

CSC 252: Computer Organization Fall 2021: Lecture 6

Assembly Programming:

- Data movement
- Compute
- Control

Instructor: Alan Beadle

Department of Computer Science
University of Rochester

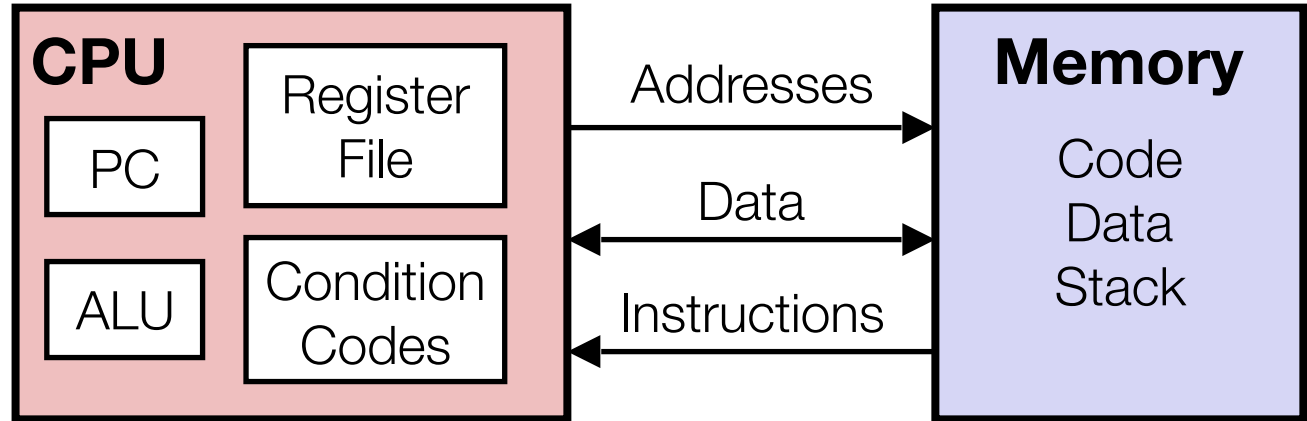
Announcements

A1 is over unless you are using slip days (should have told us already)

A2 is out (Due 9/30)

Instruction Processing Sequence

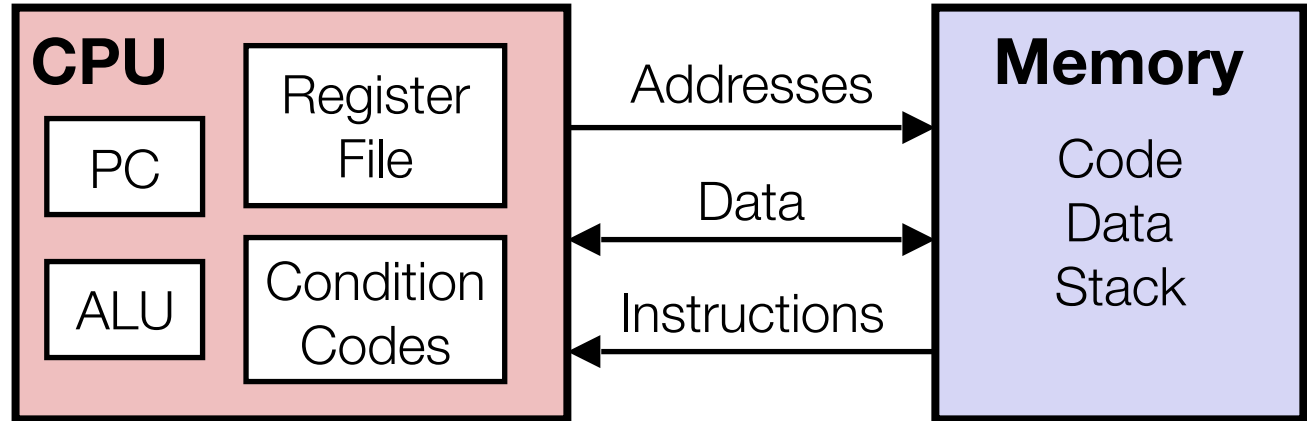
Assembly
Programmer's
Perspective
of a Computer



Fetch Instruction
(According to PC)

Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer

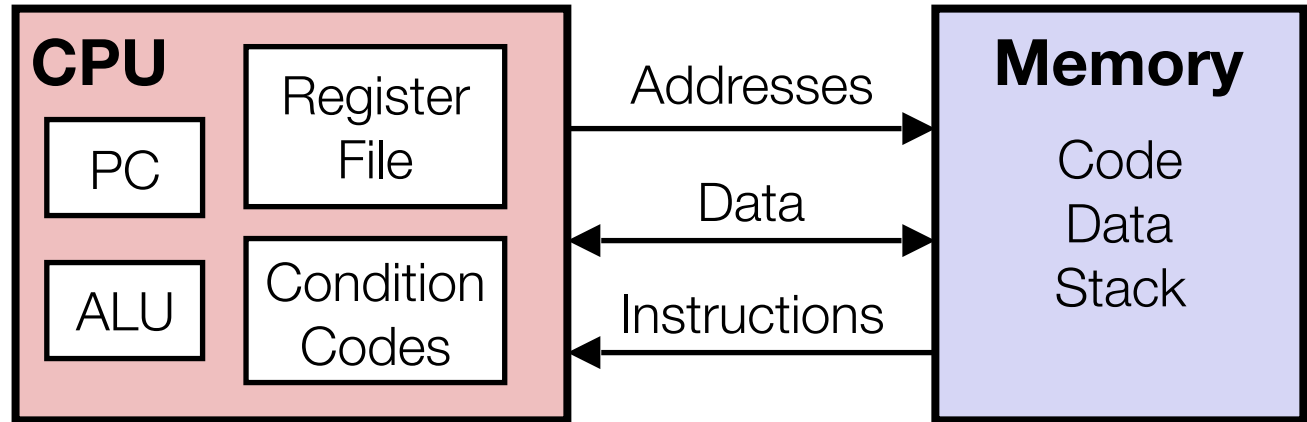


Fetch Instruction
(According to PC)

0x4801d8

Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer

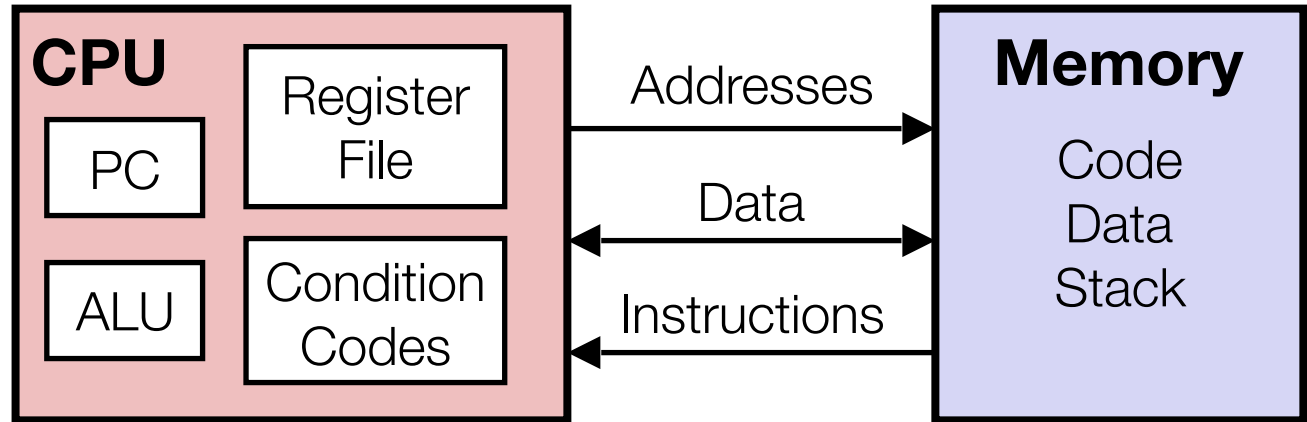


Fetch Instruction (According to PC) → Decode Instruction

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

Instruction Processing Sequence

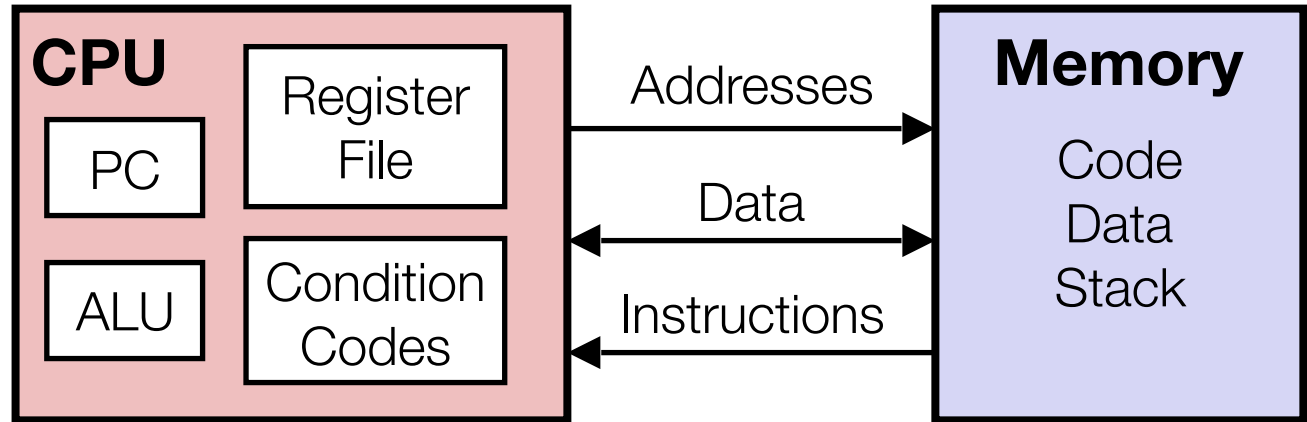
Assembly
Programmer's
Perspective
of a Computer



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

Instruction Processing Sequence

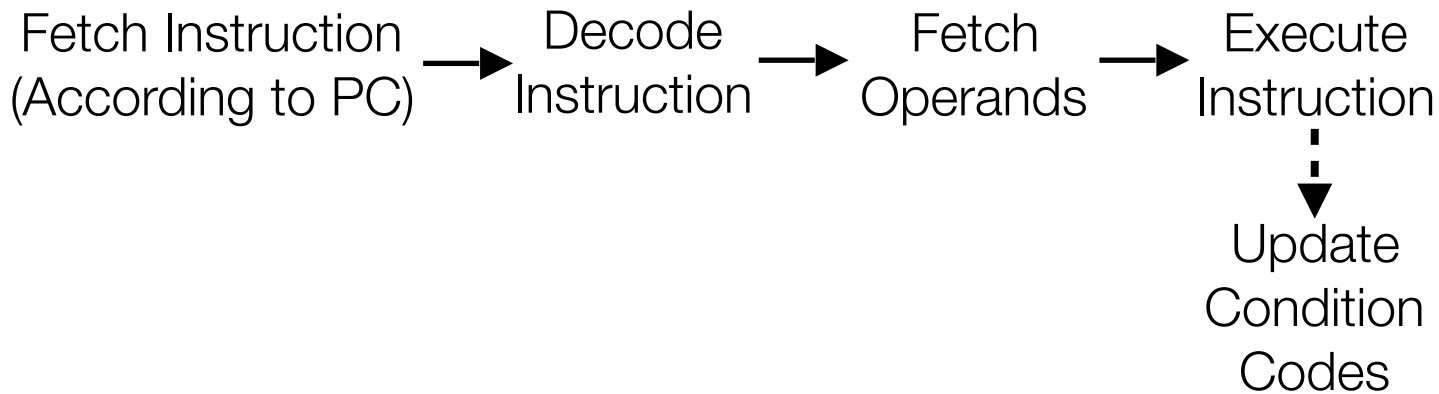
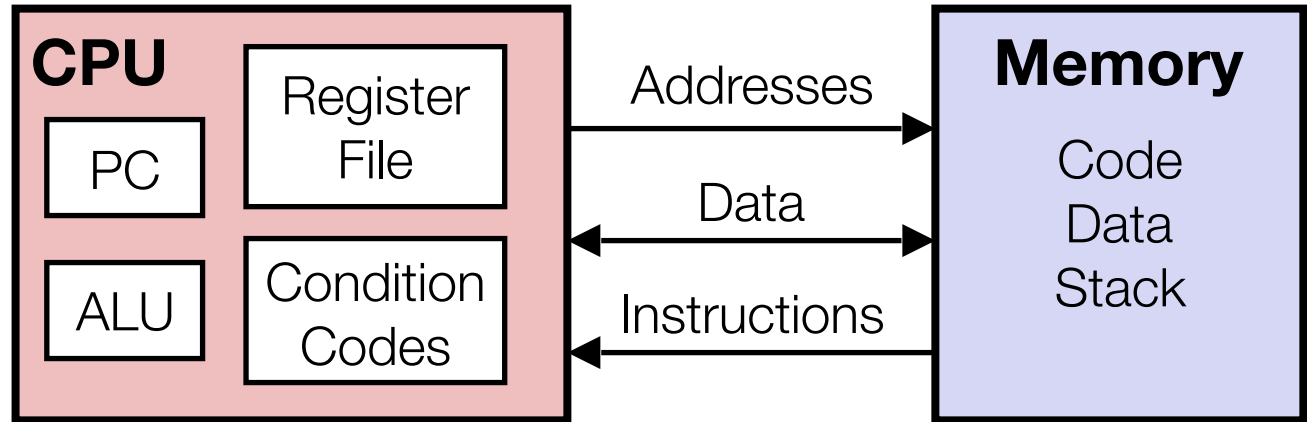
Assembly
Programmer's
Perspective
of a Computer



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

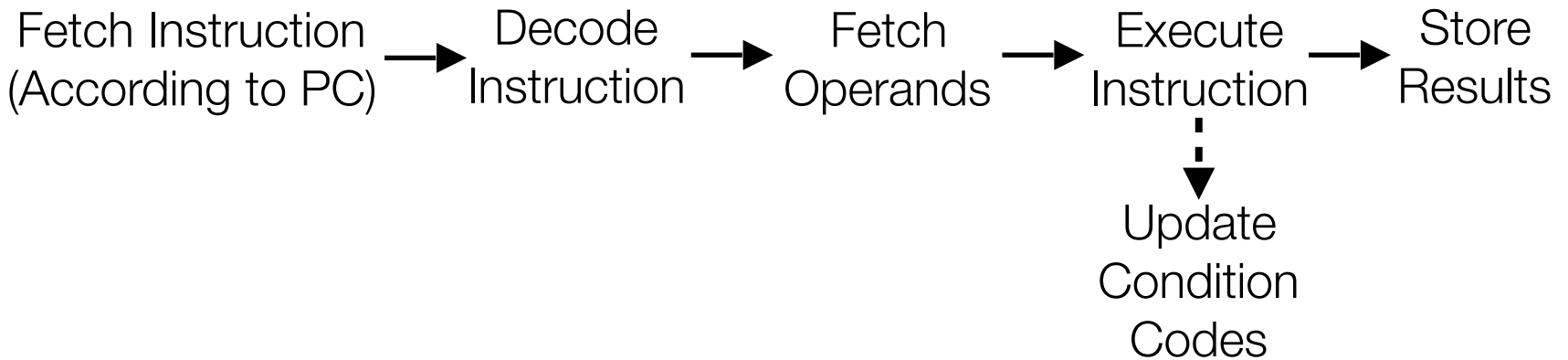
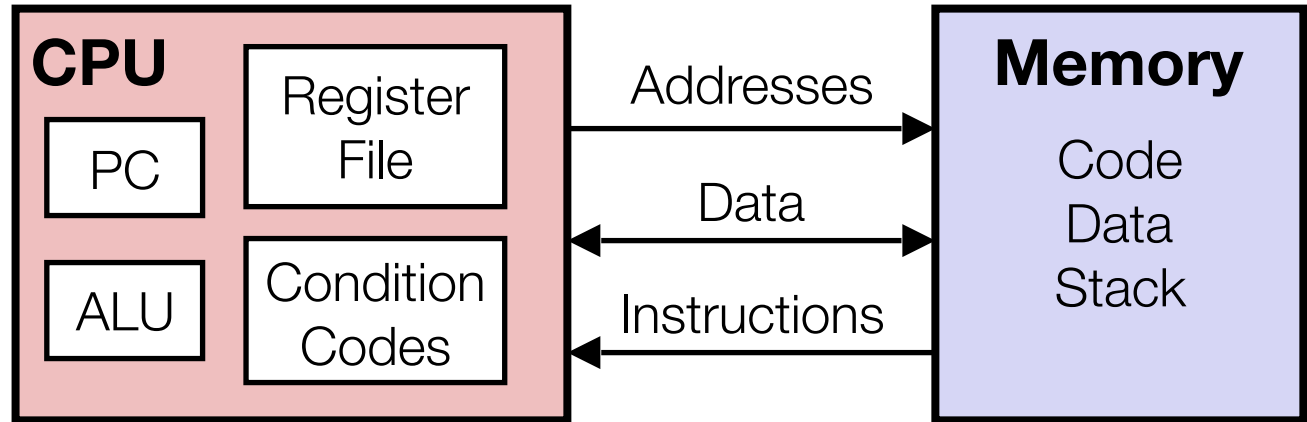
Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



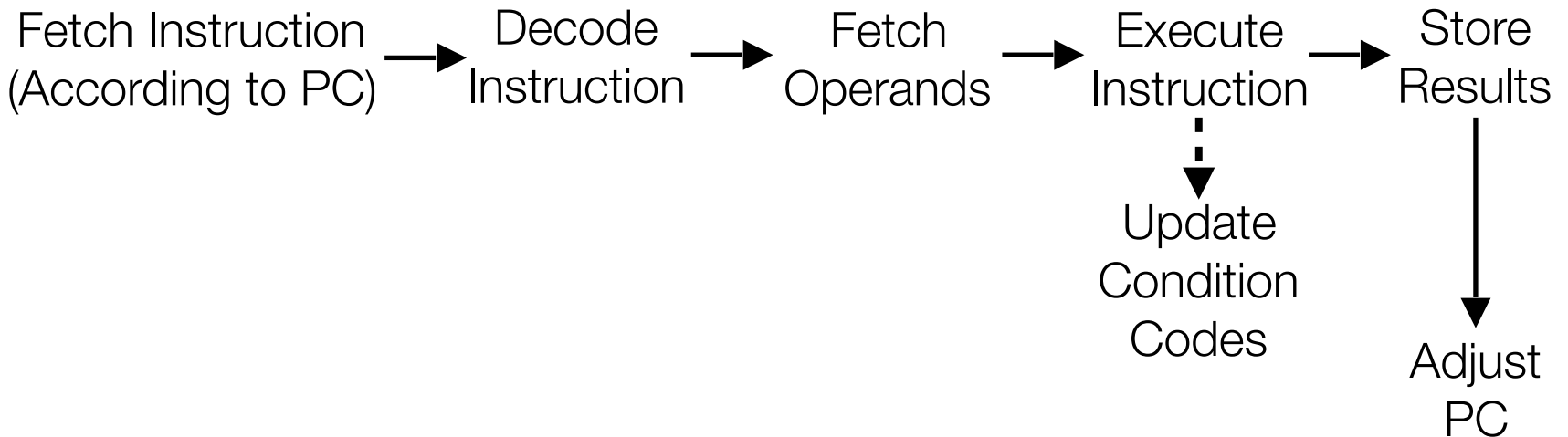
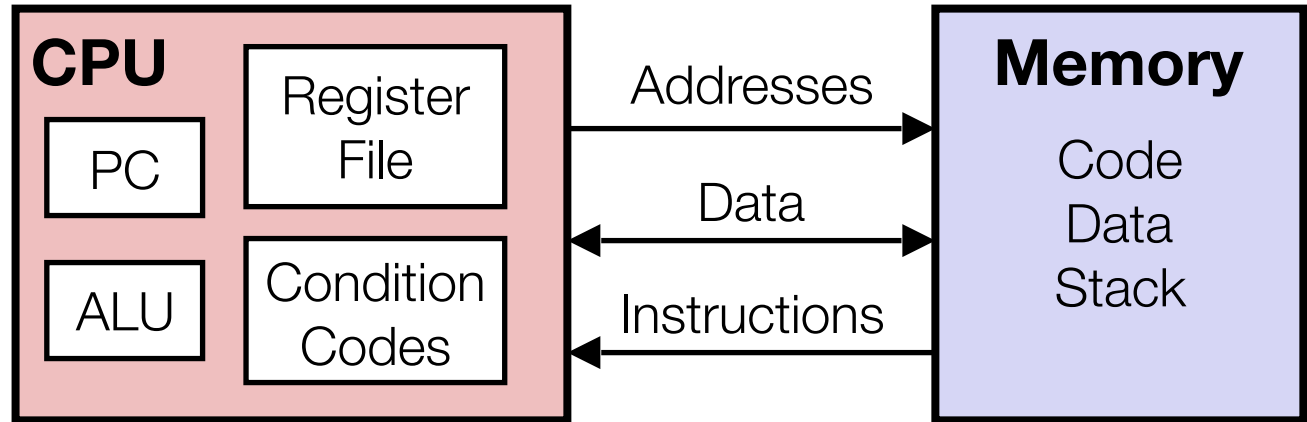
Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



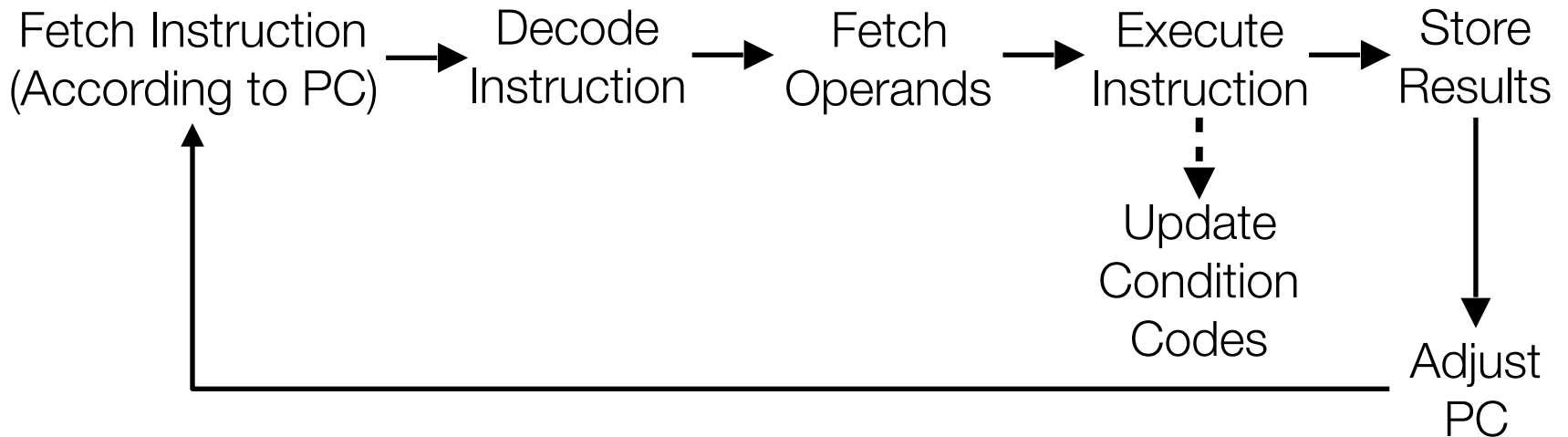
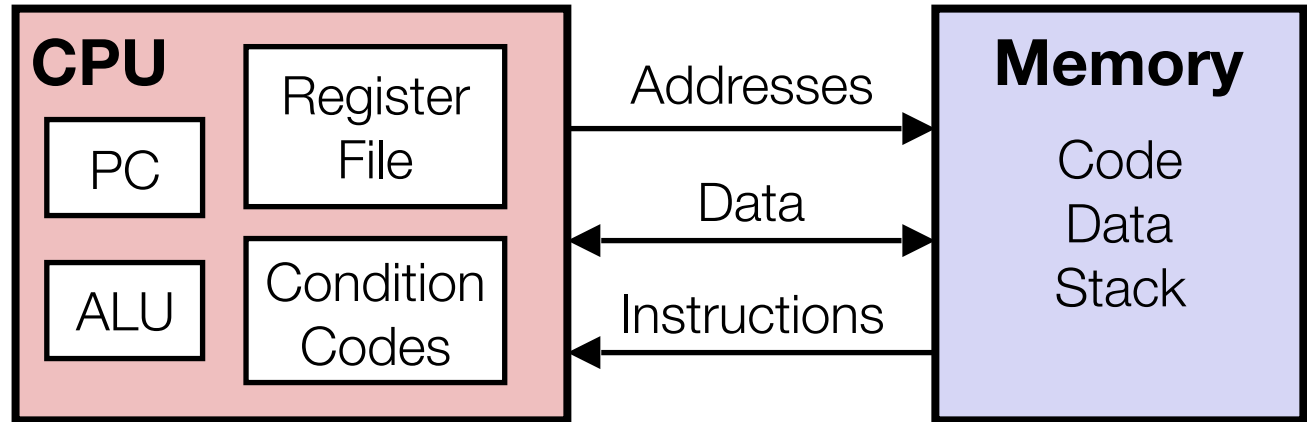
Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



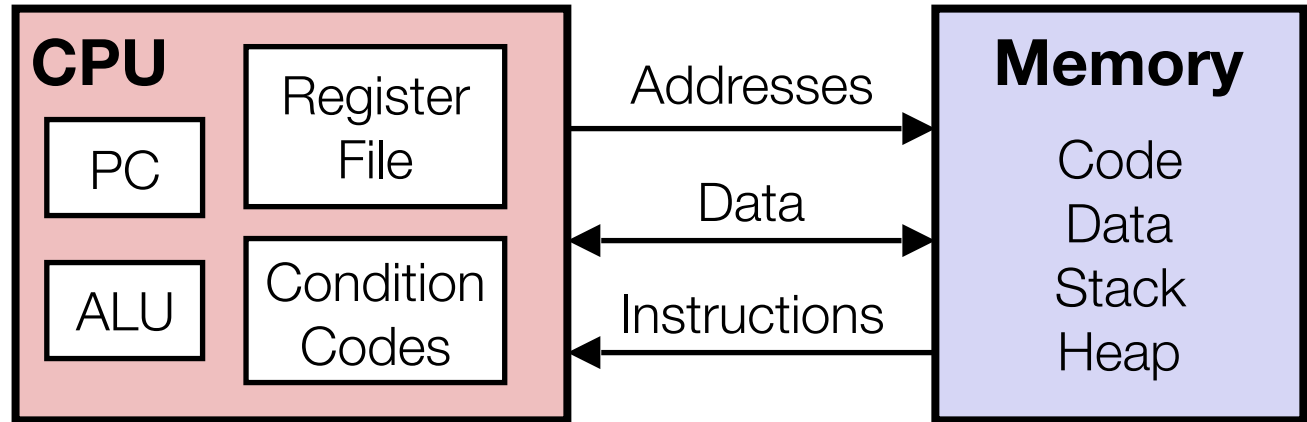
Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



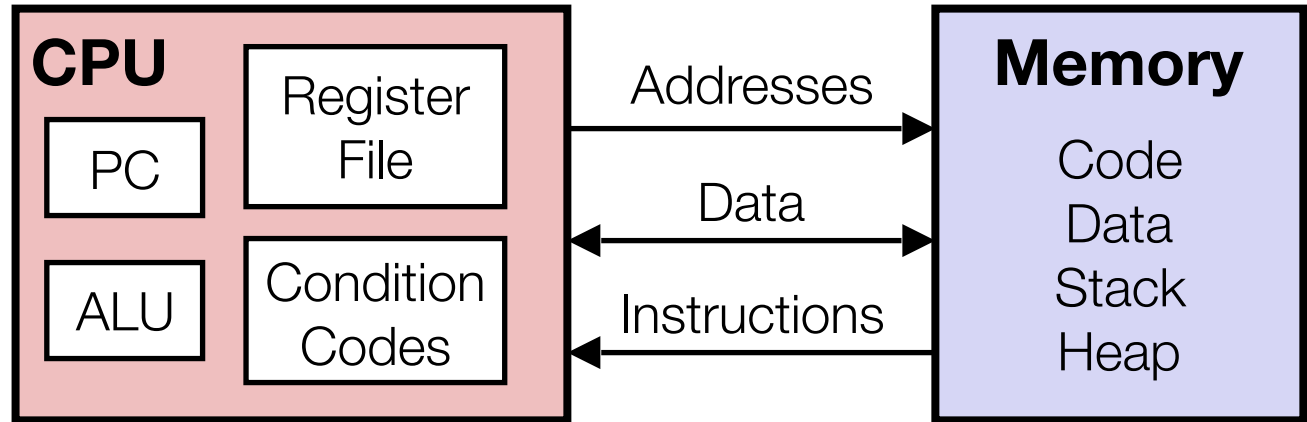
Assembly Program Instructions

Assembly
Programmer's
Perspective
of a Computer



Assembly Program Instructions

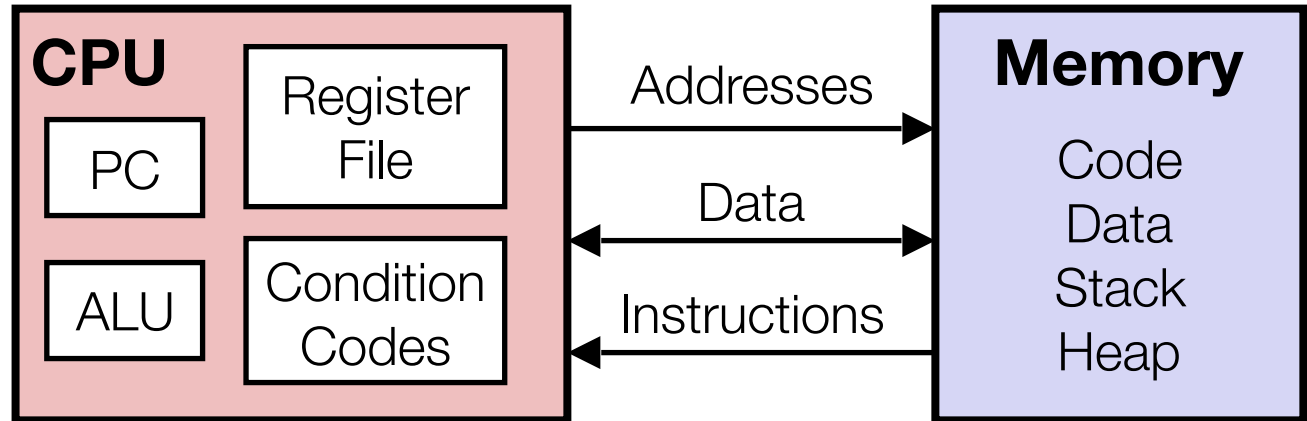
Assembly
Programmer's
Perspective
of a Computer



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

Assembly Program Instructions

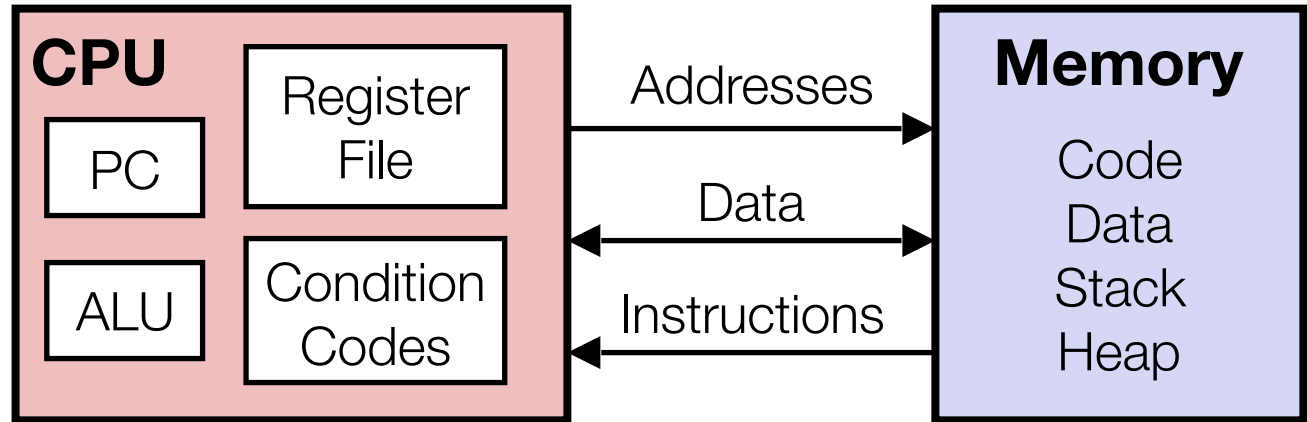
Assembly Programmer's Perspective of a Computer



- *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



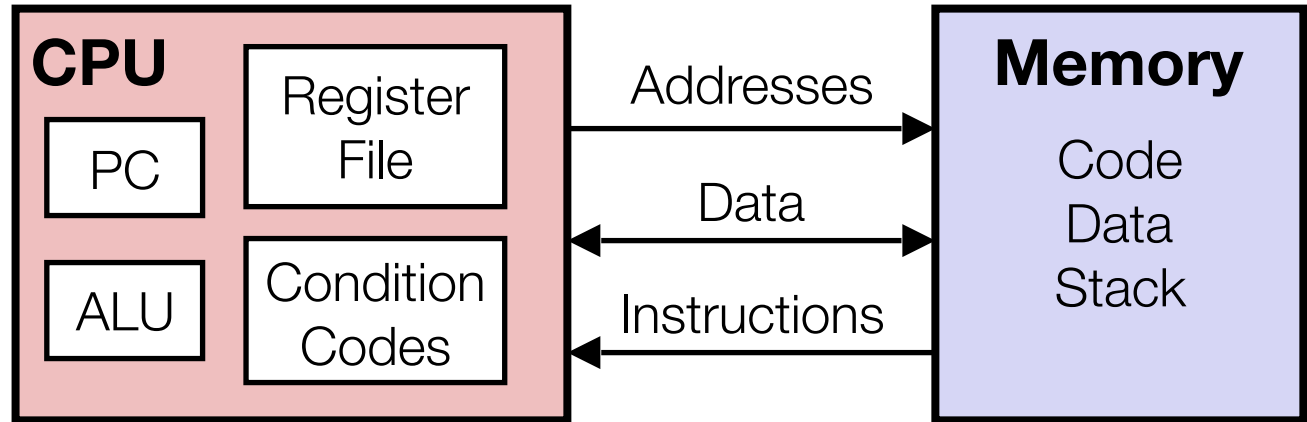
- **Compute Instruction**: Perform arithmetics on register or memory data
 - `addq %eax, %ebx`
 - C constructs: +, -, >>, etc.
- **Data Movement Instruction**: Transfer data between memory and register
 - `movq %eax, (%ebx)`
- **Control Instruction**: Alter the sequence of instructions (by changing PC)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

Today: Compute and Control Instructions

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

Data Movement in Processors

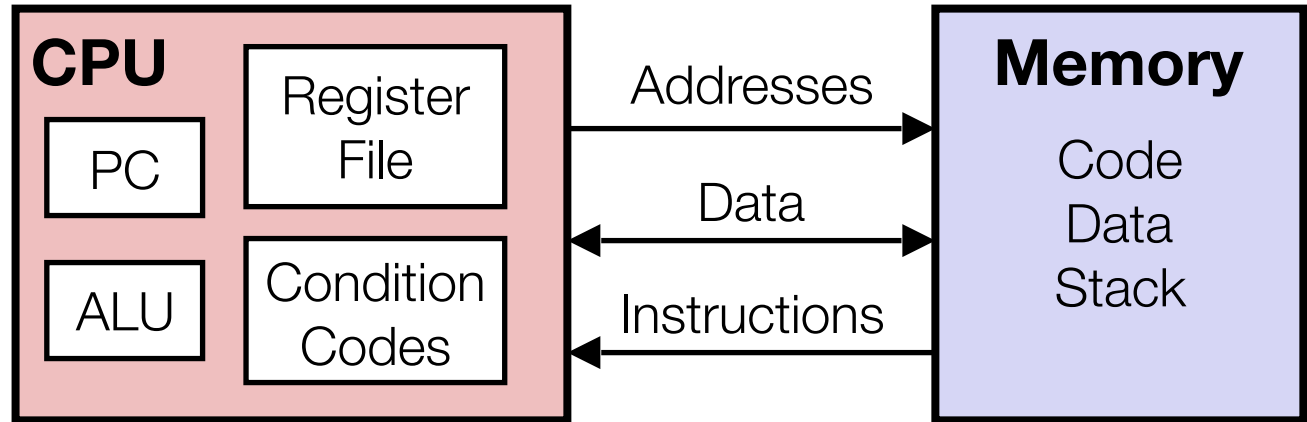
Assembly
Programmer's
Perspective
of a Computer



- Initially all data is in the memory

Data Movement in Processors

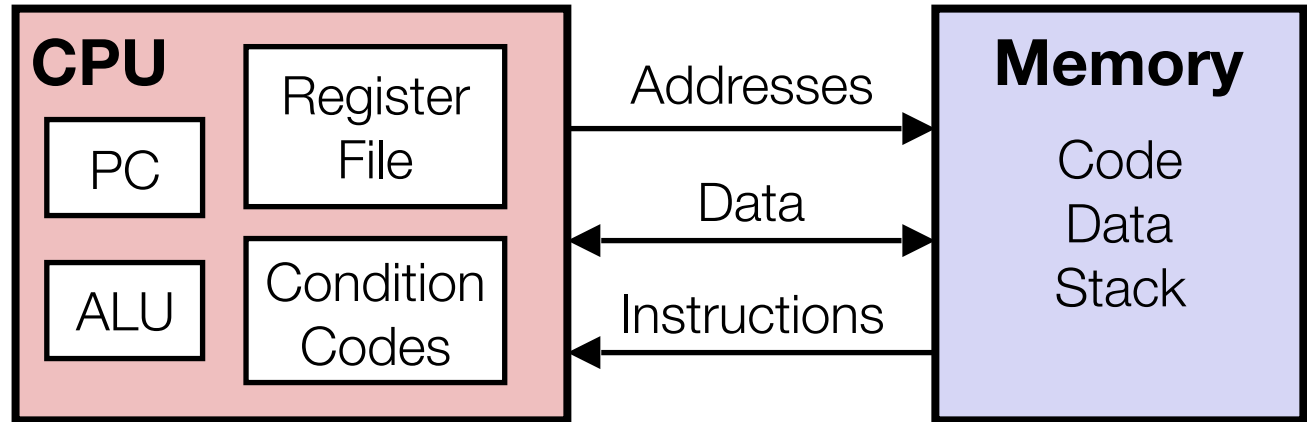
Assembly
Programmer's
Perspective
of a Computer



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

Data Movement in Processors

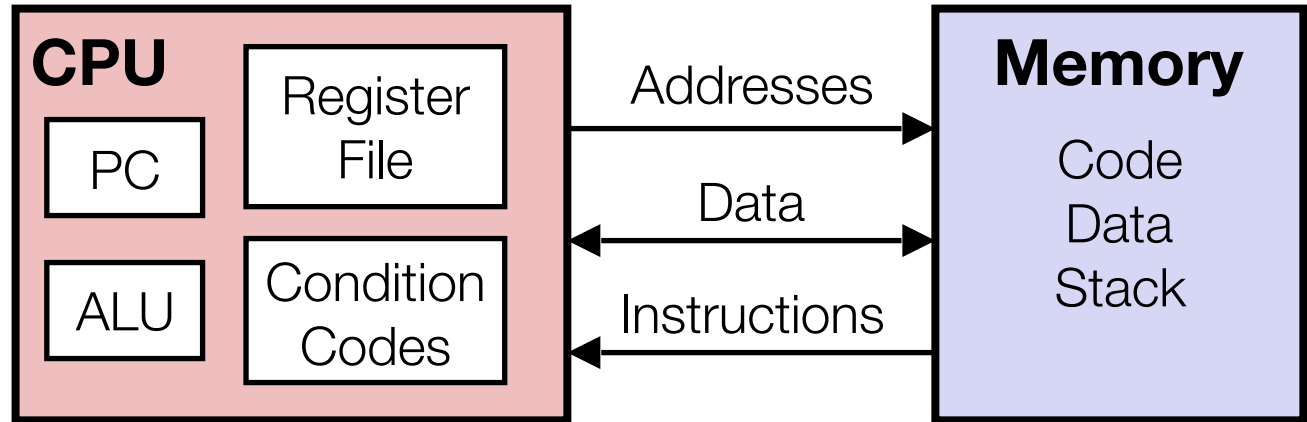
Assembly
Programmer's
Perspective
of a Computer



- 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

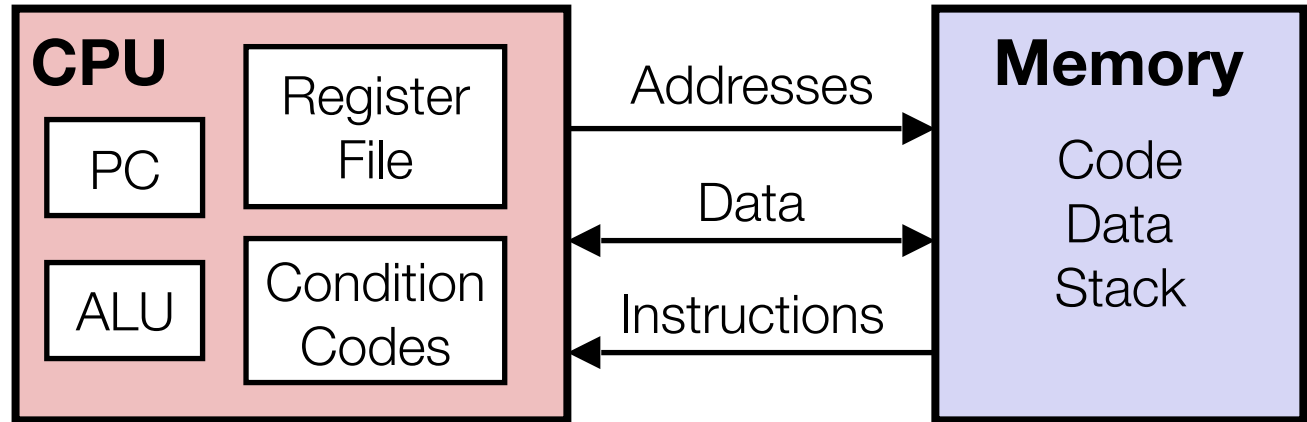
Assembly Programmer's Perspective of a Computer



- 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

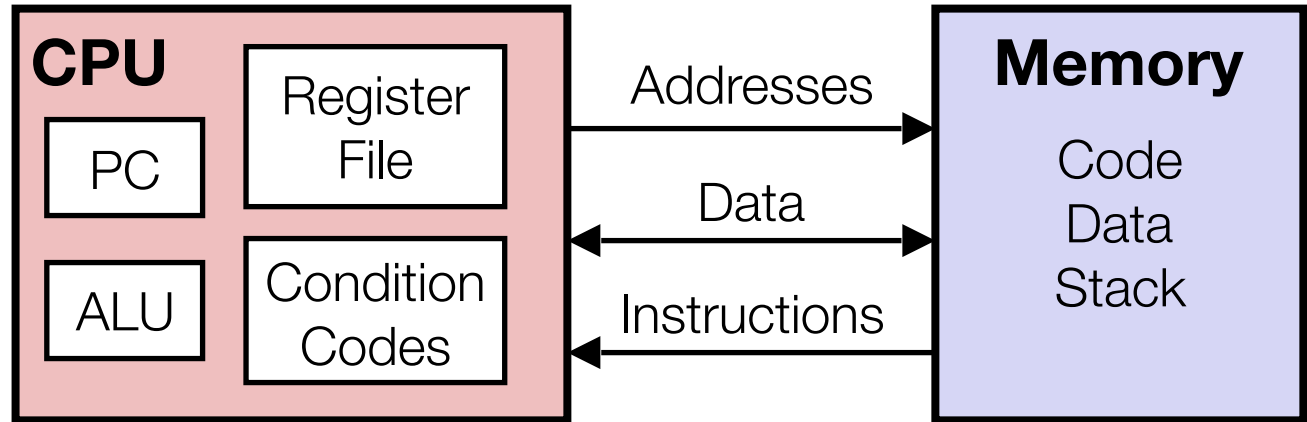
Assembly Programmer's Perspective of a Computer



- 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.

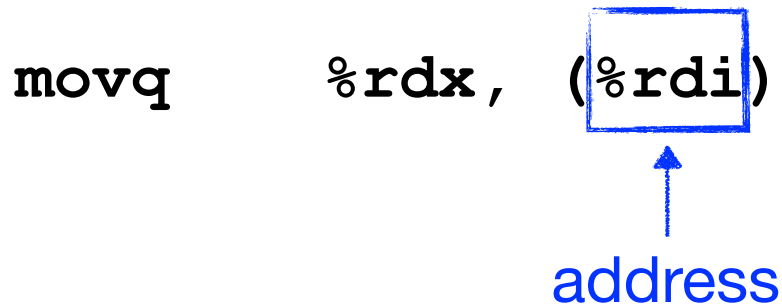
Data Movement Instruction Example

```
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

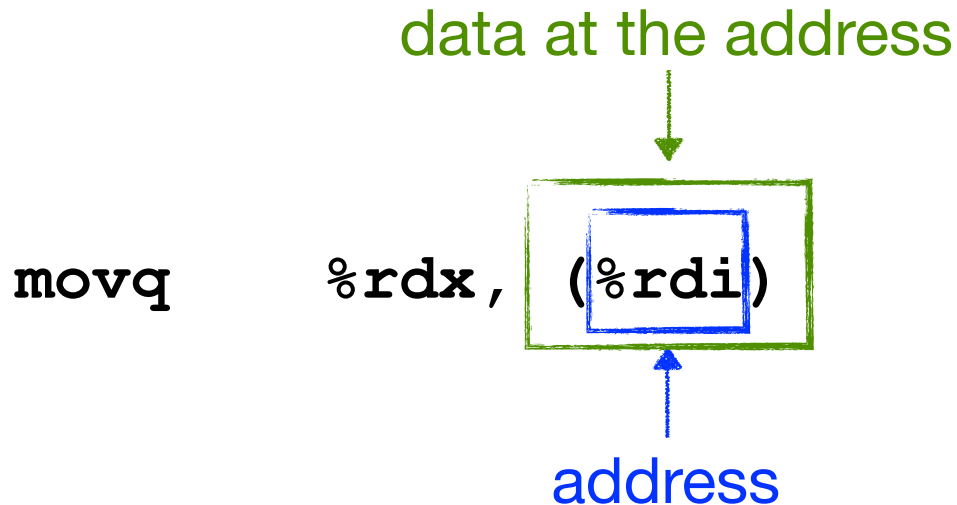
`movq %rdx, (%rdi)`



address

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

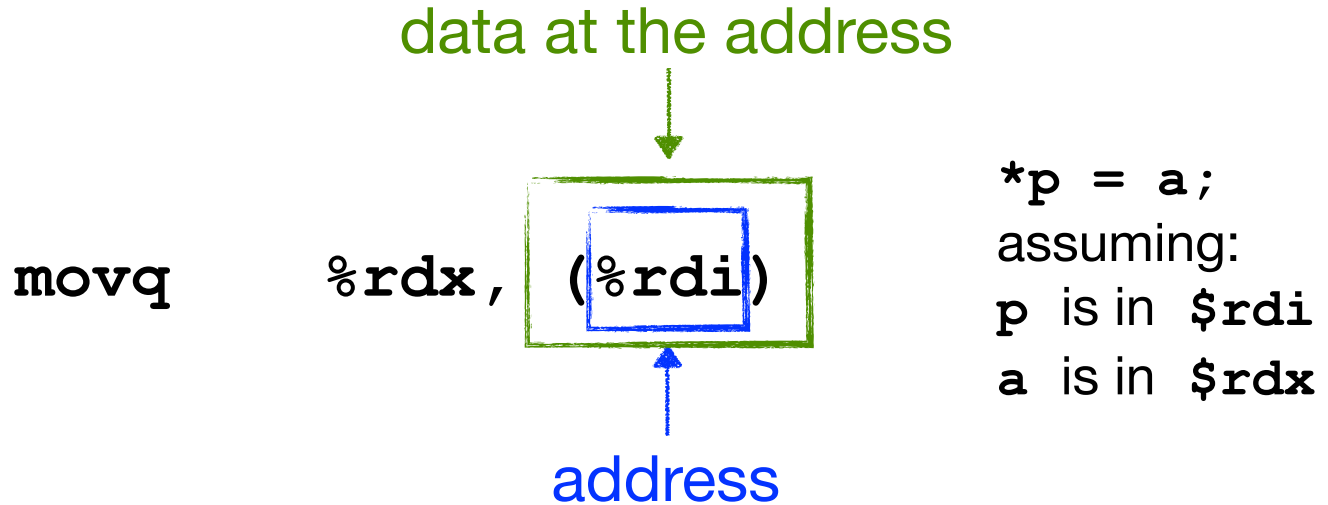
Data Movement Instruction Example



- 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

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

`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

	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

	Source	Dest	Example	C Analog
movq	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;
		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;
}
```

Registers

%rdi	xp
%rsi	yp
%rax	
%rdx	

Memory Addr

*xp	xp
*yp	yp

Example of Simple Addressing Modes

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

Registers

%rdi	xp
%rsi	yp
%rax	
%rdx	

Memory Addr

*xp	xp
*yp	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

<code>%rdi</code>	<code>0x120</code>
<code>%rsi</code>	<code>0x100</code>
<code>%rax</code>	
<code>%rdx</code>	

Memory

123
456

Addr

<code>0x120</code>	<code>xp</code>
<code>0x118</code>	
<code>0x110</code>	
<code>0x108</code>	
<code>0x100</code>	<code>yp</code>

`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

<code>%rdi</code>	<code>0x120</code>
<code>%rsi</code>	<code>0x100</code>
<code>%rax</code>	
<code>%rdx</code>	

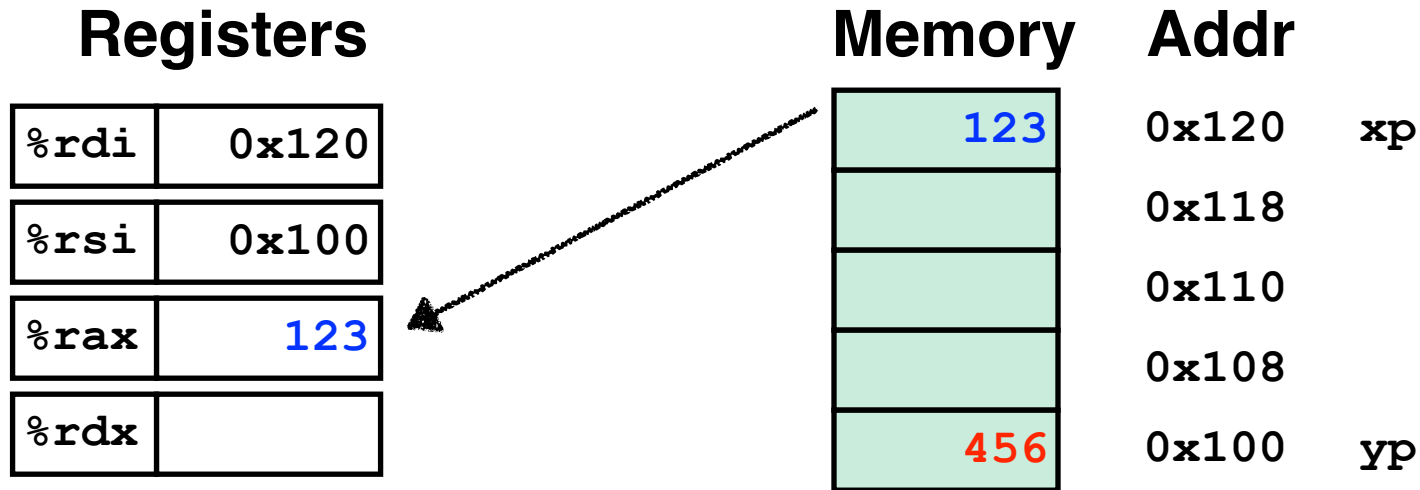
Memory

Memory	Addr
123	<code>0x120</code> <code>xp</code>
	<code>0x118</code>
	<code>0x110</code>
	<code>0x108</code>
456	<code>0x100</code> <code>yp</code>

`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

<code>%rdi</code>	<code>0x120</code>
<code>%rsi</code>	<code>0x100</code>
<code>%rax</code>	<code>123</code>
<code>%rdx</code>	

Memory

<code>123</code>
<code>456</code>

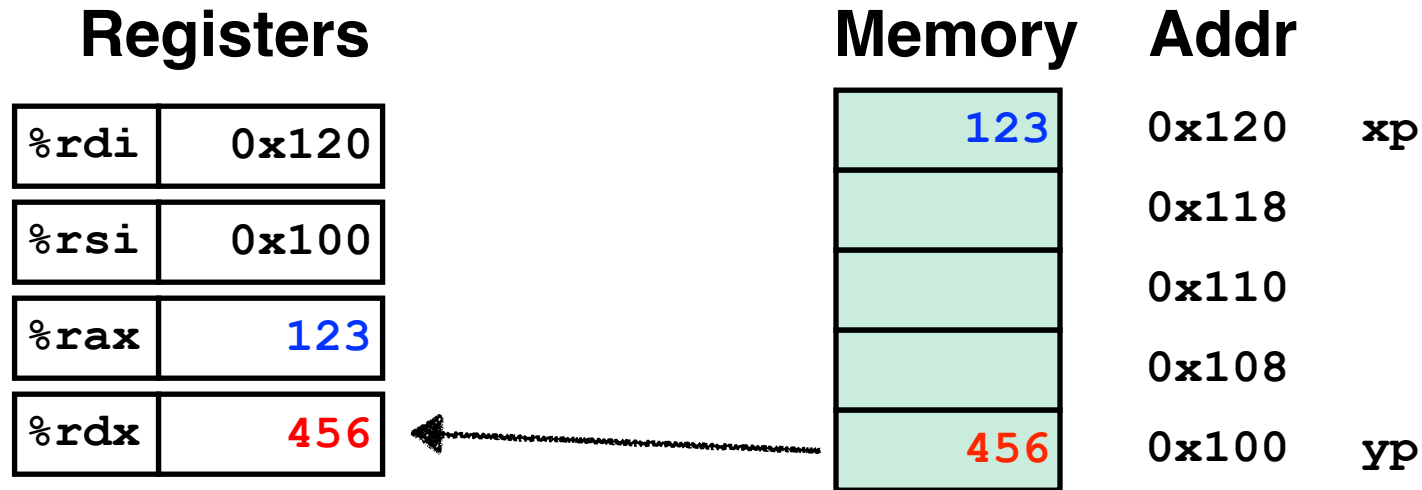
Addr

<code>0x120</code>	<code>xp</code>
<code>0x118</code>	
<code>0x110</code>	
<code>0x108</code>	
<code>0x100</code>	<code>yp</code>

`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

<code>%rdi</code>	<code>0x120</code>
<code>%rsi</code>	<code>0x100</code>
<code>%rax</code>	<code>123</code>
<code>%rdx</code>	<code>456</code>

Memory

Memory	Addr
<code>123</code>	<code>0x120</code> <code>xp</code>
	<code>0x118</code>
	<code>0x110</code>
	<code>0x108</code>
<code>456</code>	<code>0x100</code> <code>yp</code>

`swap:`

```
movq    (%rdi), %rax    # t0 = *xp
```

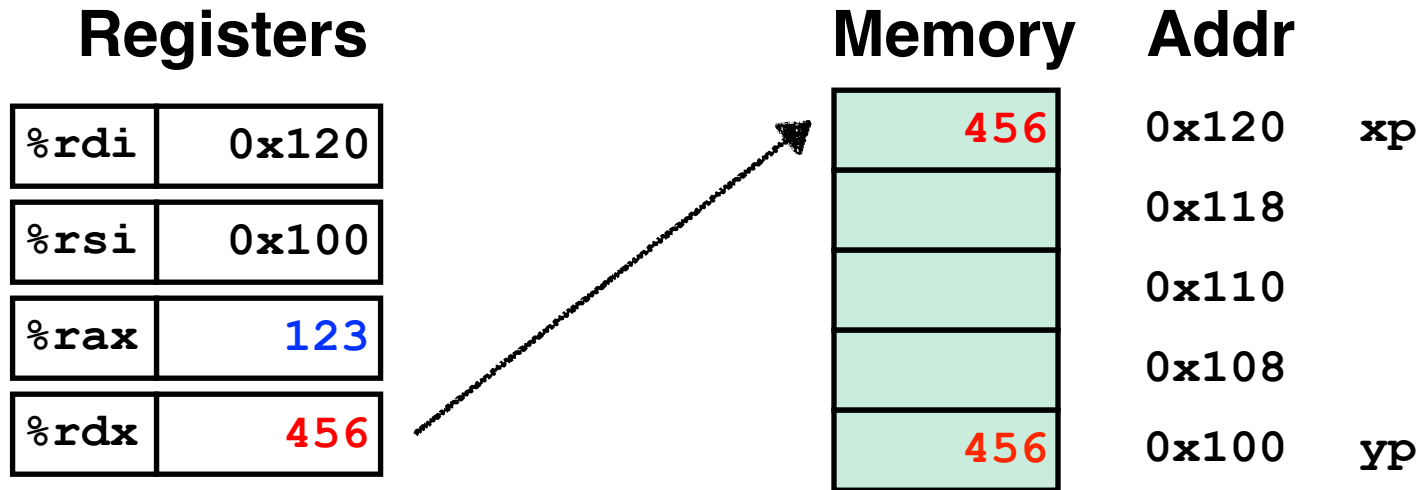
```
movq    (%rsi), %rdx    # t1 = *yp
```

```
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

<code>%rdi</code>	<code>0x120</code>
<code>%rsi</code>	<code>0x100</code>
<code>%rax</code>	<code>123</code>
<code>%rdx</code>	<code>456</code>

Memory

Memory	Addr
<code>456</code>	<code>0x120</code> <code>xp</code>
	<code>0x118</code>
	<code>0x110</code>
	<code>0x108</code>
<code>456</code>	<code>0x100</code> <code>yp</code>

`swap:`

```
movq    (%rdi), %rax    # t0 = *xp
```

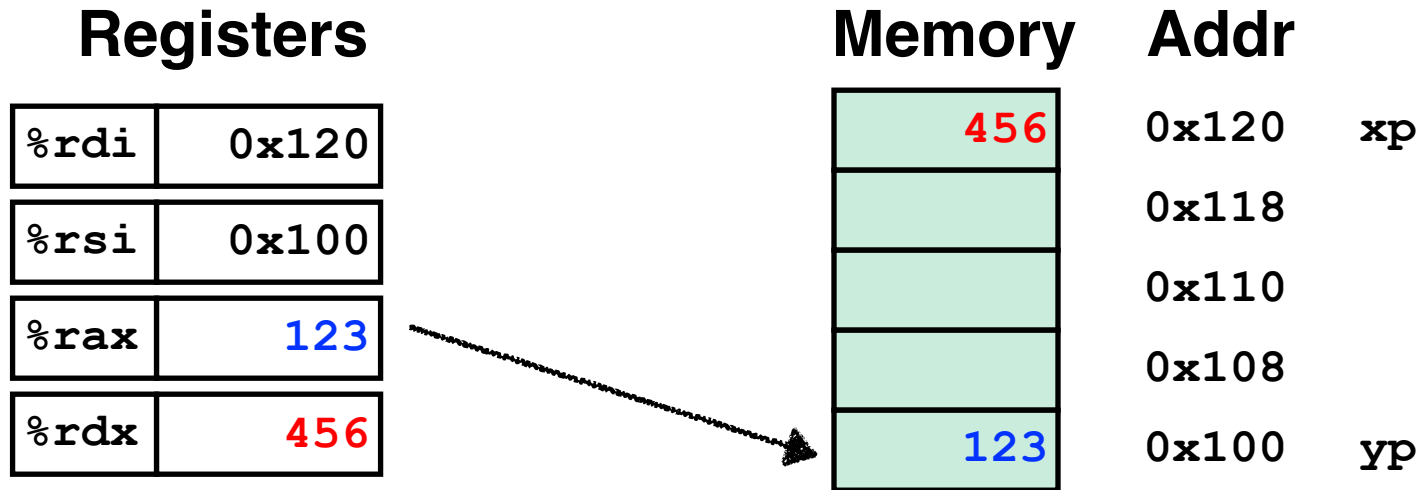
```
movq    (%rsi), %rdx    # t1 = *yp
```

```
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

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

Expression	Address Computation	Address
<code>0x8(%rdx)</code>		
<code>(%rdx,%rcx)</code>		
<code>(%rdx,%rcx,4)</code>		
<code>0x80(,%rdx,2)</code>		

Address Computation Examples

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

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

Address Computation Examples

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

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

Address Computation Examples

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

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

Address Computation Examples

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

Expression	Address Computation	Address
<code>0x8 (%rdx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%rdx, %rcx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%rdx, %rcx, 4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80 (, %rdx, 2)</code>	<code>2*0xf000 + 0x80</code>	<code>0x1e080</code>

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**

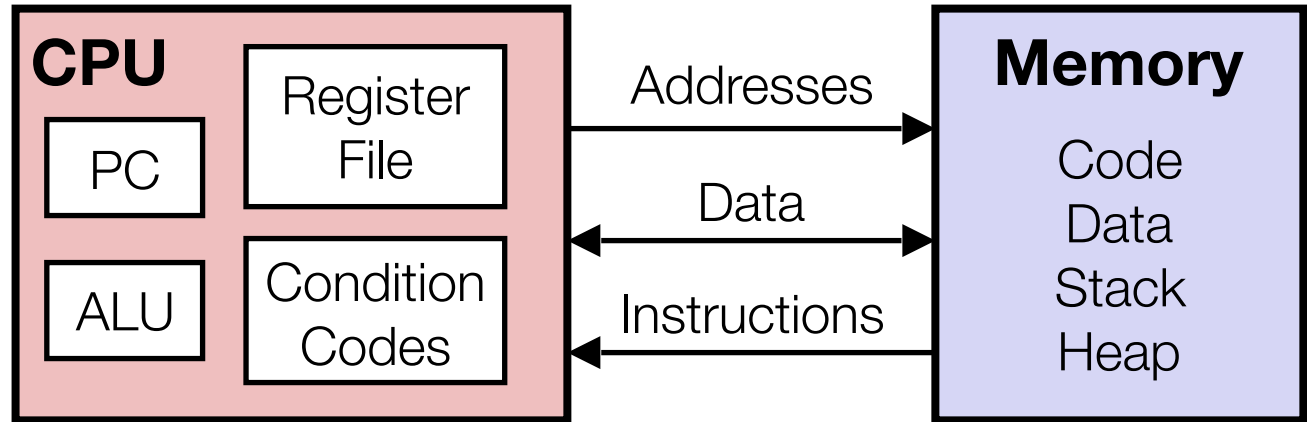
- *Src* is address mode expression
- Set *Dst* to address denoted by expression
- No actual memory reference is made

- **Uses**

- Computing addresses without a memory reference
 - E.g., translation of `p = &x[i];`

Assembly Program Instructions

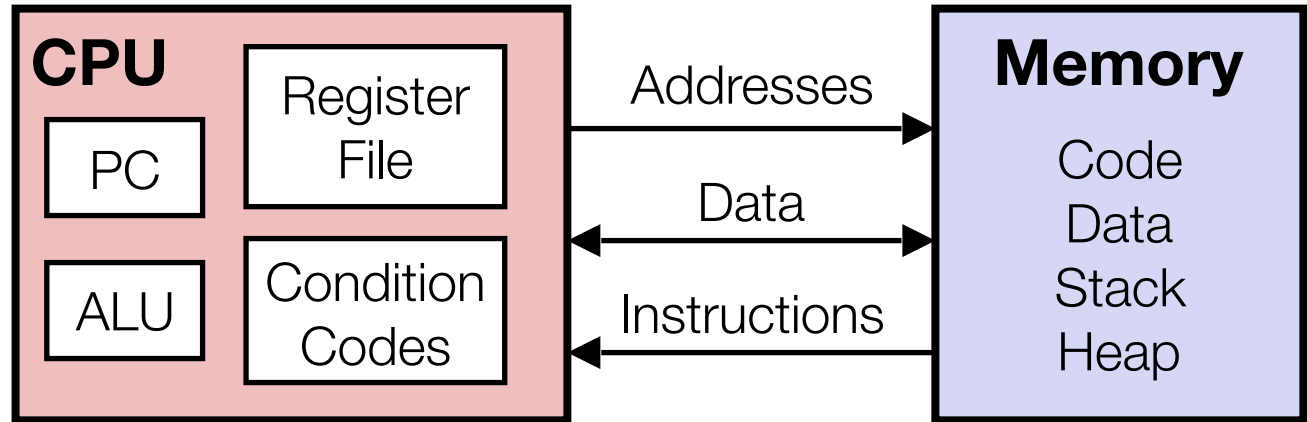
Assembly
Programmer's
Perspective
of a Computer



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

Assembly Program Instructions

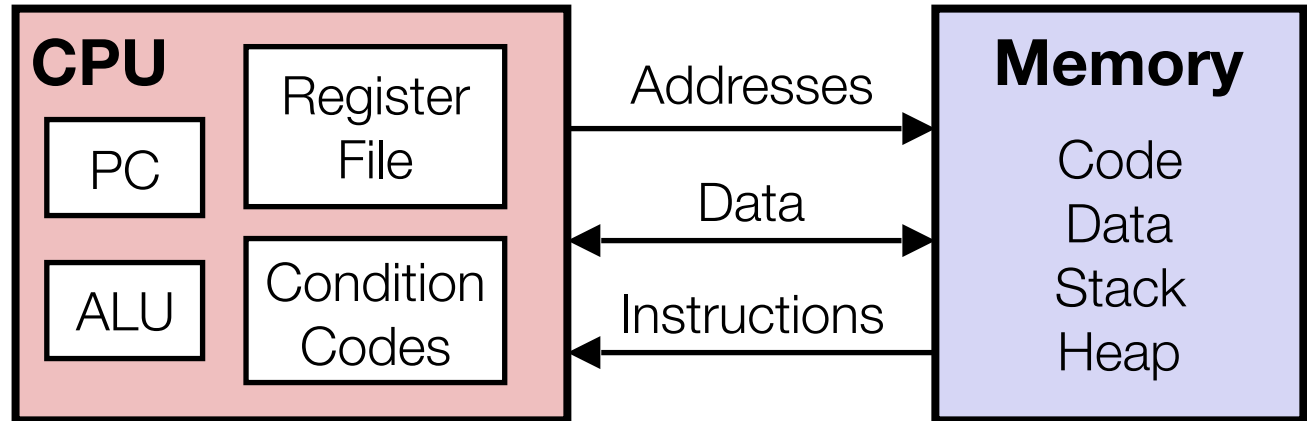
Assembly Programmer's Perspective of a Computer



- *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: Conditional branches (`if... else...`)
- Control: Loops (`for, while`)
- Control: Switch Statements (`case... switch...`)

Some Arithmetic Operations (2 Operands)

Format	Computation	Notes
<code>addq src, dest</code>	$\text{Dest} = \text{Dest} + \text{Src}$	

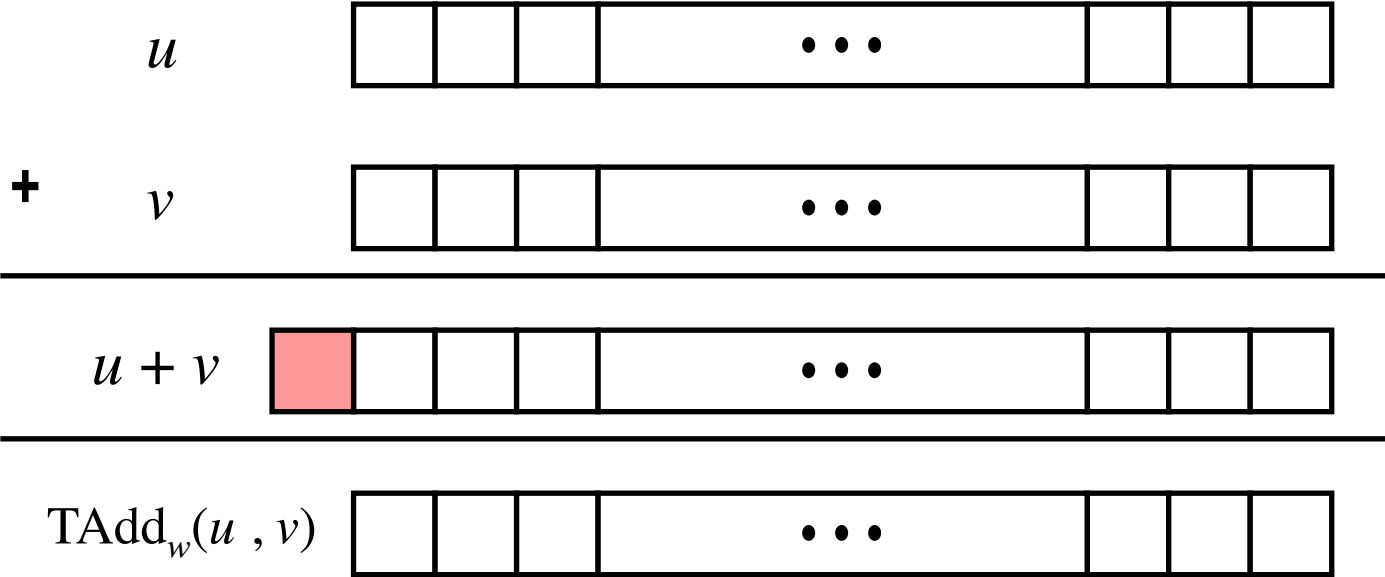
Some Arithmetic Operations (2 Operands)

Format	Computation	Notes
<code>addq src, dest</code>	$\text{Dest} = \text{Dest} + \text{Src}$	

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

Some Arithmetic Operations (2 Operands)

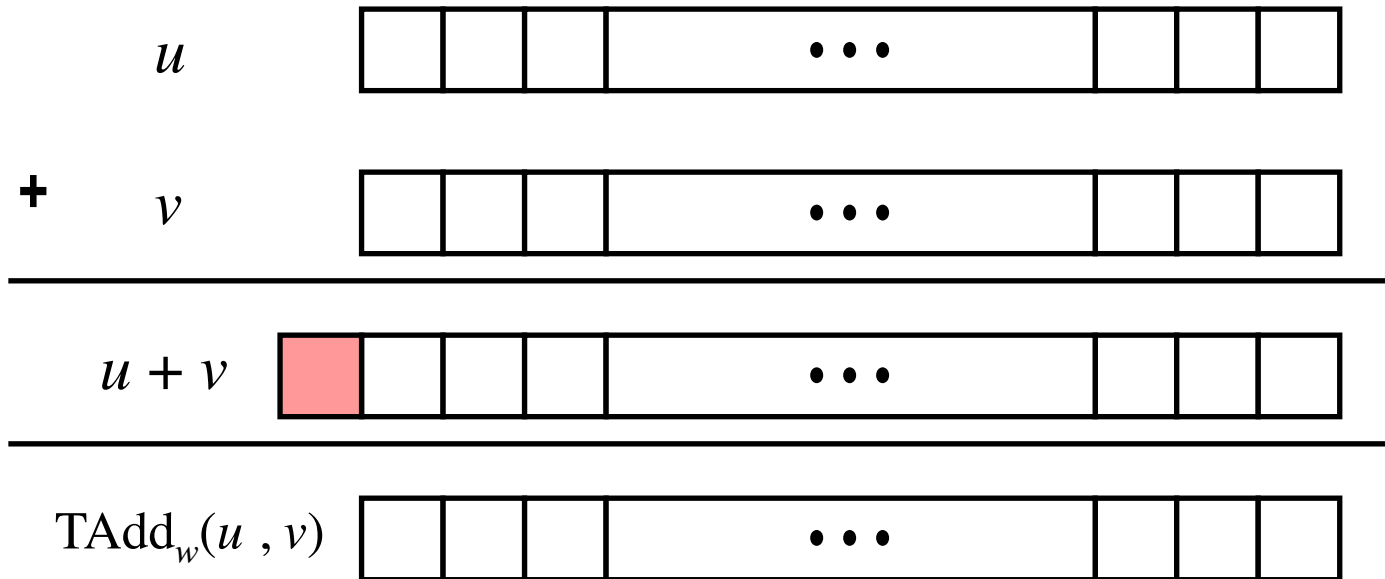
Format	Computation	Notes
<code>addq src, dest</code>	$\text{Dest} = \text{Dest} + \text{Src}$	



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

Some Arithmetic Operations (2 Operands)

Format	Computation	Notes
<code>addq src, dest</code>	$\text{Dest} = \text{Dest} + \text{Src}$	



`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
010
+) 101

111

Signed
2
+) -3

-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

$$\begin{array}{r} \text{Bit-level} \\ 010 \\ +) 101 \\ \hline 111 \end{array}$$

$$\begin{array}{r} \text{Signed} \\ 2 \\ +) -3 \\ \hline -1 \end{array}$$

$$\begin{array}{r} \text{Unsigned} \\ 2 \\ +) 5 \\ \hline 7 \end{array}$$

```
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

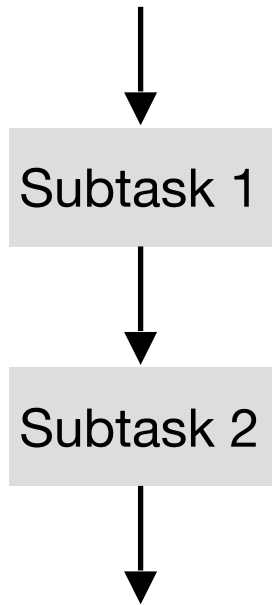
Today: Compute and Control Instructions

- 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



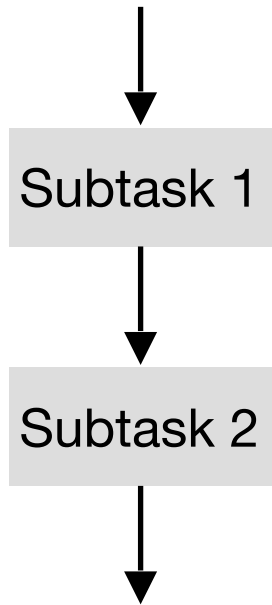
```
a = x + y;
```

```
y = a - c;
```

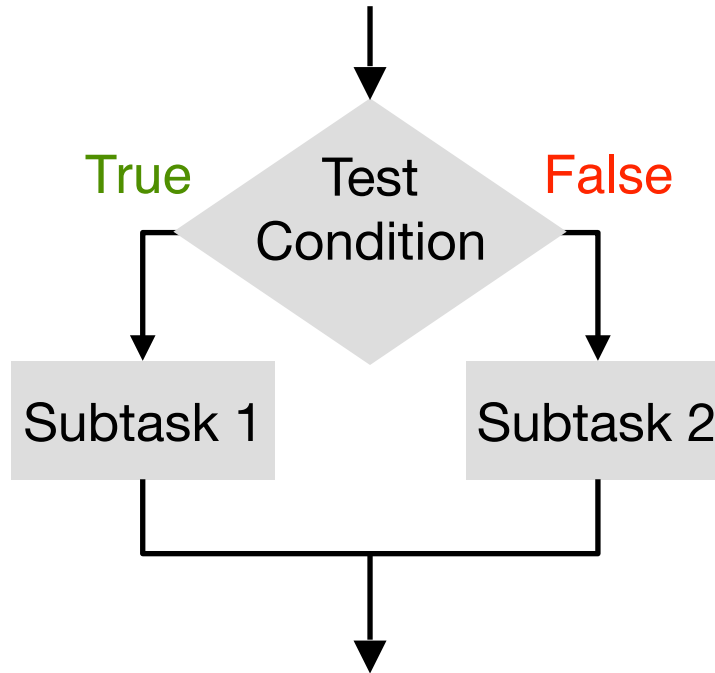
```
...
```


Three Basic Programming Constructs

Sequential



Conditional



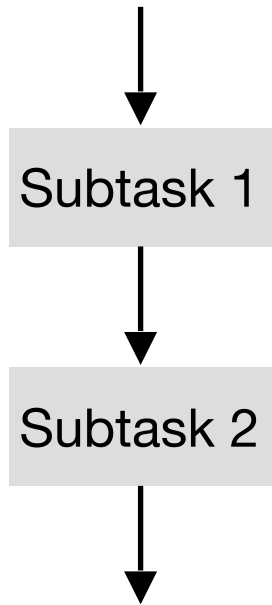
```
a = x + y;  
y = a - c;
```

...

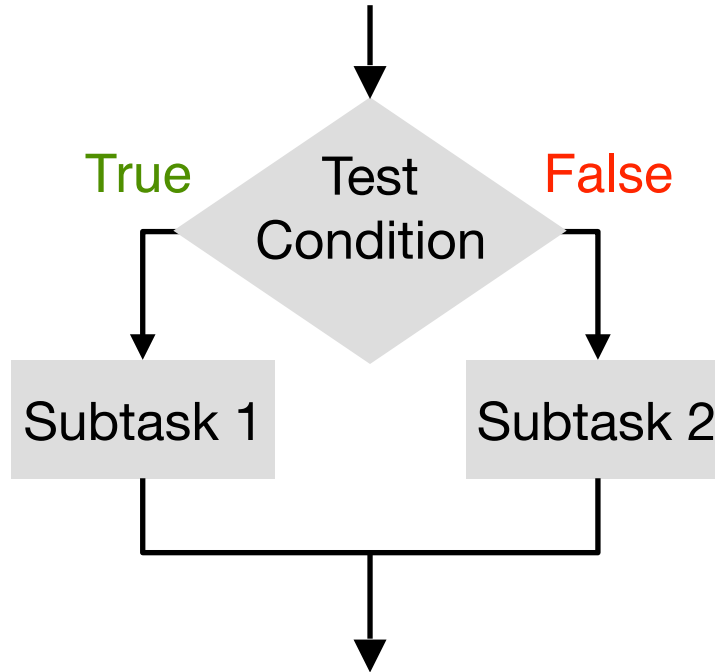
```
if (x > y) r = x - y;  
else r = y - x;
```

Three Basic Programming Constructs

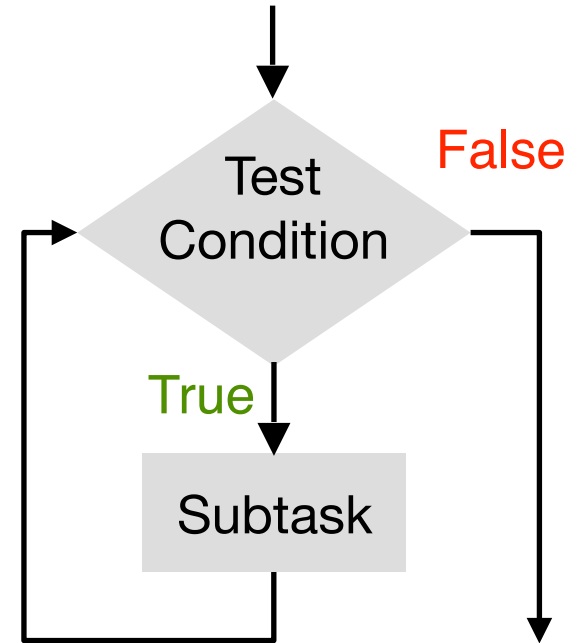
Sequential



Conditional



Iterative



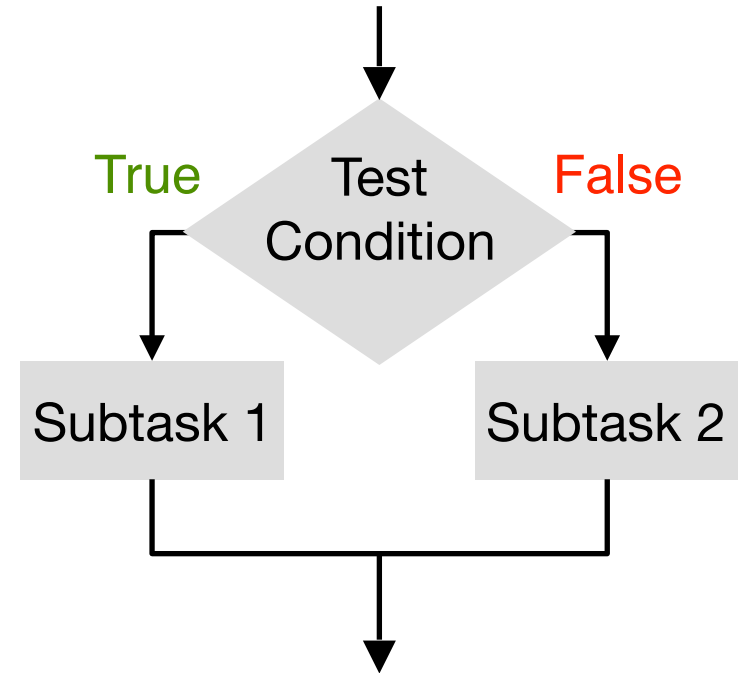
```
a = x + y;  
y = a - c;  
...
```

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

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

Three Basic Programming Constructs

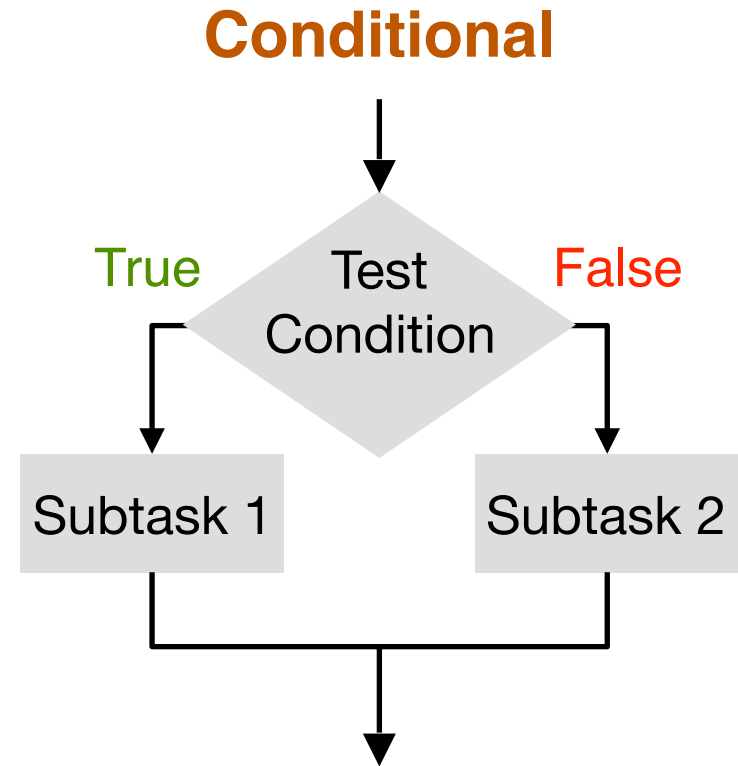
Conditional



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

Three Basic Programming Constructs

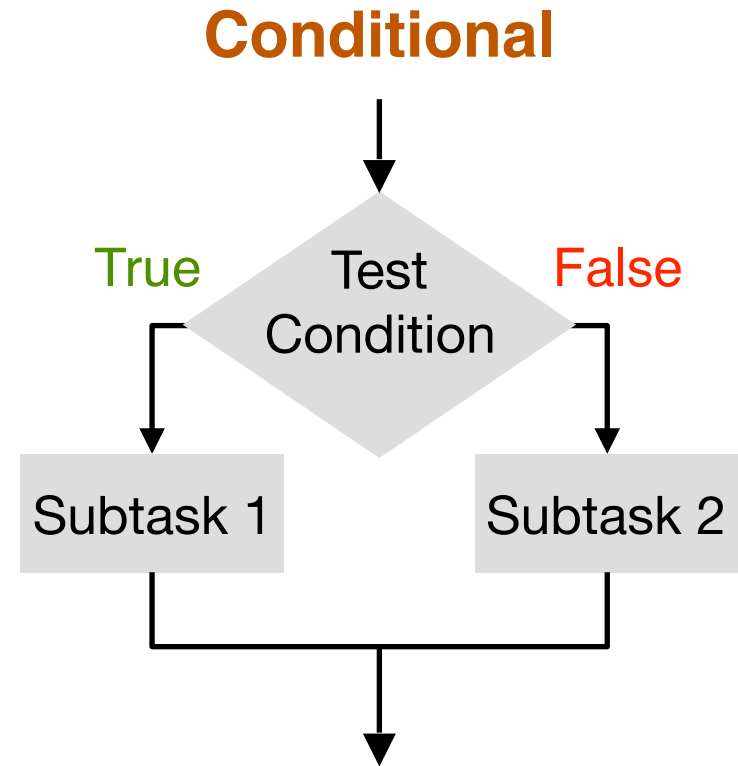
- Both conditional and iterative programming requires altering the sequence of instructions (control flow)



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

Three Basic Programming Constructs

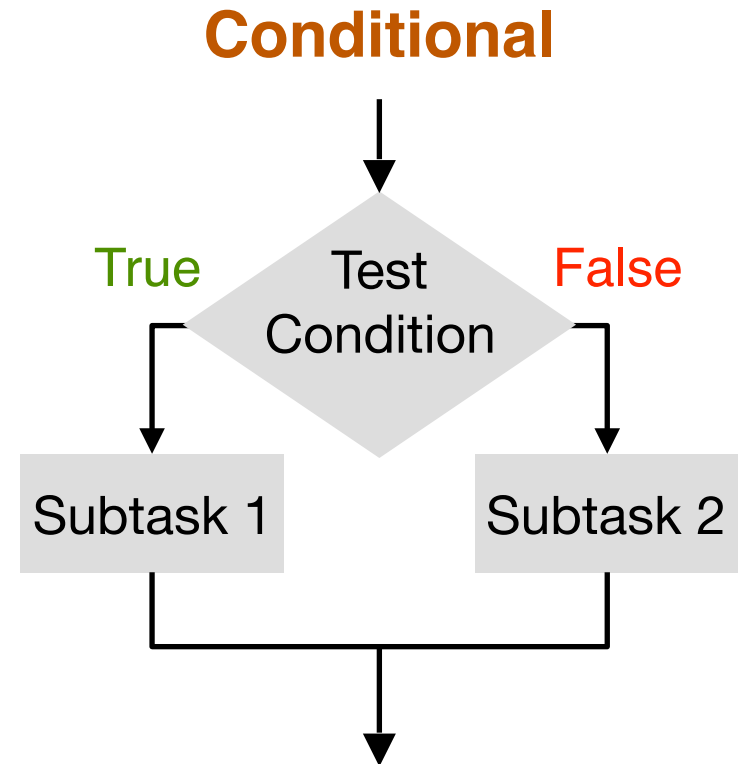
- Both conditional and iterative programming requires altering the sequence of instructions (control flow)
- We need a set of *control instructions* to do so



```
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?



```
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;
}
```

```
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;
}
```

```
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
```

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

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
<hr/>		
	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
	<hr/>	<hr/>
	111	-1
Overflow	101	-3
	-) 011	-) 3
	<hr/>	<hr/>
	010	-6

ZF Zero Flag (result is zero)



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
	<u>111</u>	<u>-1</u>
Overflow	101	-3
	-) 011	-) 3
	<u>010</u>	<u>-6</u>

ZF Zero Flag (result is zero)



How Should `cmpq` Set Condition Codes?

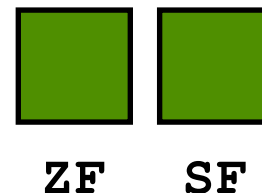
`cmpq` `%rsi, %rdi`

- 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	$\begin{array}{r} 001 \\ -) 010 \\ \hline 111 \end{array}$	$\begin{array}{r} 1 \\ -) 2 \\ \hline -1 \end{array}$
	$\begin{array}{r} 101 \\ -) 011 \\ \hline 010 \end{array}$	$\begin{array}{r} -3 \\ -) 3 \\ \hline -6 \end{array}$

ZF Zero Flag (result is zero)

SF Sign Flag (result is negative)



How Should `cmpq` Set Condition Codes?

`cmpq` `%rsi, %rdi`

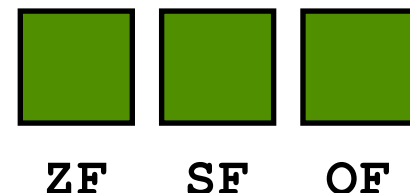
- Essentially, how do we know `%rdi <= %rsi`?
- Calculate `%rdi - %rsi`
- `%rdi == %rsi` if and only if `%rdi - %rsi == 0`
- `%rdi < %rsi` if and only if: ~~`%rdi - %rsi < 0` (is it correct??)~~
 - `%rdi - %rsi < 0` and the result doesn't overflow, or
 - `%rdi - %rsi > 0` and the result does overflow

No Overflow	001	1
	-) 010	-) 2
	<hr/> 111	<hr/> -1
Overflow	101	-3
	-) 011	-) 3
	<hr/> 010	<hr/> -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
<hr/>	<hr/>
10000001	-127

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
<hr/>	<hr/>
10000001	-127

ZF Zero Flag (result is zero)
SF Sign Flag (result is negative)
OF Overflow Flag (result overflow)

0	1	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

- `%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;
}
```

```
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

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	$\begin{array}{r} 001 \\ -) 010 \\ \hline 111 \end{array}$	$\begin{array}{r} 1 \\ -) 2 \\ \hline -1 \end{array}$
----------------	--	---

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	$\begin{array}{r} 001 \\ -) 010 \\ \hline 111 \end{array}$	$\begin{array}{r} 1 \\ -) 2 \\ \hline -1 \end{array}$
	$\begin{array}{r} 101 \\ -) 011 \\ \hline 010 \end{array}$	$\begin{array}{r} -3 \\ -) 3 \\ \hline 2 \end{array}$

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	$\begin{array}{r} 001 \\ -) 010 \\ \hline 111 \end{array}$	$\begin{array}{r} 1 \\ -) 2 \\ \hline -1 \end{array}$		
Overflow	$\begin{array}{r} 101 \\ -) 011 \\ \hline 010 \end{array}$	$\begin{array}{r} -3 \\ -) 3 \\ \hline 2 \end{array}$	$\begin{array}{r} 011 \\ -) 100 \\ \hline 111 \end{array}$	$\begin{array}{r} 3 \\ -) -4 \\ \hline -1 \end{array}$

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	$\begin{array}{r} 001 \\ -) 010 \\ \hline 111 \end{array}$	$\begin{array}{r} 1 \\ -) 2 \\ \hline -1 \end{array}$		
Overflow	$\begin{array}{r} 101 \\ -) 011 \\ \hline 010 \end{array}$	$\begin{array}{r} -3 \\ -) 3 \\ \hline 2 \end{array}$	$\begin{array}{r} 011 \\ -) 100 \\ \hline 111 \end{array}$	$\begin{array}{r} 3 \\ -) -4 \\ \hline -1 \end{array}$