Machine Program: Arithmetic and Control

Jinyang Li

based on Tiger Wang's slides

What we've learnt so far

- Basic hardware organization
  - CPU (PC/RIP, general-purpose registers)
  - Memory (byte-addressable)
- x86 ISA
  - b, w, l, q suffix
- **`mov`** instruction
  - register to register, memory to register, register to memory
  - Addressing modes: D(Rb, Ri, S)

Today's lesson plan

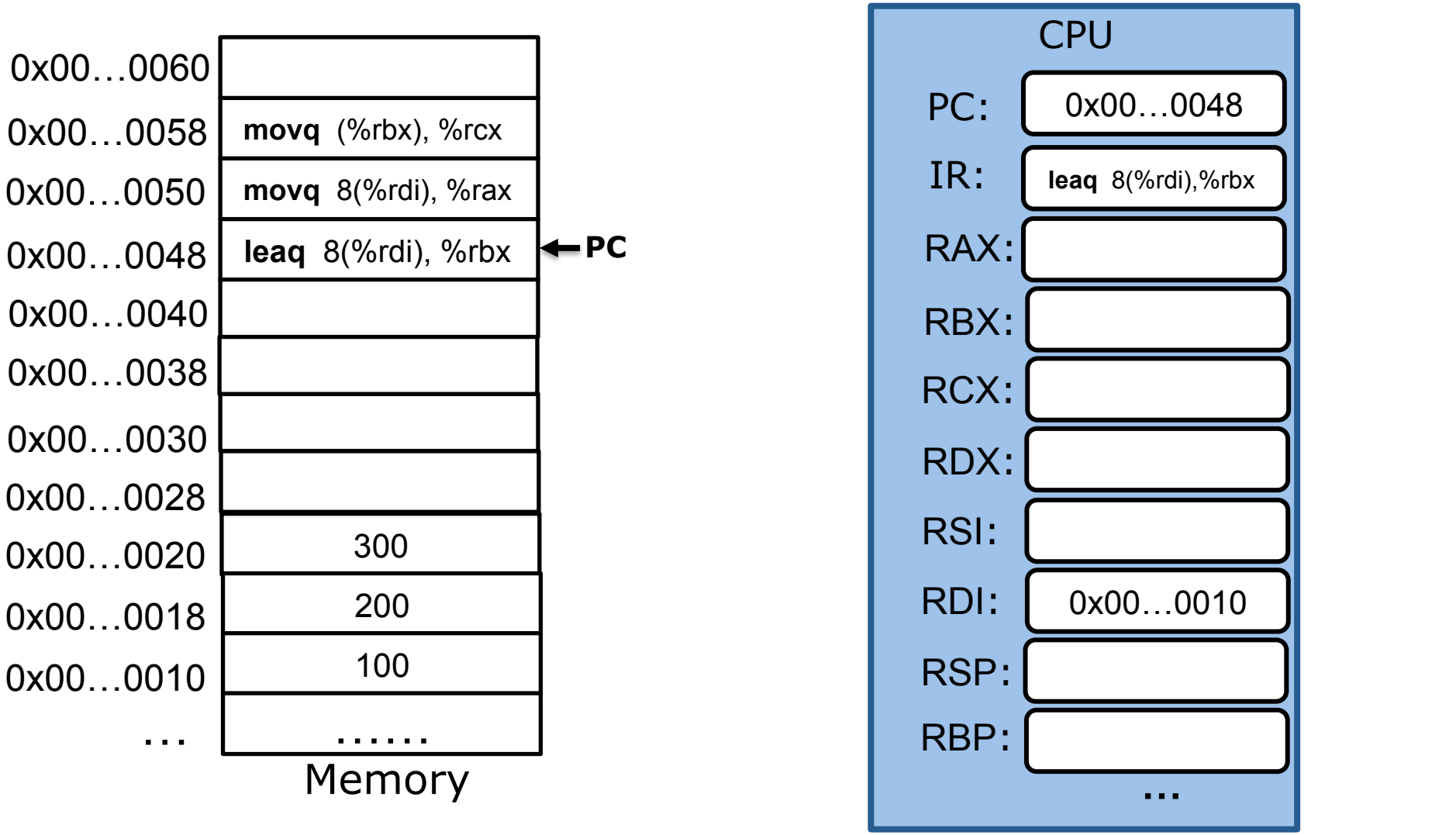
- Arithmetic instructions
- Control instructions

The lea instruction

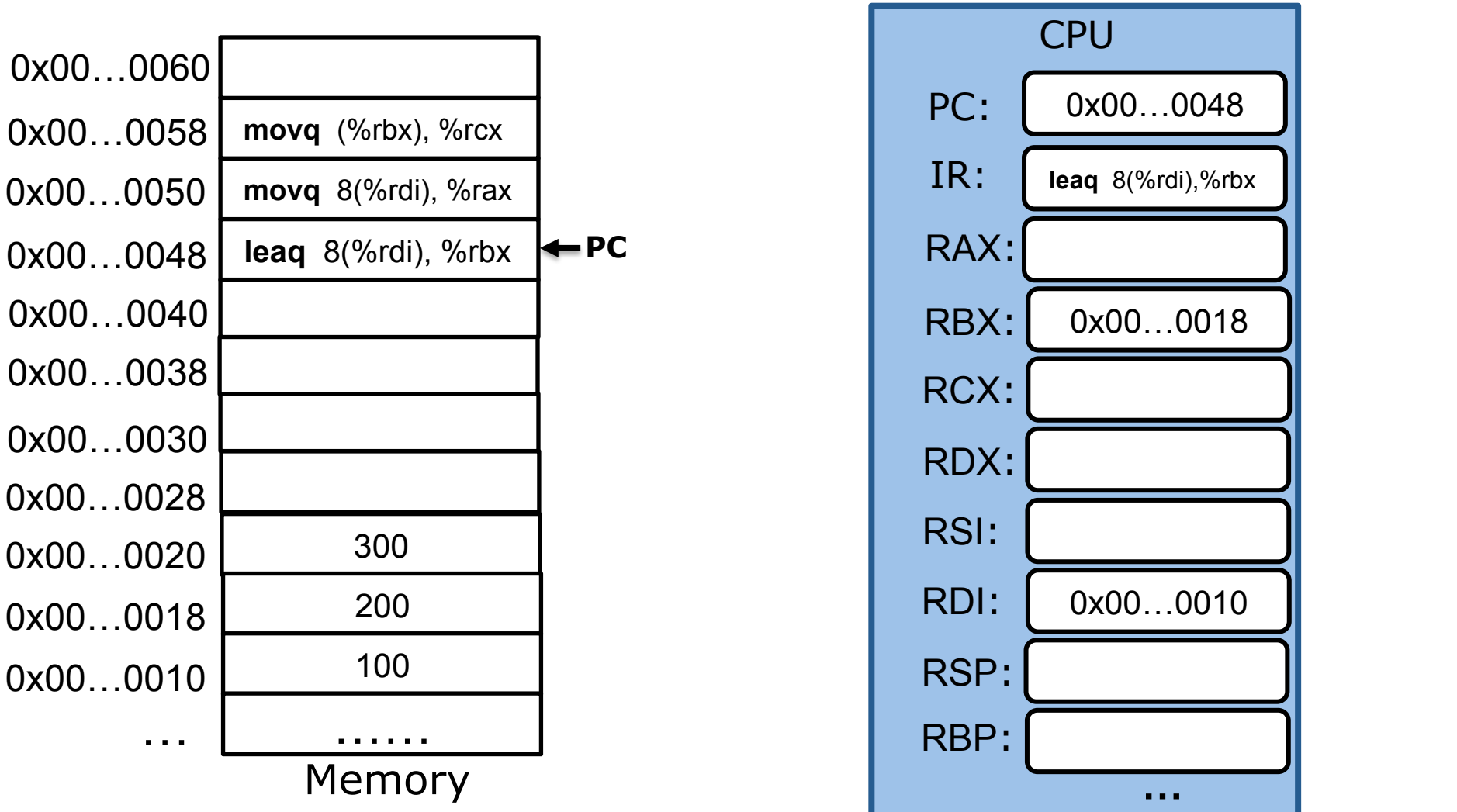
leaq *Source, Dest*

- 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:
…

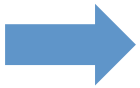
Example


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;
}
```



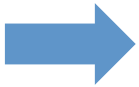
```
leaq (%rdi, %rdi,2), %rax
```

Assume %rdi has the value of x

Arithmetic Expression Puzzle

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

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



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

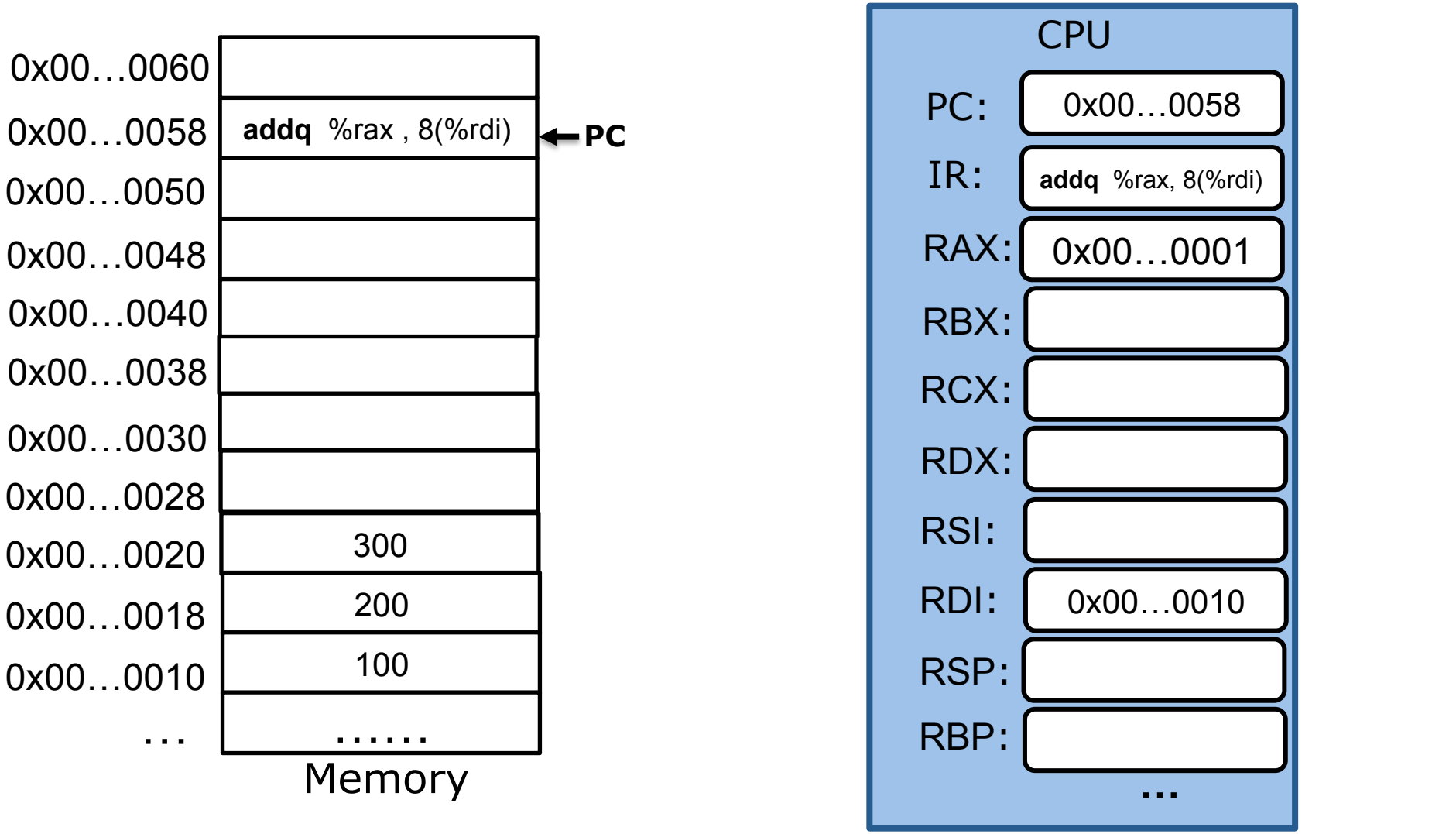
Basic Arithmetic Operations

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

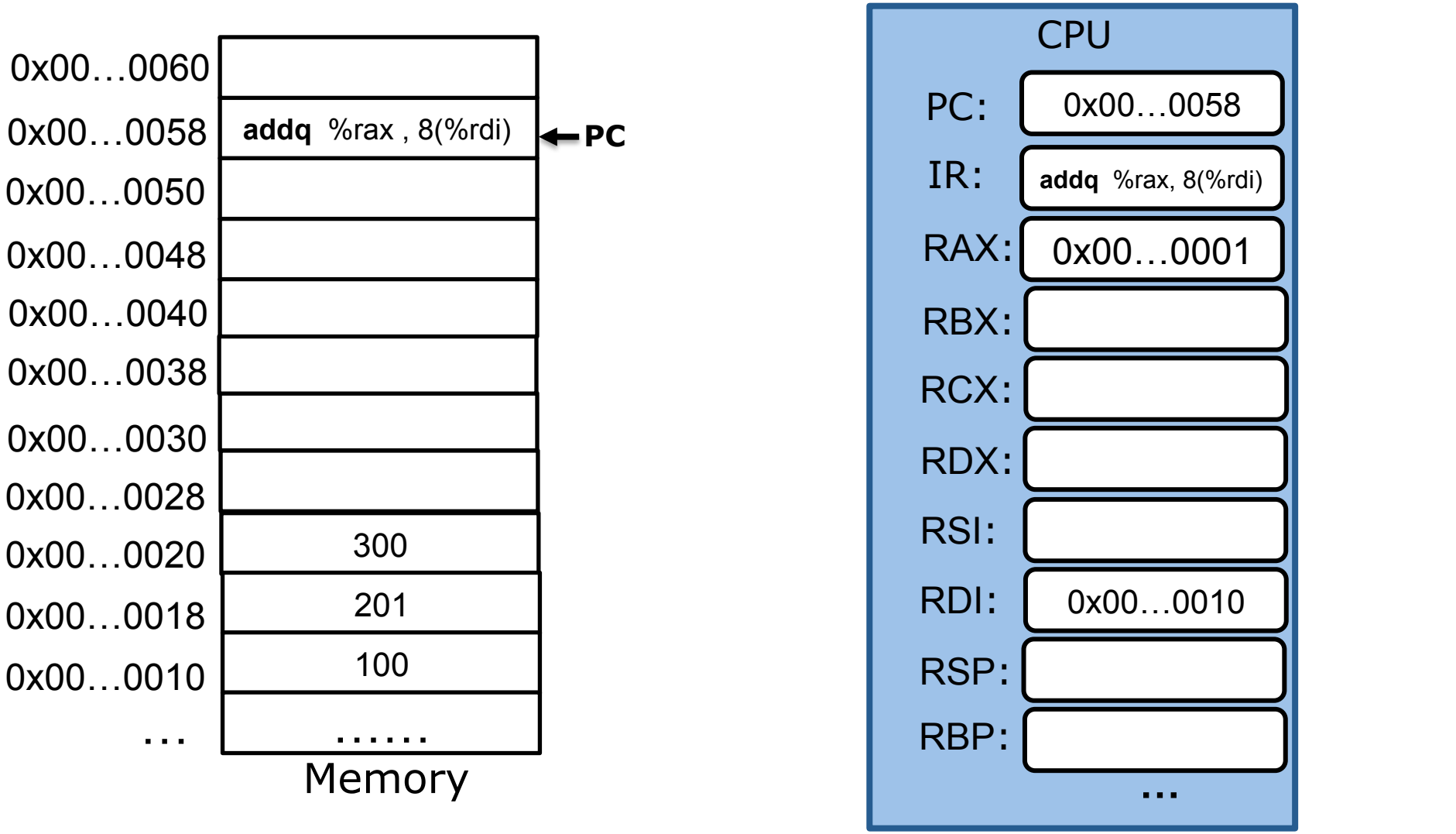
Bitwise Operations

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:
…

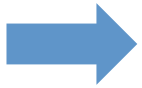
Example


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:
…

Control instructions

How is control flow realized?

```
if (*a > 0)
  *a--;
```



???

Control flow uses EFLAGS register

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:

- Contain status of the instruction executed
- All called “**EFLAGS**” in x86_64

EFLAGS register overview

- EFLAGS is a special purpose register
- Different bits represent different status flags
- Various instructions may set certain flags
  - regular arithmetic instructions
  - **`cmp, test, set`** instructions
- Control instructions use flags to determine control flow

EFLAGS register: ZF

- ZF (Zero Flag):
  - Set if the result of the instruction is zero; cleared otherwise.

```
movq $2, %rax
subq $2, %rax
```

EFLAGS register: SF

- SF (Sign Flag):
  - Set to be the most-significant bit of the result.

```
movq $2, %rax
subq $10, %rax
```

EFLAGS register: CF

- CF (Carry Flag):
  1. Set if adding two numbers carries out of the most significant bit
  2. Set if subtracting one number from the other borrows out of the most significant bit

```
movq $0xffffffffffffffff, %rax
addq $2, %rax
```

```
movq $0, %rax
subq $1, %rax
```

EFLAGS register: OF

- OF (Overflow Flag):
  - Overflow for signed integer (2's complement) arithmetic.

```
movq $0x7fffffffffffffff, %rax
addq $1, %rax
```

```
movq $0x7fffffffffffffff, %rax
subq $-1, %rax
```

CF and OF are different flags

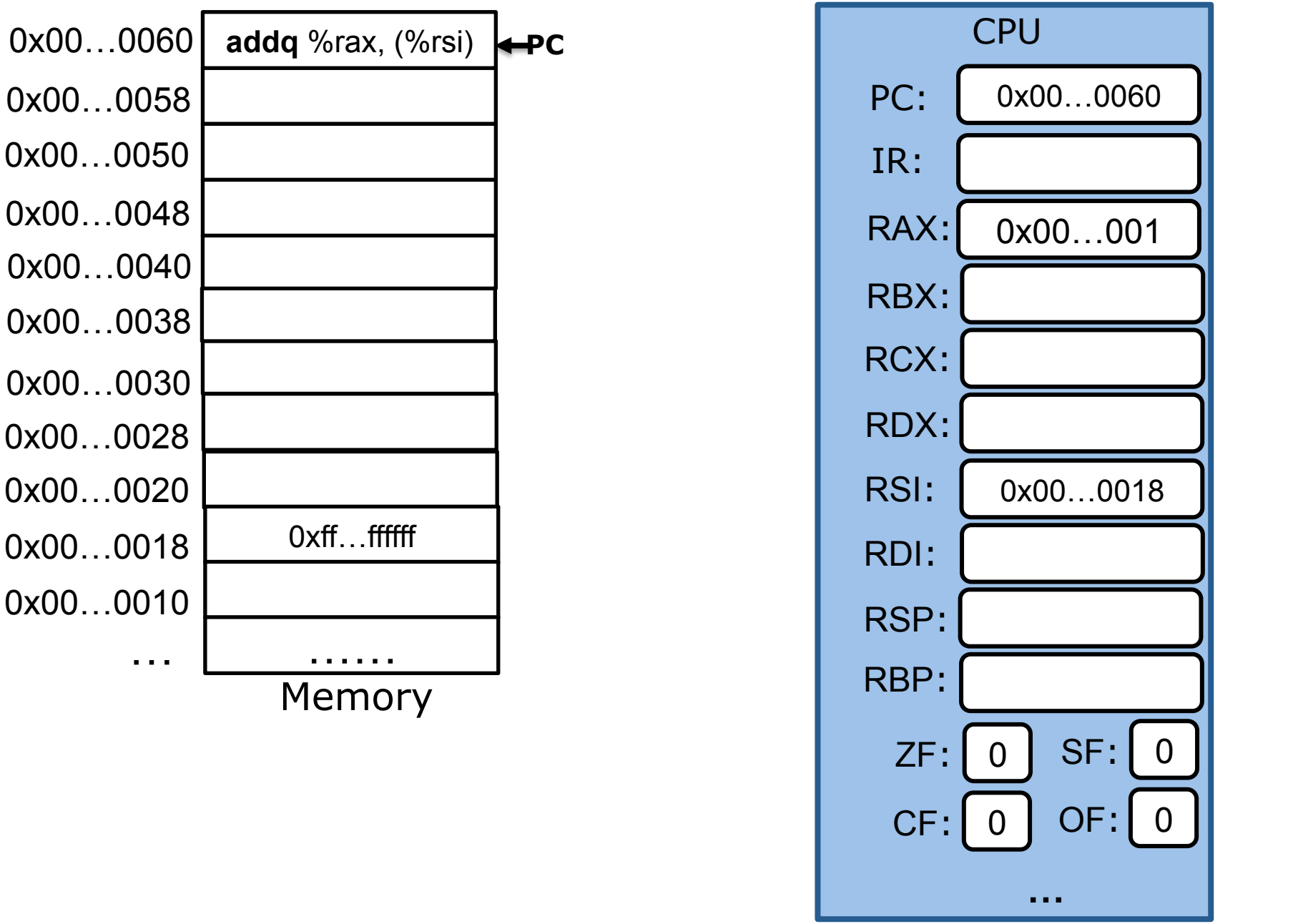
- CPU is not aware of whether data is signed or unsigned integer:
  - OF and CF flags are set by examining carry/borrow and MSB (sign bit).
- Up to programmer/compiler to check the right flag

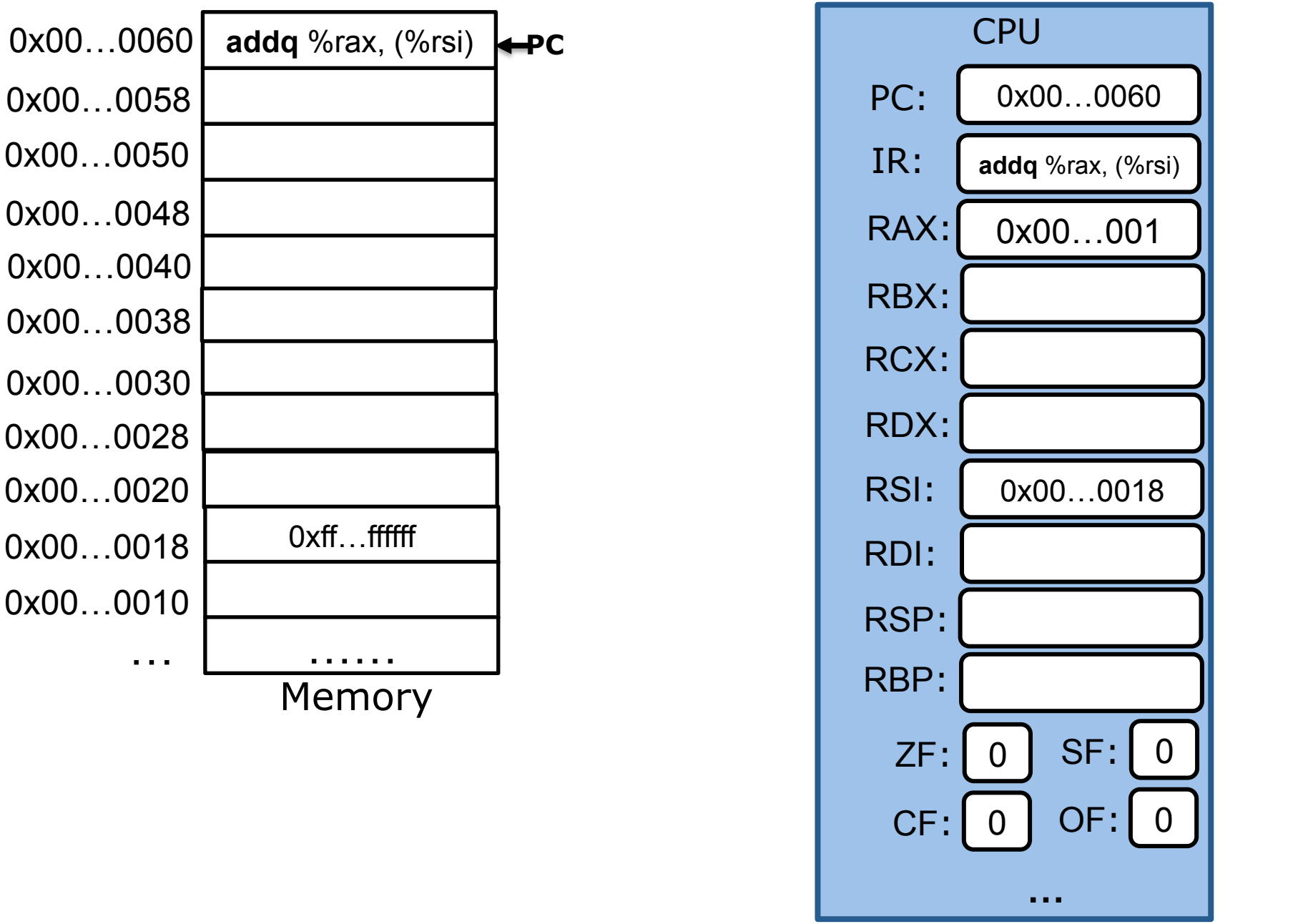
Status flags summary

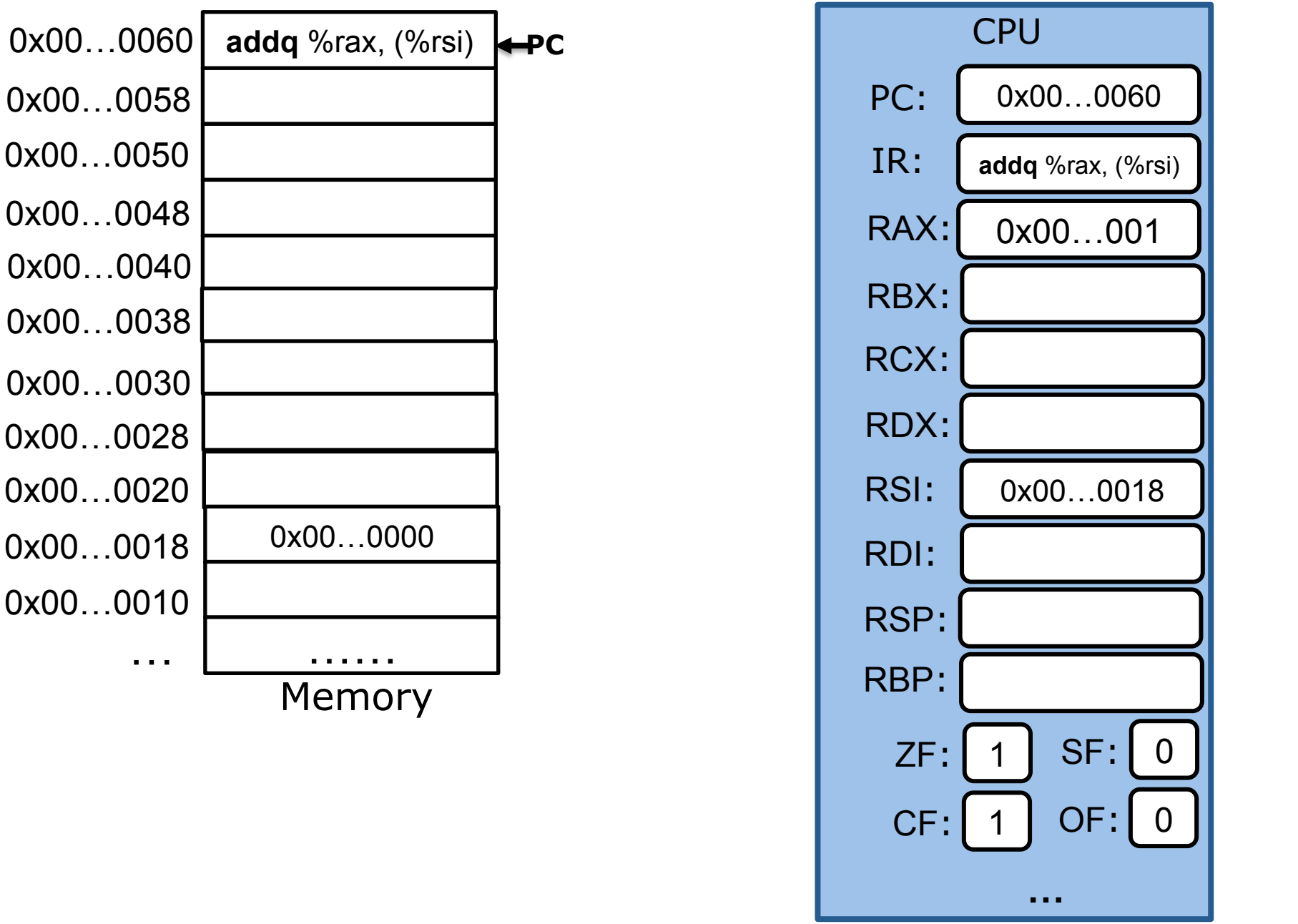
flag	status
ZF (Zero Flag)	set if the result is zero.
SF (Sign Flag)	set if the result is negative.
CF (Carry Flag)	Overflow for unsigned-integer arithmetic
OF (Overflow Flag)	Overflow for signed-integer arithmetic

Set by arithmetic instructions, e.g. add, inc, and, sal

Not set by **`lea`**, **`mov`**


0x00…0060
addq %rax, (%rsi)
PC
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0xff…ffffff
0x00…0010
…
……
Memory
CPU
PC:
0x00…0060
IR:
RAX:
0x00…001
RBX:
RCX:
RDX:
RSI:
0x00…0018
RDI:
RSP:
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…


0x00…0060
addq %rax, (%rsi)
PC
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0xff…ffffff
0x00…0010
…
……
Memory
CPU
PC:
0x00…0060
IR:
addq %rax, (%rsi)
RAX:
0x00…001
RBX:
RCX:
RDX:
RSI:
0x00…0018
RDI:
RSP:
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…


0x00…0060
addq %rax, (%rsi)
PC
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0000
0x00…0010
…
……
Memory
CPU
PC:
0x00…0060
IR:
addq %rax, (%rsi)
RAX:
0x00…001
RBX:
RCX:
RDX:
RSI:
0x00…0018
RDI:
RSP:
RBP:
ZF:
1
SF:
0
CF:
1
OF:
0
…

Exercises

src	dest	operation	ZF	SF	CF	OF
0xffffffff	0x1	addl				
0xffffffff						
0xffffffff						
0xffffffff						

Exercises

src	dest	operation	ZF	SF	CF	OF
0xffffffff	0x1	addl	1	0	1	0
0xffffffff	0x80000000	addl				
0xffffffff						
0xffffffff						

Exercises

src	dest	operation	ZF	SF	CF	OF
0xffffffff	0x1	addl	1	0	1	0
0xffffffff	0x80000000	addl	0	0	1	1
0xffffffff	0x80000000	subl				
0xffffffff						

Exercises

src	dest	operation	ZF	SF	CF	OF
0xffffffff	0x1	addl	1	0	1	0
0xffffffff	0x80000000	addl	0	0	1	1
0xffffffff	0x80000000	subl	0	1	1	0
0xffffffff	0x1	subl				

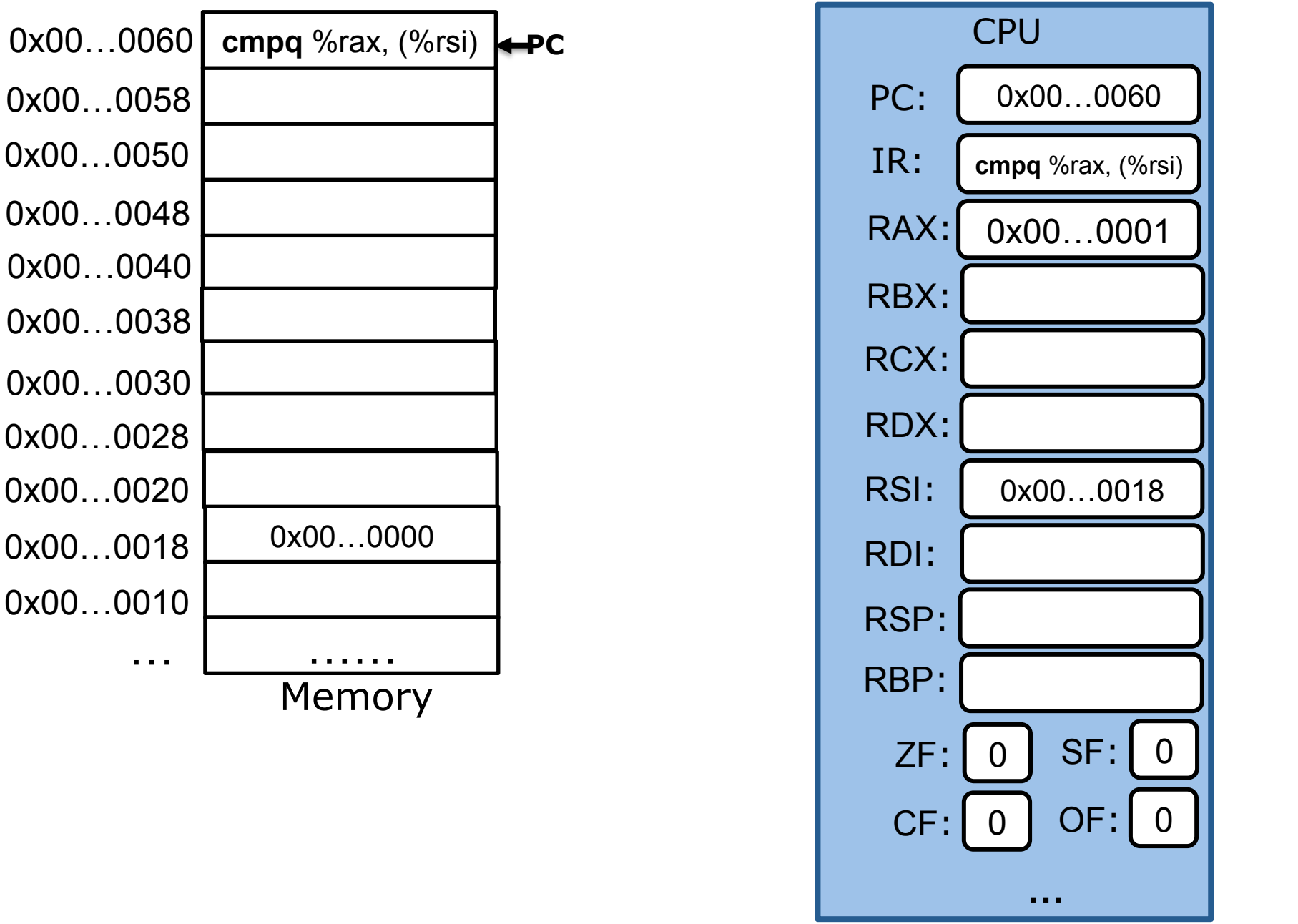
Exercises

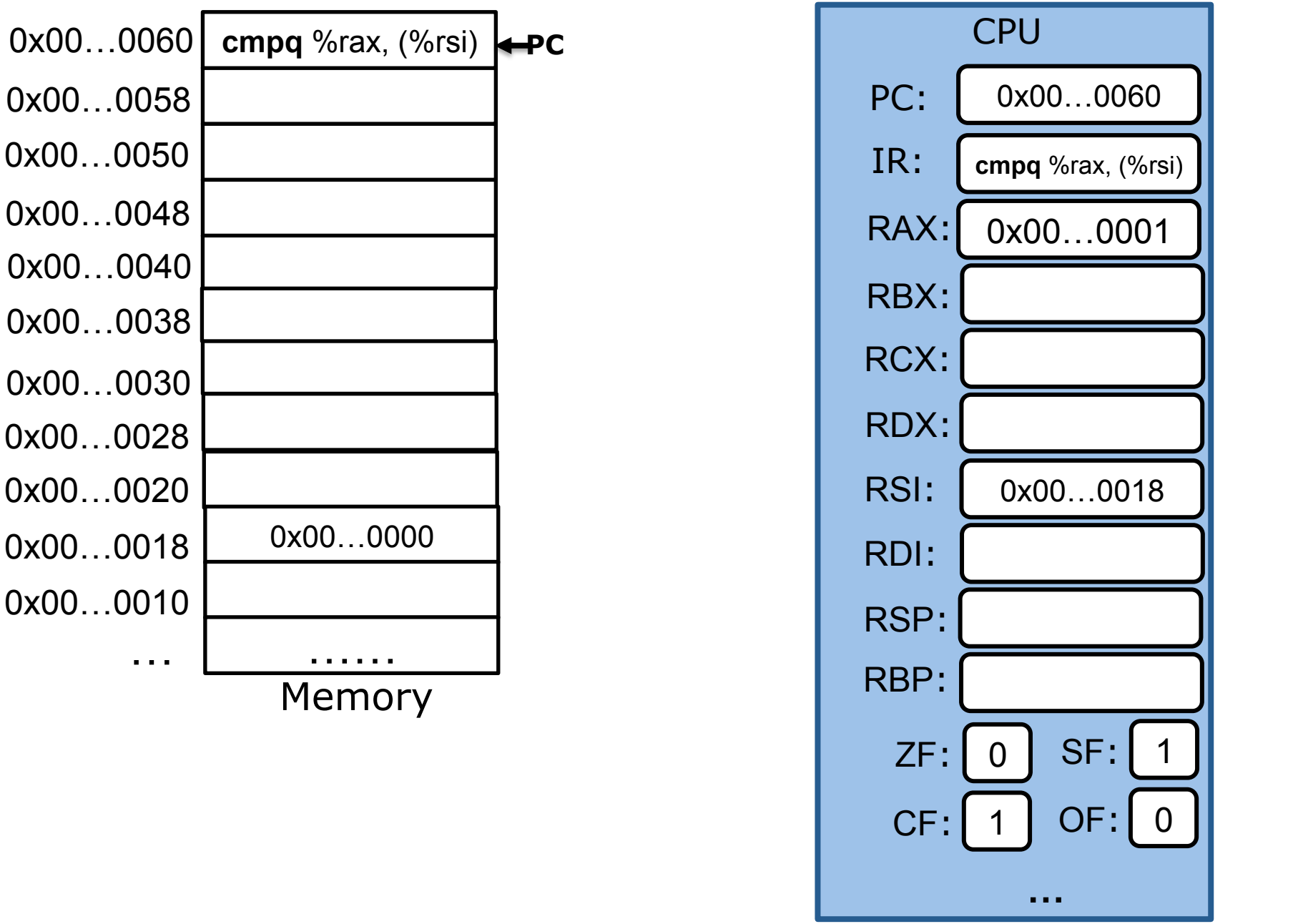
src	dest	operation	ZF	SF	CF	OF
0xffffffff	0x1	addl	1	0	1	0
0xffffffff	0x80000000	addl	0	0	1	1
0xffffffff	0x80000000	subl	0	1	1	0
0xffffffff	0x1	subl	0	0	1	0

Compare two numbers

`cmpq a, b`

- Like `subq a, b`, except destination (b) unchanged
- Set CF, ZF, SF and OF appropriately


0x00…0060
cmpq %rax, (%rsi)
PC
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0000
0x00…0010
…
……
Memory
CPU
PC:
0x00…0060
IR:
cmpq %rax, (%rsi)
RAX:
0x00…0001
RBX:
RCX:
RDX:
RSI:
0x00…0018
RDI:
RSP:
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…


0x00…0060
cmpq %rax, (%rsi)
PC
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0000
0x00…0010
…
……
Memory
CPU
PC:
0x00…0060
IR:
cmpq %rax, (%rsi)
RAX:
0x00…0001
RBX:
RCX:
RDX:
RSI:
0x00…0018
RDI:
RSP:
RBP:
ZF:
0
SF:
1
CF:
1
OF:
0
…

Exercises

```
cmpq $0x10, %rax
```

%rax	ZF	SF	CF	OF
0x10				
0x20				
0x0				
0x8000000000000000				

Exercises

```
cmpq $0x10, %rax
```

%rax	ZF	SF	CF	OF
0x10	1	0	0`	0
0x20				
0x0				
0x8000000000000000				

Exercises

```
cmpq $0x10, %rax
```

%rax	ZF	SF	CF	OF
0x10	1	0	0`	0
0x20	0	0	0	0
0x0				
0x8000000000000000				

Exercises

```
cmpq $0x10, %rax
```

%rax	ZF	SF	CF	OF
0x10	1	0	0`	0
0x20	0	0	0	0
0x0	0	1	1	0
0x8000000000000000				

Exercises

```
cmpq $0x10, %rax
```

%rax	ZF	SF	CF	OF
0x10	1	0	0`	0
0x20	0	0	0	0
0x0	0	1	1	0
0x8000000000000000	0	0	0	1

Test: logical compare

`testq` `a, b`

- Like `andq a, b`, except destination(b) unchanged
- Set ZF, SF appropriately

Questions

testq %rax, %rax

- When is ZF set?
- When is SF set?

Questions

testq %rax, %rax

- When is ZF set? 0x0
- When is SF set? val(%rax) < 0

Read status flags

setX dest

- set dest to 0 or 1 depending on the status flag (CF, SF, OF and ZF) in the EFLAGS register.
- dest is either a (1-byte) register or a byte in memory.
- Condition code suffix (X) indicates the condition being tested for.

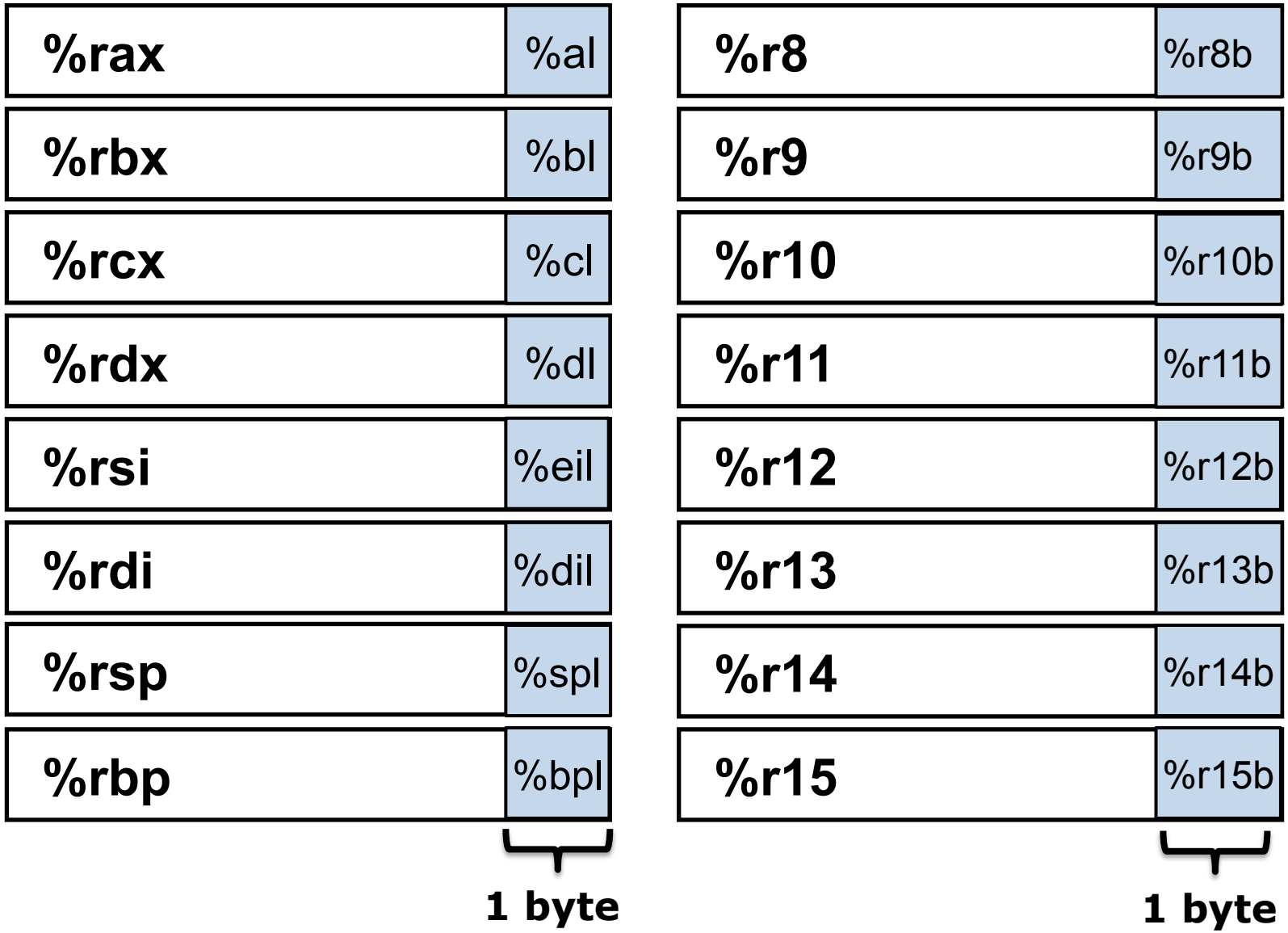
setX dest

```
cmpq a, b
setX c
```

setX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

Dest is greater than source (aka b is greater than a)

1 byte register

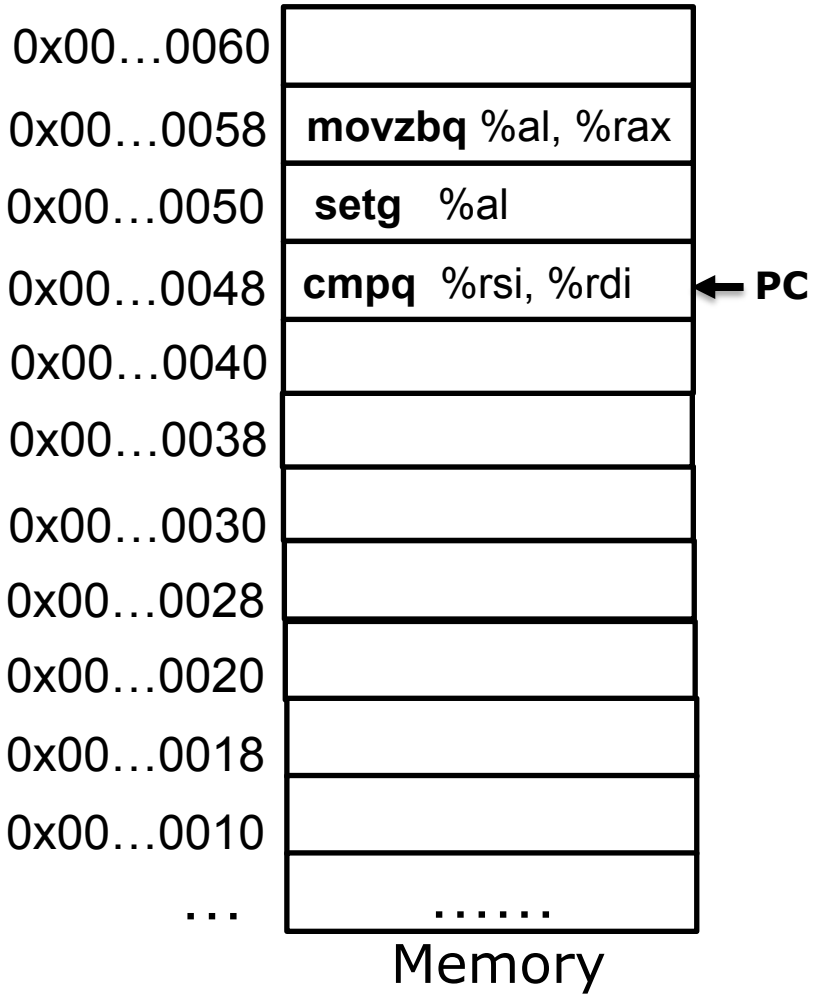


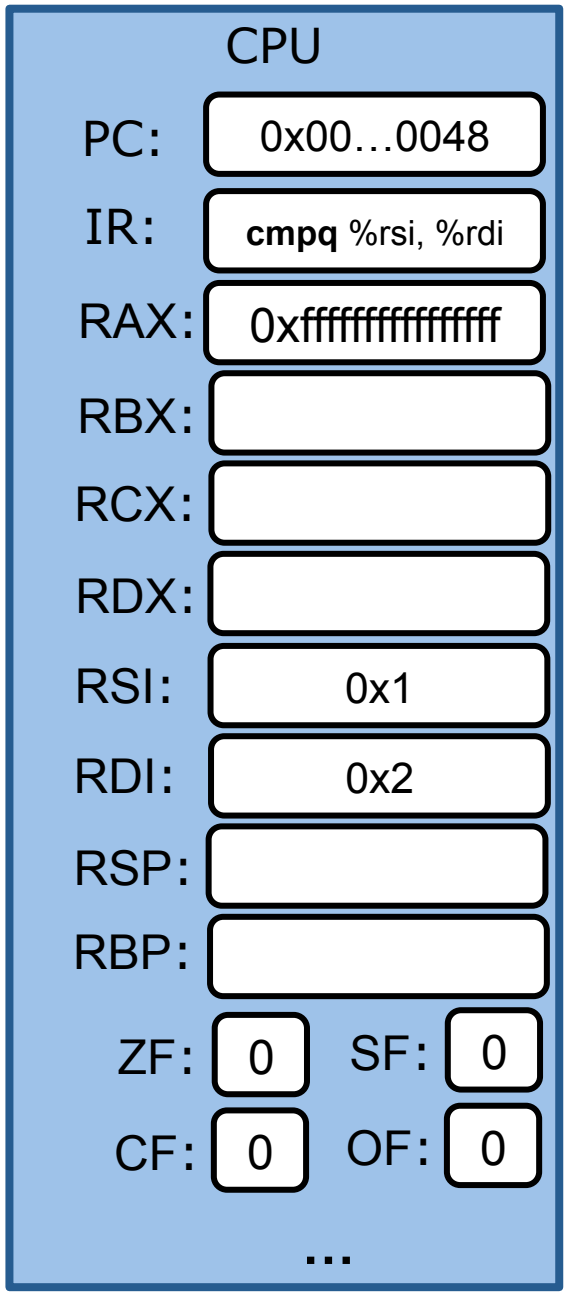
Example

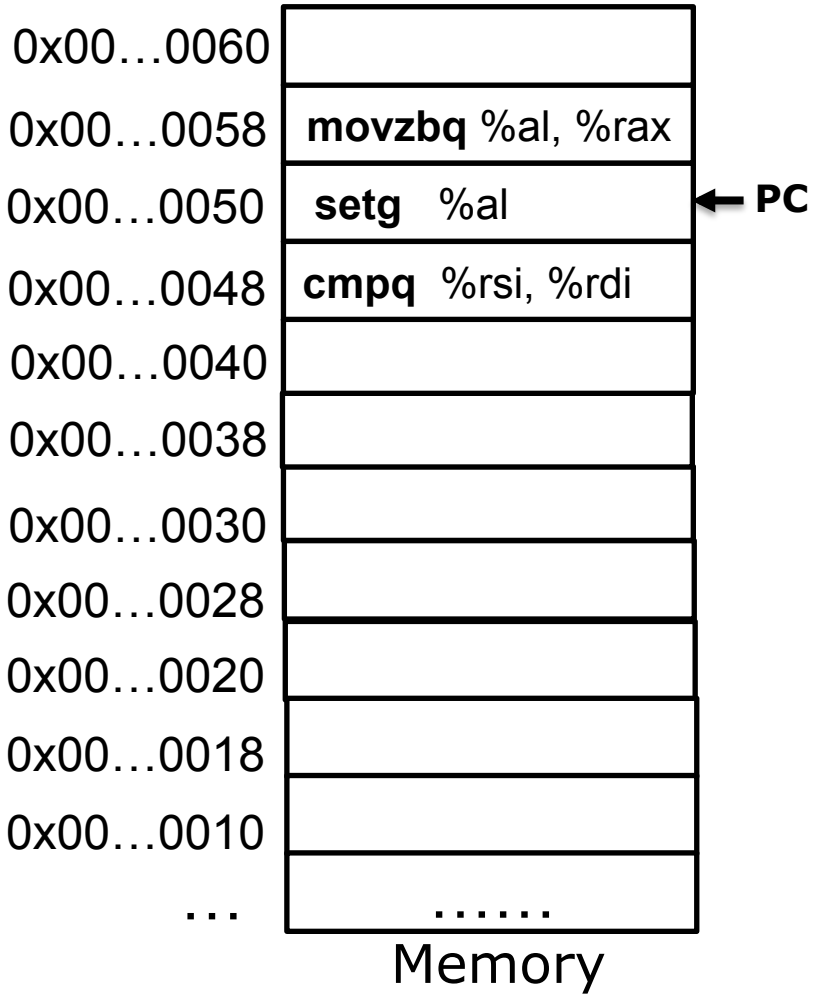
```
int gt (long x, long y)
{
  return x > y;
}
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

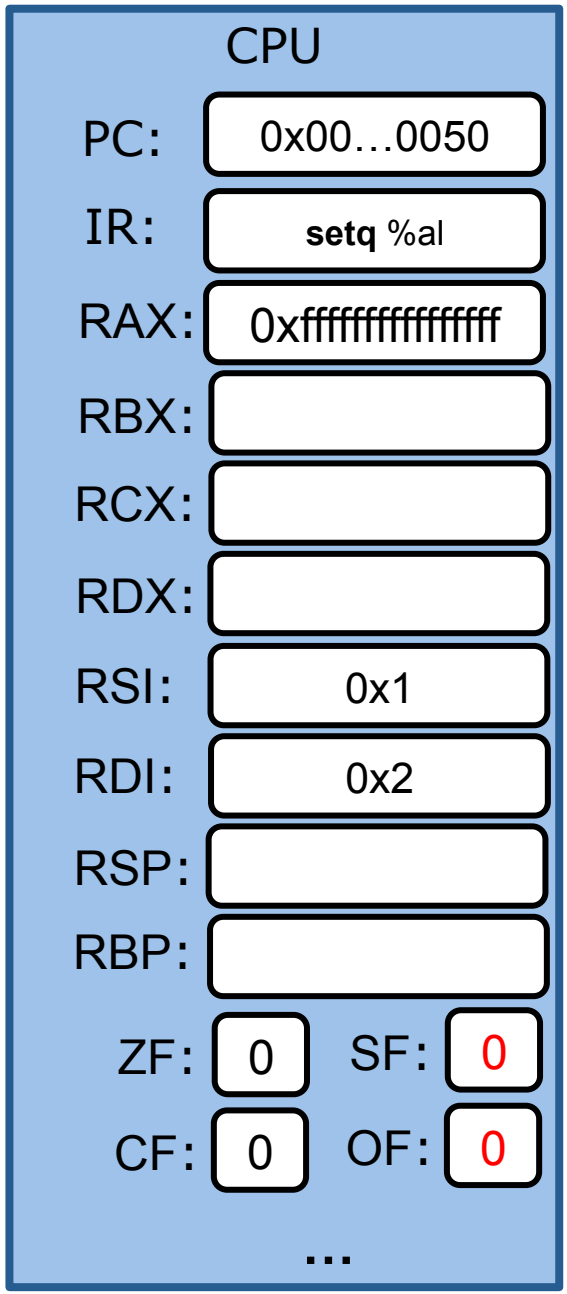
```
cmpq   %rsi, %rdi    # cmpq y x
setg   %al           # set when >
movzbq %al, %rax     # zero extend %rax
```

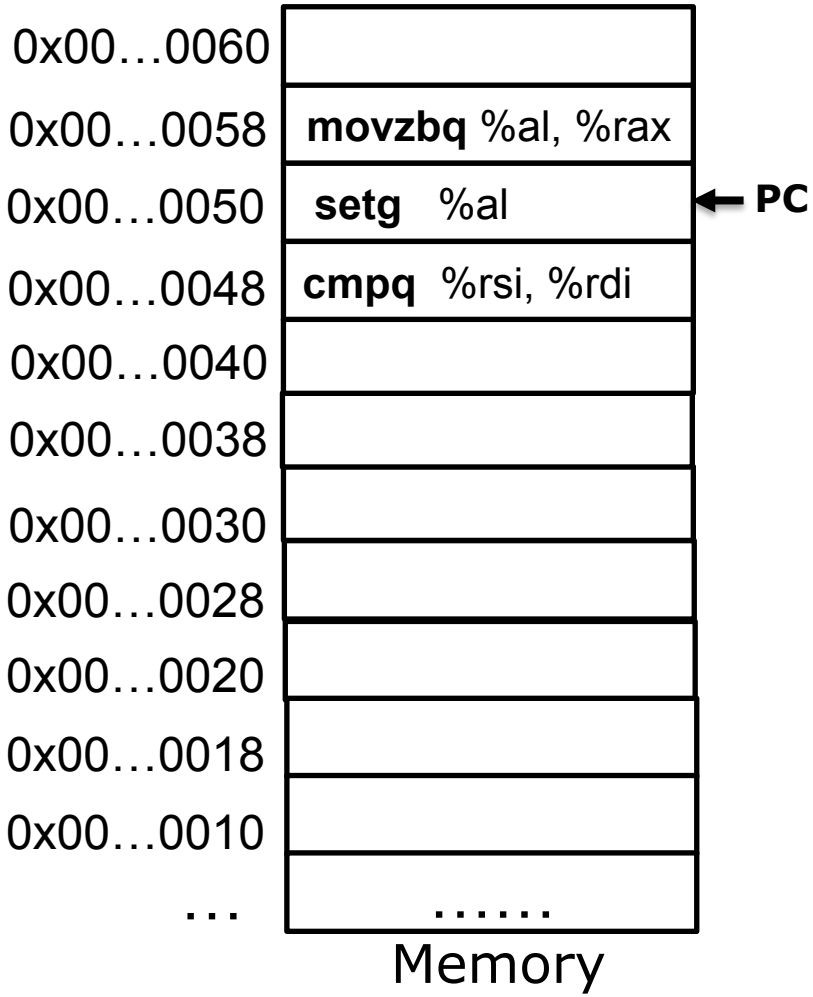

0x00…0060
0x00…0058
movzbq %al, %rax
0x00…0050
setg %al
0x00…0048
cmpq %rsi, %rdi
PC
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory


CPU
PC:
0x00…0048
IR:
cmpq %rsi, %rdi
RAX:
0xffffffffffffffff
RBX:
RCX:
RDX:
RSI:
0x1
RDI:
0x2
RSP:
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…

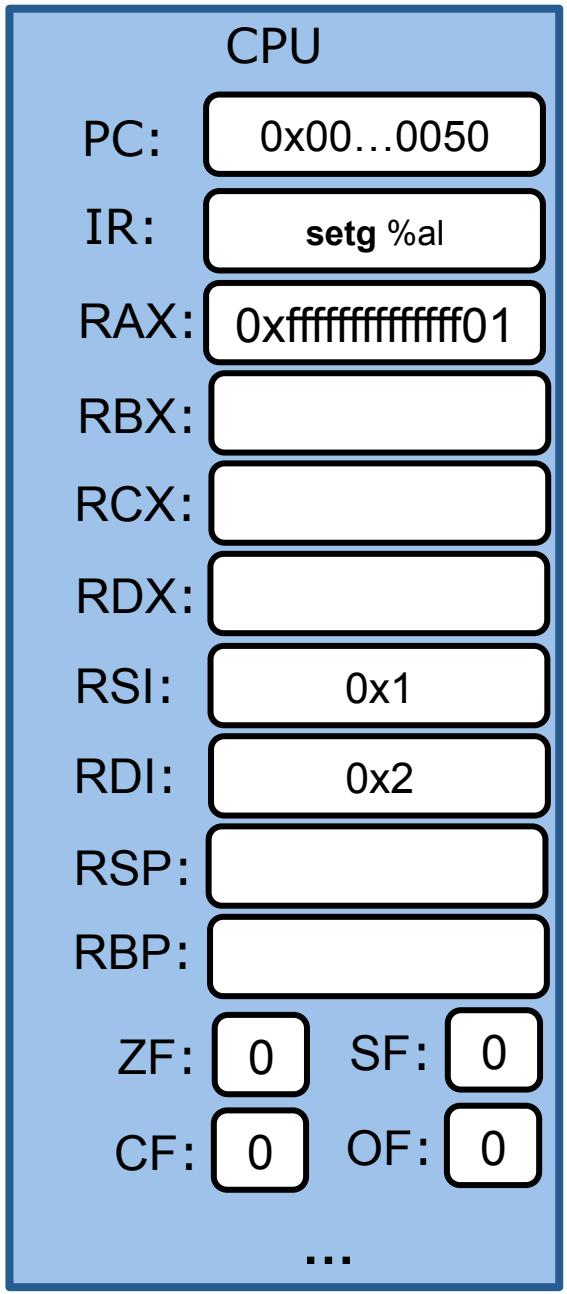

0x00…0060
0x00…0058
movzbq %al, %rax
0x00…0050
setg %al
PC
0x00…0048
cmpq %rsi, %rdi
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory

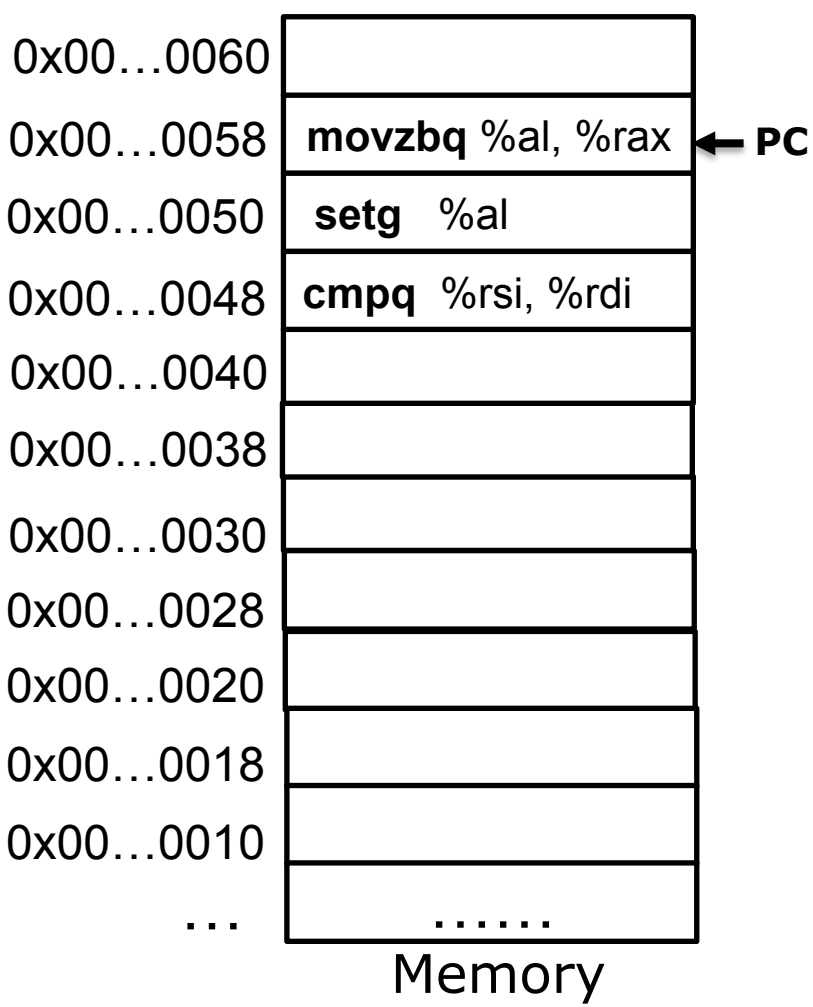
setg	~(SF^OF)&~ZF

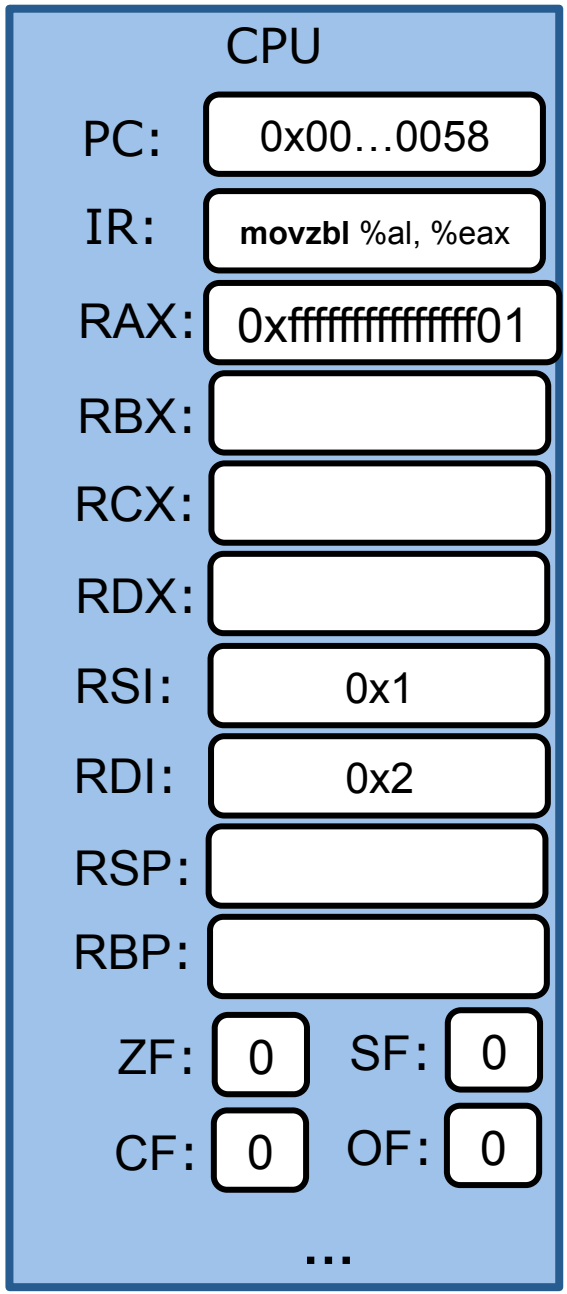

CPU
PC:
0x00…0050
IR:
setq %al
RAX:
0xffffffffffffffff
RBX:
RCX:
RDX:
RSI:
0x1
RDI:
0x2
RSP:
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…

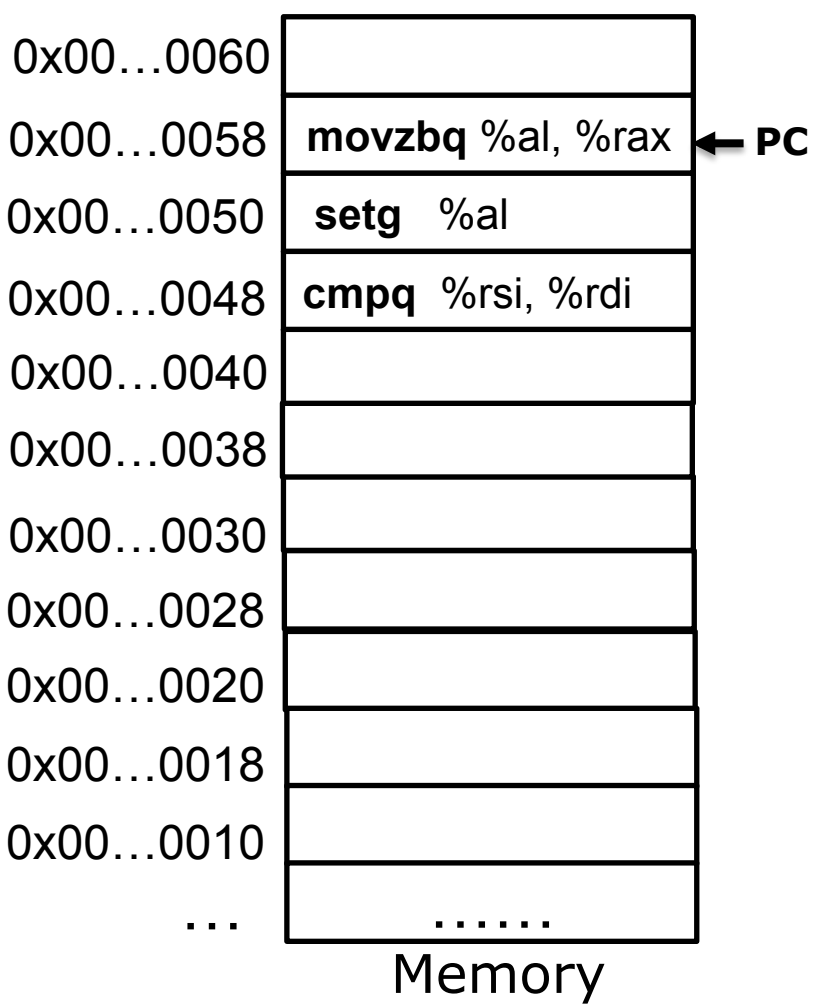

0x00…0060
0x00…0058
movzbq %al, %rax
0x00…0050
setg %al
PC
0x00…0048
cmpq %rsi, %rdi
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory

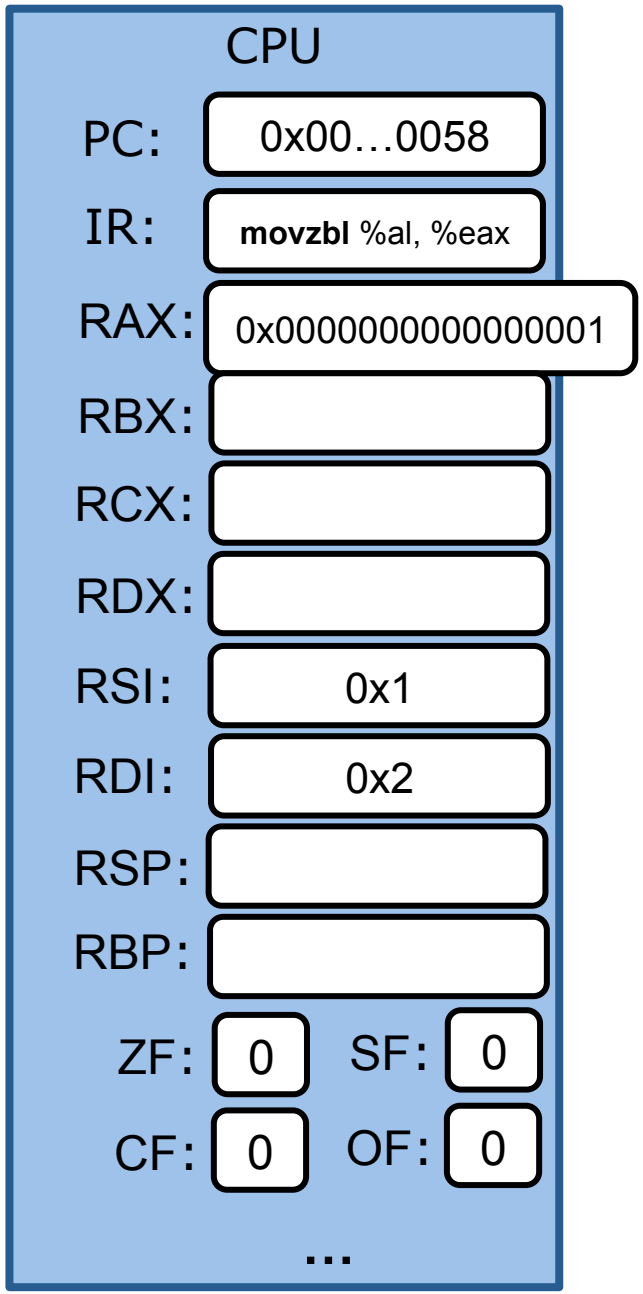
setg	~(SF^OF)&~ZF


CPU
PC:
0x00…0050
IR:
setg %al
RAX:
0xffffffffffffff01
RBX:
RCX:
RDX:
RSI:
0x1
RDI:
0x2
RSP:
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…


0x00…0060
0x00…0058
movzbq %al, %rax
PC
0x00…0050
setg %al
0x00…0048
cmpq %rsi, %rdi
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory


CPU
PC:
0x00…0058
IR:
movzbl %al, %eax
RAX:
0xffffffffffffff01
RBX:
RCX:
RDX:
RSI:
0x1
RDI:
0x2
RSP:
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…


0x00…0060
0x00…0058
movzbq %al, %rax
PC
0x00…0050
setg %al
0x00…0048
cmpq %rsi, %rdi
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory

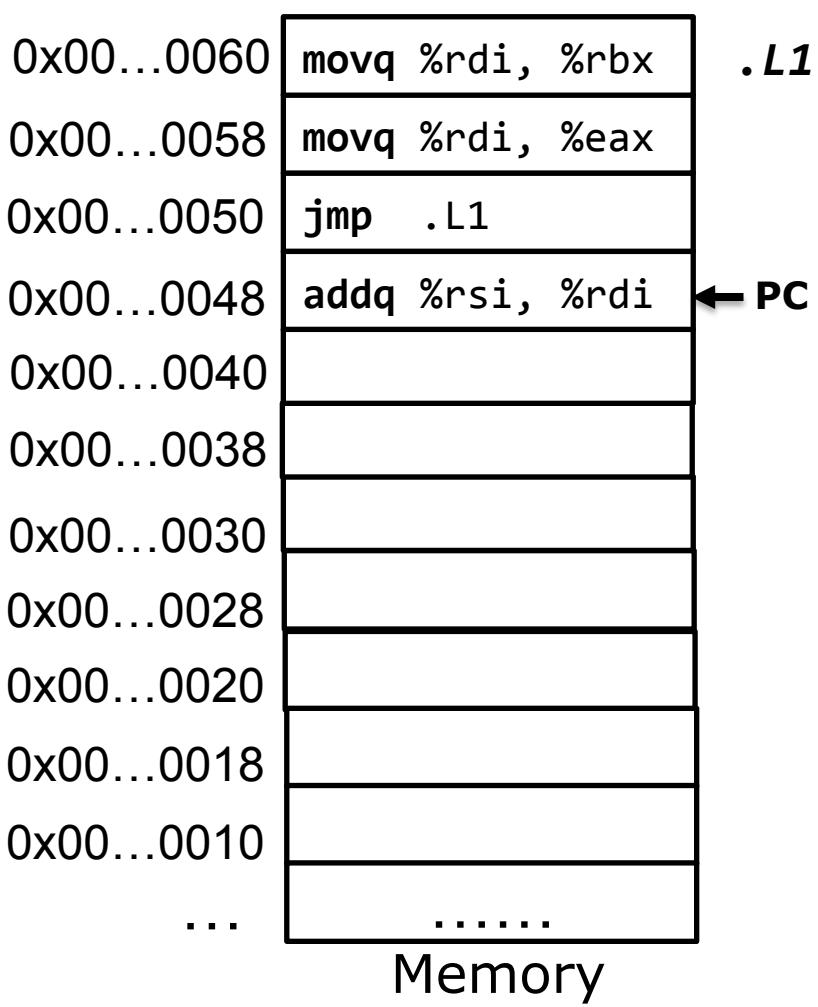

CPU
PC:
0x00…0058
IR:
movzbl %al, %eax
RAX:
0x0000000000000001
RBX:
RCX:
RDX:
RSI:
0x1
RDI:
0x2
RSP:
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…

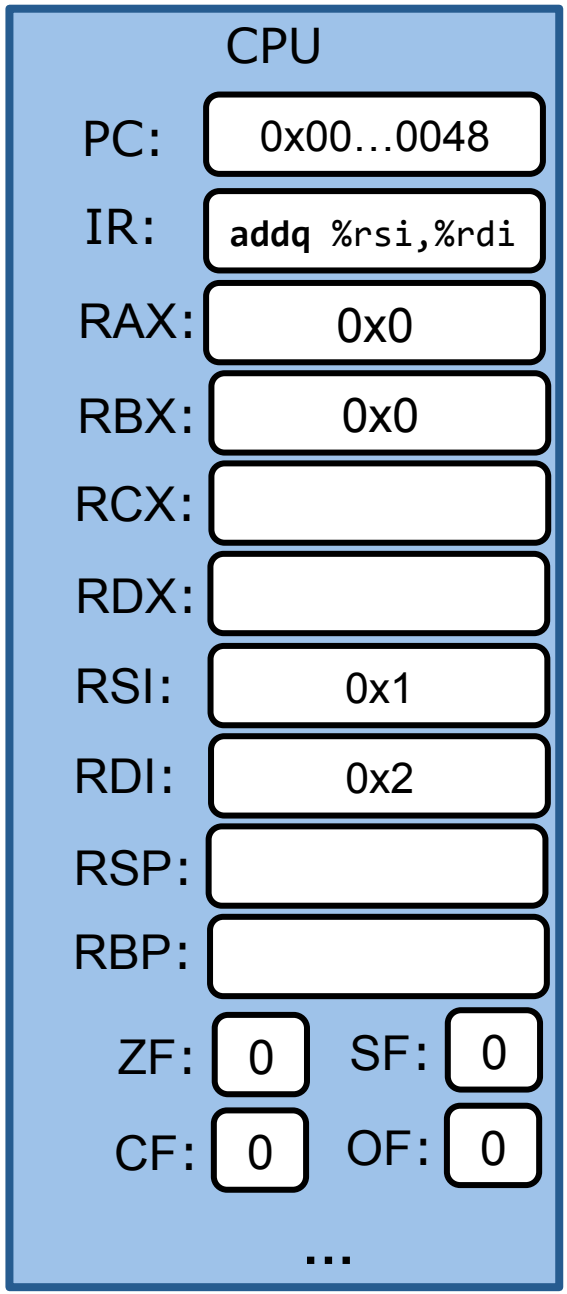
Jump instruction

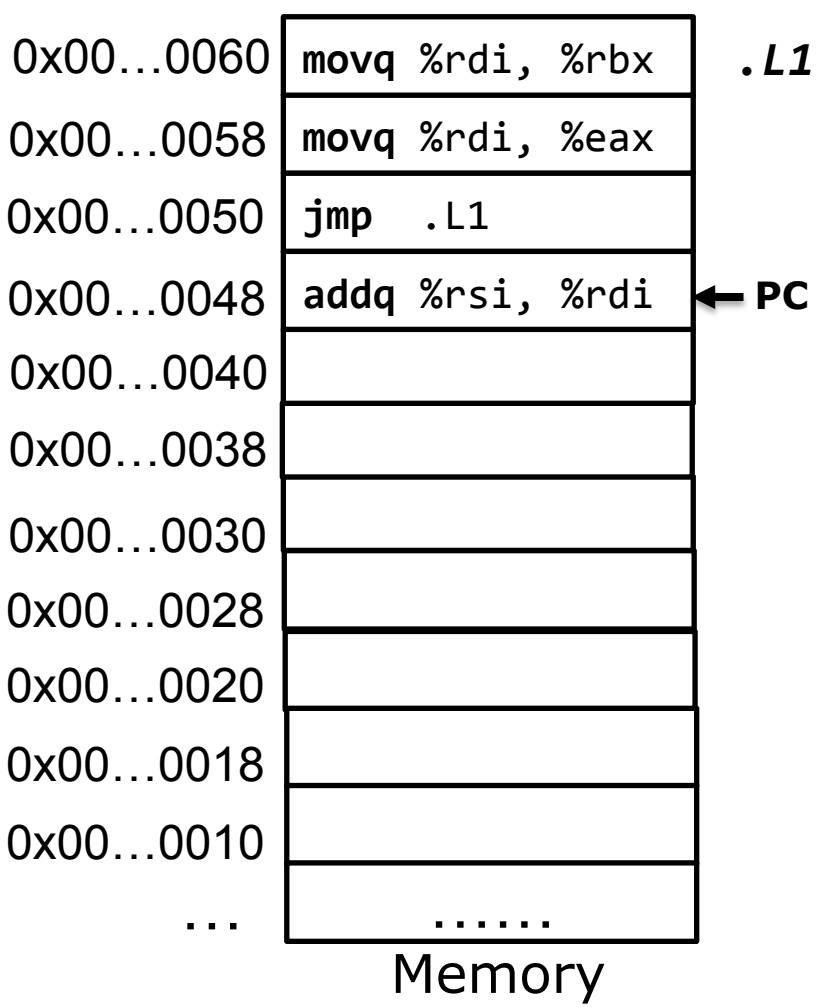
`jmp label`

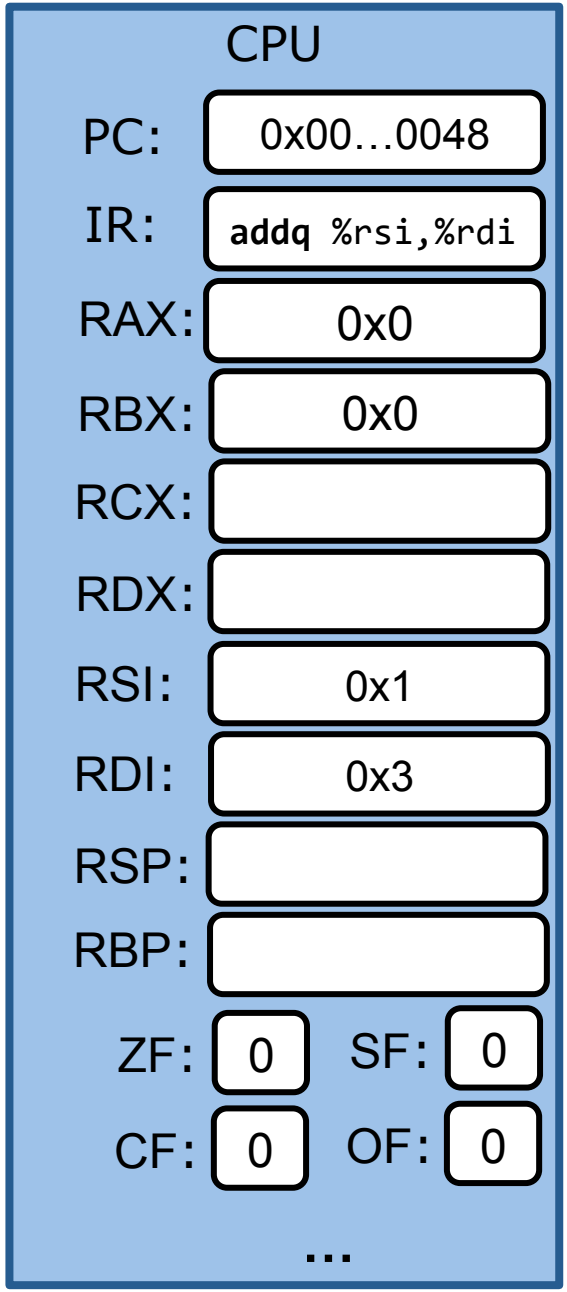
- Transfer control to a different point in the instruction stream by changing %rip
- Label specifies the address to jump to
- jmp is like ***goto***

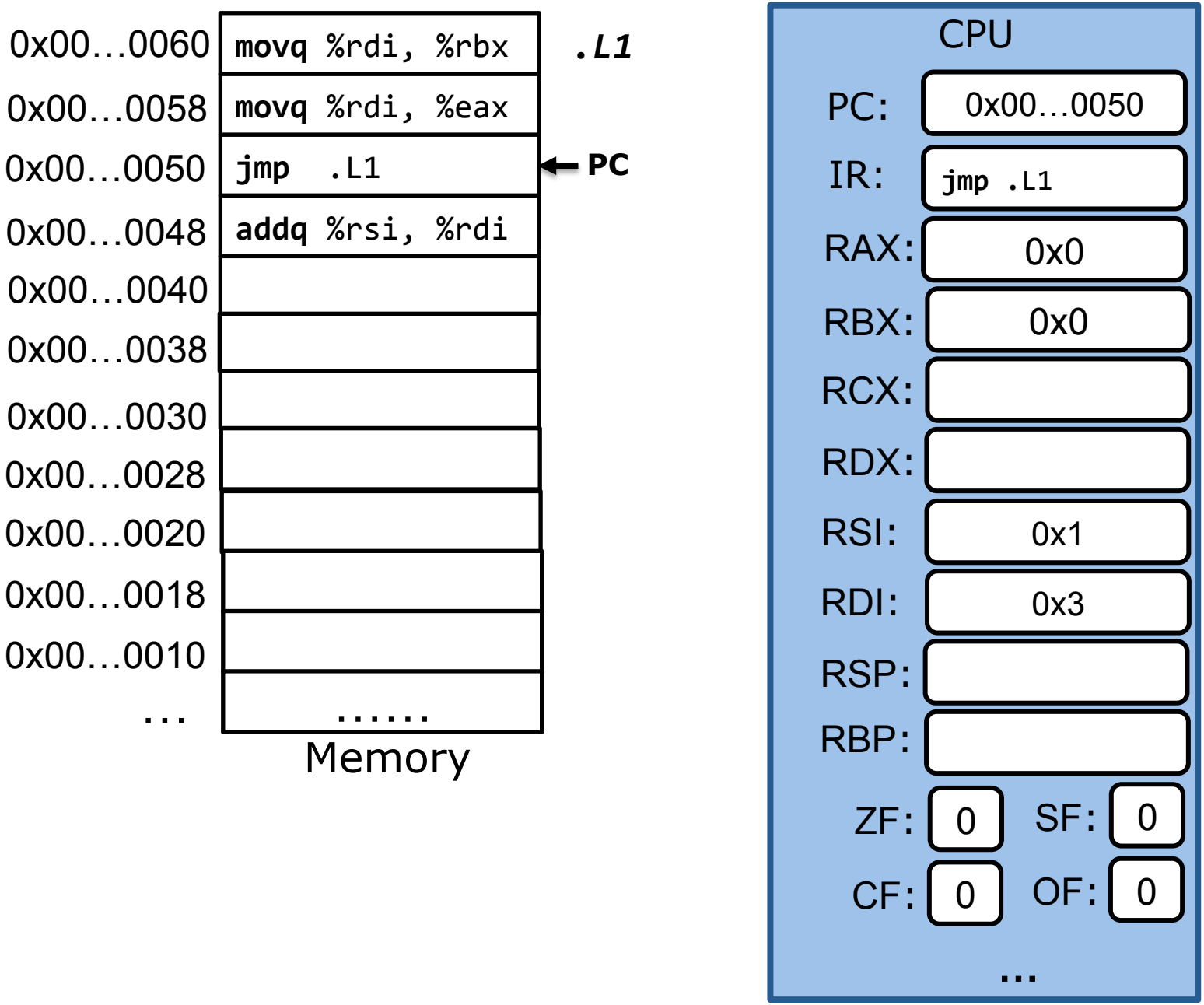
```
  addq %rsi, %rdi
  jmp  .L1
  movq %rdi, %eax
.L1
  movq %rdi, %rbx
```

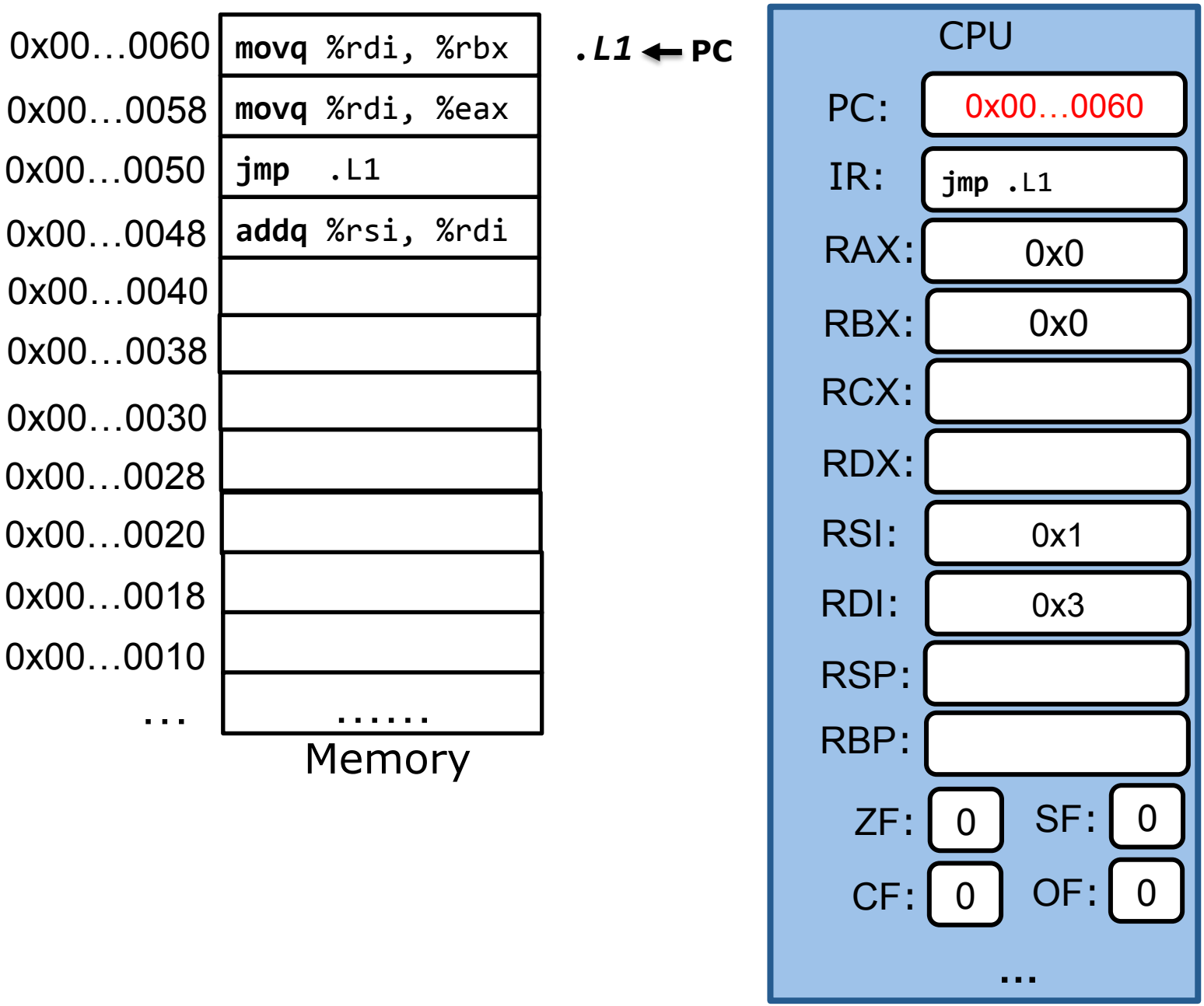

0x00…0060
movq %rdi, %rbx
.L1
0x00…0058
movq %rdi, %eax
0x00…0050
jmp .L1
0x00…0048
addq %rsi, %rdi
PC
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory

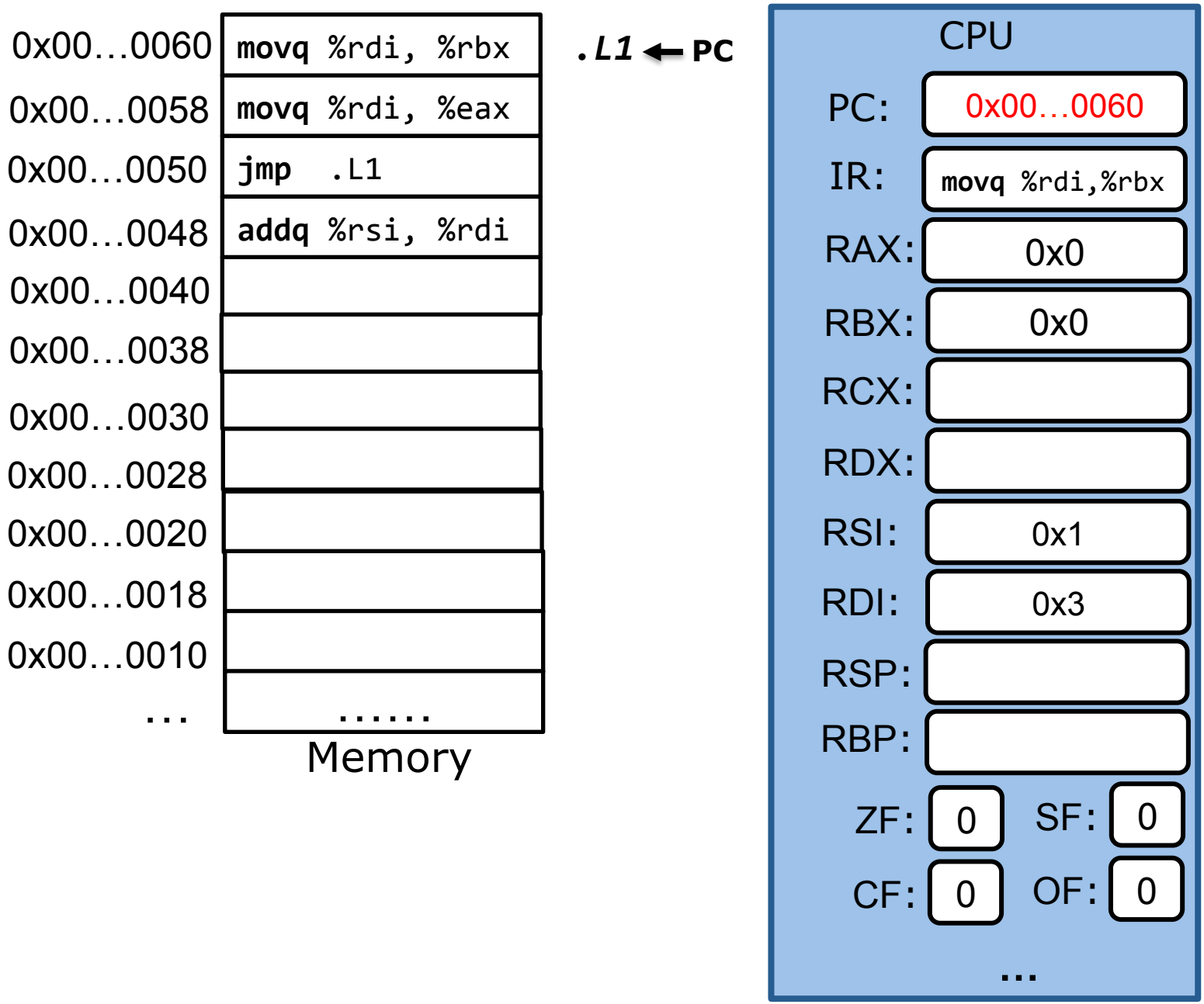

CPU
PC:
0x00…0048
IR:
addq %rsi,%rdi
RAX:
0x0
RBX:
0x0
RCX:
RDX:
RSI:
0x1
RDI:
0x2
RSP:
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…

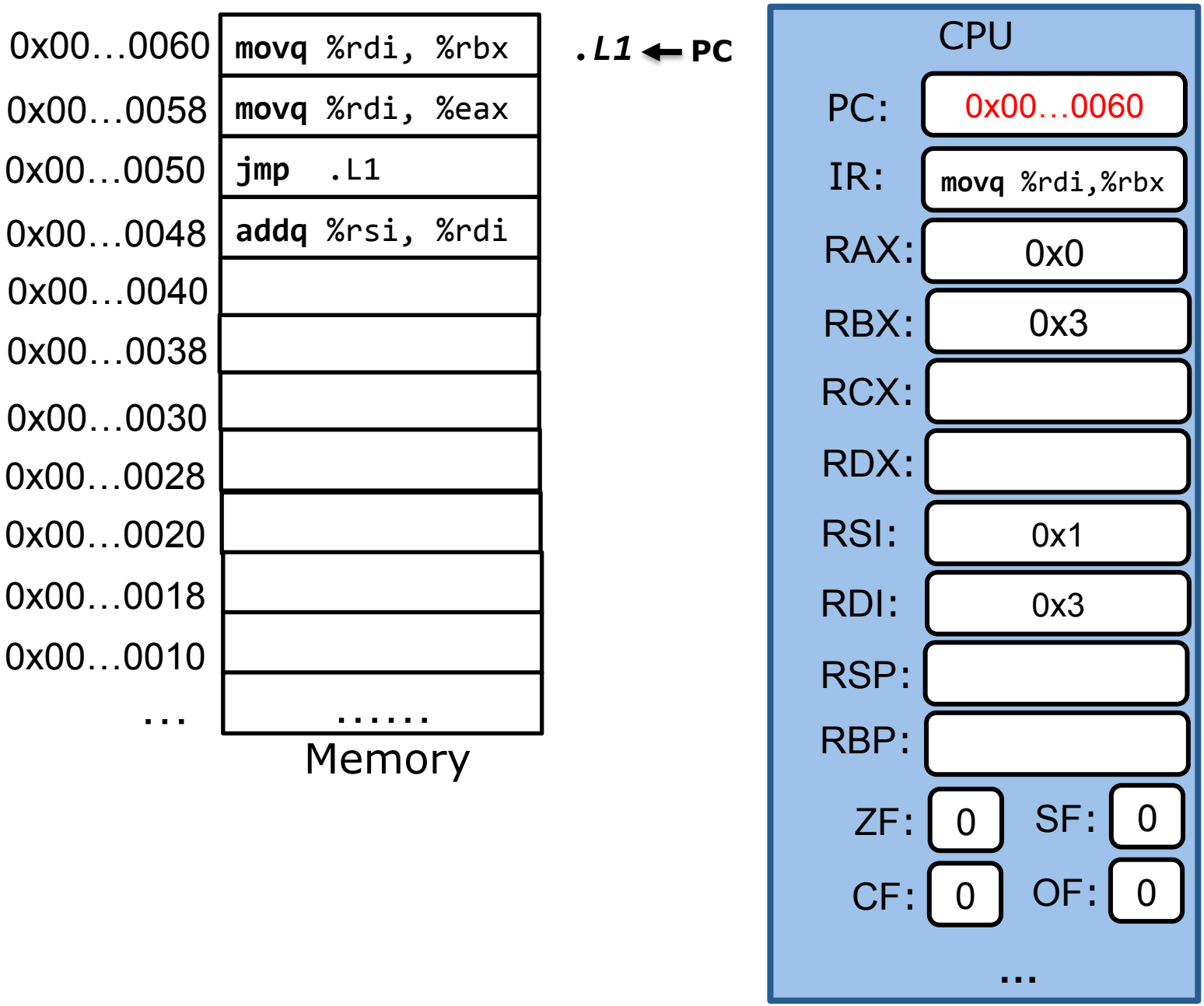

0x00…0060
movq %rdi, %rbx
.L1
0x00…0058
movq %rdi, %eax
0x00…0050
jmp .L1
0x00…0048
addq %rsi, %rdi
PC
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory


CPU
PC:
0x00…0048
IR:
addq %rsi,%rdi
RAX:
0x0
RBX:
0x0
RCX:
RDX:
RSI:
0x1
RDI:
0x3
RSP:
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…


0x00…0060
movq %rdi, %rbx
.L1
0x00…0058
movq %rdi, %eax
0x00…0050
jmp .L1
PC
0x00…0048
addq %rsi, %rdi
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
… ……
Memory
CPU
PC: 0x00…0050
IR: jmp .L1
RAX: 0x0
RBX: 0x0
RCX:
RDX:
RSI: 0x1
RDI: 0x3
RSP:
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
…


0x00…0060
movq %rdi, %rbx
0x00…0058
movq %rdi, %eax
0x00…0050
jmp .L1
0x00…0048
addq %rsi, %rdi
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
… ……
Memory
.L1 ← PC
CPU
PC: 0x00…0060
IR: jmp .L1
RAX: 0x0
RBX: 0x0
RCX:
RDX:
RSI: 0x1
RDI: 0x3
RSP:
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
…


0x00…0060
movq %rdi, %rbx
0x00…0058
movq %rdi, %eax
0x00…0050
jmp .L1
0x00…0048
addq %rsi, %rdi
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory
.L1 ← PC
CPU
PC:
0x00…0060
IR:
movq %rdi,%rbx
RAX:
0x0
RBX:
0x0
RCX:
RDX:
RSI:
0x1
RDI:
0x3
RSP:
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…

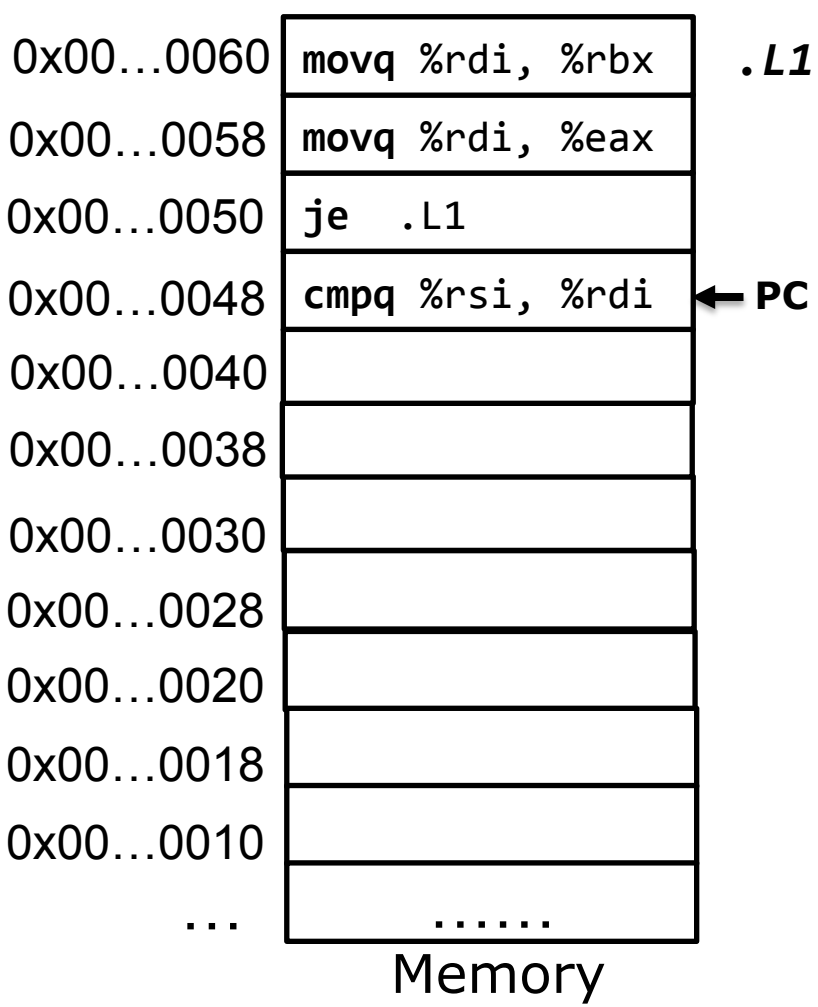

0x00…0060
movq %rdi, %rbx
0x00…0058
movq %rdi, %eax
0x00…0050
jmp .L1
0x00…0048
addq %rsi, %rdi
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory
.L1 ← PC
CPU
PC:
0x00…0060
IR:
movq %rdi,%rbx
RAX:
0x0
RBX:
0x3
RCX:
RDX:
RSI:
0x1
RDI:
0x3
RSP:
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…

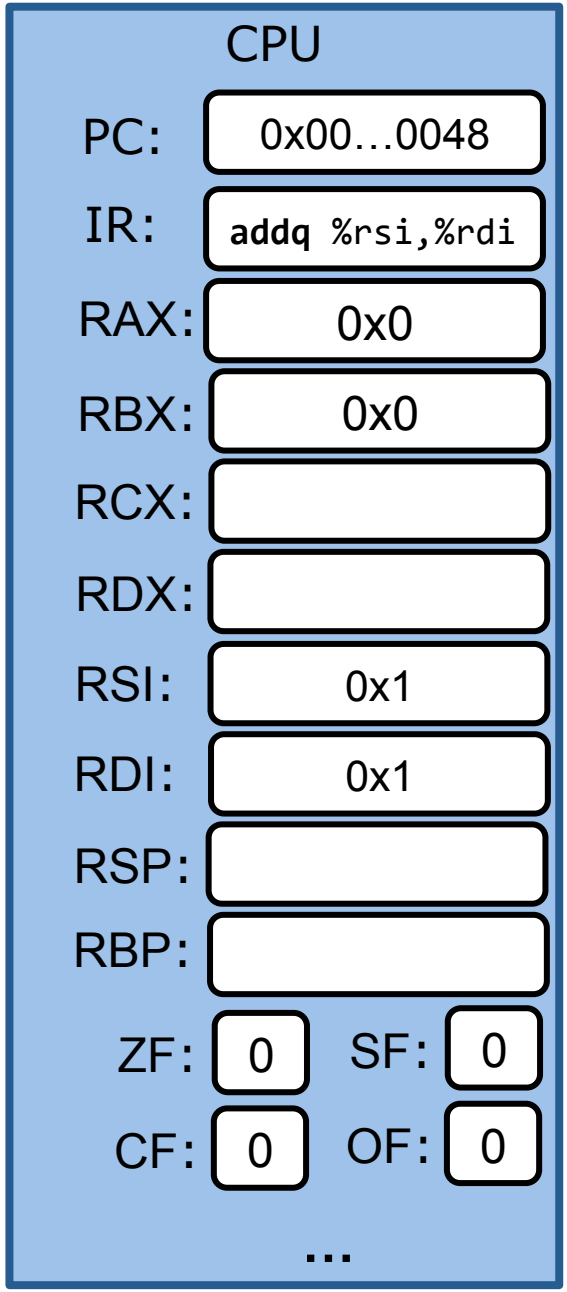
Jump instruction

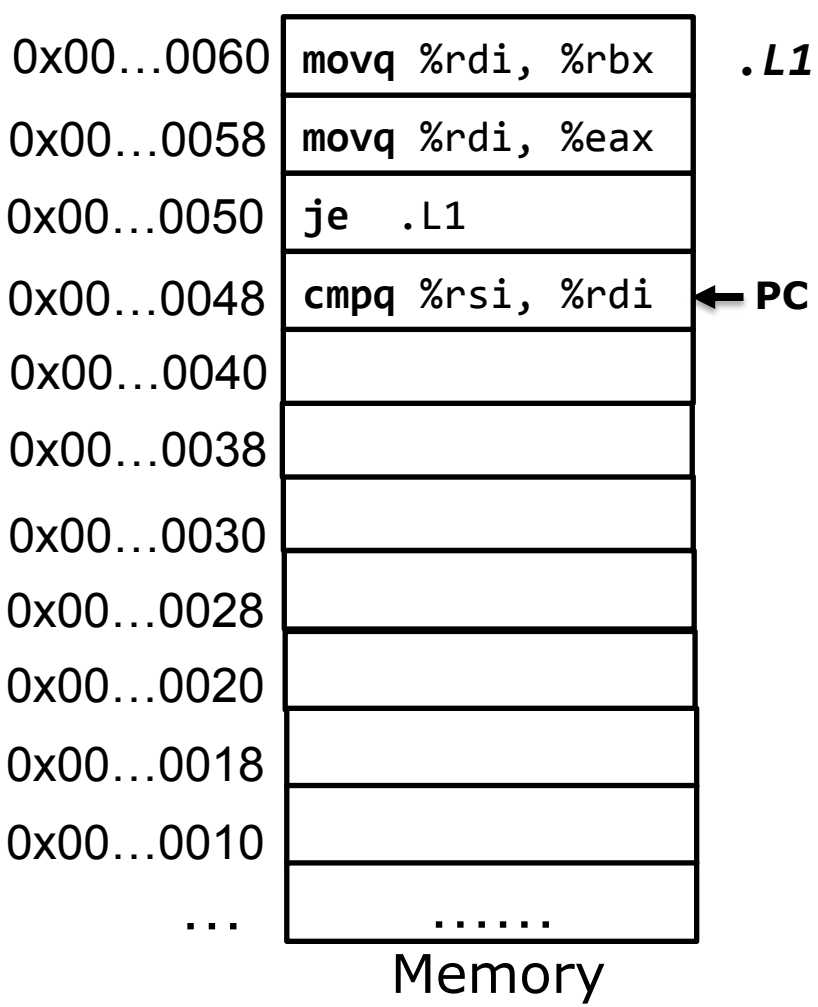
`jX label`

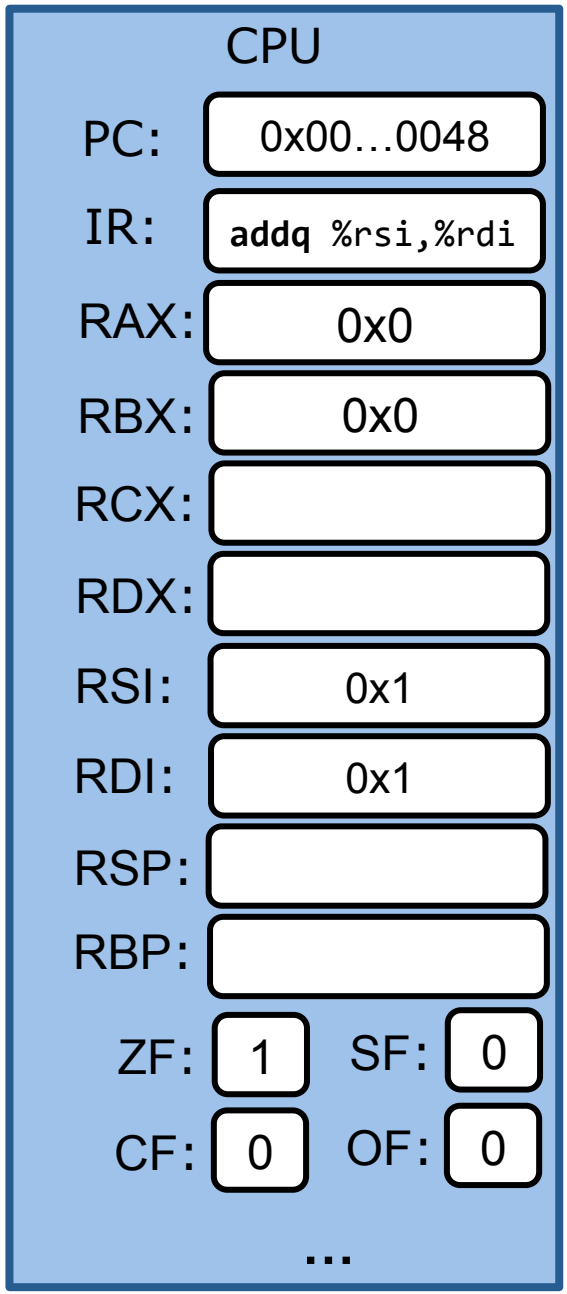
– If condition **X** is met, jump to the label

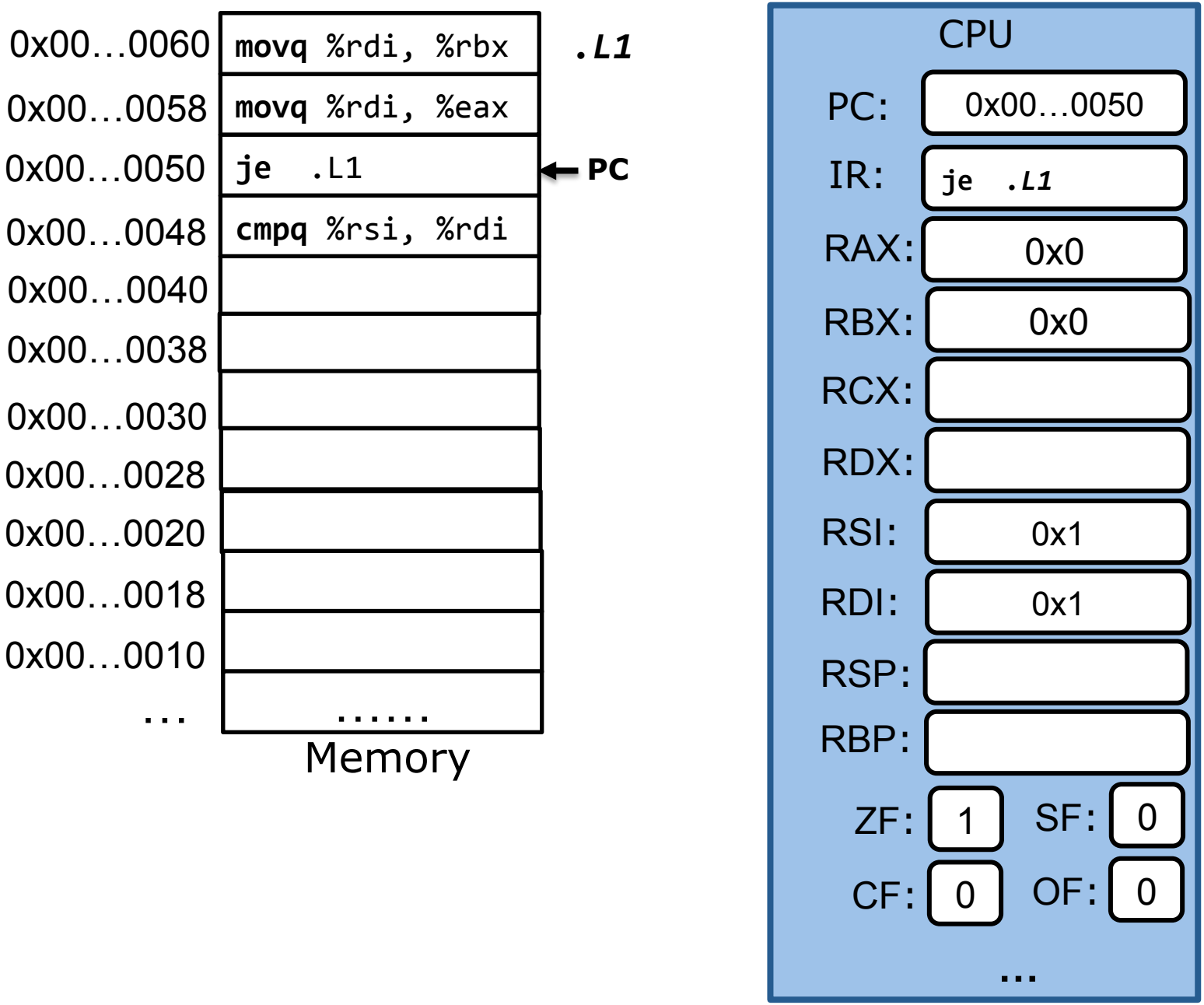
jX	Condition	Description
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

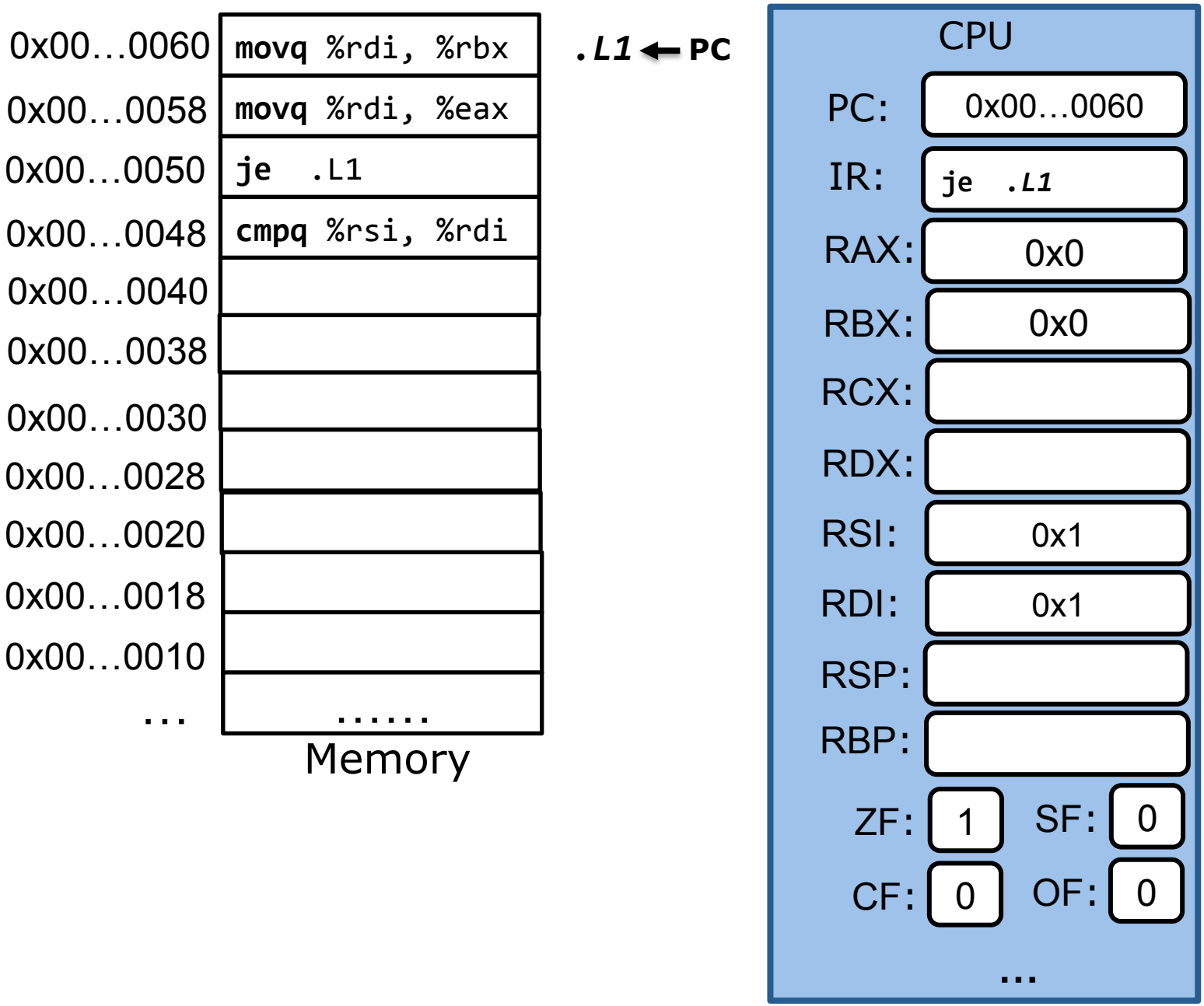

0x00…0060
movq %rdi, %rbx
.L1
0x00…0058
movq %rdi, %eax
0x00…0050
je .L1
0x00…0048
cmpq %rsi, %rdi
PC
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory

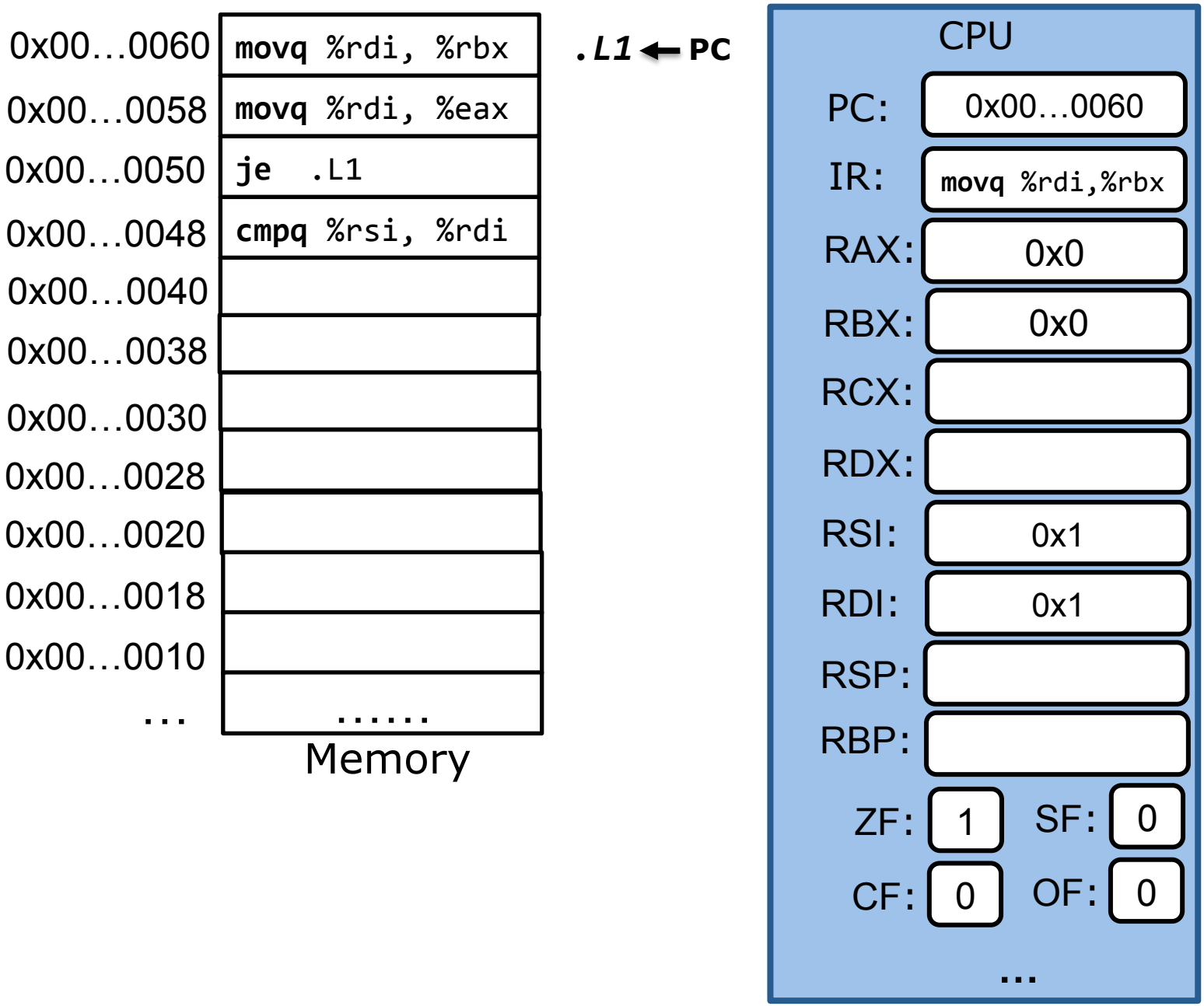

CPU
PC:
0x00…0048
IR:
addq %rsi,%rdi
RAX:
0x0
RBX:
0x0
RCX:
RDX:
RSI:
0x1
RDI:
0x1
RSP:
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…

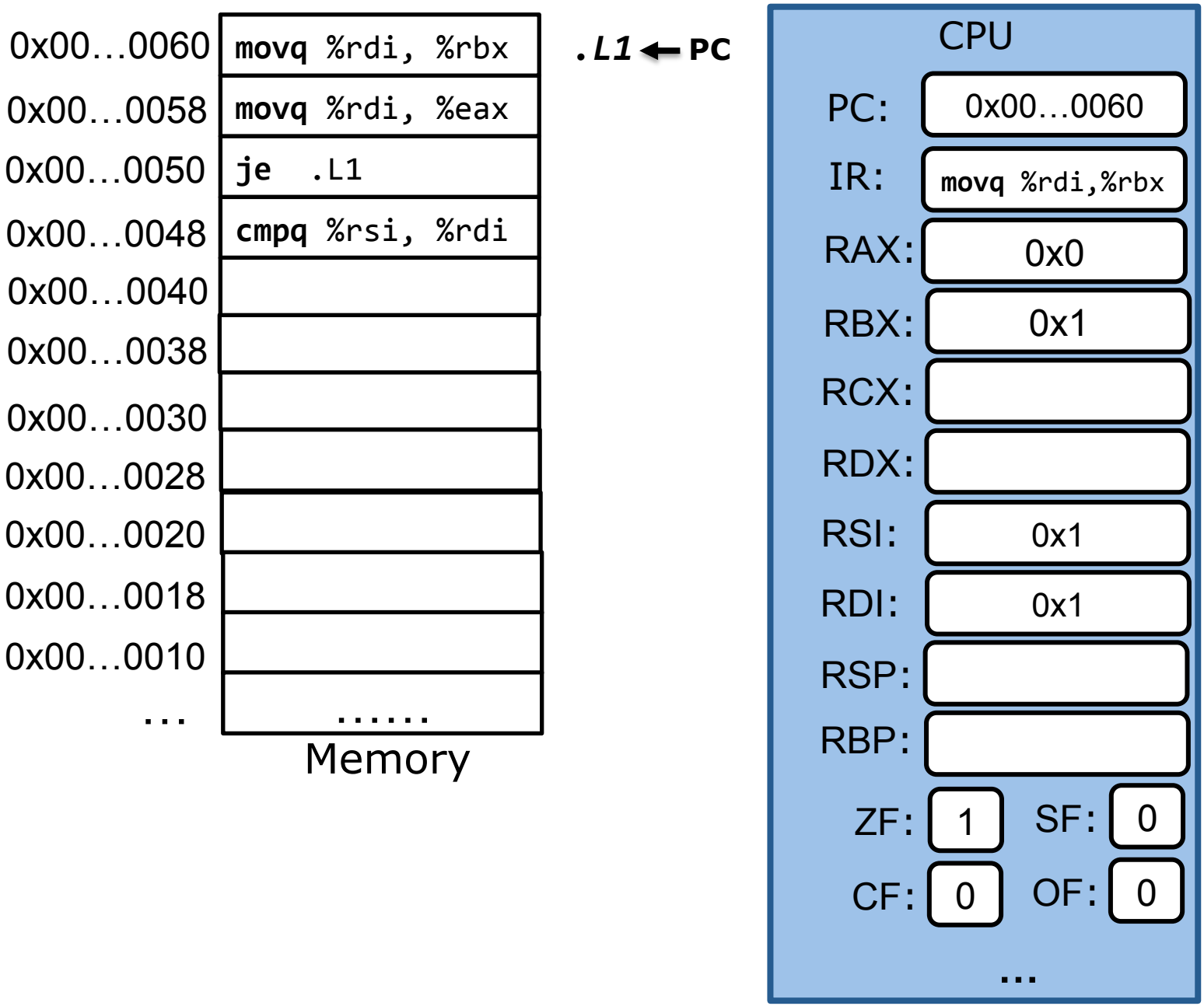

0x00…0060
movq %rdi, %rbx
.L1
0x00…0058
movq %rdi, %eax
0x00…0050
je .L1
0x00…0048
cmpq %rsi, %rdi
PC
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory


CPU
PC:
0x00…0048
IR:
addq %rsi,%rdi
RAX:
0x0
RBX:
0x0
RCX:
RDX:
RSI:
0x1
RDI:
0x1
RSP:
RBP:
ZF:
1
SF:
0
CF:
0
OF:
0
…


0x00…0060
movq %rdi, %rbx
.L1
0x00…0058
movq %rdi, %eax
0x00…0050
je .L1
PC
0x00…0048
cmpq %rsi, %rdi
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory
CPU
PC:
0x00…0050
IR:
je .L1
RAX:
0x0
RBX:
0x0
RCX:
RDX:
RSI:
0x1
RDI:
0x1
RSP:
RBP:
ZF:
1
SF:
0
CF:
0
OF:
0
…


0x00…0060
movq %rdi, %rbx
0x00…0058
movq %rdi, %eax
0x00…0050
je .L1
0x00…0048
cmpq %rsi, %rdi
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory
.L1 ← PC
CPU
PC: 0x00…0060
IR: je .L1
RAX: 0x0
RBX: 0x0
RCX:
RDX:
RSI: 0x1
RDI: 0x1
RSP:
RBP:
ZF: 1
SF: 0
CF: 0
OF: 0
…


0x00…0060
movq %rdi, %rbx
0x00…0058
movq %rdi, %eax
0x00…0050
je .L1
0x00…0048
cmpq %rsi, %rdi
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory
.L1 ← PC
CPU
PC:
0x00…0060
IR:
movq %rdi,%rbx
RAX:
0x0
RBX:
0x0
RCX:
RDX:
RSI:
0x1
RDI:
0x1
RSP:
RBP:
ZF:
1
SF:
0
CF:
0
OF:
0
…


0x00…0060
movq %rdi, %rbx
0x00…0058
movq %rdi, %eax
0x00…0050
je .L1
0x00…0048
cmpq %rsi, %rdi
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
……
Memory
.L1 ← PC
CPU
PC: 0x00…0060
IR: movq %rdi,%rbx
RAX: 0x0
RBX: 0x1
RCX:
RDX:
RSI: 0x1
RDI: 0x1
RSP:
RBP:
ZF: 1
SF: 0
CF: 0
OF: 0
…

Conditional Branch Example

- `gcc –Og –S compare.c`

```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

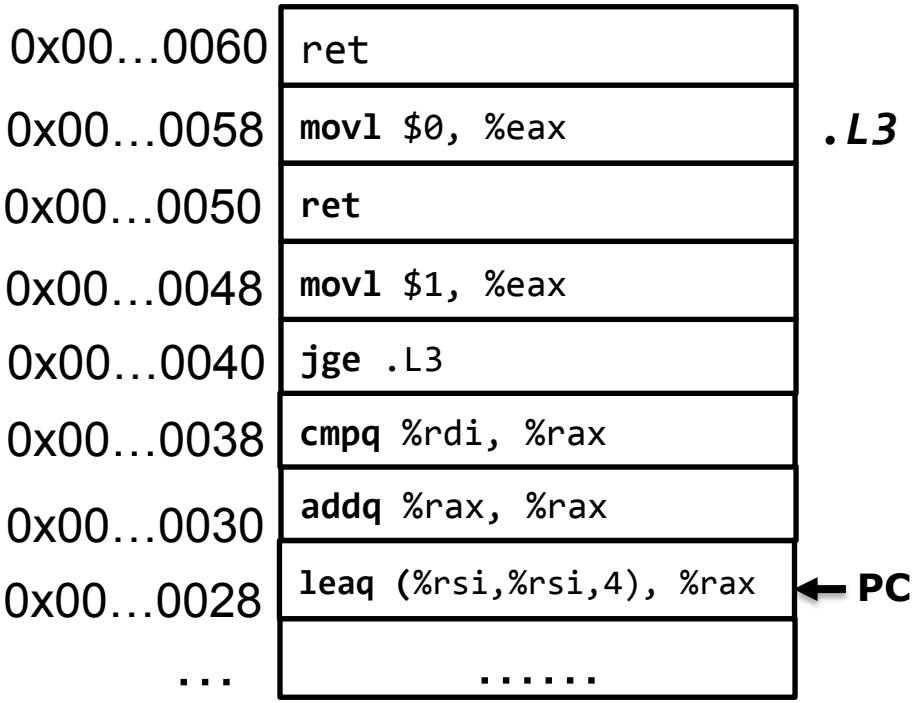
Conditional Branch Example

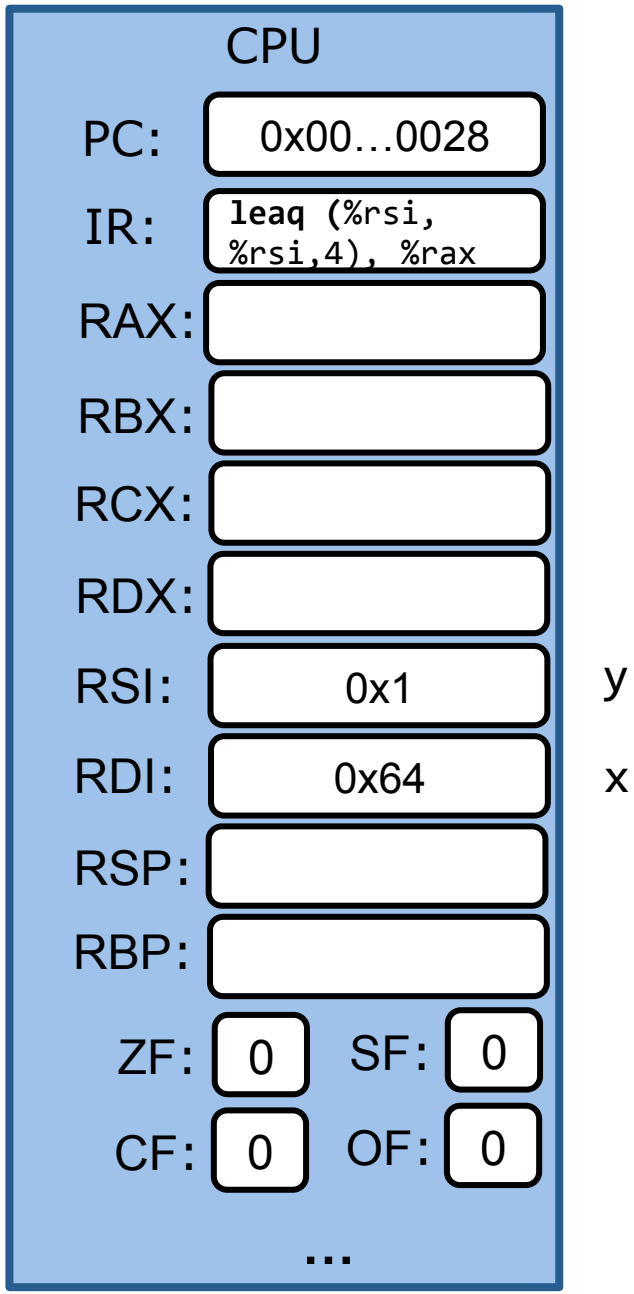
- `gcc –Og –S compare.c`

```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}
```

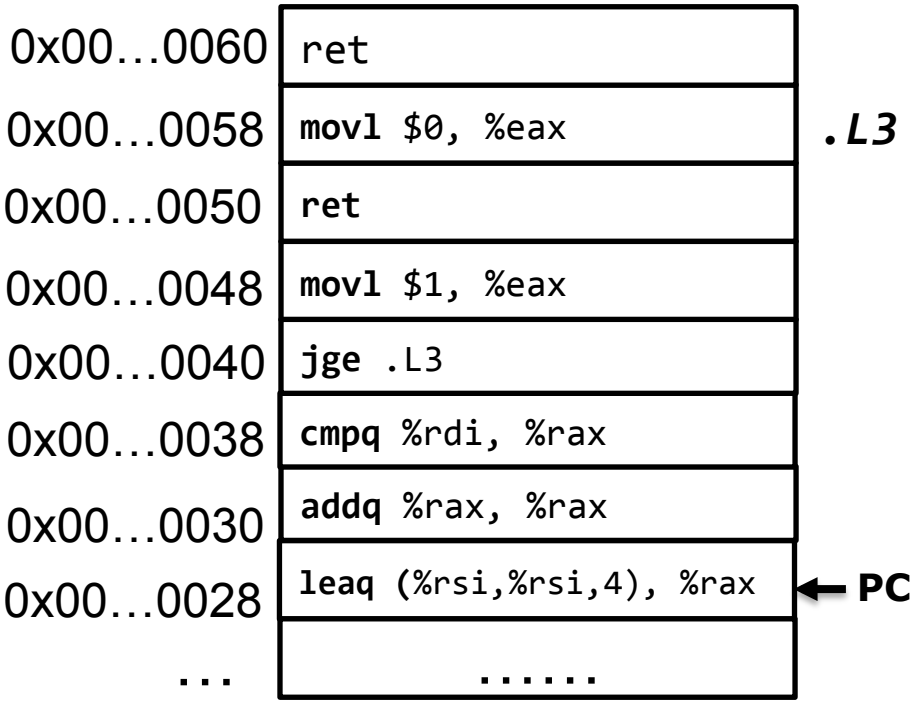
```
compare:
    leaq    (%rsi,%rsi,4), %rax
    addq    %rax, %rax
    cmpq    %rdi, %rax
    jge     .L3
    movl    $1, %eax
    ret
.L3:
    movl    $0, %eax
    ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

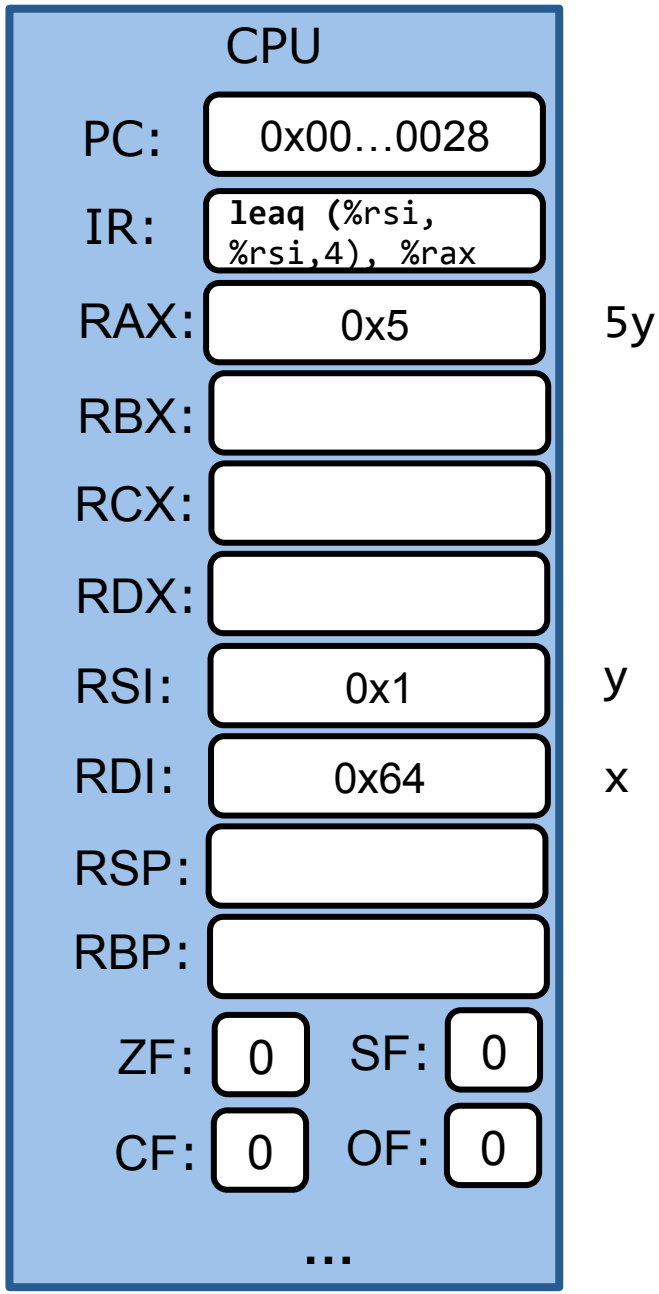


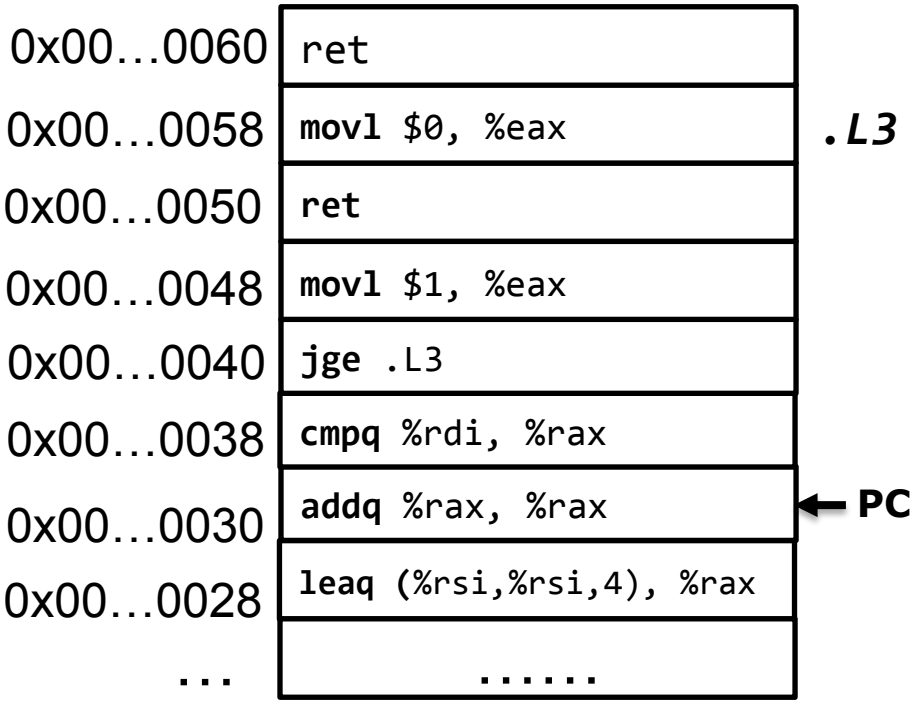



```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}

x: 100
y: 1
```

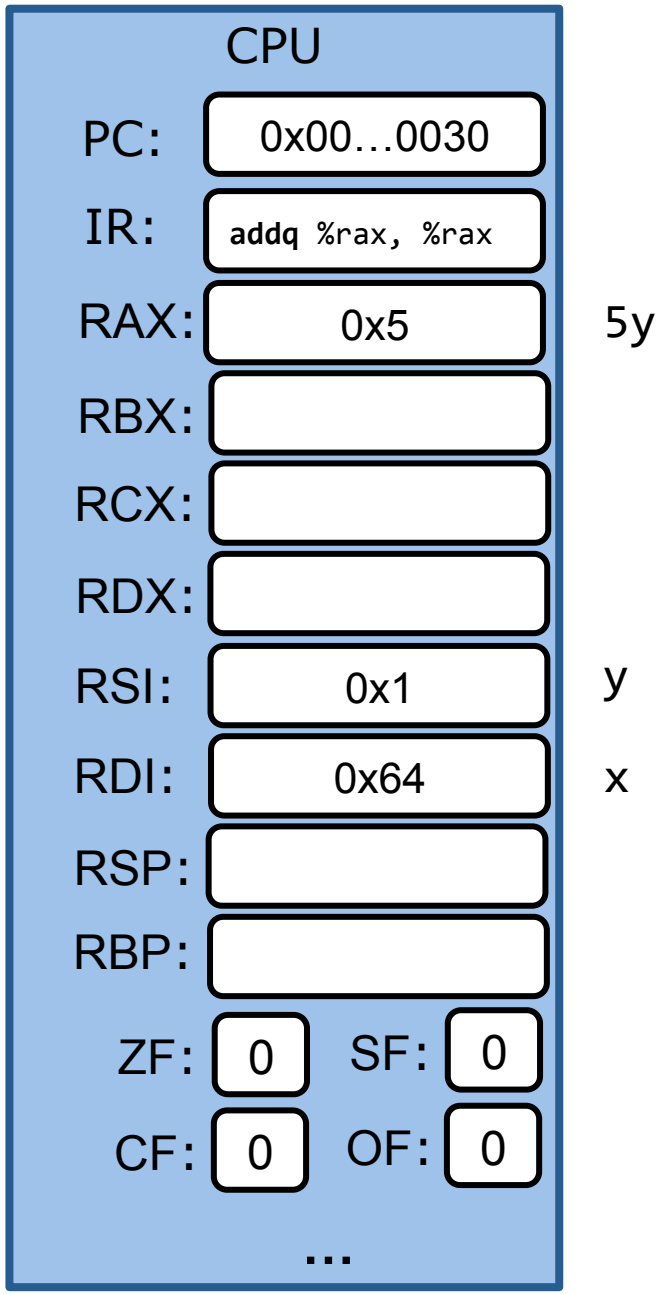


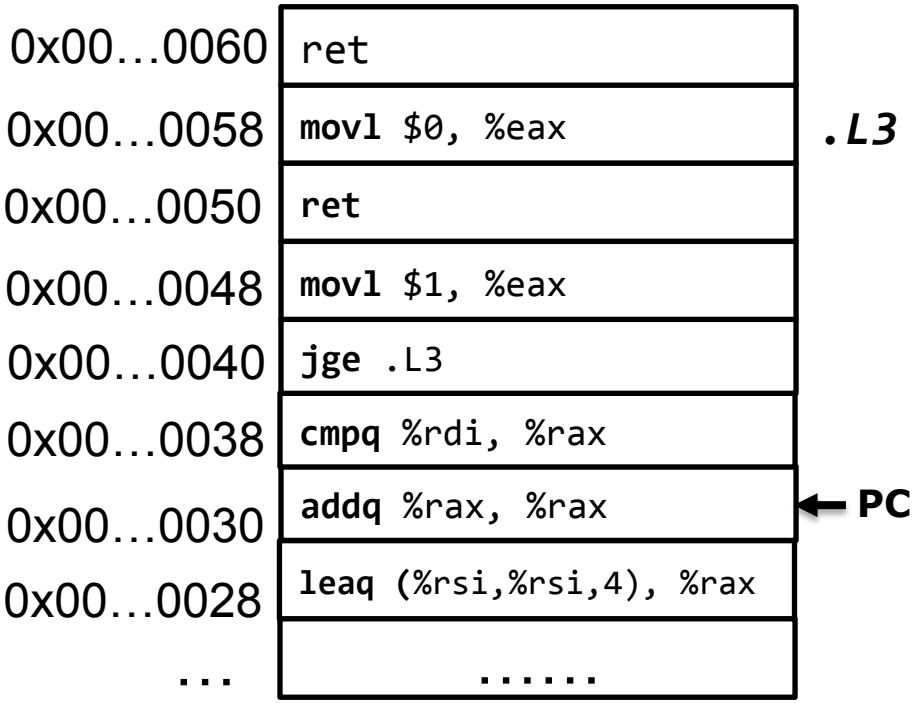

```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}

x: 100
y: 1
```



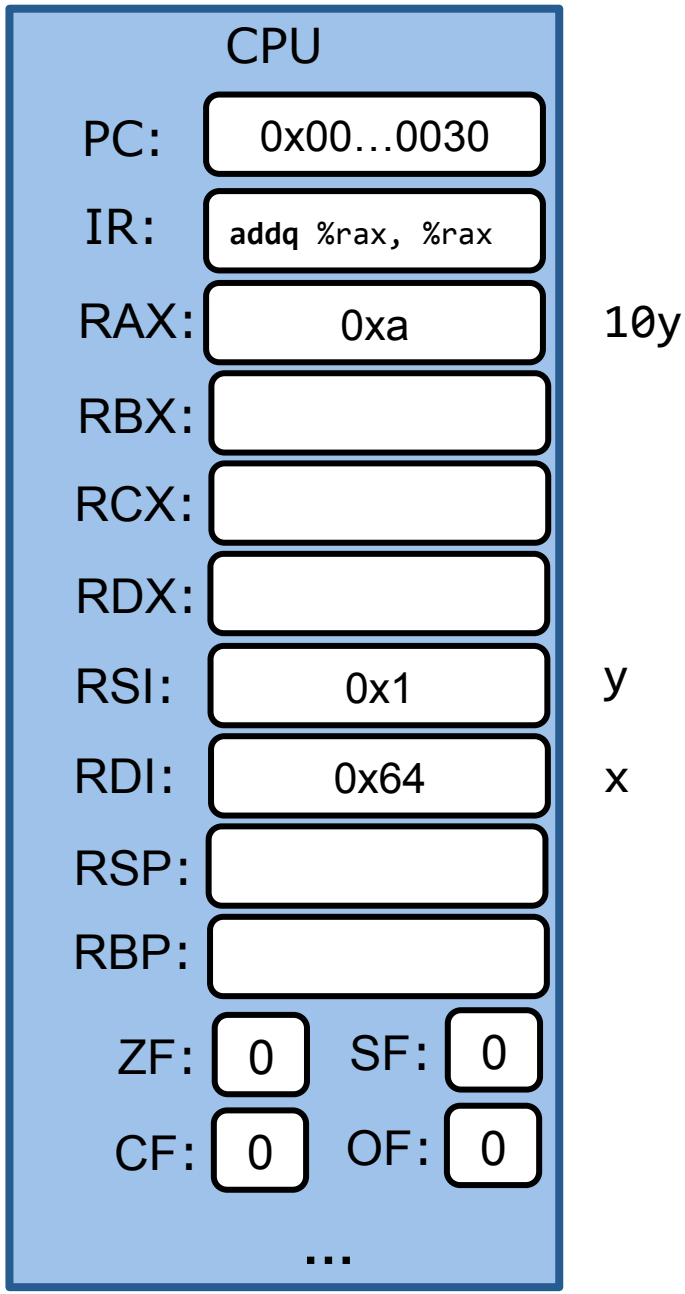


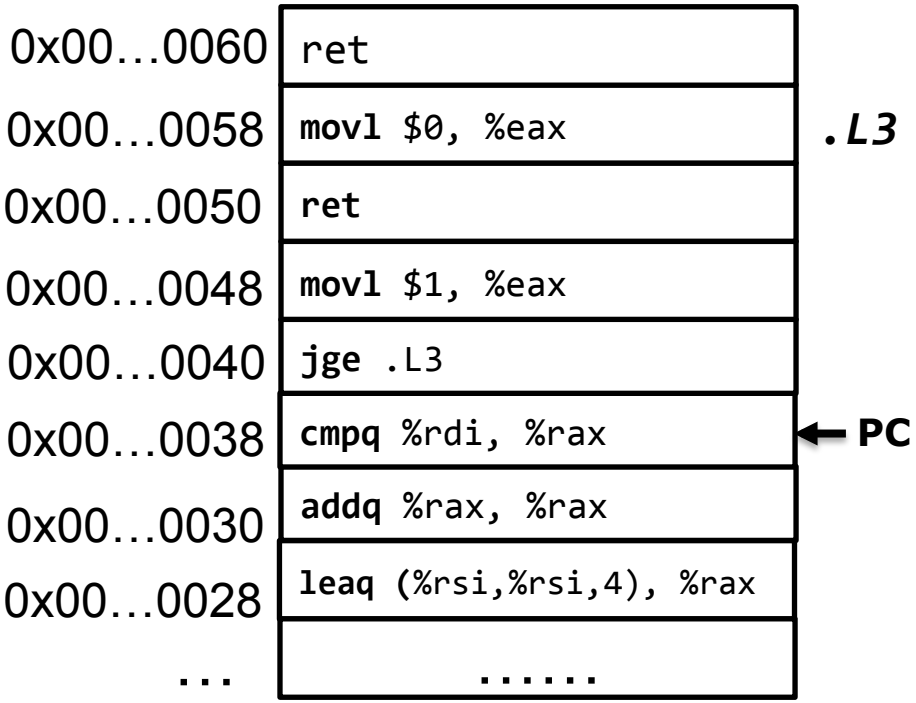

```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}

x: 100
y: 1
```

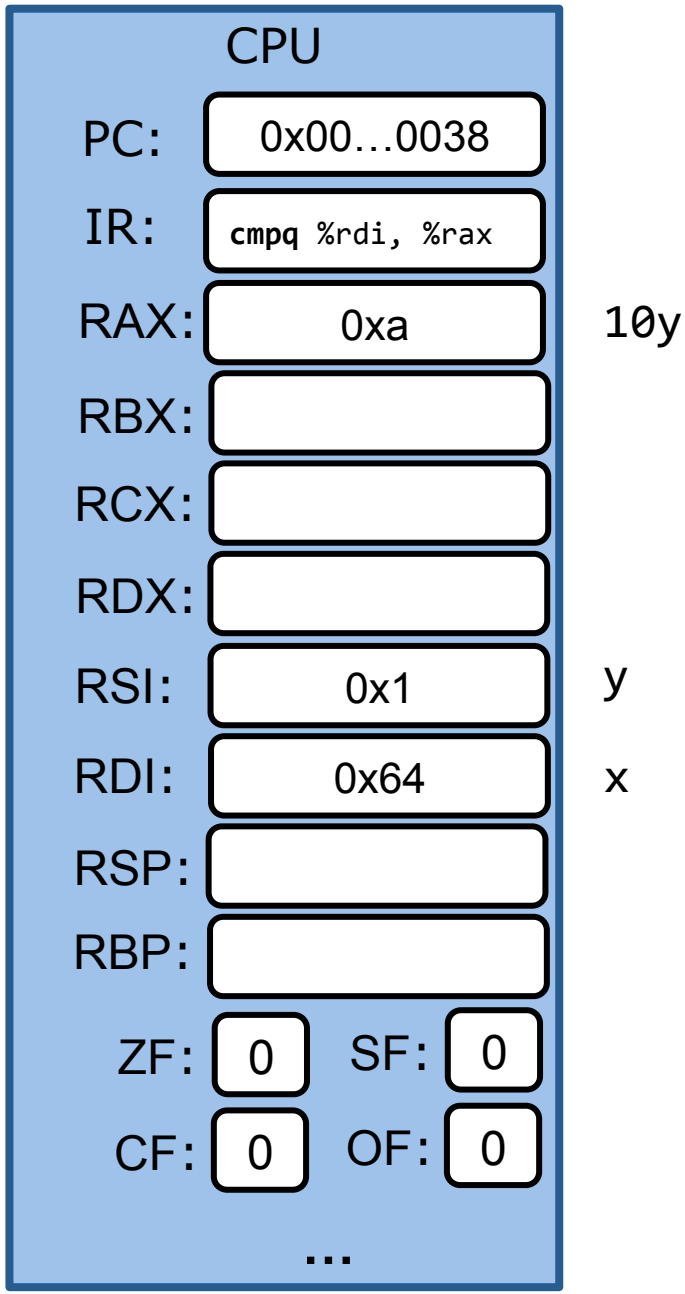




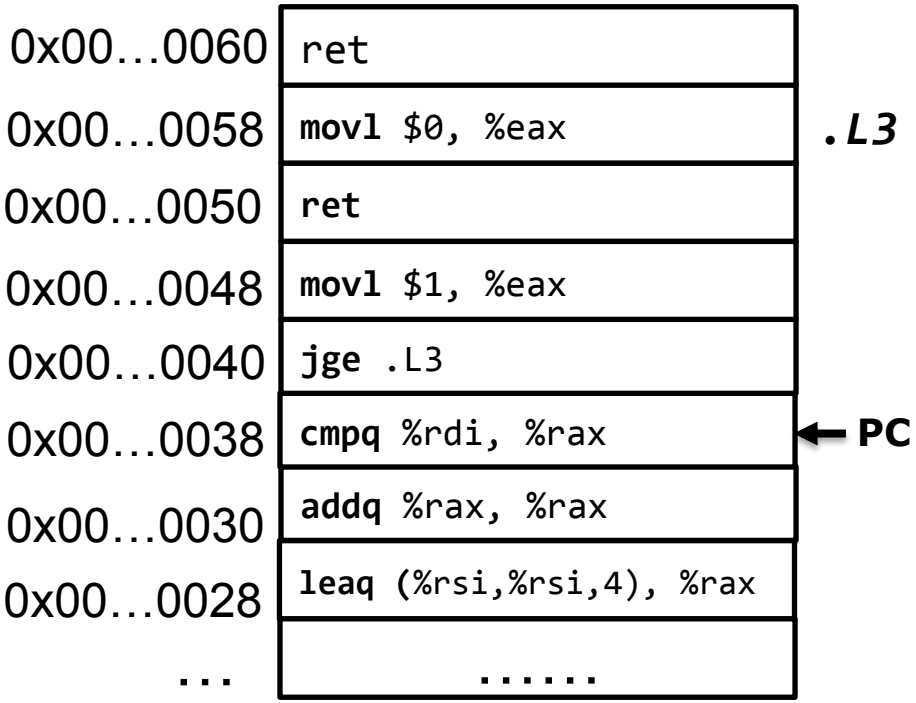

```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}

x: 100
y: 1
```



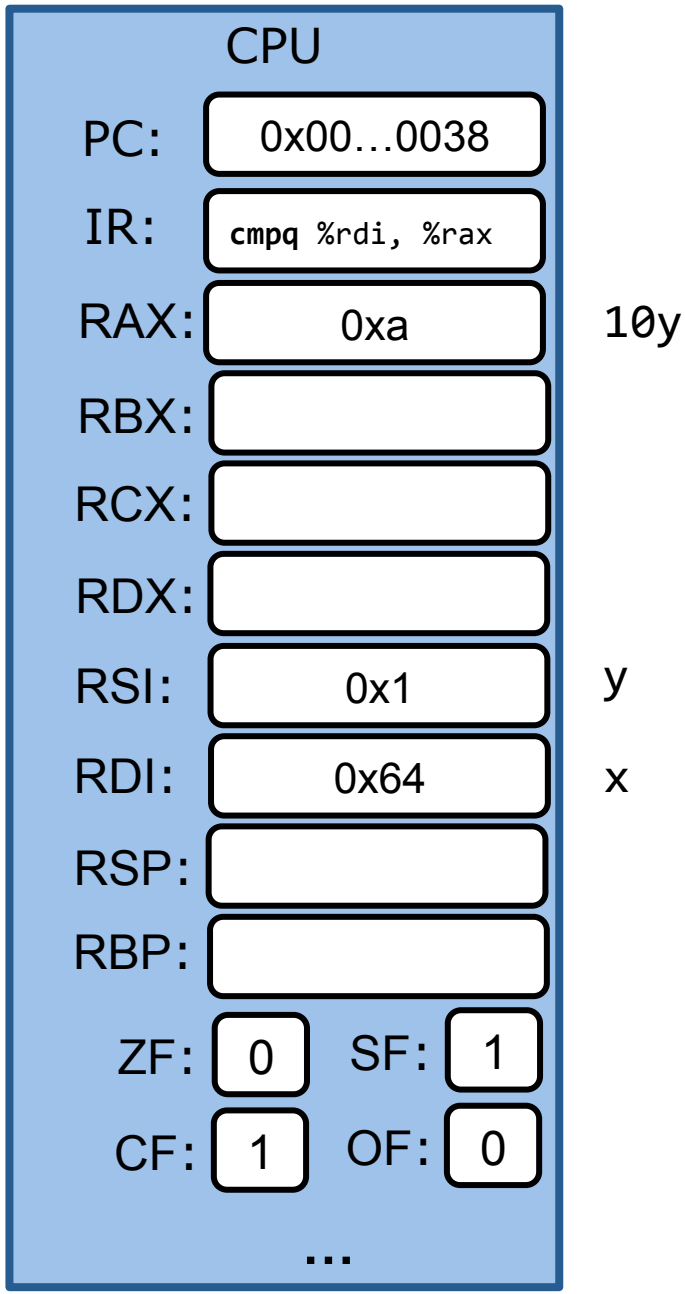


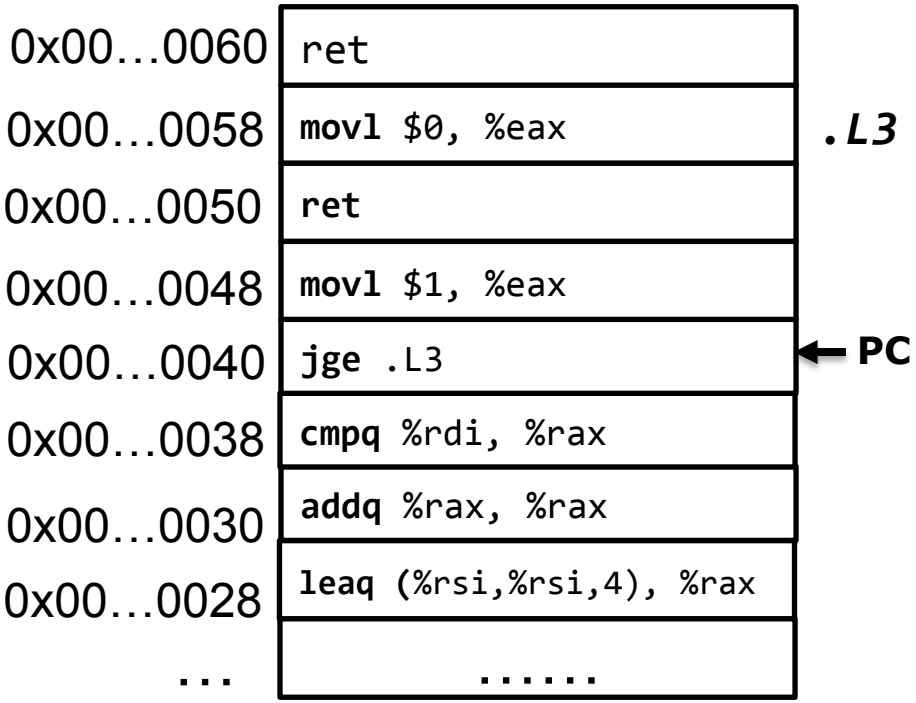


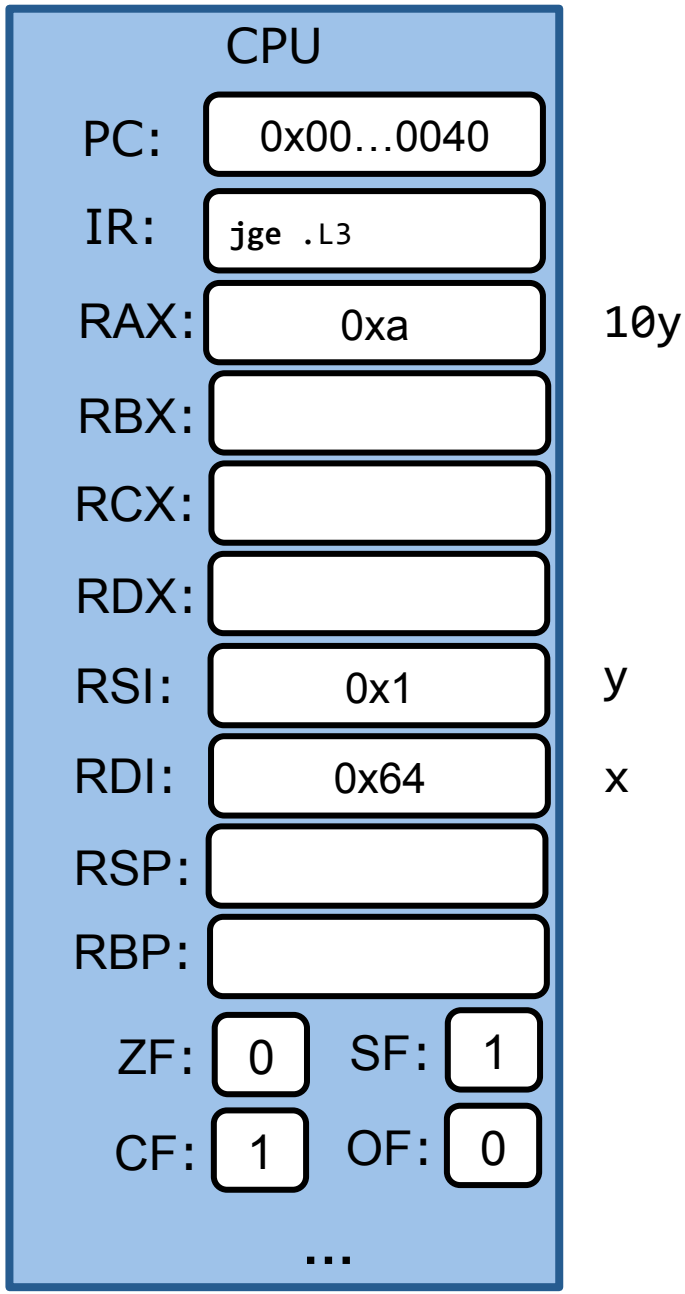

```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}

x: 100
y: 1
```




```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}

x: 100
y: 1
```



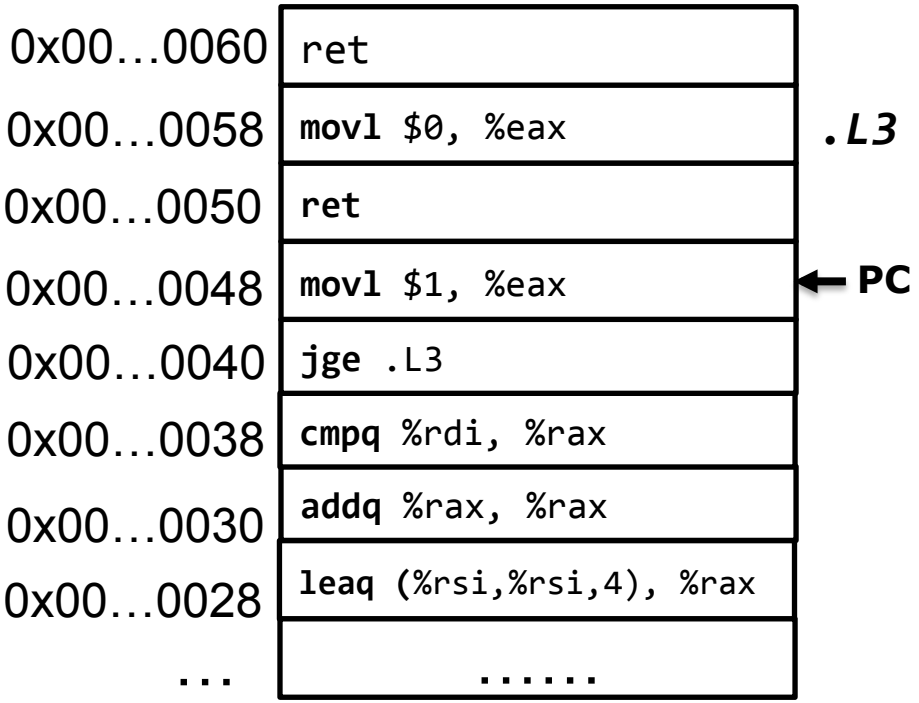


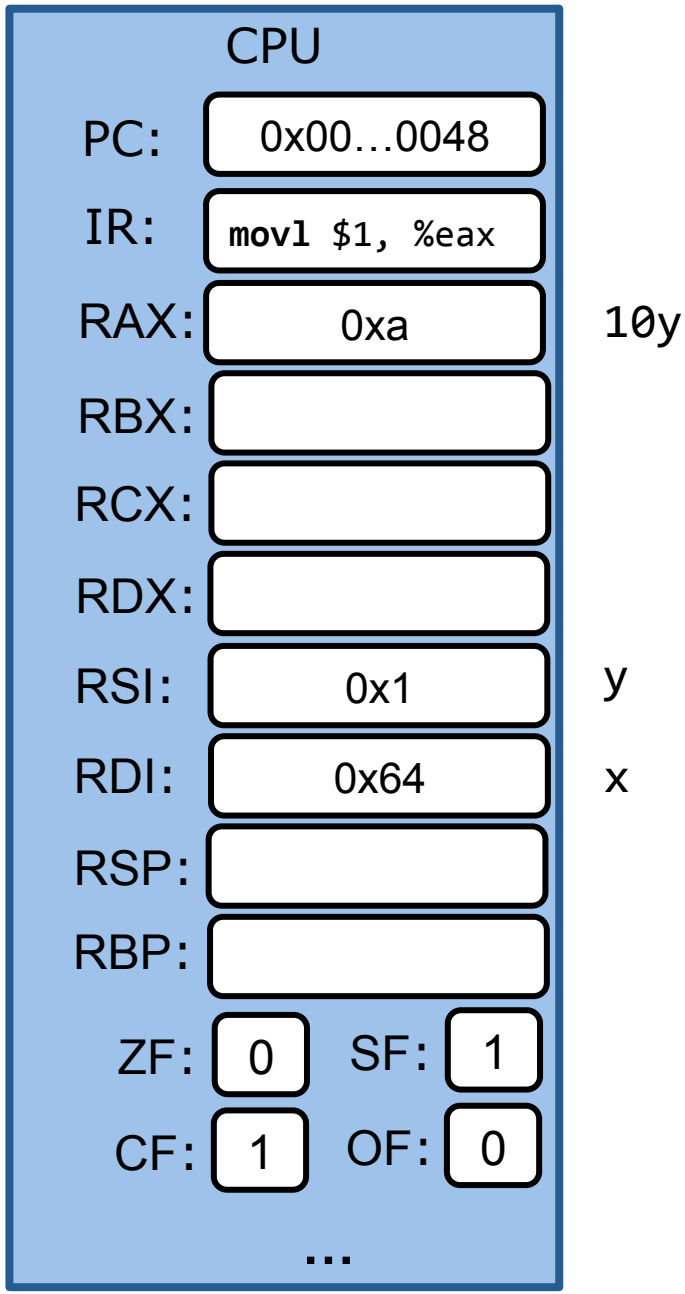



```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}
```

x: 100
y: 1

jge	~(SF^OF)

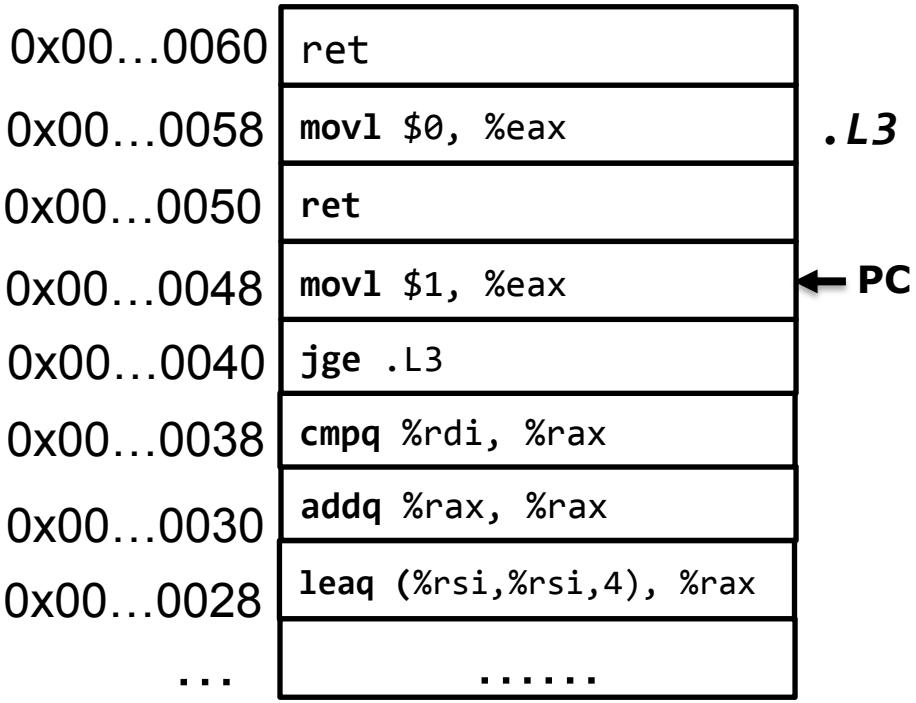


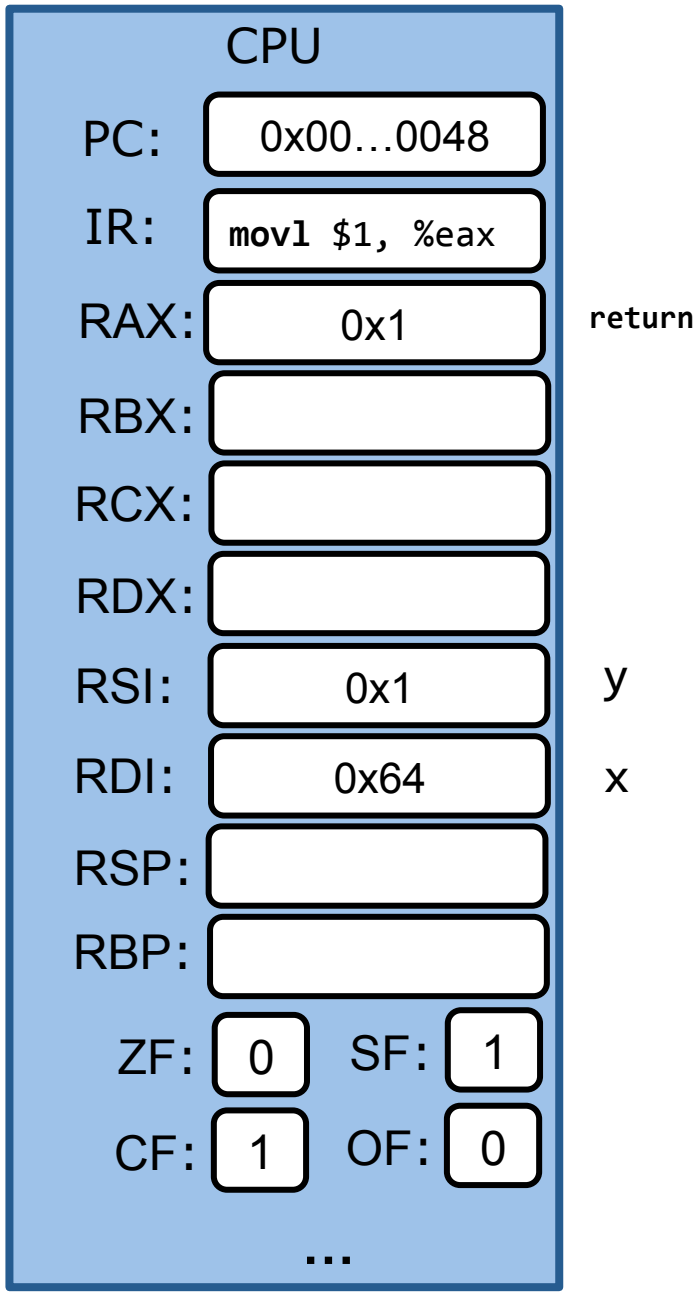



```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}
```

x: 100
y: 1

jge	~(SF^OF)

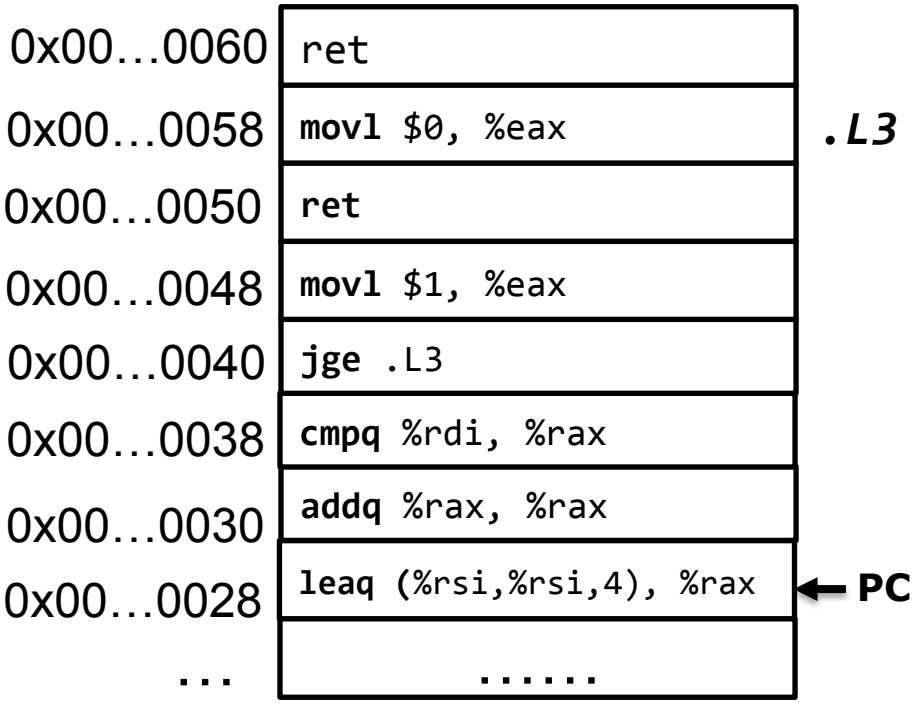


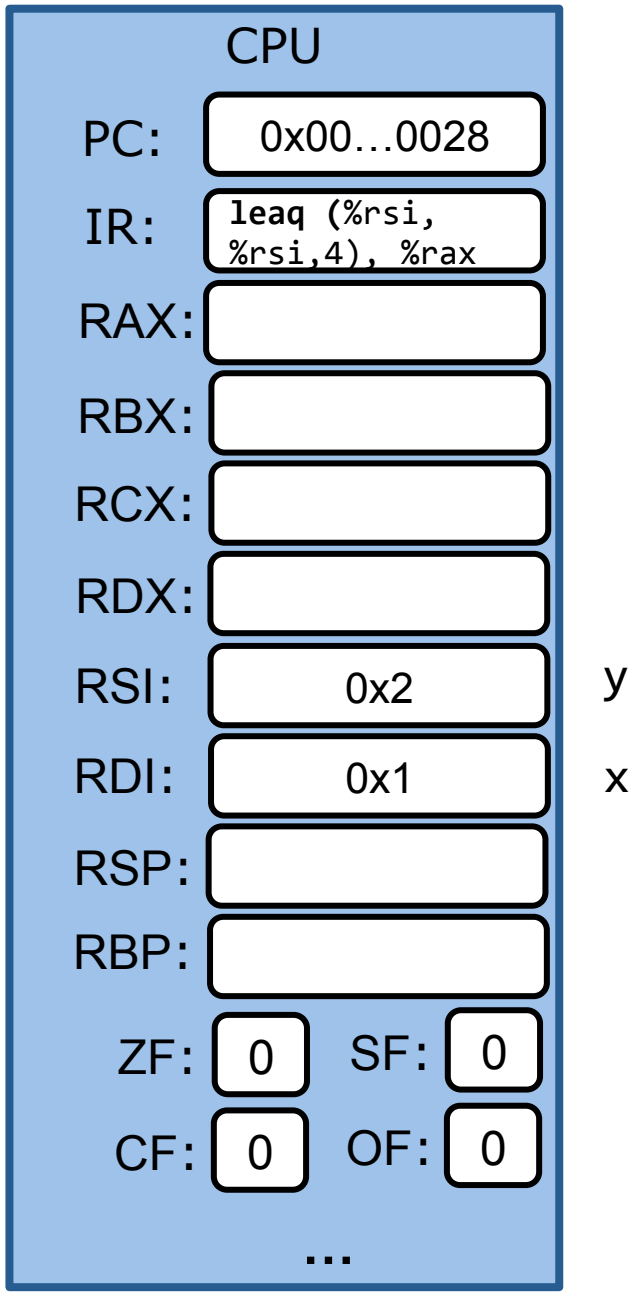



```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}
```

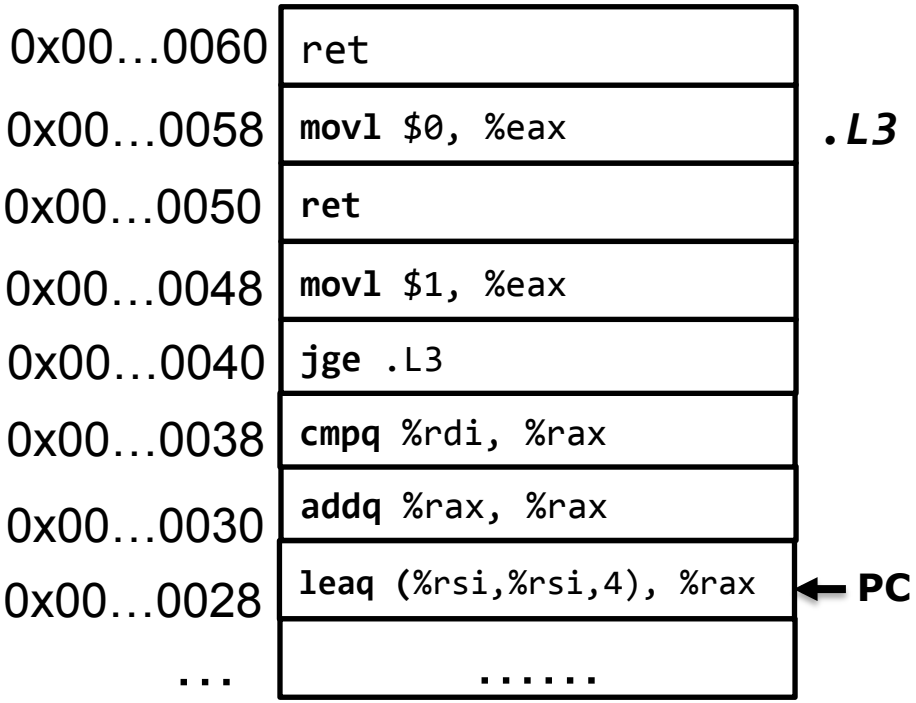
x: 100
y: 1

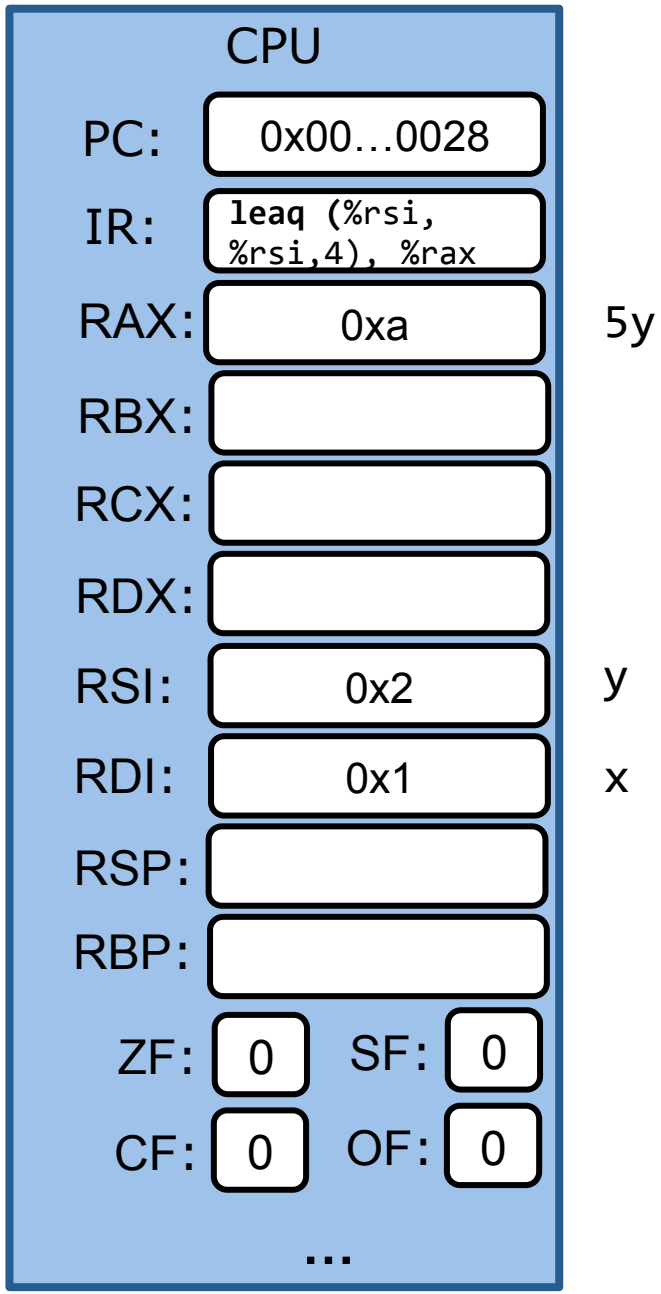
jge	~(SF^OF)



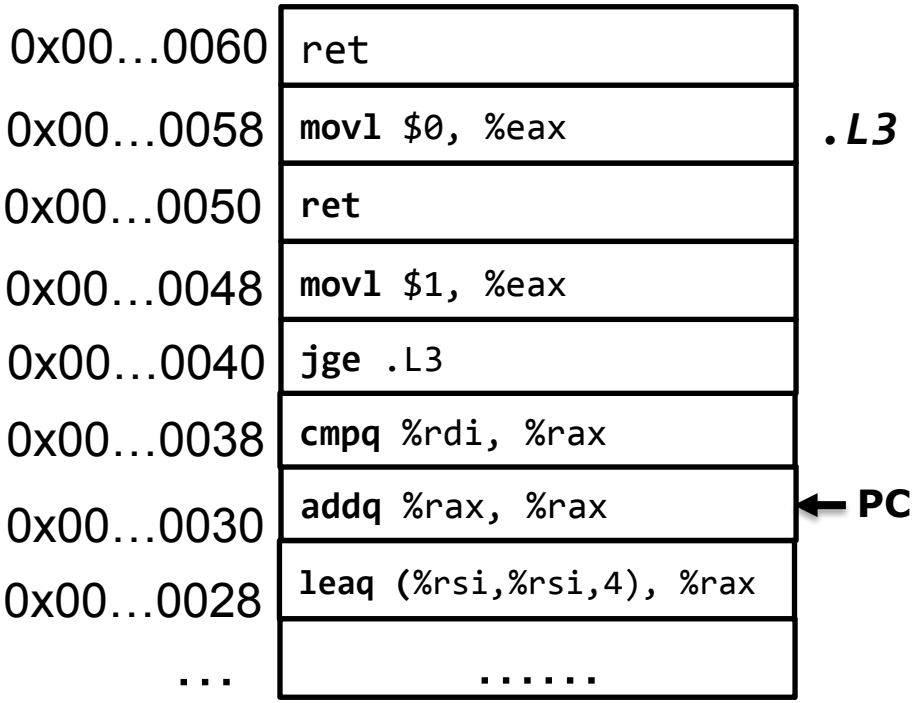



```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}

x: 1
y: 2
```



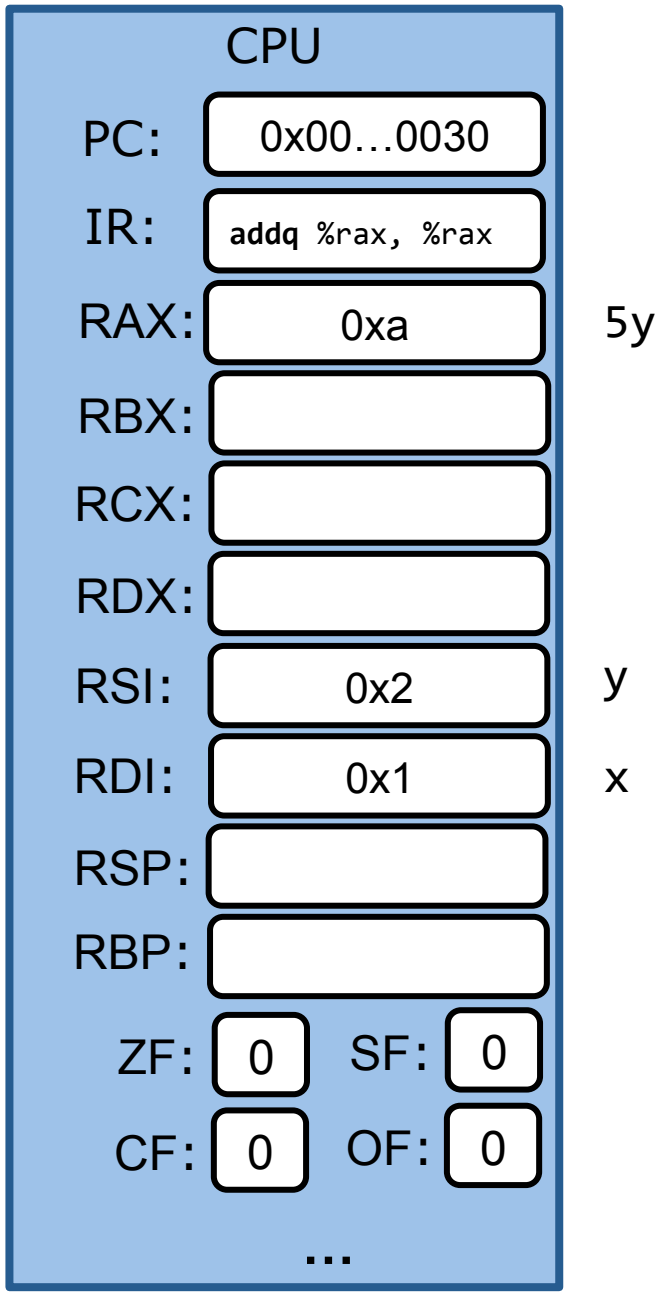


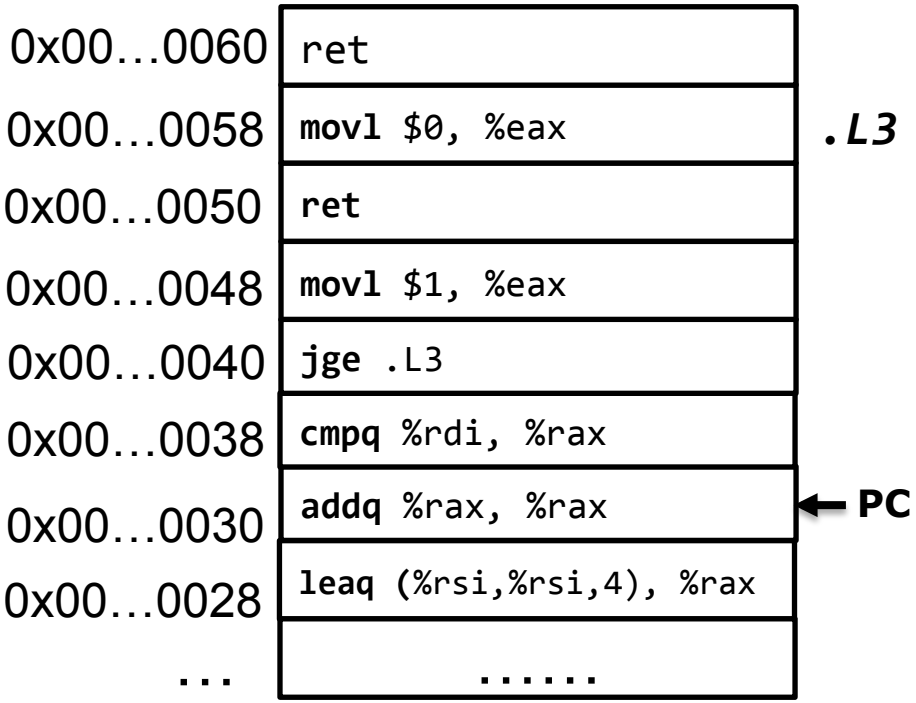

```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}

x: 1
y: 2
```




```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}
```

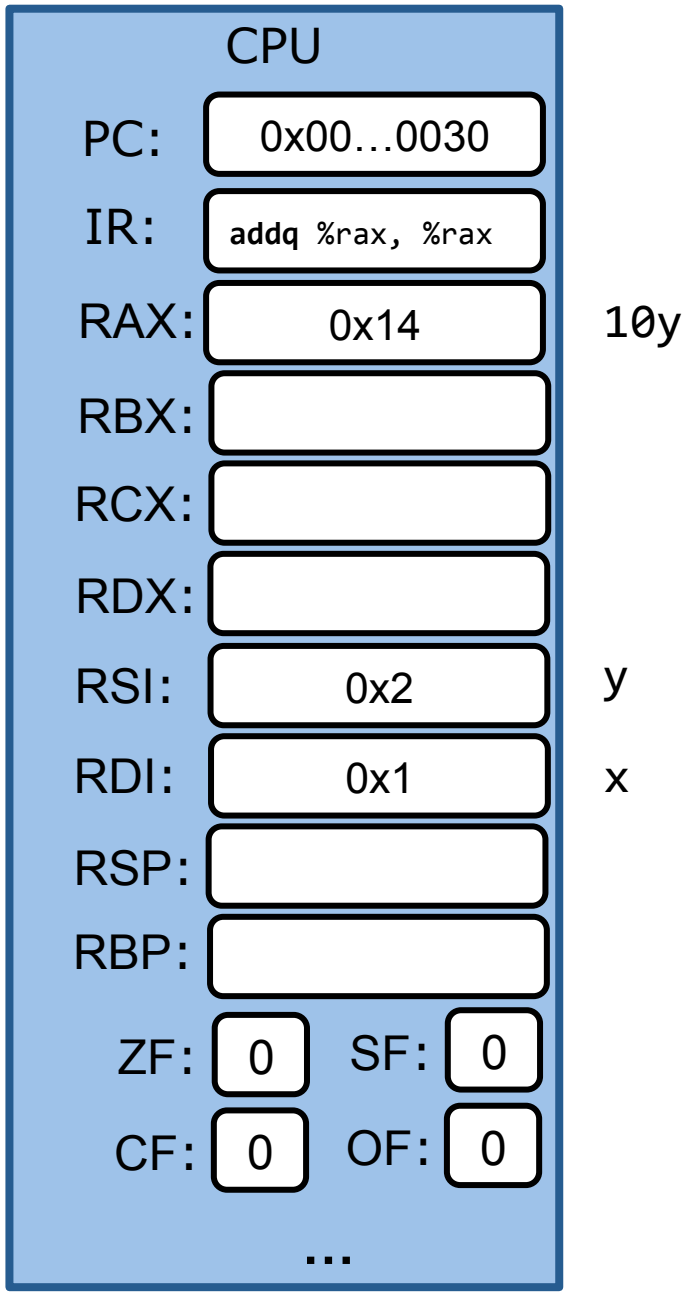
x: 1
y: 2

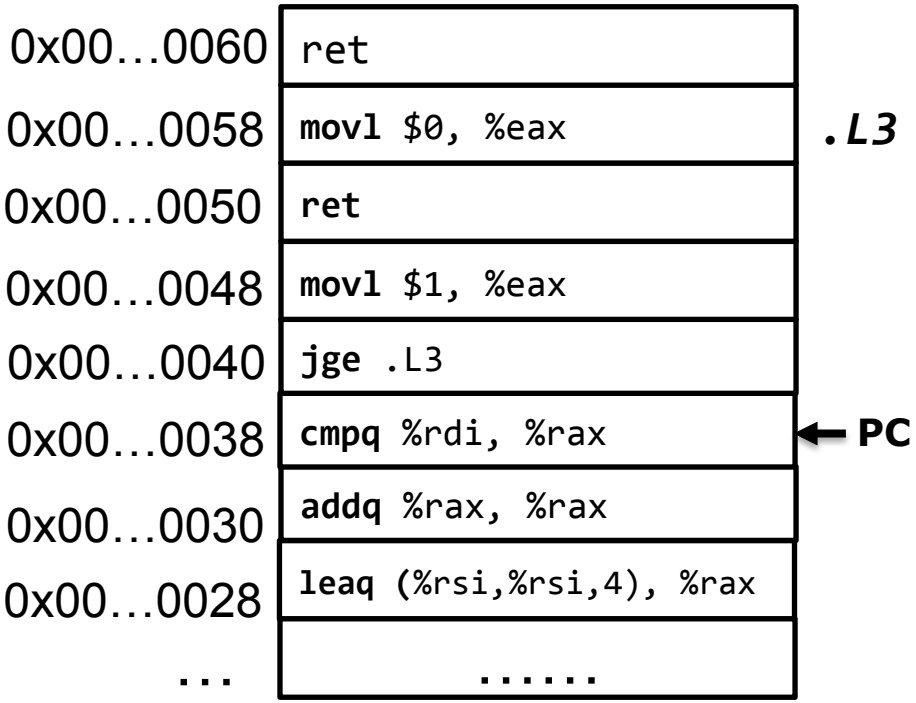


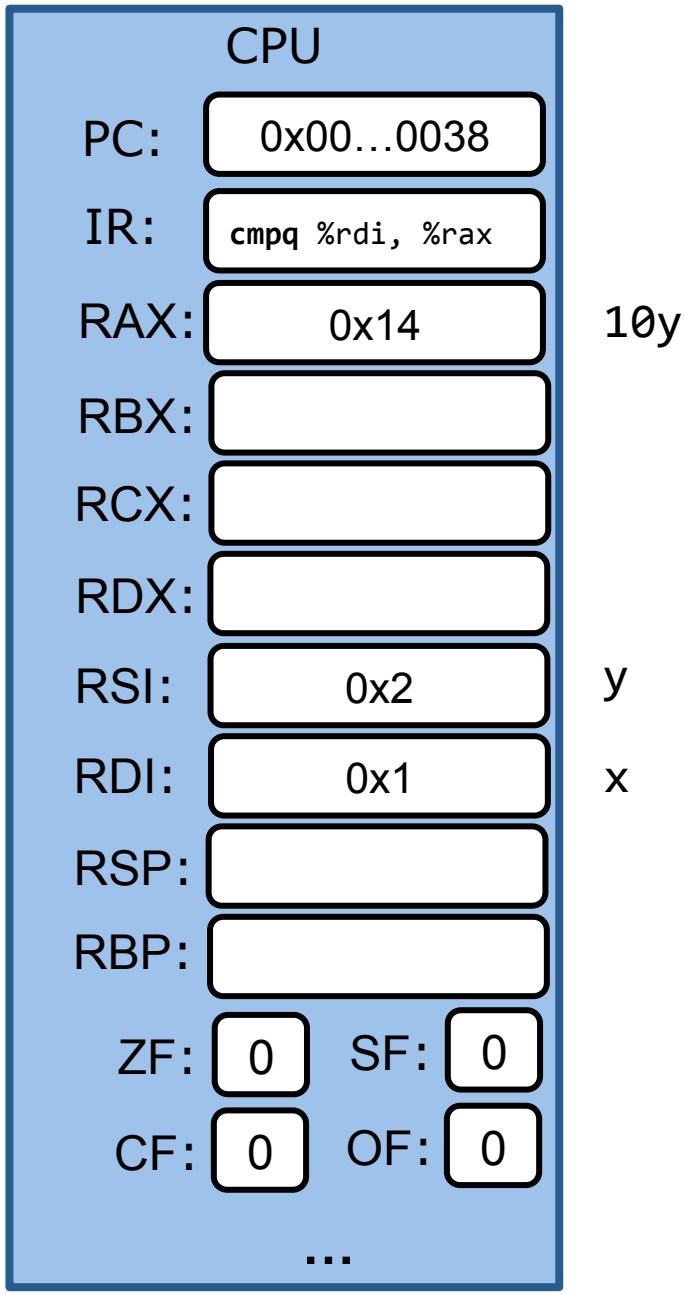



```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}
```

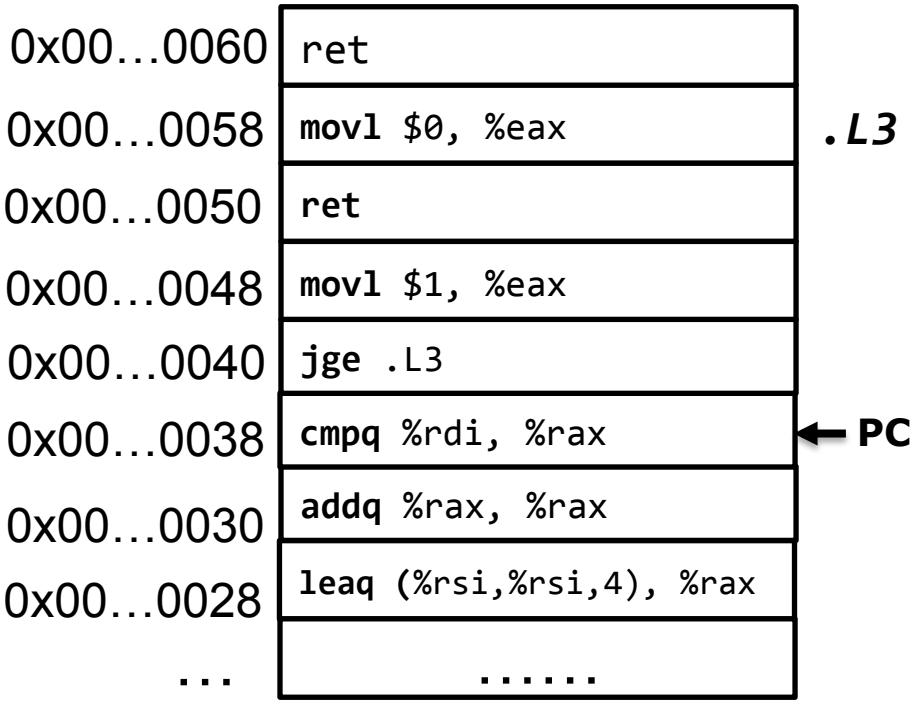
x: 1
y: 2

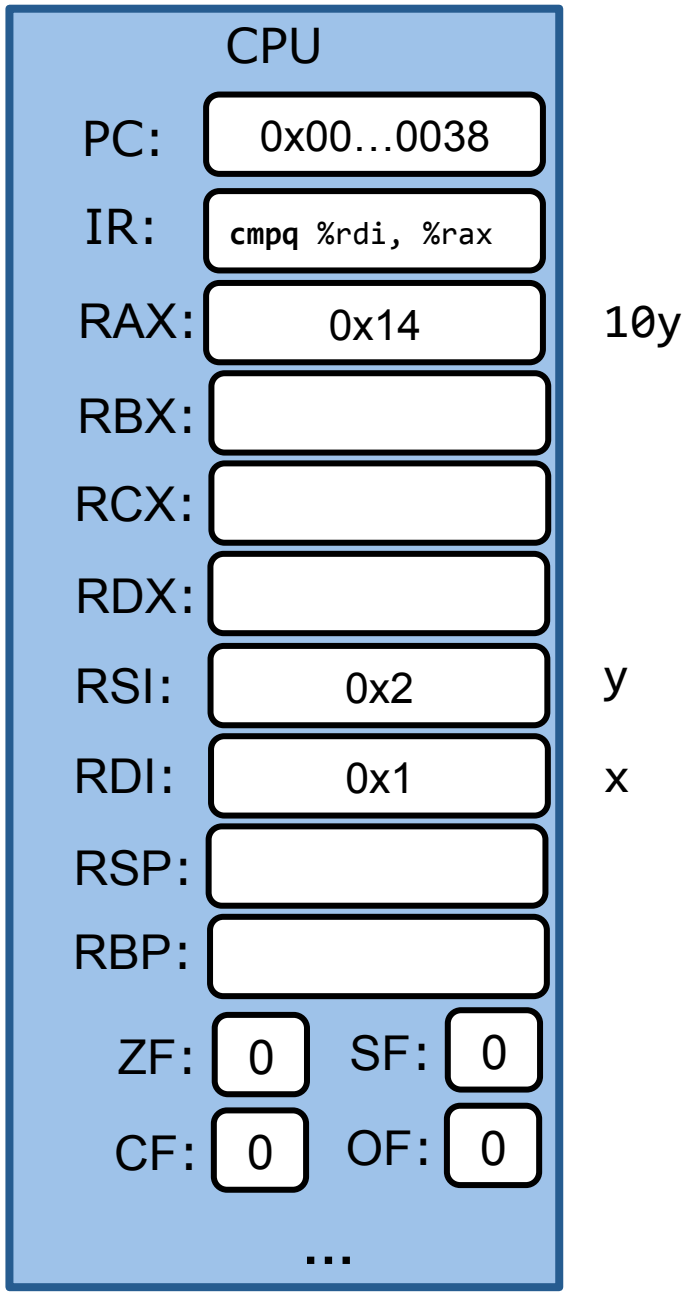




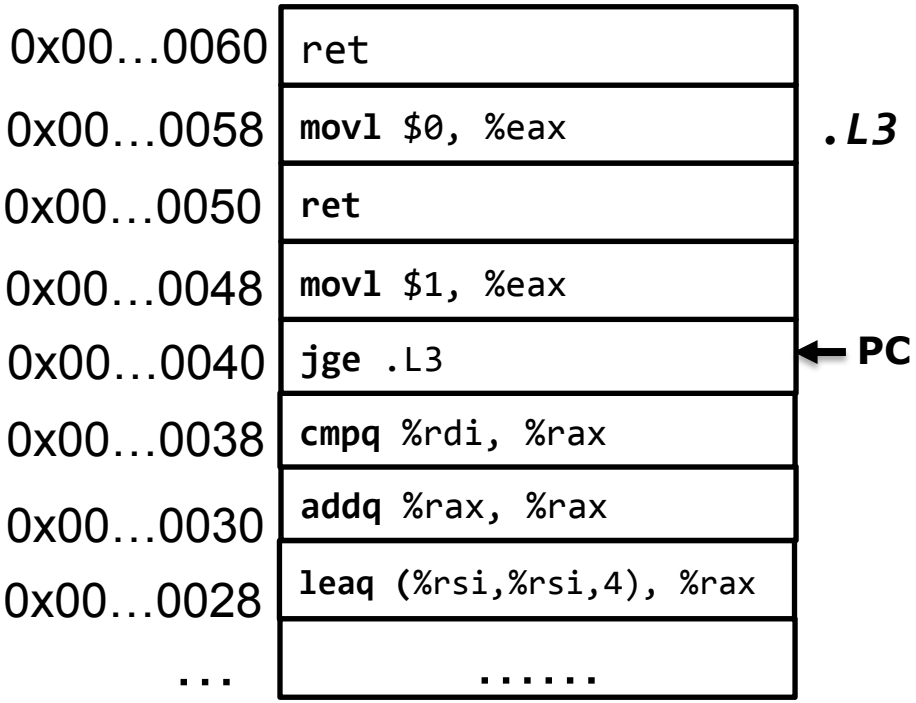


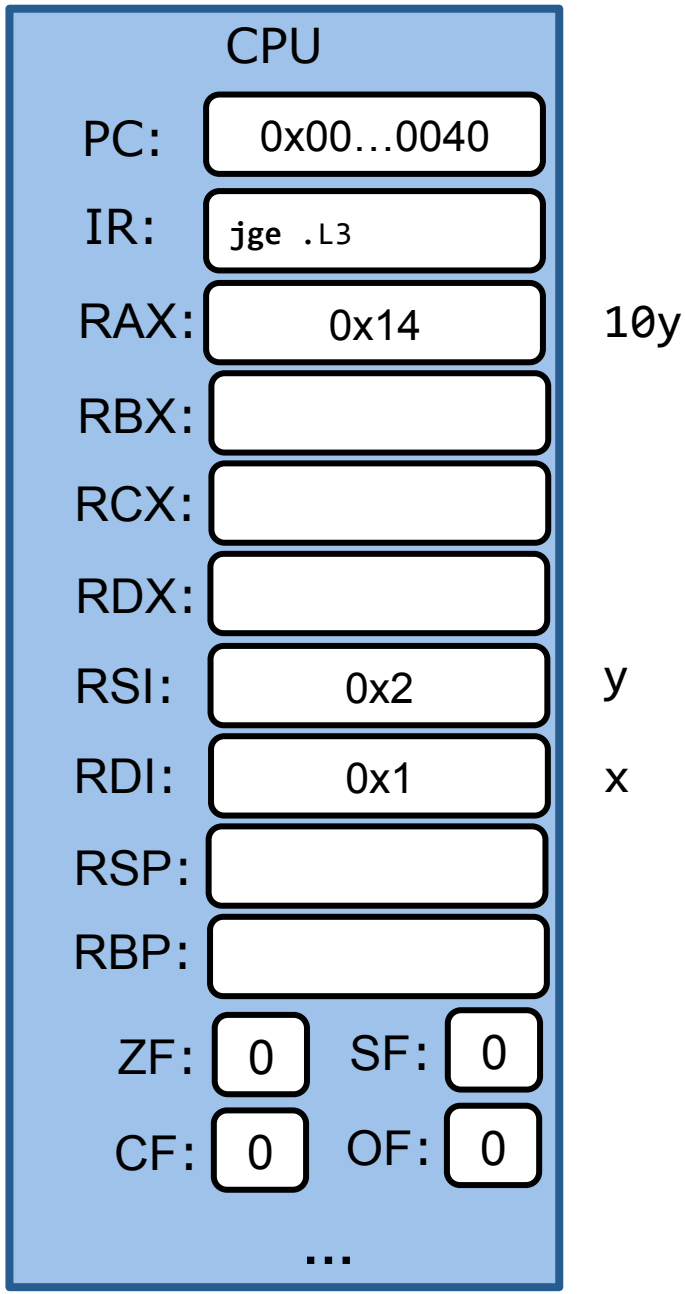

```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}

x: 1
y: 2
```






```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}

x: 1
y: 2
```

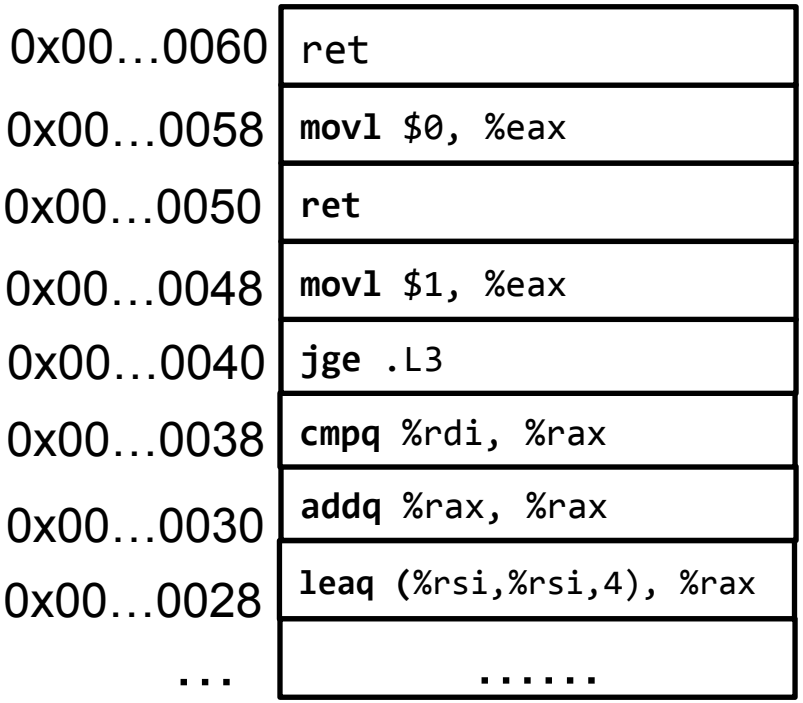


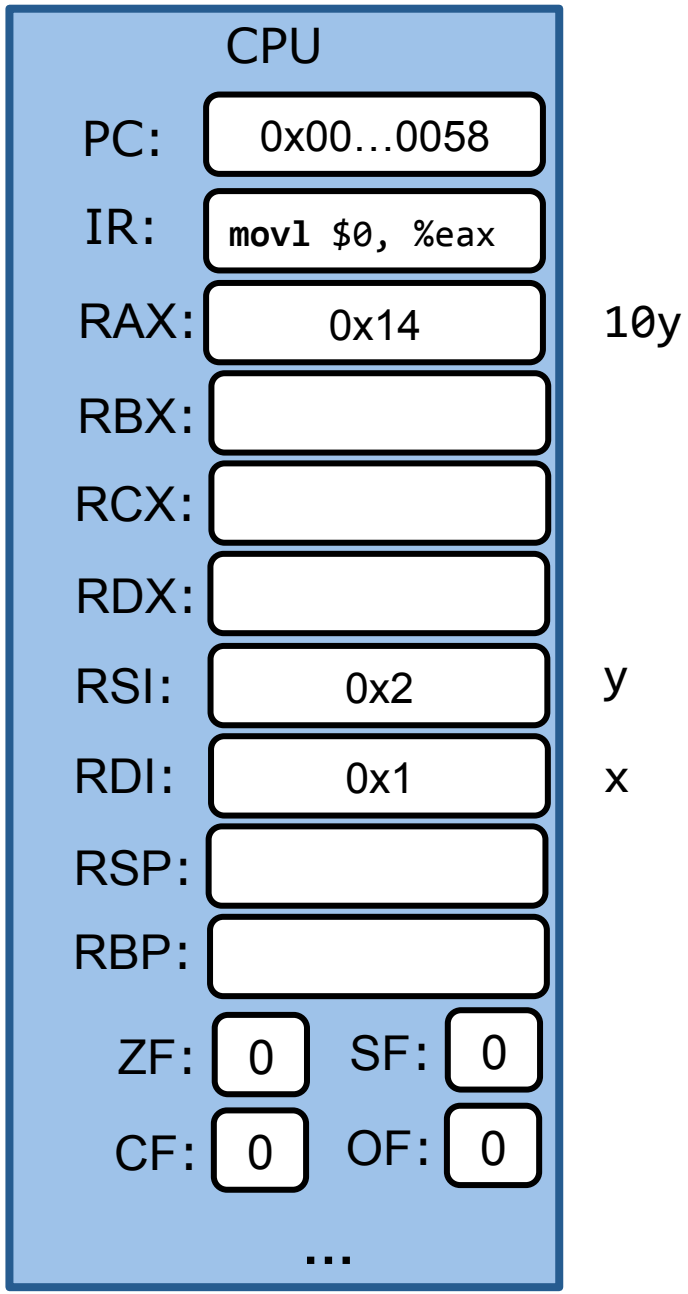



```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}
```

x: 1
y: 2

jge	~(SF^OF)

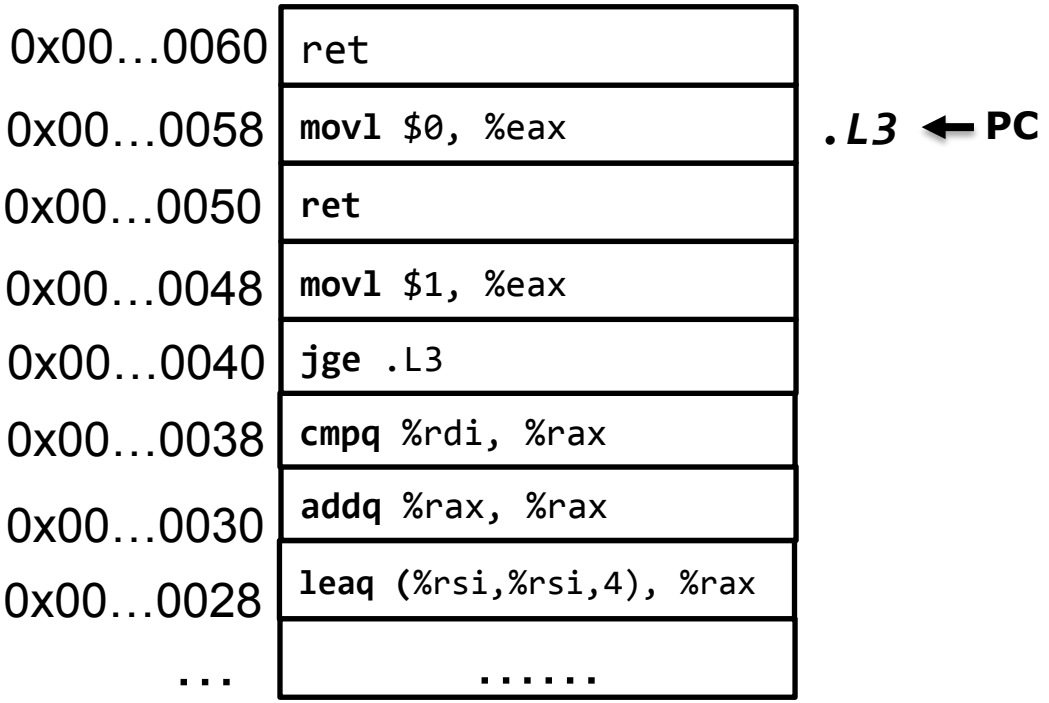


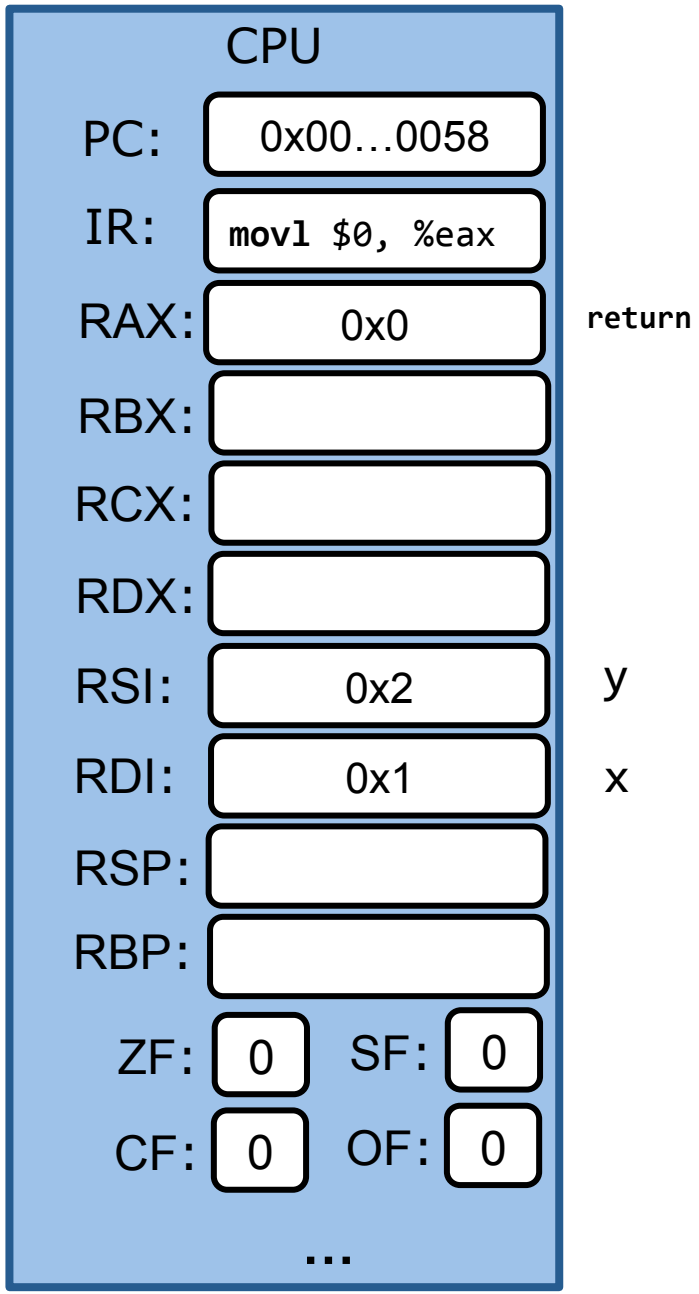



```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}
```

x: 1
y: 2

jge	~(SF^OF)






```
long compare(long x, long y)
{
  long result;
  if (x > 10*y)
    result = 1;
  else
    result = 0;
  return result;
}
```

x: 1
y: 2

jge	~(SF^OF)

"While" Translation example

`gcc –Og –S *.c`

```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
    x = x >> 1;
    cnt++;
   }
   return cnt;
}
```

Register	Use(s)
`%rdi`	Argument **`x`**
`%rax`	Return value

"While" Translation example

```
gcc –Og –S *.c
```

```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
    x = x >> 1;
    cnt++;
   }
   return cnt;
}
```

```
count:
    movq $0, %rax
    jmp .L2
.L3:
    shrq %rdi
    addq $1, %rax
.L2:
    testq %rdi, %rdi
    jne .L3
    ret
```

Register	Use(s)
%rdi	Argument **x**
%rax	Return value

shrq: logical right shift

"While" Translation example

```
gcc –Og –S *.c
```

```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
    x = x >> 1;
    cnt++;
   }
   return cnt;
}
```

```
count:
    movq $0, %rax          long cnt = 0;
    jmp .L2
.L3:
    shrq %rdi
    addq $1, %rax
.L2:
    testq %rdi, %rdi
    jne .L3
    ret
```

Register	Use(s)
%rdi	Argument **x**
%rax	Return value

shrq: logical right shift

"While" Translation example

gcc –Og –S *.c

```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
    x = x >> 1;
    cnt++;
   }
   return cnt;
}
```

```
count:
    movq $0, %rax          long cnt = 0;
    jmp .L2                goto .L2
.L3:
    shrq %rdi
    addq $1, %rax
.L2:
    testq %rdi, %rdi
    jne .L3
    ret
```

Register	Use(s)
%rdi	Argument **x**
%rax	Return value

shrq: logical right shift

"While" Translation example

gcc –Og –S *.c

```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
    x = x >> 1;
    cnt++;
   }
   return cnt;
}
```

```
count:
    movq $0, %rax
    jmp .L2
.L3:
    shrq %rdi
    addq $1, %rax
.L2:
    testq %rdi, %rdi
    jne .L3
    ret
```

```
long cnt = 0;
goto .L2

.L2:
if x != 0
  goto .L3

return cnt
```

Register	Use(s)
%rdi	Argument x
%rax	Return value

shrq: logical right shift

"While" Translation example

gcc –Og –S *.c

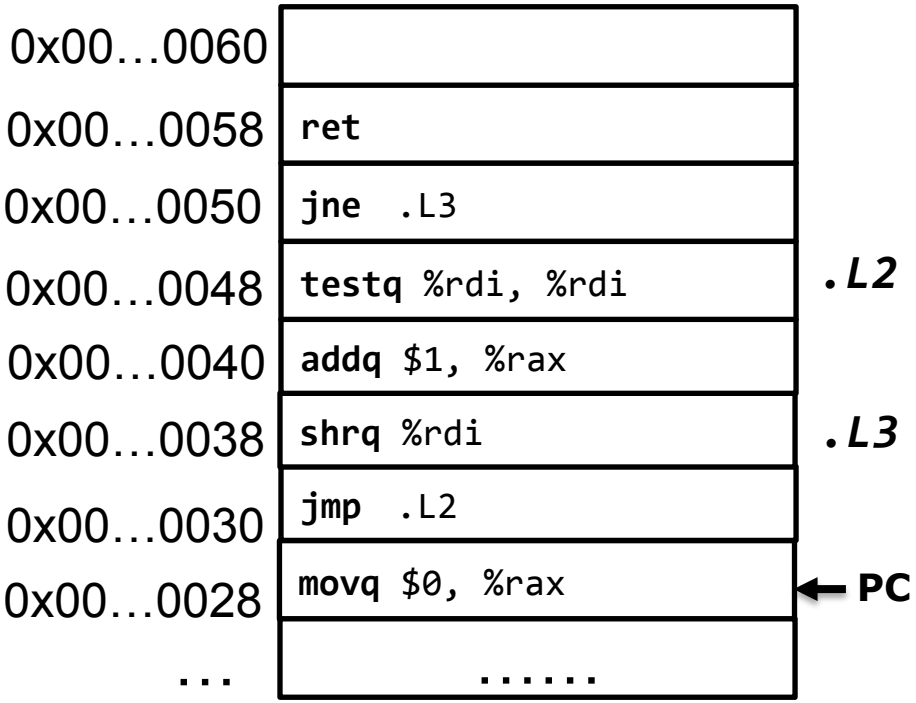
```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
    x = x >> 1;
    cnt++;
   }
   return cnt;
}
```

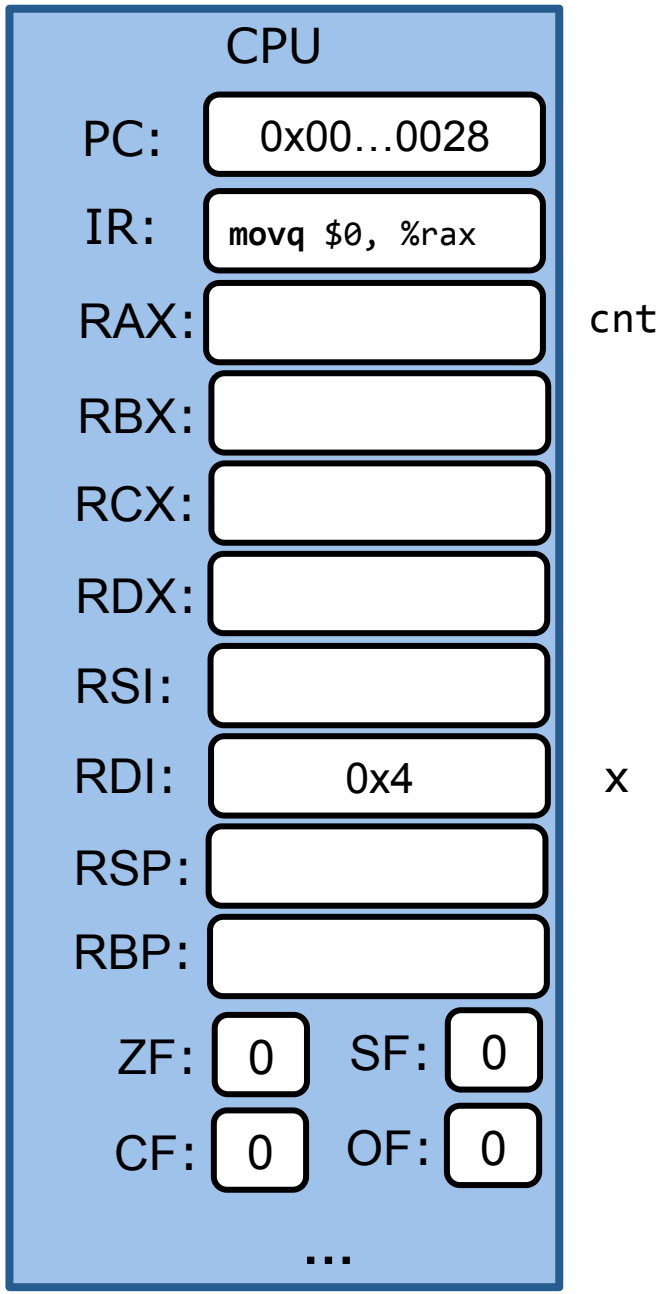
```
count:
    movq $0, %rax
    jmp .L2
.L3:
    shrq %rdi
    addq $1, %rax
.L2:
    testq %rdi, %rdi
    jne .L3
    ret
```

```
long cnt = 0;
goto .L2
.L3:
x = x >> 1
cnt = cnt + 1
.L2:
if x != 0
  goto .L3

return cnt
```

Register	Use(s)
%rdi	Argument **x**
%rax	Return value

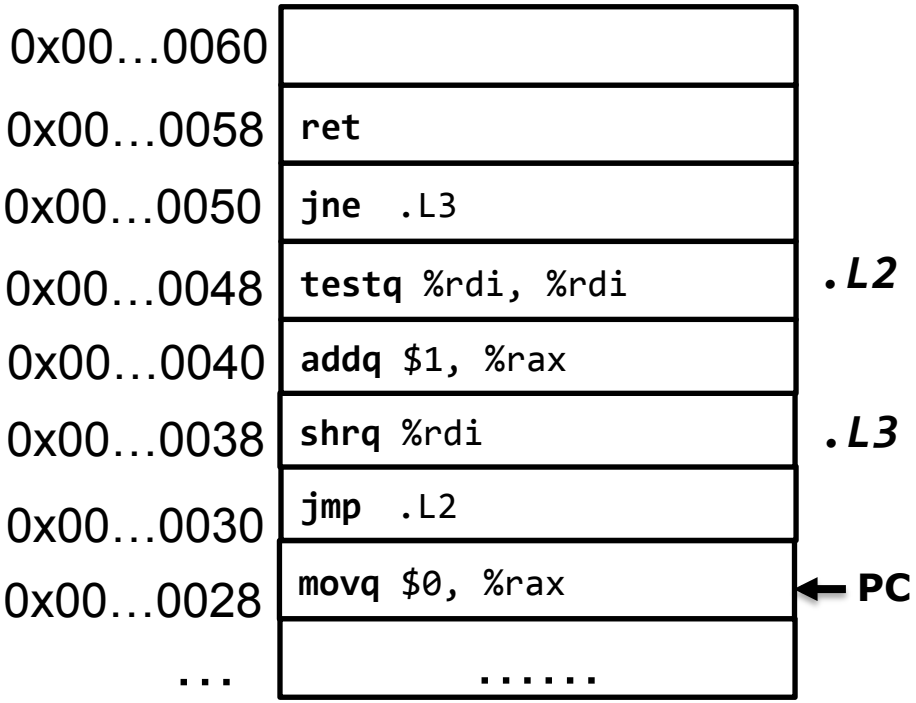
shrq: logical right shift

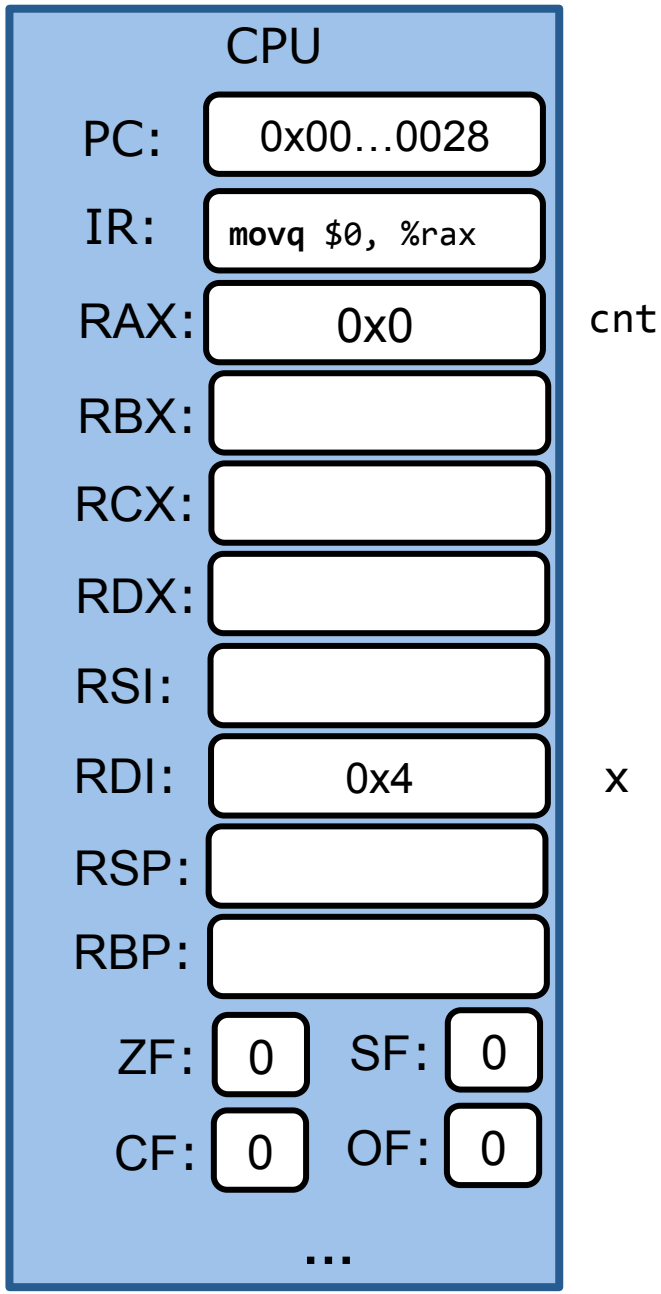





```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

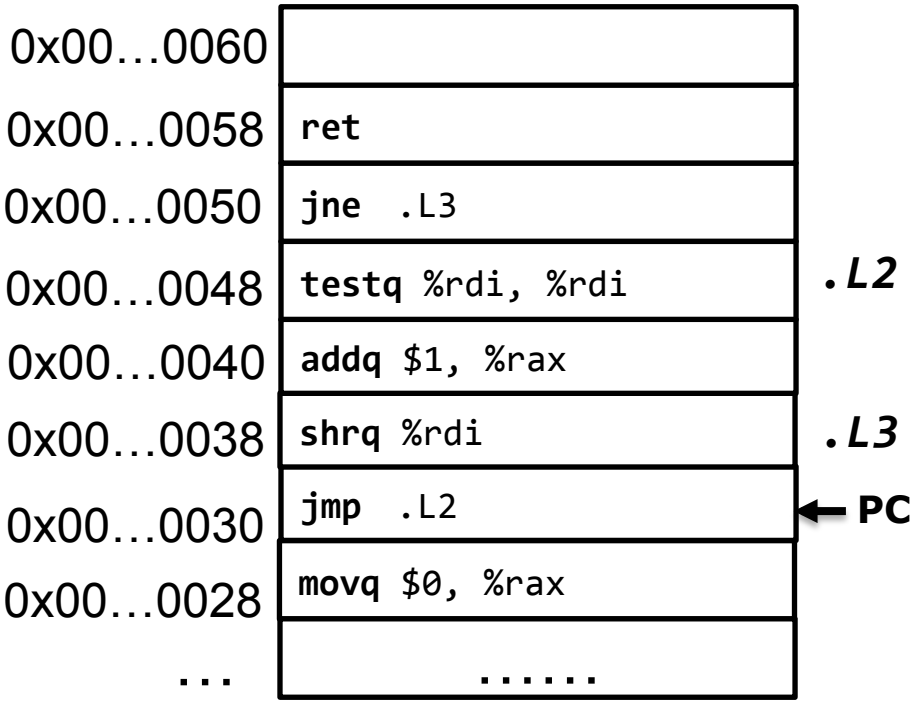
x: 4 $(100)_2$

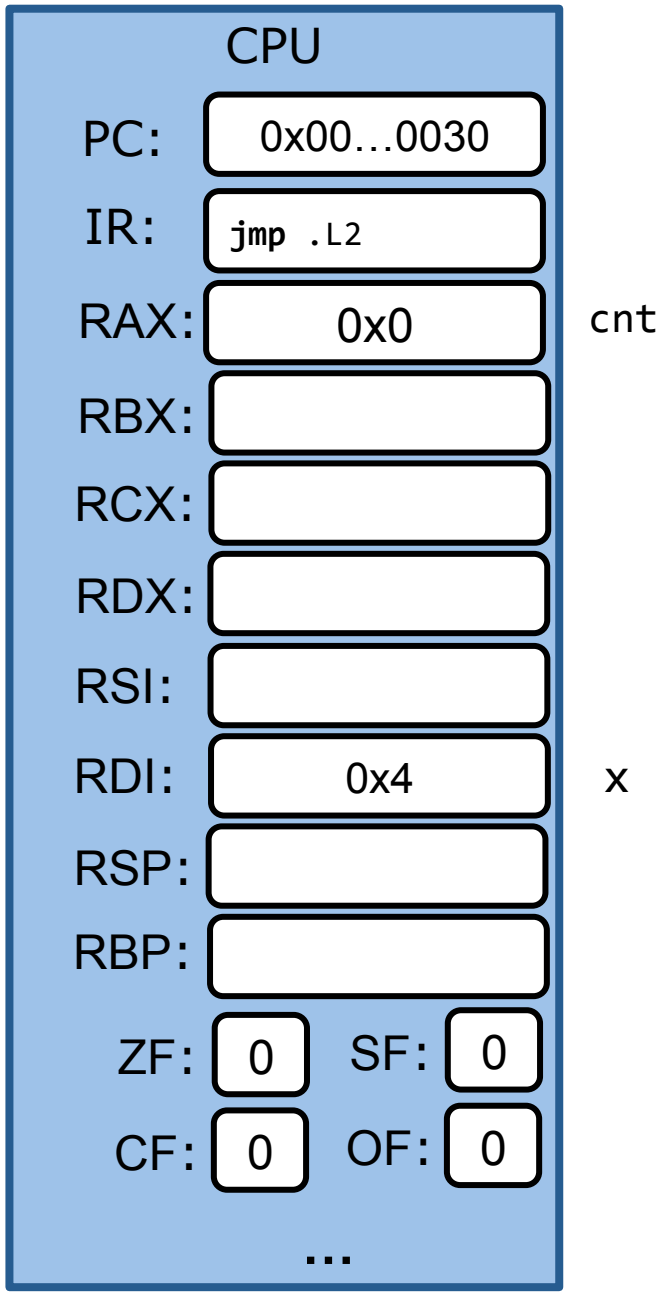





```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

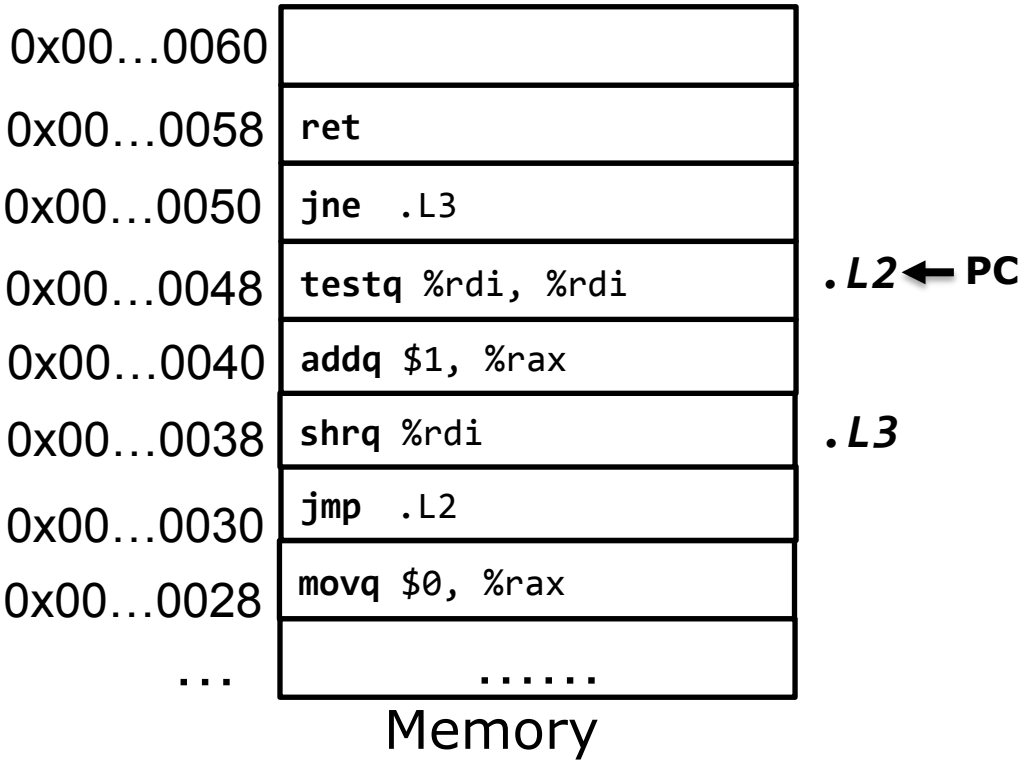
x: 4 $(100)_2$

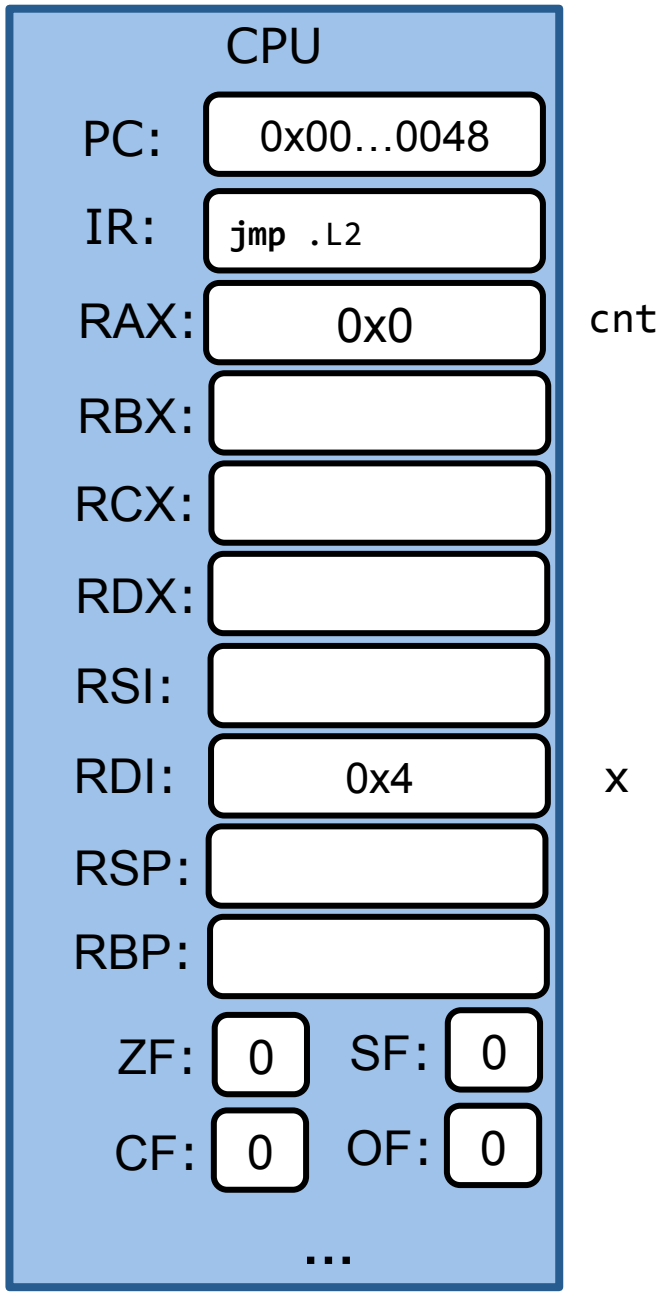





```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

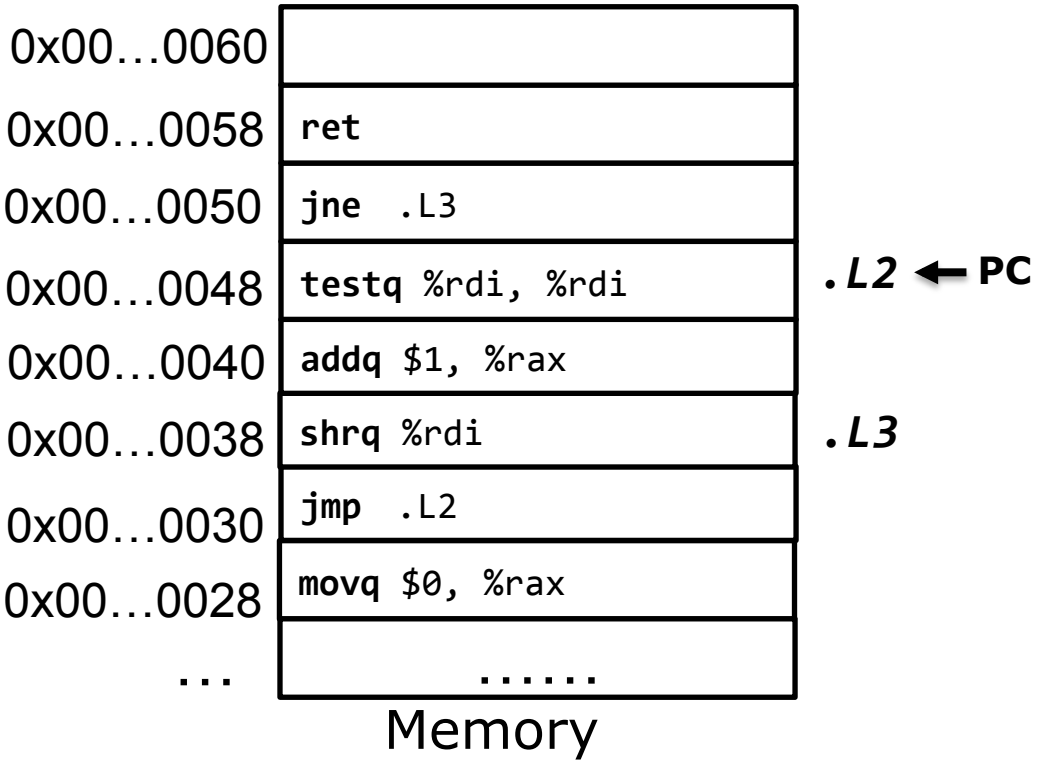
x: 4 $(100)_2$

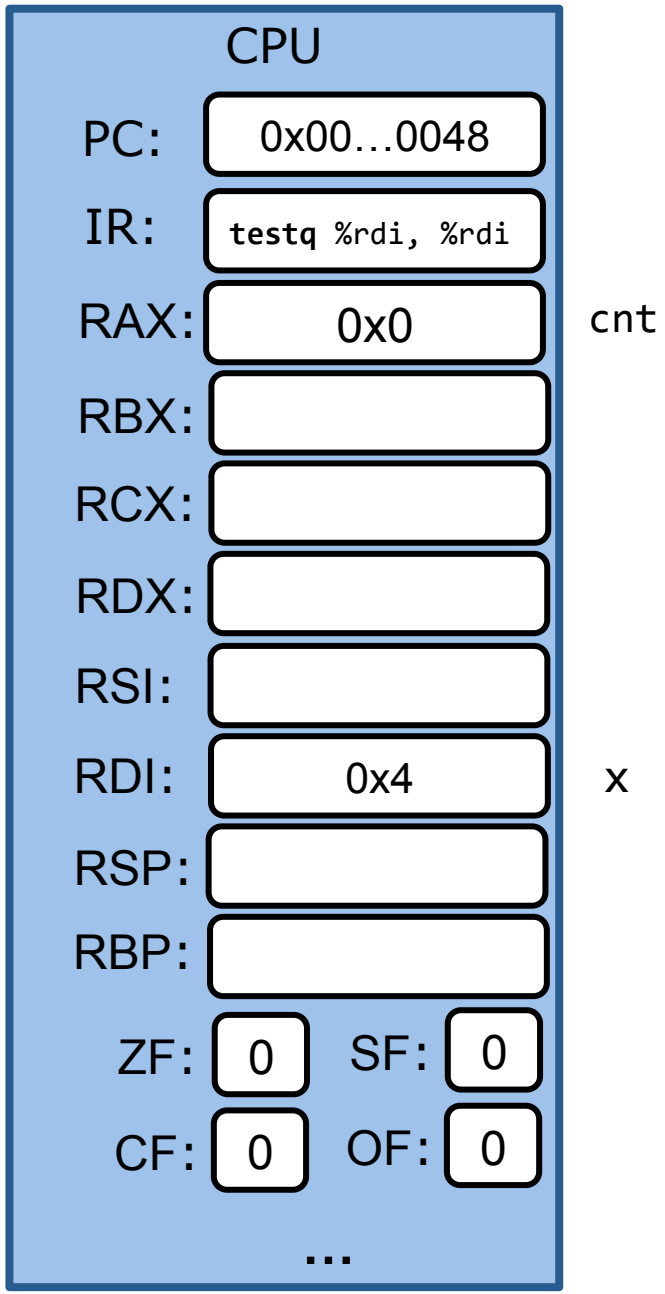





```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return log;
}
```

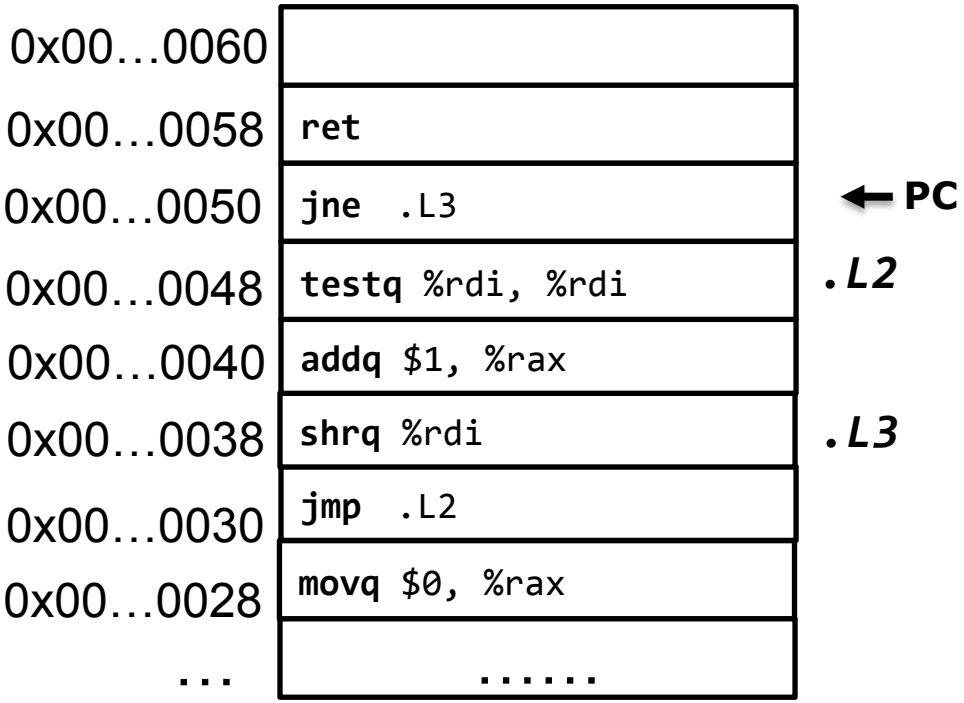
x: 4 $(100)_2$

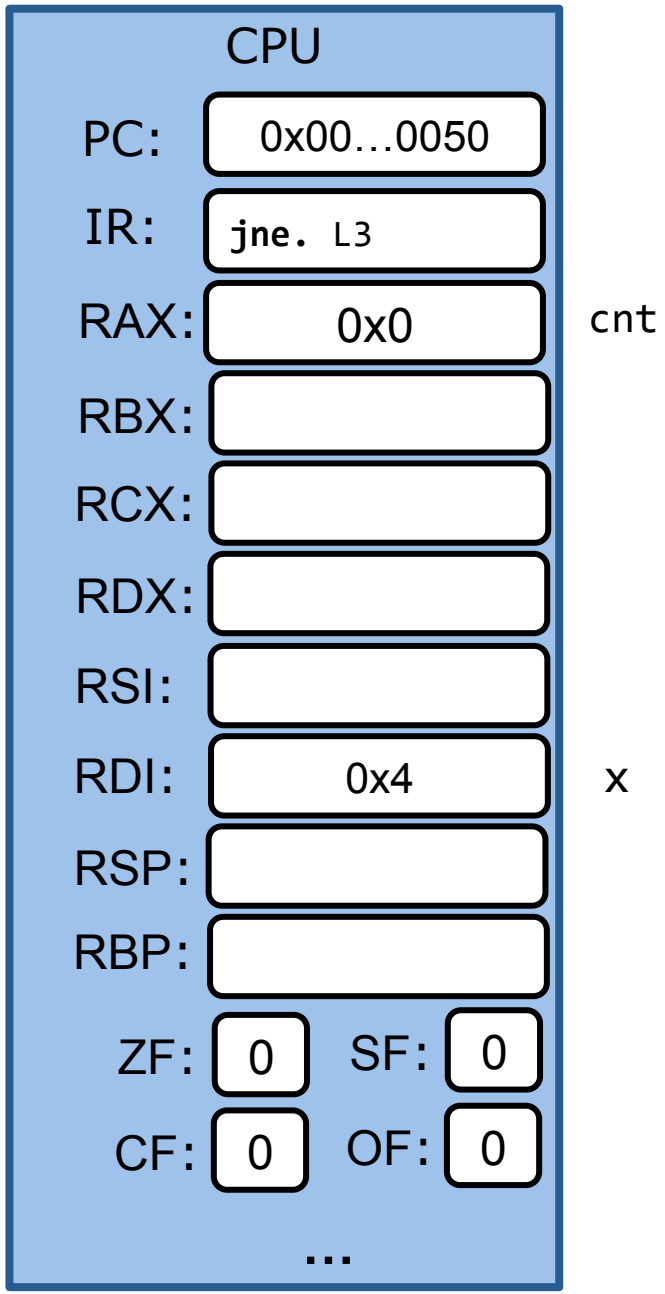





```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

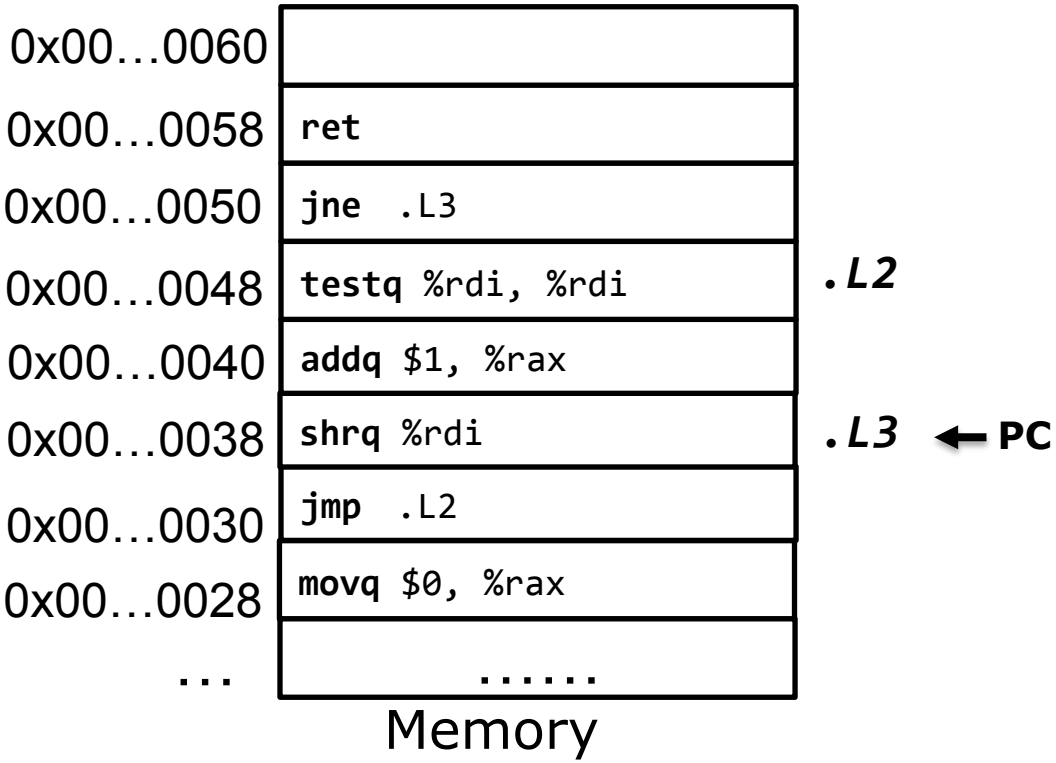


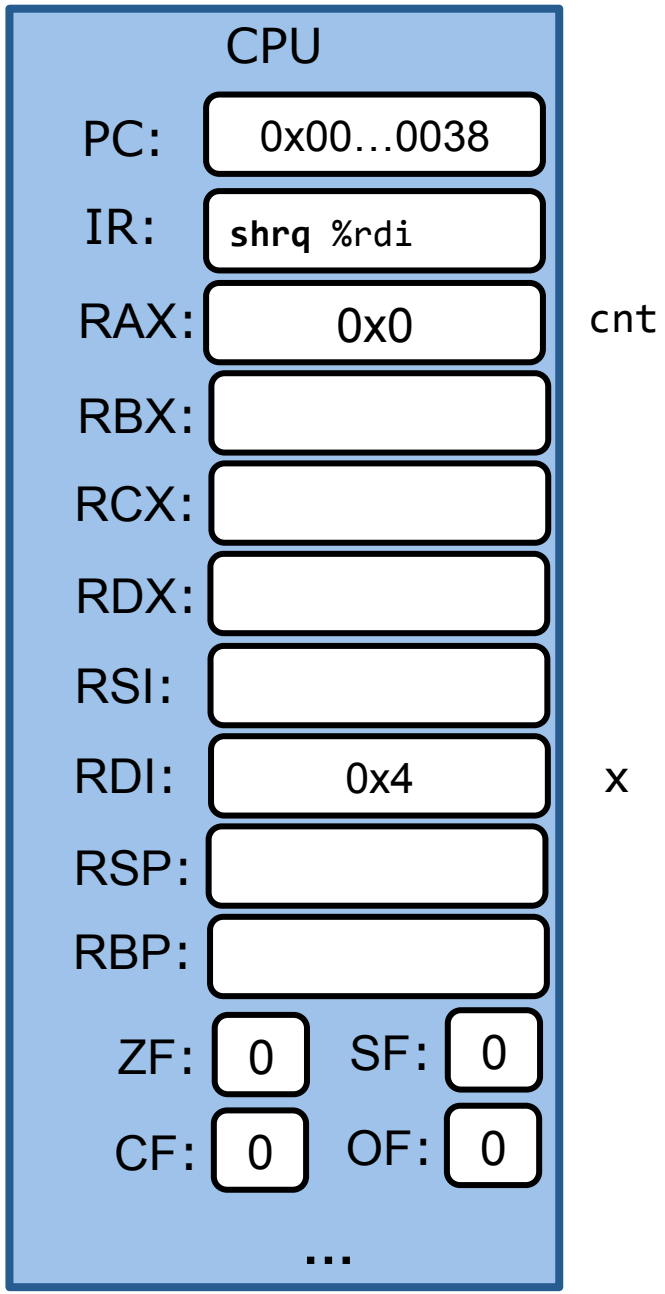



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

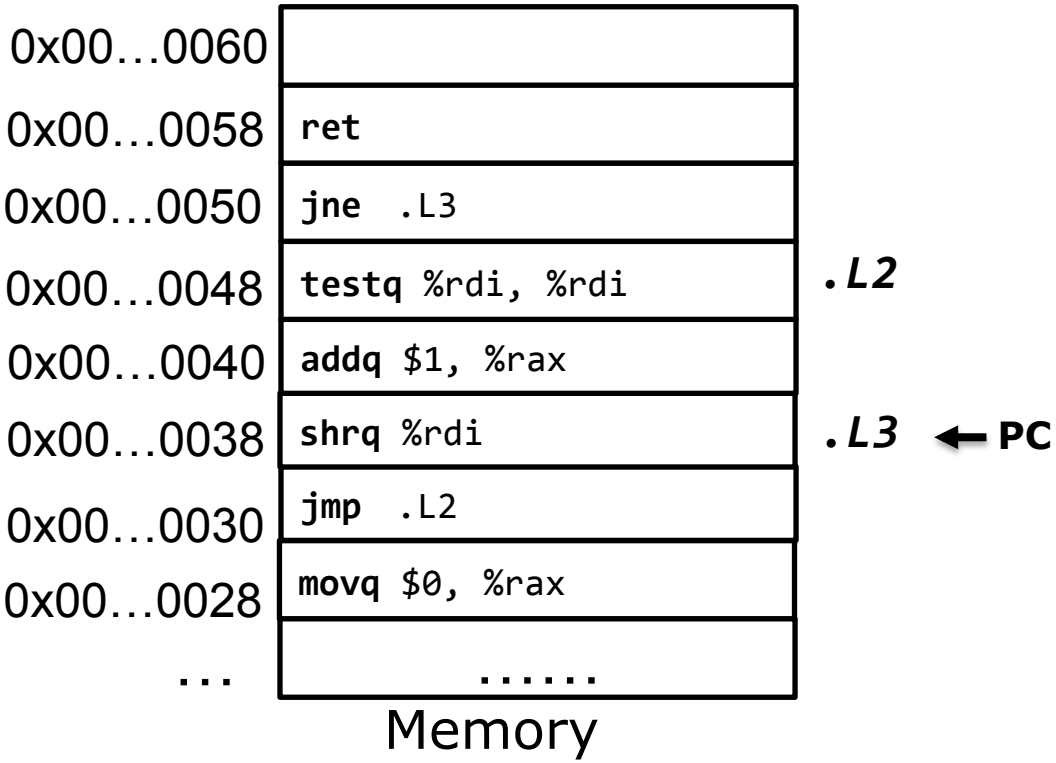


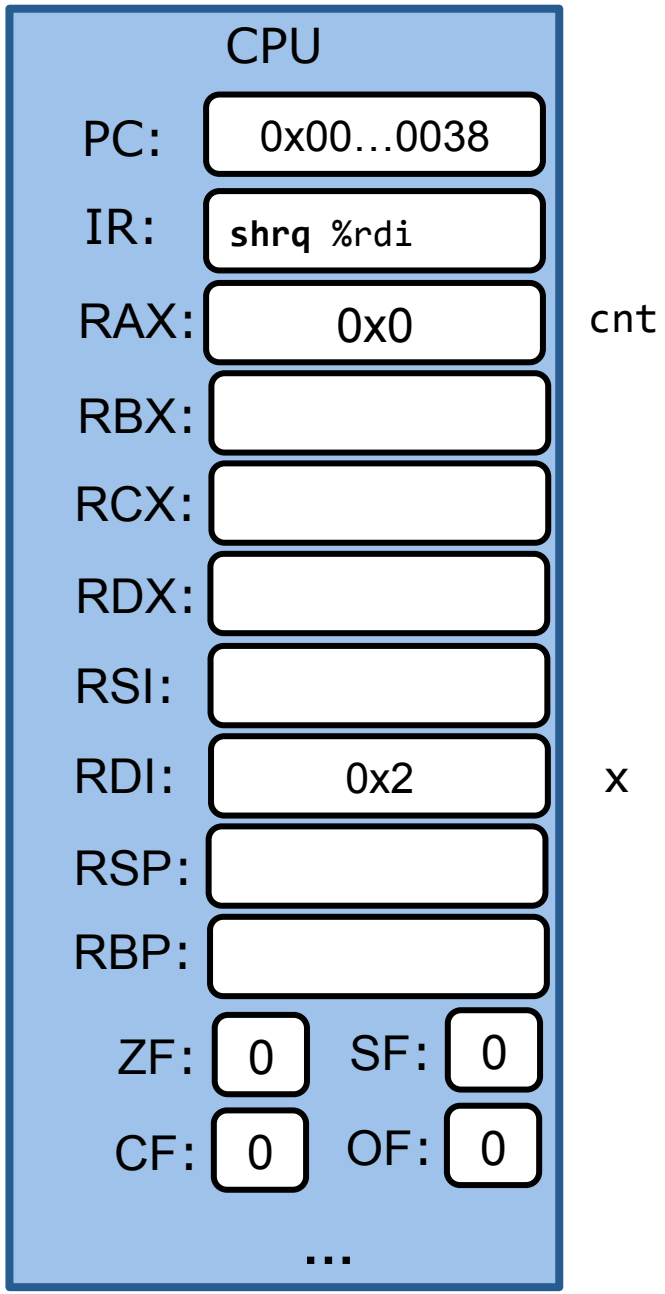



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

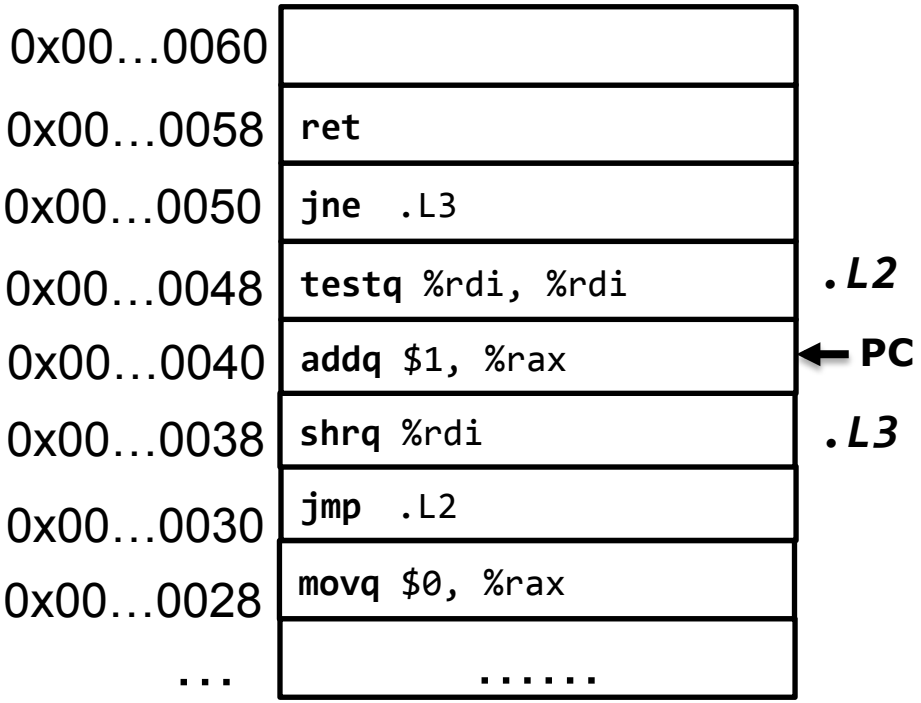


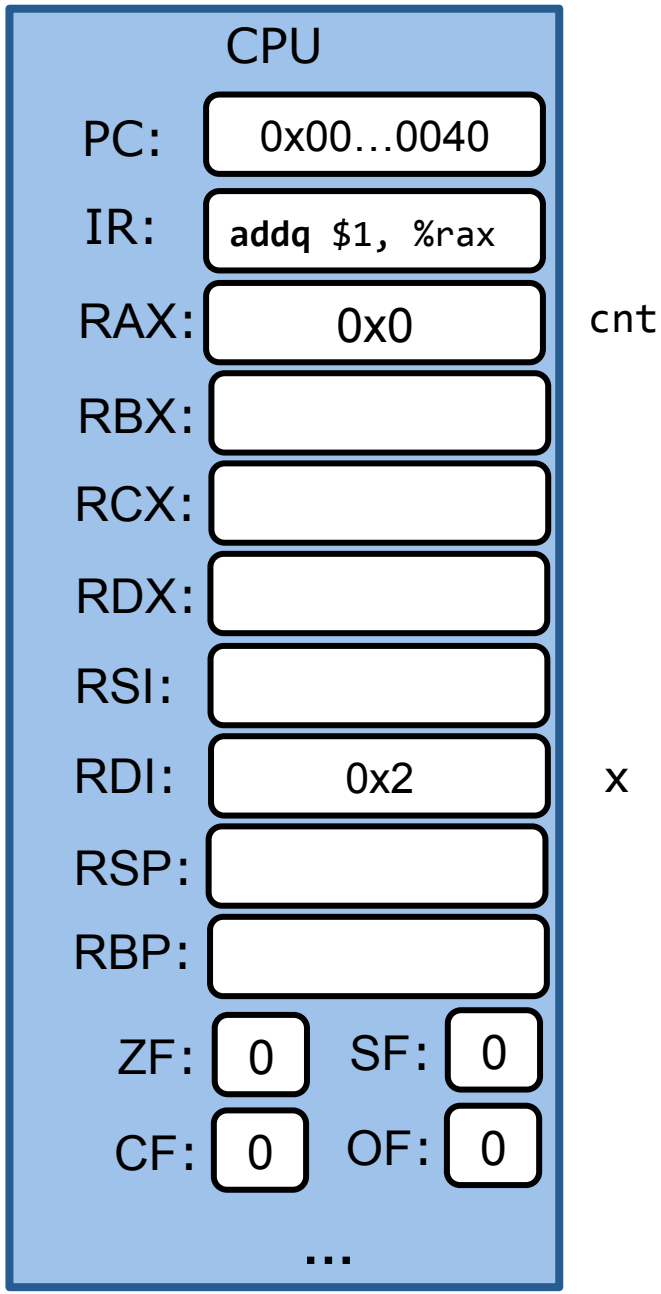



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

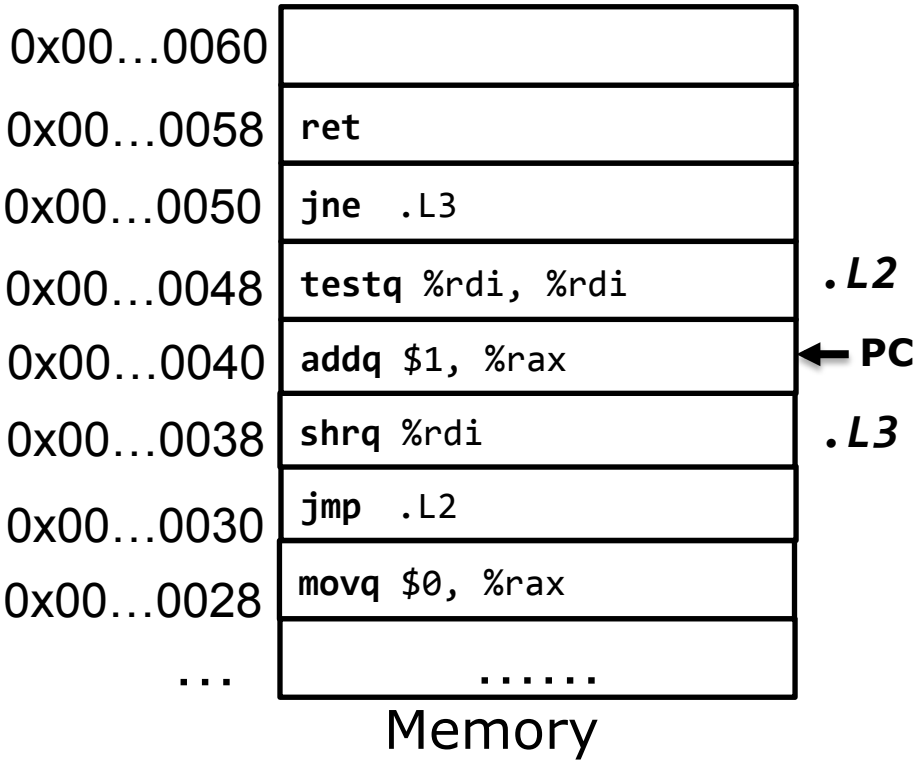


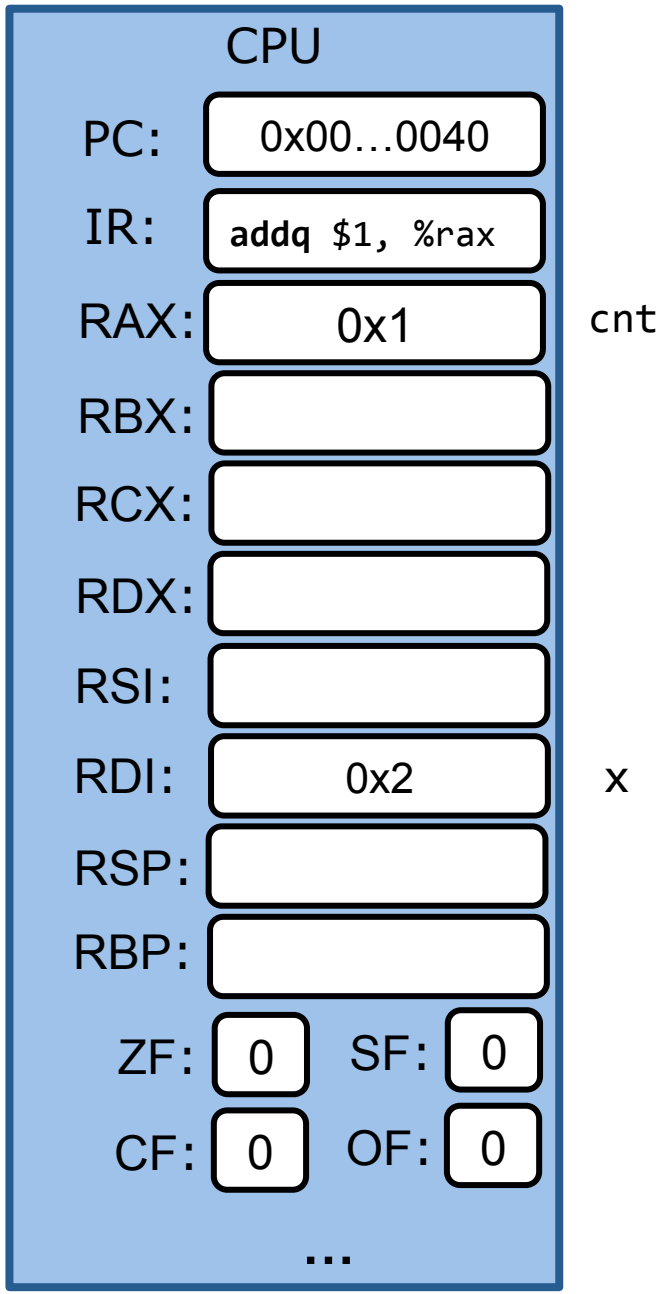



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

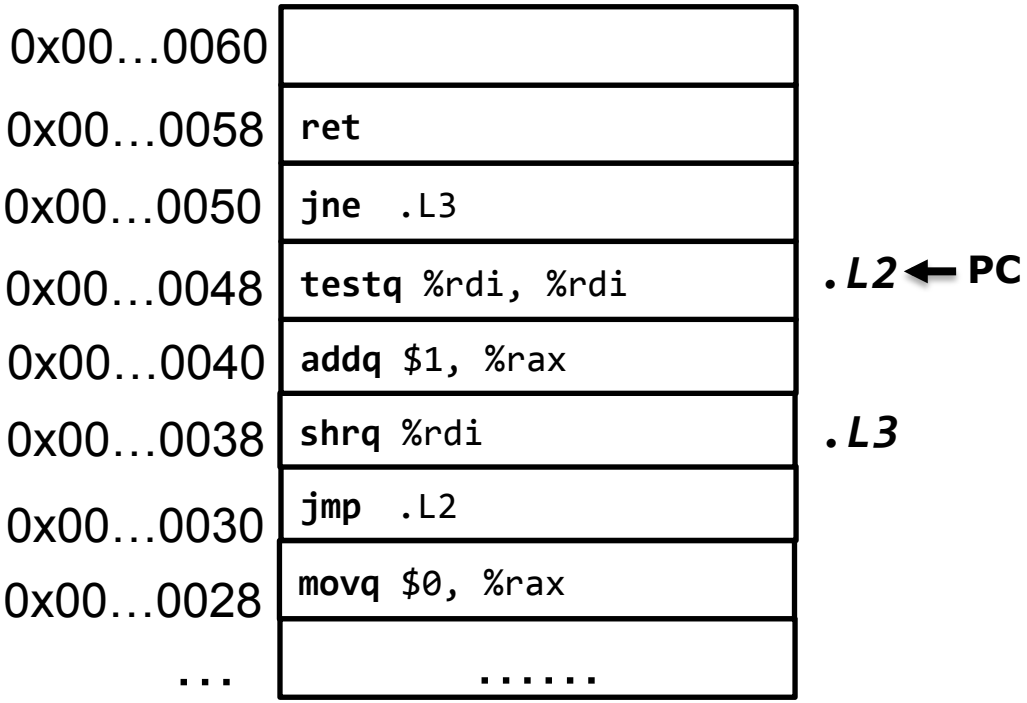


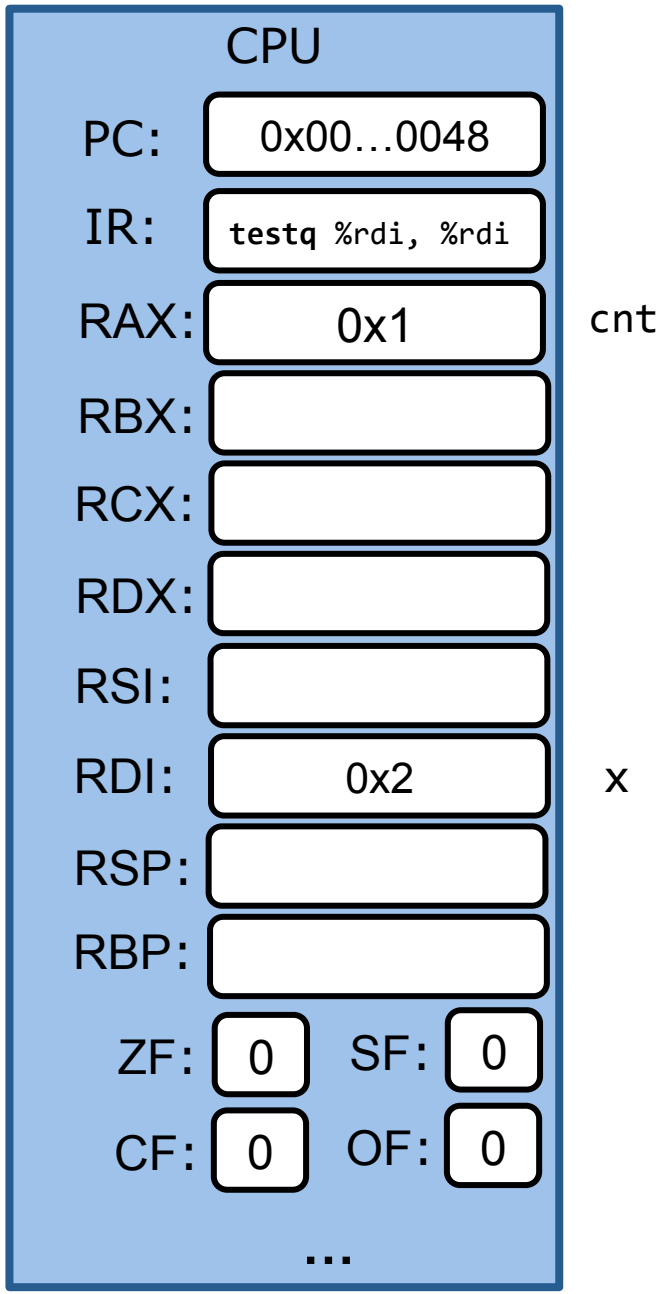



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

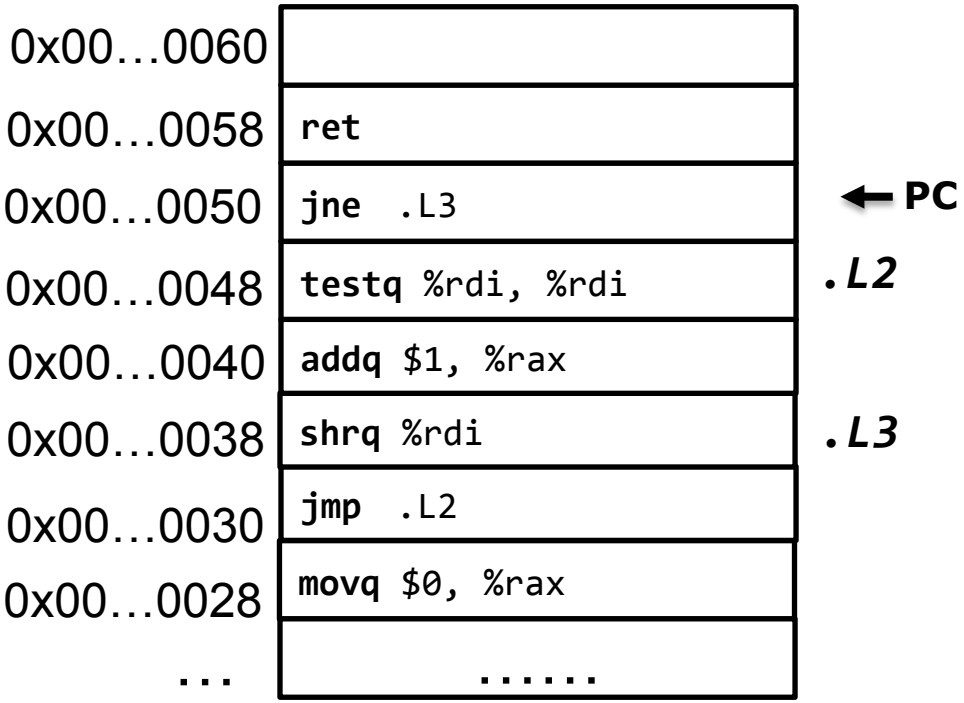


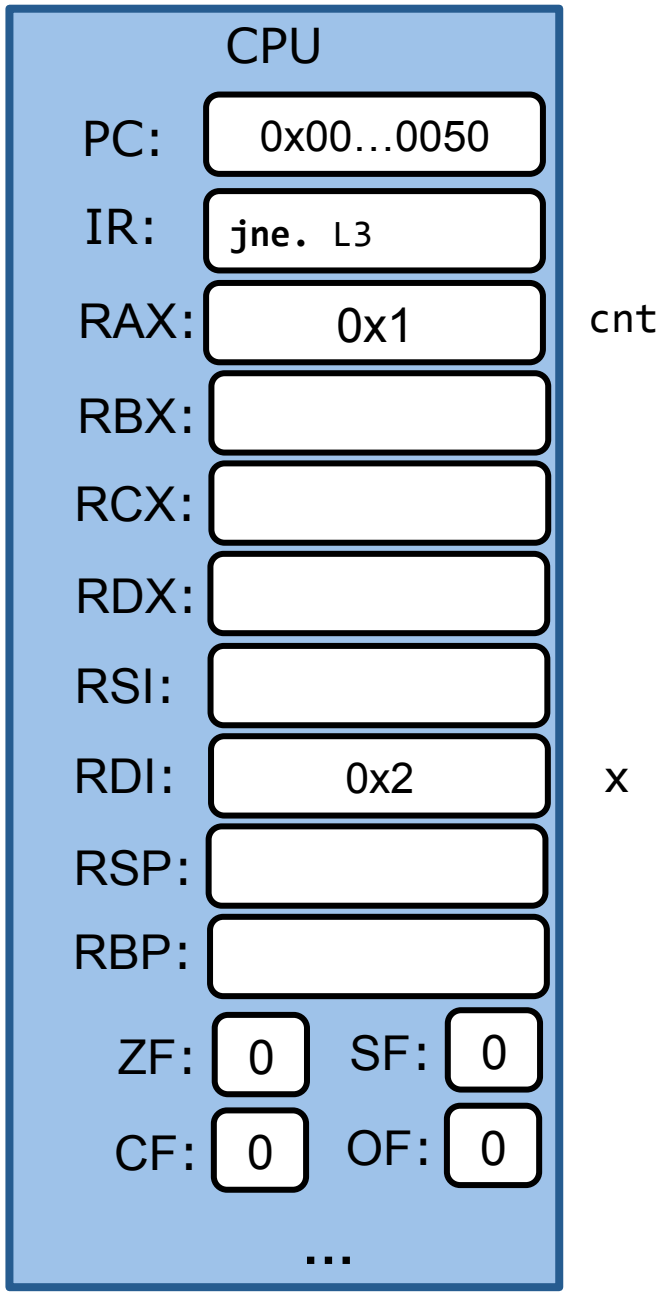



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

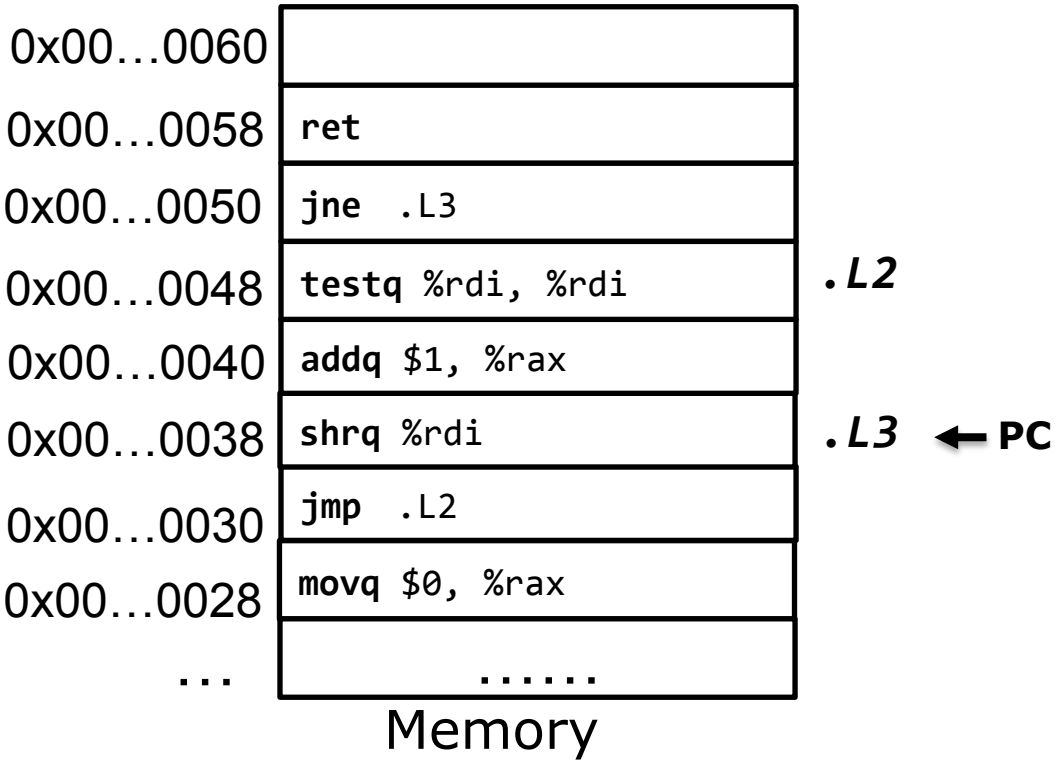


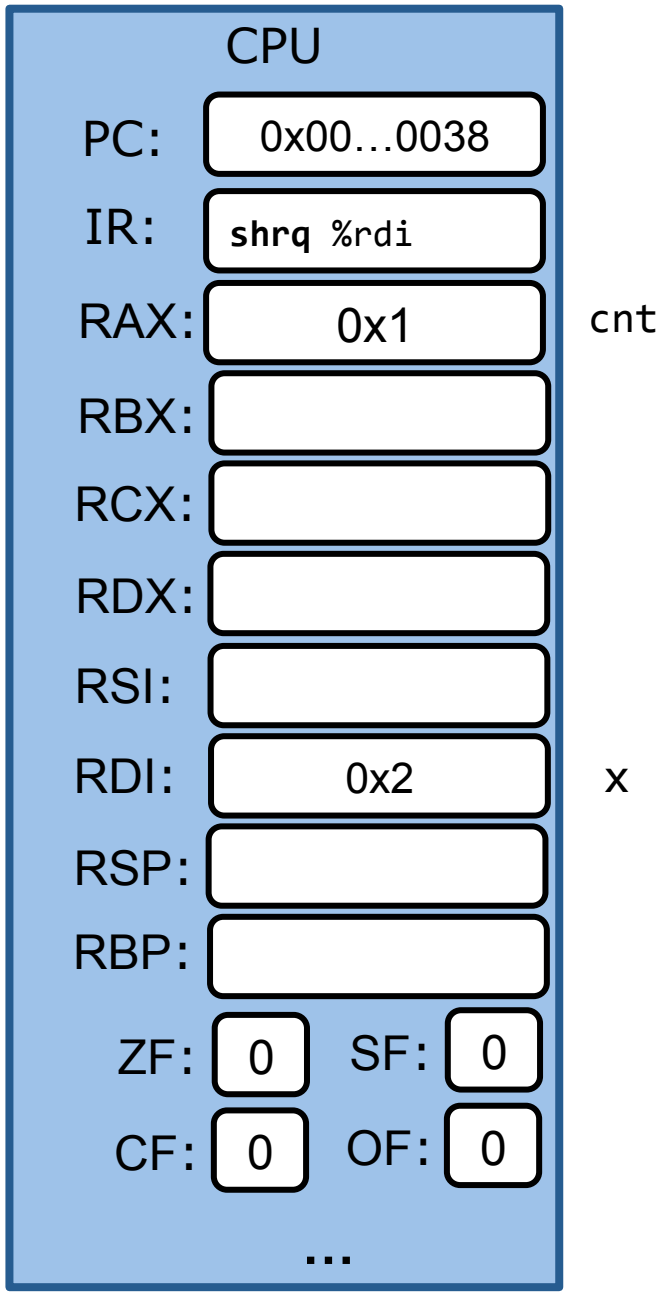



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

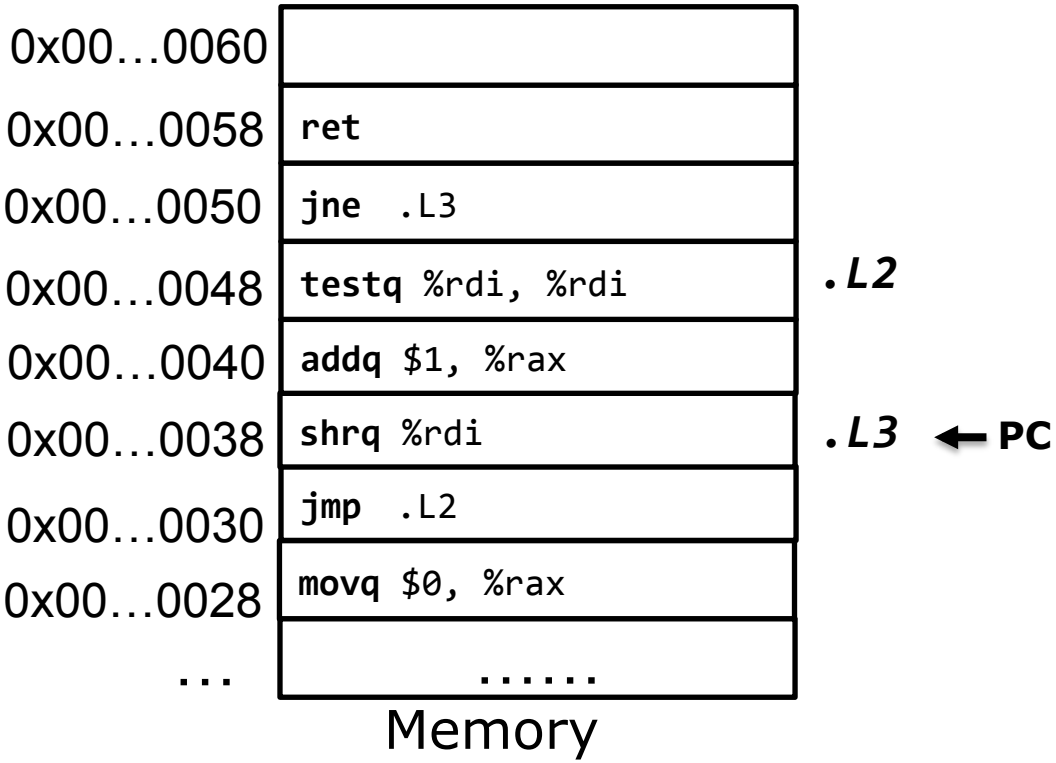


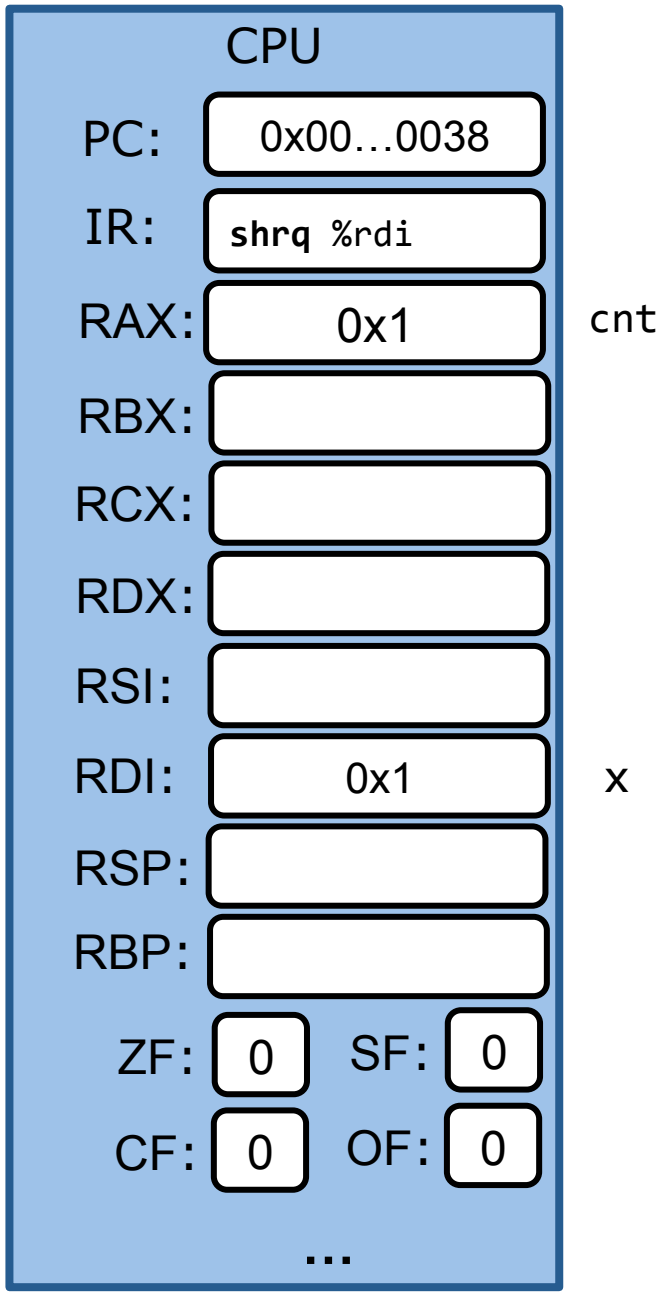



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

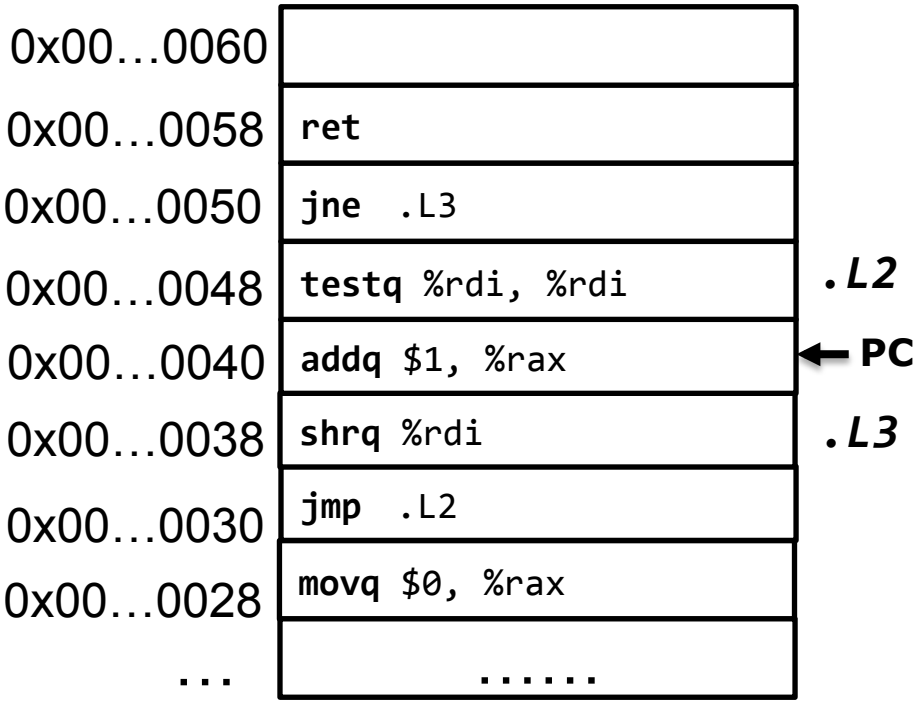


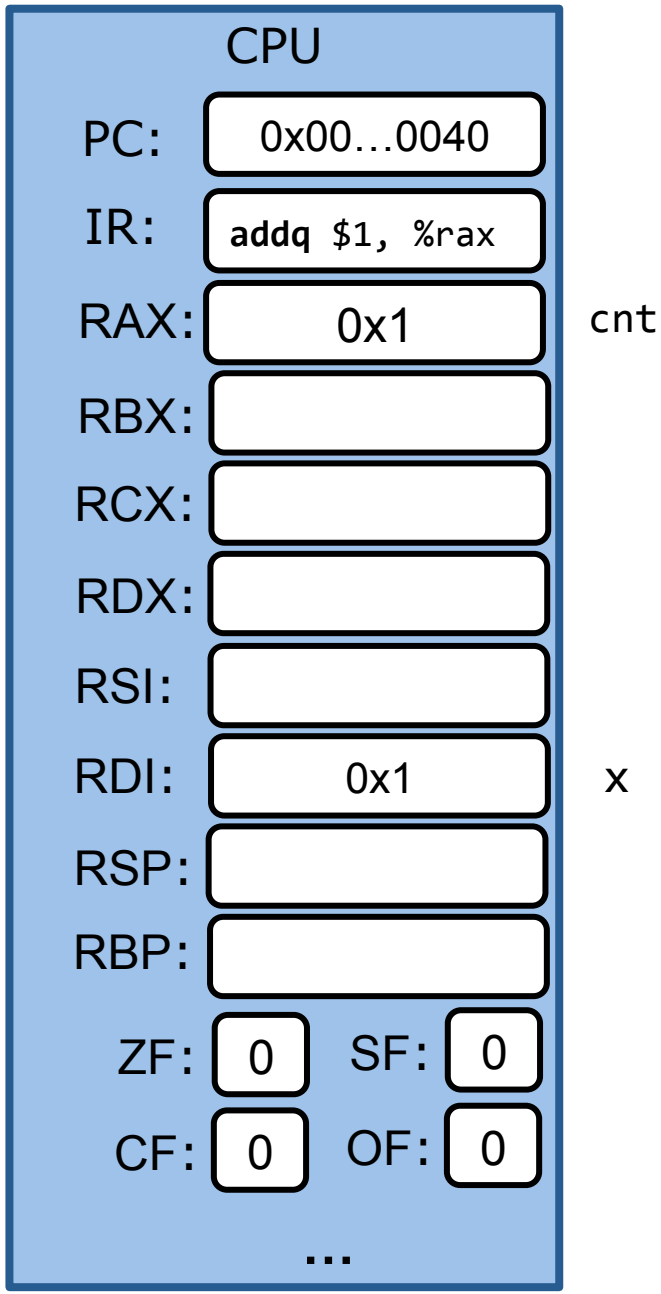



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

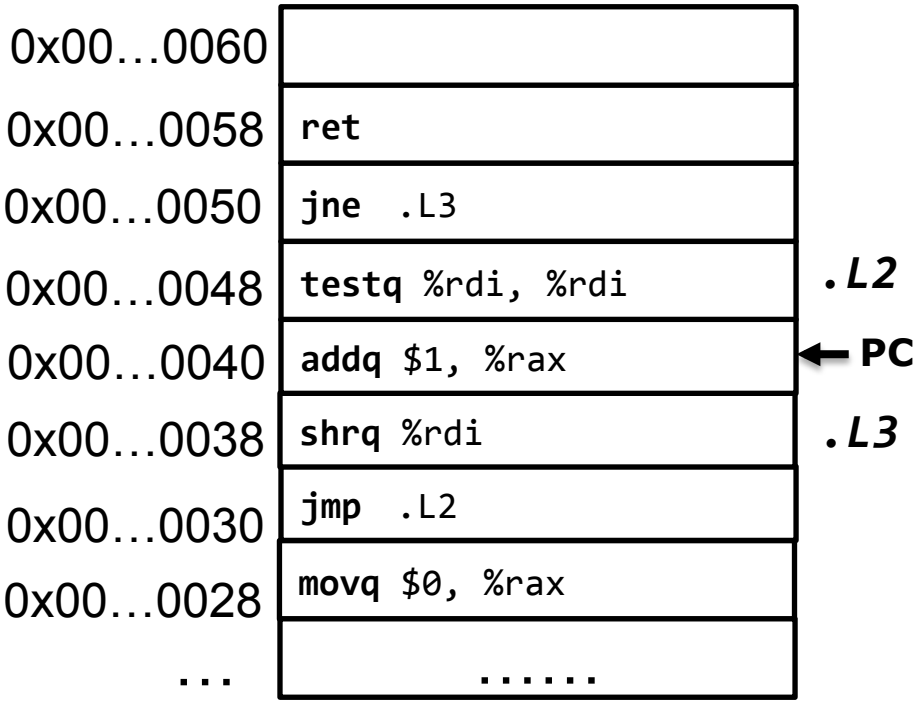


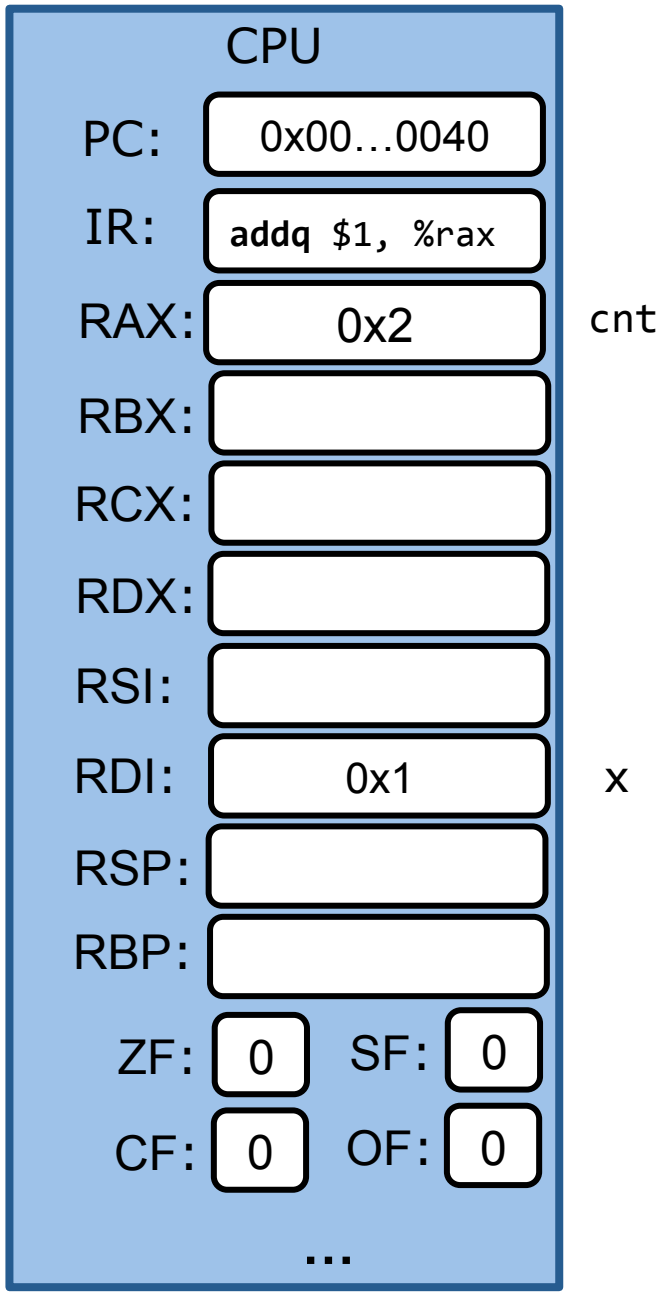



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

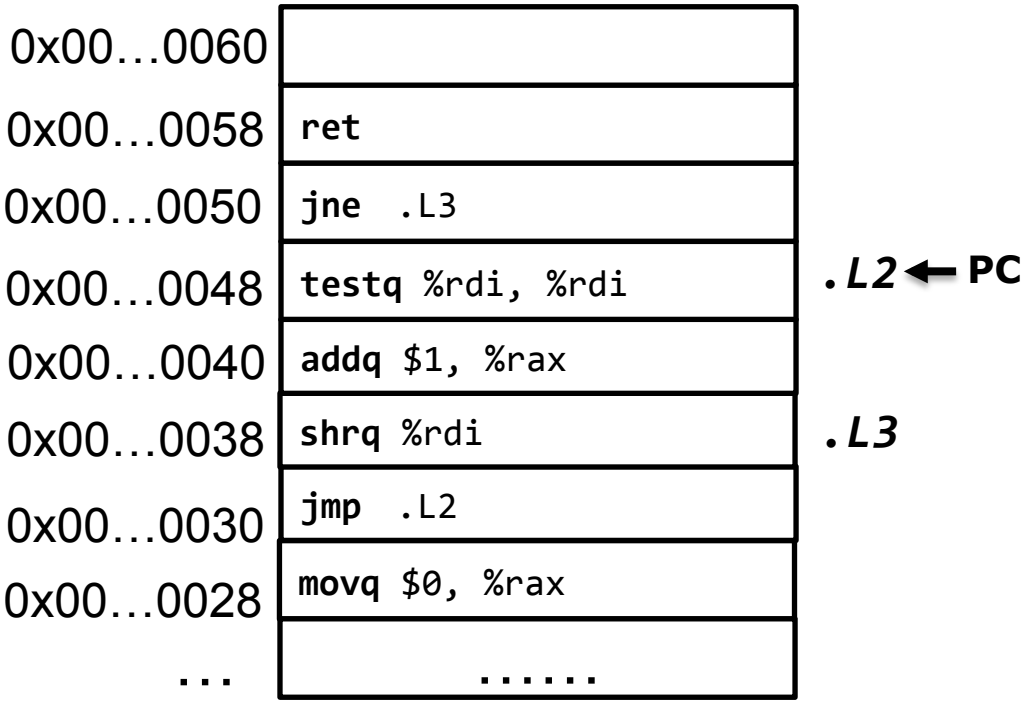


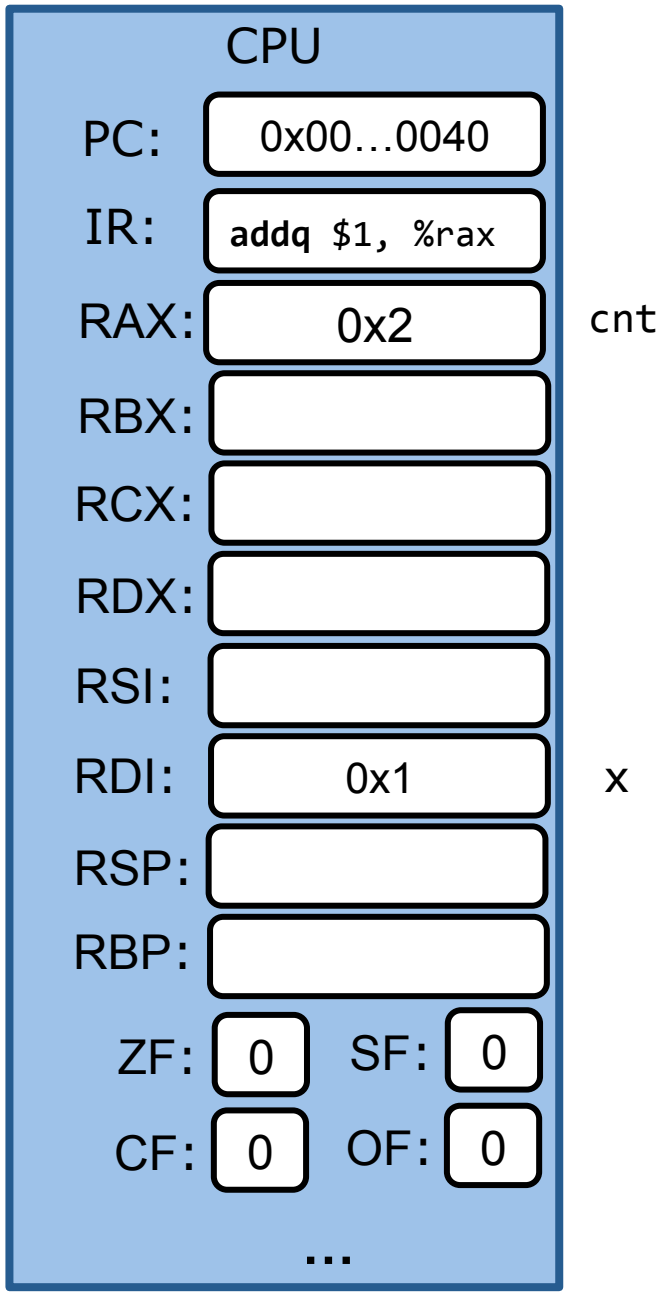



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

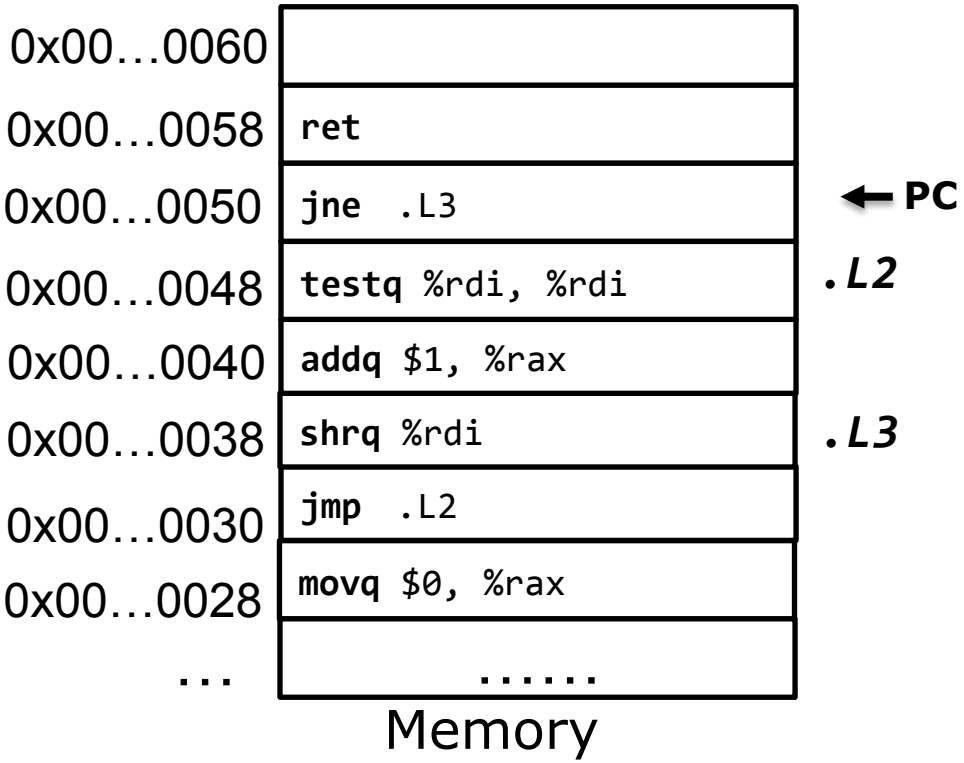


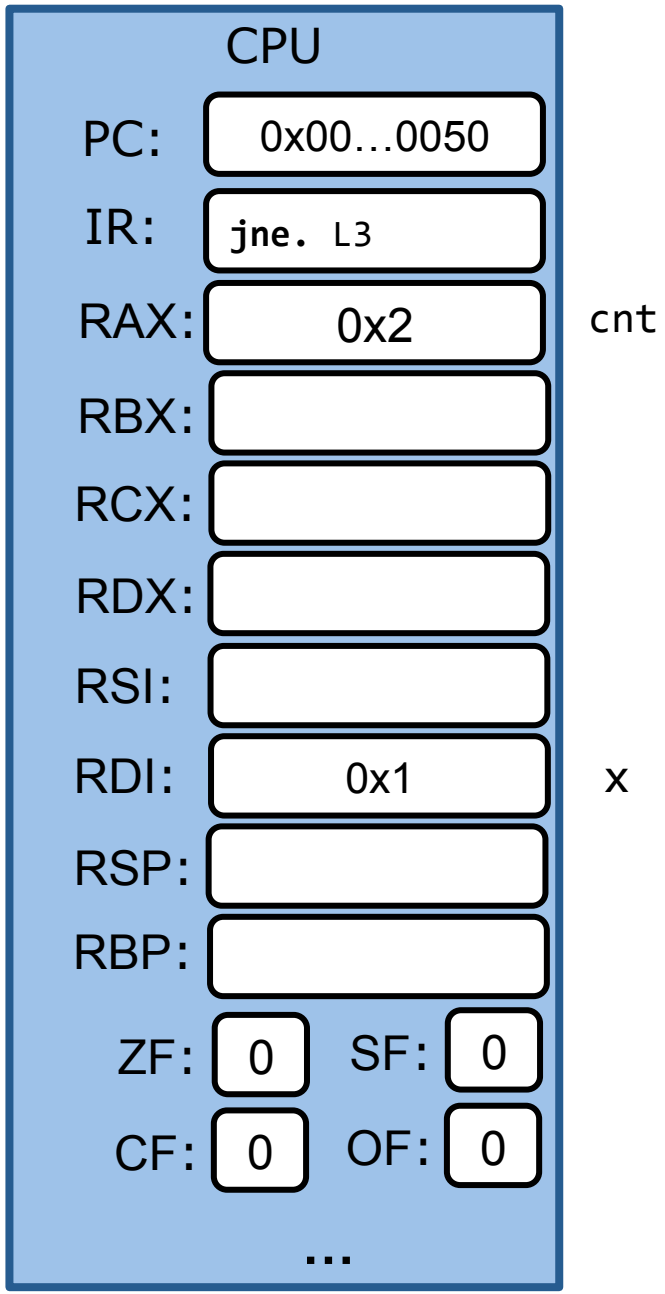



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

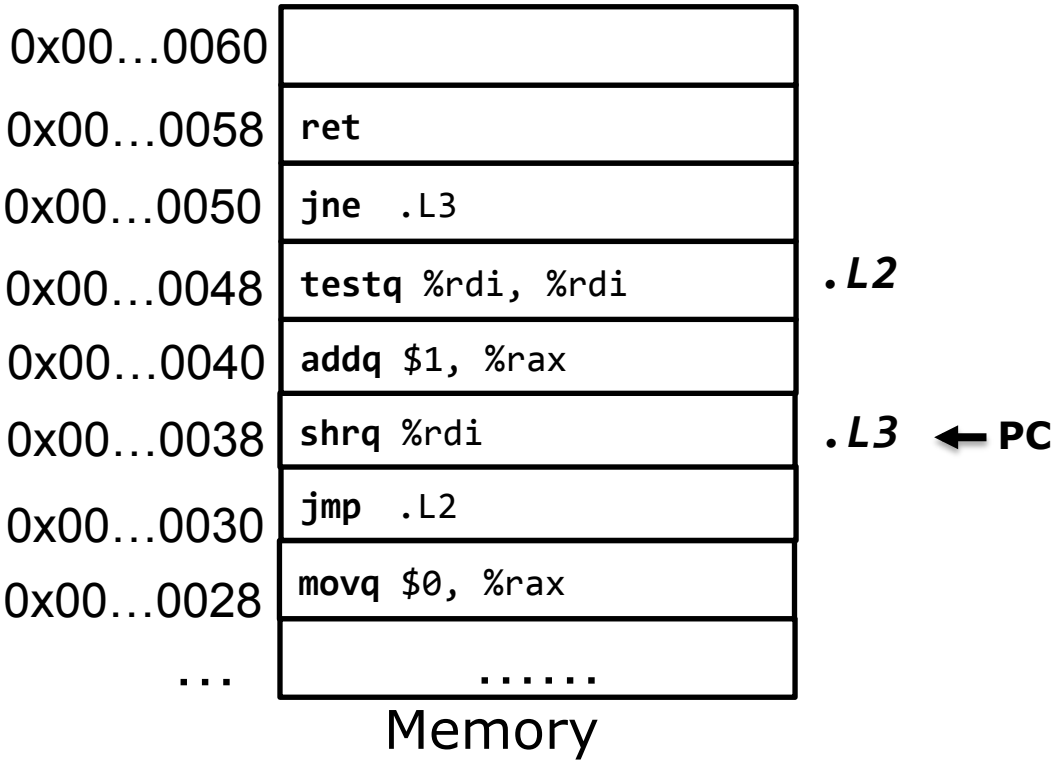


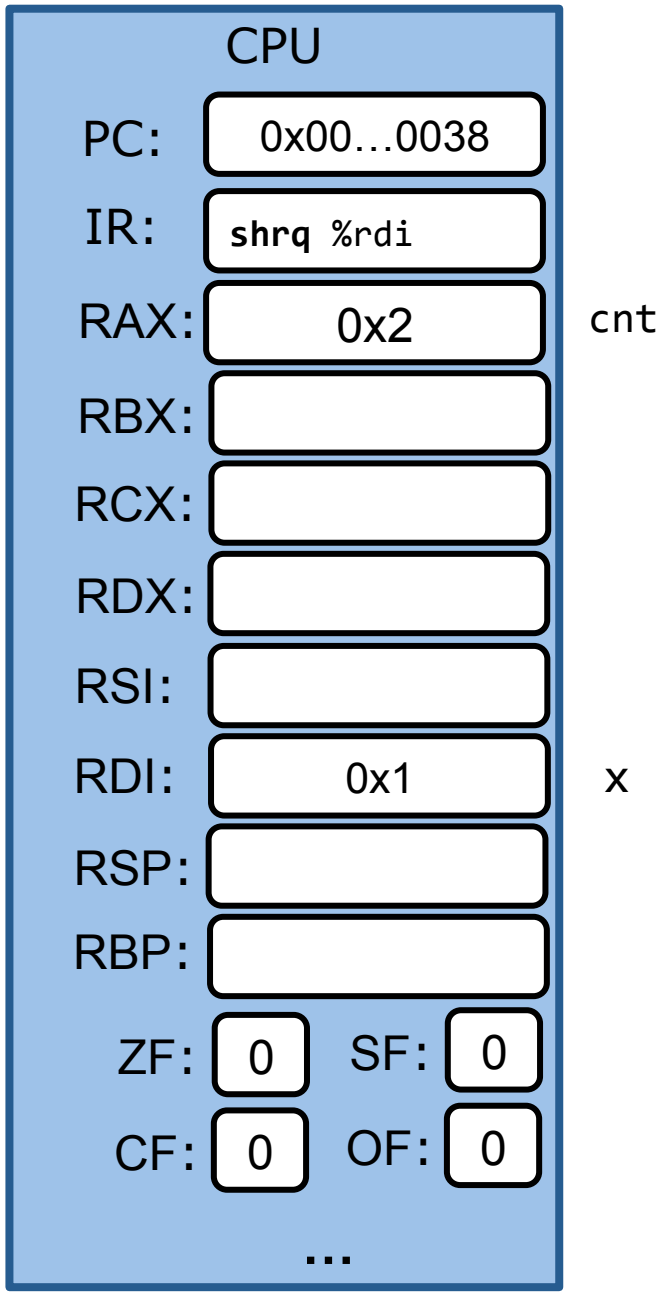



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

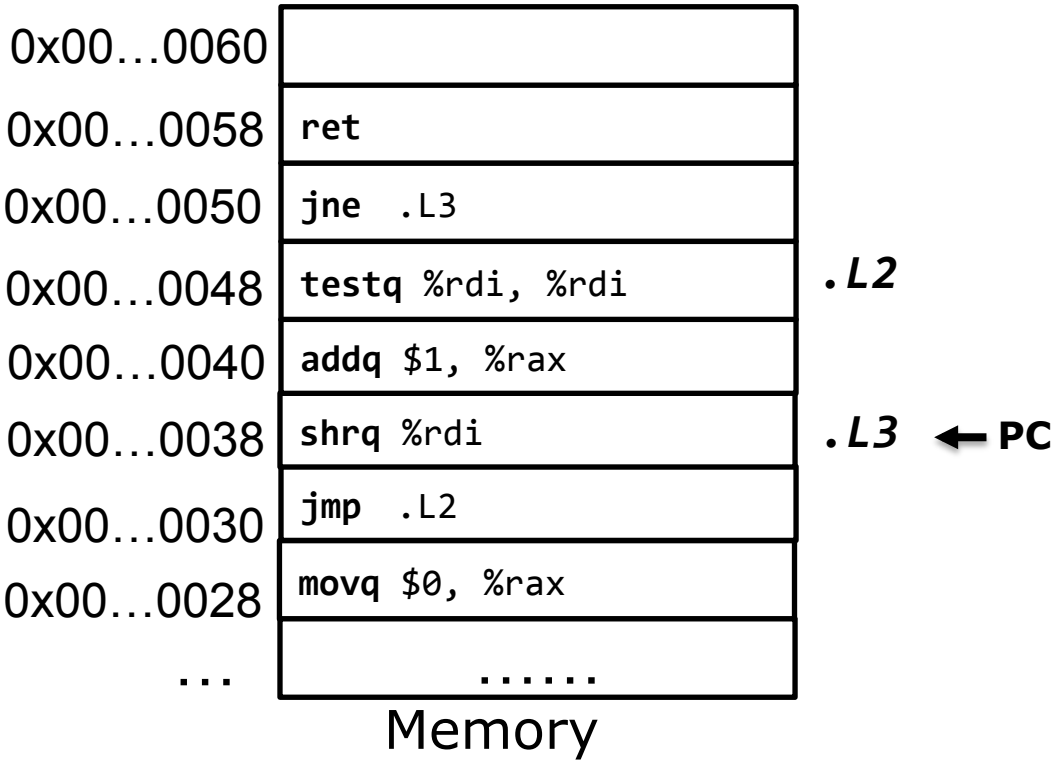


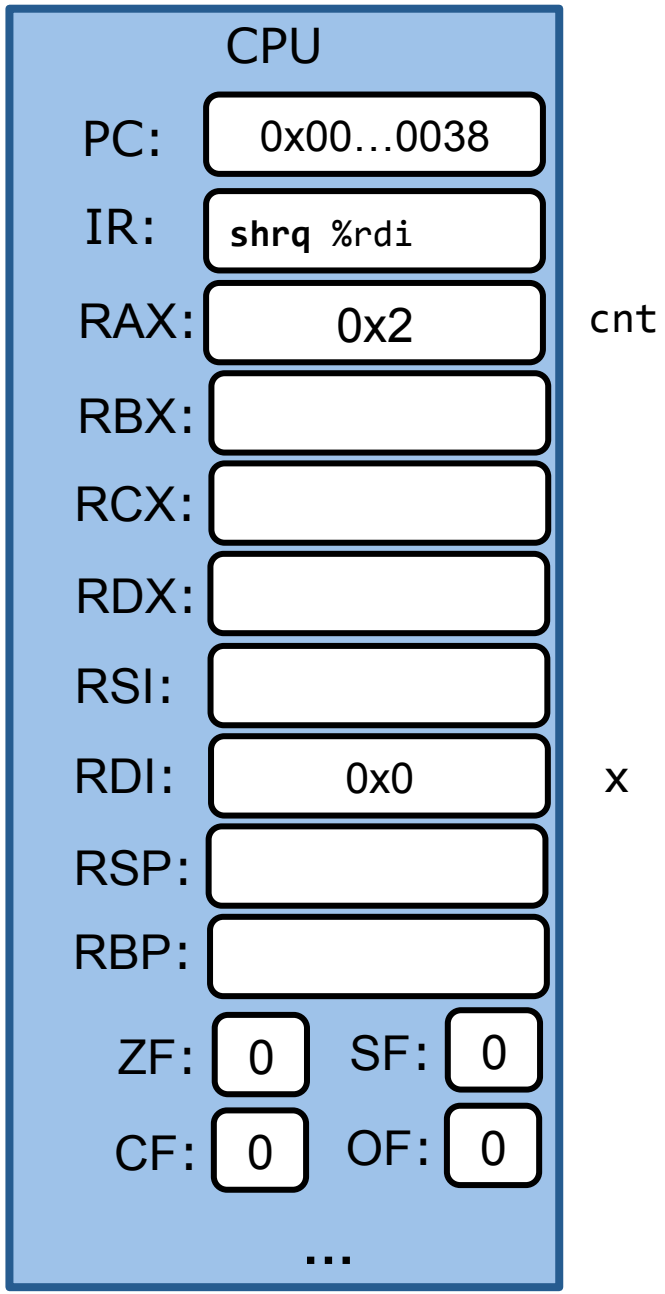



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

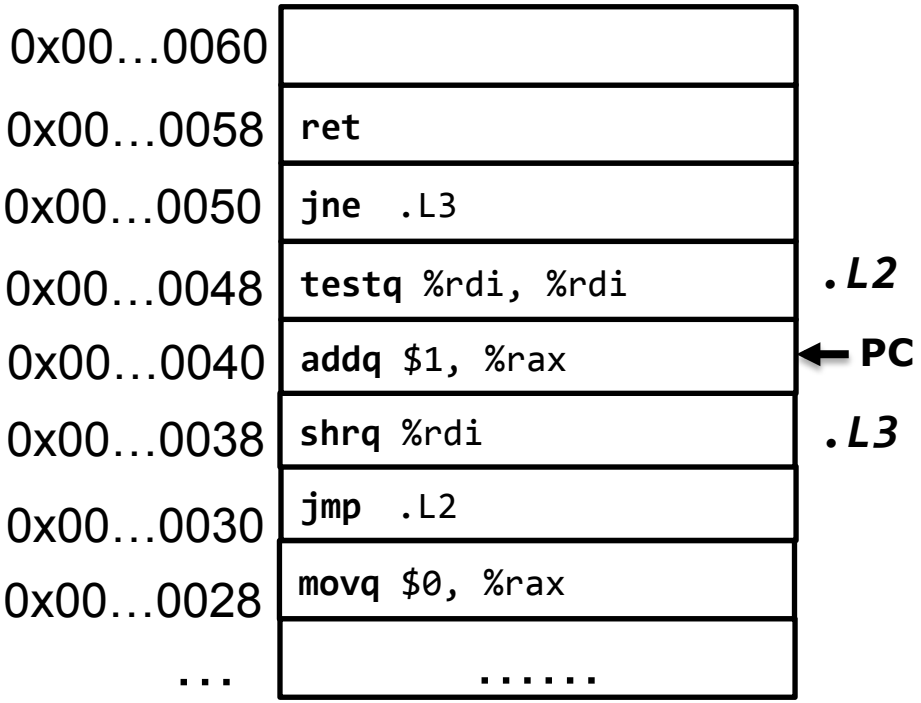


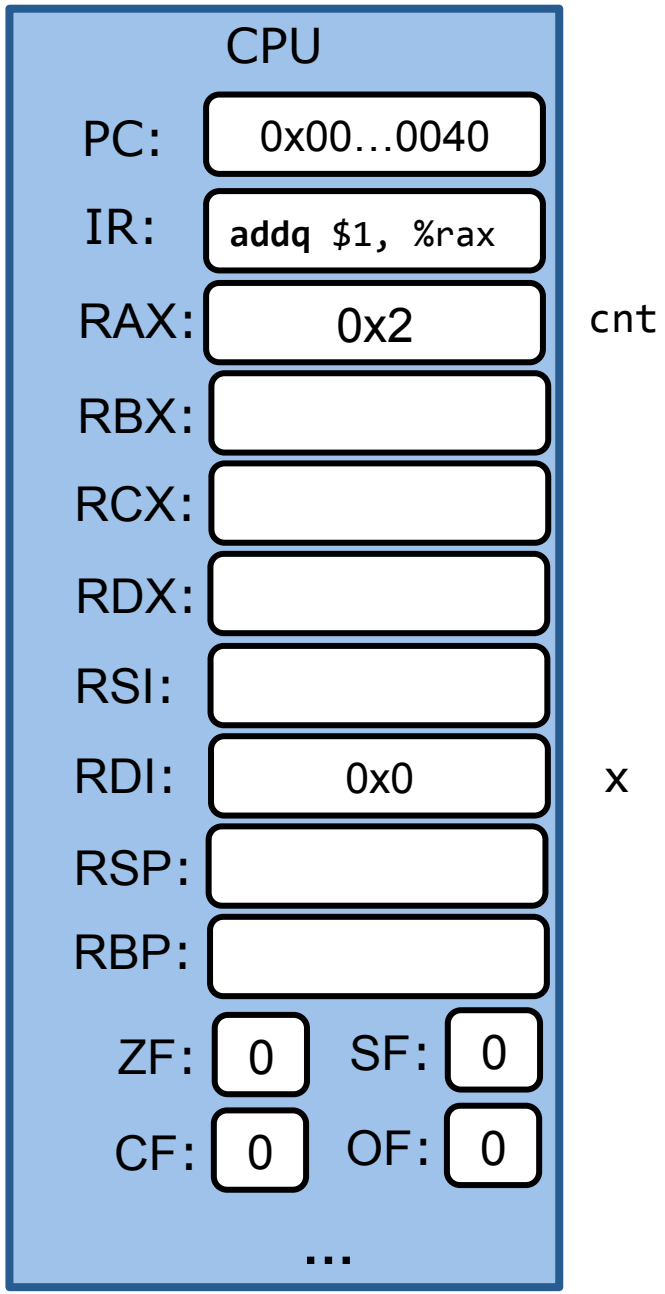



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

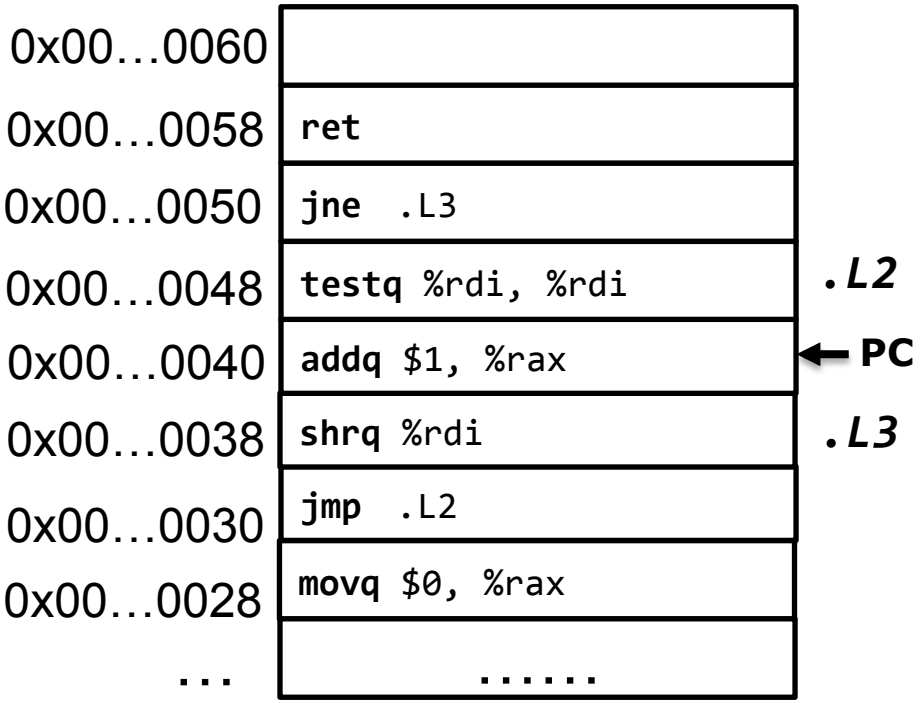


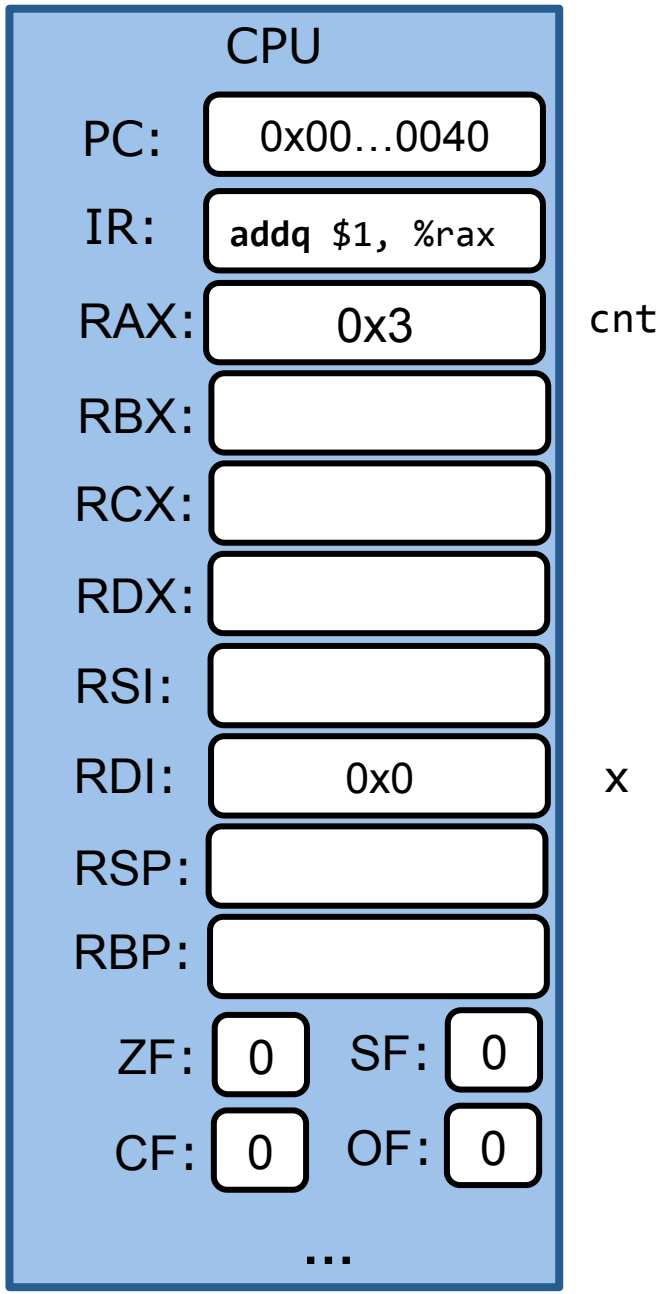



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

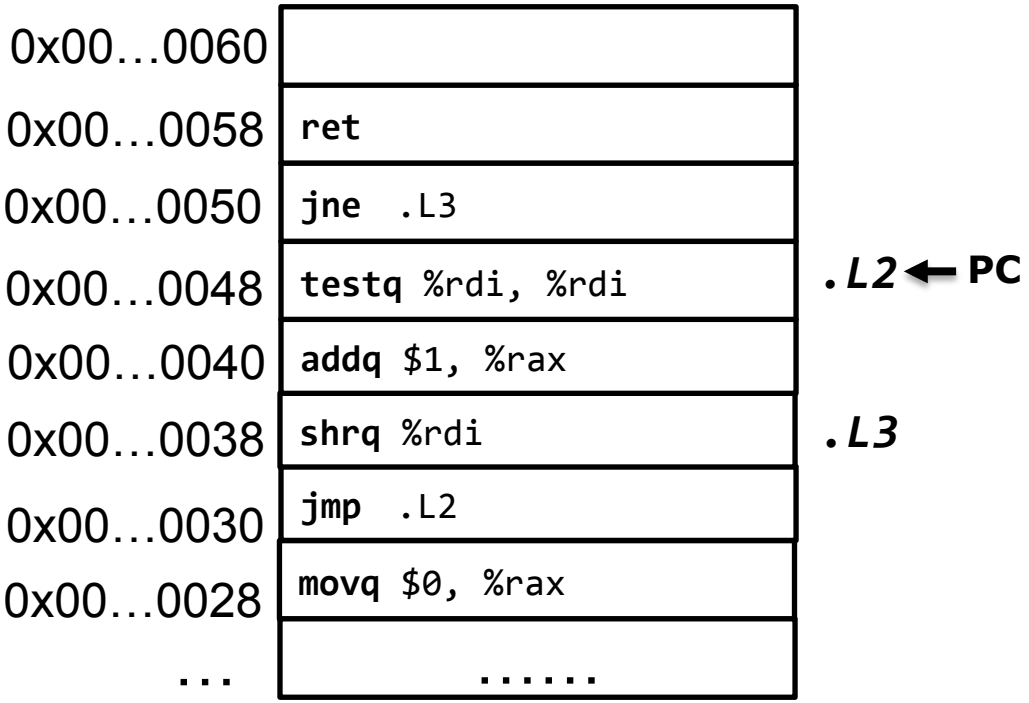


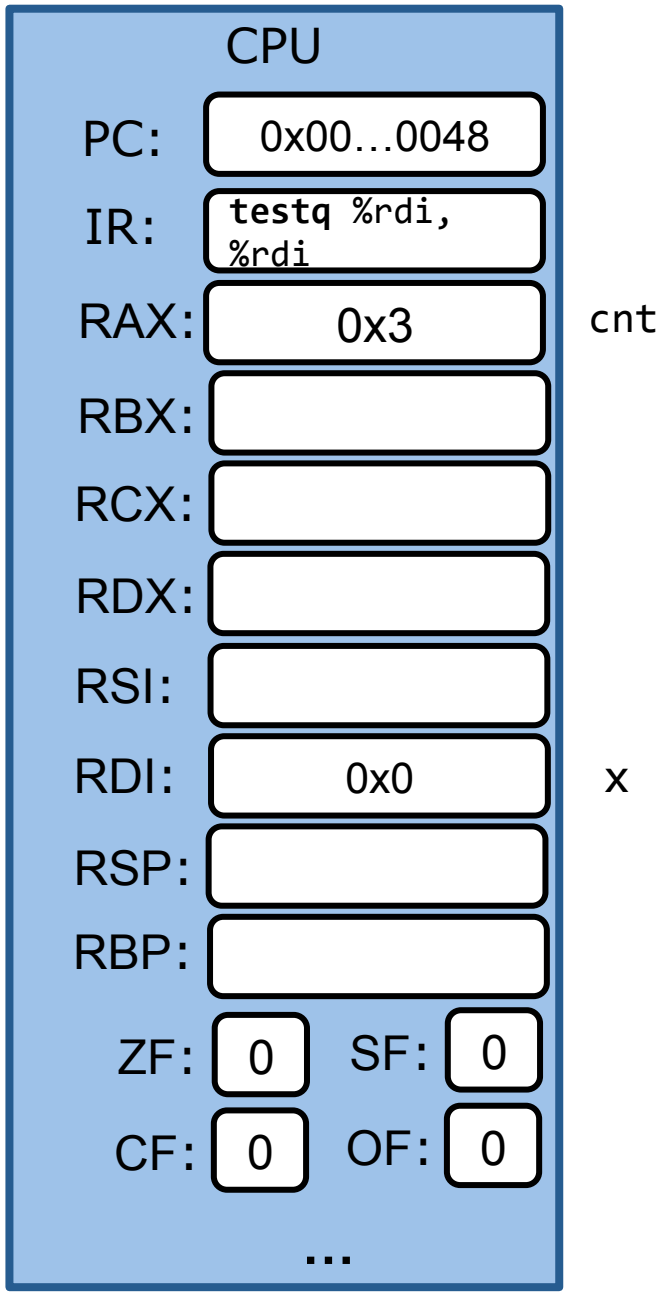



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

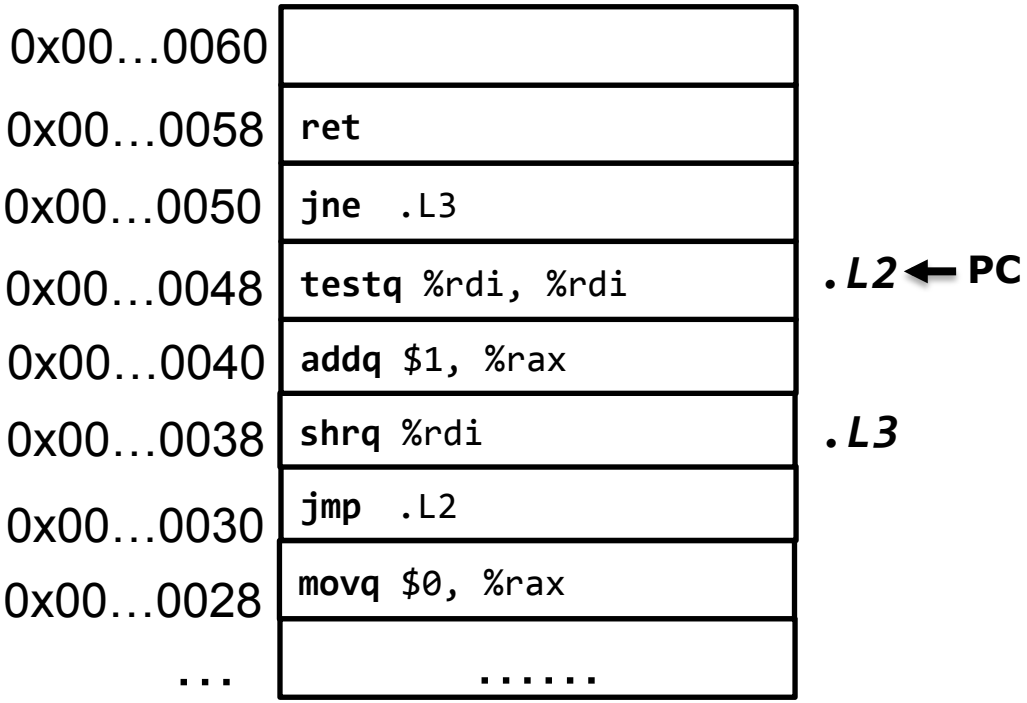


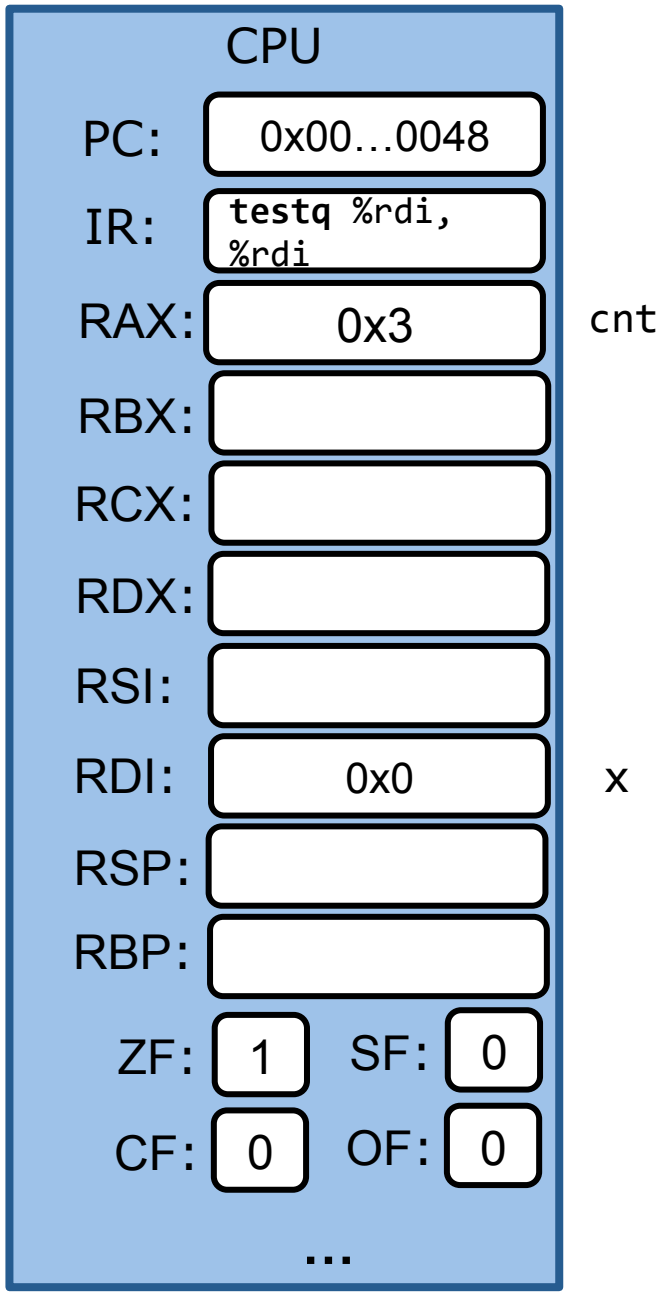



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

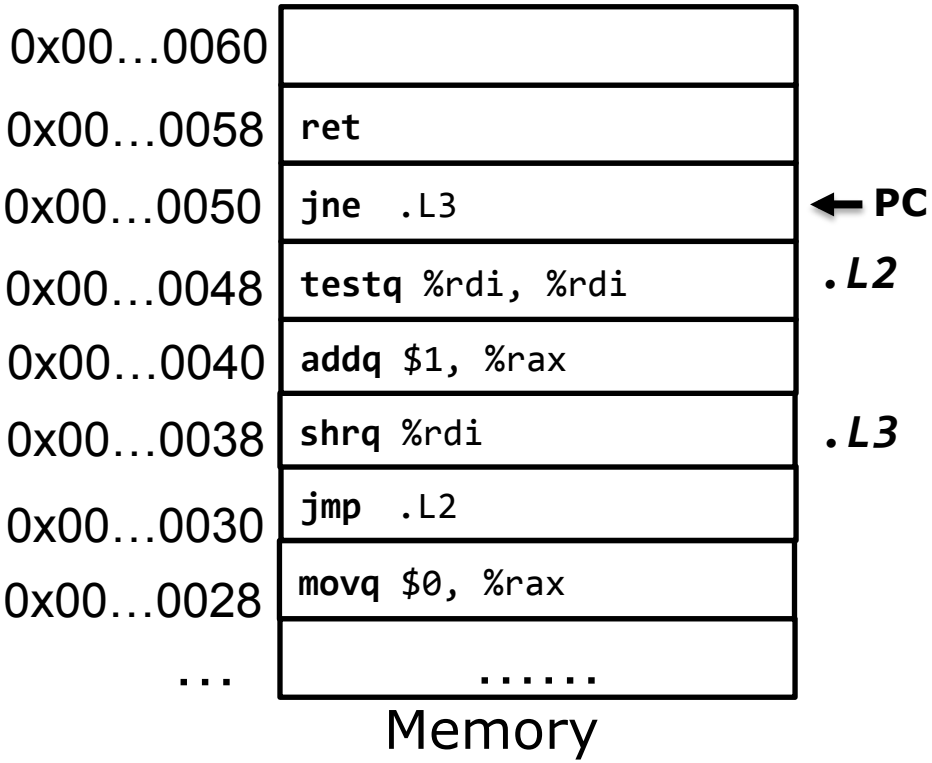


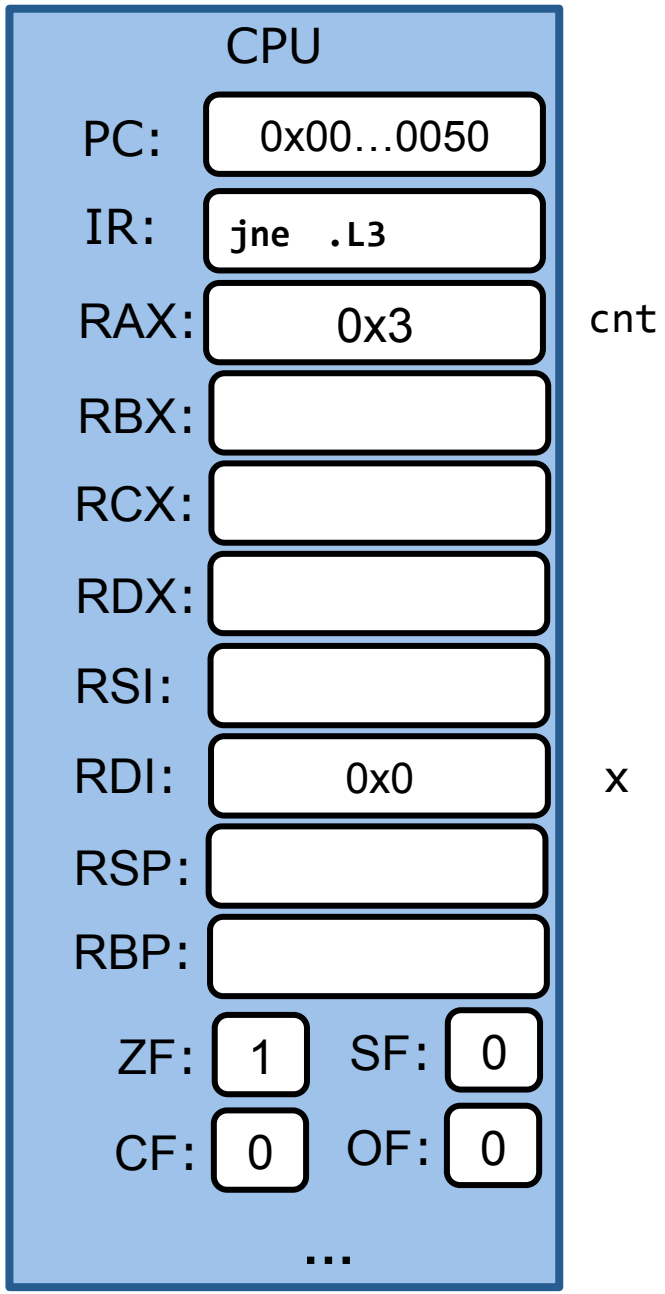



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

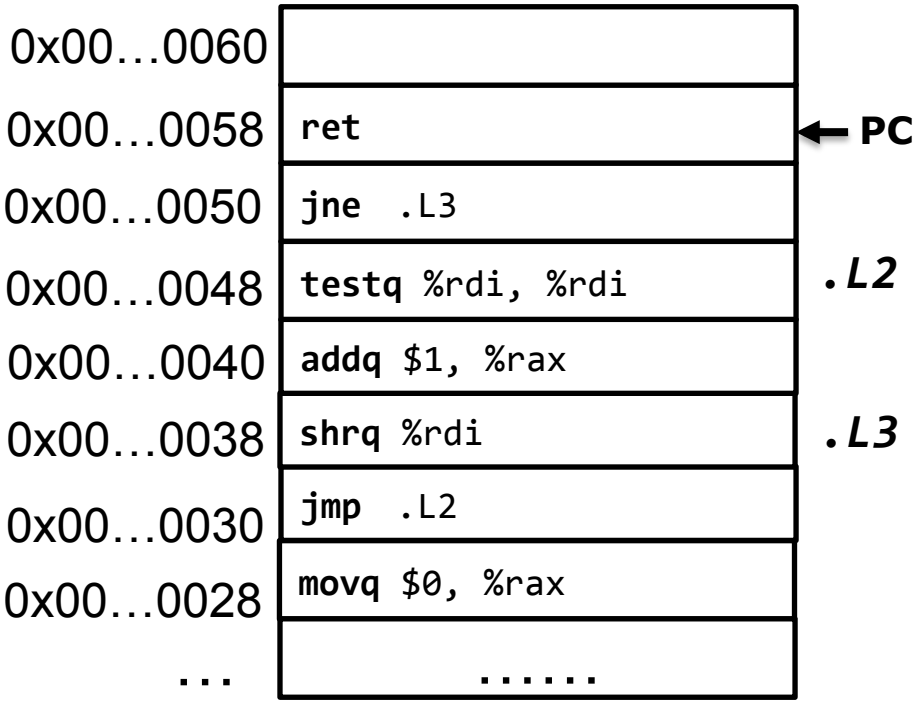


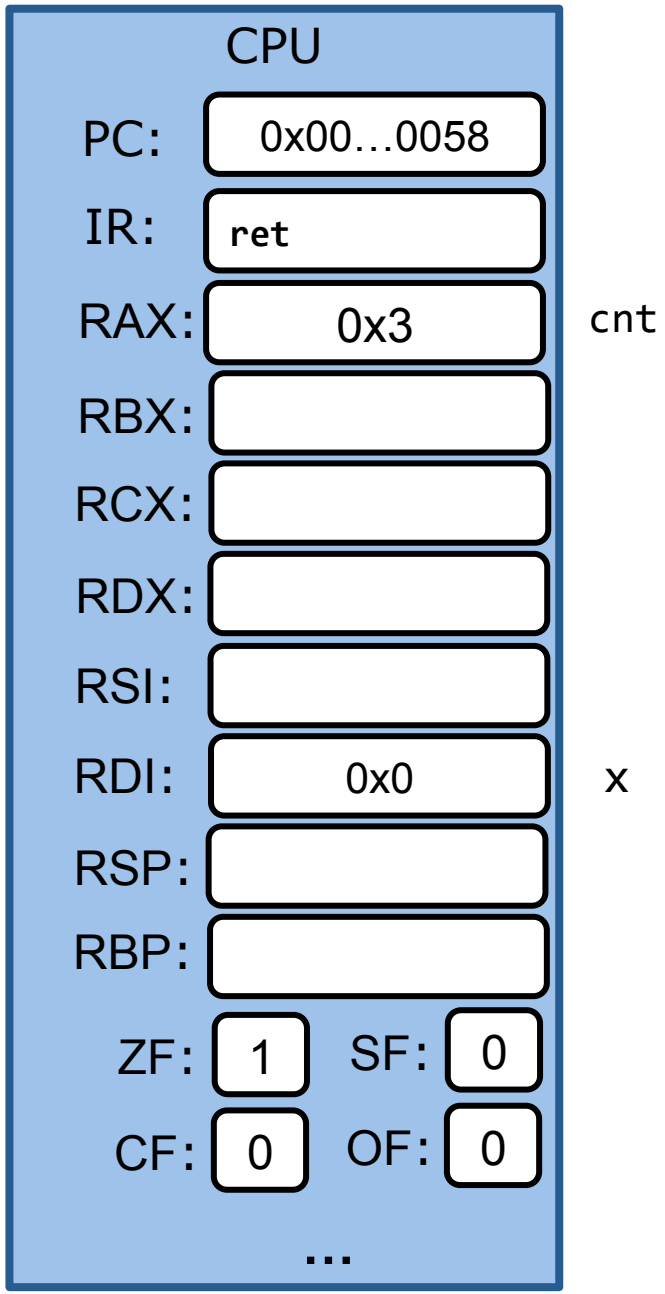



```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF






```
long count(unsigned long x)
{
  long cnt = 0;
  while (x != 0) {
        x = x >> 1;
        cnt++;
   }
   return cnt;
}
```

x: 4 $(100)_2$

jne	~ZF

"For" Loop translation

For Version

```
for (Init; Test; Update)
    Body
```

While Version

```
Init;
while (Test) {
    Body
    Update;
}
```

"Loop" Translation example

- `gcc –Og –S *.c`

```
int sum(int n)
{
  int sum = 0;
  for (int i=0; i<n; i++){
    sum += i;
  }
  return sum;
}
```

```
sum:
    movl $0, %edx
    movl $0, %eax
    jmp .L5
.L6:
    addl %edx, %eax
    addl $1, %edx
.L5:
    cmpl %edi, %edx
    jl  .L6
    ret
```

Register	Use(s)
%edi	n
%eax	sum
%edx	i

"Loop" Translation example

- `gcc –Og –S *.c`

```
int sum(int n)
{
  int sum = 0;
  for (int i=0; i<n; i++){
    sum += i;
  }
  return sum;
}
```

```
sum:
    movl $0, %edx          int i = 0
    movl $0, %eax
    jmp .L5
.L6:
    addl %edx, %eax
    addl $1, %edx
.L5:
    cmpl %edi, %edx
    jl  .L6
    ret
```

Register	Use(s)
%edi	n
%eax	sum
%edx	i

"Loop" Translation example

- `gcc –Og –S *.c`

```
int sum(int n)
{
  int sum = 0;
  for (int i=0; i<n; i++){
    sum += i;
  }
  return sum;
}
```

```
sum:
    movl $0, %edx          int i = 0;
    movl $0, %eax          int sum = 0;
    jmp .L5
.L6:
    addl %edx, %eax
    addl $1, %edx
.L5:
    cmpl %edi, %edx
    jl  .L6
    ret
```

Register	Use(s)
%edi	n
%eax	sum
%edx	i

"Loop" Translation example

- `gcc –Og –S *.c`

```
int sum(int n)
{
  int sum = 0;
  for (int i=0; i<n; i++){
    sum += i;
  }
  return sum;
}
```

```
sum:
    movl $0, %edx          int i = 0;
    movl $0, %eax          int sum = 0;
    jmp .L5                goto L5;
.L6:
    addl %edx, %eax
    addl $1, %edx
.L5:
    cmpl %edi, %edx
    jl  .L6
    ret
```

Register	Use(s)
%edi	n
%eax	sum
%edx	i

"Loop" Translation example

- `gcc –Og –S *.c`

```
int sum(int n)
{
  int sum = 0;
  for (int i=0; i<n; i++){
    sum += i;
  }
  return sum;
}
```

```
sum:
    movl $0, %edx          int i = 0;
    movl $0, %eax          int sum = 0;
    jmp .L5                goto L5;
.L6:
    addl %edx, %eax        sum = sum + i
    addl $1, %edx          i = i + 1
.L5:
    cmpl %edi, %edx
    jl  .L6
    ret
```

Register	Use(s)
%edi	n
%eax	sum
%edx	i

"Loop" Translation example

- `gcc –Og –S *.c`

```
int sum(int n)
{
  int sum = 0;
  for (int i=0; i<n; i++){
    sum += i;
  }
  return sum;
}
```

Register	Use(s)
%edi	n
%eax	sum
%edx	i

```
sum:
    movl $0, %edx       int i = 0;
    movl $0, %eax       int sum = 0;
    jmp .L5             goto L5;
.L6:
    addl %edx, %eax     sum = sum + i;
    addl $1, %edx       i = i + 1;
.L5:
    cmpl %edi, %edx     if i < n
    jl  .L6               goto L6;
    ret                 return;
```