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???

```c
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```
**How Does Pointer Work in C??**

```c
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```
How Does Pointer Work in C???

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

<table>
<thead>
<tr>
<th>C Variable</th>
<th>Memory Content</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0x11</td>
</tr>
</tbody>
</table>

---

Memory Address:

0x10

0x11

0x16

...
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char *c;
c = &a;
b += (*c);
```
## How Does Pointer Work in C???

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

<table>
<thead>
<tr>
<th>C Variable</th>
<th>Memory Content</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0x11</td>
</tr>
</tbody>
</table>

- The content of a pointer variable is memory address.

![Memory representation](Image)
How Does Pointer Work in C???

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

- The content of a pointer variable is memory address.

<table>
<thead>
<tr>
<th>C Variable</th>
<th>Memory Content</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0x11</td>
</tr>
<tr>
<td>c</td>
<td>random</td>
<td>0x16</td>
</tr>
</tbody>
</table>

...
How Does Pointer Work in C???

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

- The content of a pointer variable is memory address.

<table>
<thead>
<tr>
<th>C Variable</th>
<th>Memory Content</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0x11</td>
</tr>
<tr>
<td>c</td>
<td>random</td>
<td>0x16</td>
</tr>
</tbody>
</table>

...
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.

<table>
<thead>
<tr>
<th>C Variable</th>
<th>Memory Content</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0x11</td>
</tr>
<tr>
<td>c</td>
<td>random</td>
<td>0x16</td>
</tr>
</tbody>
</table>

Memory Content
How Does Pointer Work in C???

```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.

<table>
<thead>
<tr>
<th>C Variable</th>
<th>Memory Content</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0x11</td>
</tr>
<tr>
<td>c</td>
<td>0x10</td>
<td>0x16</td>
</tr>
</tbody>
</table>

Memory Address:

- `0x10`
- `0x11`

...
How Does Pointer Work in C???

```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???

```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)

<table>
<thead>
<tr>
<th>C Variable</th>
<th>Memory Content</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0x11</td>
</tr>
<tr>
<td>c</td>
<td>0x10</td>
<td>0x16</td>
</tr>
</tbody>
</table>

Memory Address Table:

- Memory Address: 0x10
- Memory Address: 0x11
- Memory Address: 0x16

Memory Content Table:

- Memory Content: 4
- Memory Content: 3
- Memory Content: 0x10
How Does Pointer Work in C???

```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

  \[
  \text{movq}\ (%\text{rcx}),\%\text{rax};\ //\ \text{address} = \%\text{rcx}
  \]
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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%rdx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td>0x0100</td>
</tr>
</tbody>
</table>
## Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0x8(%rdx))</td>
<td>(0xf000 + 0x8)</td>
<td>(0xf008)</td>
</tr>
<tr>
<td>((%rdx, %rcx))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((%rdx, %rcx, 4))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0x80(, %rdx, 2))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%rdx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td>0x0100</td>
</tr>
</tbody>
</table>
Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
%rdx \quad 0xf000 \\
%rcx \quad 0x0100
\]
# Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

### Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdx</td>
<td>0xf000</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x0100</td>
</tr>
</tbody>
</table>
Address Computation Instruction

```
leaq 4(%rsi,%rdi,2), %rax
```
Address Computation Instruction

\[
\text{leaq } 4(\%rsi, \%rdi, 2), \%rax \\
\%rax = \%rsi + \%rdi * 2 + 4
\]
Address Computation Instruction

\[ \text{leaq } 4(\%rsi,\%rdi,2), \%rax \]

\[ \%rax = \%rsi + \%rdi \times 2 + 4 \]

- `leaq` \text{Src, Dst}
  - \text{Src} is address mode expression
  - Set \text{Dst} to address denoted by expression
  - No actual memory reference is made
Address Computation Instruction

\texttt{leaq 4(\%rsi,\%rdi,2), \%rax}

\[ \%rax = \%rsi + \%rdi \times 2 + 4 \]

\begin{itemize}
  \item \texttt{leaq Src, Dst}
    \begin{itemize}
      \item \textit{Src} is address mode expression
      \item Set \textit{Dst} to address denoted by expression
      \item No actual memory reference is made
    \end{itemize}
  \item Uses
    \begin{itemize}
      \item Computing addresses without a memory reference
        \begin{itemize}
          \item E.g., translation of \texttt{p = \&x[i];}
        \end{itemize}
    \end{itemize}
\end{itemize}
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

```c
long m12(long x) {
    return x*12;
}
```
Address Computation Instruction

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

```c
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

CPU
- PC
- ALU

Condition Codes

Register File

Addresses

Data

Instructions

Memory
- Code
- Data
- Stack
- Heap
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)

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>src, dest</td>
<td>Dest = Dest + Src</td>
</tr>
</tbody>
</table>
### Some Arithmetic Operations (2 Operands)

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq src, dest</td>
<td>Dest = Dest + Src</td>
<td></td>
</tr>
</tbody>
</table>

```c
addq %rax, %rbx
```
### Some Arithmetic Operations (2 Operands)

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq</code> <code>src, dest</code></td>
<td><code>Dest = Dest + Src</code></td>
<td></td>
</tr>
</tbody>
</table>

- **Addition**

  \[
  u \quad \cdot \cdot \cdot \\
  + \quad v \quad \cdot \cdot \cdot \\
  \hline
  u + v \quad \cdot \cdot \cdot \\
  \hline
  \text{TAdd}_w(u, v) \quad \cdot \cdot \cdot \\
  \hline
  \text{addq } \%rax, \%rbx \\
  \hline
  
  \]
Some Arithmetic Operations (2 Operands)

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq</code></td>
<td><code>src, dest</code></td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Truncation if overflow,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set carry bit (more later…)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cc}
  u & \cdots \\
  + & v \\
  u + v & \cdots \\
  \text{TAdd}_w(u, v) & \cdots \\
\end{array}
\]

\[
\text{addq } \%rax, \%rbx \quad \%rbx = \%rax + \%rbx
\]

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

---

Carnegie Mellon

---

[699x7]11

[635x527]
Some Arithmetic Operations (2 Operands)

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq src, dest</code></td>
<td><code>Dest = Dest + Src</code></td>
<td></td>
</tr>
<tr>
<td><code>subq src, dest</code></td>
<td><code>Dest = Dest - Src</code></td>
<td></td>
</tr>
<tr>
<td><code>imulq src, dest</code></td>
<td><code>Dest = Dest * Src</code></td>
<td></td>
</tr>
<tr>
<td><code>salq src, dest</code></td>
<td><code>Dest = Dest &lt;&lt; Src</code></td>
<td>Also called <code>shlq</code></td>
</tr>
<tr>
<td><code>sarq src, dest</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
<td>Arithmetic shift</td>
</tr>
<tr>
<td><code>shrq src, dest</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
<td>Logical shift</td>
</tr>
<tr>
<td><code>xorq src, dest</code></td>
<td><code>Dest = Dest ^ Src</code></td>
<td></td>
</tr>
<tr>
<td><code>andq src, dest</code></td>
<td><code>Dest = Dest &amp; Src</code></td>
<td></td>
</tr>
<tr>
<td><code>orq src, dest</code></td>
<td>`Dest = Dest</td>
<td>Src`</td>
</tr>
</tbody>
</table>
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

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

```assembly
#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

```c
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

\[
\begin{array}{c}
010 \\
+ \quad 101 \\
\hline
111
\end{array}
\]

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 (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;
}
```

---

Bit-level

<table>
<thead>
<tr>
<th>010</th>
<th>101</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>

Signed

<table>
<thead>
<tr>
<th>2</th>
<th>-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

#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;
}
```

### Bit-level

<table>
<thead>
<tr>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>101</td>
<td>-3</td>
</tr>
<tr>
<td>111</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
#x in %rdx, y in %rax addq %rdx, %rax

<table>
<thead>
<tr>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
#x in %rdx, y in %rax addq %rdx, %rax
Some Arithmetic Operations (1 Operand)

- Unary Instructions (one operand)

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>incq dest</td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td>decq dest</td>
<td>Dest = Dest - 1</td>
</tr>
<tr>
<td>negq dest</td>
<td>Dest = -Dest</td>
</tr>
<tr>
<td>notq dest</td>
<td>Dest = ~Dest</td>
</tr>
</tbody>
</table>
Some Arithmetic Operations (1 Operand)

- Unary Instructions (one operand)

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>incq</code> dest</td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td><code>decq</code> dest</td>
<td>Dest = Dest - 1</td>
</tr>
<tr>
<td><code>negq</code> dest</td>
<td>Dest = -Dest</td>
</tr>
<tr>
<td><code>notq</code> dest</td>
<td>Dest = ~Dest</td>
</tr>
</tbody>
</table>

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

Subtask 1

Subtask 2

\[ a = x + y; \]
\[ y = a - c; \]

...
Three Basic Programming Constructs

**Sequential**

1. Subtask 1
2. Subtask 2

**Conditional**

- **Test Condition**
  - **True**: Subtask 1
  - **False**: Subtask 2

### Code Examples

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

if (x > y) r = x - y;
else r = y - x;
```
Three Basic Programming Constructs

**Sequential**
- Subtask 1
- Subtask 2

**Conditional**
- Test Condition
- True: Subtask 1
- False: Subtask 2

**Iterative**
- Test Condition
- True: Subtask
- False: Subtask

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

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

```plaintext
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

\[
\text{if} \ (x > y) \ r = x - y; \\
\text{else} \ r = y - x;
\]
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?

\[
\text{if (x > y) } r = x - y; \\
\text{else } r = y - x;
\]
Condition Codes Hold Test Results

Assembly Programmer’s Perspective of a Computer

CPU
- PC
- ALU
  - Register File
  - Condition Codes

Memory
- Addresses
- Data
- Instructions
- Code
- Data
- Stack
- Heap
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 ZF SF OF
Condition Codes Hold Test Results

- 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

\texttt{cmpq} \textit{Src2, Src1}
Explicit Set Condition Codes: Compare

\texttt{cmpq} \textit{Src2, Src1}

- Explicit Setting by Compare Instruction
Explicit Set Condition Codes: Compare

\[
\text{cmpq } Src2, Src1
\]

- Explicit Setting by Compare Instruction
  - \text{cmpq } b,a \text{ like computing } a-b, \text{ and sets condition codes accordingly}
Explicit Set Condition Codes: Compare

\texttt{cmpq} \texttt{Src2, Src1}

• Explicit Setting by Compare Instruction
  • \texttt{cmpq b,a} like computing \( a-b \), and sets condition codes accordingly

\begin{verbatim}
11111111 10000000
\end{verbatim}

\texttt{cmpq 0xFF, 0x80}

(assuming 8-bit word size here)
Explicit Set Condition Codes: Compare

\texttt{cmpq} \textit{Src2, Src1}

• Explicit Setting by Compare Instruction
  • \texttt{cmpq b, a} like computing \(a - b\), and sets condition codes accordingly
  • \textbf{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

\begin{equation}
11111111\ 10000000
\end{equation}

\texttt{cmpq} \textit{0xFF, 0x80}

(assuming 8-bit word size here)
Explicit Set Condition Codes: Compare

\texttt{cmpq} \textit{Src2, Src1}

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

\[
\begin{array}{c}
11111111 \\
10000000 \\
\end{array}
\]

\texttt{cmpq 0xFF, 0x80}

(assuming 8-bit word size here)
Explicit Set Condition Codes: Compare

\( \text{cmpq } Src2, Src1 \)

- Explicit Setting by Compare Instruction
  - \( \text{cmpq } b, a \) 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)

\[
\begin{array}{cccc}
11111111 & 10000000 \\
\text{cmpq } 0xFF, 0x80 \\
\end{array}
\]

(assuming 8-bit word size here)
Explicit Set Condition Codes: Compare

\texttt{cmpq \textit{Src2}, \textit{Src1}}

- Explicit Setting by Compare Instruction

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

\begin{verbatim}
11111111 10000000
cmpq 0xFF, 0x80
\end{verbatim}
Explicit Set Condition Codes: Compare

\texttt{cmpq Src2, Src1}

- Explicit Setting by Compare Instruction
  - \texttt{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)

\begin{verbatim}
11111111 10000000
\end{verbatim}

\texttt{cmpq 0xFF, 0x80}

(assuming 8-bit word size here)
Explicit Set Condition Codes: Compare

\texttt{cmpq} \textit{Src2, Src1}

- Explicit Setting by Compare Instruction
  - \texttt{cmpq b, a} like computing \( a - b \), and sets condition codes accordingly
  - \textbf{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
  - \textbf{ZF (Zero Flag)}: set if \( a == b \) (i.e., bit patterns are the same)
  - \textbf{SF (Sign Flag)}: set if \( (a-b) < 0 \) (i.e., most significant bit is 1)
  - \textbf{OF (Overflow Flag)} set if \( a-b \) overflows (treat \( a \) and \( b \) as signed)
    \[(a>0 \&\& b<0 \&\& (a-b)<0) || (a<0 \&\& b>0 \&\& (a-b)>0)\]

\begin{align*}
11111111 & 10000000 \\
\text{cmpq} \ 0xFF, \ 0x80 & \text{ (assuming 8-bit word size here)}
\end{align*}
Explicit Set Condition Codes: Test

\texttt{testq Src2, Src1}

- Explicit Setting by Test Instruction
  - \texttt{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

\texttt{addq \textit{Src}, Dest}
Implicit Set Condition Codes

\texttt{addq \textit{Src}, \textit{Dest}}

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

\[ \text{addq } \text{Src, Dest} \]

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

\texttt{addq} \textit{Src, Dest}

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

\texttt{addq} \textit{Src}, \textit{Dest}

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

\texttt{addq} \texttt{Src, Dest}

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

\texttt{addq} \textit{Src, Dest}

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

```c
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

```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

```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
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
</tr>
<tr>
<td>%rsi</td>
<td>y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Conditional Branch Example

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

```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
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
</tr>
<tr>
<td>%rsi</td>
<td>y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

Labels are symbolic names used to refer to instruction addresses.