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?

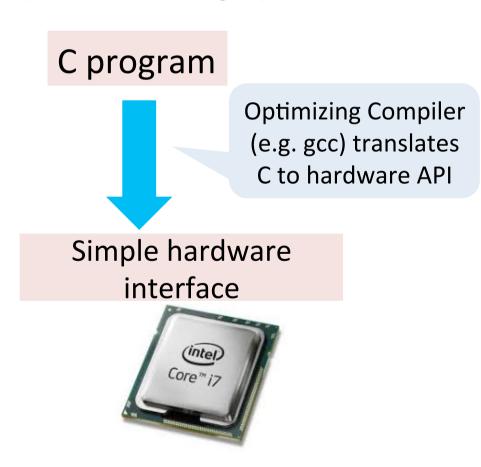
a CPU to execute C directly?



- 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

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

<pre>long x; long y;</pre>	
y = x; y = 2*y;	compile to



instruction 0x00...0058 0x00...0050 instruction instruction 0x00...0048 0x00...0040 0x00...0038 data **Y**:0x00...0030 data 0x00...0028 data X:0x00...0020 data 0x00...0018 0x00...0010 Memory

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

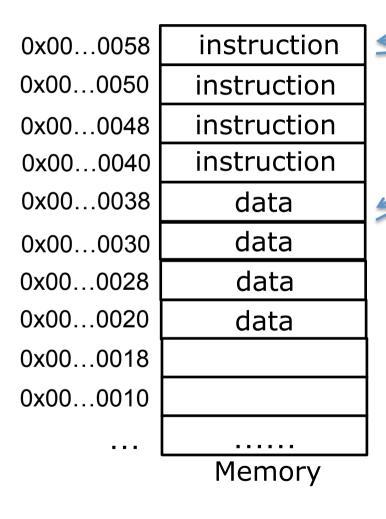
How CPU executes a program

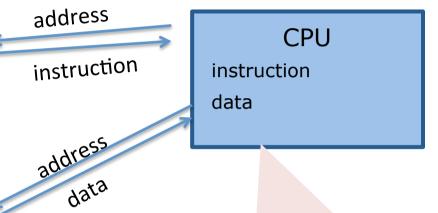
0x000058	instruction		
0x000050	instruction		
0x000048	instruction		
0x000040	instruction		
0x000038	data		
0x000030	data		
0x000028	data		
0x000020	data		
0x000018			
0x000010			
Memory			

instruction CPU instruction

64 bit machine

How CPU executes a program



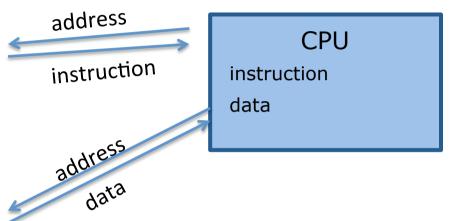


CPU can execute billions of instructions per second

CPU can do ~10 million fetches/sec from memory

How CPU executes a program

0x000058	instruction	
0x000050	instruction	
0x000048	instruction	
0x000040	instruction	
0x000038	data	
0x000030	data	
0x000028	data	
0x000020	data	
0x000018		
0x000010		
Memory		



Questions

How does CPU know which instruction to fetch?

Where does CPU keep the instruction and data?

Register – temporary storage area built into a CPU

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

- Store memory address of next instruction
- Called "RIP" in x86_64

IR: instruction register

Store the fetched instruction

General purpose registers:

Store data and address used by program

Program status and control register:

- Status of the program being executed
- Called "EFLAGS" in x86_64

Register – temporary storage area built into a CPU

PC: Program counter

- Store memory address of next instruction
- Also called "RIP" in x86_64

IR: instruction register

Store the fetched instruction

General purpose registers:

Store operands and pointers used by program

Program status and control register:

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

Visible to programmers (aka part of hardware interface)

Г		addr	CPU
0x000058	instruction		
0x000050	instruction		PC: 0x000058
0x000048	instruction		IR:
0x000040	instruction		GPRs:
0x000038	data		
0x000030	data		
0x000028	data		
0x000020	data		
0x000018			
0x000010			
	Memory	_	

addr

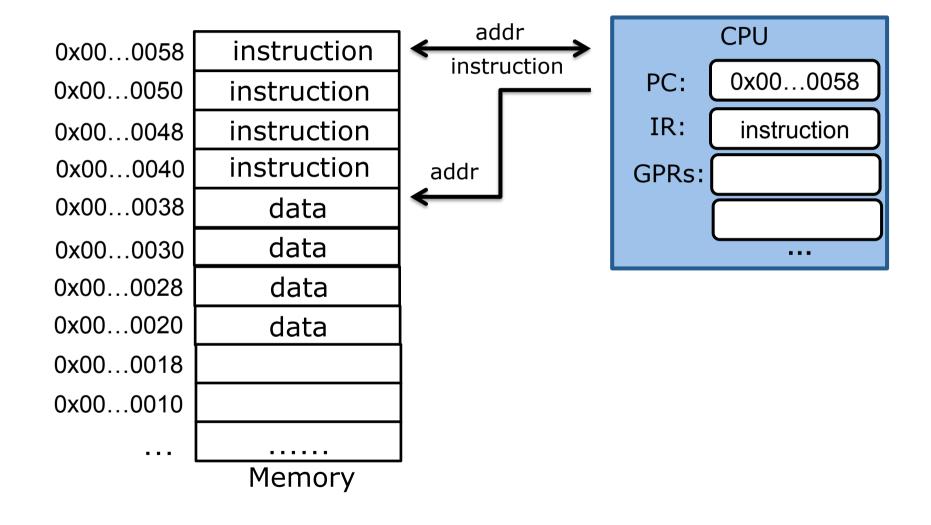
instruction

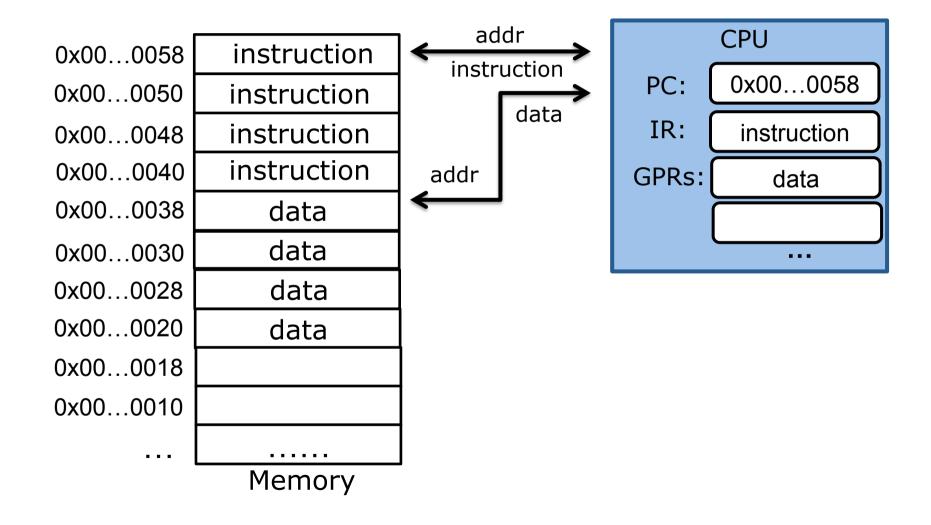
0x000058	instruction	
0x000050	instruction	
0x000048	instruction	
0x000040	instruction	
0x000038	data	
0x000030	data	
0x000028	data	
0x000020	data	
0x000018		
0x000010		
Memory		

PC: 0x00...0058

IR: instruction

GPRs:

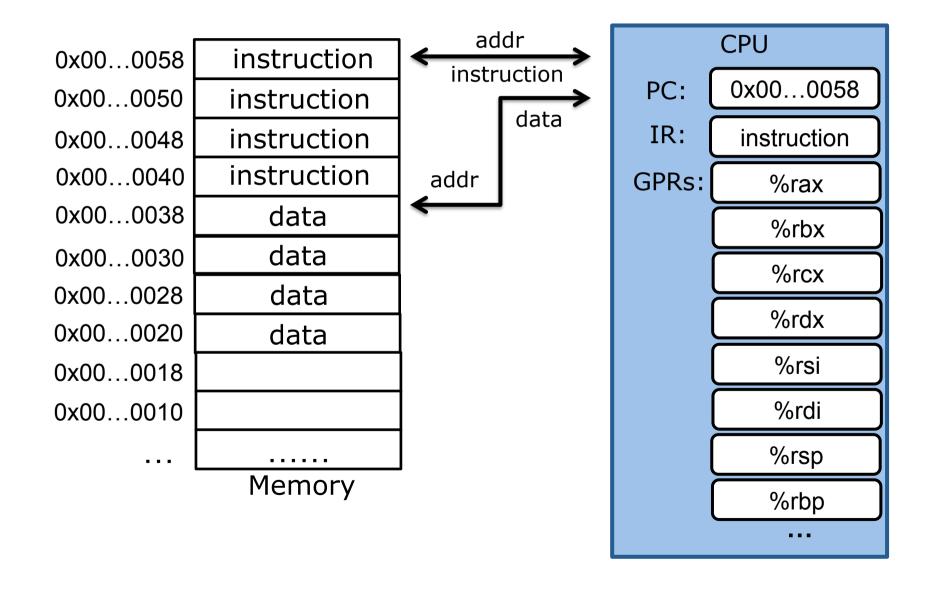




General Purpose Registers (intel x86-64)

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

8 bytes



General Purpose Registers (intel x86-64)

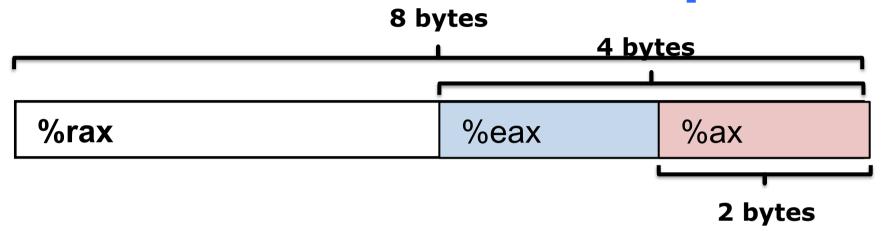
%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

8 bytes

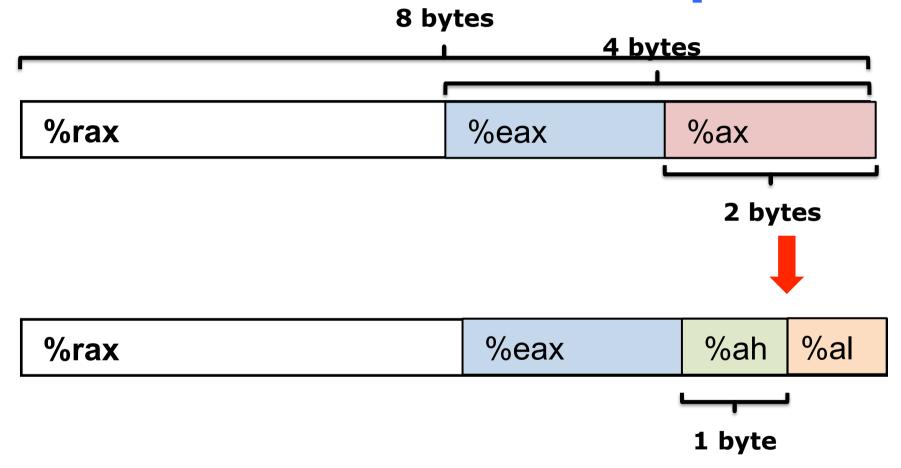
%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

4 bytes

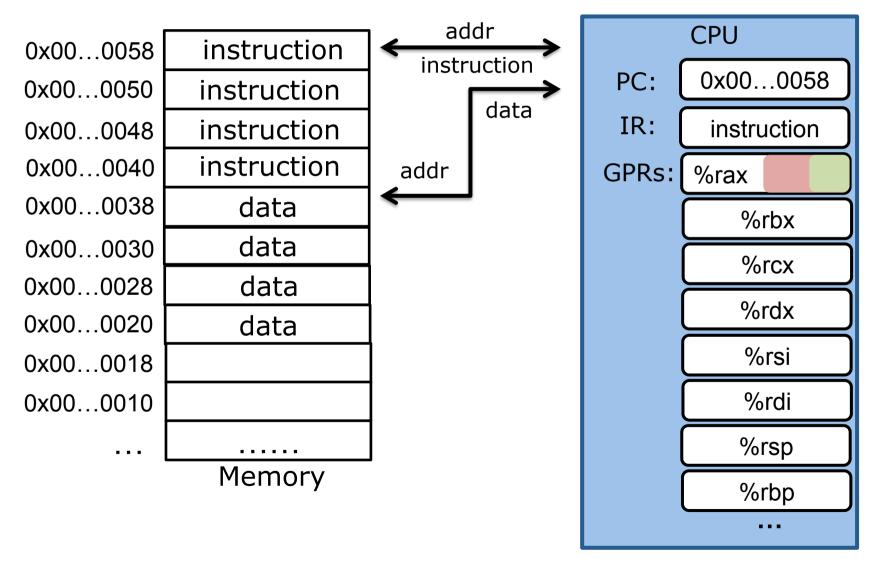
Use %rax as an example



Use %rax as an example



CPU execution (intel x86-64)



Steps of execution

- 1. PC contains the instruction's address
- 2. Fetch the instruction into IR
- 3. Execute the instruction

Instruction Set Architecture (ISA)

The interface exposed by hardware to software writers

X86_64 is the ISA implemented by Intel/AMD CPUs

- 64-bit version of x86

taught by CSO

ARM is another common ISA

Phones, tablets, Raspberry Pi

X86 ISA

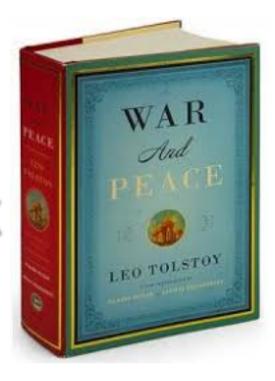


Intel® 64 and IA-32 Architectures Software Developer's Manual

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

Moving data

movq Source, Dest

 Copy a quadword (64 bits) from source operand to destination operand.

Moving data

suffix movq Source, Dest

 Copy a quadword (64 bits) from source operand to destination operand.

Suffix	Name	Size (byte)
b	Byte	1
W	Word	2
1	Long	4
q	Quadword	8

Why using a size suffix?

movq Source, Dest

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

Moving data

movq Source, Dest

Operand Types

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

movq Operand combinations

Source Dest Source, Dest $\begin{cases} Imm & Reg & movq $0x4,\%rax \\ Mem & movq $0x4,(\%rax) \end{cases}$ Reg & Reg & movq \$movq \$mo

- 1. Immediate can only be Source
- 2. No memory-memory mov

How CPU executes an instruction

- 1. PC contains the instruction's address
- 2. Load the instruction into IR

- 3. Execute the instruction
- 4. CPU automatically increments PC by the size of the executed (retired) instruction.

movq Imm, Reg

				_
0x000058	movq	%rax,%r	·bх	
0x000050	movq	\$0x4,%r	ax	← PC
0x000048				
0x000040				
0x000038				
0x000030				
0x000028				
0x000020				
0x000018				
0x000010				
		1emory		-

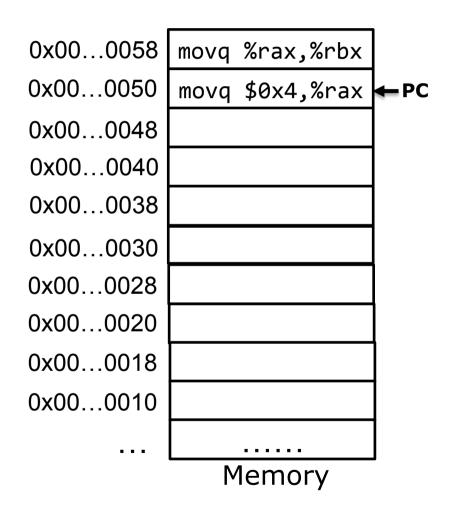
	CPU
PC:	0x000050
IR:	
RAX:	
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	

movq Imm, Reg

				_
0x000058	movq	%rax,%r	·bх	
0x000050	movq	\$0x4,%r	ax	← PC
0x000048				
0x000040				
0x000038				
0x000030				
0x000028				
0x000020				
0x000018				
0x000010				
		1emory		-

CPU	
PC:	0x000050
IR:	movq \$0x4, %rax
RAX:	
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	

movq Imm, Reg



CPU	
PC:	0x000050
IR:	movq \$0x4, %rax
RAX:	0x000004
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	

movq Reg, Reg

		_
0x000058	movq %rax,%rbx	← PC
0x000050	movq \$0x4,%rax	
0x000048		
0x000040		
0x000038		
0x000030		
0x000028		
0x000020		
0x000018		
0x000010		
	Memory	•

CPU	
PC:	0x000058
IR:	movq \$0x4, %rax
RAX:	0x000004
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	
	•••

movq Reg, Reg

0x000058	movq %rax,%rbx	← PC
0x000050	movq \$0x4,%rax	
0x000048		
0x000040		
0x000038		
0x000030		
0x000028		
0x000020		
0x000018		
0x000010		
	Memory	•

CPU	
PC:	0x000058
IR:	movq %rax, %rbx
RAX:	0x000004
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	
	•••

movq Reg, Reg

	<u></u>
0x000058	movq %rax,%rbx ←PC
0x000050	movq \$0x4,%rax
0x000048	
0x000040	
0x000038	
0x000030	
0x000028	
0x000020	
0x000018	
0x000010	
	Memory

CPU	
PC:	0x000058
IR:	movq %rax, %rbx
RAX:	0x000004
RBX:	0x000004
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	
	•••

movq Mem, Reg

How to represent a "memory" operand?

Direct addressing: use registers to index the memory

(Register)

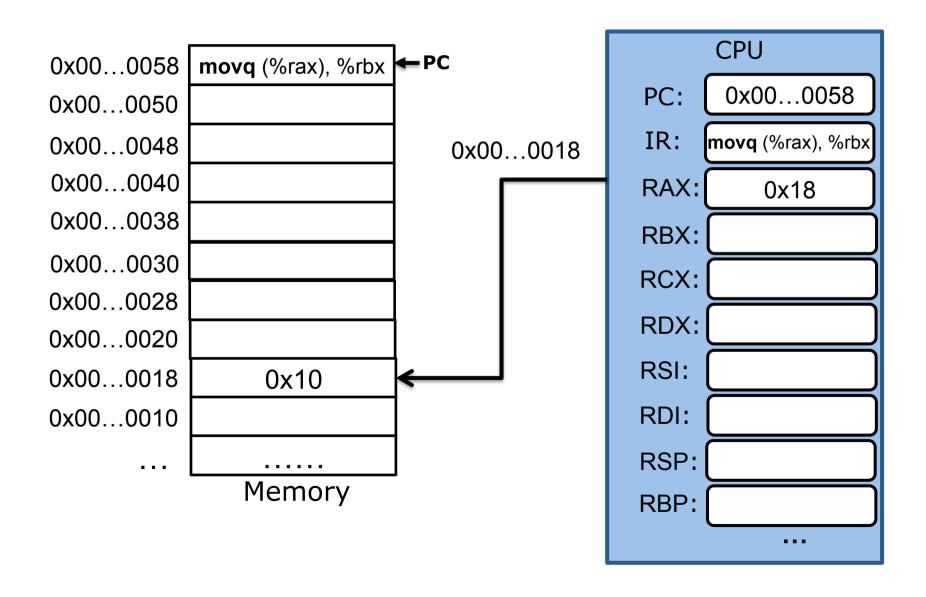
- Register contains a memory address
- movq (%rax), %rbx

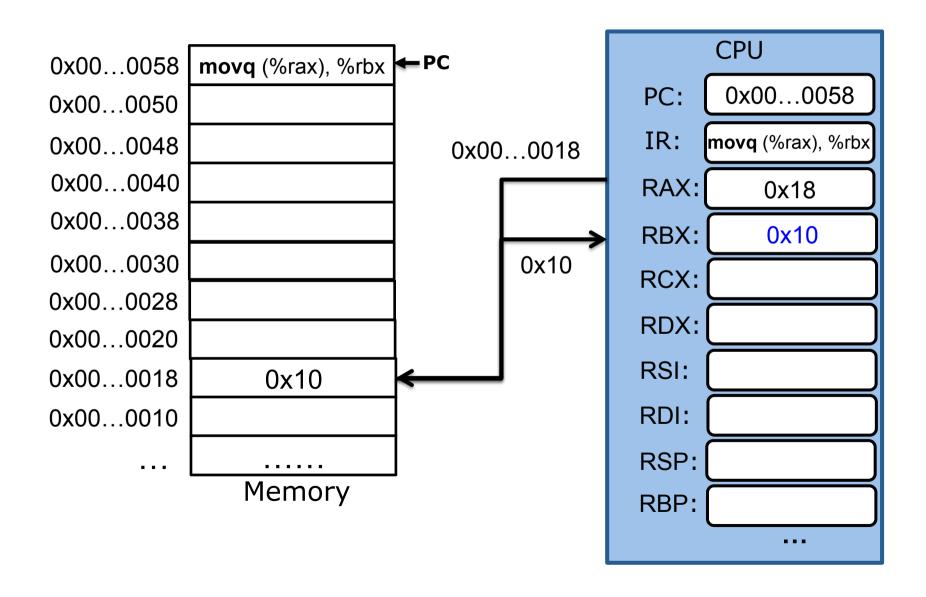
0x000058	movq (%rax), %rbx	← PC
0x000050		
0x000048		
0x000040		
0x000038		
0x000030		
0x000028		
0x000020		
0x000018	0x10	
0x000010		
	Memory	-

	CPU
PC:	0x000058
IR:	
RAX:	0x18
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	
	•••

		_
0x000058	movq (%rax), %rbx	← PC
0x000050		
0x000048		
0x000040		
0x000038		
0x000030		
0x000028		
0x000020		
0x000018	0x10	
0x000010		
	Memory	-

	CPU
PC:	0x000058
IR:	movq (%rax), %rbx
RAX:	0x18
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	
	•••





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

    long tmp = *a;
    *a = *b;
    *b = tmp;
    gcc -S -O3 swap.c
}
```

```
%rdi stores a
                                                              %rsi stores b
void
                                            swap:
swap(long *a, long* b) {
                                                          (%rdi),
                                                                   %rax
                                                movq
                                                          (%rsi), /%rdx
                                                movq
   long tmp = *a;
                                                           %rdx,/(%rdi)
                                                movq
   *a = *b;
                                                           %rax,
                                                movq
                         gcc -S -O3 swap.c
   *b = tmp;
                                                %rax is local variable tmp
```

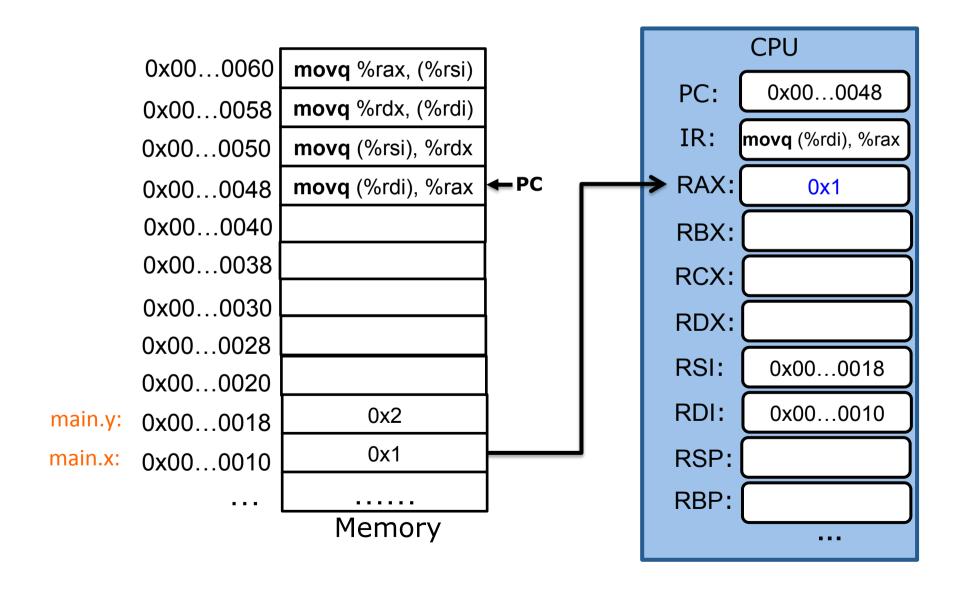
Use two instructions and %rdx to perform memory to memory move

			_
	0x000060	movq %rax, (%rsi)	
	0x000058	movq %rdx, (%rdi)	
	0x000050	movq (%rsi), %rdx	
	0x000048	movq (%rdi), %rax	← PC
	0x000040		
	0x000038		
	0x000030		
	0x000028		
	0x000020		
main.y:	0x000018	0x2	
main.x:	0x000010	0x1	
		Memory	

	CPU
PC:	0x000048
IR:	
RAX:	
RBX:	
RCX:	
RDX:	
RSI:	0x000018
RDI:	0x000010
RSP:	
RBP:	
	•••

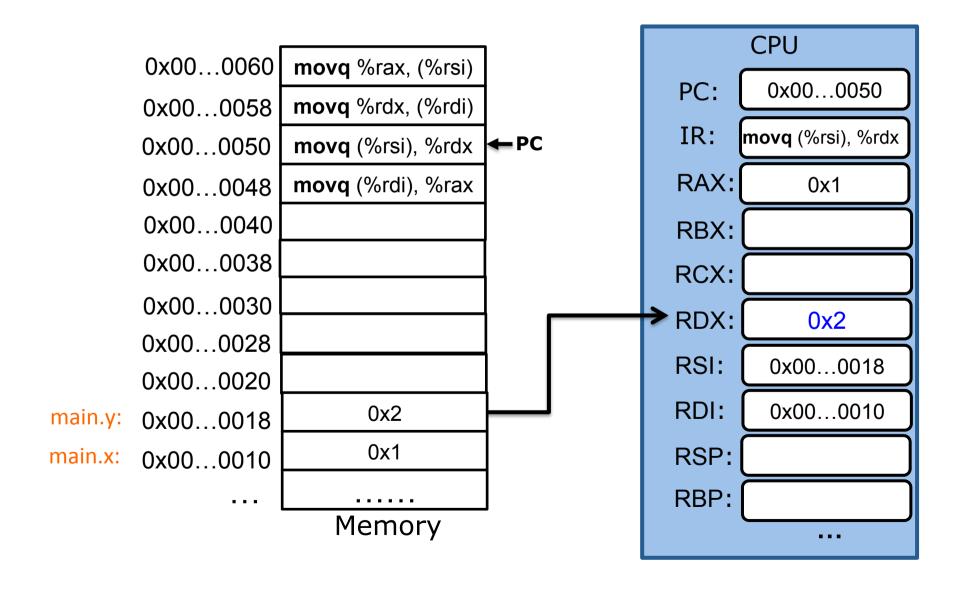
	_		_
	0x000060	movq %rax, (%rsi)	
	0x000058	movq %rdx, (%rdi)	
	0x000050	movq (%rsi), %rdx	
	0x000048	movq (%rdi), %rax	← PC
	0x000040		
	0x000038		
	0x000030		
	0x000028		
	0x000020		
main.y:	0x000018	0x2	
main.x:	0x000010	0x1	
		Memory	_

	CPU
PC:	0x000048
IR:	movq (%rdi), %rax
RAX:	
RBX:	
RCX:	
RDX:	
RSI:	0x000018
RDI:	0x000010
RSP:	
RBP:	
	•••



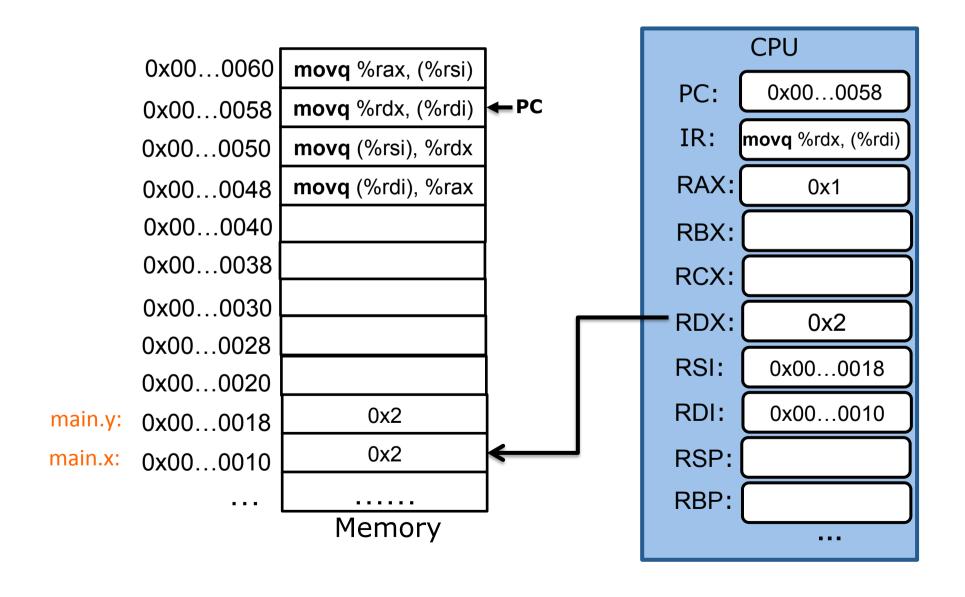
	_		_
	0x000060	movq %rax, (%rsi)	
	0x000058	movq %rdx, (%rdi)	
	0x000050	movq (%rsi), %rdx	← PC
	0x000048	movq (%rdi), %rax	
	0x000040]
	0x000038		
	0x000030		ļ
	0x000028		
	0x000020		
main.y:	0x000018	0x2	
main.x:	0x000010	0x1	
		Memory	

	CPU
PC:	0x000050
IR:	movq (%rsi), %rdx
RAX:	0x1
RBX:	
RCX:	
RDX:	
RSI:	0x000018
RDI:	0x000010
RSP:	
RBP:	
	•••



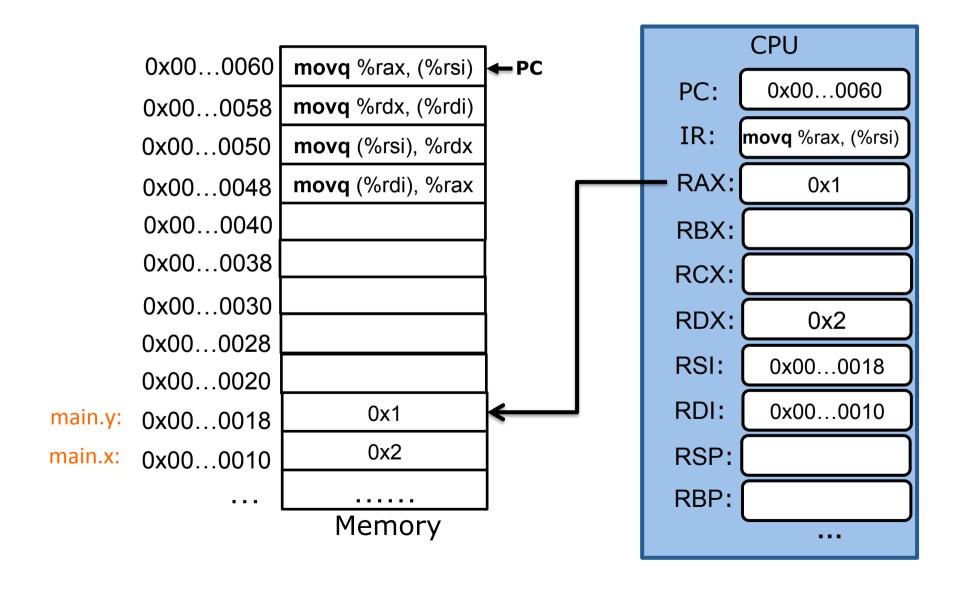
			_
	0x000060	movq %rax, (%rsi)	
	0x000058	movq %rdx, (%rdi)	← PC
	0x000050	movq (%rsi), %rdx	
	0x000048	movq (%rdi), %rax	
	0x000040		
	0x000038		
	0x000030		
	0x000028		
	0x000020		
main.y:	0x000018	0x2	
main.x:	0x000010	0x1	
		Memory	

	CPU
PC:	0x000058
IR:	movq %rdx, (%rdi)
RAX:	0x1
RBX:	
RCX:	
RDX:	0x2
RDX: RSI:	0x2 0x000018
RSI:	0x000018
RSI: RDI:	0x000018



	0x000060	movq %rax, (%rsi)	← PC
		. ,	
	0x000058	movq %rdx, (%rdi)	
	0x000050	movq (%rsi), %rdx	
	0x000048	movq (%rdi), %rax	
	0x000040		
	0x000038		
	0x000030		
	0x000028		
	0x000020		
main.y:	0x000018	0x2	
main.x:	0x000010	0x2	
		Memory	

CPU	
PC:	0x000060
IR:	movq %rax, (%rsi)
RAX:	0x1
RBX:	
RCX:	
RDX:	0x2
RDX: RSI:	0x2 0x000018
RSI:	0x000018
RSI: RDI:	0x000018



Limitation of direct addressing

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

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

Must use 2 instructions:

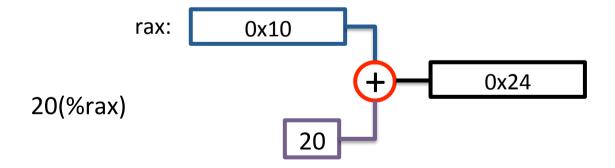
- 1. calculate &a[i] and store in %rax
- 2. mov \$0, (%rax)

Solution: Addressing mode with displacement

D(Register): val(Register) + D

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





0x000060		
0x000058		
0x000050		
0x000048	movq \$0x2, 16(%rax)	
0x000040	movq \$0x1, 8(%rax)	
0x000038	movq \$0x0, (%rax)	
0x000030	movq \$0x10, %rax	← PC
0x000028		
0x000020	0x1	a[2]
0x000018	0x2	a[1]
0x000010	0x3	a[0]
	Memory	-

CPU		
PC:	0x000030	
IR:	movq \$0x10, %rax	
RAX:		
RBX:		
RCX:		
RDX:		
RSI:		
RDI:		
RSP:		
RBP:		
	•••	

0x000060		
0x000058		
0x000050		
0x000048	movq \$0x2, 16(%rax)	
0x000040	movq \$0x1, 8(%rax)	
0x000038	movq \$0x0, (%rax)	
0x000030	movq \$0x10, %rax	← PC
0x000028		
0x000020	0x1	a[2]
0x000018	0x2	a[1]
0x000010	0x3	a[0]
	Memory	

CPU	
PC:	0x000030
IR:	movq \$0x10, %rax
RAX:	0x000010
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	
	•••

0x000060		
0x000058		
0x000050		
0x000048	movq \$0x2, 16(%rax)	
0x000040	movq \$0x1, 8(%rax)	
0x000038	movq \$0x0, (%rax)	← PC
0x000030	movq \$0x10, %rax	
0x000028		
0x000020	0x1	a[2]
0x000018	0x2	a[1]
0x000010	0x3	a[0]
	Memory	

CPU		
PC:	0x000038	
IR:	movq \$0x0, (%rax)	
RAX:	0x000010	
RBX:		
RCX:		
RDX:		
RSI:		
RDI:		
RSP:		
RBP:		
	•••	

0x000060		
0x000058		
0x000050		
0x000048	movq \$0x2, 16(%rax)	
0x000040	movq \$0x1, 8(%rax)	
0x000038	movq \$0x0, (%rax)	← PC
0x000030	movq \$0x10, %rax	
0x000028		
0x000020	0x1	a[2]
0x000018	0x2	a[1]
0x000010	0x0	a[0]
	Memory	

CPU	
PC:	0x000038
IR:	movq \$0x0, (%rax)
RAX:	0x000010
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	
	•••

		•
0x000060		
0x000058		
0x000050		
0x000048	movq \$0x2, 16(%rax)	
0x000040	movq \$0x1, 8(%rax)	← PC
0x000038	movq \$0x0, (%rax)	r:
0x000030	movq \$0x10, %rax	
0x000028		ı
0x000020	0x1	a[2]
0x000018	0x2	a[1]
0x000010	0x0	a[0]
	Memory	

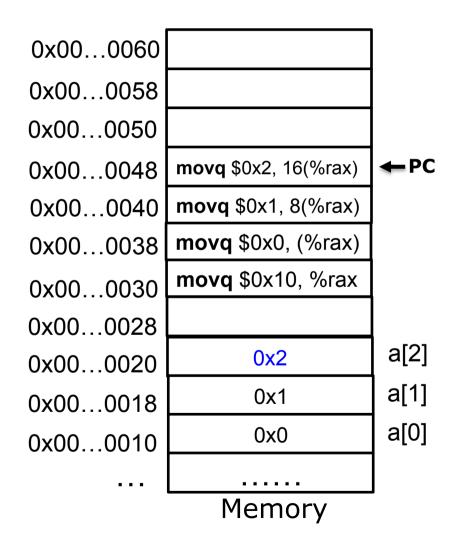
CPU	
PC:	0x000040
IR:	movq \$0x1, 8(%rax)
RAX:	0x000010
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	
	•••

		•
0x000060		
0x000058		
0x000050		
0x000048	movq \$0x2, 16(%rax)	
0x000040	movq \$0x1, 8(%rax)	← PC
0x000038	movq \$0x0, (%rax)	r:
0x000030	movq \$0x10, %rax	
0x000028		ı
0x000020	0x1	a[2]
0x000018	0x1	a[1]
0x000010	0x0	a[0]
	Memory	

CPU		
PC:	0x000040	
IR:	movq \$0x1, 8(%rax)	
RAX:	0x000010	
RBX:		
RCX:		
RDX:		
RSI:		
RDI:		
RSP:		
RBP:		

0x000060		
0x000058		
0x000050		
0x000048	movq \$0x2, 16(%rax)	← PC
0x000040	movq \$0x1, 8(%rax)	
0x000038	movq \$0x0, (%rax)	1
0x000030	movq \$0x10, %rax	
0x000028		
0x000020	0x1	a[2]
0x000018	0x1	a[1]
0x000010	0x0	a[0]
	Memory	_

CPU	
PC:	0x000048
IR:	movq \$0x2, 16(%rax)
RAX:	0x000010
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	
	•••



CPU	
PC:	0x000048
IR:	movq \$0x2, 16(%rax)
RAX:	0x000010
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	
RSP:	
RBP:	
	•••

Complete Memory Addressing Mode

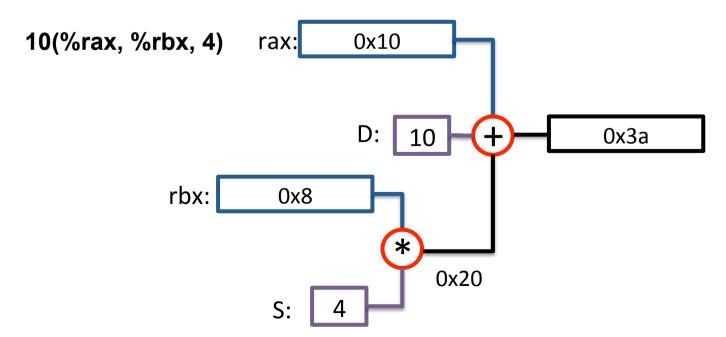
D(Rb, Ri, S): val(Rb) + S * val(Ri) + D

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)

%rdx	0xf000
%rcx	0x100

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

%rdx	0xf000
%rcx	0x100

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

%rdx	0xf000
%rcx	0x100

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

%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)		

%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

•		•
0x000060		
0x000058	movq \$2, (%rdi, %rbx, 8)	
0x000050	movq \$1, 8(%rdi, %rax, 8)	←PC
0x000048		
0x000040	a[6]	
0x000038	a[5]	
0x000030	a[4]	t.
0x000028	a[3]	
0x000020	a[2]	
0x000018	a[1]	
0x000010	a[0]	
	Memory	

CPU		
PC:	0x000050	
IR:	movq \$1, 8(%rdi, %rax, 8)	
RAX:	0x000001	
RBX:	0x000002	
RCX:		
RDX:		
RSI:		
RDI:	0x000010	
RSP:		
RBP:		
	•••	

0x000060		
0x000058	movq \$2, (%rdi, %rbx, 8)	
0x000050	movq \$1, 8(%rdi, %rax, 8)	←PC
0x000048		
0x000040	a[6]	
0x000038	a[5]	
0x000030	a[4]	
0x000028	a[3]	
0x000020	a[2]: 1	
0x000018	a[1]	
0x000010	a[0]	
	Memory	-

CPU		
PC:	0x000050	
IR:	movq \$1, 8(%rdi, %rax, 8)	
RAX:	0x000001	
RBX:	0x000002	
RCX:		
RDX:		
RSI:		
RDI:	0x000010	
RSP:		
RBP:		
•••		

0x000060		
00000000		
0x000058	movq \$2, (%rdi, %rbx, 8)	← PC
0x000050	movq \$1, 8(%rdi, %rax, 8)	
0x000048		
0x000040	a[6]	
0x000038	a[5]	
0x000030	a[4]	t.
0x000028	a[3]	
0x000020	a[2]: 2	
0x000018	a[1]	
0x000010	a[0]	
	Memory	-

CPU		
PC:	0x000058	
IR:	movq \$1, (%rdi, %rbx, 8)	
RAX:	0x000001	
RBX:	0x000002	
RCX:		
RDX:		
RSI:		
RDI:	0x000010	
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
movI src, dest	Copy a long (32 bits) from the source operand to the destination operand. e.g., movl %eax, %ebx
movq src, dest	Copy a quadword from the source operand to the destination operand. e.g., movq %rax, %rbx

The lea instruction

leaq Source, Dest

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

000 0000		
0x000060		
0x000058	movq (%rbx), %rcx	
0x000050	movq 8(%rdi), %rax	
0x000048	leaq 8(%rdi), %rbx	← PC
0x000040		
0x000038		
0x000030		
0x000028		
0x000020	300	
0x000018	200	
0x000010	100	
,	Memory	•

CPU		
PC:	0x000048	
IR:	leaq 8(%rdi),%rbx	
RAX:		
RBX:		
RCX:		
RDX:		
RSI:		
RDI:	0x000010	
RSP:		
RBP:		
	•••	

000 0000		
0x000060		
0x000058	movq (%rbx), %rcx	
0x000050	movq 8(%rdi), %rax	
0x000048	leaq 8(%rdi), %rbx	← PC
0x000040		
0x000038		
0x000030		
0x000028		
0x000020	300	
0x000018	200	
0x000010	100	
,	Memory	•

CPU		
PC:	0x000048	
IR:	leaq 8(%rdi),%rbx	
RAX:		
RBX:	0x000018	
RCX:		
RDX:		
RSI:		
RDI:	0x000010	
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

Arithmetic Expression Puzzle

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

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

Src,Dest Dest = Dest << Src salq Src,Dest Dest = Dest >> Src sarq shlq Src,Dest Dest = Dest << Src shrq Src,Dest Dest = Dest >> Src Src,Dest Dest = Dest ^ Src xorq Src,Dest andq Dest = Dest & Src Src,Dest Dest = Dest | Src orq Dest Dest = ~Dest notq

Arithmetic left shift
Arithmetic right shift
Logical left shift
Logical right shift

0x000060		
0x000058	addq %rax, 8(%rdi)	← PC
0x000050		
0x000048		
0x000040		
0x000038		l
0x000030		
0x000028		
0x000020	300	
0x000018	200	
0x000010	100	
	Memory	

CPU ————————————————————————————————————		
PC:	0x000058	
IR:	addq %rax, 8(%rdi)	
RAX:	0x000001	
RBX:		
RCX:		
RDX:		
RSI:		
RDI:	0x000010	
RSP:		
RBP:		
•••		

		_
0x000060		
0x000058	addq %rax , 8(%rdi)	← PC
0x000050		
0x000048		
0x000040		
0x000038		
0x000030		ļ
0x000028		_
0x000020	300	
0x000018	201	
0x000010	100	
	Memory	-

CPU	
PC:	0x000058
IR:	addq %rax, 8(%rdi)
RAX:	0x000001
RBX:	
RCX:	
RDX:	
RSI:	
RDI:	0x000010
RSP:	
RBP:	
•••	