CSC 252: Computer Organization Spring 2021: Lecture 6

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcement

- Programming Assignment 2 is out
  - Details: https://www.cs.rochester.edu/courses/252/spring2021/labs/assignment2.html
  - Due on **March 3**, 11:59 PM
  - You (may still) have 3 slip days

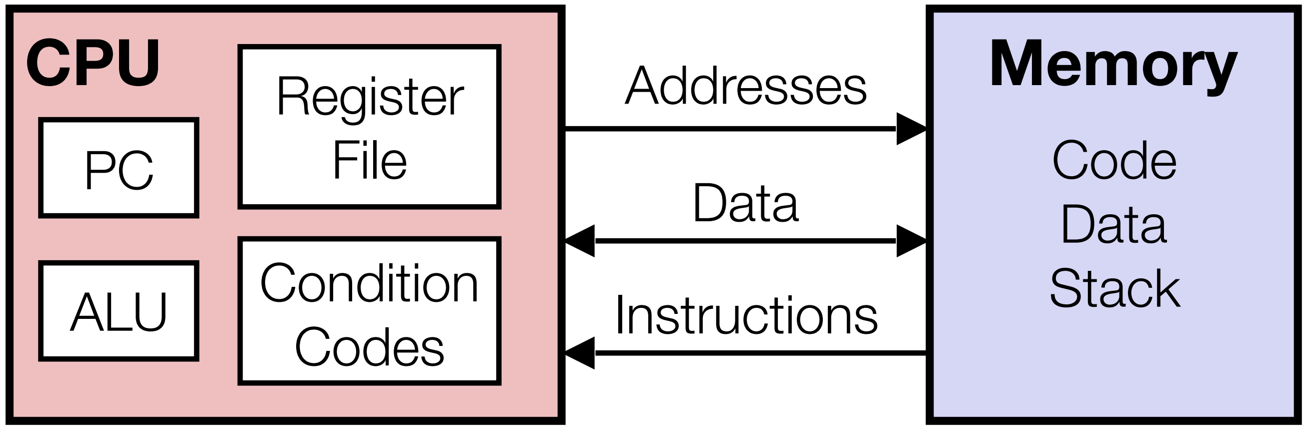
14	15	16	17	18 **Today**	19	20
21	22	23	24	25	26	27
28	Mar 1	2	3 **Due**	4	5	6

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.
- Problem set on arithmetics: https://www.cs.rochester.edu/courses/252/spring2021/handouts.html.
  - Not to be turned in.

Instruction Processing Sequence

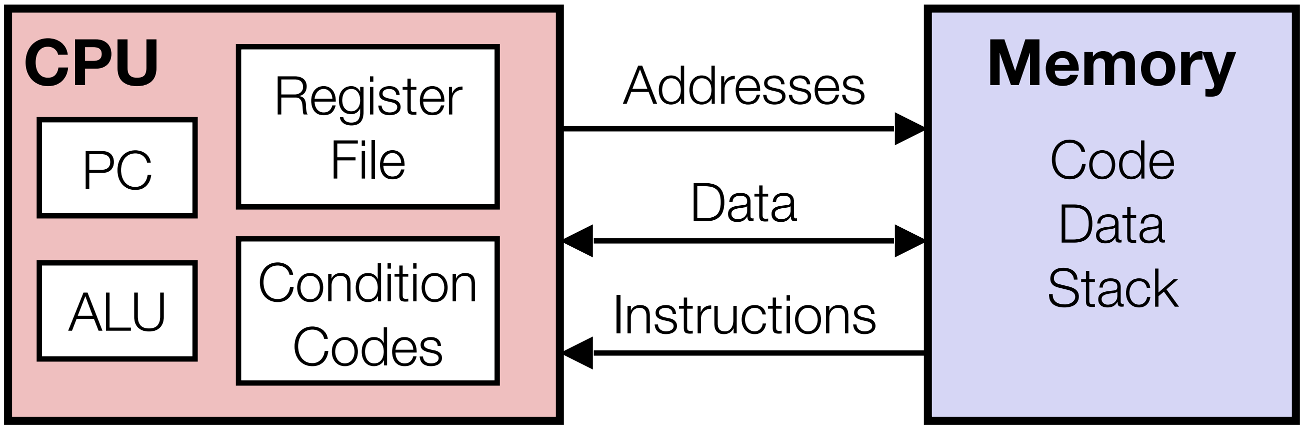
Assembly Programmer's Perspective of a Computer



Fetch Instruction (According to PC)

Instruction Processing Sequence

Assembly Programmer's Perspective of a Computer

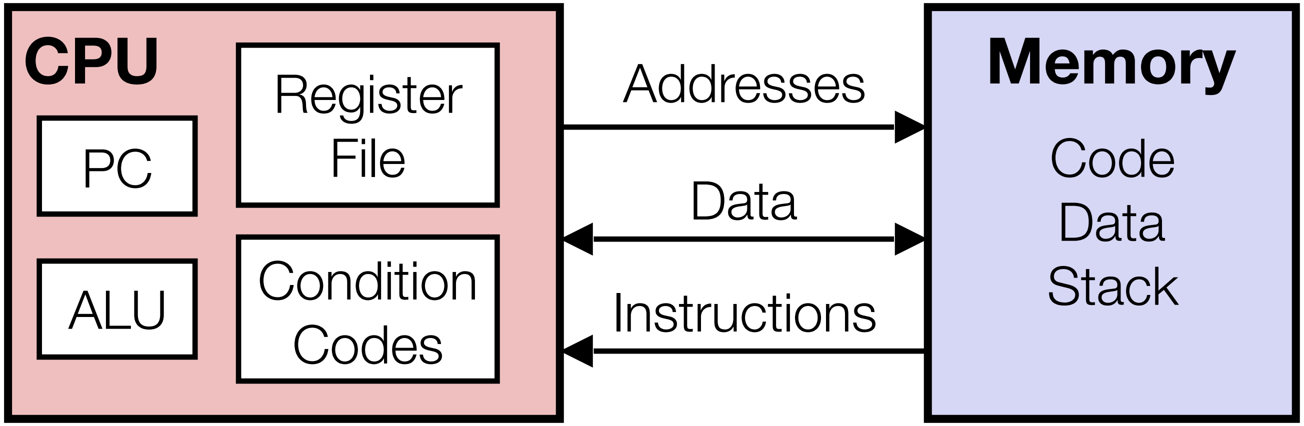


Fetch Instruction (According to PC)

`0x4801d8`

Instruction Processing Sequence

Assembly Programmer's Perspective of a Computer

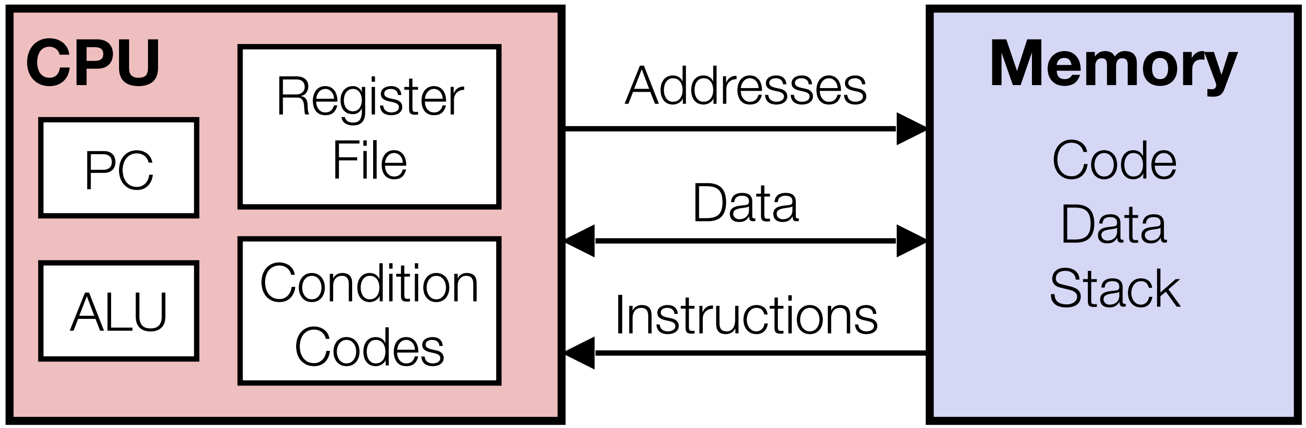


Fetch Instruction (According to PC) → Decode Instruction

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

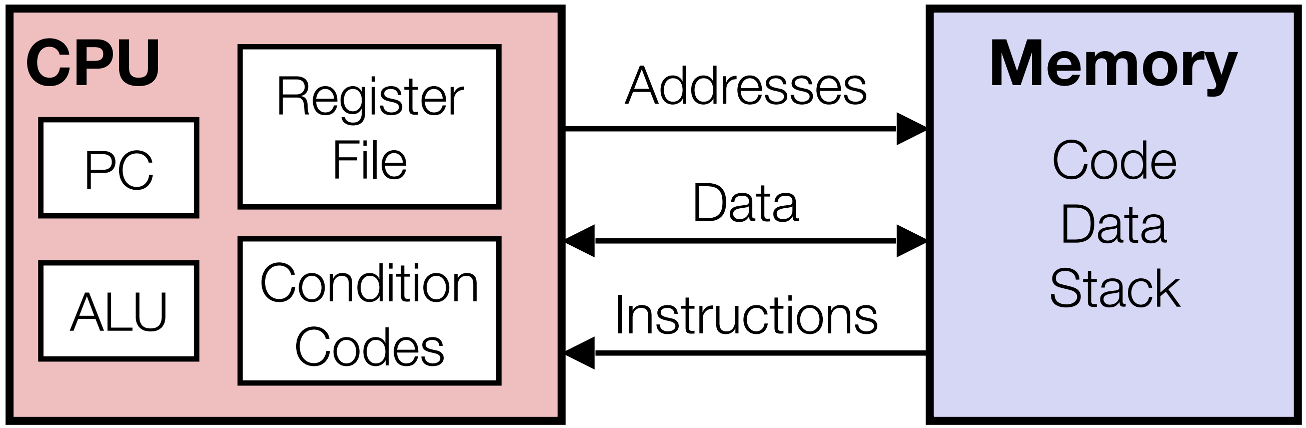
Instruction Processing Sequence

Assembly Programmer's Perspective of a Computer



Instruction Processing Sequence

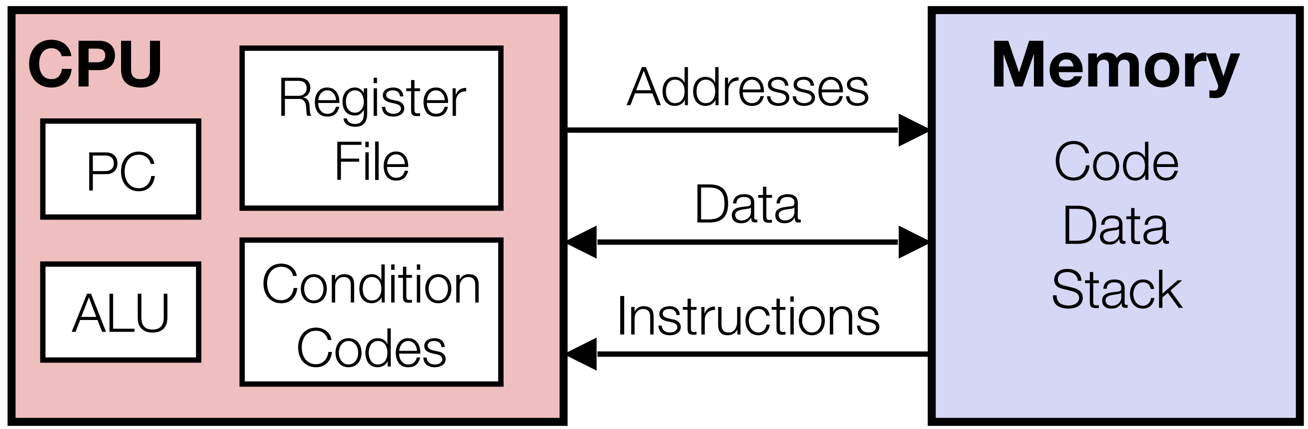
Assembly Programmer's Perspective of a Computer

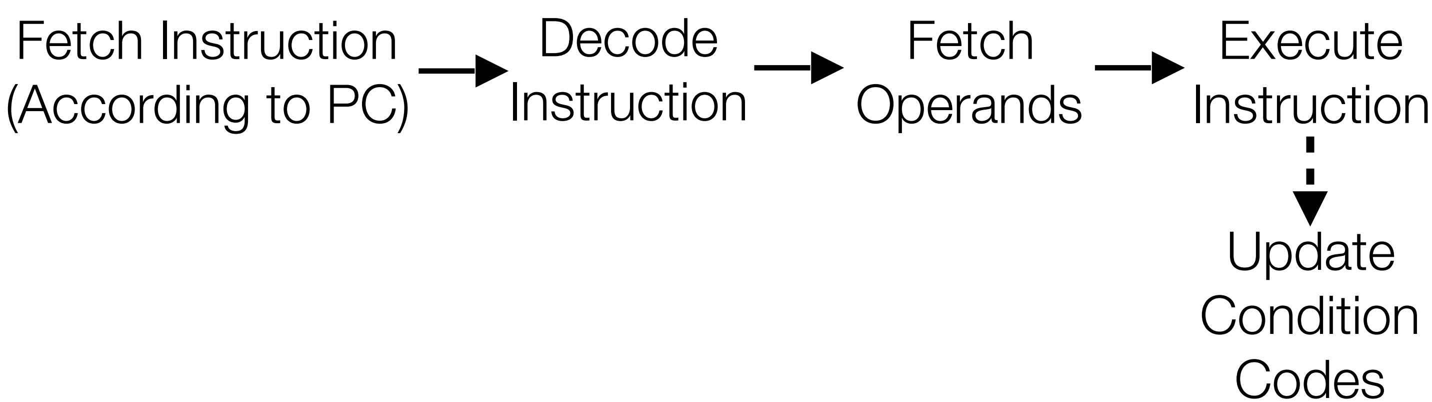


Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands → Execute Instruction

Instruction Processing Sequence

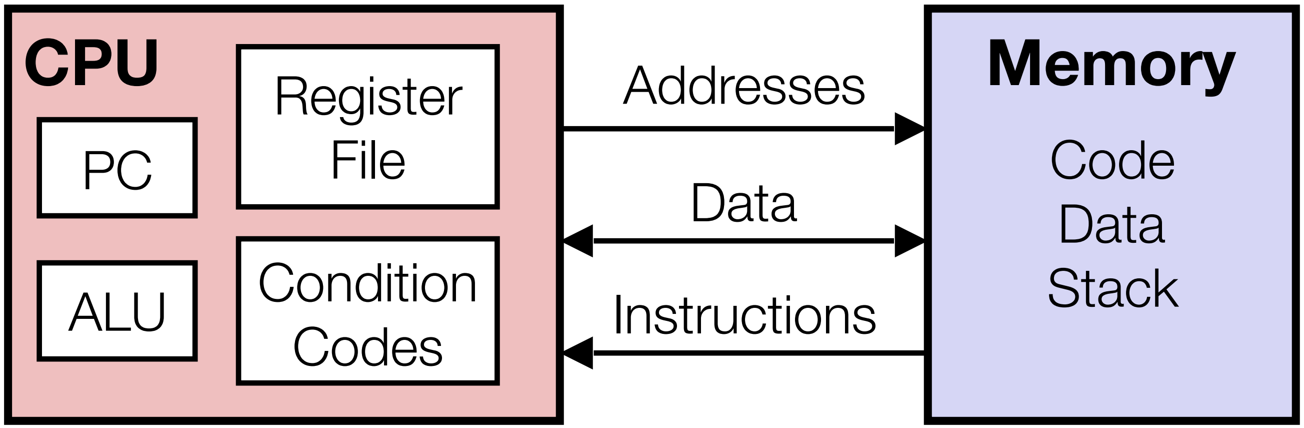
Assembly Programmer's Perspective of a Computer





Instruction Processing Sequence

Assembly Programmer's Perspective of a Computer

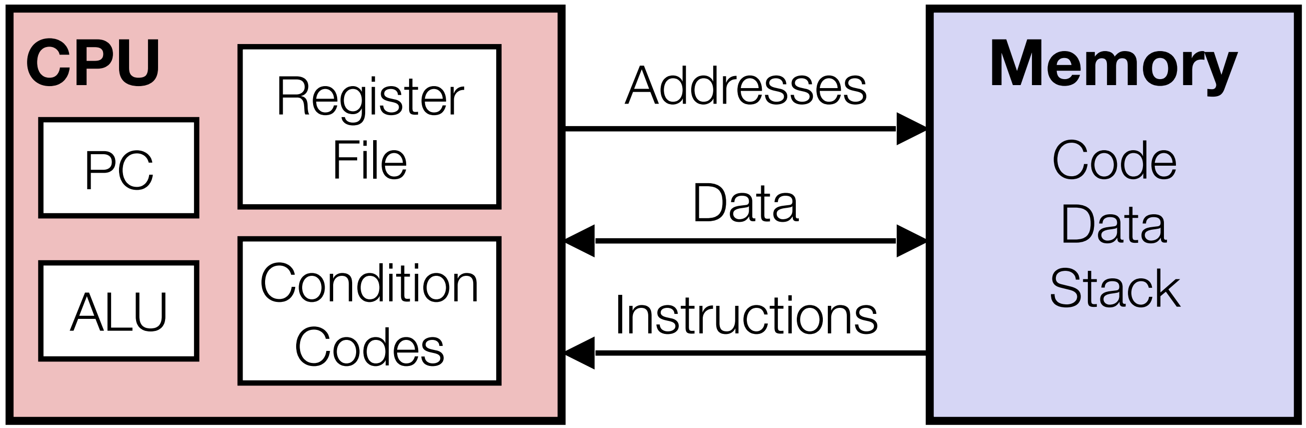


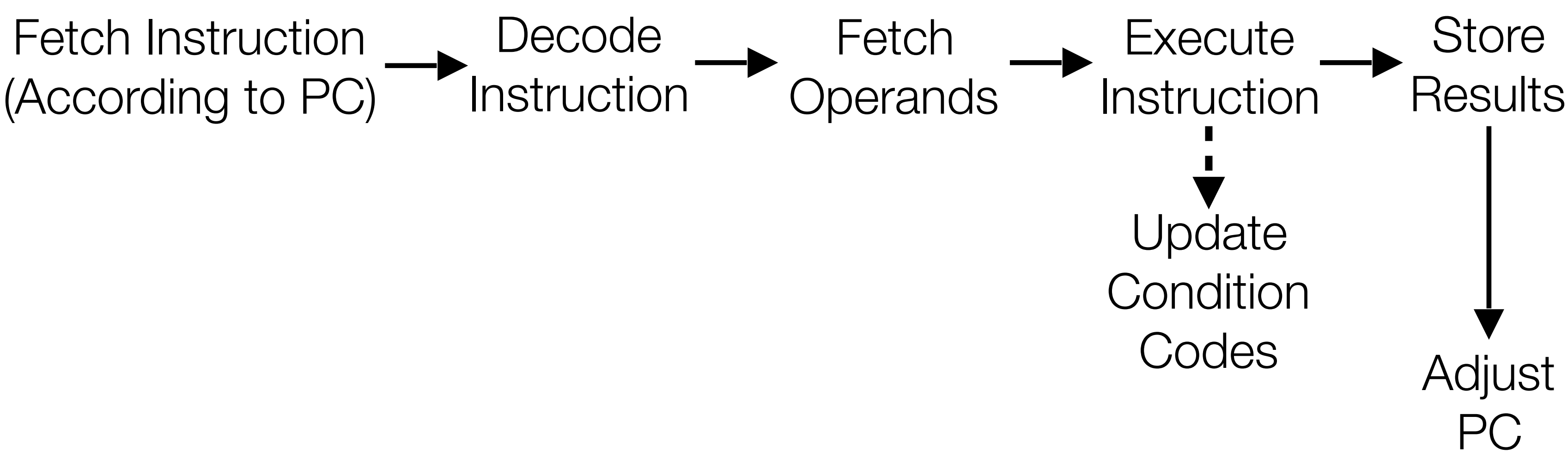
Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands → Execute Instruction → Store Results

Execute Instruction → Update Condition Codes

Instruction Processing Sequence

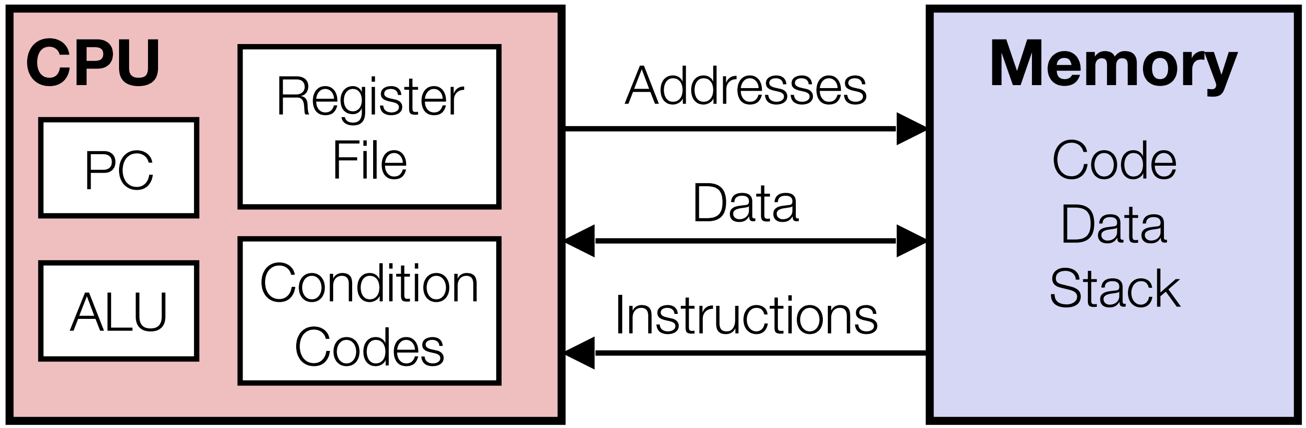
Assembly
Programmer's
Perspective
of a Computer

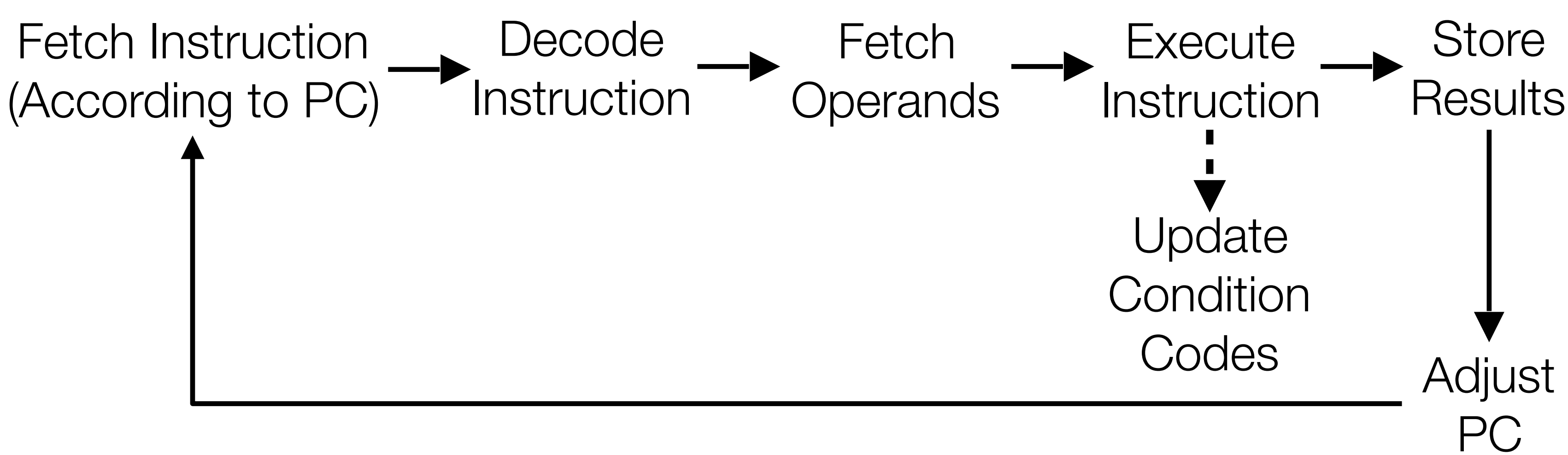




Instruction Processing Sequence

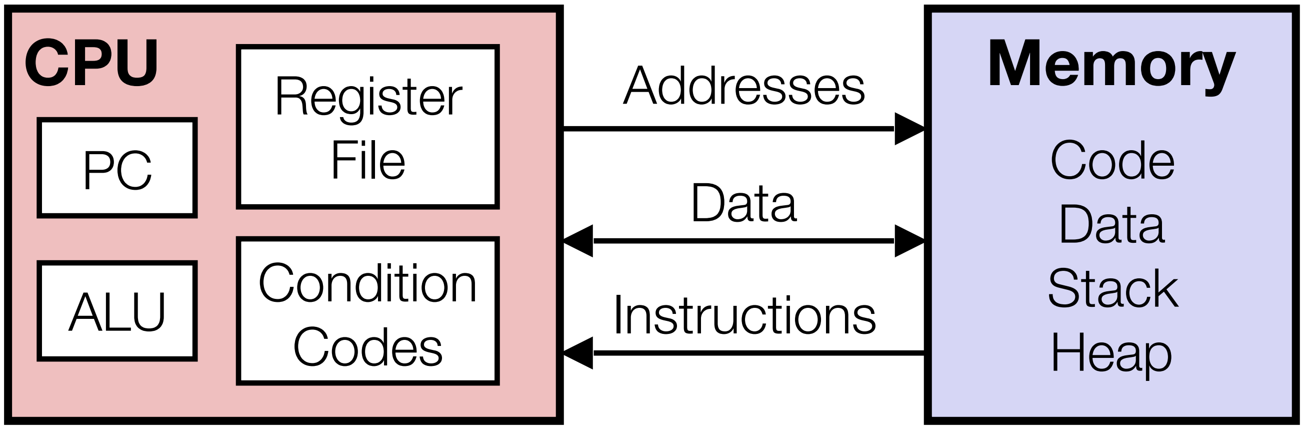
Assembly Programmer's Perspective of a Computer





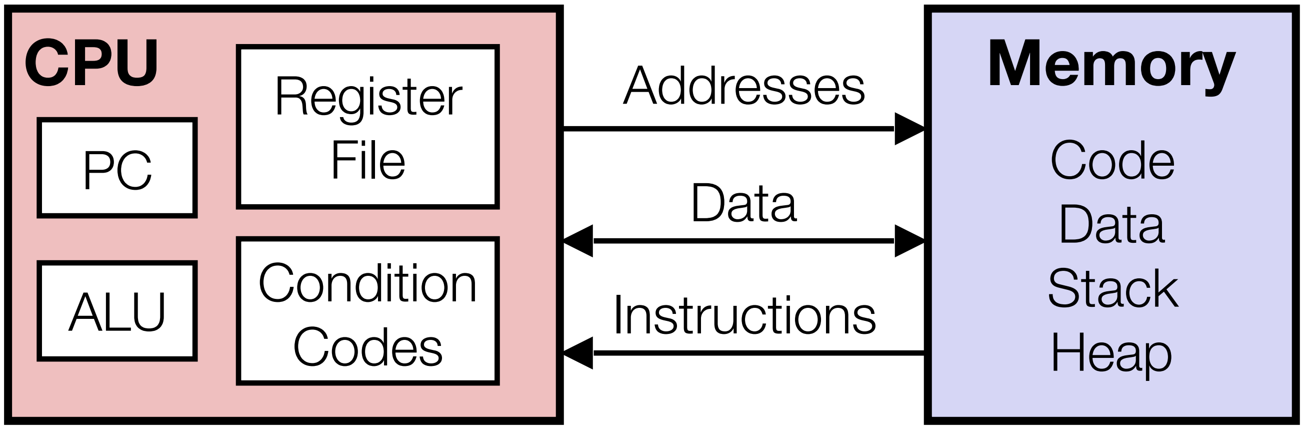
Assembly Program Instructions

Assembly Programmer's Perspective of a Computer



Assembly Program Instructions

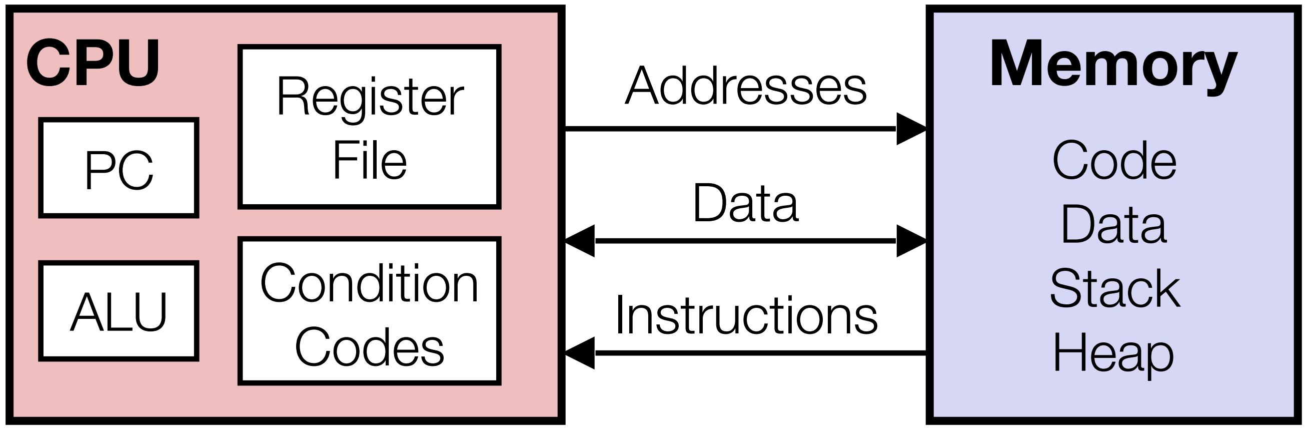
Assembly Programmer's Perspective of a Computer



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

Assembly Program Instructions

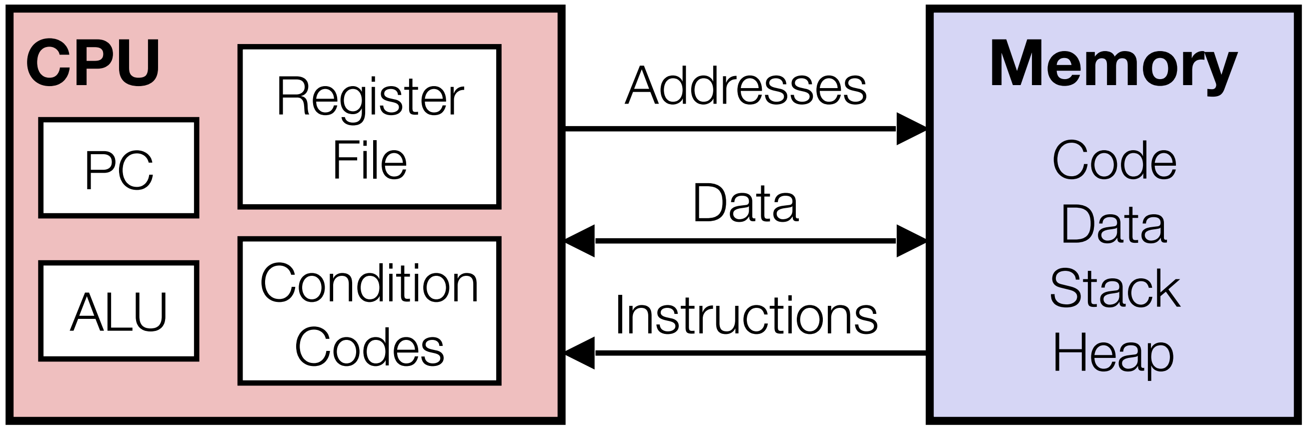
Assembly Programmer's Perspective of a Computer



- *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



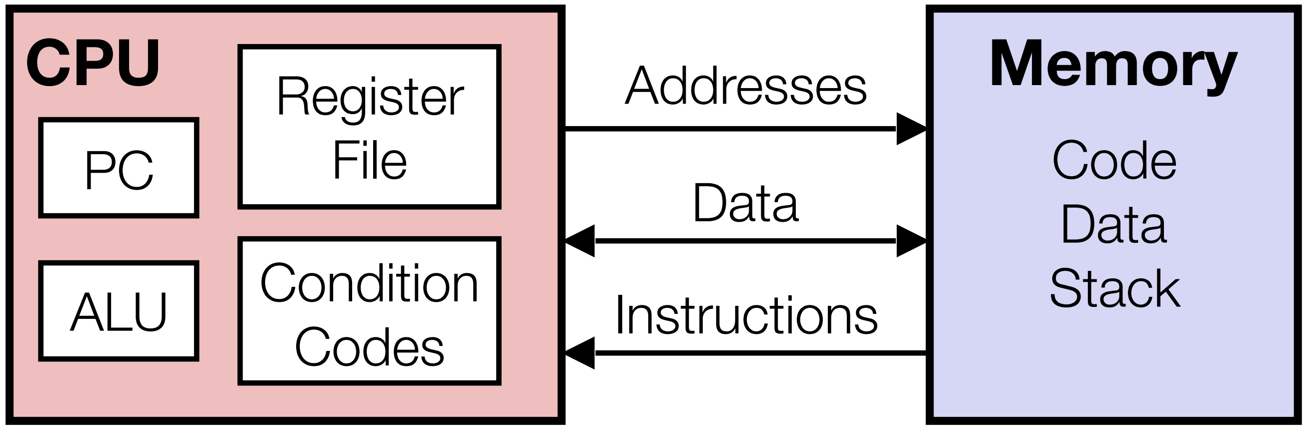
- *Compute Instruction*: Perform arithmetics on register or memory data
  - **`addq %eax, %ebx`**
  - C constructs: +, -, >>, etc.
- *Data Movement Instruction*: Transfer data between memory and register
  - **`movq %eax, (%ebx)`**
- *Control Instruction*: Alter the sequence of instructions (by changing 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
- Control: Conditional branches (**`if`**… **`else`**…)
- Control: Loops (**`for`**, **`while`**)
- Control: Switch Statements (**`case`**… **`switch`**…)

Data Movement in Processors

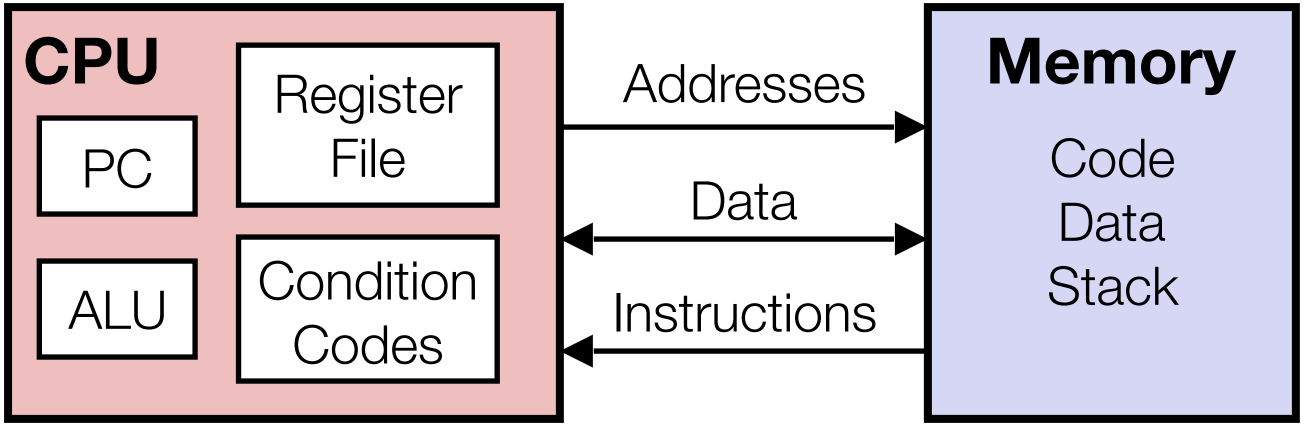
Assembly Programmer's Perspective of a Computer



- Initially all data is in the memory

Data Movement in Processors

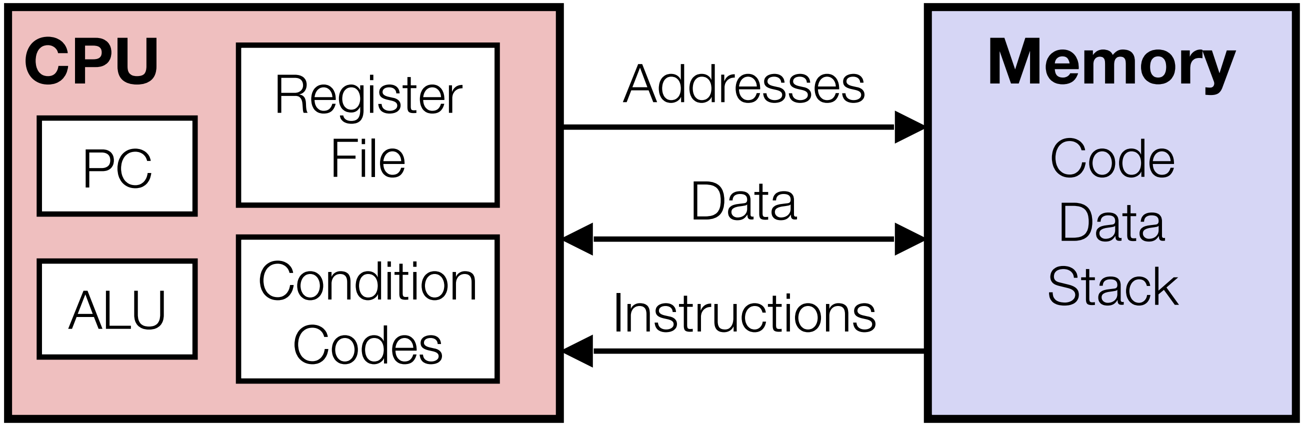
Assembly Programmer's Perspective of a Computer



- Initially all data is in the memory
- But memory is slow: e.g., 15 ns for each access

Data Movement in Processors

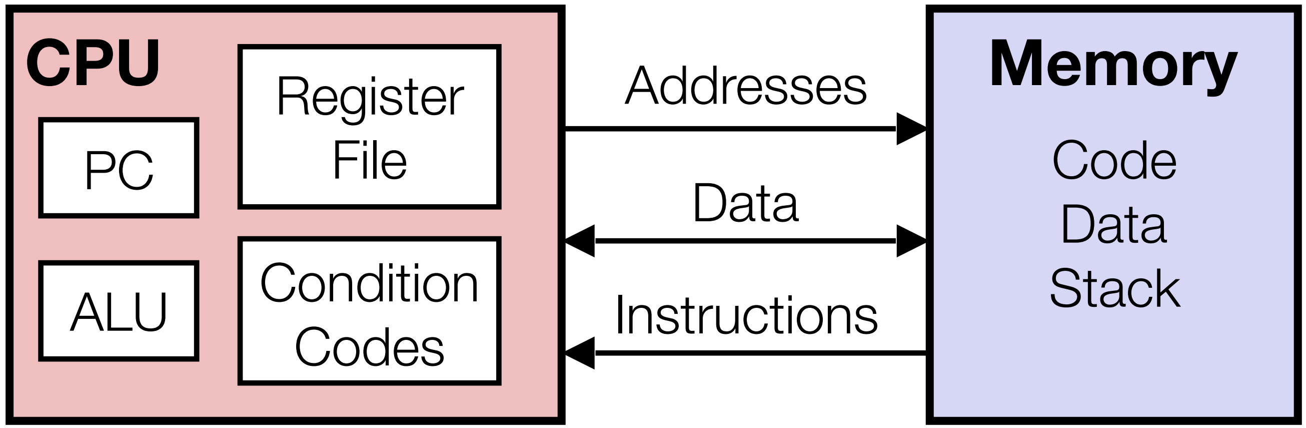
Assembly Programmer's Perspective of a Computer



- Initially all data is in the memory
- But memory is slow: e.g., 15 ns for each access
- Idea: move the frequently used data to a faster memory

Data Movement in Processors

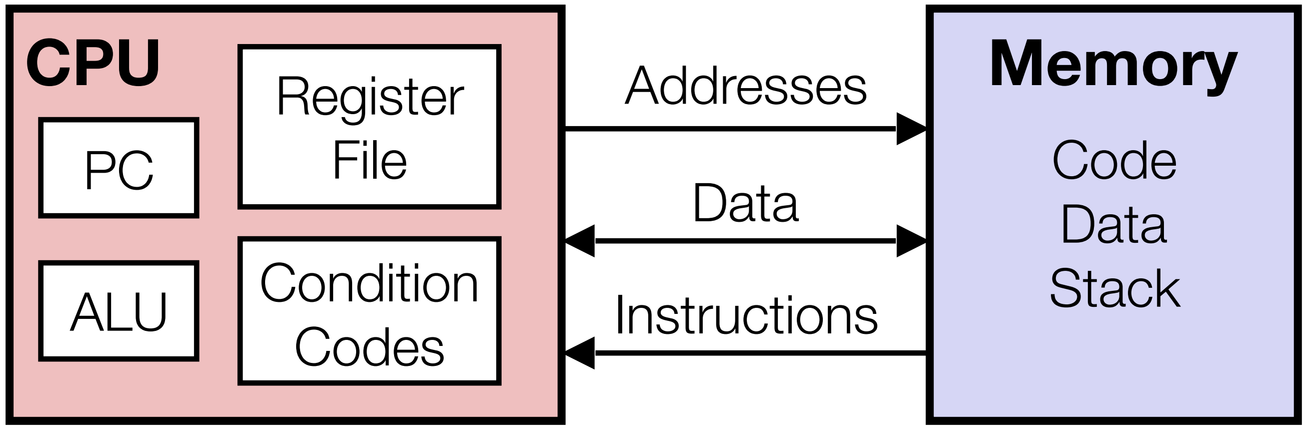
Assembly Programmer's Perspective of a Computer



- Initially all data is in the memory
- But memory is slow: e.g., 15 ns for each access
- Idea: move the frequently used data to a faster memory
- Register file is faster (but much smaller) memory: e.g., 0.5 ns

Data Movement in Processors

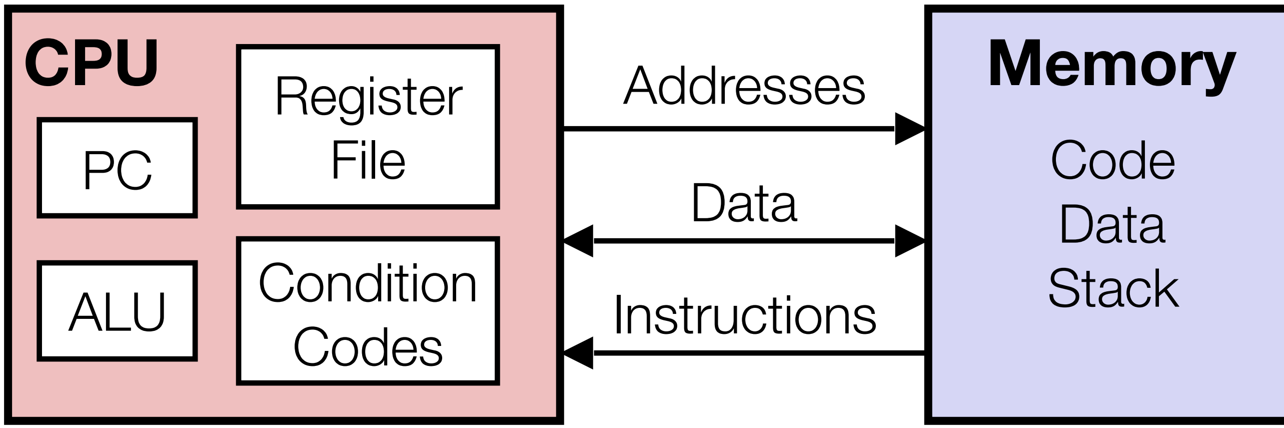
Assembly Programmer's Perspective of a Computer



- Initially all data is in the memory
- But memory is slow: e.g., 15 ns for each access
- Idea: move the frequently used data to a faster memory
- Register file is faster (but much smaller) memory: e.g., 0.5 ns
- There are other kinds of faster memory that we will talk about later

Data Movement in Processors

Assembly Programmer's Perspective of a Computer



- Initially all data is in the memory
- But memory is slow: e.g., 15 ns for each access
- Idea: move the frequently used data to a faster memory
- Register file is faster (but much smaller) memory: e.g., 0.5 ns
- There are other kinds of faster memory that we will talk about later
- Key: register file is programmer visible, i.e., you could use instructions to explicitly move data between memory and register file.

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)
```

address

- 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

data at the address

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

address

- 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

data at the address

`movq      %rdx, (%rdi)`

address

`*p = a;`
assuming:
`p` is in `$rdi`
`a` is in `$rdx`

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

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
```

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

$\underbrace{\texttt{movq}}_{\text{Operator}}\ \underbrace{\textit{Source, Dest}}_{\text{Operands}}$

- 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

	Source	Dest	Example	C Analog
`movq`	*Imm*	*Reg*	**`movq $0x4,%rax`**	
		Mem		
	Reg	*Reg*		
		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		
	Reg	*Reg*		
		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)`**	
	Reg	*Reg*		
		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*		
		Mem		
	Mem	*Reg*		

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

`movq` Operand Combinations

`movq`

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

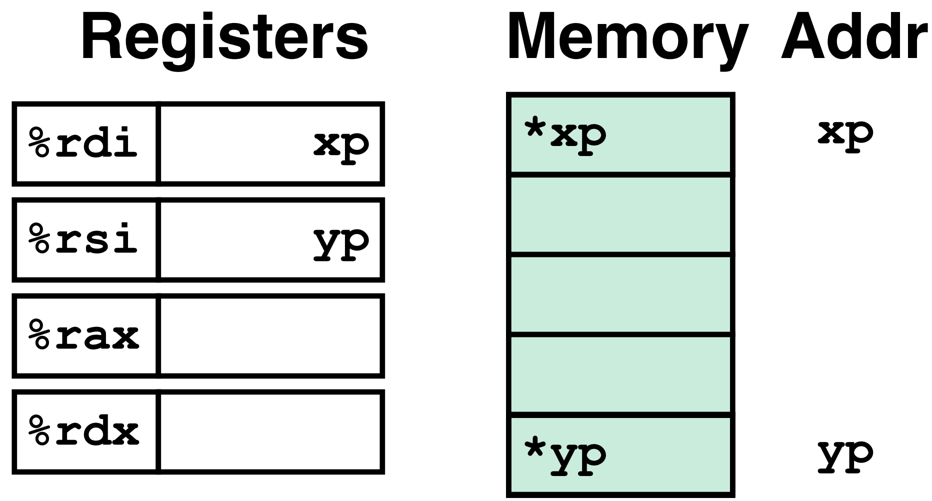
Cannot do memory-memory transfer with a single instruction in x86.

Example of Simple Addressing Modes

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

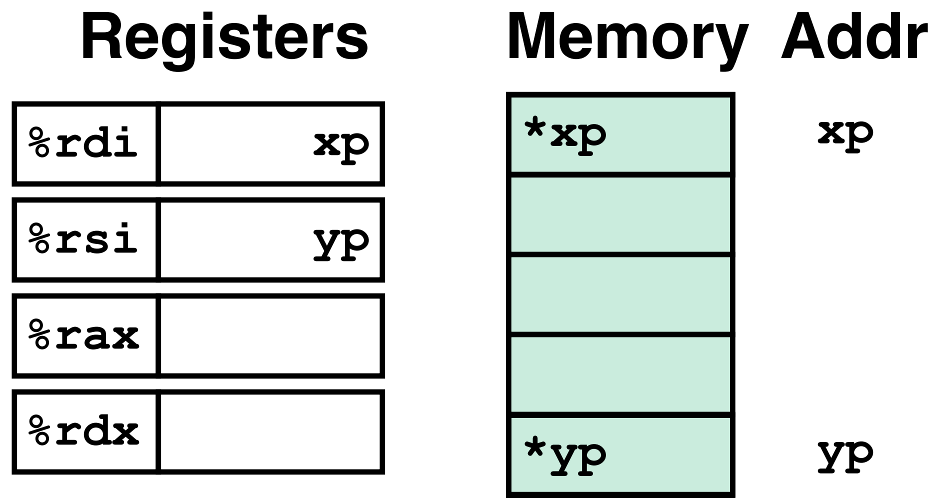
Example of Simple Addressing Modes

```
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()`

Registers

%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory / **Addr**

Memory	Addr	
123	0x120	xp
	0x118	
	0x110	
	0x108	
456	0x100	yp

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

Understanding Swap()

Registers

%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory **Addr**

Memory	Addr	
123	0x120	xp
	0x118	
	0x110	
	0x108	
456	0x100	yp

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

Understanding `Swap()`

Registers

`%rdi`	`0x120`
`%rsi`	`0x100`
`%rax`	`123`
`%rdx`	

Memory

Memory	Addr	
123	0x120	xp
	0x118	
	0x110	
	0x108	
456	0x100	yp

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

Understanding `Swap()`

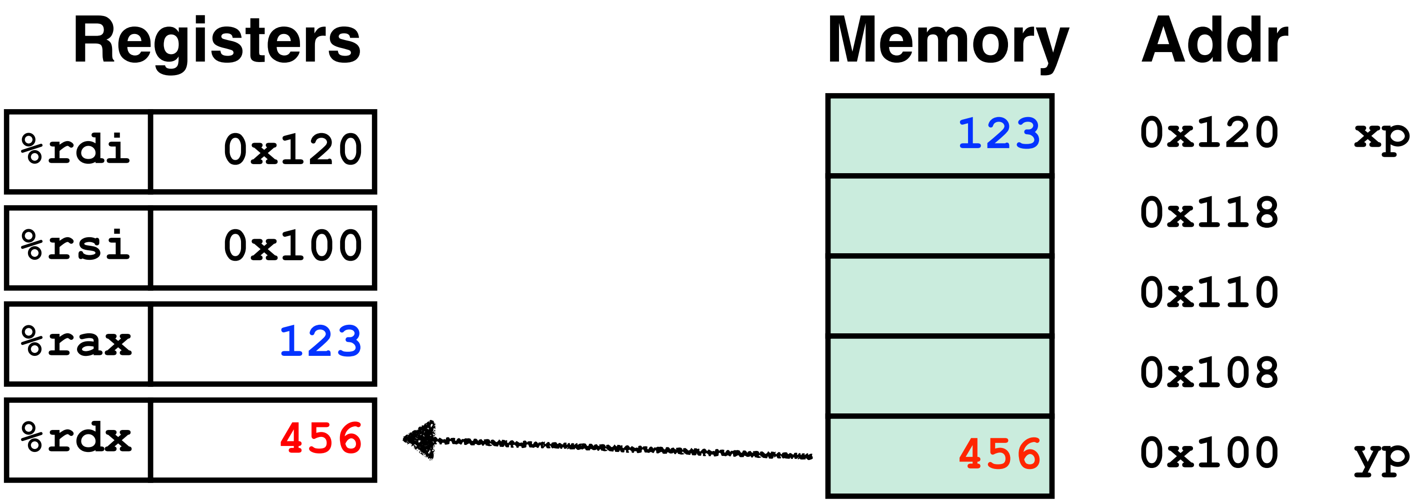
Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	

Memory	Addr	
123	0x120	xp
	0x118	
	0x110	
	0x108	
456	0x100	yp

```
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()`

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory / **Addr**

Memory	Addr	
123	0x120	xp
	0x118	
	0x110	
	0x108	
456	0x100	yp

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

Understanding `Swap()`

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory / **Addr**

Memory	Addr	
456	0x120	xp
	0x118	
	0x110	
	0x108	
456	0x100	yp

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

Understanding `Swap()`

Registers

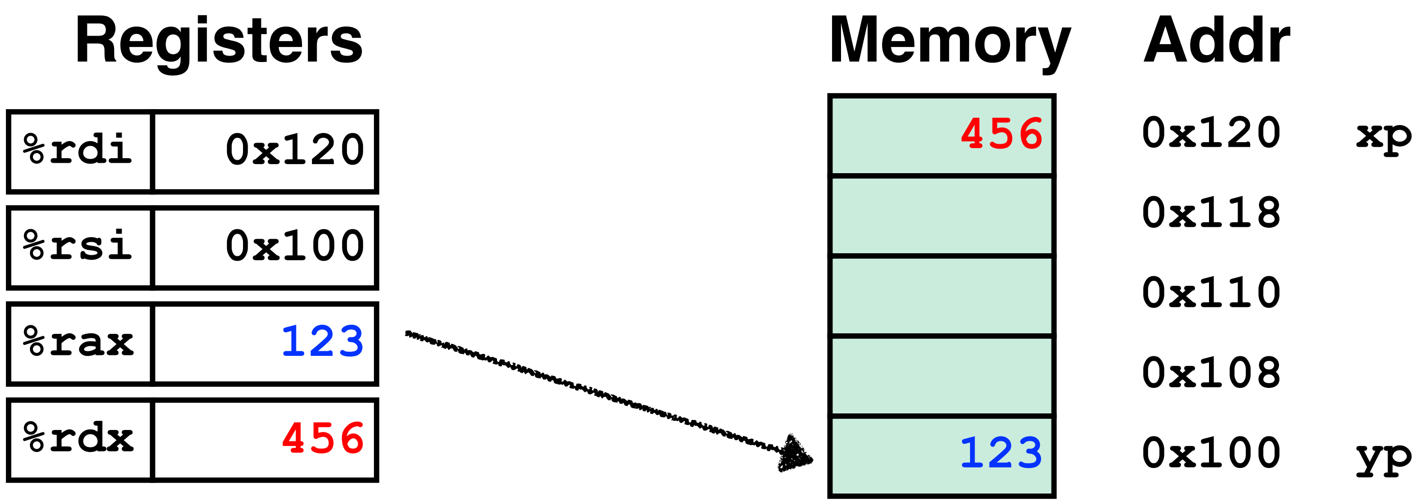
%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory **Addr**

Memory	Addr	
456	0x120	xp
	0x118	
	0x110	
	0x108	
456	0x100	yp

```
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
```

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`

$$\downarrow$$

%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
```

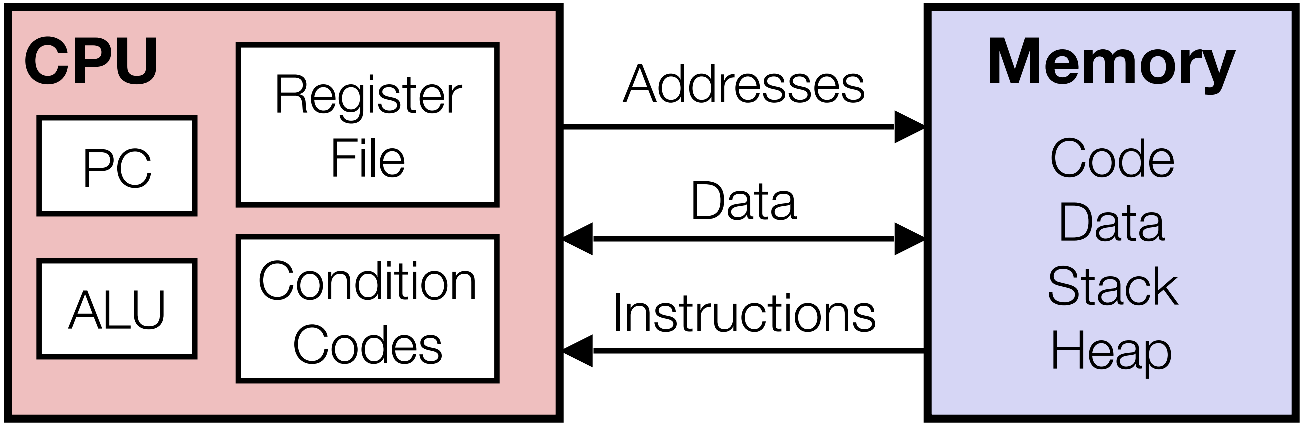
↓

%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];

Assembly Program Instructions

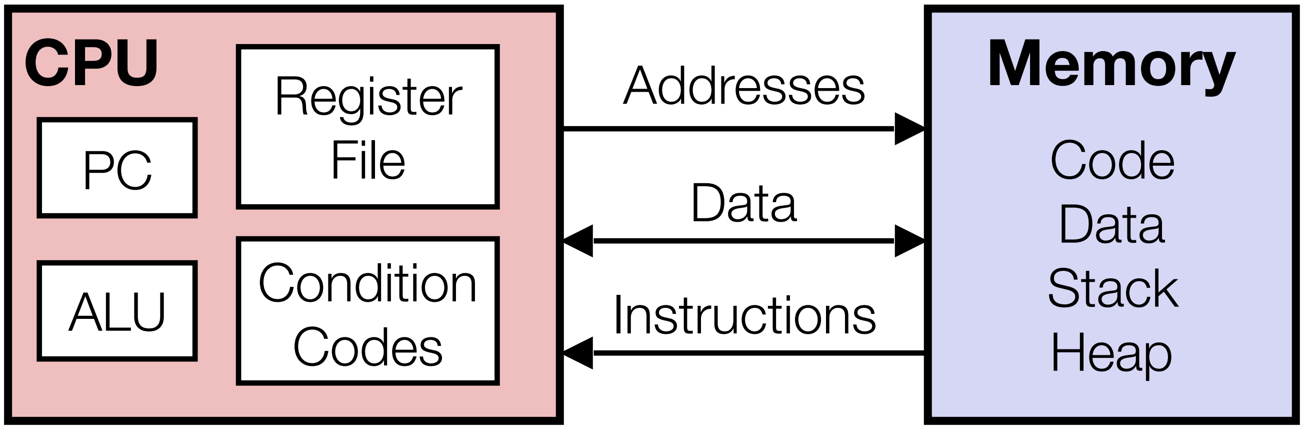
Assembly Programmer's Perspective of a Computer



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

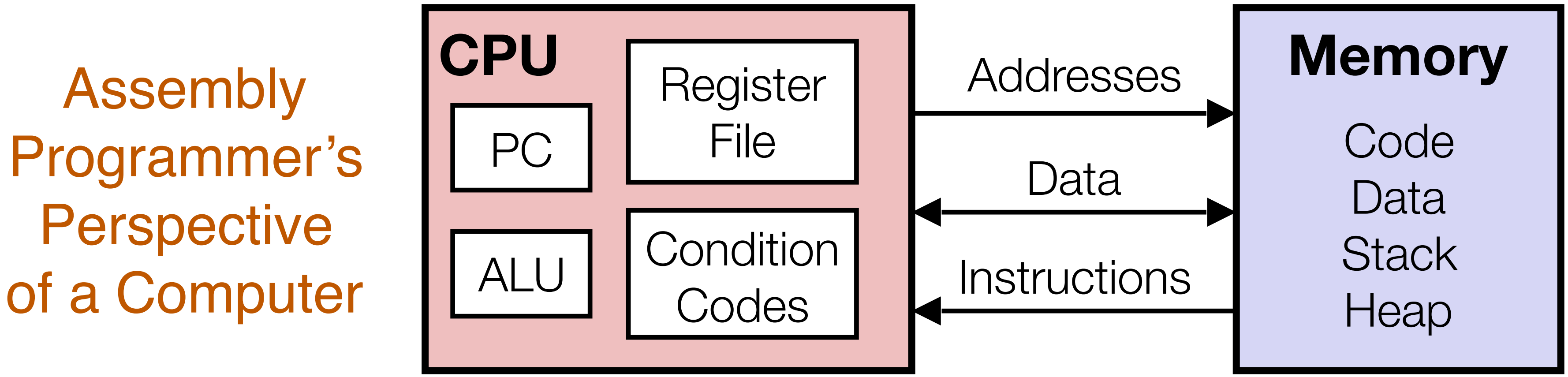
Assembly Program Instructions

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



- *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: Conditional 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	

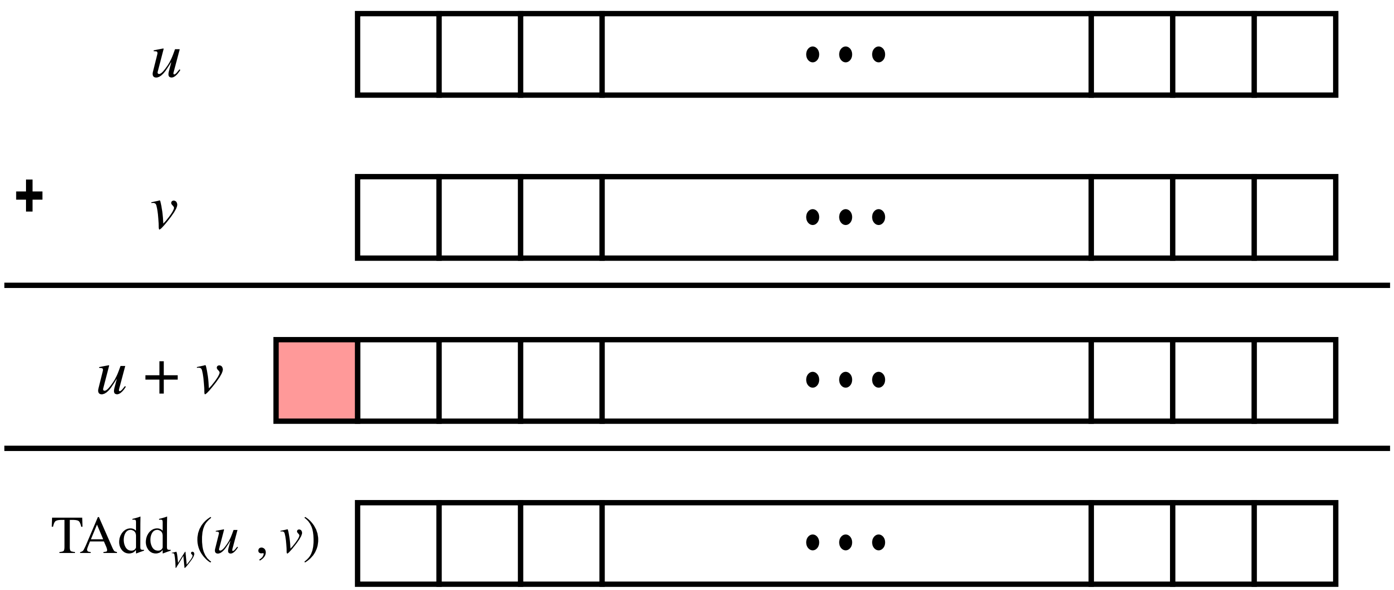
Some Arithmetic Operations (2 Operands)

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

`addq %rax, %rbx`

Some Arithmetic Operations (2 Operands)

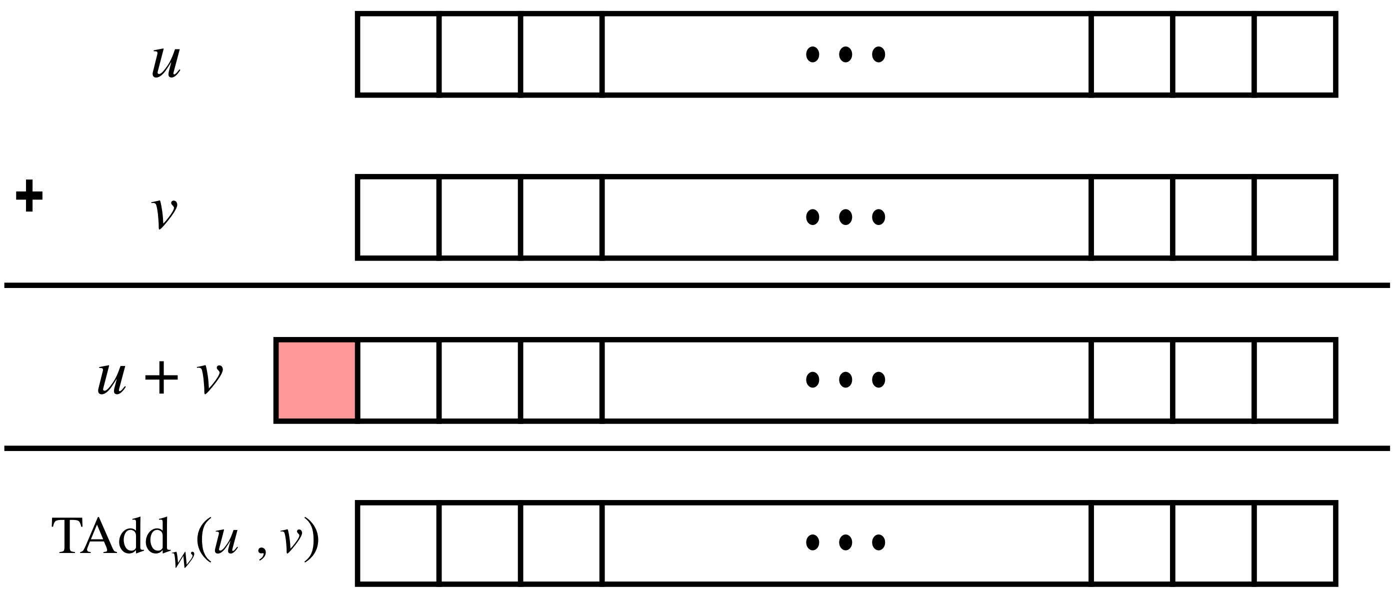
Format	Computation	Notes
addq `src, dest`	Dest = Dest + Src	



`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;
}
```

```
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
```

```
#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;
}
```

```
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
```

```
#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;
}
```

```
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
```

```
#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;
}
```

```
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
```

```
#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

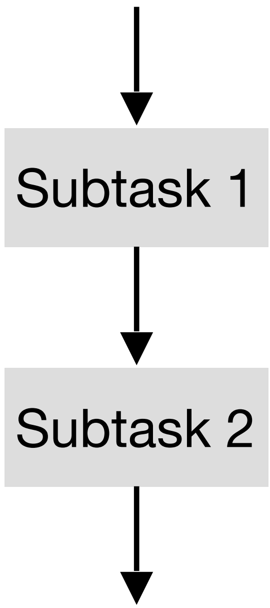
Today: Compute and Control Instructions

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

Three Basic Programming Constructs

Three Basic Programming Constructs

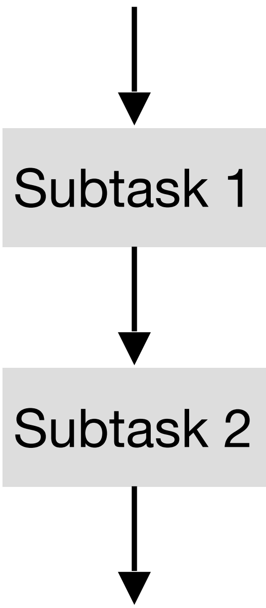
Sequential




```
a = x + y;
y = a - c;
...
```

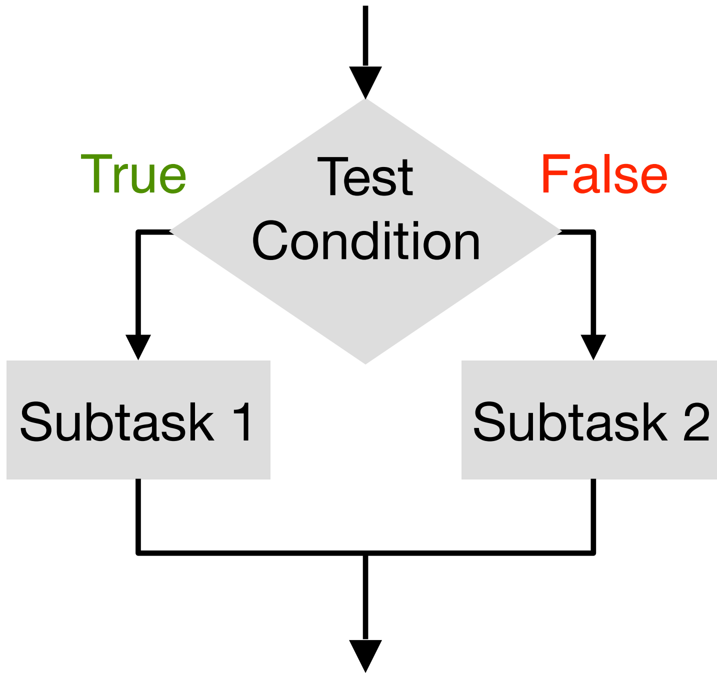
Three Basic Programming Constructs

Sequential




```
a = x + y;
y = a - c;
…
```

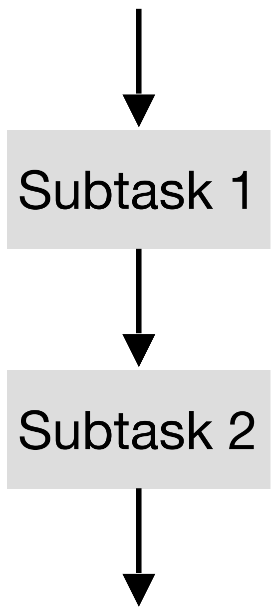
Conditional




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

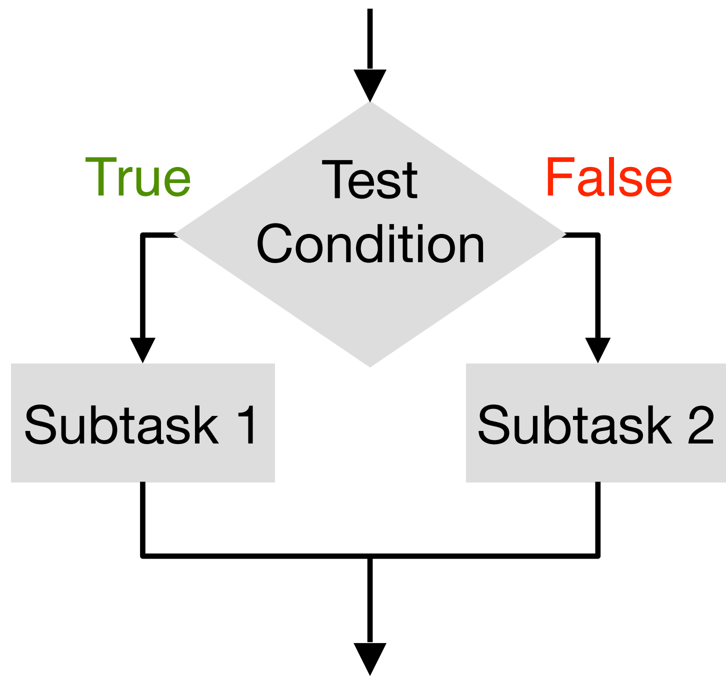
Three Basic Programming Constructs

Sequential



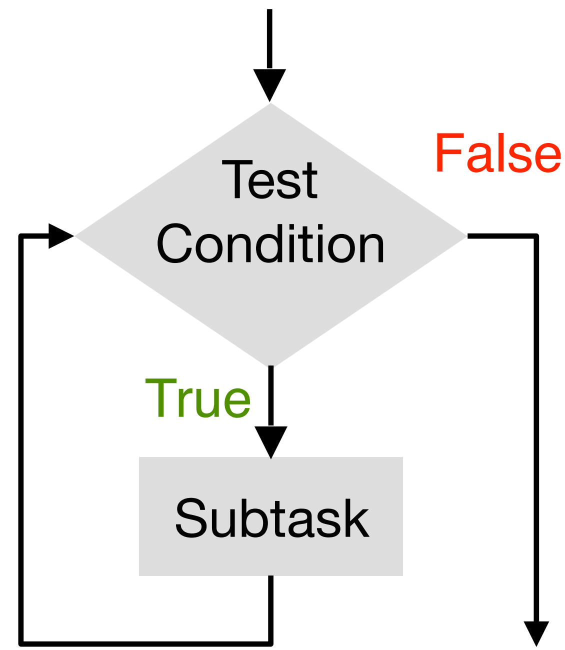

```
a = x + y;
y = a - c;
…
```

Conditional




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

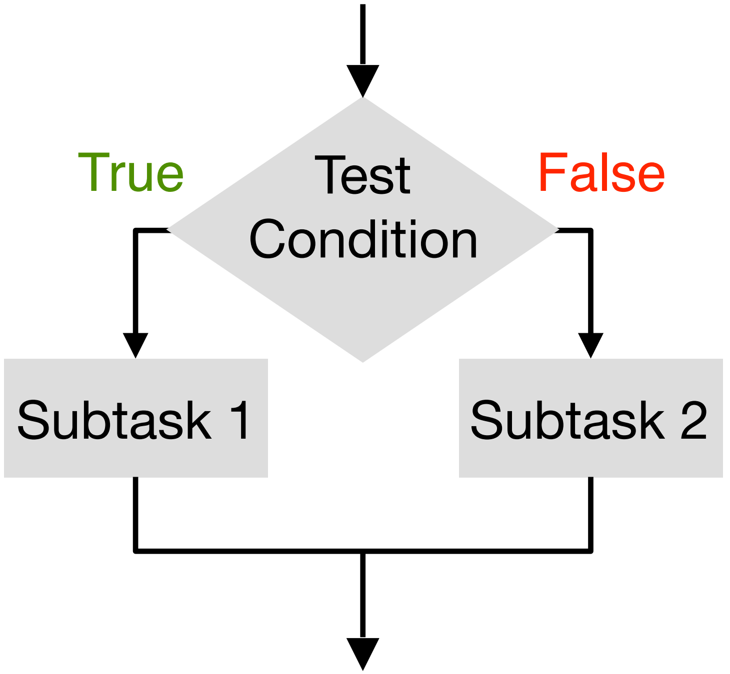
Iterative




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

Three Basic Programming Constructs

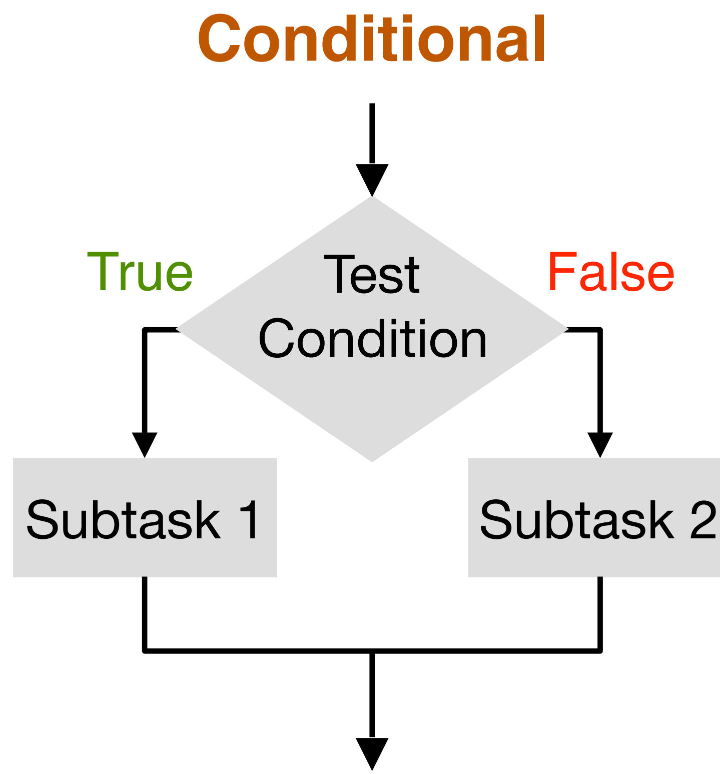
Conditional




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

Three Basic Programming Constructs

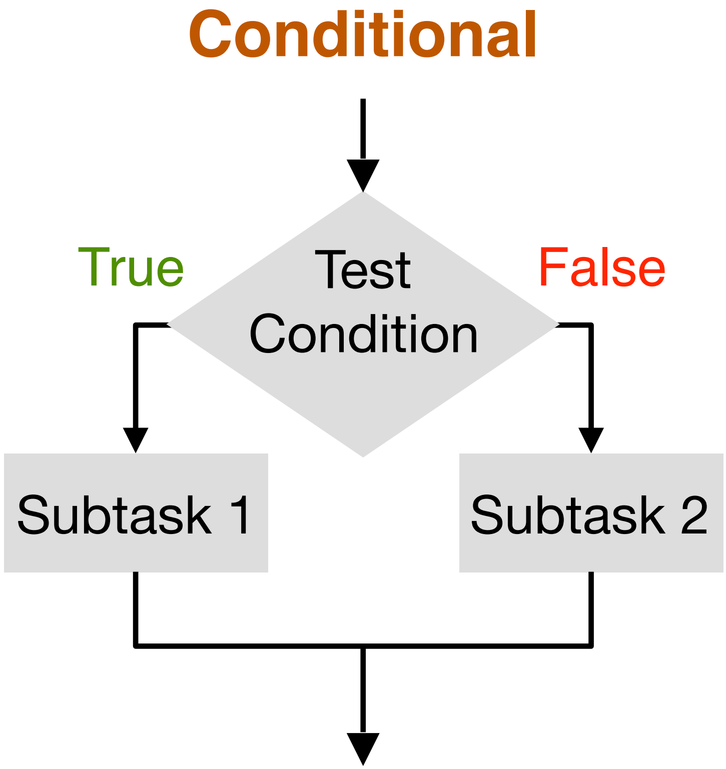
- Both conditional and iterative programming requires altering the sequence of instructions (control flow)




```
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

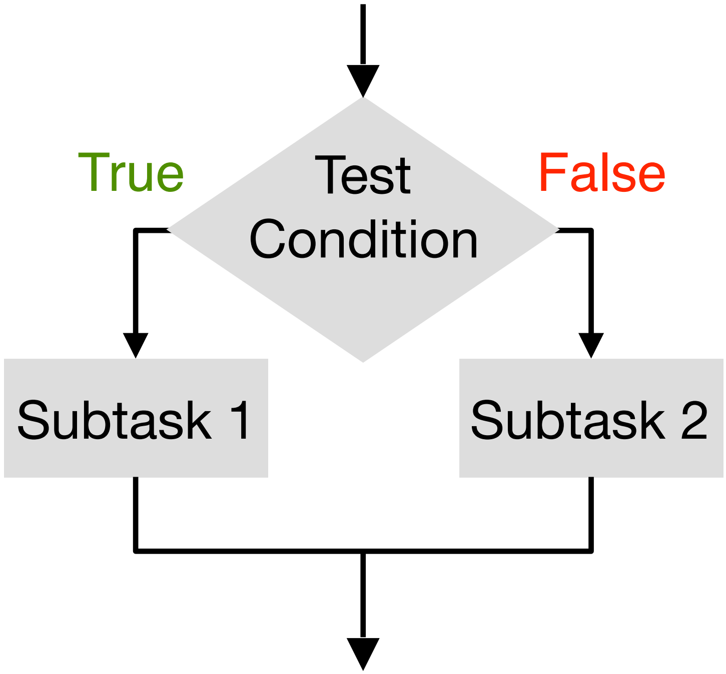



```
if (x > y) r = x - y;
else r = y - x;
```

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?

Conditional




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

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;
}
```

```
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
```

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

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
```

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

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`**

How Should `cmpq` Set Condition Codes?

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

How Should `cmpq` Set Condition Codes?

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

- Essentially, how do we know **`%rdi`** <= **`%rsi`**?

How Should `cmpq` Set Condition Codes?

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

- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`

How Should `cmpq` Set Condition Codes?

`cmpq      %rsi, %rdi`

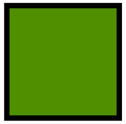
- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`

How Should `cmpq` Set Condition Codes?

`cmpq     %rsi, %rdi`

- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`

ZF Zero Flag (result is zero)



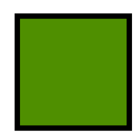
ZF

How Should `cmpq` Set Condition Codes?

`cmpq     %rsi, %rdi`

- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: `%rdi - %rsi < 0` (is it correct??)

ZF Zero Flag (result is zero)



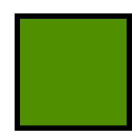
ZF

How Should `cmpq` Set Condition Codes?

`cmpq    %rsi, %rdi`

- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: ~~`%rdi - %rsi < 0` (is it correct??)~~
  - `%rdi` - `%rsi` < 0 and the result doesn't overflow, or

ZF Zero Flag (result is zero)



ZF

How Should `cmpq` Set Condition Codes?

`cmpq        %rsi, %rdi`

- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: ~~`%rdi - %rsi < 0` (is it correct??)~~
  - `%rdi` - `%rsi` < 0 and the result doesn't overflow, or

No Overflow

```
    001          1
-)  010     -)   2
-------     ------
    111         -1
```

ZF Zero Flag (result is zero)

ZF

How Should `cmpq` Set Condition Codes?

`cmpq %rsi, %rdi`

- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: ~~`%rdi - %rsi < 0` (is it correct??)~~
  - `%rdi` - `%rsi` < 0 and the result doesn't overflow, or

No Overflow

```
    001        1
-)  010    -)  2
-------    -----
    111       -1
```

Overflow

```
    101       -3
-)  011    -)  3
-------    -----
    010       -6
```

ZF Zero Flag (result is zero)

ZF

How Should `cmpq` Set Condition Codes?

`cmpq    %rsi, %rdi`

- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%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

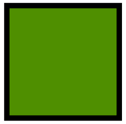
No Overflow

```
    001        1
-)  010    -)  2
-------    -----
    111       -1
```

Overflow

```
    101       -3
-)  011    -)  3
-------    -----
    010       -6
```

ZF Zero Flag (result is zero)



ZF

How Should `cmpq` Set Condition Codes?

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

- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%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

No Overflow

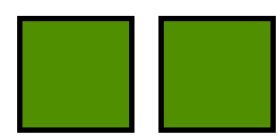
```
    001        1
-)  010    -)  2
-------    -----
    111       -1
```

Overflow

```
    101       -3
-)  011    -)  3
-------    -----
    010       -6
```

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)



ZF **SF**

How Should `cmpq` Set Condition Codes?

`cmpq %rsi, %rdi`

- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%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

No Overflow

```
    001          1
-)  010     -)   2
    ---        ---
    111         -1
```

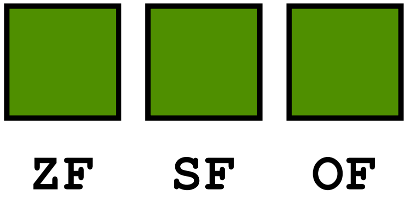
Overflow

```
    101         -3
-)  011     -)   3
    ---        ---
    010         -6
```

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`

- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%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

11111111 10000000

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

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (result overflow)

0	0	0
ZF	**SF**	**OF**

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

`cmpq      %rsi, %rdi`

- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%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

11111111   10000000

`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)

ZF	SF	OF
0	0	0

How Should `cmpq` Set Condition Codes?

`cmpq %rsi, %rdi`

- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%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

11111111 10000000

`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)

ZF	SF	OF
0	1	0

How Should `cmpq` Set Condition Codes?

`cmpq %rsi, %rdi`

- Essentially, how do we know **`%rdi`** <= **`%rsi`**?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%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

> - `%rdi <= %rsi` if and only if
>   - ZF is set, or
>   - SF is set but OF is not set, or
>   - SF is not set, but OF is set
> - or simply: ZF | (SF ^ OF)

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)

OF Overflow Flag (result overflow)

0	1	0
ZF	**SF**	**OF**

Conditional Branch Example

```
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
```

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

0	0	0
ZF	SF	OF

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)

0	0	0
ZF	**SF**	**OF**

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits
- How about OF? How do we know A-B leads to overflow (A and B are treated as signed)

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits
- How about OF? How do we know A-B leads to overflow (A and B are treated as signed)
  - If $A < 0$ & $B > 0$, but the result > 0, or

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits
- How about OF? How do we know A-B leads to overflow (A and B are treated as signed)
  - If $A < 0$ & $B > 0$, but the result > 0, or
  - If $A > 0$ & $B < 0$, but the result < 0

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits
- How about OF? How do we know A-B leads to overflow (A and B are treated as signed)
  - If A < 0 & B > 0, but the result > 0, or
  - If A > 0 & B < 0, but the result < 0

No Overflow

```
    001          1
-)  010     -)   2
-------     ------
    111         -1
```

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits
- How about OF? How do we know A-B leads to overflow (A and B are treated as signed)
  - If A < 0 & B > 0, but the result > 0, or
  - If A > 0 & B < 0, but the result < 0

No Overflow

```
    001          1
-)  010     -)   2
-------     ------
    111         -1
```

Overflow

```
    101         -3
-)  011     -)   3
-------     ------
    010          2
```

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits
- How about OF? How do we know A-B leads to overflow (A and B are treated as signed)
  - If A < 0 & B > 0, but the result > 0, or
  - If A > 0 & B < 0, but the result < 0

No Overflow

```
    001         1
-)  010    -)   2
-------    ------
    111        -1
```

Overflow

```
    101        -3         011         3
-)  011    -)   3     -)  100    -)  -4
-------    ------     -------    ------
    010         2         111        -1
```

How Does the Hardware Check Overflow?

- ZF and SF are easily set by just examining the bits
- How about OF? How do we know A-B leads to overflow (A and B are treated as signed)
  - If $A < 0$ & $B > 0$, but the result > 0, or
  - If $A > 0$ & $B < 0$, but the result < 0
  - So again, just have to check the bits

No Overflow

```
     001          1
-)   010     -)   2
--------     ------
     111         -1
```

Overflow

```
     101         -3          011          3
-)   011     -)   3     -)   100     -)  -4
--------     ------     --------     ------
     010          2          111         -1
```