# Machine Program: Basics

Jinyang Li

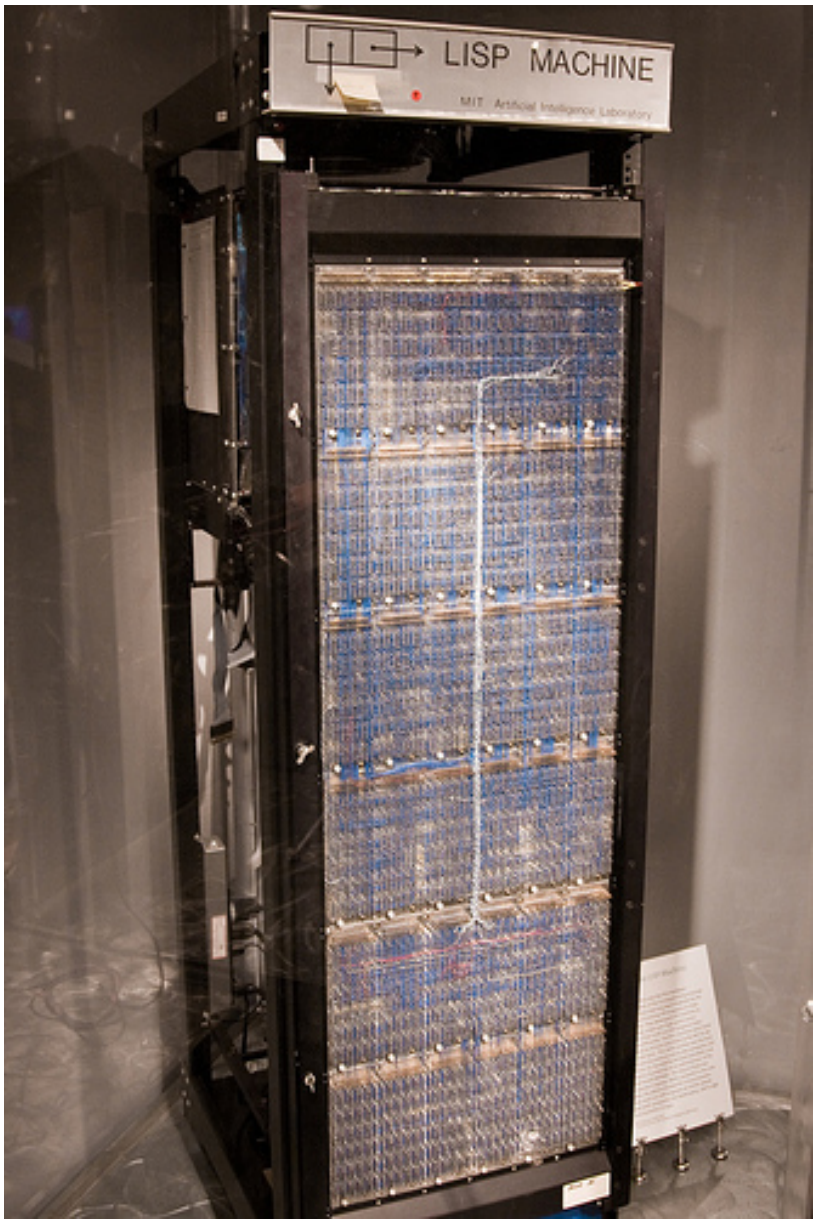
Some are based on Tiger Wang's slides

# What we've learnt so far

- Programming in C

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

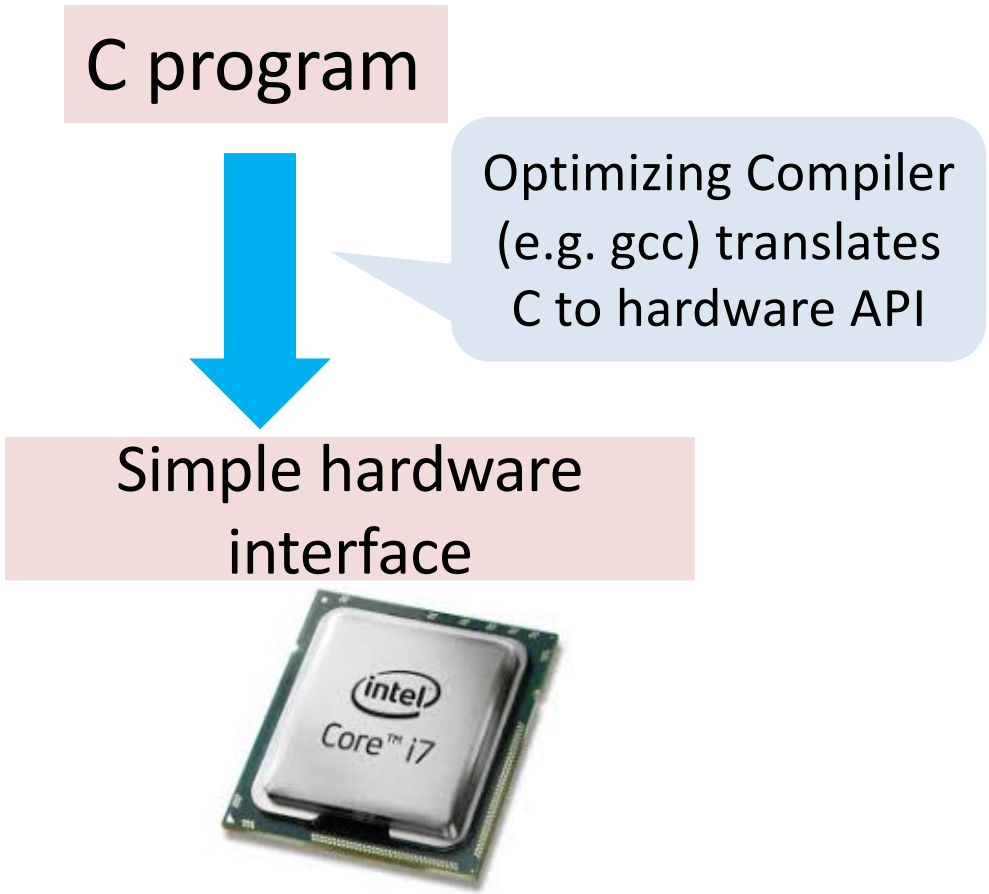
# a CPU to execute C directly?




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

# Why not build a CPU that directly executes C?

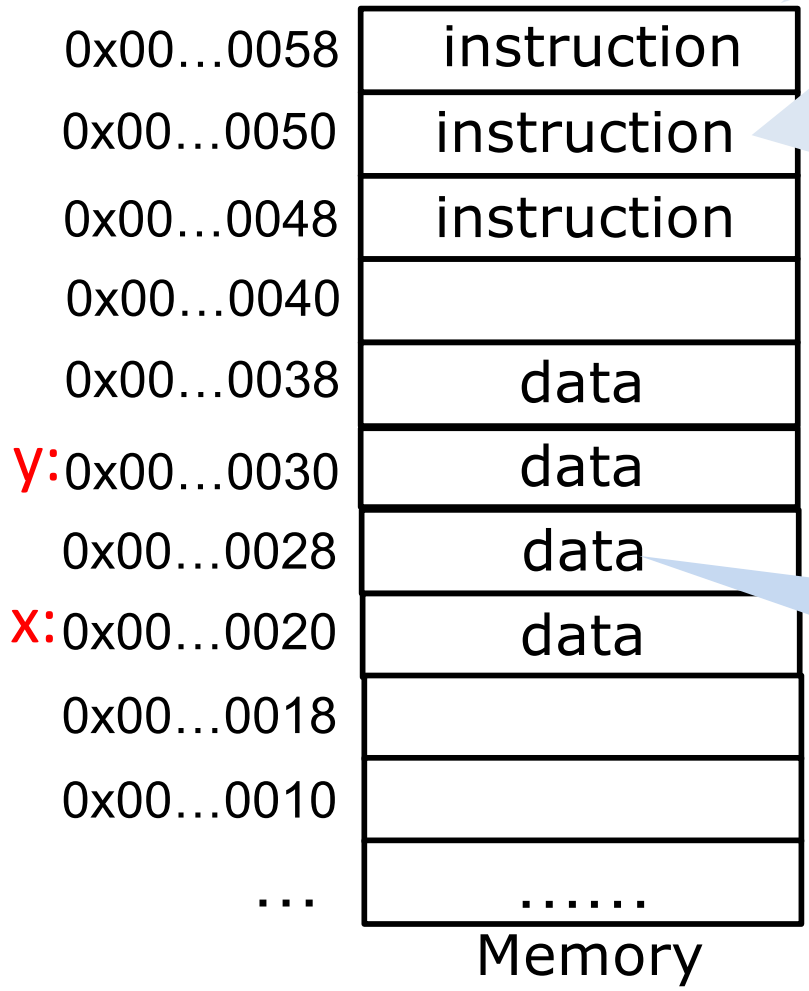
- Leads to very complex hardware design
  - Complex → Hard to implement w/ high performance
- A better approach:



# C vs. machine code

```
long x;
long y;

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

compile to
x86 executable

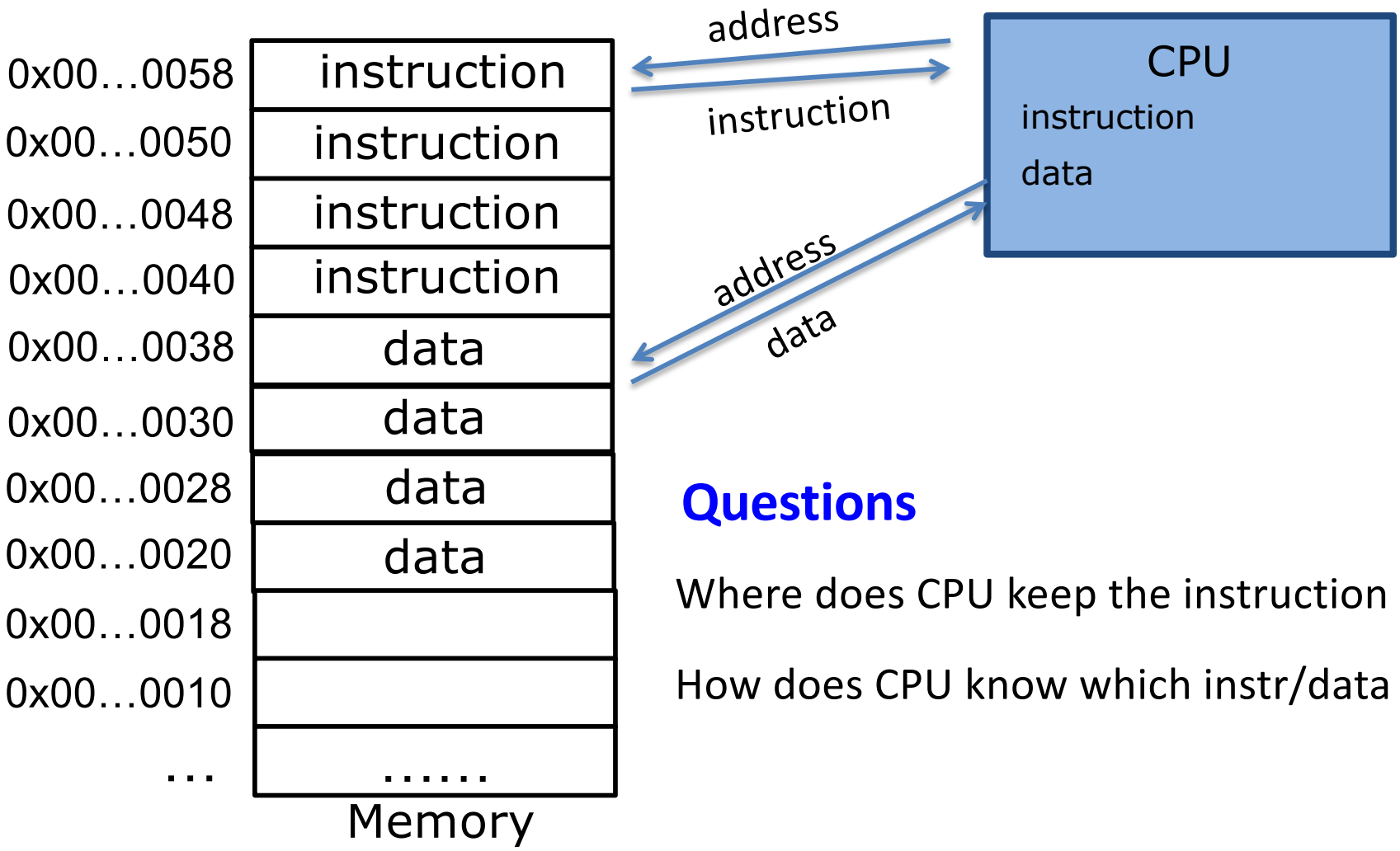



E.g. move data from one memory location to another

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

No concept of variables, scopes, types

# How CPU executes a program

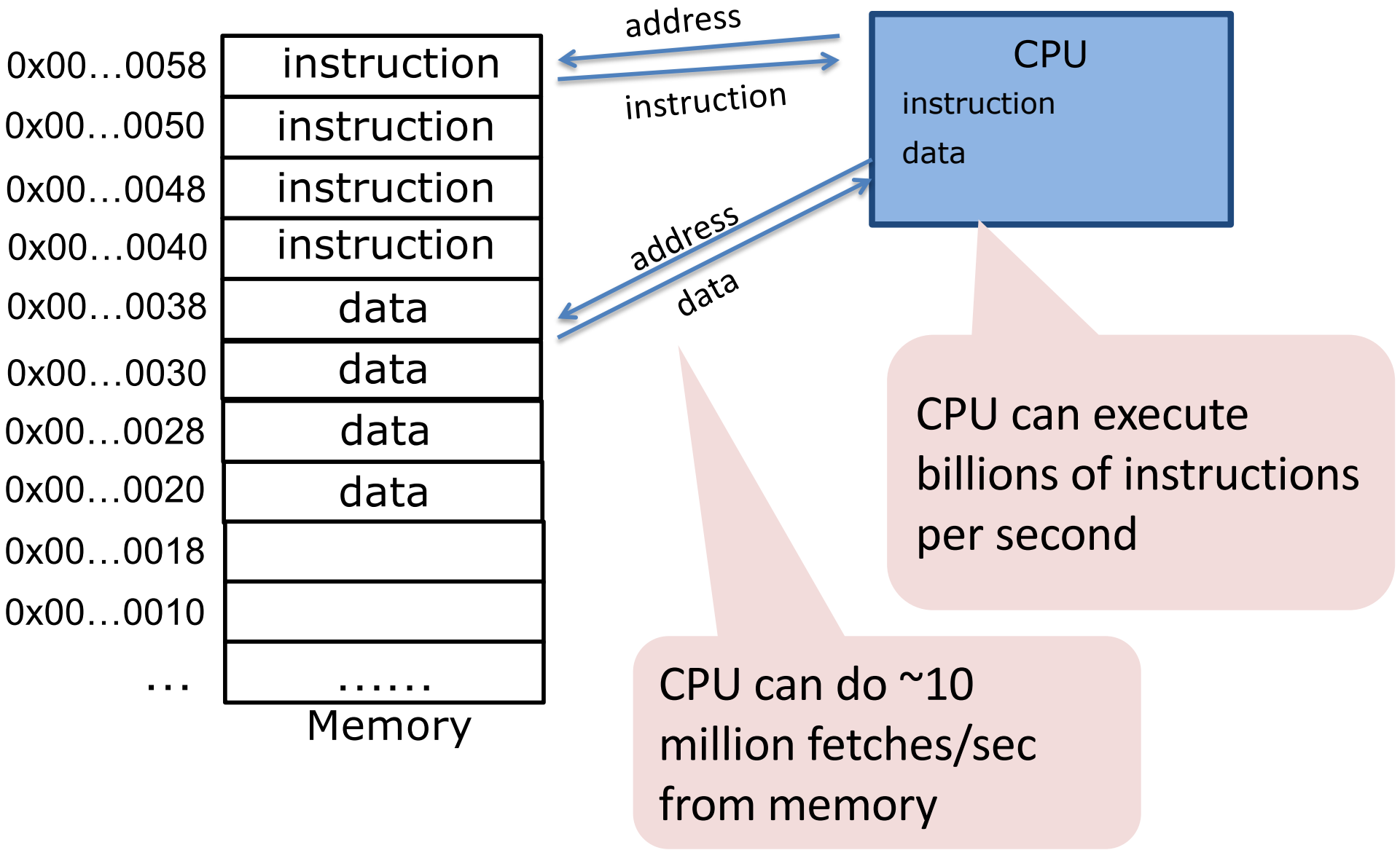



## Questions

Where does CPU keep the instruction and data?

How does CPU know which instr/data to fetch?

# How CPU executes a program



# 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 "RFLAGS" 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

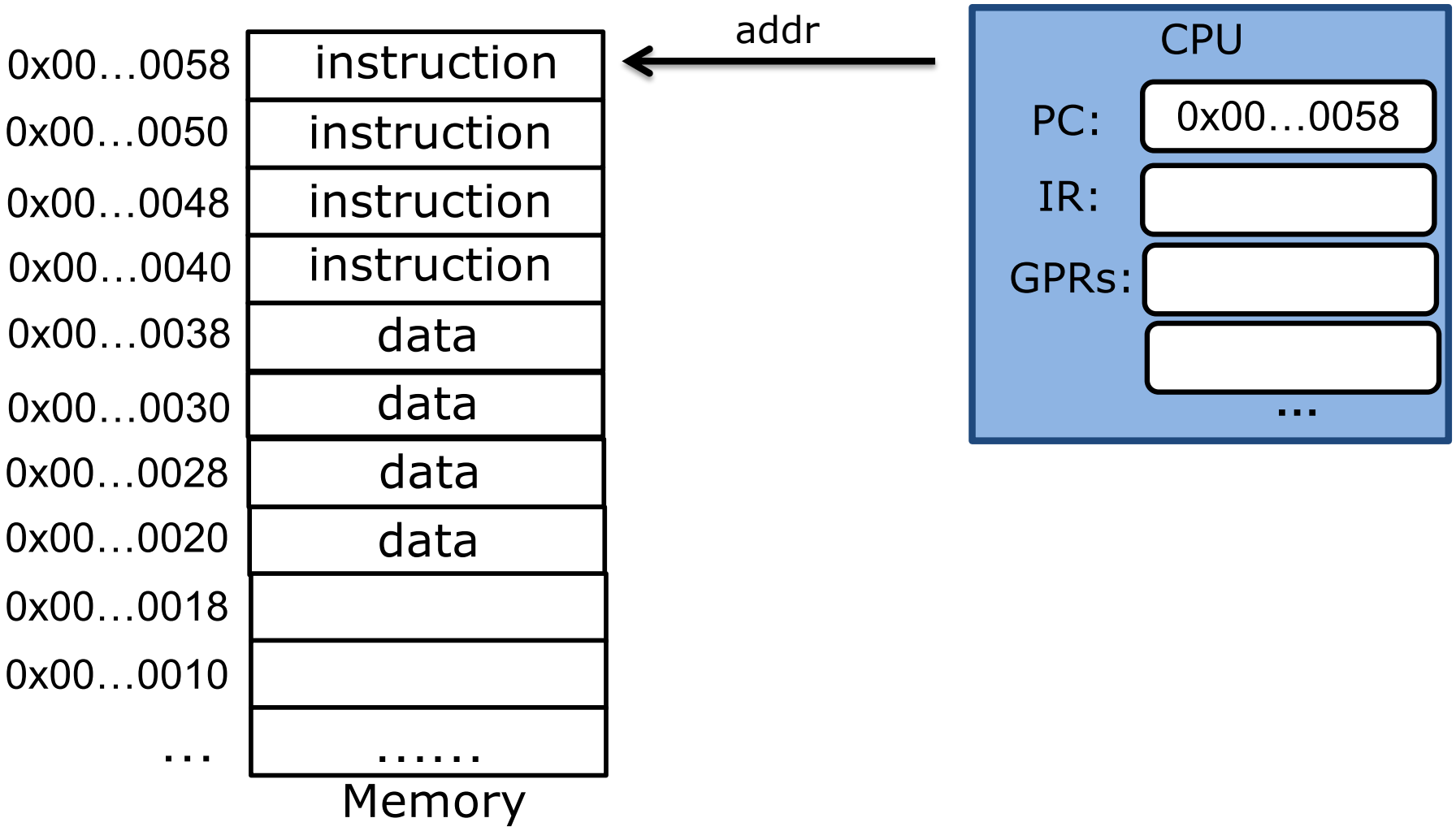
IR: instruction register
- Store the fetched instruction

General purpose registers:
- Store operands and pointers used by program

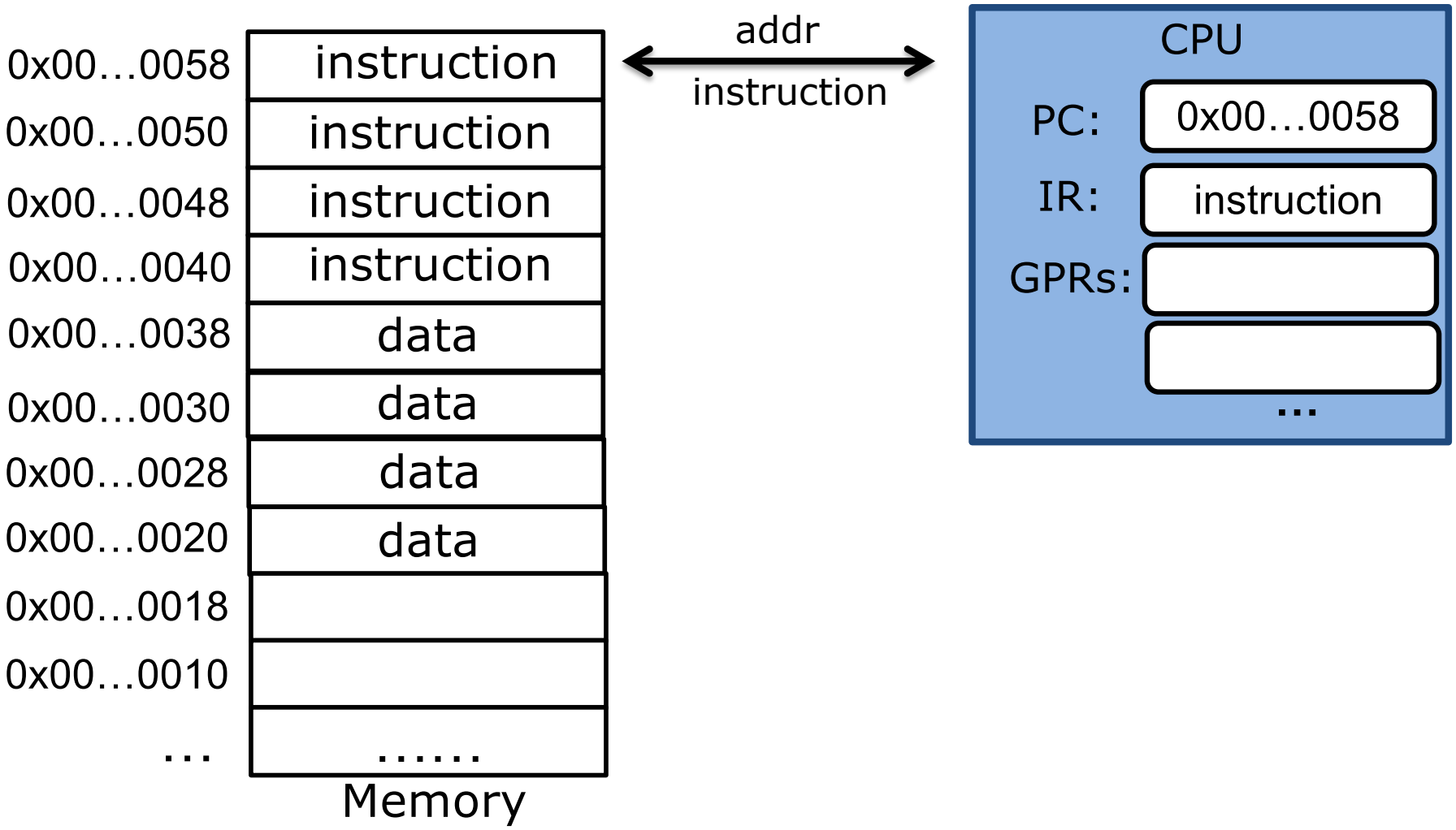
Program status and control register:
- Status of the program being executed
- All called “RFLAGS” in x86_64

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

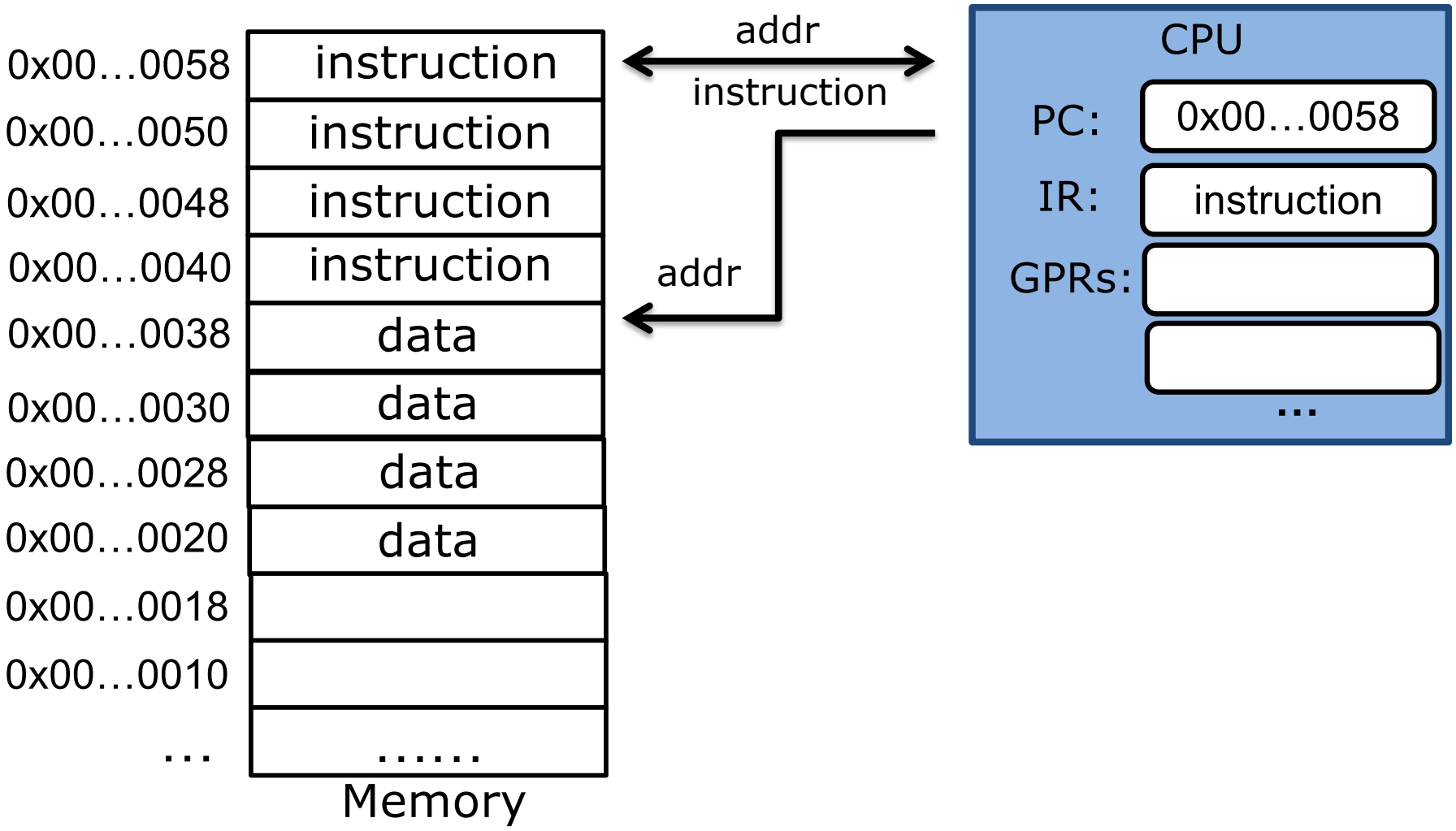
# CPU execution



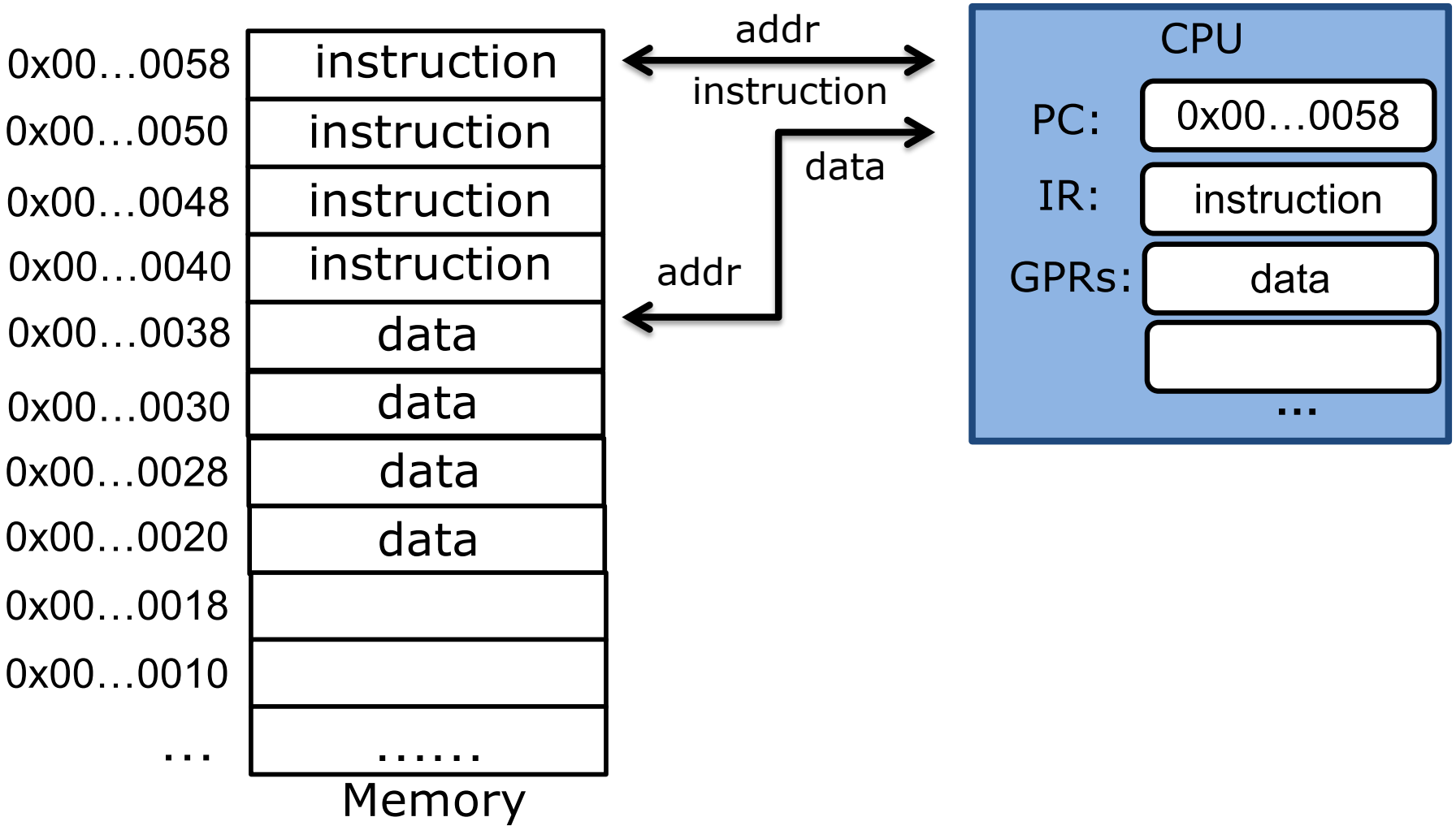
# CPU execution



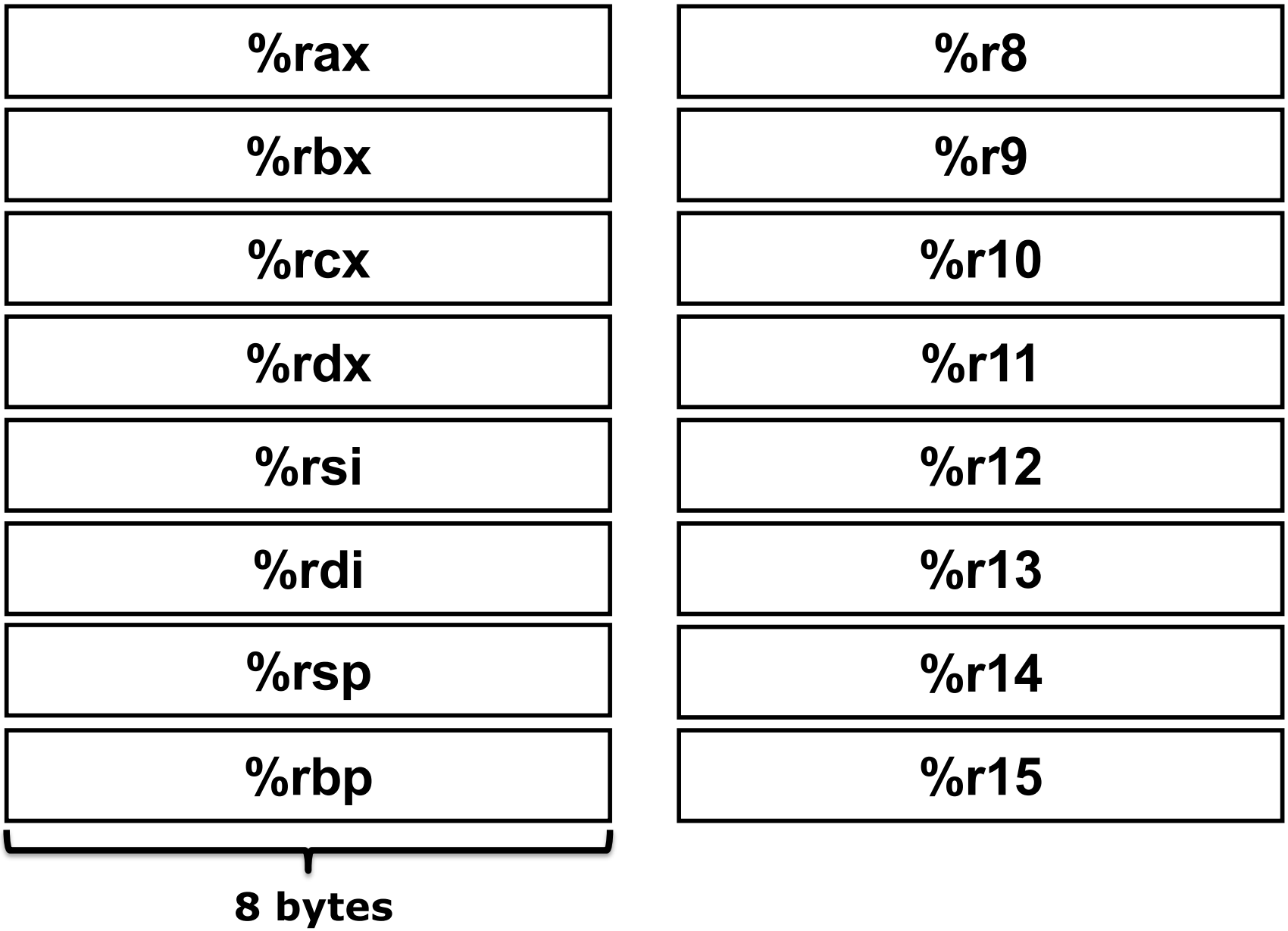
# CPU execution



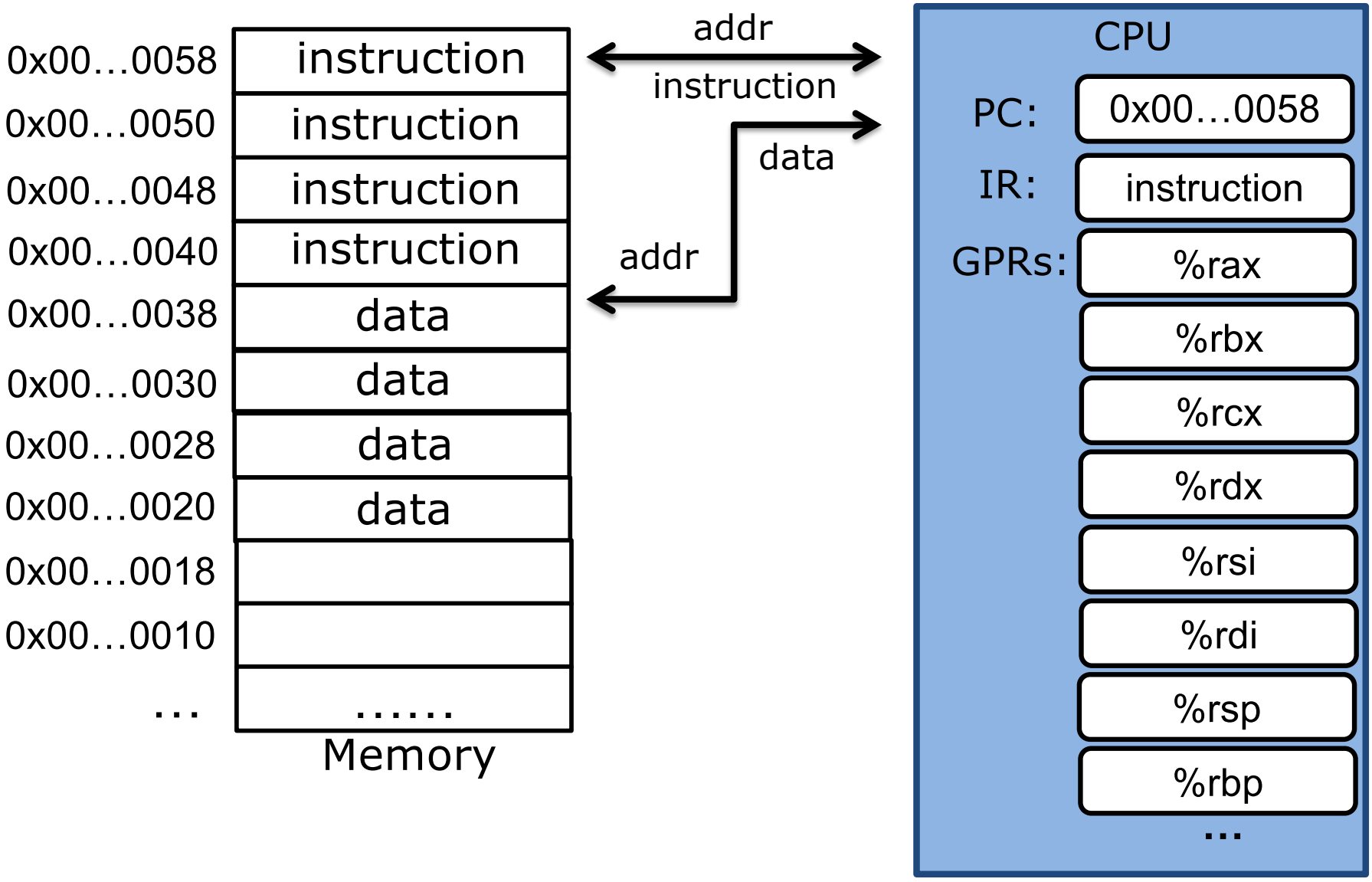
# CPU execution



# General Purpose Registers (intel x86-64)



# CPU execution

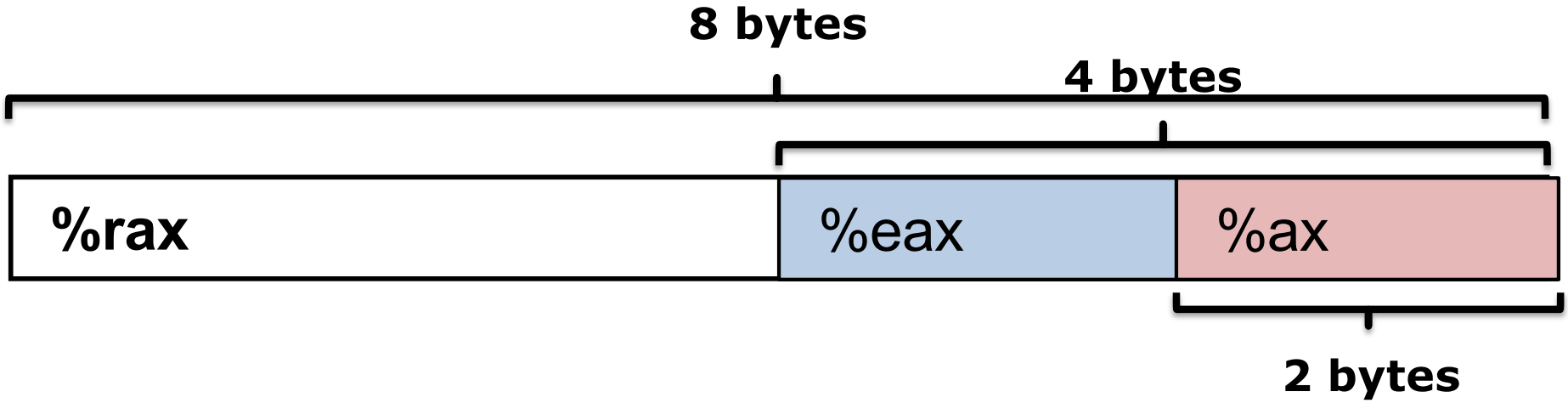


# General Purpose Registers (intel x86-64)

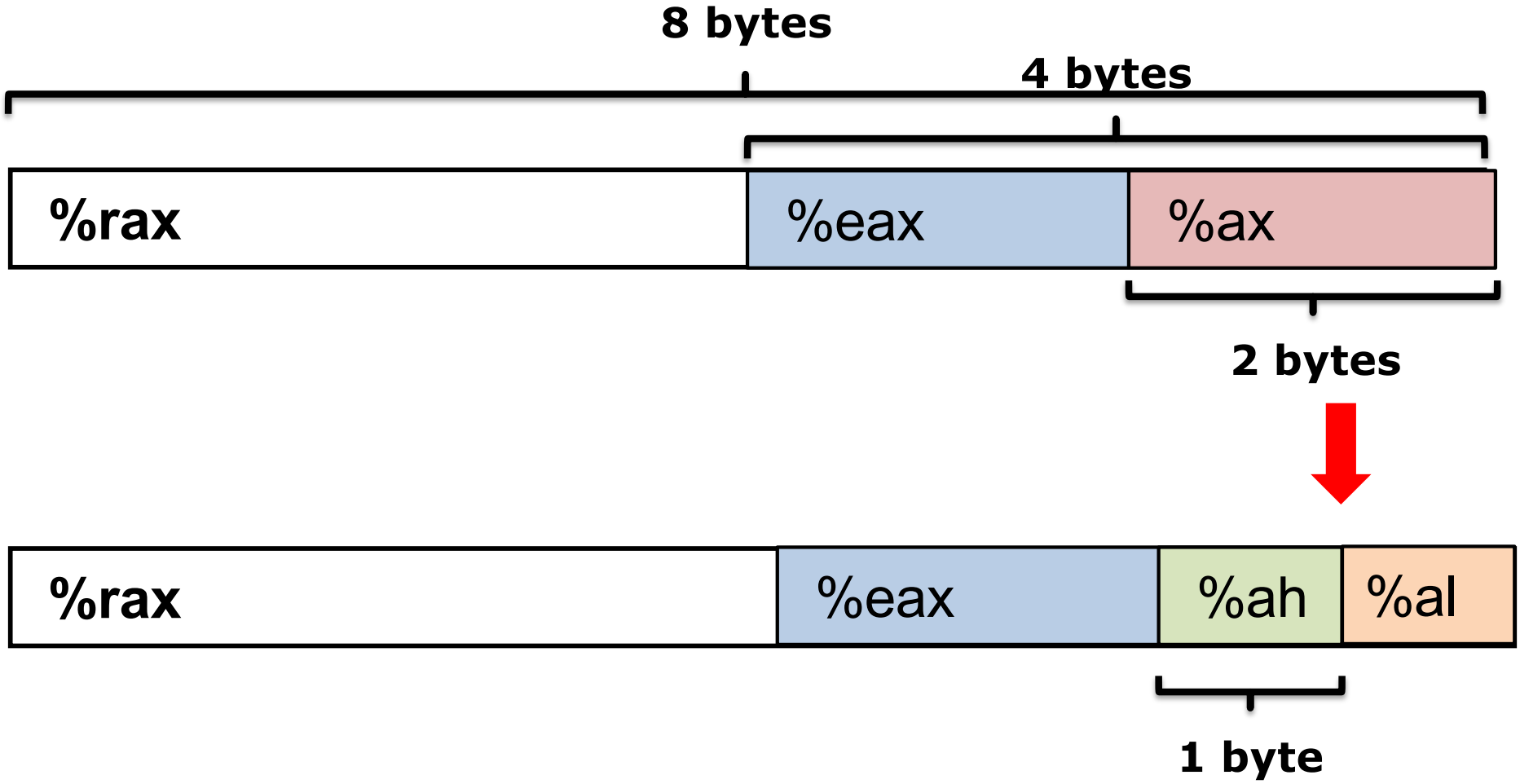
| | | | |
|---|---|---|---|
| **%rax** | %eax | **%r8** | %r8d |
| **%rbx** | %ebx | **%r9** | %r9d |
| **%rcx** | %ecx | **%r10** | %r10d |
| **%rdx** | %edx | **%r11** | %r11d |
| **%rsi** | %esi | **%r12** | %r12d |
| **%rdi** | %edi | **%r13** | %r13d |
| **%rsp** | %esp | **%r14** | %r14d |
| **%rbp** | %ebp | **%r15** | %r15d |

**8 bytes** **4 bytes**

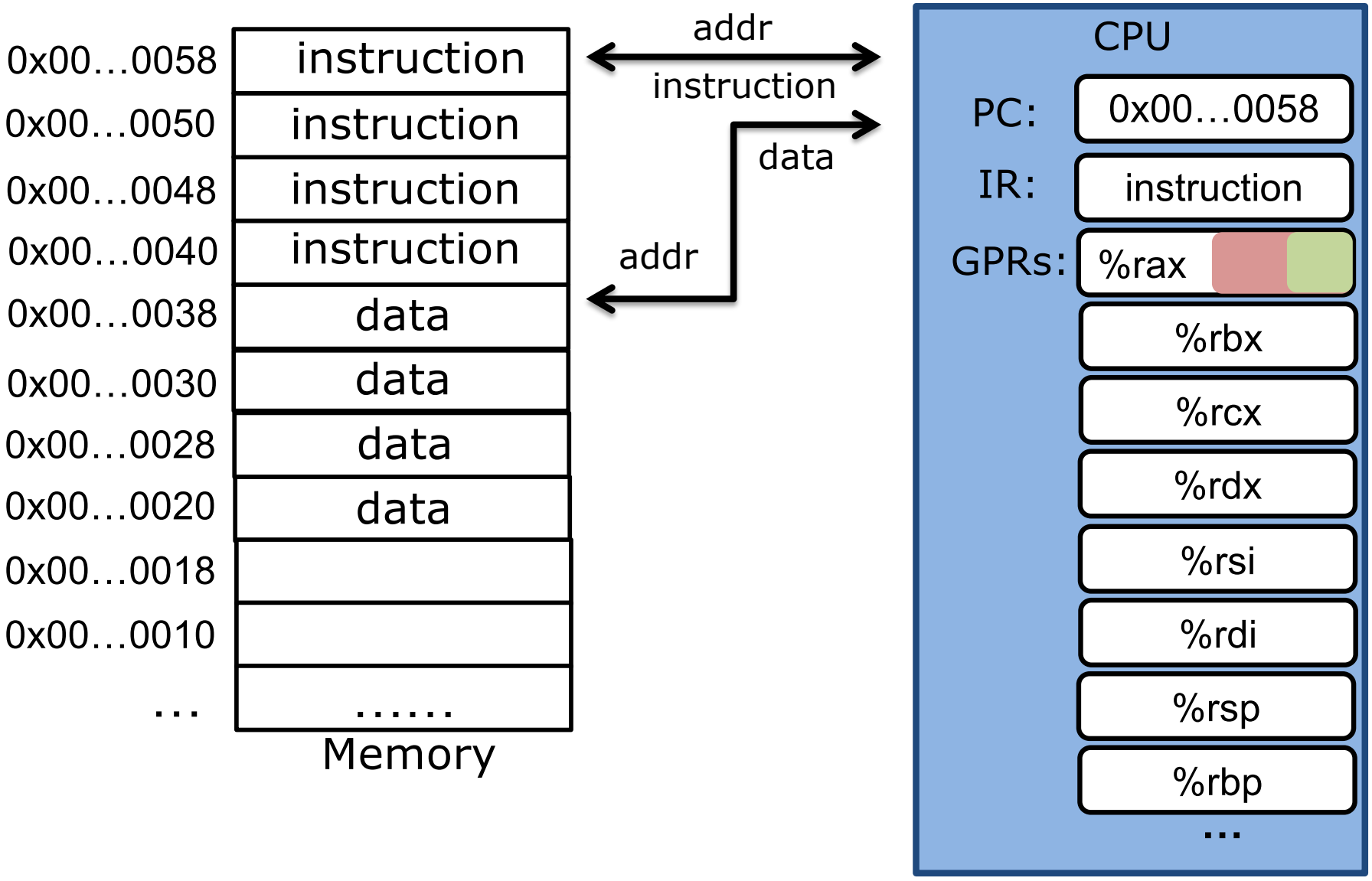
# Use %rax as an example



# 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)

- ISA: interface exposed by hardware to software writers

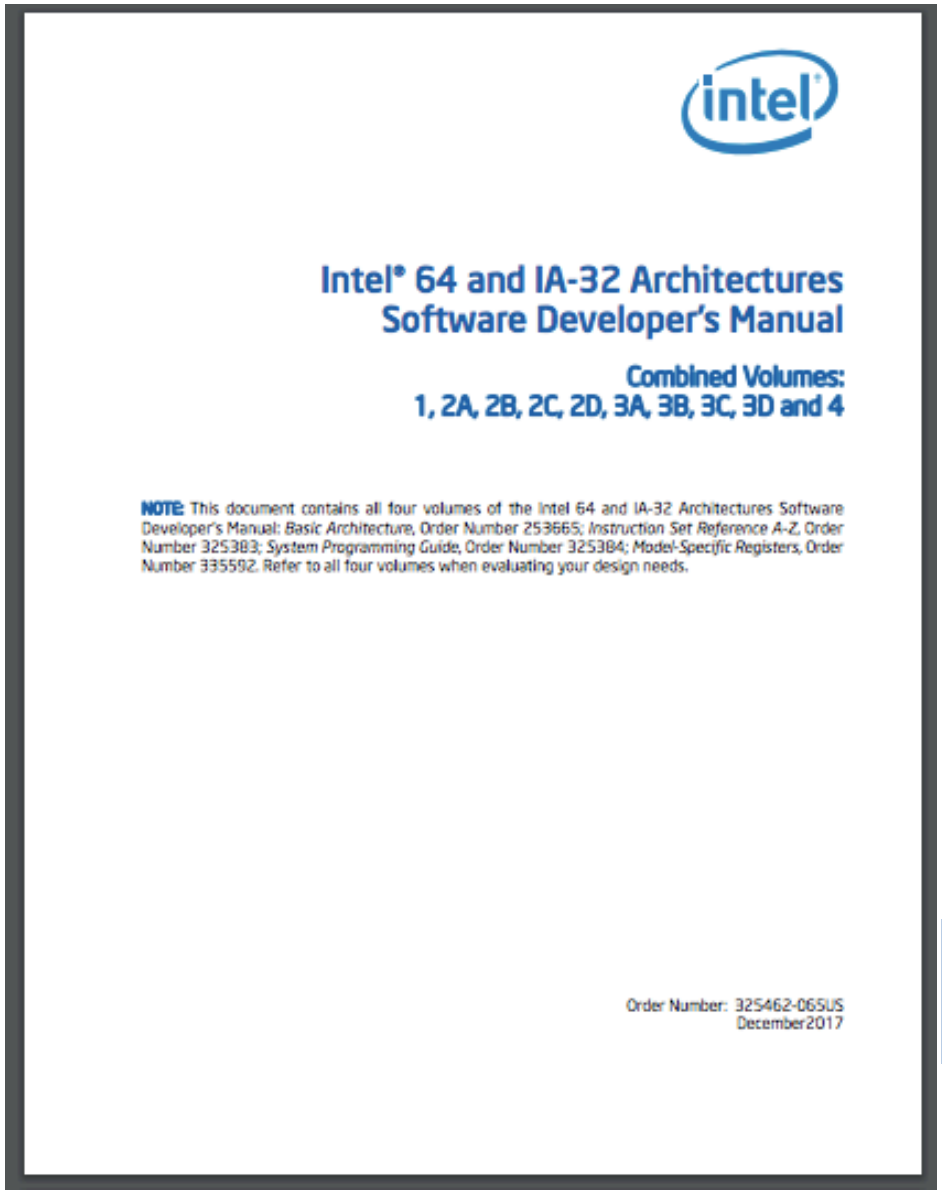
- X86_64 is the ISA implemented by Intel/AMD CPUs
  - 64-bit version of x86

taught by CSO

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

- RISC-V is yet another ISA
  - P&H textbook's ISA.
  - Open-sourced, royalty-free

# 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-bit) from the source operand (first operand) to the destination operand (second operand).

# Moving data

suffix

**movq** *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*

- Support **full backward compatibility**
  - New processor can run the same binary file compiled for older processors
- In the Intel x86 world, a word = 16 bits.
  - 8086 refers to 16 bits as a word

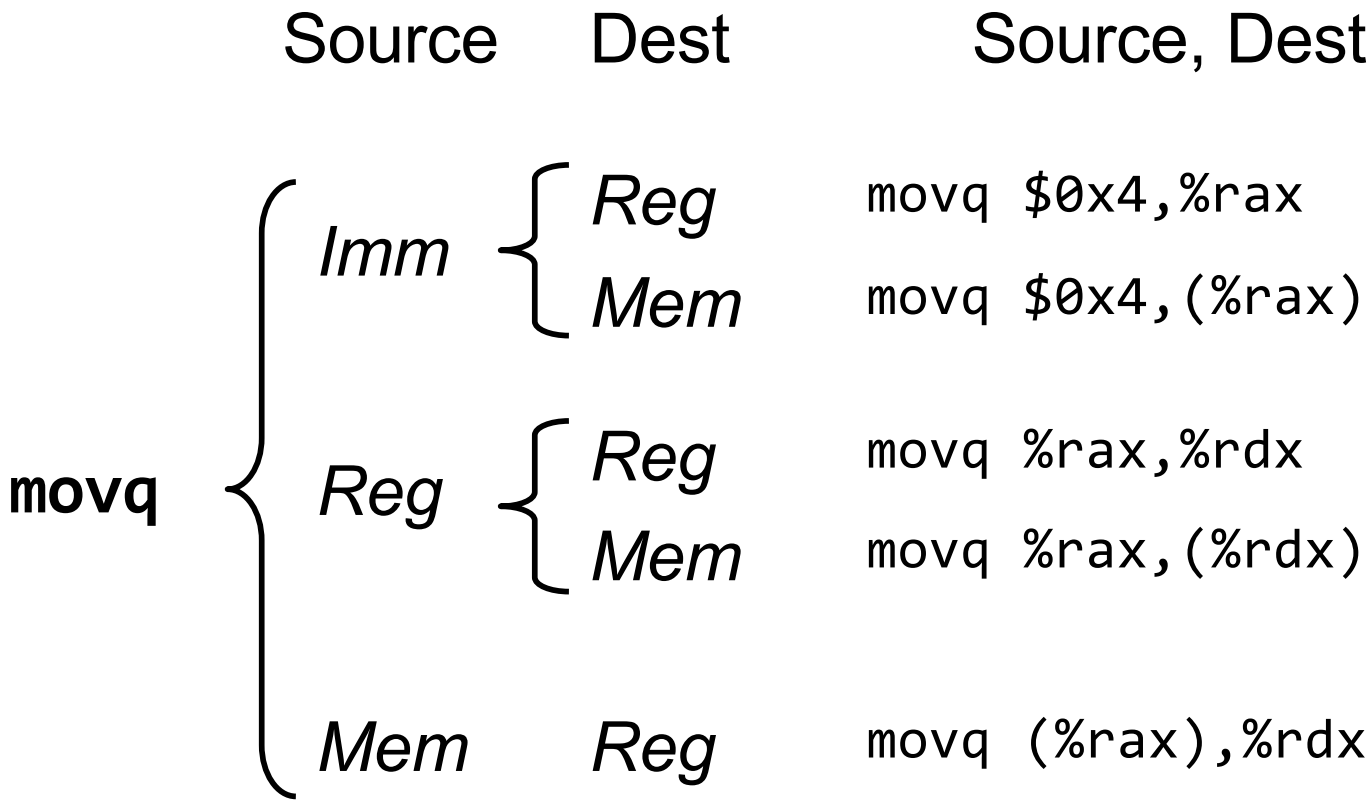
# Moving data

**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)`**

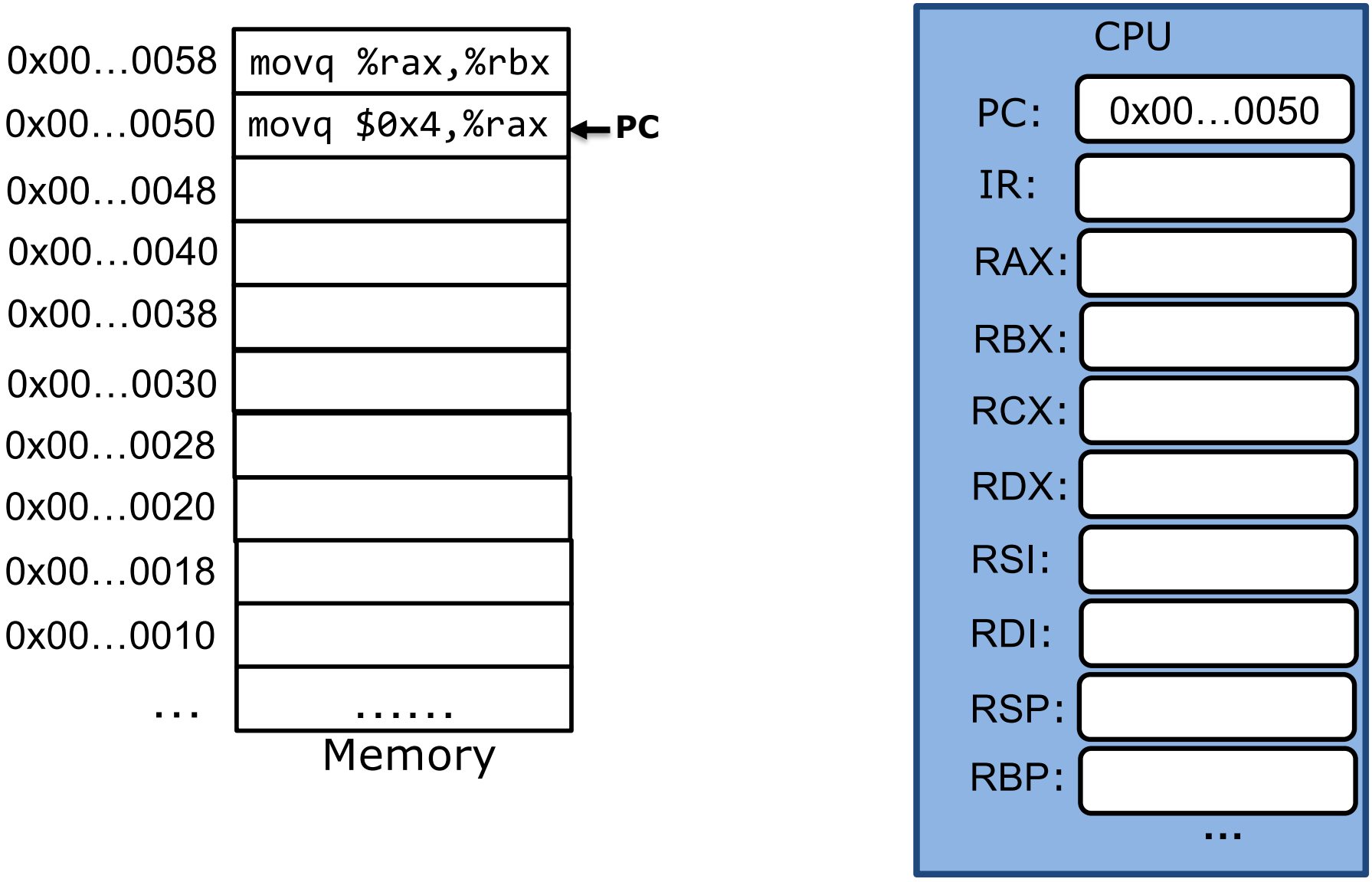
# movq Operand combinations



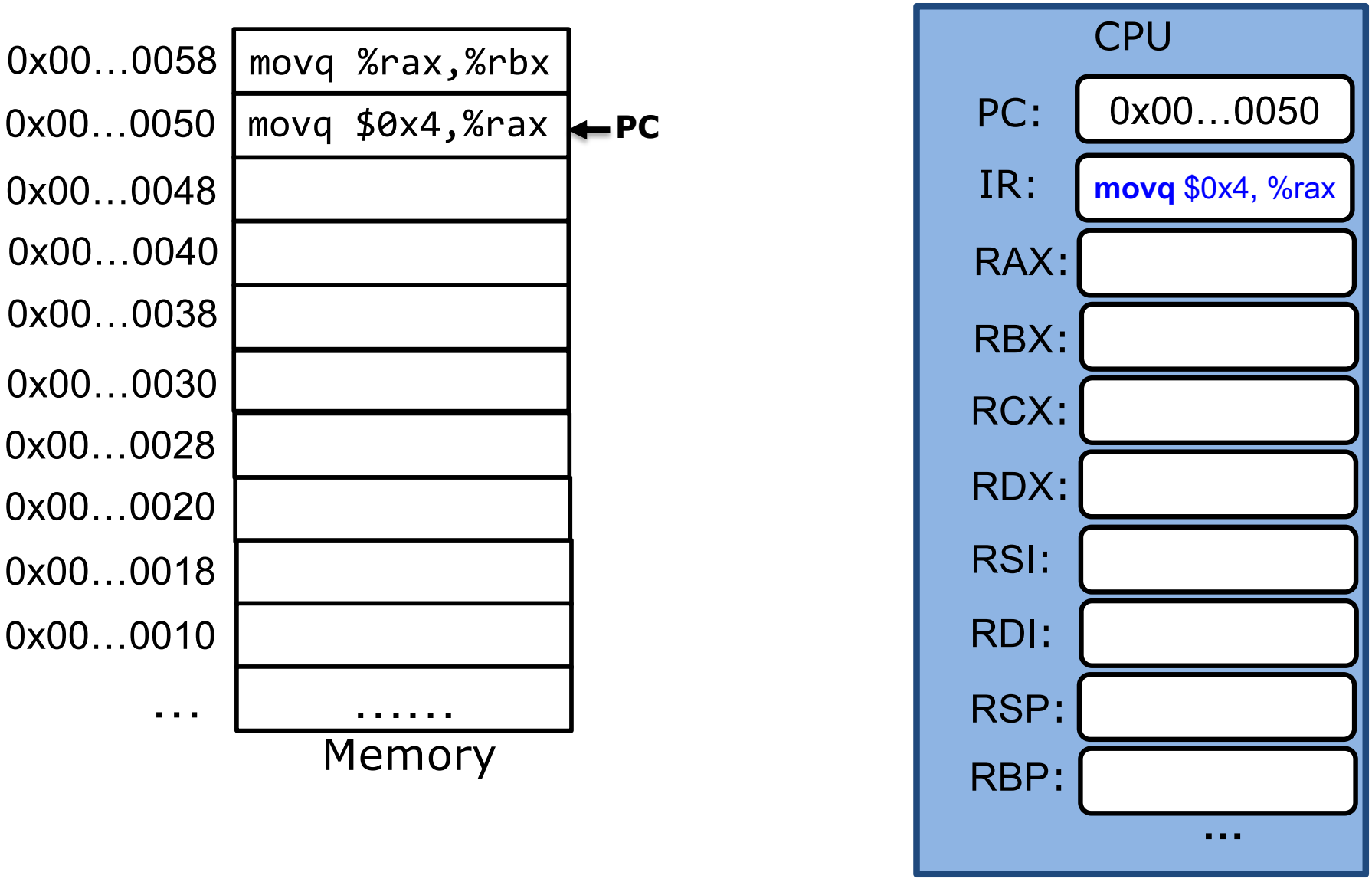

1. Immediate can only be *Source*

2. No memory-memory mov

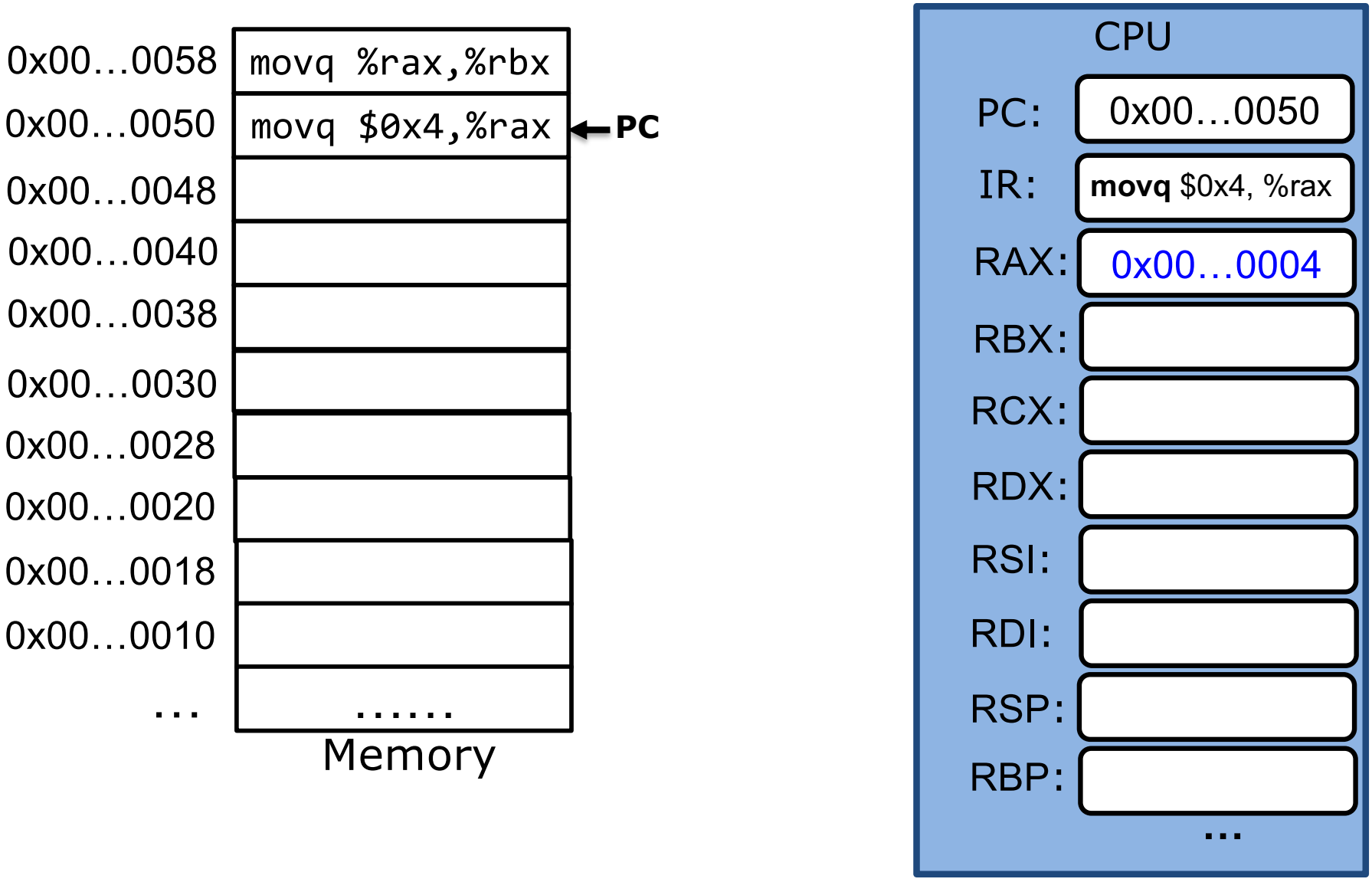
# **movq** *Imm, Reg*


0x00…0058
movq %rax,%rbx
0x00…0050
movq $0x4,%rax
PC
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory
CPU
PC:
0x00…0050
IR:
RAX:
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
RBP:
…

# **movq** *Imm, Reg*


0x00…0058
movq %rax,%rbx
0x00…0050
movq $0x4,%rax
PC
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory
CPU
PC:
0x00…0050
IR:
movq $0x4, %rax
RAX:
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
RBP:
…

# **movq** *Imm, Reg*


0x00…0058
movq %rax,%rbx
0x00…0050
movq $0x4,%rax
PC
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory
CPU
PC:
0x00…0050
IR:
movq $0x4, %rax
RAX:
0x00…0004
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
RBP:
…

# Steps

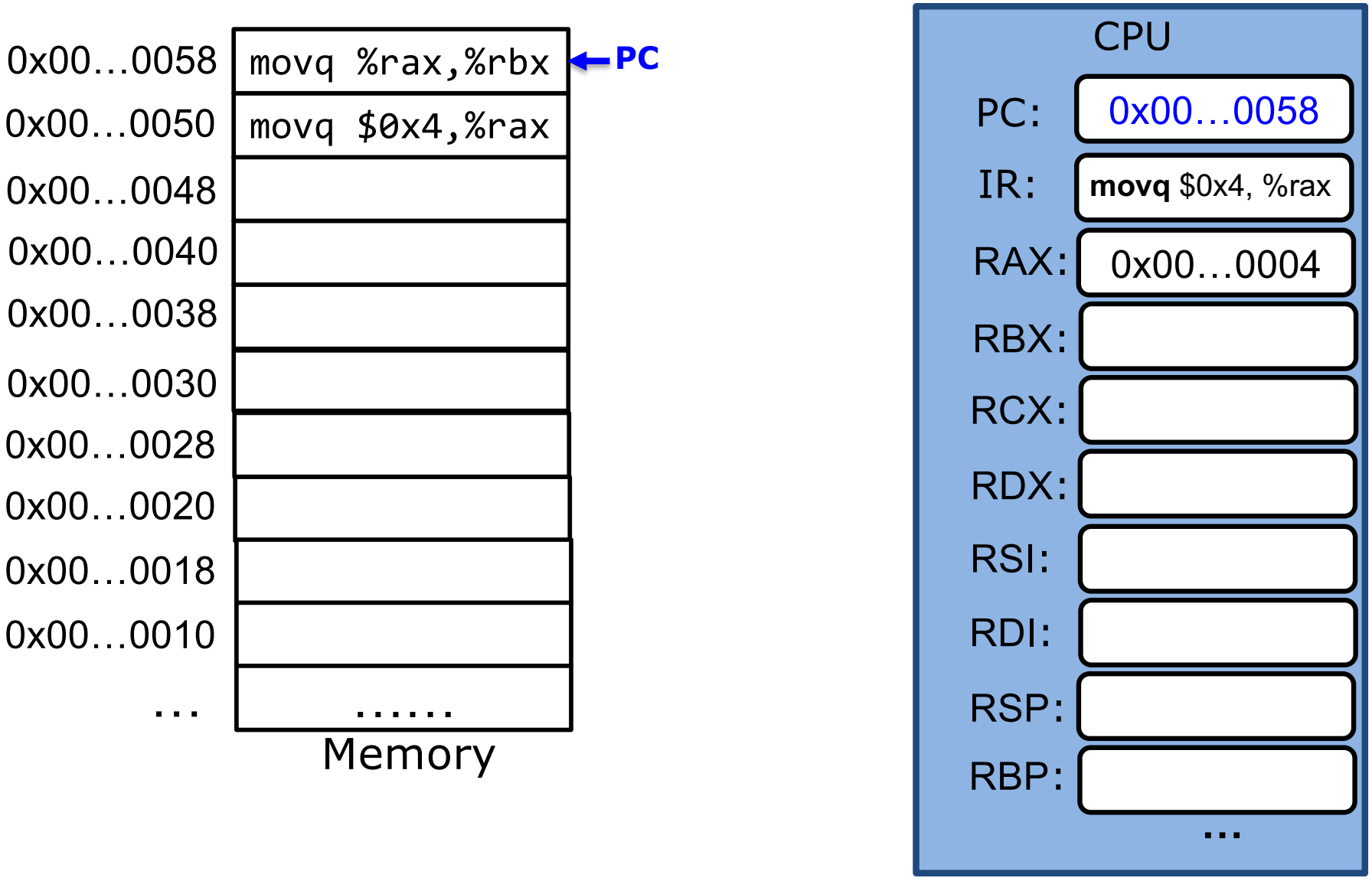
1. PC contains the instruction's address

2. Load the instruction into IR

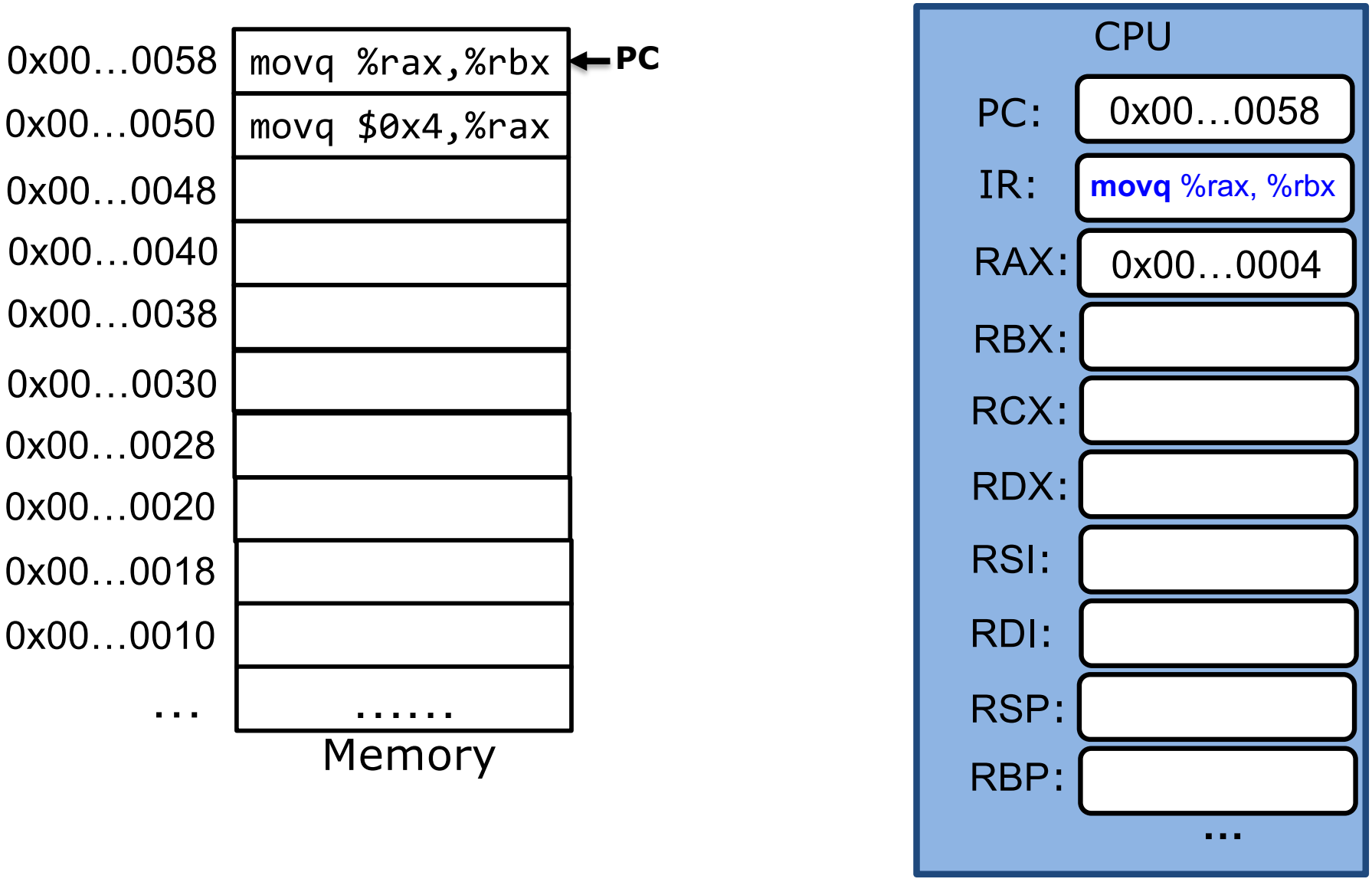
3. Execute the instruction

4. CPU automatically updates PC after current instruction finishes (is retired).

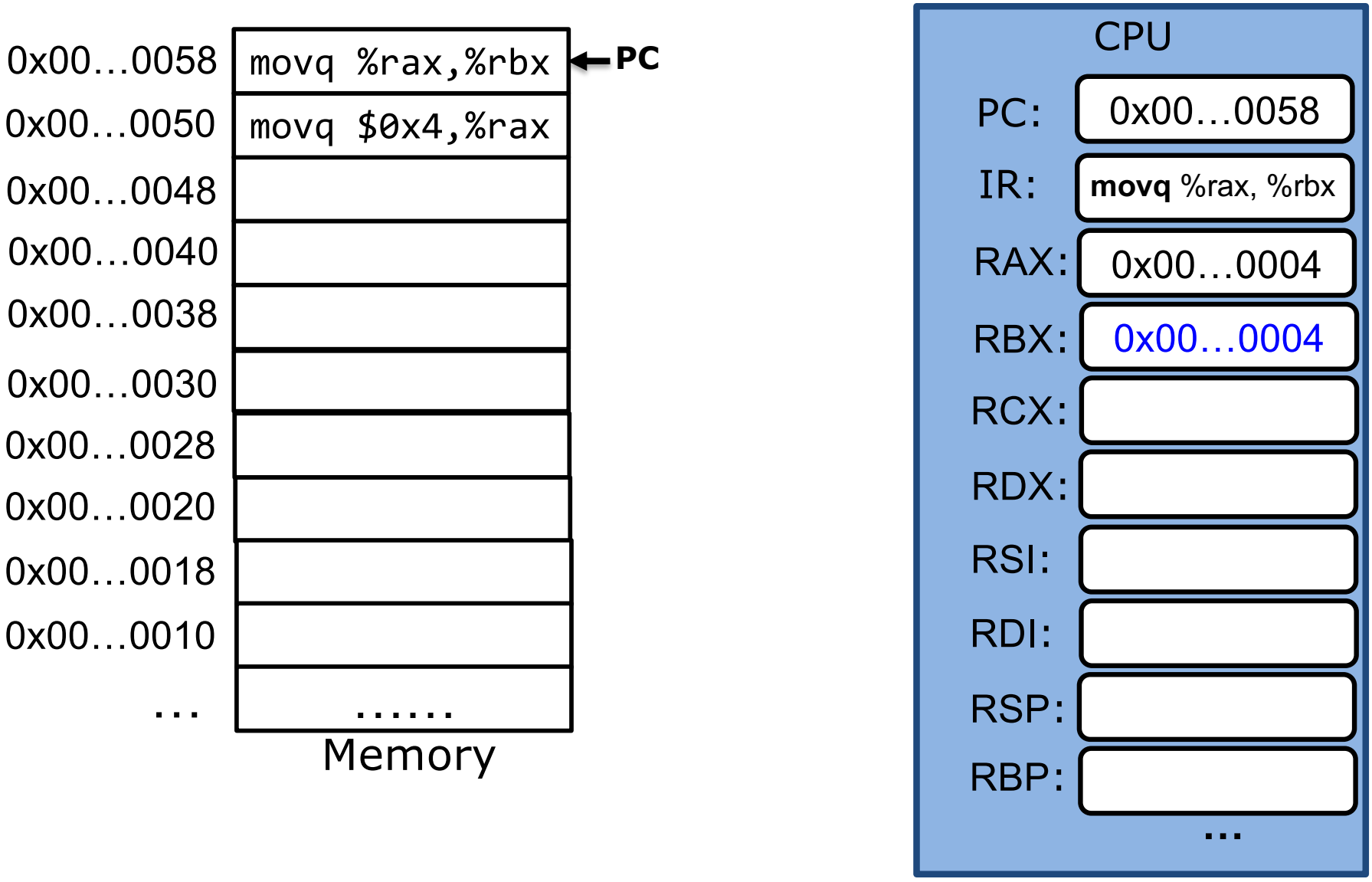
# **movq** *Reg, Reg*


0x00…0058
movq %rax,%rbx
PC
0x00…0050
movq $0x4,%rax
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory
CPU
PC:
0x00…0058
IR:
movq $0x4, %rax
RAX:
0x00…0004
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
RBP:
…

# **movq** *Reg, Reg*


0x00…0058
movq %rax,%rbx
PC
0x00…0050
movq $0x4,%rax
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory
CPU
PC:
0x00…0058
IR:
movq %rax, %rbx
RAX:
0x00…0004
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
RBP:
…

# **movq** *Reg, Reg*


0x00…0058
movq %rax,%rbx
PC
0x00…0050
movq $0x4,%rax
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory
CPU
PC:
0x00…0058
IR:
movq %rax, %rbx
RAX:
0x00…0004
RBX:
0x00…0004
RCX:
RDX:
RSI:
RDI:
RSP:
RBP:
…

# **movq** *Mem, Reg*

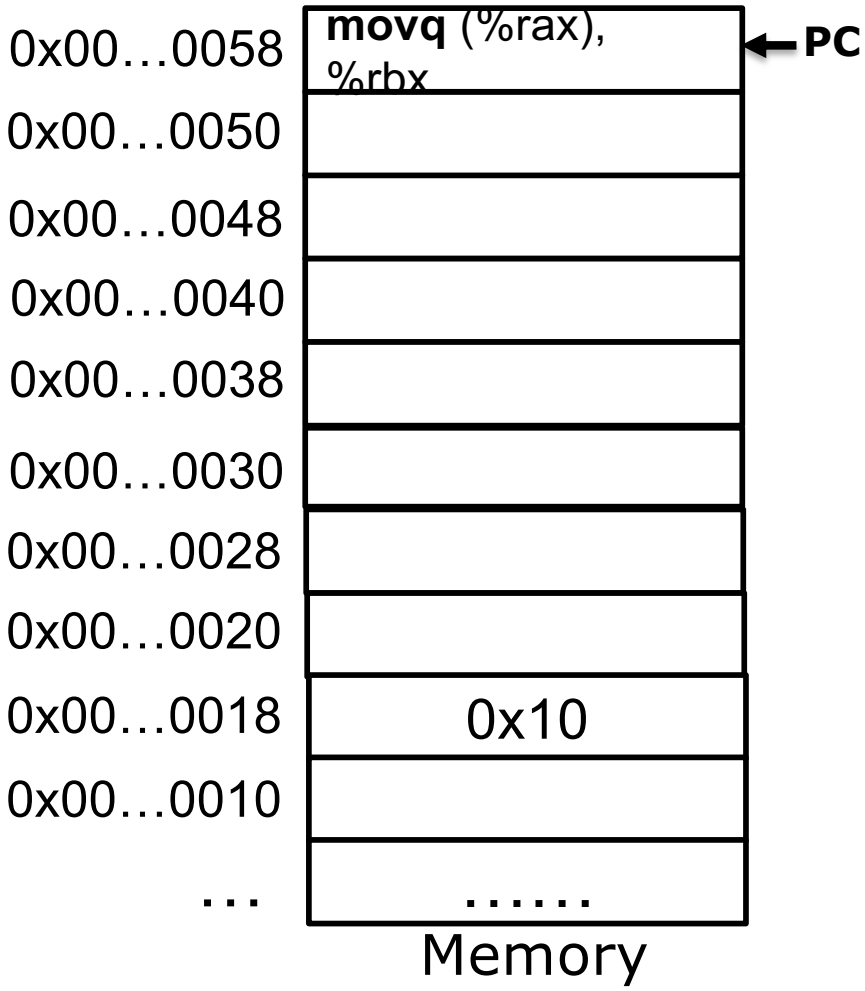
How to represent a “memory” operand?

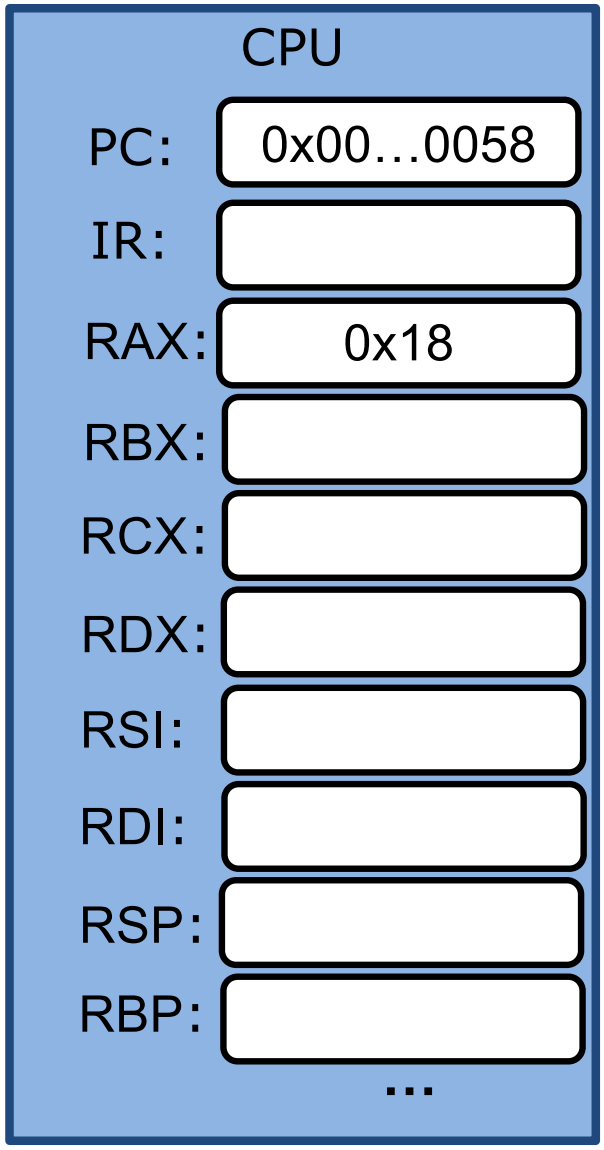
# Direct addressing: use registers to index the memory

(Register)

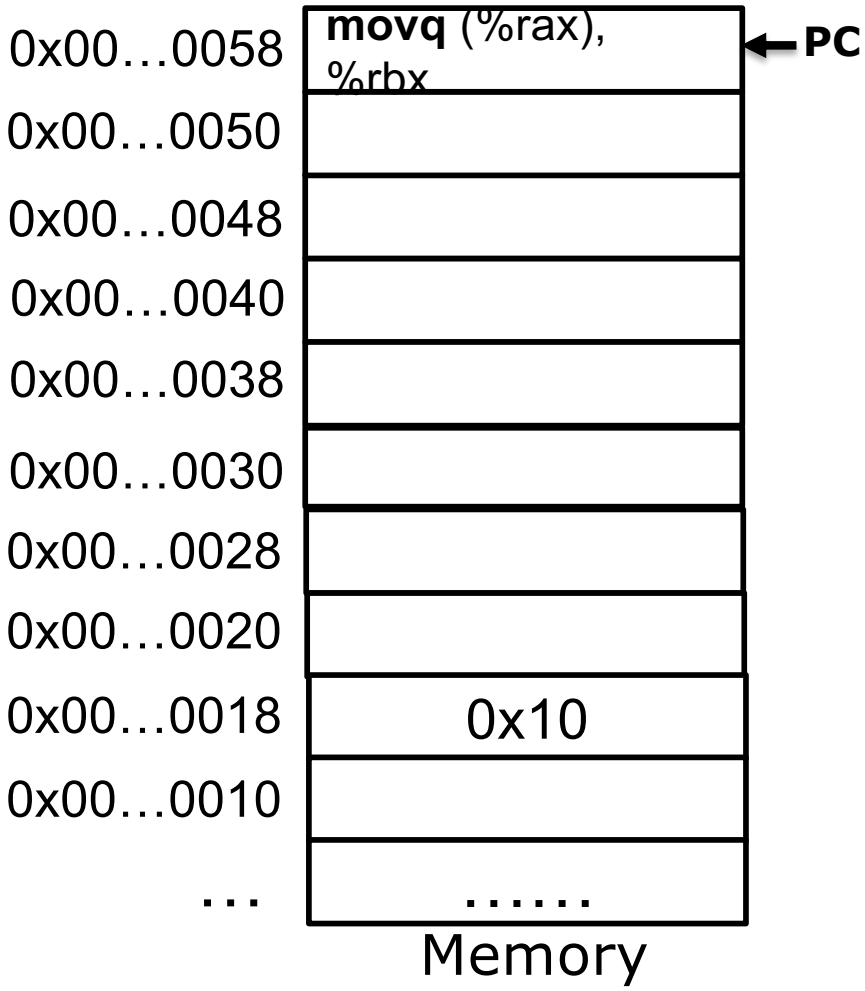
- The content of the register specifies memory address
- movq (%rax), %rbx

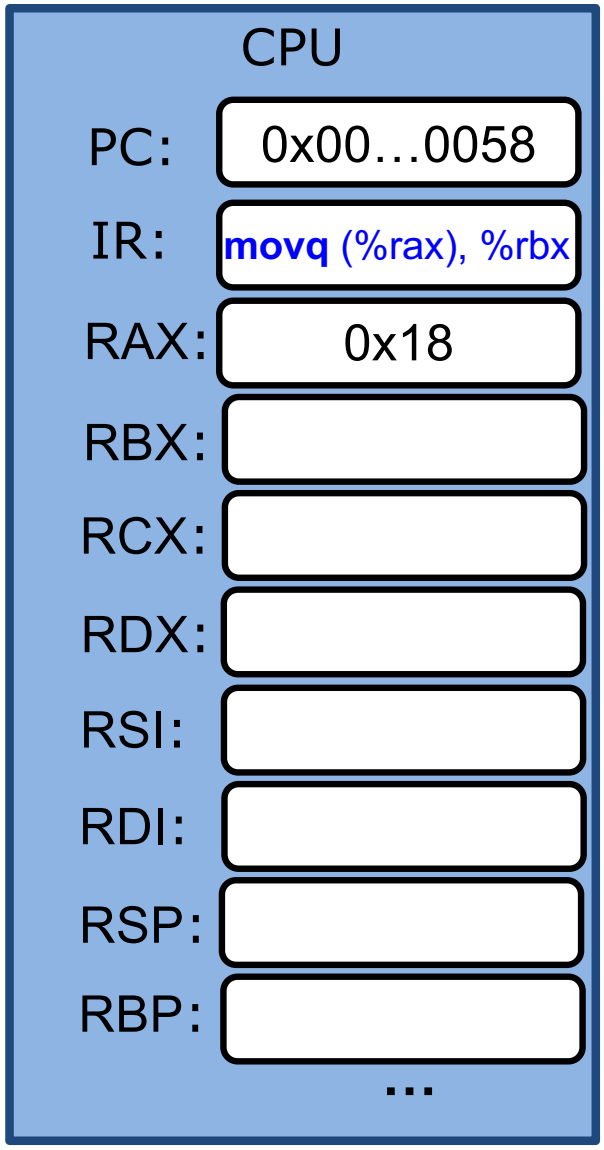
# **movq** *(%rax), %rbx*


0x00…0058
movq (%rax), %rbx
PC
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x10
0x00…0010
…
……
Memory

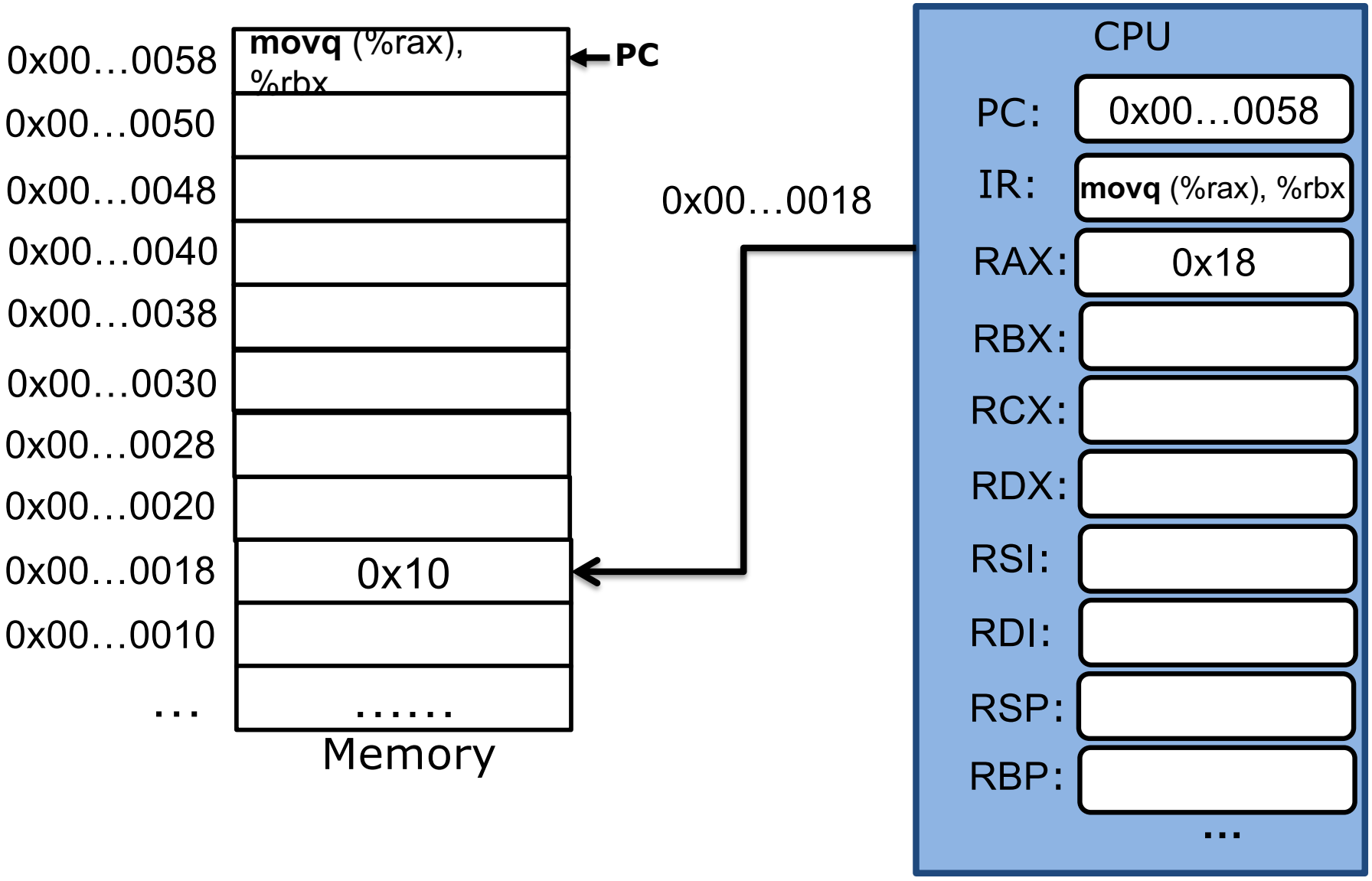

CPU
PC:
0x00…0058
IR:
RAX:
0x18
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
RBP:
…

# **movq** *(%rax), %rbx*

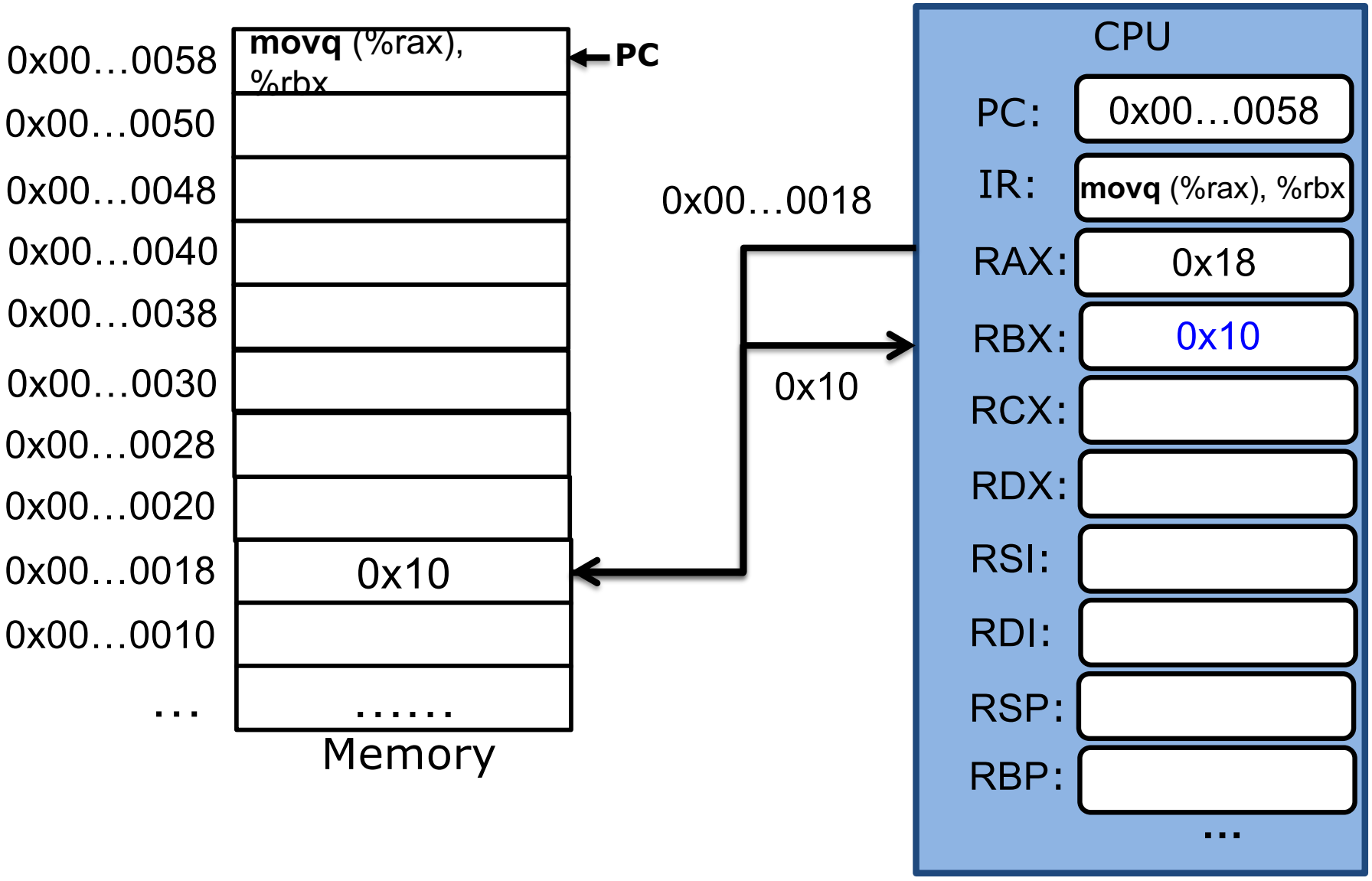

0x00…0058
movq (%rax), %rbx
PC
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x10
0x00…0010
…
……
Memory


CPU
PC:
0x00…0058
IR:
movq (%rax), %rbx
RAX:
0x18
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
RBP:
…

# **movq** *(%rax), %rbx*


0x00…0058
movq (%rax), %rbx
PC
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x10
0x00…0010
…
……
Memory
0x00…0018
CPU
PC:
0x00…0058
IR:
movq (%rax), %rbx
RAX:
0x18
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
RBP:
…

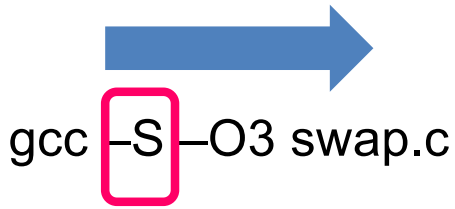
# **movq** *(%rax), %rbx*


0x00…0058
movq (%rax), %rbx
PC
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x10
0x00…0010
…
……
Memory
0x00…0018
0x10
CPU
PC:
0x00…0058
IR:
movq (%rax), %rbx
RAX:
0x18
RBX:
0x10
RCX:
RDX:
RSI:
RDI:
RSP:
RBP:
…

# swap function

```
void
swap(long *a, long* b) {

    long tmp = *a;
    *a = *b;
    *b = tmp;

}
```

**swap:**




Makes gcc output assembly
(human readable machine instructions)

# 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

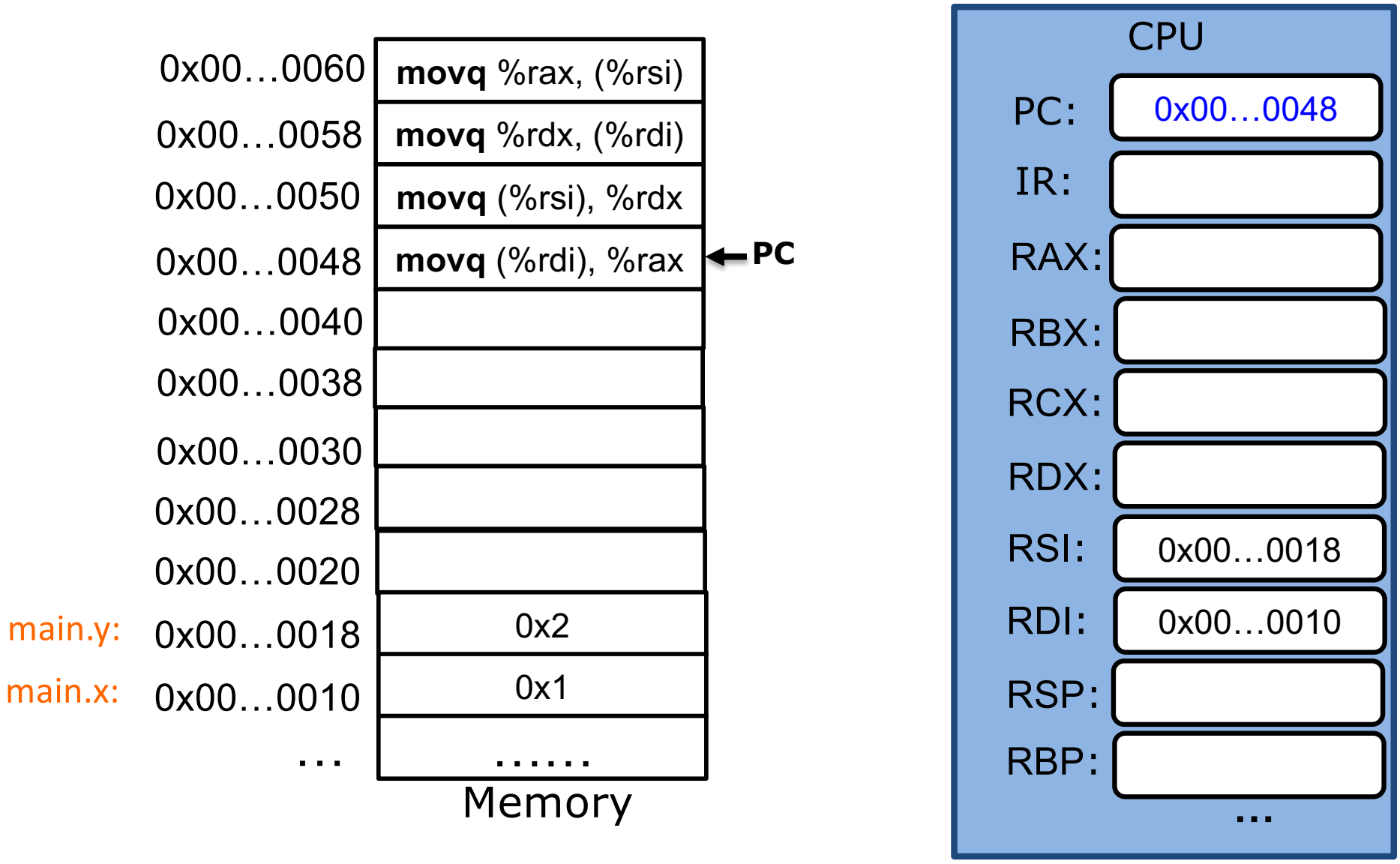
# swap function

```
void
swap(long *a, long* b) {

    long tmp = *a;
    *a = *b;
    *b = tmp;

}
```

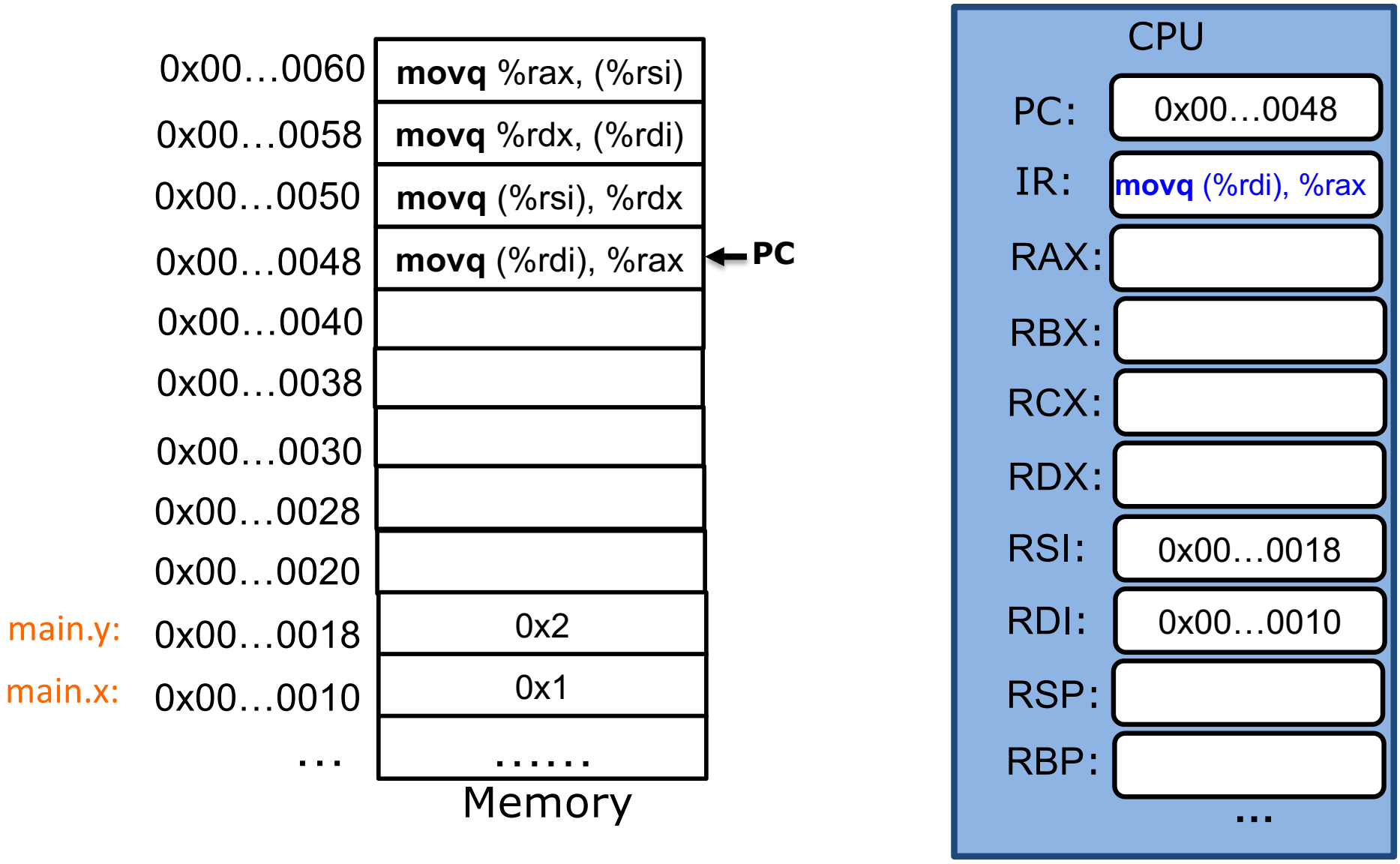
gcc –S –O3 swap.c

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

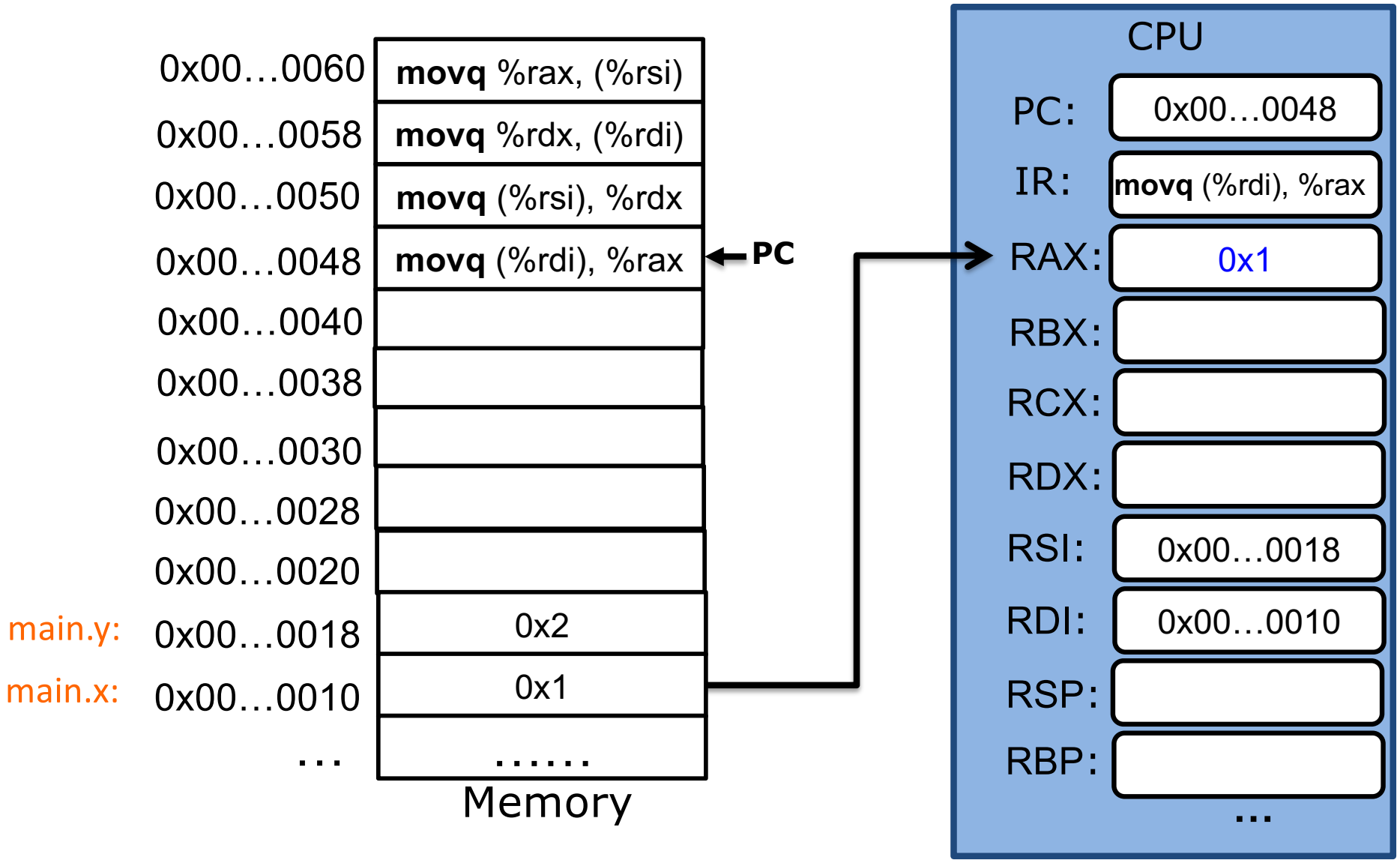
# swap func


0x00…0060
movq %rax, (%rsi)
0x00…0058
movq %rdx, (%rdi)
0x00…0050
movq (%rsi), %rdx
0x00…0048
movq (%rdi), %rax
PC
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
main.y:
0x00…0018
0x2
main.x:
0x00…0010
0x1
…
……
Memory
CPU
PC:
0x00…0048
IR:
RAX:
RBX:
RCX:
RDX:
RSI:
0x00…0018
RDI:
0x00…0010
RSP:
RBP:
…

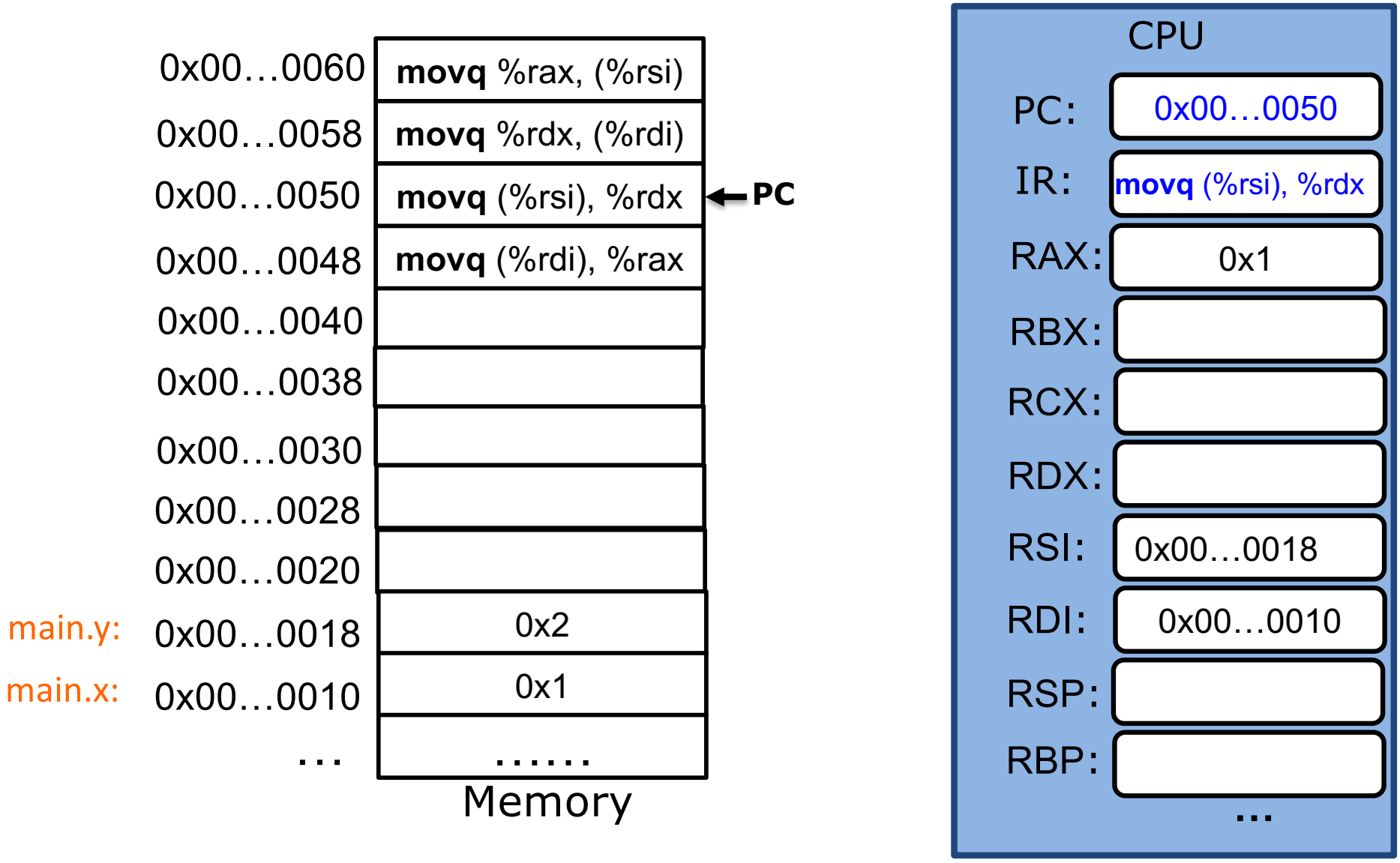
# swap func


0x00…0060
movq %rax, (%rsi)
0x00…0058
movq %rdx, (%rdi)
0x00…0050
movq (%rsi), %rdx
0x00…0048
movq (%rdi), %rax
PC
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
main.y:
0x00…0018
0x2
main.x:
0x00…0010
0x1
…
……
Memory
CPU
PC:
0x00…0048
IR:
movq (%rdi), %rax
RAX:
RBX:
RCX:
RDX:
RSI:
0x00…0018
RDI:
0x00…0010
RSP:
RBP:
…

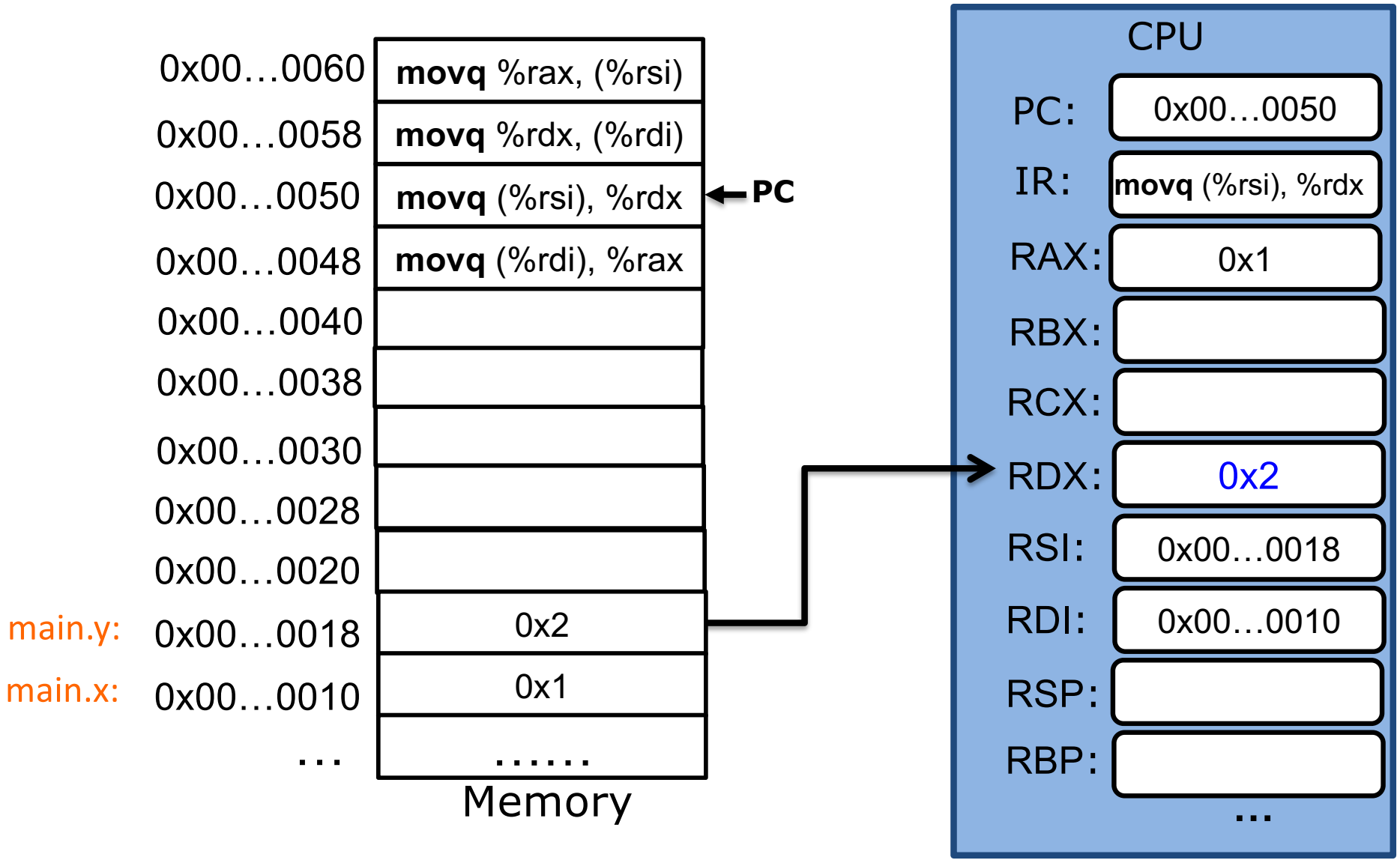
# swap func


0x00…0060
movq %rax, (%rsi)
0x00…0058
movq %rdx, (%rdi)
0x00…0050
movq (%rsi), %rdx
0x00…0048
movq (%rdi), %rax
PC
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
main.y:
0x00…0018
0x2
main.x:
0x00…0010
0x1
…
……
Memory
CPU
PC:
0x00…0048
IR:
movq (%rdi), %rax
RAX:
0x1
RBX:
RCX:
RDX:
RSI:
0x00…0018
RDI:
0x00…0010
RSP:
RBP:
…

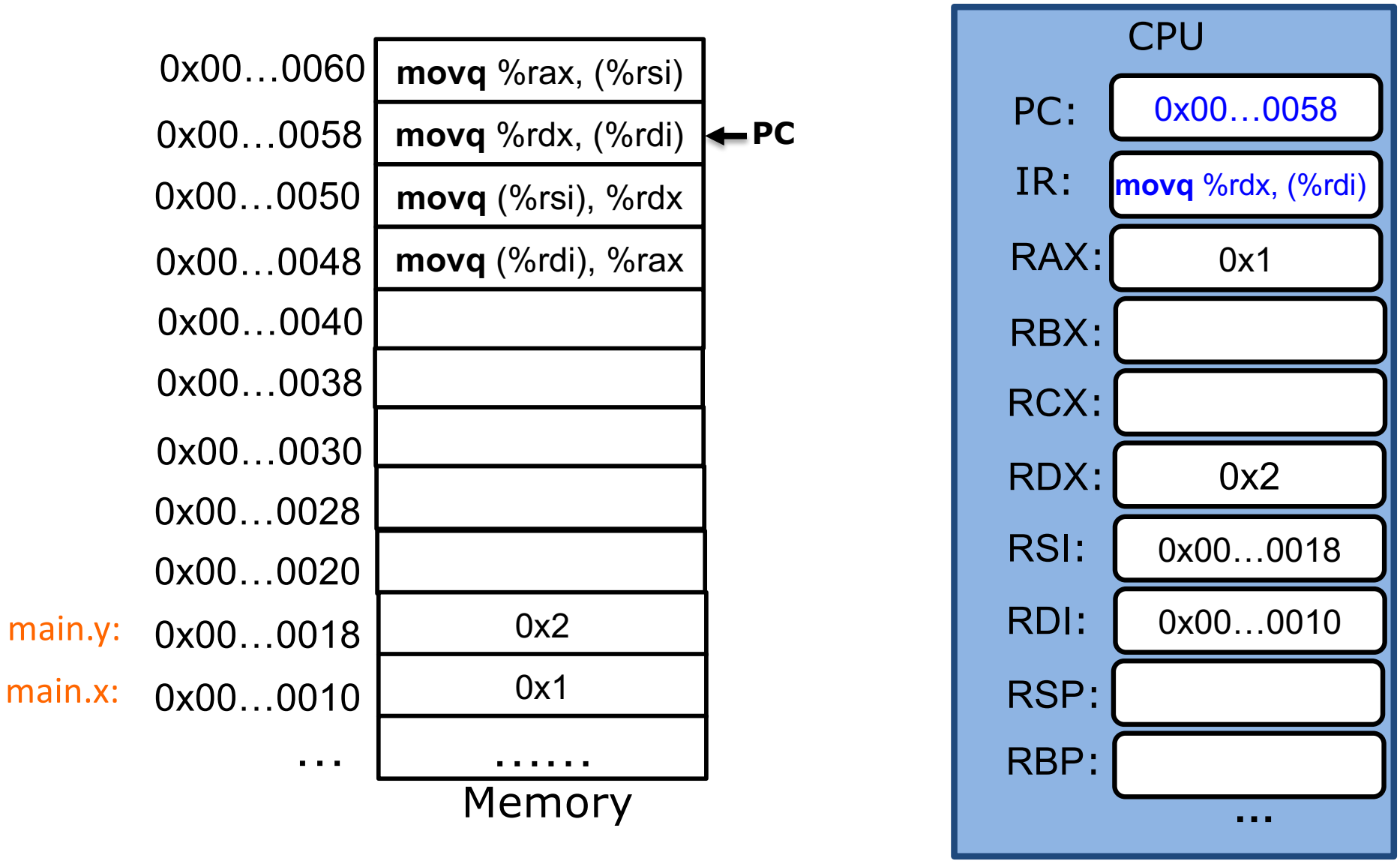
# swap func


0x00…0060
movq %rax, (%rsi)
0x00…0058
movq %rdx, (%rdi)
0x00…0050
movq (%rsi), %rdx
PC
0x00…0048
movq (%rdi), %rax
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
main.y:
0x00…0018
0x2
main.x:
0x00…0010
0x1
…
……
Memory
CPU
PC:
0x00…0050
IR:
movq (%rsi), %rdx
RAX:
0x1
RBX:
RCX:
RDX:
RSI:
0x00…0018
RDI:
0x00…0010
RSP:
RBP:
…

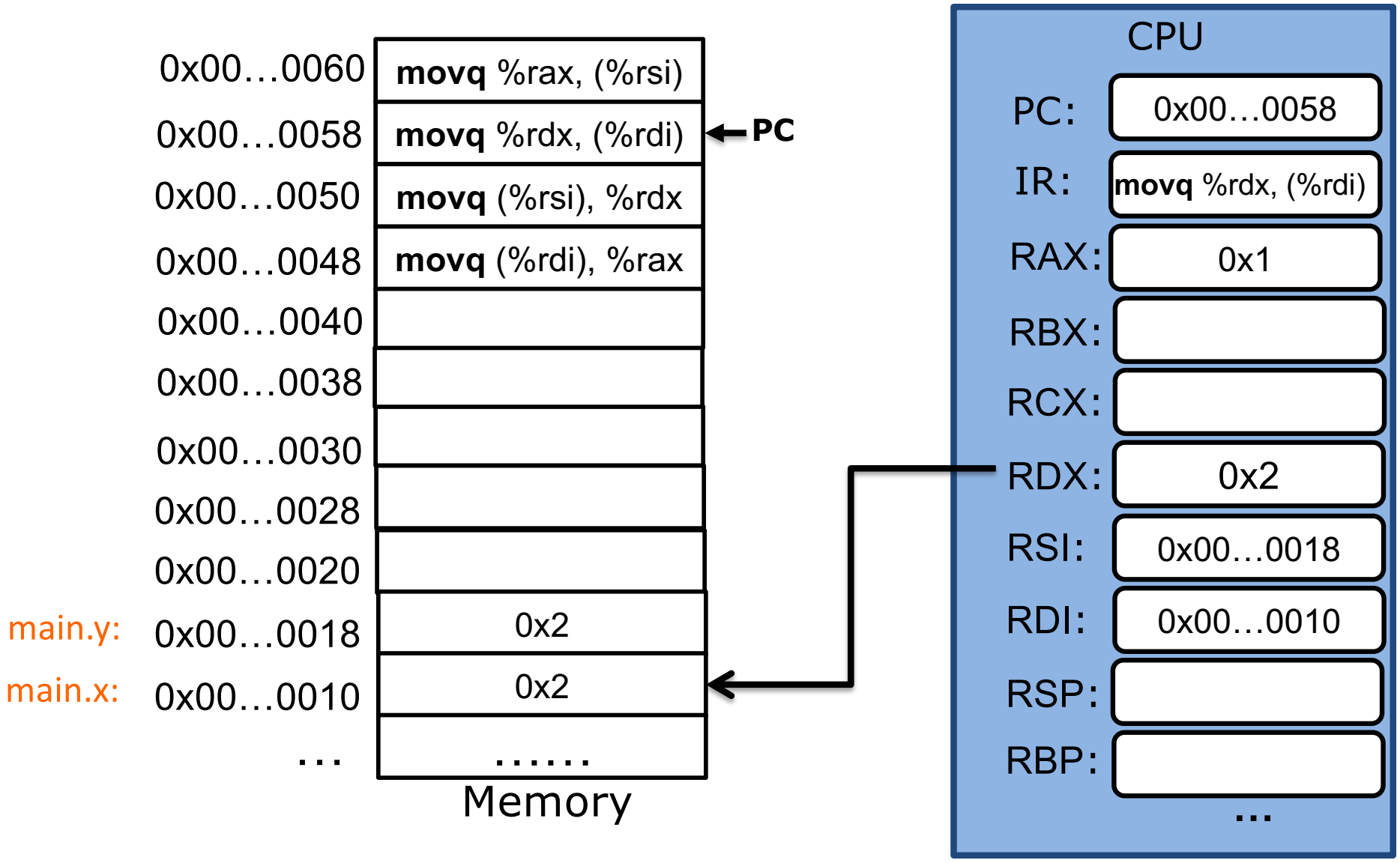
# swap func


0x00…0060
movq %rax, (%rsi)
0x00…0058
movq %rdx, (%rdi)
0x00…0050
movq (%rsi), %rdx
PC
0x00…0048
movq (%rdi), %rax
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
main.y:
0x00…0018
0x2
main.x:
0x00…0010
0x1
…
……
Memory
CPU
PC:
0x00…0050
IR:
movq (%rsi), %rdx
RAX:
0x1
RBX:
RCX:
RDX:
0x2
RSI:
0x00…0018
RDI:
0x00…0010
RSP:
RBP:
…

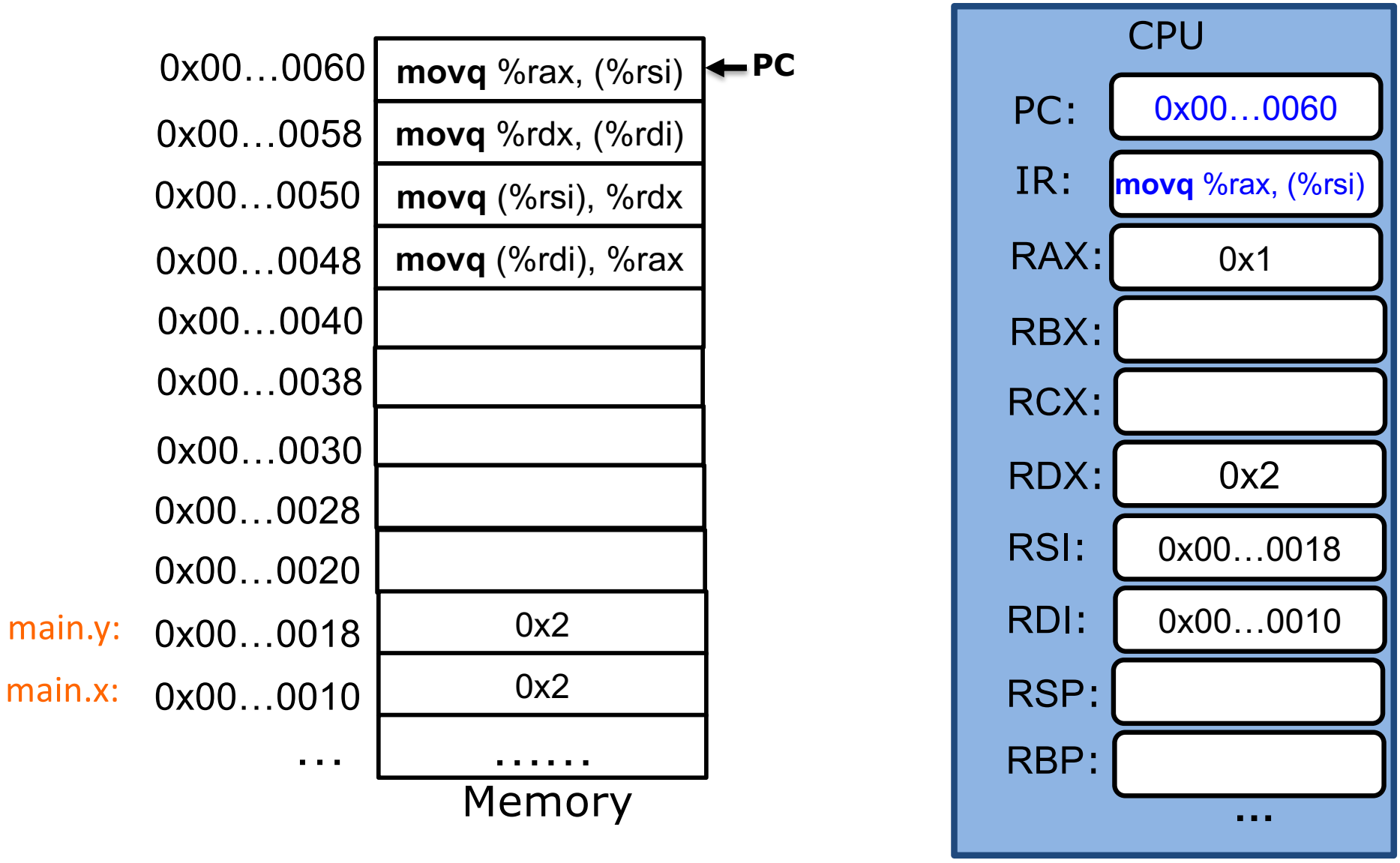
# swap func


0x00…0060
movq %rax, (%rsi)
0x00…0058
movq %rdx, (%rdi)
PC
0x00…0050
movq (%rsi), %rdx
0x00…0048
movq (%rdi), %rax
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
main.y:
0x00…0018
0x2
main.x:
0x00…0010
0x1
…
……
Memory
CPU
PC:
0x00…0058
IR:
movq %rdx, (%rdi)
RAX:
0x1
RBX:
RCX:
RDX:
0x2
RSI:
0x00…0018
RDI:
0x00…0010
RSP:
RBP:
…

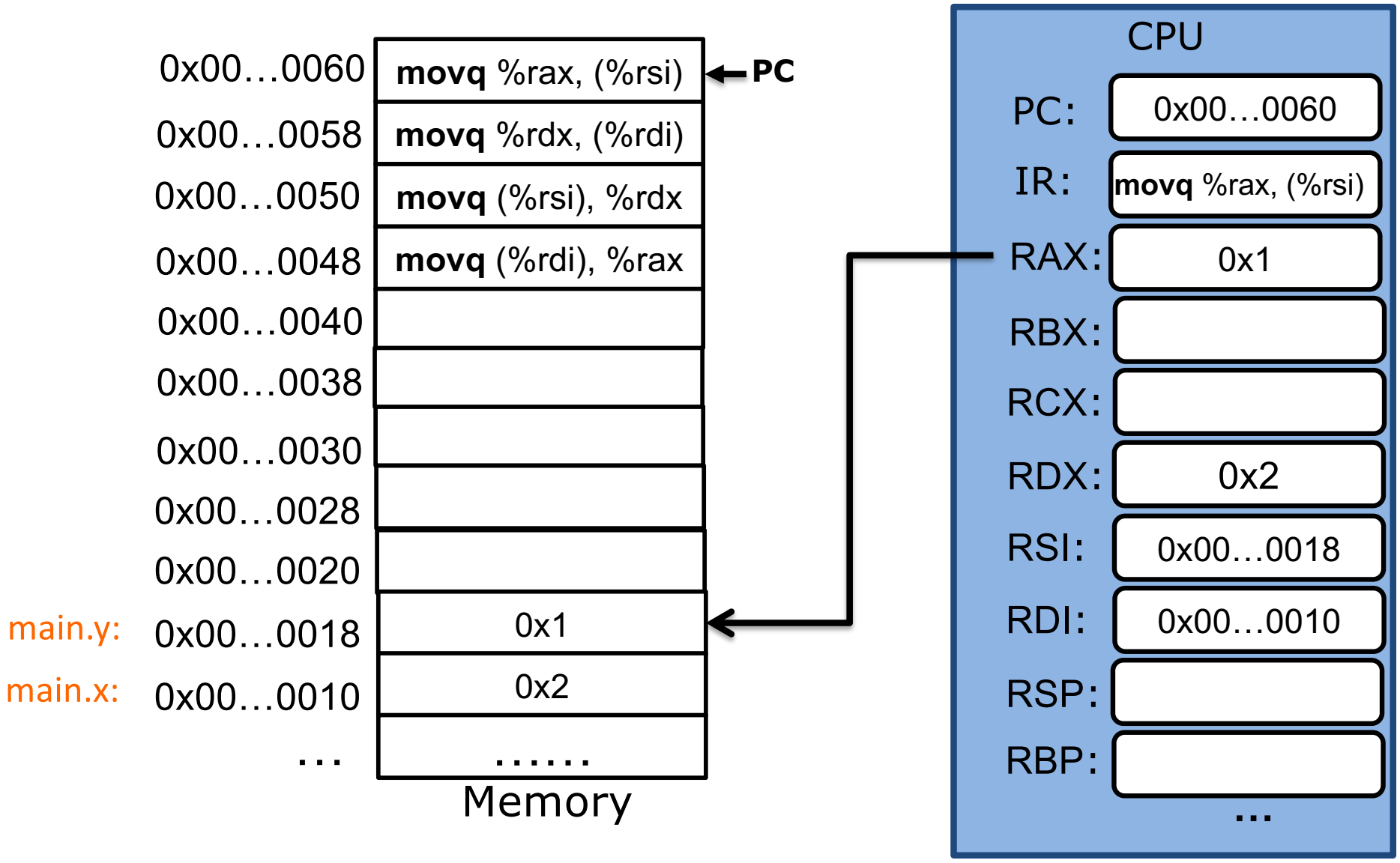
# swap func


0x00…0060
movq %rax, (%rsi)
0x00…0058
movq %rdx, (%rdi)
PC
0x00…0050
movq (%rsi), %rdx
0x00…0048
movq (%rdi), %rax
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
main.y:
0x00…0018
0x2
main.x:
0x00…0010
0x2
…
……
Memory
CPU
PC:
0x00…0058
IR:
movq %rdx, (%rdi)
RAX:
0x1
RBX:
RCX:
RDX:
0x2
RSI:
0x00…0018
RDI:
0x00…0010
RSP:
RBP:
…

# swap func


0x00…0060
movq %rax, (%rsi)
PC
0x00…0058
movq %rdx, (%rdi)
0x00…0050
movq (%rsi), %rdx
0x00…0048
movq (%rdi), %rax
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
main.y:
0x00…0018
0x2
main.x:
0x00…0010
0x2
…
……
Memory
CPU
PC:
0x00…0060
IR:
movq %rax, (%rsi)
RAX:
0x1
RBX:
RCX:
RDX:
0x2
RSI:
0x00…0018
RDI:
0x00…0010
RSP:
RBP:
…

# swap func


0x00…0060
movq %rax, (%rsi)
PC
0x00…0058
movq %rdx, (%rdi)
0x00…0050
movq (%rsi), %rdx
0x00…0048
movq (%rdi), %rax
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
main.y:
0x00…0018
0x1
main.x:
0x00…0010
0x2
…
……
Memory
CPU
PC:
0x00…0060
IR:
movq %rax, (%rsi)
RAX:
0x1
RBX:
RCX:
RDX:
0x2
RSI:
0x00…0018
RDI:
0x00…0010
RSP:
RBP:
…

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

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

# Observation

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

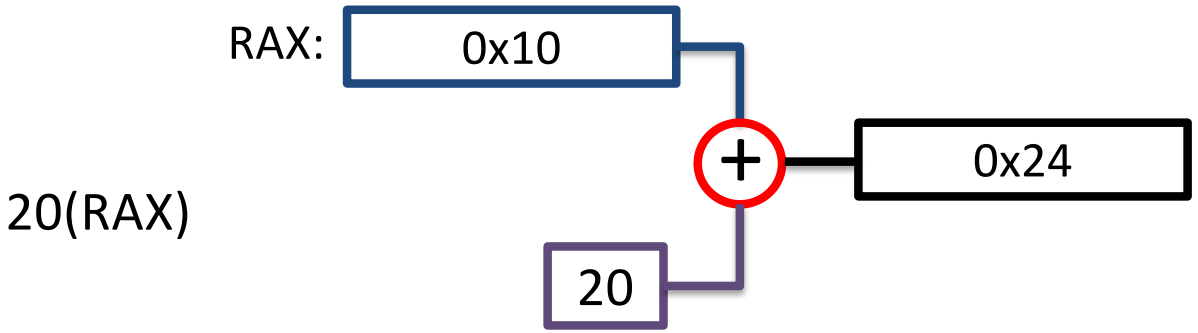
a[0], a[1] and a[2] have the same base address:&a[0]

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

# Address mode with displacement

D(Register): val(Register) + D

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

# Address mode with displacement

D(Register): val(Register) + D

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

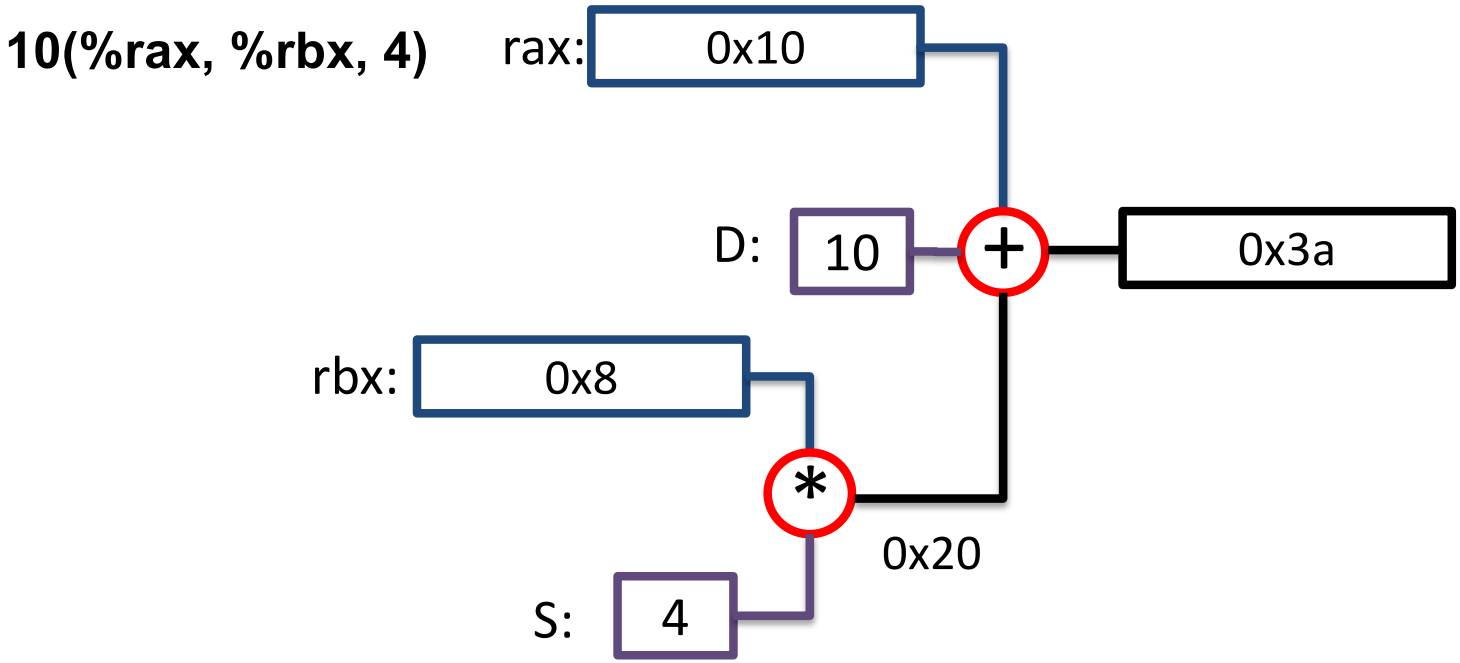
```
long a[] = {3, 2, 1};
for(int i = 0; i < 3; i++) {
    a[i] = i;
}
```

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

# 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

| | |
|---|---|
| `%rdx` | `0xf000` |
| `%rcx` | `0x100` |

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

# Address Computation Examples

| | |
|---|---|
| `%rdx` | `0xf000` |
| `%rcx` | `0x100` |

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

# Address Computation Examples

| | |
|---|---|
| `%rdx` | `0xf000` |
| `%rcx` | `0x100` |

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

# Address Computation Examples

| | |
|---|---|
| `%rdx` | `0xf000` |
| `%rcx` | `0x100` |

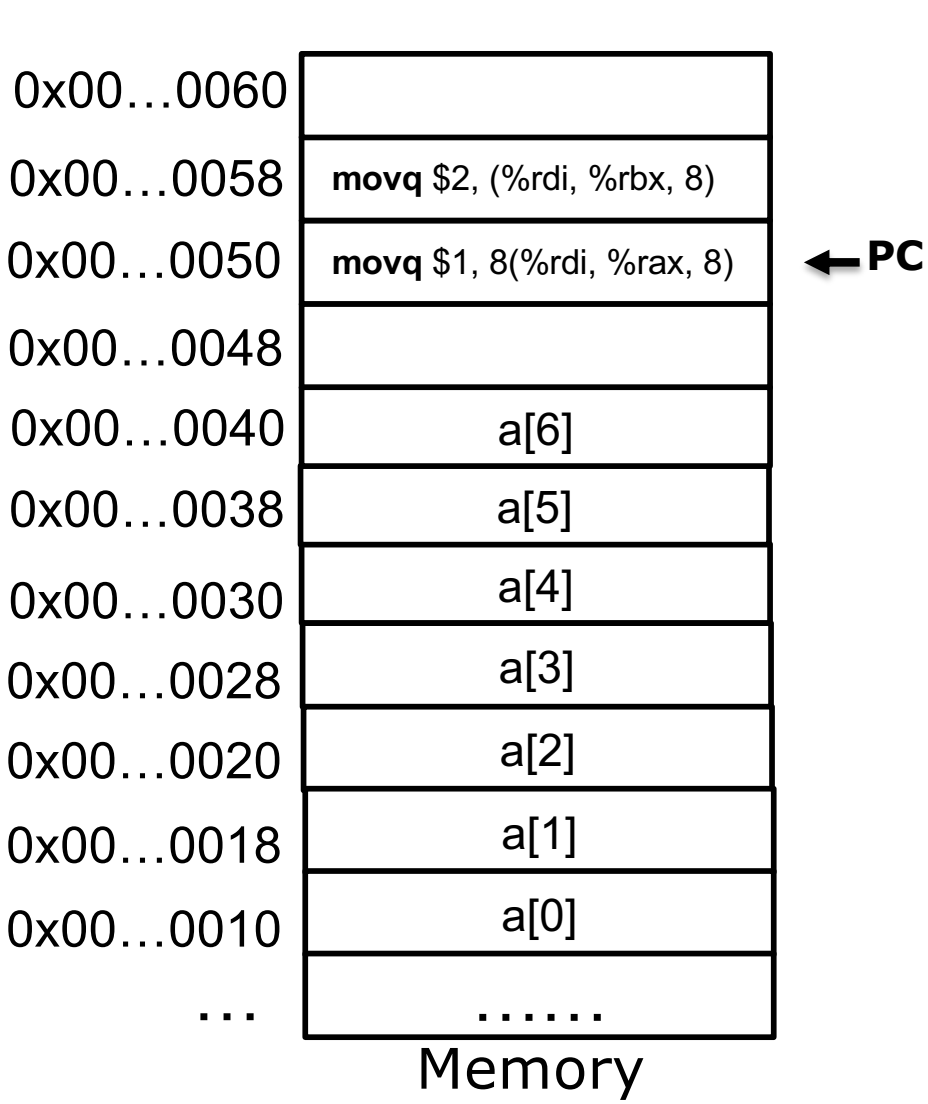
| Expression | Address Computation | Address |
|---|---|---|
| `0x8(%rdx)` | `0xf000 + 0x8` | `0xf008` |
| `(%rdx,%rcx)` | `0xf000 + 0x100` | `0xf100` |
| `(%rdx,%rcx,4)` | `0xf000 + 4*0x100` | `0xf400` |
| `0x80(,%rdx,2)` | | |

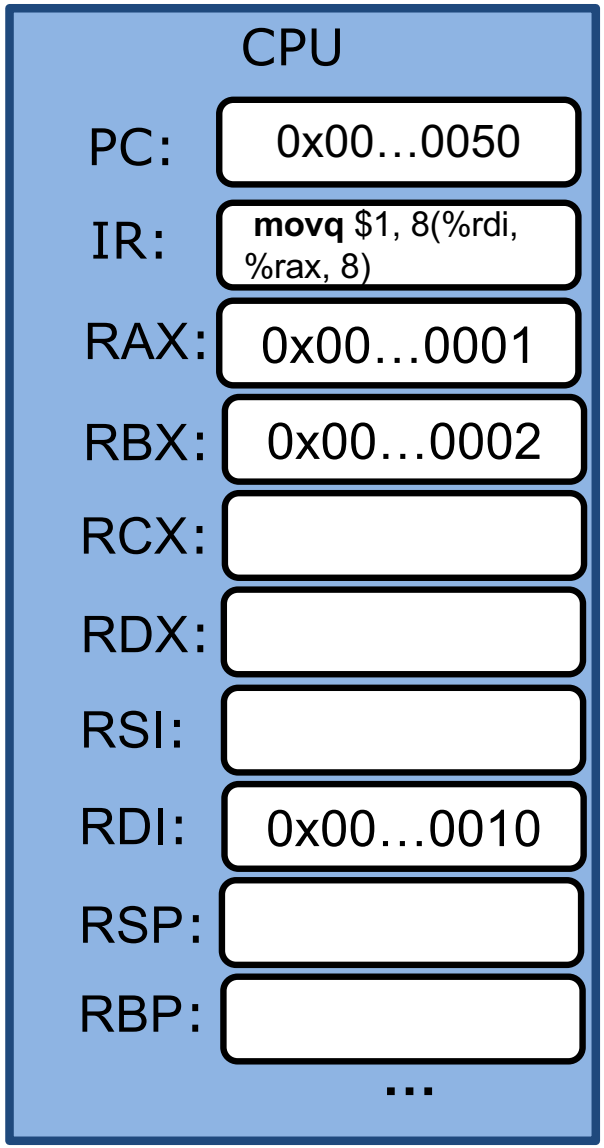
# Address Computation Examples

| | |
|---|---|
| `%rdx` | `0xf000` |
| `%rcx` | `0x100` |

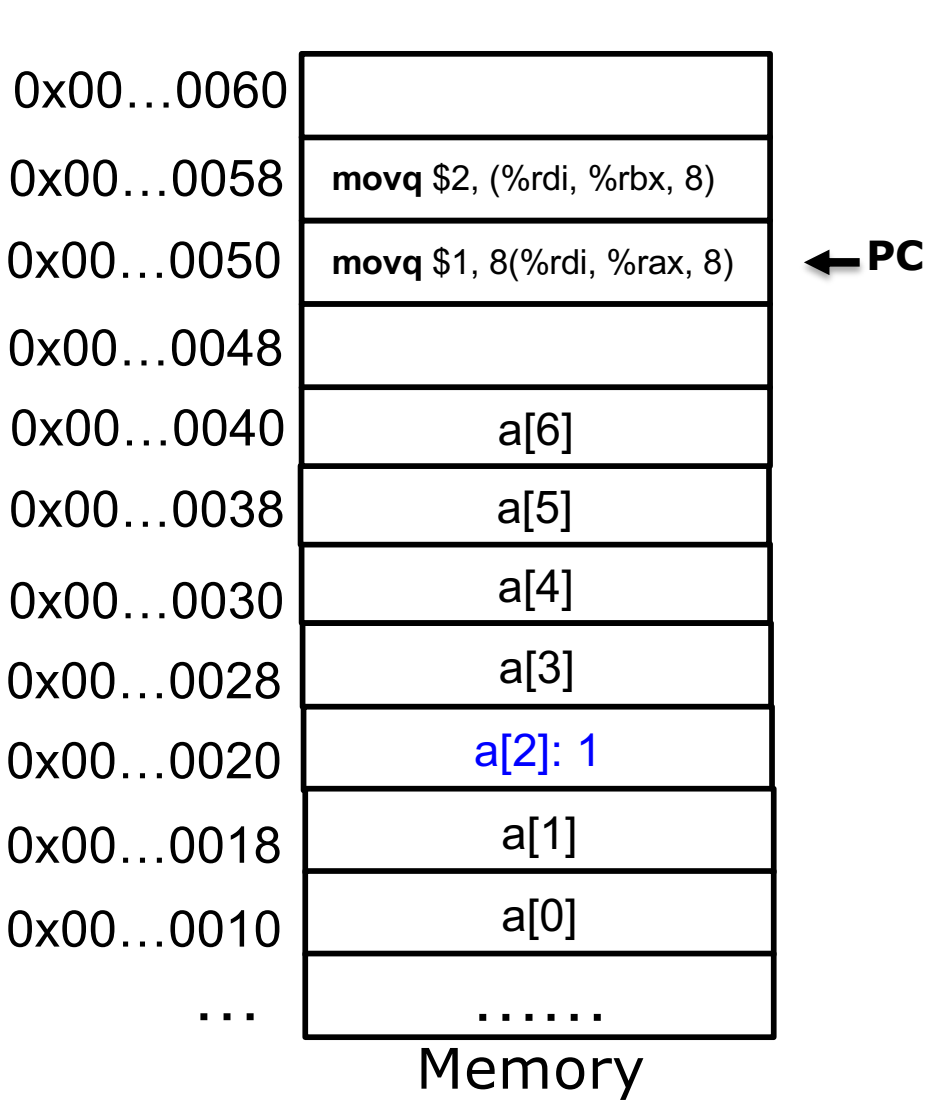
| Expression | Address Computation | Address |
|---|---|---|
| `0x8(%rdx)` | `0xf000 + 0x8` | `0xf008` |
| `(%rdx,%rcx)` | `0xf000 + 0x100` | `0xf100` |
| `(%rdx,%rcx,4)` | `0xf000 + 4*0x100` | `0xf400` |
| `0x80(,%rdx,2)` | `2*0xf000 + 0x80` | `0x1e080` |

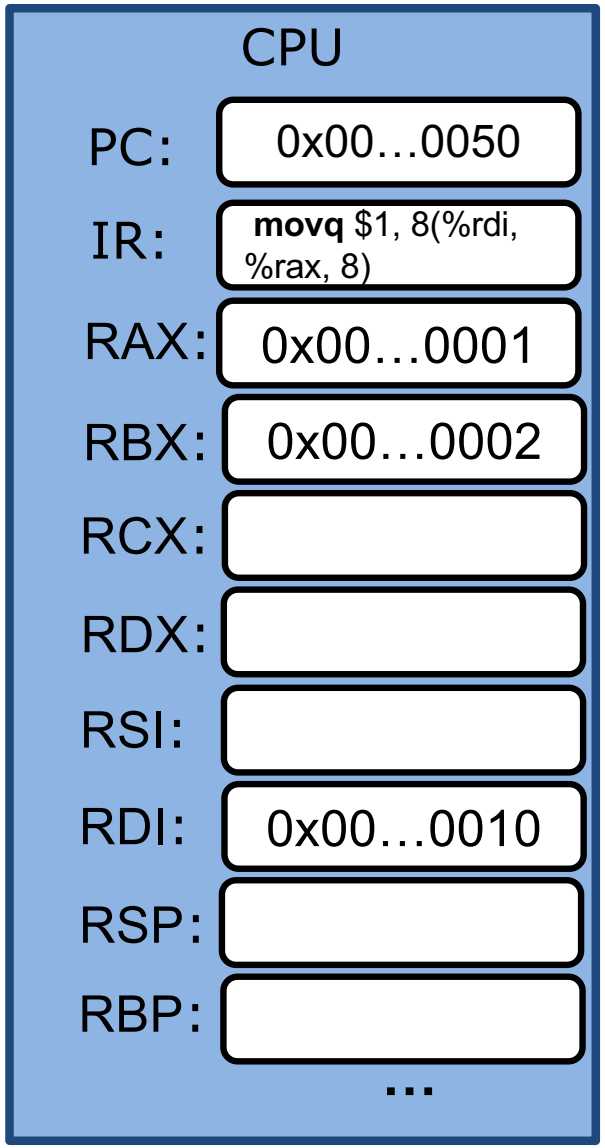
# Example


0x00…0060
0x00…0058
movq $2, (%rdi, %rbx, 8)
0x00…0050
movq $1, 8(%rdi, %rax, 8)
PC
0x00…0048
0x00…0040
a[6]
0x00…0038
a[5]
0x00…0030
a[4]
0x00…0028
a[3]
0x00…0020
a[2]
0x00…0018
a[1]
0x00…0010
a[0]
…
……
Memory

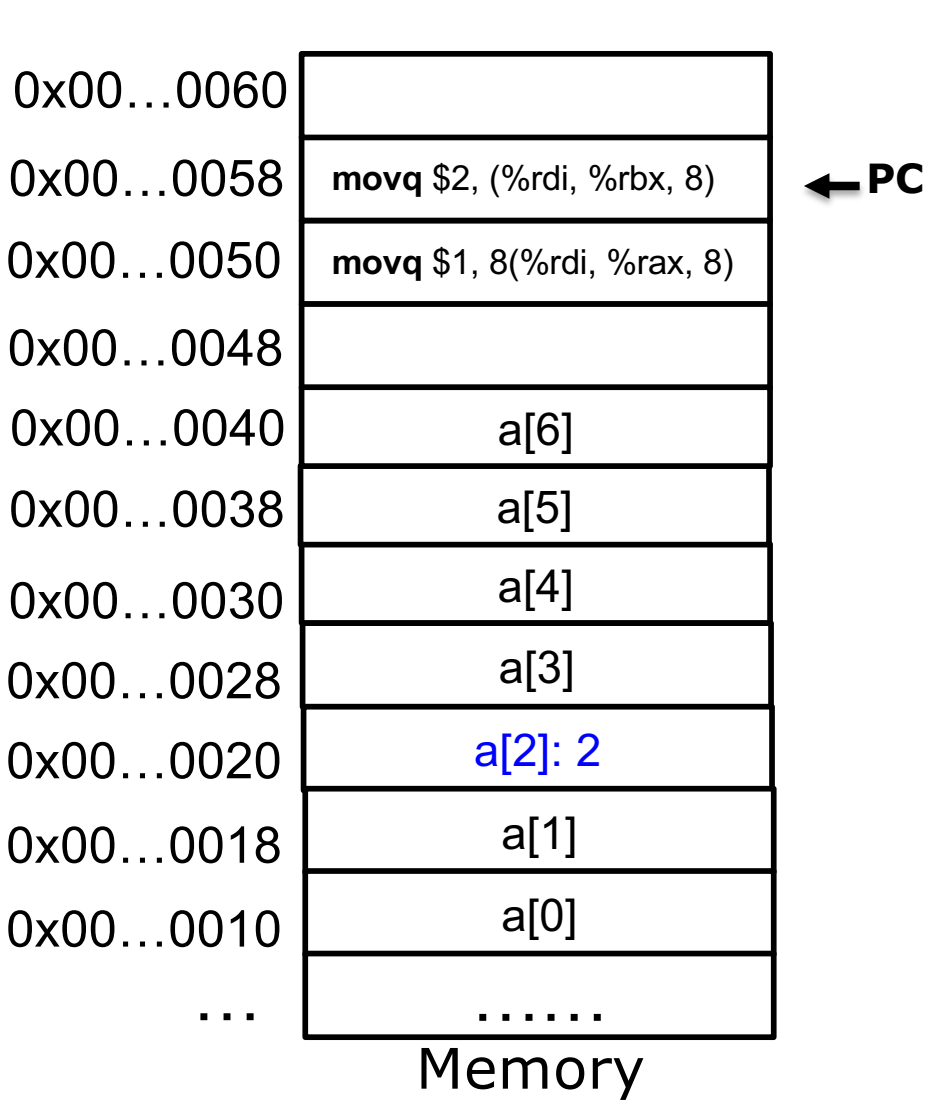

CPU
PC:
0x00…0050
IR:
movq $1, 8(%rdi, %rax, 8)
RAX:
0x00…0001
RBX:
0x00…0002
RCX:
RDX:
RSI:
RDI:
0x00…0010
RSP:
RBP:
…

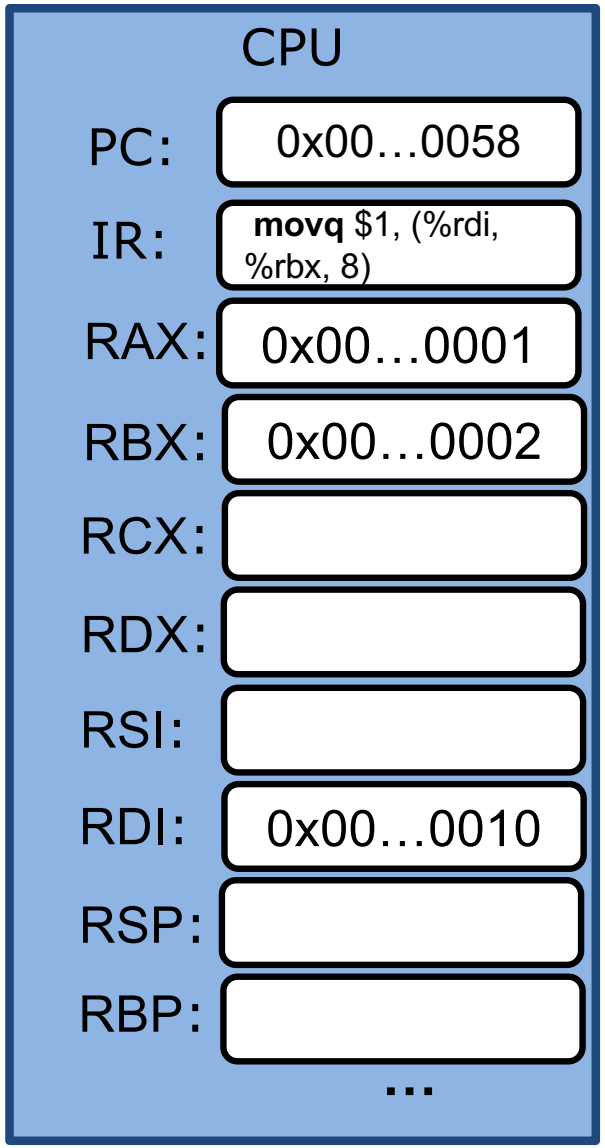
# Example


0x00…0060
0x00…0058
movq $2, (%rdi, %rbx, 8)
0x00…0050
movq $1, 8(%rdi, %rax, 8)
PC
0x00…0048
0x00…0040
a[6]
0x00…0038
a[5]
0x00…0030
a[4]
0x00…0028
a[3]
0x00…0020
a[2]: 1
0x00…0018
a[1]
0x00…0010
a[0]
…
……
Memory


CPU
PC:
0x00…0050
IR:
movq $1, 8(%rdi, %rax, 8)
RAX:
0x00…0001
RBX:
0x00…0002
RCX:
RDX:
RSI:
RDI:
0x00…0010
RSP:
RBP:
…

# Example


0x00…0060
0x00…0058
movq $2, (%rdi, %rbx, 8)
PC
0x00…0050
movq $1, 8(%rdi, %rax, 8)
0x00…0048
0x00…0040
a[6]
0x00…0038
a[5]
0x00…0030
a[4]
0x00…0028
a[3]
0x00…0020
a[2]: 2
0x00…0018
a[1]
0x00…0010
a[0]
…
……
Memory


CPU
PC:
0x00…0058
IR:
movq $1, (%rdi, %rbx, 8)
RAX:
0x00…0001
RBX:
0x00…0002
RCX:
RDX:
RSI:
RDI:
0x00…0010
RSP:
RBP:
…

# mov{bwlq}

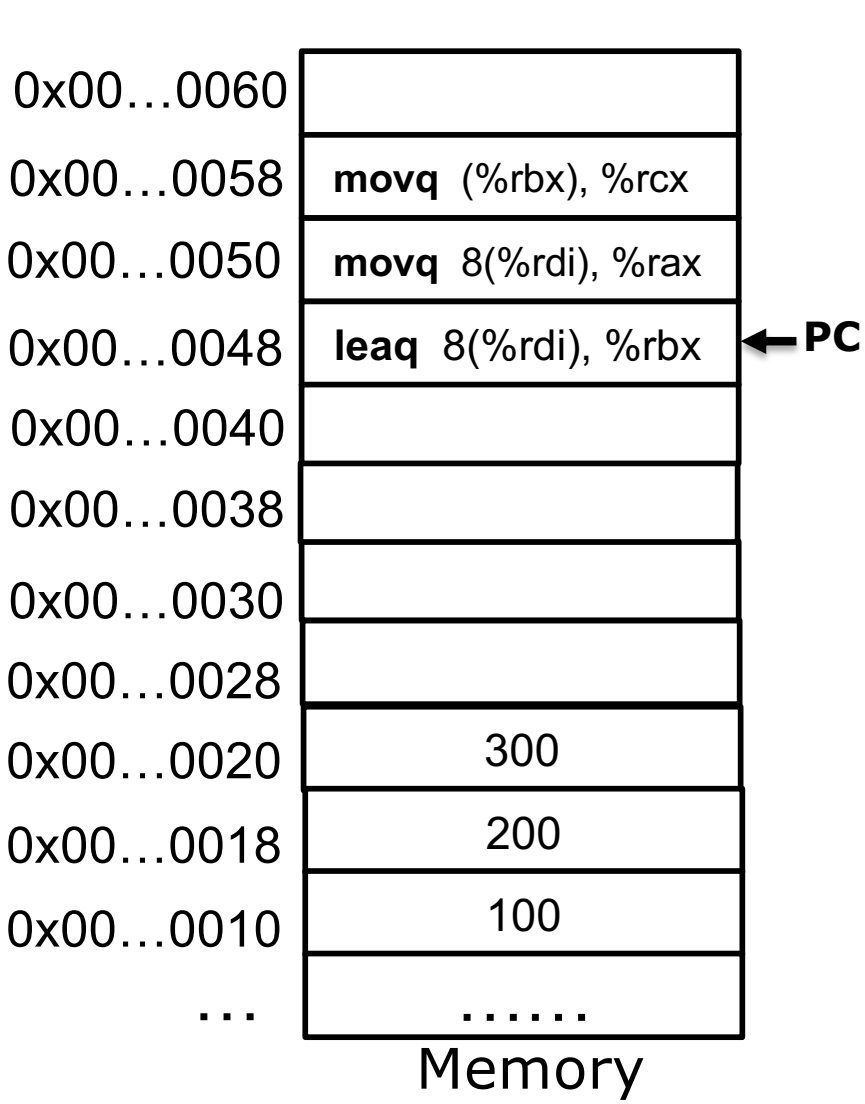
| | |
|---|---|
| **movb** src, dest | Copy a **byte** from the source operand to the destination operand. e.g., **movb** %al, %bl |
| **movw** src, dest | Copy a **word** from the source operand to the destination operand. e.g., **movw** %ax, %bx |
| **movl** src, dest | Copy a **long** (32 bits) from 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 |

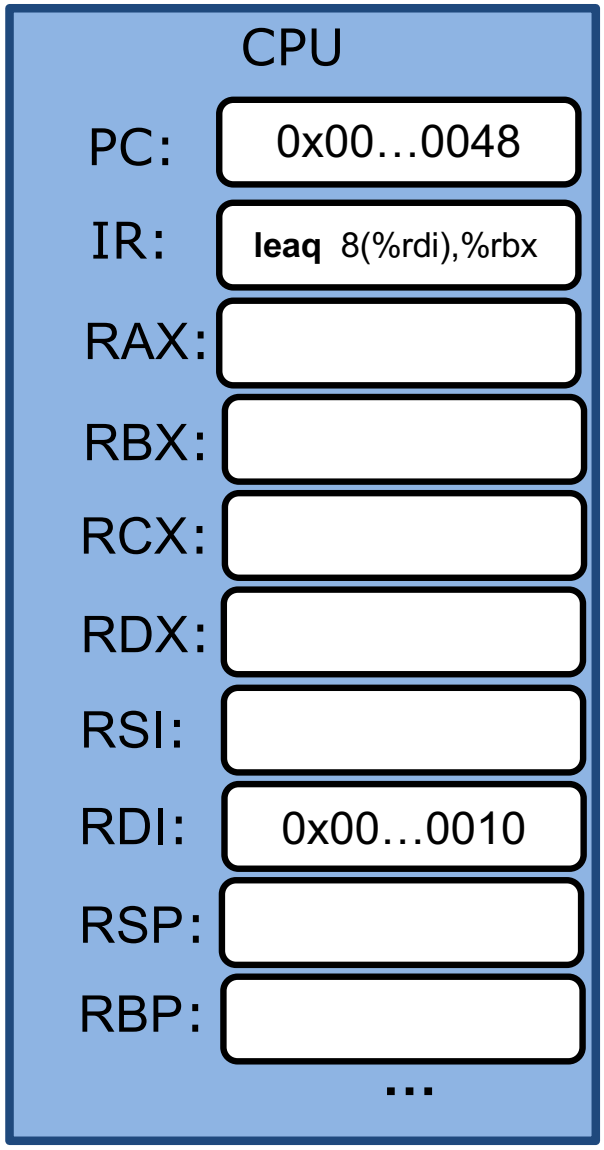
# The lea instruction

**leaq** *Source, Dest*

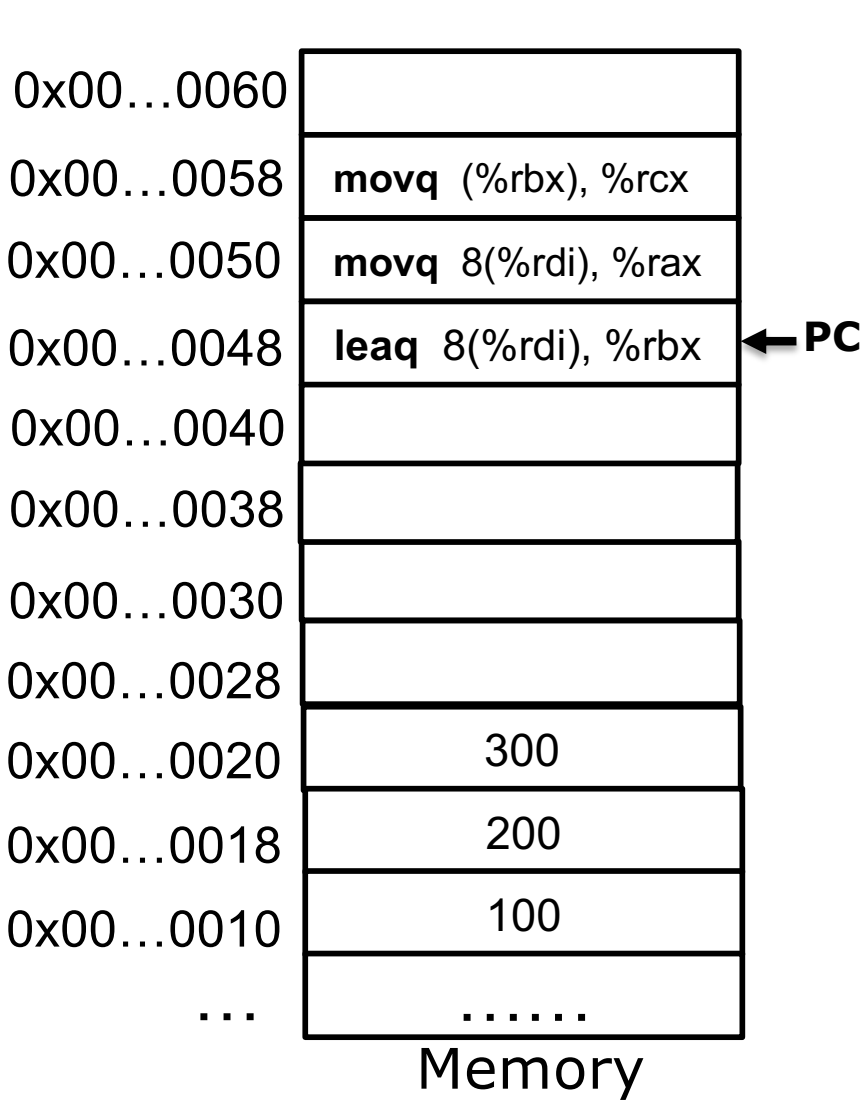
- load effective address: set *Dest* to the address denoted by *Source* address mode expression

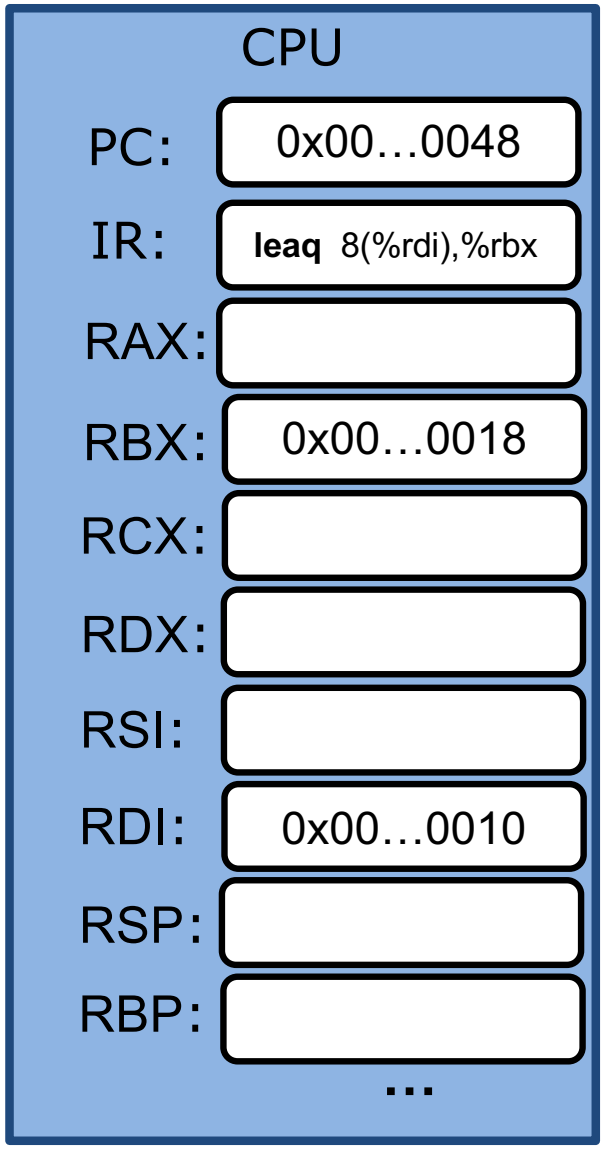
# Example


0x00…0060
0x00…0058
movq (%rbx), %rcx
0x00…0050
movq 8(%rdi), %rax
0x00…0048
leaq 8(%rdi), %rbx
PC
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
300
0x00…0018
200
0x00…0010
100
…
……
Memory


CPU
PC:
0x00…0048
IR:
leaq 8(%rdi),%rbx
RAX:
RBX:
RCX:
RDX:
RSI:
RDI:
0x00…0010
RSP:
RBP:
…

# Example

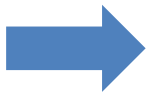

0x00…0060
0x00…0058
movq (%rbx), %rcx
0x00…0050
movq 8(%rdi), %rax
0x00…0048
leaq 8(%rdi), %rbx
PC
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
300
0x00…0018
200
0x00…0010
100
…
……
Memory


CPU
PC:
0x00…0048
IR:
leaq 8(%rdi),%rbx
RAX:
RBX:
0x00…0018
RCX:
RDX:
RSI:
RDI:
0x00…0010
RSP:
RBP:
…

# A common usage of leaq

Compute expressions: x + K*y + d (K=1, 2, 4, or 8)

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



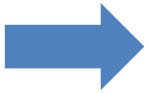
```
leaq (%rdi, %rdi,2), %rax
```

**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
```

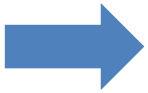


```
long f(long x, long y)
{
    long s = ??;
    return s;
}
```

# Arithmetic Expression Puzzle

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

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



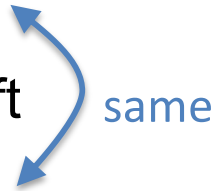
```
long f(long x, long y)
{
    long s = 5(x + 2y);
    return s;
}
```

# Basic Arithmetic Operations

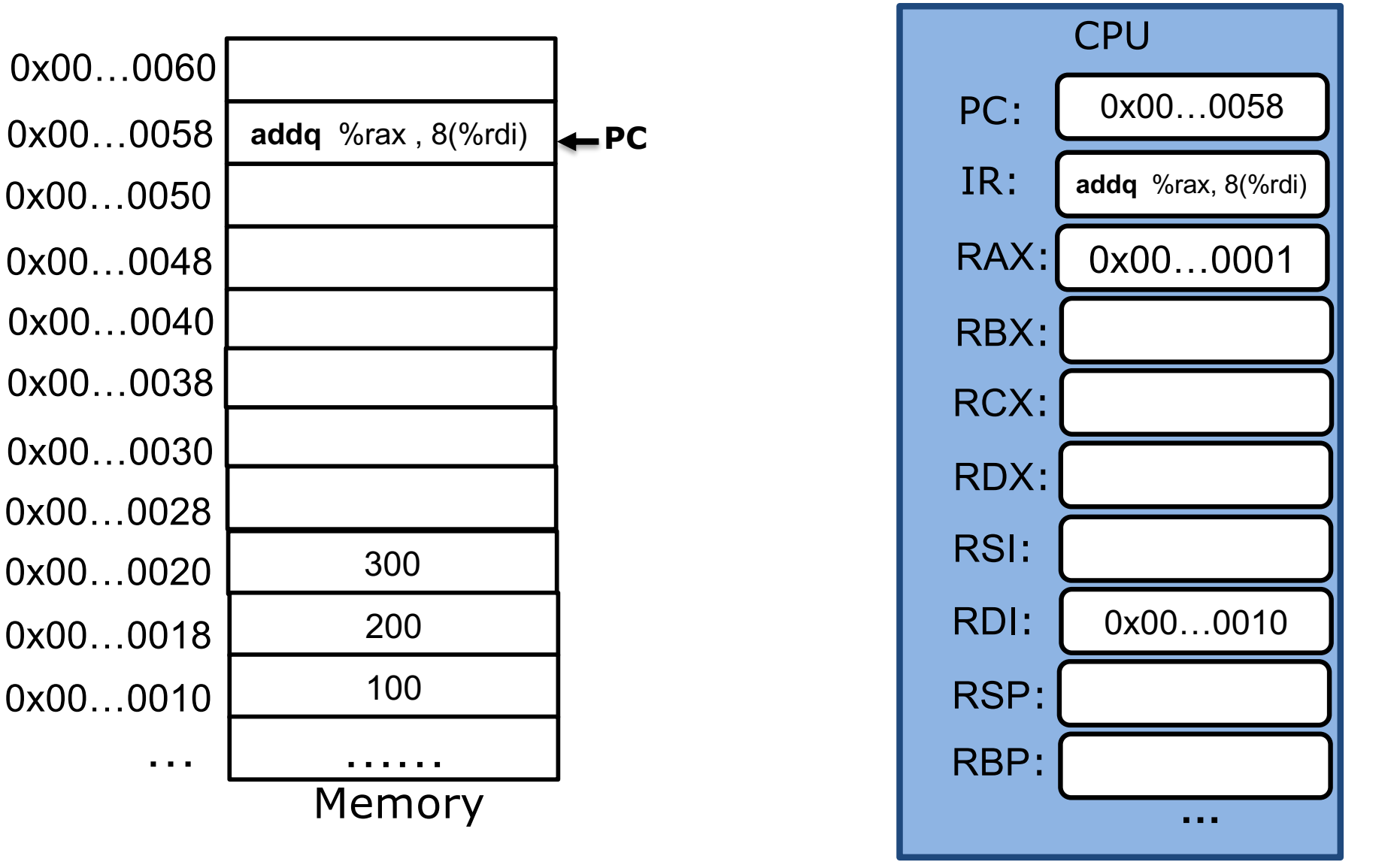
| | | |
|---|---|---|
| **addq** | Src, Dest | Dest = Dest + Src |
| **subq** | Src, Dest | Dest = Dest – Src |
| **imulq** | Src, Dest | Dest = Dest * Src |
| | | |
| **incq** | Dest | Dest = Dest + 1 |
| **decq** | Dest | Dest = Dest – 1 |
| **negq** | Dest | Dest = – Dest |

# Bitwise Operations

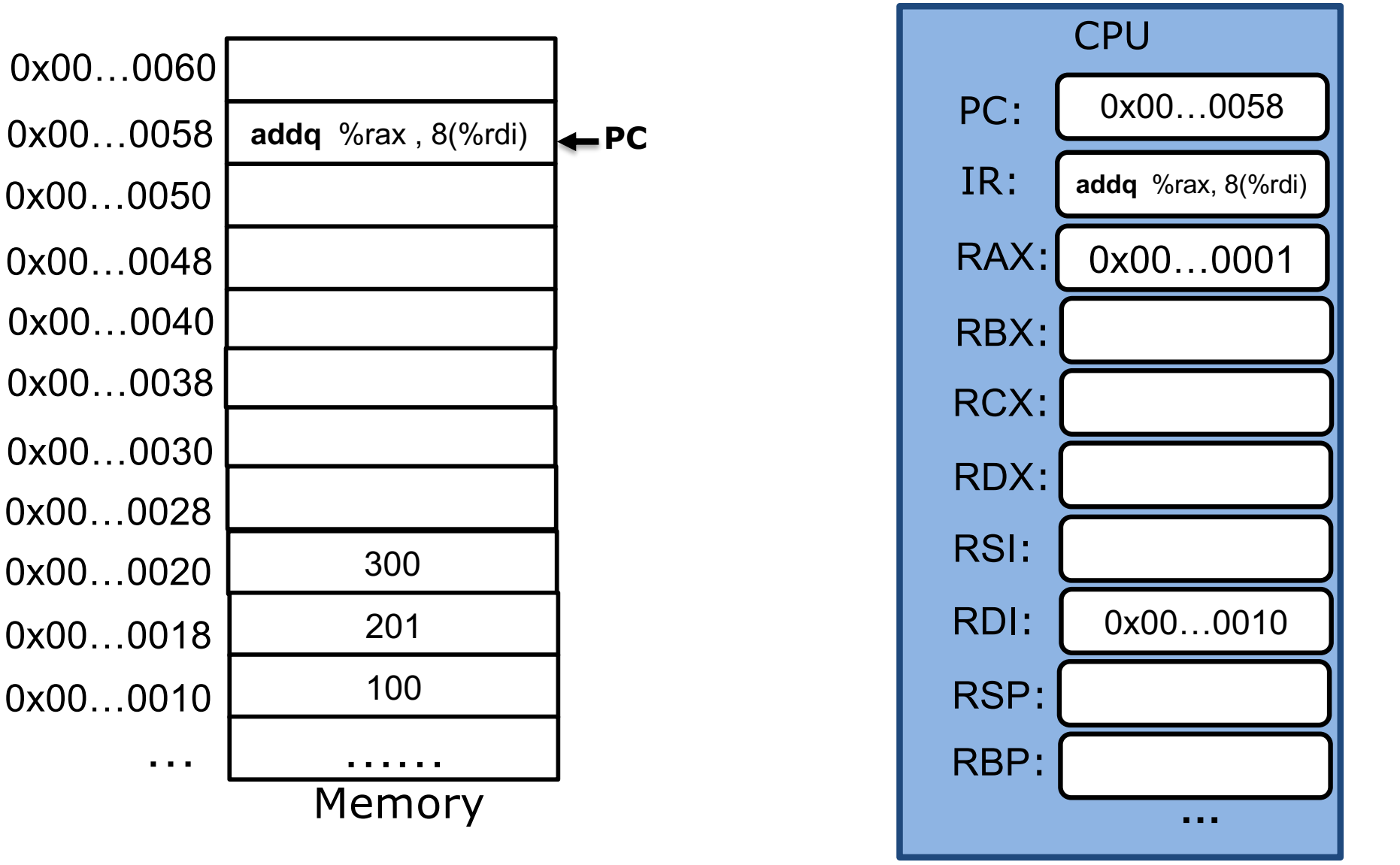
| | | | |
|---|---|---|---|
| **salq** | Src,Dest | Dest = Dest << Src | Arithmetic left shift |
| **sarq** | Src,Dest | Dest = Dest >> Src | Arithmetic right shift |
| **shlq** | Src,Dest | Dest = Dest << Src | Logical left shift |
| **shrq** | Src,Dest | Dest = Dest >> Src | Logical right shift |
|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



# Example


0x00…0060
0x00…0058
addq %rax , 8(%rdi)
PC
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
300
0x00…0018
200
0x00…0010
100
…
……
Memory
CPU
PC:
0x00…0058
IR:
addq %rax, 8(%rdi)
RAX:
0x00…0001
RBX:
RCX:
RDX:
RSI:
RDI:
0x00…0010
RSP:
RBP:
…

# Example


0x00…0060
0x00…0058
addq %rax , 8(%rdi)
PC
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
300
0x00…0018
201
0x00…0010
100
…
……
Memory
CPU
PC:
0x00…0058
IR:
addq %rax, 8(%rdi)
RAX:
0x00…0001
RBX:
RCX:
RDX:
RSI:
RDI:
0x00…0010
RSP:
RBP:
…