Lecture 5: Introduction to Assembly

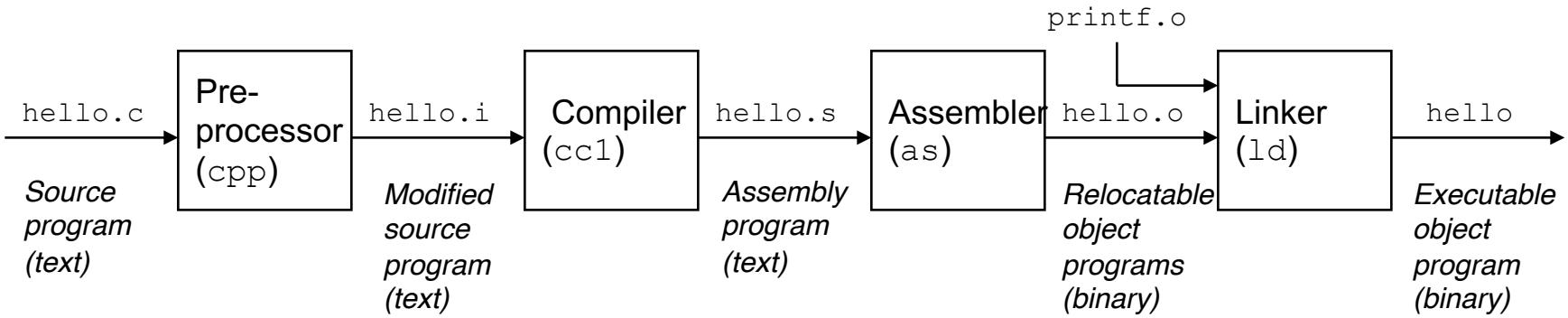
CS 105

Programs

```
#include<stdio.h>

int main(int argc,
         char** argv){

  printf("Hello
          world!\n");

  return 0;
}
```

```
55
48 89 e5
48 83 ec 20
48 8d 05 25 00 00 00
c7 45 fc 00 00 00 00
89 7d f8
48 89 75 f0
48 89 c7
b0 00
e8 00 00 00 00
31 c9
89 45 ec
89 c8
48 83 c4 20
5d
c3
```

Compilation




```
#include<stdio.h>

int main(int argc,
         char ** argv){

  printf("Hello
          world!\n");
  return 0;
}
```

```
…
int printf(const char *
           restrict,
           ...)
__attribute__((__format_
_ (__printf__, 1, 2)));
…
int main(int argc,
         char ** argv){

  printf("Hello
          world!\n");
  return 0;
}
```

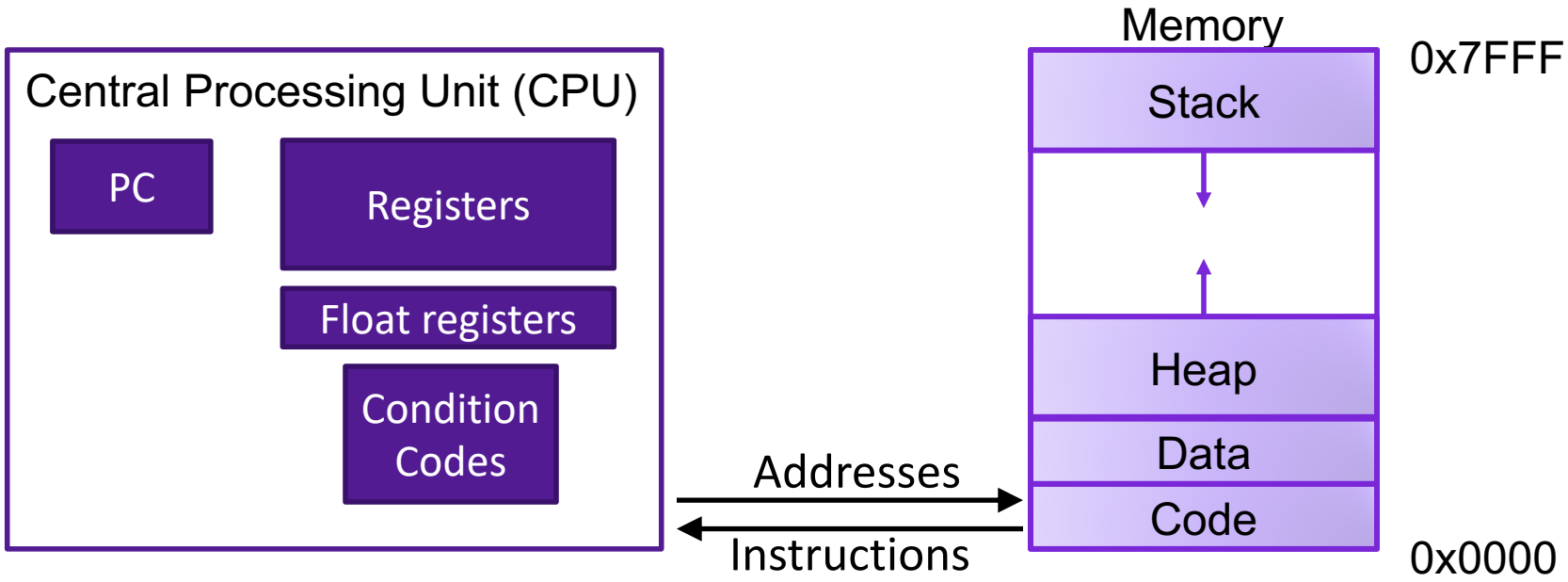
```
pushq   %rbp
movq    %rsp, %rbp
subq  $32, %rsp
leaq  L_.str(%rip), %rax
movl  $0, -4(%rbp)
movl  %edi, -8(%rbp)
movq  %rsi, -16(%rbp)
movq  %rax, %rdi
movb  $0, %al
callq _printf
xorl  %ecx, %ecx
movl  %eax, -20(%rbp)
movl  %ecx, %eax
addq  $32, %rsp
popq  %rbp
retq
```

```
55
48 89 e5
48 83 ec 20
48 8d 05 25 00 00 00
c7 45 fc 00 00 00 00
89 7d f8
48 89 75 f0
48 89 c7
b0 00
e8 00 00 00 00
31 c9
89 45 ec
89 c8
48 83 c4 20
5d
c3
```

x86-64 Assembly Language

- Evolutionary design, going back to 8086 in 1978
  - Basis for original IBM Personal Computer, 16-bits
- Intel Pentium 4E (2004): 64 bit instruction set
- High-level languages are translated into x86 instructions and then executed on the CPU
  - Actual instructions are sequences of bytes
  - We give them mnemonic names

Assembly/Machine Code View



Programmer-Visible State

- PC: Program counter (%rip)
- Register file: 16 Registers
- Float registers
- Condition codes

Memory

- Byte addressable array
- Code and user data
- Stack to support procedures

Assembly Characteristics: Instructions

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Perform arithmetic operations on register or memory data
- Transfer control
  - Conditional branches
  - Unconditional jumps to/from procedures

DATA TRANSFER IN ASSEMBLY

Data Movement Instructions

- MOV source, dest

Moves data source->dest

```
dest = source
```

Operand Forms

- Immediate:
  - Syntax: $Imm Value: Imm Example: $47
- Register:
  - Syntax: r Value: R[r] Example: %rbp
- Memory (Absolute):
  - Syntax: Imm Value: M[Imm] Example: 0x4050
- Memory (Indirect):
  - Syntax: (r) Value: M[R[r]] Example: (%rbp)

Exercise: Operands

Register	Value
%rax	0x100
%rcx	0x01
%rdx	0x03

Memory Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13

- What are the values of the following operands (assuming register and memory state shown above)?
  1. `%rax`
  2. `0x104`
  3. `$0x108`
  4. `(%rax)`

Operand Forms

- Immediate:
  - Syntax: $Imm Value: Imm Example: $47
- Register:
  - Syntax: r Value: R[r] Example: %rbp
- Memory (Absolute):
  - Syntax: Imm Value: M[Imm] Example: 0x4050
- Memory (Indirect):
  - Syntax: (r) Value: M[R[r]] Example: (%rbp)
- Memory (Base+displacement):
  - Syntax: Imm(r) Value: M[Imm+R[r]] Example: -12(%rbp)
- Memory (Scaled indexed):
  - Syntax: Imm(r1, r2, s) Value: M[Imm+R[r1]+R[r2]*s] Example: 7(%rdx, %rdx, 4)

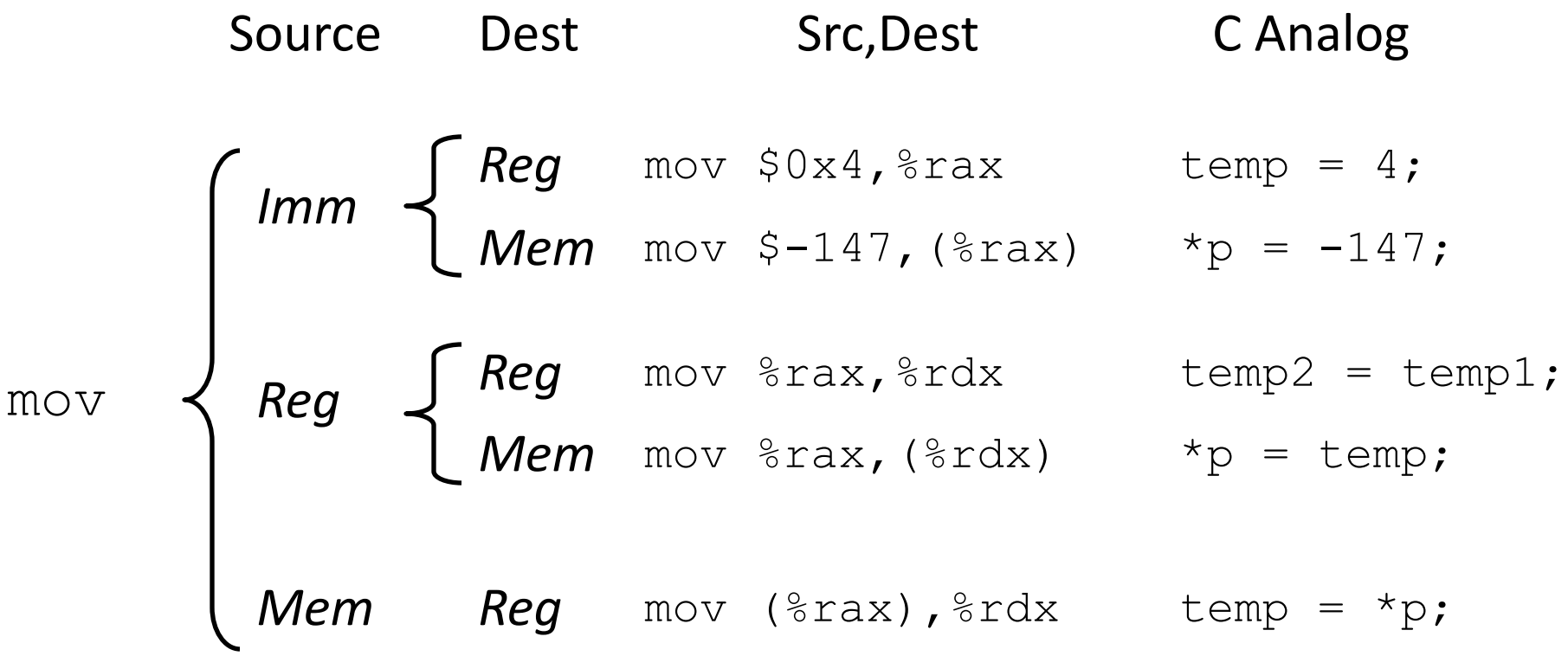
Exercise: Operands

Register	Value
%rax	0x100
%rcx	0x01
%rdx	0x03

Memory Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x47

- What are the values of the following operands (assuming register and memory state shown above)?
  1. `4(%rax)`
  2. `0(%rax,%rcx,4)`
  3. `0(%rax,%rdx,4)`
  4. `4(%rax,%rcx,4)`

`mov` Operand Combinations



Cannot do memory-memory transfer with a single instruction

Exercise: Moving Data

- For each of the following move instructions, write an equivalent C assignment

1. `mov $0x40604a, %rbx`
2. `mov %rbx, %rax`
3. `mov $47, (%rax)`

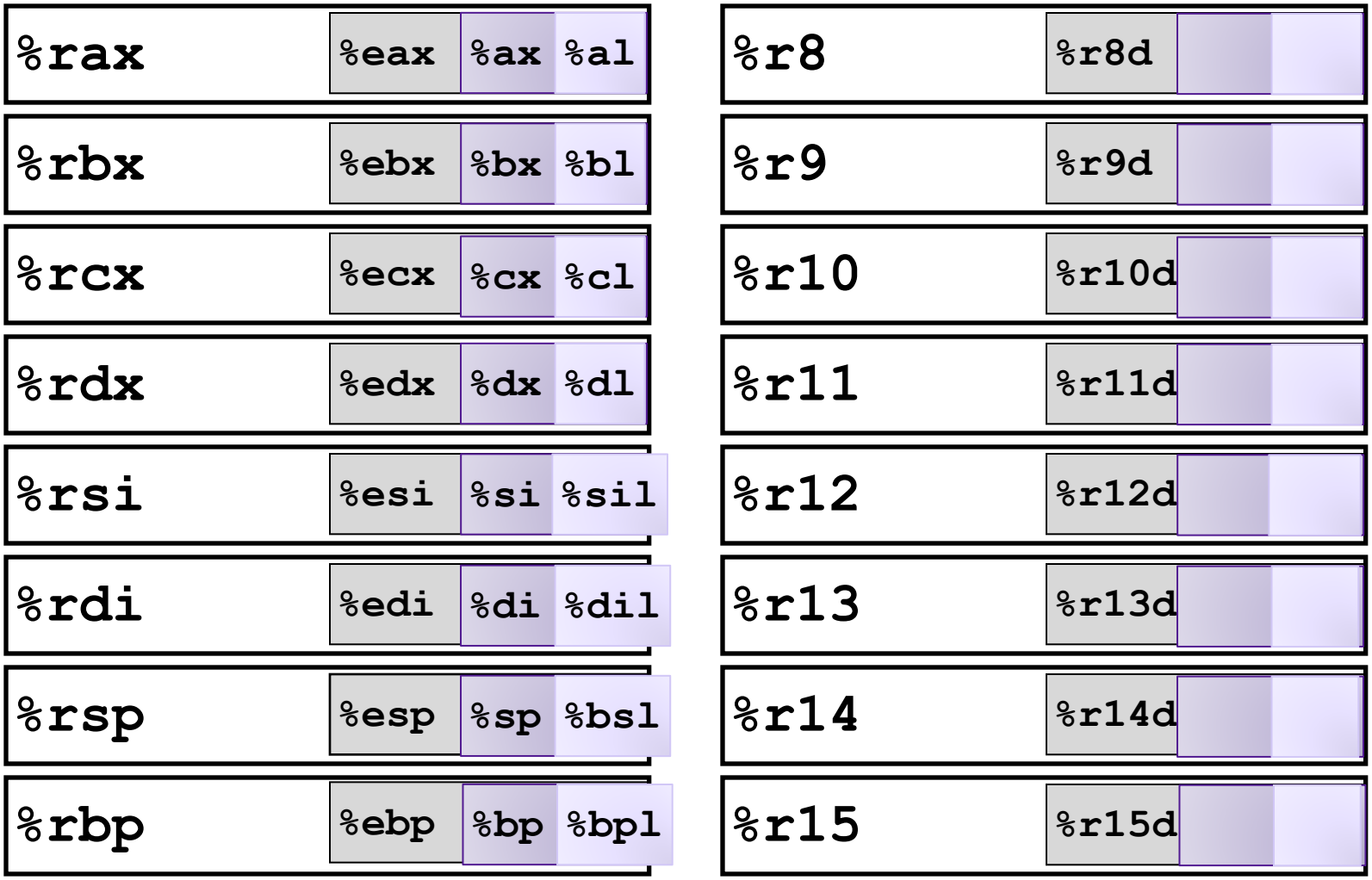
Sizes of C Data Types in x86-64

C declaration	Size (bytes)	Intel data type	Assembly suffix
char	1	Byte	b
short	2	Word	w
int	4	Double word	l
long	8	Quad word	q
char *	8	Quad word	q
float	4	Single precision	s
double	8	Double precision	l

Data Movement Instructions

- MOV source, dest — Move data source->dest
  - `movb` — Move 1 byte
  - `movw` — Move 2 bytes
  - `movl` — Move 4 bytes
  - `movq` — Move 8 bytes

X86-64 Integer Registers



X86-64 Integer Registers

`%rax` (function result)	`%r8` (fifth argument)
`%rbx`	`%r9` (sixth argument)
`%rcx` (fourth argument)	`%r10`
`%rdx` (third argument)	`%r11`
`%rsi` (second argument)	`%r12`
`%rdi` (first argument)	`%r13`
`%rsp` (stack pointer)	`%r14`
`%rbp`	`%r15`

Exercise: Translating Assembly

- Write a C function `void decode1(long *xp, long *yp)` that will do the same thing as the following assembly code:

```
decode:

        movq (%rdi), %rax
        movq (%rsi), %rcx
        movq %rax, (%rsi)
        movq %rcx, (%rdi)
        ret
```

```
void decode(long *xp, long *yp){




}
```

Register	Use(s)
`%rdi`	Argument **`xp`**
`%rsi`	Argument **`yp`**

C is close to Machine Language

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e:  48 89 03
```

- C Code
  - Store value **t** where designated by **dest**
- Assembly
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - **t:** Register **%rax**
    - **dest:** Register **%rbx**
    - ***dest:** Memory **M[%rbx]**
- Object Code
  - 3-byte instruction
  - at address **0x40059e**

ARITHMETIC IN ASSEMBLY

Some Arithmetic Operations

- Two Operand Instructions:

Format		Computation	
`andq`	Src,Dest	Dest = Dest & Src	
`orq`	Src,Dest	Dest = Dest | Src	
`xorq`	Src,Dest	Dest = Dest ^ Src	
`shlq`	Src,Dest	Dest = Dest << Src	Also called salq
`shrq`	Src,Dest	Dest = Dest >> Src	Logical
`sarq`	Src,Dest	Dest = Dest >> Src	Arithmetic
`addq`	Src,Dest	Dest = Dest + Src	
`subq`	Src,Dest	Dest = Dest – Src	
`imulq`	Src,Dest	Dest = Dest * Src	

Suffixes

`char`	`b`	`1`
`short`	`w`	`2`
`int`	`l`	`4`
`long`	`q`	`8`
`pointer`	`q`	`8`

Some Arithmetic Operations

- One Operand Instructions

notq	Dest	Dest = ~Dest
incq	Dest	Dest = Dest + 1
decq	Dest	Dest = Dest – 1
negq	Dest	Dest = – Dest

Suffixes

char	**b**	**1**
short	**w**	**2**
int	**l**	**4**
long	**q**	**8**
pointer	**q**	**8**

Exercise: Assembly Operations

Register	Value
%rax	0x100
%rbx	0x108
%rdi	0x01

Address	Value
0x100	0x012
0x108	0x89a
0x110	0x909

1. `addq $0x47, %rax`
2. `addq %rbx, %rax`
3. `addq (%rbx), %rax`
4. `addq %rbx, (%rax)`
5. `addq 8(%rax,%rdi,8), %rax`

Sum	Location

Example: Translating Assembly

```
arith:
  orq     %rsi, %rdi
  sarq    $3, %rdi
  notq    %rdi
  movq    %rdx, %rax
  subq    %rdi, %rax
  ret
```

```
long arith(long x, long y, long z){
  x = x | y;
  x = x >> 3;
  x = ~x;

  long ret = z - x;
  return ret
}
```

Interesting Instructions

- **`sarq`**: arithmetic right shift

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rdx`	Argument **`z`**
`%rax`	return value

Exercise: Translating Assembly

```
arith:
  movq     %rdi, %rax
  addq     %rsi, %rax
  addq     %rdx, %rax
  movq     %rsi, %rdx
  salq     $3,   %rdx
  movq     $47,  %rcx
  addq     %rdx, %rcx
  imulq    %rcx, %rax
  ret
```

```
long arith(long x, long y,
           long z){




}
```

Interesting Instructions

- **`leaq`**: address computation
- **`salq`**: shift
- **`imulq`**: multiplication
  - But, only used once

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rdx`	Argument **`z`**
`%rax`	return value

lea Instruction

Scaled Memory Operands

`movq (%rdi,%rsi,8), %rax`

```
void ex(long* xp, long* yp){

  long* p = xp + 8*yp;
  long ret = *p;

}
```

`leaq` Source, Dest

`leaq (%rdi,%rsi,8), %rax`

```
void ex(long xp, long yp){
  long ret = xp + 8*yp;
}
```

- pointer arithmetic
  - E.g., p = x + i;
- arithmetic
  - expressions x + k*y (k=1, 2, 4, 8)

Converted to ASM by compiler:

```
long m12(long x){

  return x*12;

}
```

```
leaq (%rdi,%rdi,2), %rax # ret <- x+x*2
salq $2, %rax            # return ret<<2
```