

Machine Program: Basics

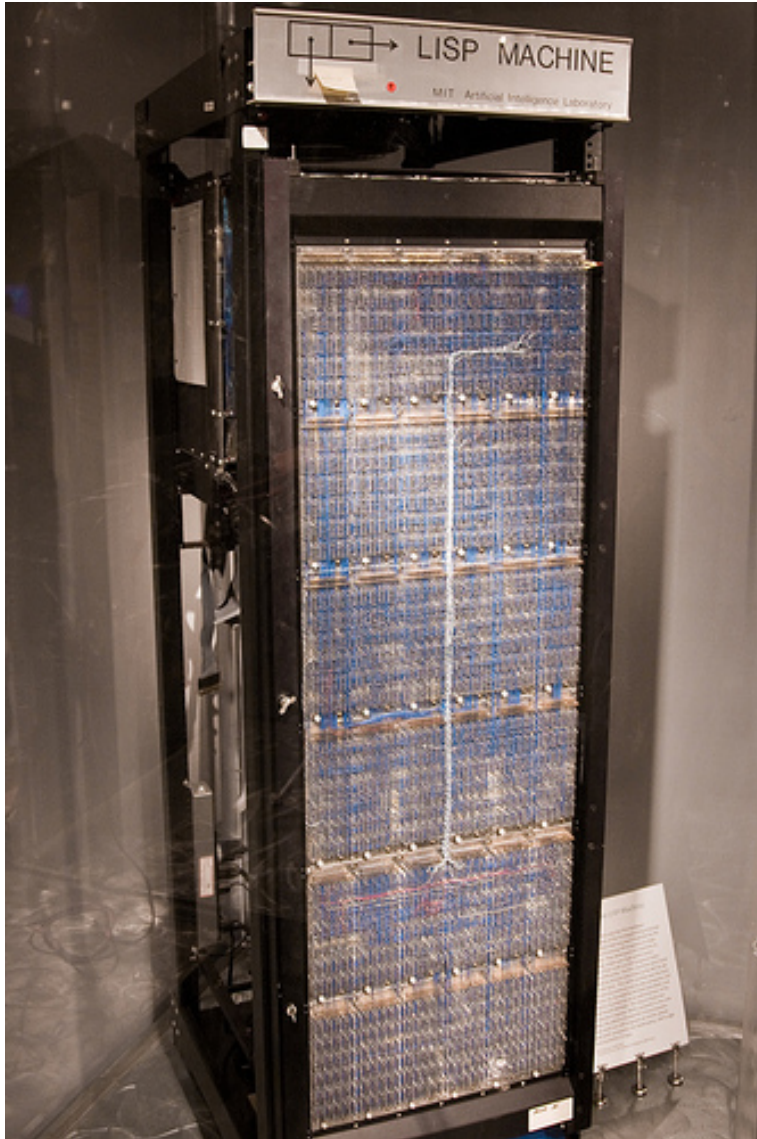
Jinyang Li

some are 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?

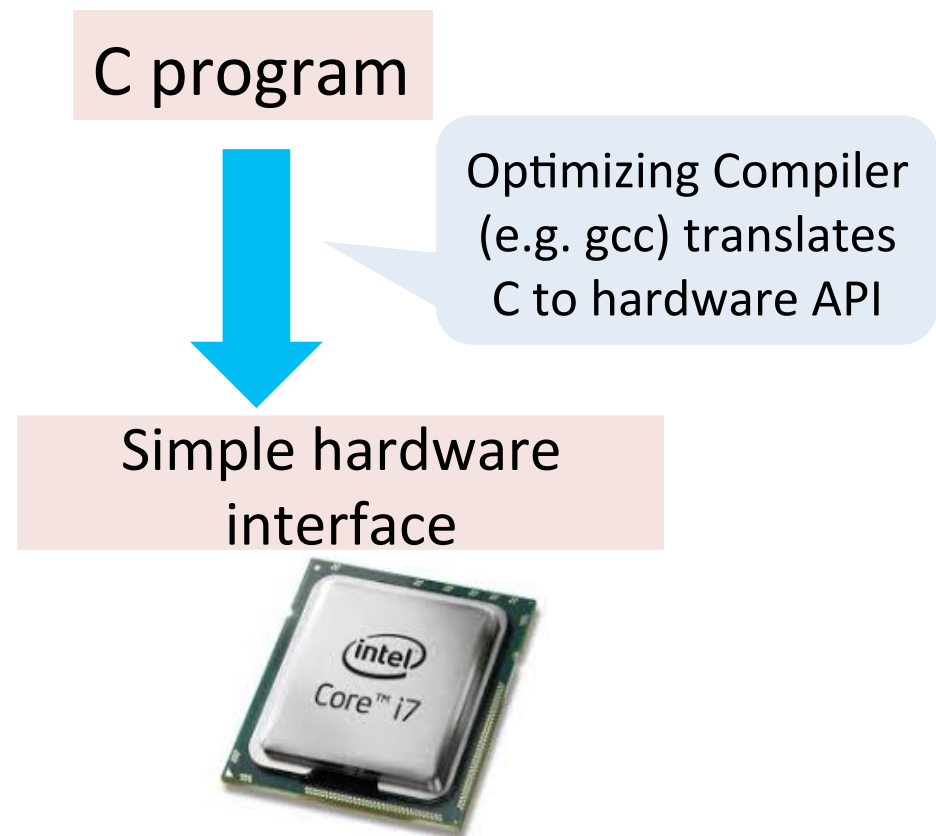
a CPU to execute C directly?



- 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:

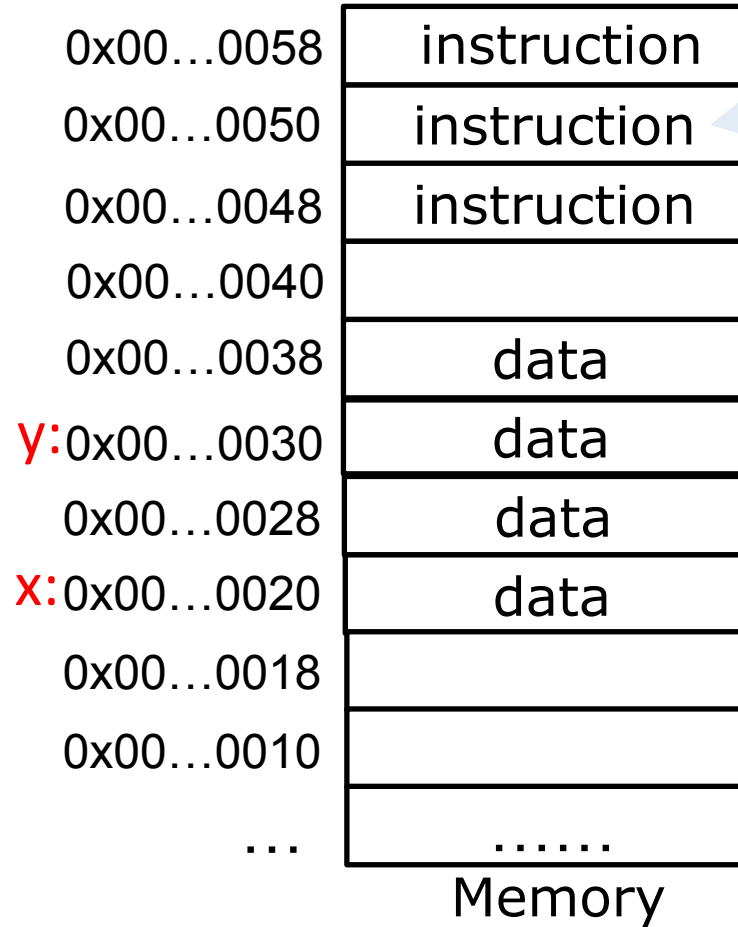


C vs. machine code

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

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

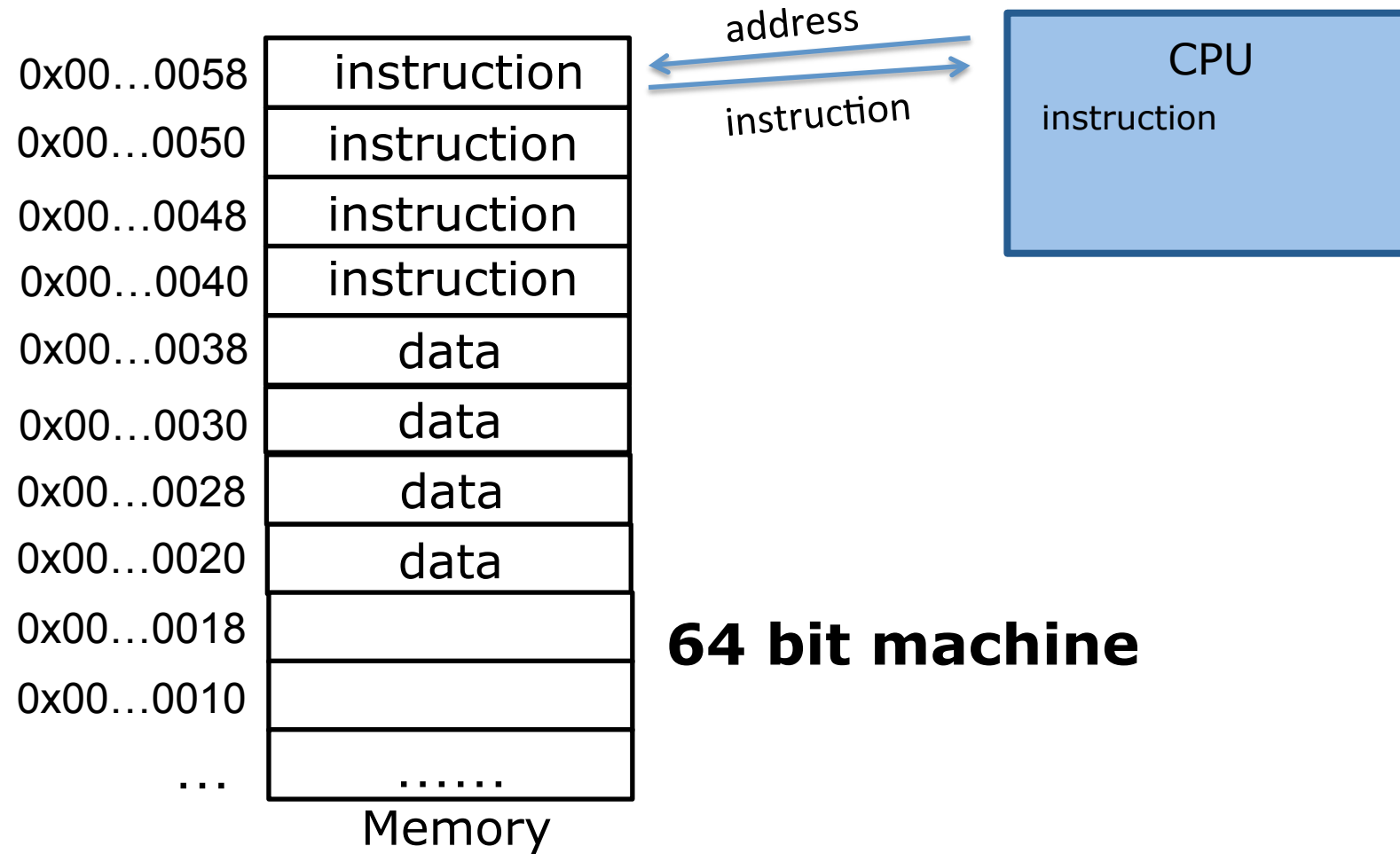

compile to
x86 executable



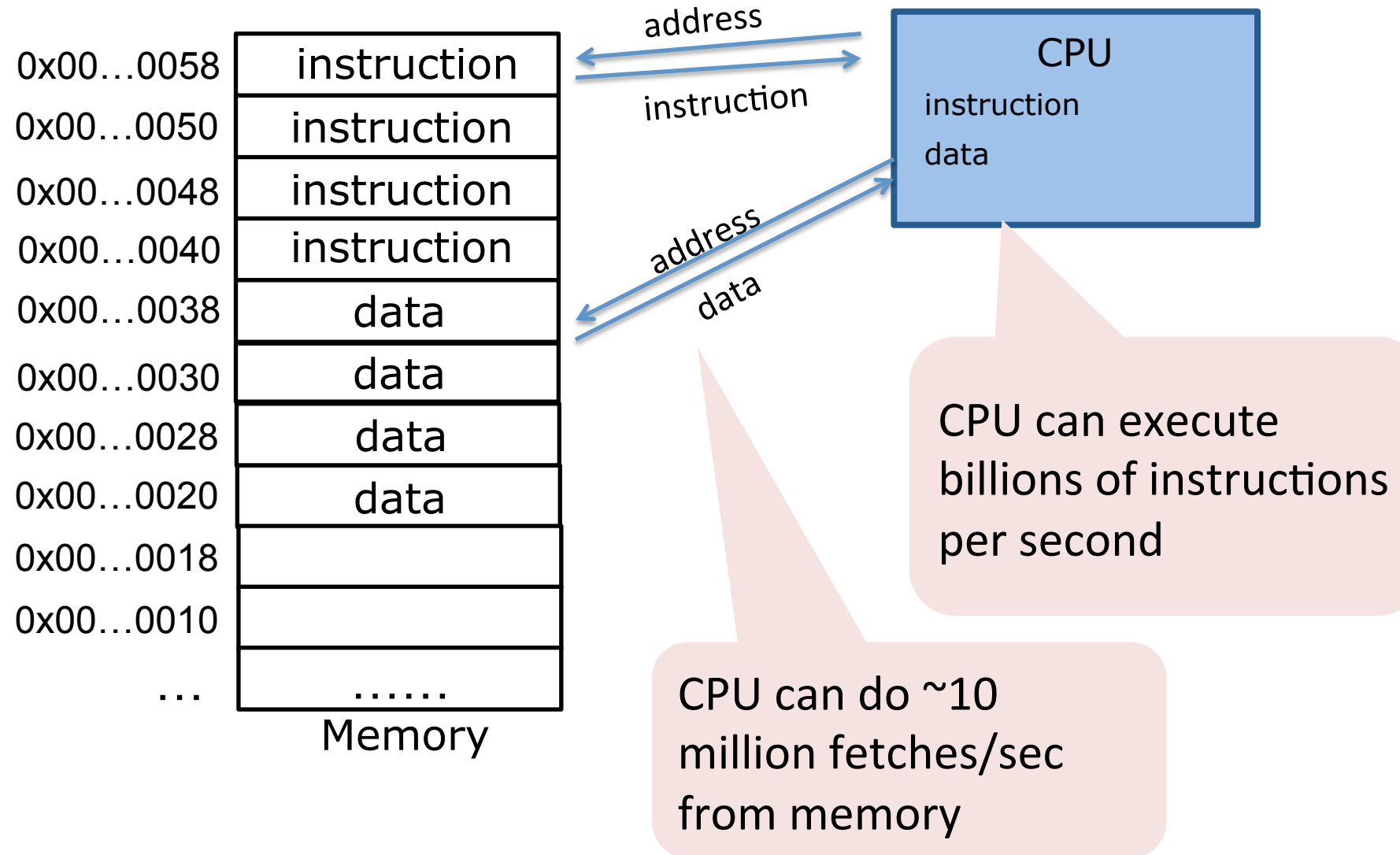
E.g. move data
from one
memory location
to another

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

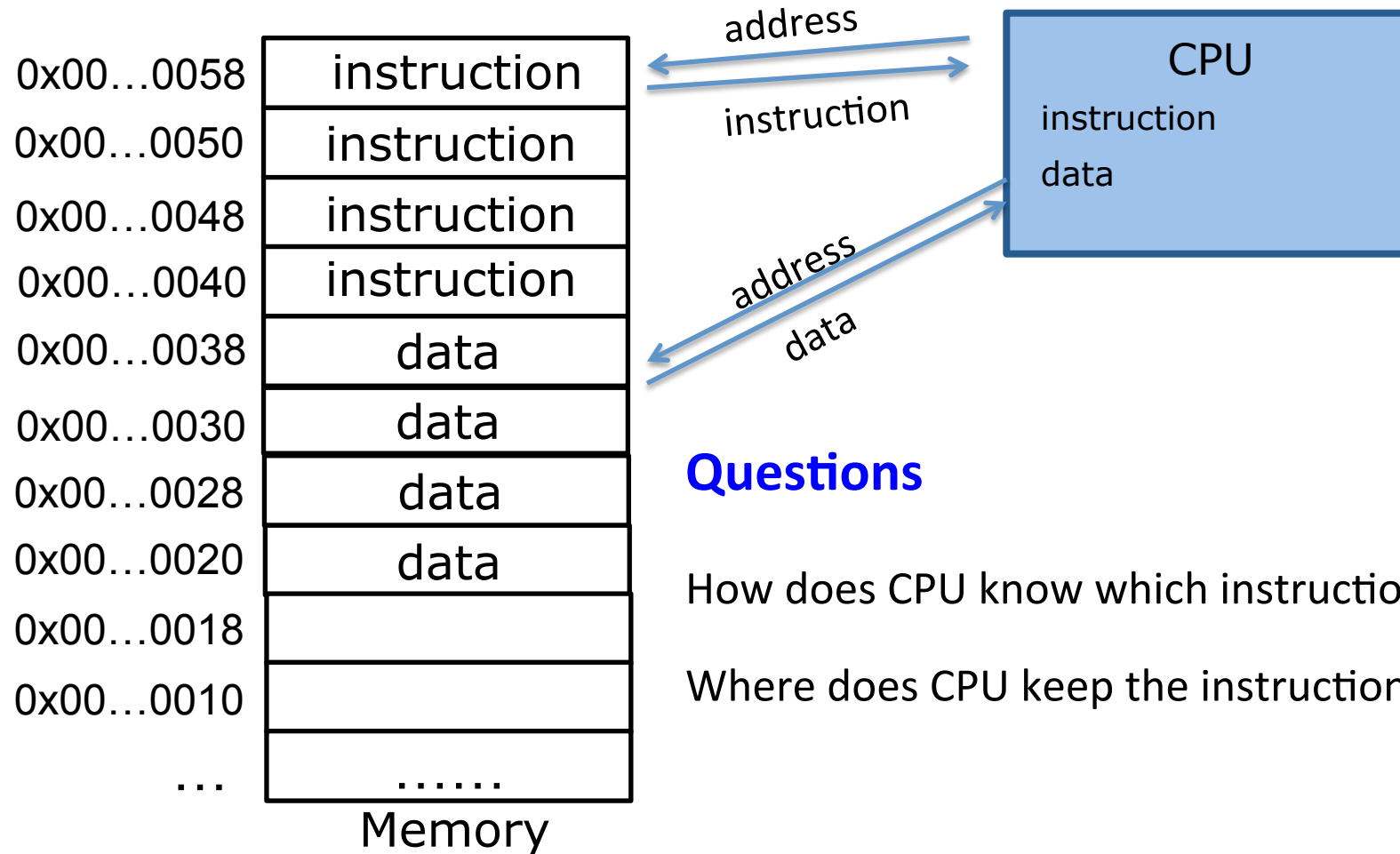
How CPU executes a program



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:

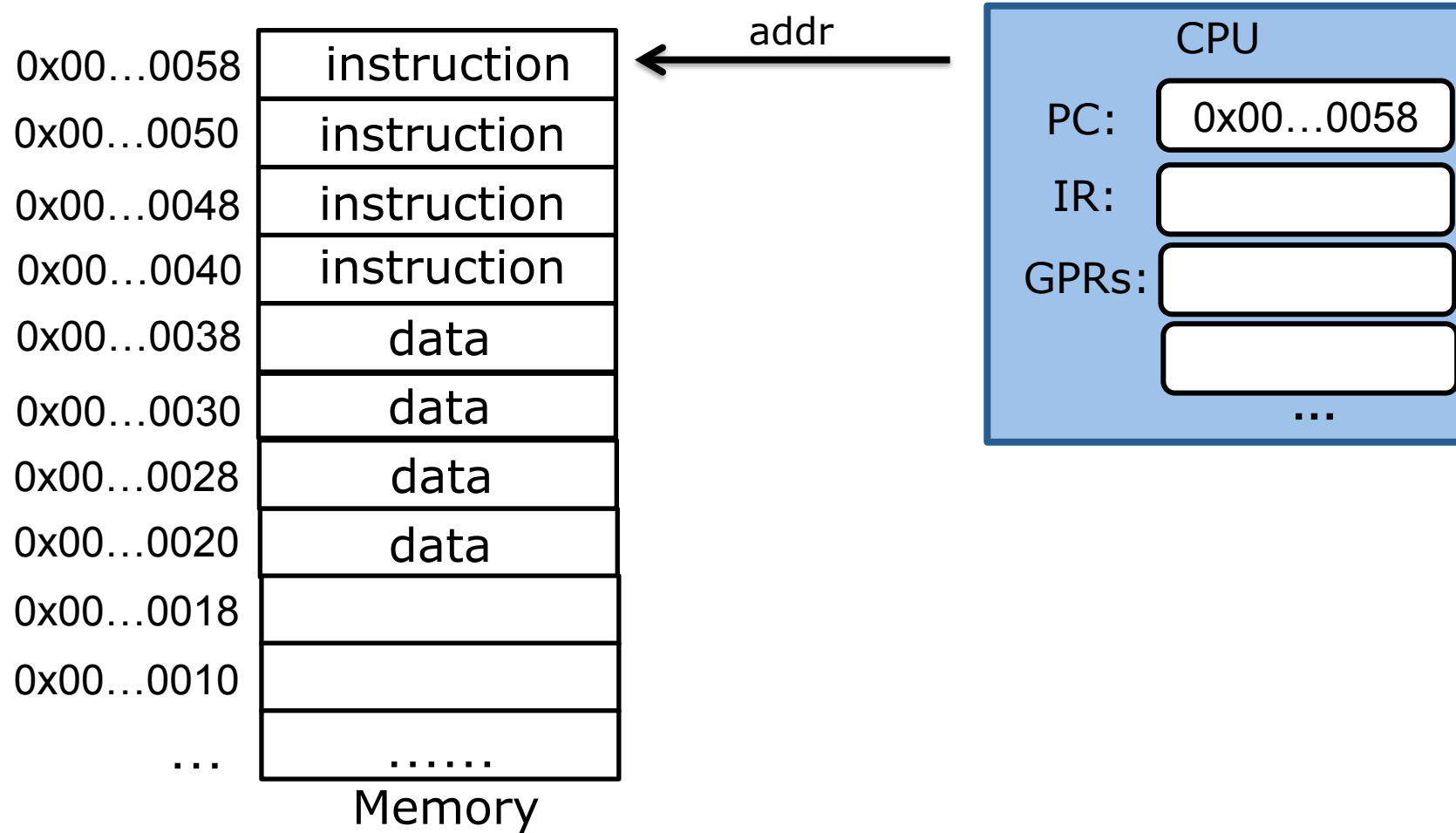
- Store operands and pointers used by program

Program status and control register:

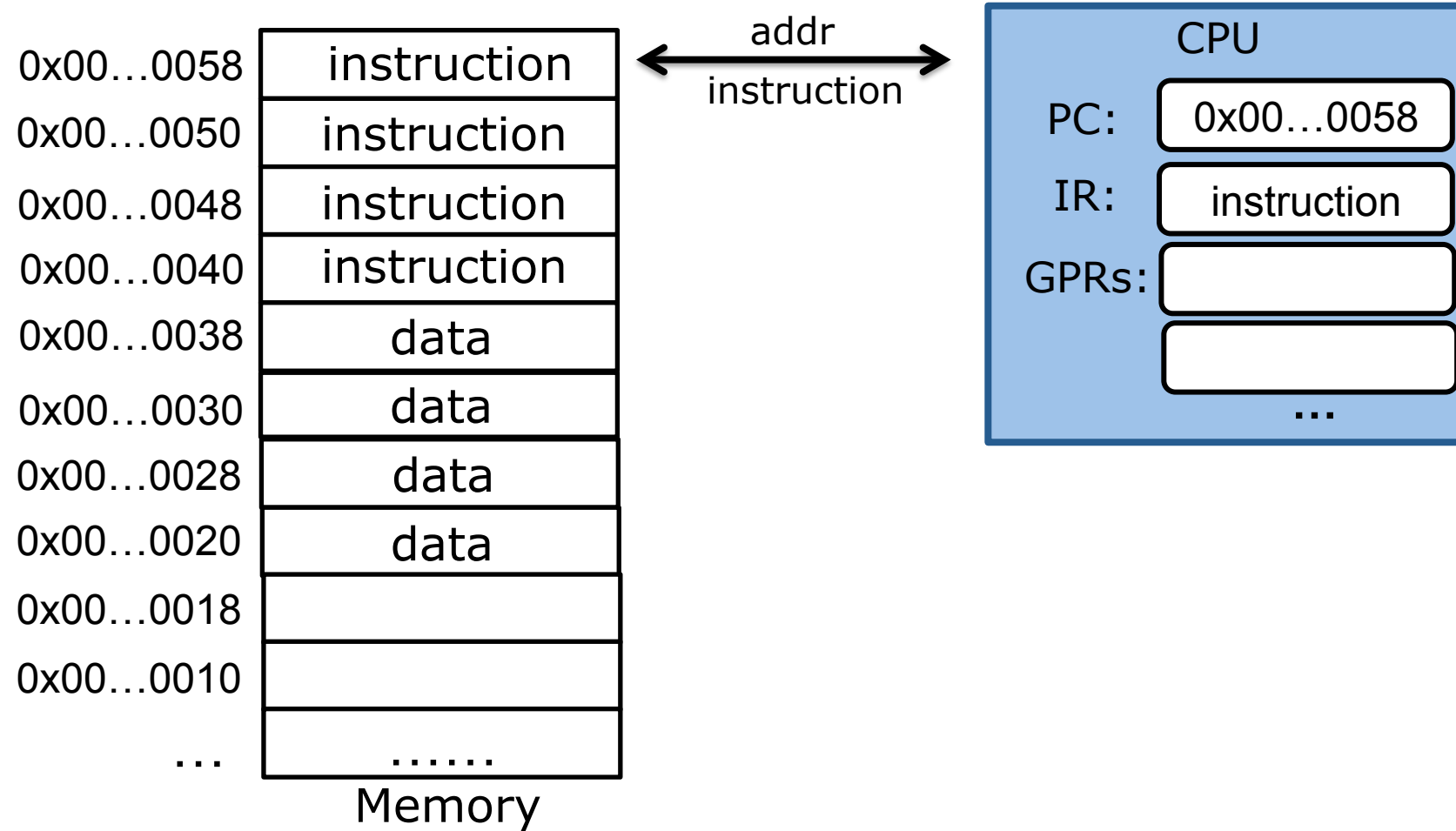
- Status of the program being executed
- All called “EFLAGS” in x86_64

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

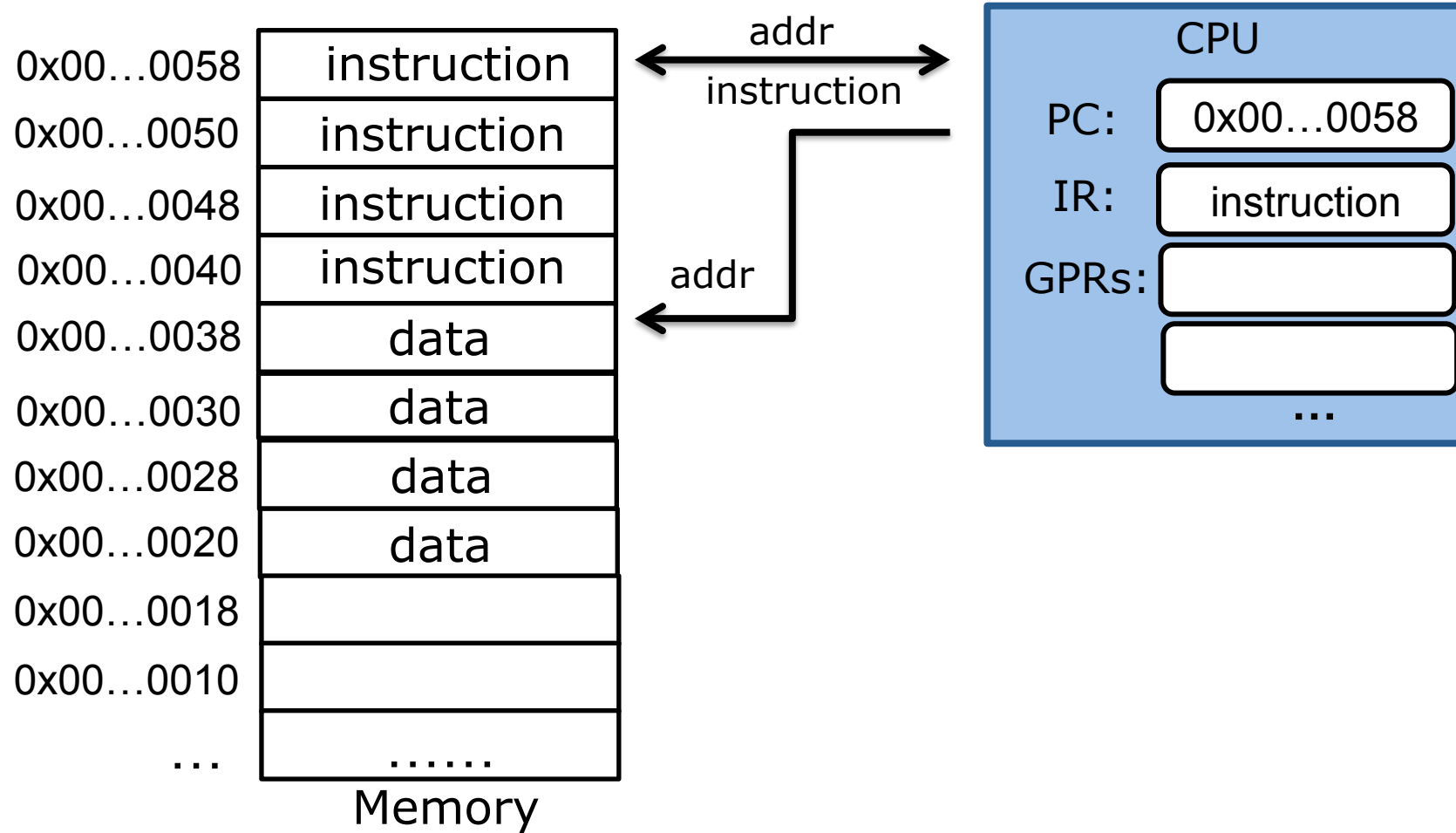
CPU execution



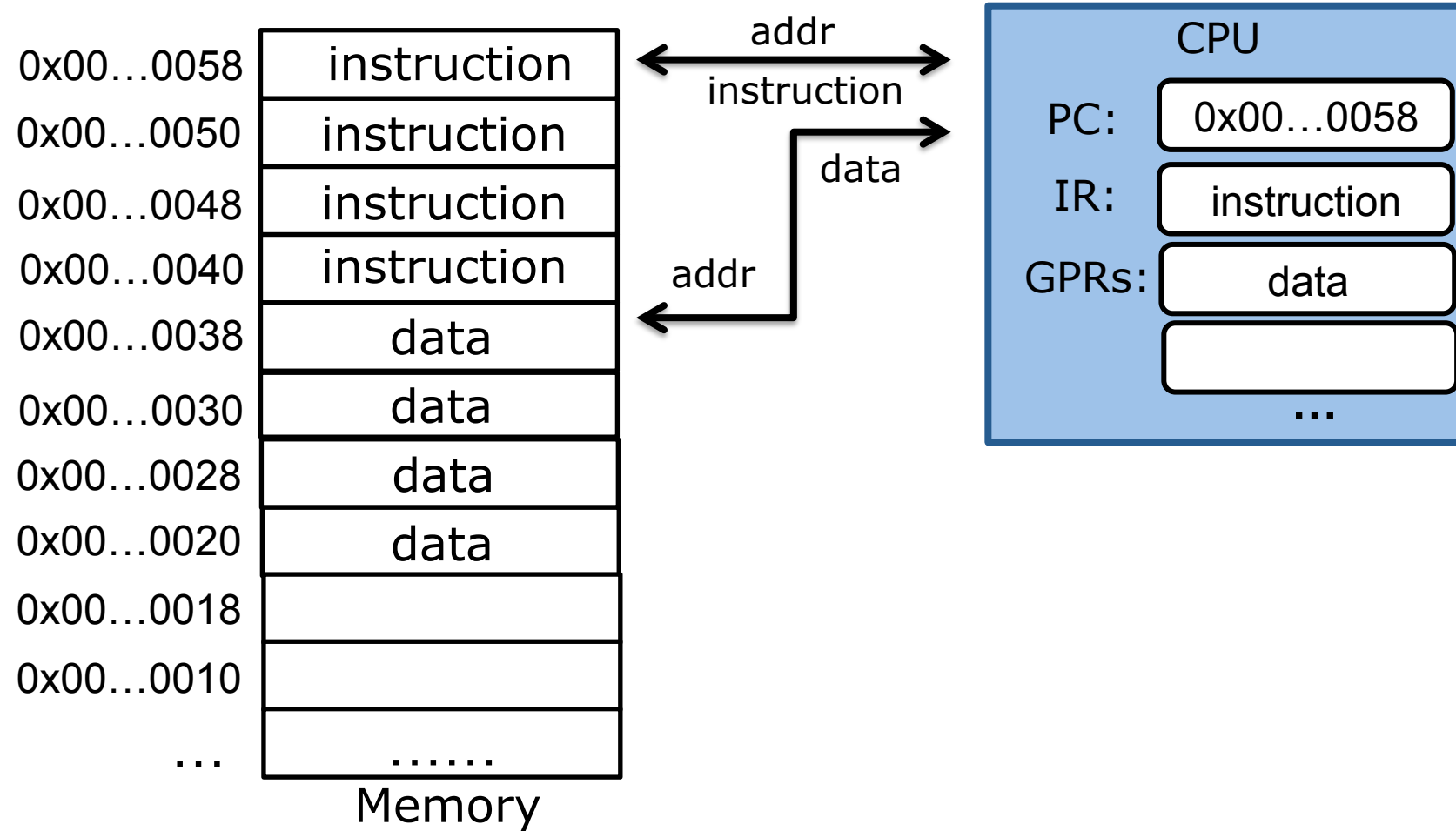
CPU execution



CPU execution



CPU execution



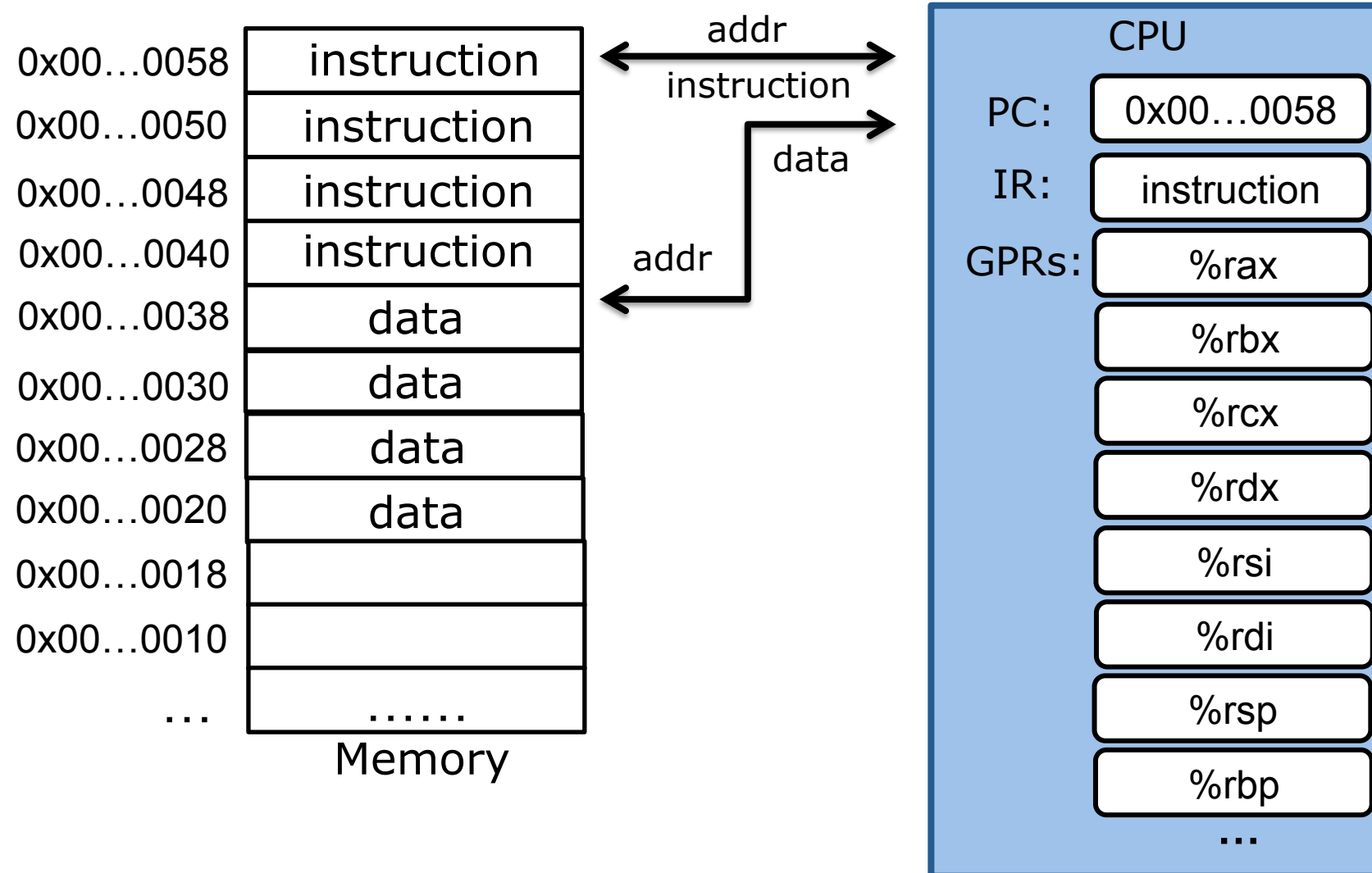
General Purpose Registers (intel x86-64)

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

8 bytes

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

CPU execution



General Purpose Registers (intel x86-64)

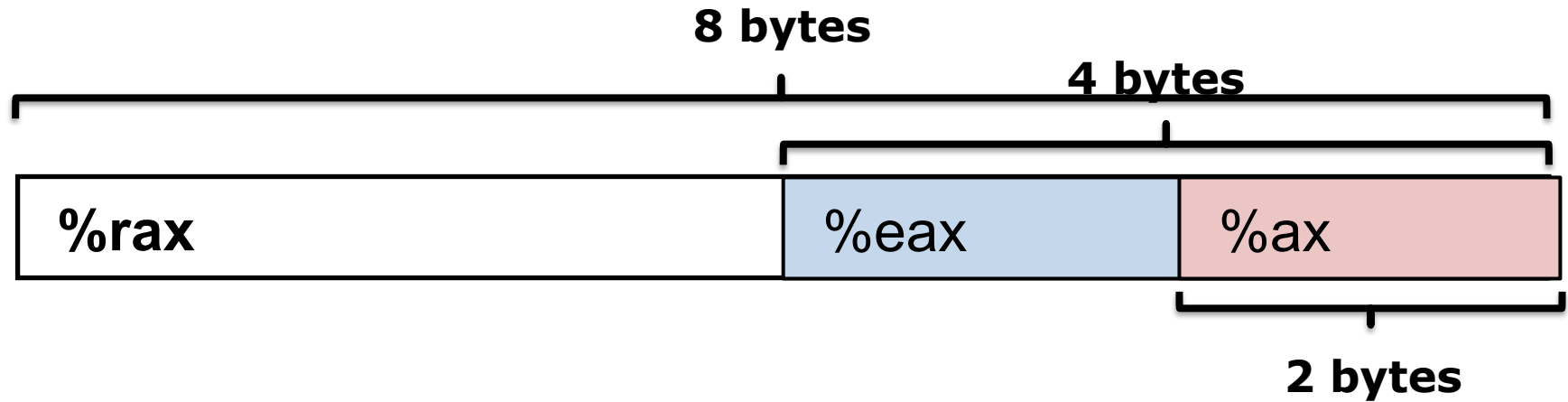
%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

8 bytes

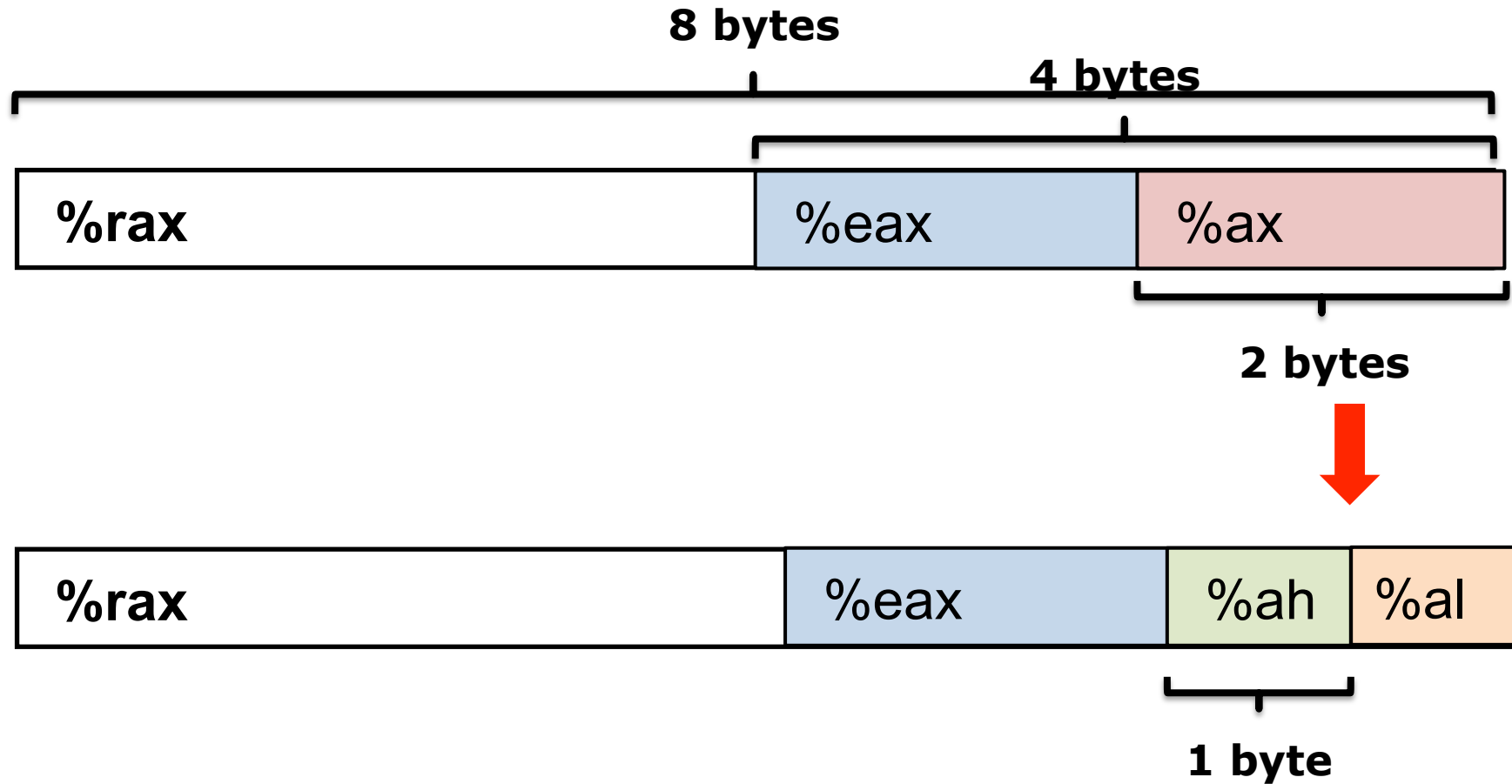
%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

4 bytes

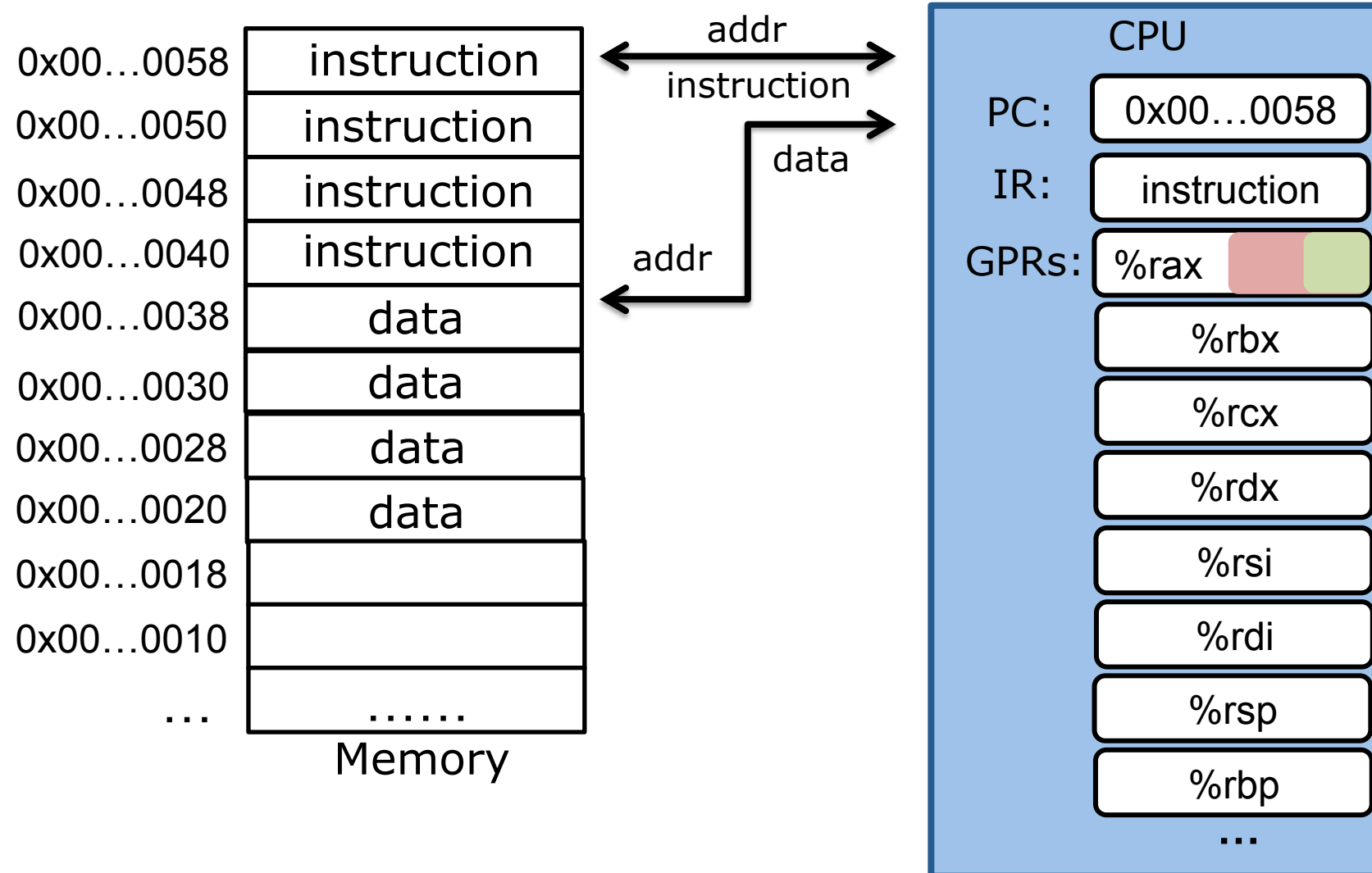
Use %rax as an example



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

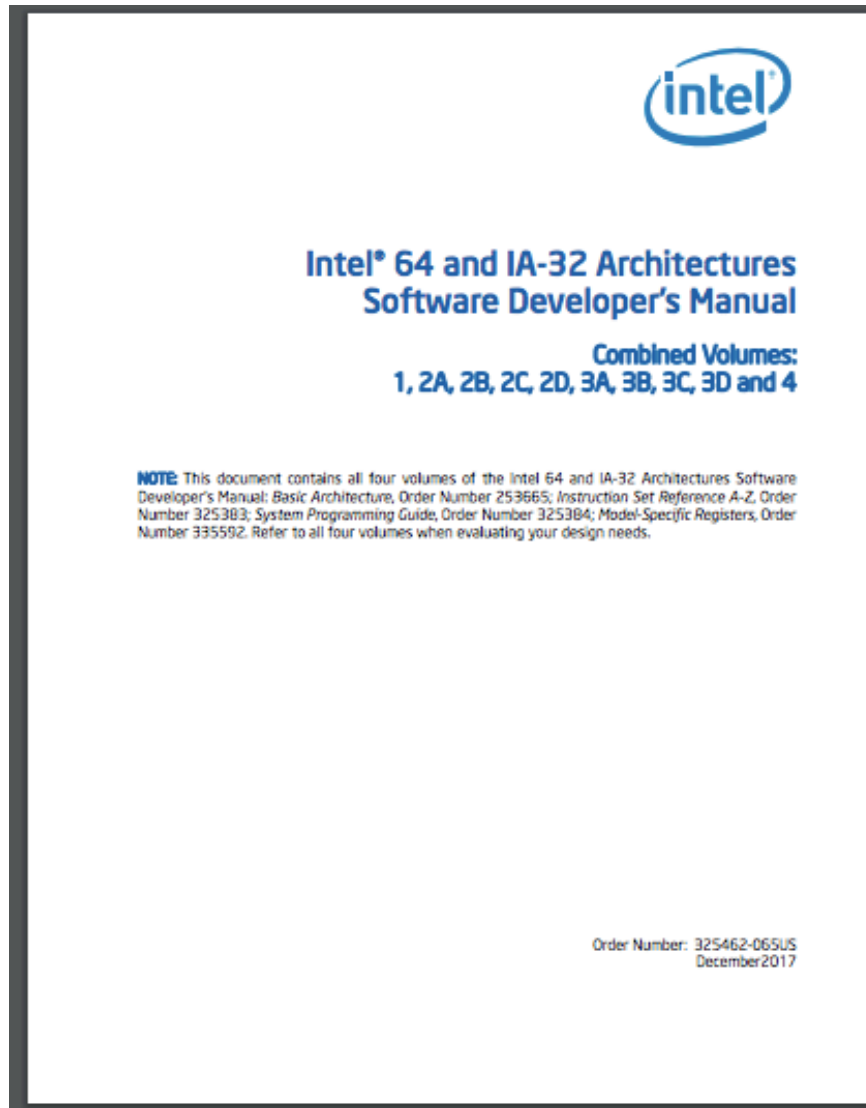
- 64-bit version of x86

← taught by CSO

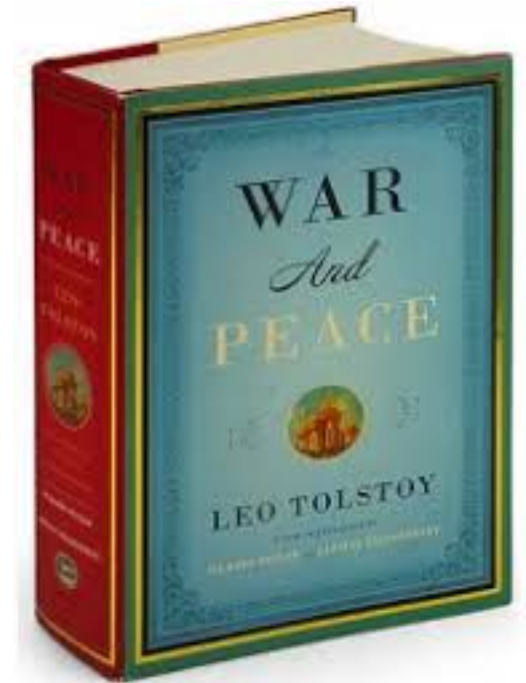
ARM is another common ISA

- Phones, tablets, Raspberry Pi

X86 ISA



= 4x



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 source operand to destination operand.

Moving data

mov *Source, Dest*

- Copy a quadword (64 bits) from source operand to destination 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 source operand to destination operand.
- For x86, a word is 16 bits due to historical reasons
 - 8086 uses 16 bits as a word
 - Subsequent intel processors are **backward compatible** with earlier ones
 - Allows binary files compiled for older processors to run unmodified on new 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

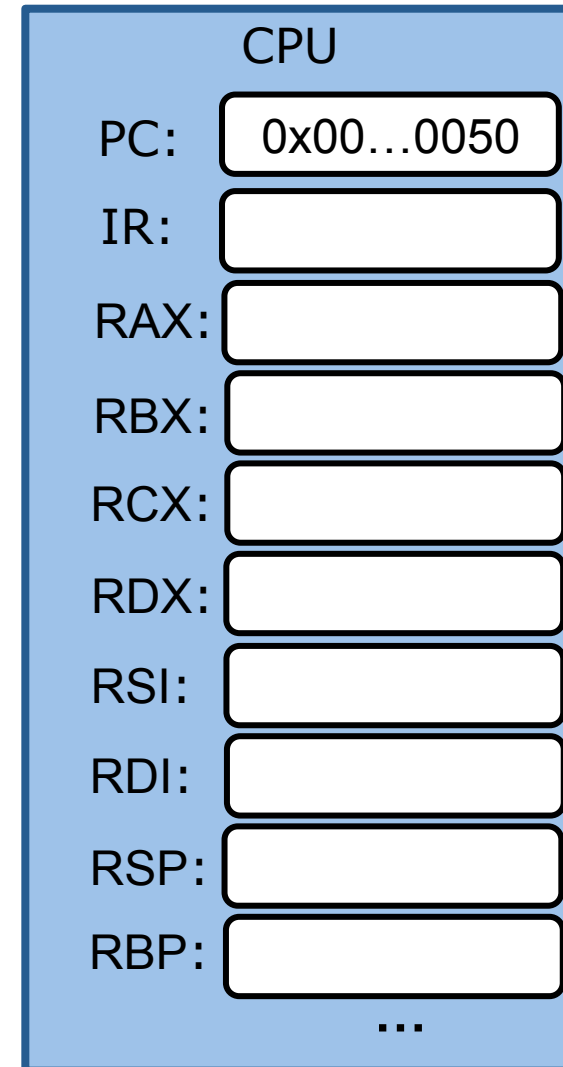
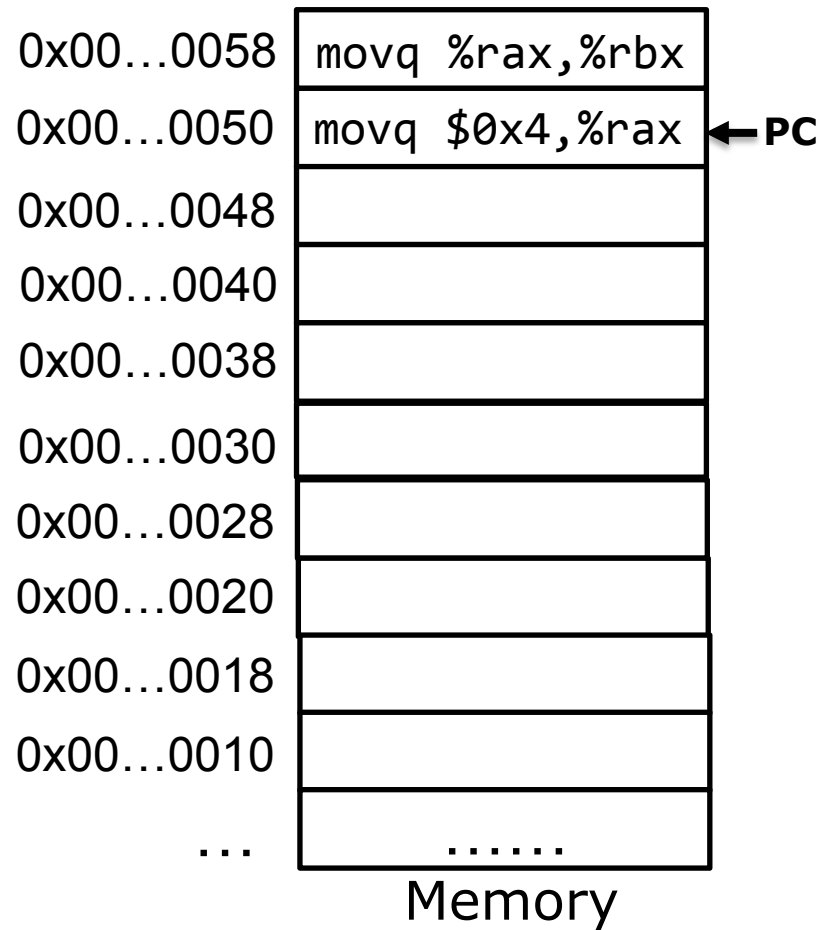
1. Immediate can only be *Source*

2. No memory-memory mov

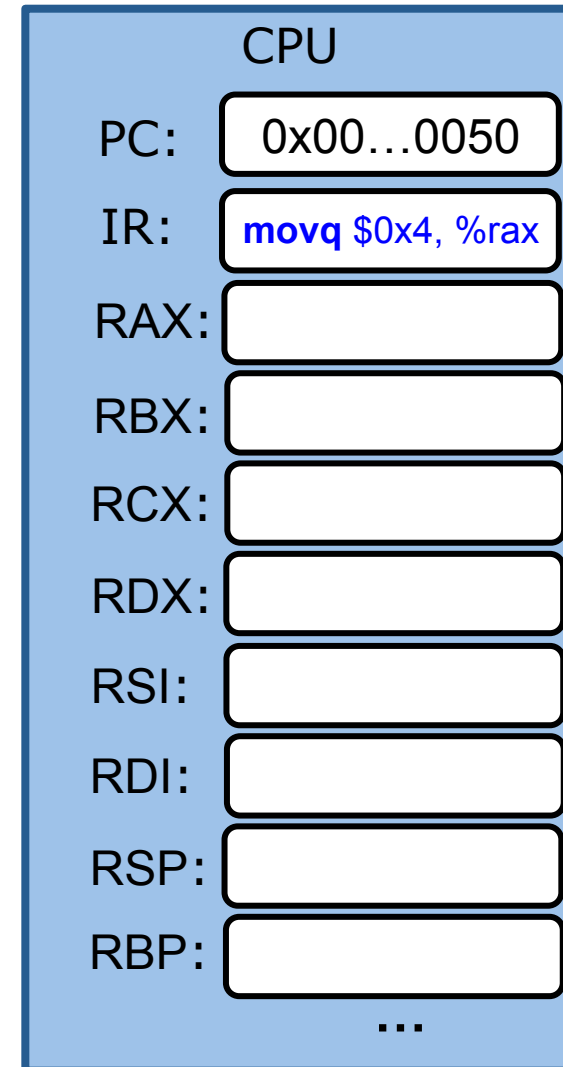
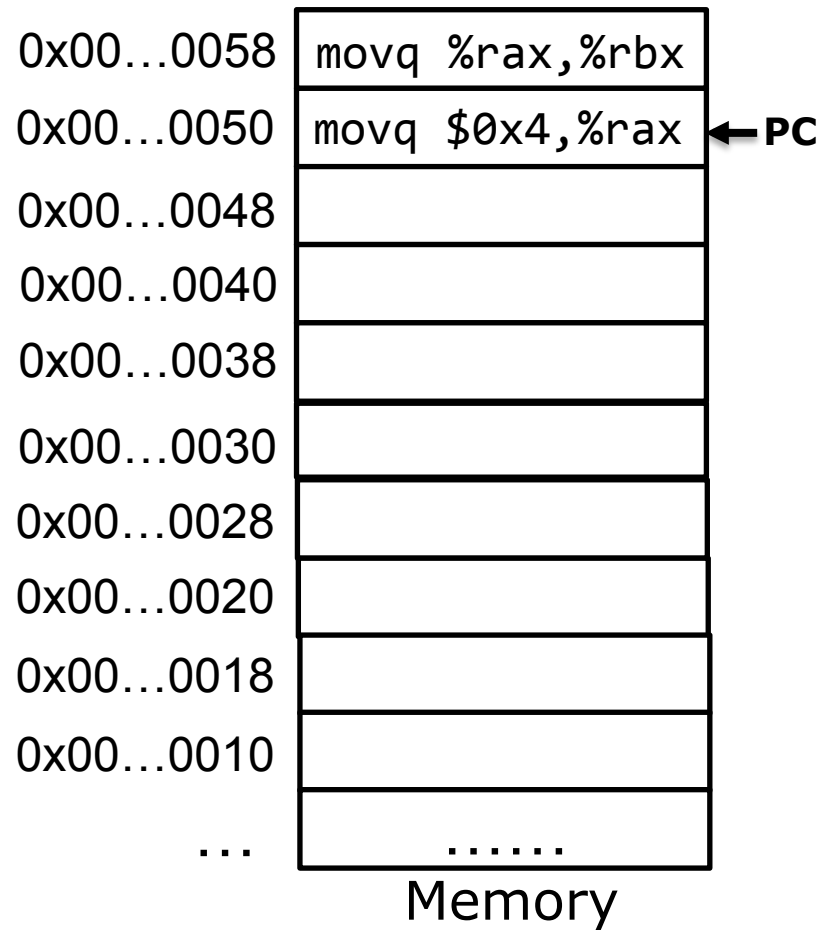
How CPU executes an instruction

1. PC contains the instruction's address
2. Load the instruction into IR
3. Execute the instruction
4. CPU automatically increments PC by the size of the executed (retired) instruction.

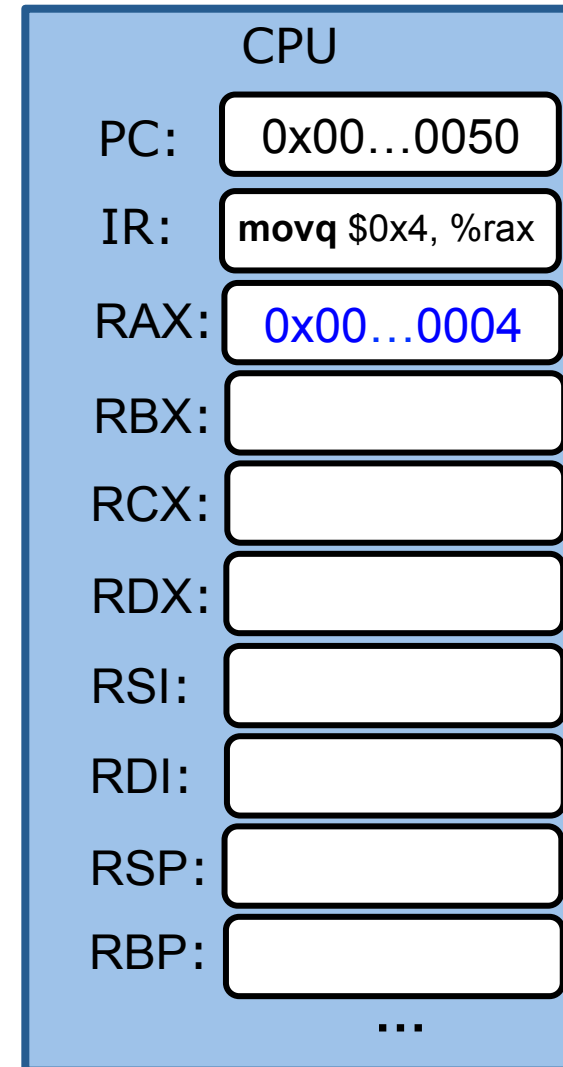
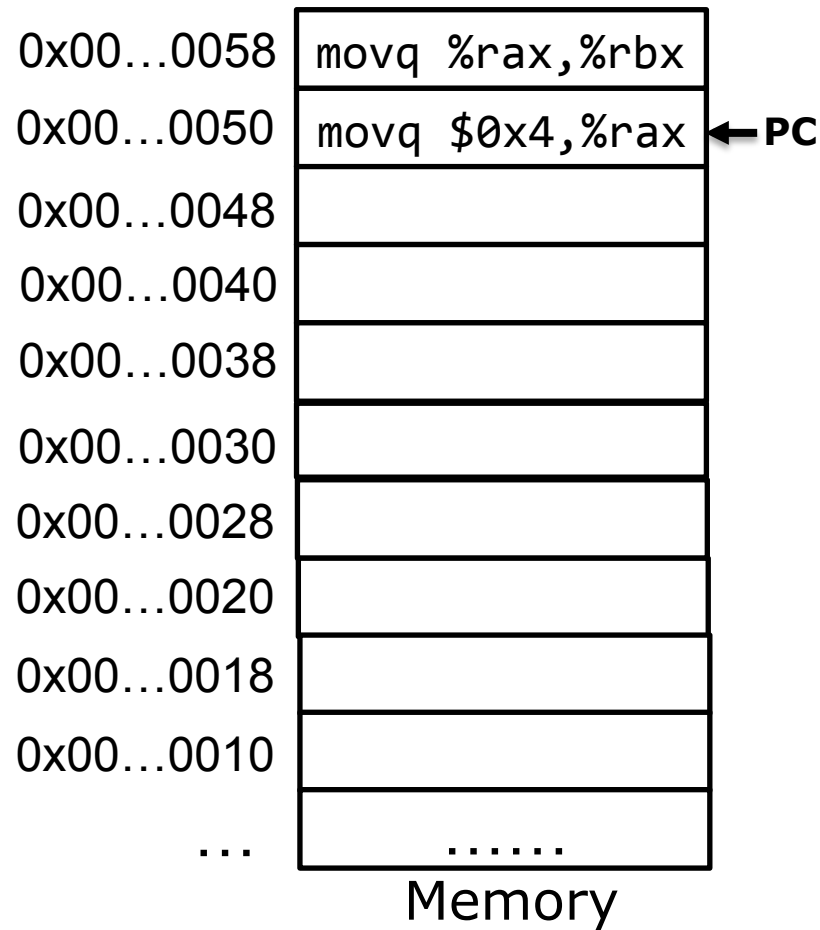
movq Imm, Reg



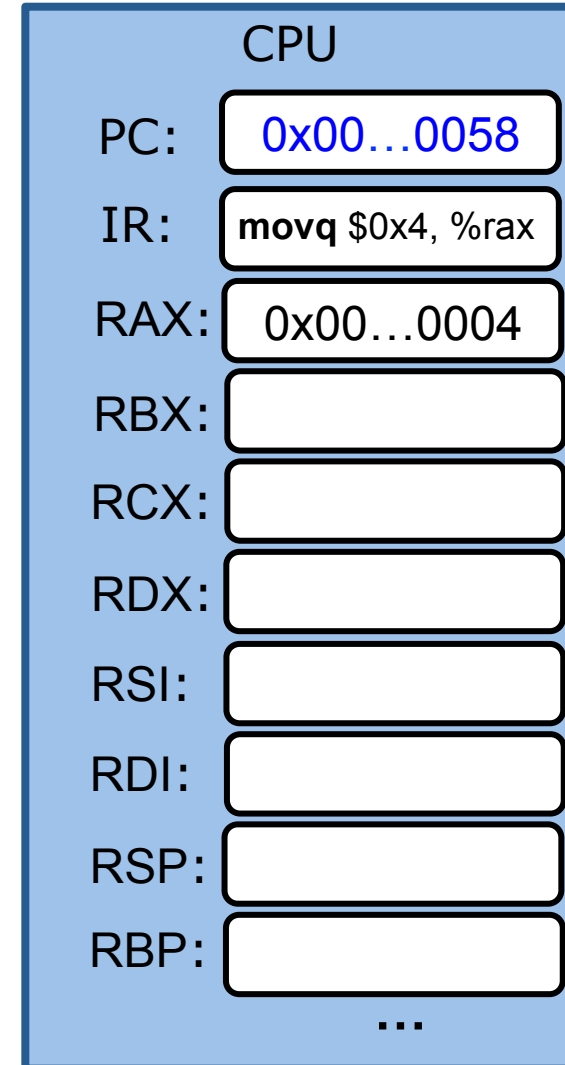
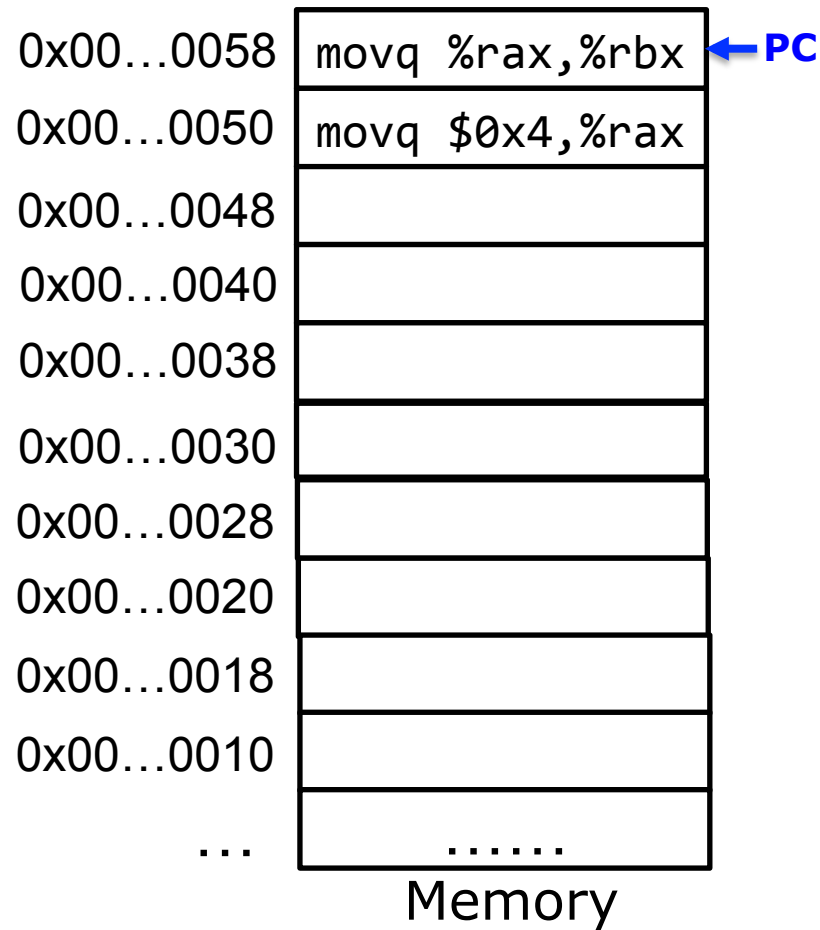
movq Imm, Reg



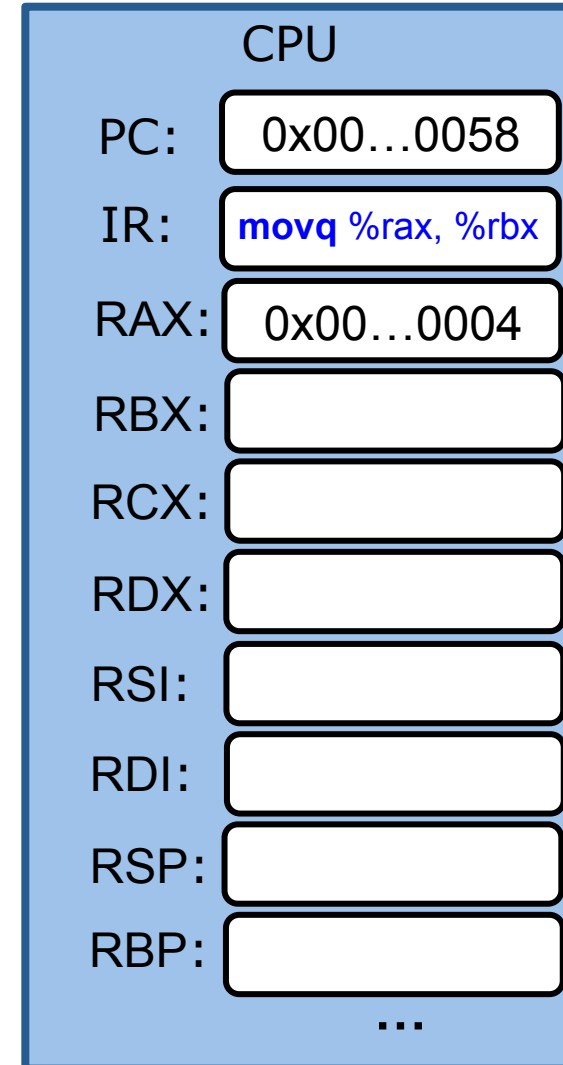
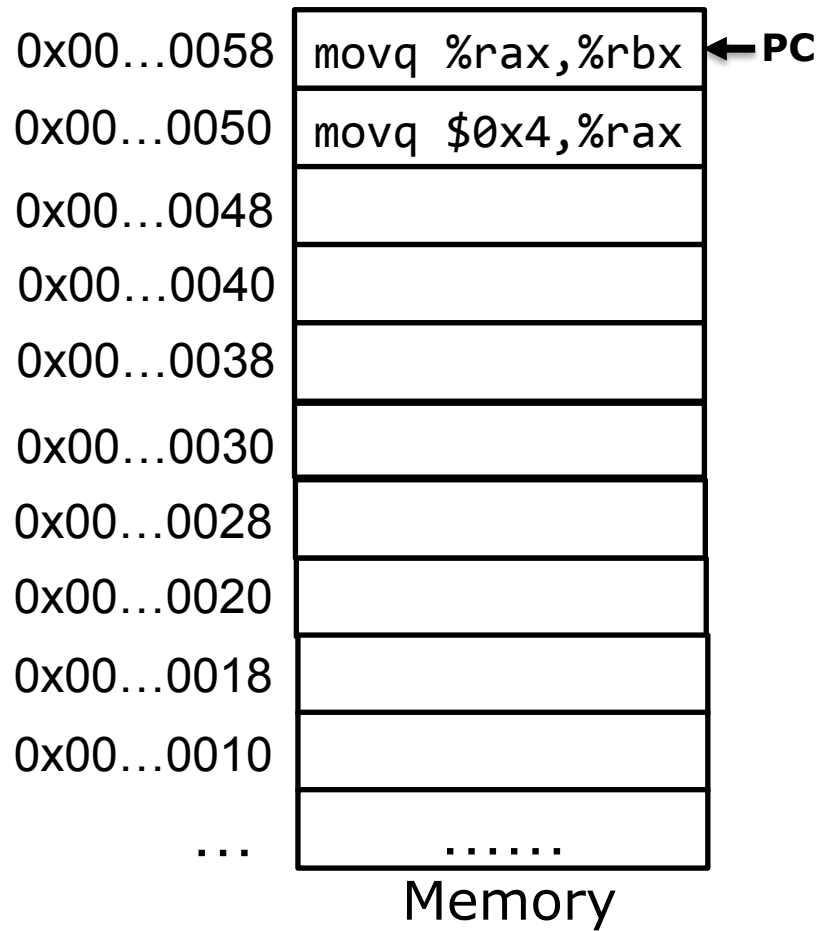
movq Imm, Reg



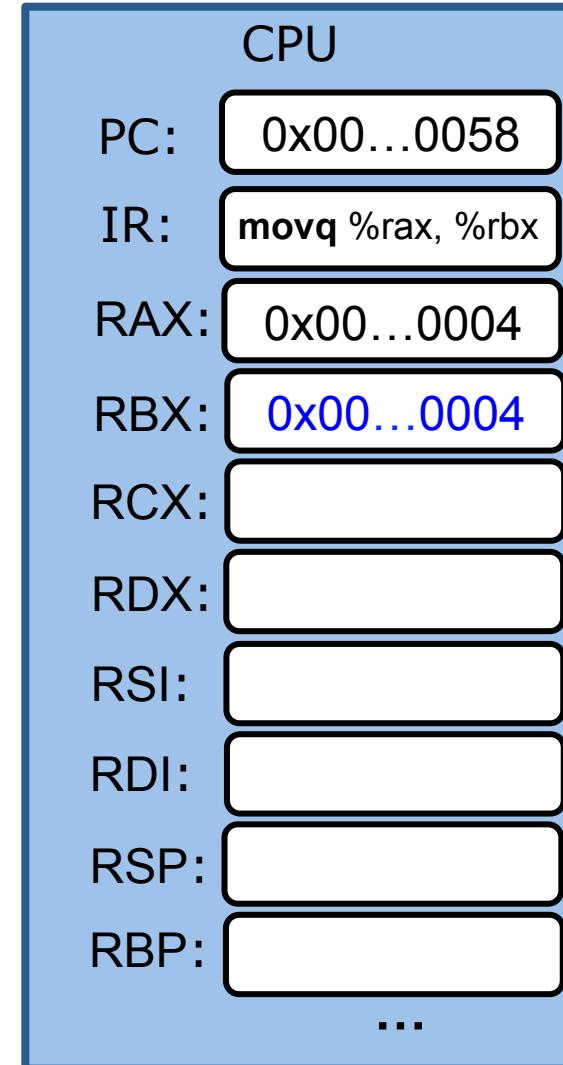
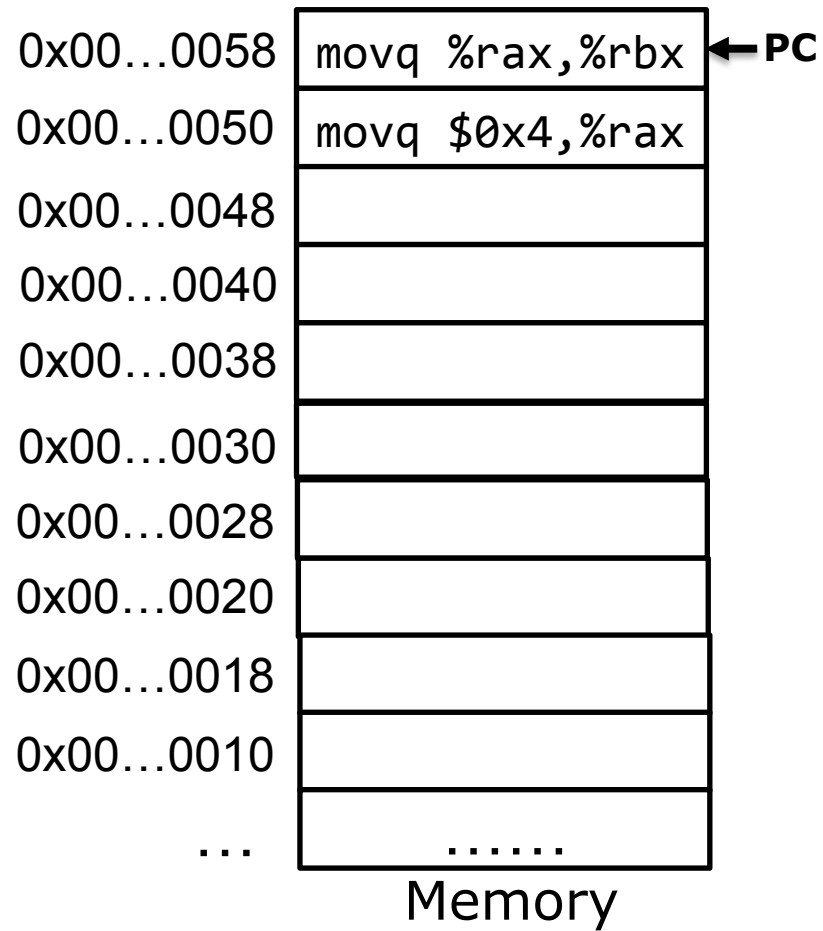
movq *Reg, Reg*



movq *Reg, Reg*



`movq Reg, Reg`



movq *Mem, Reg*

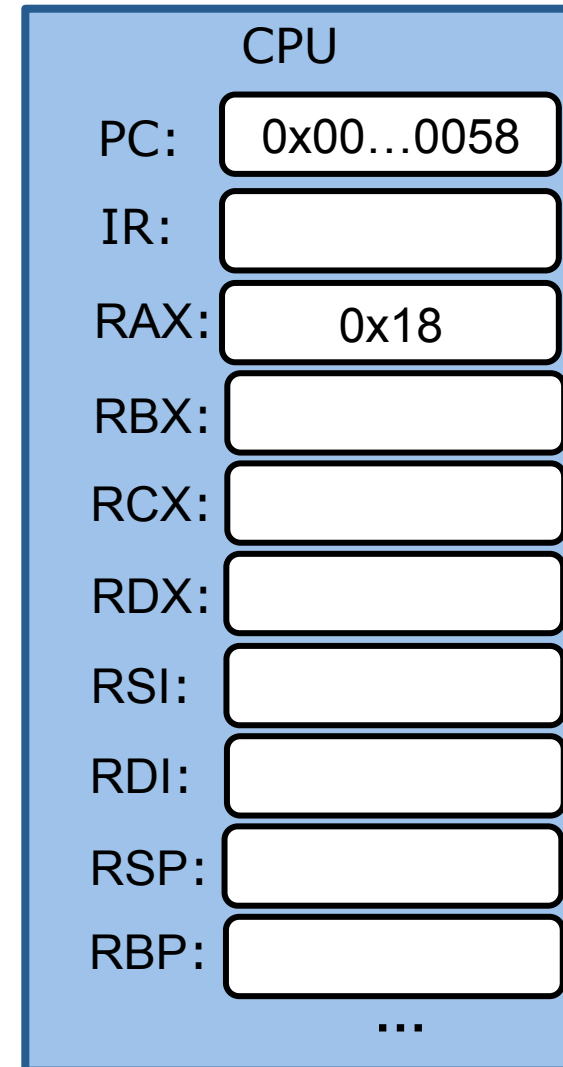
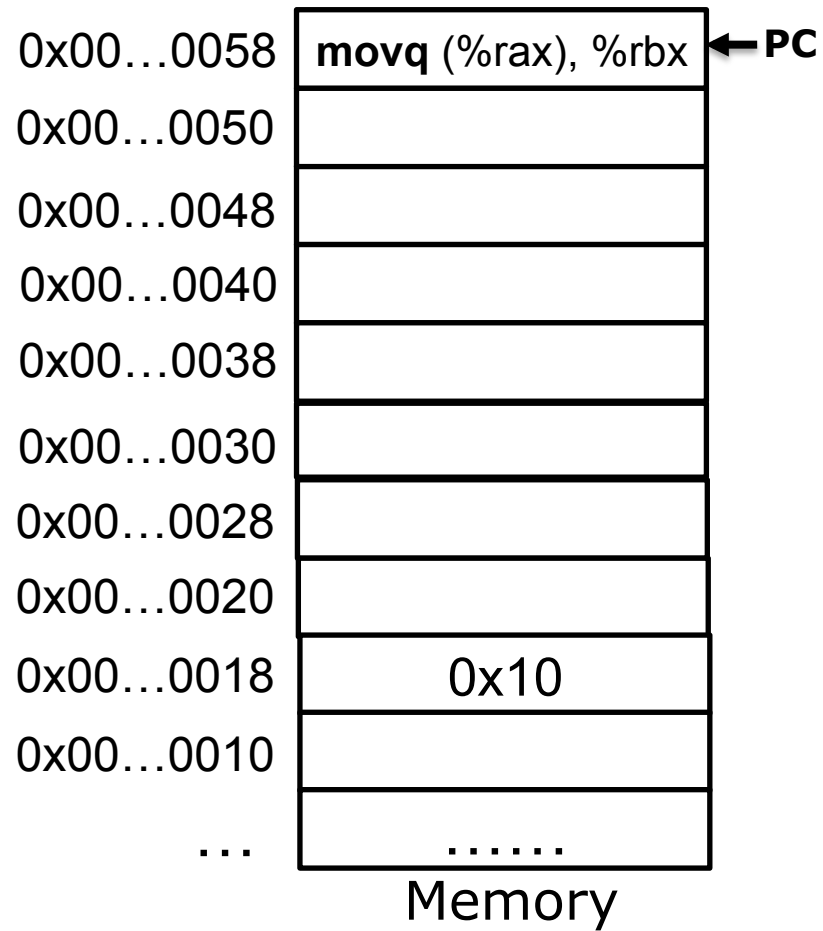
How to represent a “memory” operand?

Direct addressing: use registers to index the memory

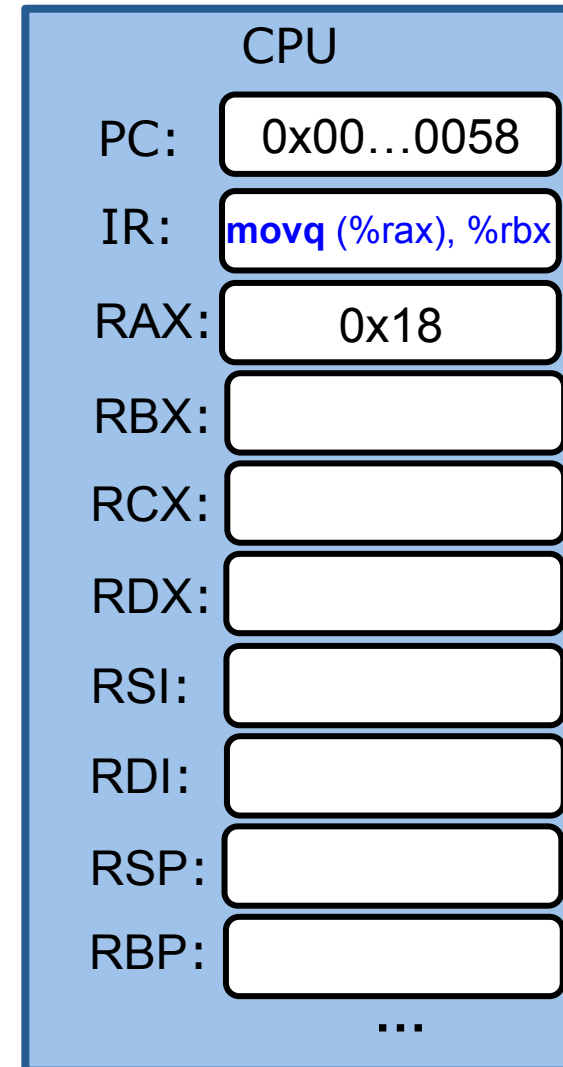
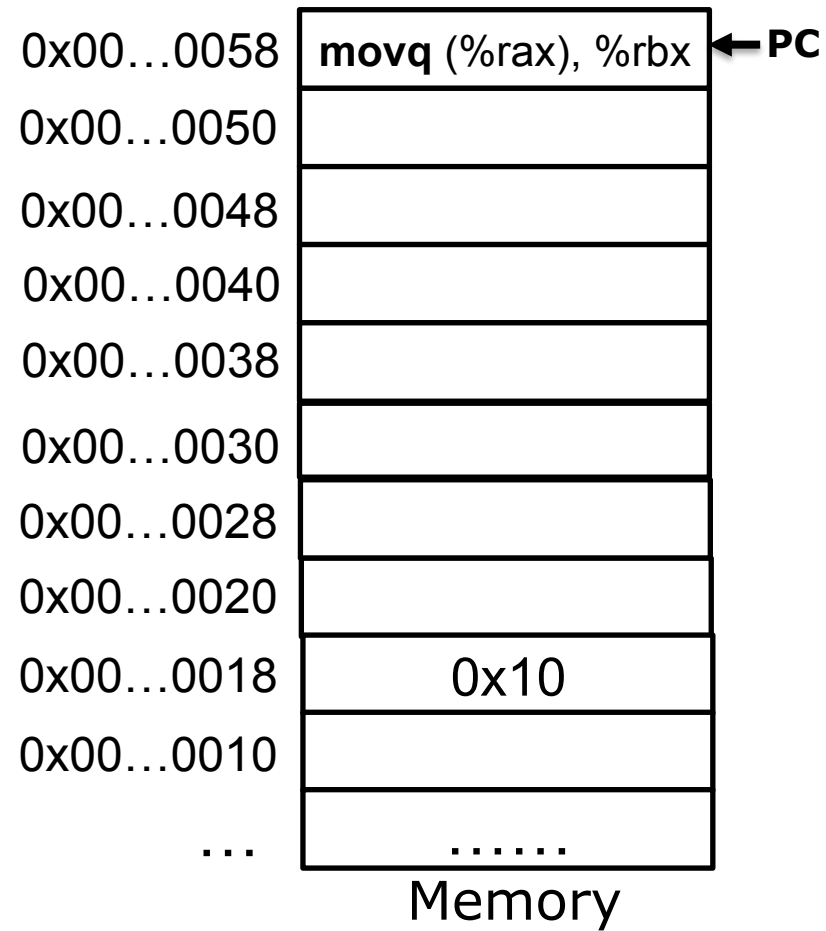
(Register)

- Register contains a memory address
- `movq (%rax), %rbx`

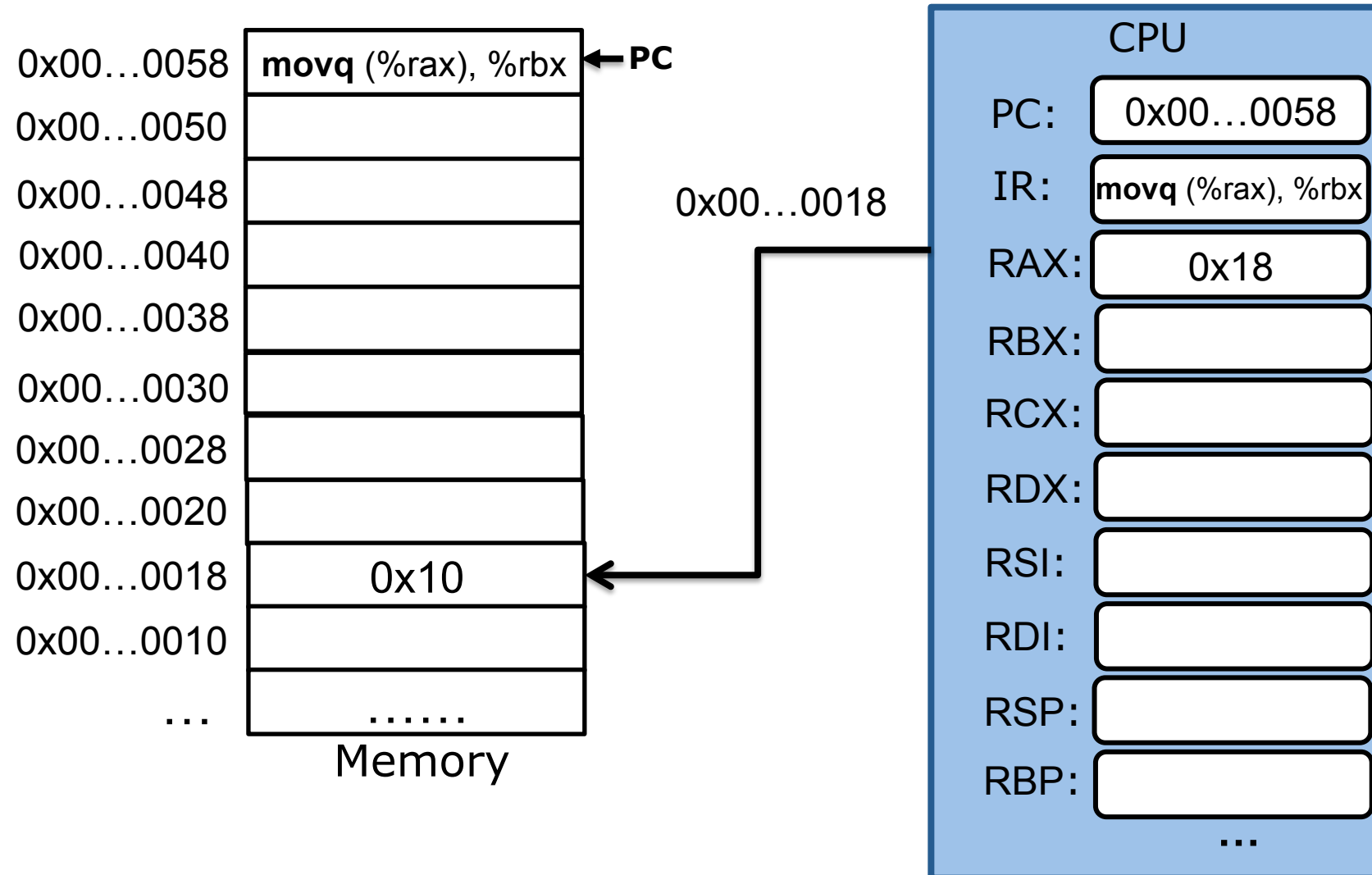
movq (%rax), %rbx



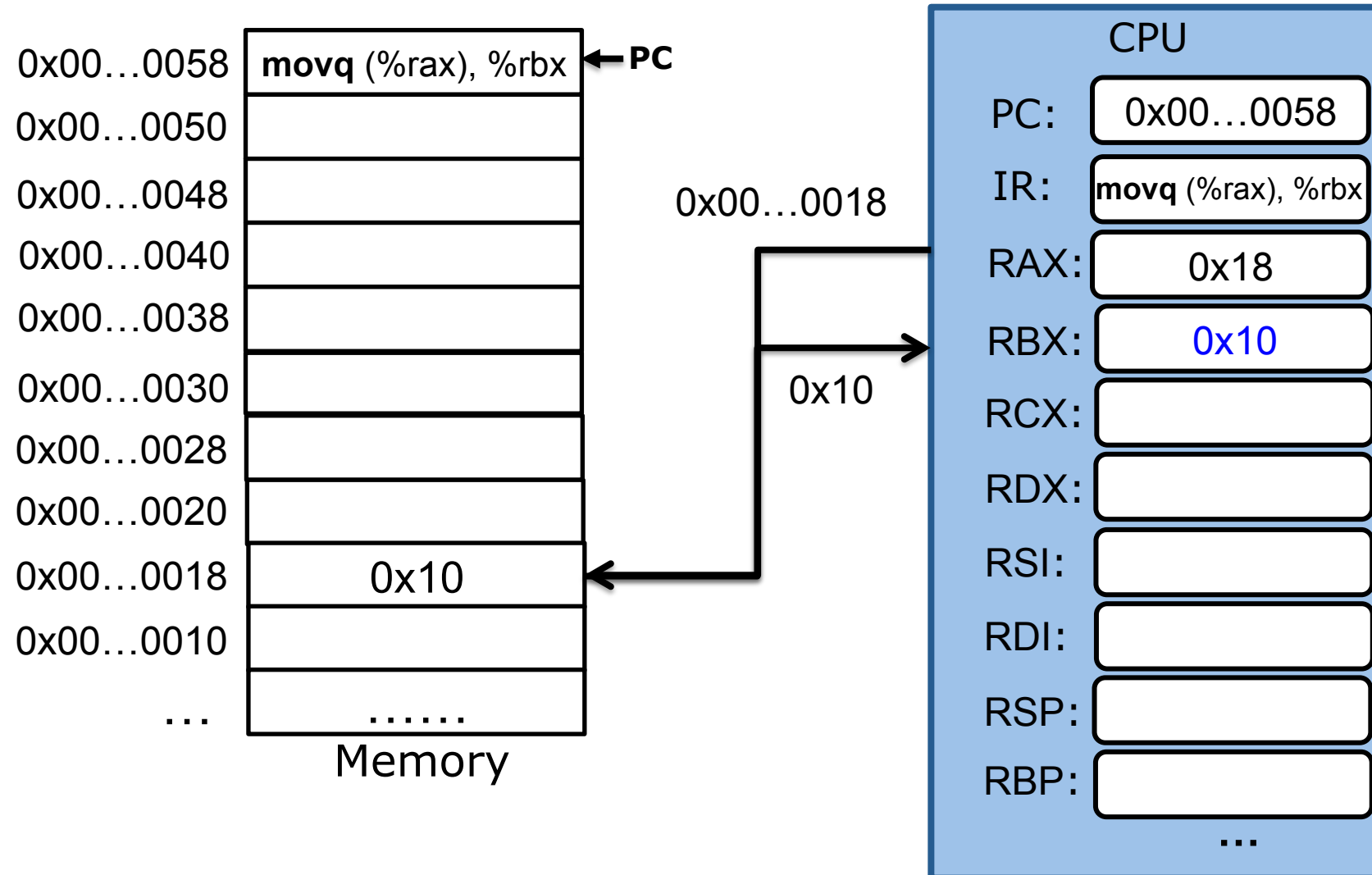
movq (%rax), %rbx



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

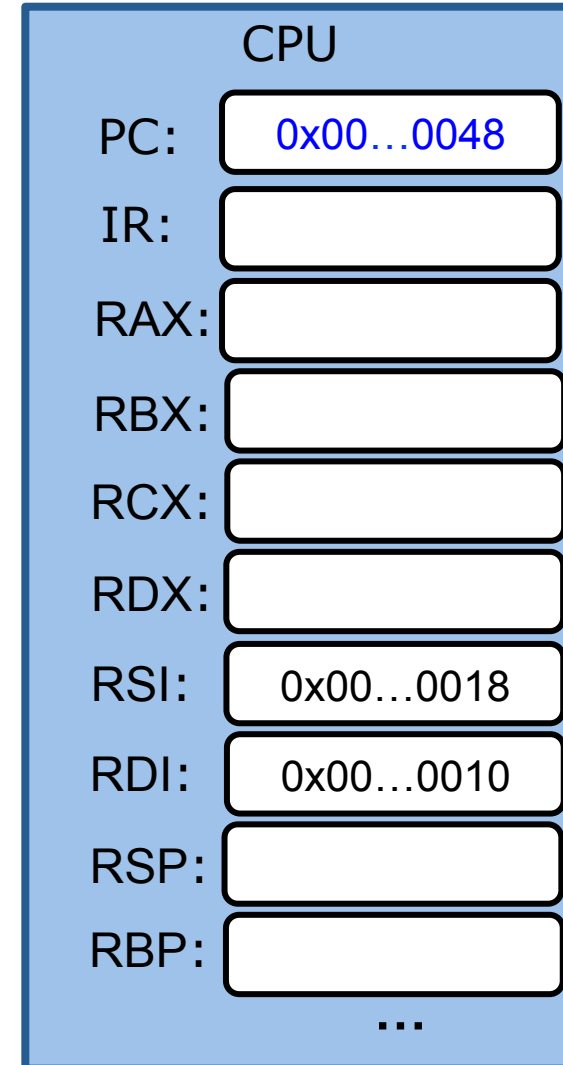
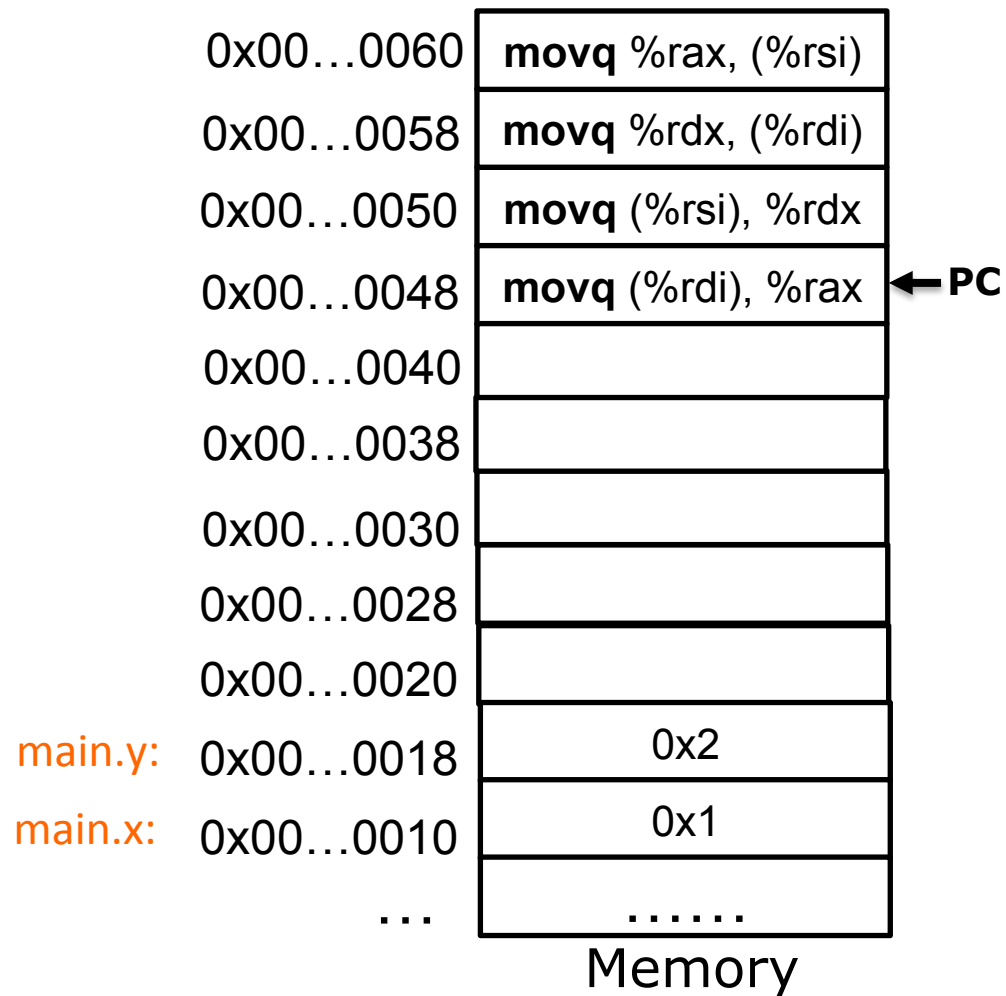
```
void  
swap(long *a, long* b) {  
  
    long tmp = *a;  
    *a = *b;  
    *b = tmp;  
  
}
```



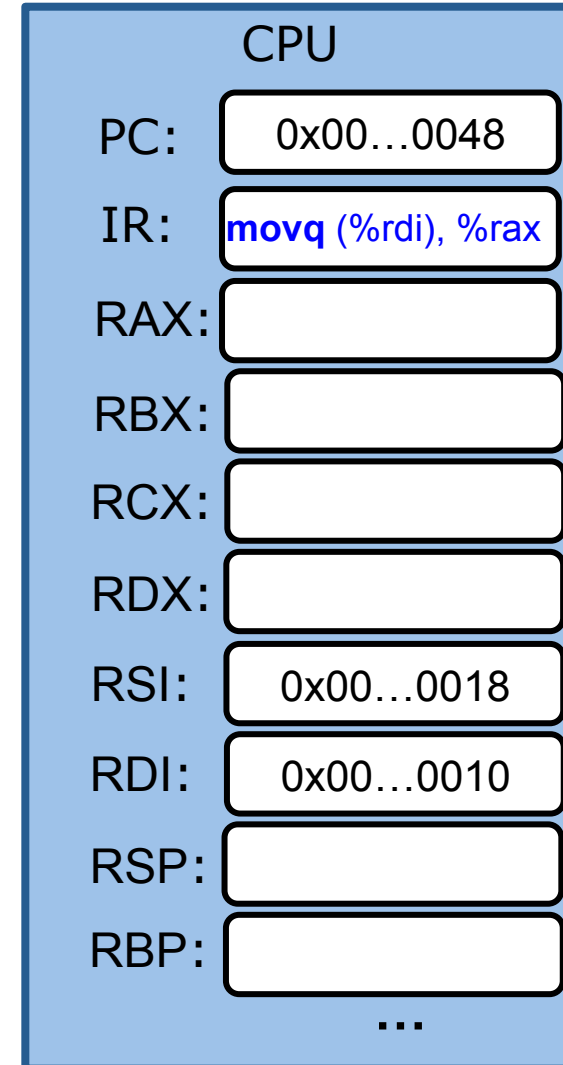
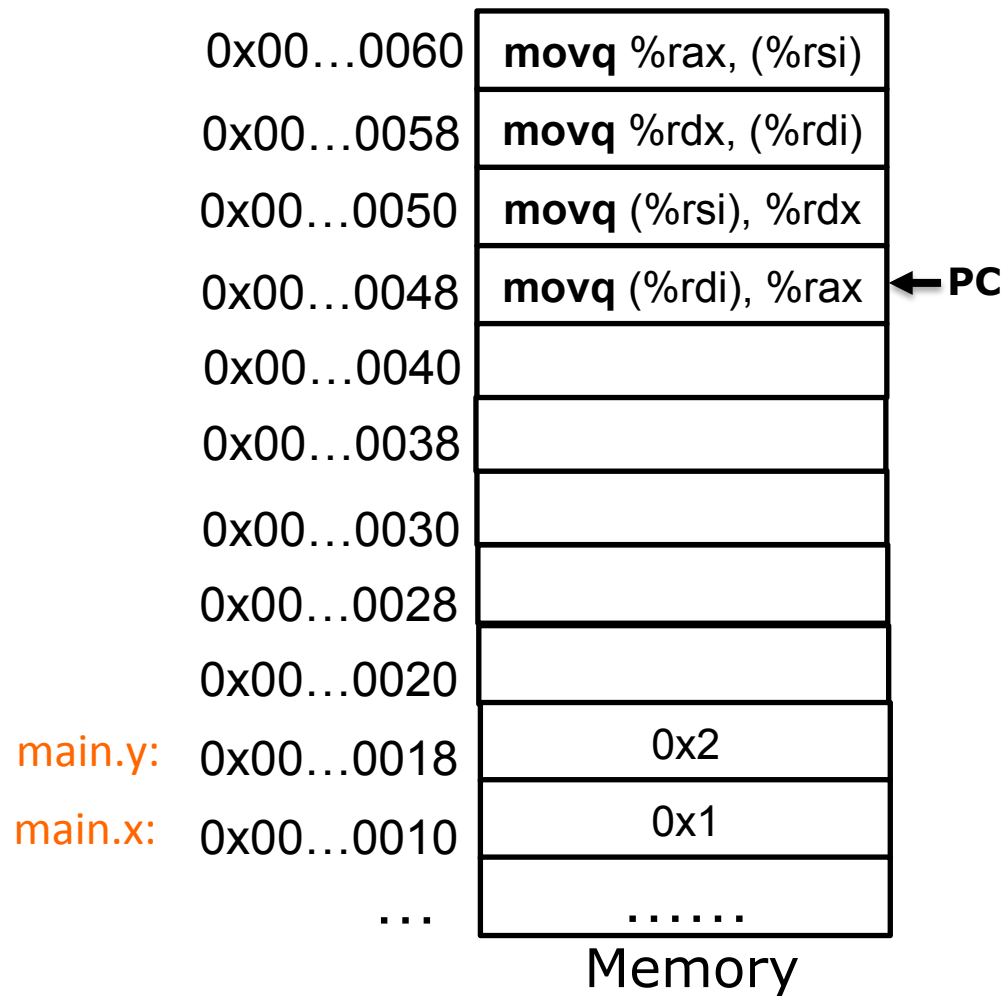
gcc -S -O3 swap.c

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

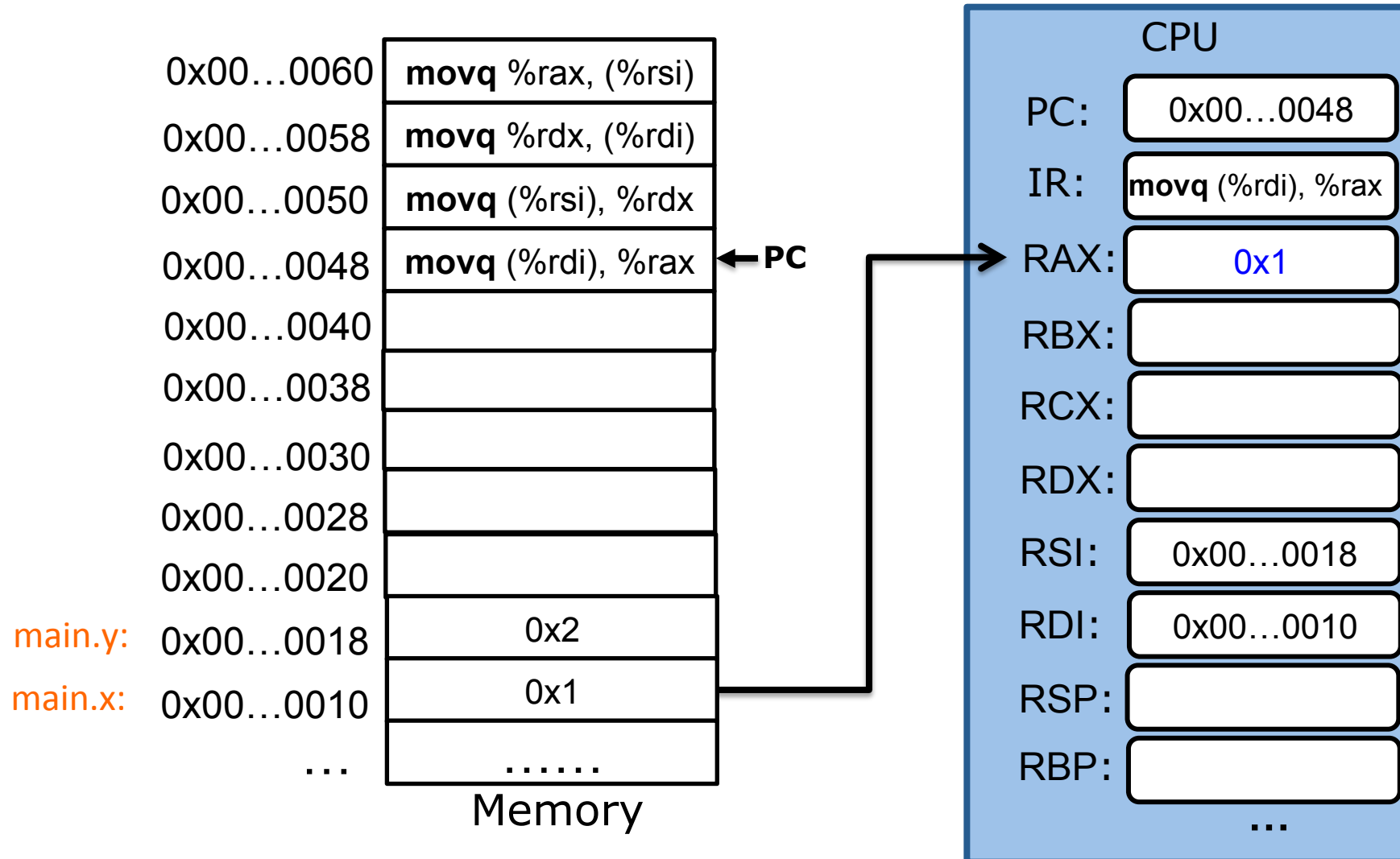
swap func



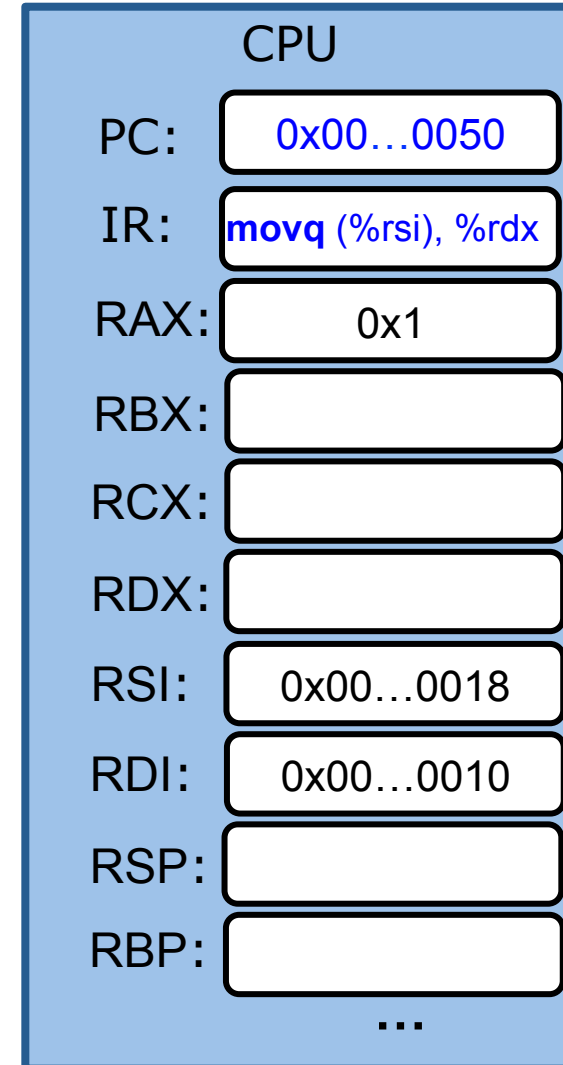
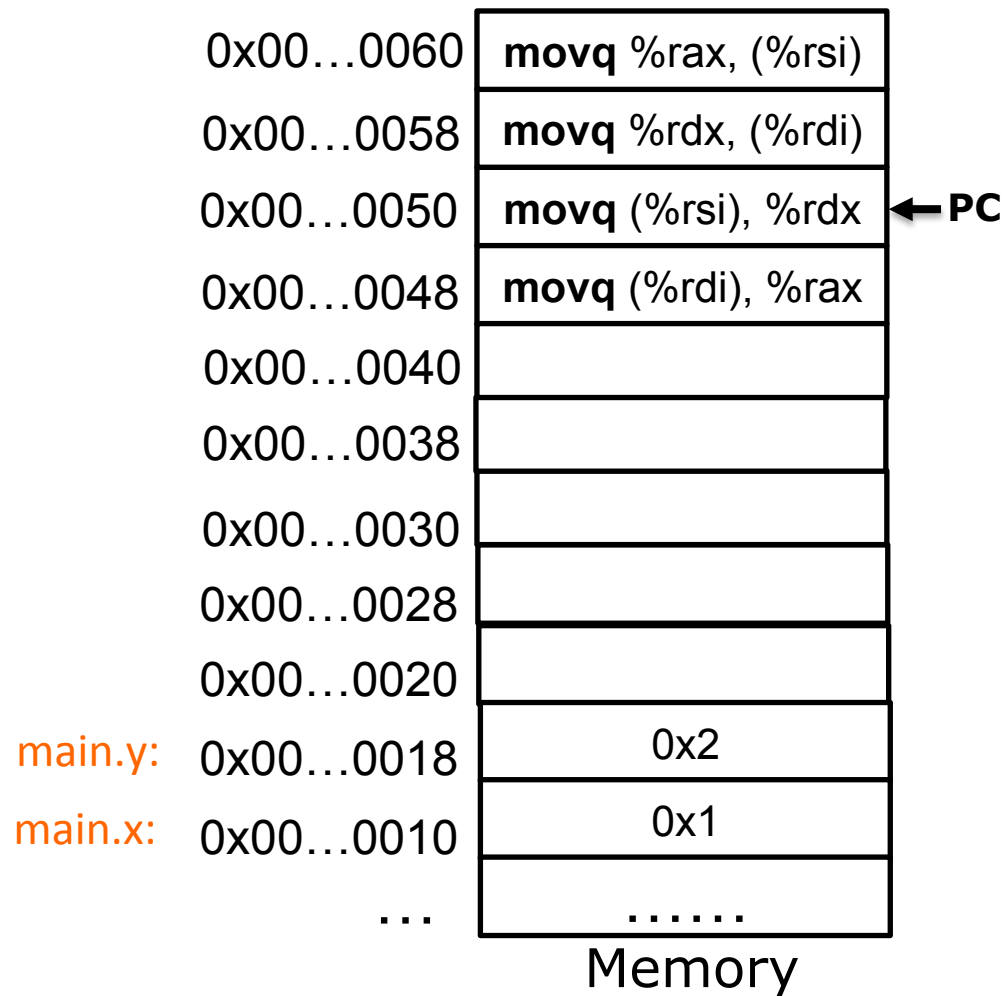
swap func



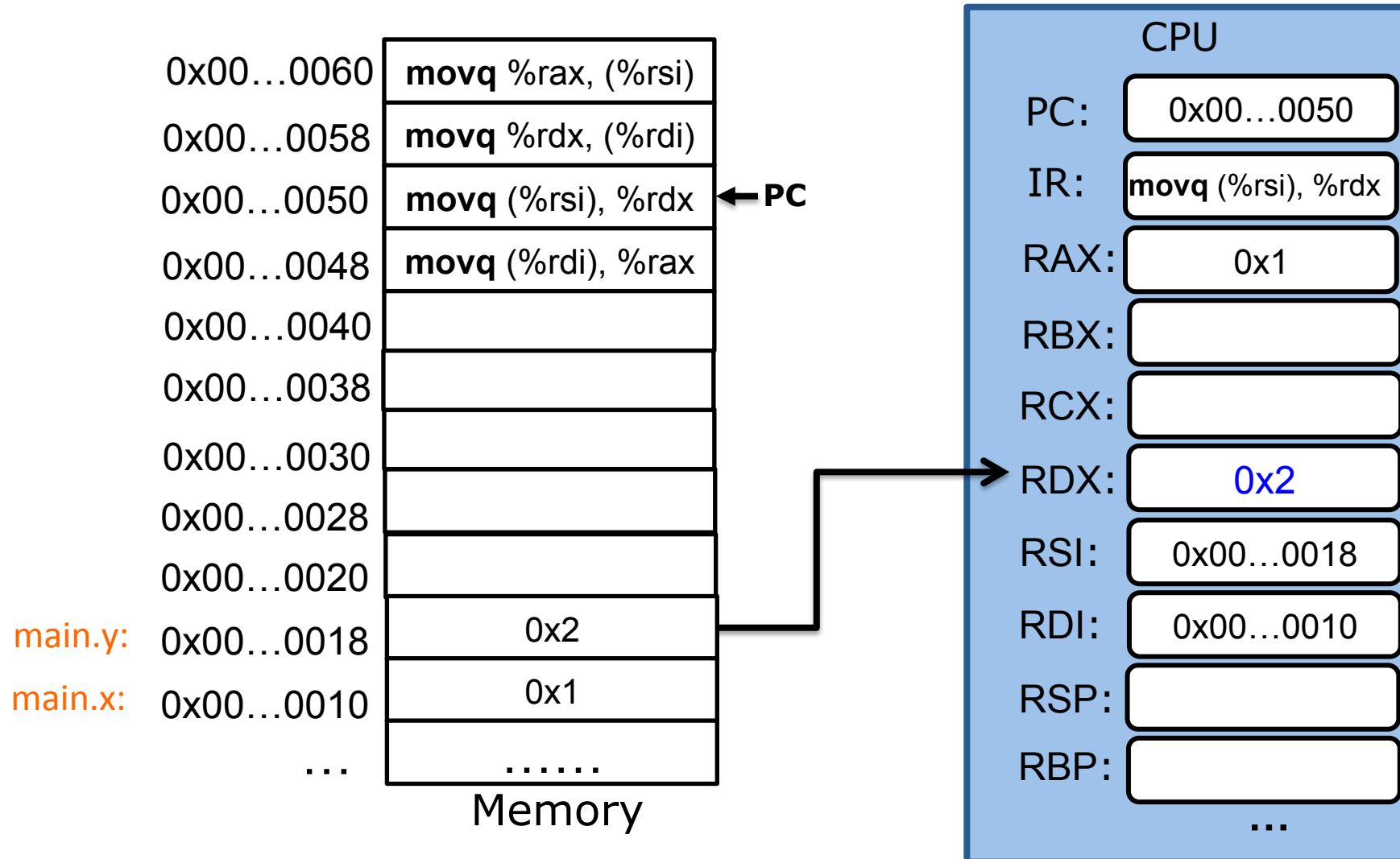
swap func



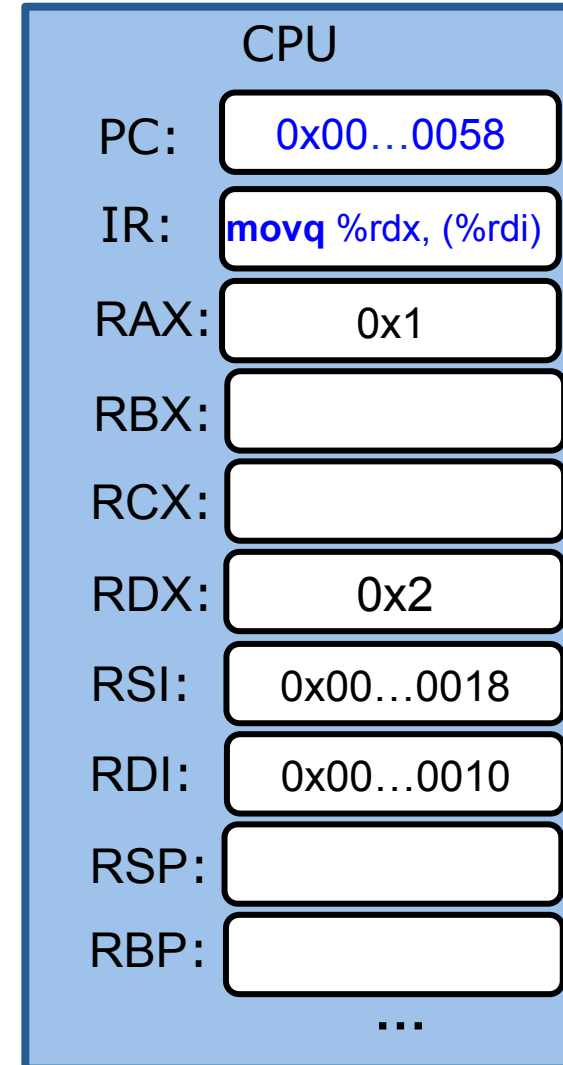
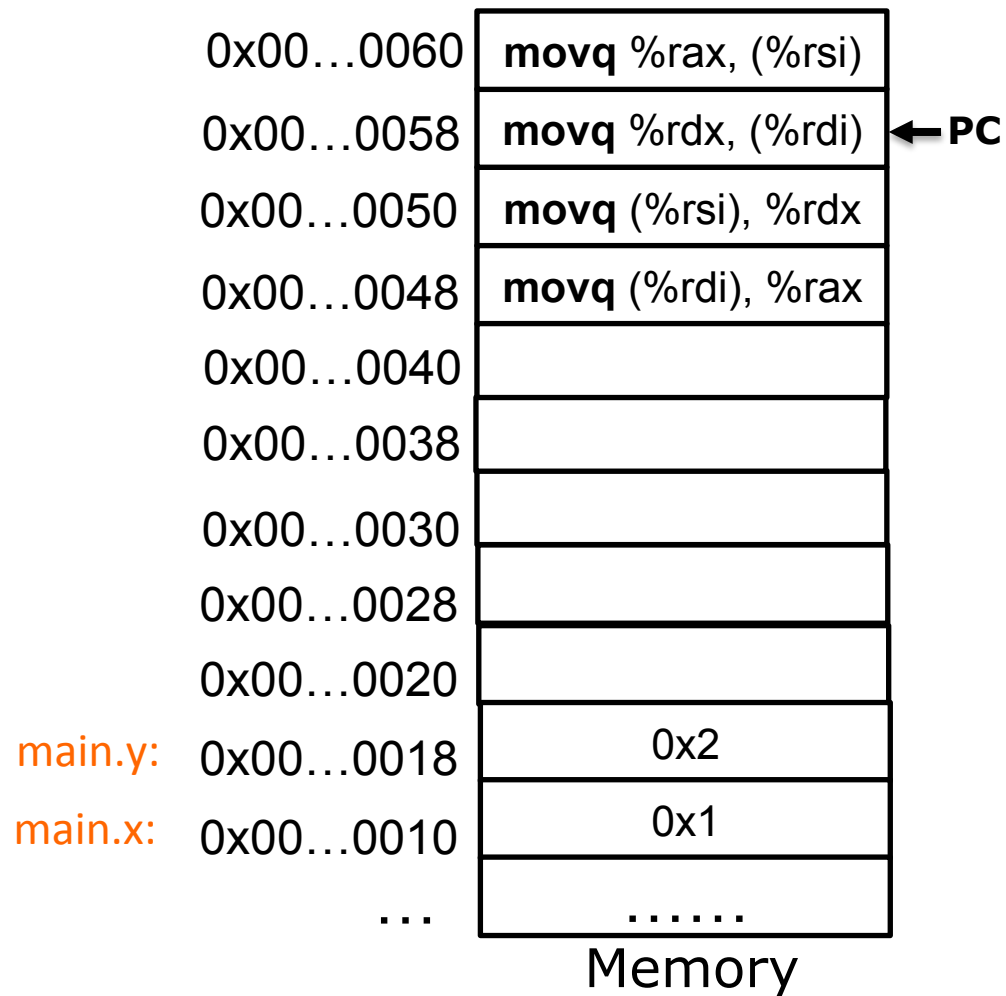
swap func



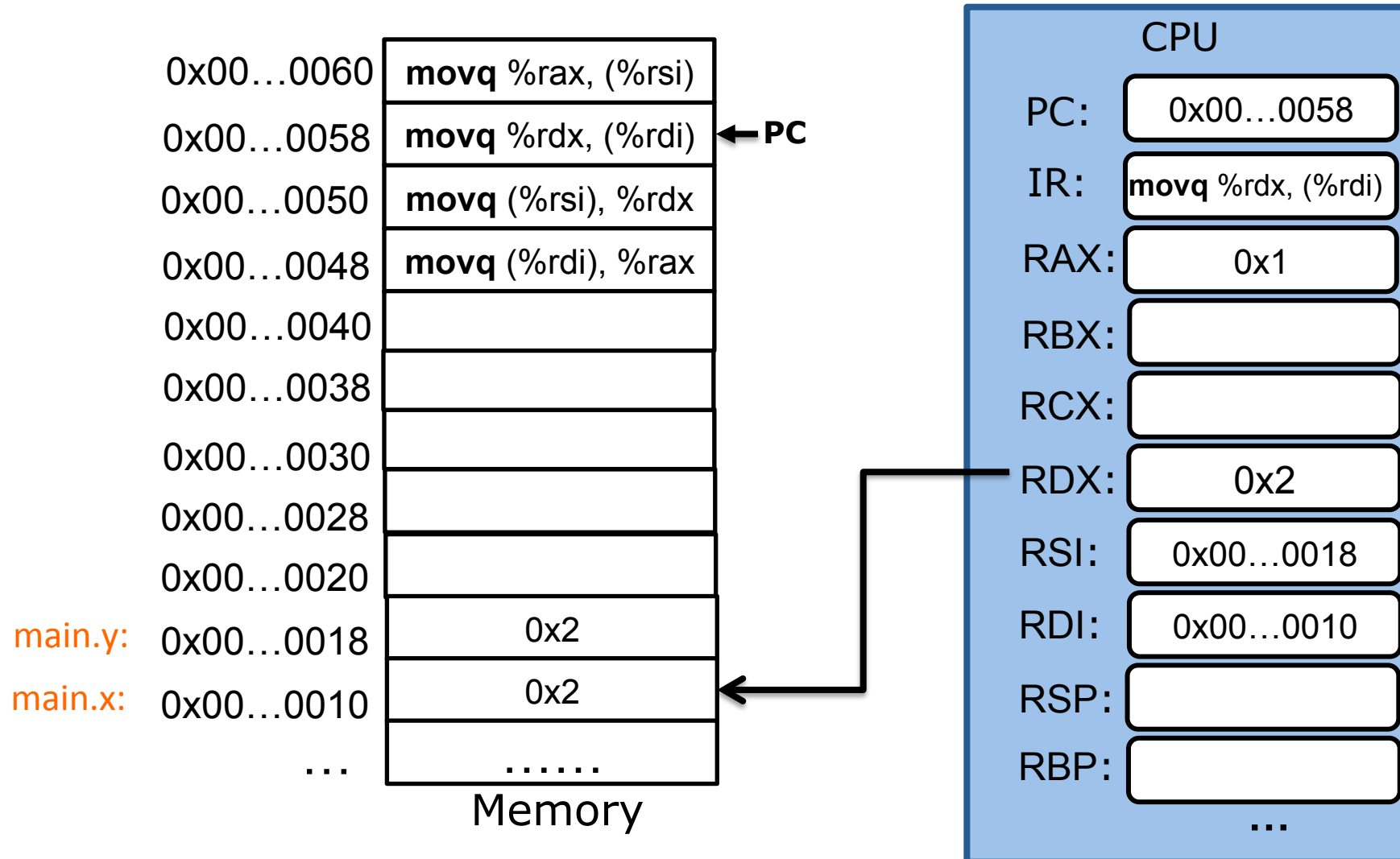
swap func



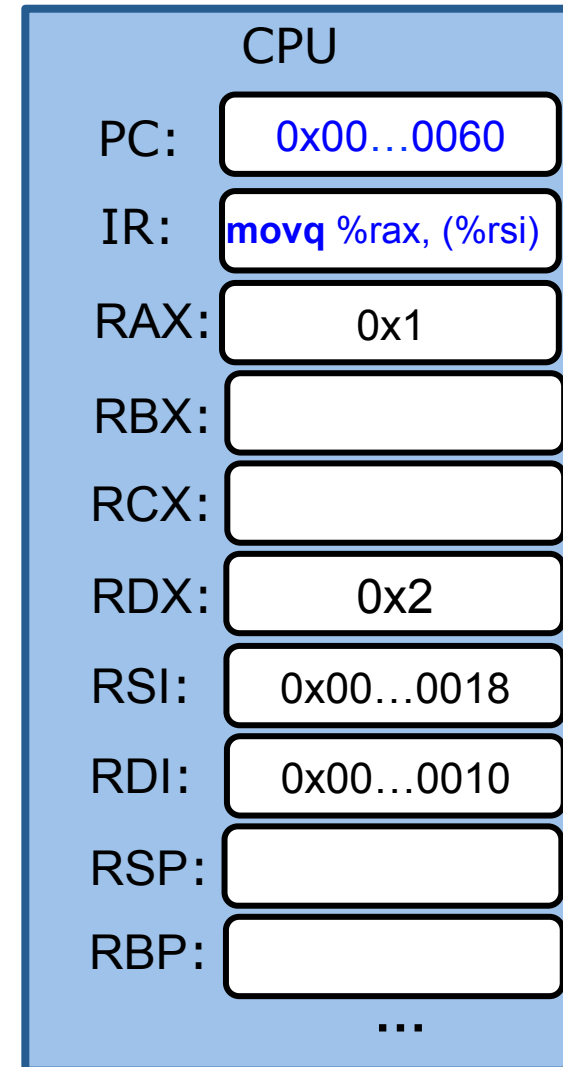
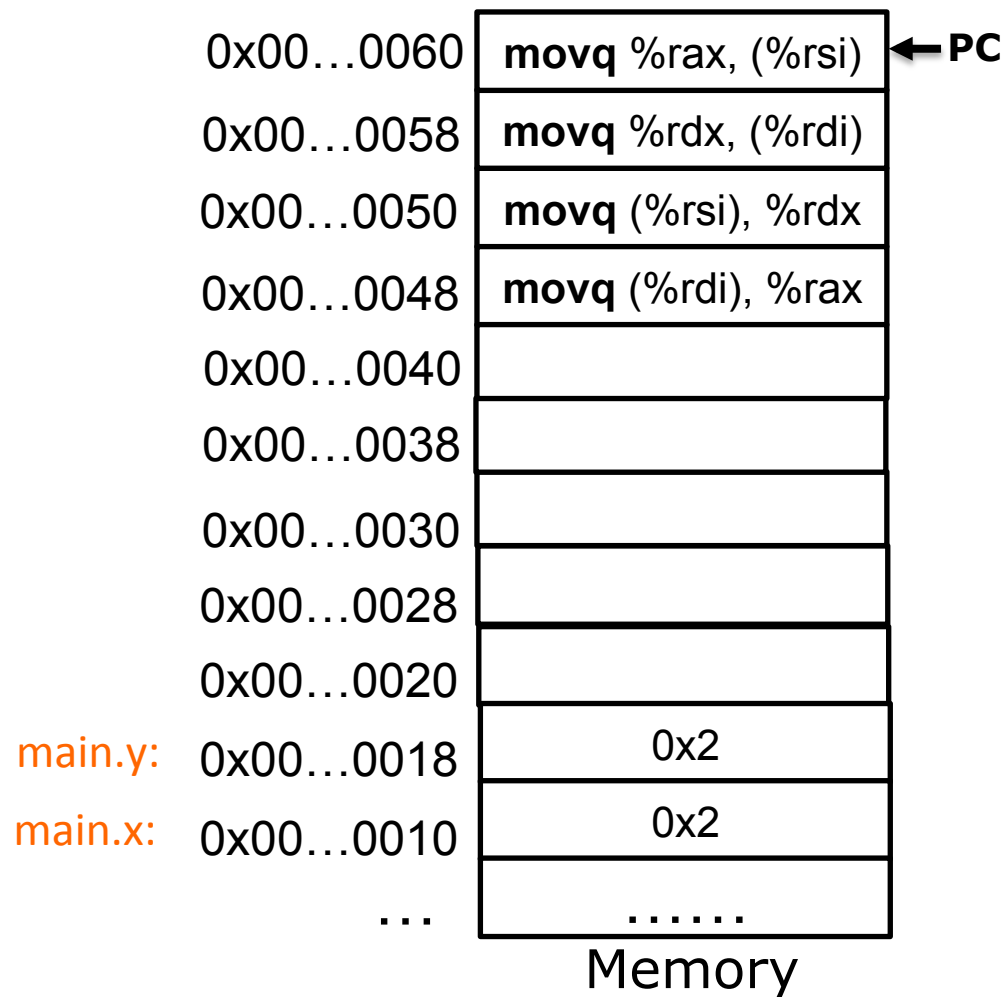
swap func



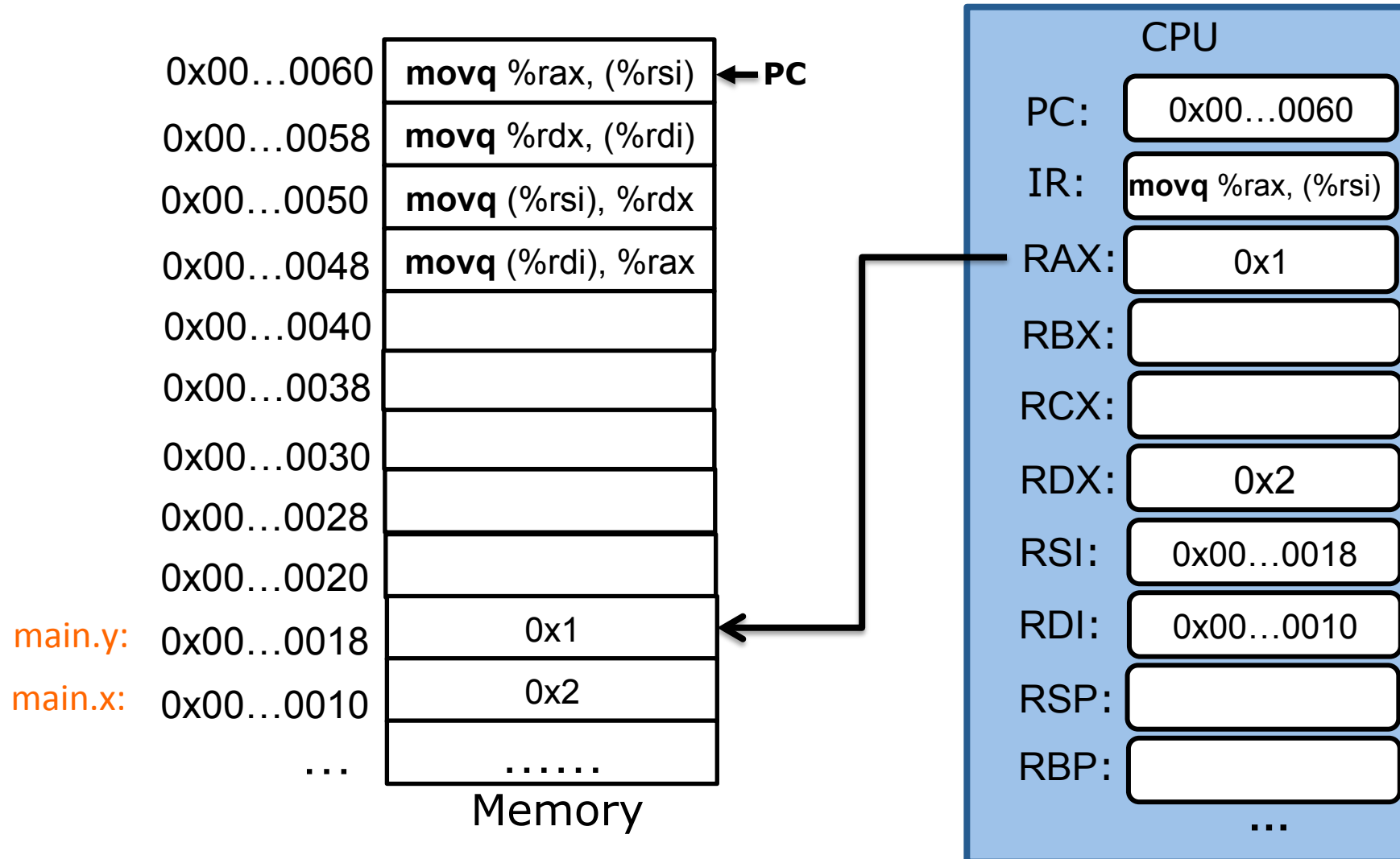
swap func



swap func



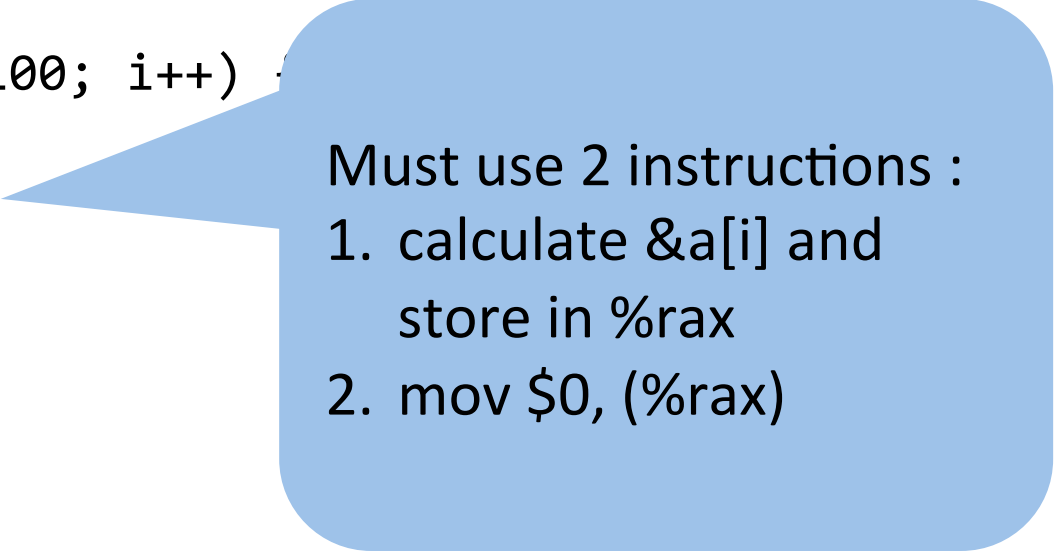
swap func



Limitation of direct addressing

- Direct addressing requires the register to contain the final address.
 - Not efficient for common operations like array indexing

```
for (int i = 0; i < 100; i++) {  
    a[i] = 0;  
}
```



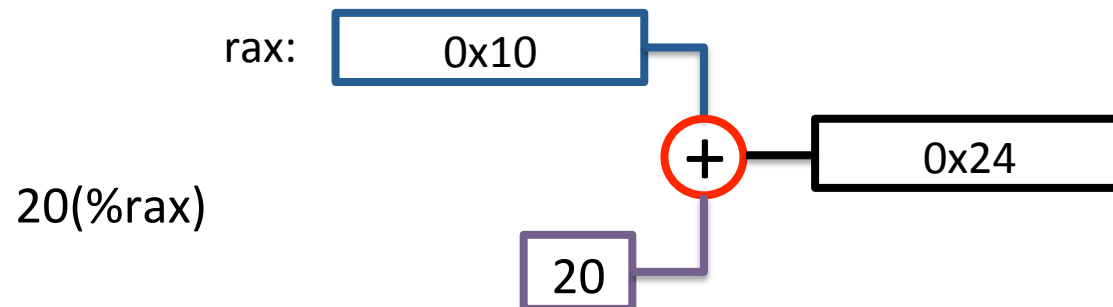
Must use 2 instructions :
1. calculate &a[i] and store in %rax
2. mov \$0, (%rax)

Solution: Addressing mode with displacement

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

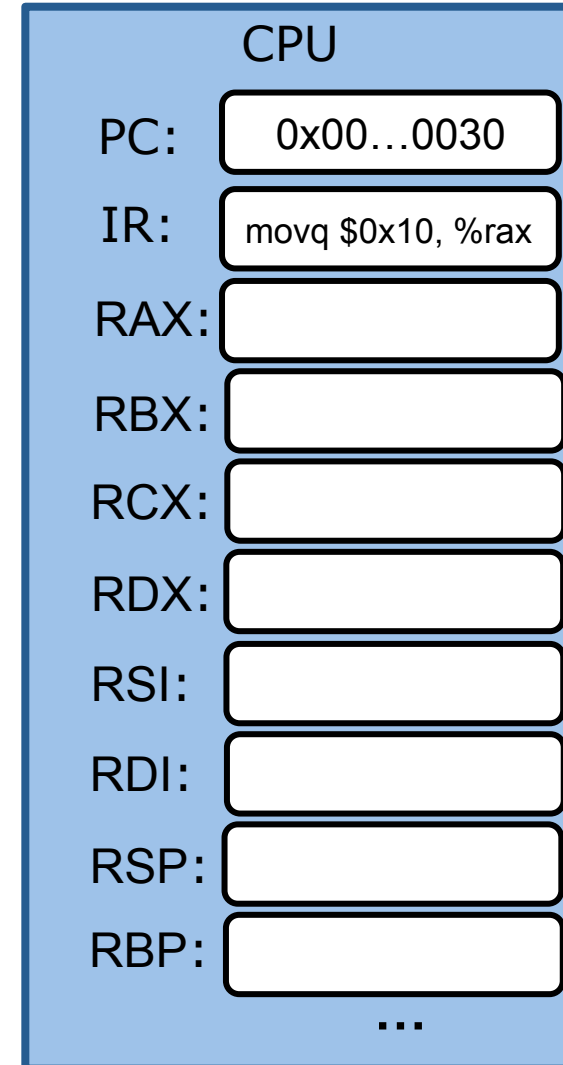
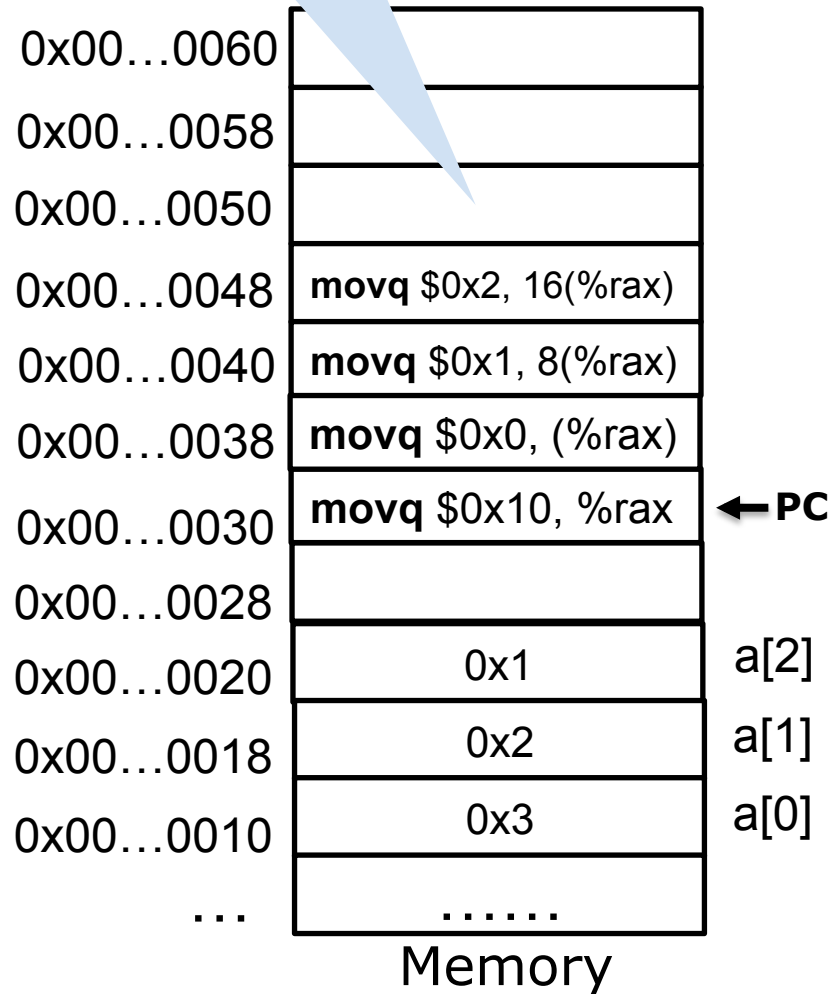
- Register specifies the start of the memory region
- Constant D specifies the offset

rax: 0x10

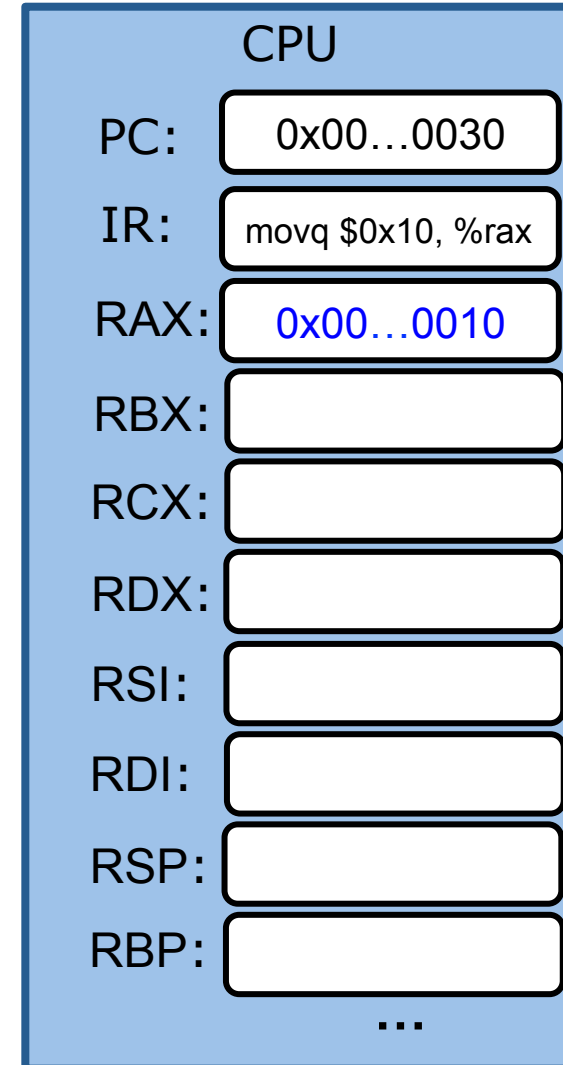
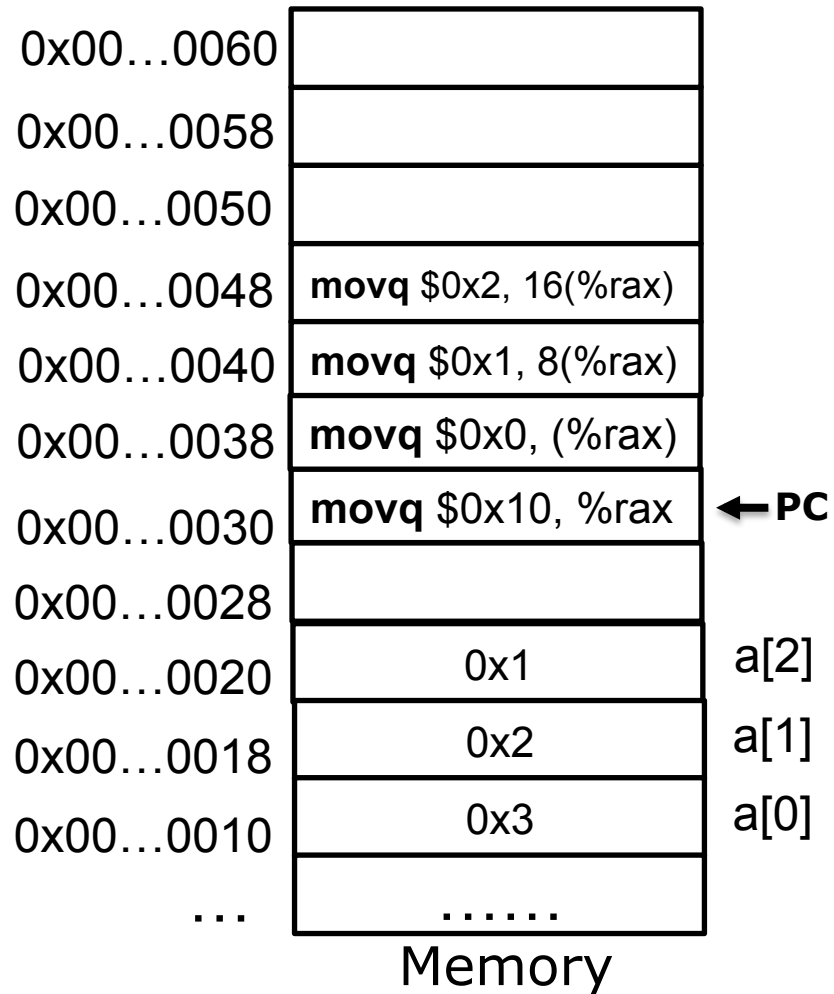


a[0] = 0;
a[1] = 1;
a[2] = 2;

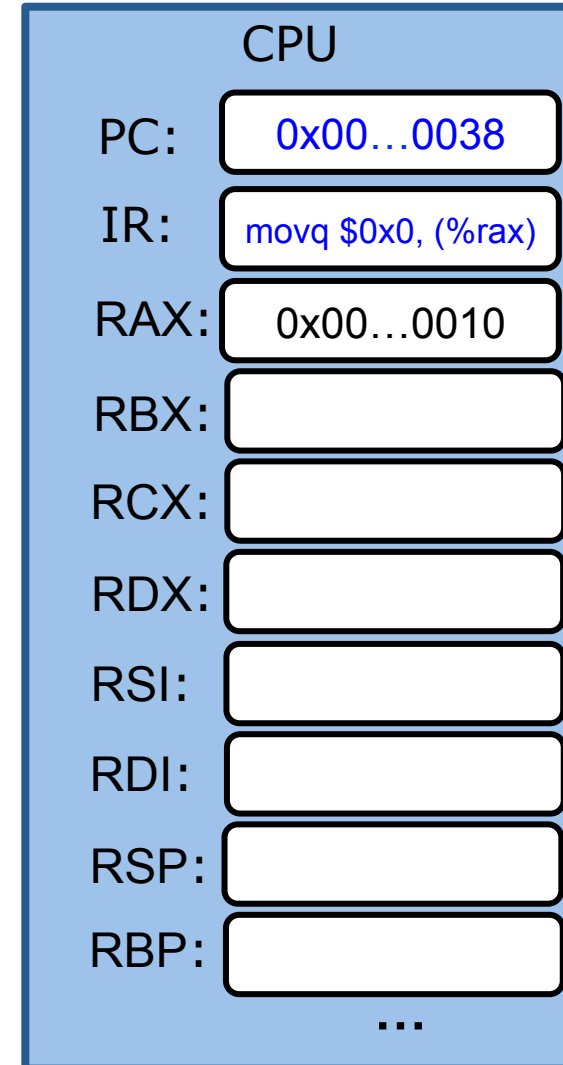
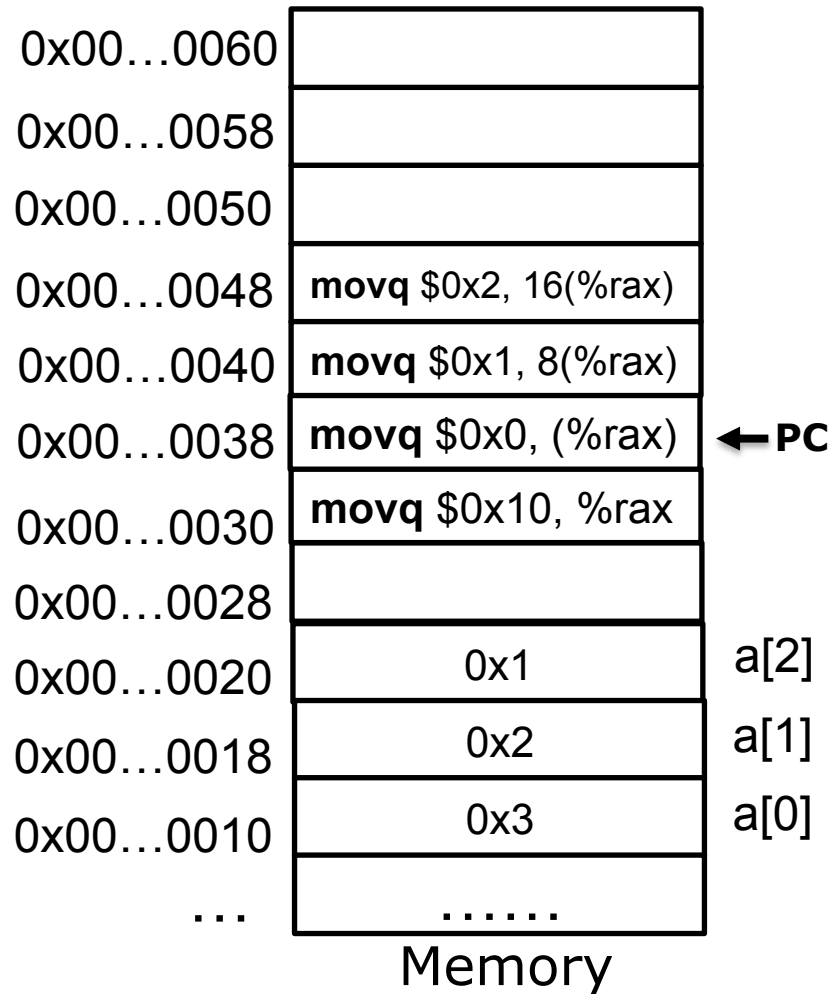
Example



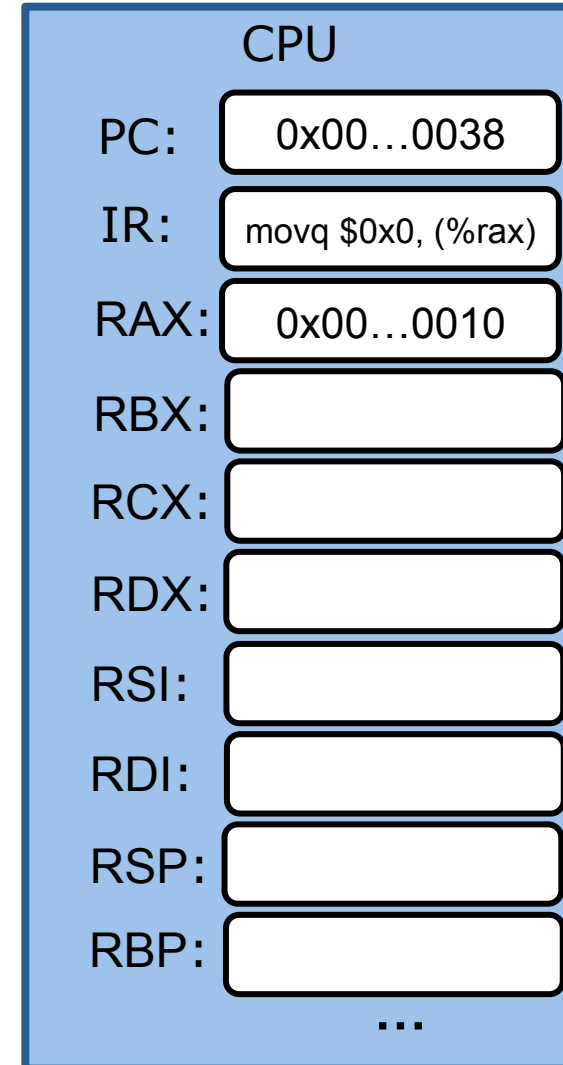
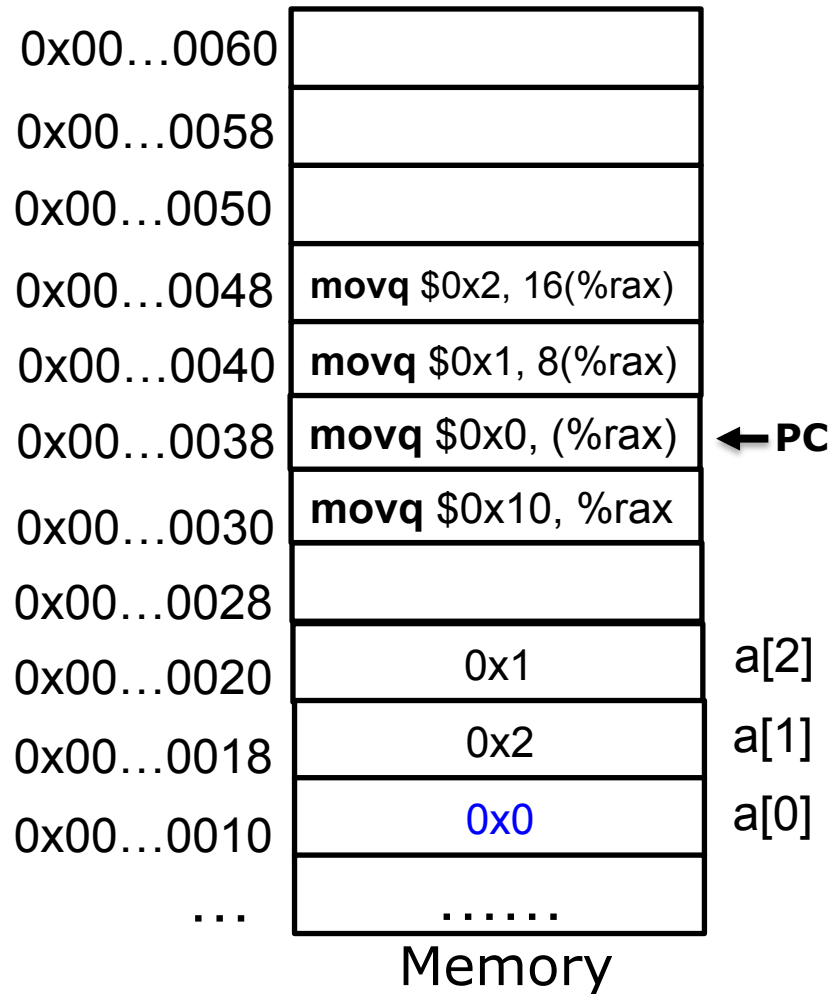
Example



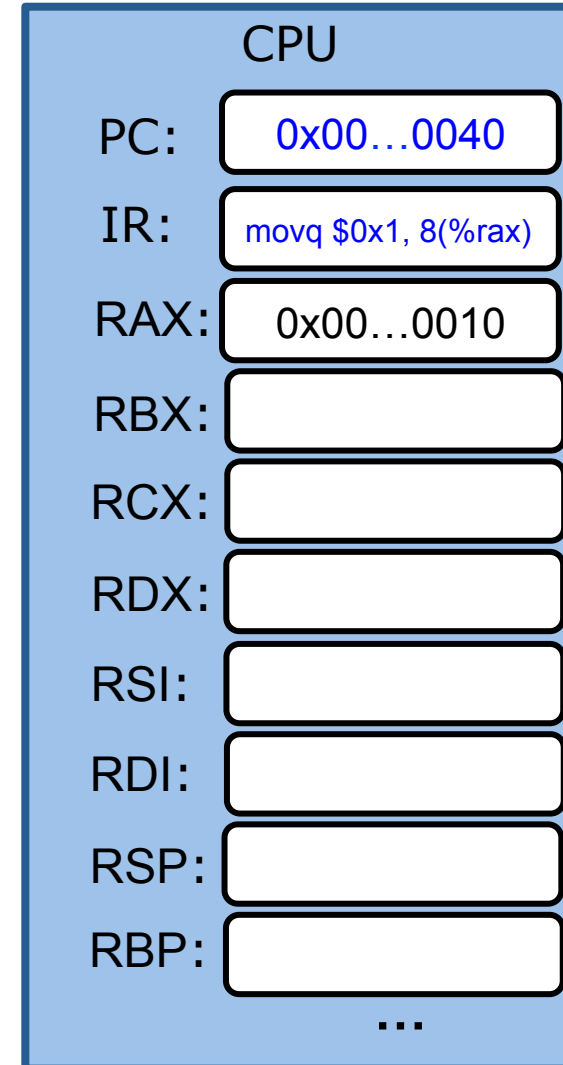
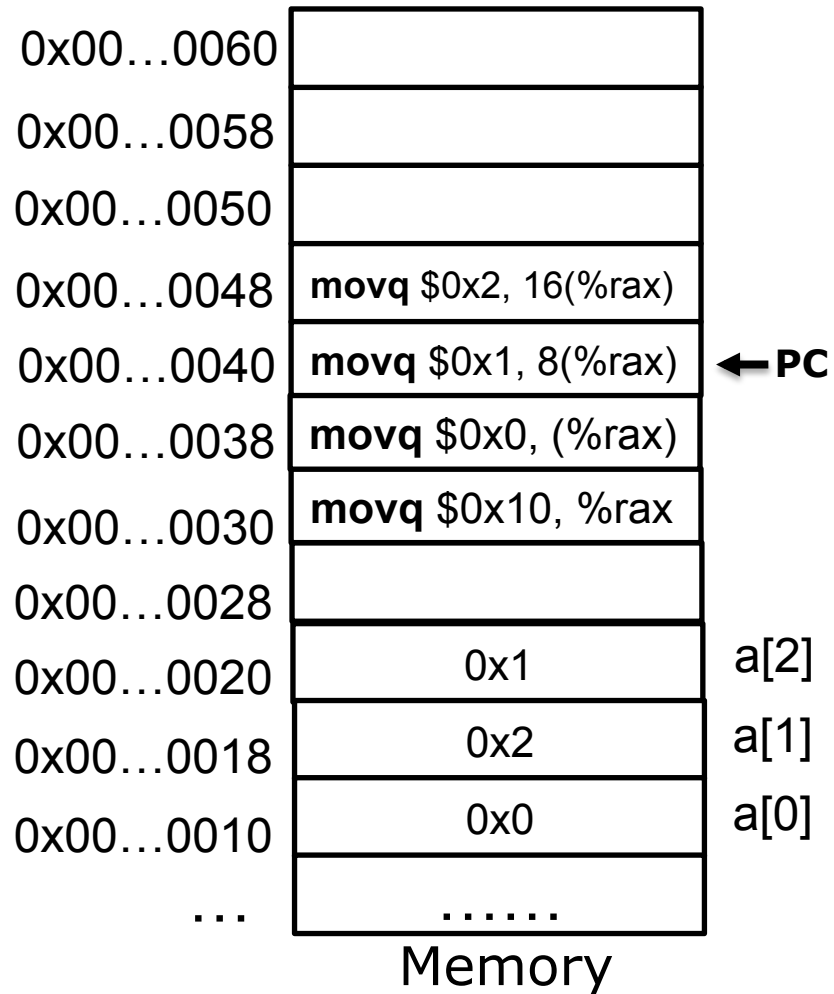
Example



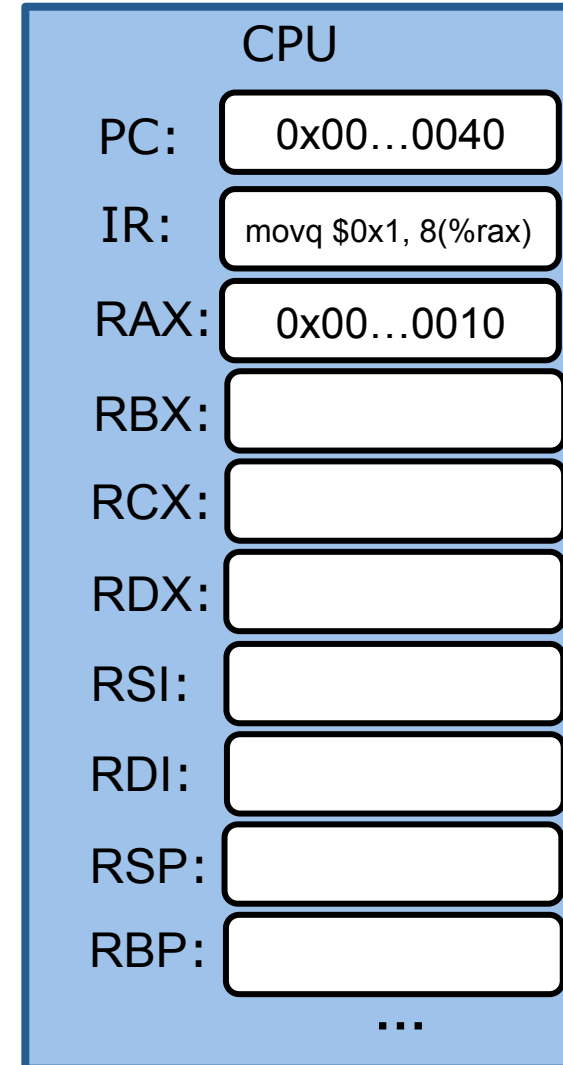
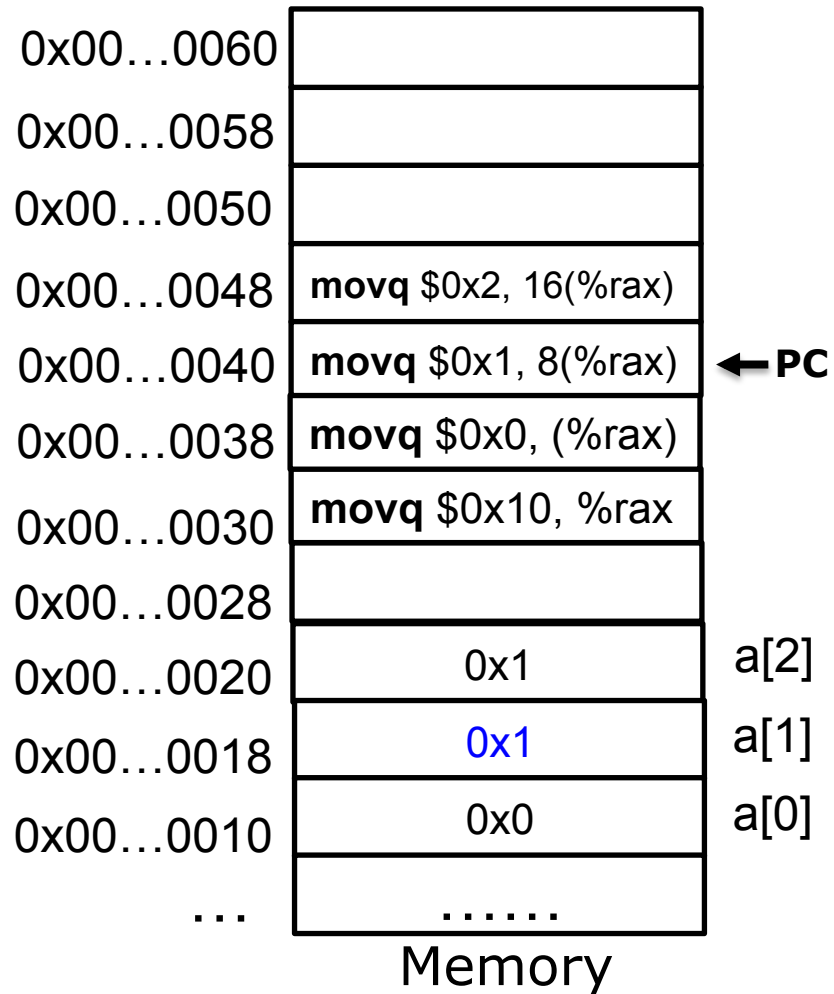
Example



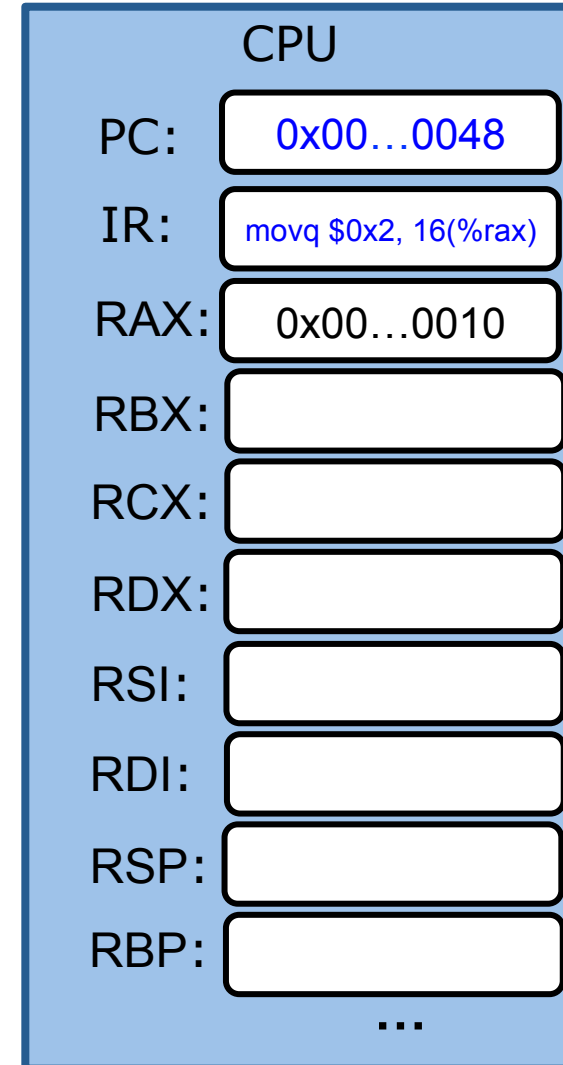
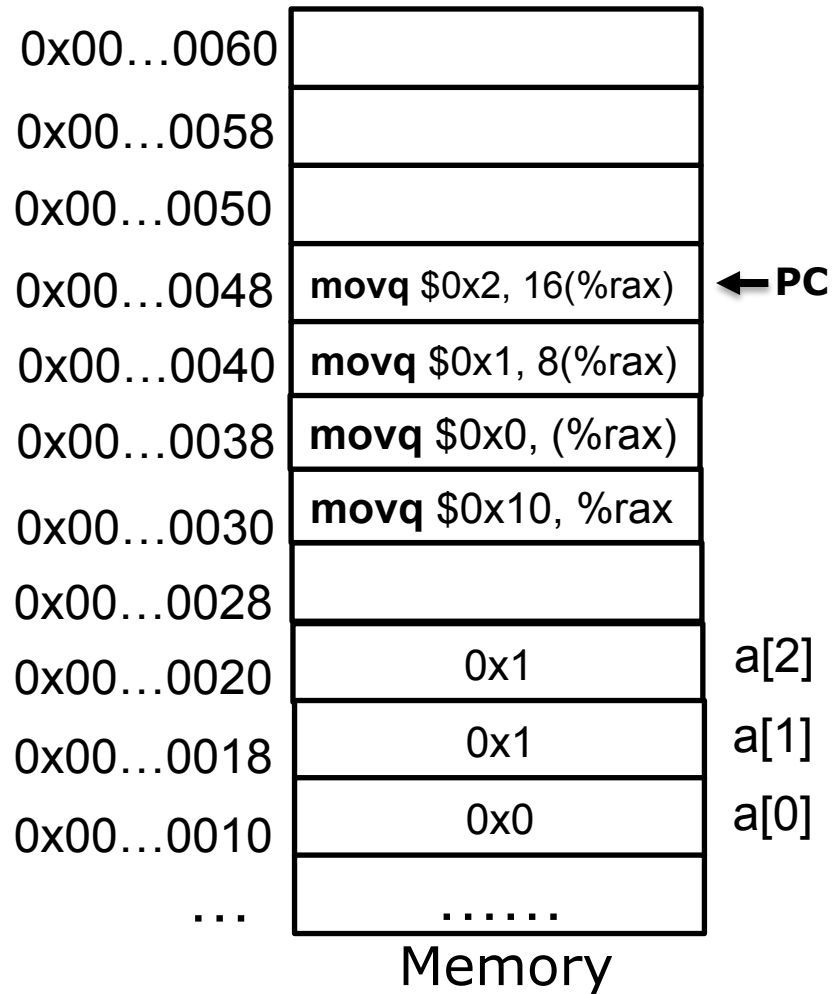
Example



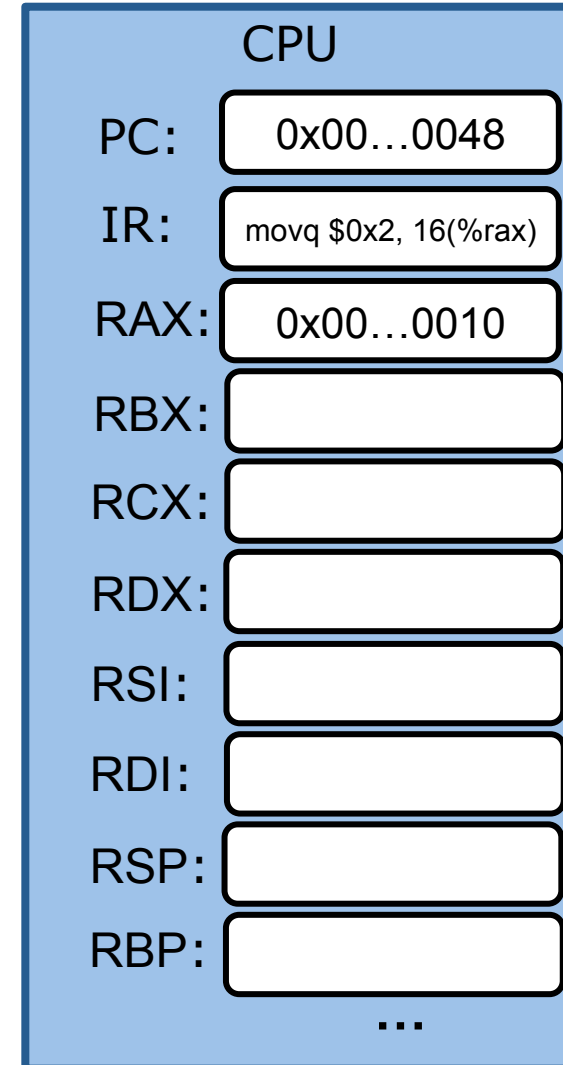
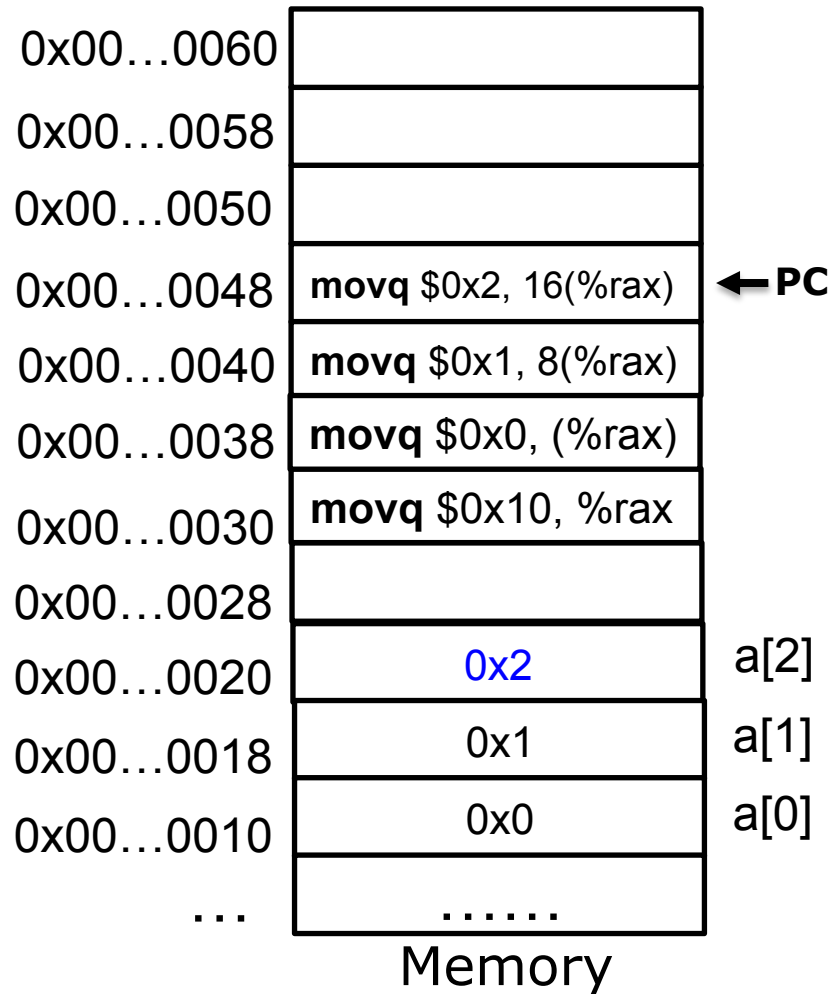
Example



Example



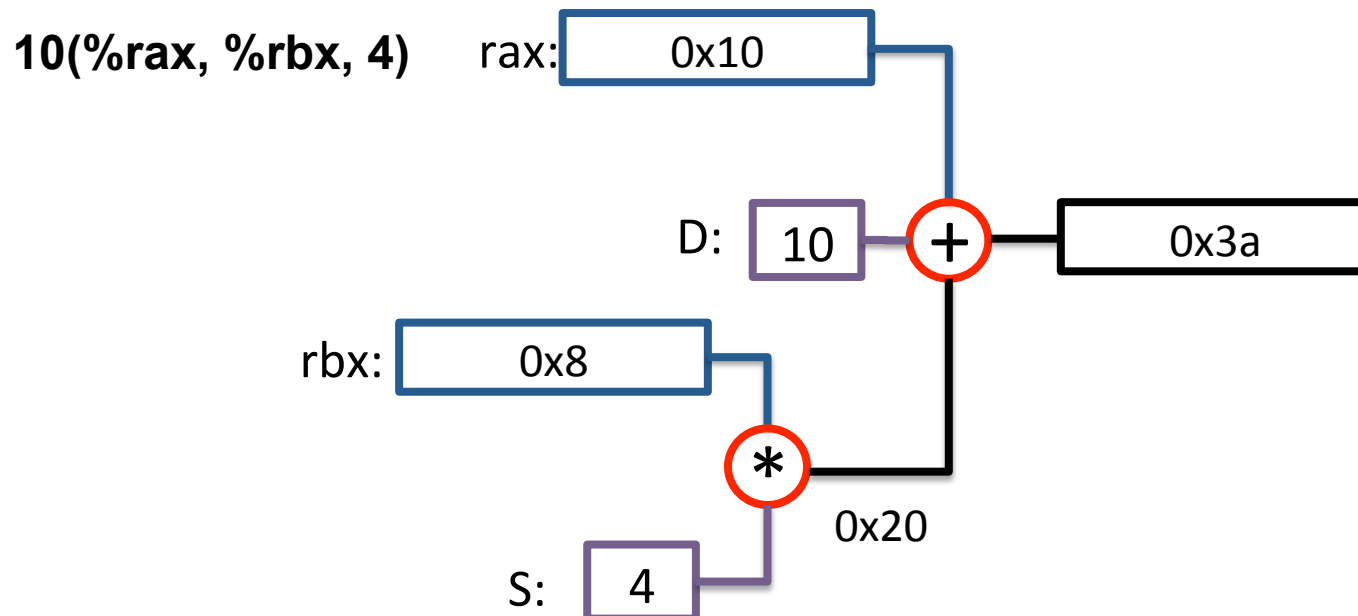
Example



Complete Memory Addressing Mode

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

- 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>

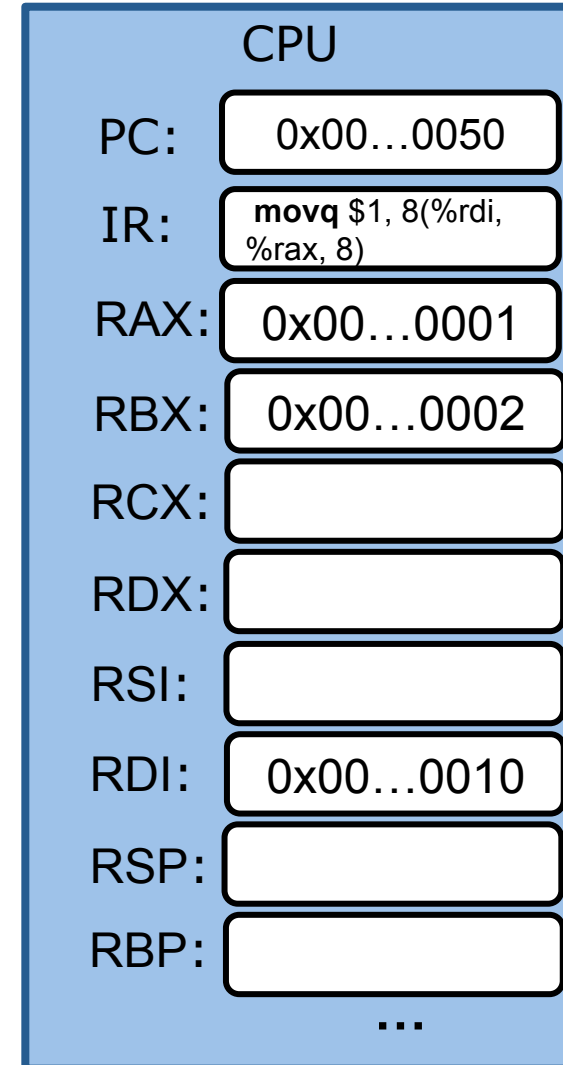
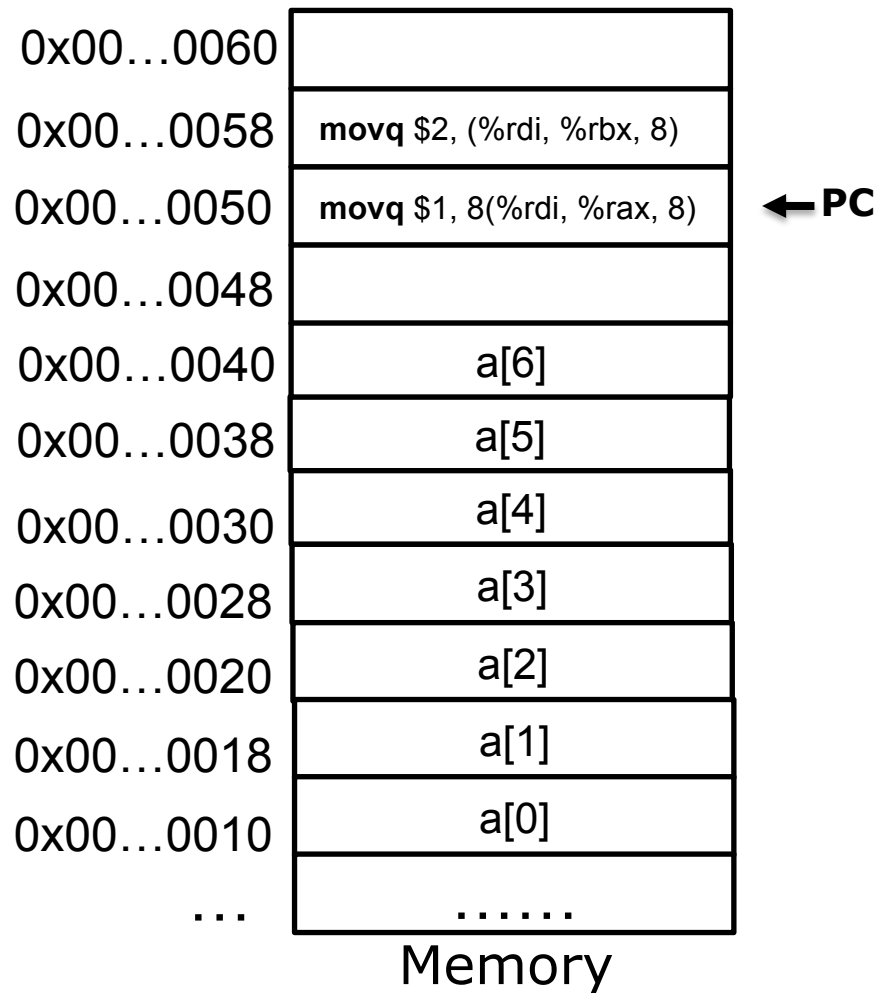
Expression	Address Computation	Address
<code>0x8(%rdx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%rdx,%rcx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%rdx,%rcx,4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80(,%rdx,2)</code>		

Address Computation Examples

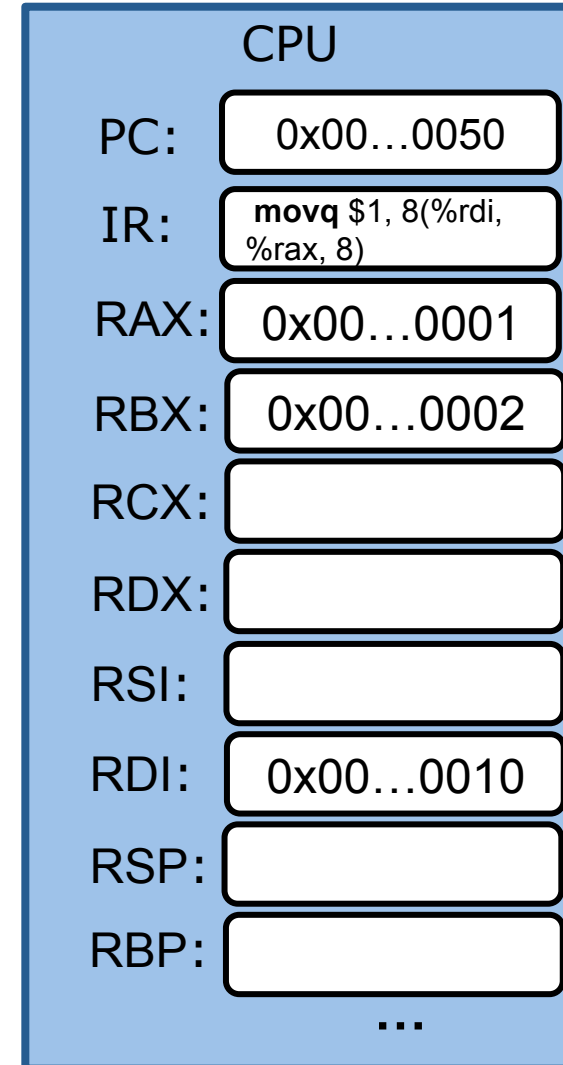
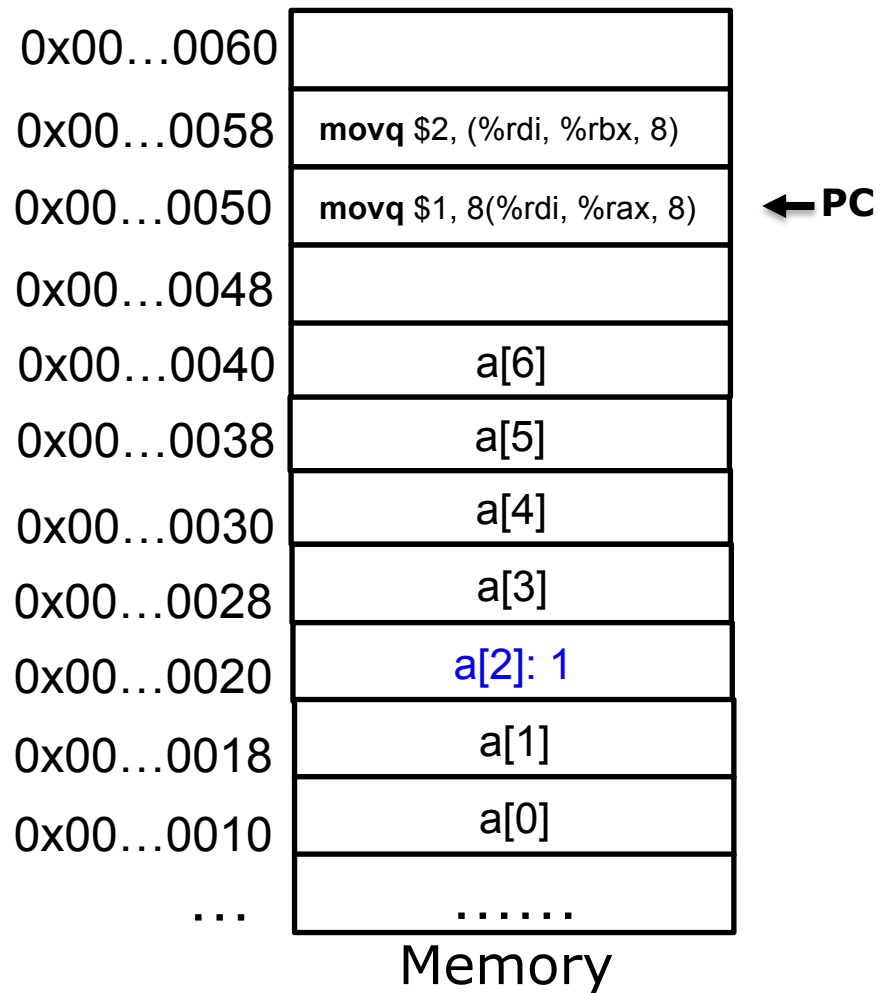
<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x100</code>

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>

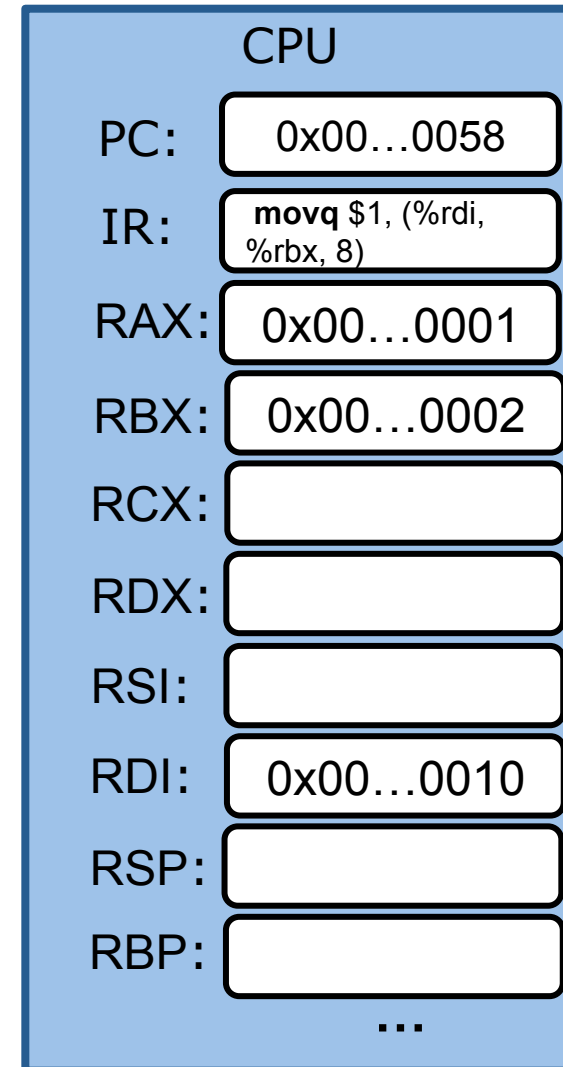
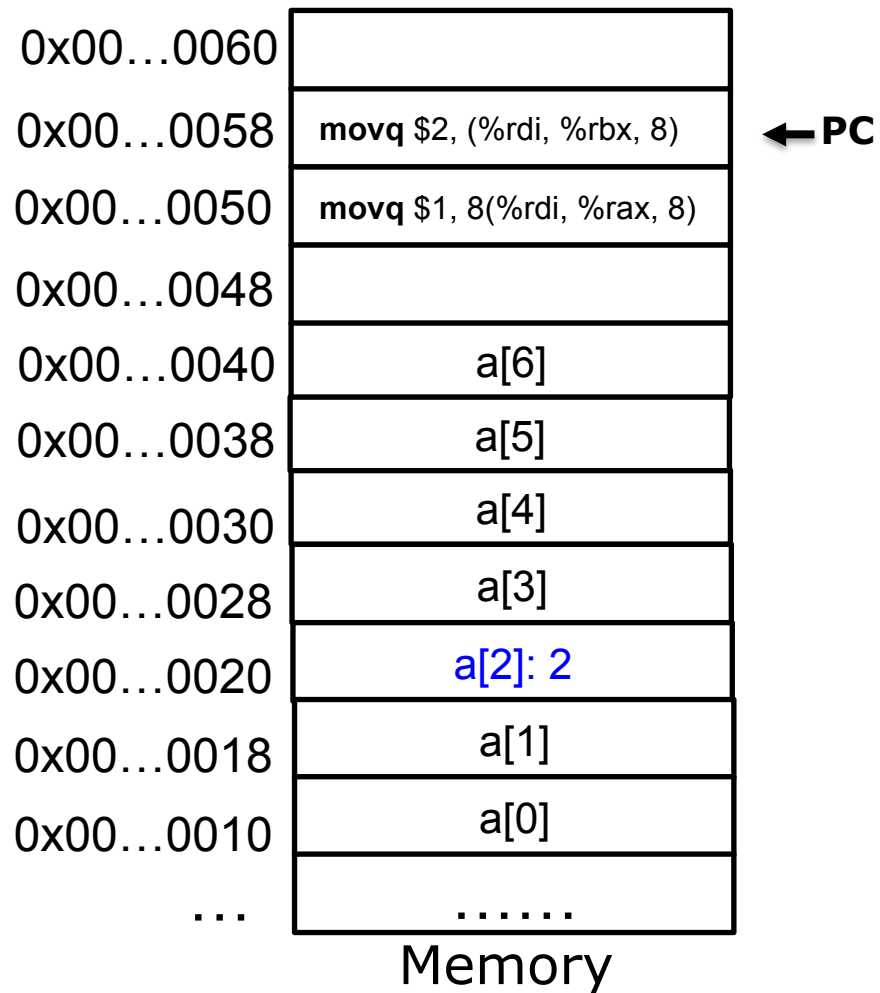
Example



Example



Example



mov{bwlq}

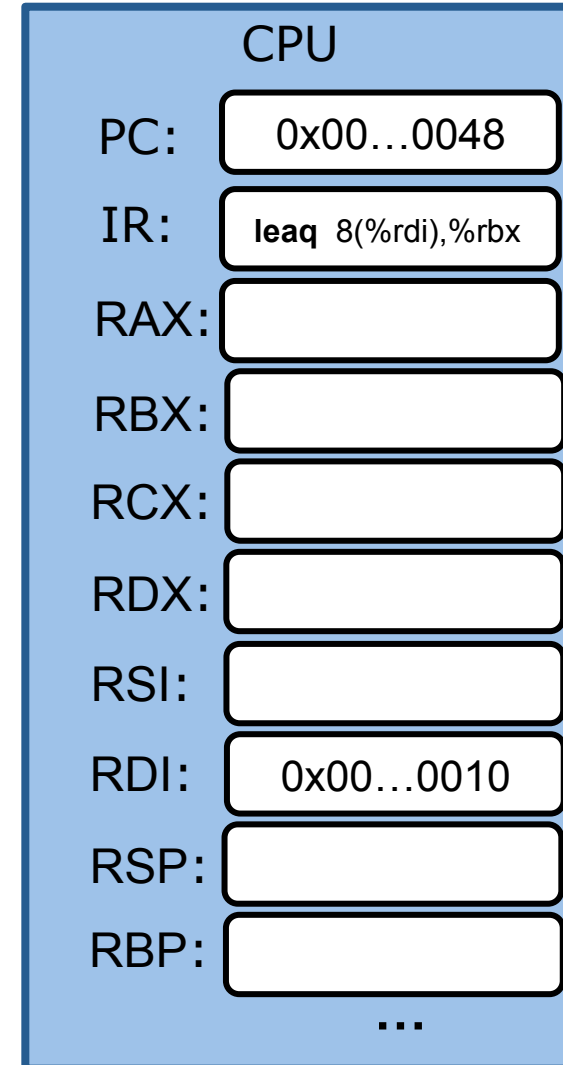
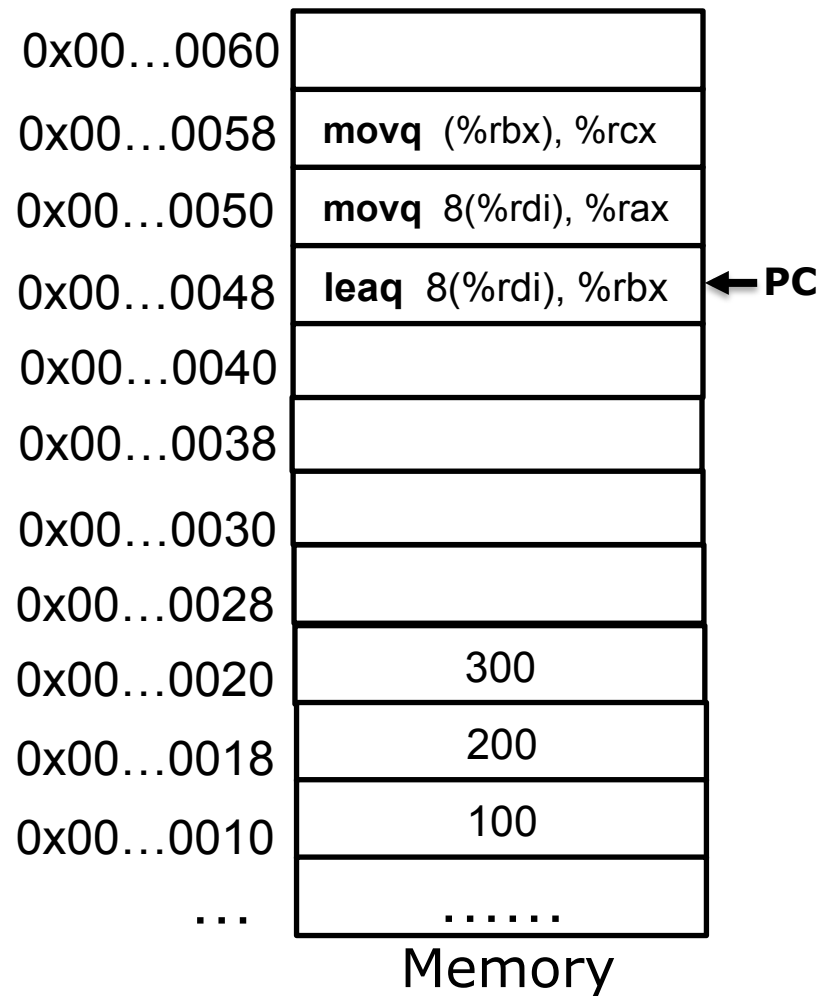
movb src, dest	Copy a byte from the source operand to the destination operand. e.g., movb %al, %bl
movw src, dest	Copy a word from the source operand to the destination operand. e.g., movw %ax, %bx
movl src, dest	Copy a long (32 bits) from the source operand to the destination operand. e.g., movl %eax,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

The lea instruction

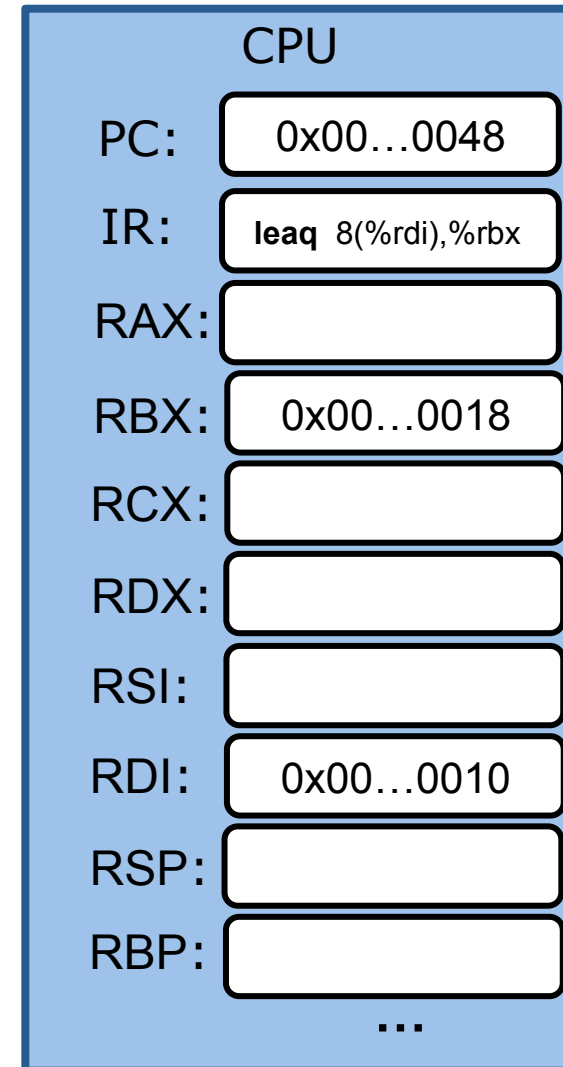
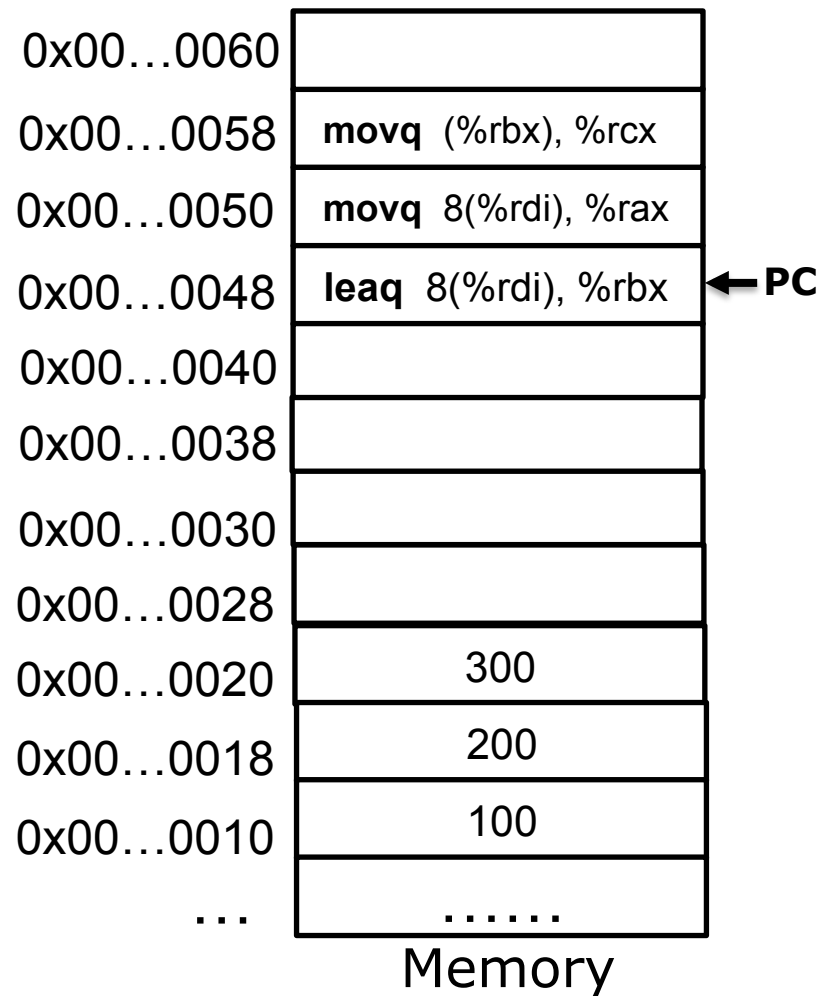
leaq *Source*, *Dest*

- load effective address: set *Dest* to the address given by *Source* address mode expression
- No memory access

Example



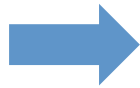
Example



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

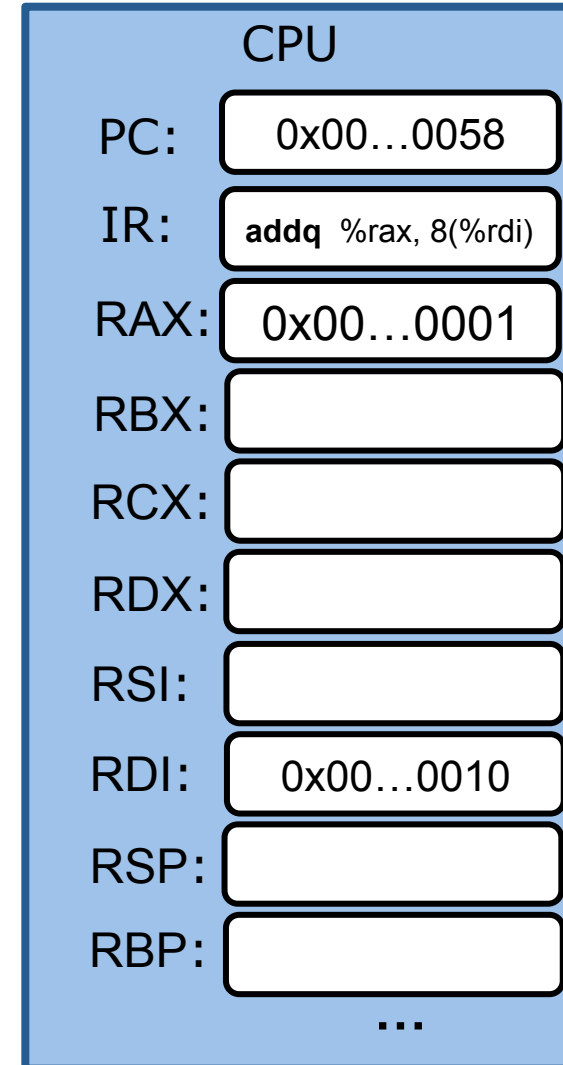
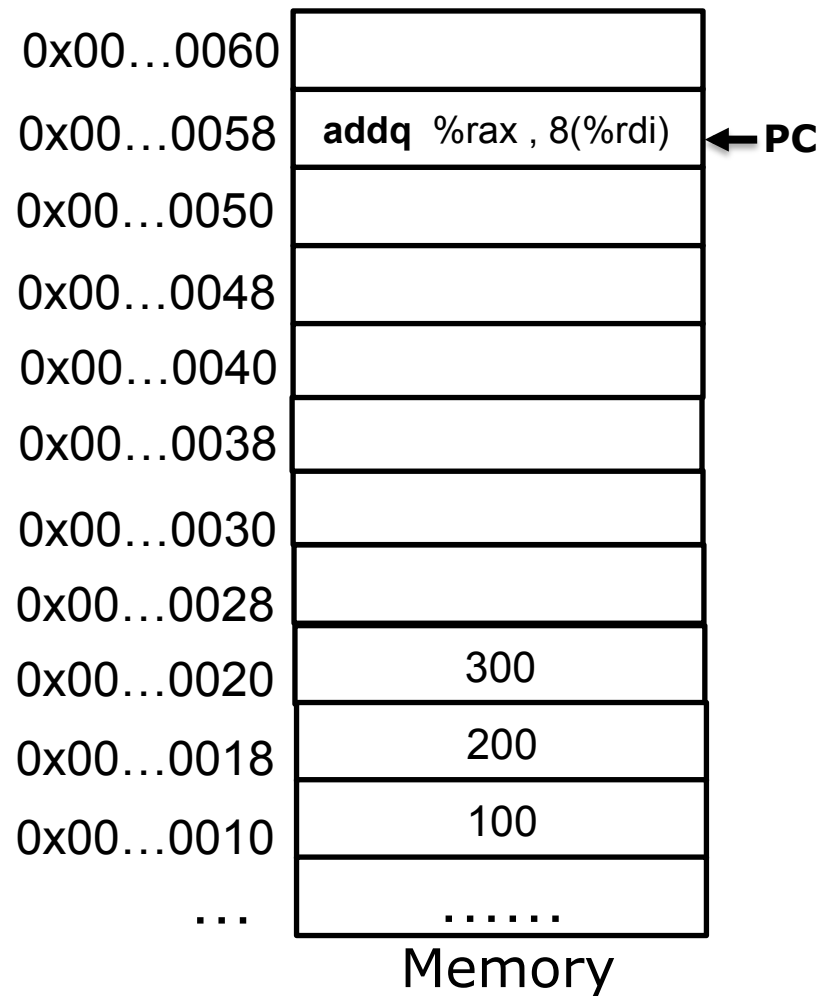
negq Dest $\text{Dest} = - \text{Dest}$

Bitwise Operations

salq	Src, Dest	Dest = Dest << Src	Arithmetic left shift
sarq	Src, Dest	Dest = Dest >> Src	Arithmetic right shift
shlq	Src, Dest	Dest = Dest << Src	Logical left shift
shrq	Src, Dest	Dest = Dest >> Src	Logical right shift
xorq	Src, Dest	Dest = Dest ^ Src	
andq	Src, Dest	Dest = Dest & Src	
orq	Src, Dest	Dest = Dest Src	
notq	Dest	Dest = ~Dest	



Example



Example

