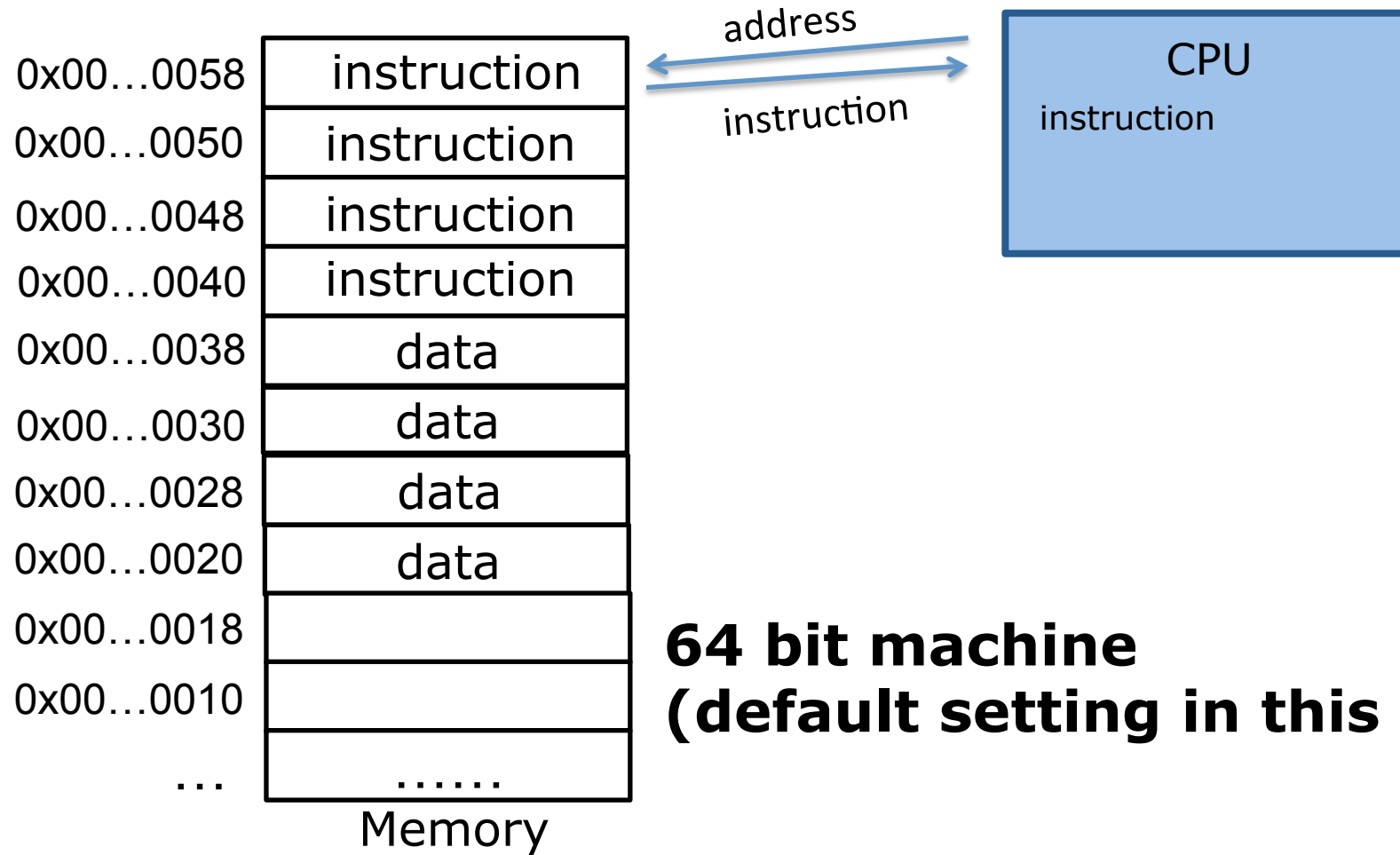


Machine Program: Basics

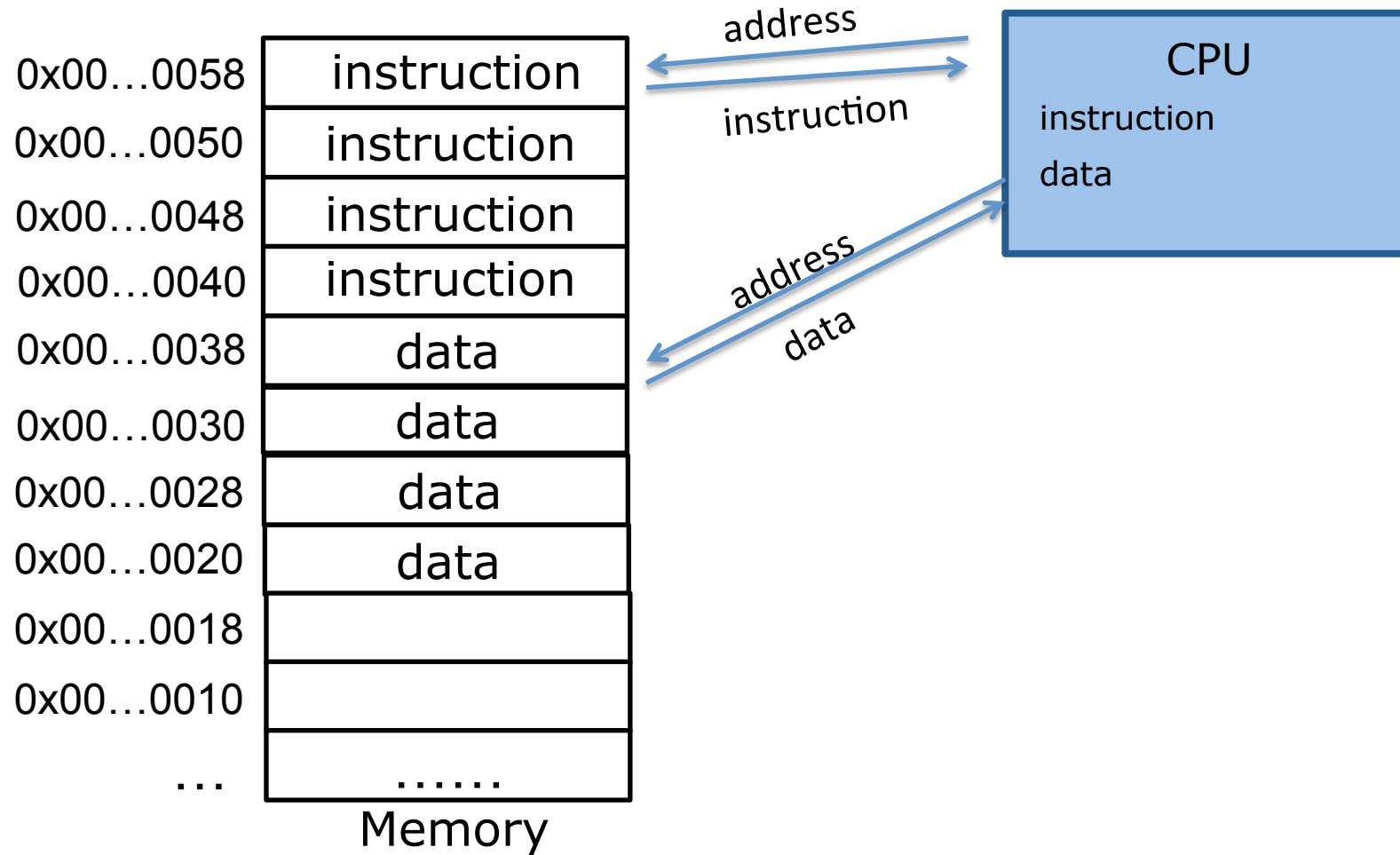
Jinyang Li

based on Tiger Wang's slides

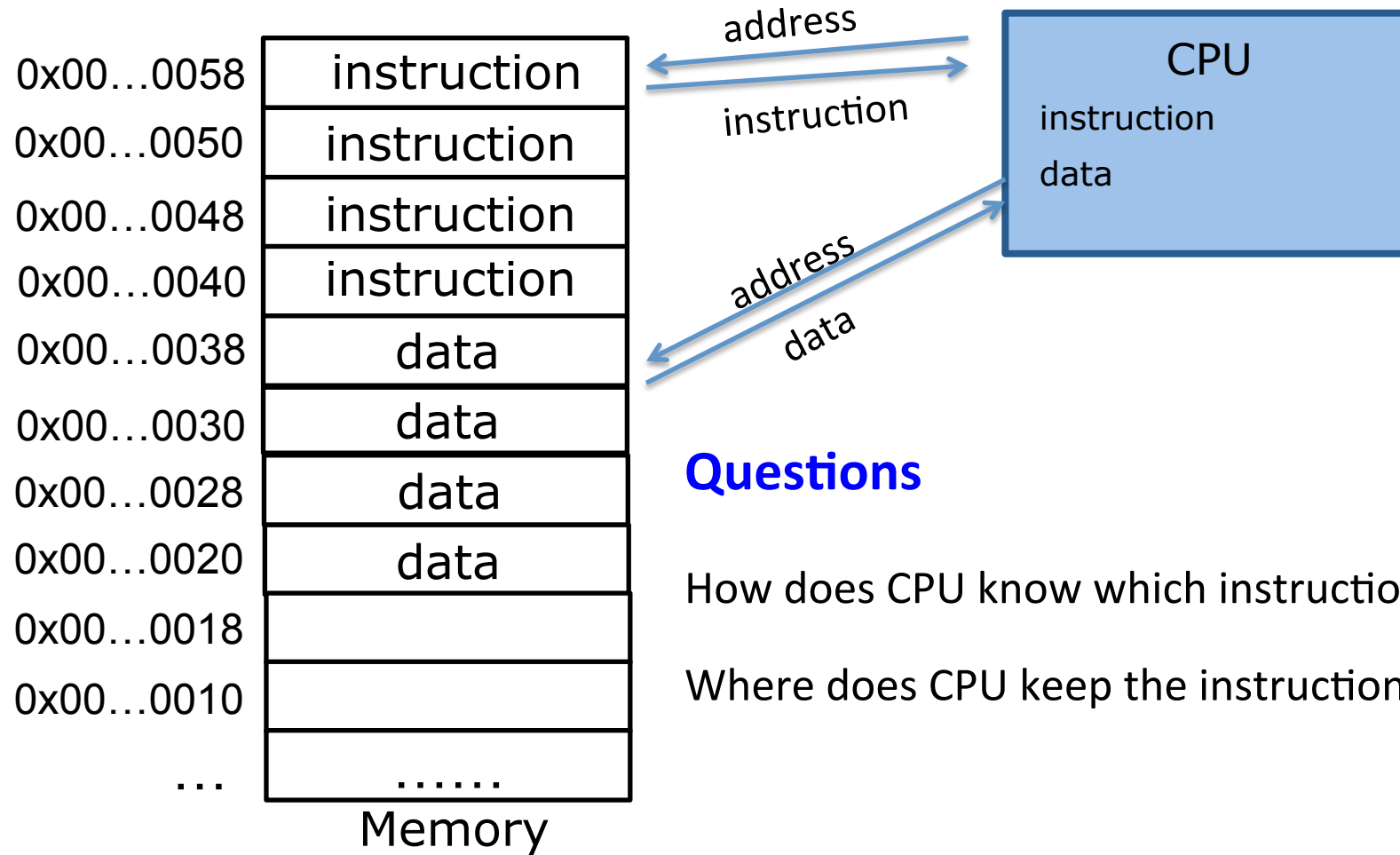
Your mental model



Your mental model



Your mental model



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

- Store memory address of next instruction
- 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
- 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:

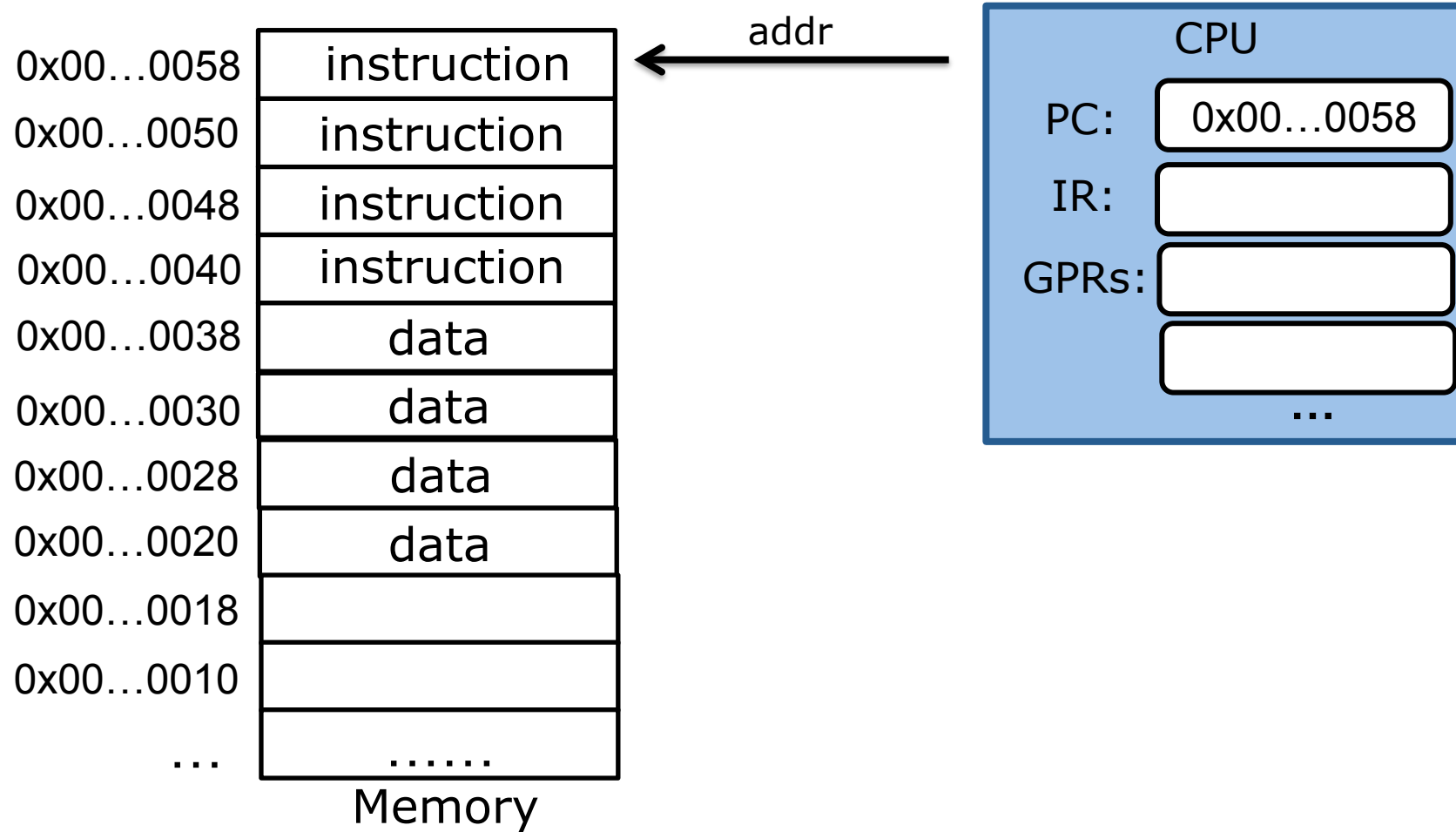
- Store operands and pointers used by program

Program status and control register:

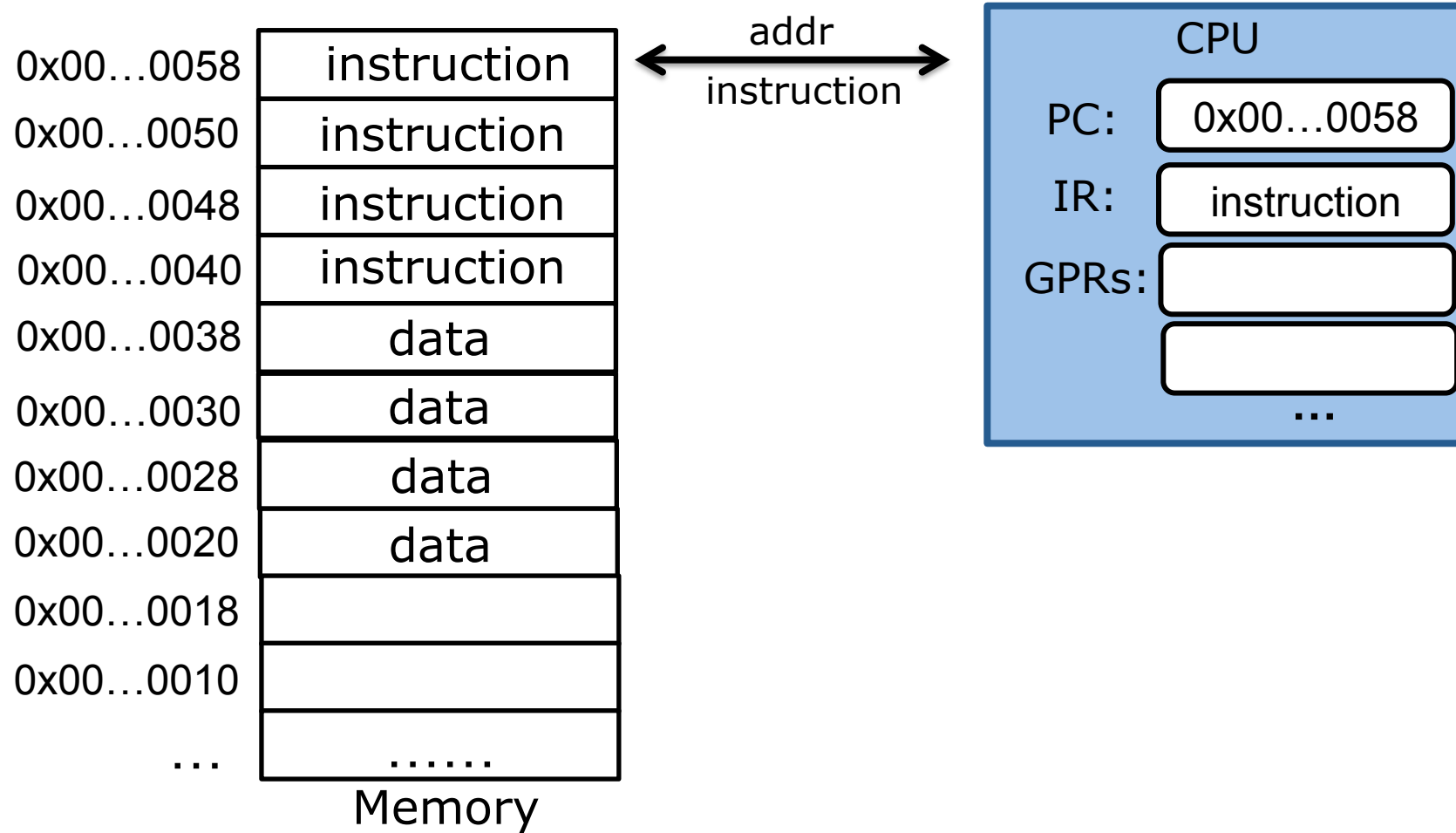
- Status of the program being executed
- All called “EFLAGS” in x86_64

Visible to programmers

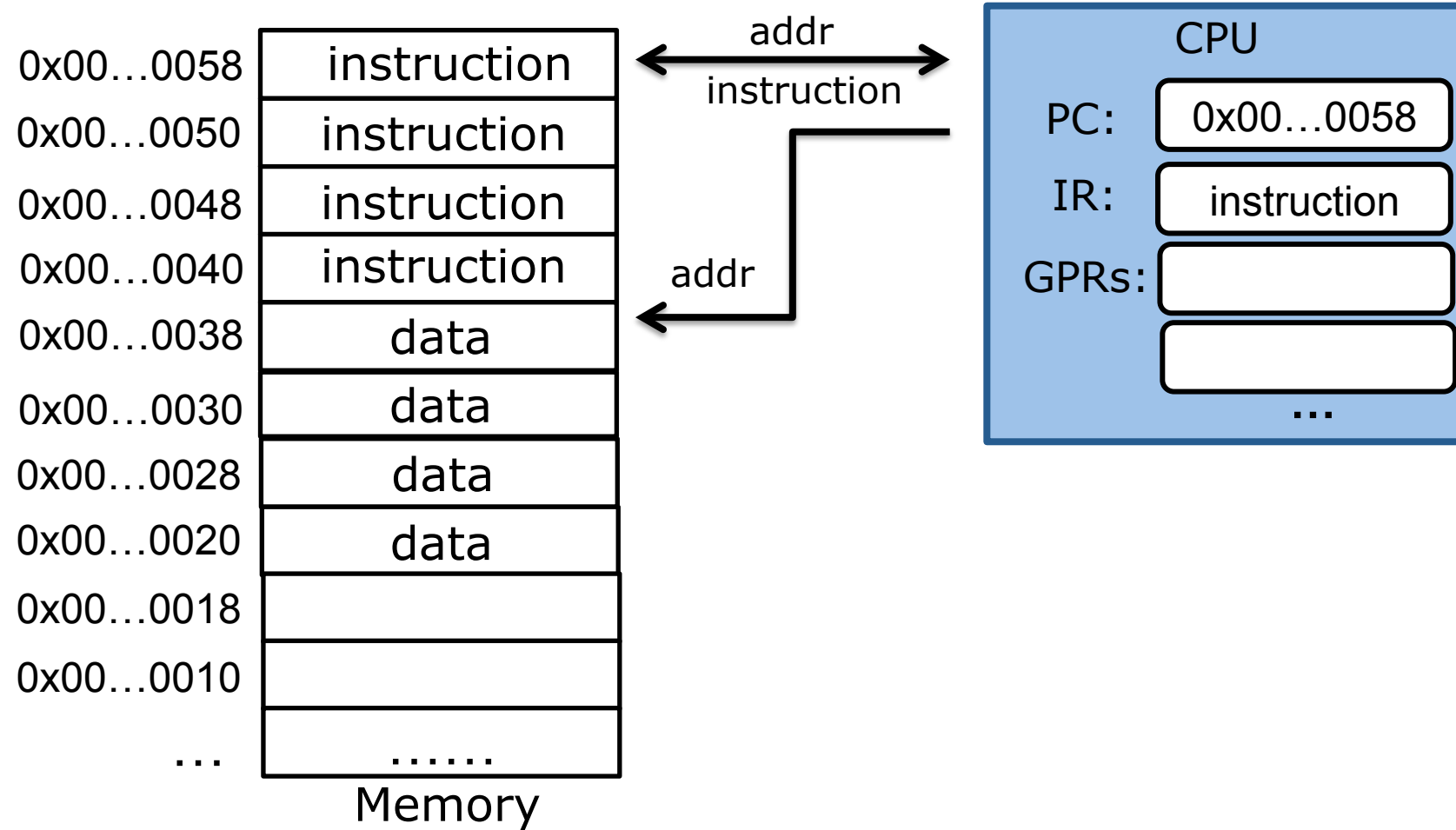
Your mental model



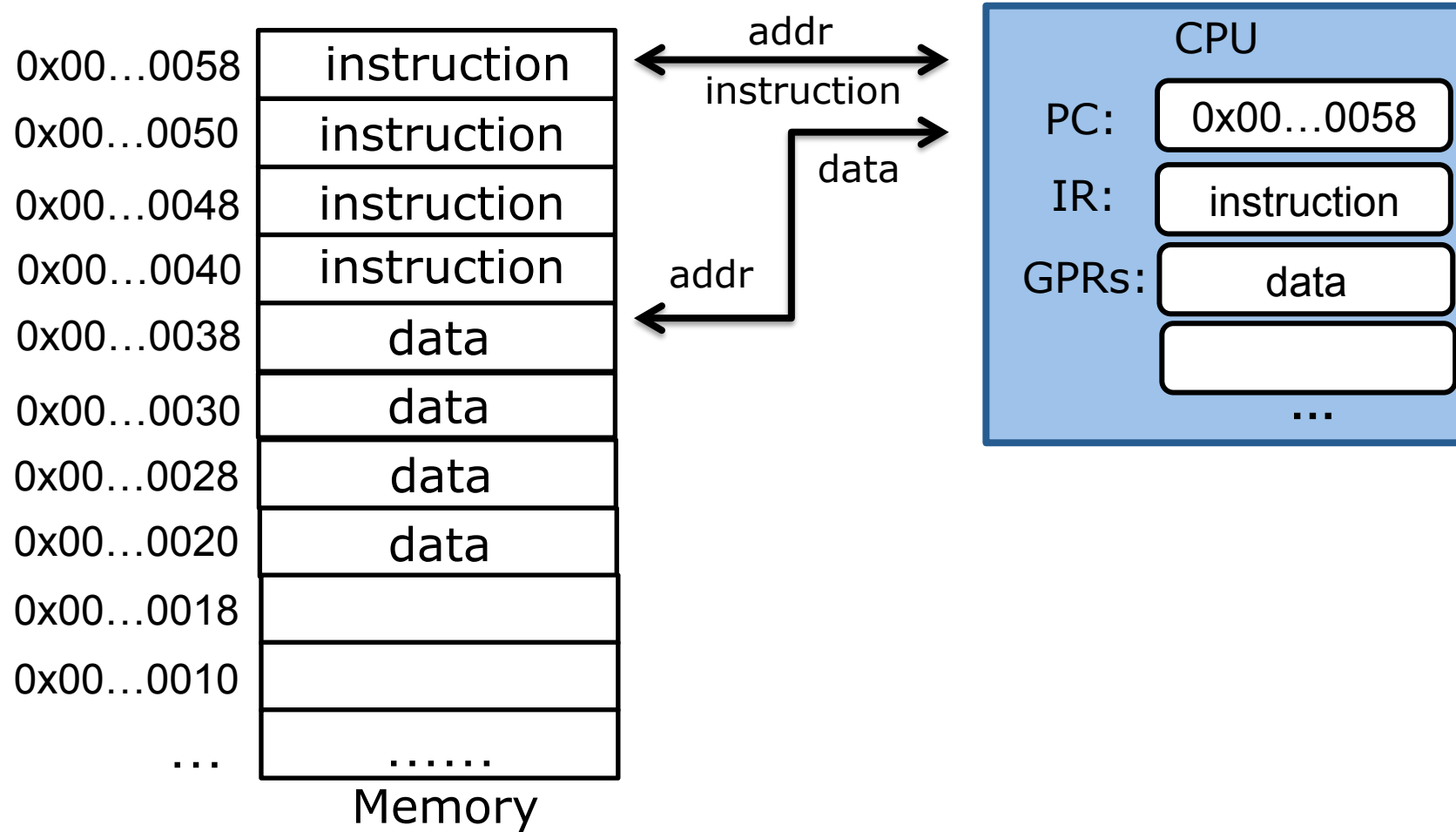
Your mental model



Your mental model



Your mental model



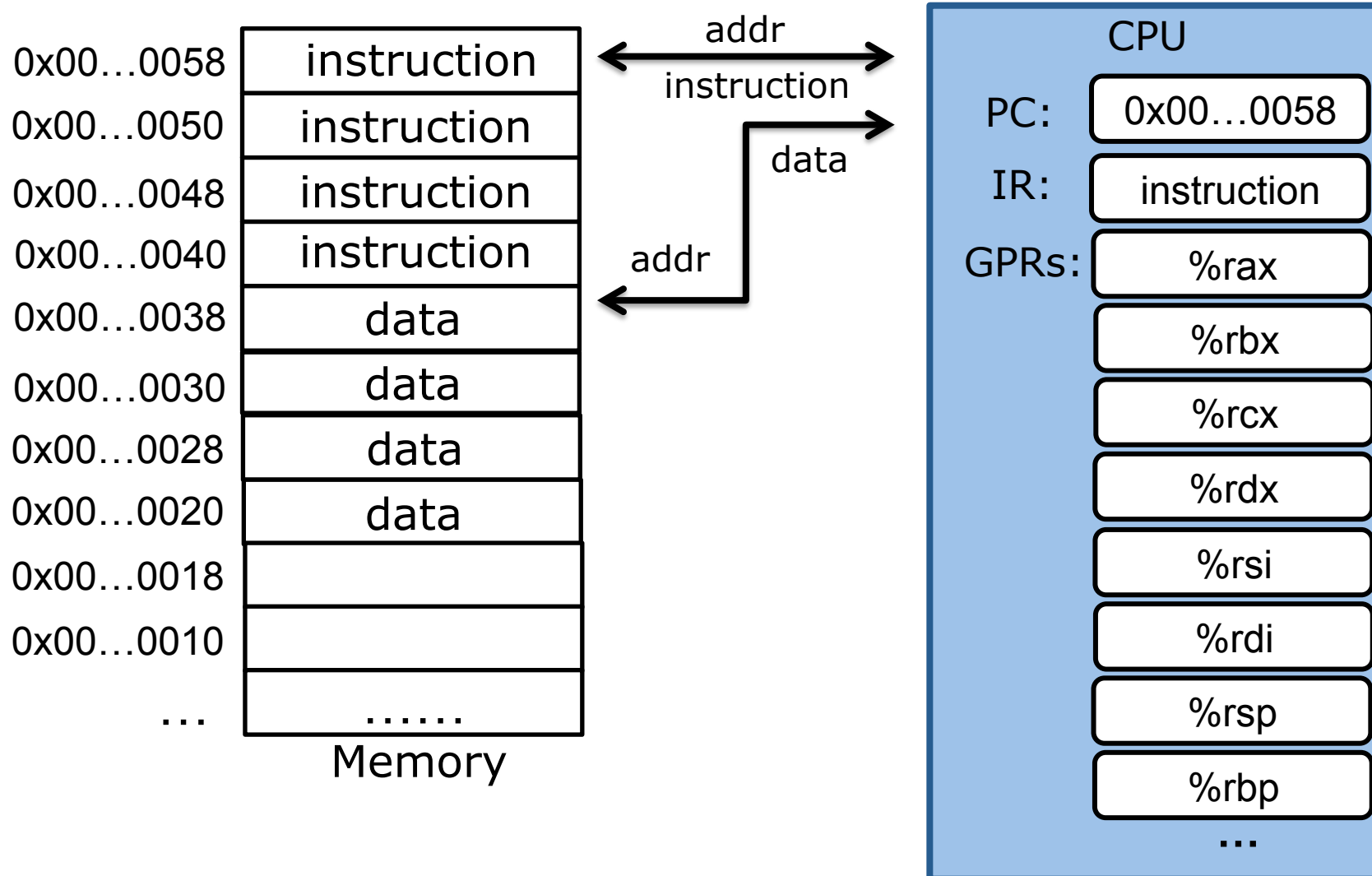
General Purpose Registers (intel x86-64)

%rax
%rbx
%rcx
%rdx
%rsi
%rdi
%rsp
%rbp

8 bytes

%r8
%r9
%r10
%r11
%r12
%r13
%r14
%r15

Your mental model



General Purpose Registers (intel x86-64)

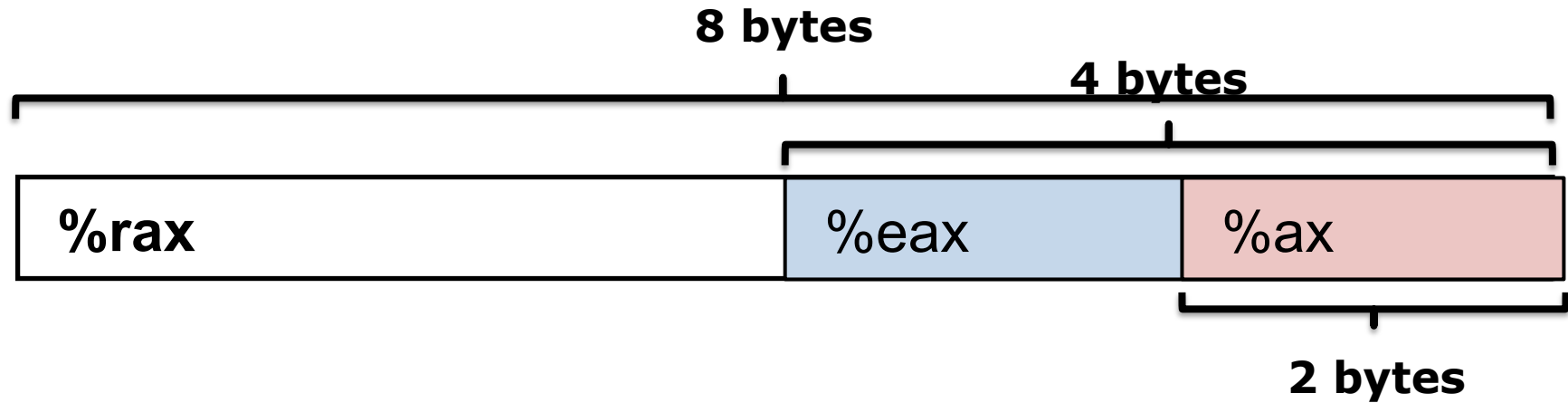
%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

8 bytes

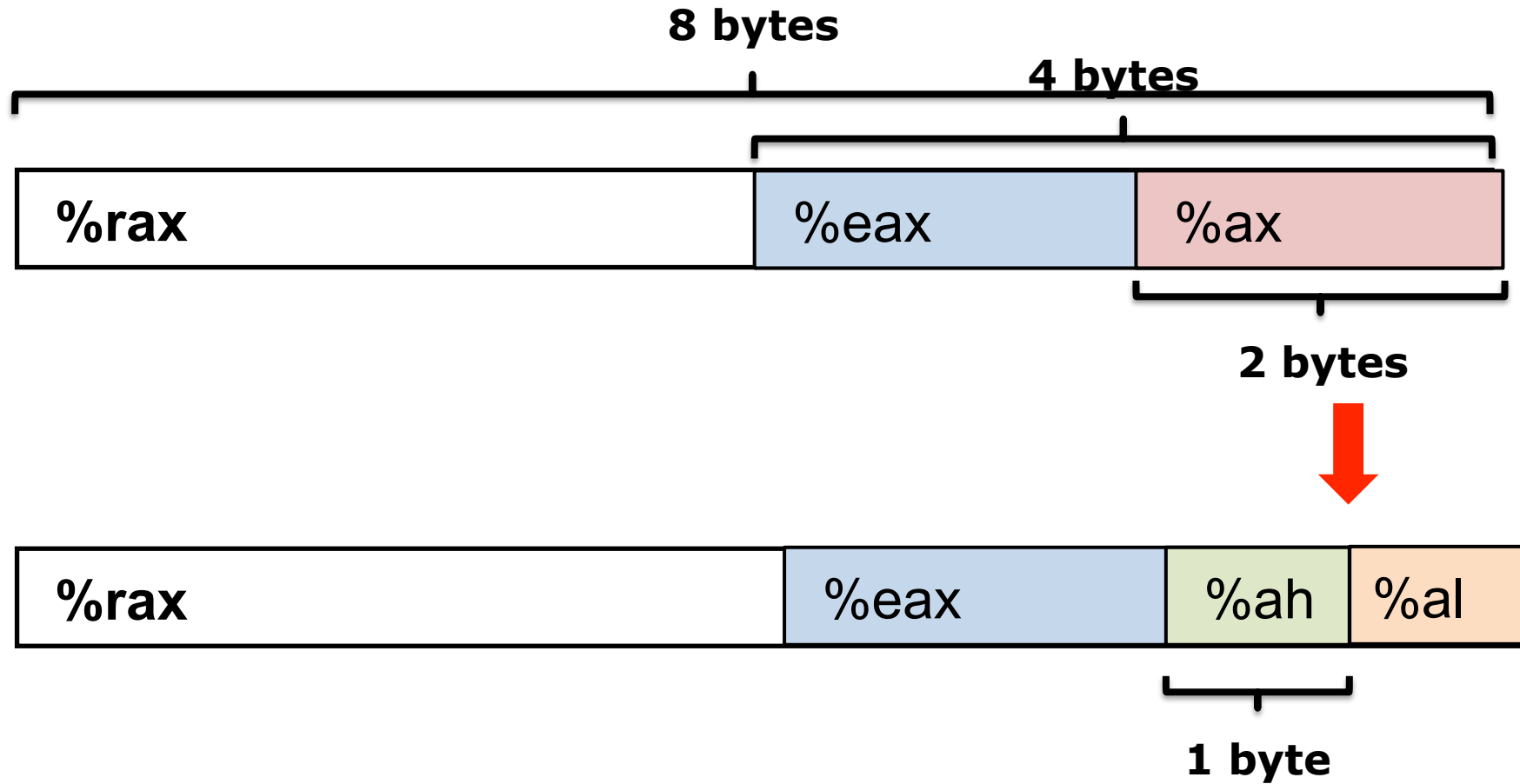
%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

4 bytes

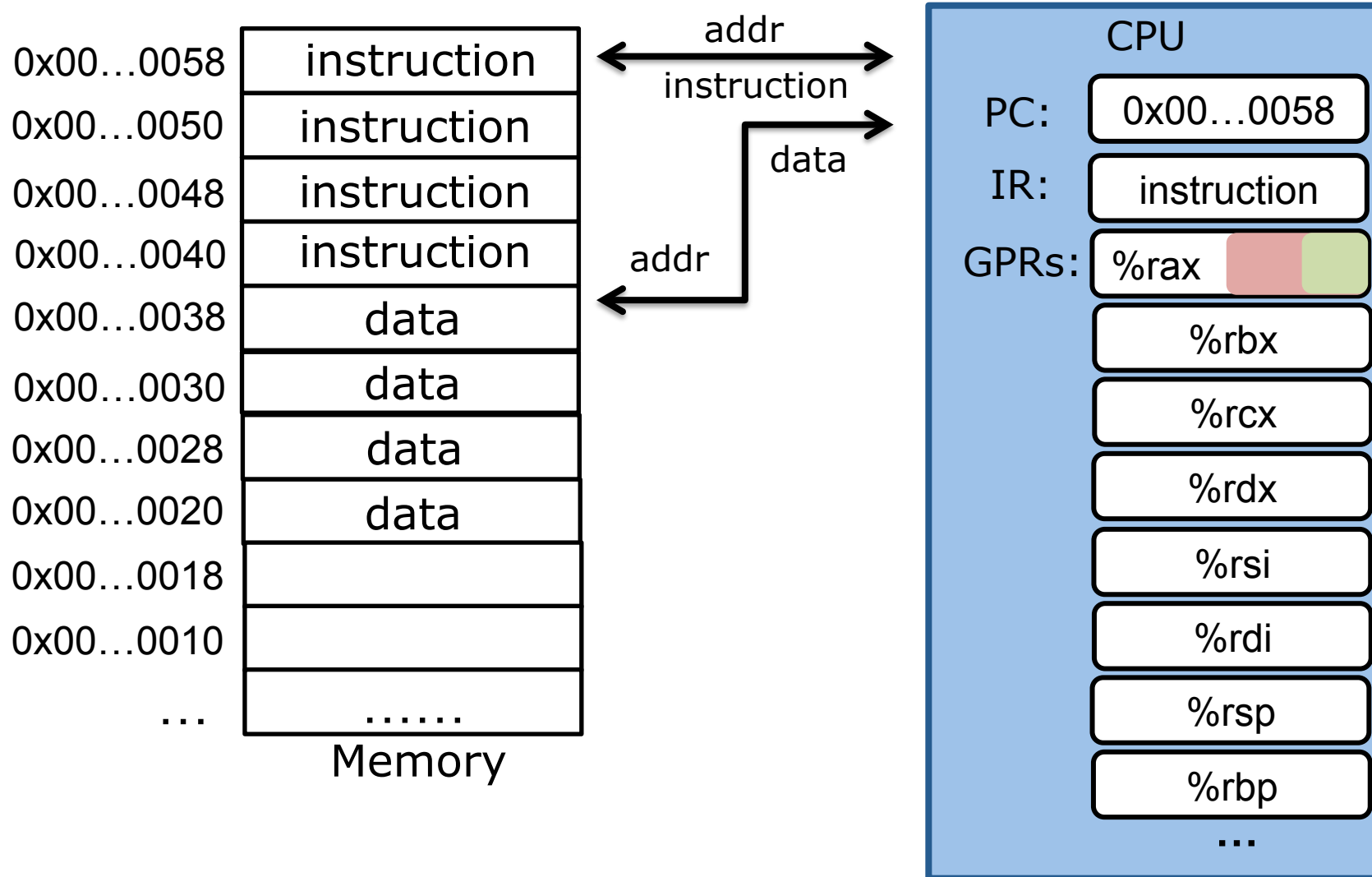
Use `%rax` and `%rbx` as examples



Use %rax as an example



Your mental model (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)

An abstract model of a computer

X86_64 is the ISA implemented by Intel/AMD CPUs
– 64-bit version of x86

this class' focus

ARM is another common ISA
– Phones, tablets, Raspberry Pi

X86 ISA



Intel® 64 and IA-32 Architectures Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4

NOTE: This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: Basic Architecture, Order Number 253665; Instruction Set Reference A-Z, Order Number 325383; System Programming Guide, Order Number 325384; Model-Specific Registers, Order Number 335592. Refer to all four volumes when evaluating your design needs.

Order Number: 325462-065US
December 2017

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^{suffix}**q** *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

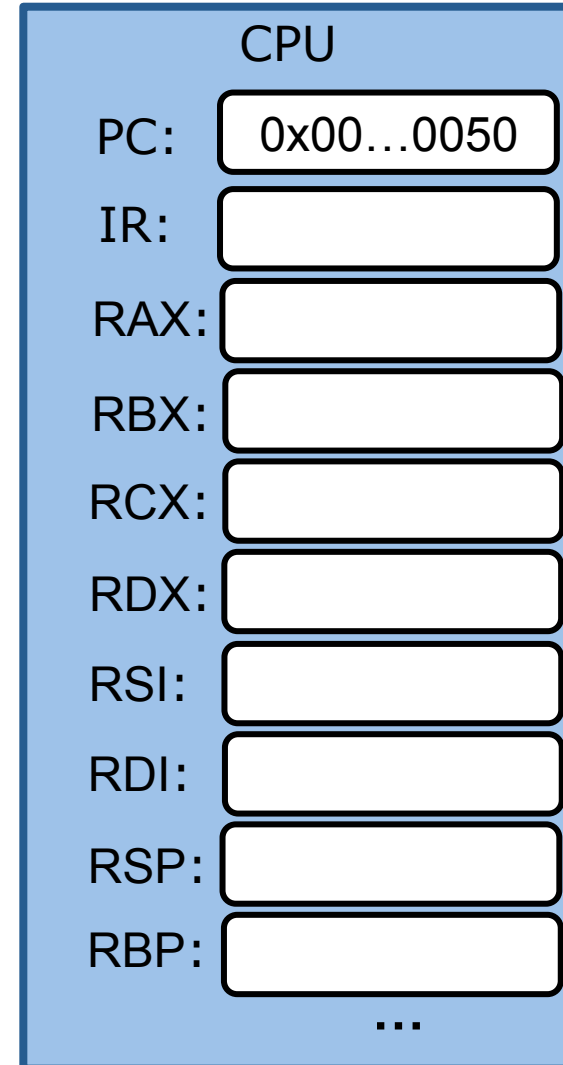
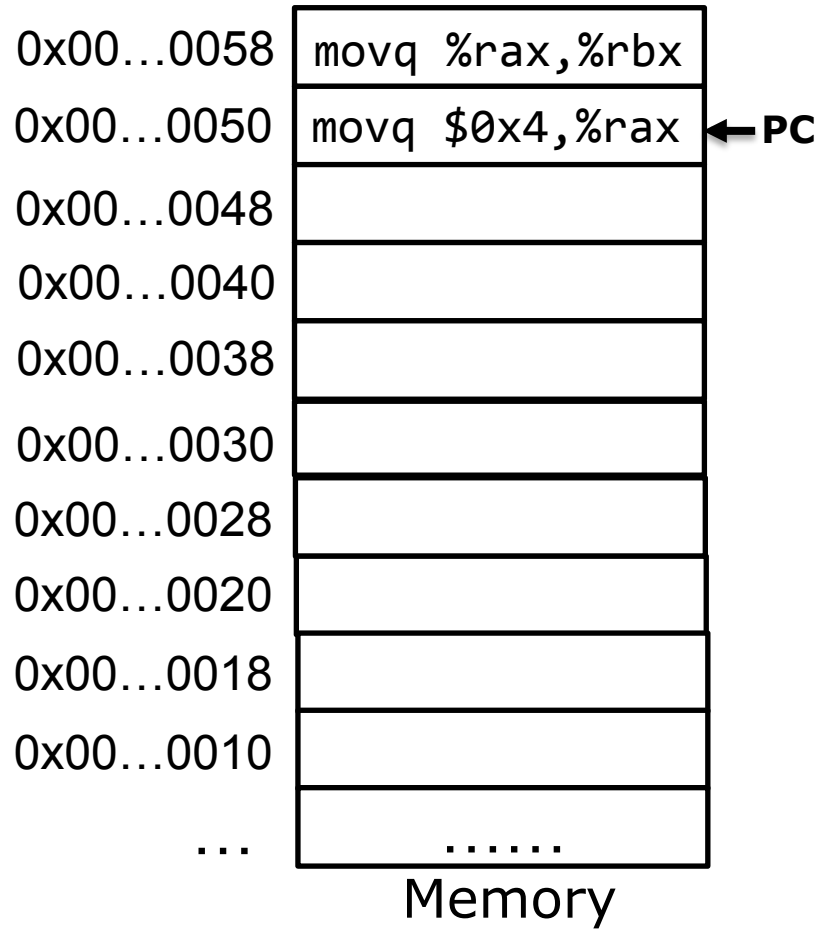
- **Immediate:** Constant integer data
 - Prefixed with \$
 - Example: \$0x400, \$-533
- **Register:** One of general purpose registers
 - Example: %rax, %rsi
- **Memory:** 8 consecutive bytes of memory
 - Indexed by register with various “address modes”
 - Simplest example: (%rax)

movq Operand combinations

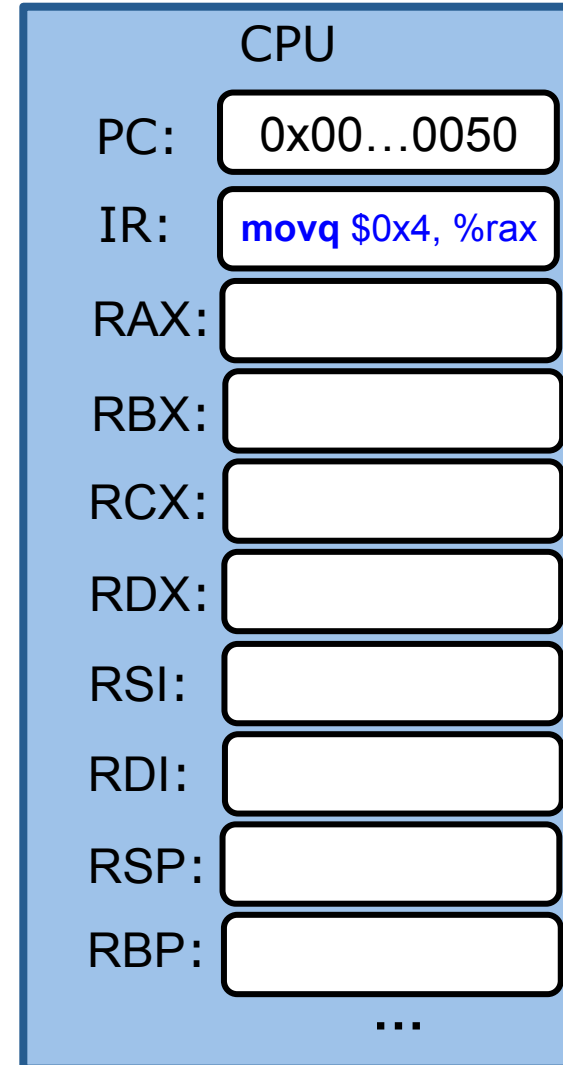
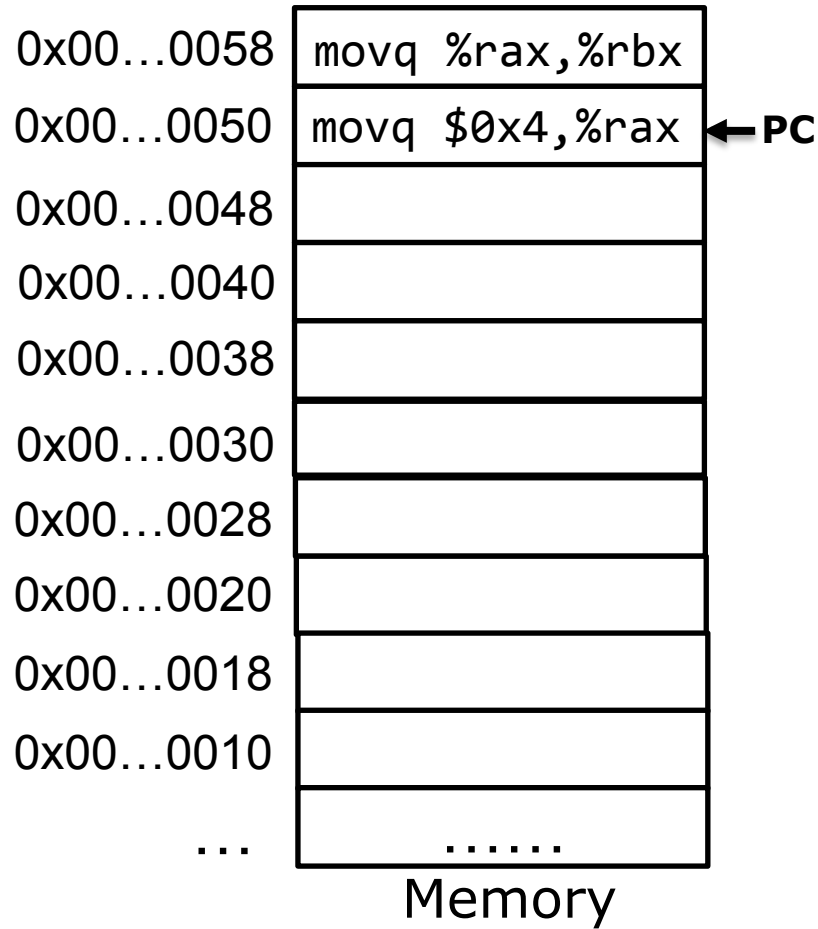
	Source	Dest	Source, Dest
movq	Imm	Reg	movq \$0x4,%rax
		Mem	movq \$0x4,(%rax)
	Reg	Reg	movq %rax,%rdx
		Mem	movq %rax,(%rdx)
	Mem	Reg	movq (%rax),%rdx

1. Immediate can only be *Source*
2. Cannot do memory-memory transfer with a single instruction

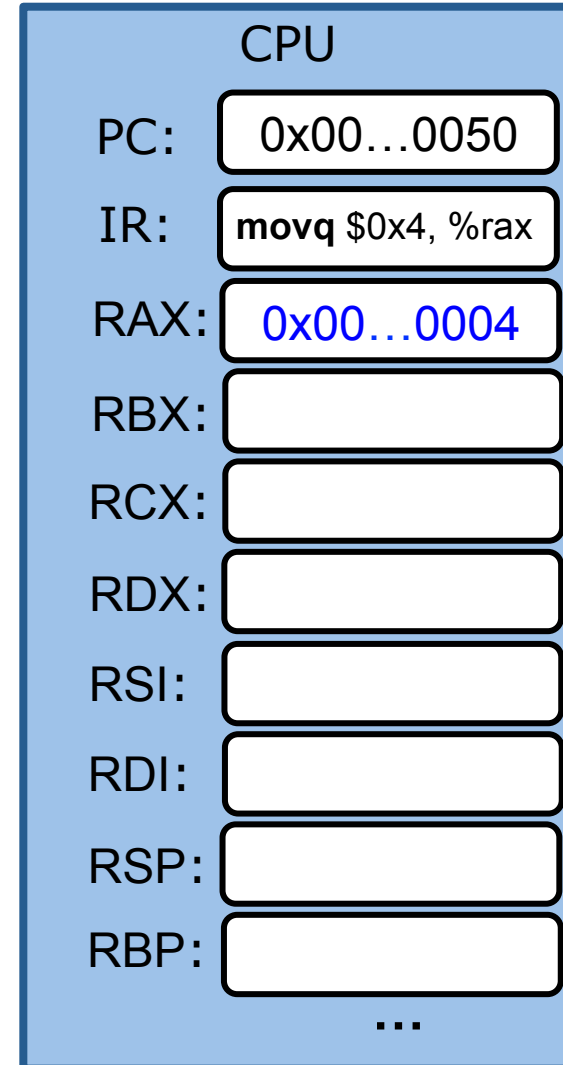
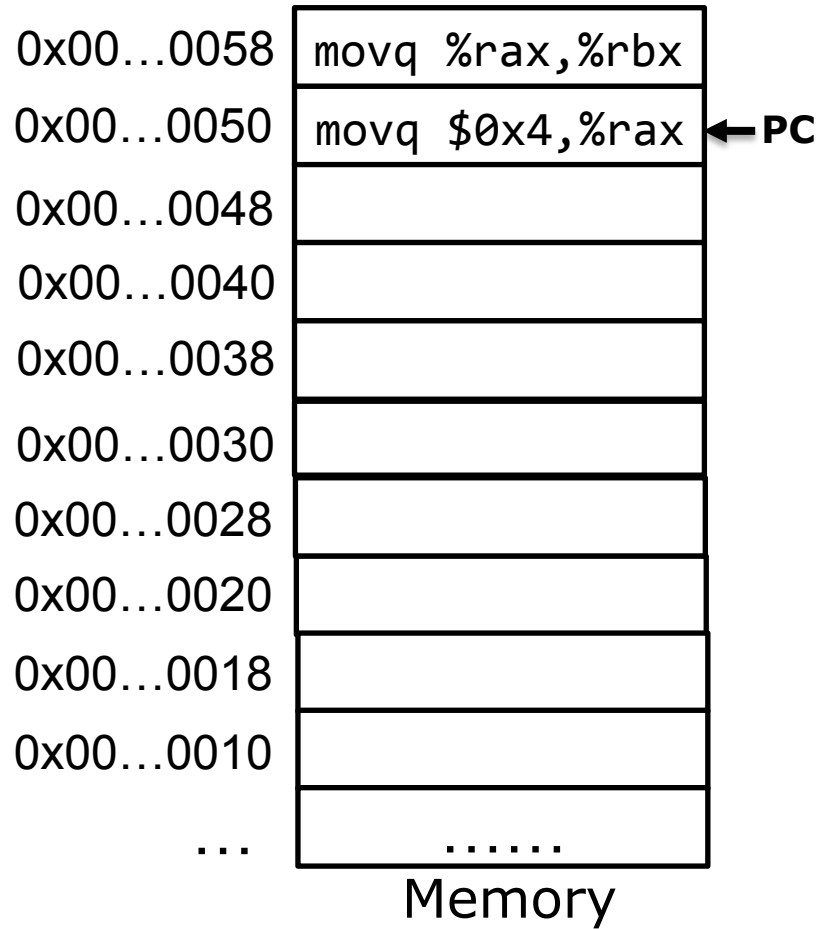
movq Imm, Reg



movq Imm, Reg



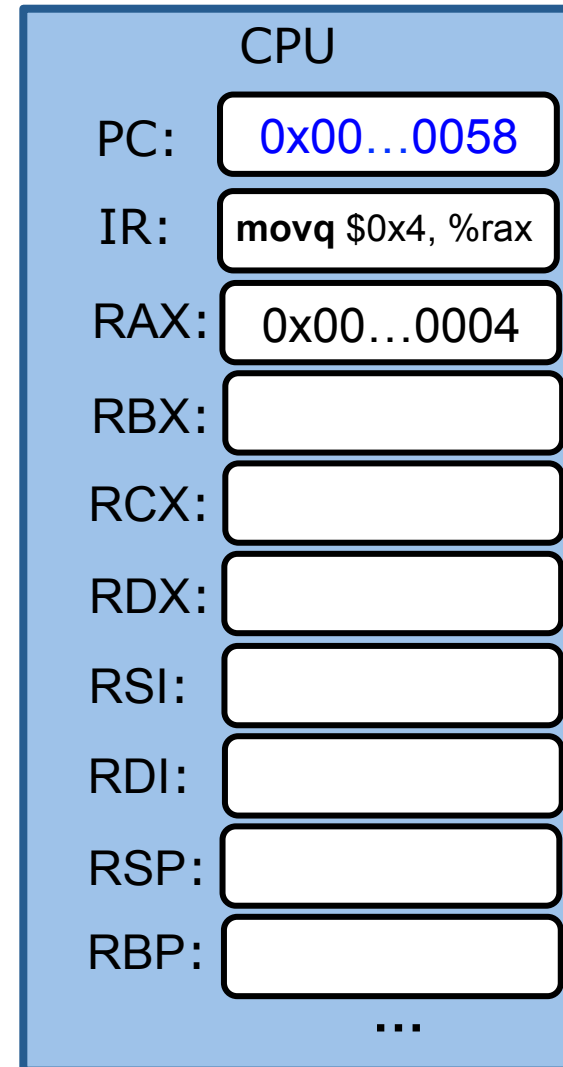
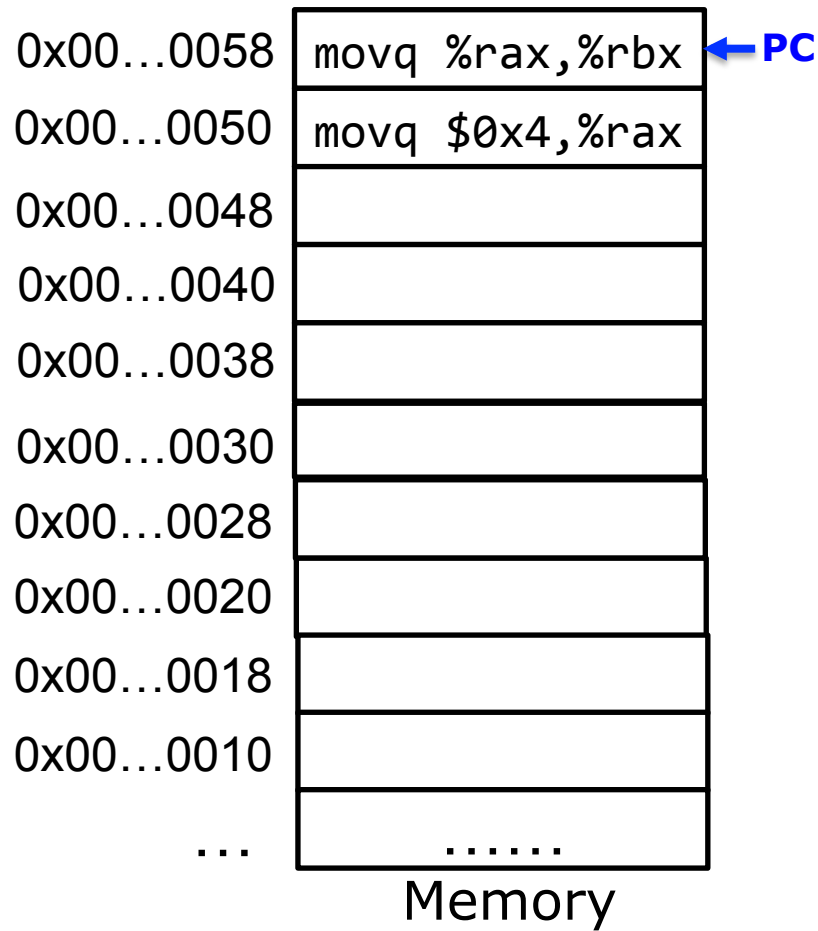
movq Imm, Reg



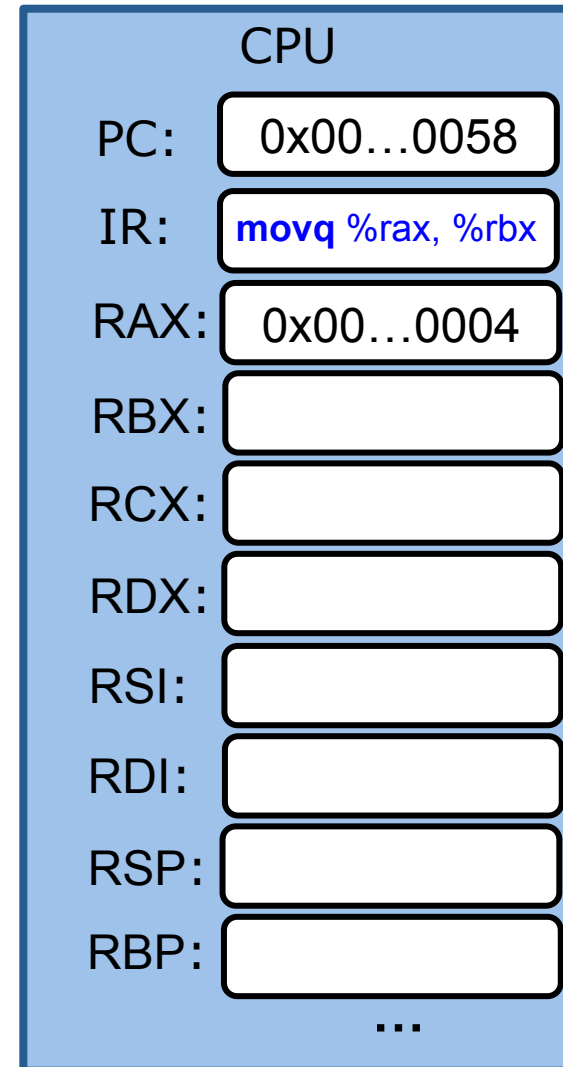
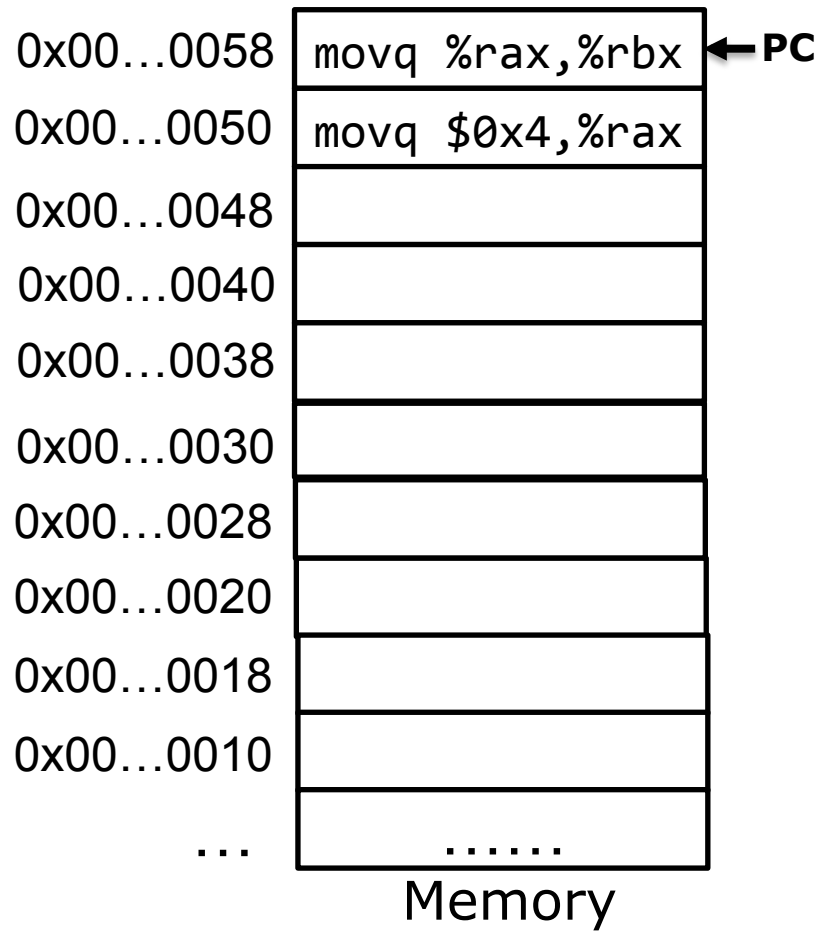
Steps

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).

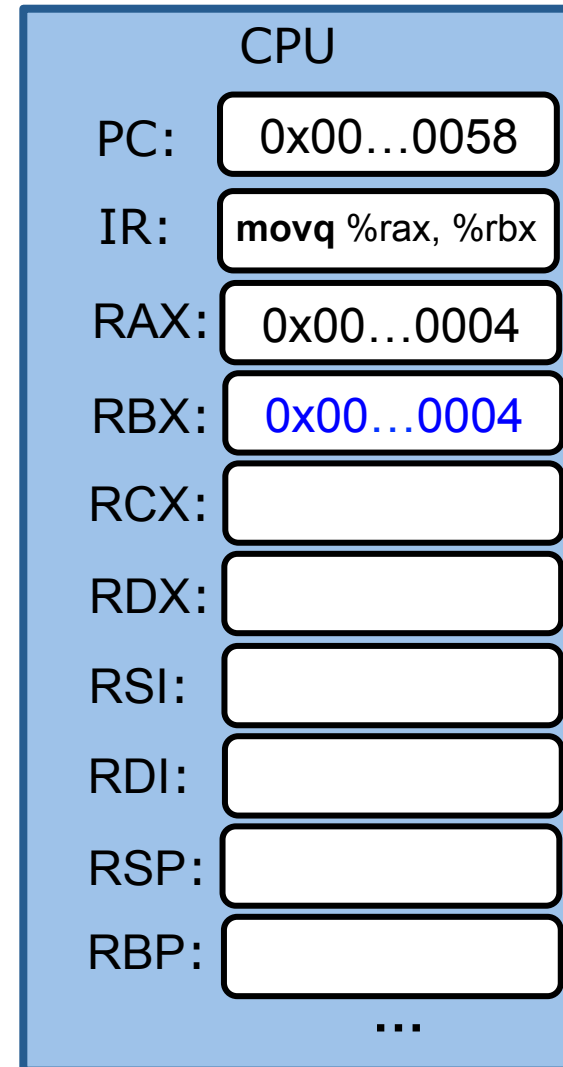
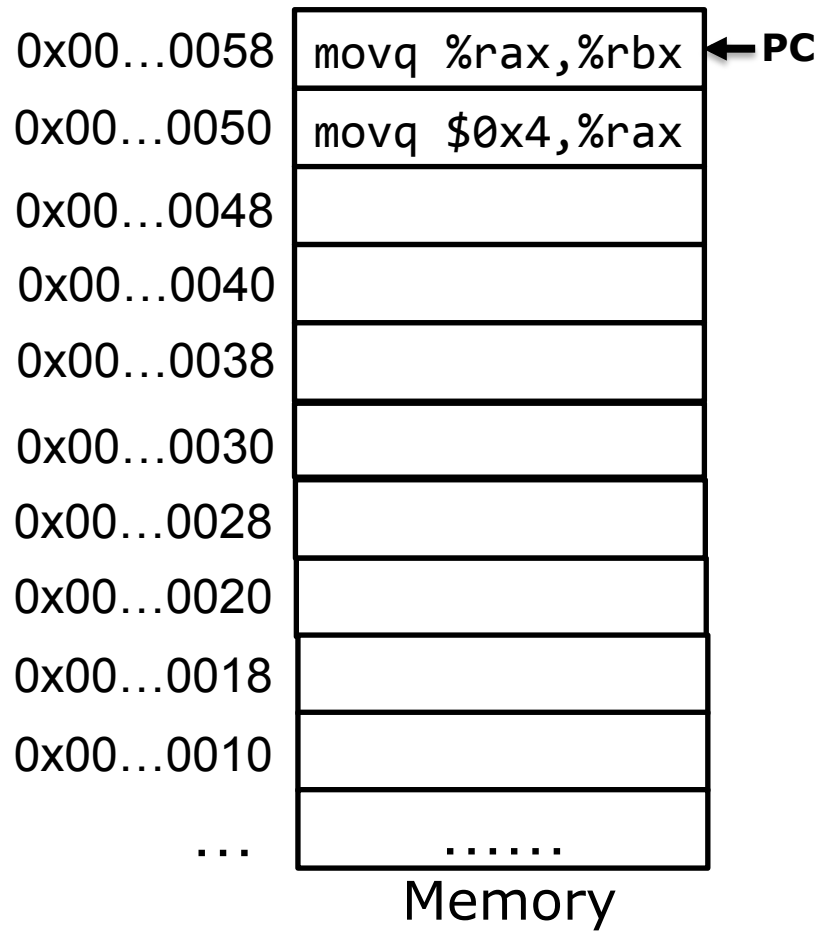
movq Reg, Reg



movq Reg, Reg



movq Reg, Reg



movq *Mem, Reg*

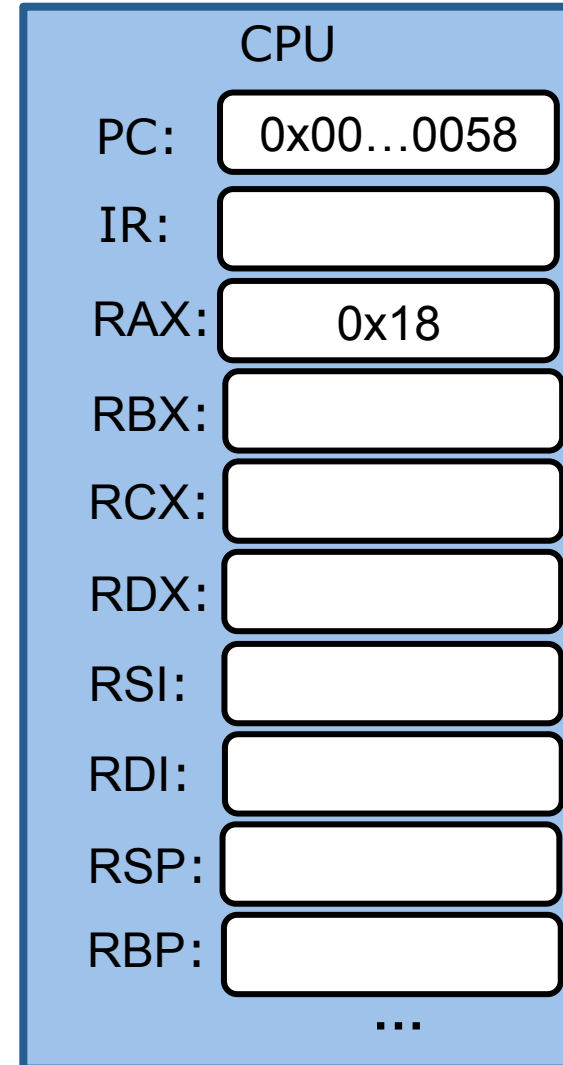
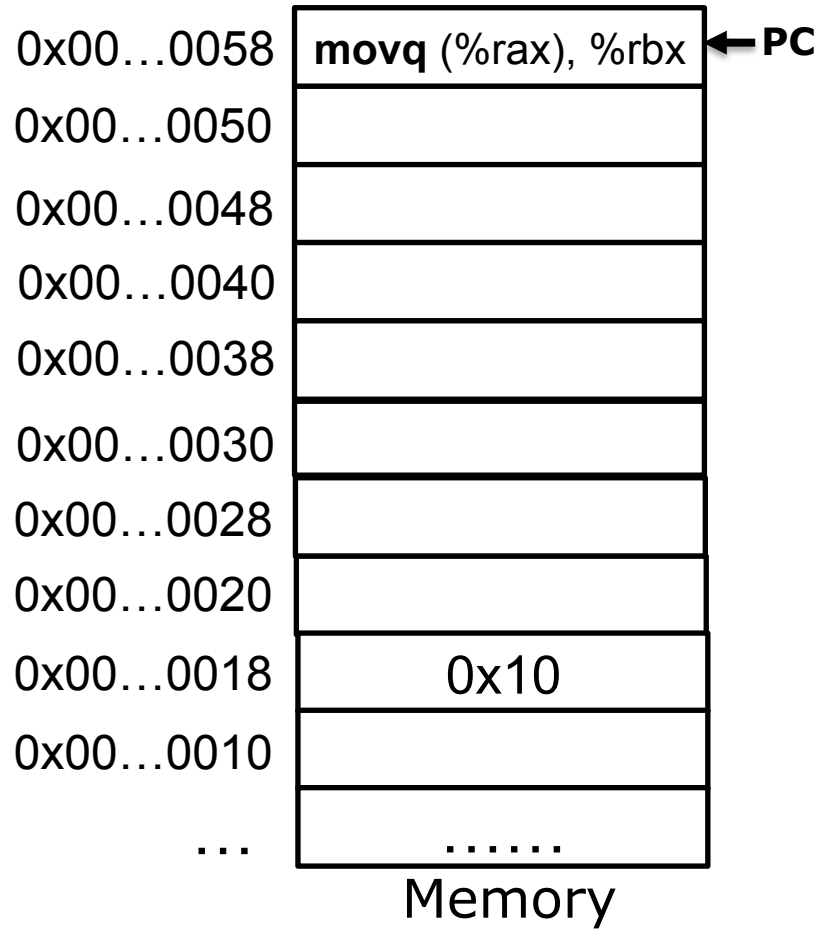
How to represent a “memory” operand?

Direct addressing: use registers to index the memory

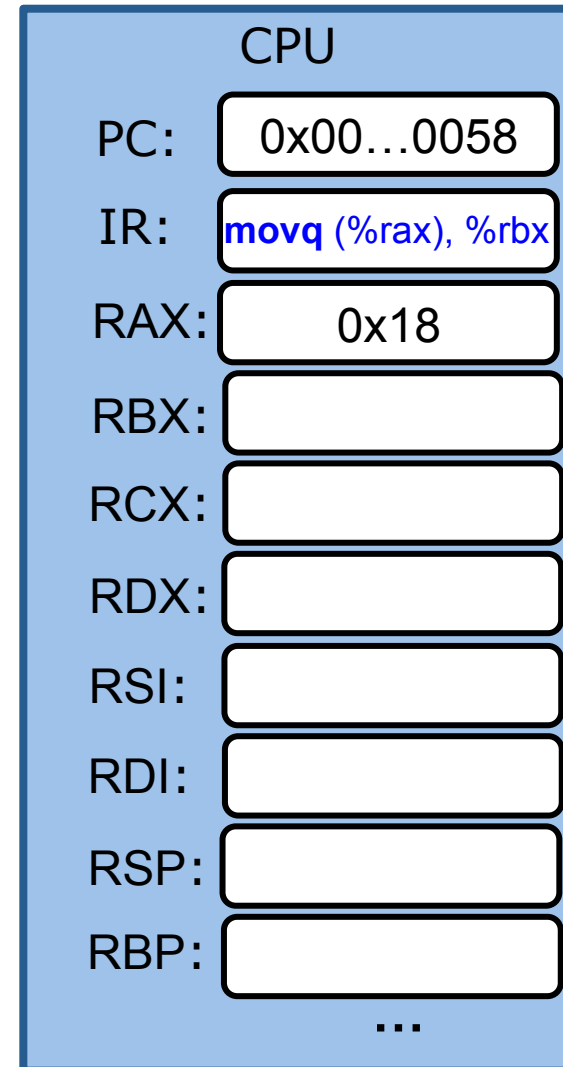
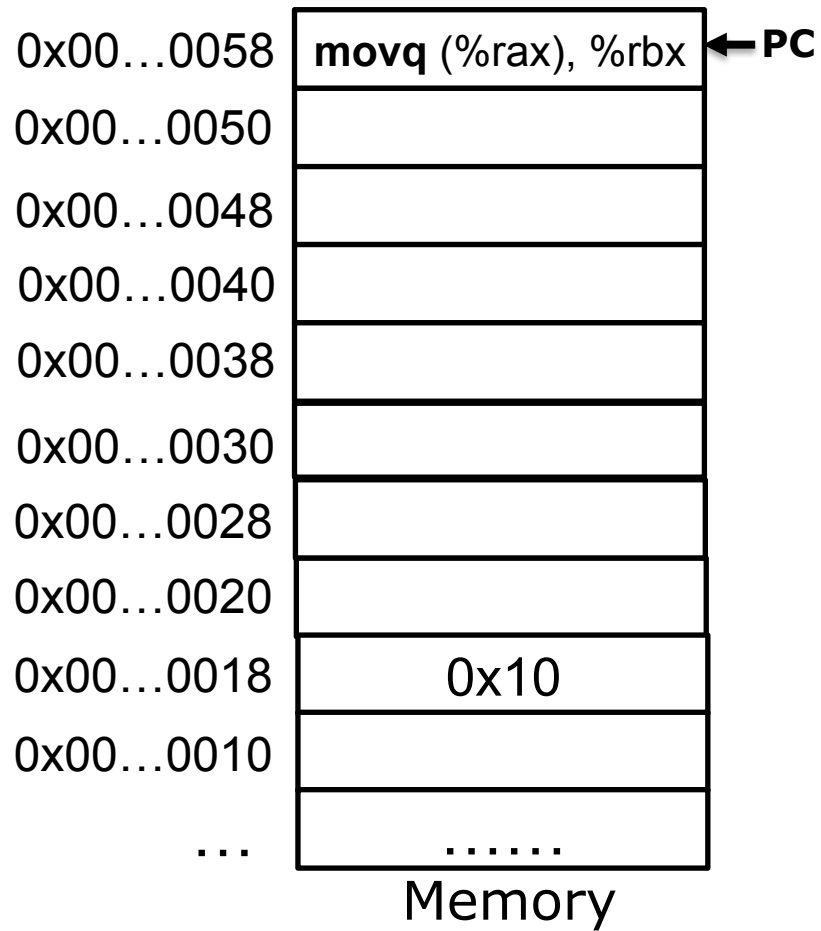
(Register)

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

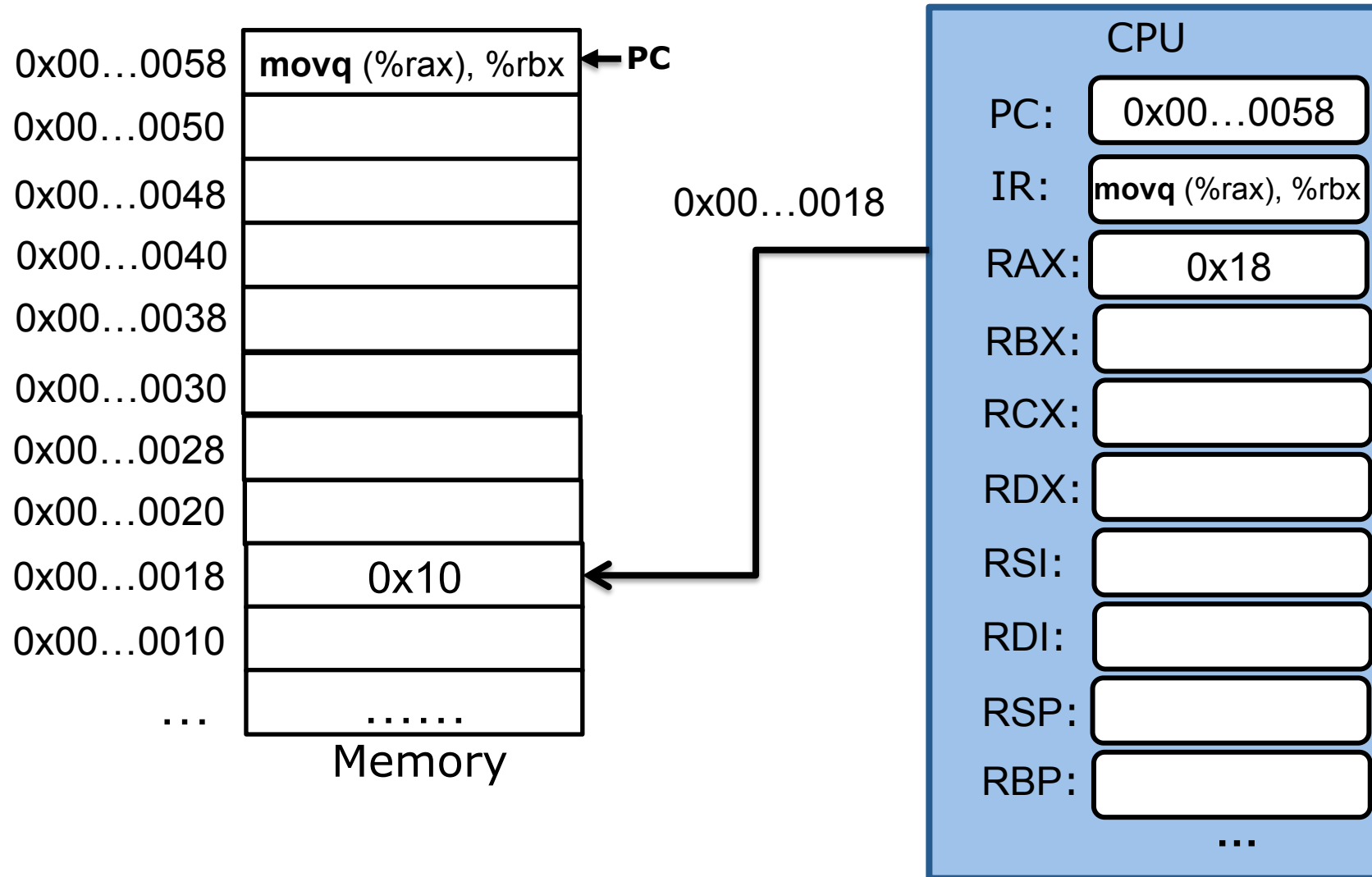
movq (%rax), %rbx



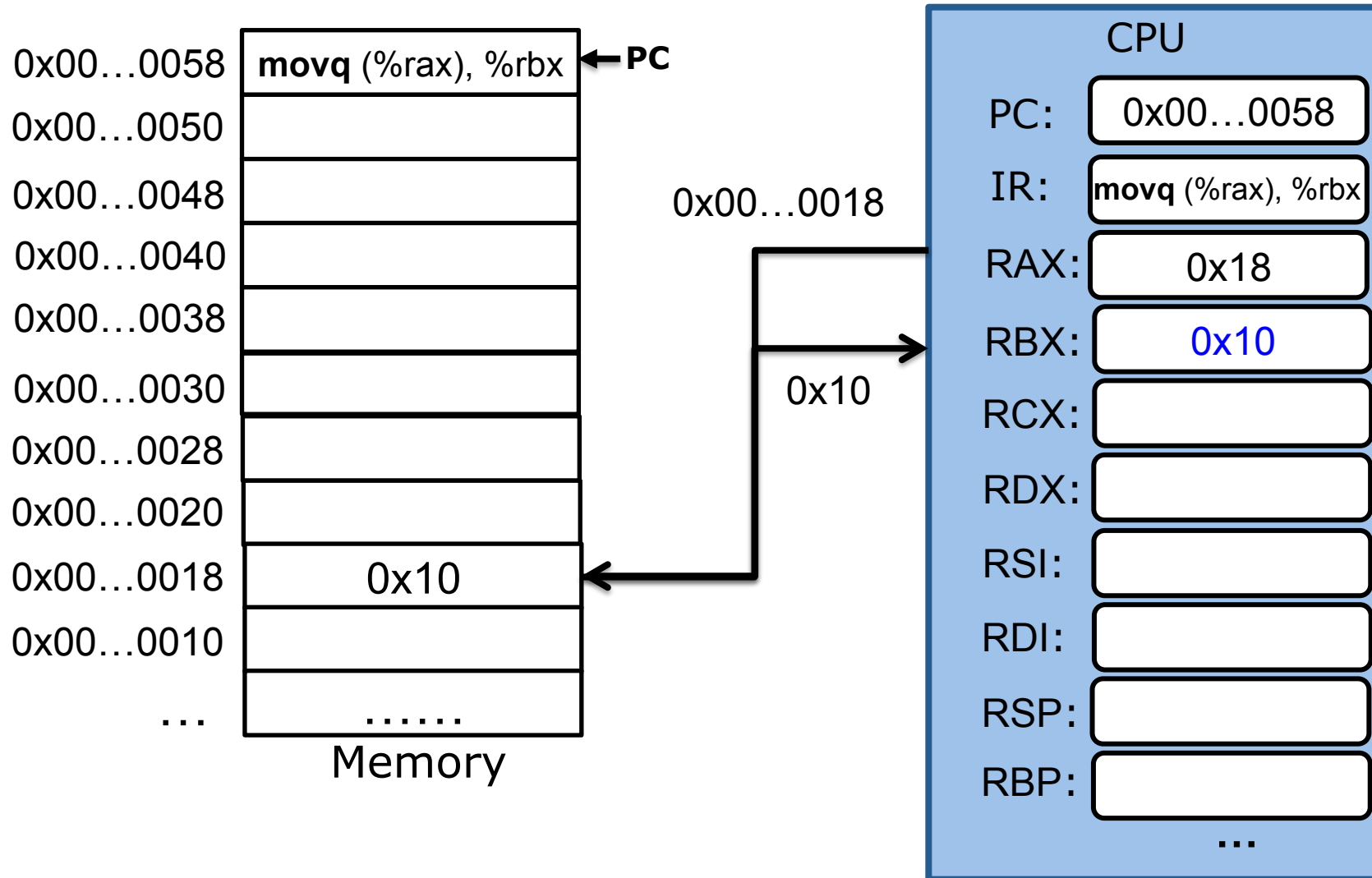
movq (%rax), %rbx



movq (%rax), %rbx



movq (%rax), %rbx



swap function

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



GCC -S -O3 swap.c

swap function

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

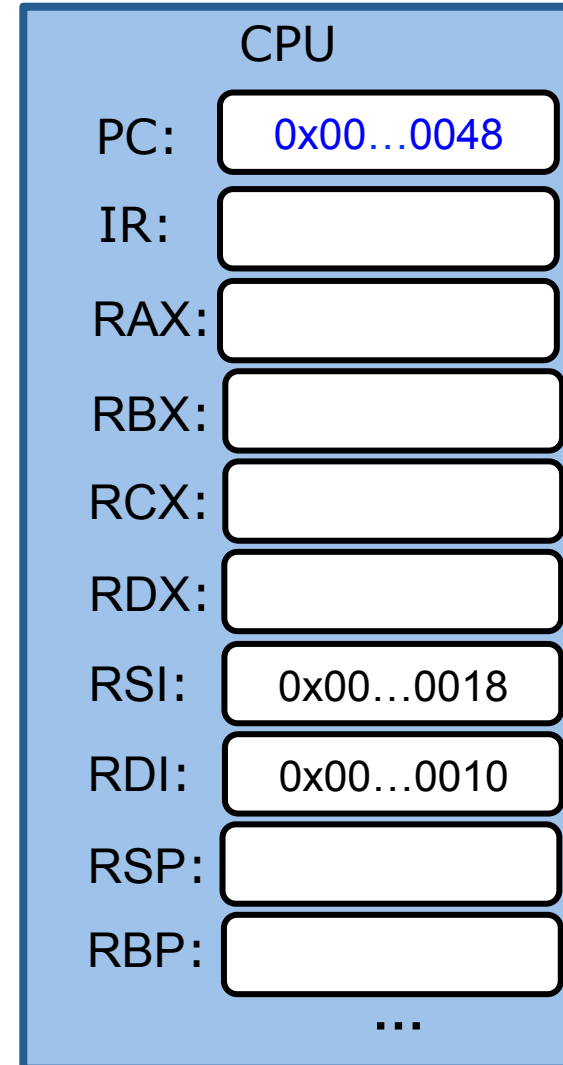
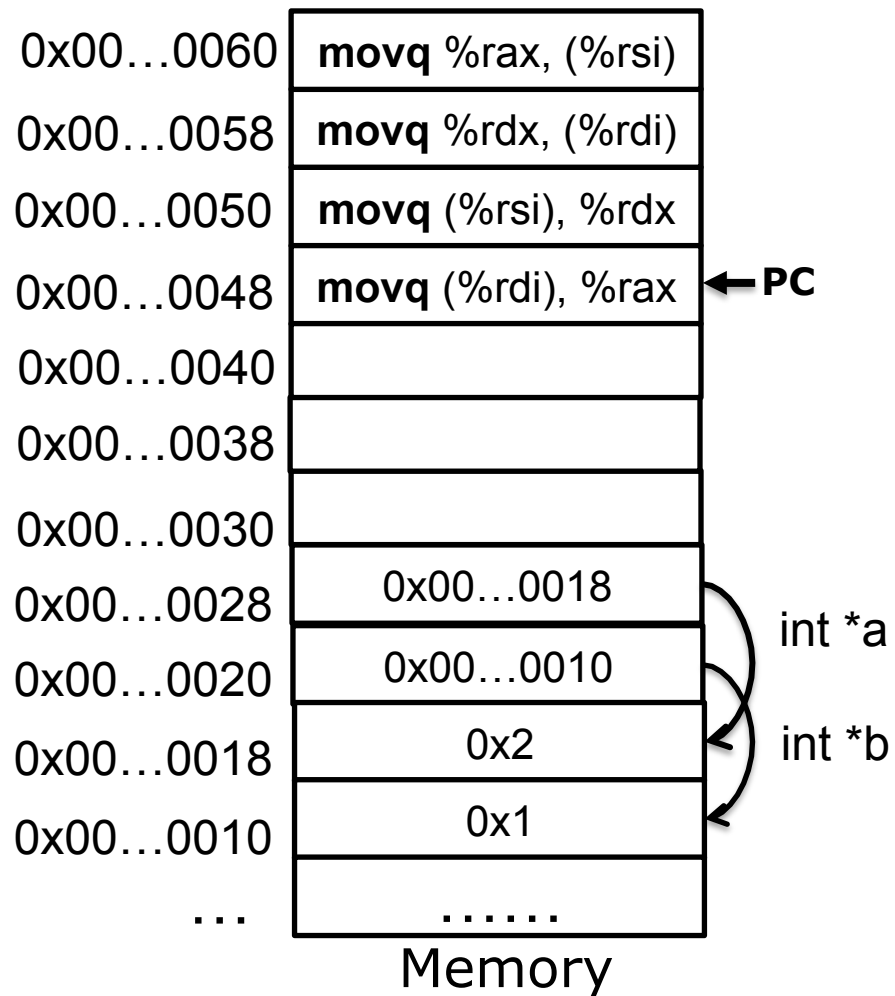


GCC -S -O3 swap.c

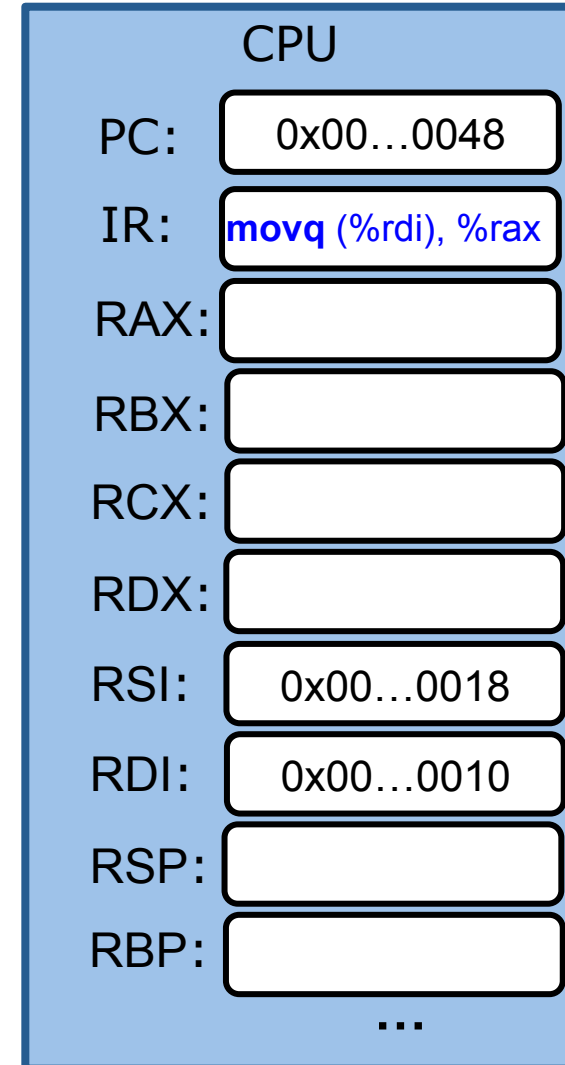
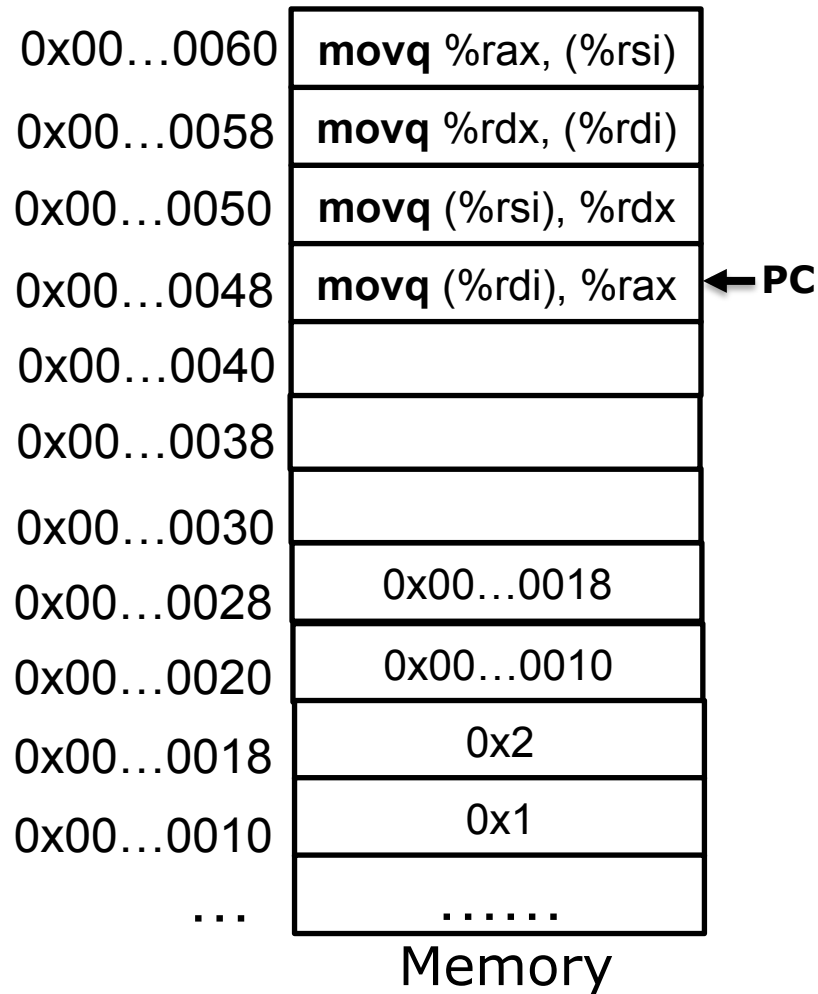
swap:

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

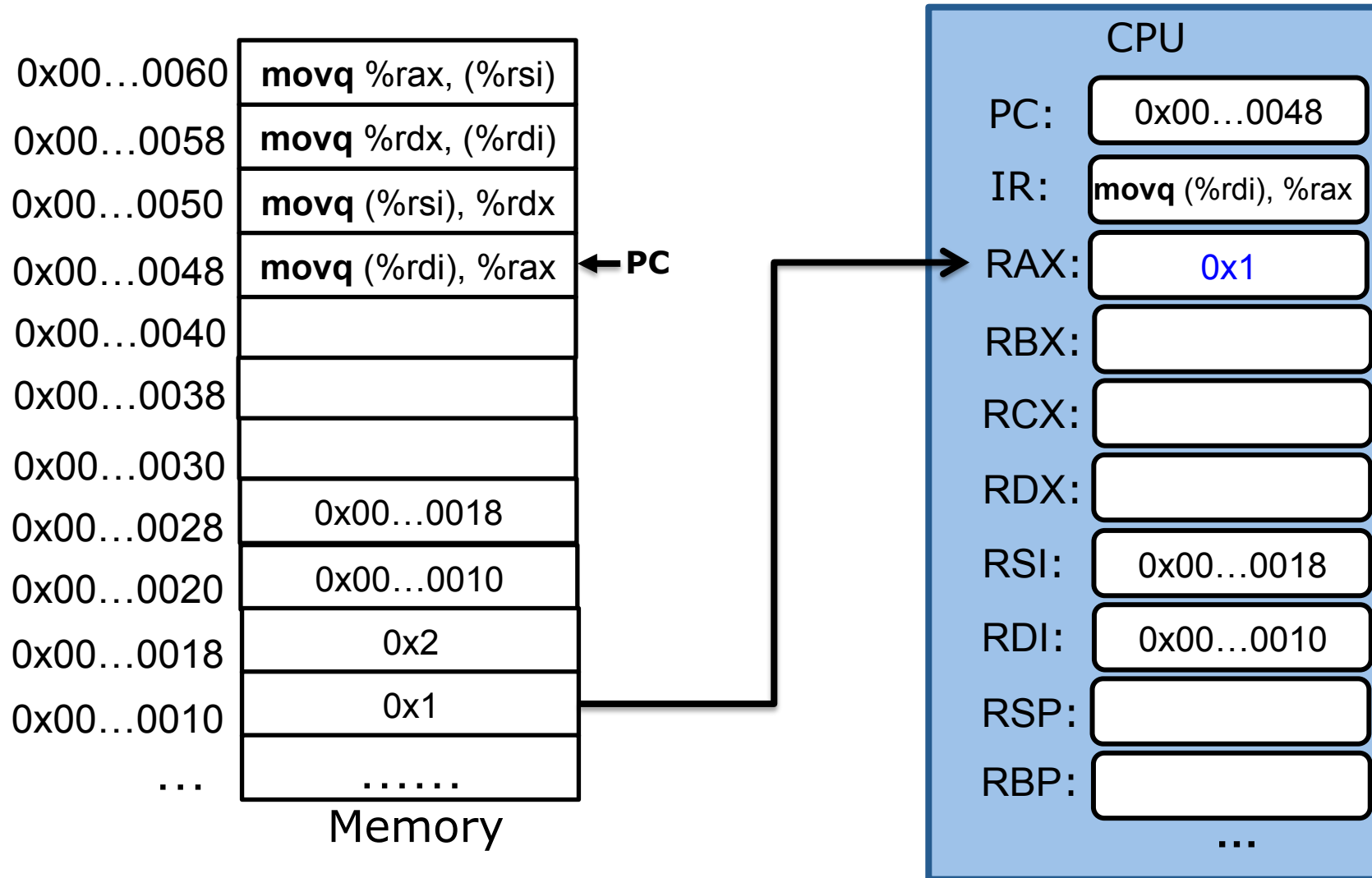
swap func



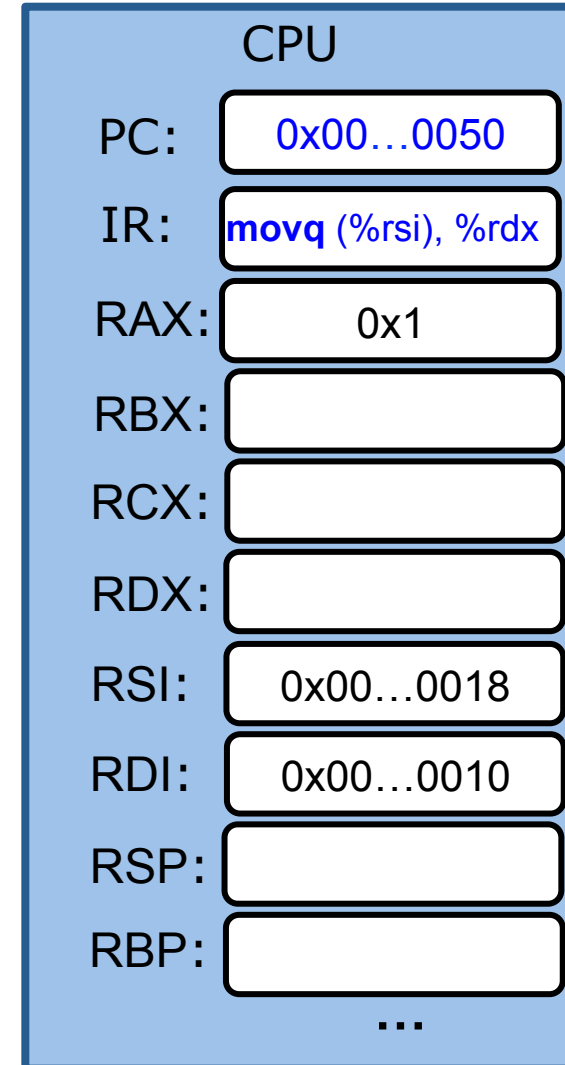
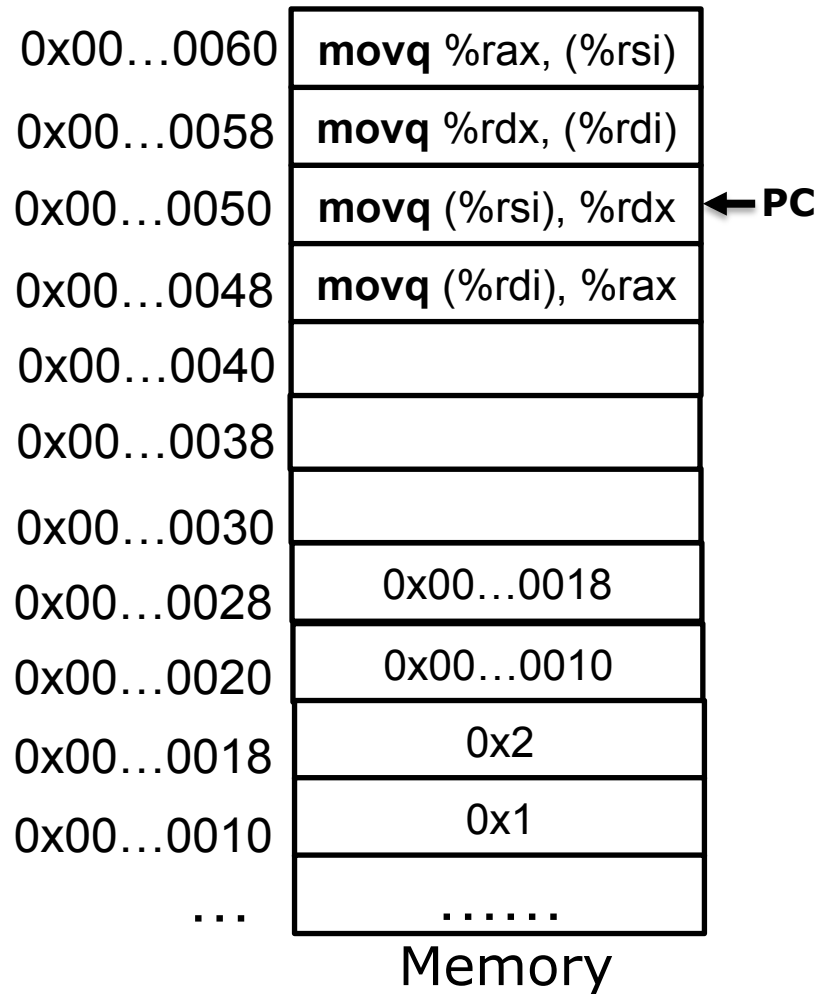
swap func



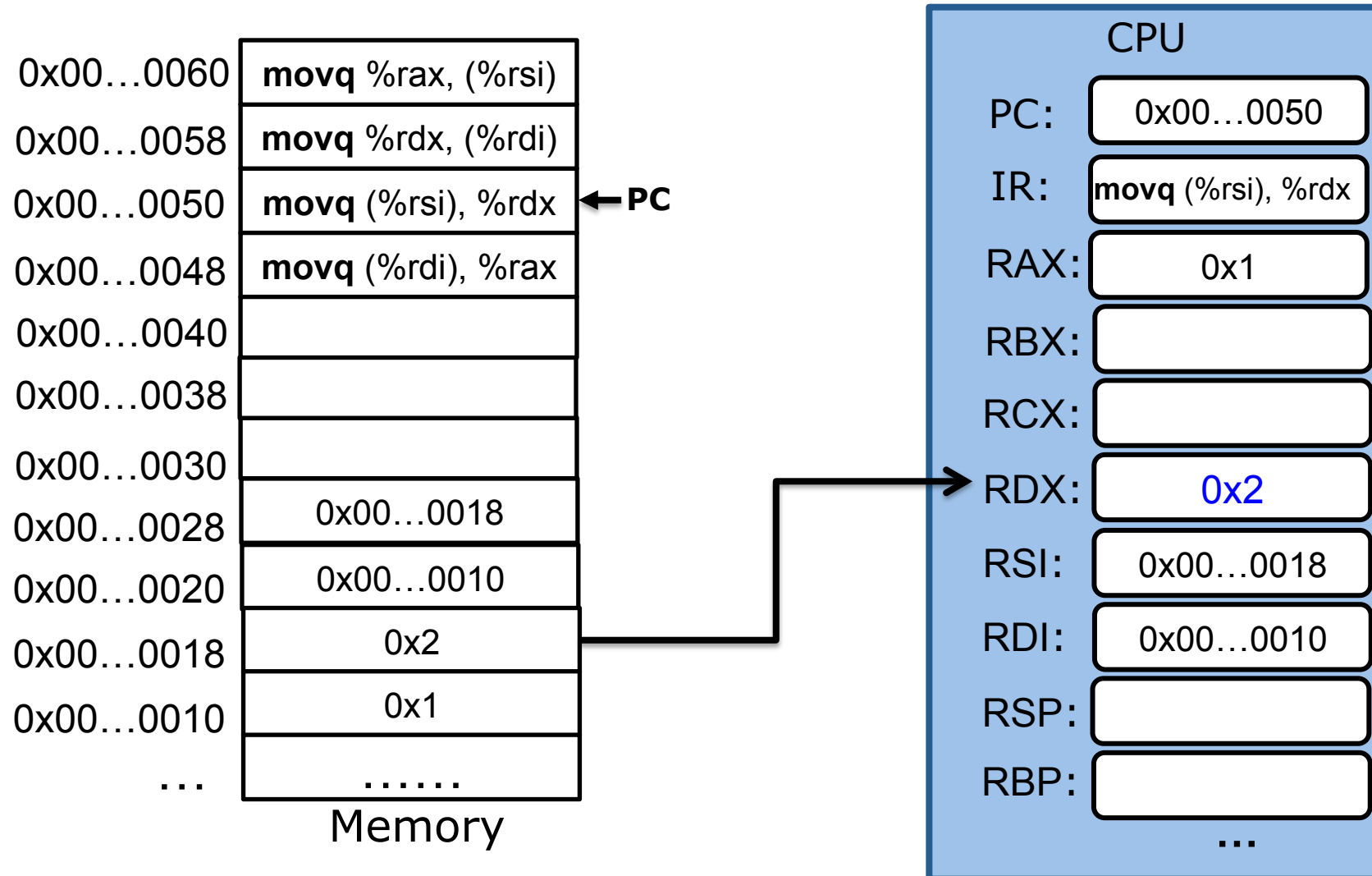
swap func



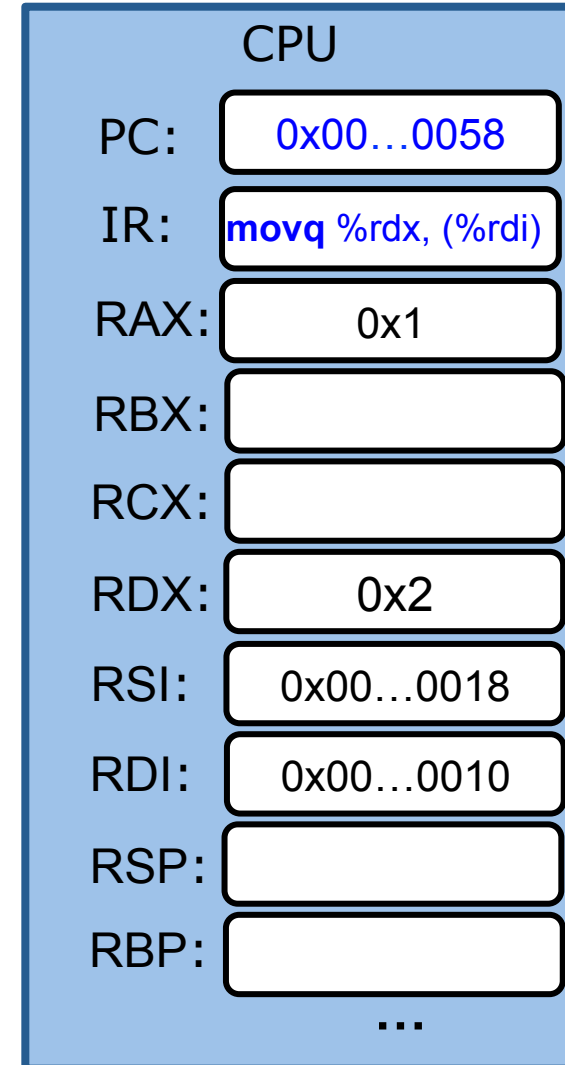
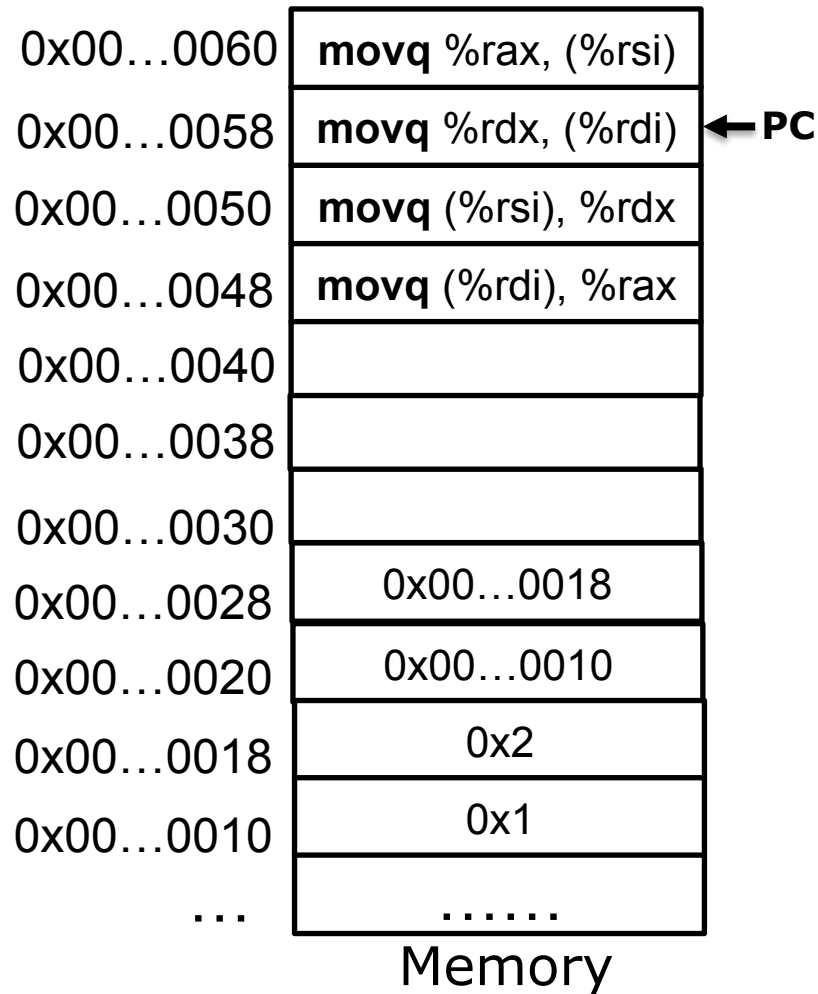
swap func



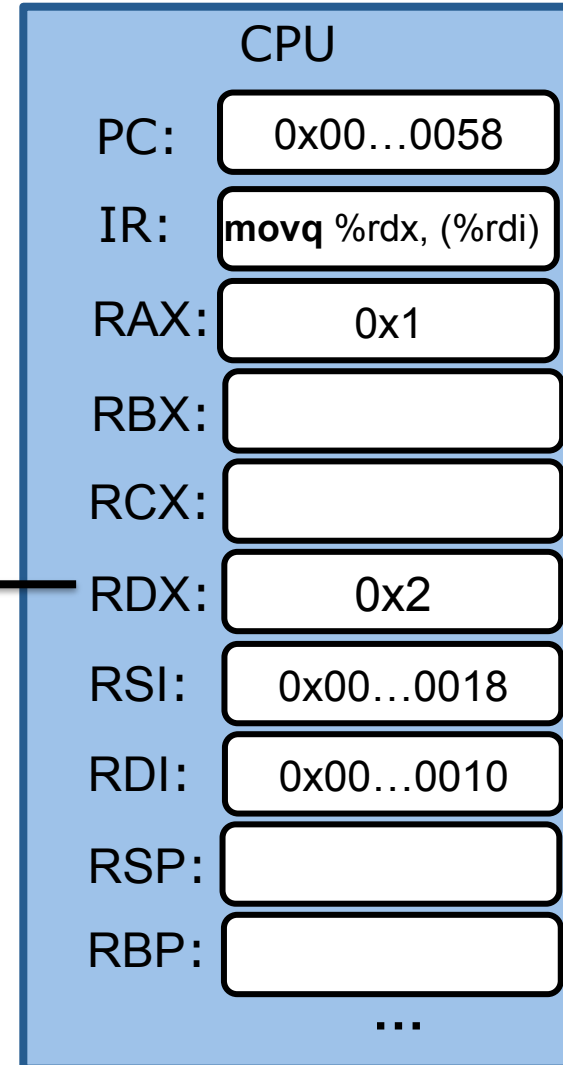
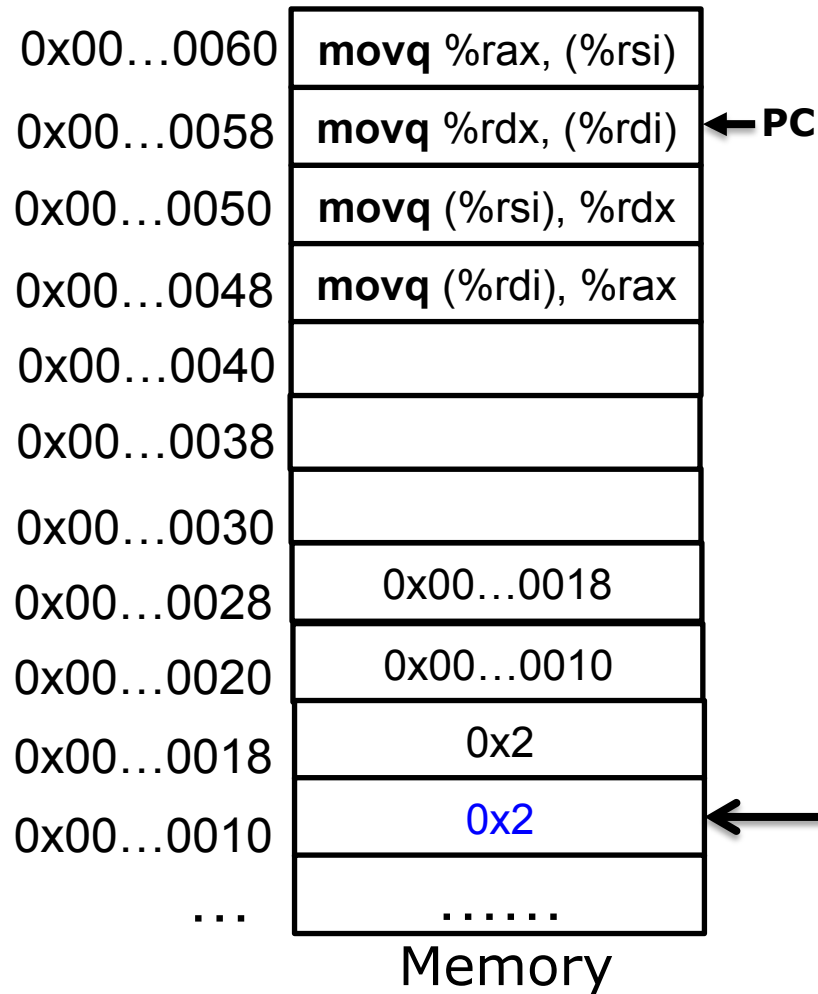
swap func



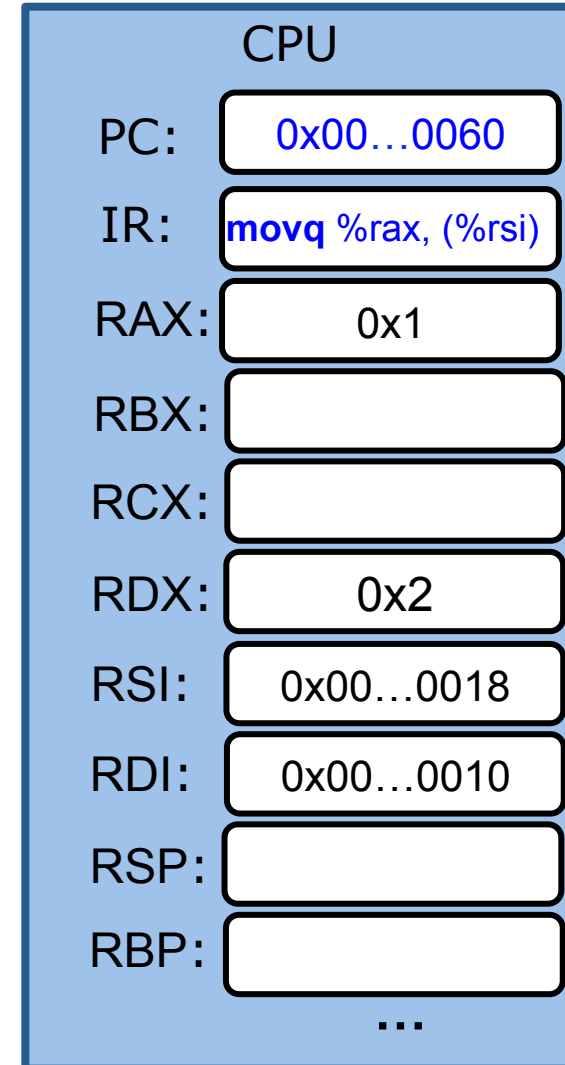
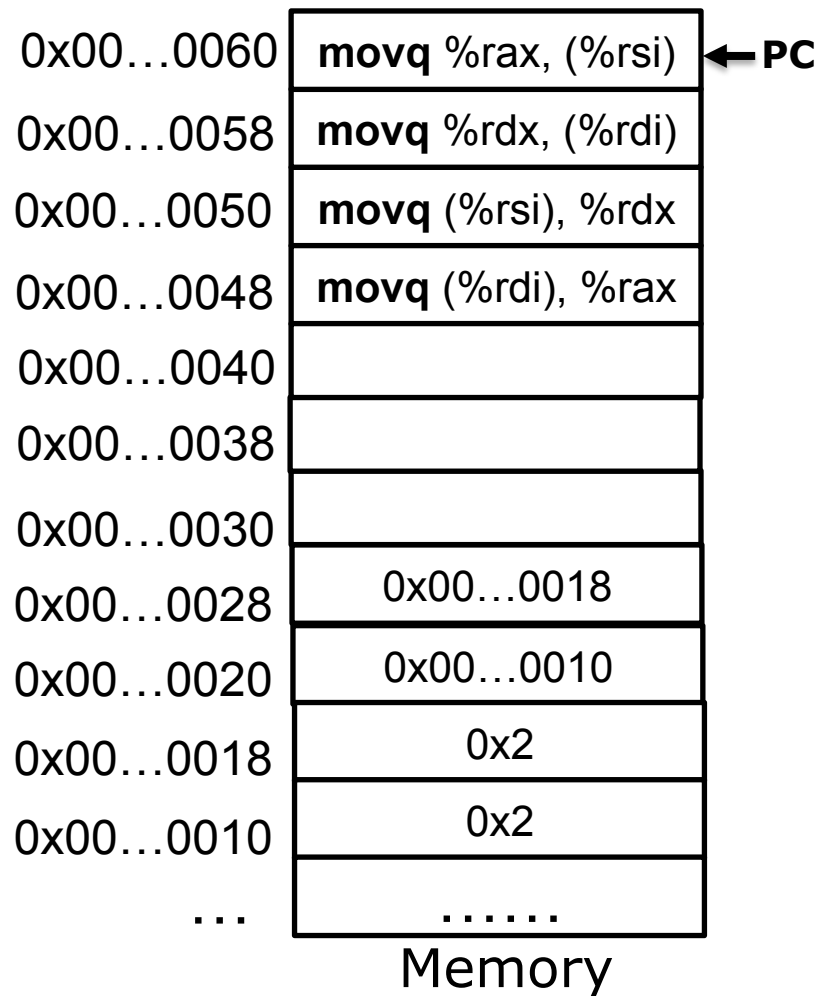
swap func



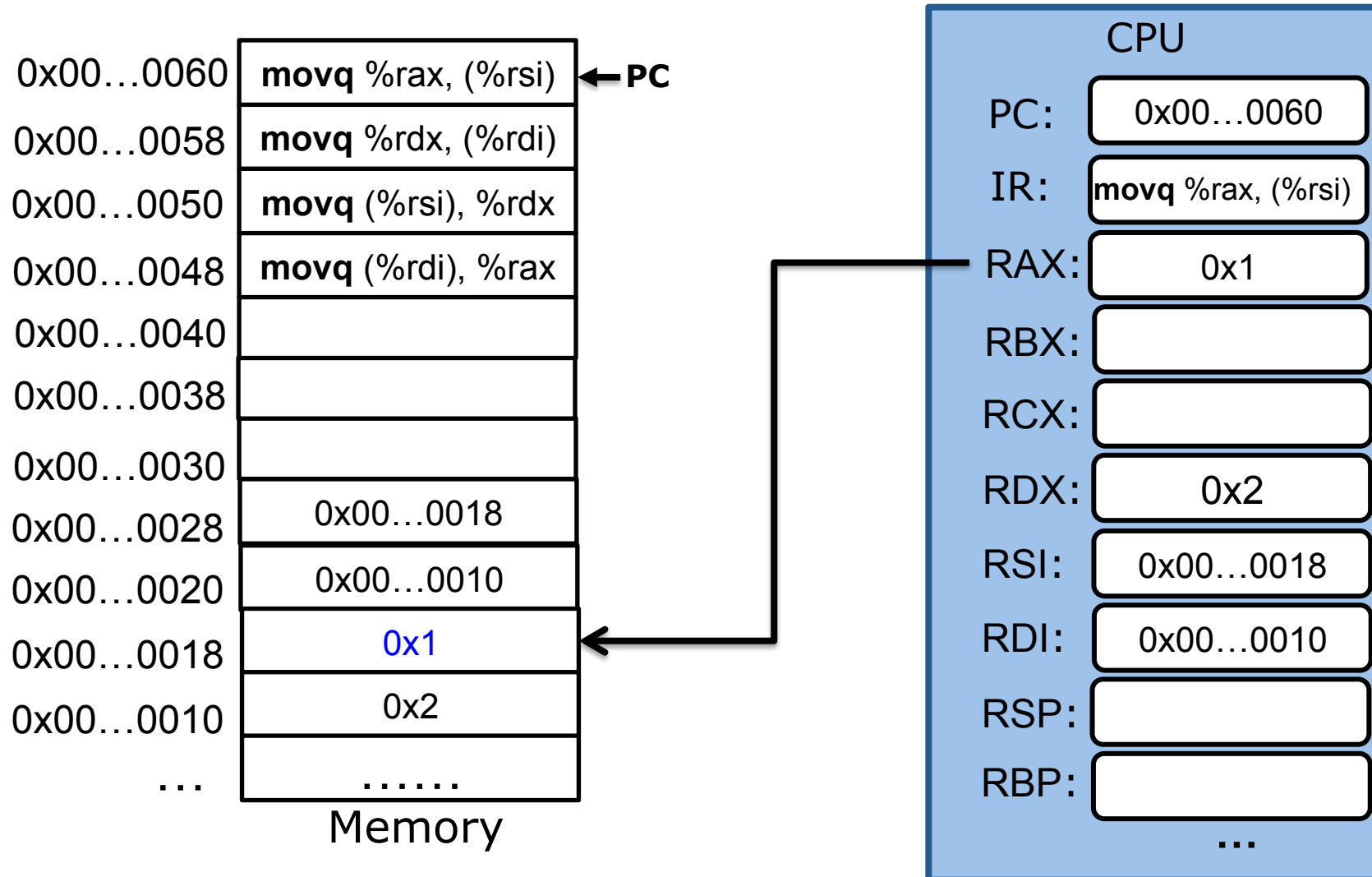
swap func



swap func



swap func



Limitation of direct addressing

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

Issue

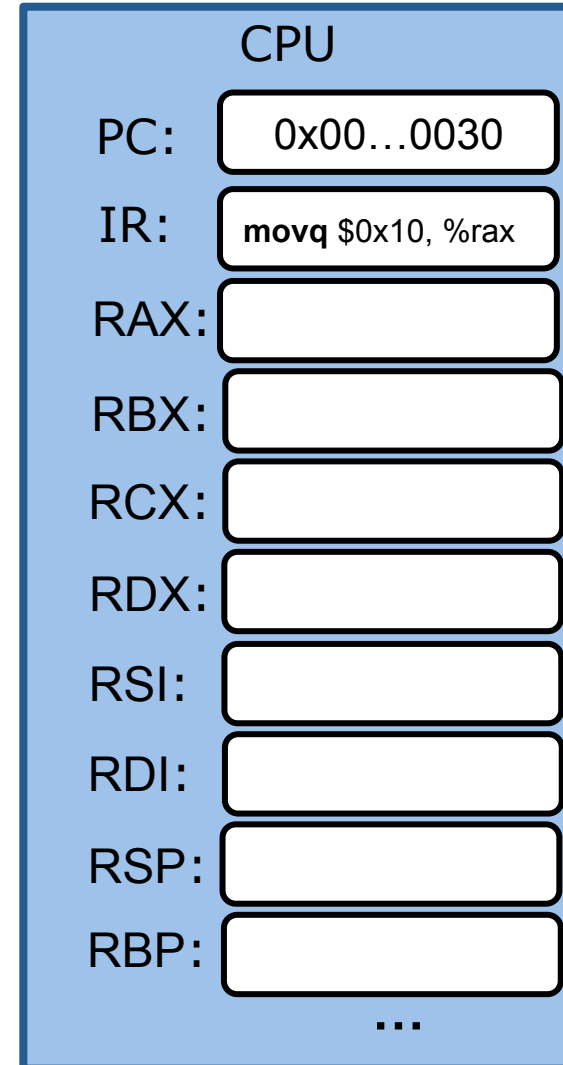
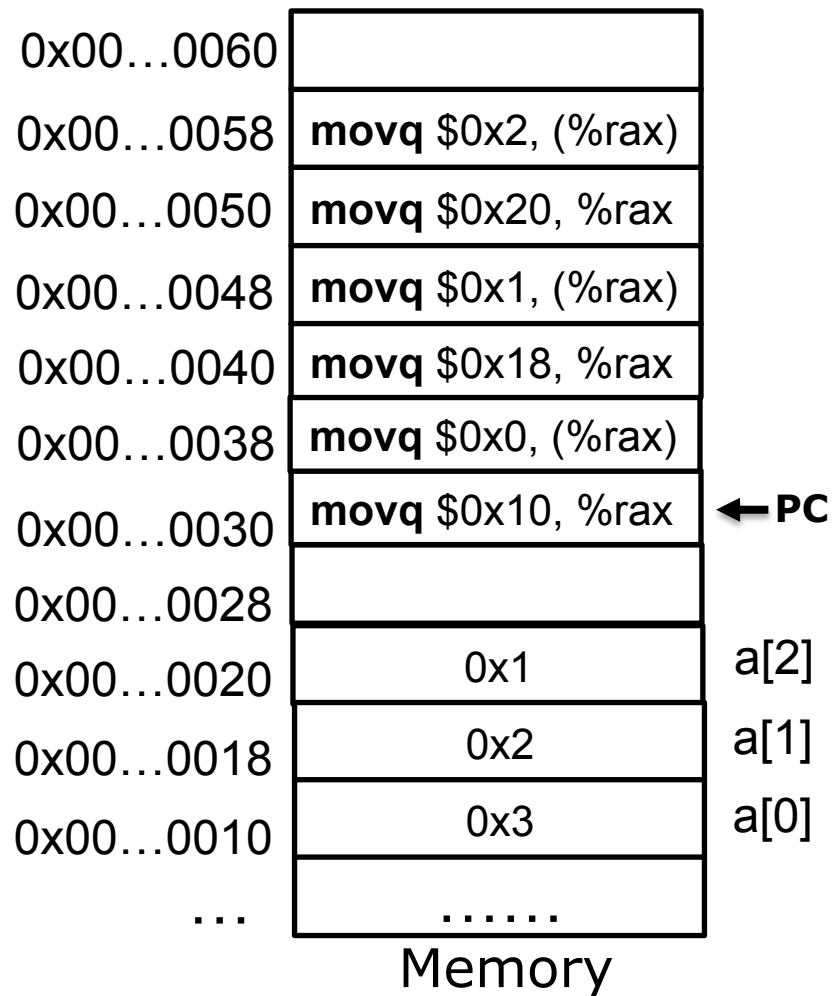
Issue: the address must be calculated and stored in the register before each memory access.

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

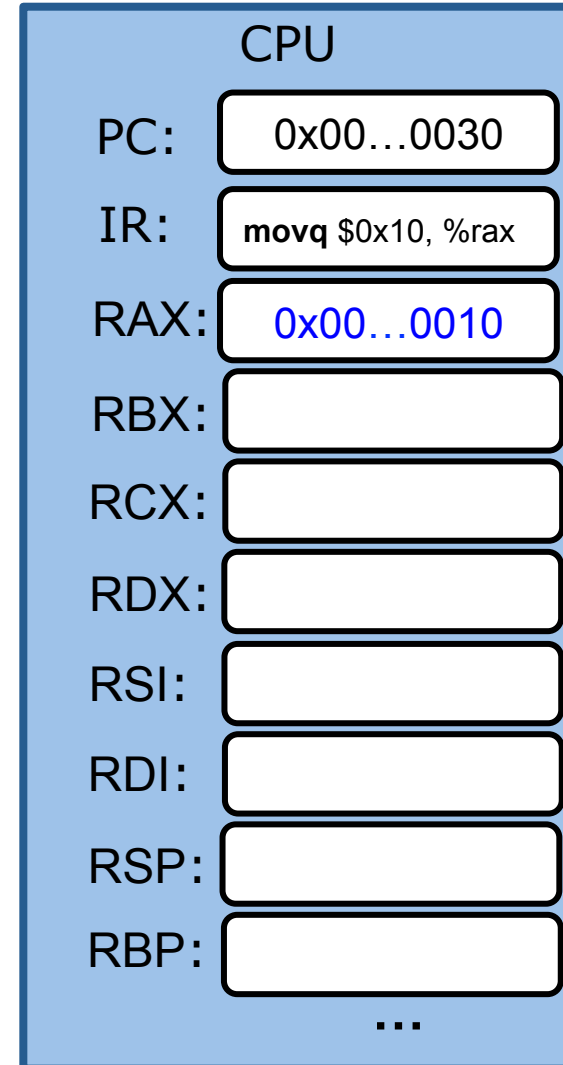
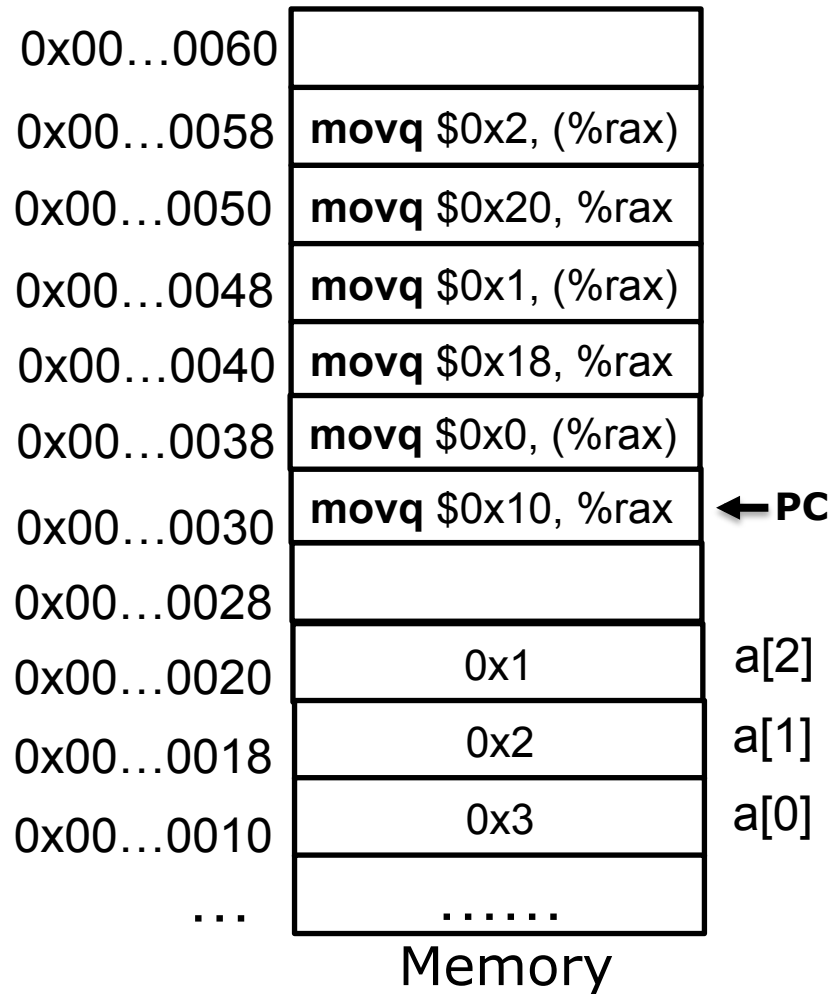


```
long a[] = {1, 2, 3};  
for(int i = 0; i < 3; i++) {  
    1. put &a[i] into reg  
    2. mov $i, (reg)  
}
```

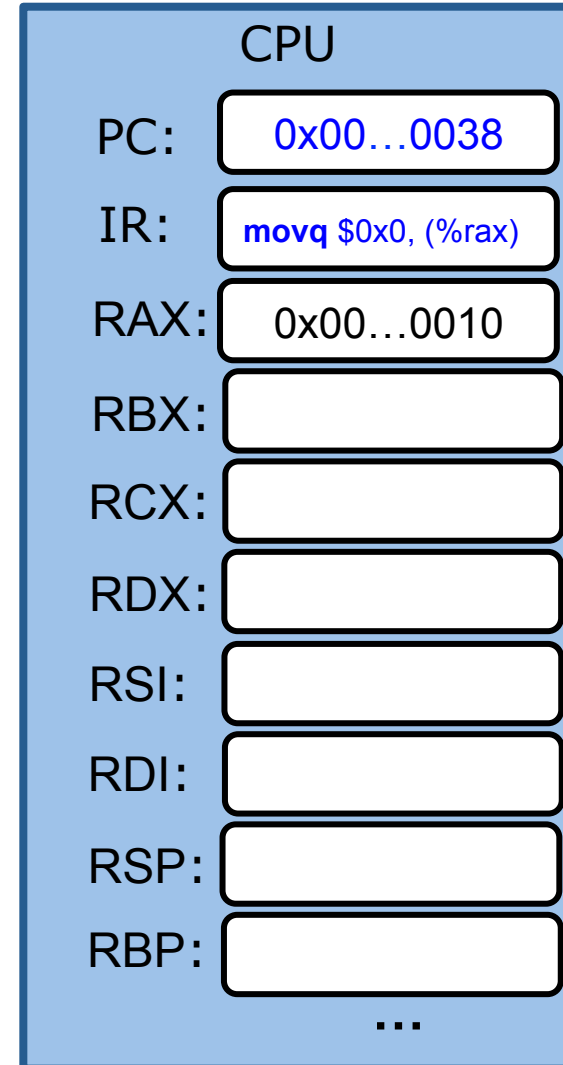
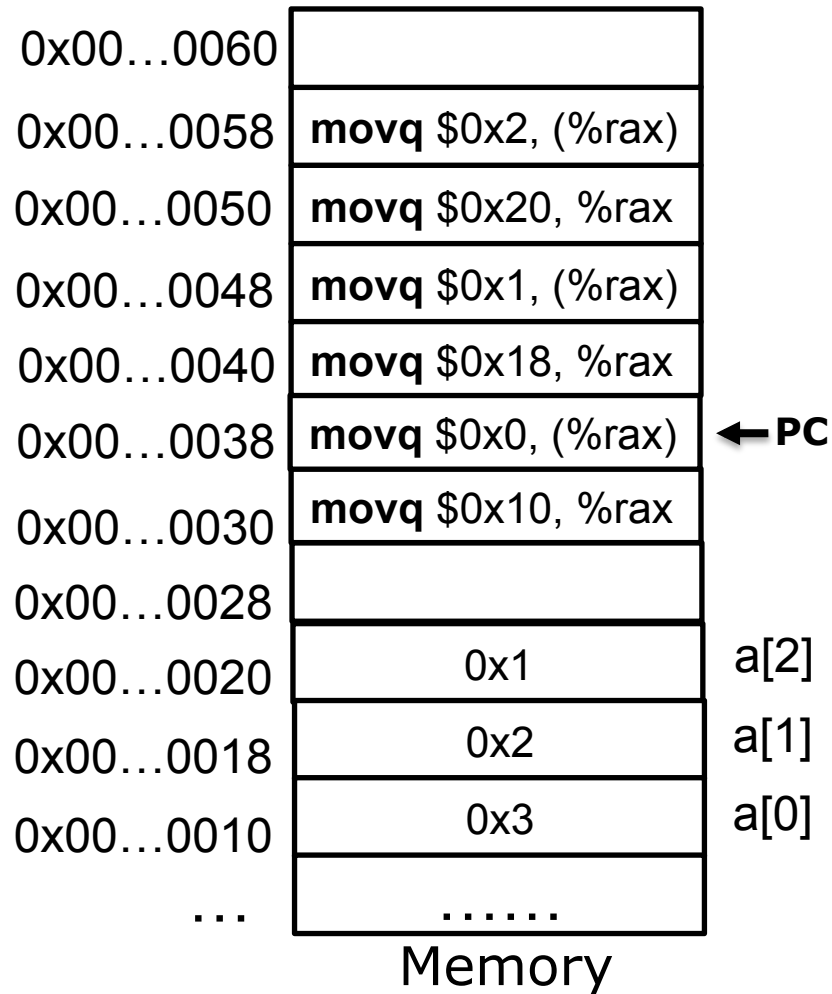
Example



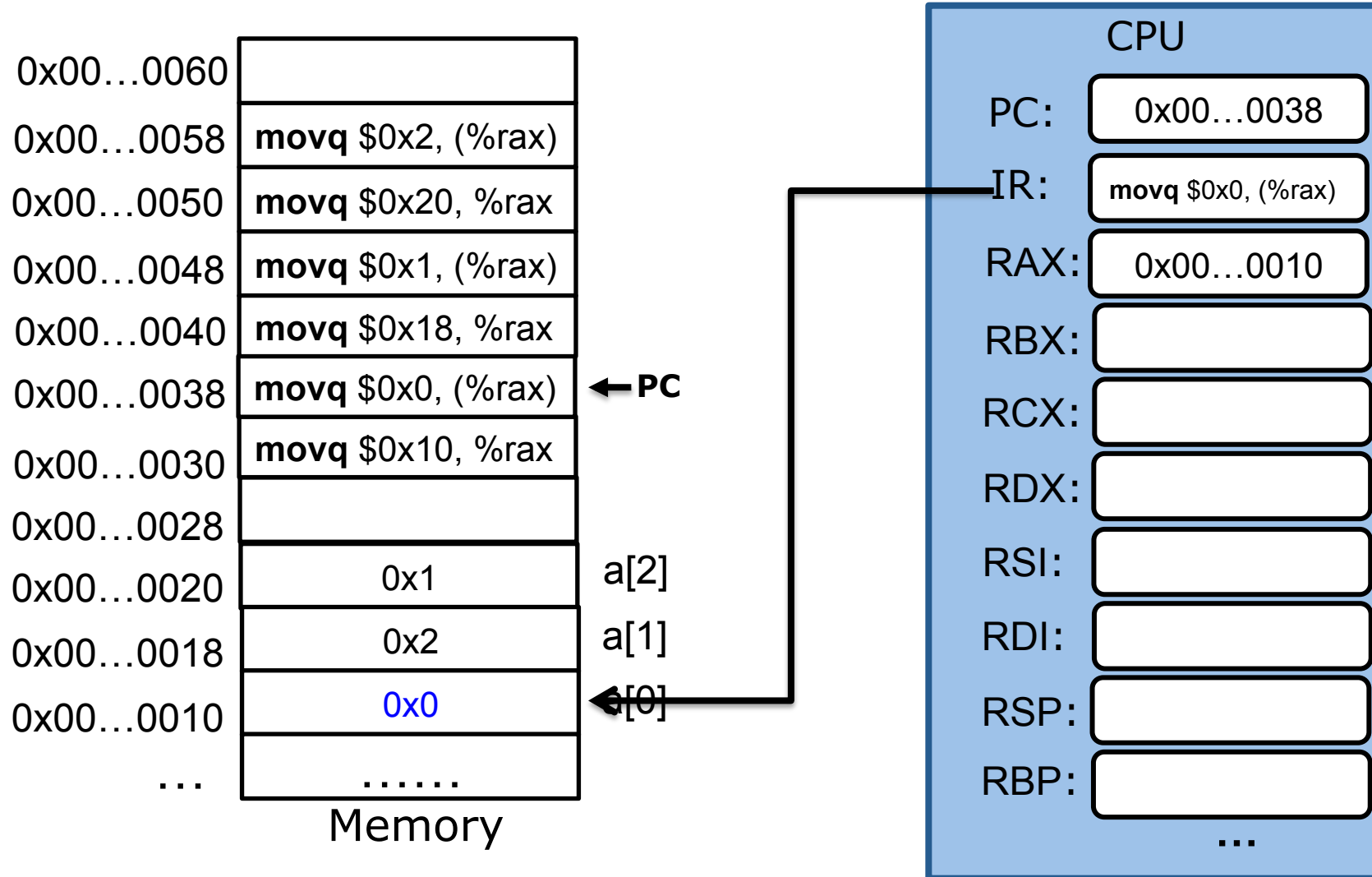
Example



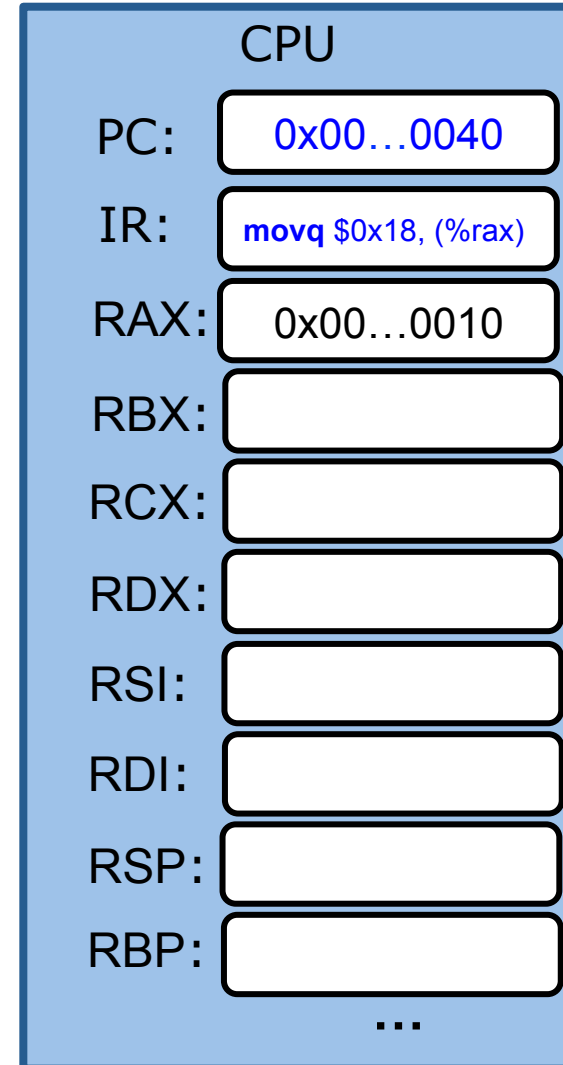
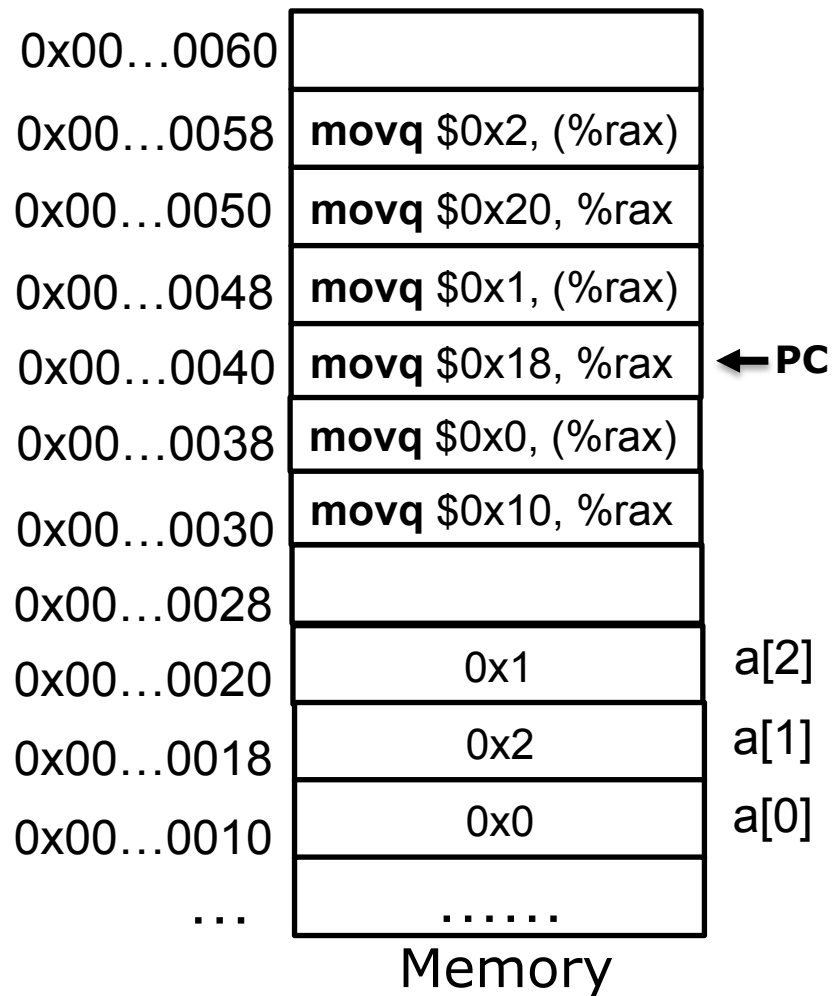
Example



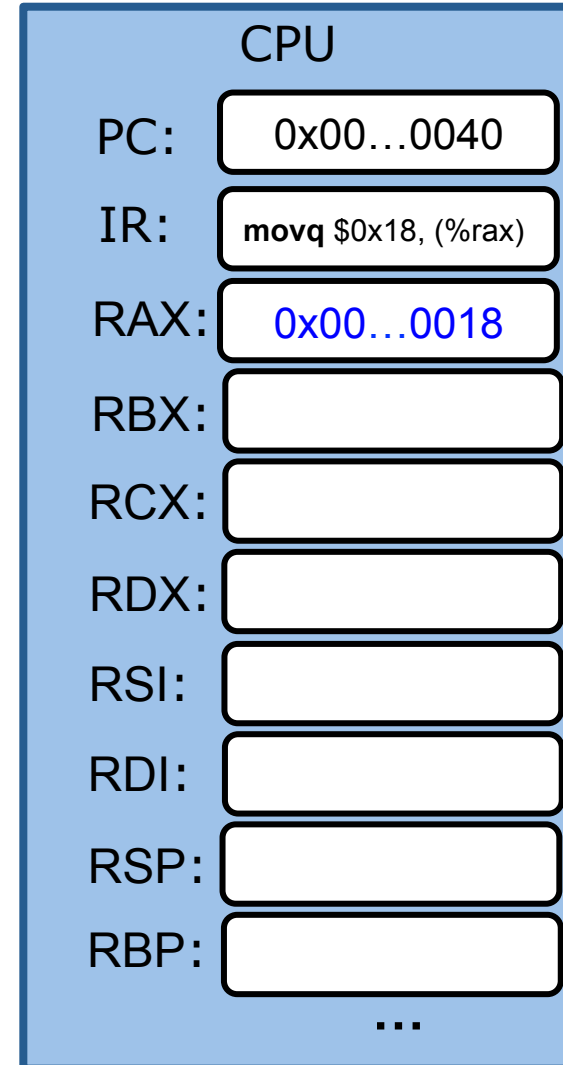
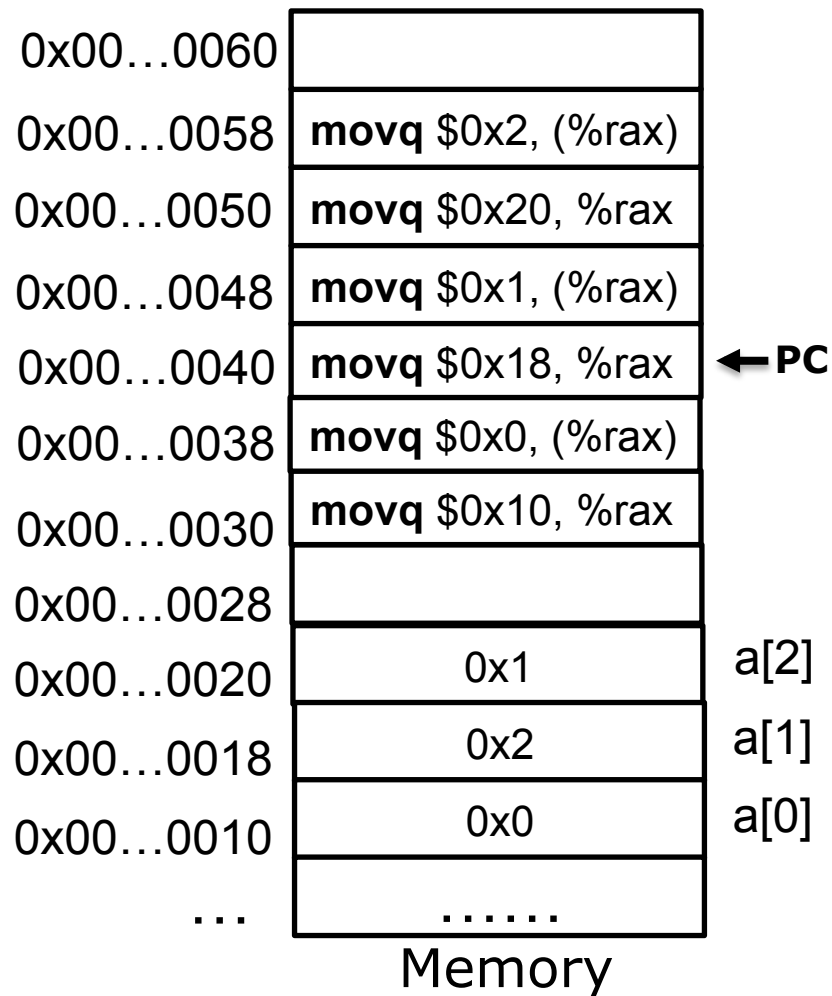
Example



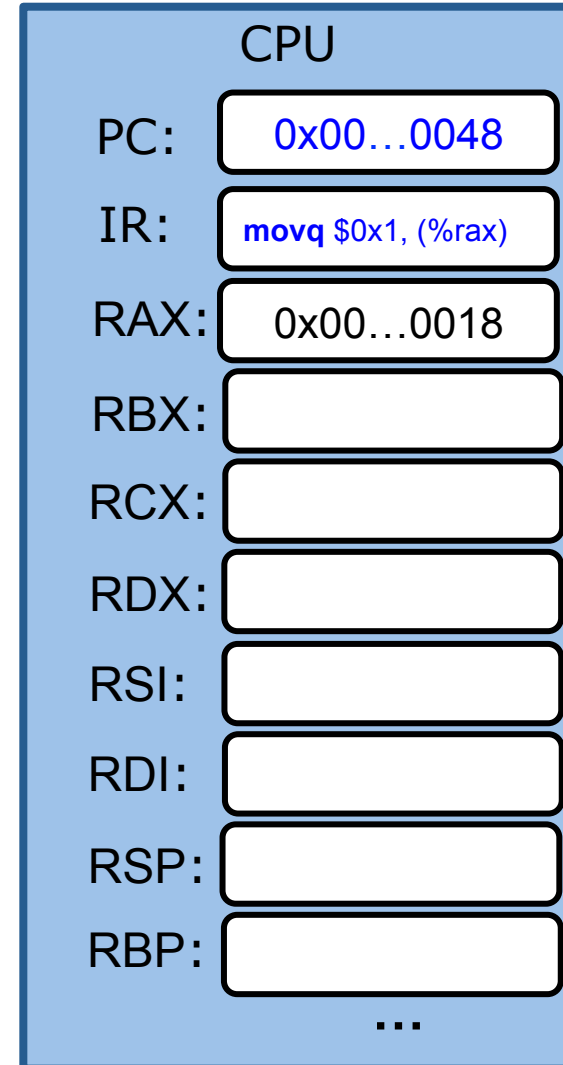
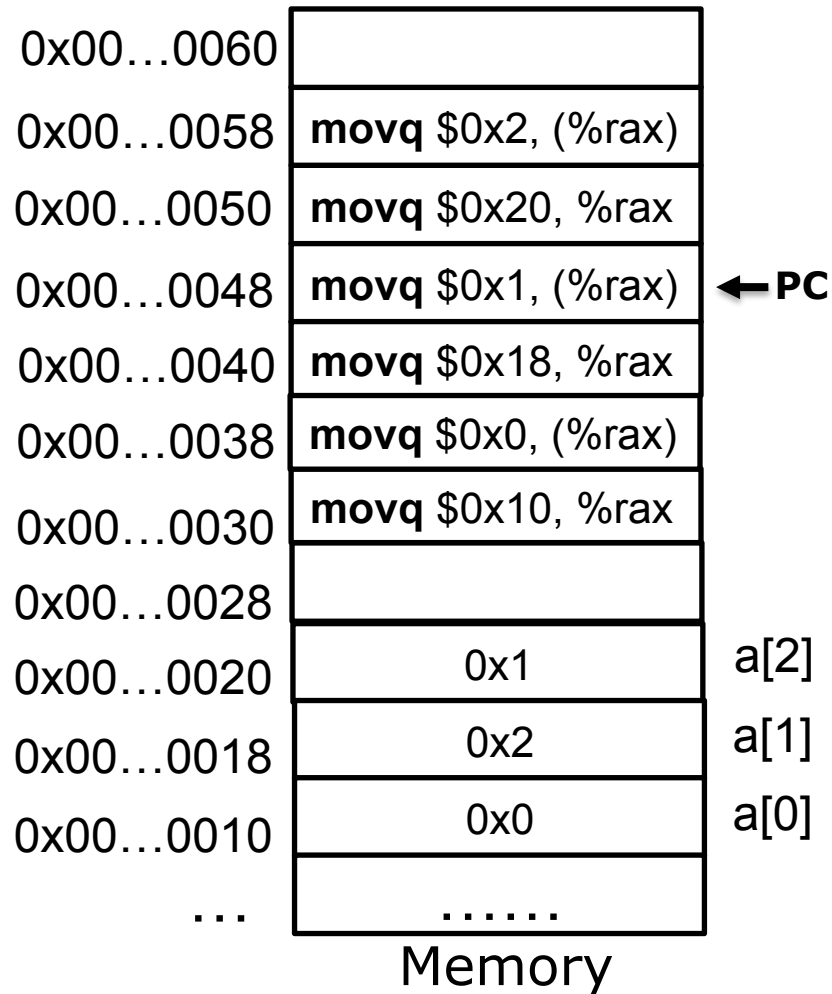
Example



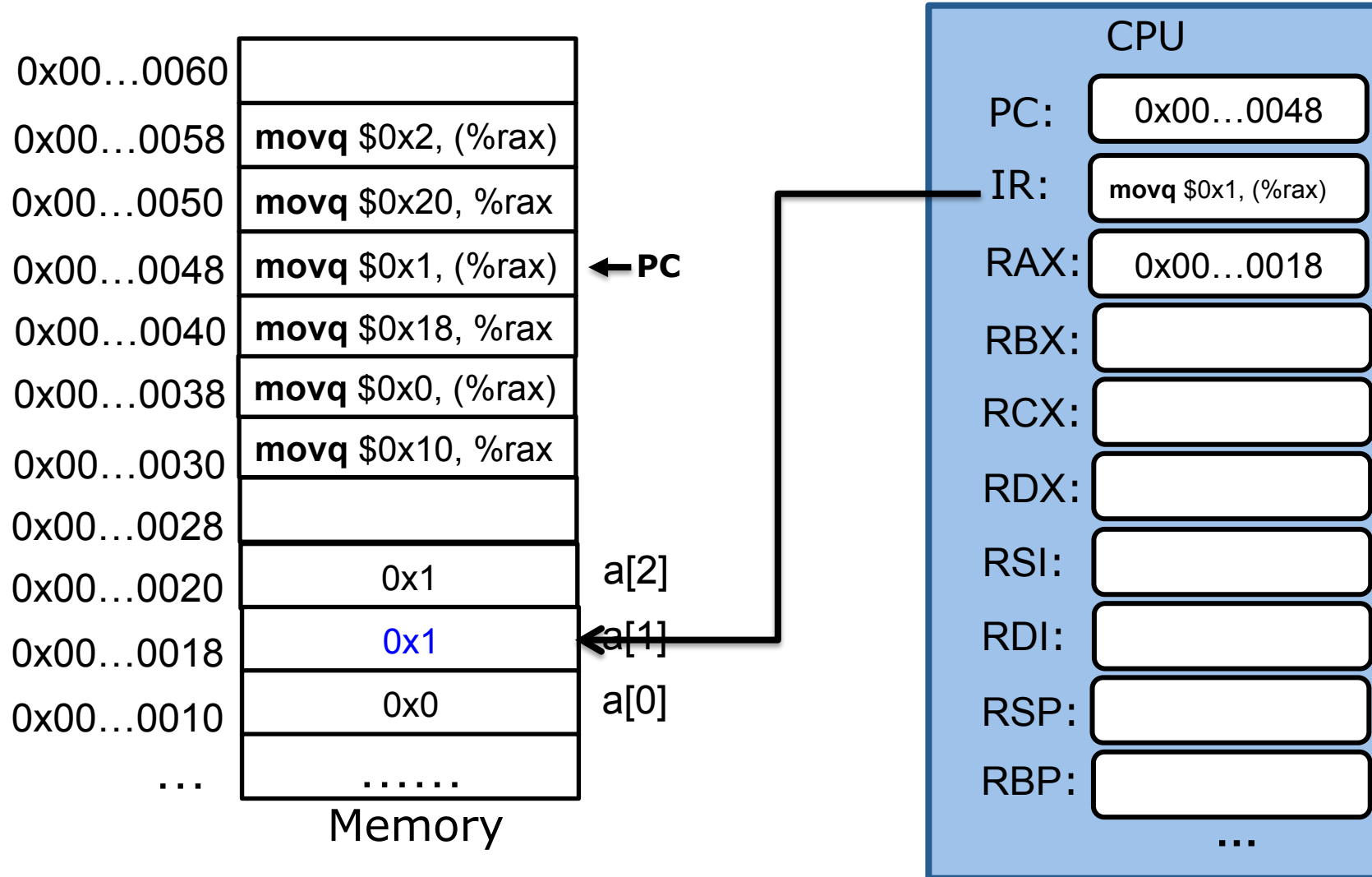
Example



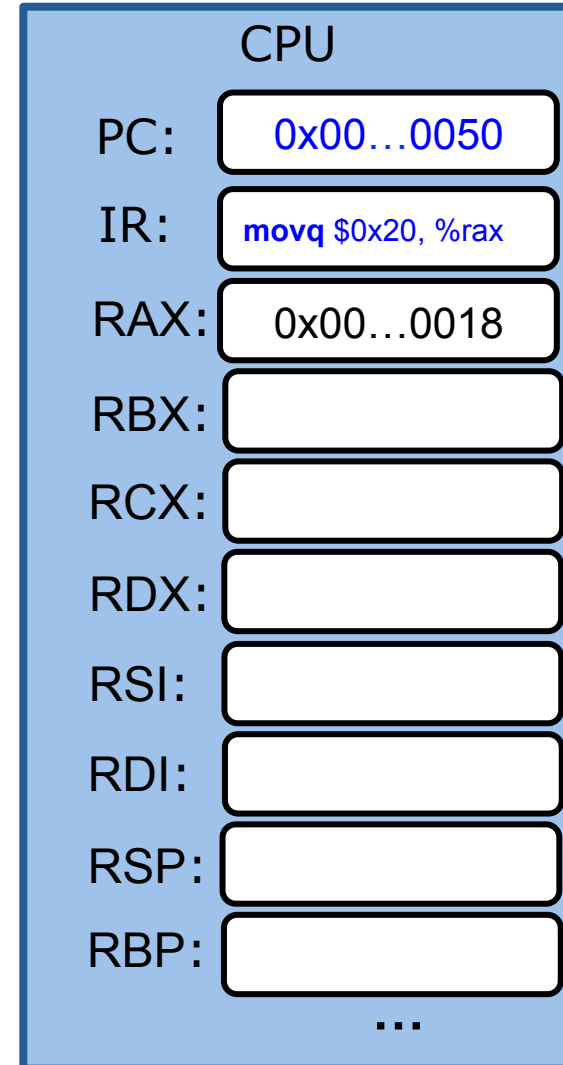
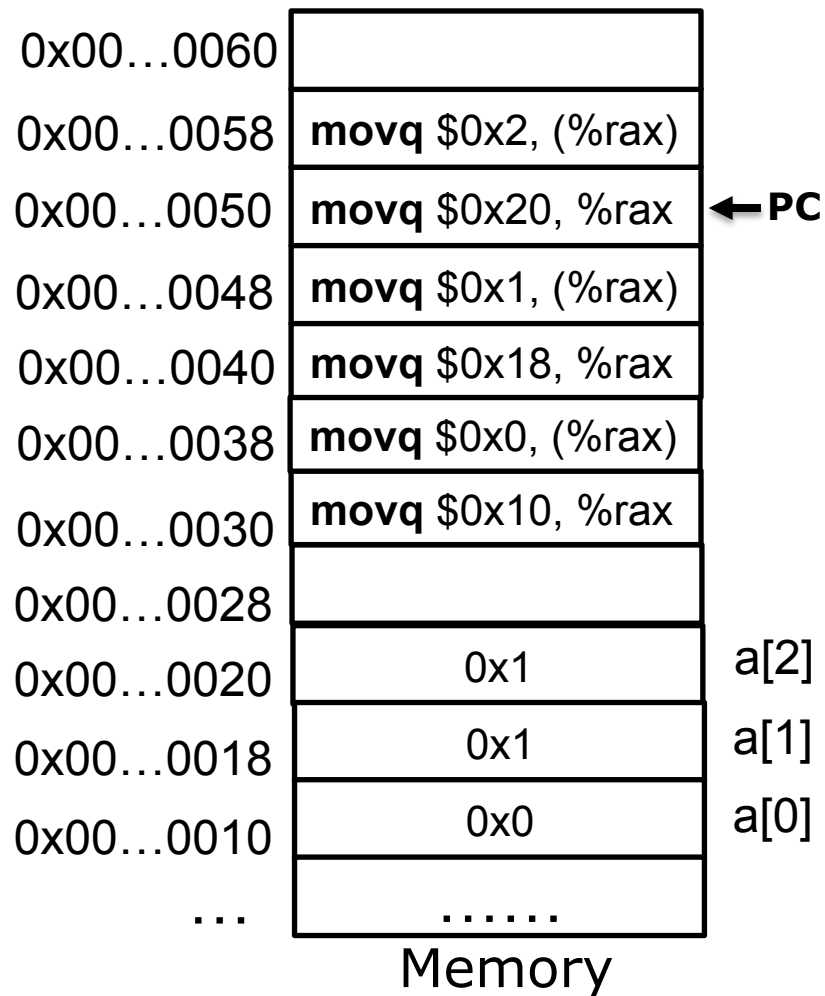
Example



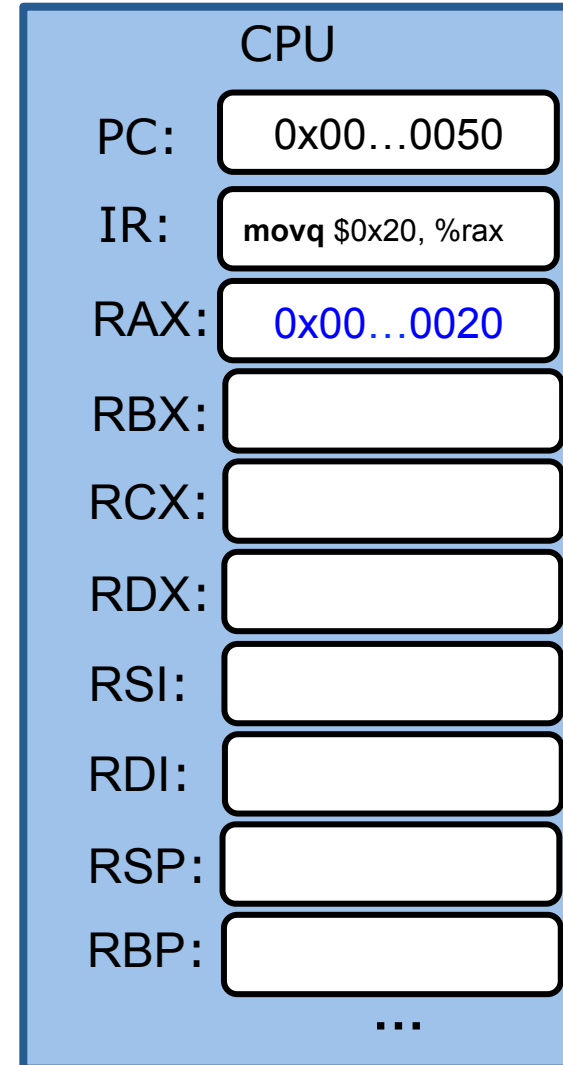
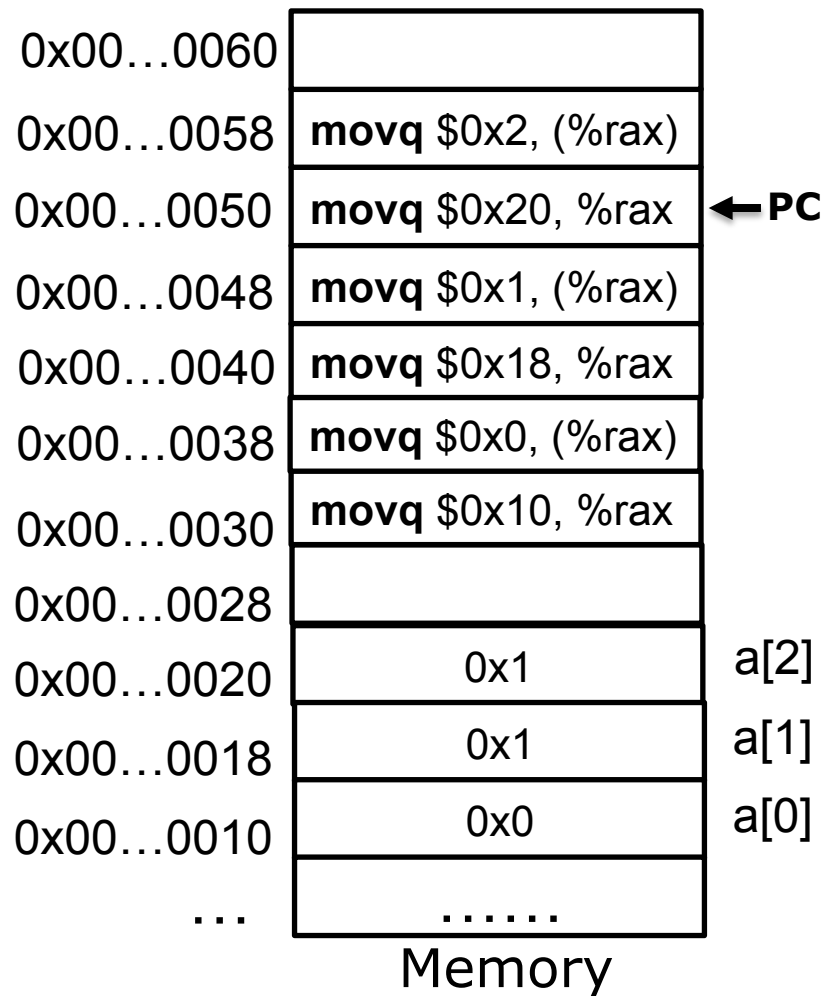
Example



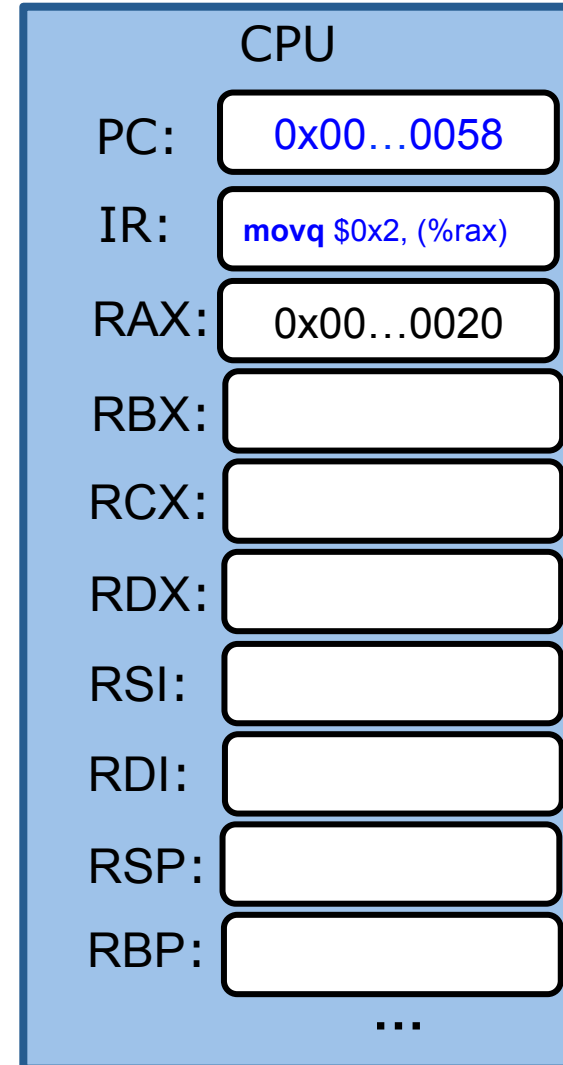
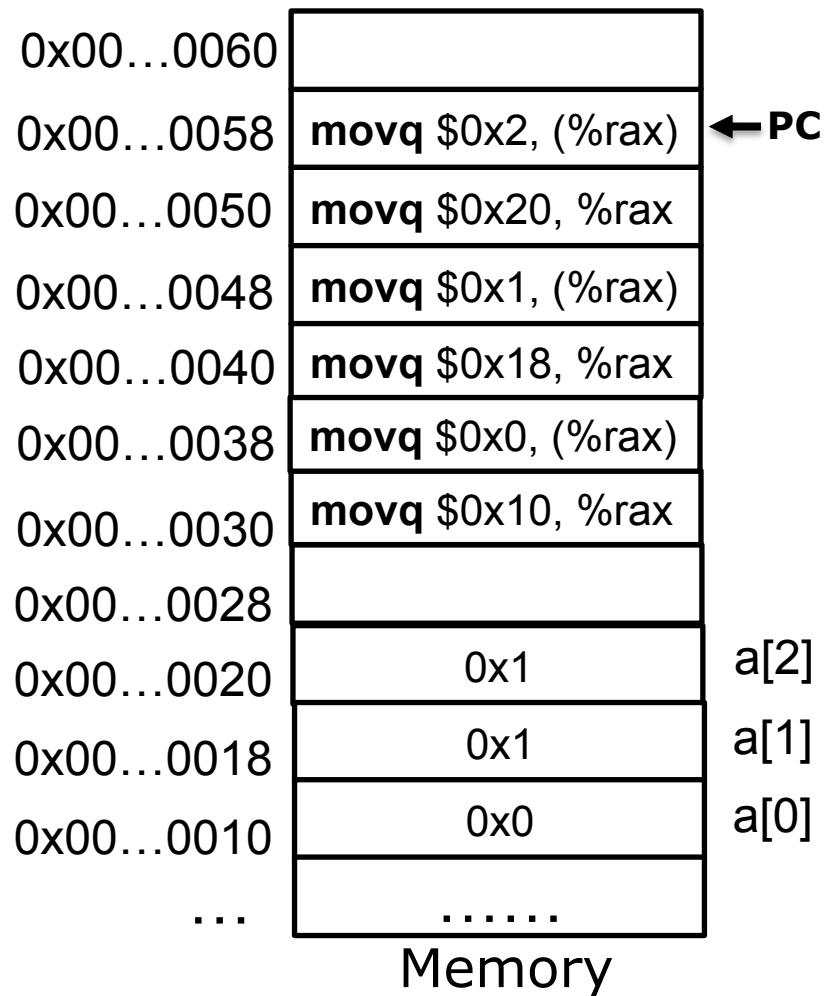
Example



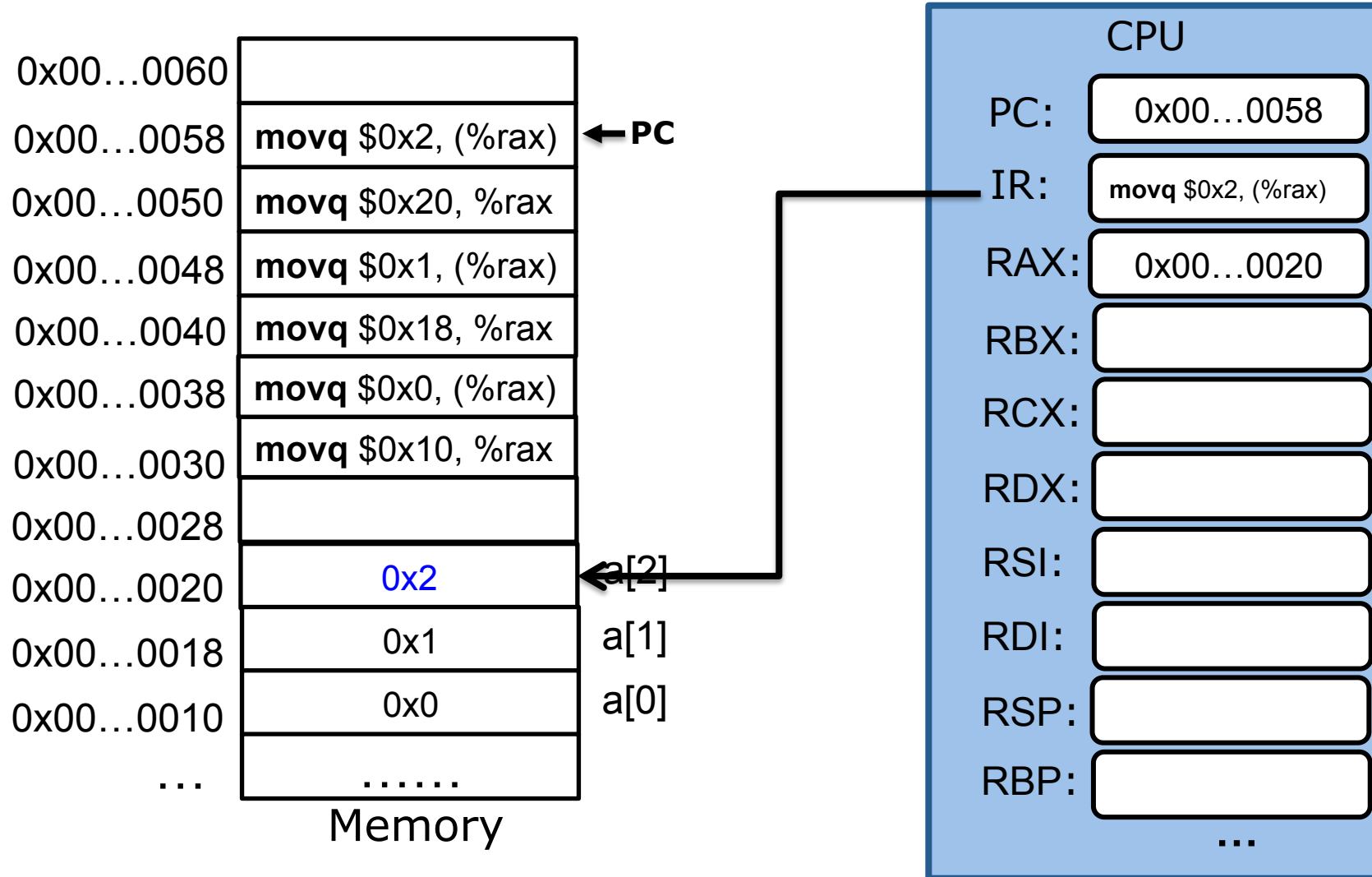
Example



Example



Example



Observation

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

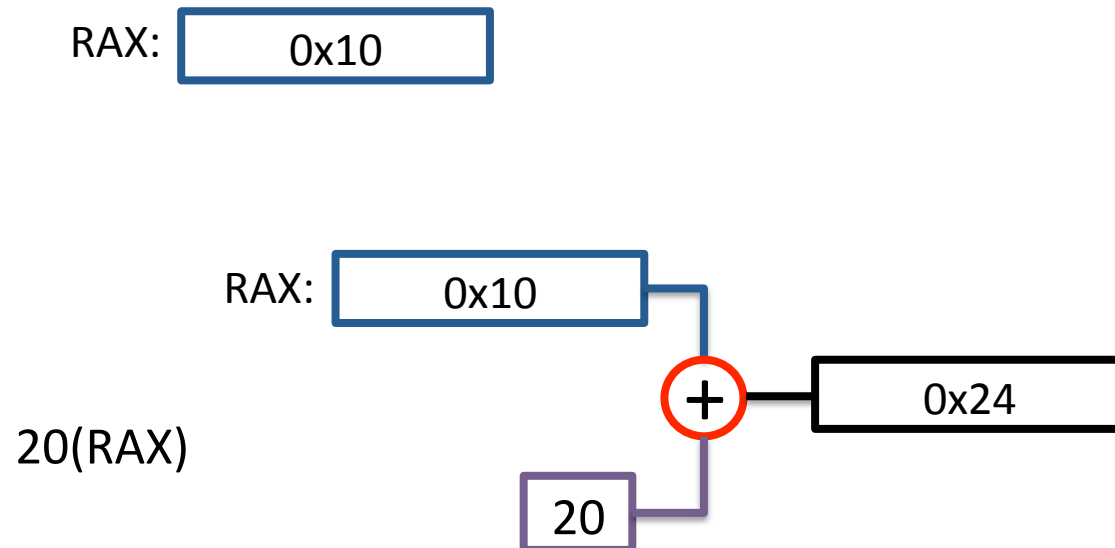
a[0], a[1] and a[2] have the same base address:&a[0]

- &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



Address mode with displacement

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

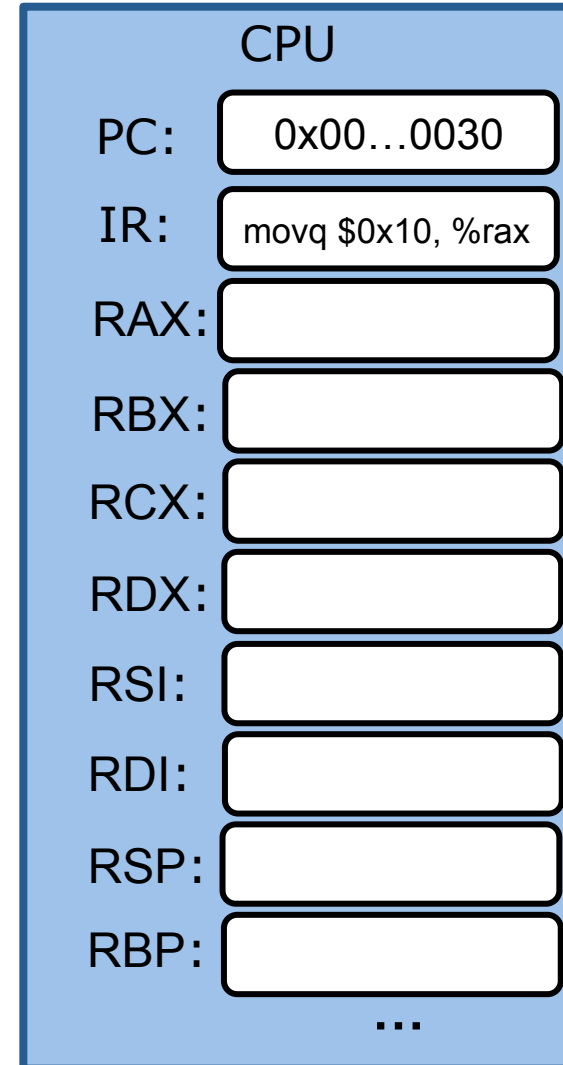
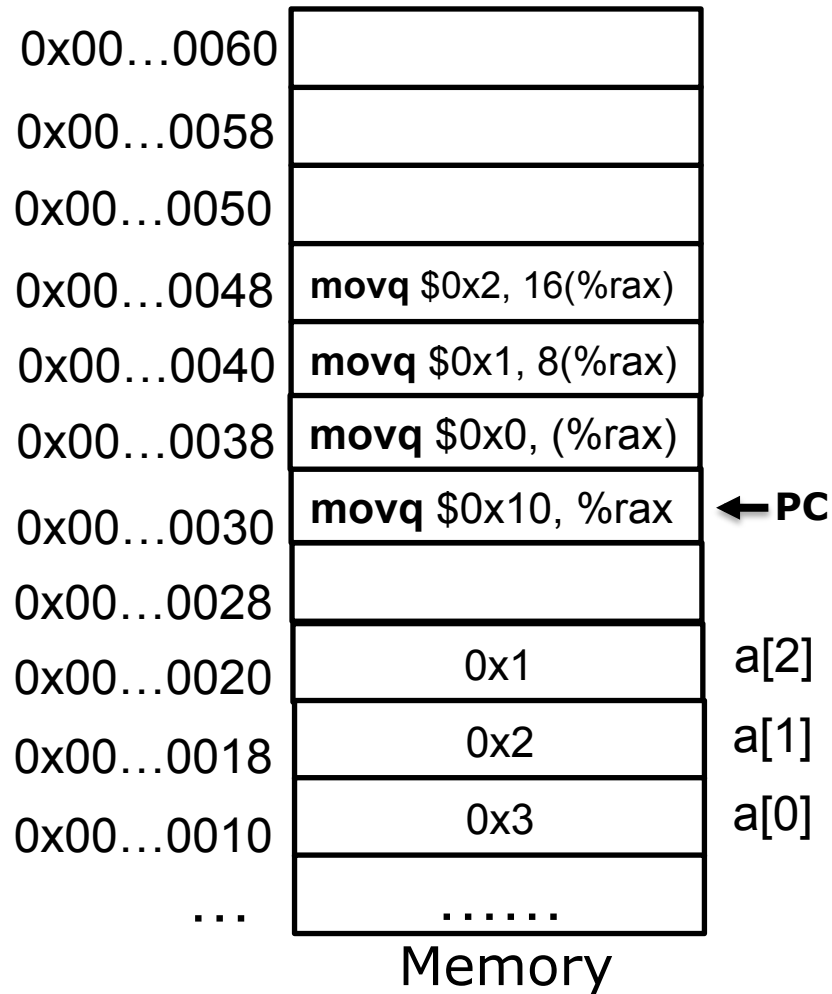
- Register specifies the start of the memory region
- Constant D specifies the offset

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

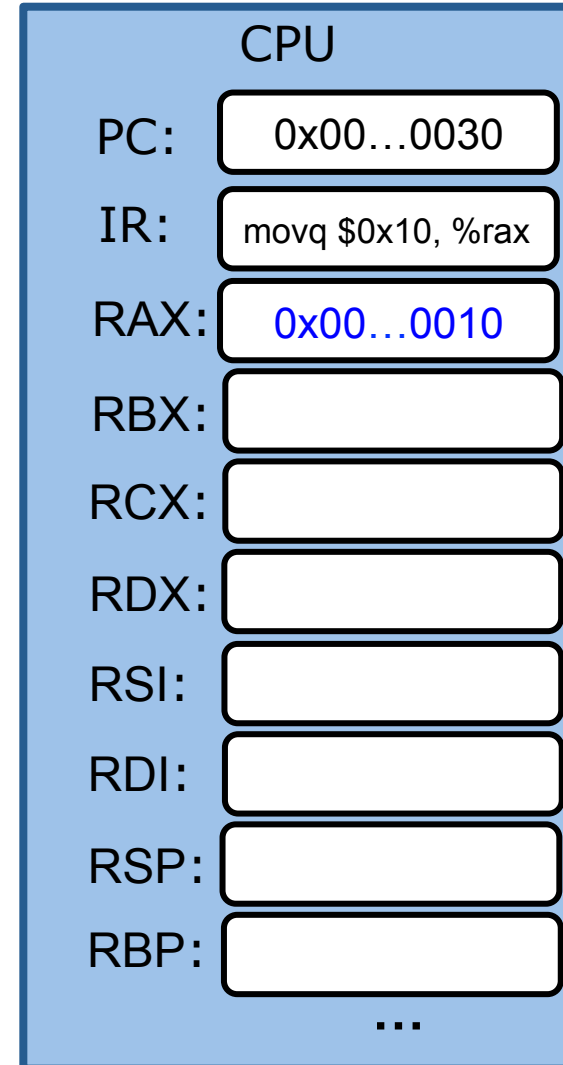
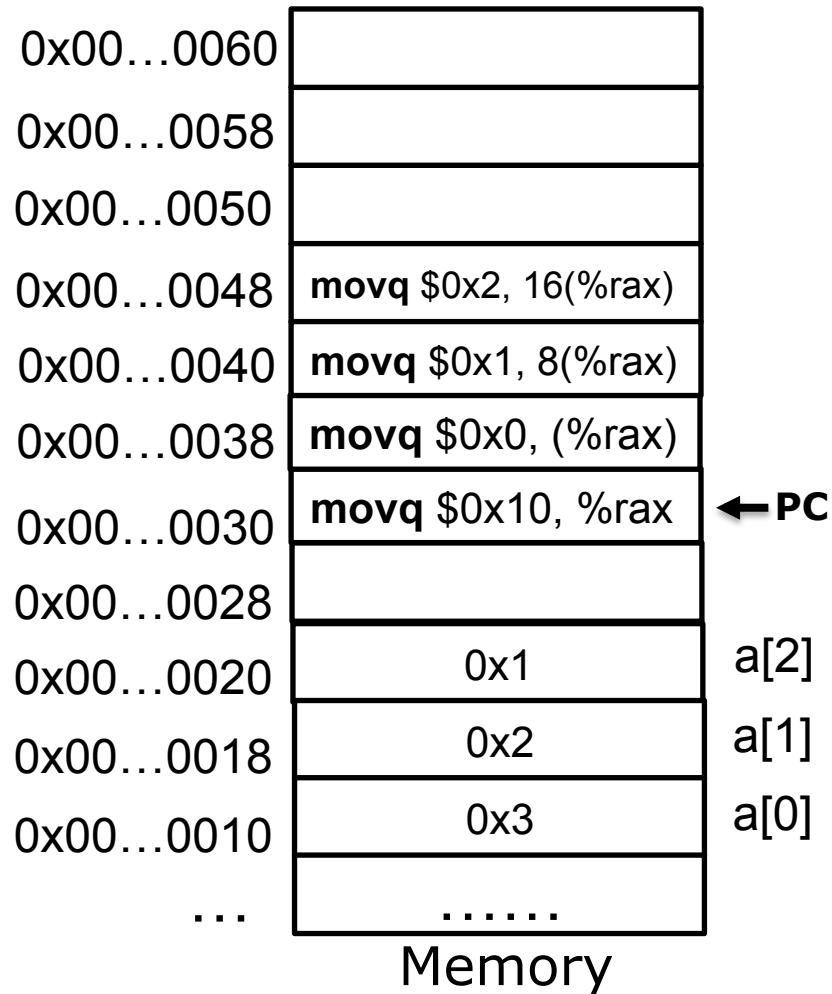


```
long a[] = {1, 2, 3};  
for(int i = 0; i < 3; i++) {  
    mov $i, D(reg); // D = i * 8, reg = &a[0]  
}
```

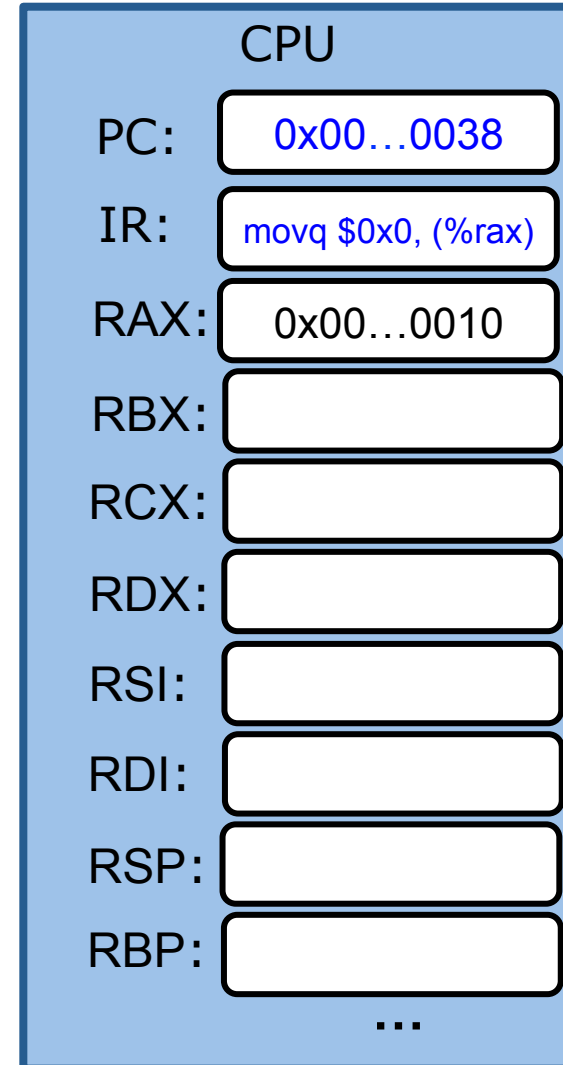
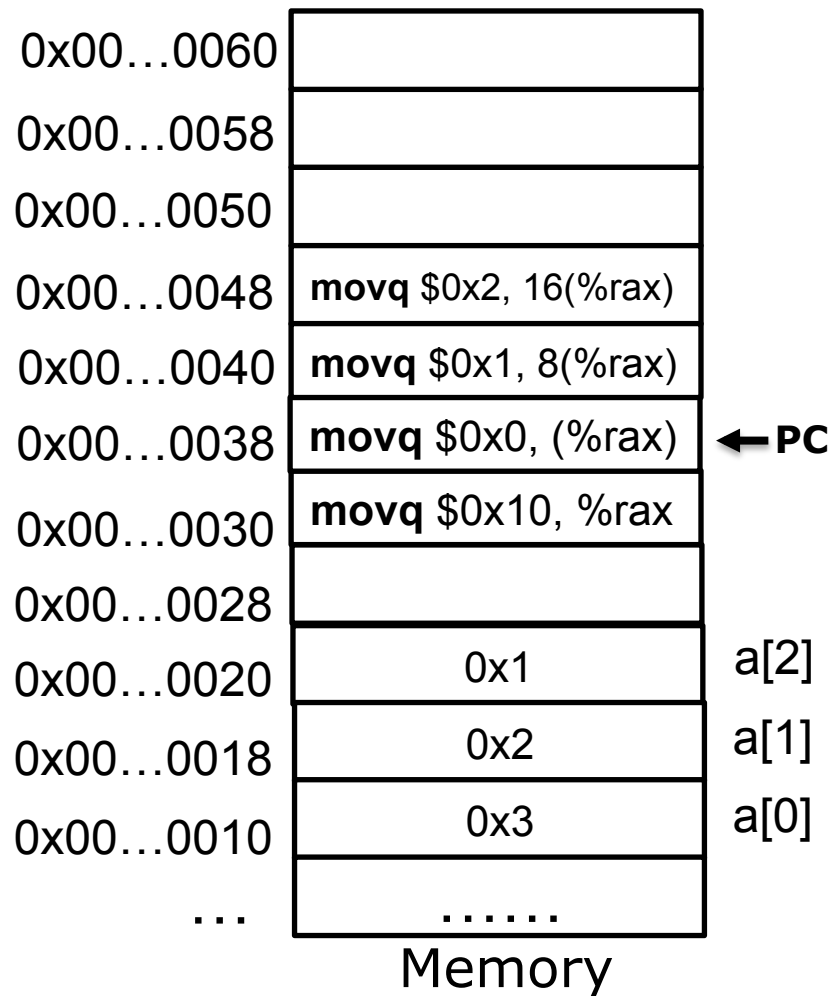
Example



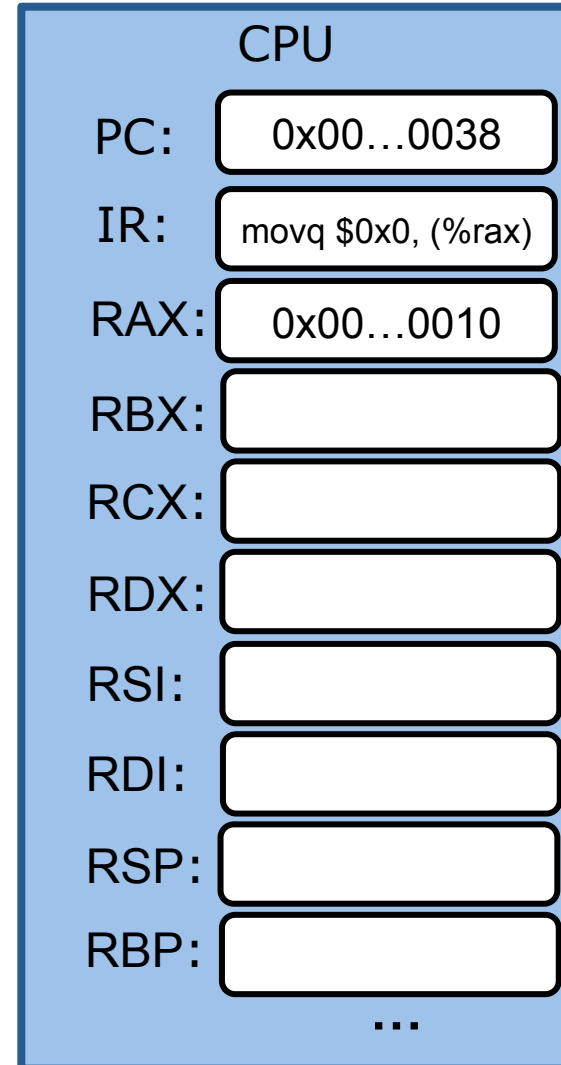
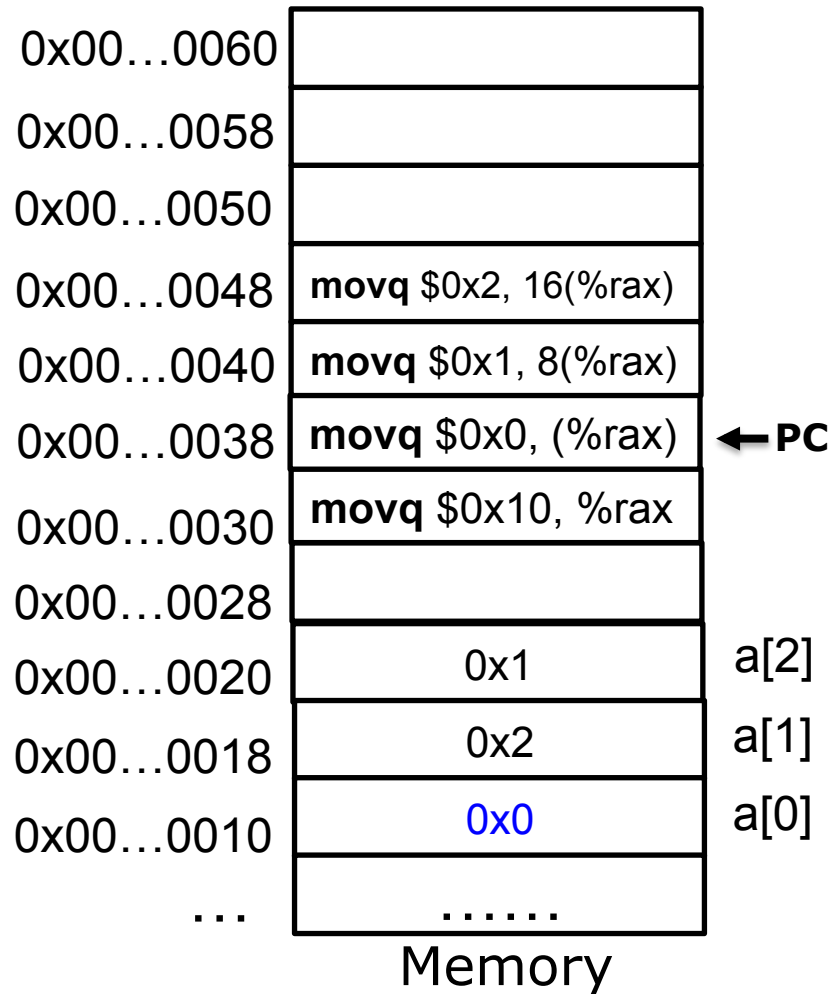
Example



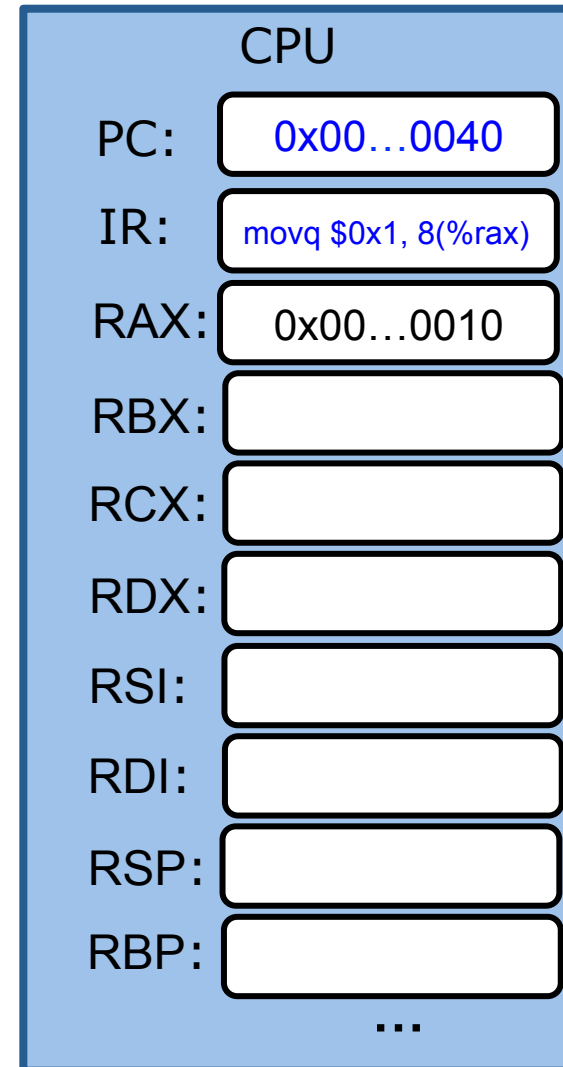
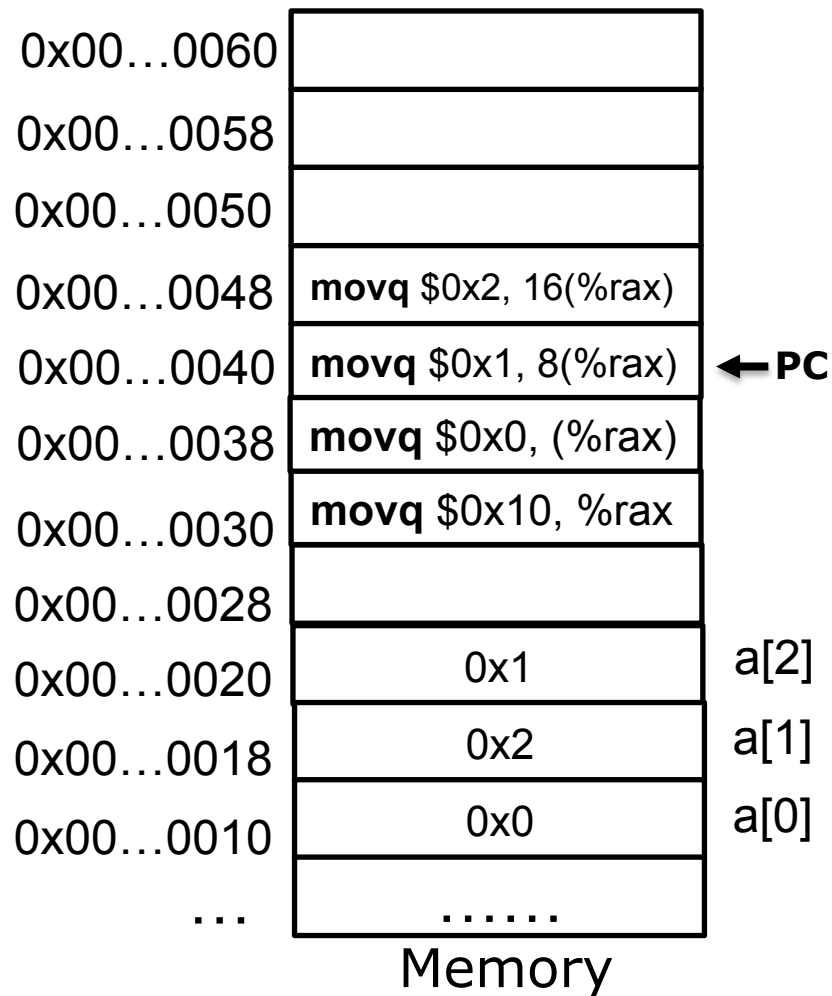
Example



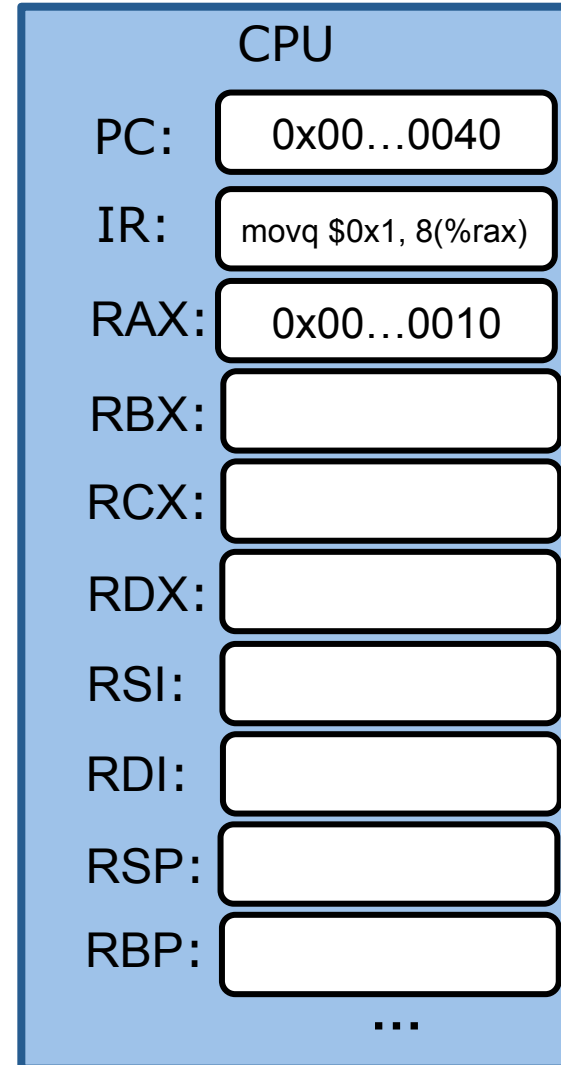
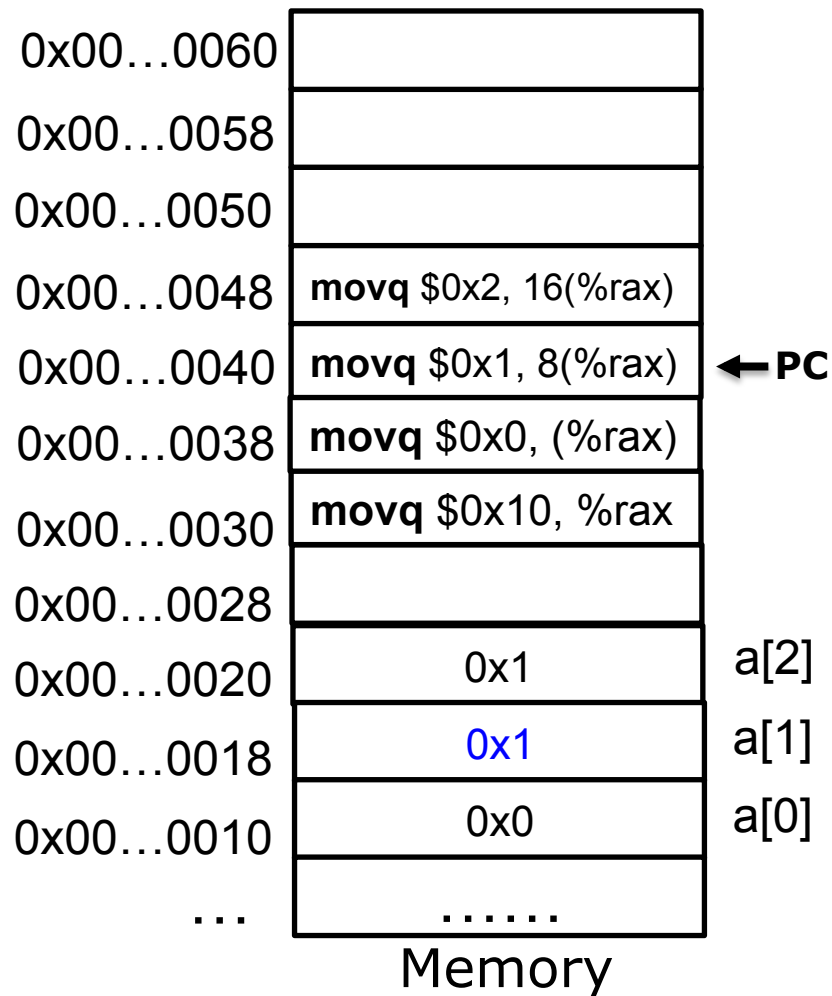
Example



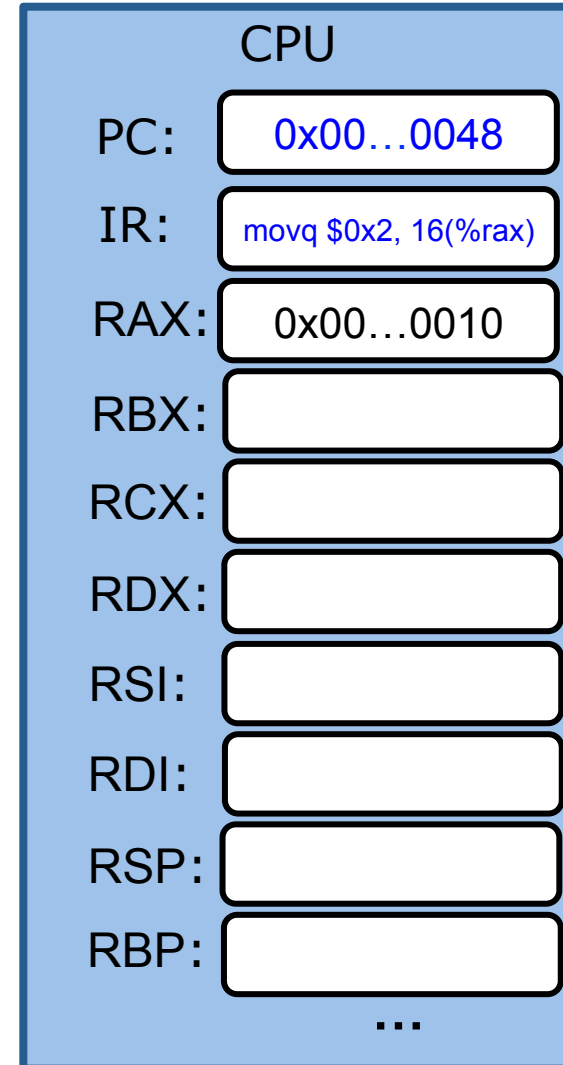
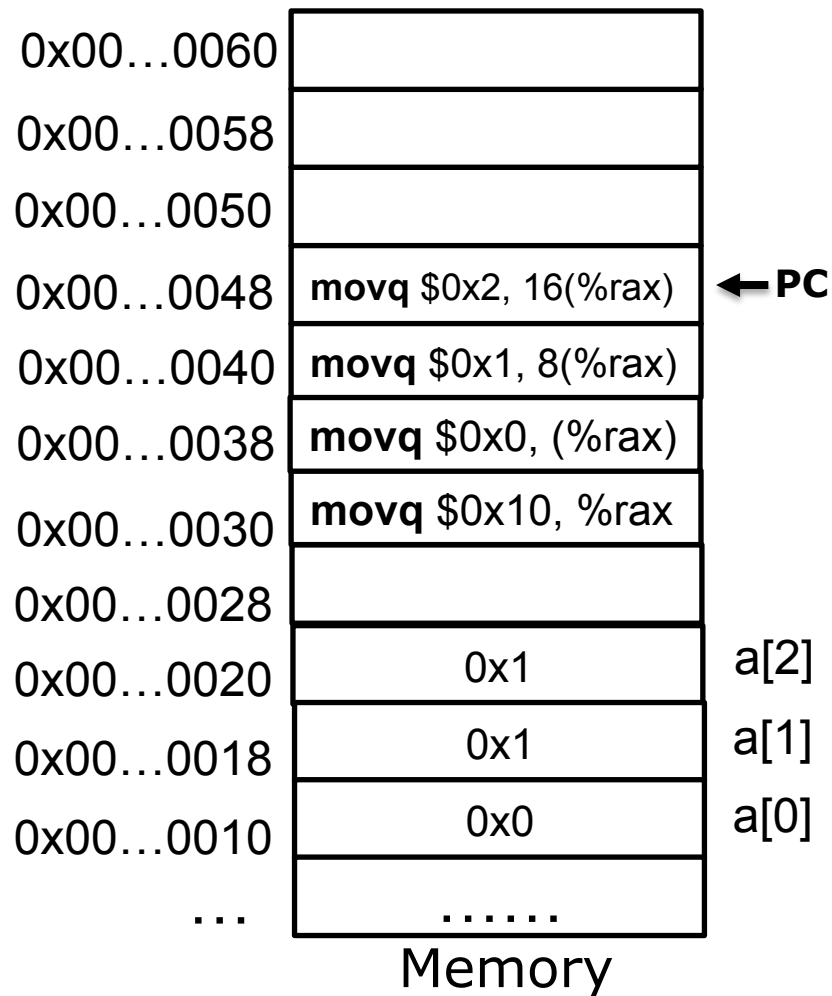
Example



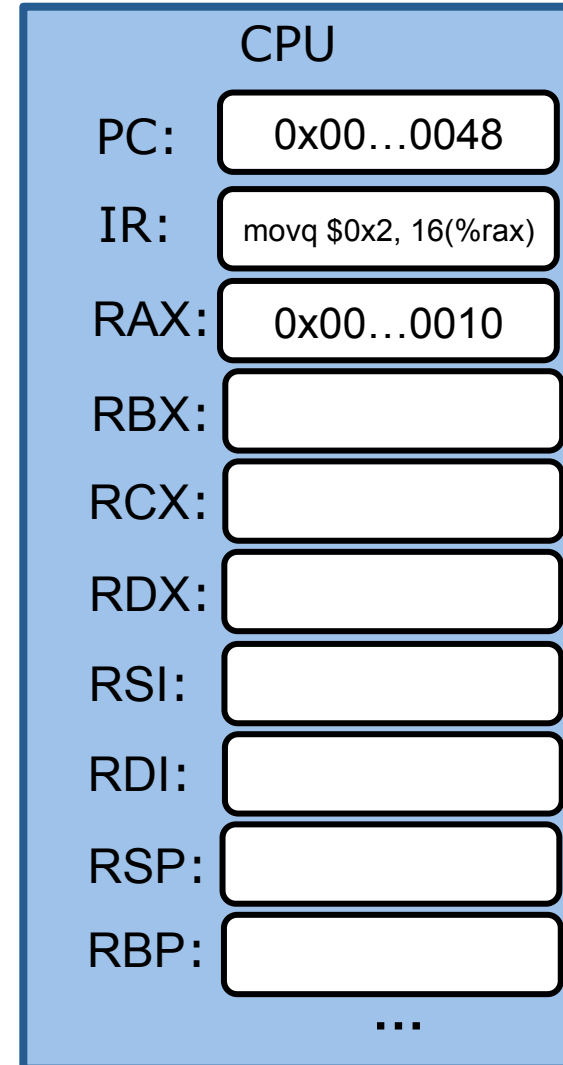
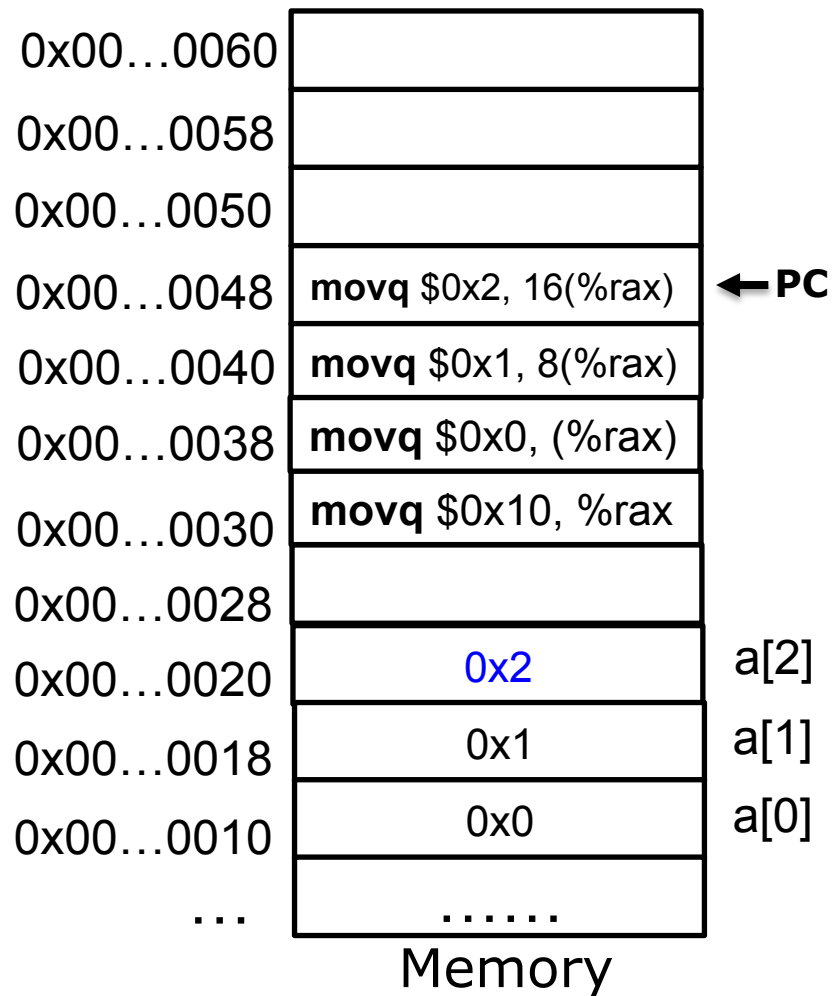
Example



Example



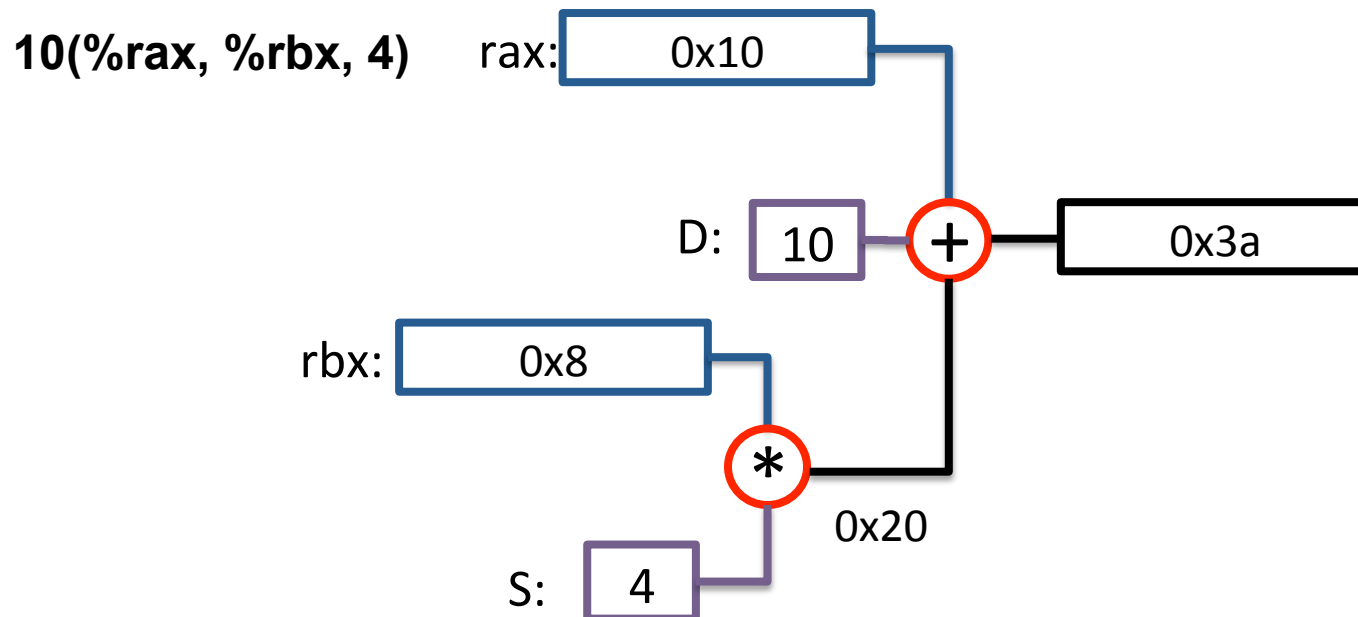
Example



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, we can just get rid of them

- (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>

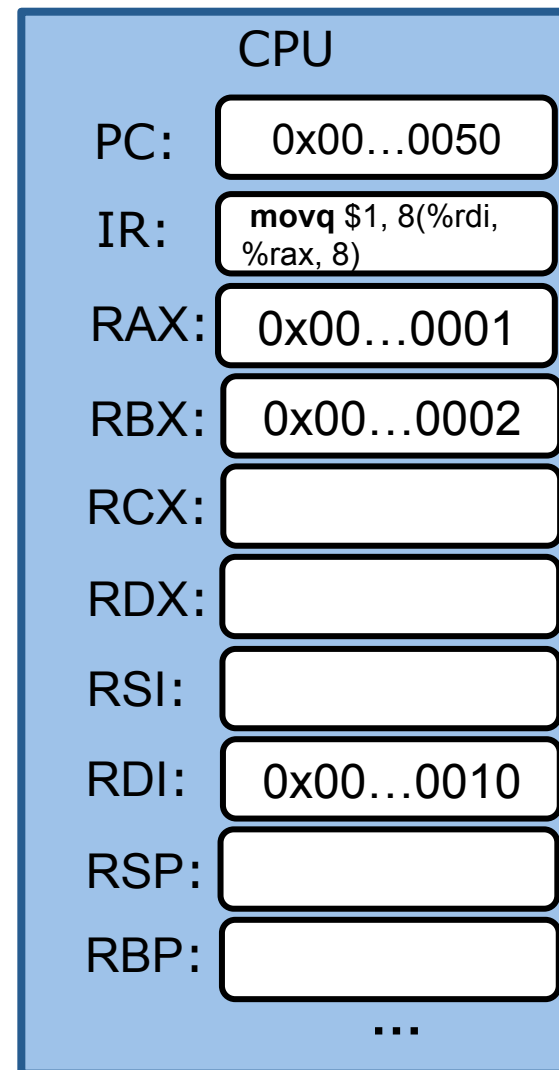
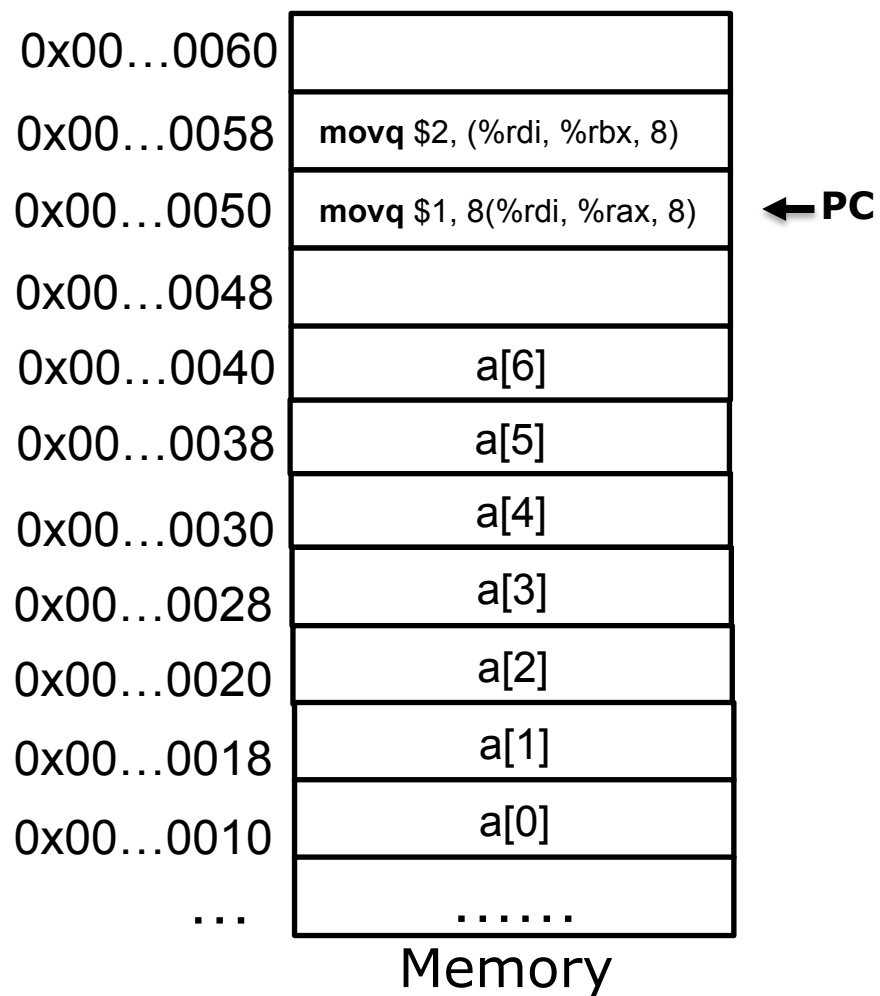
Expression	Address Computation	Address
<code>0x8(%rdx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%rdx,%rcx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%rdx,%rcx,4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80(,%rdx,2)</code>		

Address Computation Examples

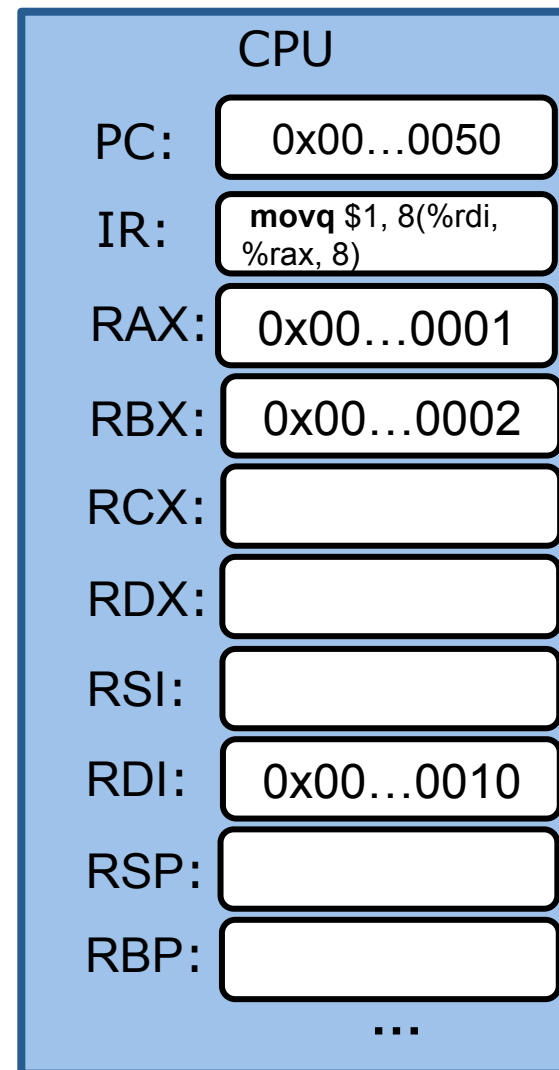
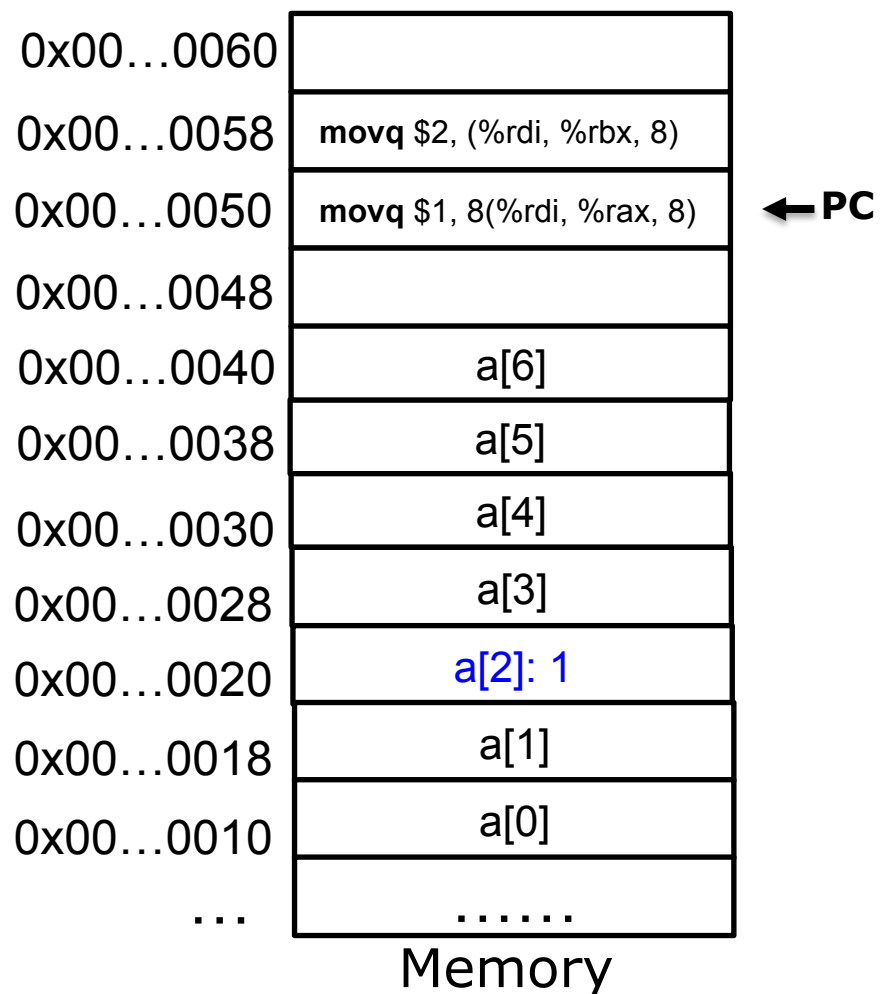
<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x100</code>

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>

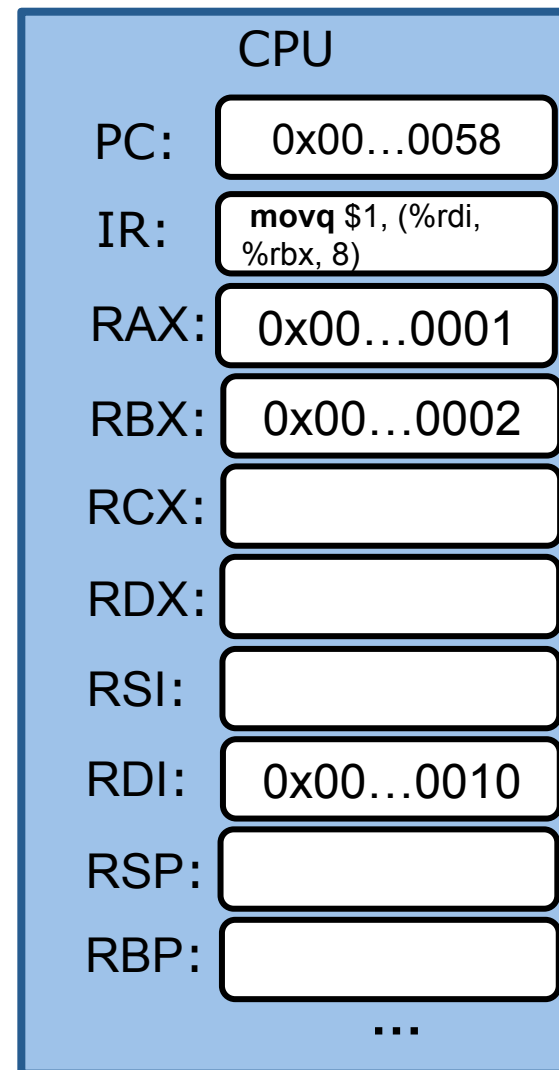
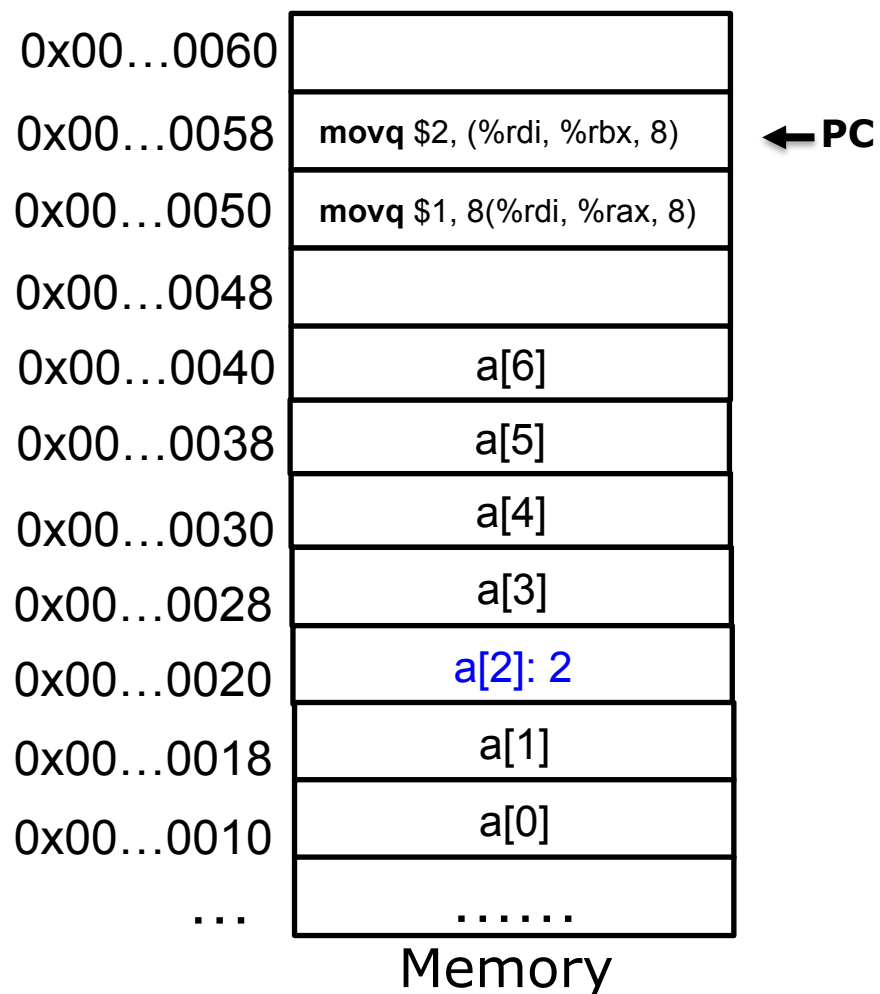
Example



Example



Example



mov{bwlq}

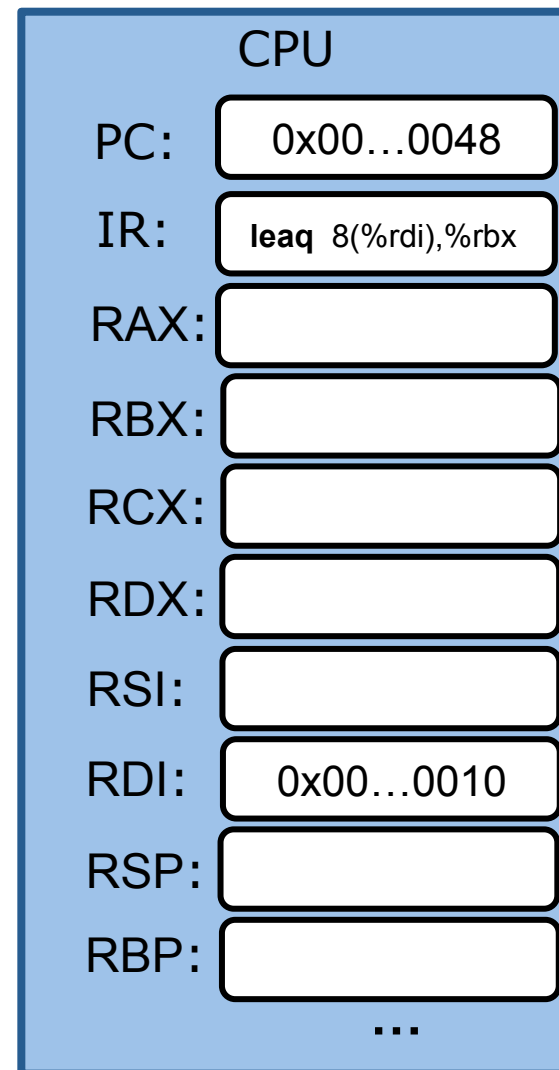
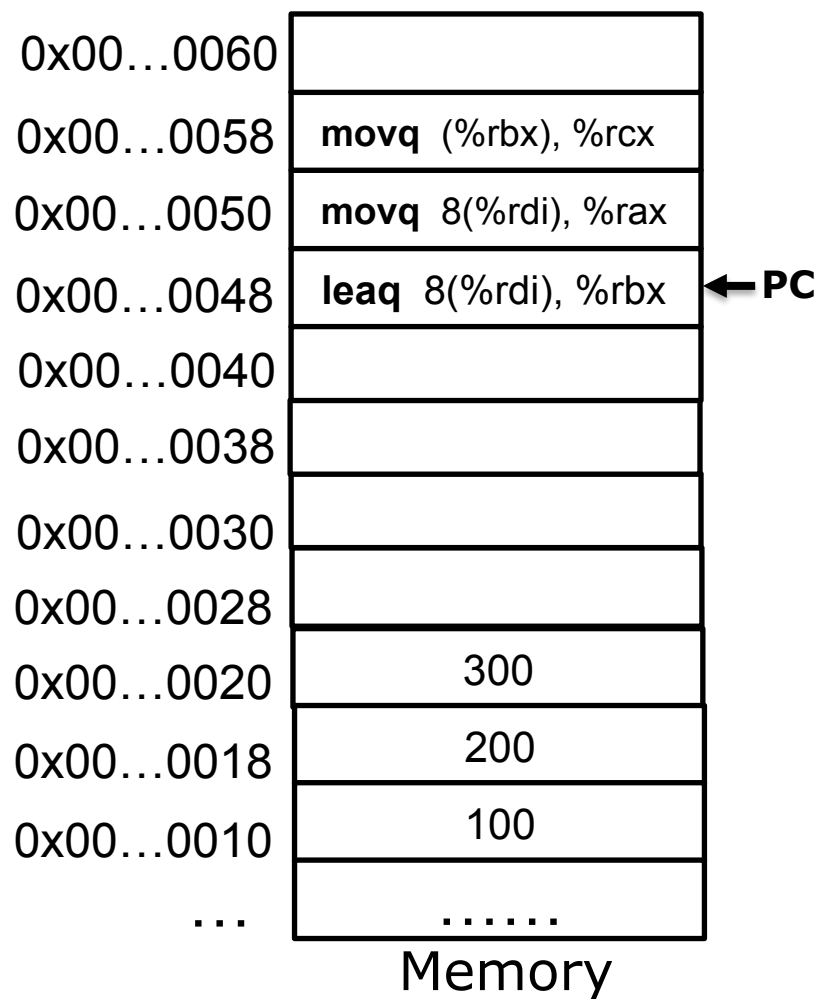
movb src, dest	Copy a byte from the source operand to the destination operand. e.g., movb %al, %bl
movw src, dest	Copy a word from the source operand to the destination operand. e.g., movw %ax, %bx
movl src, dest	Copy a long (32 bits) from the source operand to the destination operand. e.g., movl %eax,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

The lea instruction

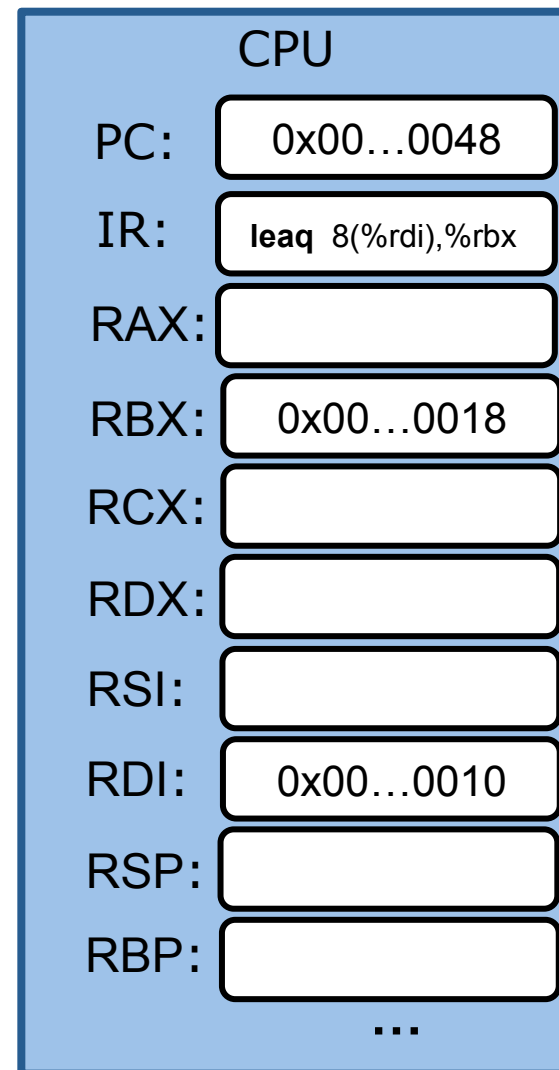
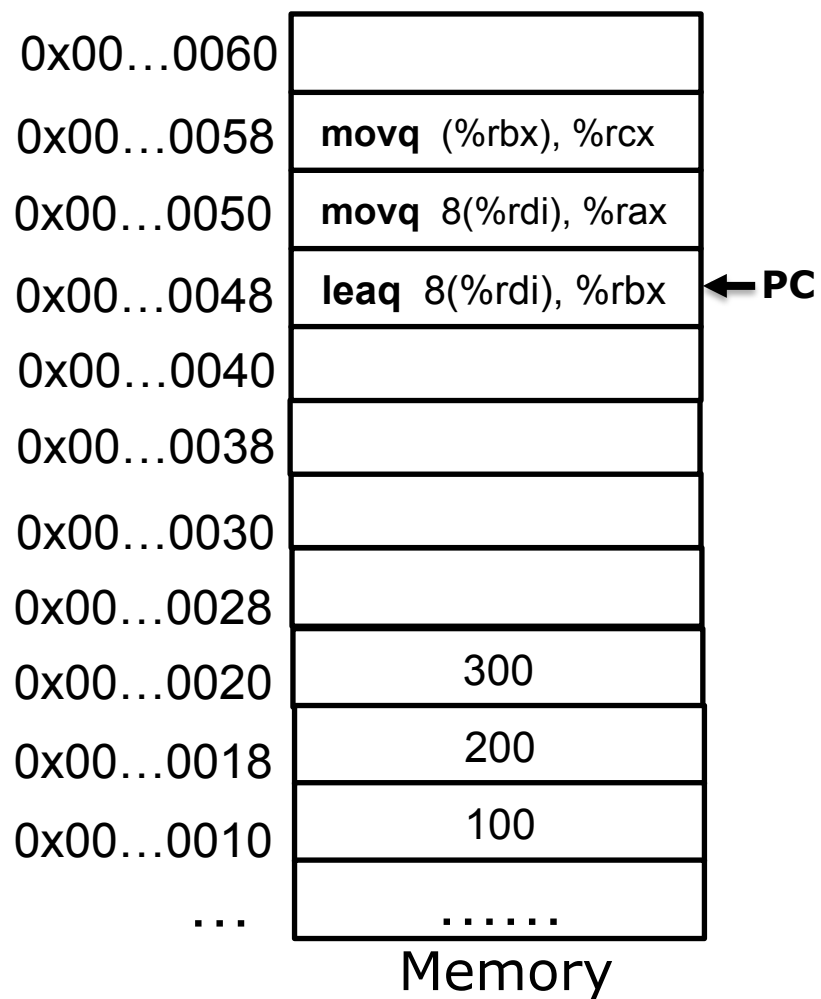
leaq *Source*, *Dest*

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

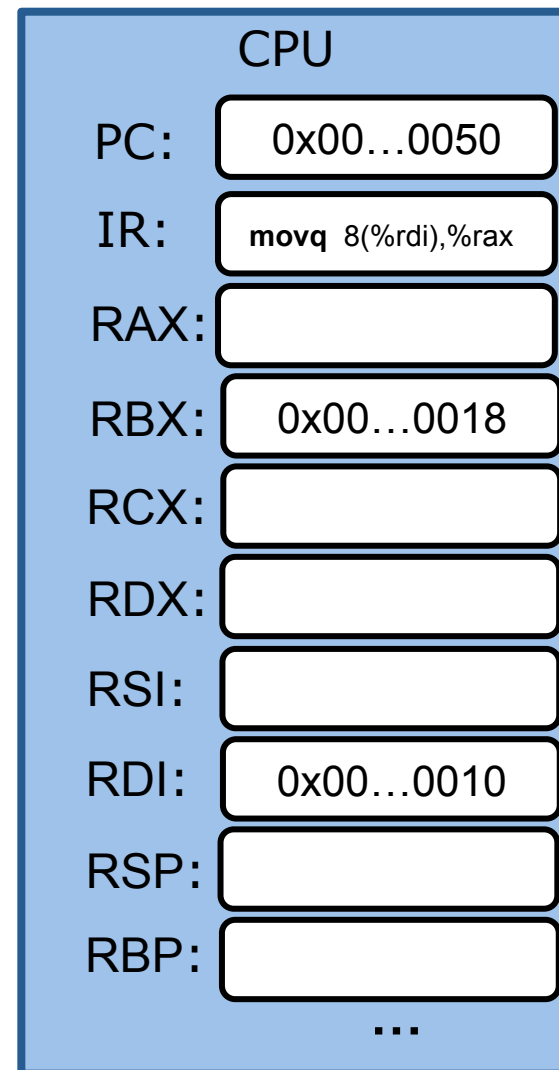
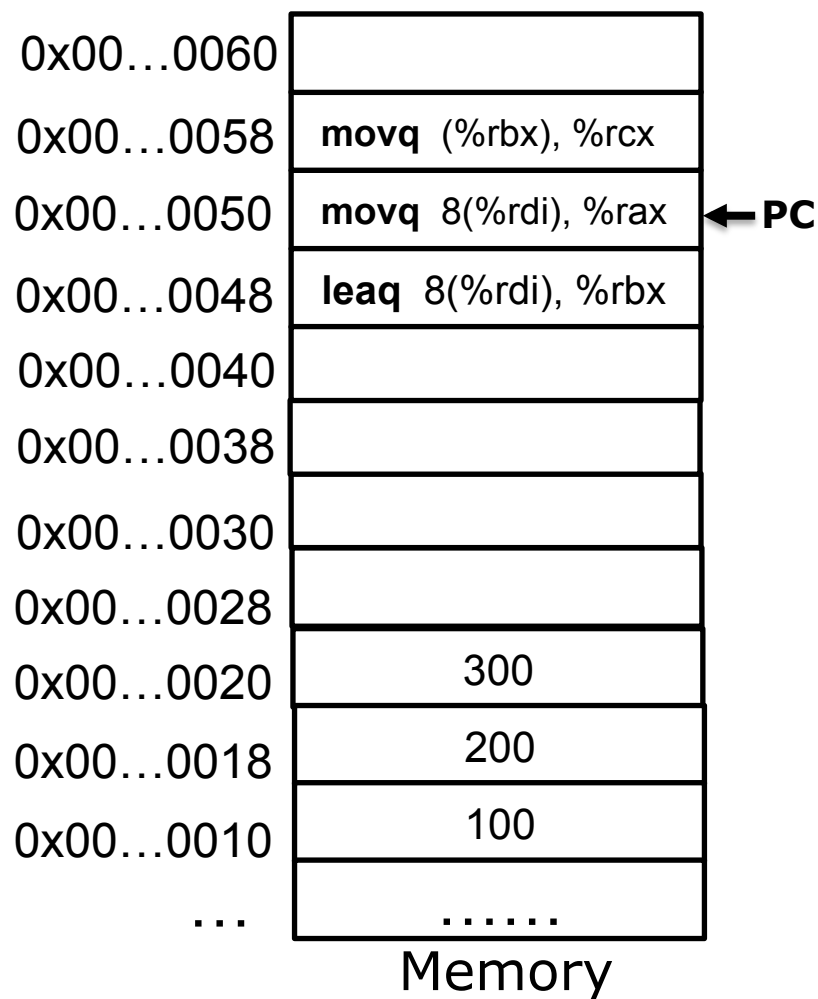
Example



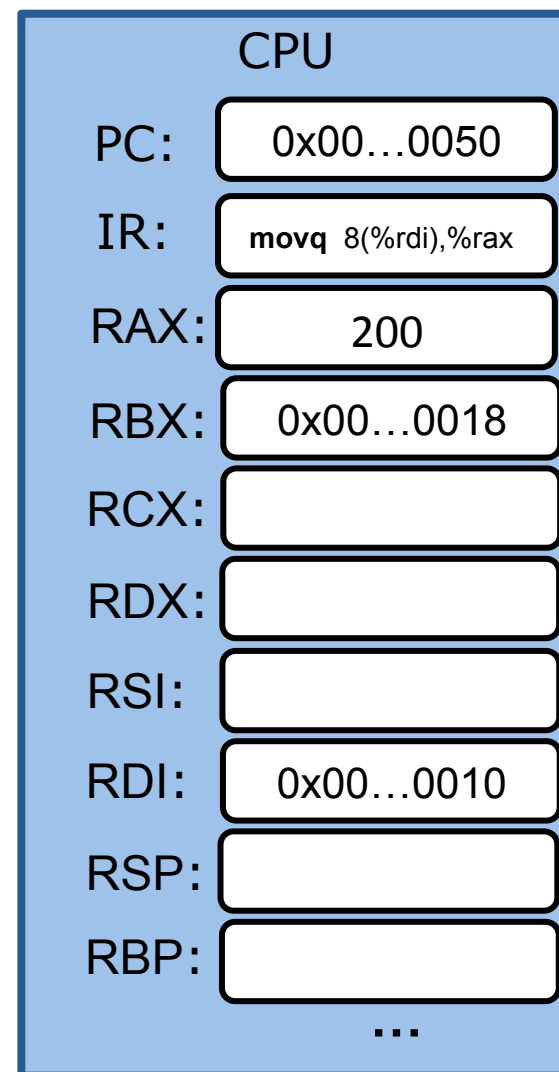
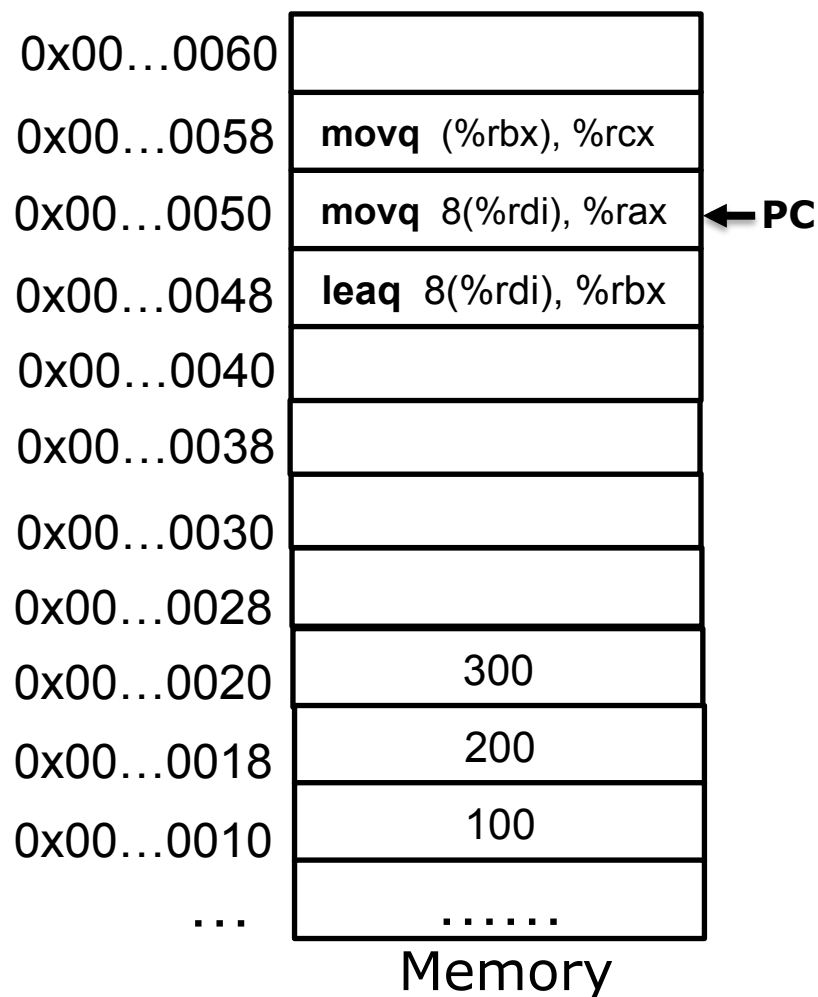
Example



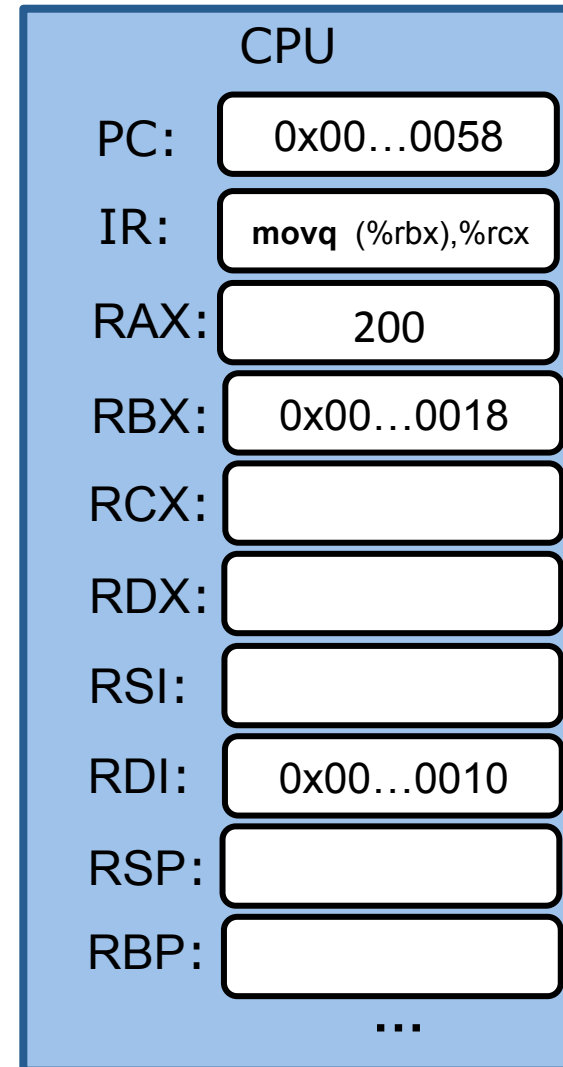
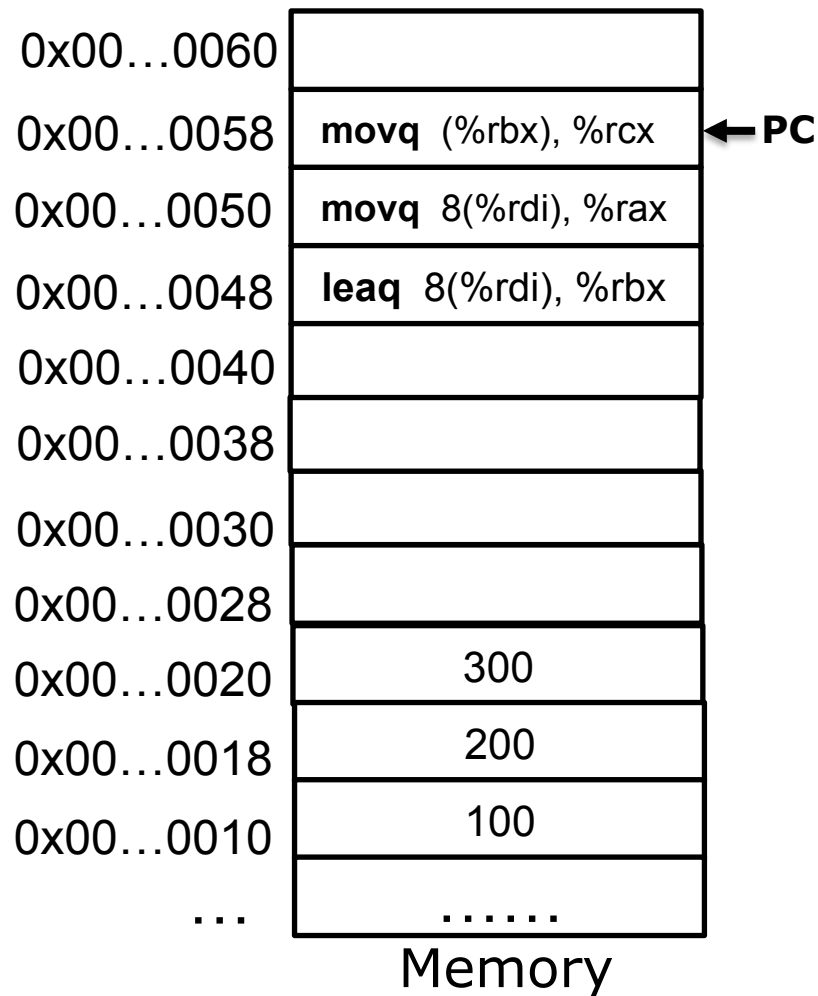
Example



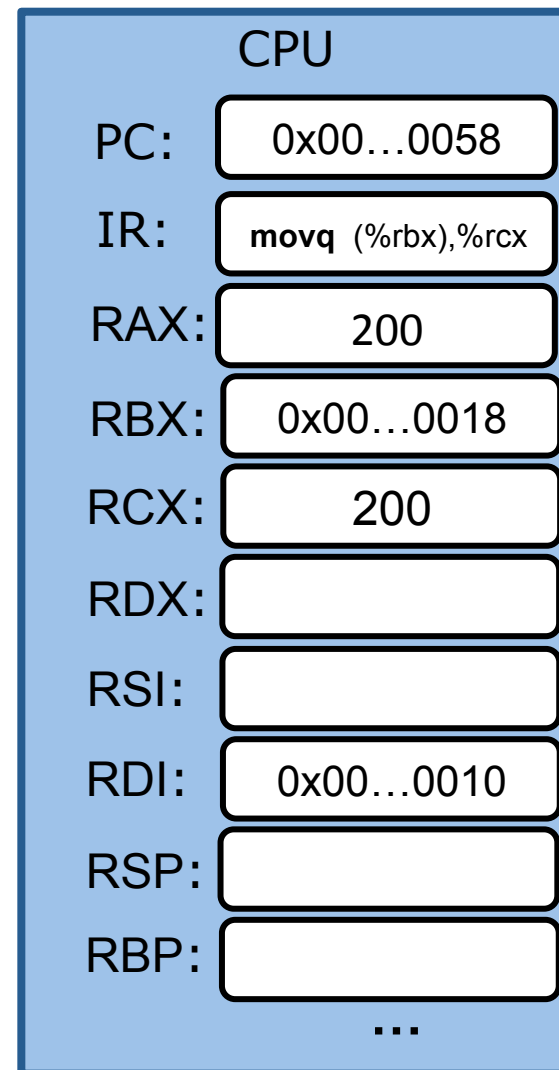
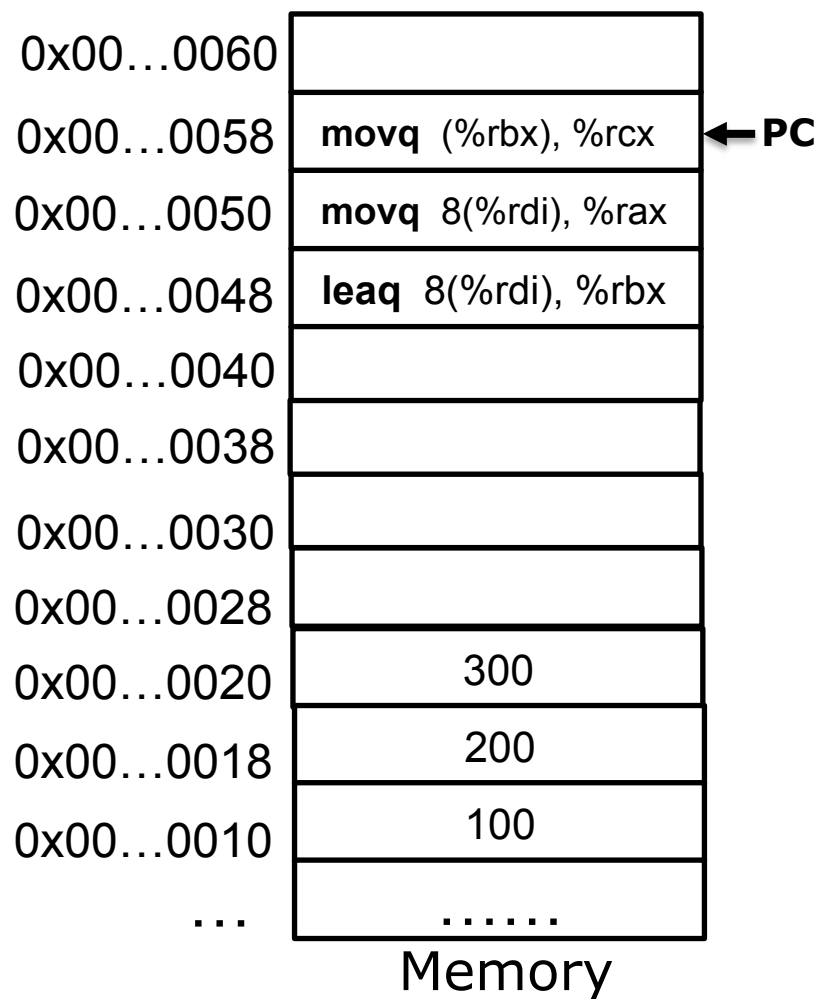
Example



Example



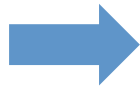
Example



Other usage of leaq

Computing arithmetic expressions of the form $x + k*y + d$ ($k = 1, 2, 4, \text{ or } 8$)

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

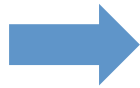


Assume %rdi has the value of x

Other usage of leaq

Computing arithmetic expressions of the form $x + k*y + d$ ($k = 1, 2, 4, \text{ or } 8$)

```
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;  
}
```

Some Arithmetic Operations

Two Operand Instructions:

addq Src, Dest Dest = Dest + Src

subq Src, Dest Dest = Dest - Src

imulq Src, Dest Dest = Dest * Src

salq Src, Dest Dest = Dest << Src

sarq 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

Also called shlq

Arithmetic

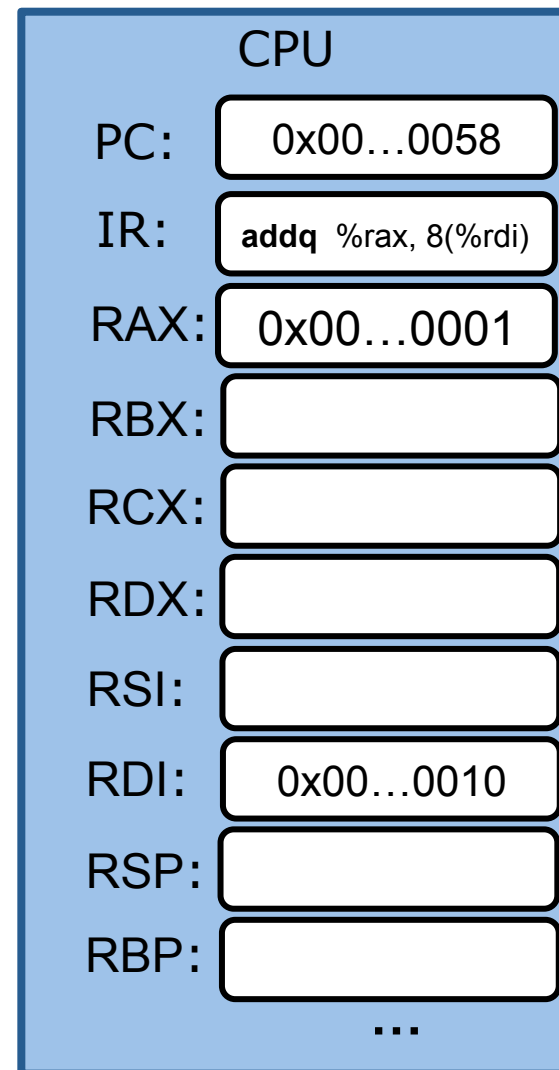
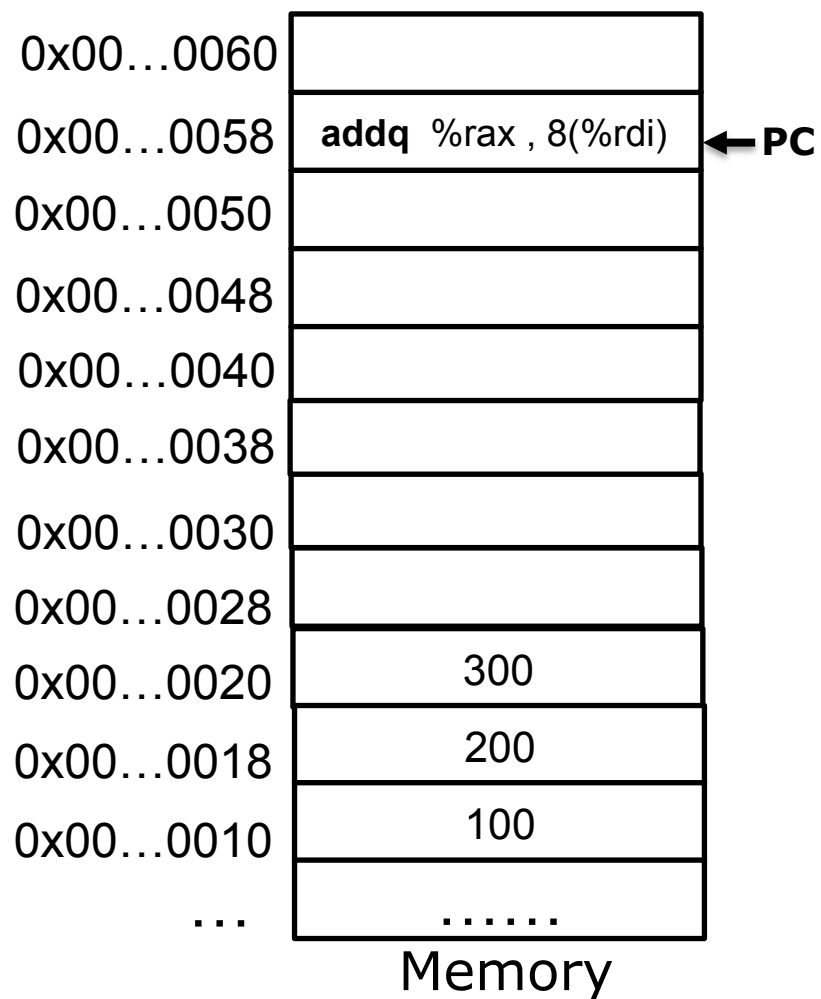
Logical

Some Arithmetic Operations

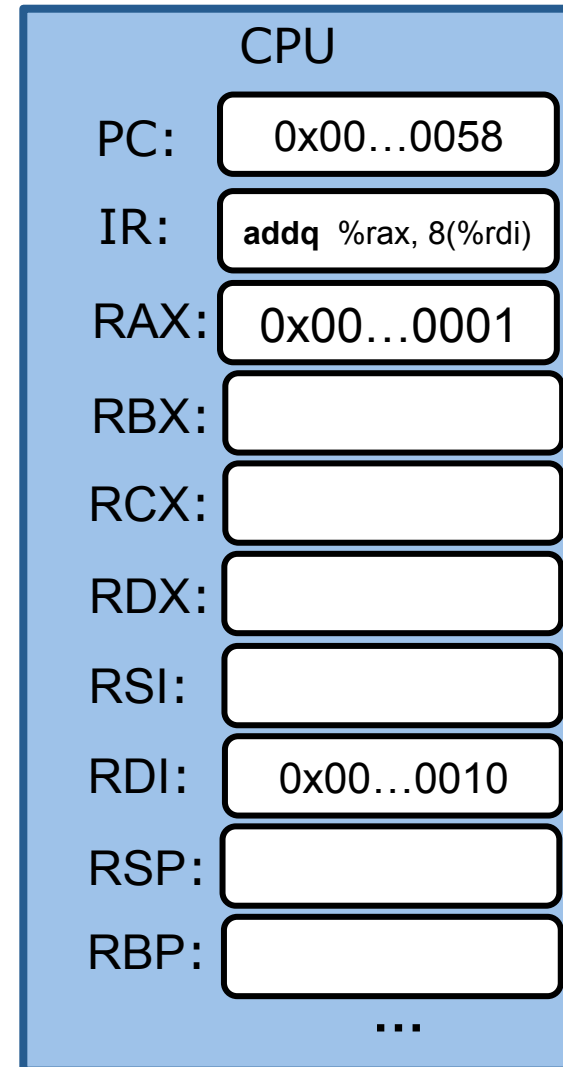
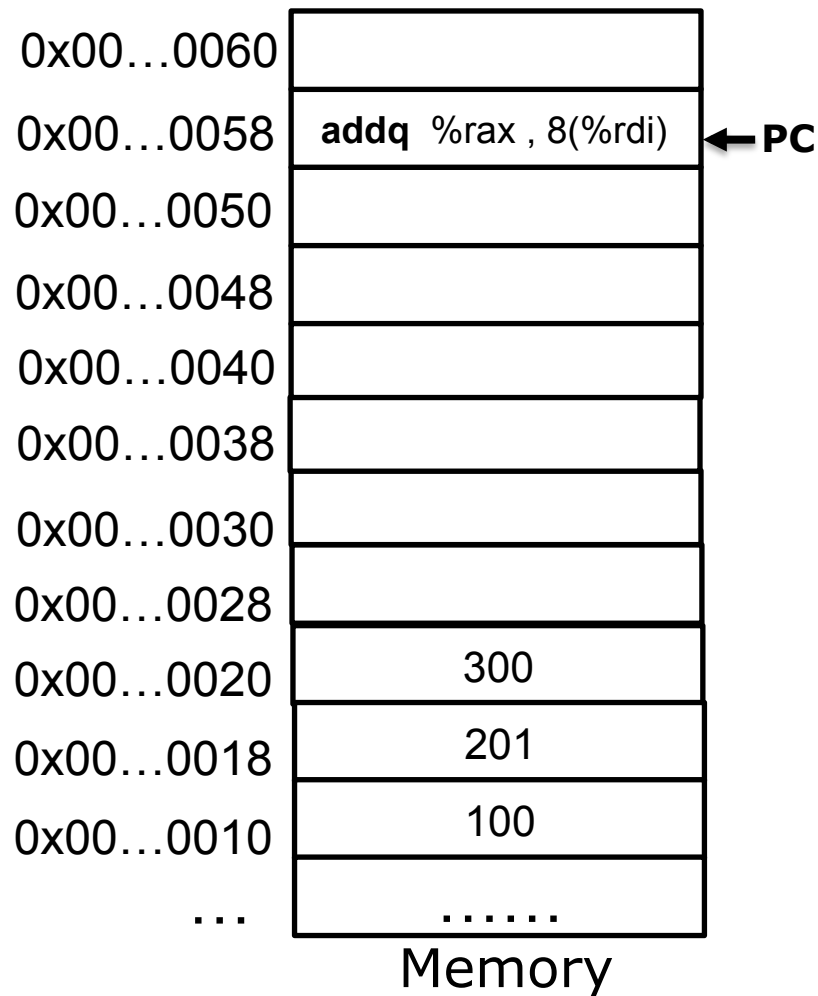
One Operand Instructions

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

Example



Example



Some Arithmetic Operations

Two Operand Instructions:

add{bwlq} Src, Dest Dest = Dest + Src

sub{bwlq} Src, Dest Dest = Dest - Src

imul{bwlq} Src, Dest Dest = Dest * Src

sal{bwlq} Src, Dest Dest = Dest << Src

sar{bwlq} Src, Dest Dest = Dest >> Src

shr{bwlq} Src, Dest Dest = Dest >> Src

xor{bwlq} Src, Dest Dest = Dest ^ Src

and{bwlq} Src, Dest Dest = Dest & Src

or{bwlq} Src, Dest Dest = Dest | Src

Also called shlq

Arithmetic

Logical

Some Arithmetic Operations

One Operand Instructions

inc{bwlq}	Dest	Dest = Dest + 1
dec{bwlq}	Dest	Dest = Dest - 1
neg{bwlq}	Dest	Dest = - Dest
not{bwlq}	Dest	Dest = ~Dest