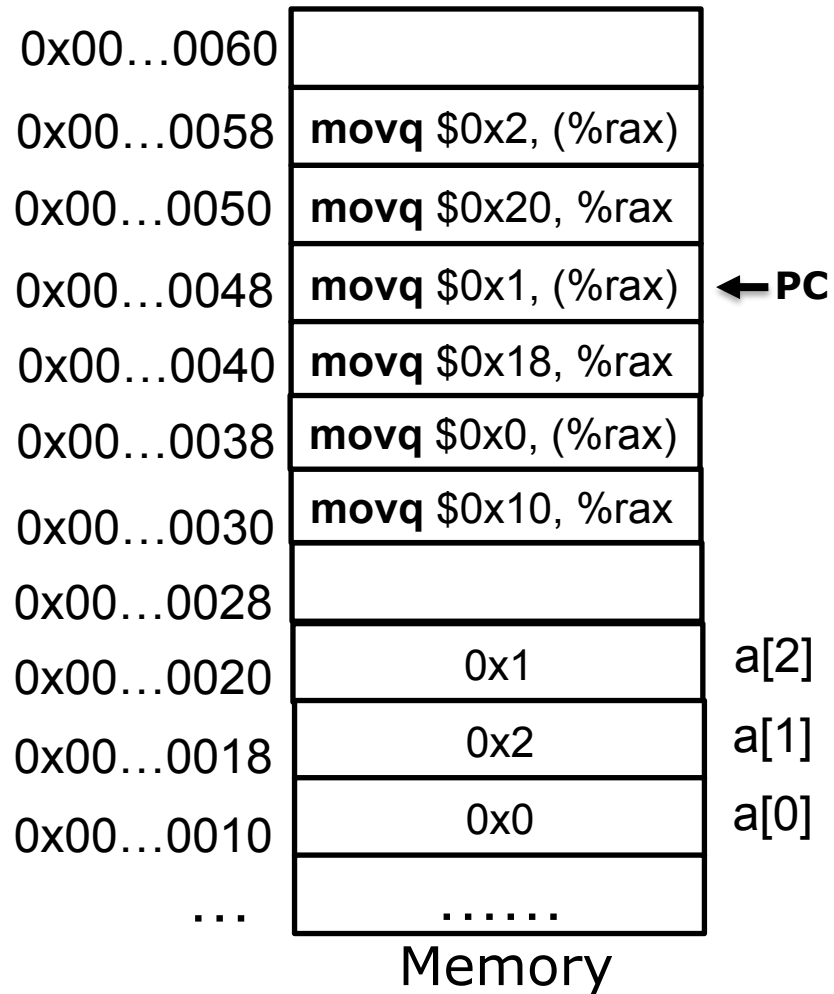
# Example



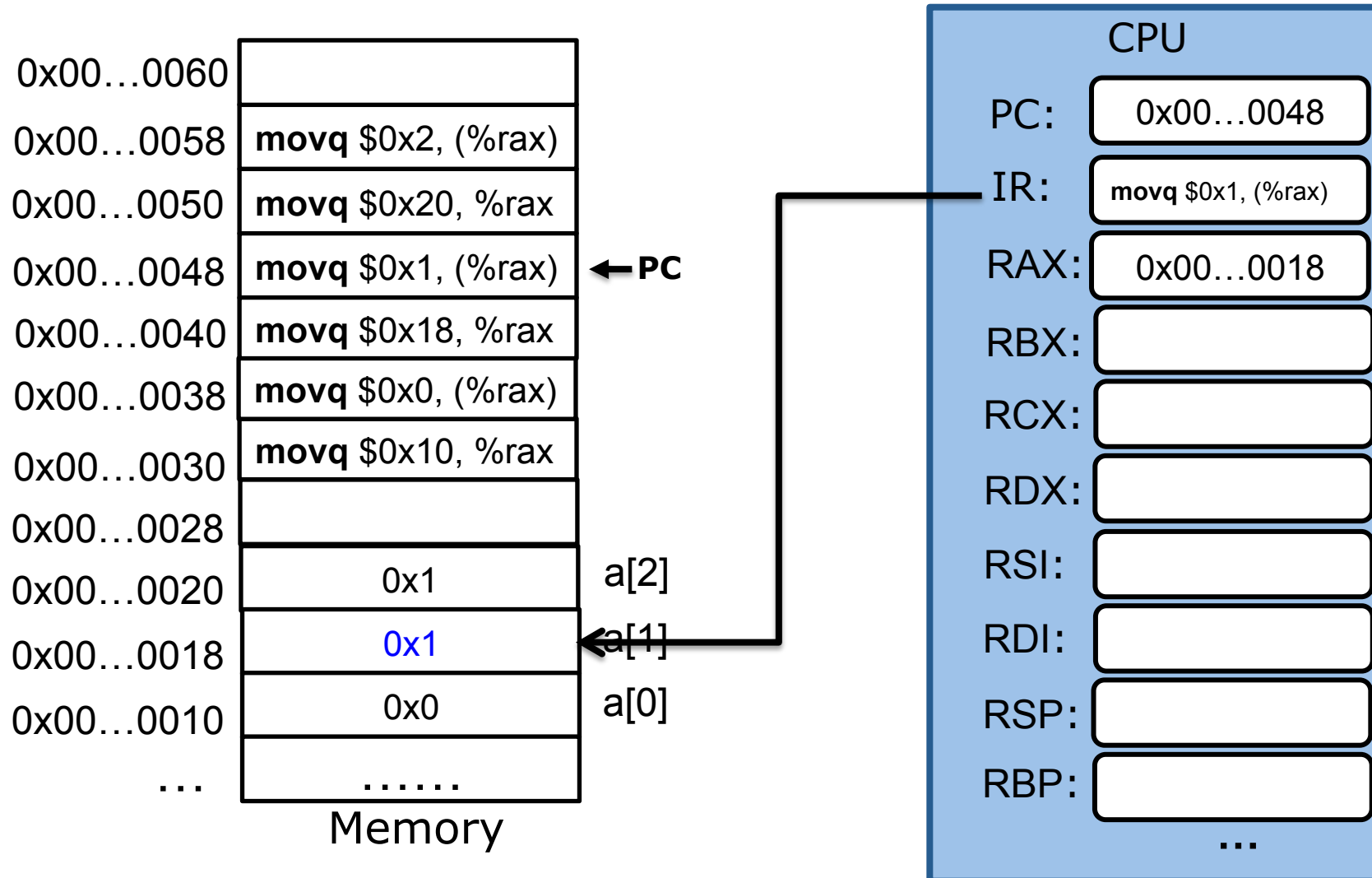
# Example



# Example

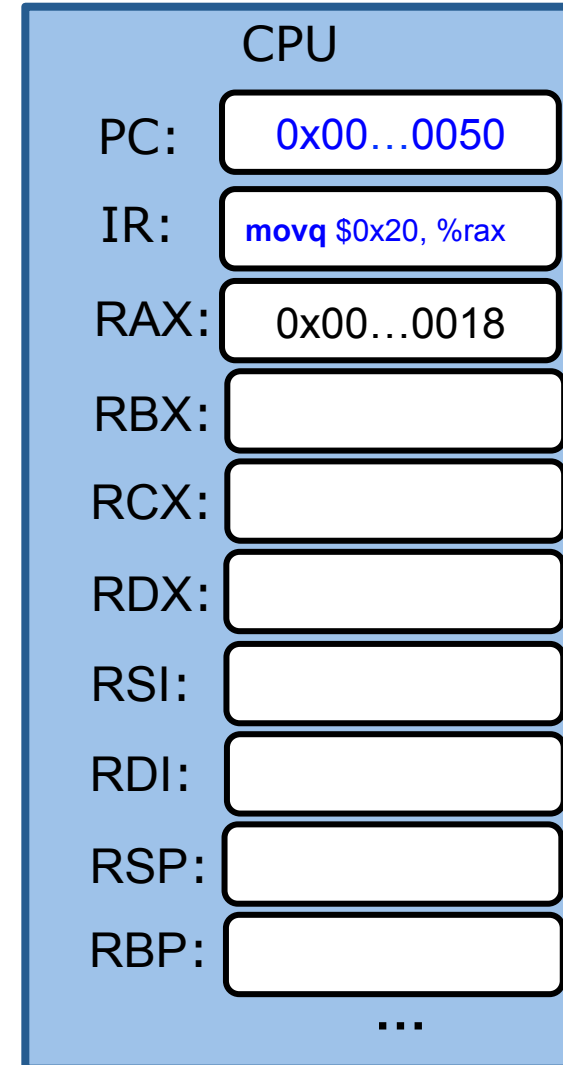
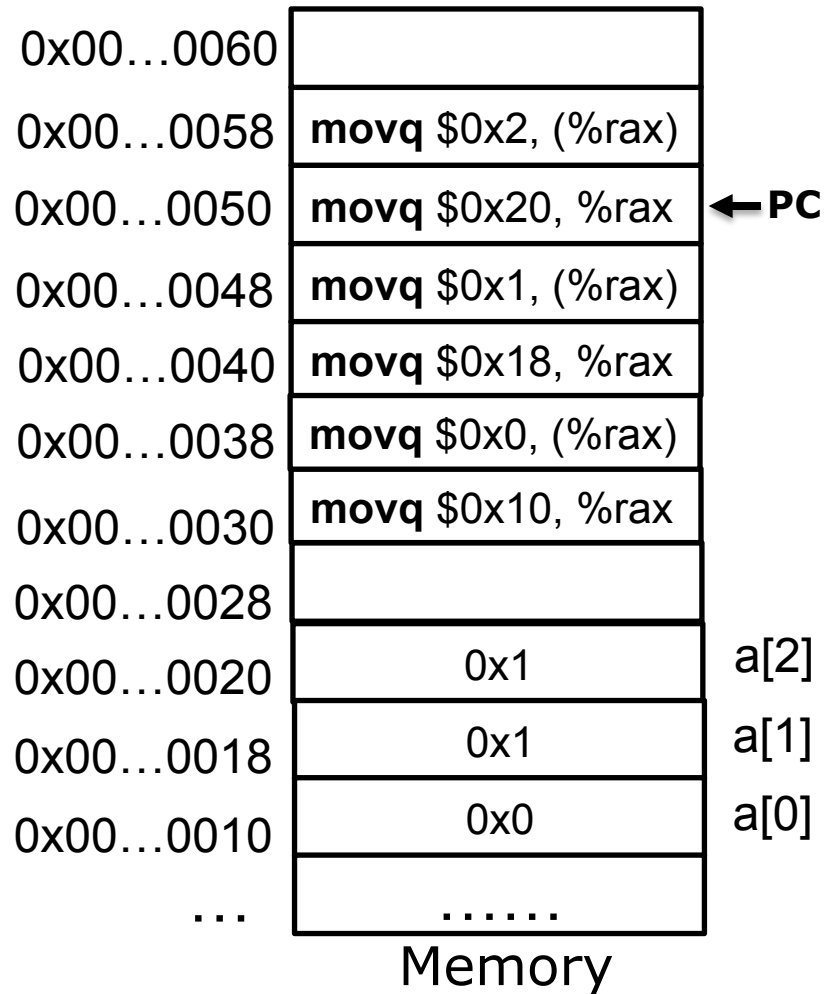


# Example

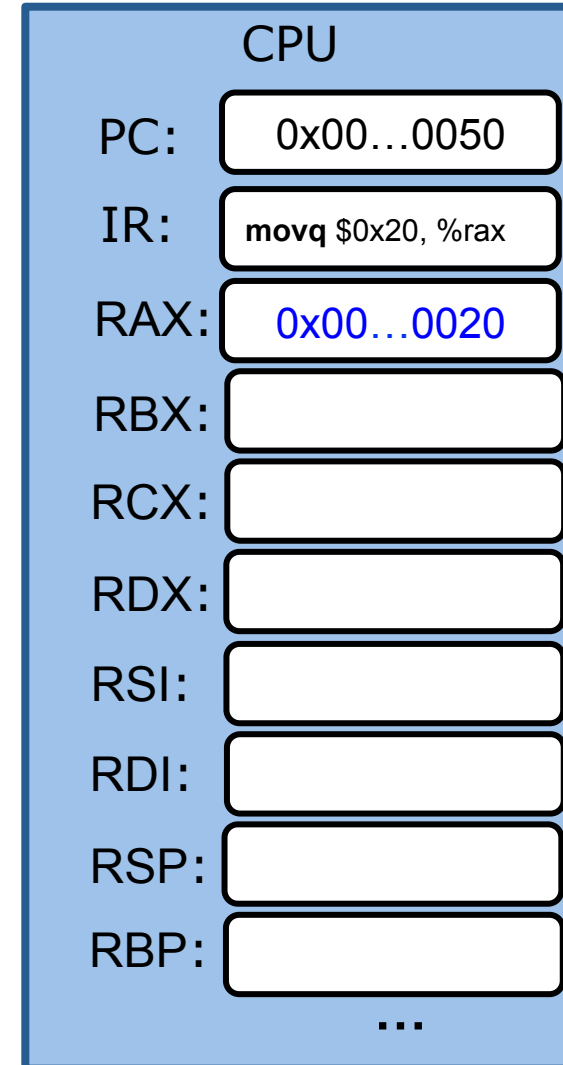
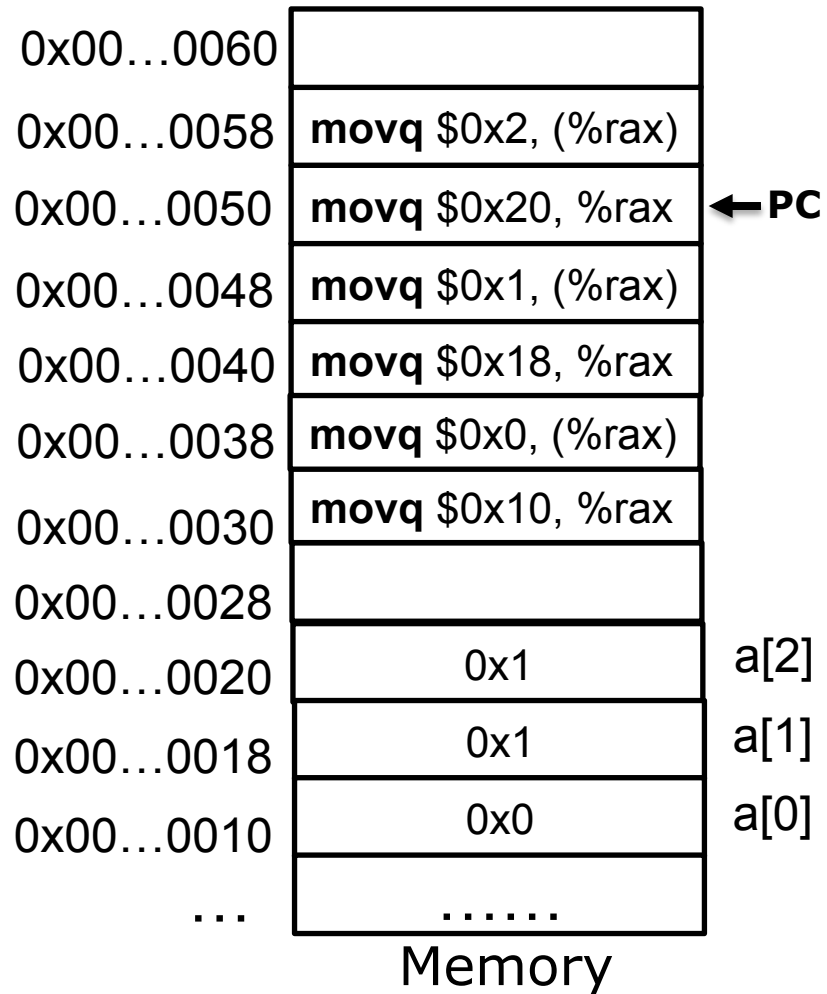




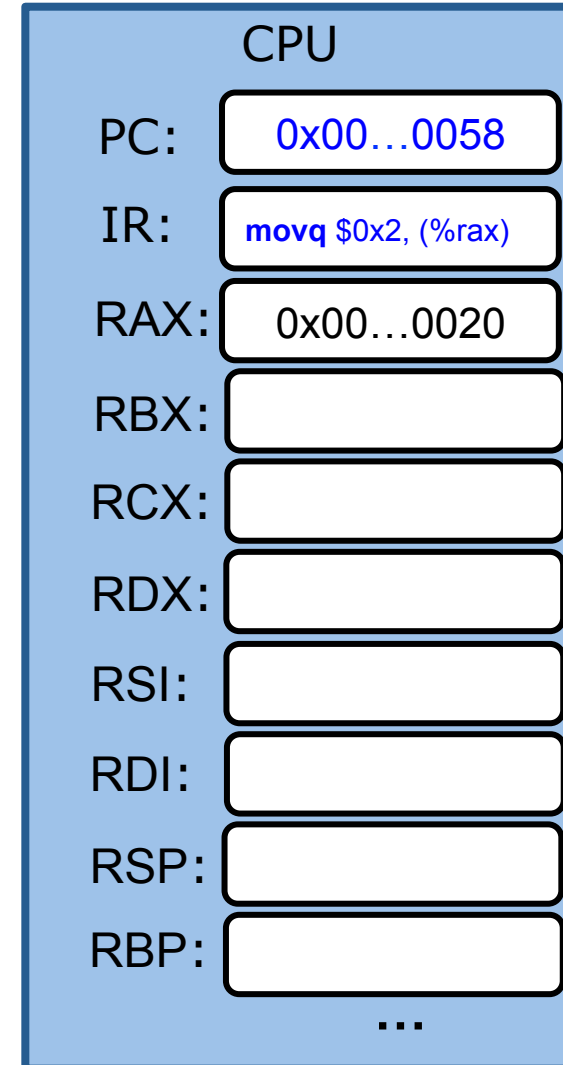
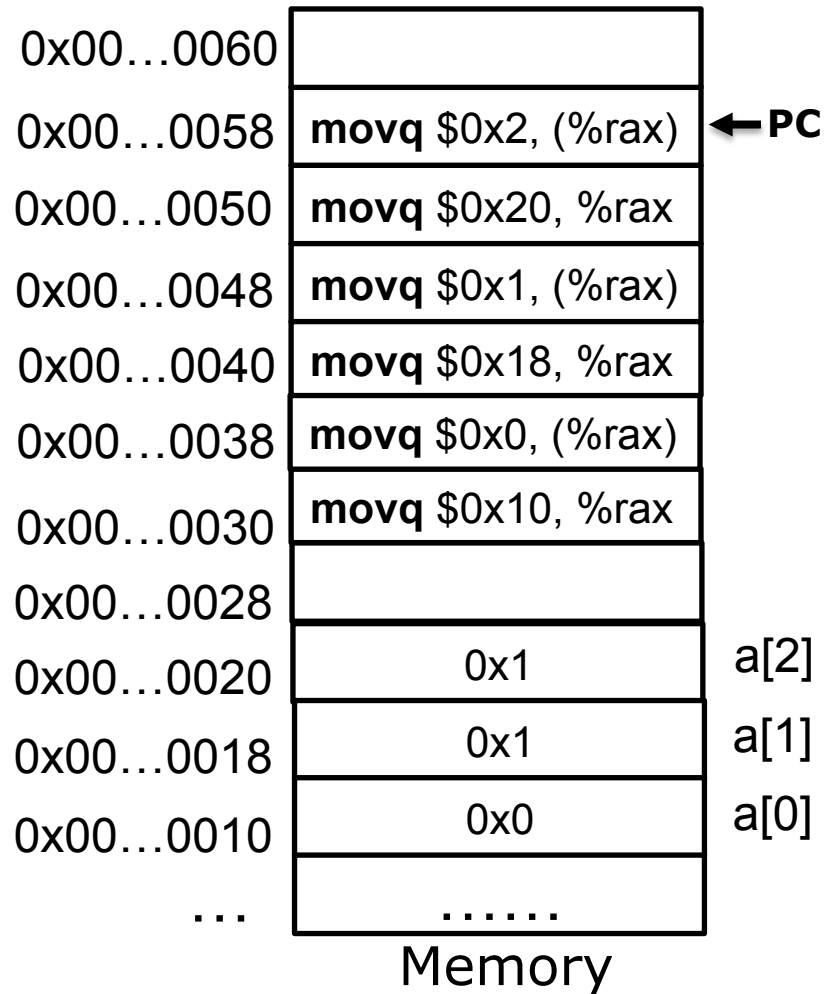
# Example



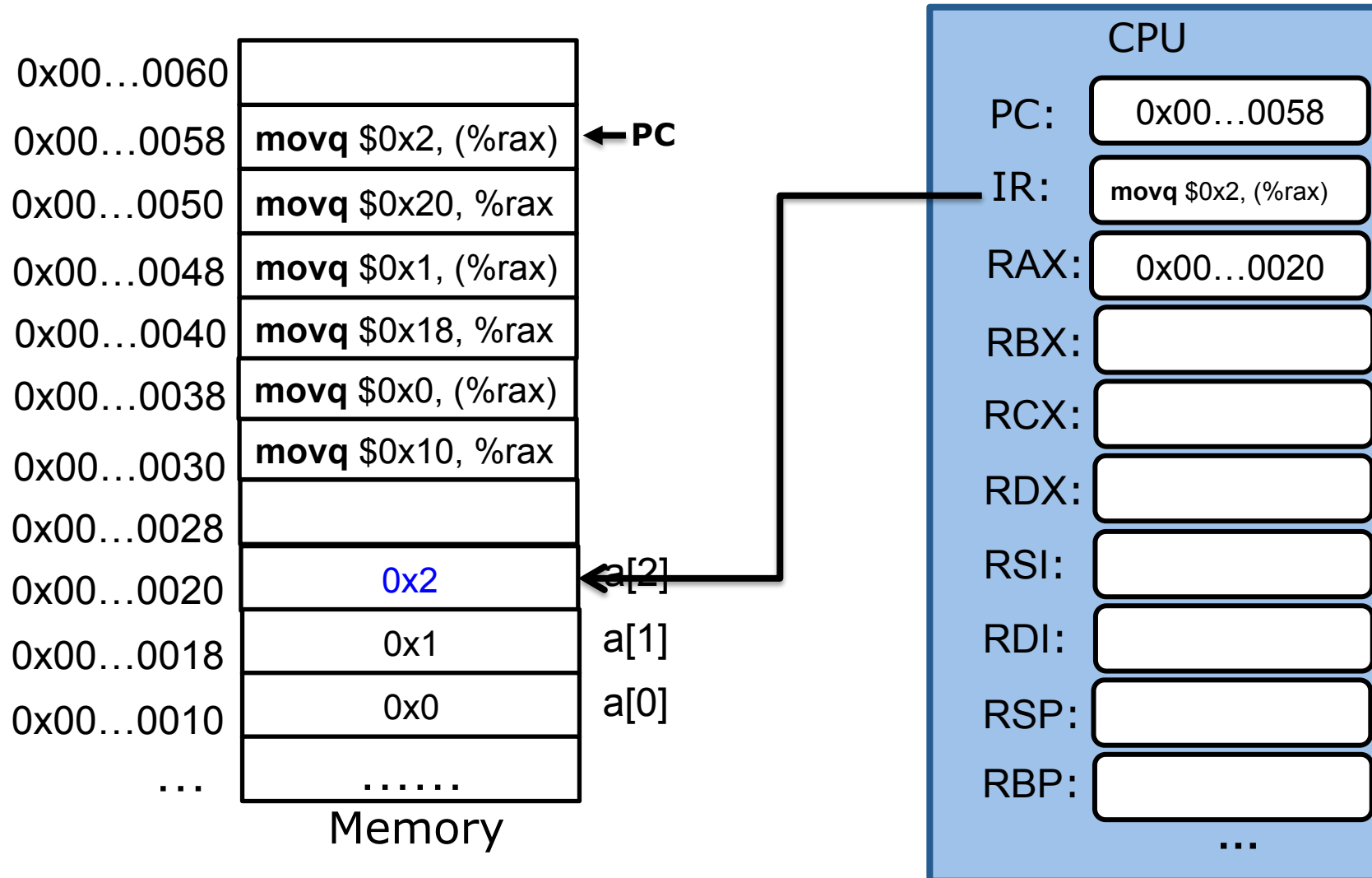
# Example



# Example



# Example



# Observation

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

a[0], a[1] and a[2] have the same base address:&a[0]

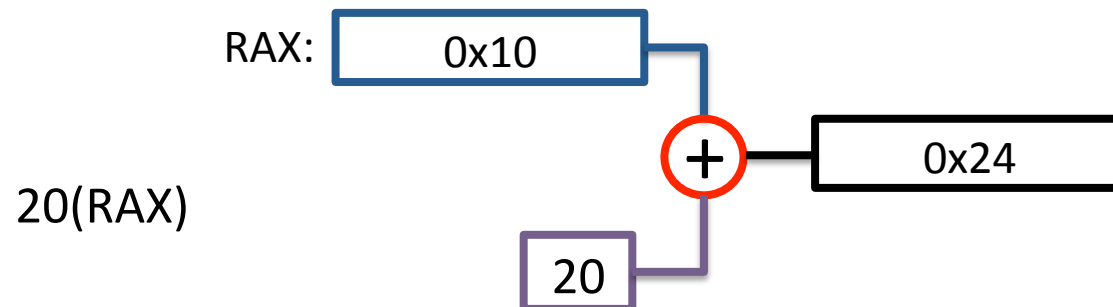
- &a[0]: &a[0] + 0
- &a[1]: &a[0] + 1
- &a[2]: &a[0] + 2

# 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

RAX: 0x10



# Address mode with displacement

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

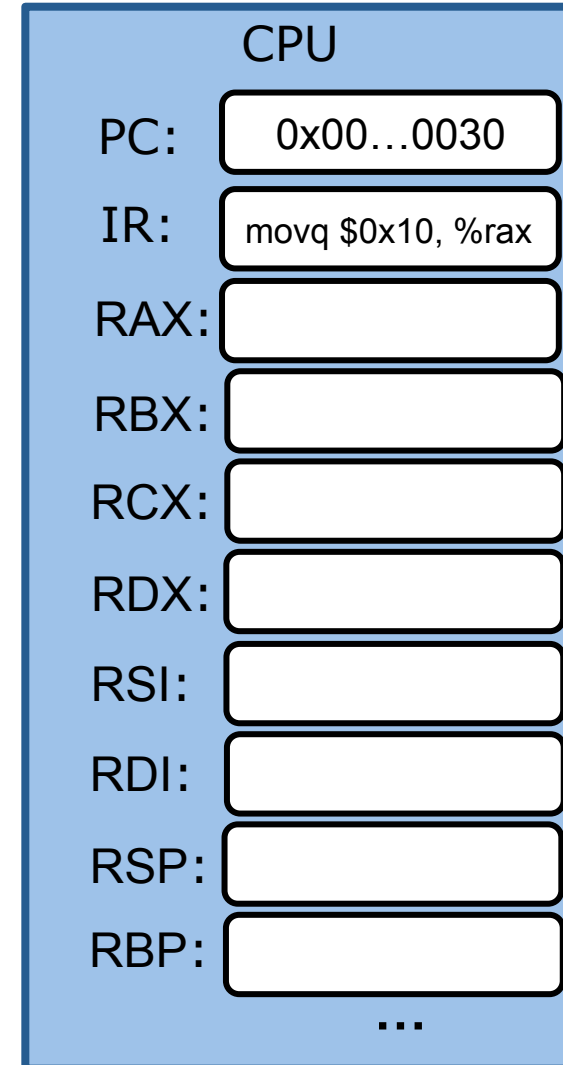
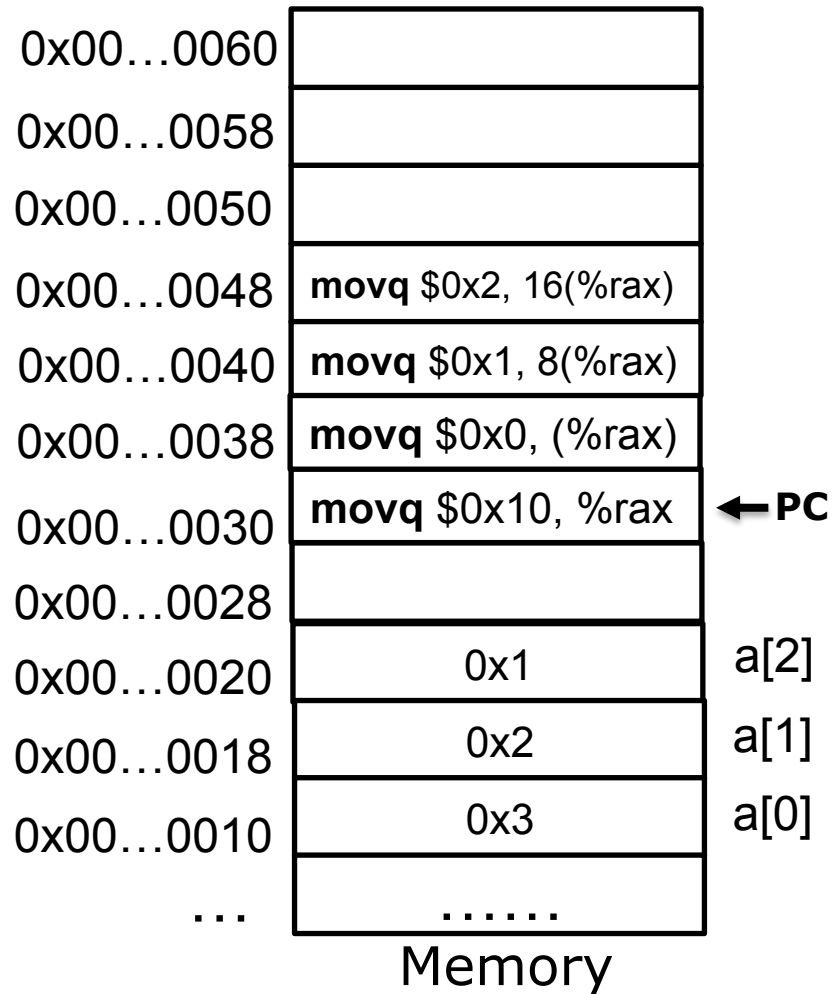
- Register specifies the start of the memory region
- Constant D specifies the offset

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



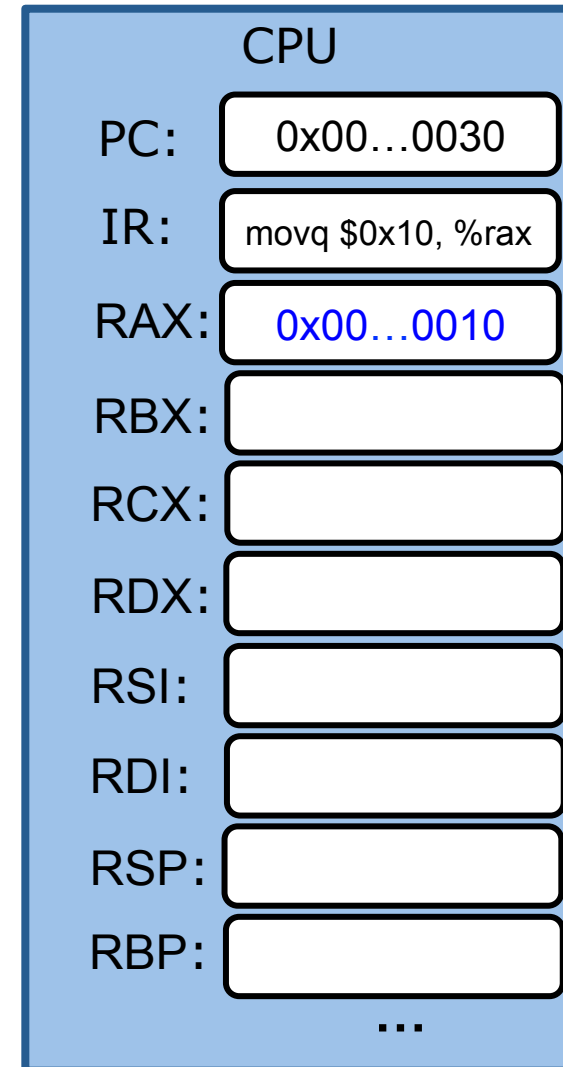
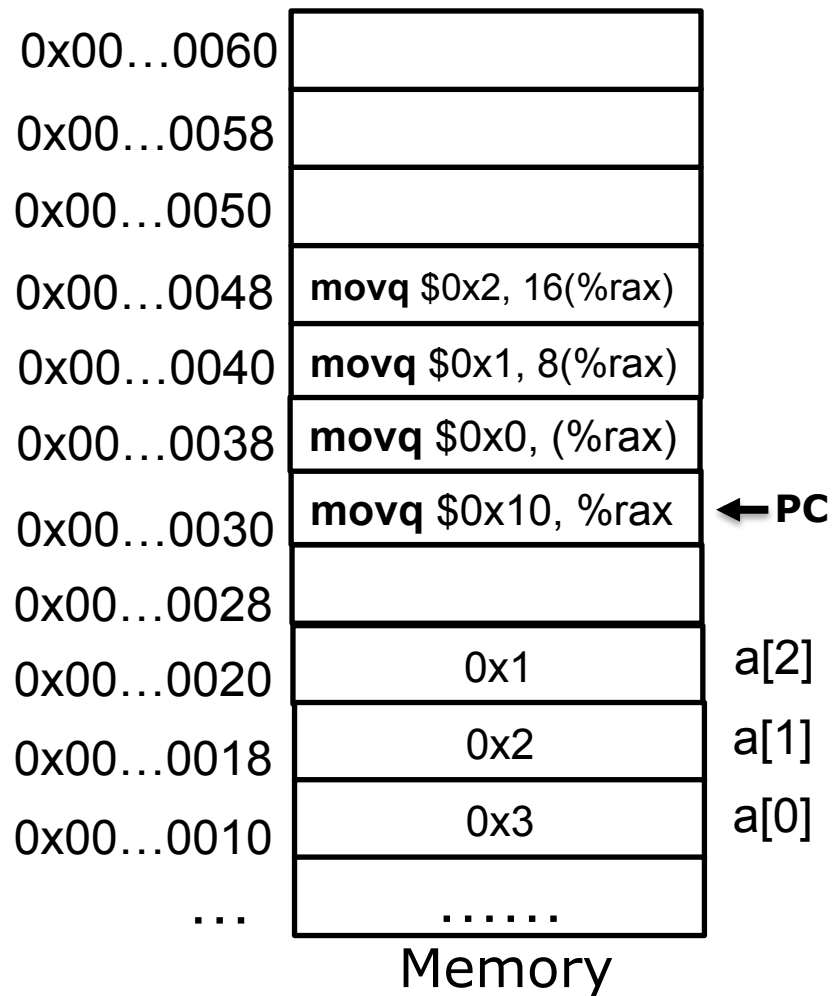
```
long a[] = {1, 2, 3};  
for(int i = 0; i < 3; i++) {  
    mov $i, D(reg); // D = i * 8, reg = &a[0]  
}
```

# Example

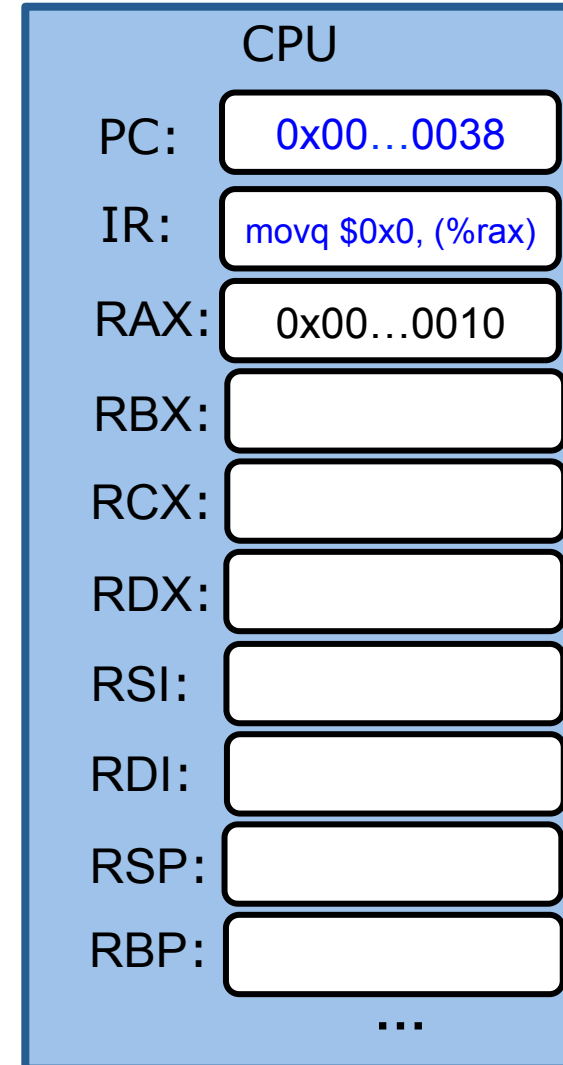
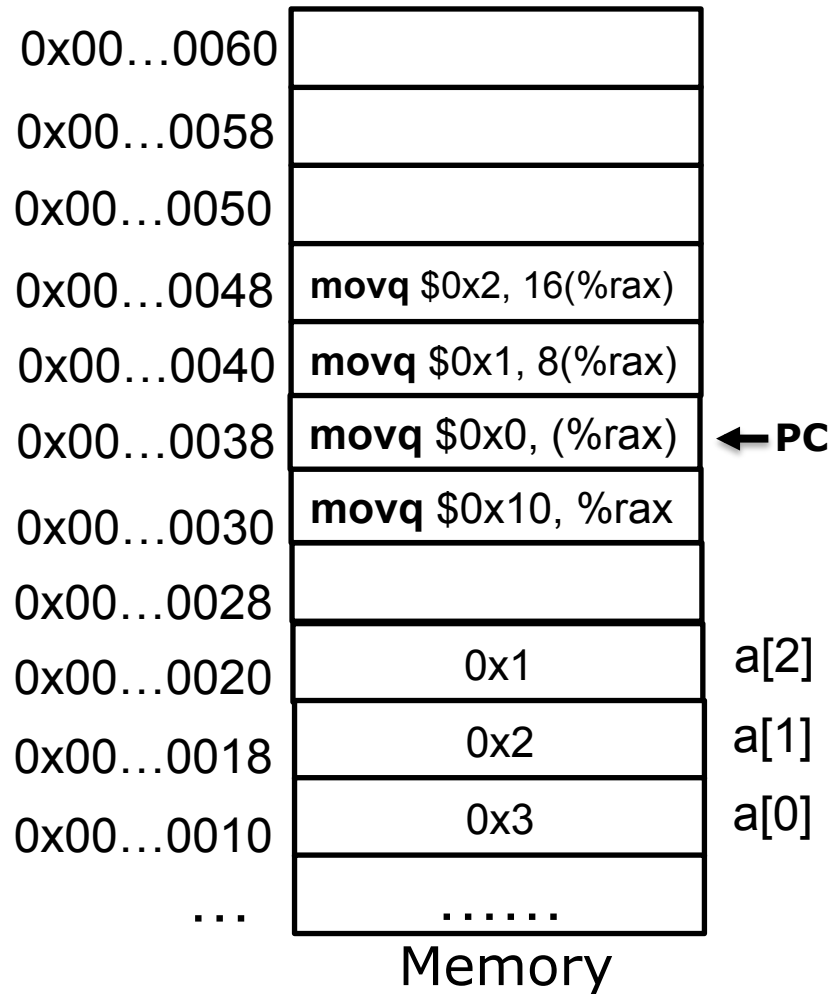




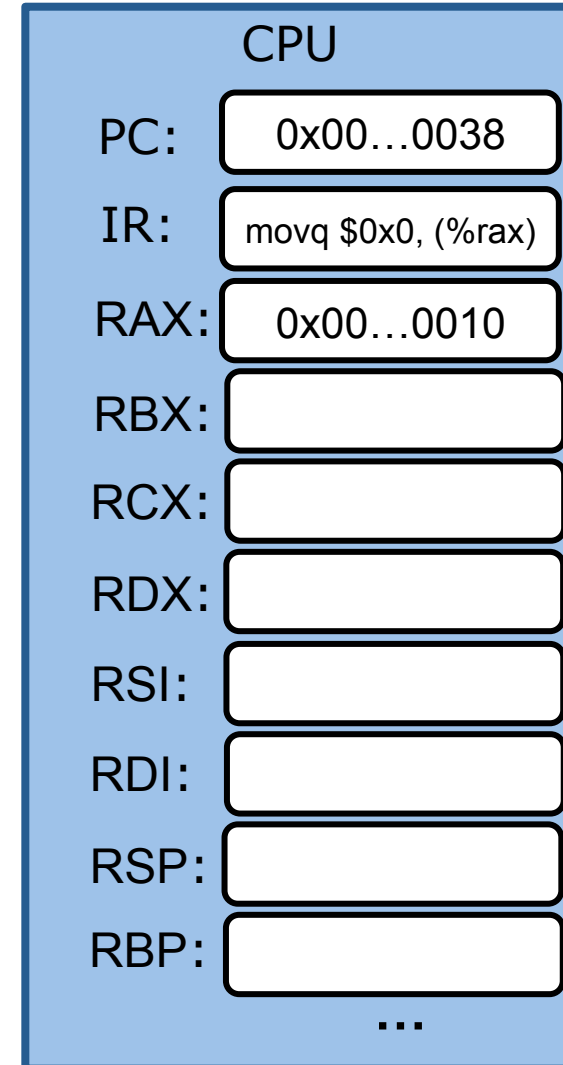
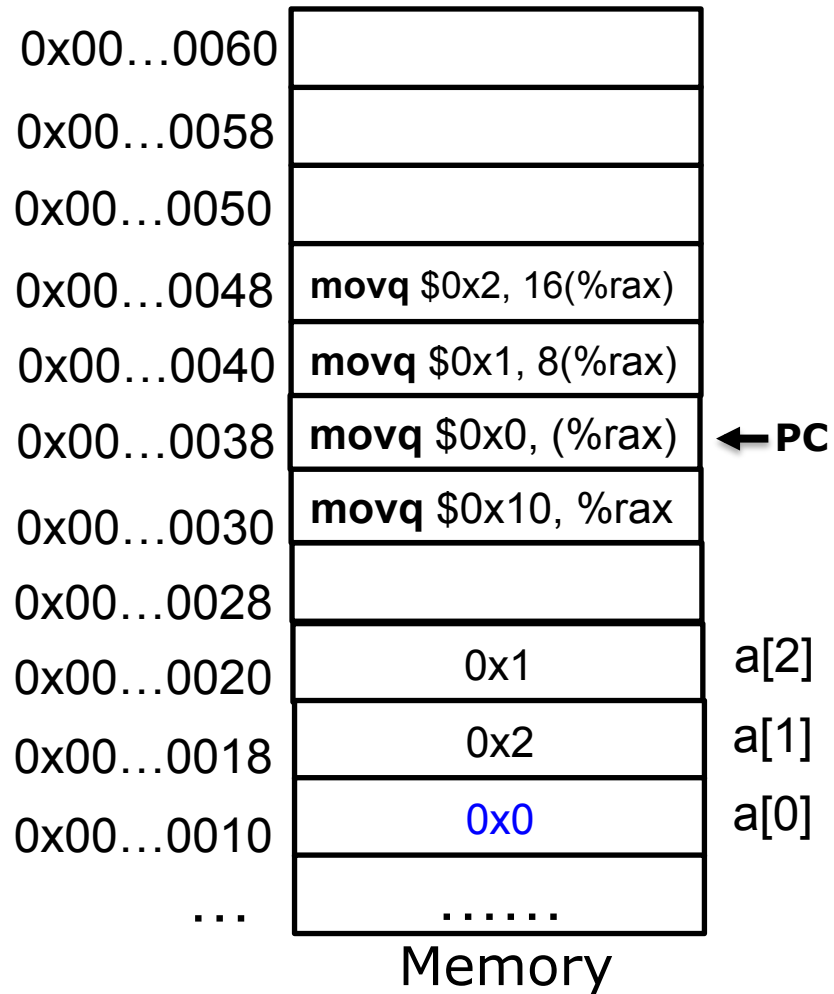
# Example



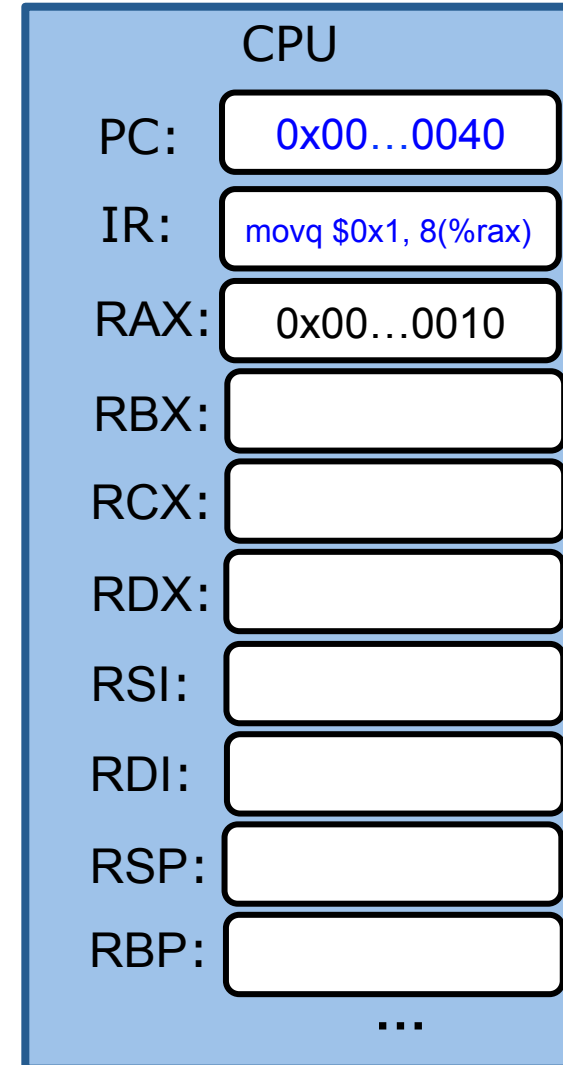
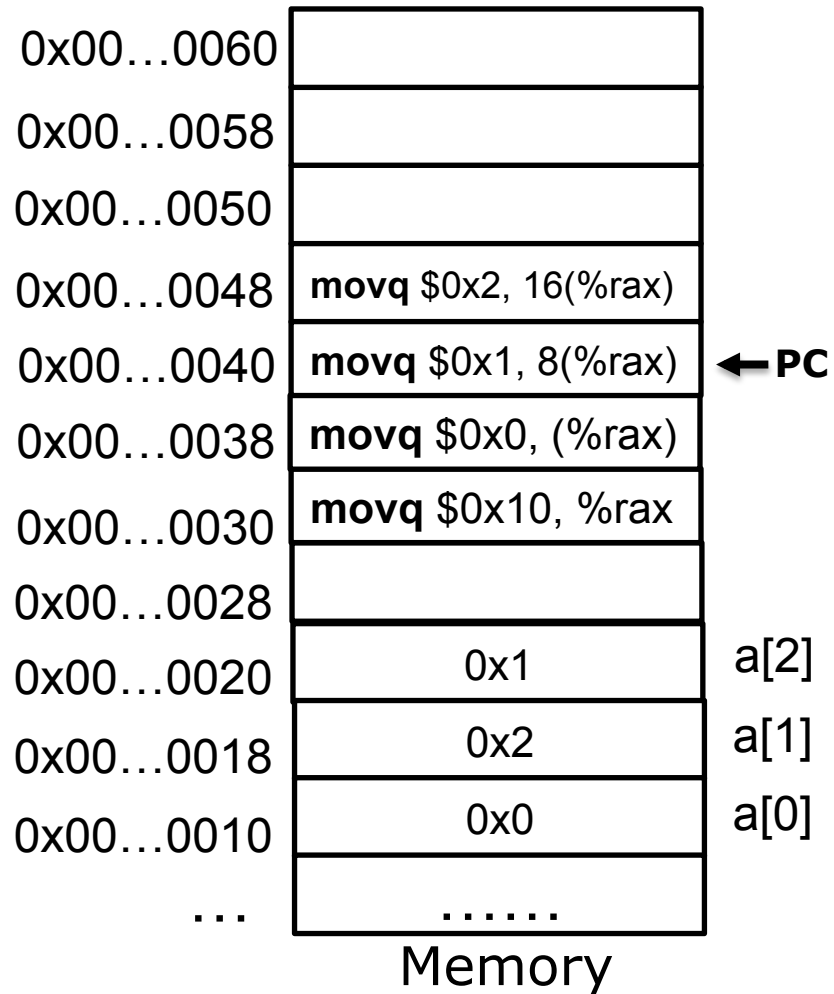
# Example



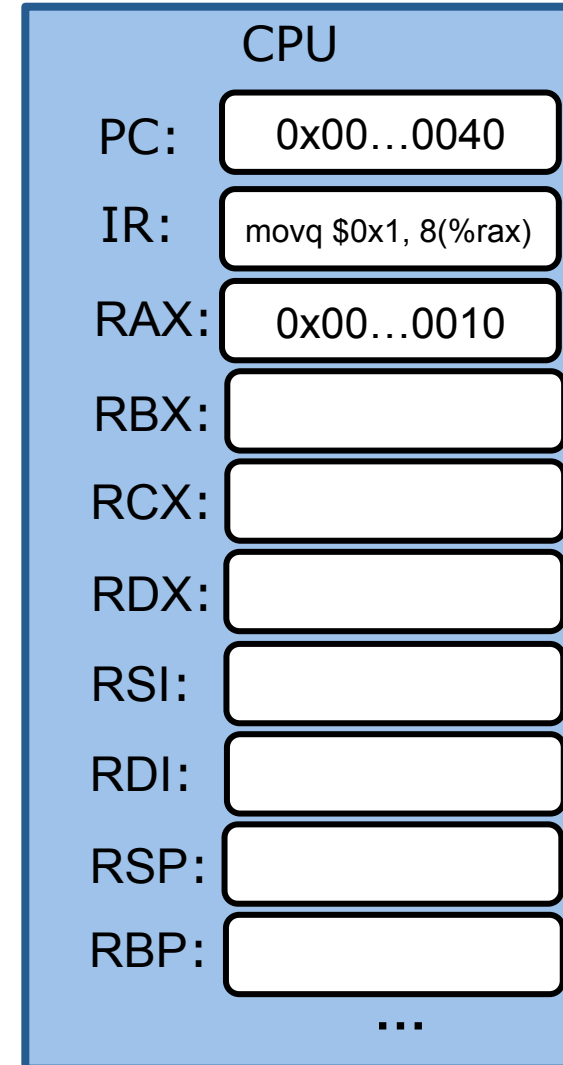
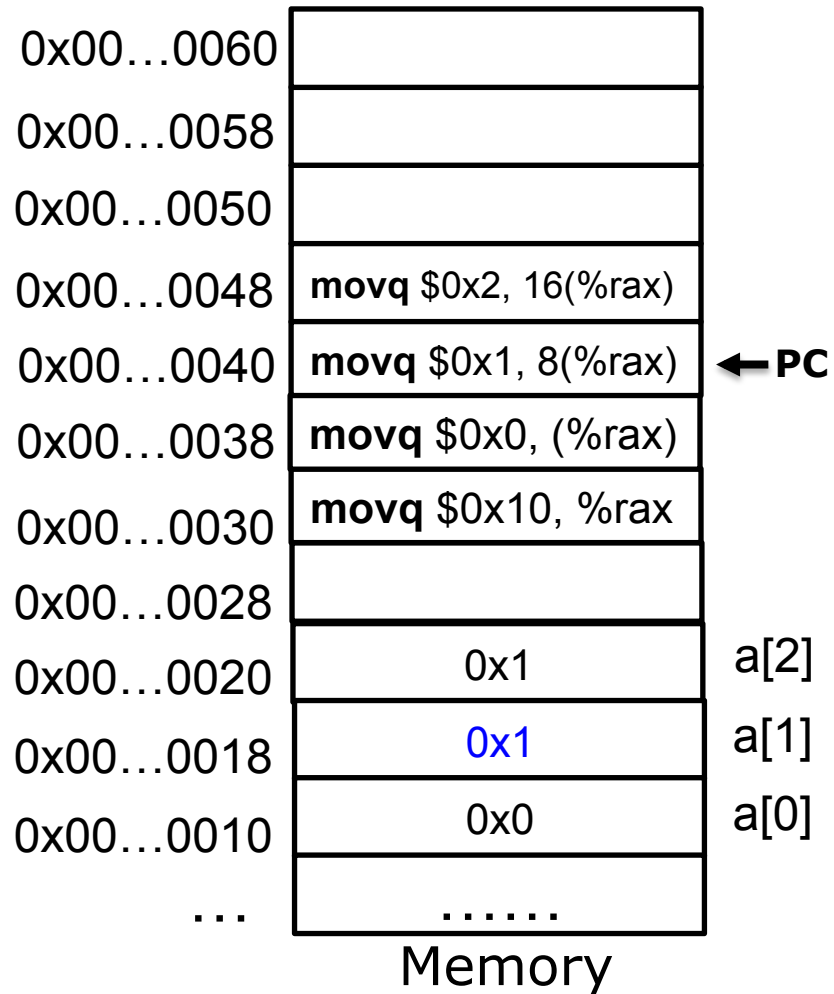
# Example



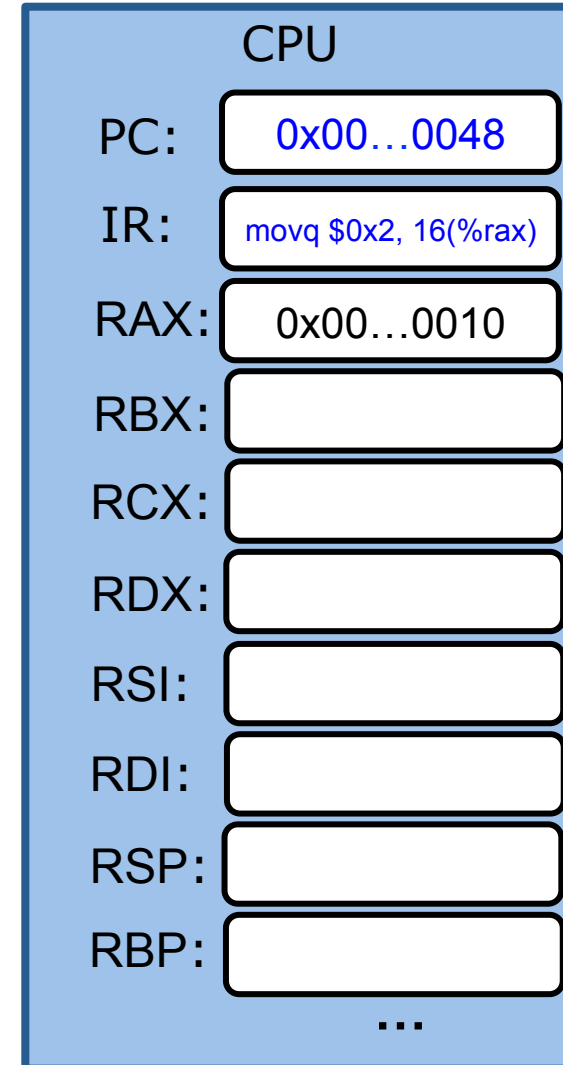
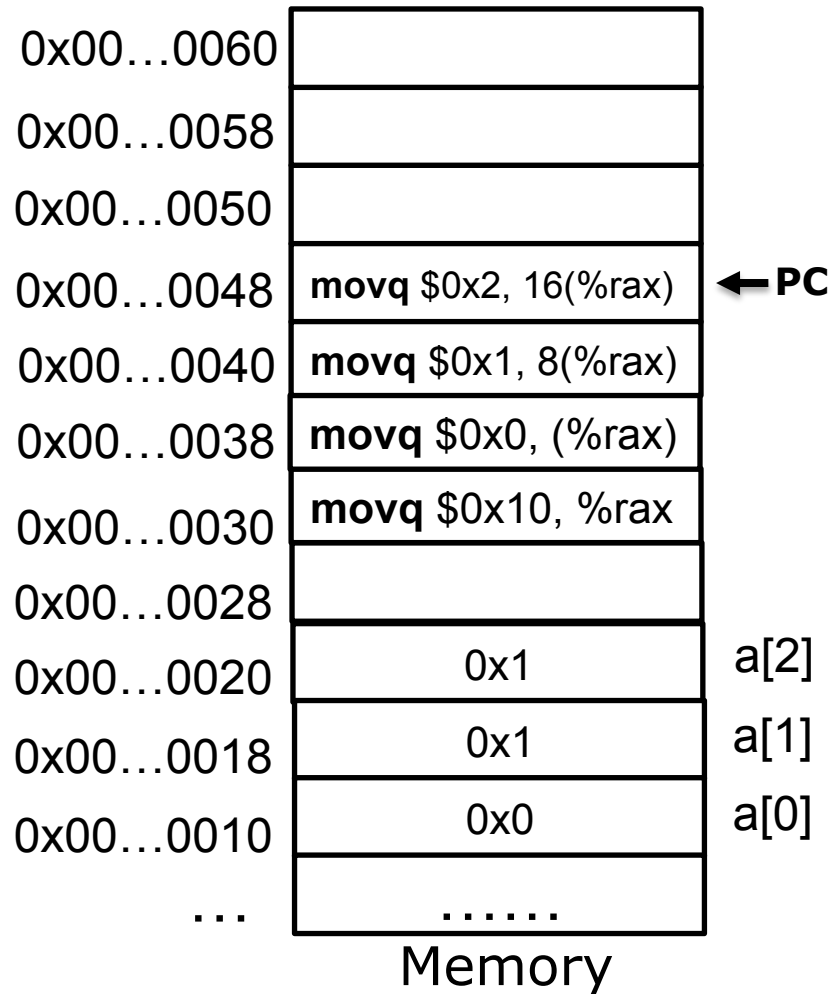
# Example



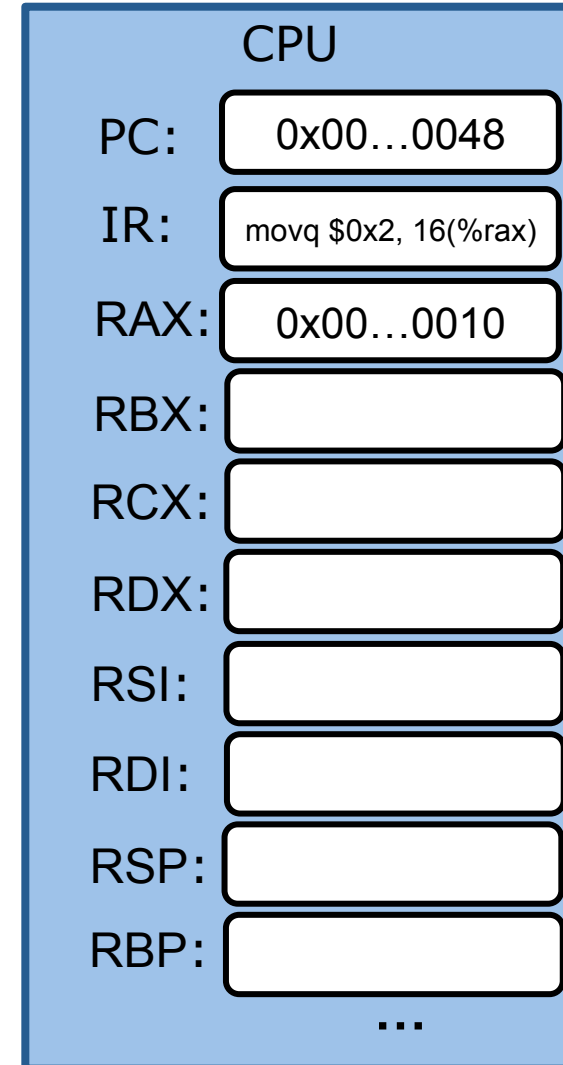
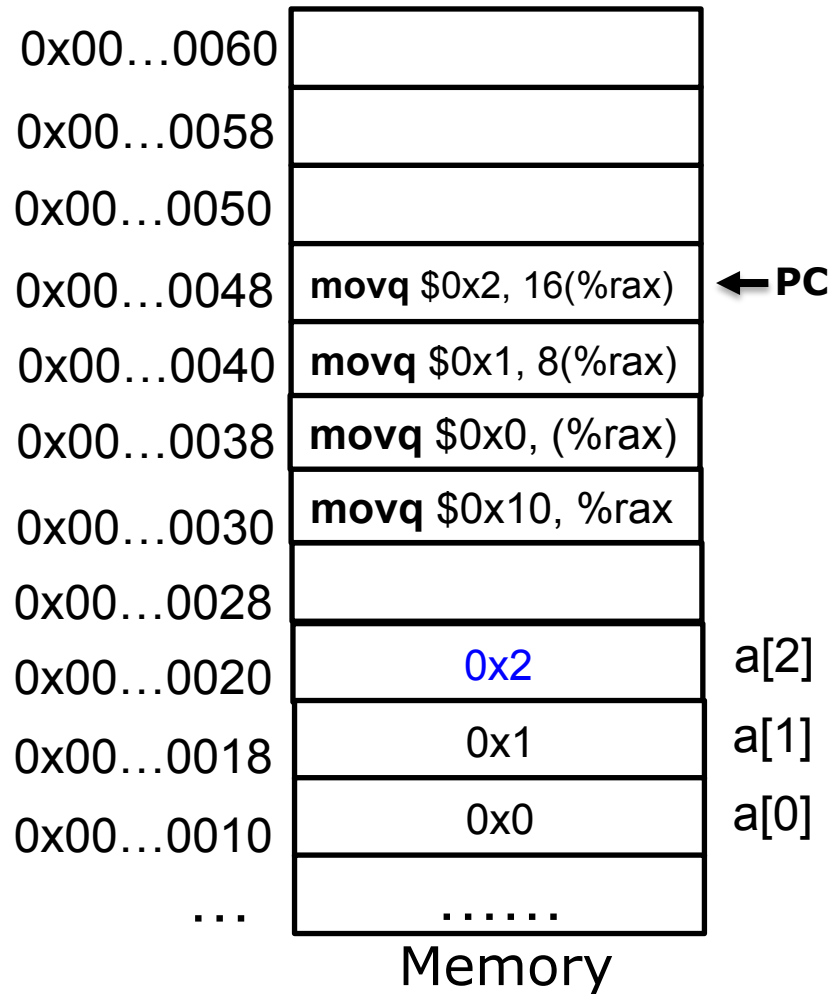
# Example



# Example



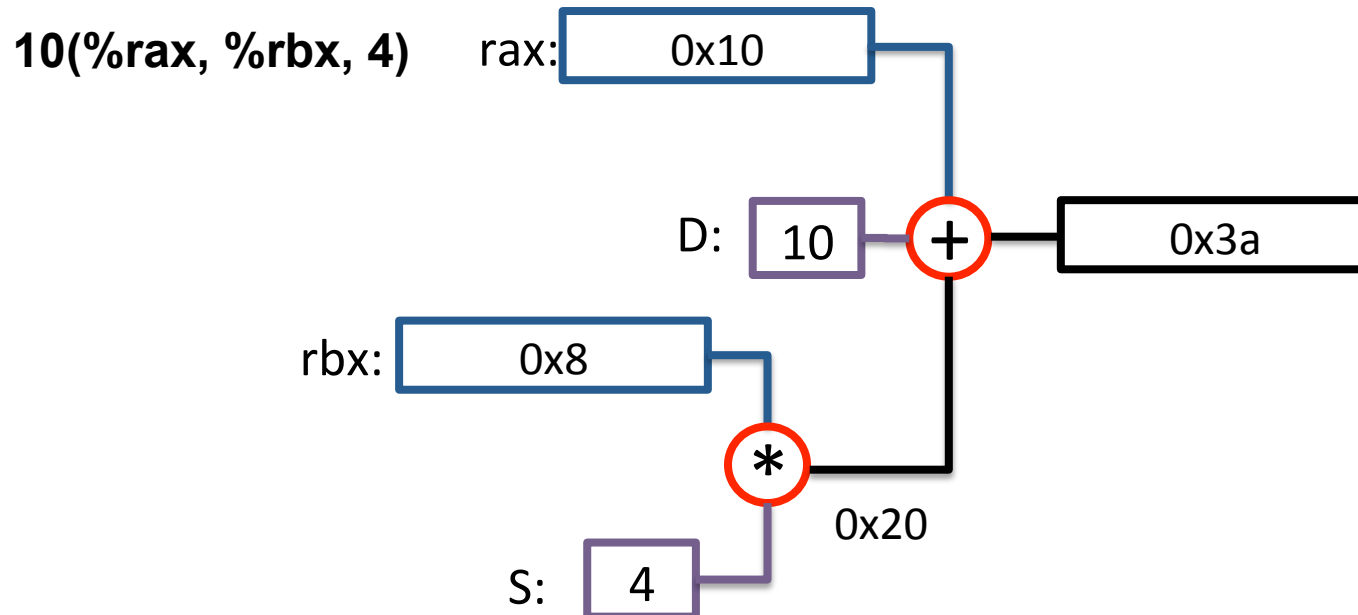
# Example



# 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

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

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

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

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

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

# Address Computation Examples

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

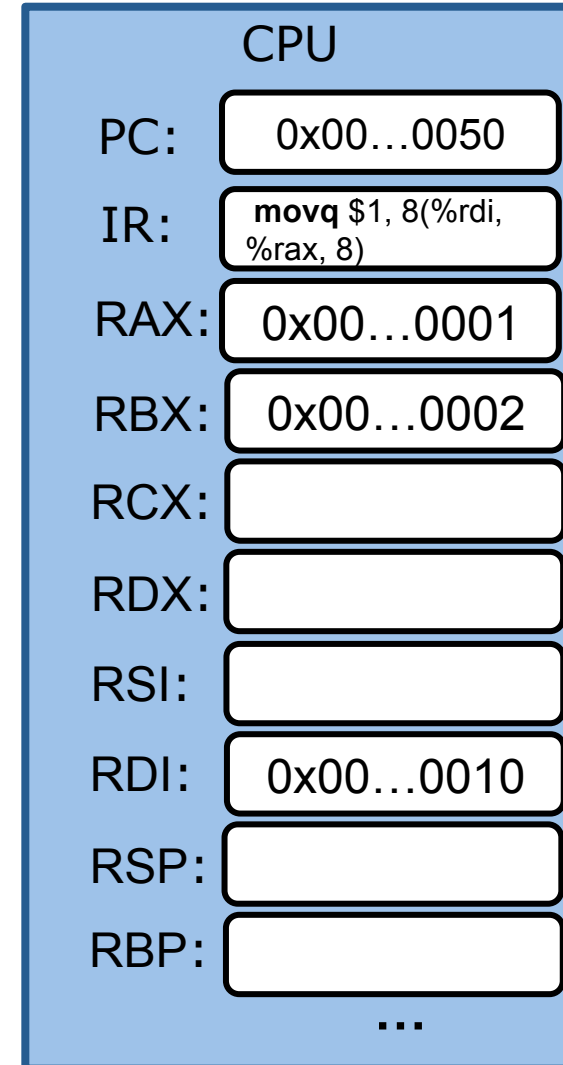
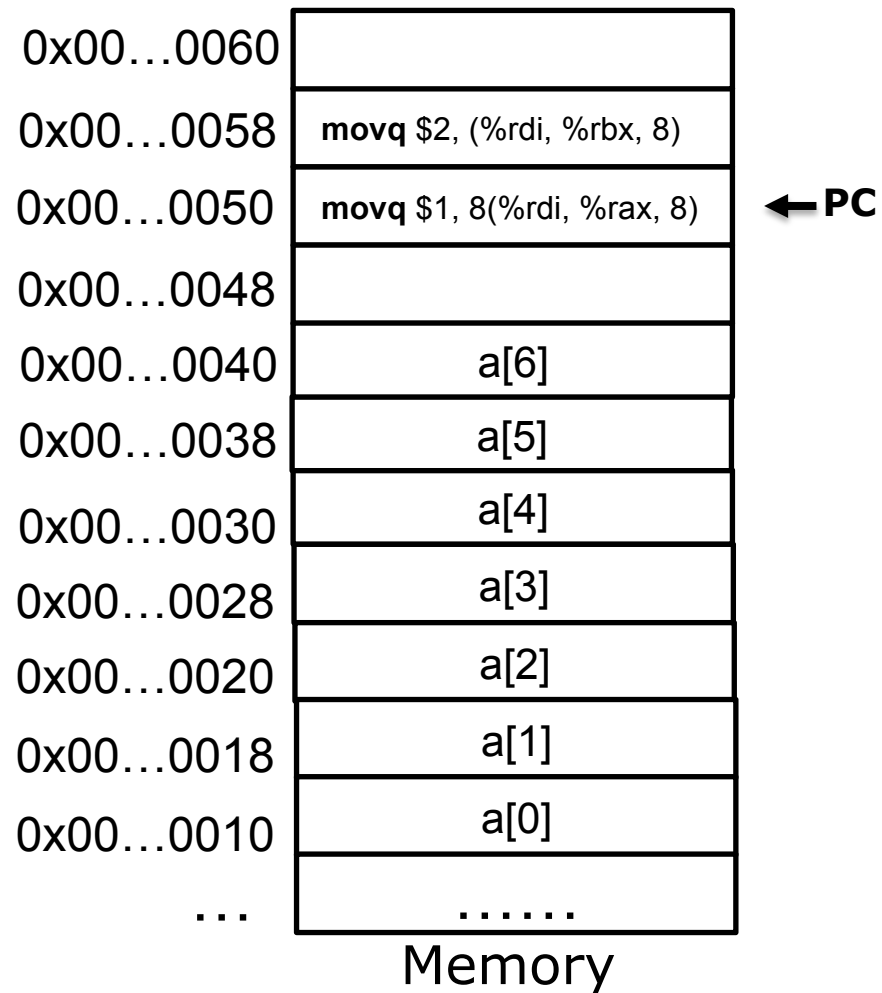
Expression	Address Computation	Address
<code>0x8(%rdx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%rdx,%rcx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%rdx,%rcx,4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80(,%rdx,2)</code>		

# Address Computation Examples

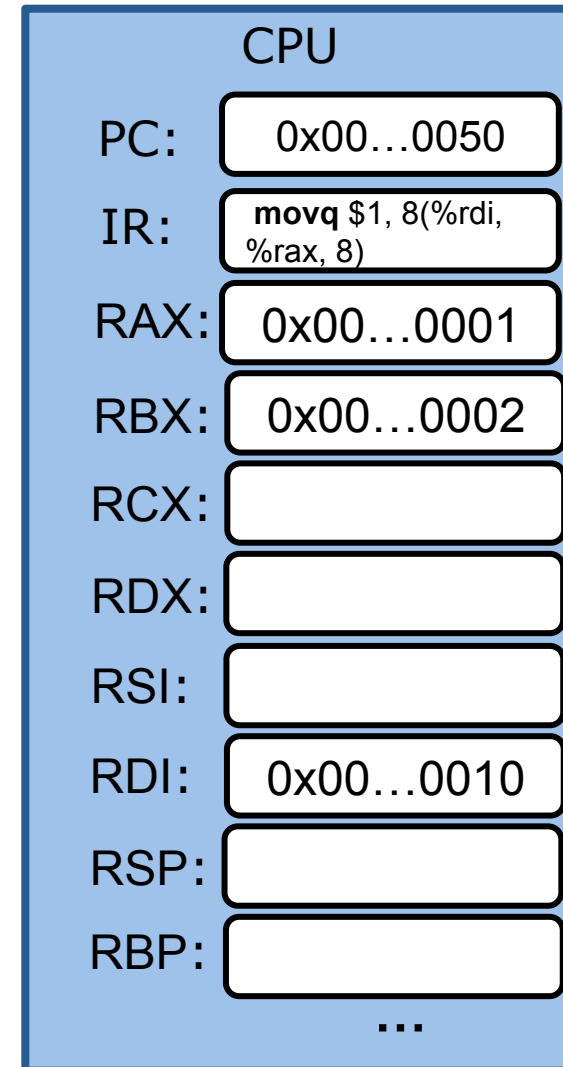
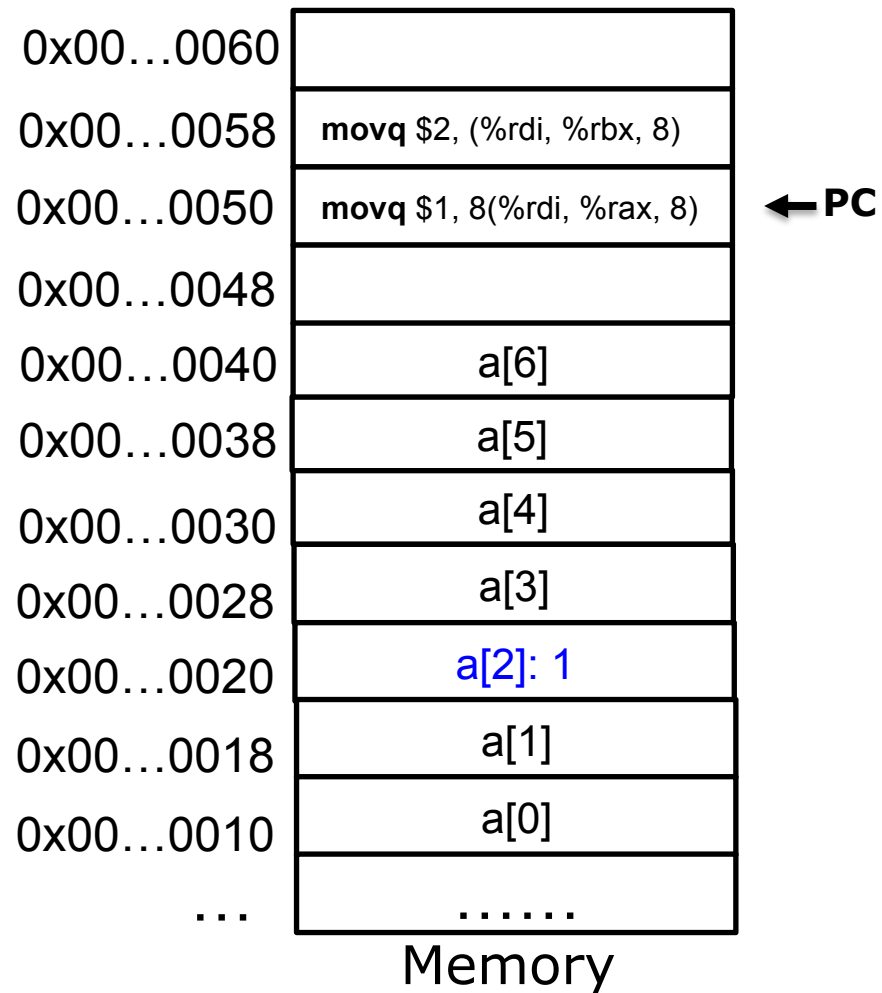
<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x100</code>

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

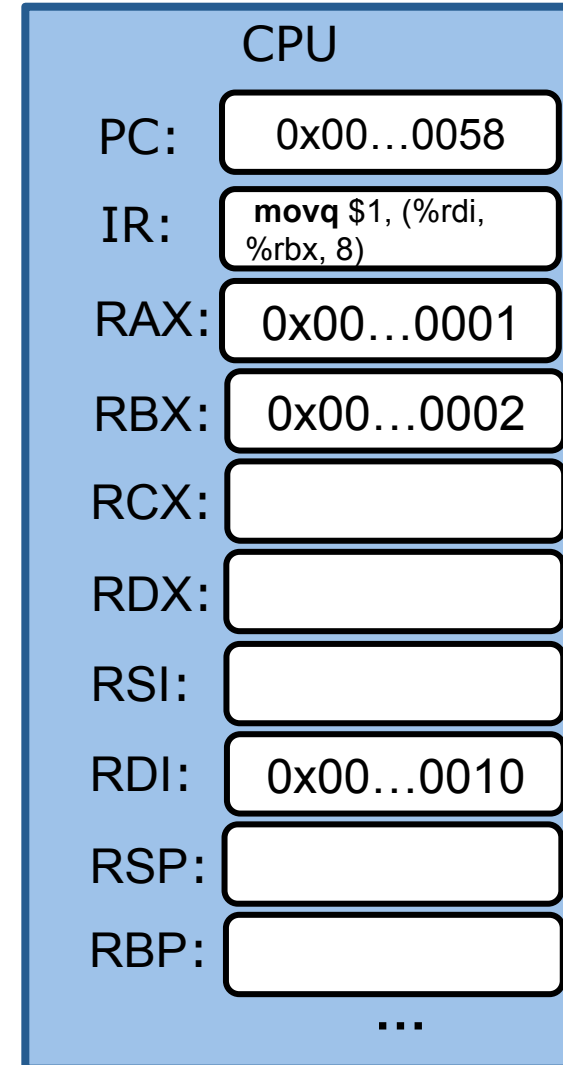
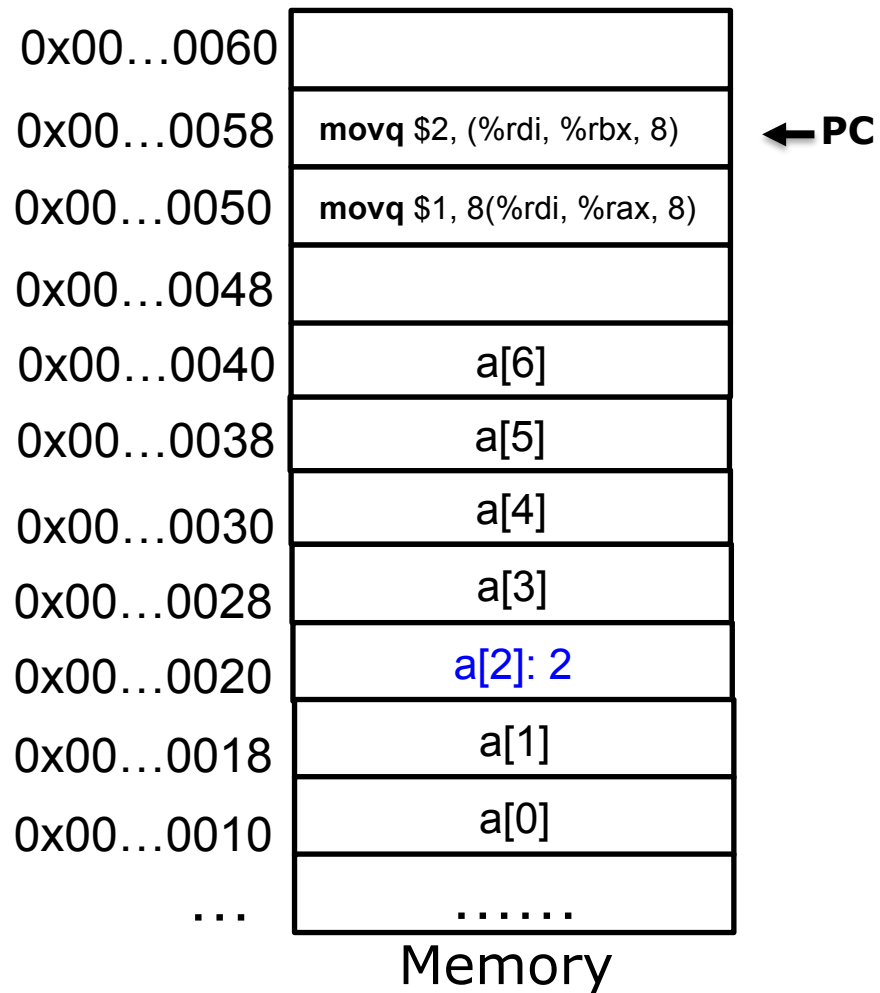


# Example





# Example



# mov{bwlq}

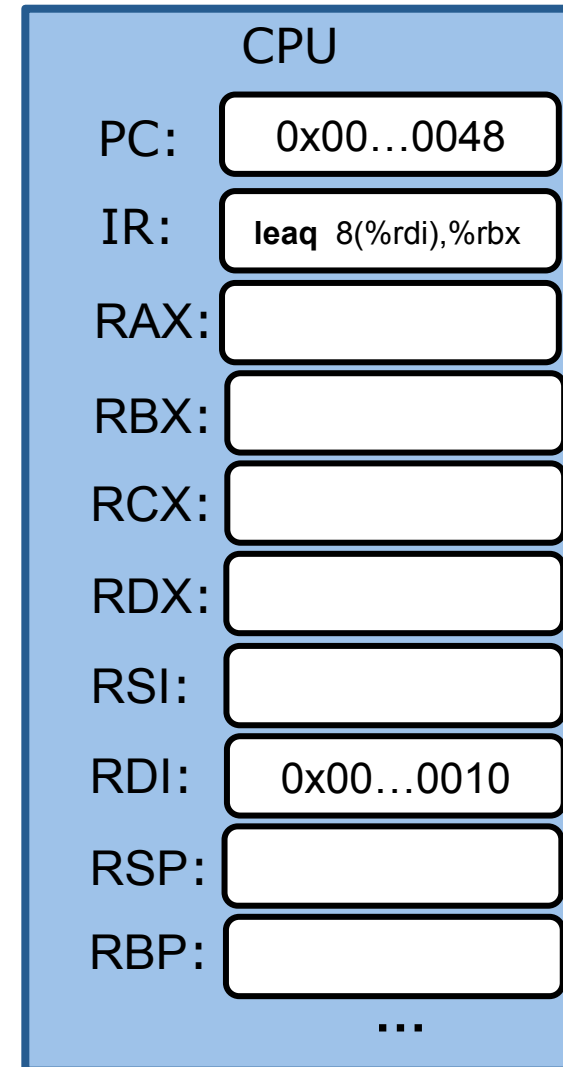
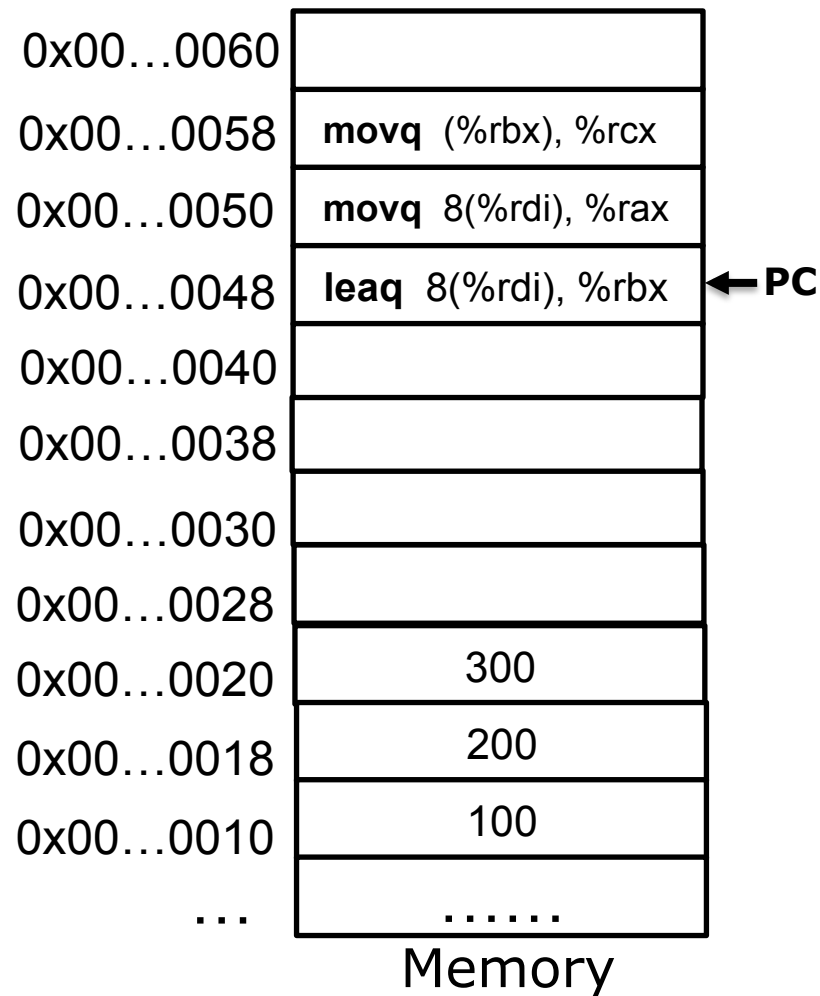
<b>movb</b> src, dest	Copy a <b>byte</b> from the source operand to the destination operand. e.g., <b>movb</b> %al, %bl
<b>movw</b> src, dest	Copy a <b>word</b> from the source operand to the destination operand. e.g., <b>movw</b> %ax, %bx
<b>movl</b> src, dest	Copy a <b>long</b> (32 bits) from the source operand to the destination operand. e.g., <b>movl</b> %eax, %ebx
<b>movq</b> src, dest	Copy a <b>quadword</b> from the source operand to the destination operand. e.g., <b>movq</b> %rax, %rbx

# The lea instruction

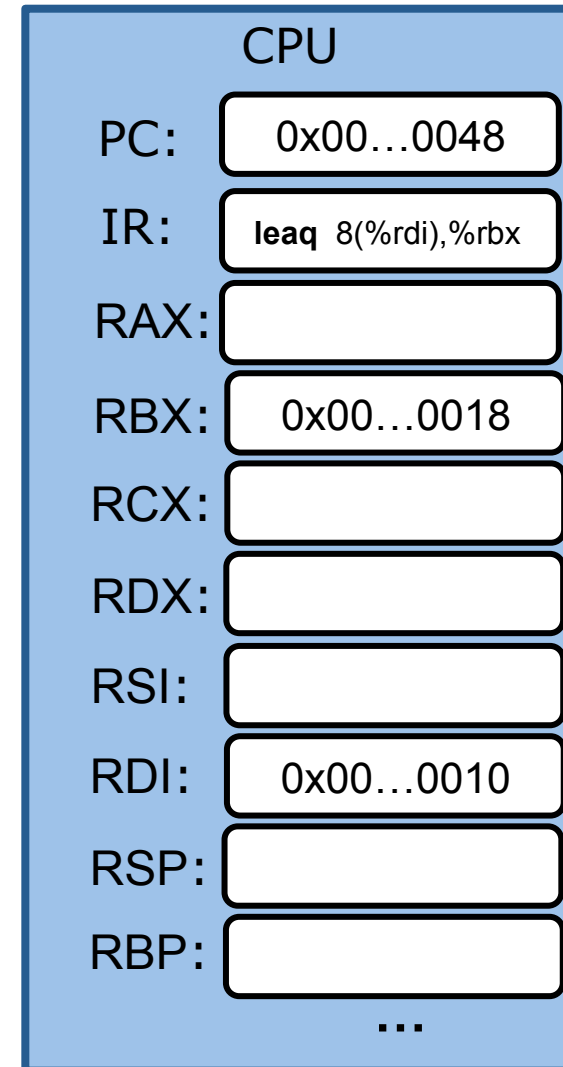
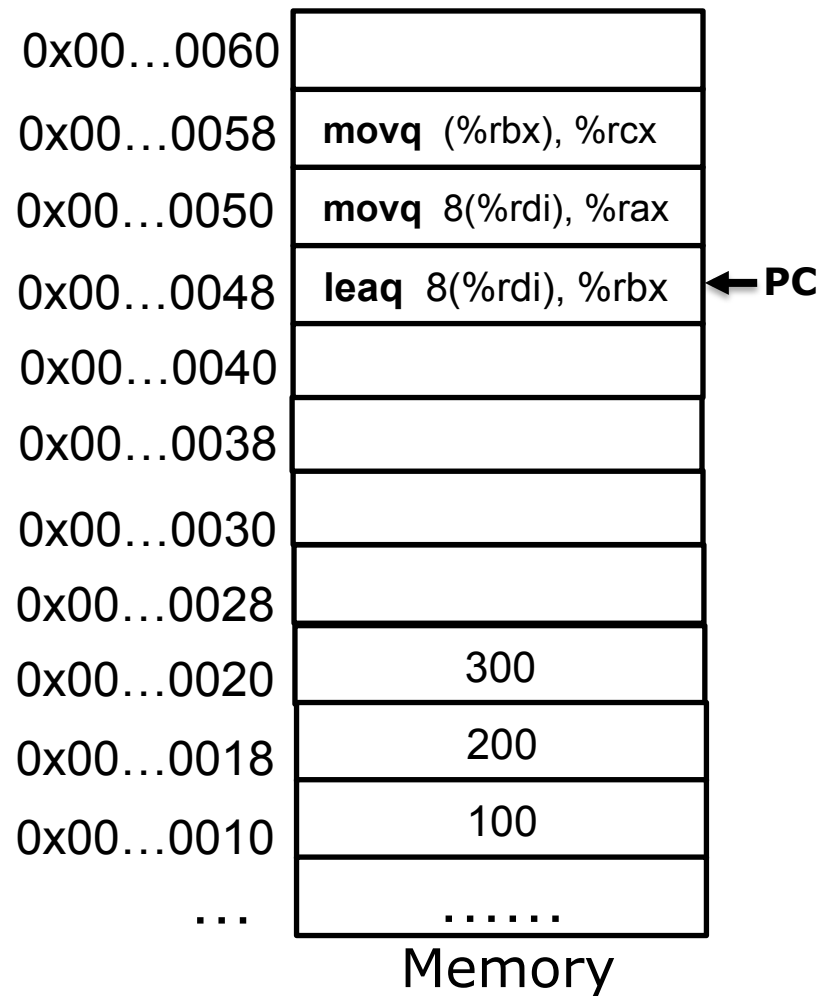
**leaq** *Source*, *Dest*

- load effective address: set *Dest* to the address denoted by *Source* address mode expression

# Example



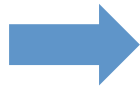
# Example



# A common usage of leaq

Compute expressions:  $x + K*y + d$  ( $K=1, 2, 4, \text{ or } 8$ )

```
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           $\text{Dest} = \text{Dest} - 1$

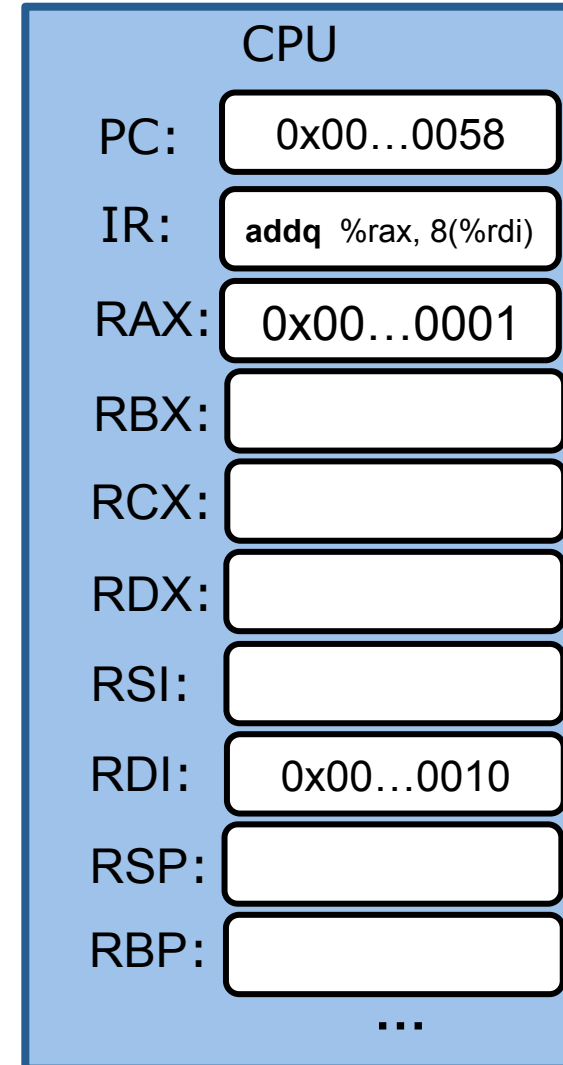
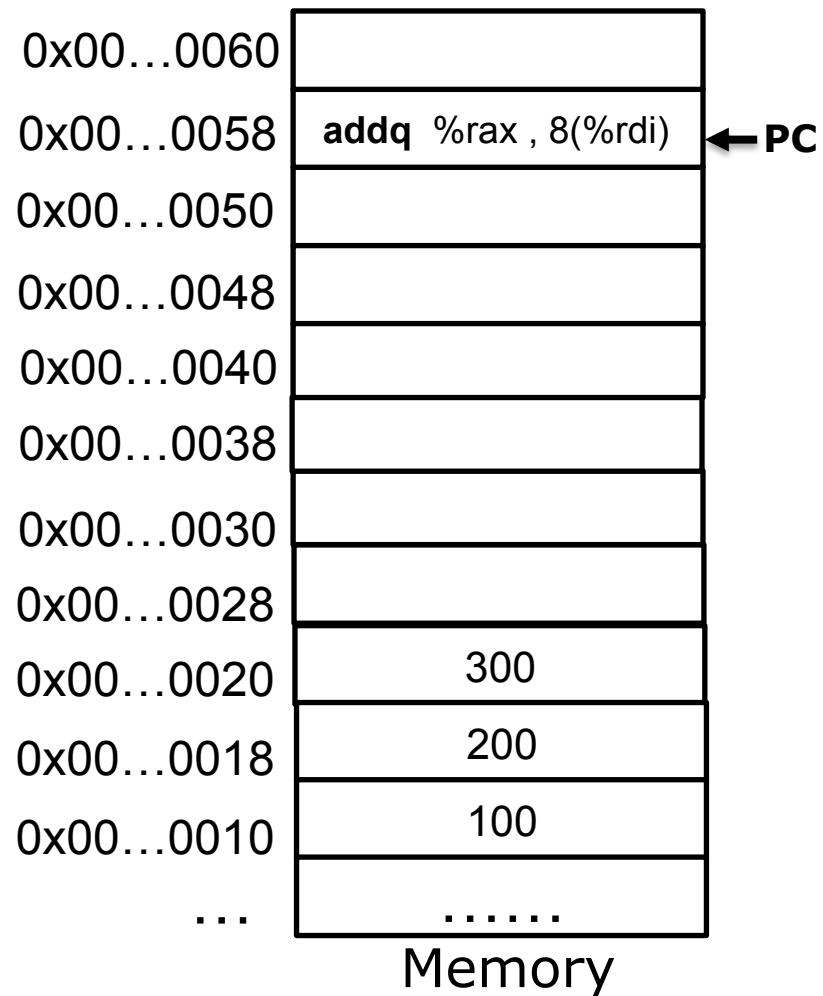
**negq**      Dest           $\text{Dest} = - \text{Dest}$

# Bitwise Operations

<b>salq</b>	Src, Dest	Dest = Dest << Src	Arithmetic left shift
<b>sarq</b>	Src, Dest	Dest = Dest >> Src	Arithmetic right shift
<b>shlq</b>	Src, Dest	Dest = Dest << Src	Logical left shift
<b>shrq</b>	Src, Dest	Dest = Dest >> Src	Logical right shift
<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	



# Example



# Example

