Machine Program: Basics

Jinyang Li

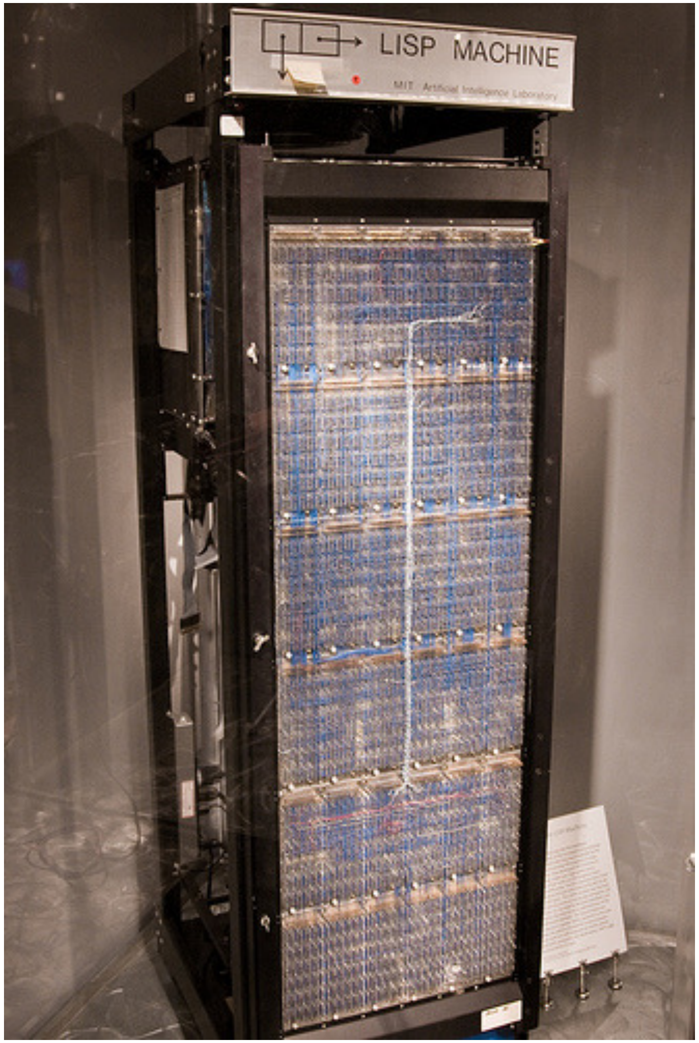
Some are based on Tiger Wang's slides

Lesson plan

- What we’ve learnt so far:
  - How integers/reals/characters are represented by computers
  - C programming

- Today:
  - Basic hardware execution of a program
  - x86 registers
  - x86 move instruction

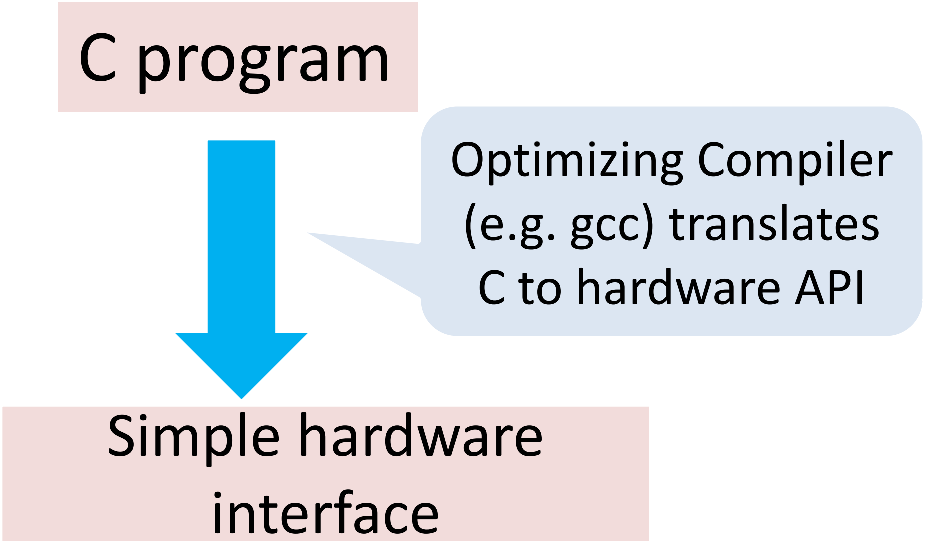
Can we build a machine to execute C directly?



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

Why not directly execute C?

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

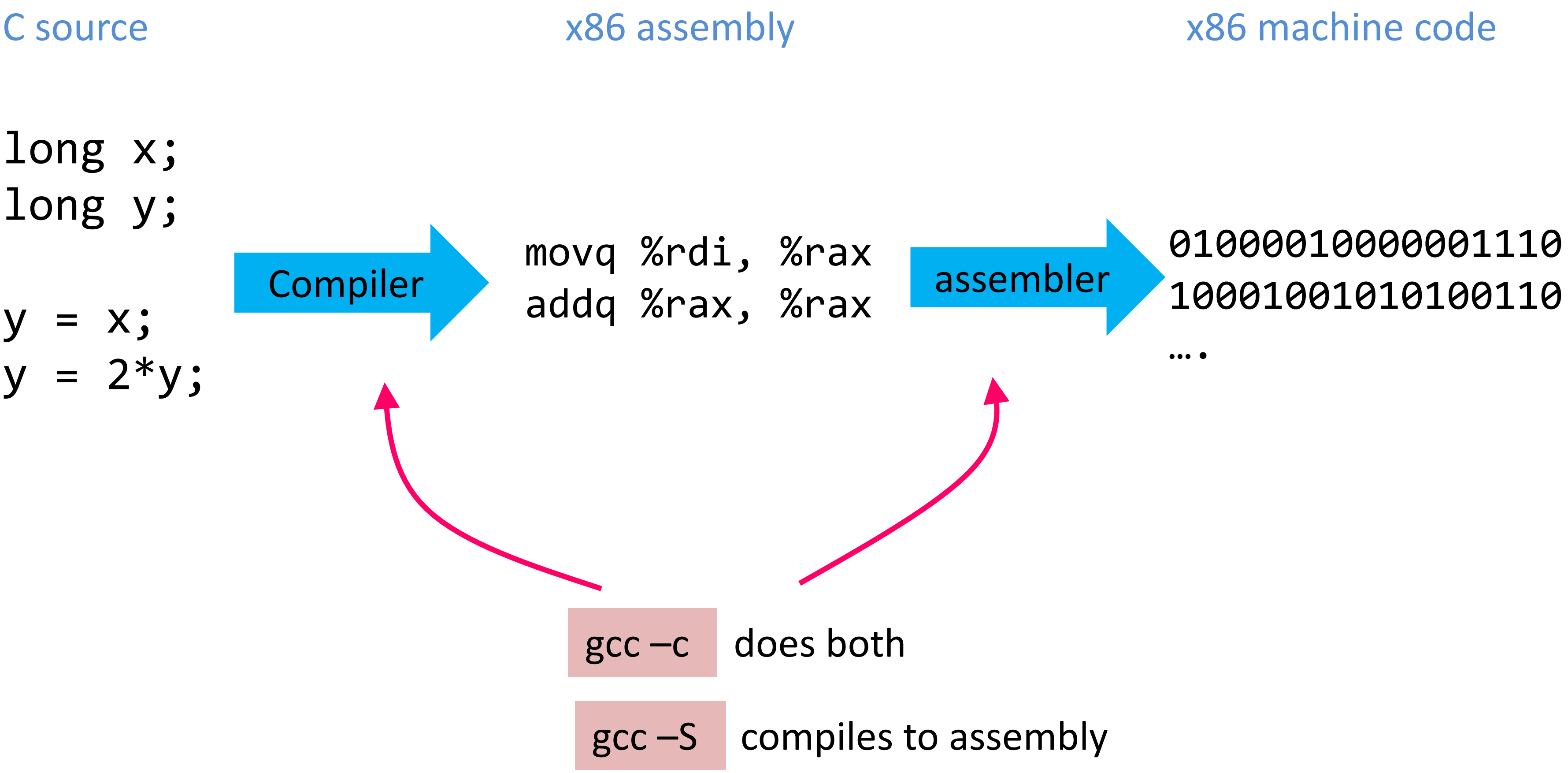


C program

Optimizing Compiler (e.g. gcc) translates C to hardware API

Simple hardware interface

C vs. assembly vs. machine code



C source

```
long x;
long y;

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

x86 assembly

```
movq %rdi, %rax
addq %rax, %rax
```

x86 machine code

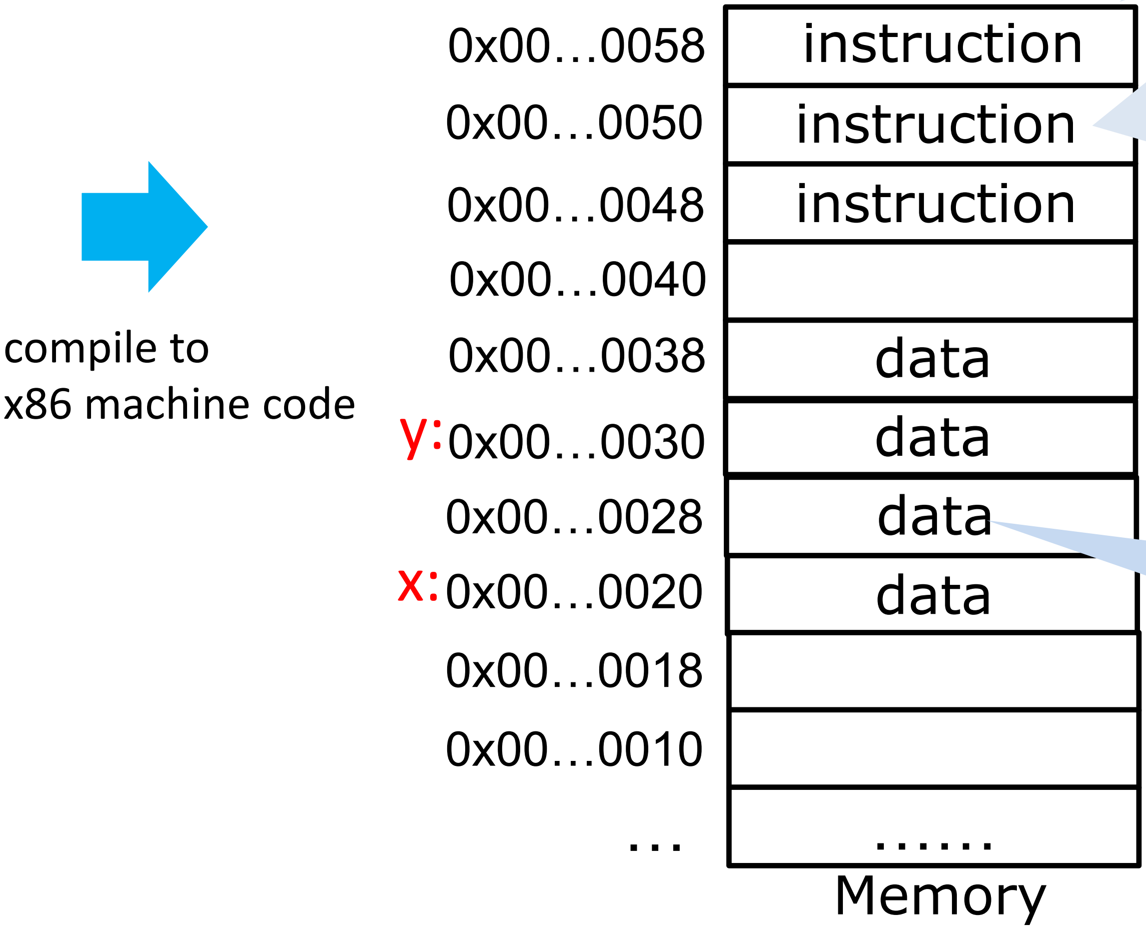
```
01000010000001110
10001001010100110
….
```

gcc –c does both

gcc –S compiles to assembly

C vs. machine code

```
long x;
long y;

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



compile to
x86 machine code

	0x00…0058	instruction
	0x00…0050	instruction
	0x00…0048	instruction
	0x00…0040	
	0x00…0038	data
y:	0x00…0030	data
	0x00…0028	data
x:	0x00…0020	data
	0x00…0018	
	0x00…0010	
	…	……

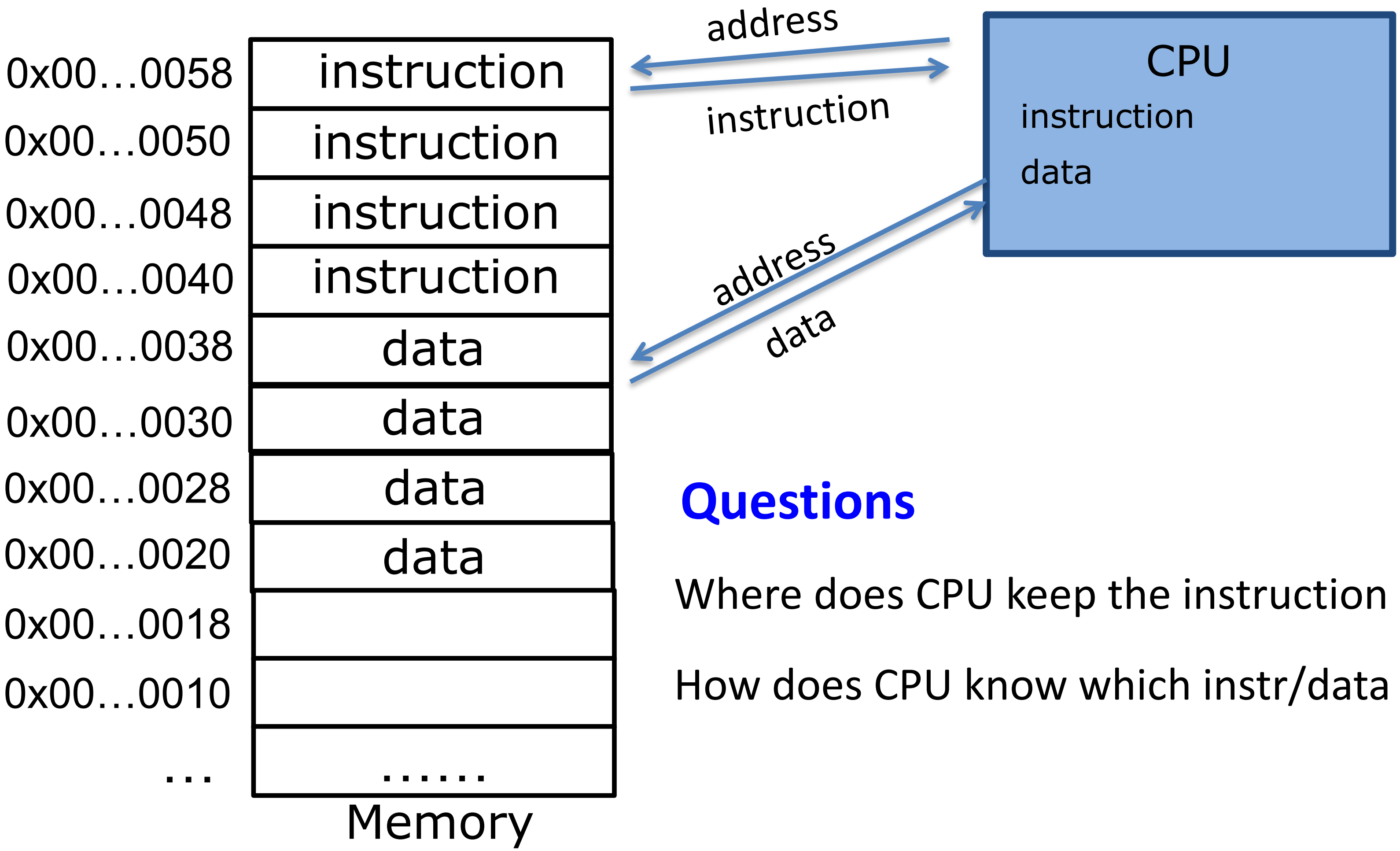
Memory

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



0x00…0058	instruction
0x00…0050	instruction
0x00…0048	instruction
0x00…0040	instruction
0x00…0038	data
0x00…0030	data
0x00…0028	data
0x00…0020	data
0x00…0018	
0x00…0010	
…	……

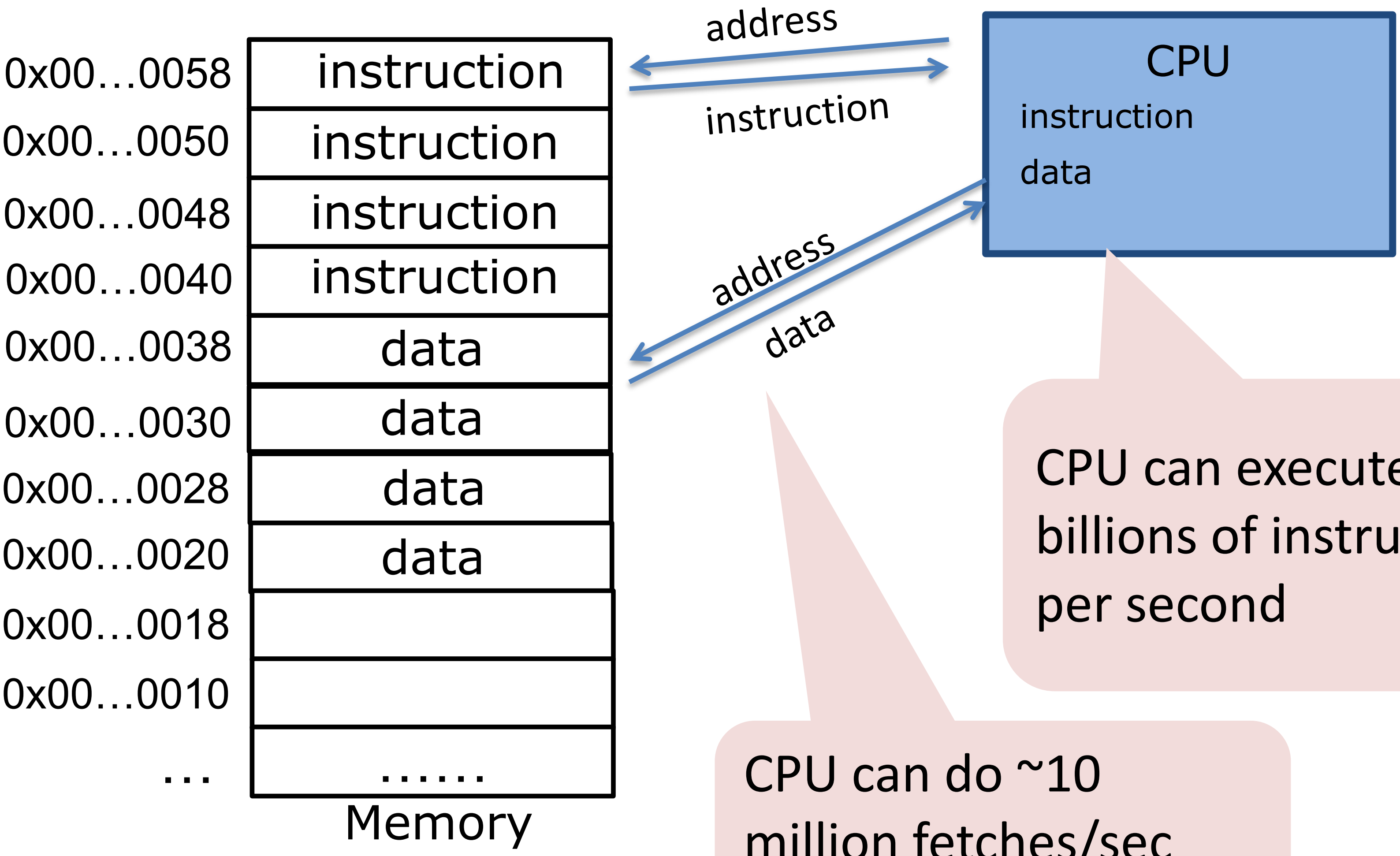
Memory

Questions

Where does CPU keep the instruction and data?

How does CPU know which instr/data to fetch?

How CPU executes a program



0x00…0058	instruction
0x00…0050	instruction
0x00…0048	instruction
0x00…0040	instruction
0x00…0038	data
0x00…0030	data
0x00…0028	data
0x00…0020	data
0x00…0018	
0x00…0010	
…	……

CPU can execute billions of instructions per second

CPU can do ~10 million fetches/sec from memory

Register – temporary storage area built into a CPU

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

- Store memory address of next instruction

IR: CPU’s internal buffer storing the fetched instruction

General purpose registers:

- Store data and address used by programs

Program status and control register:

- Status of the instruction executed

Steps of execution in CPU

1. PC contains the instruction’s address

2. Fetch the instruction to internal buffer

3. Execute the instruction which does one of following:
   - Memory operations: move data from memory to register (or opposite)
   - Arithmetic operations: add, shift etc.
   - Control flow operations.

4. PC is updated to contain the next instruction’s address.

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

Lectures on assembly

- ARM is another common ISA
  - Phones, tablets, Raspberry Pi, Apple's new M1 laptop

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

Lectures on hardware

Question:
Can you run on snappy1 the executable (a.out) compiled on your apple M1 laptop?

X86-64 ISA: registers

Program counter:

- called %rip in x86_64

IR: CPU’s internal buffer storing the fetched instruction

Visible to programmers
(aka part of ISA)

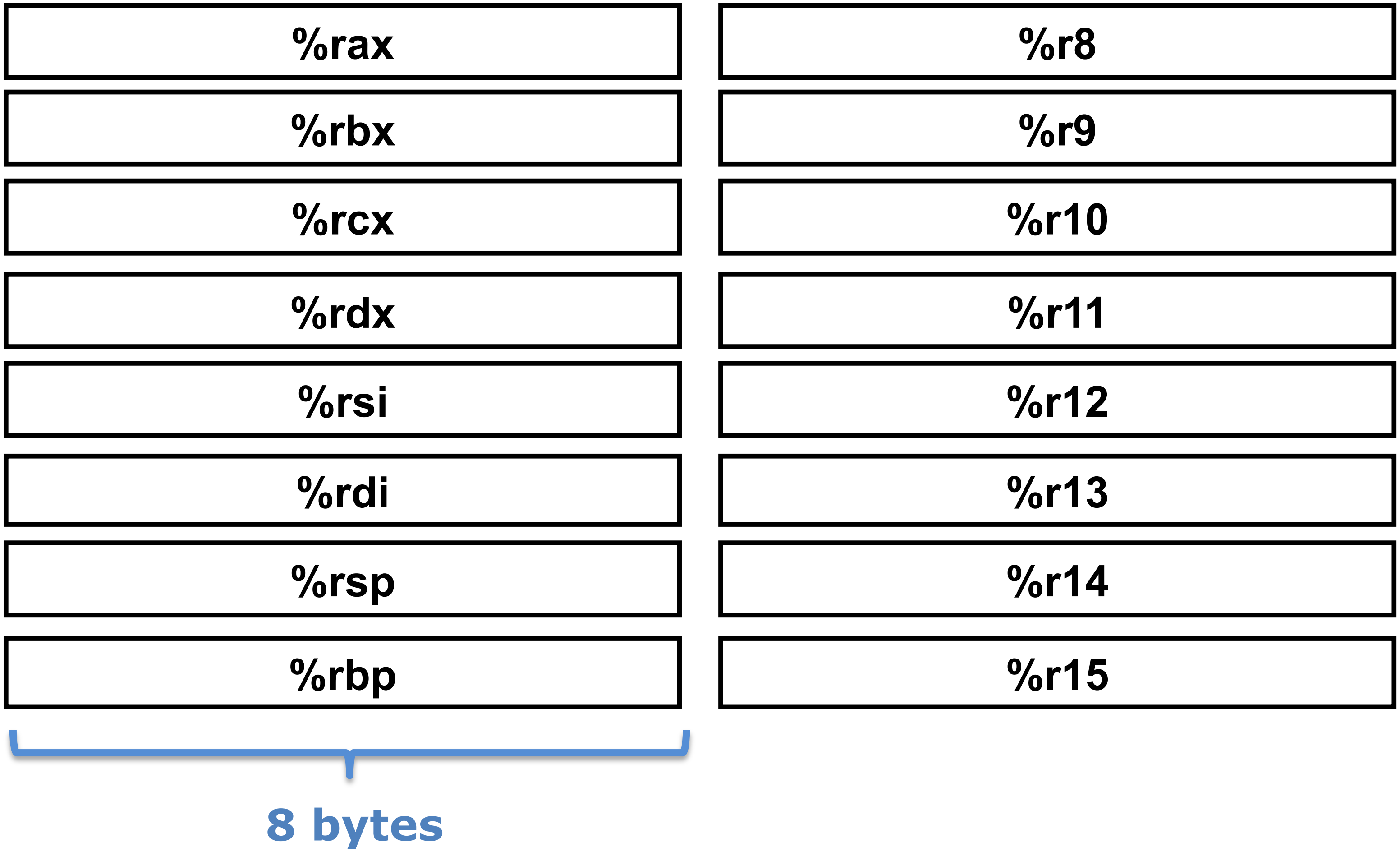
General purpose registers:

- 16 8-byte registers: %rax, %rbx …

Program status and control register:

- Called “RFLAGS” in x86_64

X86-64 general purpose registers: 8-byte

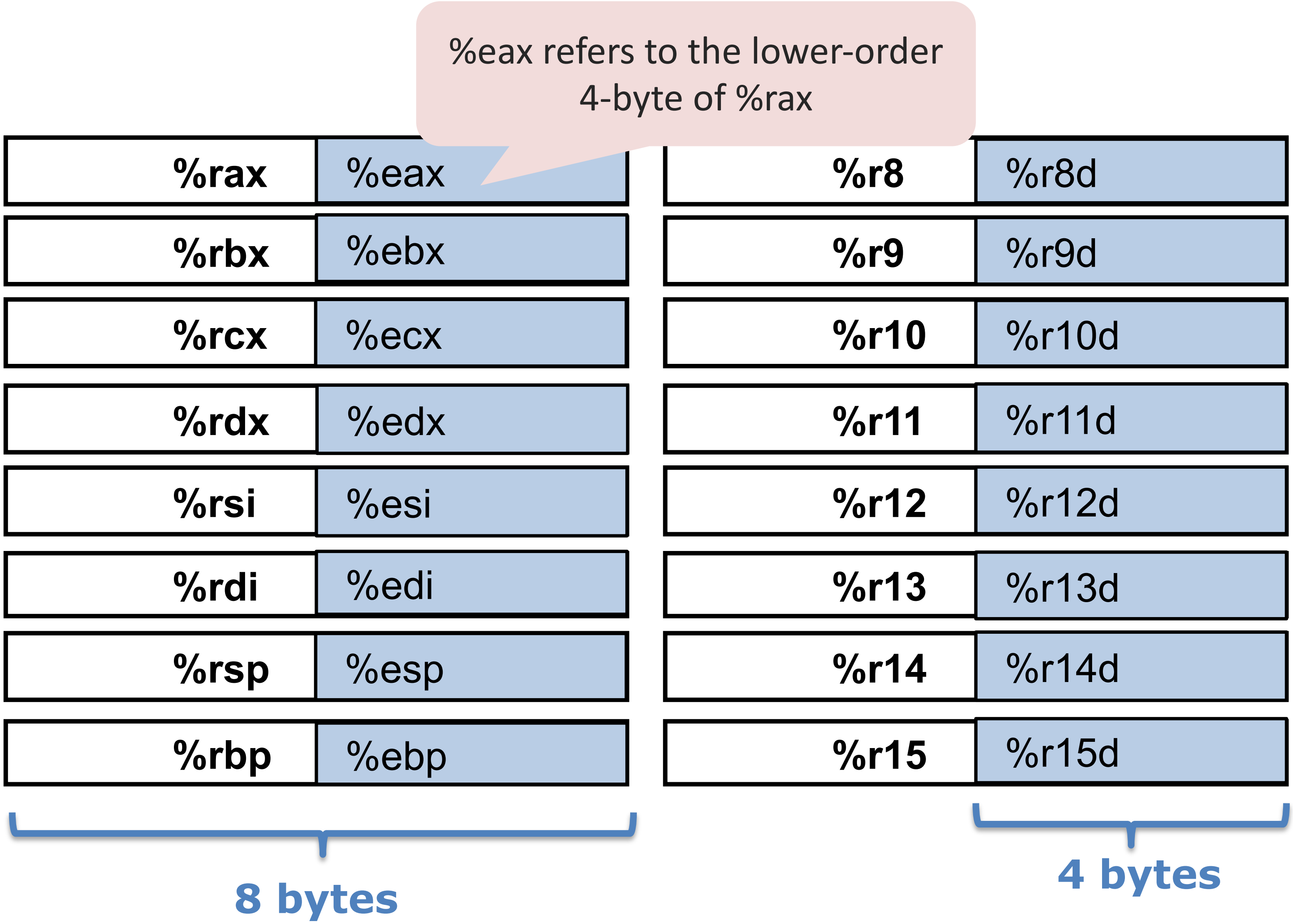


%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

8 bytes

X86-64 general purpose registers: 4-byte

4-byte registers refer to the lower-order 4-bytes of original registers.



%eax refers to the lower-order 4-byte of %rax

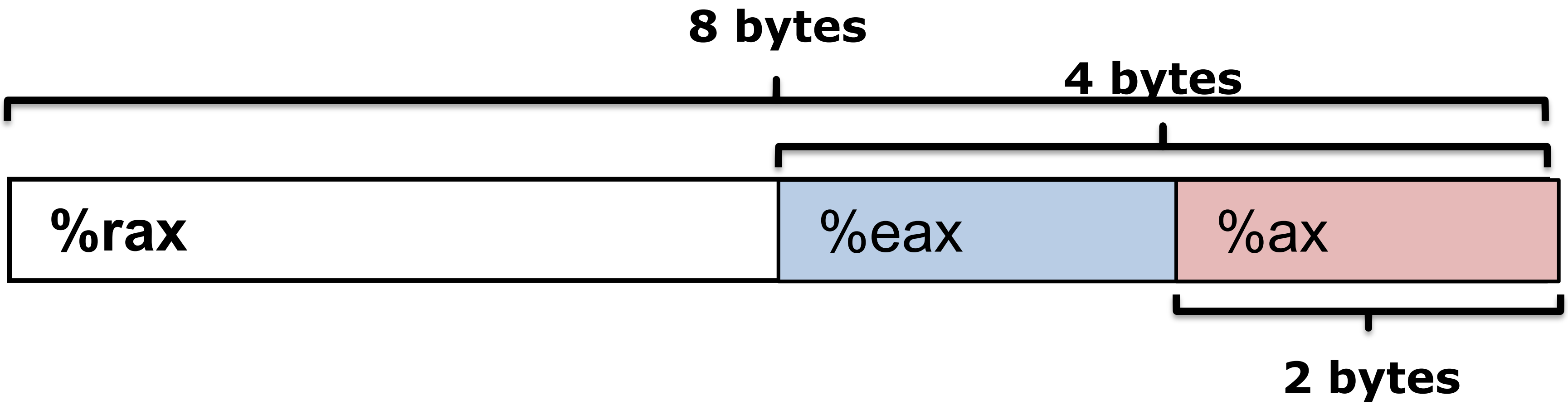
%rax	%eax	**%r8**	%r8d
%rbx	%ebx	**%r9**	%r9d
%rcx	%ecx	**%r10**	%r10d
%rdx	%edx	**%r11**	%r11d
%rsi	%esi	**%r12**	%r12d
%rdi	%edi	**%r13**	%r13d
%rsp	%esp	**%r14**	%r14d
%rbp	%ebp	**%r15**	%r15d

8 bytes

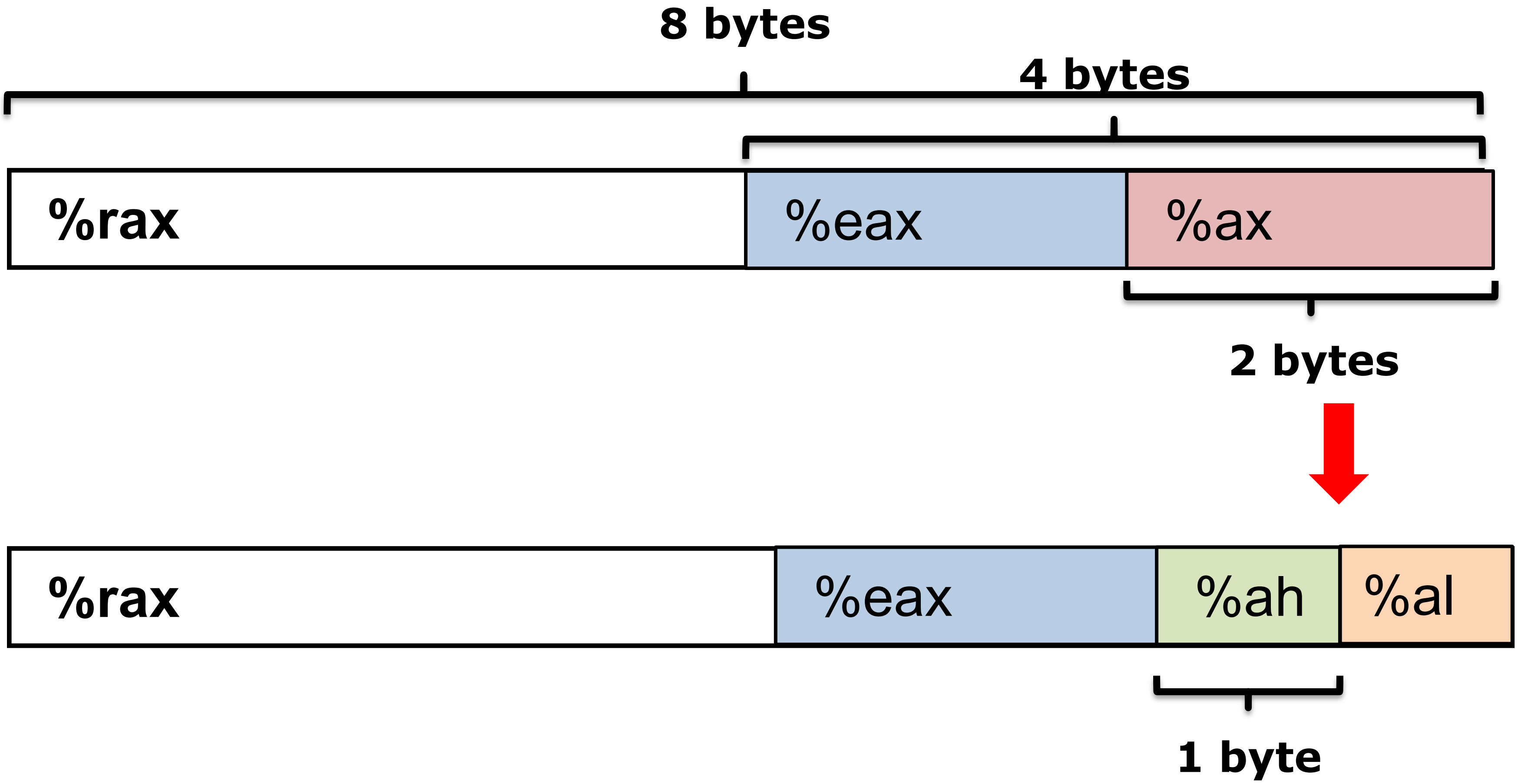
4 bytes

X86-64 general purpose registers: 2-byte

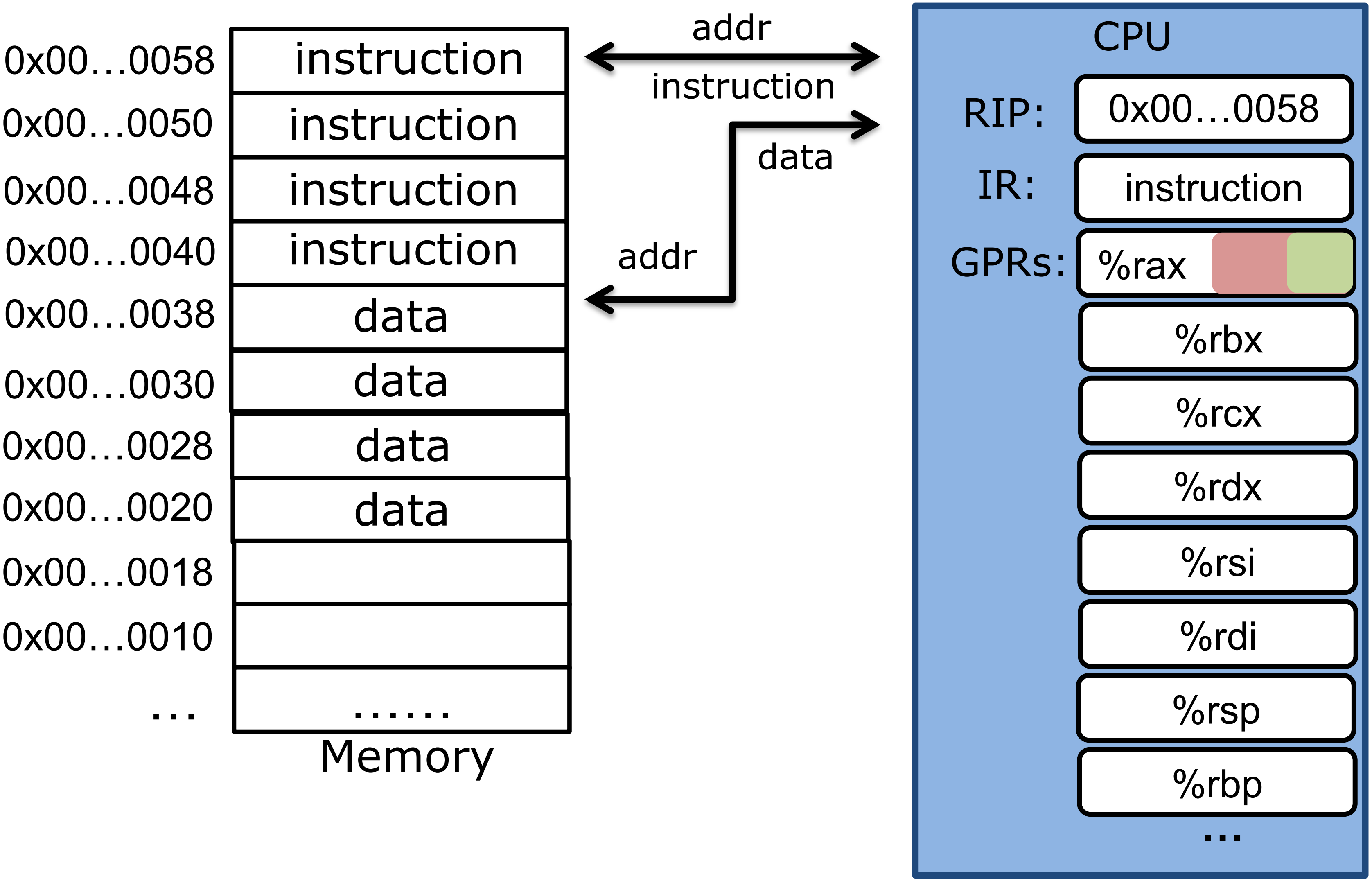
2-byte registers refer to the lower-order 2-bytes of original registers.



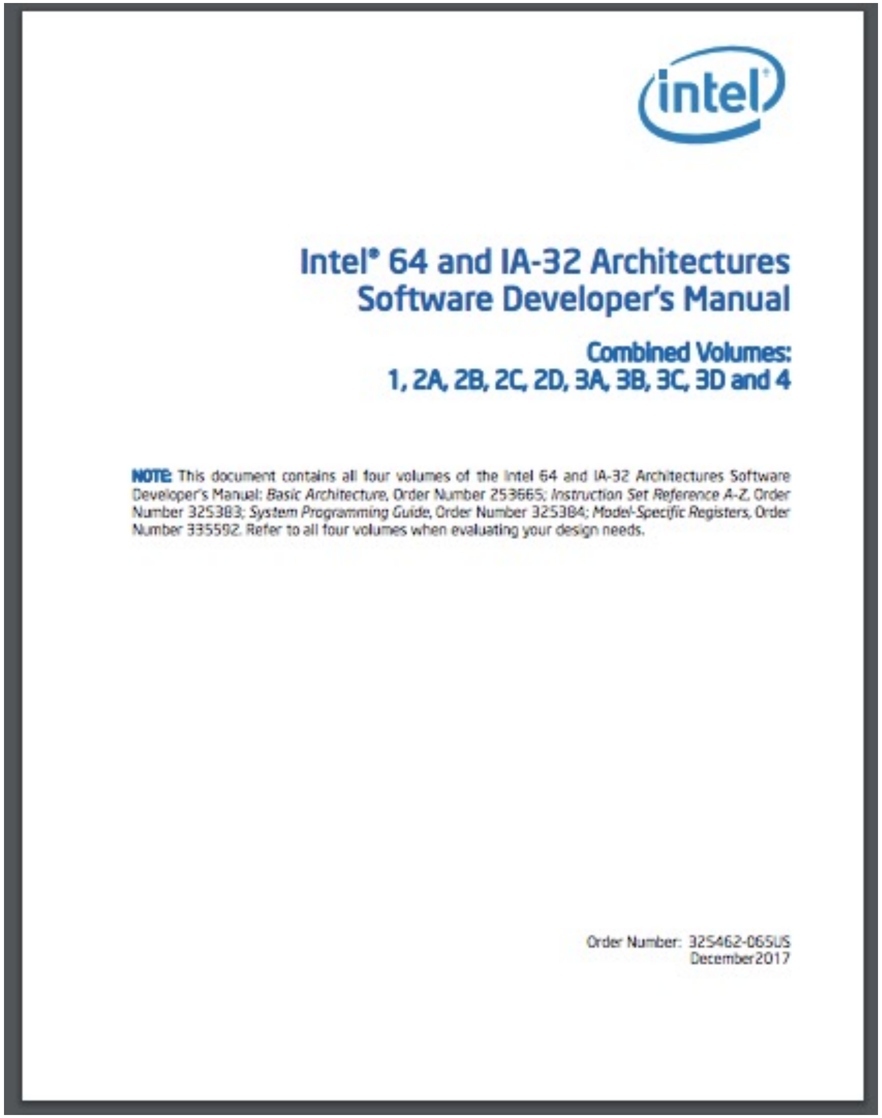
X86-64 general purpose registers: 1-byte



x86-64 execution



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

x86 instruction: Moving data

movq *Source*, *Dest*

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

We use AT&T (instead of Intel) syntax for assembly

Moving data

suffix

movq *Source*, *Dest*

– Copy a quadword (8-bytes) from the source operand to the 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*

- 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

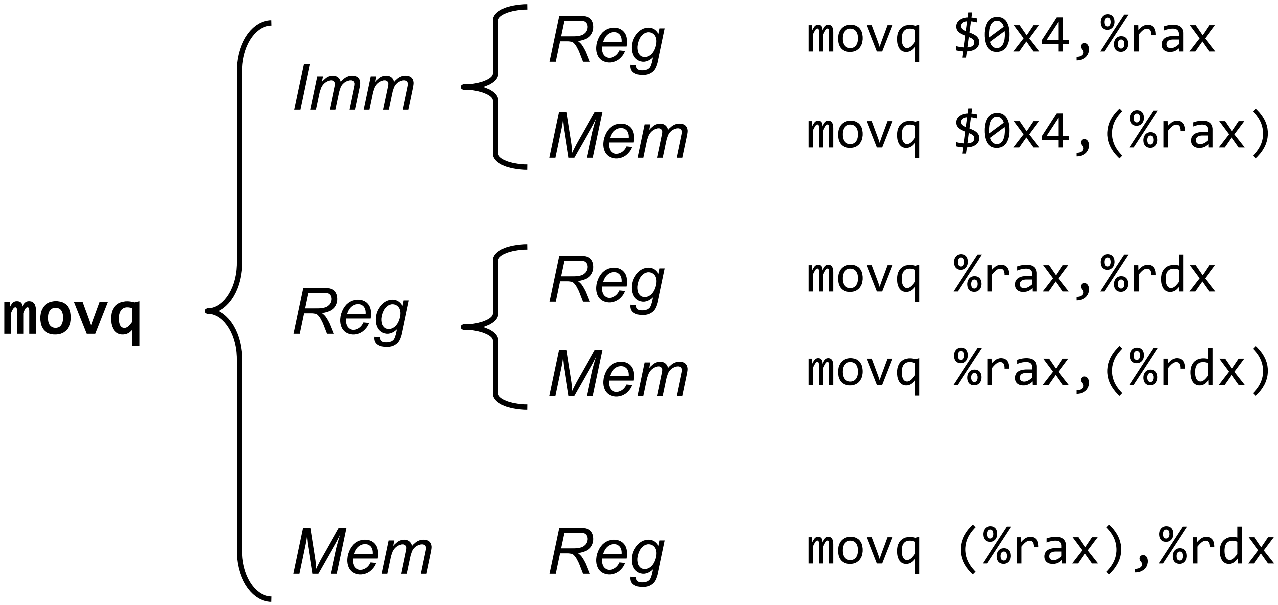
Moving data

movq *Source*, *Dest*

Operand Types

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

movq Operand combinations

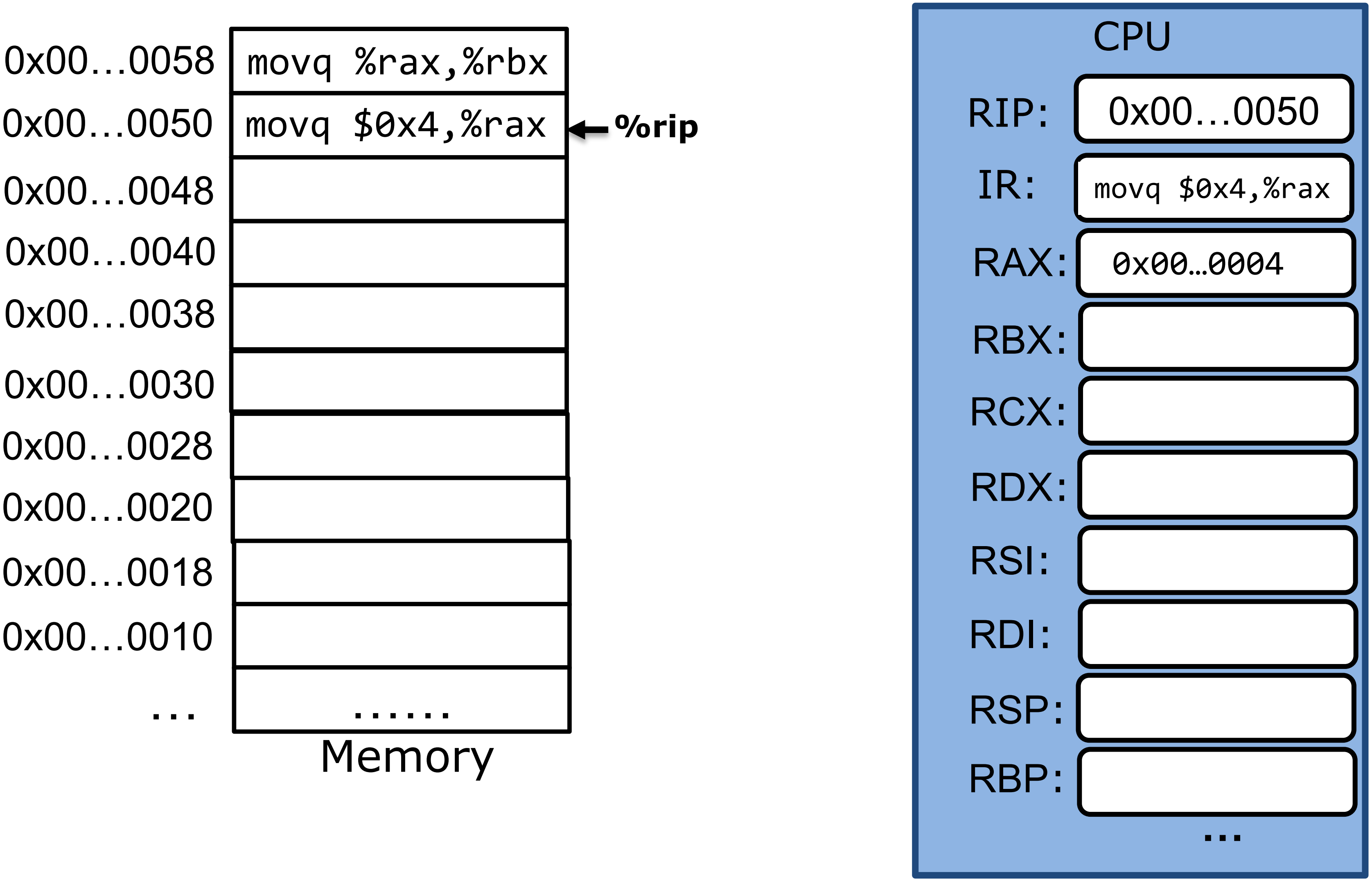


	Source	Dest	Example
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`
		Mem	`movq $0x4,(%rax)`
	Reg	*Reg*	`movq %rax,%rdx`
		Mem	`movq %rax,(%rdx)`
	Mem	*Reg*	`movq (%rax),%rdx`

1. Immediate can only be *source*

2. No memory-memory mov

movq *Imm, Reg*



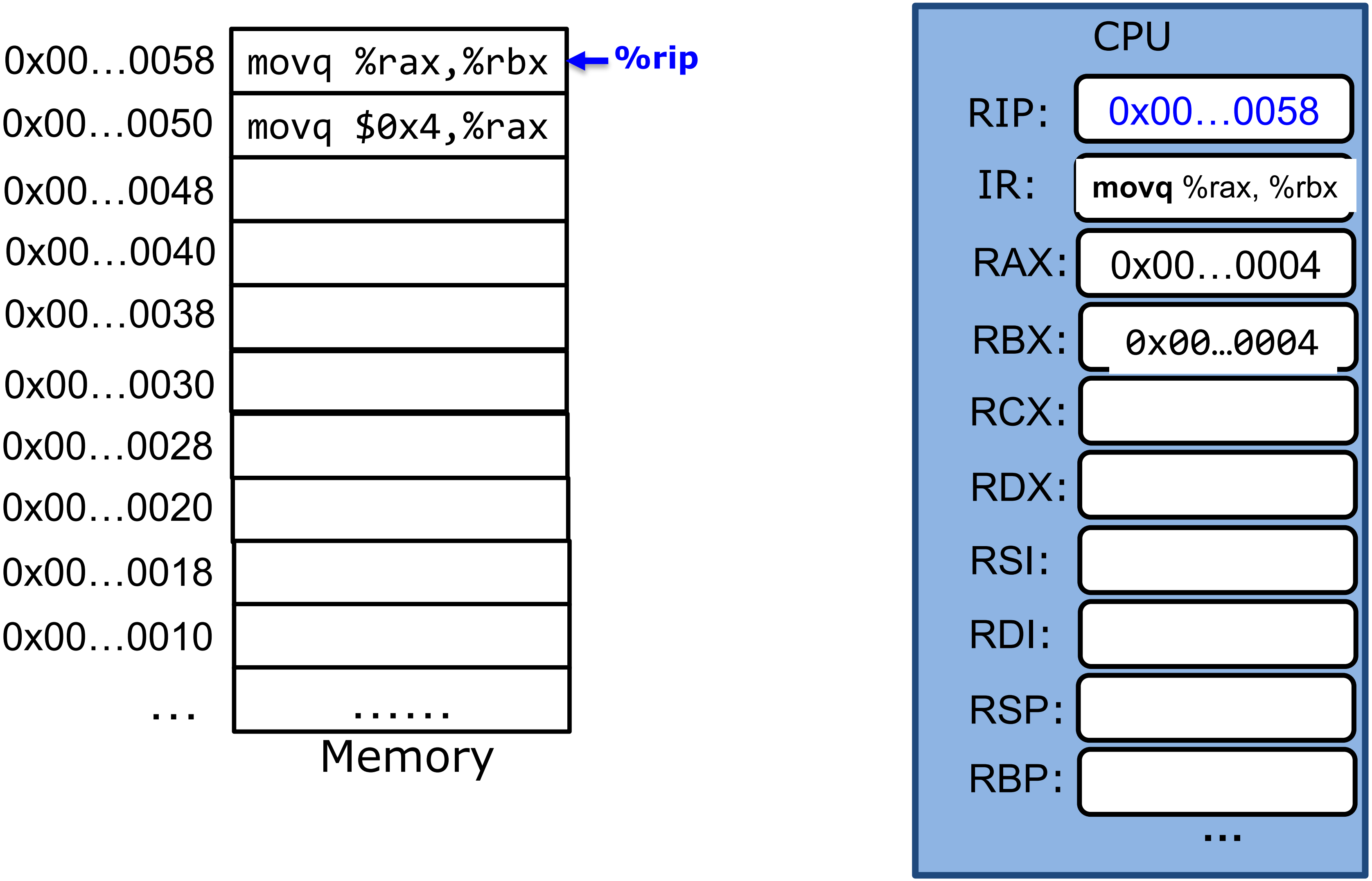
0x00…0058	movq %rax,%rbx
0x00…0050	movq $0x4,%rax ← %rip
0x00…0048	
0x00…0040	
0x00…0038	
0x00…0030	
0x00…0028	
0x00…0020	
0x00…0018	
0x00…0010	
…	……

Memory

CPU

RIP:	0x00…0050
IR:	movq $0x4,%rax
RAX:	0x00…0004
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	
	…

movq *Reg, Reg*



0x00…0058	movq %rax,%rbx ← %rip
0x00…0050	movq $0x4,%rax
0x00…0048	
0x00…0040	
0x00…0038	
0x00…0030	
0x00…0028	
0x00…0020	
0x00…0018	
0x00…0010	
…	……

Memory

CPU

RIP:	0x00…0058
IR:	**movq** %rax, %rbx
RAX:	0x00…0004
RBX:	0x00…0004
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	
	…

movq *Mem, Reg*

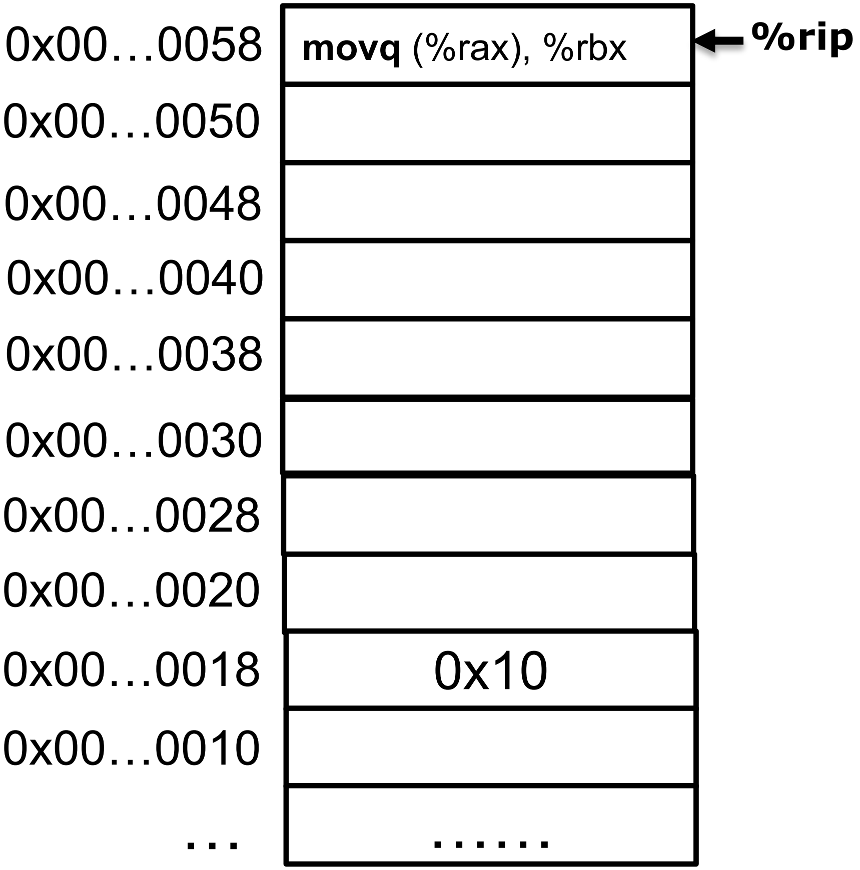
How to represent a “memory” operand?

Direct addressing: use register to index memory

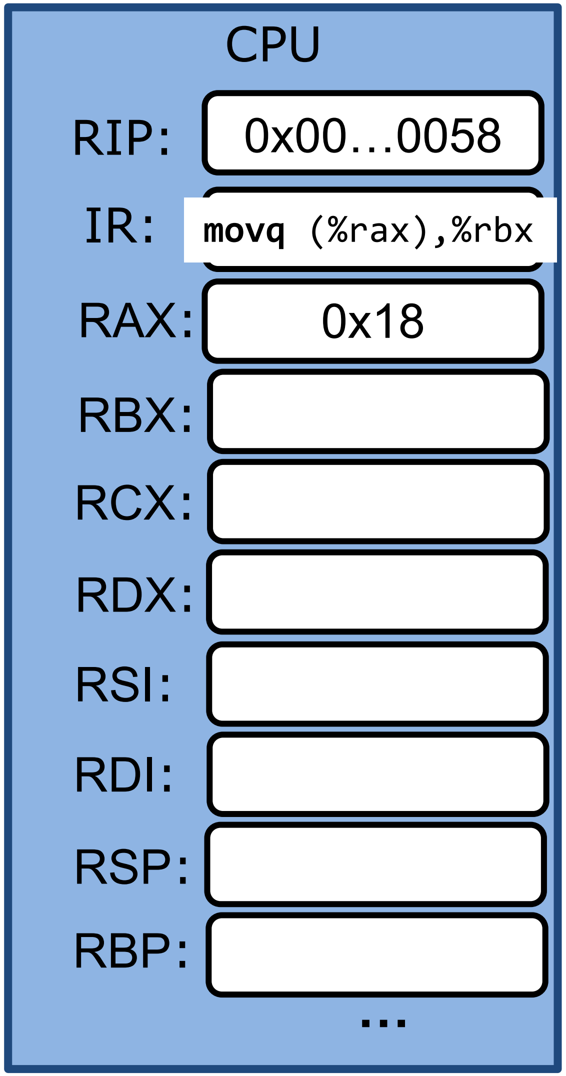
(Register)

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

movq *(%rax), %rbx*



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

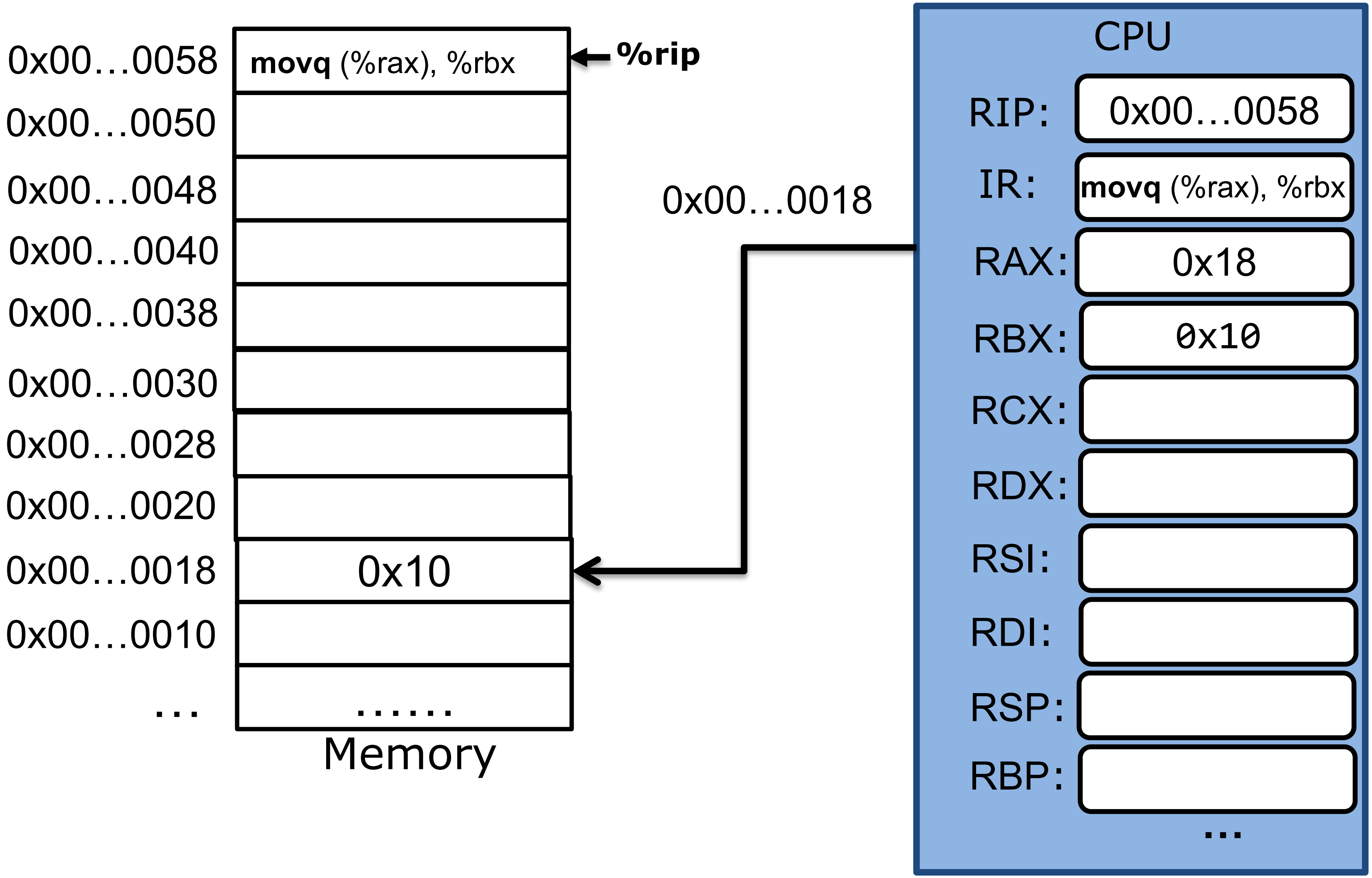


CPU

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

How many bytes are copied? Source? Destination?

movq *(%rax), %rbx*

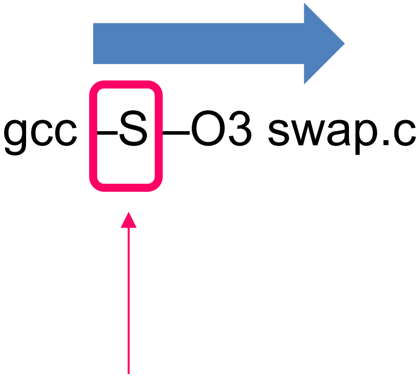


0x00…0058
movq (%rax), %rbx
%rip
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x10
0x00…0010
…
……
Memory
0x00…0018
CPU
RIP:
0x00…0058
IR:
movq (%rax), %rbx
RAX:
0x18
RBX:
0x10
RCX:
RDX:
RSI:
RDI:
RSP:
RBP:
…

swap function

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

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

}
```

```
swap:
```



gcc –S –O3 swap.c

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

%rdi stores a

%rsi stores b

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

%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

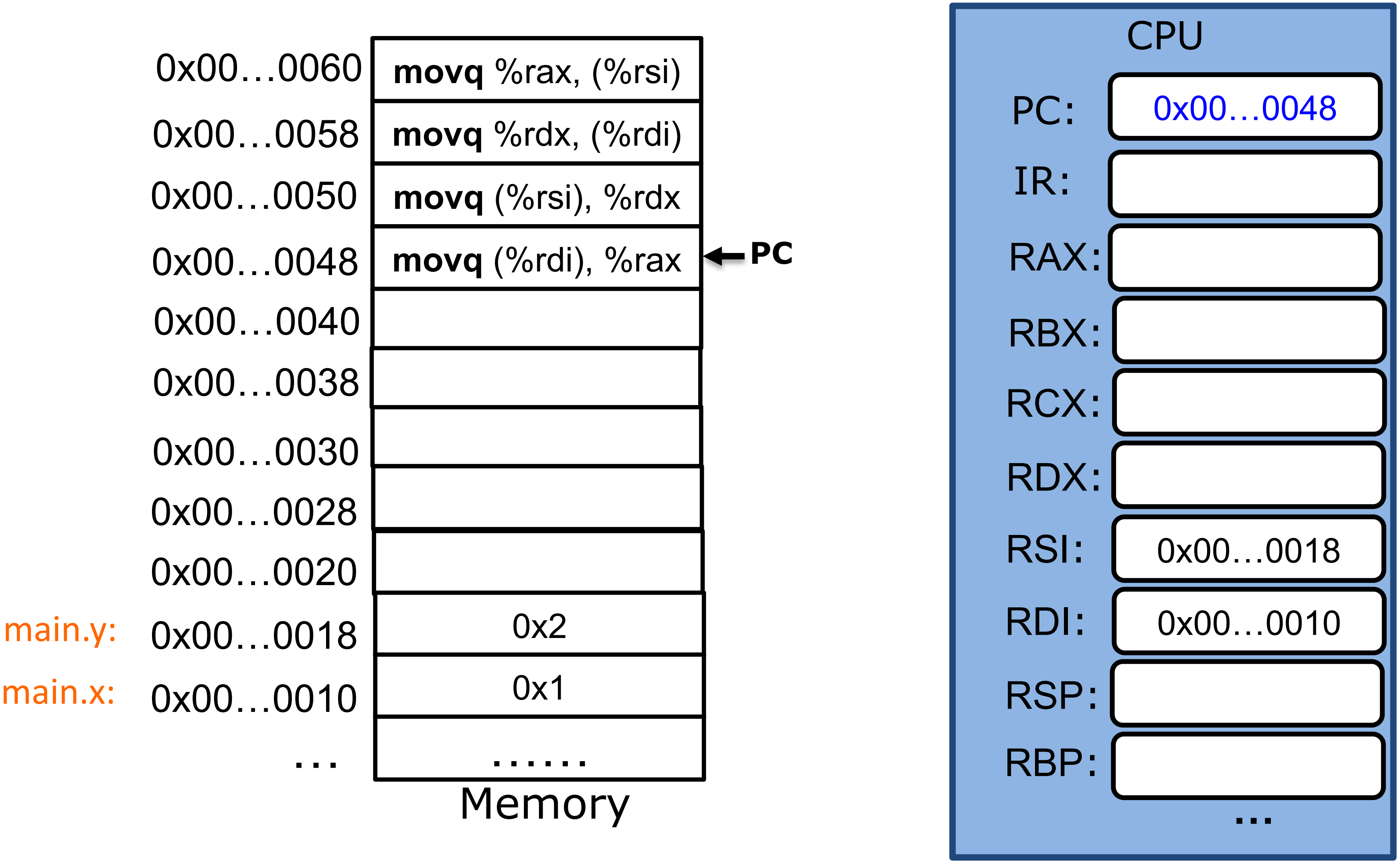
swap function

```
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



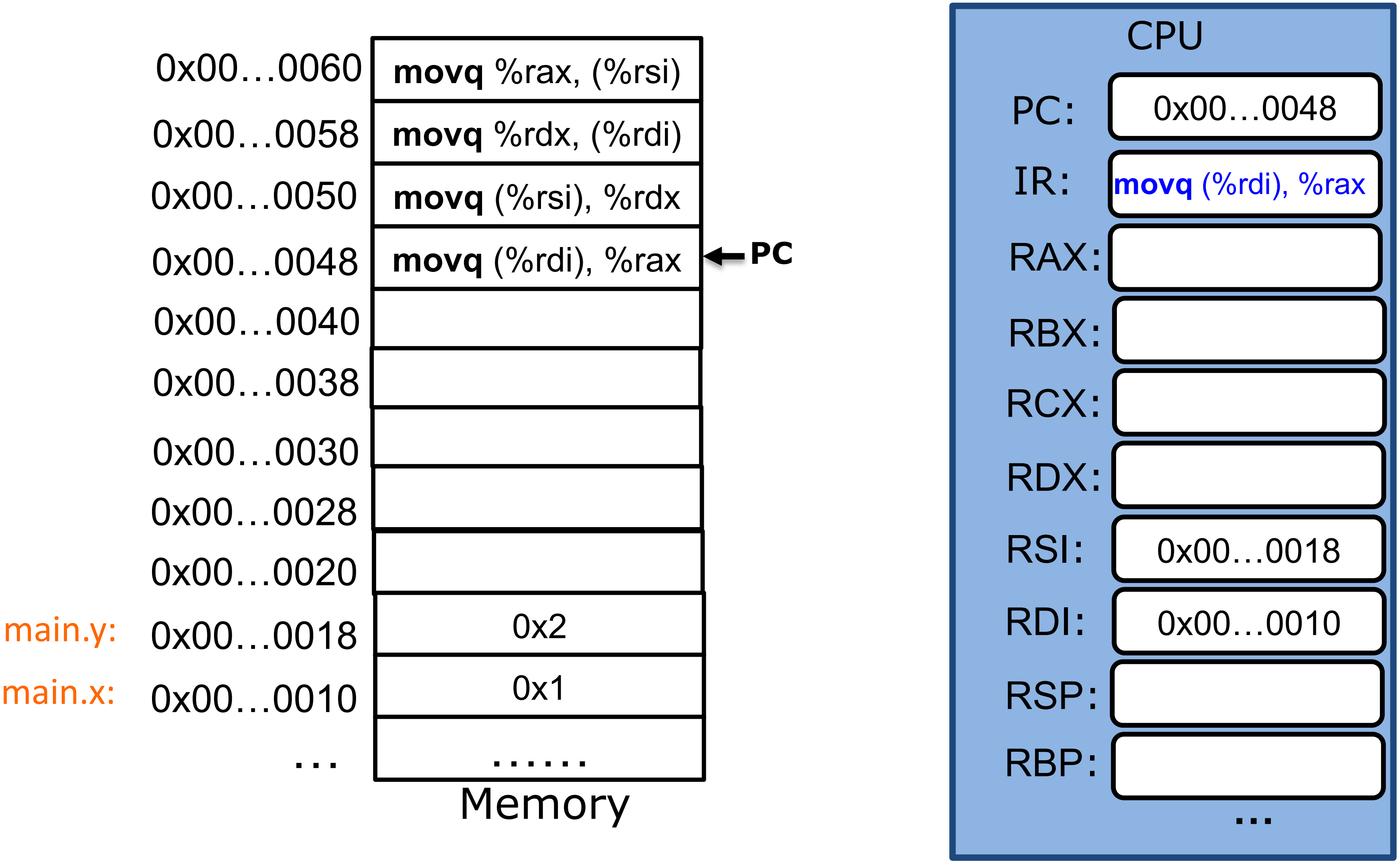
	0x00…0060	**movq** %rax, (%rsi)	
	0x00…0058	**movq** %rdx, (%rdi)	
	0x00…0050	**movq** (%rsi), %rdx	
	0x00…0048	**movq** (%rdi), %rax	← **PC**
	0x00…0040		
	0x00…0038		
	0x00…0030		
	0x00…0028		
	0x00…0020		
main.y:	0x00…0018	0x2	
main.x:	0x00…0010	0x1	
	…	……	

Memory

CPU

PC:	0x00…0048
IR:	
RAX:	
RBX:	
RCX:	
RDX:	
RSI:	0x00…0018
RDI:	0x00…0010
RSP:	
RBP:	
	…

swap func



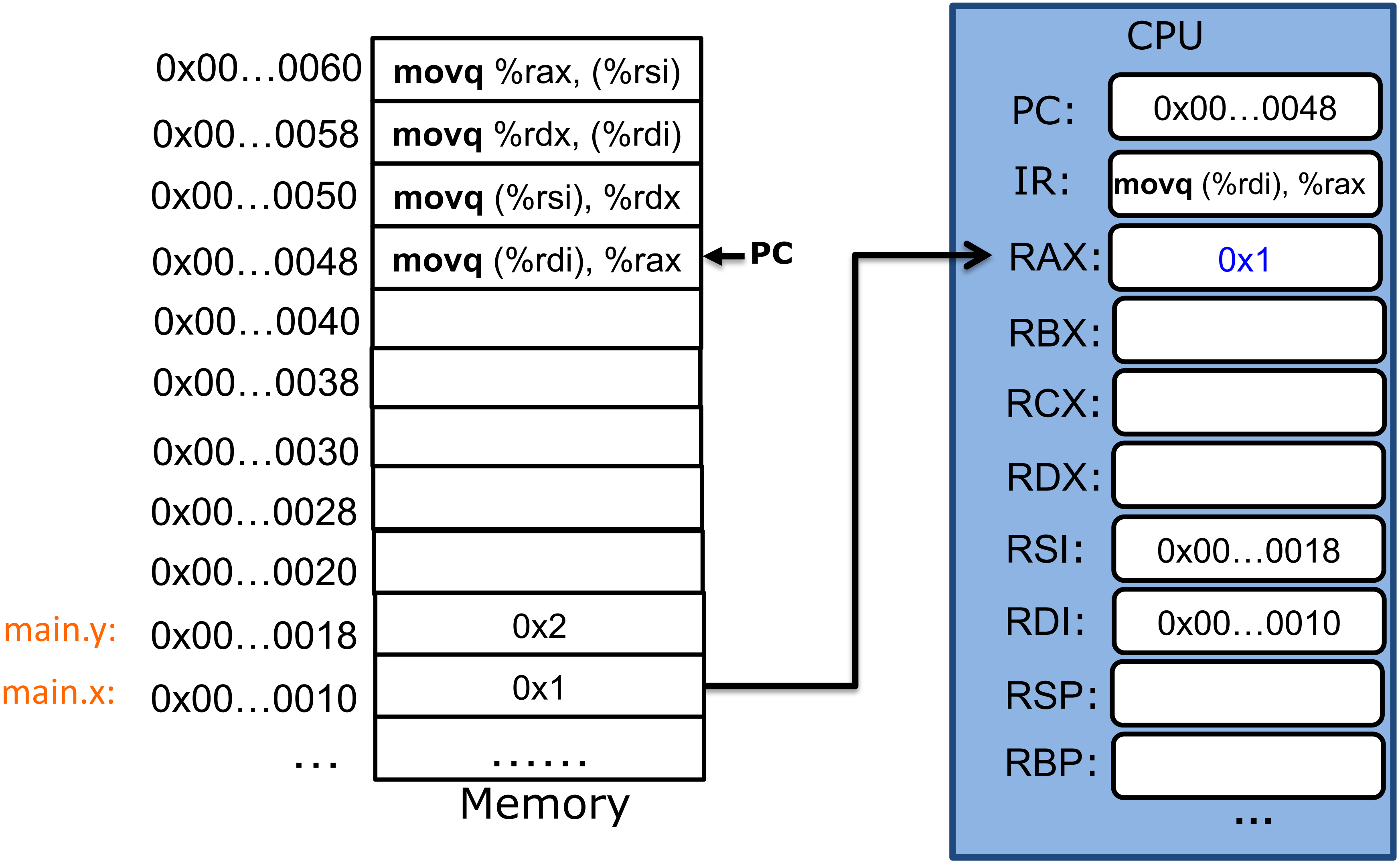
	0x00…0060	**movq** %rax, (%rsi)
	0x00…0058	**movq** %rdx, (%rdi)
	0x00…0050	**movq** (%rsi), %rdx
	0x00…0048	**movq** (%rdi), %rax ← **PC**
	0x00…0040	
	0x00…0038	
	0x00…0030	
	0x00…0028	
	0x00…0020	
main.y:	0x00…0018	0x2
main.x:	0x00…0010	0x1
	…	……

Memory

CPU

PC:	0x00…0048
IR:	**movq** (%rdi), %rax
RAX:	
RBX:	
RCX:	
RDX:	
RSI:	0x00…0018
RDI:	0x00…0010
RSP:	
RBP:	
…	

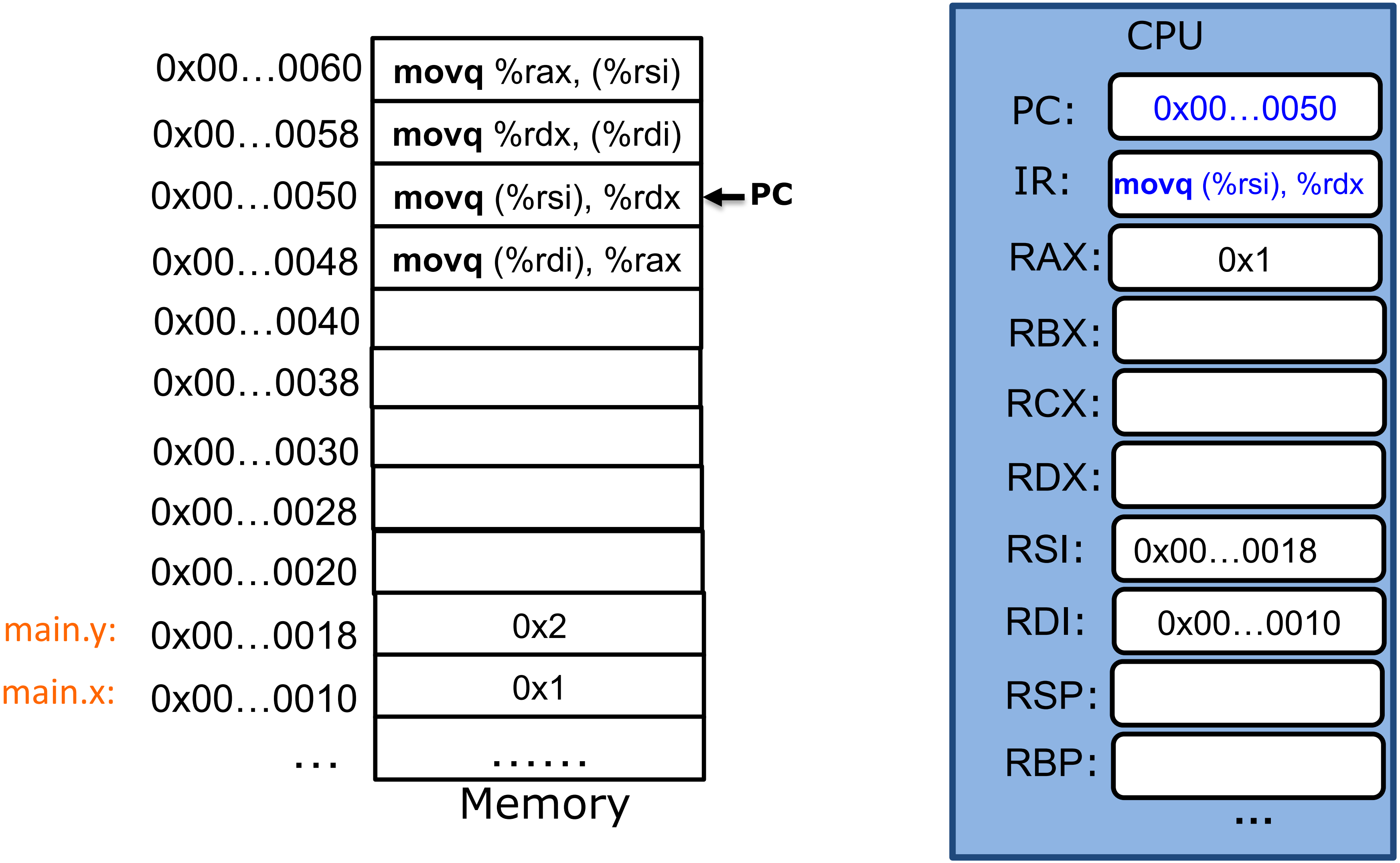
swap func



		Memory
	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
	…	……

CPU	
PC:	0x00…0048
IR:	**movq** (%rdi), %rax
RAX:	0x1
RBX:	
RCX:	
RDX:	
RSI:	0x00…0018
RDI:	0x00…0010
RSP:	
RBP:	
…	

swap func



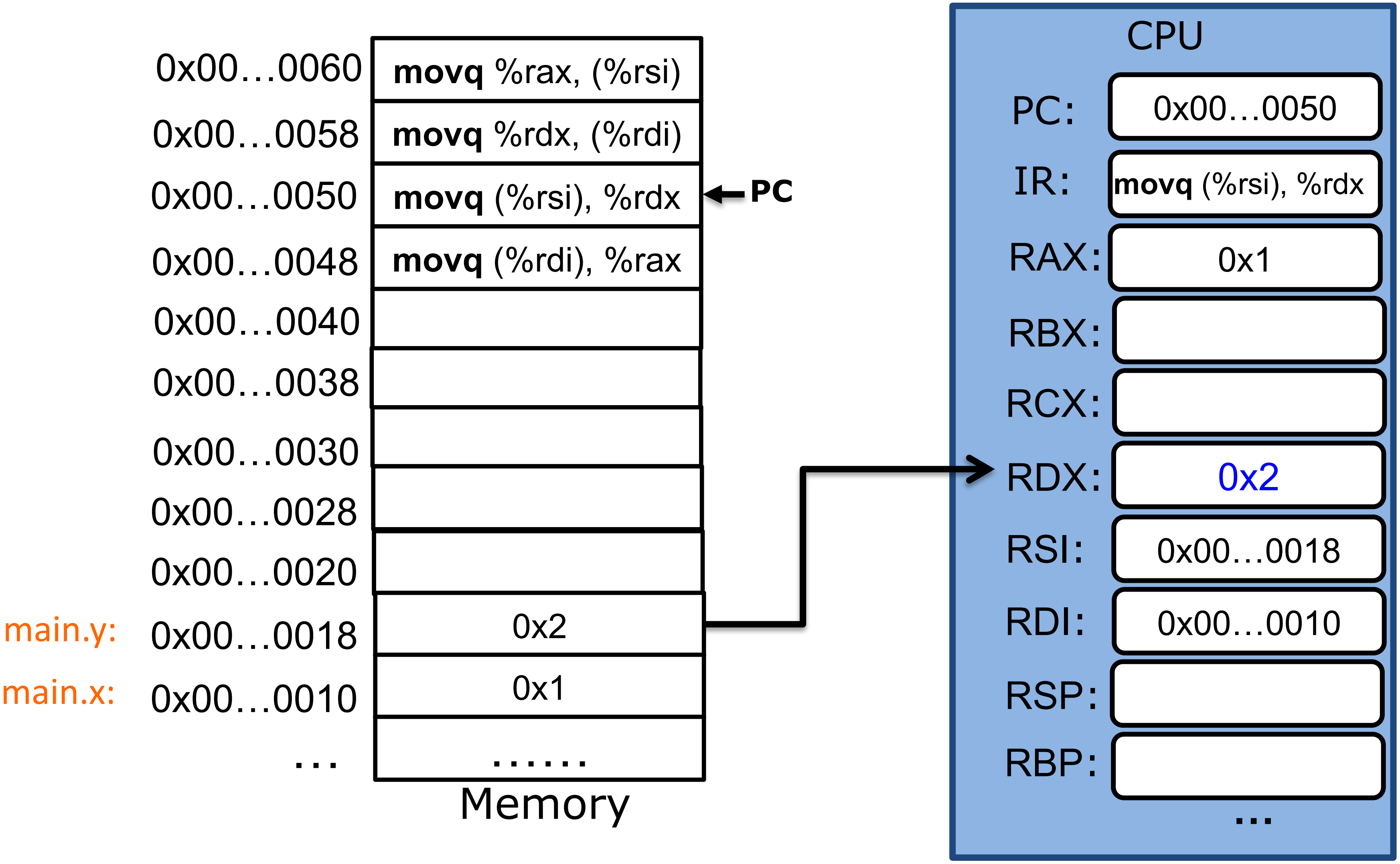
	Address	Memory
	0x00…0060	**movq** %rax, (%rsi)
	0x00…0058	**movq** %rdx, (%rdi)
	0x00…0050	**movq** (%rsi), %rdx ← **PC**
	0x00…0048	**movq** (%rdi), %rax
	0x00…0040	
	0x00…0038	
	0x00…0030	
	0x00…0028	
	0x00…0020	
main.y:	0x00…0018	0x2
main.x:	0x00…0010	0x1
	…	……

Memory

CPU

PC:	0x00…0050
IR:	**movq** (%rsi), %rdx
RAX:	0x1
RBX:	
RCX:	
RDX:	
RSI:	0x00…0018
RDI:	0x00…0010
RSP:	
RBP:	
	…

swap func

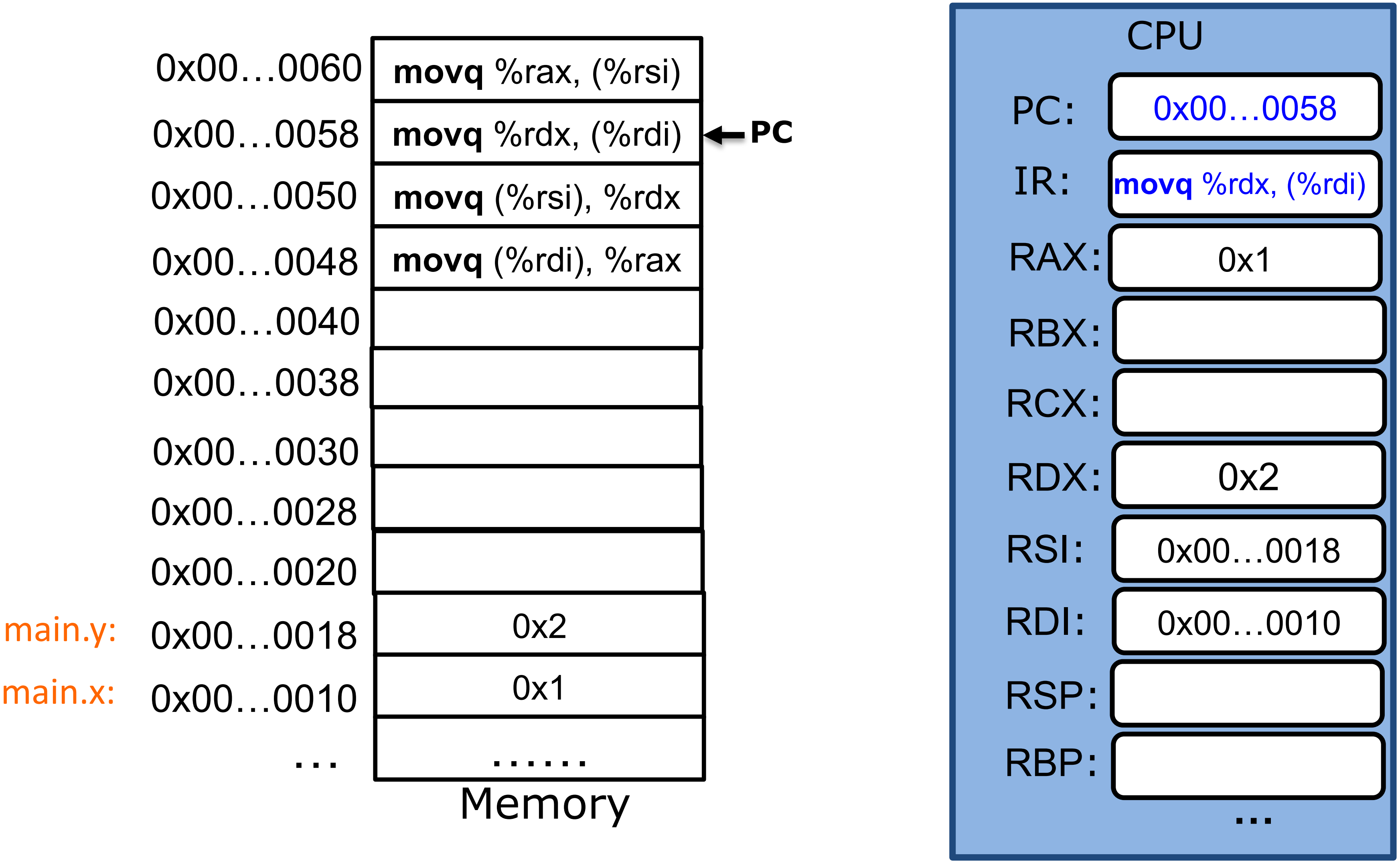


	0x00…0060	**movq** %rax, (%rsi)
	0x00…0058	**movq** %rdx, (%rdi)
	0x00…0050	**movq** (%rsi), %rdx
	0x00…0048	**movq** (%rdi), %rax
	0x00…0040	
	0x00…0038	
	0x00…0030	
	0x00…0028	
	0x00…0020	
main.y:	0x00…0018	0x2
main.x:	0x00…0010	0x1
	…	……

CPU

PC:	0x00…0050
IR:	**movq** (%rsi), %rdx
RAX:	0x1
RBX:	
RCX:	
RDX:	0x2
RSI:	0x00…0018
RDI:	0x00…0010
RSP:	
RBP:	
	…

swap func



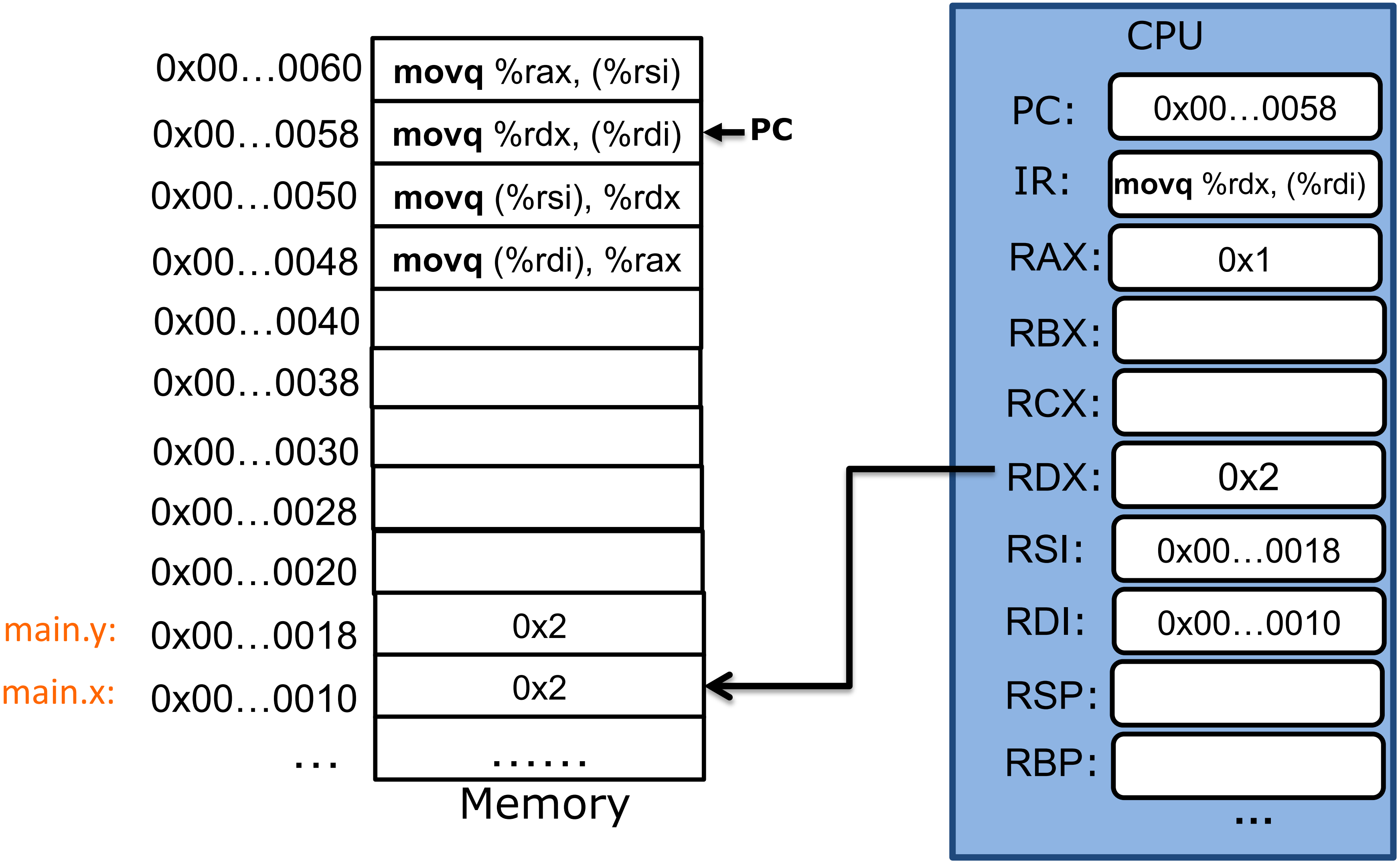
	Address	Memory	
	0x00…0060	**movq** %rax, (%rsi)	
	0x00…0058	**movq** %rdx, (%rdi)	← **PC**
	0x00…0050	**movq** (%rsi), %rdx	
	0x00…0048	**movq** (%rdi), %rax	
	0x00…0040		
	0x00…0038		
	0x00…0030		
	0x00…0028		
	0x00…0020		
main.y:	0x00…0018	0x2	
main.x:	0x00…0010	0x1	
	…	……	

Memory

CPU

PC:	0x00…0058
IR:	**movq** %rdx, (%rdi)
RAX:	0x1
RBX:	
RCX:	
RDX:	0x2
RSI:	0x00…0018
RDI:	0x00…0010
RSP:	
RBP:	
	…

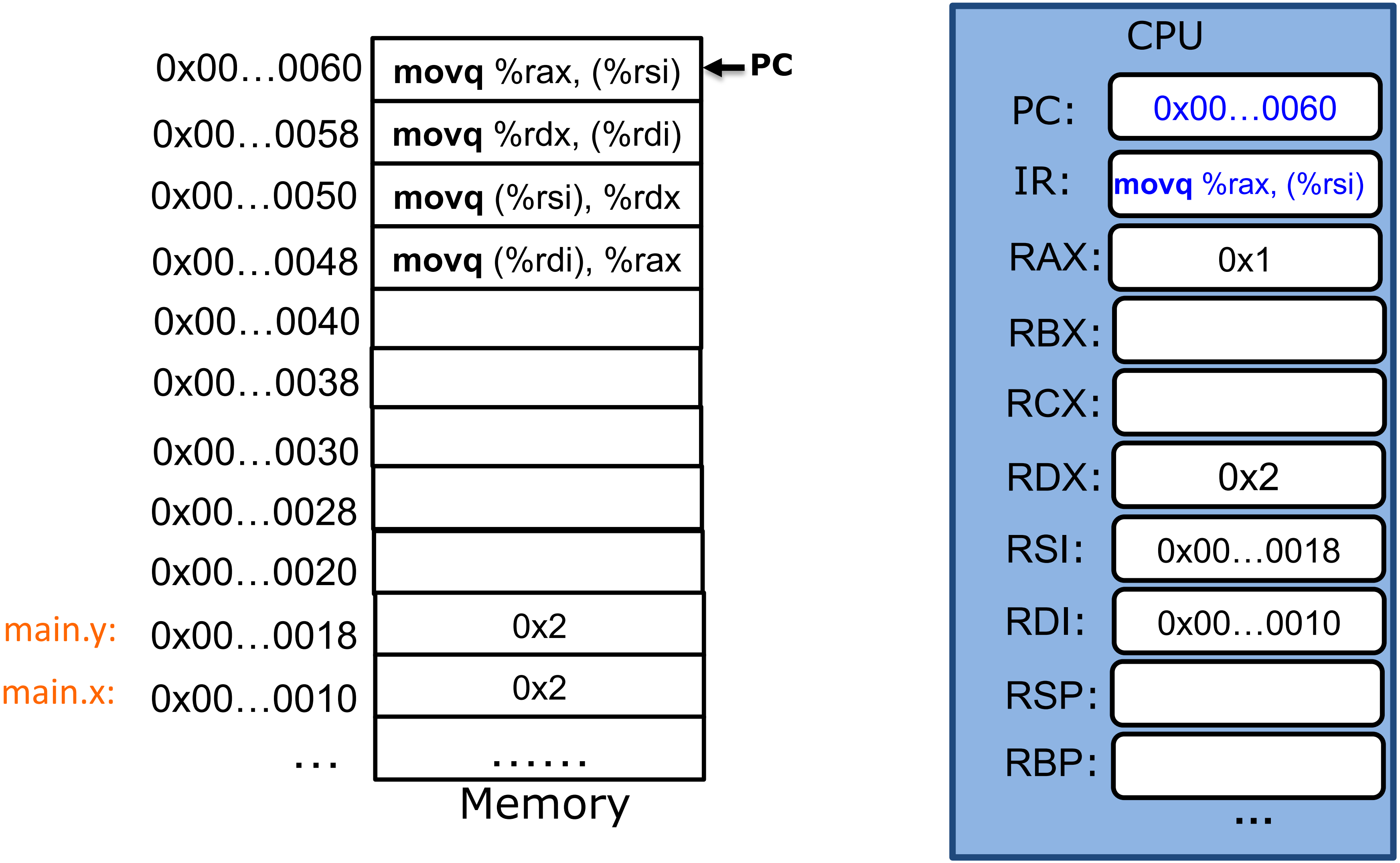
swap func



		Memory
	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
	…	……

CPU	
PC:	0x00…0058
IR:	**movq** %rdx, (%rdi)
RAX:	0x1
RBX:	
RCX:	
RDX:	0x2
RSI:	0x00…0018
RDI:	0x00…0010
RSP:	
RBP:	
	…

swap func

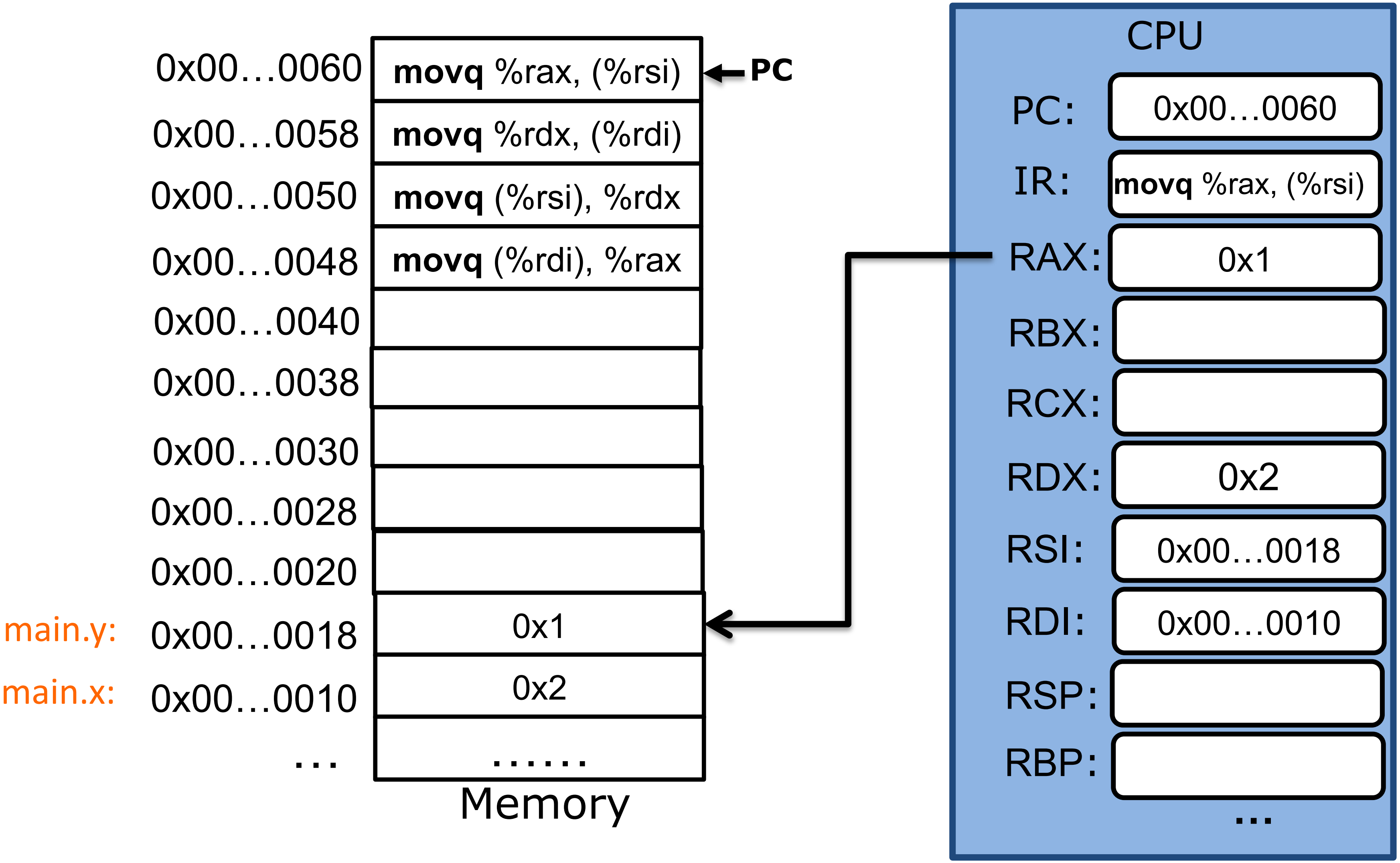


		Memory
	0x00…0060	**movq** %rax, (%rsi) ← **PC**
	0x00…0058	**movq** %rdx, (%rdi)
	0x00…0050	**movq** (%rsi), %rdx
	0x00…0048	**movq** (%rdi), %rax
	0x00…0040	
	0x00…0038	
	0x00…0030	
	0x00…0028	
	0x00…0020	
main.y:	0x00…0018	0x2
main.x:	0x00…0010	0x2
	…	……

CPU

PC:	0x00…0060
IR:	**movq** %rax, (%rsi)
RAX:	0x1
RBX:	
RCX:	
RDX:	0x2
RSI:	0x00…0018
RDI:	0x00…0010
RSP:	
RBP:	
	…

swap func



	Address	Memory
	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:	
…	

Summary

- Basic hardware execution
  - Instructions and data stored in memory
  - CPU fetches instructions one at a time according to PC
- X86-64 ISA
  - %rip (PC), 16 general-purpose registers
  - movq allows copying data across registers or memory ↔register.