CSC 252: Computer Organization
Spring 2018: Lecture 6

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:

- **Assignment 2 is out**

Announcement

- Programming Assignment 2 is out
  - Due on **Feb 16, 11:59 PM**
  - You may still have 3 slip days…

Announcement

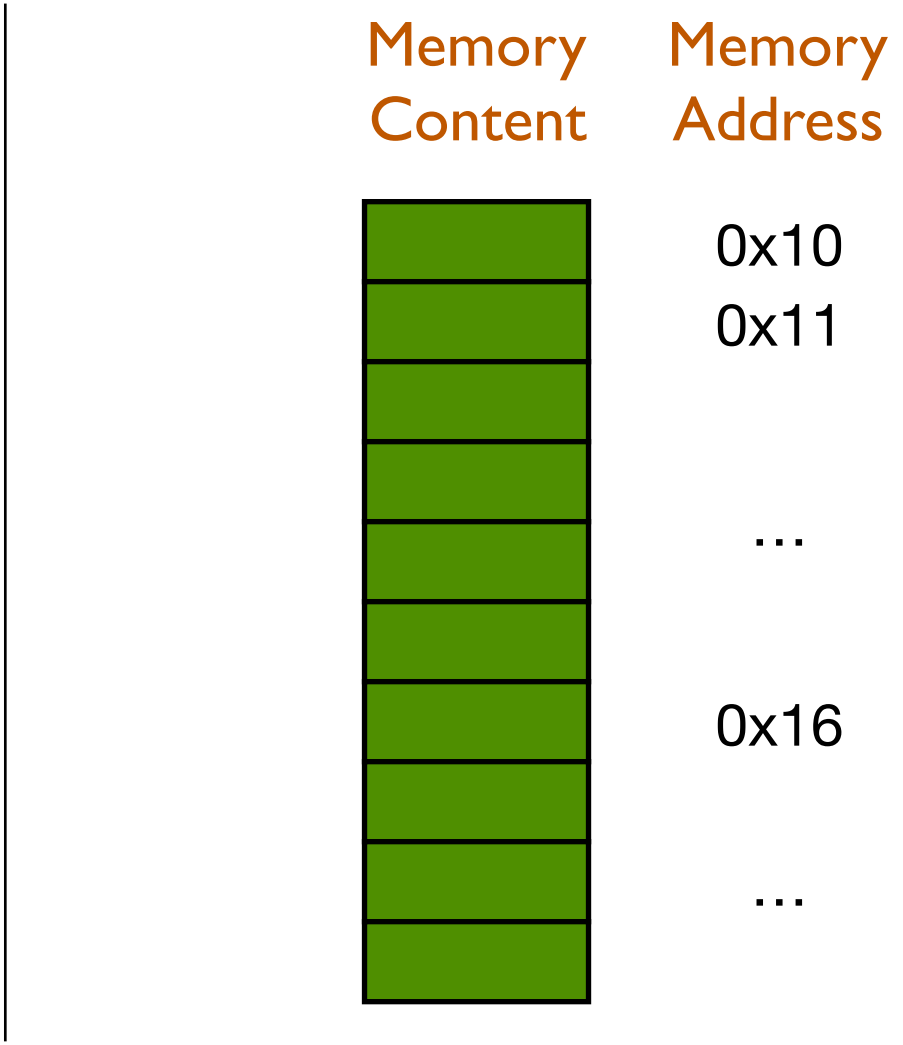
- Programming Assignment 2 is out
  - Due on **Feb 16, 11:59 PM**
  - You may still have 3 slip days…
- Read the instructions before getting started!!!
  - You get 1/4 point off for every wrong answer
  - Maxed out at 10

How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```

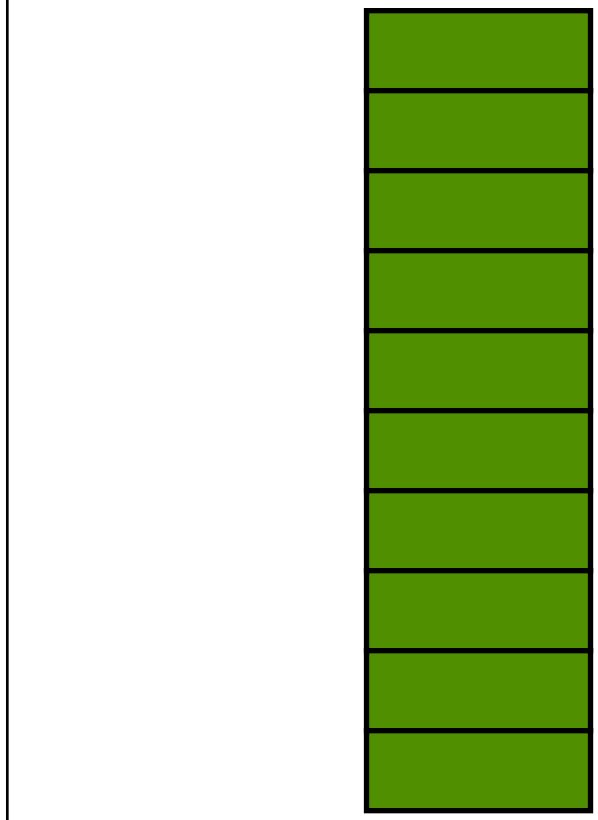
How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```



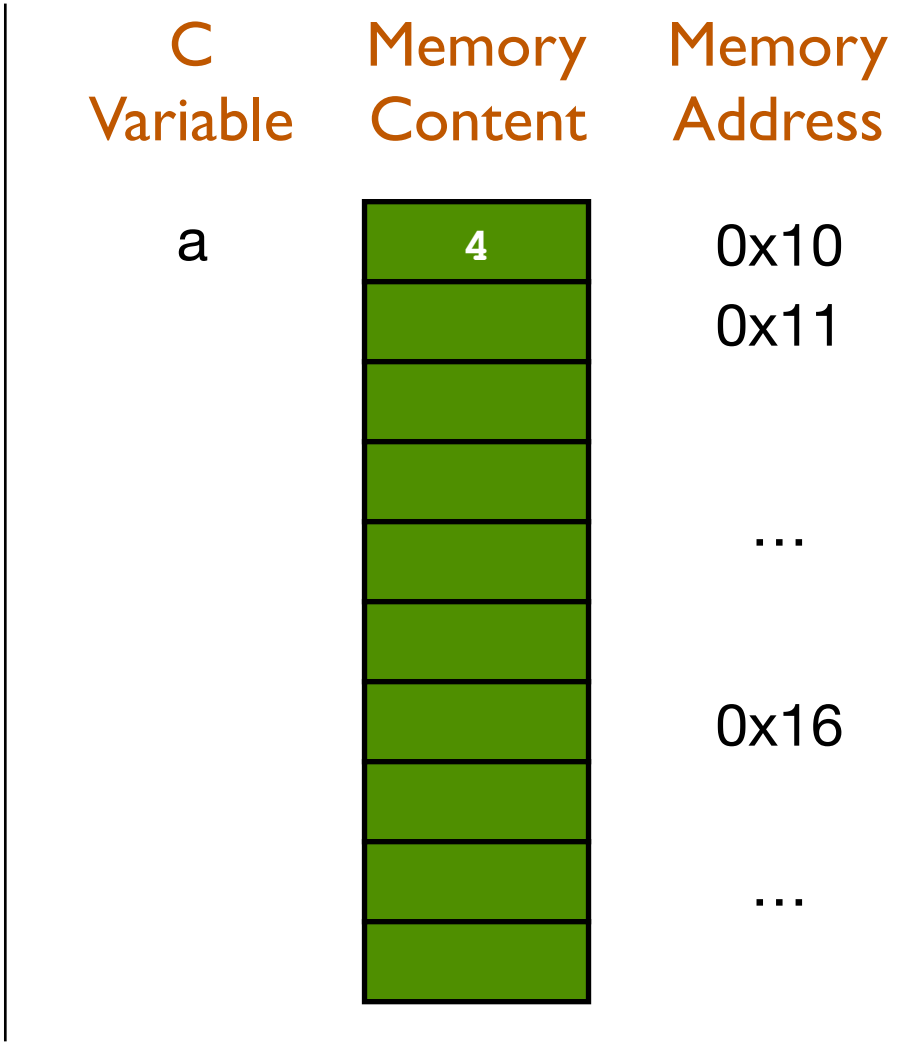
How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```



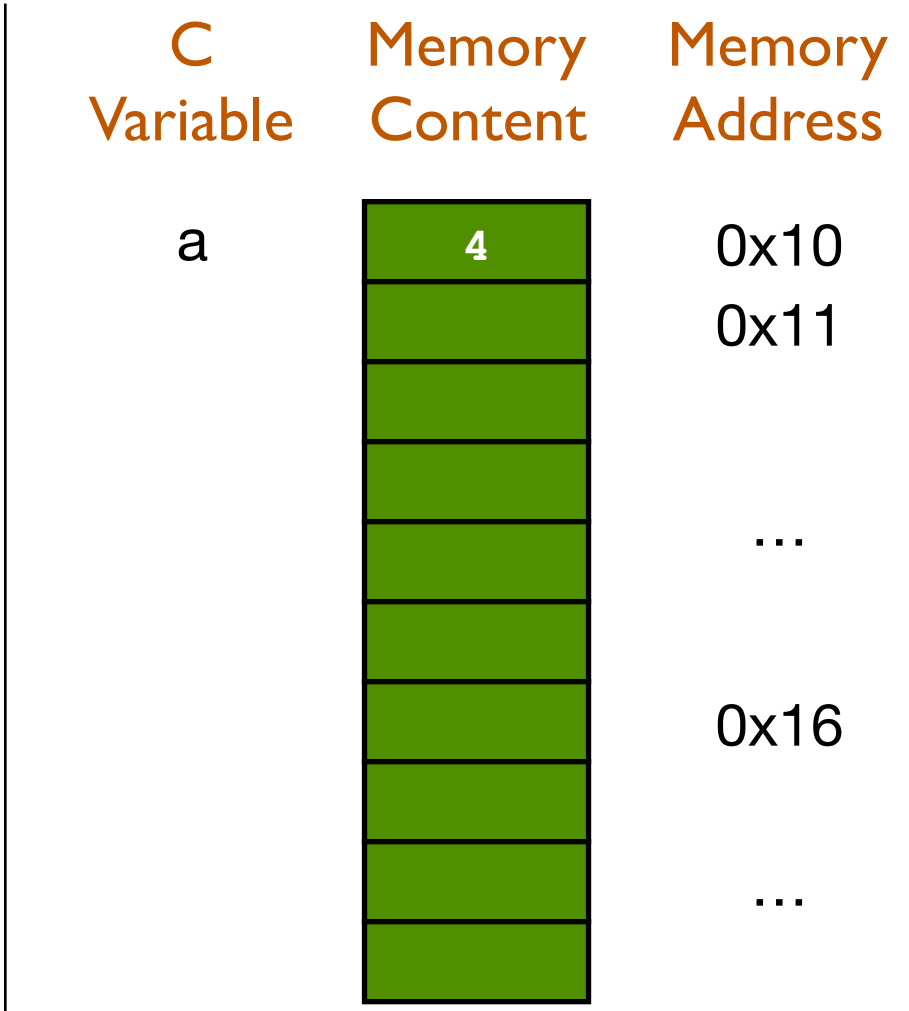
How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```



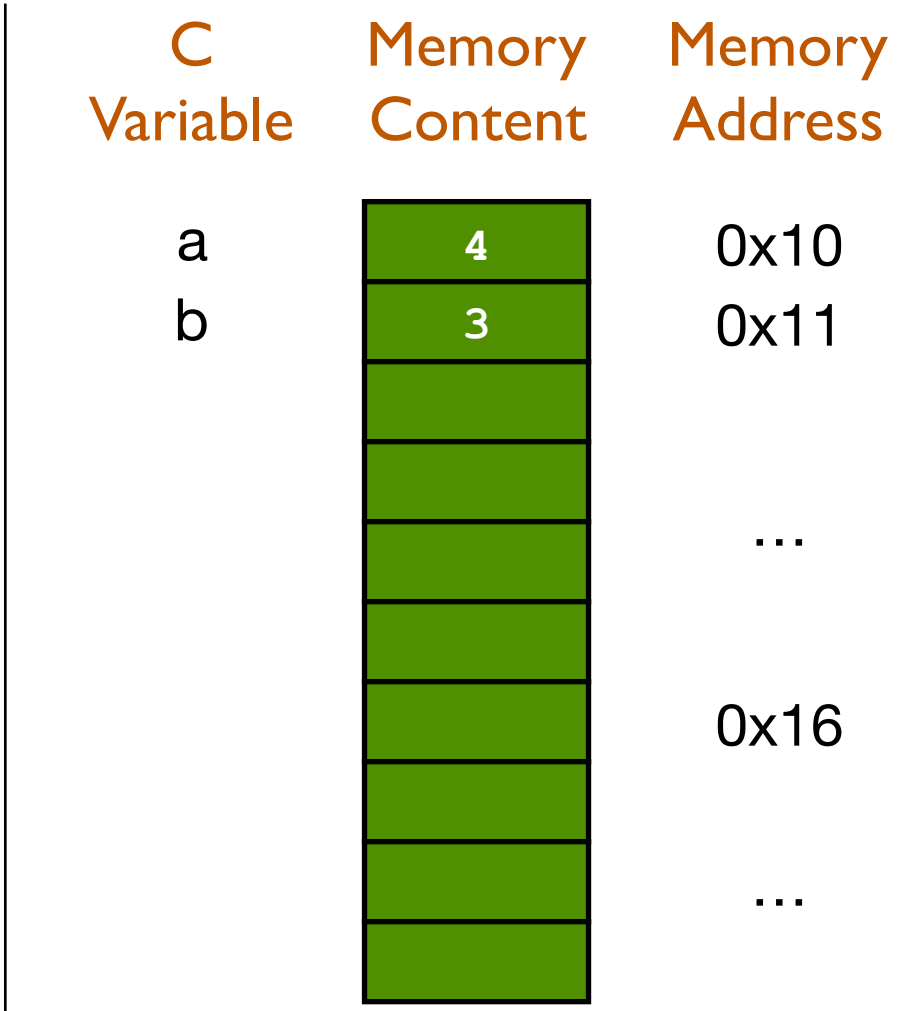
How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```



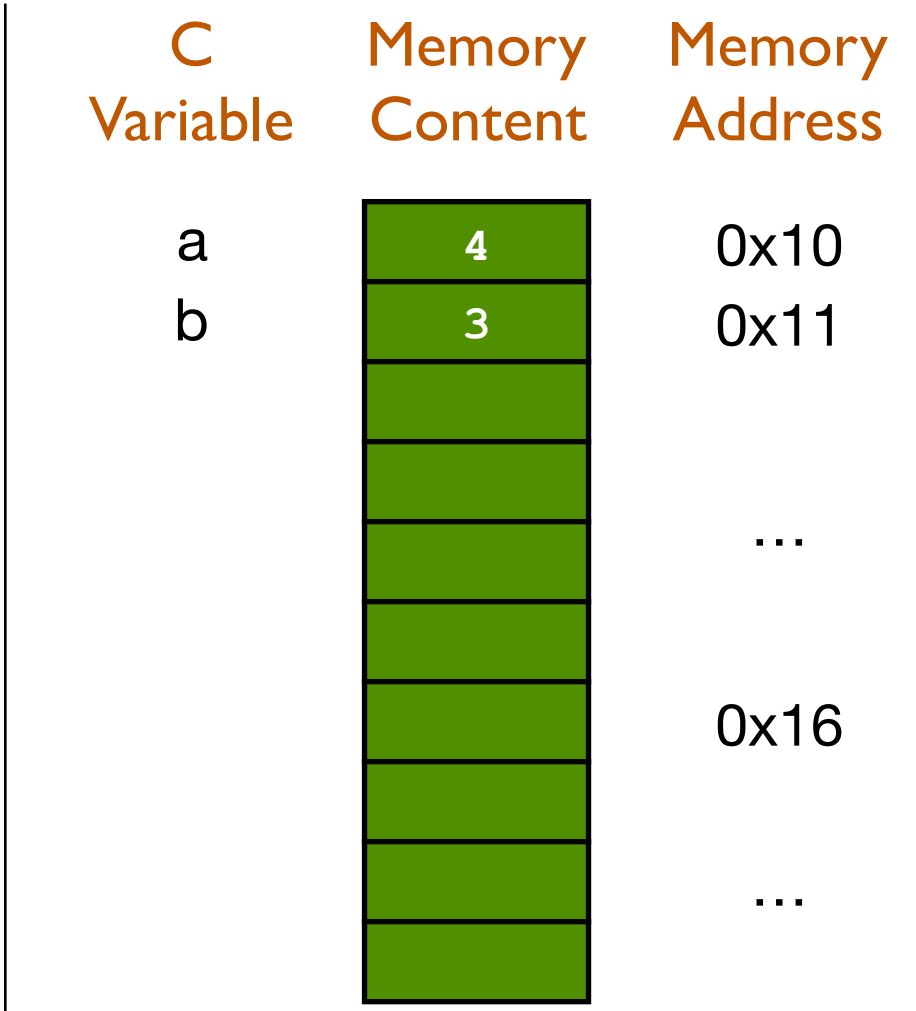
How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```



How Does Pointer Work in C???

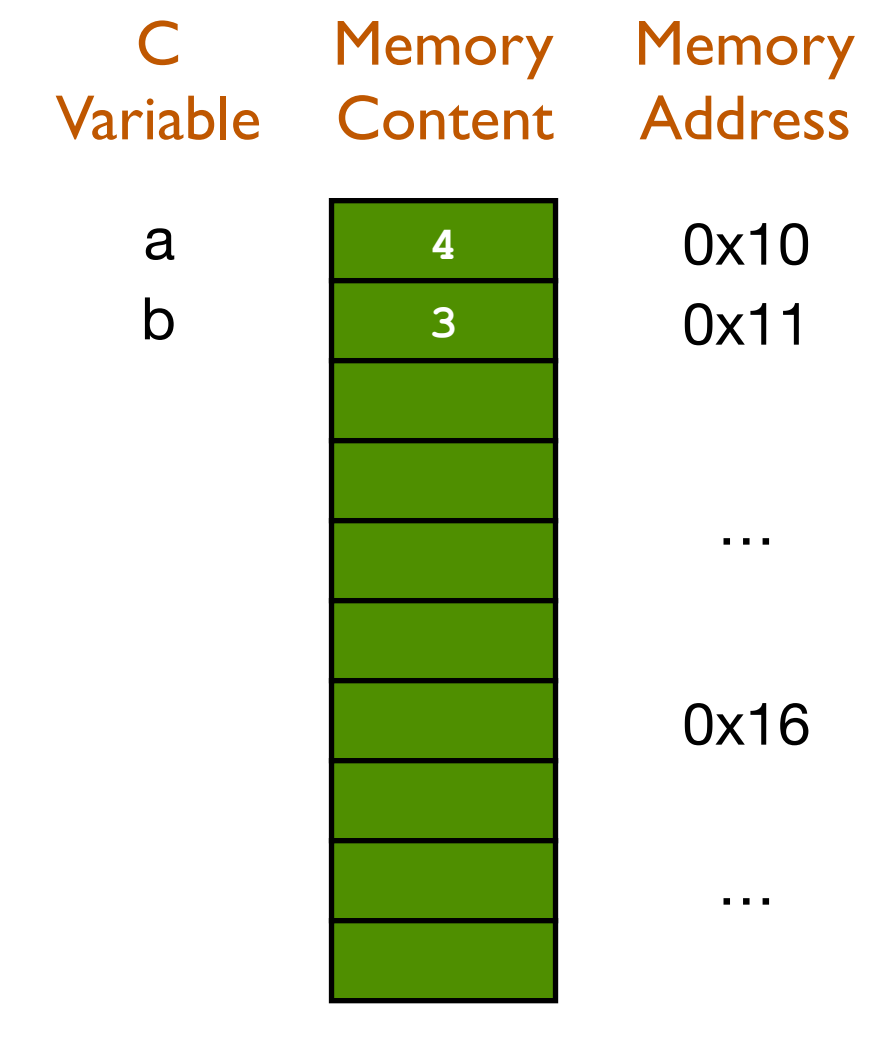
```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```



How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```

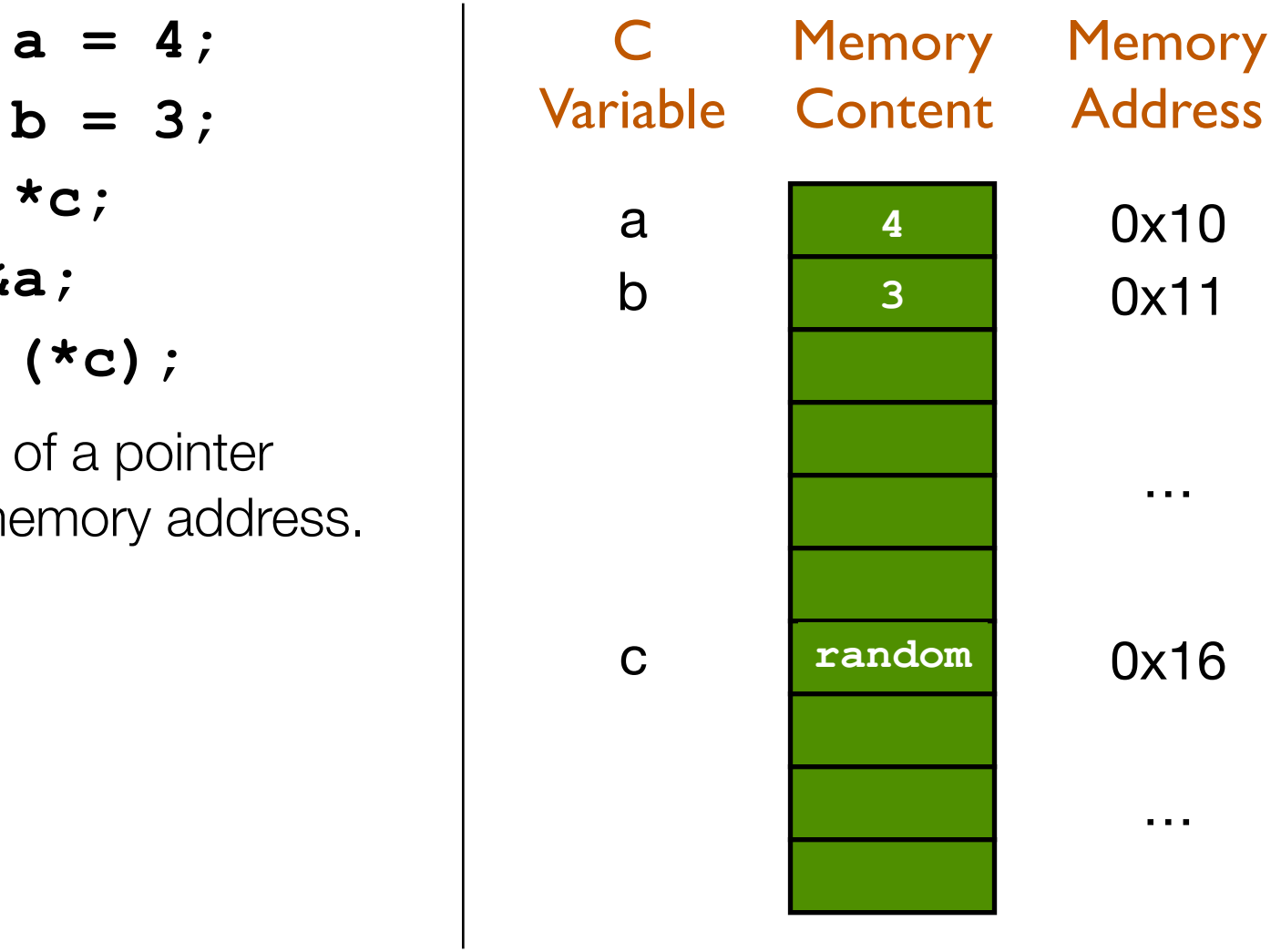
- The content of a pointer variable is memory address.



How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```

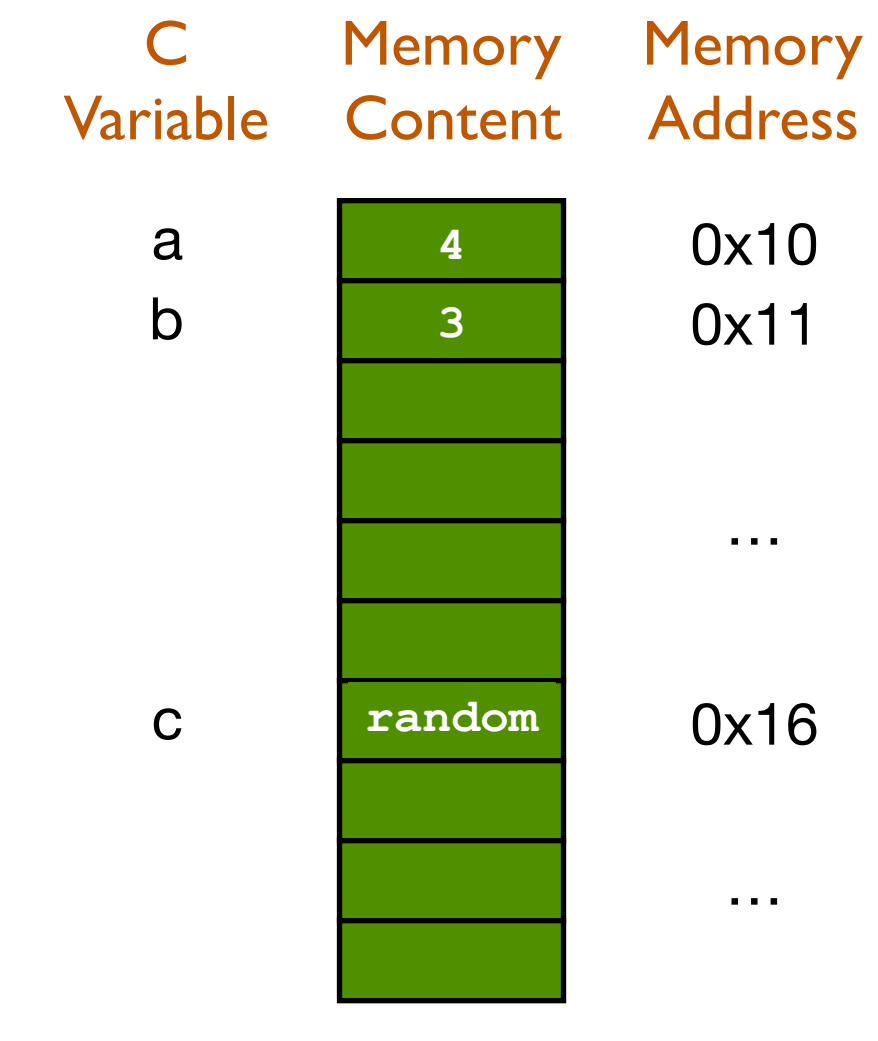
- The content of a pointer variable is memory address.



How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```

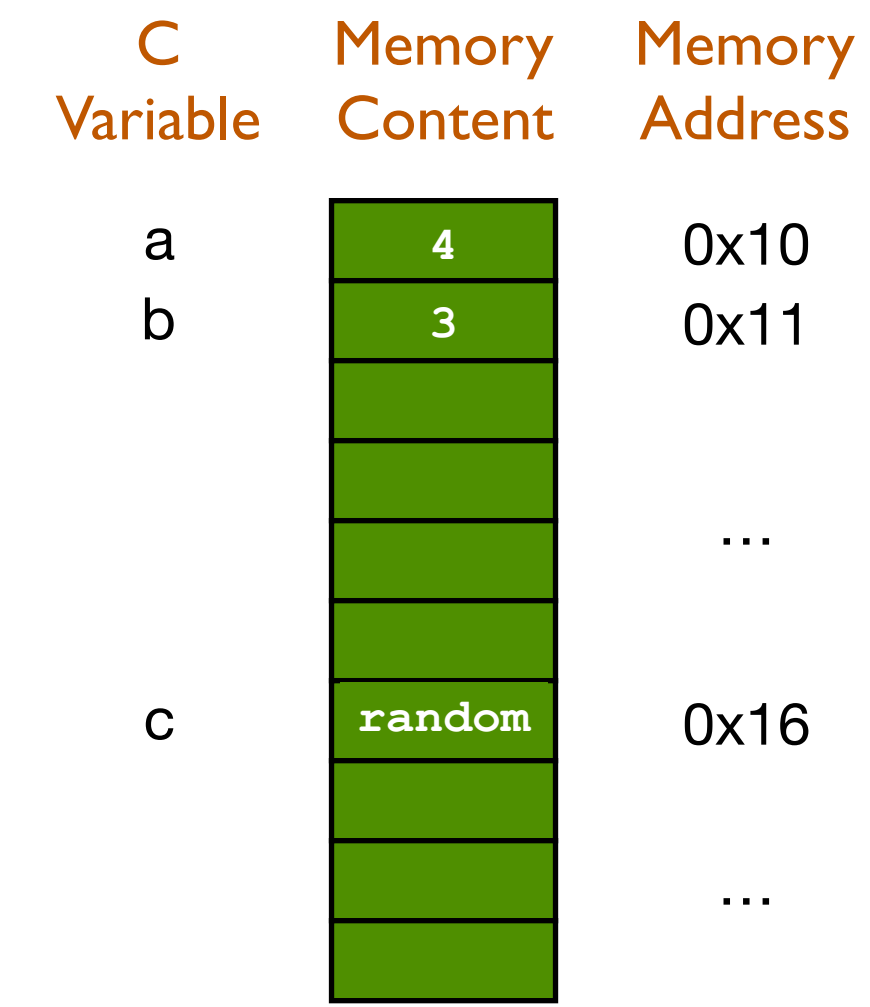
- The content of a pointer variable is memory address.



How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```

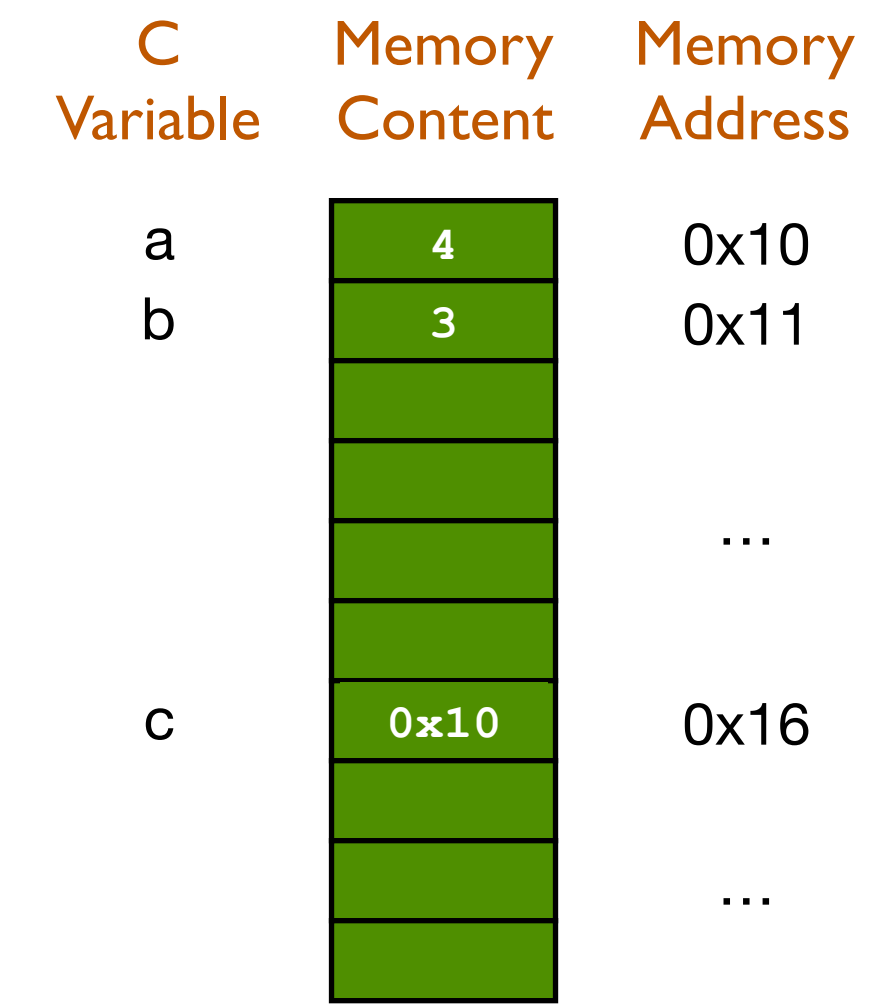
- The content of a pointer variable is memory address.
- The '**&**' operator (address-of operator) returns the memory address of a variable.



How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```

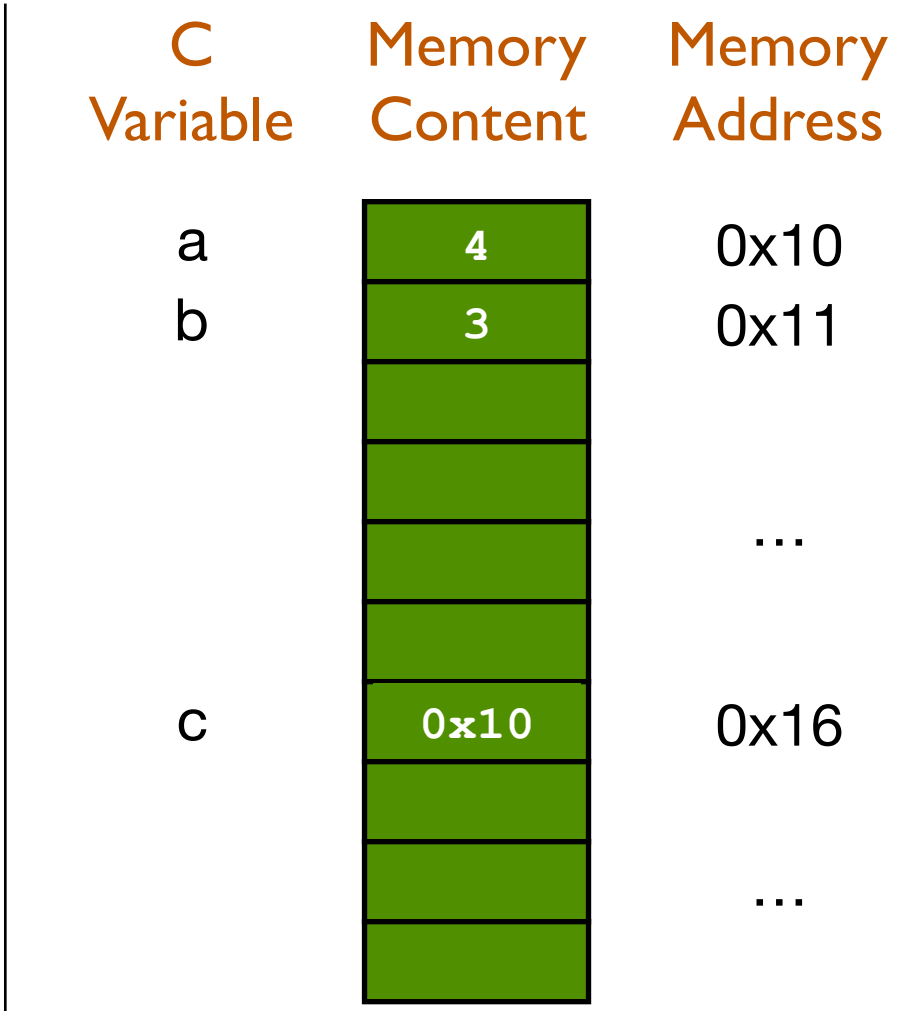
- The content of a pointer variable is memory address.
- The '**&**' operator (address-of operator) returns the memory address of a variable.



How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```

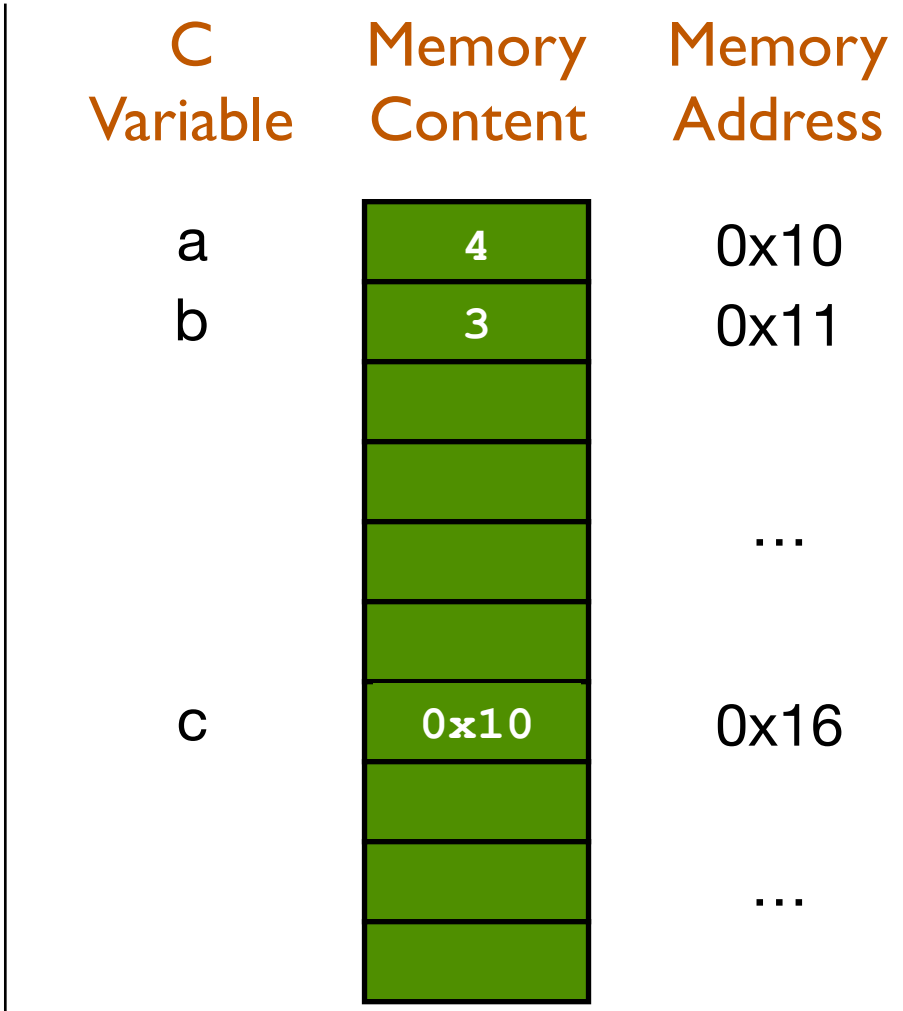
- The content of a pointer variable is memory address.
- The '**&**' operator (address-of operator) returns the memory address of a variable.



How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```

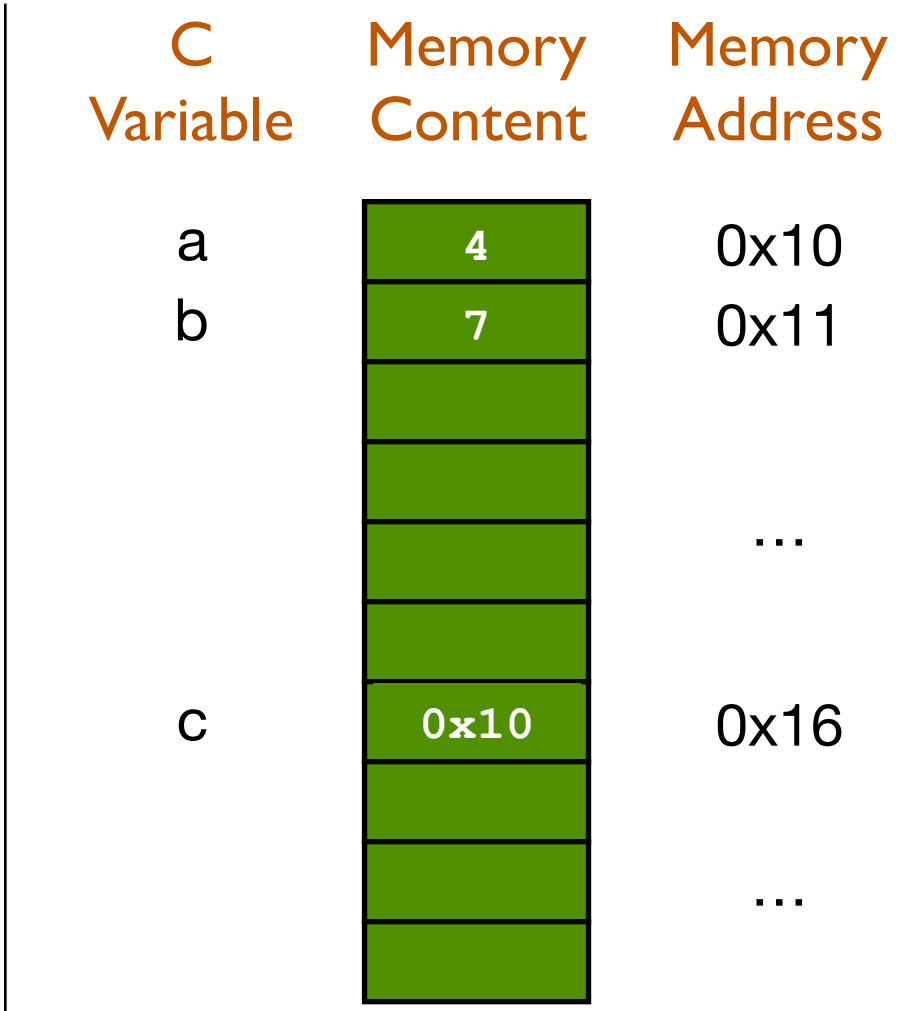
- The content of a pointer variable is memory address.
- The '**&**' operator (address-of operator) returns the memory address of a variable.
- The '*****' operator returns the content stored at the memory location pointed by the pointer variable (dereferencing)



How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```

- The content of a pointer variable is memory address.
- The '**&**' operator (address-of operator) returns the memory address of a variable.
- The '*****' operator returns the content stored at the memory location pointed by the pointer variable (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]**)
  - Essentially 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]**)
  - Essentially pointer dereferencing in C

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

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

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

Complete Memory Addressing Modes

- Most 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

- Most 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

- Special Cases

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

Address Computation Examples

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

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

Address Computation Examples

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

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

Address Computation Examples

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

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

Address Computation Examples

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

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

Address Computation Examples

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

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

Address Computation Instruction

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

Address Computation Instruction

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

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

Address Computation Instruction

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

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

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

Address Computation Instruction

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

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

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

Address Computation Instruction

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

Address Computation Instruction

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

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

Address Computation Instruction

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

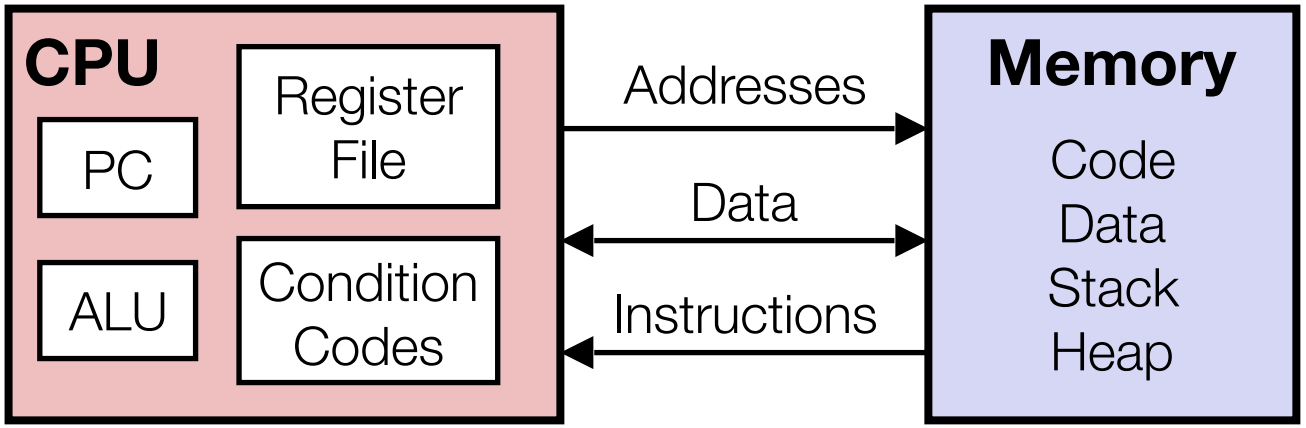
```
long m12(long x)
{
  return x*12;
}
```

Converted to
ASM by compiler:

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

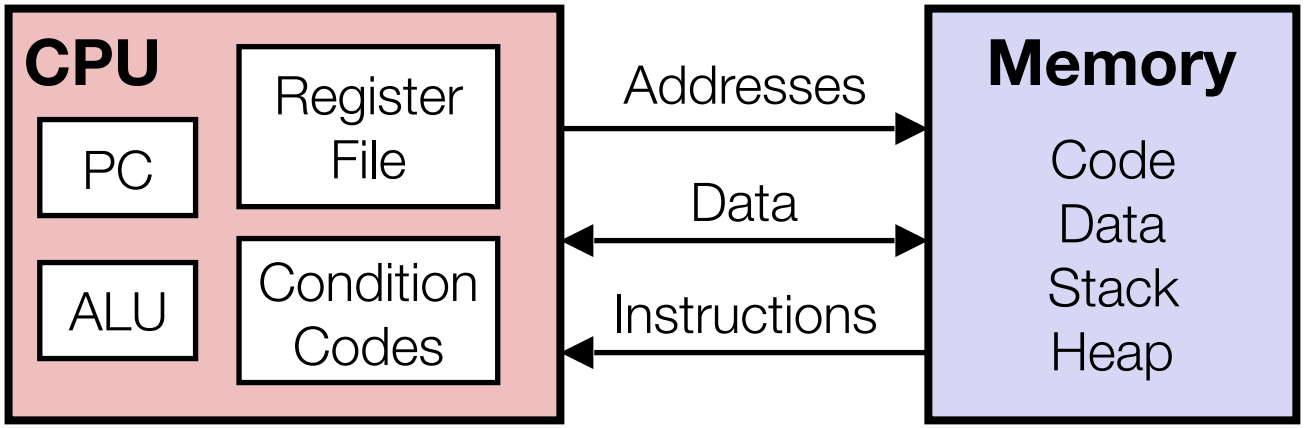
Assembly Program Instructions

Assembly Programmer's Perspective of a Computer



Assembly Program Instructions

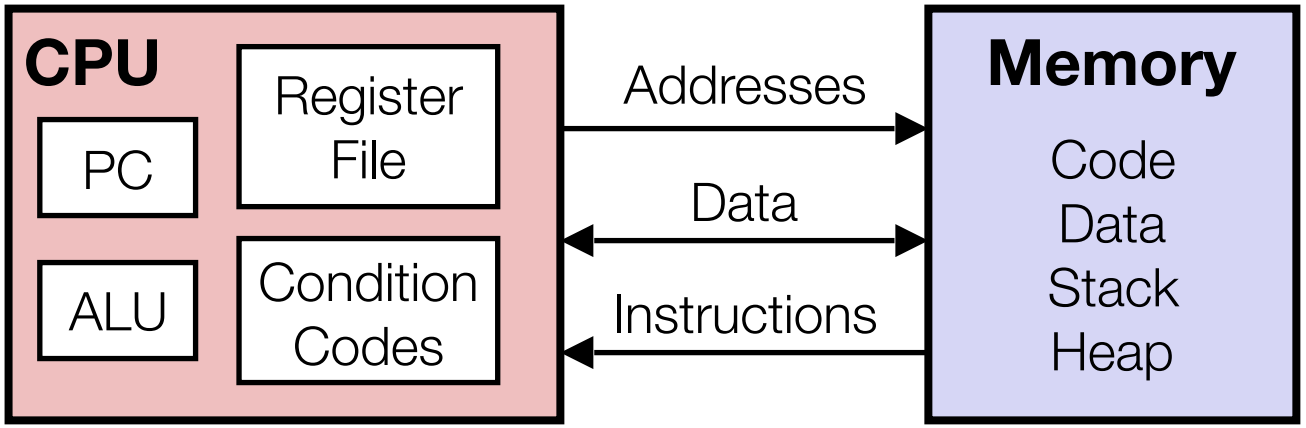
Assembly Programmer's Perspective of a Computer



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

Assembly Program Instructions

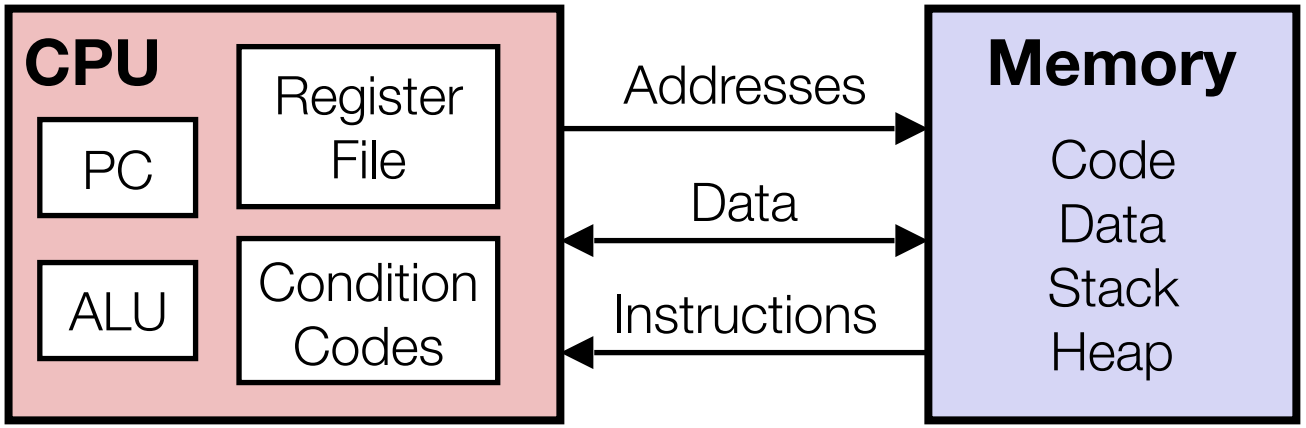
Assembly Programmer's Perspective of a Computer



- *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

- Arithmetic & logical operations
- Control: Condition codes
- Conditional branches (**`if`**... **`else`**...)
- Loops (**`for`**, **`while`**)
- Switch Statements (**`case`**... **`switch`**...)

Some Arithmetic Operations (2 Operands)

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

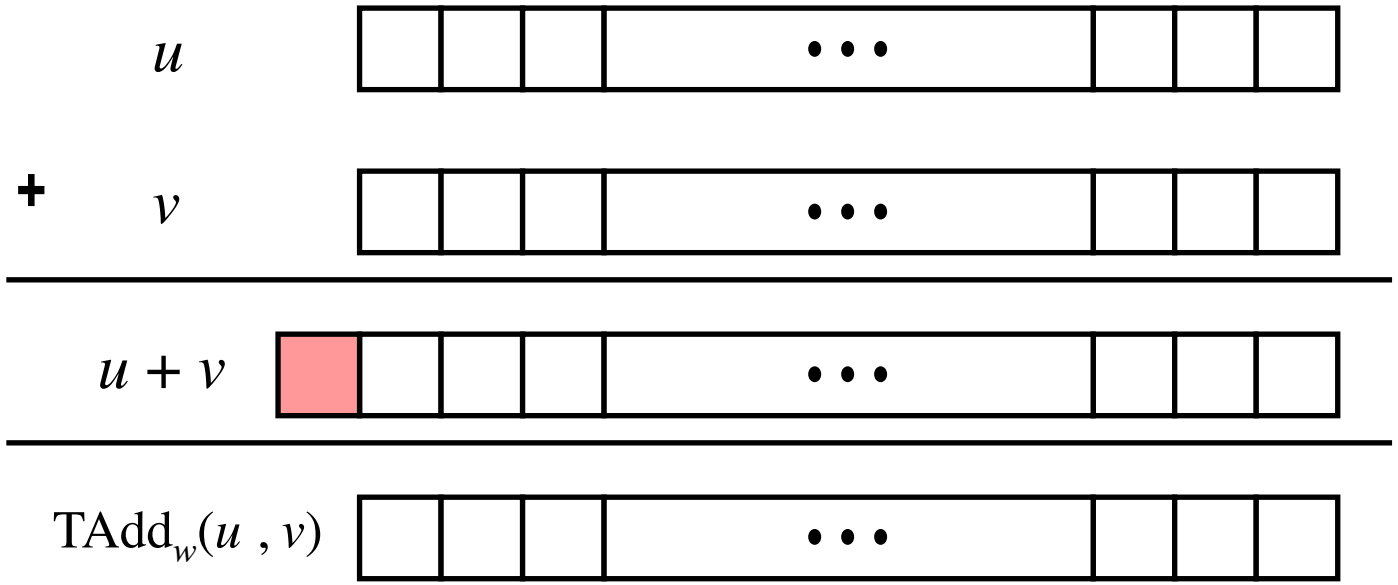
Some Arithmetic Operations (2 Operands)

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

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

Some Arithmetic Operations (2 Operands)

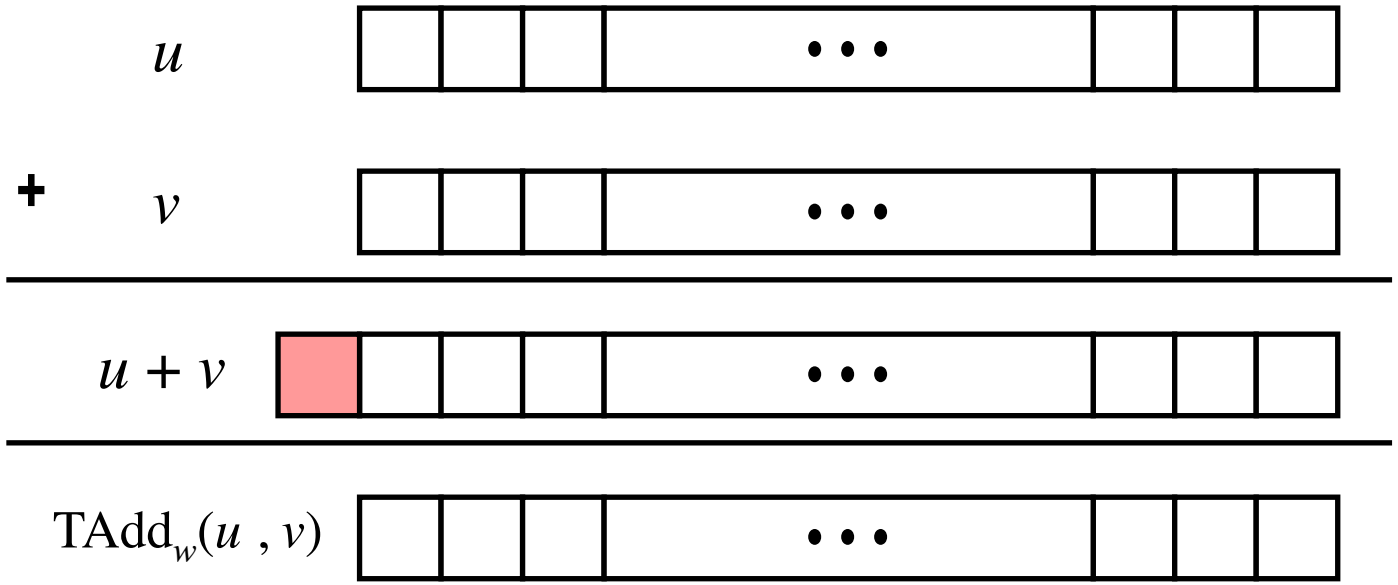
Format	Computation	Notes
addq src, dest	Dest = Dest + Src	



addq %rax, %rbx

Some Arithmetic Operations (2 Operands)

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



addq %rax, %rbx

%rbx = %rax + %rbx
Truncation if overflow,
set carry bit (more later…)

Some Arithmetic Operations (2 Operands)

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

Some Arithmetic Operations (2 Operands)

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

Some Arithmetic Operations (2 Operands)

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

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

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

Some Arithmetic Operations (2 Operands)

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

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

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

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

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

Some Arithmetic Operations (2 Operands)

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

Bit-level

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

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

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

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

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

Some Arithmetic Operations (2 Operands)

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

Bit-level

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

Signed

```
    2
+) -3
-----
   -1
```

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

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

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

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

Some Arithmetic Operations (2 Operands)

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

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

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

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

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

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

Some Arithmetic Operations (1 Operand)

- Unary Instructions (one operand)

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

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

Questions?

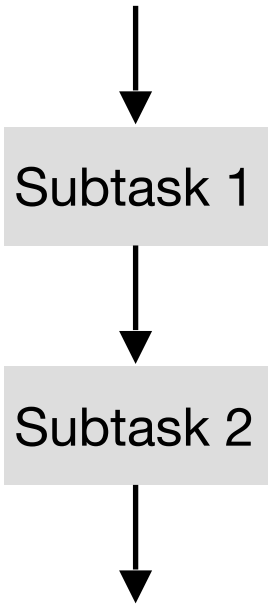
Today: Compute and Control Instructions

- Arithmetic & logical operations
- Control: Condition codes
- Conditional branches (**`if`**... **`else`**...)
- Loops (**`for`**, **`while`**)
- Switch Statements (**`case`**... **`switch`**...)

Three Basic Programming Constructs

Three Basic Programming Constructs

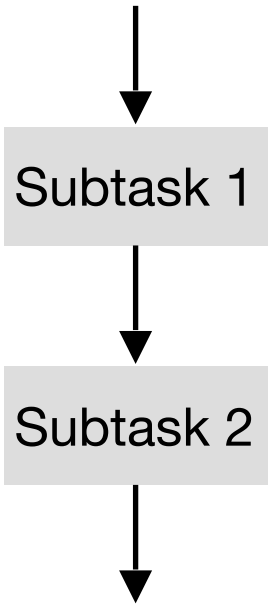
Sequential




```
a = x + y;
y = a - c;
…
```

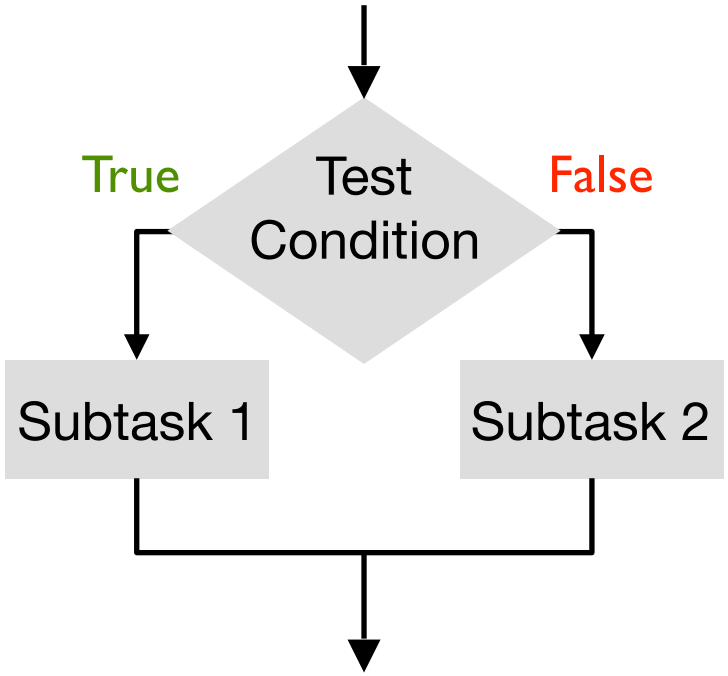
Three Basic Programming Constructs

Sequential




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

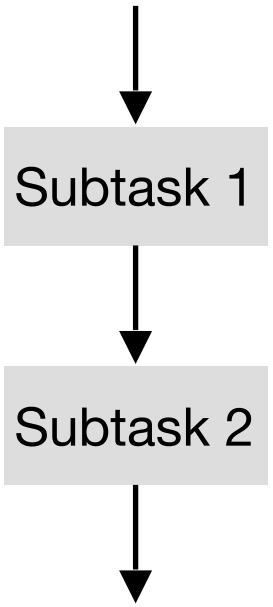
Conditional




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

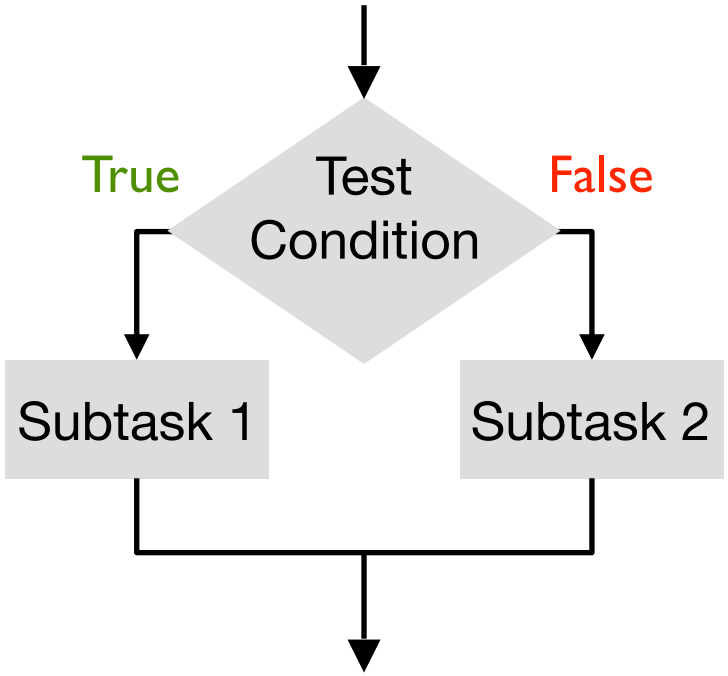
Three Basic Programming Constructs

Sequential



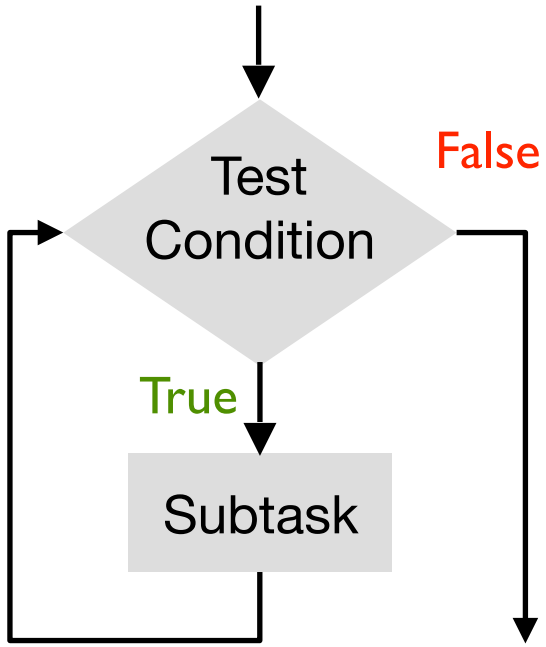

```
a = x + y;
y = a - c;
…
```

Conditional



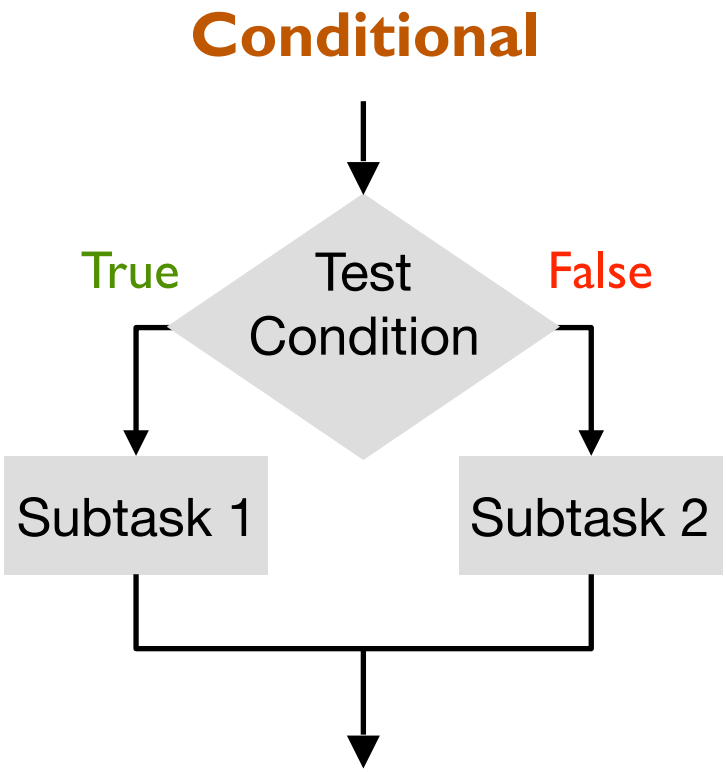

```
if (x > y) r = x - y;
else r = y - x;
```

Iterative




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

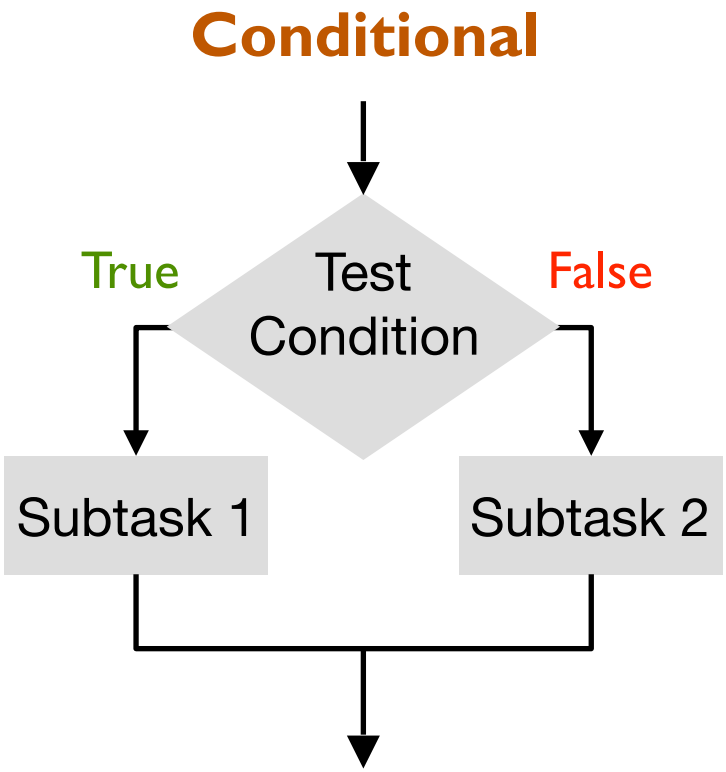
Three Basic Programming Constructs




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

Three Basic Programming Constructs

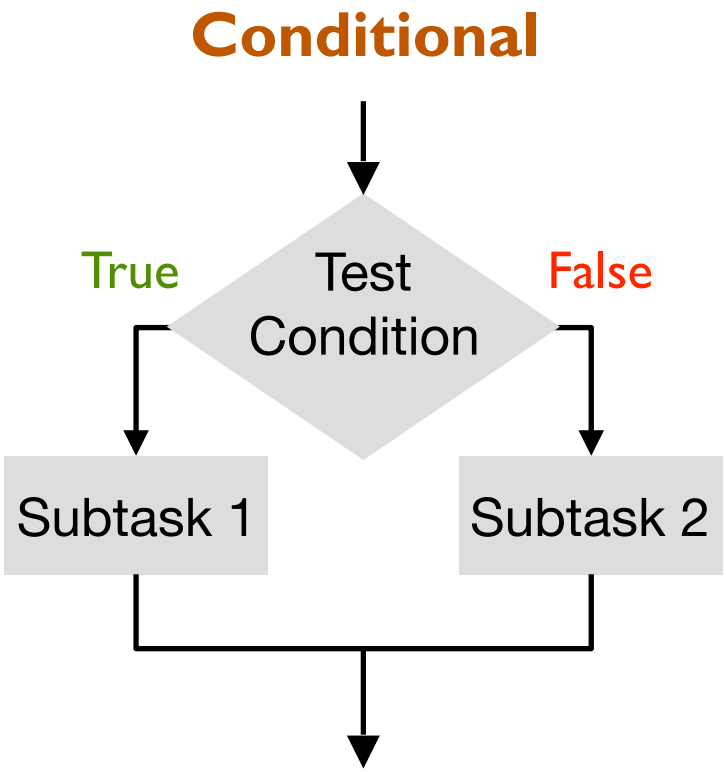
- Both conditional and iterative programming requires altering the sequence of instructions (control flow)




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

Three Basic Programming Constructs

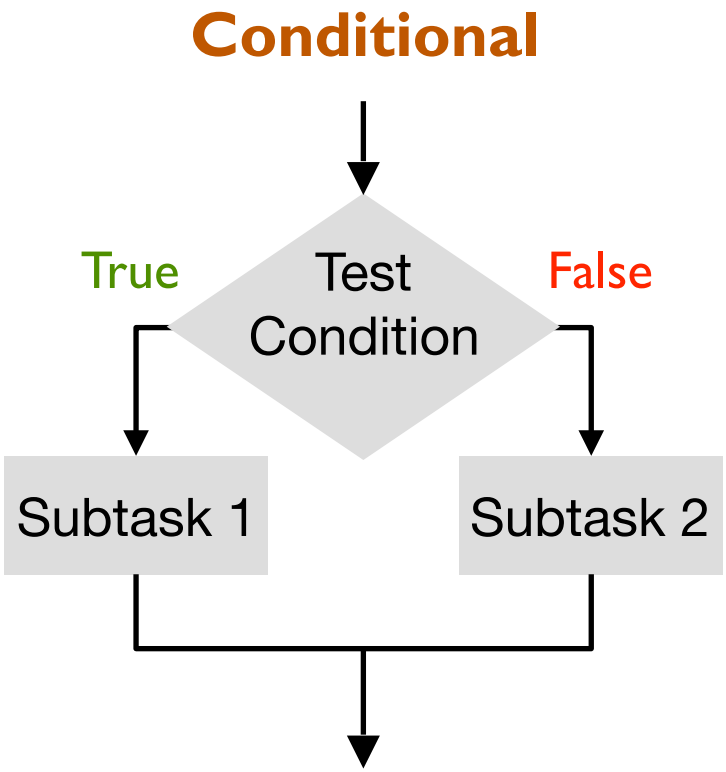
- Both conditional and iterative programming requires altering the sequence of instructions (control flow)
- We need a set of *control instructions* to do so




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

Three Basic Programming Constructs

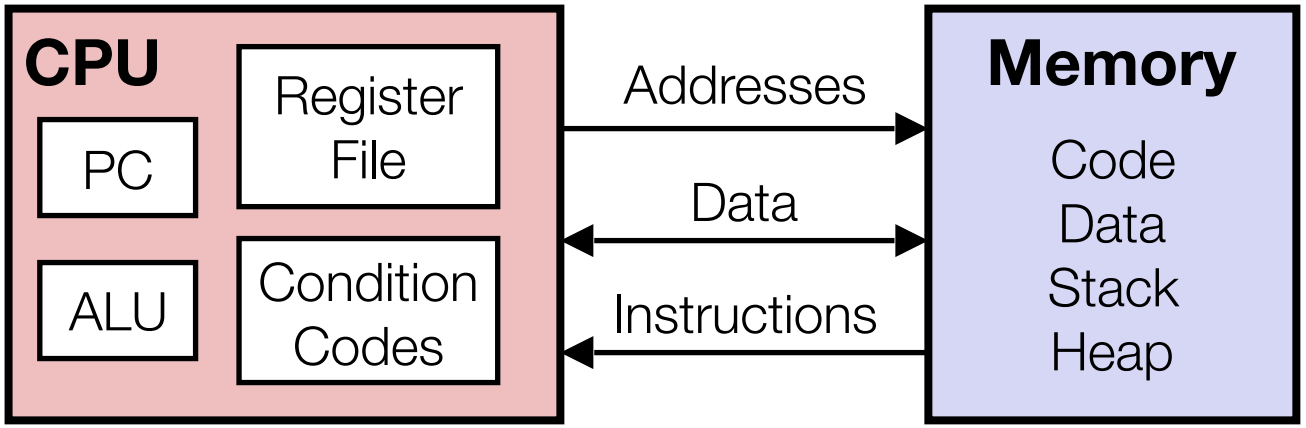
- Both conditional and iterative programming requires altering the sequence of instructions (control flow)
- We need a set of *control instructions* to do so
- Two fundamental questions:
  - How to test condition and how to represent test results?
  - How to alter control flow according to the test results?




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

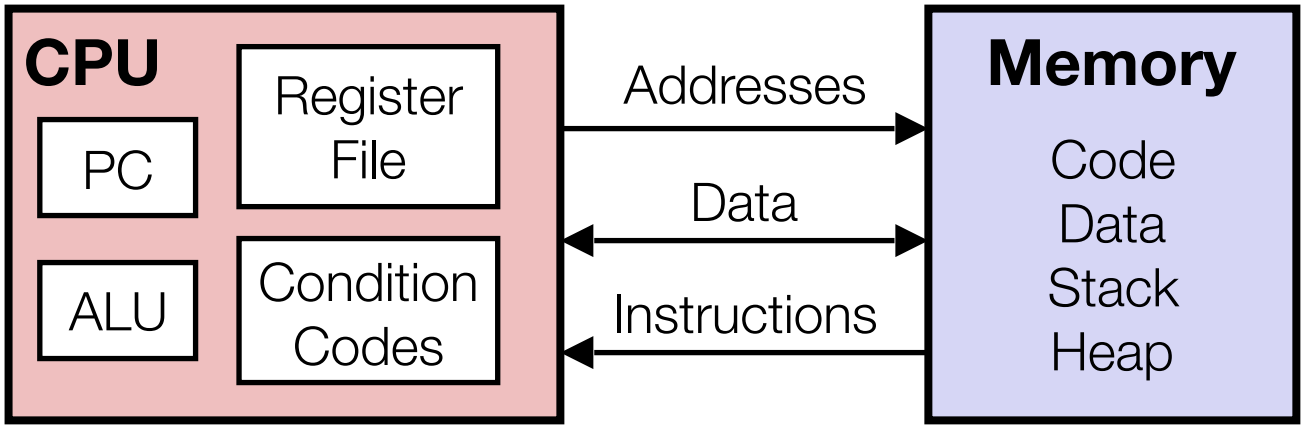
Condition Codes Hold Test Results

Assembly Programmer's Perspective of a Computer



Condition Codes Hold Test Results

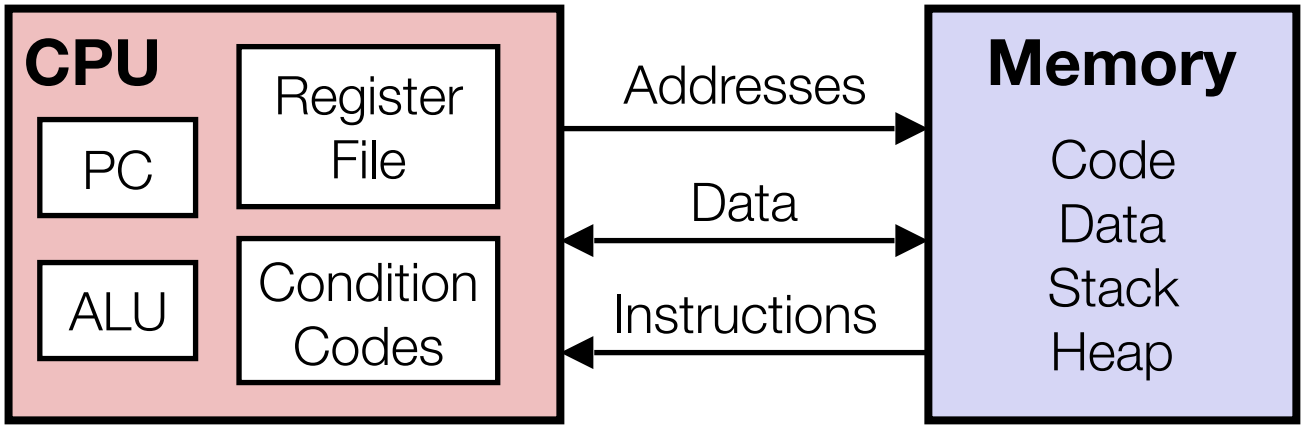
Assembly Programmer's Perspective of a Computer



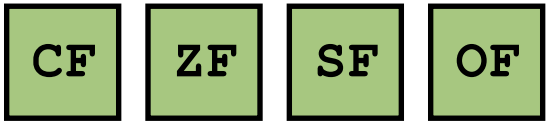
- Condition Codes
  - Hold the status of most recent test
  - 4 common condition codes in x86-64
  - A set of special registers (more often: bits in one single register)
  - Sometimes also called: Status Register, Flag Register

Condition Codes Hold Test Results

Assembly Programmer's Perspective of a Computer

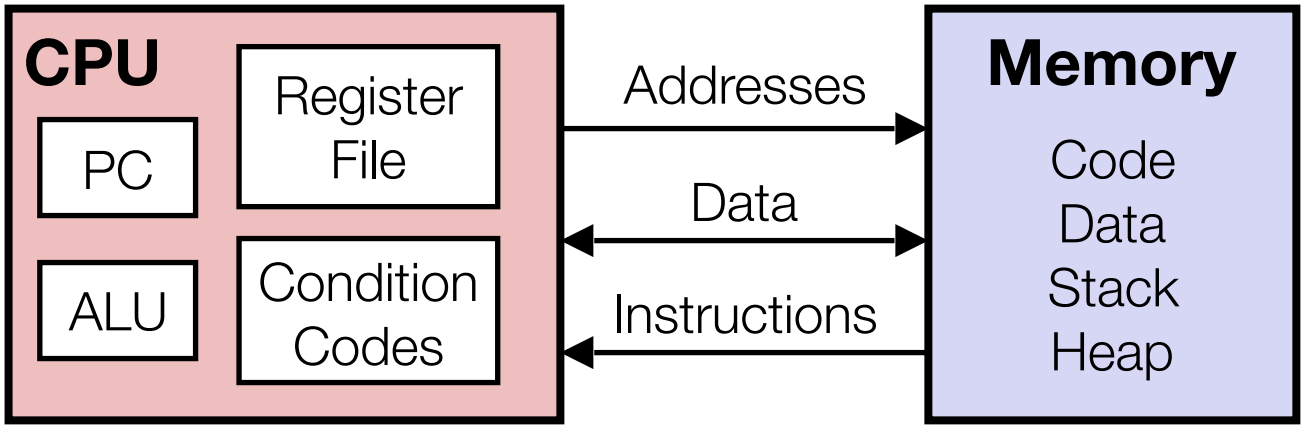


- Condition Codes
  - Hold the status of most recent test
  - 4 common condition codes in x86-64
  - A set of special registers (more often: bits in one single register)
  - Sometimes also called: Status Register, Flag Register

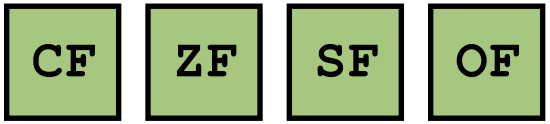


Condition Codes Hold Test Results

Assembly Programmer's Perspective of a Computer



- Condition Codes
  - Hold the status of most recent test
  - 4 common condition codes in x86-64
  - A set of special registers (more often: bits in one single register)
  - Sometimes also called: Status Register, Flag Register



CF Carry Flag (for unsigned)

ZF Zero Flag

SF Sign Flag

OF Overflow Flag (for signed)

Explicit Set Condition Codes: Compare

`cmpq` *Src2*, *Src1*

Explicit Set Condition Codes: Compare

`cmpq` *Src2*, *Src1*

- Explicit Setting by Compare Instruction

Explicit Set Condition Codes: Compare

`cmpq` *Src2*, *Src1*

- Explicit Setting by Compare Instruction
  - **`cmpq b,a`** like computing `a-b`, and sets condition codes accordingly

Explicit Set Condition Codes: Compare

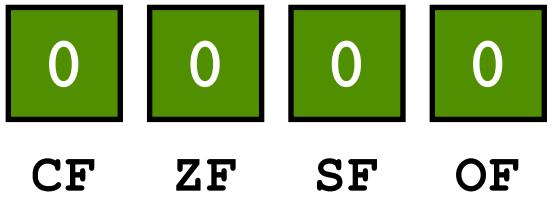
`cmpq` *Src2*, *Src1*

- Explicit Setting by Compare Instruction
  - **`cmpq b,a`** like computing `a-b`, and sets condition codes accordingly

11111111 10000000

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

(assuming 8-bit word size here)



Explicit Set Condition Codes: Compare

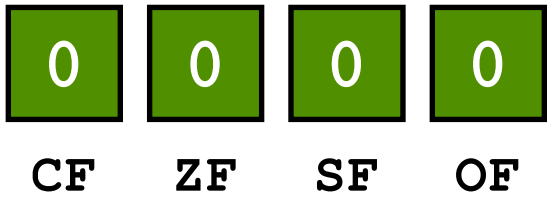
`cmpq` *Src2*, *Src1*

- Explicit Setting by Compare Instruction
  - **`cmpq b,a`** like computing `a-b`, and sets condition codes accordingly
  - **CF (Carry Flag):** Treat `a` and `b` as unsigned value, set CF if `a-b` generates a borrow out of the most significant bit (i.e., `a < b`). Effectively detecting overflow for unsigned

11111111 10000000

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

(assuming 8-bit word size here)



Explicit Set Condition Codes: Compare

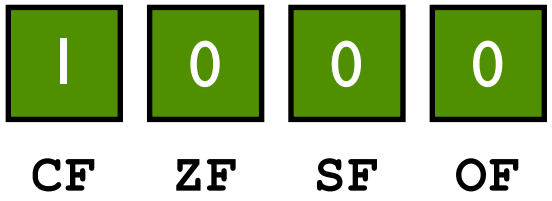
`cmpq` *Src2*, *Src1*

- Explicit Setting by Compare Instruction
  - **`cmpq b,a`** like computing `a-b`, and sets condition codes accordingly
  - **CF (Carry Flag):** Treat `a` and `b` as unsigned value, set CF if `a-b` generates a borrow out of the most significant bit (i.e., `a < b`). Effectively detecting overflow for unsigned

11111111 10000000

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

(assuming 8-bit word size here)



Explicit Set Condition Codes: Compare

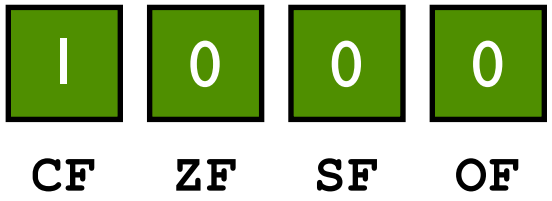
`cmpq` *Src2*, *Src1*

- Explicit Setting by Compare Instruction
  - **`cmpq b,a`** like computing `a-b`, and sets condition codes accordingly
  - **CF (Carry Flag):** Treat `a` and `b` as unsigned value, set CF if `a-b` generates a borrow out of the most significant bit (i.e., `a < b`). Effectively detecting overflow for unsigned
  - **ZF (Zero Flag):** set if `a == b` (i.e., bit patterns are the same)

11111111 10000000

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

(assuming 8-bit word size here)



Explicit Set Condition Codes: Compare

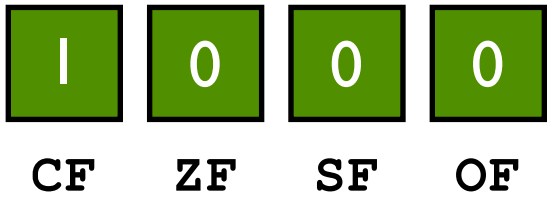
`cmpq` *Src2*, *Src1*

- Explicit Setting by Compare Instruction
  - **`cmpq b,a`** like computing `a-b`, and sets condition codes accordingly
  - **CF (Carry Flag):** Treat `a` and `b` as unsigned value, set CF if `a-b` generates a borrow out of the most significant bit (i.e., `a < b`). Effectively detecting overflow for unsigned
  - **ZF (Zero Flag):** set if `a == b` (i.e., bit patterns are the same)
  - **SF (Sign Flag):** set if `(a-b)<0` (i.e., most significant bit is 1)

11111111 10000000

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

(assuming 8-bit word size here)



Explicit Set Condition Codes: Compare

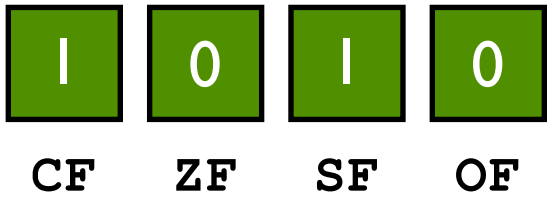
`cmpq` *Src2*, *Src1*

- Explicit Setting by Compare Instruction
  - **`cmpq b,a`** like computing `a-b`, and sets condition codes accordingly
  - **CF (Carry Flag):** Treat `a` and `b` as unsigned value, set CF if `a-b` generates a borrow out of the most significant bit (i.e., `a < b`). Effectively detecting overflow for unsigned
  - **ZF (Zero Flag):** set if `a == b` (i.e., bit patterns are the same)
  - **SF (Sign Flag):** set if `(a-b)<0` (i.e., most significant bit is 1)

11111111 10000000

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

(assuming 8-bit word size here)



Explicit Set Condition Codes: Compare

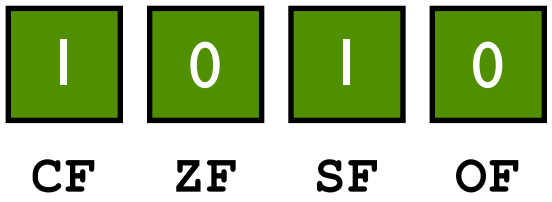
`cmpq` *Src2*, *Src1*

- Explicit Setting by Compare Instruction
  - **`cmpq b,a`** like computing `a-b`, and sets condition codes accordingly
  - **CF (Carry Flag):** Treat `a` and `b` as unsigned value, set CF if `a-b` generates a borrow out of the most significant bit (i.e., `a < b`). Effectively detecting overflow for unsigned
  - **ZF (Zero Flag):** set if `a == b` (i.e., bit patterns are the same)
  - **SF (Sign Flag):** set if `(a-b)<0` (i.e., most significant bit is 1)
  - **OF (Overflow Flag)** set if `a-b` overflows (treat *a* and *b* as signed)
    `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

11111111 10000000

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

(assuming 8-bit word size here)



Explicit Set Condition Codes: Test

`testq` *Src2*, *Src1*

- Explicit Setting by Test Instruction
  - **`test b,a`** like computing `a&b`, but instead of setting the result, it sets condition codes
  - **ZF (Zero Flag):** set if `a & b == 0`
  - **SF (Sign Flag):** set if `a & b < 0`
  - OF and CF are always set to 0

Implicit Set Condition Codes

`addq` *Src*, *Dest*

Implicit Set Condition Codes

`addq` *Src*, *Dest*

- Condition Codes could also be implicitly set by arithmetic operations (think of it as side effect)

Implicit Set Condition Codes

`addq` *Src*, *Dest*

- Condition Codes could also be implicitly set by arithmetic operations (think of it as side effect)
  - Assume *Src* is a, *Dest* is b, addition result is t = a + b (after truncation)

Implicit Set Condition Codes

`addq` *Src*, *Dest*

- Condition Codes could also be implicitly set by arithmetic operations (think of it as side effect)
  - Assume *Src* is a, *Dest* is b, addition result is t = a + b (after truncation)
  - **CF** set if carry out from most significant bit (unsigned overflow)

Implicit Set Condition Codes

`addq` *Src*, *Dest*

- Condition Codes could also be implicitly set by arithmetic operations (think of it as side effect)
  - Assume *Src* is a, *Dest* is b, addition result is t = a + b (after truncation)
  - **CF** set if carry out from most significant bit (unsigned overflow)
  - **ZF** set if `t == 0`

Implicit Set Condition Codes

`addq` *Src*, *Dest*

- Condition Codes could also be implicitly set by arithmetic operations (think of it as side effect)
  - Assume *Src* is a, *Dest* is b, addition result is t = a + b (after truncation)
  - **CF** set if carry out from most significant bit (unsigned overflow)
  - **ZF** set if `t == 0`
  - **SF** set if `t < 0` (as signed, i.e., MSB is 1)

Implicit Set Condition Codes

`addq` *Src*, *Dest*

- Condition Codes could also be implicitly set by arithmetic operations (think of it as side effect)
  - Assume *Src* is a, *Dest* is b, addition result is t = a + b (after truncation)
  - **CF** set if carry out from most significant bit (unsigned overflow)
  - **ZF** set if `t == 0`
  - **SF** set if `t < 0` (as signed, i.e., MSB is 1)
  - **OF** set if signed overflow

```
(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
```

Today: Compute and Control Instructions

- Arithmetic & logical operations
- Control: Condition codes
- Conditional branches (**`if`**... **`else`**...)
- Loops (**`for`**, **`while`**)
- Switch Statements (**`case`**... **`switch`**...)

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.