CSC 252/452: Computer Organization
Fall 2025: Lecture 6

Instructor: Yanan Guo

Department of Computer Science
University of Rochester

Announcement

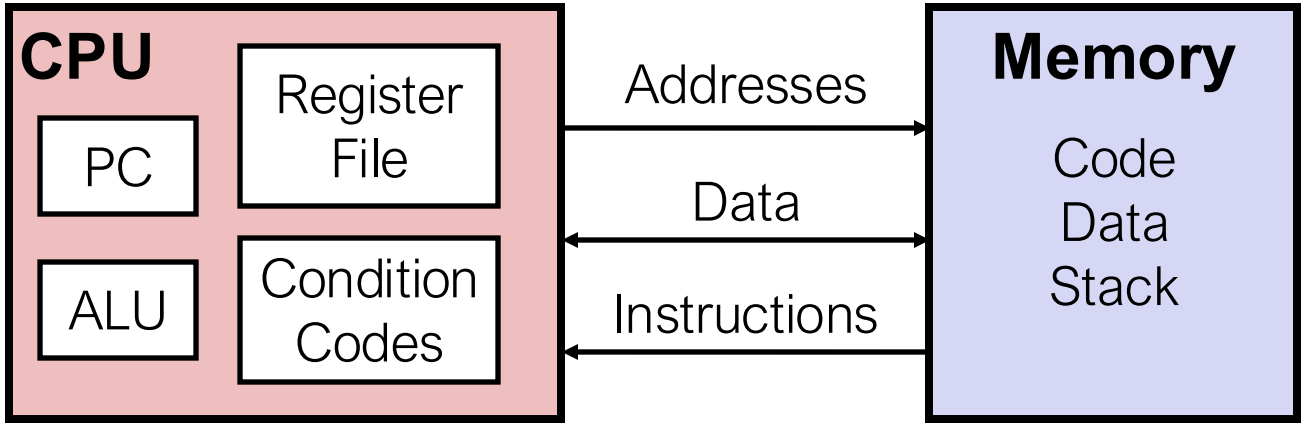
- Programming Assignment 1 is due today
  - NetID is not your 8-digit URID
  - For using slip days, email the two TAs listed in the assignment description
- Programming Assignment 2 is out today
  - Details: https://www.cs.rochester.edu/courses/252/fall2025/labs/assignment2.html
  - Due on **Oct 1st**, 11:59 PM
  - You (may still) have 3 slip days

Announcement

- Programming assignment 2 is in x86 assembly language.
- Read the instructions before getting started!!!
  - You get 1/4 point off for every wrong answer
  - Maxed out at 10
- TAs are best positioned to answer your questions about programming assignments!!!
- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.

Instruction Processing Sequence

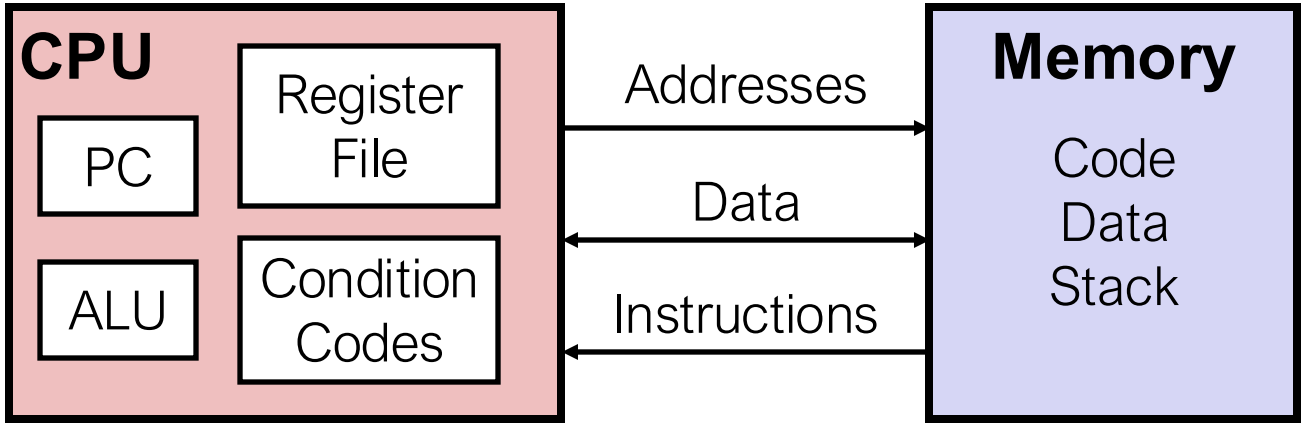
Assembly Programmer's Perspective of a Computer



Fetch Instruction (According to PC)

Instruction Processing Sequence

Assembly Programmer's Perspective of a Computer

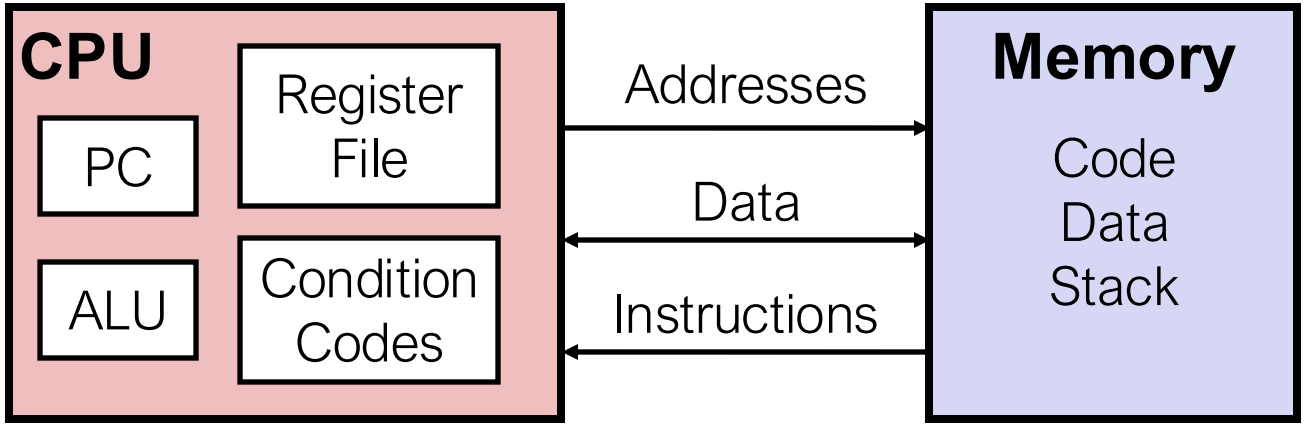


Fetch Instruction (According to PC)

`0x4801d8`

Instruction Processing Sequence

Assembly Programmer's Perspective of a Computer

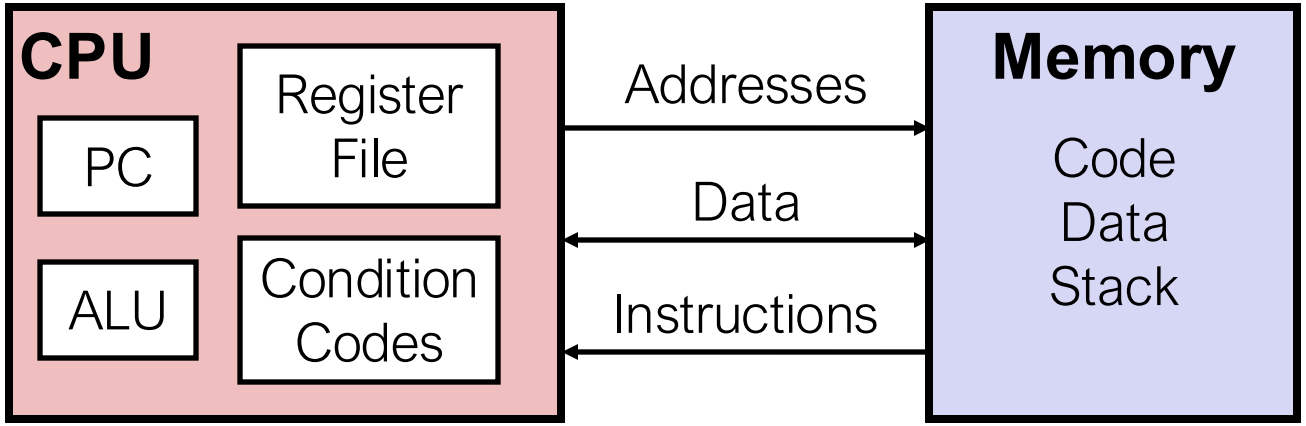


Fetch Instruction (According to PC) → Decode Instruction

```
addq %rax,(%rbx)
```

Instruction Processing Sequence

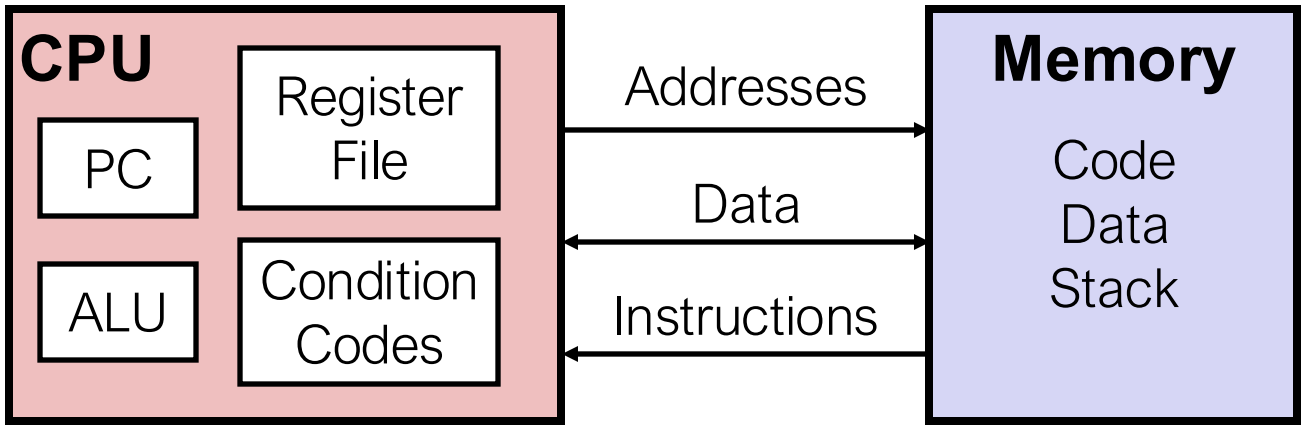
Assembly Programmer's Perspective of a Computer



Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands

Instruction Processing Sequence

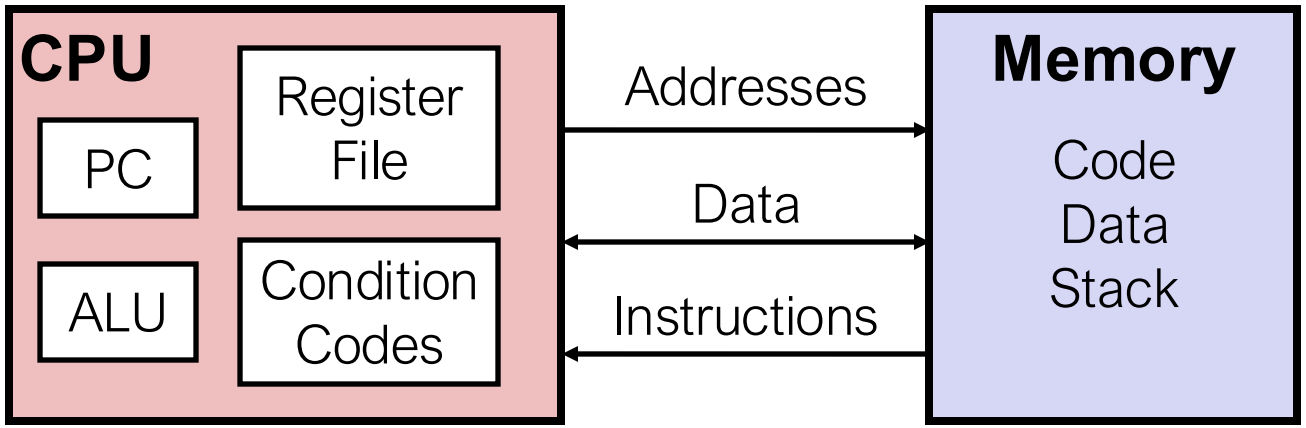
Assembly Programmer's Perspective of a Computer

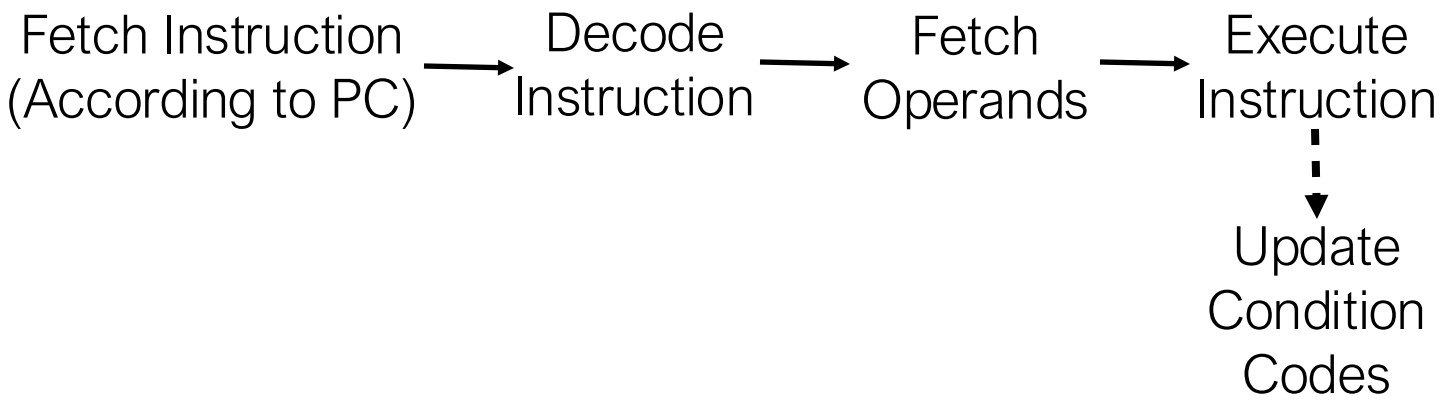


Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands → Execute Instruction

Instruction Processing Sequence

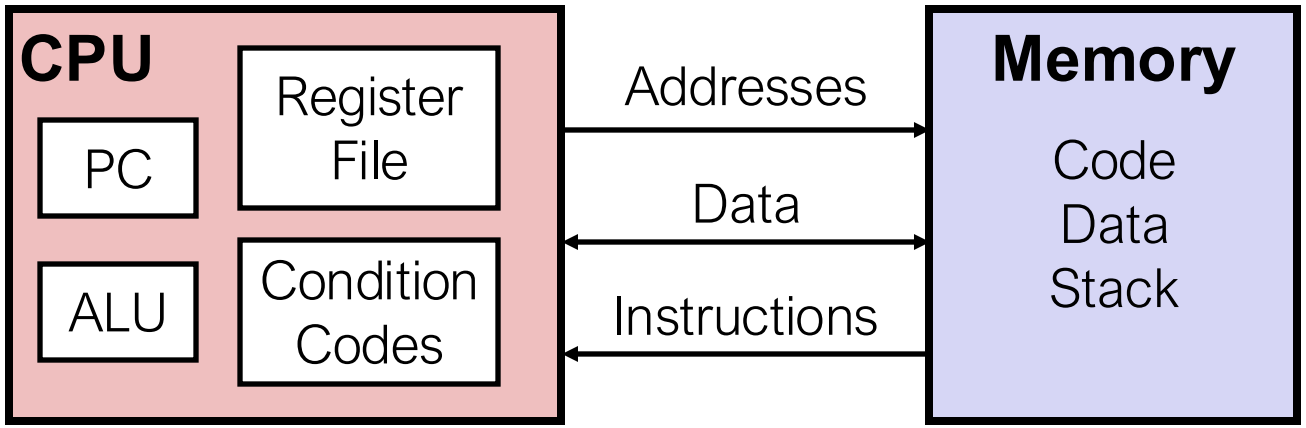
Assembly Programmer's Perspective of a Computer

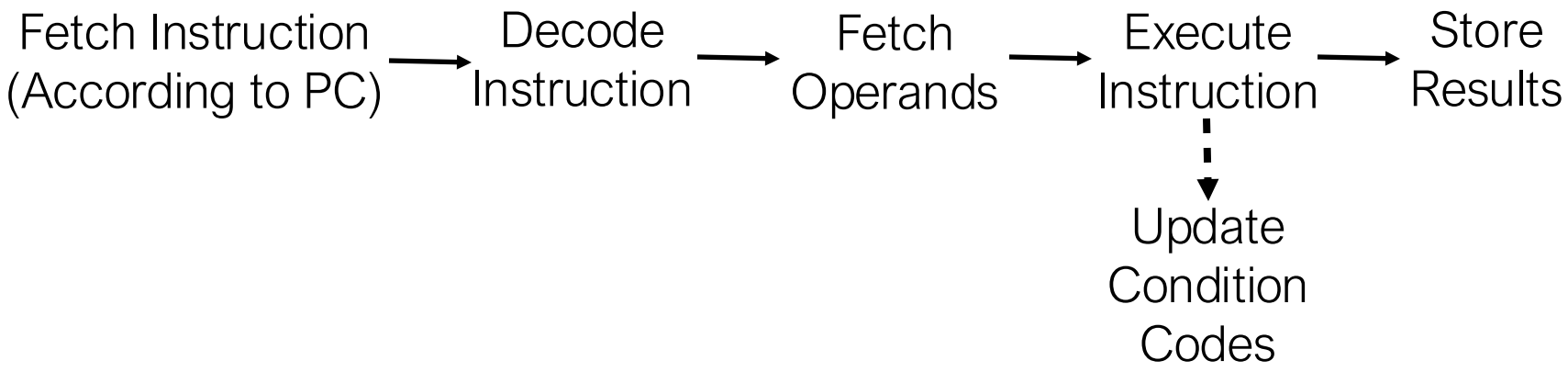




Instruction Processing Sequence

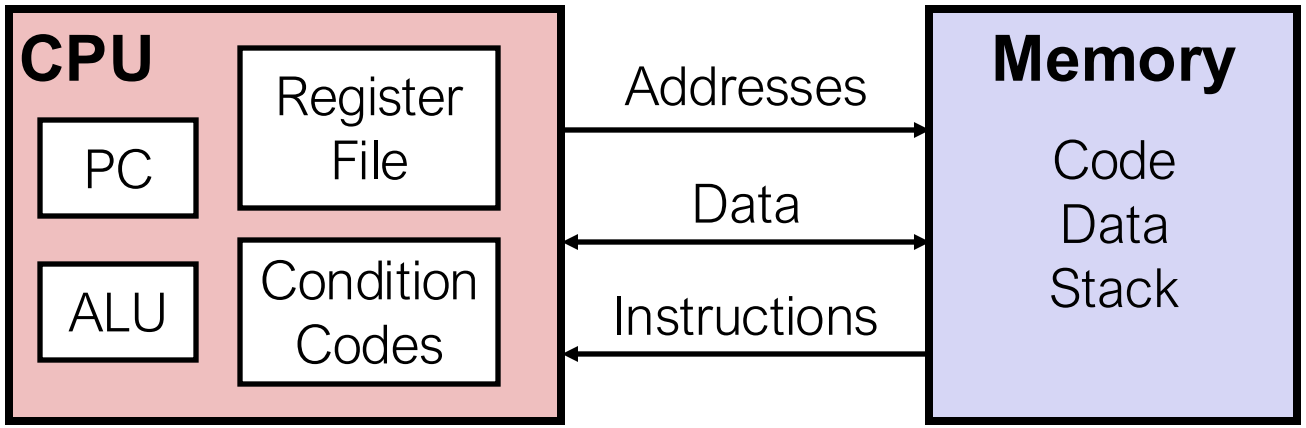
Assembly Programmer's Perspective of a Computer

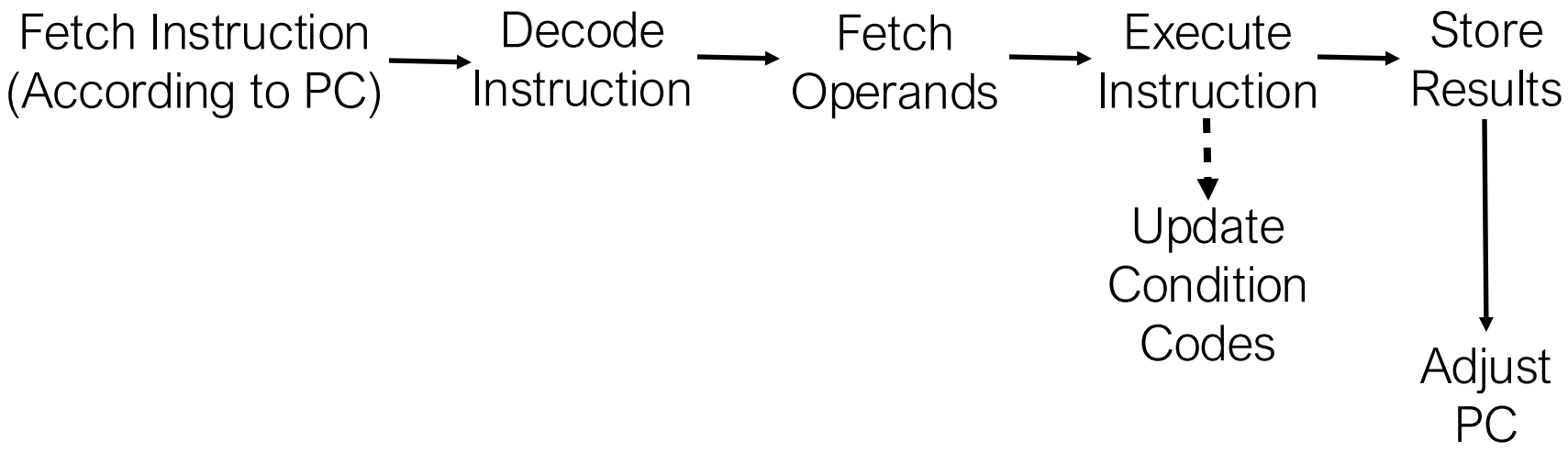




Instruction Processing Sequence

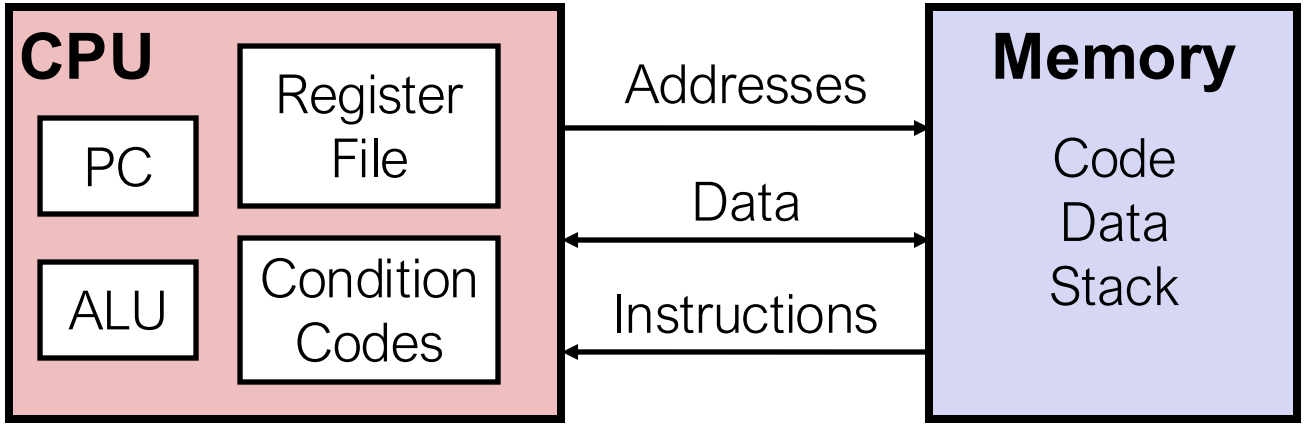
Assembly Programmer's Perspective of a Computer

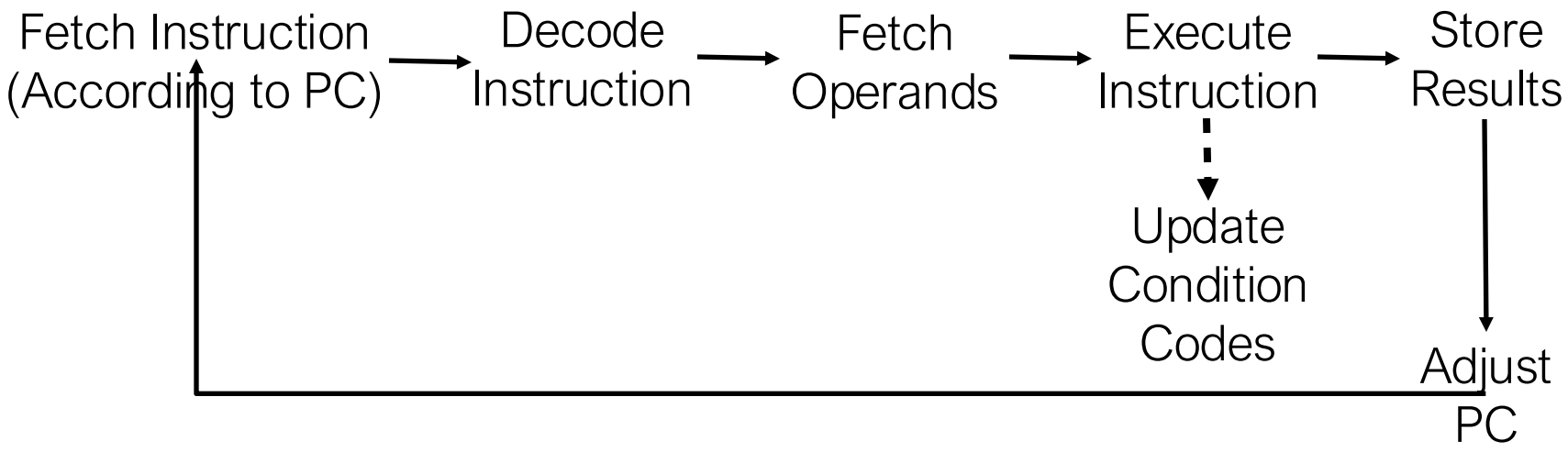




Instruction Processing Sequence

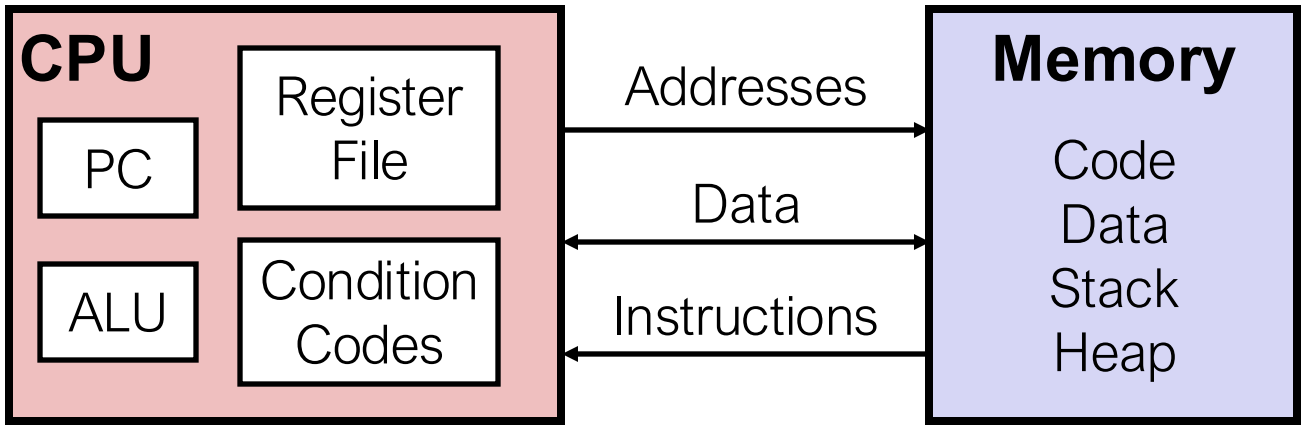
Assembly Programmer's Perspective of a Computer





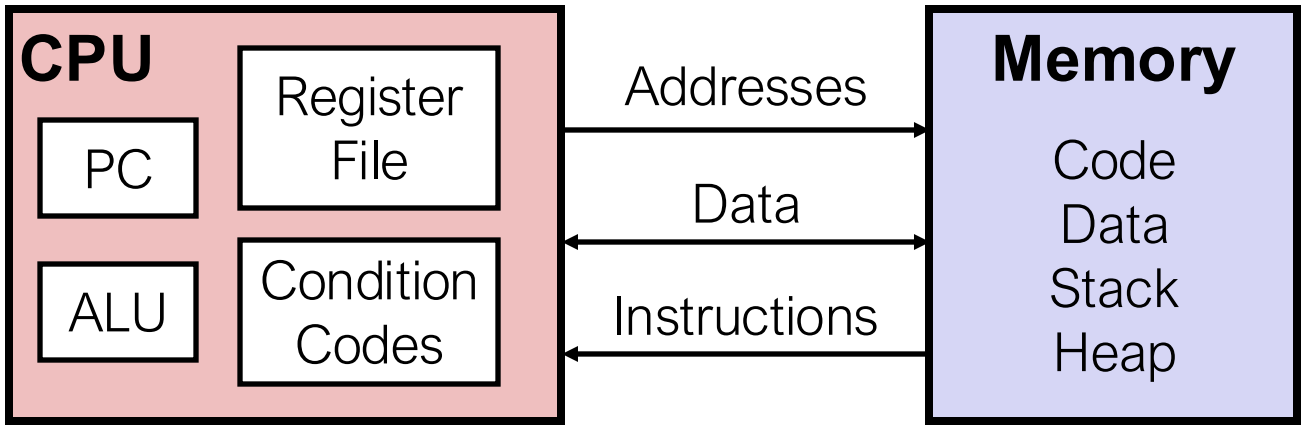
Assembly Program Instructions

Assembly Programmer's Perspective of a Computer



Assembly Program Instructions

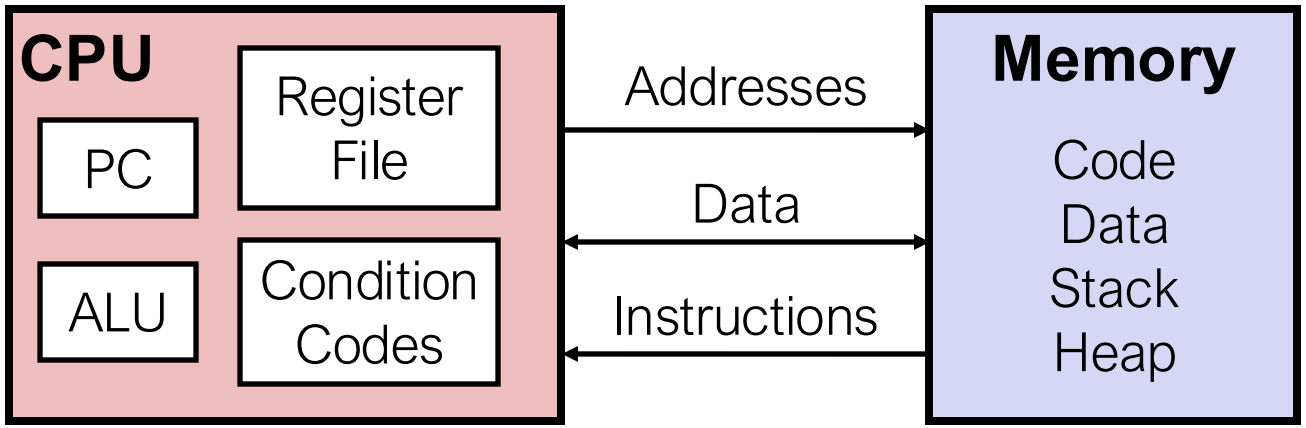
Assembly Programmer's Perspective of a Computer



- *Compute Instruction*: Perform arithmetics on register or memory data
  - **`addq %eax, %ebx`**
  - C constructs: +, -, >>, etc.

Assembly Program Instructions

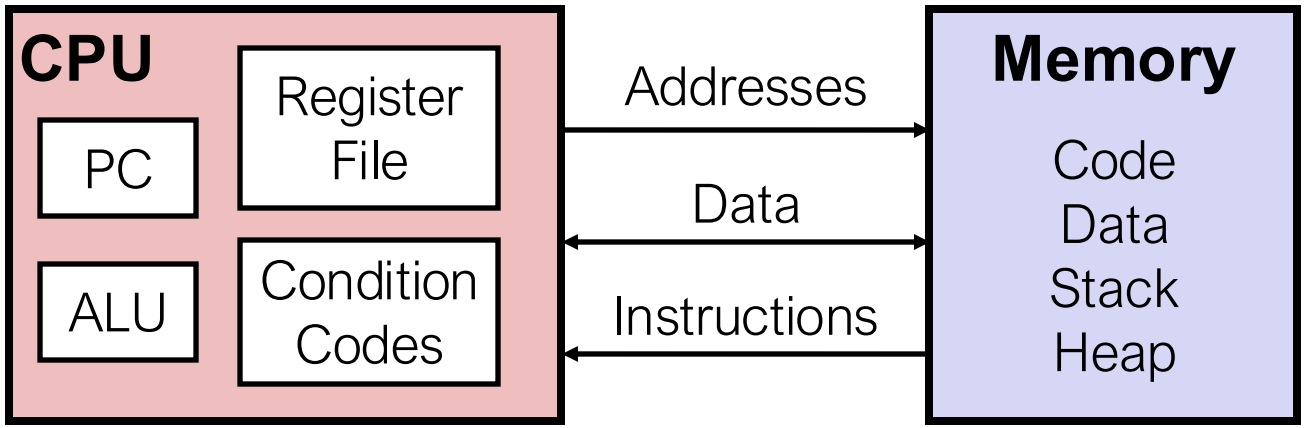
Assembly Programmer's Perspective of a Computer



- *Compute Instruction*: Perform arithmetics on register or memory data
  - **`addq %eax, %ebx`**
  - C constructs: +, -, >>, etc.
- *Data Movement Instruction*: Transfer data between memory and register
  - **`movq %eax, (%ebx)`**

Assembly Program Instructions

Assembly Programmer's Perspective of a Computer



- *Compute Instruction*: Perform arithmetics on register or memory data
  - **`addq %eax, %ebx`**
  - C constructs: +, -, >>, etc.
- *Data Movement Instruction*: Transfer data between memory and register
  - **`movq %eax, (%ebx)`**
- *Control Instruction*: Alter the sequence of instructions (by changing PC)
  - **`jmp`**, **`call`**
  - C constructs: **`if-else`**, **`do-while`**, function call, etc.

Today: Compute and Control Instructions

- Move operations (and addressing modes)
- Arithmetic & logical operations
- Condition Codes
- Control: Conditional branches (`if`… `else`…)
- Control: Loops (`for`, `while`)
- Control: Switch Statements (`case`… `switch`…)

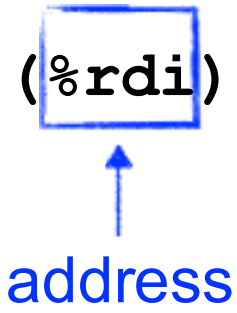
Data Movement Instruction Example

```
movq    %rdx, (%rdi)
```

- Semantics:
  - Move (really, **copy**) data in register **`%rdx`** to memory location whose address is the value stored in **`%rdi`**
  - Pointer dereferencing

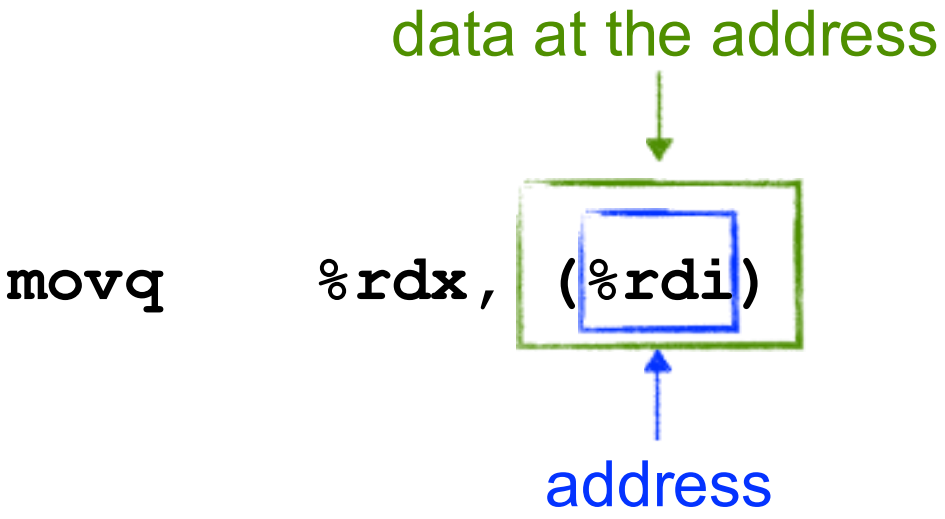
Data Movement Instruction Example

```
movq     %rdx, (%rdi)
```



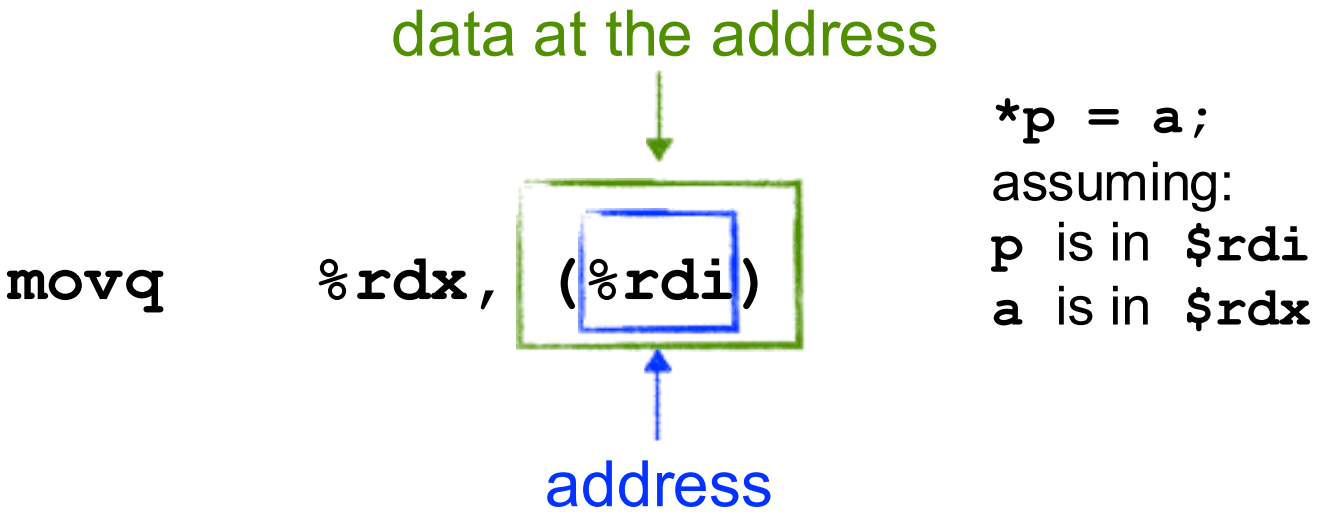
- Semantics:
  - Move (really, **copy**) data in register **`%rdx`** to memory location whose address is the value stored in **`%rdi`**
  - Pointer dereferencing

Data Movement Instruction Example



- Semantics:
  - Move (really, **copy**) data in register **`%rdx`** to memory location whose address is the value stored in **`%rdi`**
  - Pointer dereferencing

Data Movement Instruction Example



- Semantics:
  - Move (really, **copy**) data in register **`%rdx`** to memory location whose address is the value stored in **`%rdi`**
  - Pointer dereferencing

Data Movement Instructions

`movq` *Source*, *Dest*

Data Movement Instructions

`movq` *Source*, *Dest*

Operator Operands

Data Movement Instructions

`movq` *Source*, *Dest*

Operator Operands

- Memory:
  - Simplest example: **`(%rax)`**
  - How to obtain the address is called “addressing mode”

Data Movement Instructions

`movq` *Source*, *Dest*

Operator Operands

- Memory:
  - Simplest example: **`(%rax)`**
  - How to obtain the address is called “addressing mode”
- Register:
  - Example: **`%rax, %r13`**
  - But **`%rsp`** reserved for special use

Data Movement Instructions

`movq` *Source*, *Dest*

Operator Operands

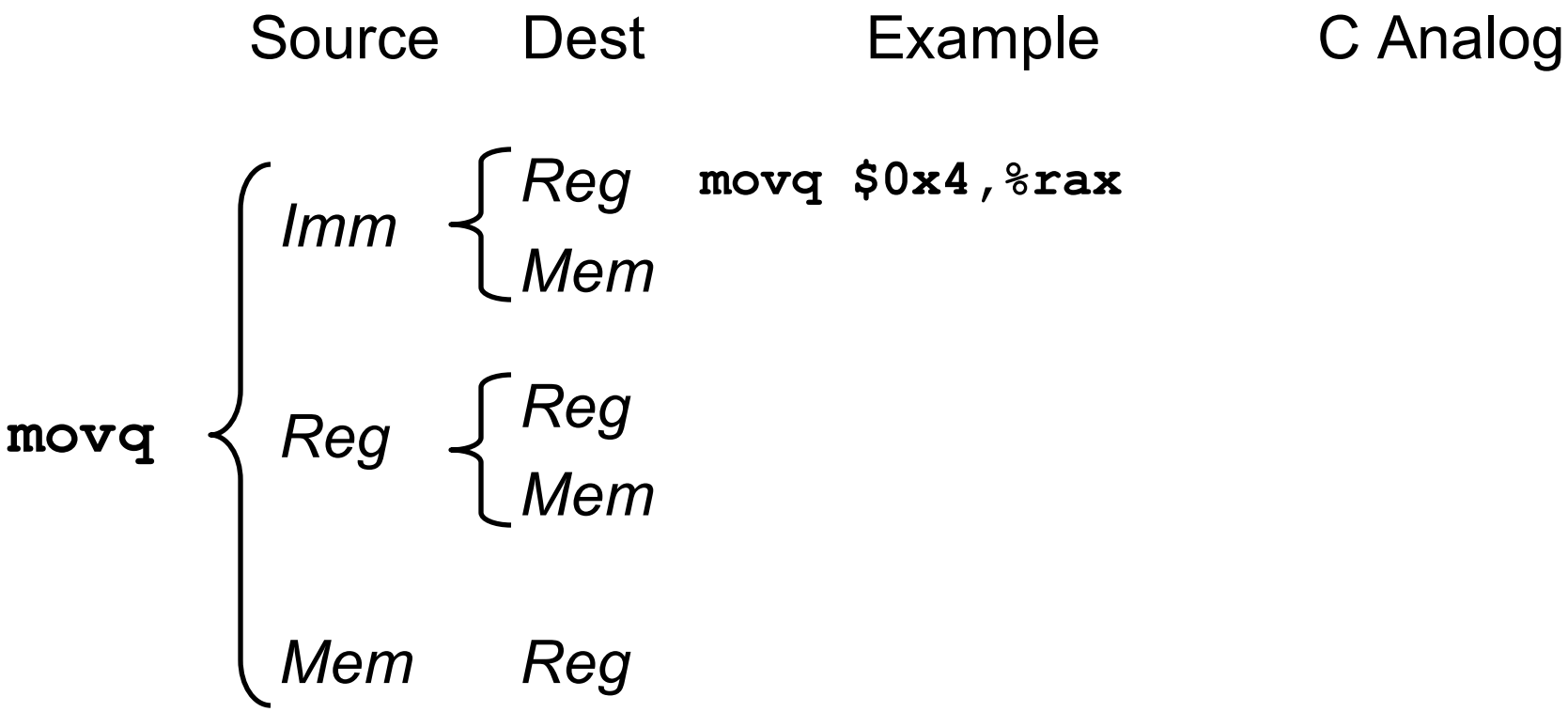
- Memory:
  - Simplest example: **`(%rax)`**
  - How to obtain the address is called "addressing mode"
- Register:
  - Example: **`%rax, %r13`**
  - But **`%rsp`** reserved for special use
- Immediate: Constant integer data
  - Example: **`$0x400`**, **`$-533;`** like C constant, but prefixed with '**`$`**'
  - Encoded with 1, 2, or 4 bytes; can only be source

`movq` Operand Combinations

	Source	Dest	Example	C Analog
movq	*Imm*	*Reg*		
		Mem		
	Reg	*Reg*		
		Mem		
	Mem	*Reg*		

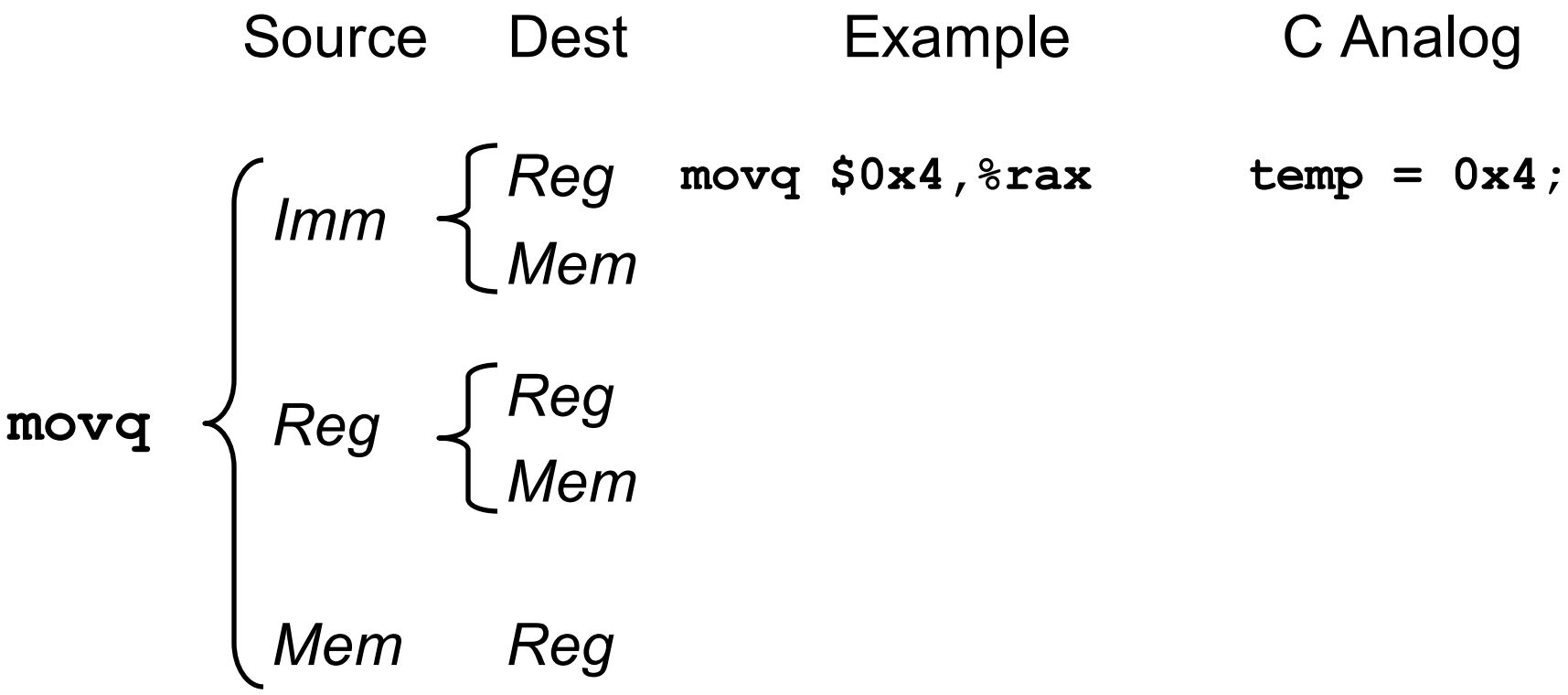
Cannot do memory-memory transfer with a single instruction in x86.

`movq` Operand Combinations



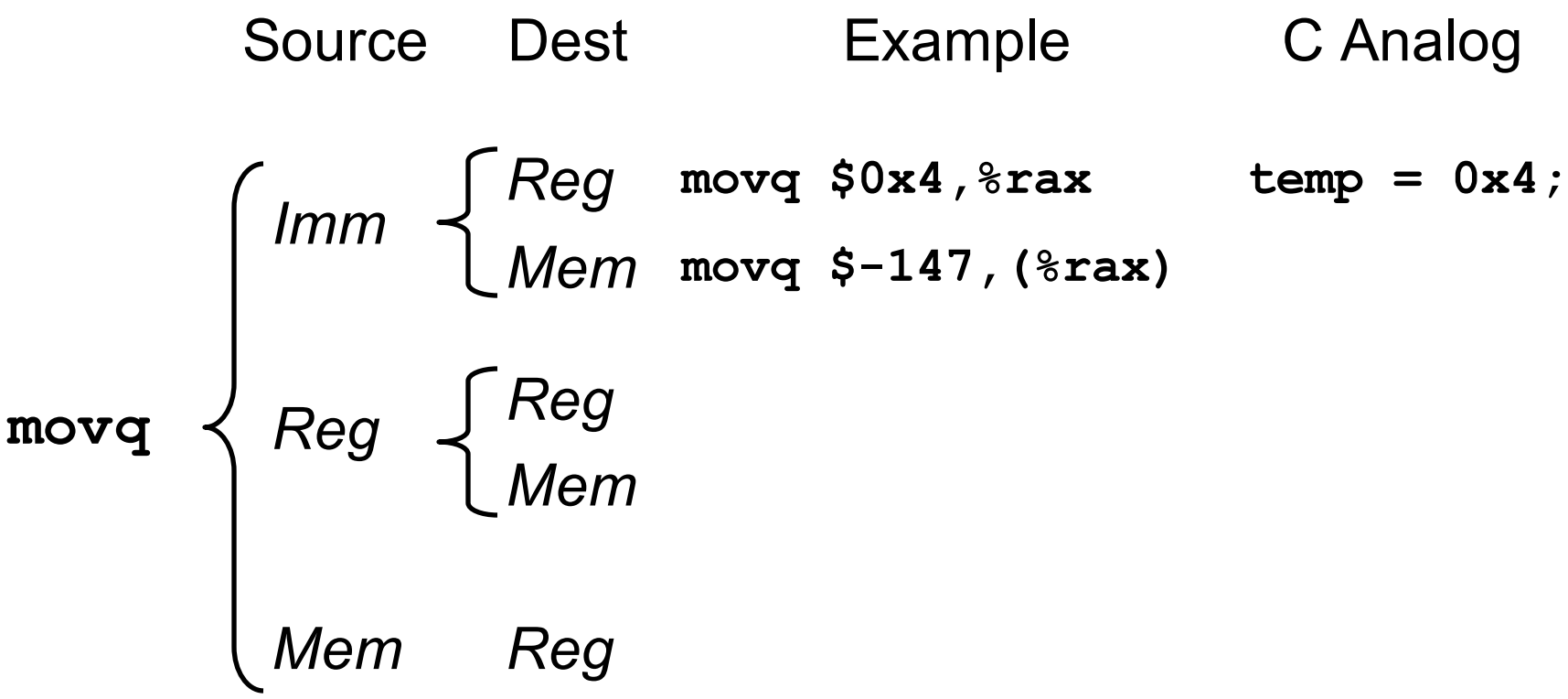
Cannot do memory-memory transfer with a single instruction in x86.

`movq` Operand Combinations



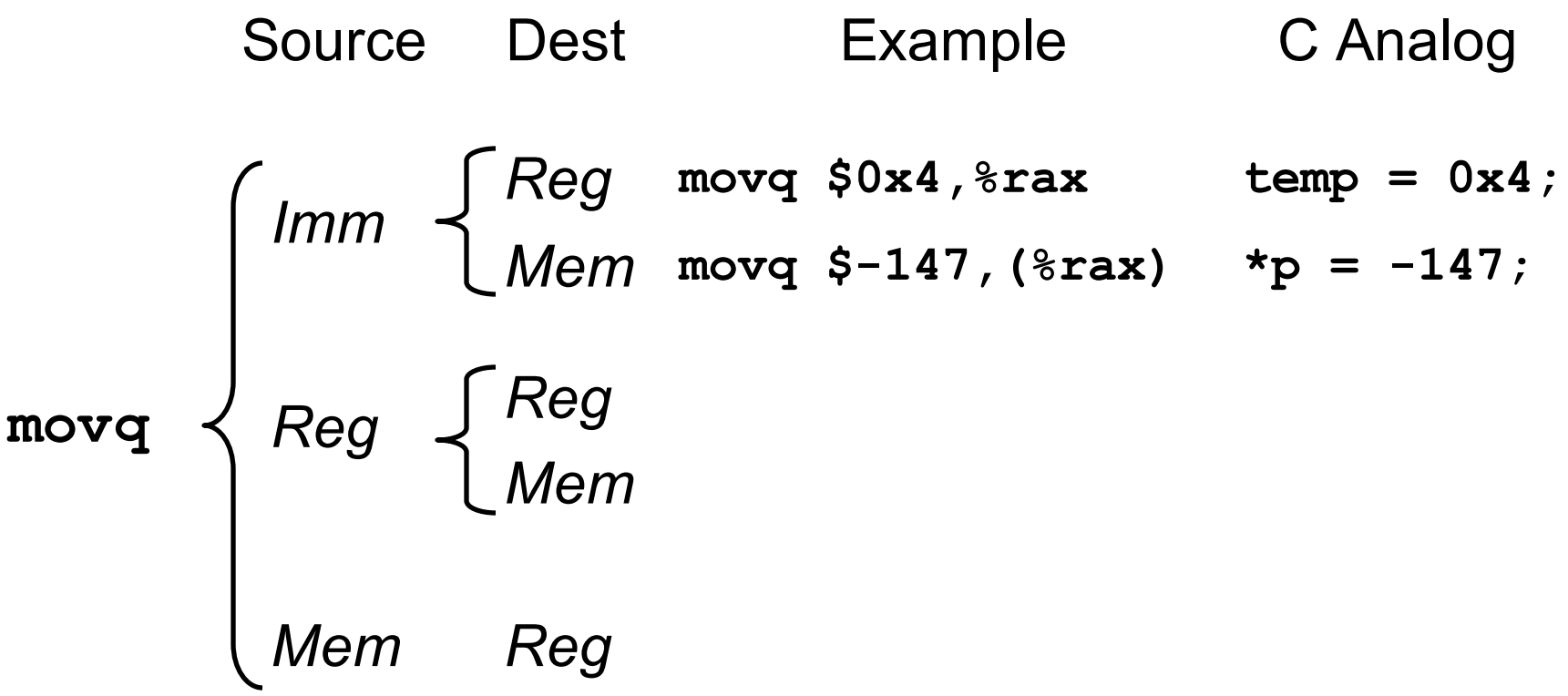
Cannot do memory-memory transfer with a single instruction in x86.

`movq` Operand Combinations



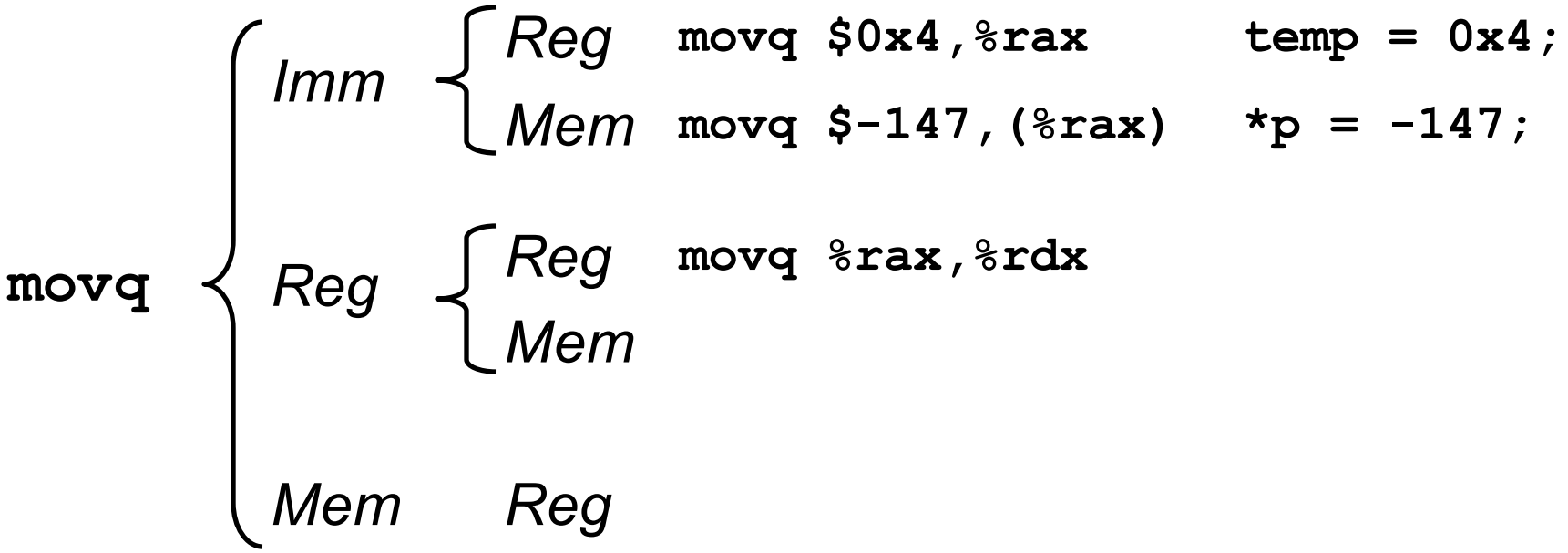
Cannot do memory-memory transfer with a single instruction in x86.

`movq` Operand Combinations



Cannot do memory-memory transfer with a single instruction in x86.

`movq` Operand Combinations



Cannot do memory-memory transfer with a single instruction in x86.

`movq` Operand Combinations

	Source	Dest	Example	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem		
	Mem	*Reg*		

Cannot do memory-memory transfer with a single instruction in x86.

`movq` Operand Combinations

	Source	Dest	Example	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	
	Mem	*Reg*		

Cannot do memory-memory transfer with a single instruction in x86.

`movq` Operand Combinations

	Source	Dest	Example	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*		

Cannot do memory-memory transfer with a single instruction in x86.

`movq` Operand Combinations

	Source	Dest	Example	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	

Cannot do memory-memory transfer with a single instruction in x86.

`movq` Operand Combinations

	Source	Dest	Example	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

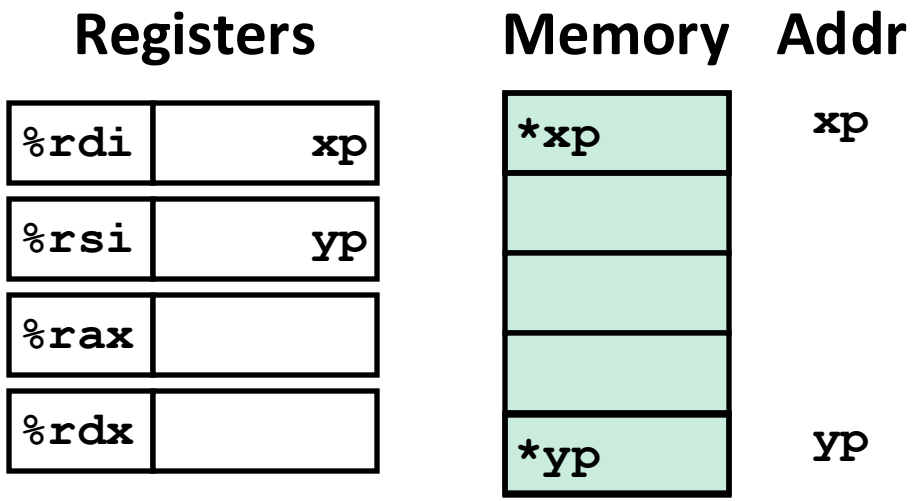
Cannot do memory-memory transfer with a single instruction in x86.

Example of Simple Addressing Modes

```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

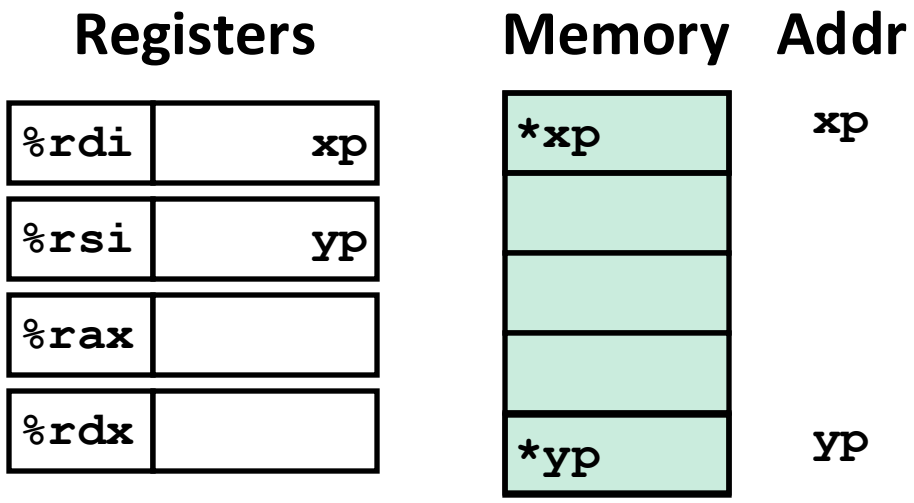
Example of Simple Addressing Modes

```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



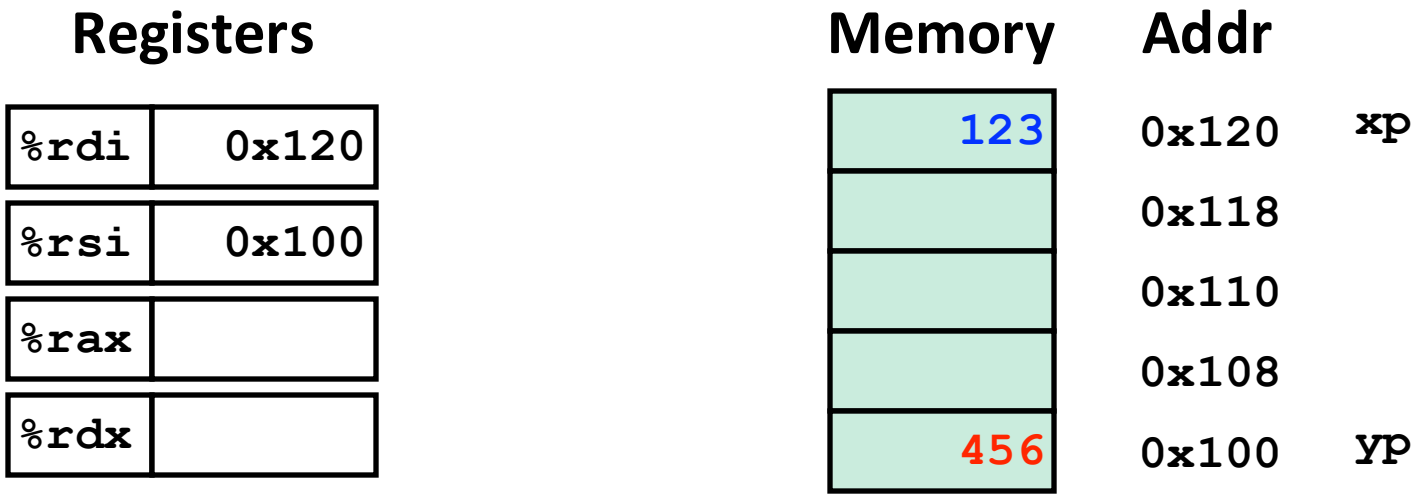
Example of Simple Addressing Modes

```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



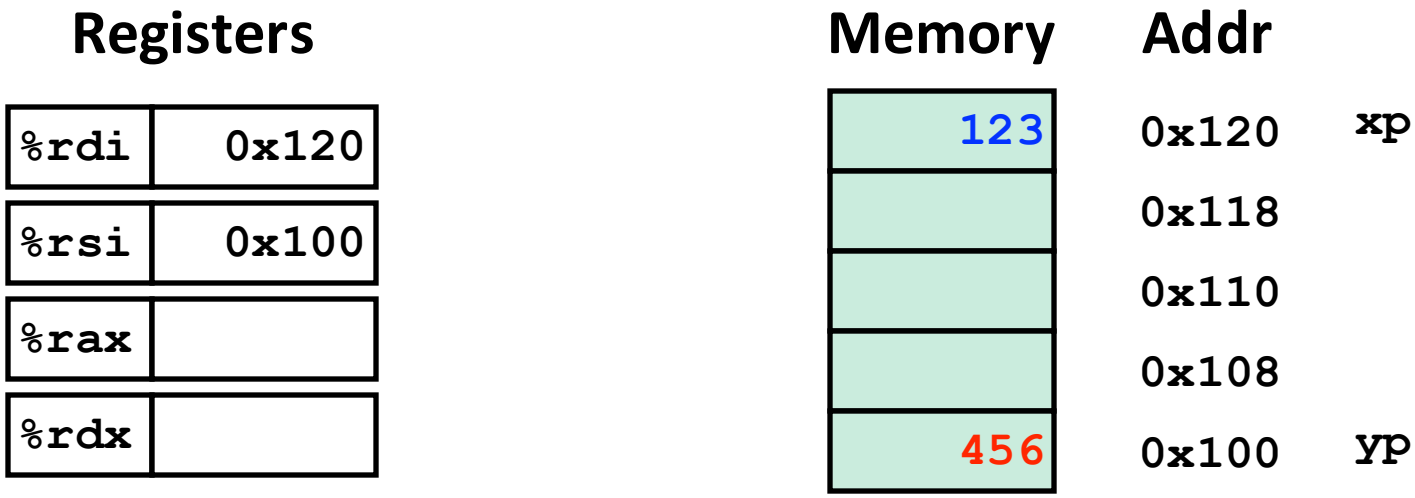

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`



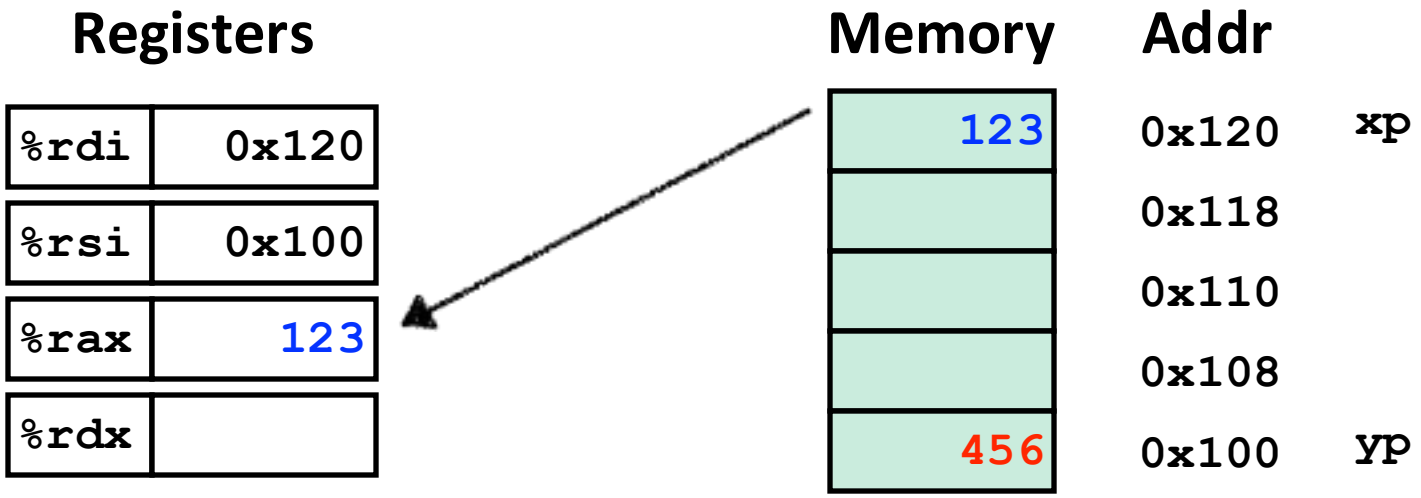

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`



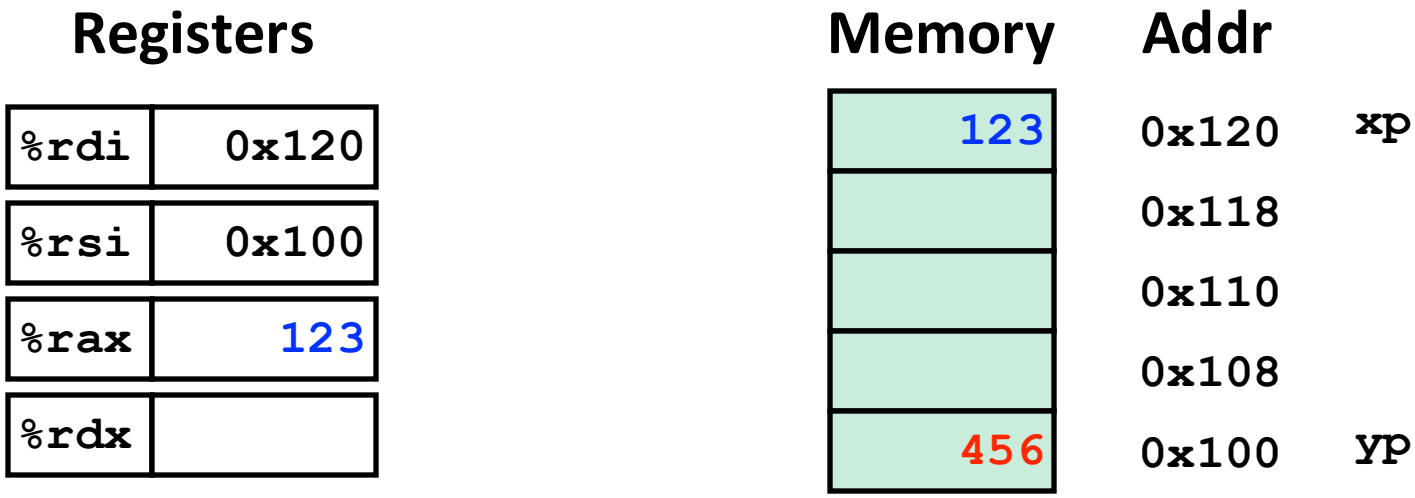

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`



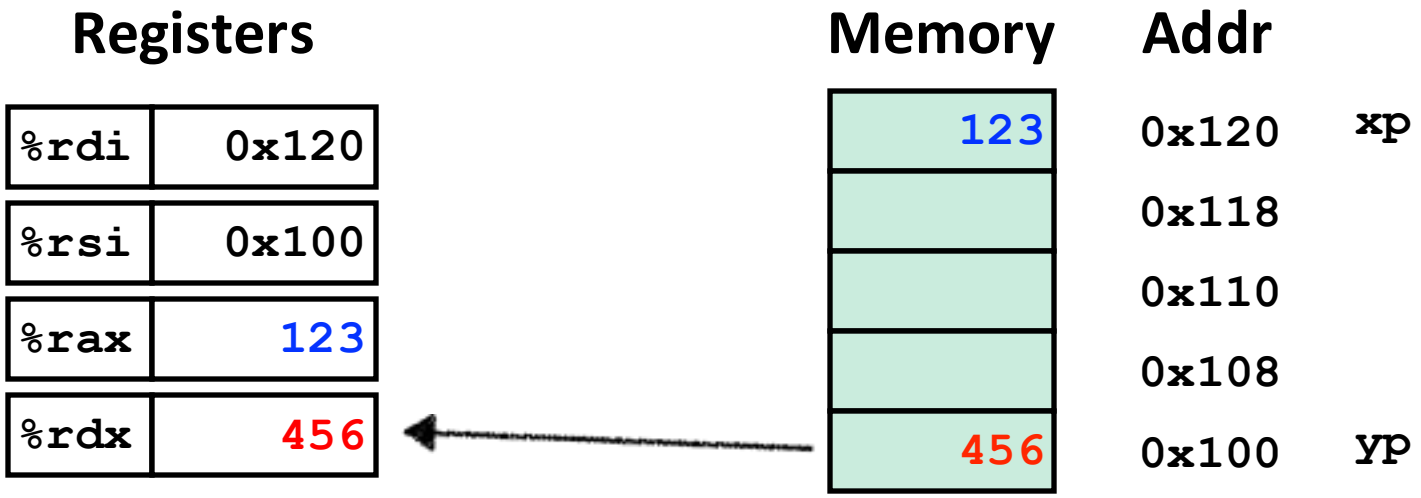

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`



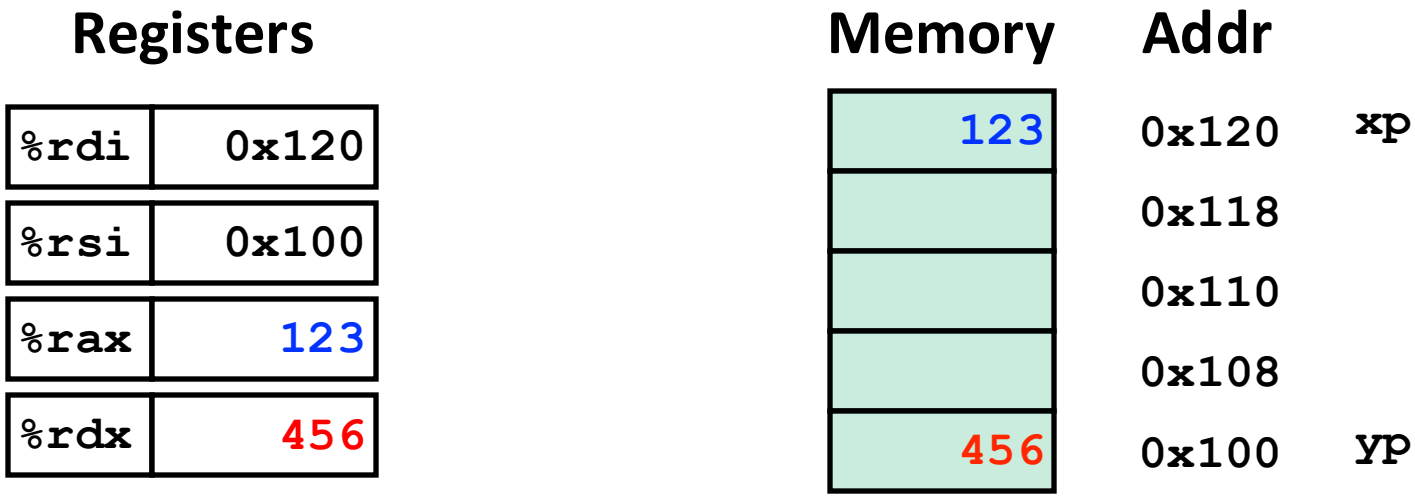

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`



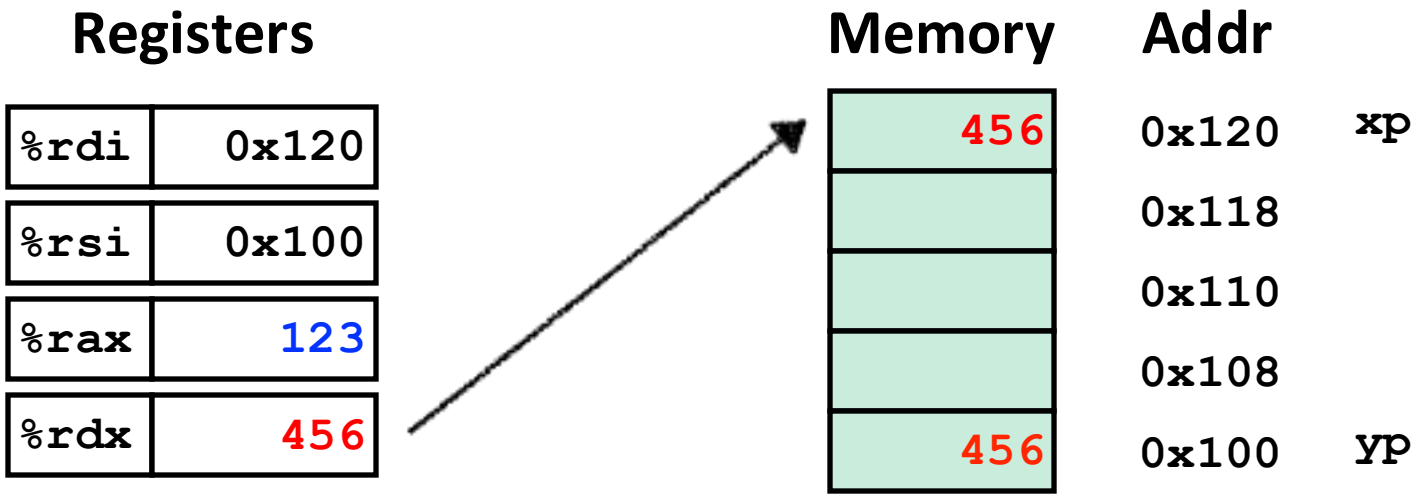

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`



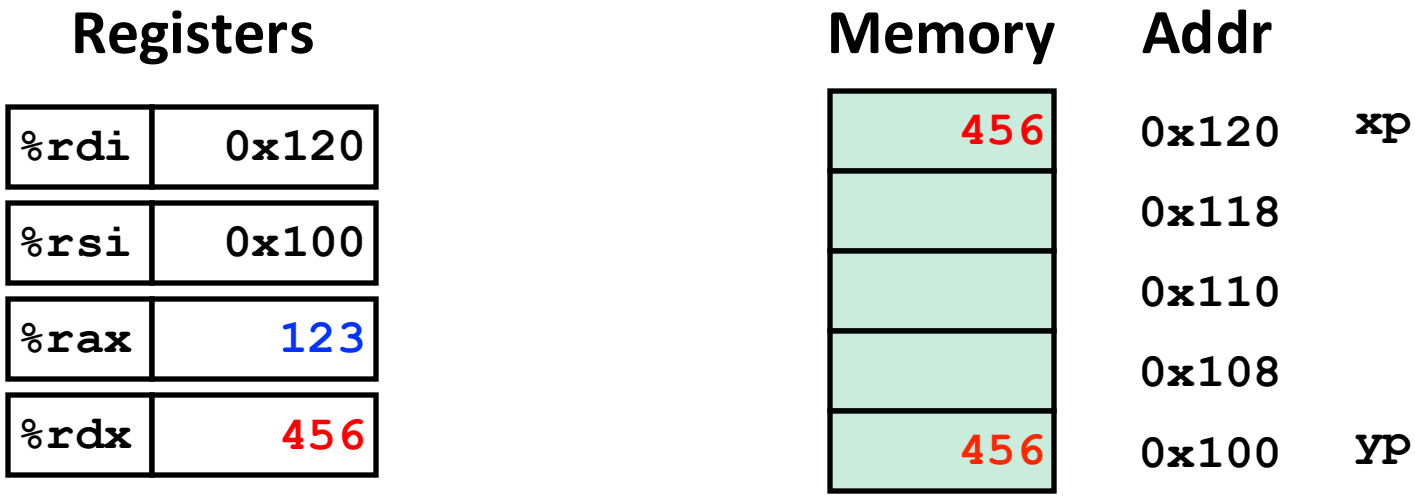

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`



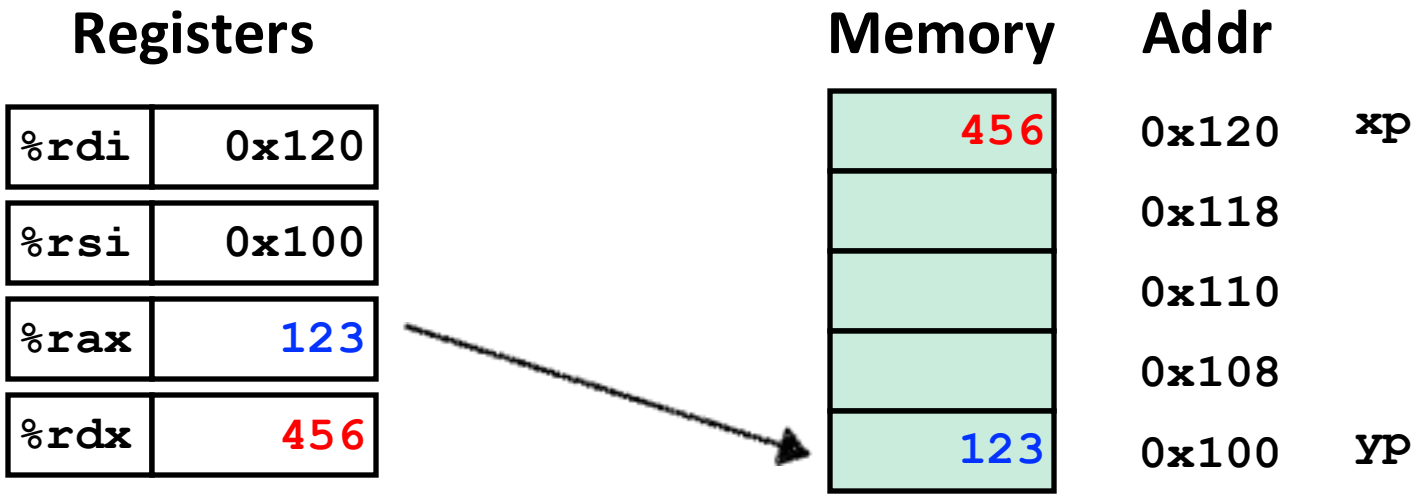

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Memory Addressing Modes

- An addressing mode specifies:
  - how to calculate the effective memory address of an operand
  - by using information held in registers and/or constants

Memory Addressing Modes

- An addressing mode specifies:
  - how to calculate the effective memory address of an operand
  - by using information held in registers and/or constants
- Normal: (R)
  - Memory address: content of Register R (**Reg[R]**)
  - Pointer dereferencing in C

```
movq (%rcx),%rax; // address = %rcx
```

Memory Addressing Modes

- An addressing mode specifies:
  - how to calculate the effective memory address of an operand
  - by using information held in registers and/or constants
- Normal: (R)
  - Memory address: content of Register R (**Reg[R]**)
  - Pointer dereferencing in C

```
movq (%rcx),%rax; // address = %rcx
```

- Displacement: D(R)
  - Memory address: **Reg[R]+D**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx; // address = %rbp + 8
```

Complete Memory Addressing Modes

- The General Form: D(Rb,Ri,S)
  - Memory address: Reg[Rb] + S * Reg[Ri] + D
  - E.g., **`8(%eax, %ebx, 4)`**; // address = **`%eax`** + 4 * **`%ebx`** + 8
  - D: Constant "displacement"
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for %rsp
  - S: Scale: 1, 2, 4, or 8

Complete Memory Addressing Modes

- The General Form: D(Rb,Ri,S)
  - Memory address: Reg[Rb] + S * Reg[Ri] + D
  - E.g., **`8(%eax, %ebx, 4)`**; // address = **`%eax`** + 4 * **`%ebx`** + 8
  - D: Constant "displacement"
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for %rsp
  - S: Scale: 1, 2, 4, or 8
- What is **`8(%eax, %ebx, 4)`** used for?

Complete Memory Addressing Modes

- The General Form: D(Rb,Ri,S)
  - Memory address: Reg[Rb] + S * Reg[Ri] + D
  - E.g., **`8(%eax, %ebx, 4)`**; // address = **`%eax`** + 4 * **`%ebx`** + 8
  - D: Constant "displacement"
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for %rsp
  - S: Scale: 1, 2, 4, or 8
- What is **`8(%eax, %ebx, 4)`** used for?
- Special Cases

(Rb,Ri)	address = Reg[Rb]+Reg[Ri]
D(Rb,Ri)	address = Reg[Rb]+Reg[Ri]+D
(Rb,Ri,S)	address = Reg[Rb]+S*Reg[Ri]

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

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

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

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

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

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

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`		

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

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`

Address Computation Instruction

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

Address Computation Instruction

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

%rax = %rsi + %rdi * 2 + 4

Address Computation Instruction

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

%rax = %rsi + %rdi * 2 + 4

- **`leaq`** Src, Dst
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression
  - No actual memory reference is made

Address Computation Instruction

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

%rax = %rsi + %rdi * 2 + 4

- **`leaq`** Src, Dst
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression
  - No actual memory reference is made
- Uses
  - Computing addresses without a memory reference
    - E.g., translation of p = &x[i];

Address Computation Instruction

- Interesting Use
  - Computing arithmetic expressions of the form x + k*y
  - Faster arithmetic computation

Address Computation Instruction

- Interesting Use
  - Computing arithmetic expressions of the form x + k*y
  - Faster arithmetic computation

```
long m12(long x)
{
  return x*12;
}
```

Address Computation Instruction

- Interesting Use
  - Computing arithmetic expressions of the form x + k*y
  - Faster arithmetic computation

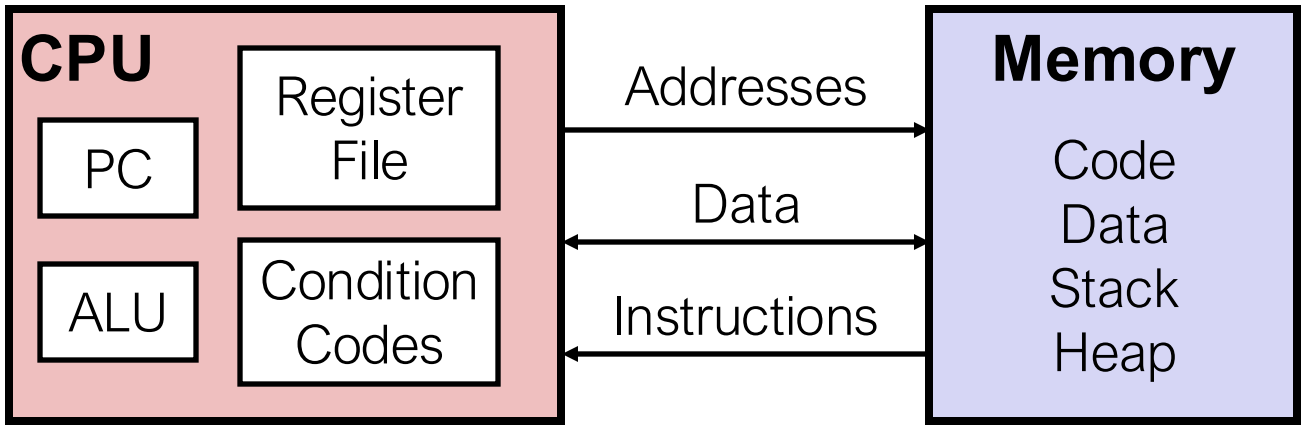
```
long m12(long x)
{
  return x*12;
}
```

Converted to
assembly by compiler:

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

Assembly Program Instructions

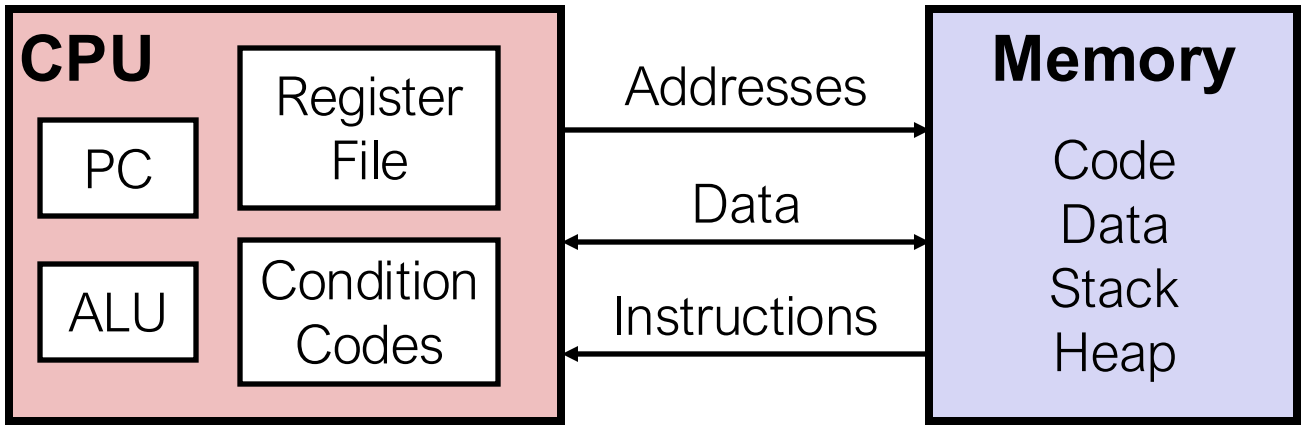
Assembly Programmer's Perspective of a Computer



- *Data Movement Instruction*: Transfer data between memory and register
  - **`movq %eax, (%ebx)`**

Assembly Program Instructions

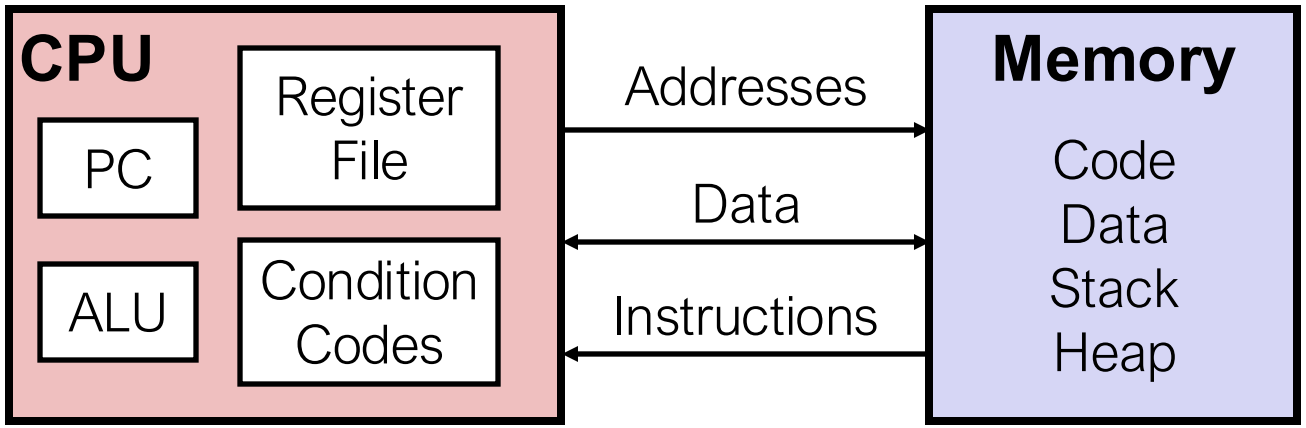
Assembly Programmer's Perspective of a Computer



- *Data Movement Instruction*: Transfer data between memory and register
  - **`movq %eax, (%ebx)`**
- *Compute Instruction*: Perform arithmetics on register or memory data
  - **`addq %eax, %ebx`**
  - C constructs: +, -, >>, etc.

Assembly Program Instructions

Assembly Programmer's Perspective of a Computer



- *Data Movement Instruction*: Transfer data between memory and register
  - **`movq %eax, (%ebx)`**
- *Compute Instruction*: Perform arithmetics on register or memory data
  - **`addq %eax, %ebx`**
  - C constructs: +, -, >>, etc.
- *Control Instruction*: Alter the sequence of instructions (by changing PC)
  - **`jmp`**, **`call`**
  - C constructs: **`if-else`**, **`do-while`**, function call, etc.

Today: Compute and Control Instructions

- Move operations (and addressing modes)
- Arithmetic & logical operations
- Control: Condition branches (**`if`**… **`else`**…)
- Control: Loops (**`for`**, **`while`**)
- Control: Switch Statements (**`case`**… **`switch`**…)

Some Arithmetic Operations (2 Operands)

Format	Computation	Notes
`addq` `src, dest`	Dest = Dest + Src	

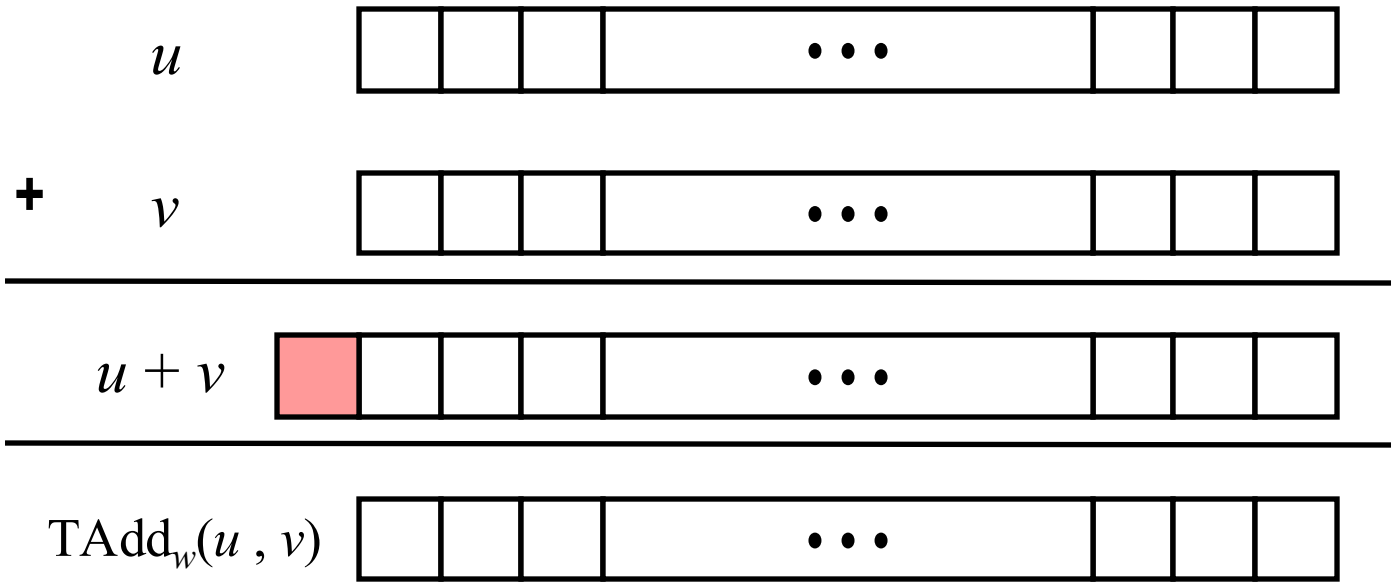
Some Arithmetic Operations (2 Operands)

Format	Computation	Notes
addq src, dest	Dest = Dest + Src	

```
addq %rax, %rbx
```

Some Arithmetic Operations (2 Operands)

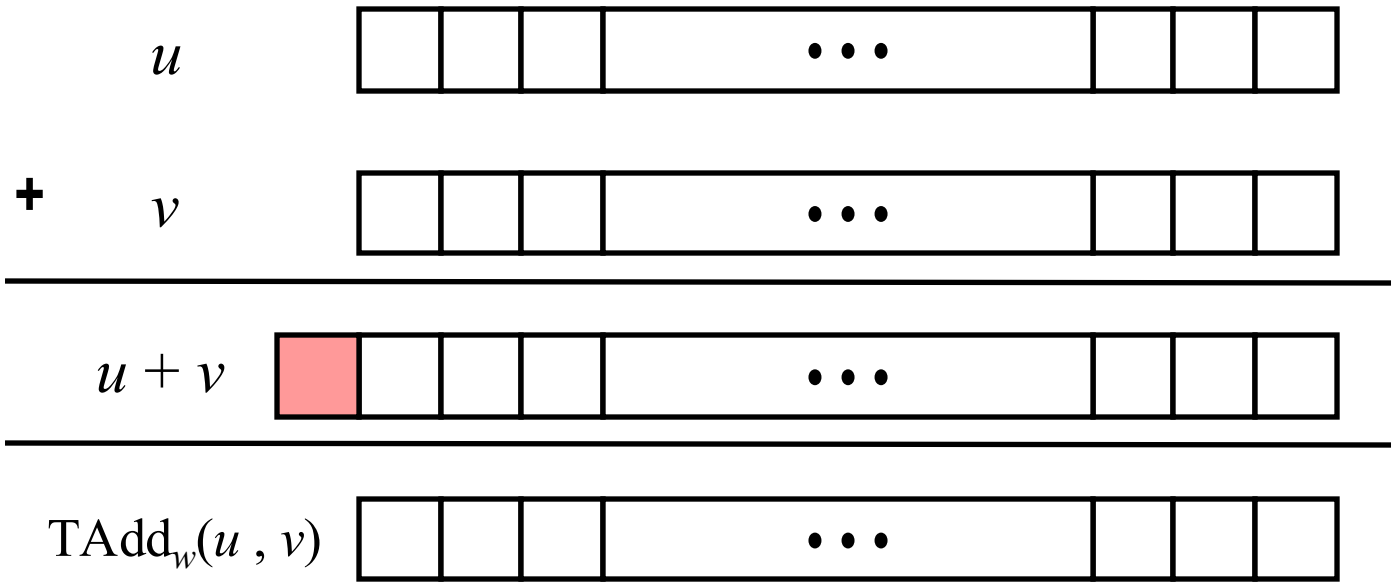
Format	Computation	Notes
addq src, dest	Dest = Dest + Src	



addq %rax, %rbx

Some Arithmetic Operations (2 Operands)

Format	Computation	Notes
addq src, dest	Dest = Dest + Src	



addq %rax, %rbx

%rbx = %rax + %rbx

Truncation if overflow,
set carry bit (more later…)

Some Arithmetic Operations (2 Operands)

Format	Computation	Notes
addq src, dest	Dest = Dest + Src	
subq src, dest	Dest = Dest - Src	
imulq src, dest	Dest = Dest * Src	
salq src, dest	Dest = Dest << Src	Also called **shlq**
sarq src, dest	Dest = Dest >> Src	Arithmetic shift
shrq src, dest	Dest = Dest >> Src	Logical shift
xorq src, dest	Dest = Dest ^ Src	
andq src, dest	Dest = Dest & Src	
orq src, dest	Dest = Dest | Src	

Some Arithmetic Operations (2 Operands)

- No distinction between signed and unsigned (why?)
  - Bit level behaviors for signed and unsigned arithmetic are exactly the same — assuming truncation

Some Arithmetic Operations (2 Operands)

- No distinction between signed and unsigned (why?)
  - Bit level behaviors for signed and unsigned arithmetic are exactly the same — assuming truncation

```
long signed_add
(long x, long y)
{
  long res = x + y;
  return res;
}
```

```
 #x in %rdx, y in %rax
 addq    %rdx, %rax
```

Some Arithmetic Operations (2 Operands)

- No distinction between signed and unsigned (why?)
  - Bit level behaviors for signed and unsigned arithmetic are exactly the same — assuming truncation

```
long signed_add
(long x, long y)
{
  long res = x + y;
  return res;
}
```

```
#x in %rdx, y in %rax
addq    %rdx, %rax
```

```
long unsigned_add
(unsigned long x, unsigned long y)
{
  unsigned long res = x + y;
  return res;
}
```

```
#x in %rdx, y in %rax
addq    %rdx, %rax
```

Some Arithmetic Operations (2 Operands)

- No distinction between signed and unsigned (why?)
  - Bit level behaviors for signed and unsigned arithmetic are exactly the same — assuming truncation

Bit-level

```
   010
+) 101
-------
   111
```

```
long signed_add
(long x, long y)
{
  long res = x + y;
  return res;
}
```

```
#x in %rdx, y in %rax
addq    %rdx, %rax
```

```
long unsigned_add
(unsigned long x, unsigned long y)
{
  unsigned long res = x + y;
  return res;
}
```

```
#x in %rdx, y in %rax
addq    %rdx, %rax
```

Some Arithmetic Operations (2 Operands)

- No distinction between signed and unsigned (why?)
  - Bit level behaviors for signed and unsigned arithmetic are exactly the same — assuming truncation

Bit-level	**Signed**
010	2
+) 101	+) -3
111	-1

```
long signed_add
(long x, long y)
{
  long res = x + y;
  return res;
}
```

```
#x in %rdx, y in %rax
addq    %rdx, %rax
```

```
long unsigned_add
(unsigned long x, unsigned long y)
{
  unsigned long res = x + y;
  return res;
}
```

```
#x in %rdx, y in %rax
addq    %rdx, %rax
```

Some Arithmetic Operations (2 Operands)

- No distinction between signed and unsigned (why?)
  - Bit level behaviors for signed and unsigned arithmetic are exactly the same — assuming truncation

Bit-level	Signed	Unsigned
010	2	2
+) 101	+) -3	+) 5
111	-1	7

```
long signed_add
(long x, long y)
{
  long res = x + y;
  return res;
}
```

```
#x in %rdx, y in %rax
addq    %rdx, %rax
```

```
long unsigned_add
(unsigned long x, unsigned long y)
{
  unsigned long res = x + y;
  return res;
}
```

```
#x in %rdx, y in %rax
addq    %rdx, %rax
```

Some Arithmetic Operations (1 Operand)

- Unary Instructions (one operand)

Format	Computation
`incq` `dest`	Dest = Dest + 1
`decq` `dest`	Dest = Dest - 1
`negq` `dest`	Dest = -Dest
`notq` `dest`	Dest = ~Dest

Arithmetic Expression Example

```
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;
}
```

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

Interesting Instructions

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

Arithmetic Expression Example

```
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;
}
```

Arithmetic Expression Example

```
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;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx   # t5
  imulq   %rcx, %rax           # rval
  ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

Today: Compute and Control Instructions

- Move operations (and addressing modes)
- Arithmetic & logical operations
- Condition Codes
- Control: Conditional branches (if… else…)
- Control: Loops (`for`, `while`)
- Control: Switch Statements (`case`… `switch`…)

Condition Codes

```
addq %rax, %rbx
```

Arithmetic instructions implicitly set condition codes (think of it as side effect)

Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)

CF set when

Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0

ZF set when

Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0

SF set when

Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

Bit-level

```
   111
+) 010
------
  1001
```

Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

Bit-level	**Signed**
111	-1
+) 010	+) 2
1001	1

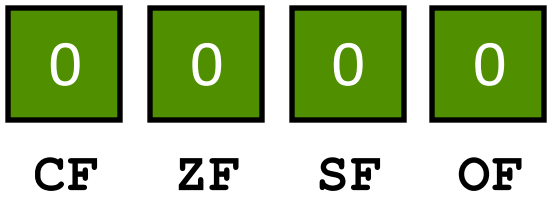
Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

Bit-level	Signed	Unsigned
111	-1	7
+) 010	+) 2	+) 2
1001	1	9



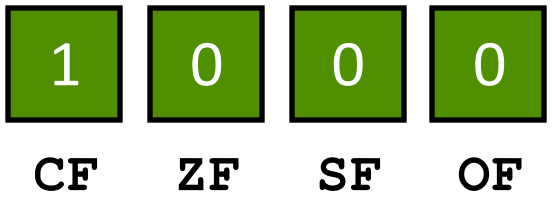
Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

Bit-level	Signed	Unsigned
111	-1	7
+) 010	+) 2	+) 2
1001	1	9



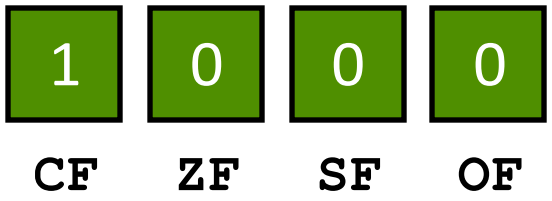
Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

Bit-level	Signed	Unsigned
111	-1	7
+) 010	+) 2	+) 2
1001	1	9



Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

Bit-level

```
   011
+) 001
-------
   100
```

Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

Bit-level	Signed
011	3
+) 001	+) 1
100	-4

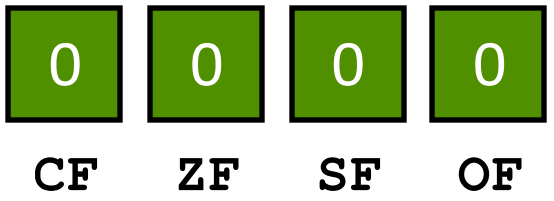
Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

Bit-level	Signed	Unsigned
011	3	3
+) 001	+) 1	+) 1
100	-4	4



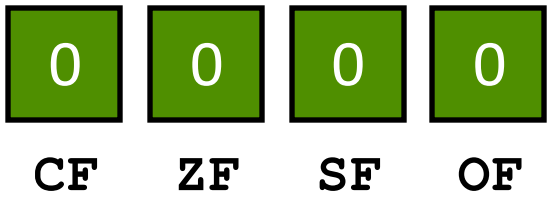
Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

Bit-level	Signed	Unsigned
011	3	3
+) 001	+) 1	+) 1
100	-4	4



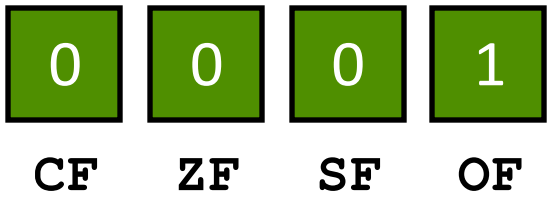
Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

Bit-level	Signed	Unsigned
011	3	3
+) 001	+) 1	+) 1
100	-4	4



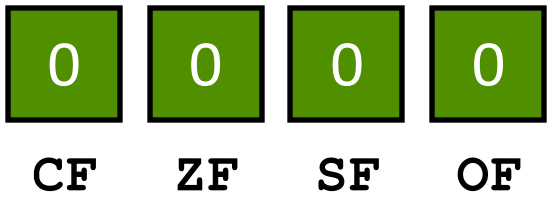
Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

```
   100
+) 111
-------
  c011
```



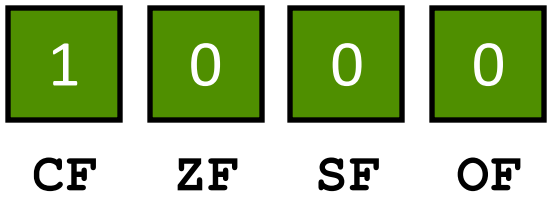
Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

```
   100
+) 111
-------
  c011
```



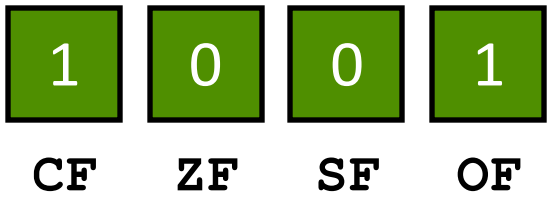
Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

```
   100
+) 111
-------
  c011
```



Compare Instruction

- **`cmpq a, b`**
  - Computes $b - a$ (just like **sub**)
  - Sets condition codes based on result, but…
  - **Does not change b**
  - All it does is setting condition codes!

How Should **`cmpq`** Set Condition Codes?

```
cmpq    %rsi, %rdi
```

```
cmpq 0xFF, 0x80
```

How Should **`cmpq`** Set Condition Codes?

`cmpq` **`%rsi, %rdi`**

`cmpq` *0xFF, 0x80*

```
   10000000      -128
-) 11111111   -)   -1
   --------   --------
   10000001      -127
```

How Should **`cmpq`** Set Condition Codes?

`cmpq %rsi, %rdi`

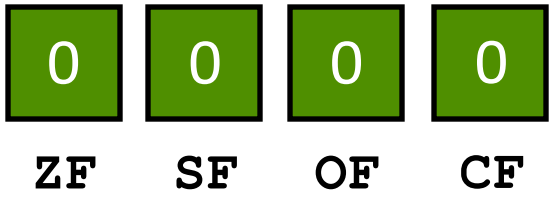
`cmpq` *0xFF, 0x80*

```
   10000000        -128
-) 11111111     -)   -1
   --------     --------
   10000001        -127
```

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (result overflow)



How Should **`cmpq`** Set Condition Codes?

`cmpq %rsi, %rdi`

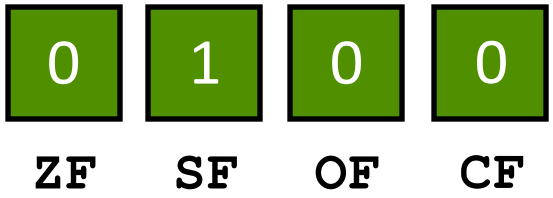
`cmpq` *0xFF, 0x80*

```
   10000000        -128
-) 11111111     -)   -1
   --------     --------
   10000001        -127
```

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (result overflow)



How Should **`cmpq`** Set Condition Codes?

`cmpq %rsi, %rdi`

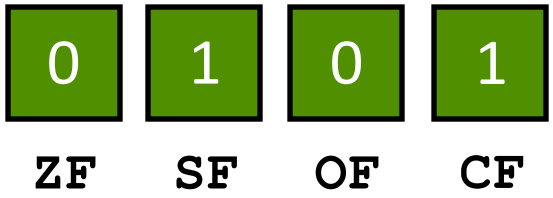
`cmpq` *0xFF, 0x80*

```
   10000000        -128
-) 11111111     -)   -1
   --------     --------
   10000001        -127
```

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (result overflow)



How Should **`cmpq`** Set Condition Codes?

`cmpq %rsi, %rdi`

`cmpq` *0xFF*, *0x80*

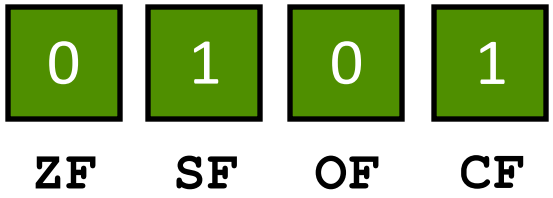
```
   10000000      -128
-) 11111111   -)   -1
   --------   --------
   10000001      -127
```

- How to know if **`%rdi`** = **`%rsi`**?
  - Check `ZF`

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (result overflow)



How Should **`cmpq`** Set Condition Codes?

`cmpq %rsi, %rdi`

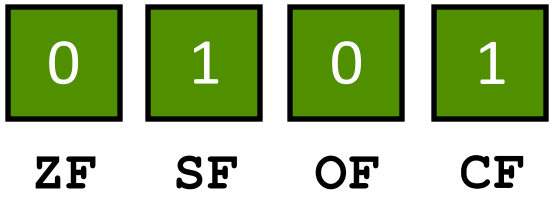
`cmpq` *0xFF, 0x80*

```
   10000000        -128
-) 11111111     -)   -1
   --------     --------
   10000001        -127
```

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (result overflow)



- How to know if **`%rdi`** = **`%rsi`**?
  - Check `ZF`
- How to know if **`%rdi < %rsi(signed)`**?
  - Check `SF` ?
  - `%rdi < %rsi` if and only if: `%rdi - %rsi < 0` (is it correct??)
  - `%rdi` - `%rsi` < 0 and the result doesn't overflow, or
  - `%rdi` - `%rsi` > 0 and the result does overflow
  - or simply: (SF ^ OF)

How Should **`cmpq`** Set Condition Codes?

`cmpq %rsi, %rdi`

`cmpq` *0xFF*, *0x80*

- How to know if **`%rdi`** = **`%rsi`**?
  - Check `ZF`
- How to know if **`%rdi < %rsi(signed)`**?
  - Check `SF` ?
  - `%rdi < %rsi` if and only if: `%rdi - %rsi < 0` (is it correct??)
  - `%rdi` - `%rsi` < 0 and the result doesn't overflow, or
  - `%rdi` - `%rsi` > 0 and the result does overflow
  - or simply: (SF ^ OF)

How Should **`cmpq`** Set Condition Codes?

`cmpq %rsi, %rdi`

`cmpq` *0xFF*, *0x80*

```
   10000000      -128
-) 01111111   -) 127
   --------   ------
   00000001         1
```

- How to know if **`%rdi`** = **`%rsi`**?
  - Check `ZF`
- How to know if **`%rdi < %rsi(signed)`**?
  - Check `SF` ?
  - `%rdi < %rsi` if and only if: `%rdi - %rsi < 0` (is it correct??)
  - `%rdi` - `%rsi` < 0 and the result doesn't overflow, or
  - `%rdi` - `%rsi` > 0 and the result does overflow
  - or simply: (SF ^ OF)

How Should **`cmpq`** Set Condition Codes?

`cmpq %rsi, %rdi`

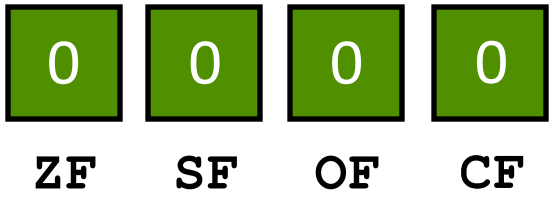
`cmpq` *0xFF*, *0x80*

```
   10000000      -128
-) 01111111   -) 127
   --------   ------
   00000001         1
```

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (result overflow)



- How to know if **`%rdi`** = **`%rsi`**?
  - Check `ZF`
- How to know if **`%rdi < %rsi(signed)`**?
  - Check `SF` ?
  - `%rdi < %rsi` if and only if: `%rdi - %rsi < 0` (is it correct??)
  - `%rdi - %rsi` < 0 and the result doesn't overflow, or
  - `%rdi - %rsi` > 0 and the result does overflow
  - or simply: (SF ^ OF)

How Should **`cmpq`** Set Condition Codes?

`cmpq %rsi, %rdi`

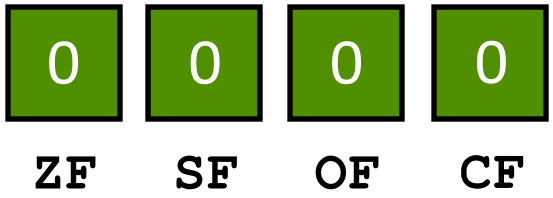
`cmpq` *0xFF*, *0x80*

```
   10000000      -128
-) 01111111   -) 127
   --------   ------
   00000001         1
```

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (result overflow)



- How to know if **`%rdi`** = **`%rsi`**?
  - Check `ZF`
- How to know if **`%rdi < %rsi(signed)`**?
  - Check `SF` ?
  - `%rdi < %rsi` if and only if: `%rdi - %rsi < 0` (is it correct??)
  - `%rdi - %rsi` < 0 and the result doesn't overflow, or
  - `%rdi - %rsi` > 0 and the result does overflow
  - or simply: (SF ^ OF)

How Should **`cmpq`** Set Condition Codes?

`cmpq %rsi, %rdi`

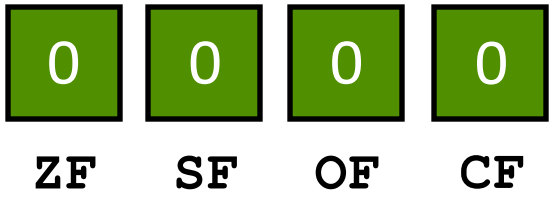
`cmpq` *0xFF, 0x80*

```
   10000000      -128
-) 01111111   -) 127
   --------   ------
   00000001         1
```

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (result overflow)



- How to know if **`%rdi`** = **`%rsi`**?
  - Check `ZF`
- How to know if **`%rdi < %rsi(signed)`**?
  - Check `SF` ?
  - `%rdi < %rsi` if and only if: `%rdi - %rsi < 0` (is it correct??)
  - `%rdi` - `%rsi` < 0 and the result doesn't overflow, or
  - `%rdi` - `%rsi` > 0 and the result does overflow
  - or simply: (SF ^ OF)

How Should **`cmpq`** Set Condition Codes?

`cmpq %rsi, %rdi`

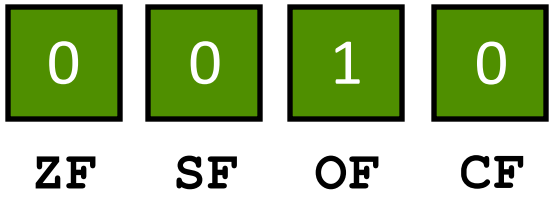
`cmpq` *0xFF, 0x80*

```
   10000000      -128
-) 01111111   -) 127
   --------   ------
   00000001         1
```

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (result overflow)



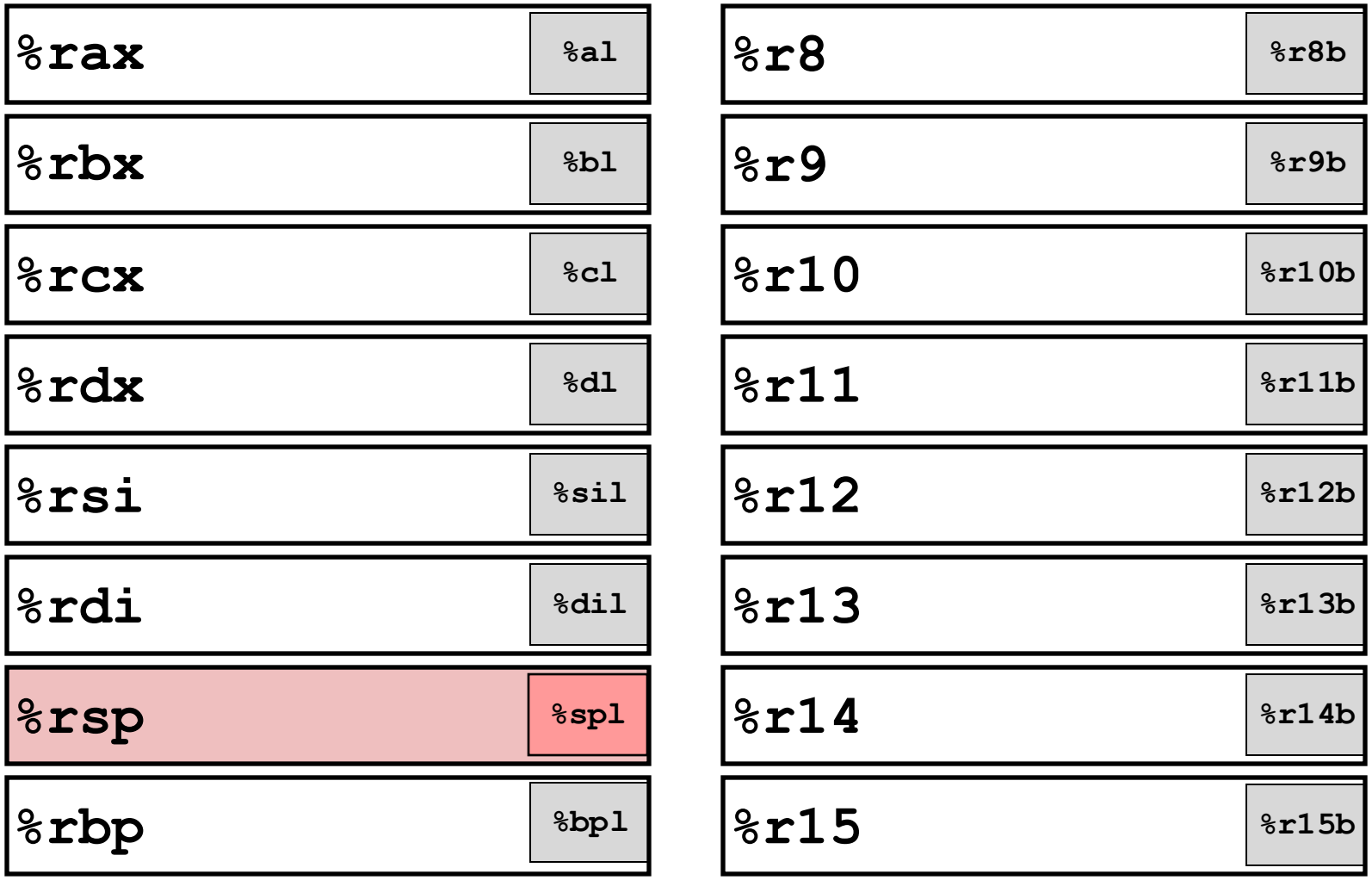
- How to know if **`%rdi`** = **`%rsi`**?
  - Check `ZF`
- How to know if **`%rdi < %rsi(signed)`**?
  - Check `SF` ?
  - `%rdi < %rsi` if and only if: `%rdi - %rsi < 0` (is it correct??)
  - `%rdi` - `%rsi` < 0 and the result doesn't overflow, or
  - `%rdi` - `%rsi` > 0 and the result does overflow
  - or simply: (SF ^ OF)

Reading Condition Codes

- SetX Instructions
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes

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)

x86-64 Integer Registers



- SetX argument is always a low byte (%al, %r8b, etc.)

Reading Condition Codes (Cont.)

- **`SetX` Instructions:**
  - Set single byte based on combination of condition codes
- One of addressable byte registers
  - Does not alter remaining bytes
  - Typically use **`movzbl`** to finish job
    - 32-bit instructions also set upper 32 bits to 0

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

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

```
      cmpq   %rsi, %rdi    # Compare x:y
      setg   %al           # Set when >
      movzbl %al, %eax     # Zero rest of %rax
      ret
```

Reading Condition Codes (Cont.)

- **`SetX` Instructions:**
  - Set single byte based on combination of condition codes
- One of addressable byte registers
  - Does not alter remaining bytes
  - Typically use **`movzbl`** to finish job
    - 32-bit instructions also set upper 32 bits to 0

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

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

```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al           # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

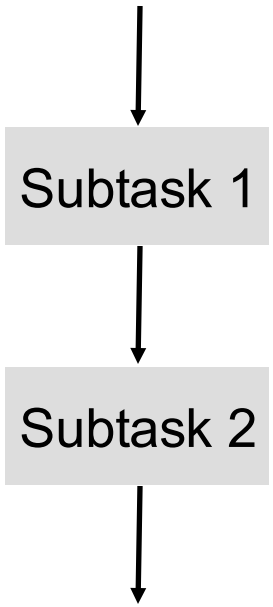
Today: Compute and Control Instructions

- Move operations (and addressing modes)
- Arithmetic & logical operations
- Condition Codes
- Control: Conditional branches (if… else…)
- Control: Loops (for, while)
- Control: Switch Statements (case… switch…)

Three Basic Programming Constructs

Three Basic Programming Constructs

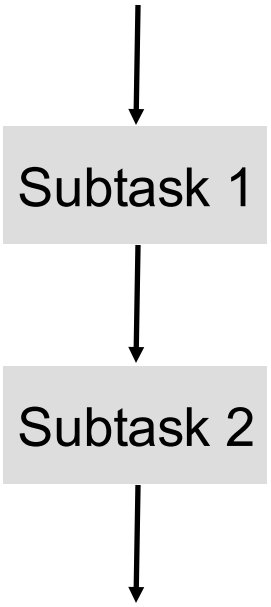
Sequential




```
a = x + y;
y = a - c;
…
```

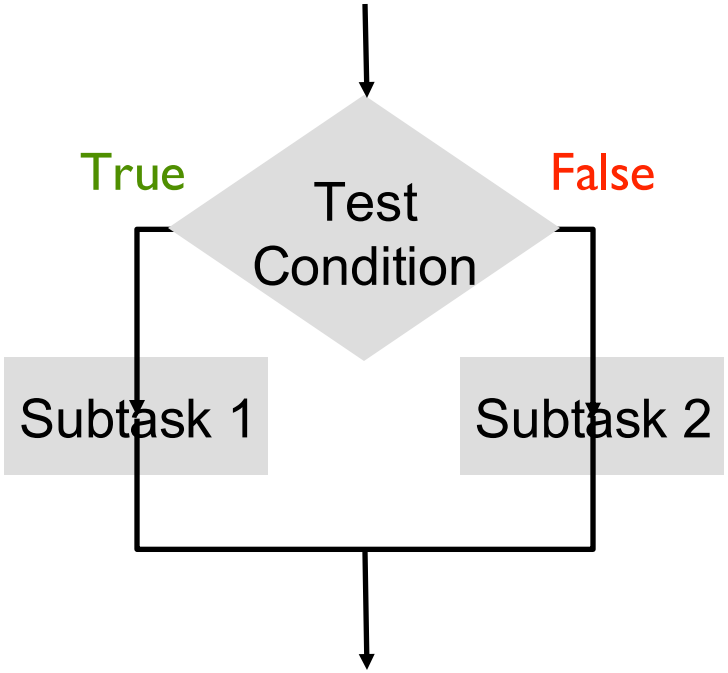
Three Basic Programming Constructs

Sequential




```
a = x + y;
y = a - c;
…
```

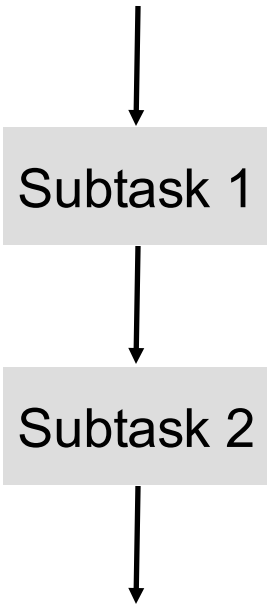
Conditional




```
if (x > y) r = x - y;
else r = y - x;
```

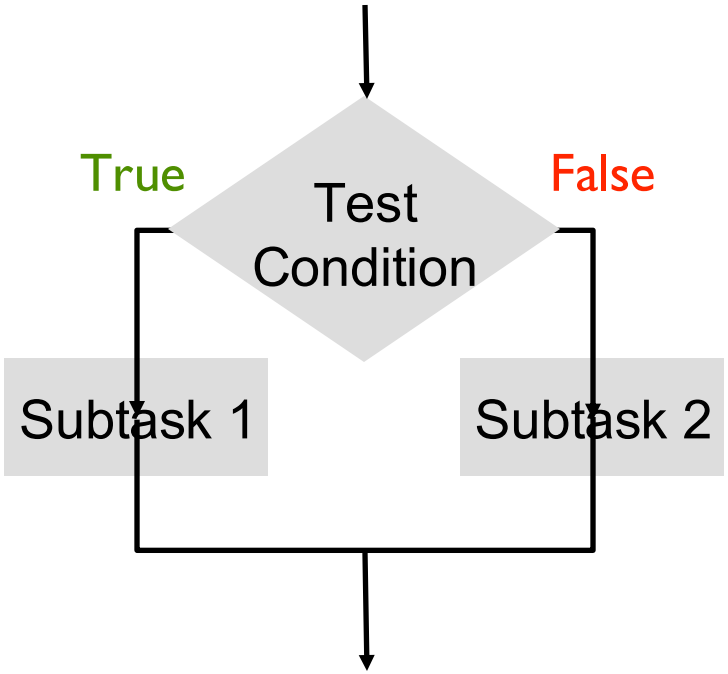
Three Basic Programming Constructs

Sequential



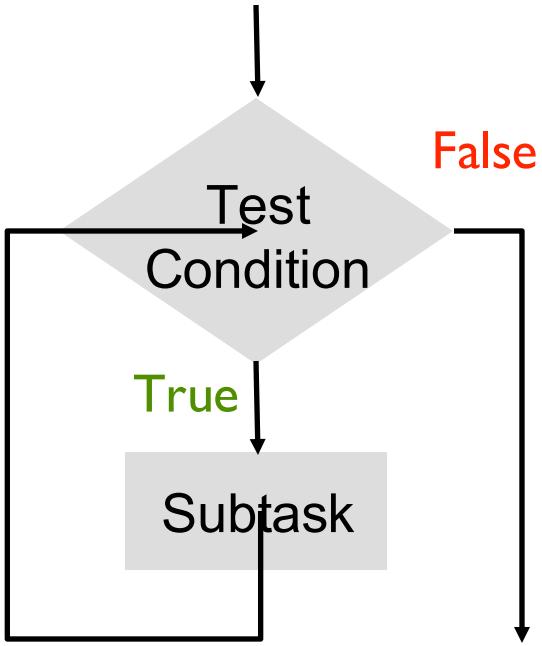

```
a = x + y;
y = a - c;
…
```

Conditional



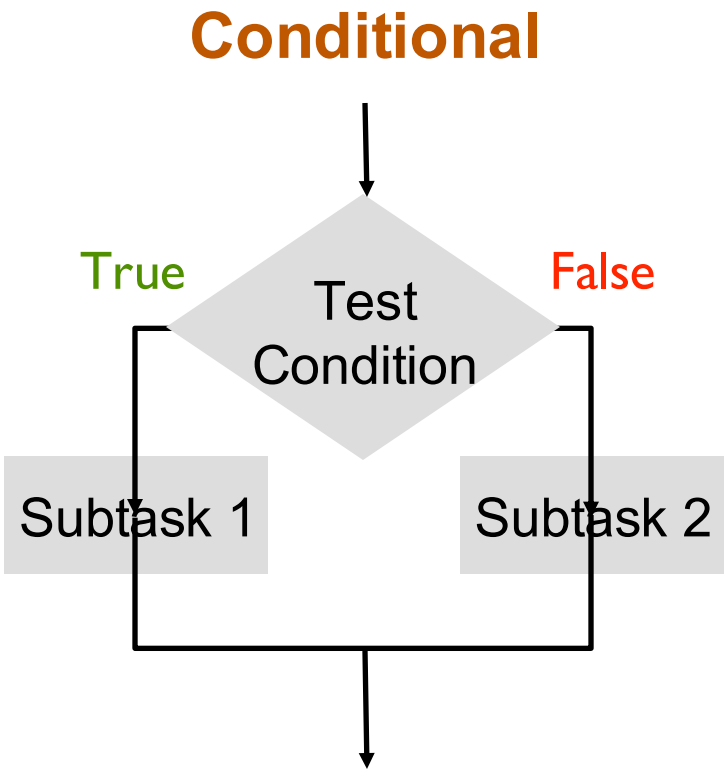

```
if (x > y) r = x - y;
else r = y - x;
```

Iterative




```
while (x > 0) {
  x--;
}
```

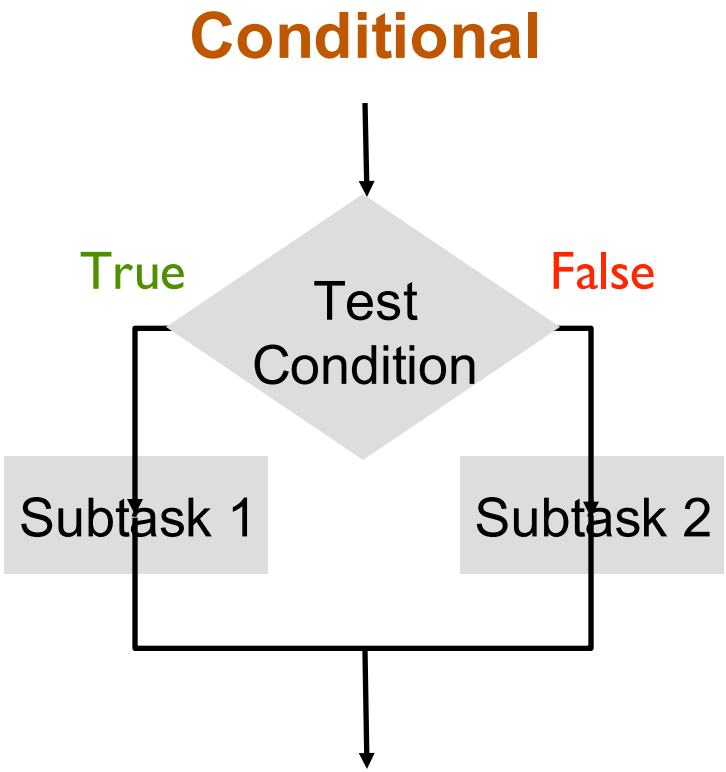
Three Basic Programming Constructs




```
if (x > y) r = x - y;
else r = y - x;
```

Three Basic Programming Constructs

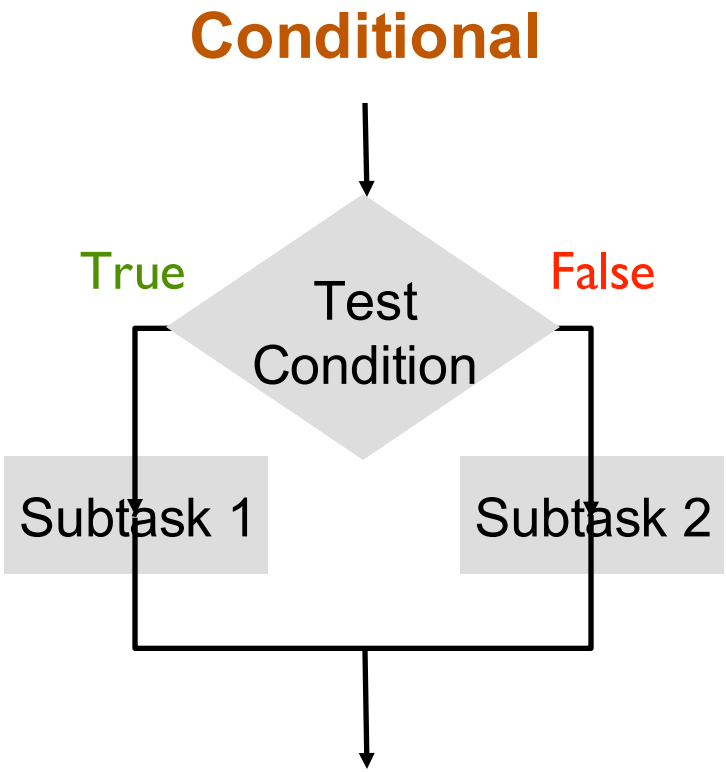
- Both conditional and iterative programming requires altering the sequence of instructions (control flow)




```
if (x > y) r = x - y;
else r = y - x;
```

Three Basic Programming Constructs

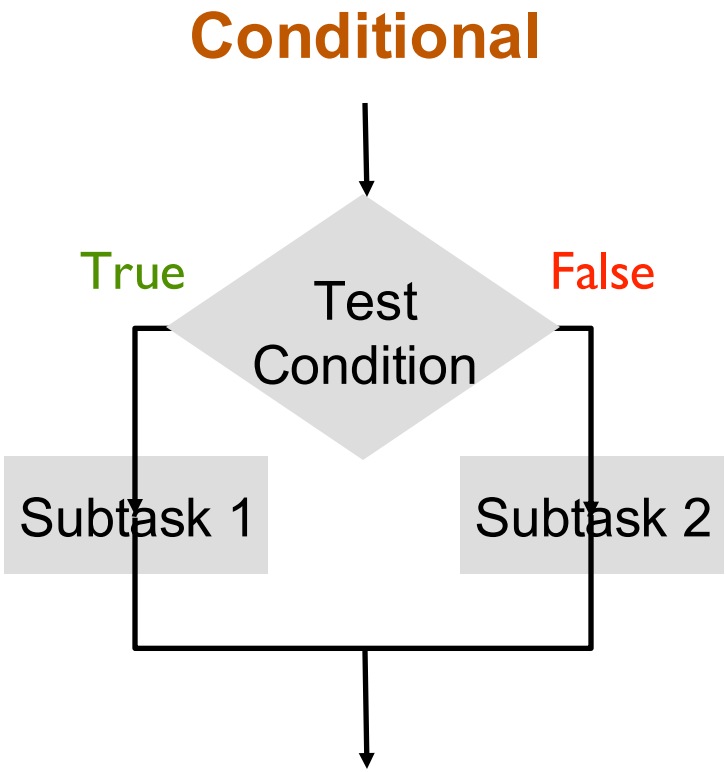
- Both conditional and iterative programming requires altering the sequence of instructions (control flow)
- We need a set of *control instructions* to do so




```
if (x > y) r = x - y;
else r = y - x;
```

Three Basic Programming Constructs

- Both conditional and iterative programming requires altering the sequence of instructions (control flow)
- We need a set of *control instructions* to do so
- Two fundamental questions:
  - How to test condition and how to represent test results?
  - How to alter control flow according to the test results?




```
if (x > y) r = x - y;
else r = y - x;
```

Jumping

- `jX` Instructions
  - Jump to different part of code depending on condition codes

jX	Condition	Description
`jmp`	`1`	Unconditional
`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)

Conditional Branch Example

Conditional Branch Example

```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

Conditional Branch Example

```
gcc –Og –S –fno-if-conversion control.c
```

```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
    cmpq    %rsi,%rdi # x:y
    jle     .L4
    movq    %rdi,%rax
    subq    %rsi,%rax
    ret
.L4:                  # x <= y
    movq    %rsi,%rax
    subq    %rdi,%rax
    ret
```

Conditional Branch Example

```
gcc –Og –S –fno-if-conversion control.c
```

```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

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

```
absdiff:
    cmpq    %rsi,%rdi # x:y
    jle     .L4
    movq    %rdi,%rax
    subq    %rsi,%rax
    ret
.L4:                  # x <= y
    movq    %rsi,%rax
    subq    %rdi,%rax
    ret
```

Conditional Branch Example

```
gcc –Og –S –fno-if-conversion control.c
```

```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

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

```
absdiff:
    cmpq    %rsi,%rdi # x:y
    jle     .L4
    movq    %rdi,%rax
    subq    %rsi,%rax
    ret
.L4:                  # x <= y
    movq    %rsi,%rax
    subq    %rdi,%rax
    ret
```

Labels are symbolic names used to refer to instruction addresses.

Conditional Branch Example

```
gcc –Og –S –fno-if-conversion control.c
```

```
unsigned long absdiff
  (unsigned long x,
unsigned long y)
{
  unsigned long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

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

```
absdiff:
    cmpq    %rsi,%rdi # x:y
    jle     .L4
    movq    %rdi,%rax
    subq    %rsi,%rax
    ret
.L4:                  # x <= y
    movq    %rsi,%rax
    subq    %rdi,%rax
    ret
```

Labels are symbolic names used to refer to instruction addresses.

Conditional Branch Example

```
gcc –Og –S –fno-if-conversion control.c
```

```
unsigned long absdiff
   (unsigned long x,
unsigned long y)
{
  unsigned long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

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

```
absdiff:
    cmpq    %rsi,%rdi # x:y
    jbe     .L4
    movq    %rdi,%rax
    subq    %rsi,%rax
    ret
.L4:                  # x <= y
    movq    %rsi,%rax
    subq    %rdi,%rax
    ret
```

Labels are symbolic names used to refer to instruction addresses.

Conditional Jump Instruction

```
cmpq    %rsi, %rdi
jle     .L4
```

Conditional Jump Instruction

```
cmpq    %rsi, %rdi
jle     .L4
```

Jump to label if less than or equal to

Conditional Jump Instruction

```
cmpq    %rsi, %rdi
jle     .L4
```

Jump to label if less than or equal to

- Semantics:
  - If **`%rdi`** is less than or equal to **`%rsi`** (both interpreted as **signed value**), jump to the part of the code with a label **`.L4`**

Conditional Jump Instruction

```
cmpq    %rsi, %rdi
jle     .L4
```

Jump to label if less than or equal to

- Semantics:
  - If **`%rdi`** is less than or equal to **`%rsi`** (both interpreted as **signed value**), jump to the part of the code with a label **`.L4`**
- Under the hood:

Conditional Jump Instruction

```
cmpq    %rsi, %rdi
jle     .L4
```

Jump to label if less than or equal to

- Semantics:
  - If **`%rdi`** is less than or equal to **`%rsi`** (both interpreted as **signed value**), jump to the part of the code with a label **`.L4`**
- Under the hood:
  - **`cmpq`** instruction sets the condition codes

Conditional Jump Instruction

```
cmpq    %rsi, %rdi
jle     .L4
```

Jump to label if less than or equal to

- Semantics:
  - If **`%rdi`** is less than or equal to **`%rsi`** (both interpreted as **signed value**), jump to the part of the code with a label **`.L4`**
- Under the hood:
  - **`cmpq`** instruction sets the condition codes
  - **`jle`** reads and checks the **condition codes**

Conditional Jump Instruction

```
cmpq    %rsi, %rdi
jle     .L4
```

Jump to label if less than or equal to

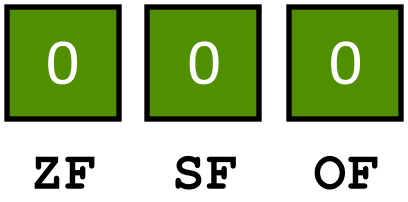
- Semantics:
  - If **`%rdi`** is less than or equal to **`%rsi`** (both interpreted as **signed value**), jump to the part of the code with a label **`.L4`**
- Under the hood:
  - **`cmpq`** instruction sets the condition codes
  - **`jle`** reads and checks the **condition codes**
  - If condition met, modify the Program Counter to point to the address of the instruction with a label **`.L4`**

Conditional Branch Example

```
long absdiff
   (long x, long y)
{
   long result;
   if (x > y)
     result = x-y;
   else
     result = y-x;
   return result;
}
```

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

```
absdiff:
    cmpq    %rsi,%rdi # x:y
    jle     .L4
    movq    %rdi,%rax
    subq    %rsi,%rax
    ret
.L4:                  # x <= y
    movq    %rsi,%rax
    subq    %rdi,%rax
    ret
```



Conditional Branch Example

```
long absdiff
   (long x, long y)
{
   long result;
   if (x > y)
     result = x-y;
   else
     result = y-x;
   return result;
}
```

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

```
absdiff:
    cmpq    %rsi,%rdi # x:y
    jle     .L4
    movq    %rdi,%rax
    subq    %rsi,%rax
    ret
.L4:                  # x <= y
    movq    %rsi,%rax
    subq    %rdi,%rax
    ret
```

`cmpq` sets ZF, SF, OF

`jle` checks ZF | (SF ^ OF)