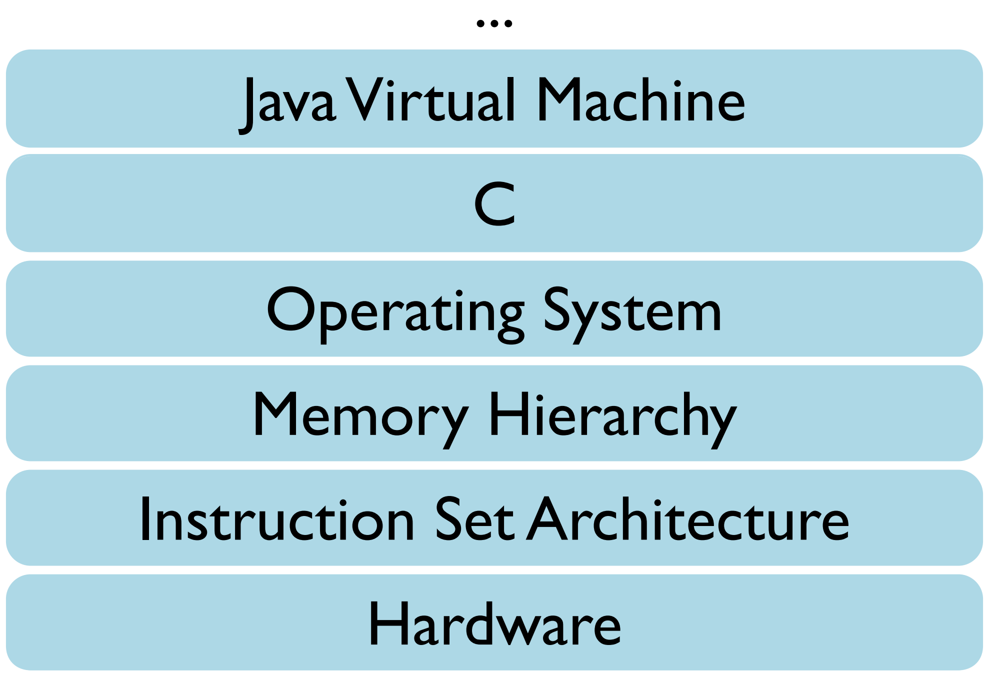
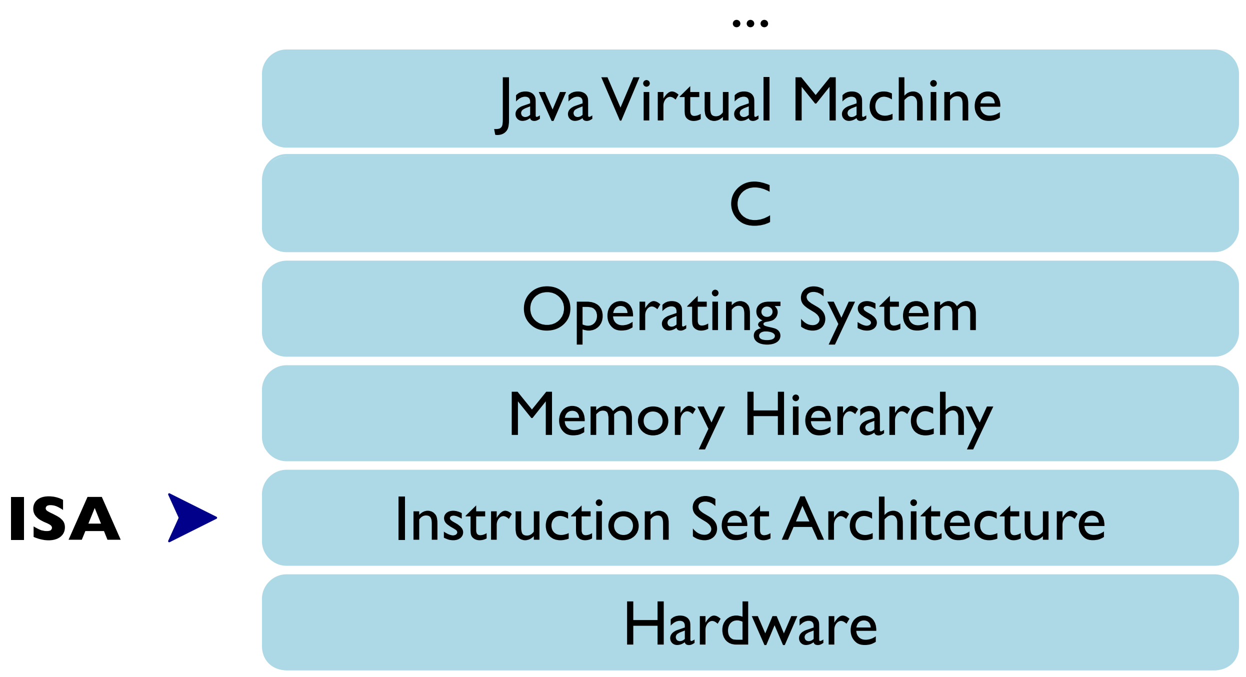
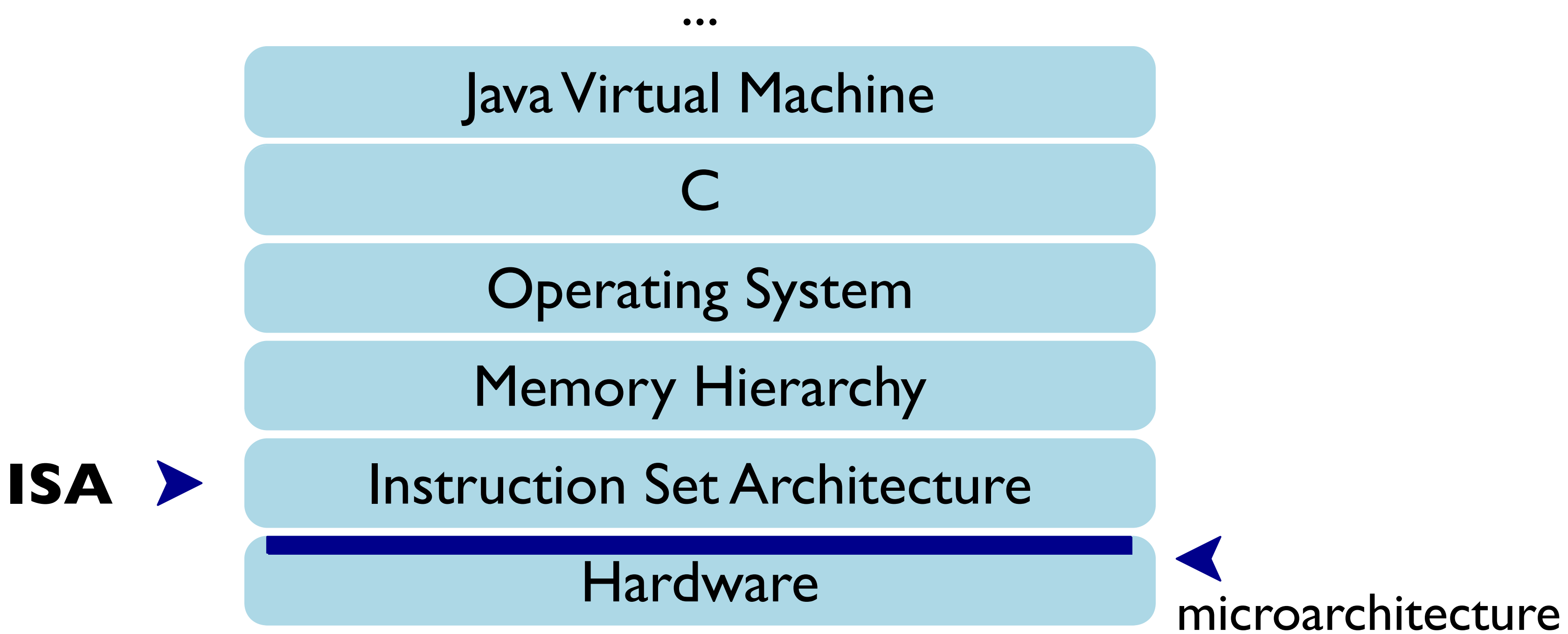
Machine Model



Machine Model



Machine Model



Instructions

```
pushq %rbx
movq %rdx, %rbx
call mult2
movq %rax, (%rbx)
popq %rbx
ret
```

Instructions

Assembly instructions

```
pushq %rbx
movq %rdx, %rbx
call mult2
movq %rax, (%rbx)
popq %rbx
ret
```

Instructions

Assembly instructions

```
pushq %rbx
movq %rdx, %rbx
call mult2
movq %rax, (%rbx)
popq %rbx
ret
```

Machine instructions

```
53
48 89 d3
e8 00 00 00 00
48 89 03
5b
c3
```

Example ISAs

- x86 and x86-64 — desktop and server
- ARM — mobile

- IA64 (Itanium)
- SPARC
- MIPS
- POWER (PowerPC)

- HPPA
- 680x0 (Motorola)
- 6502 (Motorola)

x86-64 Family History

1978: Intel 8086 and 80286

- 16-bit words

1980s: Intel 80386 and 486

- 32-bit words

1990s: Intel Pentium, AMD K5 and K6

- Still 32-bit words

2000s: AMD Athlon 64, Intel Core

- 64-bit words

Backward compatibility maintained for nearly 40 years!

Debate of the (Past) Century

CISC

complex instruction set

- Directly provide common operations
- Mixed instruction sizes; common things are compact

x86

RISC

reduced instruction set

- Provide simple and uniform primitives
- Similarly simple and uniform encoding

SPARC

MPIS

POWER

ARM

Debate of the (Past) Century

CISC
complex instruction set

- Directly provide common operations
- Mixed instruction sizes; common things are compact

RISC
reduced instruction set

- Provide simple and uniform primitives
- Similarly simple and uniform encoding

x86

CISC

SPARC

MPIS

POWER

ARM

Debate of the (Past) Century

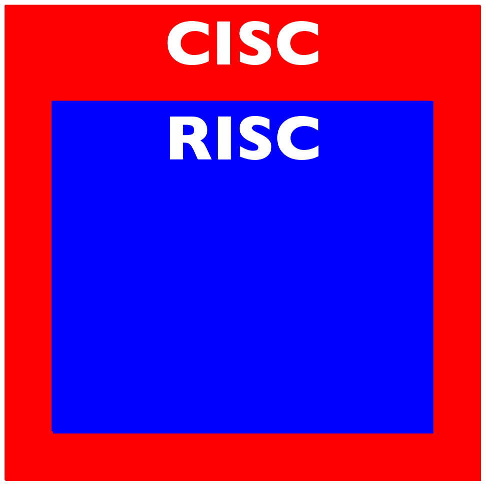
CISC
complex instruction set

- Directly provide common operations
- Mixed instruction sizes; common things are compact

RISC
reduced instruction set

- Provide simple and uniform primitives
- Similarly simple and uniform encoding

x86



SPARC

MPIS

POWER

ARM

Debate of the (Past) Century

CISC
complex instruction set

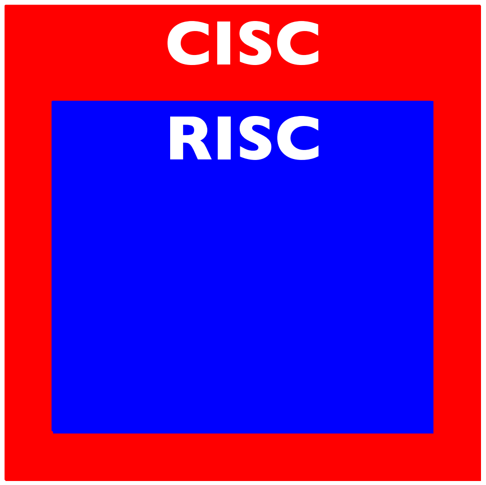
- Directly provide common operations
- Mixed instruction sizes; common things are compact

RISC
reduced instruction set

- Provide simple and uniform primitives
- Similarly simple and uniform encoding

x86

ARM Thumb



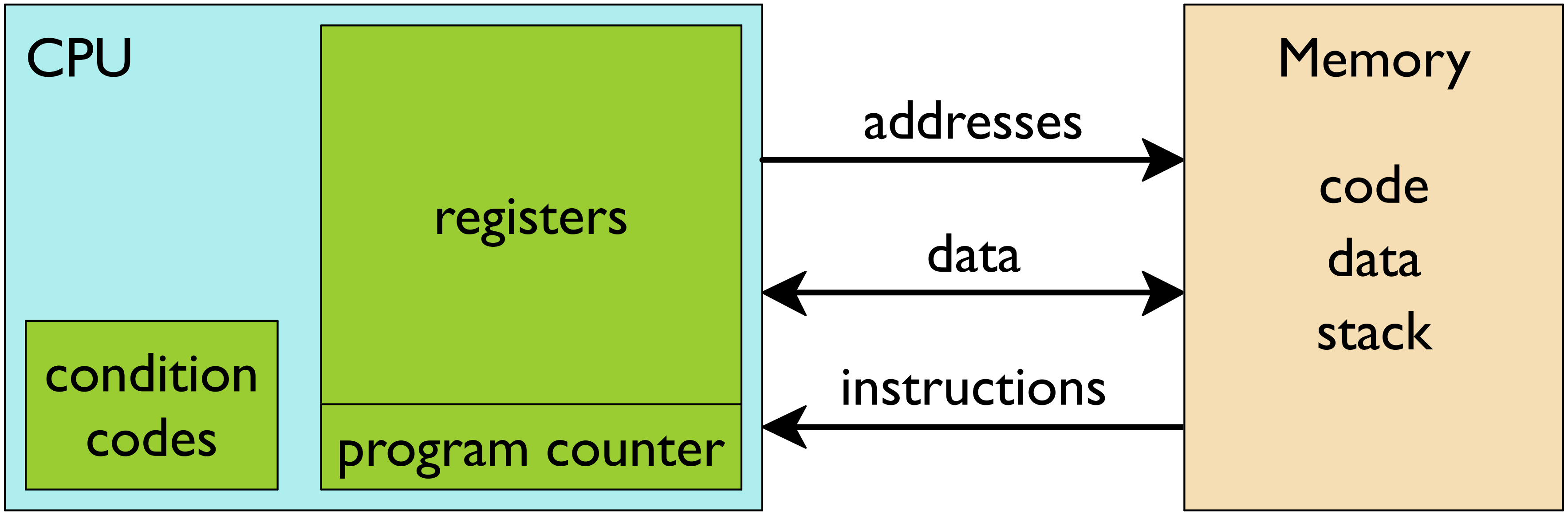
SPARC

MPIS

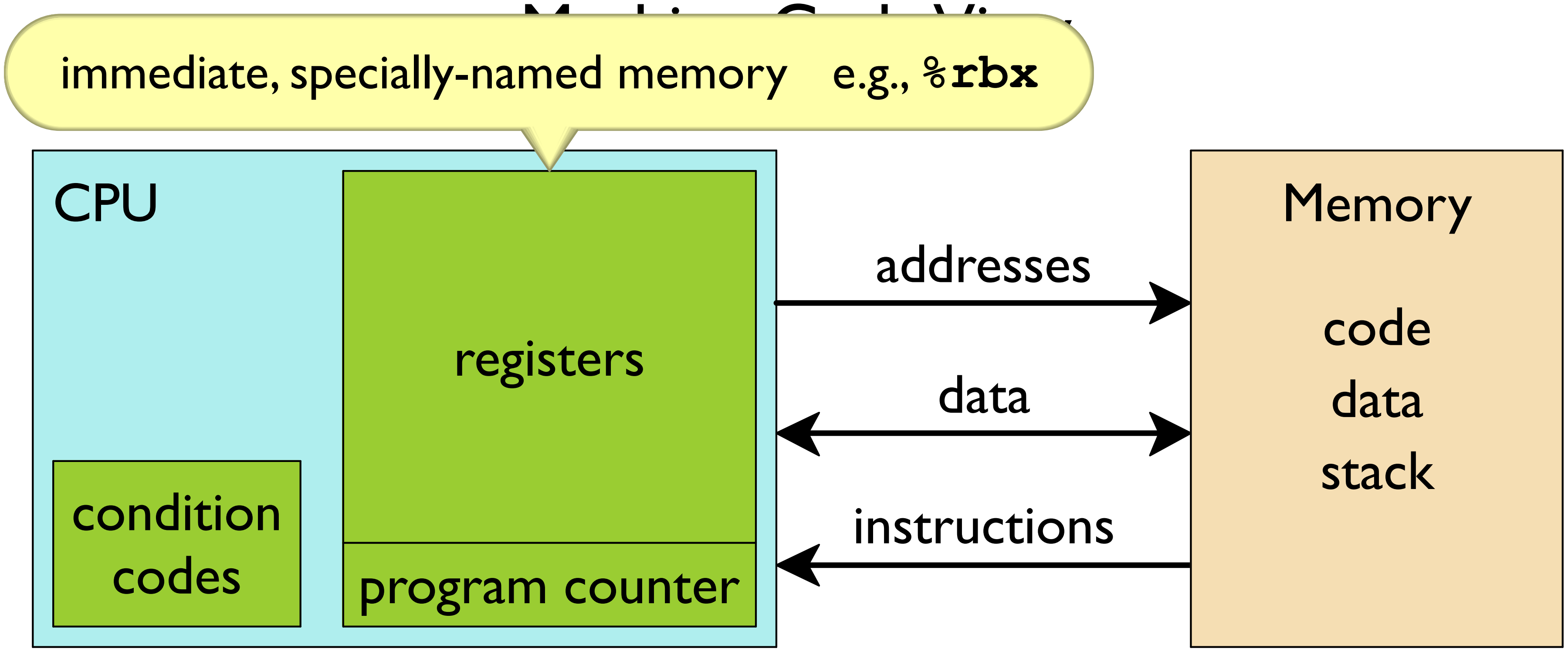
POWER

ARM

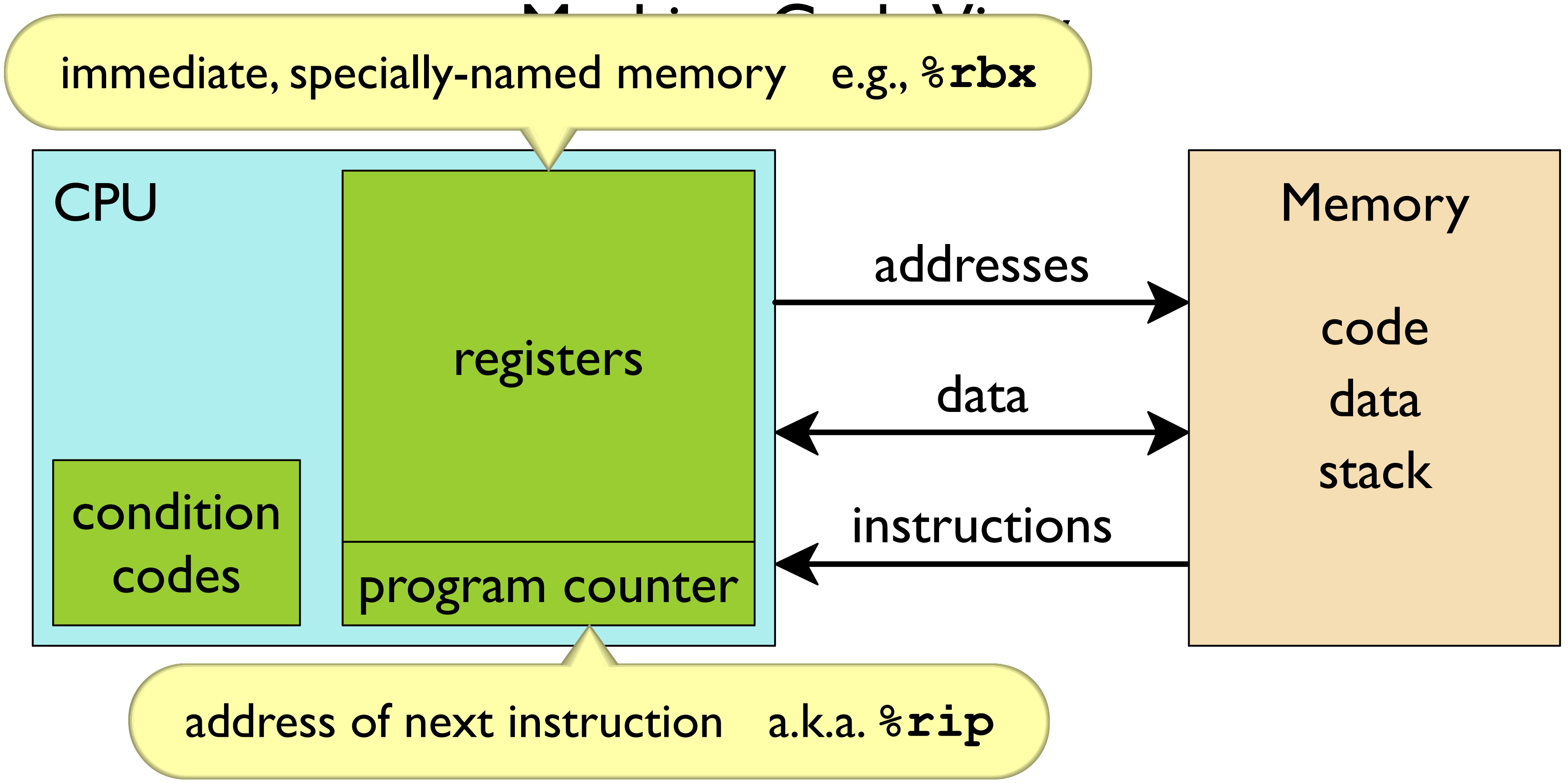
Machine Code View

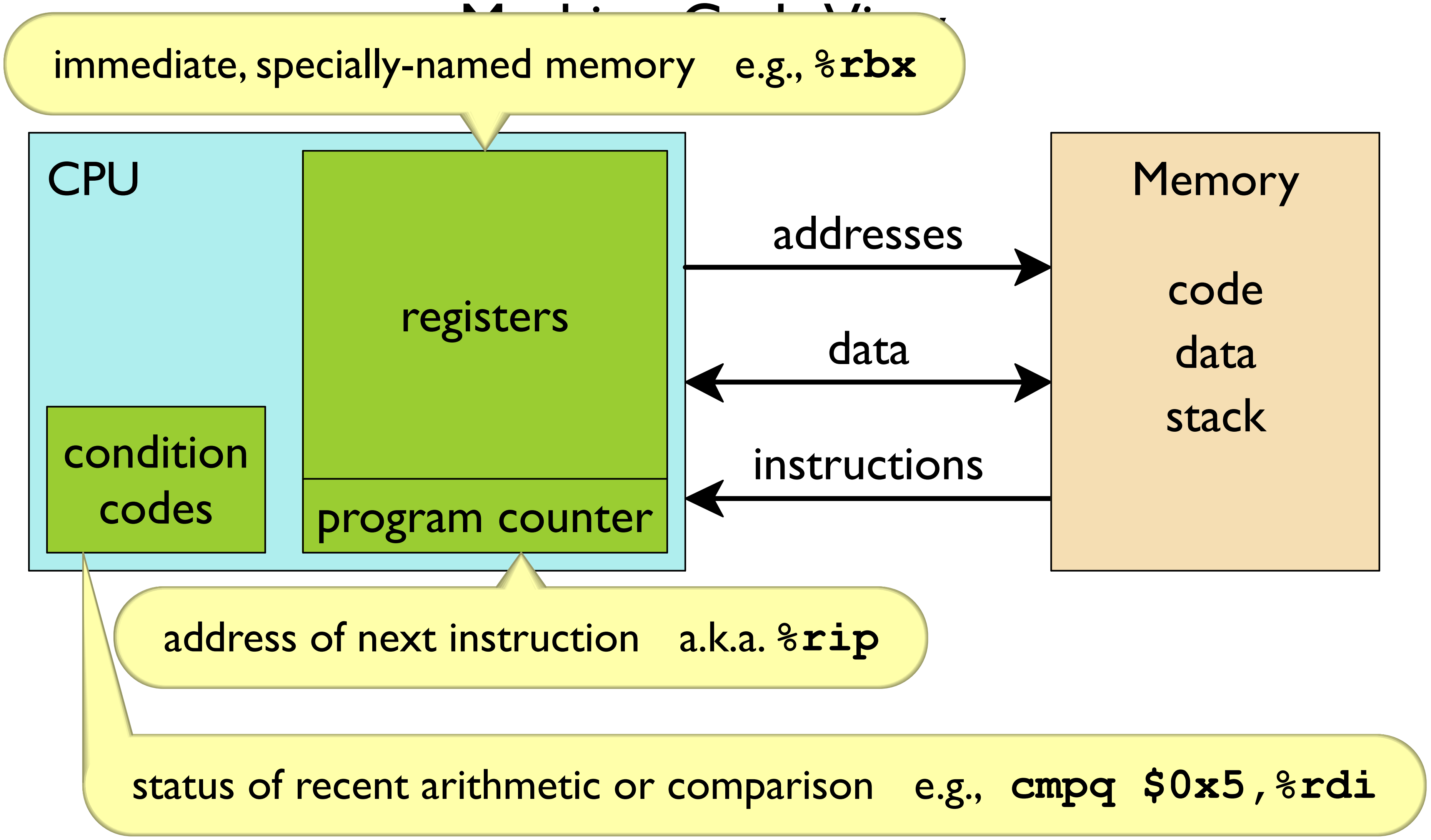


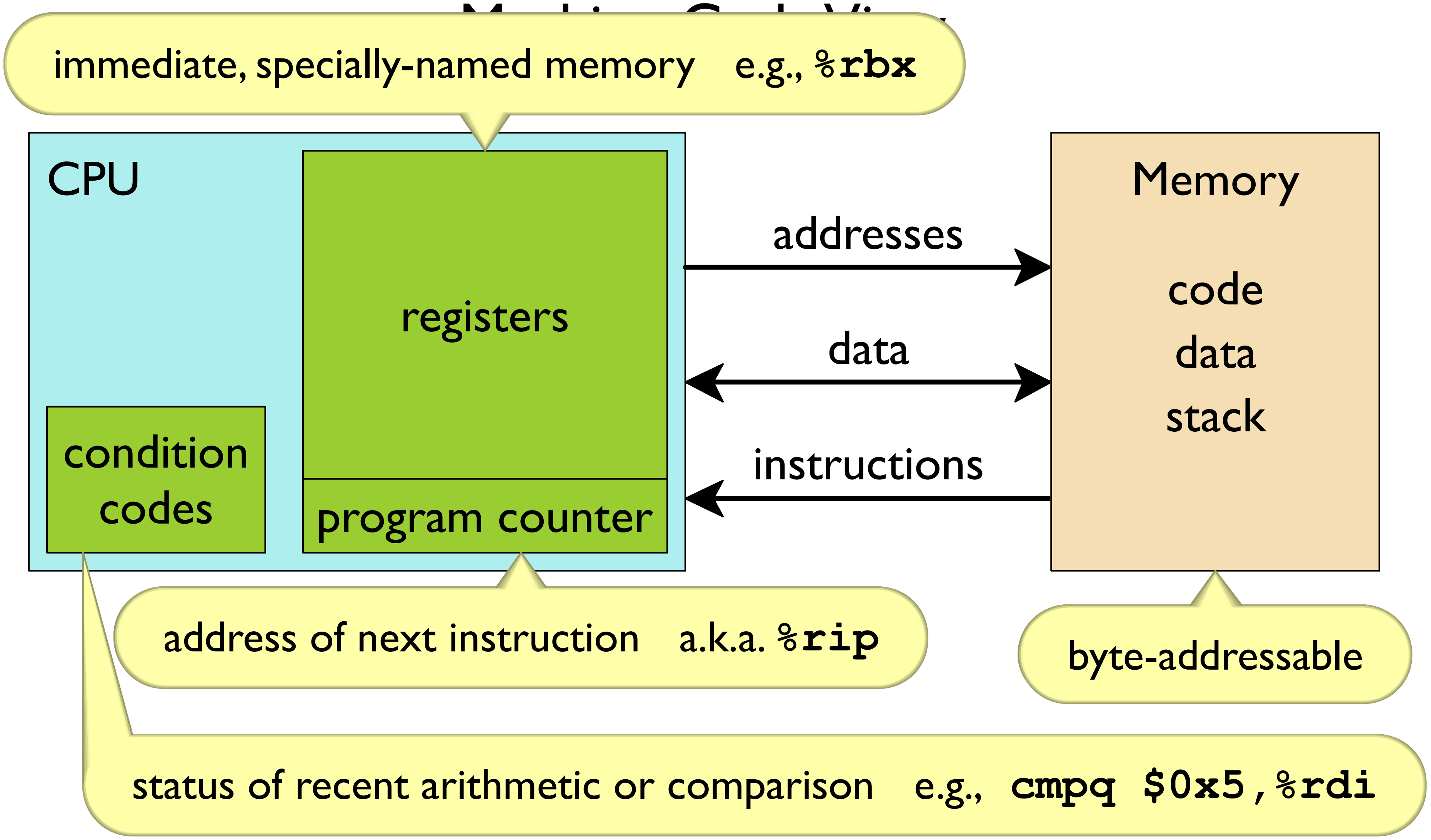
Machine Code View



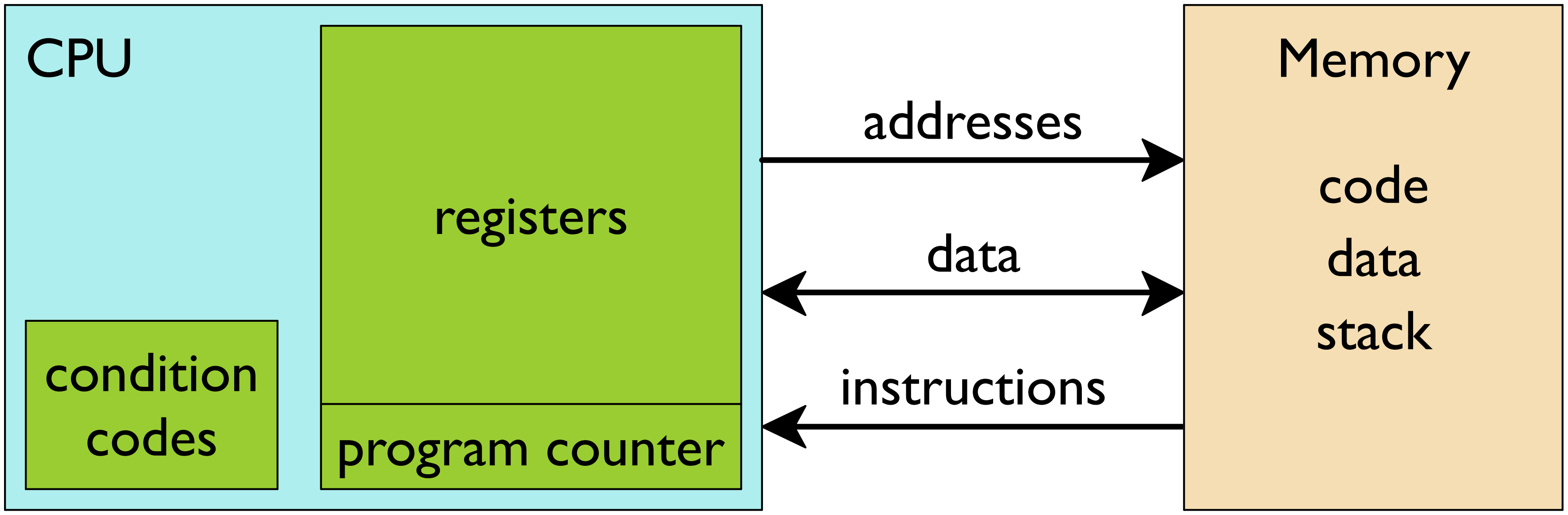
Machine Code View




immediate, specially-named memory e.g., %rbx
CPU
registers
condition codes
program counter
addresses
data
instructions
Memory
code
data
stack
address of next instruction a.k.a. %rip
status of recent arithmetic or comparison e.g., cmpq $0x5,%rdi


immediate, specially-named memory e.g., %rbx
CPU
registers
condition codes
program counter
Memory
code
data
stack
addresses
data
instructions
address of next instruction a.k.a. %rip
byte-addressable
status of recent arithmetic or comparison e.g., cmpq $0x5,%rdi

Machine Code View

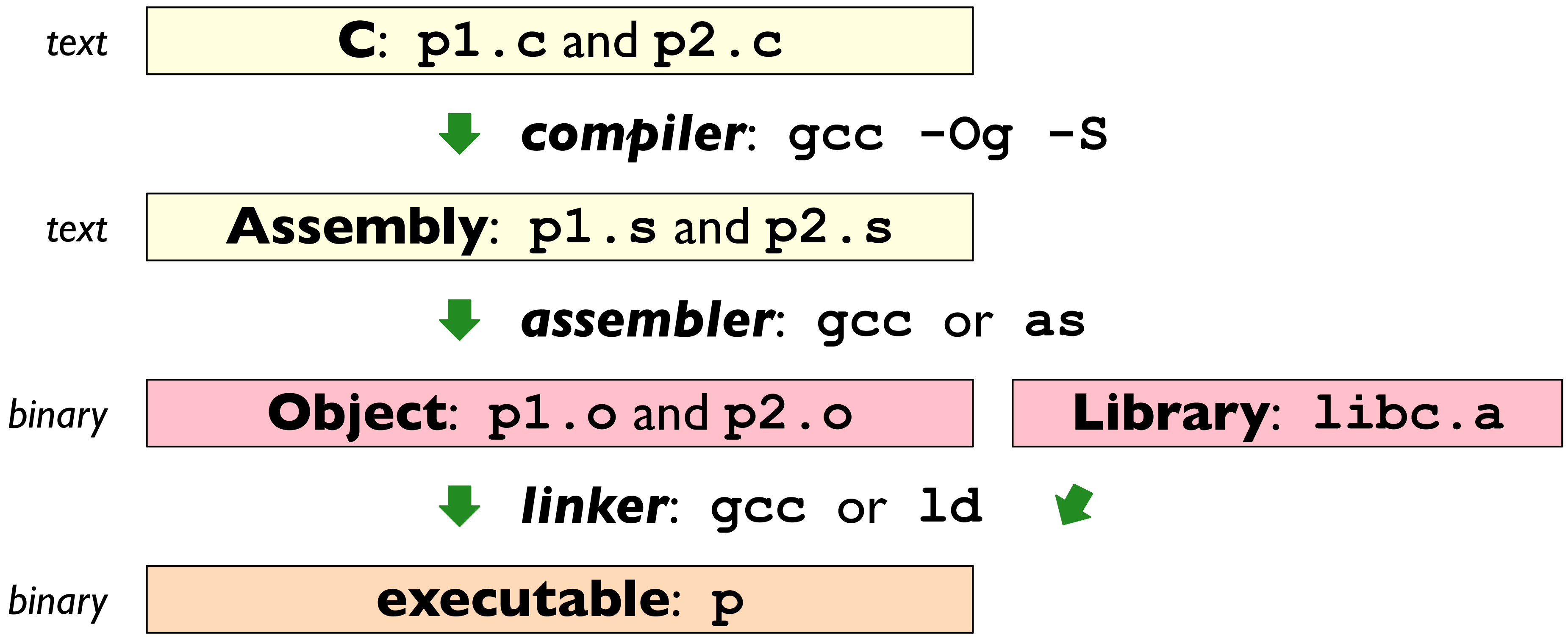


Instruction categories:

- **Arithmetic:** perform arithmetic on register or memory
- **Data:** transfer data between memory and registers
- **Control:** make program counter jump, (un)conditionally

Turning C into Machine Code

```
$ gcc -Og p1.c p2.c -lc -o p
$ ./p
```



C-to-Machine Example

```c
long mult2(long a, long b);

void multstore(long x, long y, long *dest) {
   long t = mult2(x, y);
   *dest = t;
}
```

Copy

```
$ gcc -S -Og multstore.c
```

```
pushq %rbx
movq %rdx, %rbx
call mult2
movq %rax, (%rbx)
popq %rbx
ret
```

```
53
48 89 d3
e8 00 00 00 00
48 89 03
5b
c3
```

```
$ gcc -c multstore.s or gcc -c -Og multstore.c
$ objdump -d multstore.o
```

Link

```
#include <stdio.h>
void multstore(long x, long y, long *dest);

long mult2(long a, long b) {
  return a*b;
}

int main() {
  long r;
  multstore(2, 3, &r);
  printf("%ld\n", r);
  return 0;
}
```

Copy

```
$ gcc -c -Og main.c
$ gcc -o m main.o multstore.o
$ ./m
$ objdump -d m
```

Data Formats

An assembly instruction's suffix indicates the size of the operand

Example: **`movq`** moves a "quad" word

Because x86 started as a 16-bit architecture:

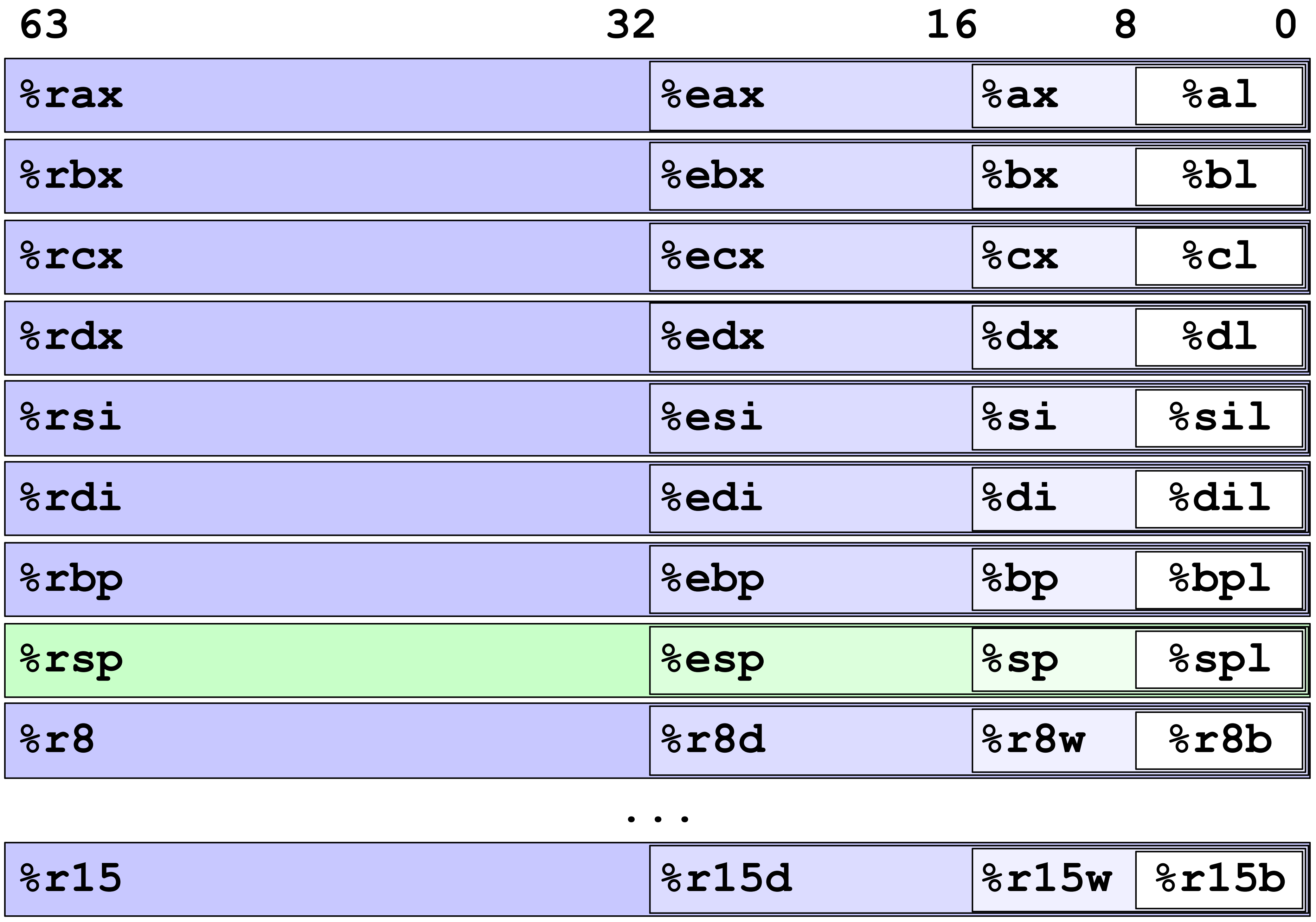
`b`	"byte"	8 bits	**`char`**
`w`	"word"	16 bits	**`short`**
`l`	"long word"	32 bits	**`int`**
`q`	"quad word"	64 bits	**`long`**

Floating-point operations:

`s`	"single precision"	32 bits	**`float`**
`l`	"double precision"	64 bits	**`double`**

No aggregate types, such as arrays and structures. Those are generated by the compiler from these primitive formats.

Register Names



%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
. . .			
%r15	%r15d	%r15w	%r15b

Operand Examples

M[*addr*] = the value stored at *addr* in memory

R[*reg*] = the value stored in the register named *reg*

`%eax`	register	R[`%eax`]
`$0x2a3`	literal	0x2a3
`0x2a3`	absolute	M[0x2a3]
`(%eax)`	indirect	M[R[`%eax`]]
`7(%edx)`	base + displacement	M[7 + R[`%edx`]]
`(%eax,%ecx)`	indexed	M[R[`%eax`] + R[`%ecx`]]
`7(%eax,%ecx)`	indexed	M[7 + R[`%eax`] + R[`%ecx`]]
`(,%eax,4)`	scaled indexed	M[R[`%eax`] × 4]
`7(,%eax,4)`	scaled indexed	M[7 + R[`%eax`] × 4]
`(%eax,%ecx,4)`	scaled indexed	M[R[`%eax`] + R[`%ecx`] × 4]
`7(%eax,%ecx,4)`	scaled indexed	M[7 + R[`%eax`] + R[`%ecx`] × 4]

Operand Practice

CPU

register	value
`%eax`	`0x200`
`%ecx`	`0x41`
`%edx`	`0x4`

Memory

address	value
`0x200`	`0x12`
`0x204`	`0x2a`
`0x208`	`0xd4`
`0x20c`	`0xfd`

operand	meaning	value
`$0x204`		
`%eax`		
`(%eax)`		
`0x208`		
`(%edx,%ecx,8)`		
`0x1f8(,%edx,4)`		

Operand Practice

CPU

register	value
`%eax`	`0x200`
`%ecx`	`0x41`
`%edx`	`0x4`

Memory

address	value
`0x200`	`0x12`
`0x204`	`0x2a`
`0x208`	`0xd4`
`0x20c`	`0xfd`

operand	meaning	value
`$0x204`	0x204	
`%eax`		
`(%eax)`		
`0x208`		
`(%edx,%ecx,8)`		
`0x1f8(,%edx,4)`		

Operand Practice

CPU

register	value
`%eax`	`0x200`
`%ecx`	`0x41`
`%edx`	`0x4`

Memory

address	value
`0x200`	`0x12`
`0x204`	`0x2a`
`0x208`	`0xd4`
`0x20c`	`0xfd`

operand	meaning	value
`$0x204`	0x204	0x204 = 516
`%eax`		
`(%eax)`		
`0x208`		
`(%edx,%ecx,8)`		
`0x1f8(,%edx,4)`		

Operand Practice

CPU

register	value
`%eax`	`0x200`
`%ecx`	`0x41`
`%edx`	`0x4`

Memory

address	value
`0x200`	`0x12`
`0x204`	`0x2a`
`0x208`	`0xd4`
`0x20c`	`0xfd`

operand	meaning	value
`$0x204`	0x204	0x204 = 516
`%eax`	R[`%eax`]	
`(%eax)`		
`0x208`		
`(%edx,%ecx,8)`		
`0x1f8(,%edx,4)`		

Operand Practice

CPU

register	value
`%eax`	**`0x200`**
`%ecx`	**`0x41`**
`%edx`	**`0x4`**

Memory

address	value
`0x200`	**`0x12`**
`0x204`	**`0x2a`**
`0x208`	**`0xd4`**
`0x20c`	**`0xfd`**

operand	meaning	value
`$0x204`	0x204	0x204 = 516
`%eax`	R[**`%eax`**]	0x200 = 512
`(%eax)`		
`0x208`		
`(%edx,%ecx,8)`		
`0x1f8(,%edx,4)`		

Operand Practice

CPU

register	value
`%eax`	**`0x200`**
`%ecx`	**`0x41`**
`%edx`	**`0x4`**

Memory

address	value
`0x200`	**`0x12`**
`0x204`	**`0x2a`**
`0x208`	**`0xd4`**
`0x20c`	**`0xfd`**

operand	meaning	value
`$0x204`	0x204	0x204 = 516
`%eax`	R[**`%eax`**]	0x200 = 512
`(%eax)`	M[R[**`%eax`**]] = M[0x200]	
`0x208`		
`(%edx,%ecx,8)`		
`0x1f8(,%edx,4)`		

Operand Practice

CPU

register	value
`%eax`	`0x200`
`%ecx`	`0x41`
`%edx`	`0x4`

Memory

address	value
`0x200`	`0x12`
`0x204`	`0x2a`
`0x208`	`0xd4`
`0x20c`	`0xfd`

operand	meaning	value
`$0x204`	0x204	0x204 = 516
`%eax`	R[`%eax`]	0x200 = 512
`(%eax)`	M[R[`%eax`]] = M[0x200]	0x12 = 18
`0x208`		
`(%edx,%ecx,8)`		
`0x1f8(,%edx,4)`		

Operand Practice

CPU

register	value
`%eax`	`0x200`
`%ecx`	`0x41`
`%edx`	`0x4`

Memory

address	value
`0x200`	`0x12`
`0x204`	`0x2a`
`0x208`	`0xd4`
`0x20c`	`0xfd`

operand	meaning	value
`$0x204`	0x204	0x204 = 516
`%eax`	R[`%eax`]	0x200 = 512
`(%eax)`	M[R[`%eax`]] = M[0x200]	0x12 = 18
`0x208`	M[0x208]	
`(%edx,%ecx,8)`		
`0x1f8(,%edx,4)`		

Operand Practice

CPU

register	value
`%eax`	`0x200`
`%ecx`	`0x41`
`%edx`	`0x4`

Memory

address	value
`0x200`	`0x12`
`0x204`	`0x2a`
`0x208`	`0xd4`
`0x20c`	`0xfd`

operand	meaning	value
`$0x204`	0x204	0x204 = 516
`%eax`	R[`%eax`]	0x200 = 512
`(%eax)`	M[R[`%eax`]] = M[0x200]	0x12 = 18
`0x208`	M[0x208]	0xd4 = 212
`(%edx,%ecx,8)`		
`0x1f8(,%edx,4)`		

Operand Practice

CPU

register	value
`%eax`	`0x200`
`%ecx`	`0x41`
`%edx`	`0x4`

Memory

address	value
`0x200`	`0x12`
`0x204`	`0x2a`
`0x208`	`0xd4`
`0x20c`	`0xfd`

operand	meaning	value
`$0x204`	0x204	0x204 = 516
`%eax`	R[`%eax`]	0x200 = 512
`(%eax)`	M[R[`%eax`]] = M[0x200]	0x12 = 18
`0x208`	M[0x208]	0xd4 = 212
`(%edx,%ecx,8)`	M[R[`%edx`]+R[`%ecx`]×8] = M[0x4+0x41×8] = M[0x20c]	
`0x1f8(,%edx,4)`		

Operand Practice

CPU

register	value
`%eax`	**`0x200`**
`%ecx`	**`0x41`**
`%edx`	**`0x4`**

Memory

address	value
`0x200`	**`0x12`**
`0x204`	**`0x2a`**
`0x208`	**`0xd4`**
`0x20c`	**`0xfd`**

operand	meaning	value
`$0x204`	0x204	0x204 = 516
`%eax`	R[**`%eax`**]	0x200 = 512
`(%eax)`	M[R[**`%eax`**]] = M[0x200]	0x12 = 18
`0x208`	M[0x208]	0xd4 = 212
`(%edx,%ecx,8)`	M[R[**`%edx`**]+R[**`%ecx`**]×8] = M[0x4+0x41×8] = M[0x20c]	0xfd = 253
`0x1f8(,%edx,4)`		

Operand Practice

CPU

register	value
`%eax`	`0x200`
`%ecx`	`0x41`
`%edx`	`0x4`

Memory

address	value
`0x200`	`0x12`
`0x204`	`0x2a`
`0x208`	`0xd4`
`0x20c`	`0xfd`

operand	meaning	value
`$0x204`	0x204	0x204 = 516
`%eax`	R[`%eax`]	0x200 = 512
`(%eax)`	M[R[`%eax`]] = M[0x200]	0x12 = 18
`0x208`	M[0x208]	0xd4 = 212
`(%edx,%ecx,8)`	M[R[`%edx`]+R[`%ecx`]×8] = M[0x4+0x41×8] = M[0x20c]	0xfd = 253
`0x1f8(,%edx,4)`	M[0x1f8+0+R[`%edx`]×4] = M[0x1f8+0+0x4×4] = M[0x208]	

Operand Practice

CPU

register	value
`%eax`	**`0x200`**
`%ecx`	**`0x41`**
`%edx`	**`0x4`**

Memory

address	value
`0x200`	**`0x12`**
`0x204`	**`0x2a`**
`0x208`	**`0xd4`**
`0x20c`	**`0xfd`**

operand	meaning	value
`$0x204`	0x204	0x204 = 516
`%eax`	R[**`%eax`**]	0x200 = 512
`(%eax)`	M[R[**`%eax`**]] = M[0x200]	0x12 = 18
`0x208`	M[0x208]	0xd4 = 212
`(%edx,%ecx,8)`	M[R[**`%edx`**]+R[**`%ecx`**]×8] = M[0x4+0x41×8] = M[0x20c]	0xfd = 253
`0x1f8(,%edx,4)`	M[0x1f8+0+R[**`%edx`**]×4] = M[0x1f8+0+0x4×4] = M[0x208]	0xd4 = 212

Copying Data

movx *source* **,** *dest*

Example:

```
movb $0xF, (%rbx)
```

movb ⇒ copy a byte

$0xF ⇒ copy the literal value 0xF

(%rbx) ⇒ copy it to the memory pointed at by **%rbx**

Copying Data

`movx` *source* **`,`** *dest*

Example:

```
movw %ax, (%rsp)
```

`movw` ⇒ copy two bytes

`%ax` ⇒ copy it from the register **`%ax`**

`(%rsp)` ⇒ copy it to the current stack

Stack Shortcuts

push_x_ _source_

pop_x_ _dest_

Combines an adjustment of **%rsp** with a copy to/from memory it points to (i.e., the stack)

The stack grows ``down"

Example:

pushl %ebp

pushl ⇒ copy four bytes

pushl ⇒ decrement **%rsp** by four bytes

%ebp ⇒ copy four bytes from **%ebp** to memory now pointed at by **%rsp**

Stack Shortcuts

`push`*x* *source*

`pop`*x* *dest*

Combines an adjustment of **`%rsp`** with a copy to/from memory it points to (i.e., the stack)

The stack grows ``down"

Example:

```
popq %rax
```

`popq` ⇒ copy eight bytes

`%rax` ⇒ copy eight bytes from memory now pointed at by **`%rsp`** to **`%rax`**

`popq` ⇒ increment **`%rsp`** by eight bytes

Exercise: Data Movement

```
int exchange(int *xp, int y) {
  int x = *xp;
  *xp = y;
  return x;
}
```

without -Og, roughly

```
...
movq -0x18(%rbp),%rax
movl -0x1c(%rbp),%edx
movl (%rax),%ecx
movl %edx,(%rax)
movl %ecx,%eax
...
```

Exercise: Data Movement

```
int exchange(int *xp, int y) {
  int x = *xp;
  *xp = y;
  return x;
}
```

without -Og, roughly

```
...
movq -0x18(%rbp),%rax
movl -0x1c(%rbp),%edx
movl (%rax),%ecx
movl %edx,(%rax)
movl %ecx,%eax
...
```

`%rax` = `xp`

Exercise: Data Movement

```
int exchange(int *xp, int y) {
  int x = *xp;
  *xp = y;
  return x;
}
```

without -Og, roughly

```
...
movq -0x18(%rbp),%rax
movl -0x1c(%rbp),%edx
movl (%rax),%ecx
movl %edx,(%rax)
movl %ecx,%eax
...
```

`%rax` = `xp`

`%edx` = `y`

Exercise: Data Movement

```
int exchange(int *xp, int y) {
  int x = *xp;
  *xp = y;
  return x;
}
```

without -Og, roughly

```
...
movq -0x18(%rbp),%rax
movl -0x1c(%rbp),%edx
movl (%rax),%ecx
movl %edx,(%rax)
movl %ecx,%eax
...
```

%rax = xp
%edx = y
%ecx = *xp

Exercise: Data Movement

```
int exchange(int *xp, int y) {
  int x = *xp;
  *xp = y;
  return x;
}
```

without -Og, roughly

```
...
movq -0x18(%rbp),%rax
movl -0x1c(%rbp),%edx
movl (%rax),%ecx
movl %edx,(%rax)
movl %ecx,%eax
...
```

%rax = **xp**

%edx = **y**

%ecx = ***xp**

put **y** in ***xp**

Exercise: Data Movement

```
int exchange(int *xp, int y) {
  int x = *xp;
  *xp = y;
  return x;
}
```

without -Og, roughly

```
...
movq -0x18(%rbp),%rax
movl -0x1c(%rbp),%edx
movl (%rax),%ecx
movl %edx,(%rax)
movl %ecx,%eax
...
```

`movq -0x18(%rbp),%rax`	`%rax` = `xp`
`movl -0x1c(%rbp),%edx`	`%edx` = `y`
`movl (%rax),%ecx`	`%ecx` = `*xp`
`movl %edx,(%rax)`	put `y` in `*xp`
`movl %ecx,%eax`	`%eax` = old `*xp`

Exercise: Data Movement

```
void decode1(int *xp, int *yp, int *zp) {




}
```

```
movq 8(%rbp),%rdi
movq 16(%rbp),%rbx
movq 24(%rbp),%rsi
movl (%rdi),%eax
movl (%rbx),%edx
movl (%rsi),%ecx
movl %eax,(%rbx)
movl %edx,(%rsi)
movl %ecx,(%rdi)
```

Exercise: Data Movement

```
void decode1(int *xp, int *yp, int *zp) {




}
```

```
movq 8(%rbp),%rdi
movq 16(%rbp),%rbx
movq 24(%rbp),%rsi
movl (%rdi),%eax
movl (%rbx),%edx
movl (%rsi),%ecx
movl %eax,(%rbx)
movl %edx,(%rsi)
movl %ecx,(%rdi)
```

%rdi = xp

Exercise: Data Movement

```
void decode1(int *xp, int *yp, int *zp) {




}
```

```
movq 8(%rbp),%rdi
movq 16(%rbp),%rbx
movq 24(%rbp),%rsi
movl (%rdi),%eax
movl (%rbx),%edx
movl (%rsi),%ecx
movl %eax,(%rbx)
movl %edx,(%rsi)
movl %ecx,(%rdi)
```

%rdi = **xp**

%rbx = **yp**

Exercise: Data Movement

```
void decode1(int *xp, int *yp, int *zp) {




}
```

```
movq 8(%rbp),%rdi
movq 16(%rbp),%rbx
movq 24(%rbp),%rsi
movl (%rdi),%eax
movl (%rbx),%edx
movl (%rsi),%ecx
movl %eax,(%rbx)
movl %edx,(%rsi)
movl %ecx,(%rdi)
```

%rdi = **xp**

%rbx = **yp**

%rsi = **zp**

Exercise: Data Movement

```
void decode1(int *xp, int *yp, int *zp) {




}
```

```
movq 8(%rbp),%rdi
movq 16(%rbp),%rbx
movq 24(%rbp),%rsi
movl (%rdi),%eax
movl (%rbx),%edx
movl (%rsi),%ecx
movl %eax,(%rbx)
movl %edx,(%rsi)
movl %ecx,(%rdi)
```

```
%rdi = xp
%rbx = yp
%rsi = zp
%eax = *xp
```

Exercise: Data Movement

```
void decode1(int *xp, int *yp, int *zp) {





}
```

```
movq 8(%rbp),%rdi
movq 16(%rbp),%rbx
movq 24(%rbp),%rsi
movl (%rdi),%eax
movl (%rbx),%edx
movl (%rsi),%ecx
movl %eax,(%rbx)
movl %edx,(%rsi)
movl %ecx,(%rdi)
```

%rdi = **xp**
%rbx = **yp**
%rsi = **zp**
%eax = ***xp**
%edx = ***yp**

Exercise: Data Movement

```
void decode1(int *xp, int *yp, int *zp) {




}
```

```
movq 8(%rbp),%rdi
movq 16(%rbp),%rbx
movq 24(%rbp),%rsi
movl (%rdi),%eax
movl (%rbx),%edx
movl (%rsi),%ecx
movl %eax,(%rbx)
movl %edx,(%rsi)
movl %ecx,(%rdi)
```

```
%rdi = xp
%rbx = yp
%rsi = zp
%eax = *xp
%edx = *yp
%ecx = *zp
```

Exercise: Data Movement

```
void decode1(int *xp, int *yp, int *zp) {




}
```

```
movq 8(%rbp),%rdi
movq 16(%rbp),%rbx
movq 24(%rbp),%rsi
movl (%rdi),%eax
movl (%rbx),%edx
movl (%rsi),%ecx
movl %eax,(%rbx)
movl %edx,(%rsi)
movl %ecx,(%rdi)
```

%rdi = **xp**

%rbx = **yp**

%rsi = **zp**

%eax = ***xp**

%edx = ***yp**

%ecx = ***zp**

put ***xp** in ***yp**

Exercise: Data Movement

```
void decode1(int *xp, int *yp, int *zp) {




}
```

```
movq 8(%rbp),%rdi
movq 16(%rbp),%rbx
movq 24(%rbp),%rsi
movl (%rdi),%eax
movl (%rbx),%edx
movl (%rsi),%ecx
movl %eax,(%rbx)
movl %edx,(%rsi)
movl %ecx,(%rdi)
```

%rdi = **xp**

%rbx = **yp**

%rsi = **zp**

%eax = ***xp**

%edx = ***yp**

%ecx = ***zp**

put ***xp** in ***yp**

put old ***yp** in ***zp**

Exercise: Data Movement

```
void decode1(int *xp, int *yp, int *zp) {




}
```

`movq 8(%rbp),%rdi`	`%rdi` = `xp`
`movq 16(%rbp),%rbx`	`%rbx` = `yp`
`movq 24(%rbp),%rsi`	`%rsi` = `zp`
`movl (%rdi),%eax`	`%eax` = `*xp`
`movl (%rbx),%edx`	`%edx` = `*yp`
`movl (%rsi),%ecx`	`%ecx` = `*zp`
`movl %eax,(%rbx)`	put `*xp` in `*yp`
`movl %edx,(%rsi)`	put old `*yp` in `*zp`
`movl %ecx,(%rdi)`	put old `*zp` in `*xp`

Exercise: Data Movement

```
void decode1(int *xp, int *yp, int *zp) {
  int old_y = *yp, old_z = *zp;
  *yp = *xp;
  *zp = old_y;
  *xp = old_z;
}
```

`movq 8(%rbp),%rdi`	`%rdi` = `xp`
`movq 16(%rbp),%rbx`	`%rbx` = `yp`
`movq 24(%rbp),%rsi`	`%rsi` = `zp`
`movl (%rdi),%eax`	`%eax` = `*xp`
`movl (%rbx),%edx`	`%edx` = `*yp`
`movl (%rsi),%ecx`	`%ecx` = `*zp`
`movl %eax,(%rbx)`	put `*xp` in `*yp`
`movl %edx,(%rsi)`	put old `*yp` in `*zp`
`movl %ecx,(%rdi)`	put old `*zp` in `*xp`

Arithmetic Operations

add*x* *source*, *dest*	*dest* = *dest* + *source*
sub*x* *source*, *dest*	*dest* = *dest* − *source*
imul*x* *source*, *dest*	*dest* = *dest* * *source*
sal*x* *source*, *dest*	signed *dest* = *dest* << *source*
sar*x* *source*, *dest*	signed *dest* = *dest* >> *source*
shl*x* *source*, *dest*	unsigned *dest* = *dest* << *source*
shr*x* *source*, *dest*	unsigned *dest* = *dest* >> *source*
xor*x* *source*, *dest*	*dest* = *dest* ^ *source*
and*x* *source*, *dest*	*dest* = *dest* & *source*
or*x* *source*, *dest*	*dest* = *dest* | *source*

Shortcuts: Simple Arithmetic

inc*x* *dest*	*dest* = *dest* + 1
dec*x* *dest*	*dest* = *dest* – 1
neg*x* *dest*	*dest* = – *dest*
not*x* *dest*	*dest* = ~ *dest*

Shortcut: Computing Addresses

`lea`*x* *source*`,` *destRegister*

Like **`mov`***x*, but only computes an address that contains a value; the destination is always a regsiter

The same effect could be achieved with arithmetic operations

Example:

```
leaq 4(%rbx, %rsp), %rax
```

`leaq` ⇒ compute address as eight bytes

`4(%rbx, %rsp)` ⇒ add **`%rbx`** and **`%rsp`** and 4

`%rax` ⇒ put that address in **`%rax`**

Shortcut: Computing Addresses

`lea`*x* *source* **`,`** *destRegister*

Like **`mov`***x*, but only computes an address that contains a value; the destination is always a regsiter

The same effect could be achieved with arithmetic operations

Example:

```
leaq (%rsp), %rax
```

`leaq` ⇒ compute address as eight bytes

`(%rsp)` ⇒ use the address in the register **`%rsp`**

`%rax` ⇒ put that same address in **`%rax`**

Example: Arithmetic

```
long arith(long x, long y, long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
```

```
...
leaq    (%rdi,%rsi), %rax
addq    %rdx, %rax
leaq    (%rsi,%rsi,2), %rdx
salq    $4, %rdx
leaq    4(%rdi,%rdx), %rcx
imulq   %rcx, %rax
ret
```

Example: Arithmetic

```
long arith(long x, long y, long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
```

```
...
leaq    (%rdi,%rsi), %rax
addq    %rdx, %rax
leaq    (%rsi,%rsi,2), %rdx
salq    $4, %rdx
leaq    4(%rdi,%rdx), %rcx
imulq   %rcx, %rax
ret
```

`%rax` = `t1` = `x+y`

Example: Arithmetic

```
long arith(long x, long y, long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
```

```
...
leaq    (%rdi,%rsi), %rax
addq    %rdx, %rax
leaq    (%rsi,%rsi,2), %rdx
salq    $4, %rdx
leaq    4(%rdi,%rdx), %rcx
imulq   %rcx, %rax
ret
```

`%rax` = `t1` = `x+y`

`%rax` = `t2` = `z+t1`

Example: Arithmetic

```
long arith(long x, long y, long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
```

```
...
leaq    (%rdi,%rsi), %rax
addq    %rdx, %rax
leaq    (%rsi,%rsi,2), %rdx
salq    $4, %rdx
leaq    4(%rdi,%rdx), %rcx
imulq   %rcx, %rax
ret
```

%rax = t1 = x+y

%rax = t2 = z+t1

%rdx = y*3

Example: Arithmetic

```
long arith(long x, long y, long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
```

```
...
leaq    (%rdi,%rsi), %rax
addq    %rdx, %rax
leaq    (%rsi,%rsi,2), %rdx
salq    $4, %rdx
leaq    4(%rdi,%rdx), %rcx
imulq   %rcx, %rax
ret
```

%rax = t1 = x+y

%rax = t2 = z+t1

%rdx = y*3

%rdx = y*48

Example: Arithmetic

```
long arith(long x, long y, long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
```

```
...
leaq    (%rdi,%rsi), %rax       %rax = t1 = x+y
addq    %rdx, %rax              %rax = t2 = z+t1
leaq    (%rsi,%rsi,2), %rdx     %rdx = y*3
salq    $4, %rdx                %rdx = y*48
leaq    4(%rdi,%rdx), %rcx      %rcx = t5 = x+4+y*48
imulq   %rcx, %rax
ret
```

Example: Arithmetic

```
long arith(long x, long y, long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
```

```
...
leaq    (%rdi,%rsi), %rax       %rax = t1 = x+y
addq    %rdx, %rax              %rax = t2 = z+t1
leaq    (%rsi,%rsi,2), %rdx     %rdx = y*3
salq    $4, %rdx                %rdx = y*48
leaq    4(%rdi,%rdx), %rcx      %rcx = t5 = x+4+y*48
imulq   %rcx, %rax              %rax = t2*t5
ret
```