CS 33

Machine Programming (1)
Data Types on Intel x86

• “Integer” data of 1, 2, or 4 bytes (plus 8 bytes on x86-64)
  – data values
    » whether signed or unsigned depends on interpretation
  – addresses (untyped pointers)

• Floating-point data of 4, 8, or 10 bytes

• No aggregate types such as arrays or structures
  – just contiguously allocated bytes in memory
Operand Size

- byte
- short
- long
- quad

- Rather than mov ...
  - movb
  - movs
  - movl
  - movq (x86-64 only)
### General-Purpose Registers (IA32)

#### General Purpose

<table>
<thead>
<tr>
<th>Register</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx</td>
<td>counter</td>
</tr>
<tr>
<td>%edx</td>
<td>data</td>
</tr>
<tr>
<td>%ebx</td>
<td>base</td>
</tr>
<tr>
<td>%esi</td>
<td>source index</td>
</tr>
<tr>
<td>%edi</td>
<td>destination index</td>
</tr>
<tr>
<td>%esp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>base pointer</td>
</tr>
</tbody>
</table>

**16-bit virtual registers (backwards compatibility)**
Moving Data: IA32

• Moving data
  movl source, dest

• Operand types
  – **Immediate:** constant integer data
    » example: $0x400, $-533
    » like C constant, but prefixed with ‘$’
    » encoded with 1, 2, or 4 bytes
  – **Register:** one of 8 integer registers
    » example: %eax, %edx
    » but %esp and %ebp reserved for special use
    » others have special uses for particular instructions
  – **Memory:** 4 consecutive bytes of memory at address given by register(s)
    » simplest example: (%eax)
    » various other “address modes”
### movl Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Reg</strong></td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Mem</strong></td>
<td>movl %eax,(%edx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td><strong>Reg</strong></td>
<td>movl (%eax),%edx</td>
<td>temp = *p;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td><strong>Mem</strong></td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td><strong>Imm</strong></td>
<td><strong>Reg</strong></td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
</tbody>
</table>

**Cannot (normally) do memory-memory transfer with a single instruction**
Simple Memory Addressing Modes

- Normal \( (R) \) \( \text{Mem}[\text{Reg}[R]] \)
  - register \( R \) specifies memory address

\[
\text{movl} \text{ } (\%ecx),\%eax
\]

- Displacement \( D(R) \) \( \text{Mem}[\text{Reg}[R]+D] \)
  - register \( R \) specifies start of memory region
  - constant displacement \( D \) specifies offset

\[
\text{movl} \text{ } 8(\%ebp),\%edx
\]
Using Simple Addressing Modes

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

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 8(%ebp), %edx
    movl 12(%ebp), %ecx
    movl (%edx), %ebx
    movl (%ecx), %eax
    movl %eax, (%edx)
    movl %ebx, (%ecx)

    popl %ebx
    popl %ebp
    ret
```
### Understanding Swap

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

#### Stack (in memory)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>yp</td>
</tr>
<tr>
<td>8</td>
<td>xp</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebx</td>
</tr>
</tbody>
</table>

#### Register Value

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>xp</td>
</tr>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<tr>
<td>%ebx</td>
<td>t0</td>
</tr>
<tr>
<td>%eax</td>
<td>t1</td>
</tr>
</tbody>
</table>

```assembly
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *yp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
```
Understanding Swap

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx</th>
<th>%ecx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x104</td>
<td>0x104</td>
</tr>
</tbody>
</table>

movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx # ebx = *xp (t0)
movl (%ecx), %eax # eax = *yp (t1)
movl %eax, (%edx) # *xp = t1
movl %ebx, (%ecx) # *yp = t0
Understanding Swap

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>0</td>
</tr>
<tr>
<td>0x120</td>
<td>12</td>
</tr>
<tr>
<td>0x11c</td>
<td>8</td>
</tr>
<tr>
<td>0x118</td>
<td>4</td>
</tr>
<tr>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

 Movl 8(%ebp), %edx # edx = xp
 Movl 12(%ebp), %ecx # ecx = yp
 Movl (%edx), %ebx # ebx = *xp (t0)
 Movl (%ecx), %eax # eax = *yp (t1)
 Movl %eax, (%edx) # *xp = t1
 Movl %ebx, (%ecx) # *yp = t0
Understanding Swap

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
<td>0x108</td>
</tr>
<tr>
<td>0x118</td>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>0x114</td>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx</th>
<th>%ecx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>0x120</td>
<td>0x120</td>
</tr>
</tbody>
</table>

```
movl 8(%%ebp), %edx  # edx = xp
movl 12(%%ebp), %ecx  # ecx = yp
movl (%edx), %ebx    # ebx = *xp (t0)
movl (%ecx), %eax    # eax = *yp (t1)
movl %eax, (%edx)    # *xp = t1
movl %ebx, (%ecx)    # *yp = t0
```
### Understanding Swap

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx</th>
<th>%ecx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x124</td>
<td>0x120</td>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td>0x104</td>
</tr>
</tbody>
</table>

#### Register Values

- **%edx**: 0x124
- **%ecx**: 0x120
- **%ebx**: 123
- **%esi**: 0
- **%edi**: 0
- **%esp**: 0x100
- **%ebp**: 0x104

#### Code Snippet

```assembly
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *yp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
```

#### Offset Table

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

#### Address Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>123</td>
</tr>
<tr>
<td>0x120</td>
<td>456</td>
</tr>
<tr>
<td>0x11c</td>
<td>0x110</td>
</tr>
<tr>
<td>0x118</td>
<td>0x10c</td>
</tr>
<tr>
<td>0x114</td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

---

**CS33 Intro to Computer Systems**

X–13
Understanding Swap

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

```assembly
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *yp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
```

Address Offset
---
123 0x124
456 0x120
-4  0x11c
  4  0x118
   8  0x114
  12  0x110
  yp  0x120
  xp  0x124
    4 Rtn adr
  0x10c
  0x108
  0x104
  0x100

Understanding Swap

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x124</td>
</tr>
<tr>
<td>%edx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ecx</td>
<td>123</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x104</td>
</tr>
<tr>
<td>%esi</td>
<td>0x100</td>
</tr>
<tr>
<td>%edi</td>
<td>0x108</td>
</tr>
<tr>
<td>%esp</td>
<td>0x110</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x114</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>0</td>
</tr>
</tbody>
</table>

```assembly
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *yp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
```
Understanding Swap

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>456</td>
</tr>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x11c</td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

- %eax = 456
- %edx = 0x124
- %ecx = 0x120
- %ebx = 123
- %esi
- %edi
- %esp
- %ebp = 0x104

Example code:

```assembly
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
```
Quiz 1

movl -4(%ebp), %eax
movl (%eax), %eax
movl (%eax), %eax
movl %eax, -8(%ebp)

Which C statements best describe the assembler code?

// a
int x;
int y;
y = x;

// b
int *x;
int y;
y = *x;

// c
int **x;
int y;
y = **x;

// d
int ***x;
int y;
y = ***x;
Complete Memory-Addressing Modes

• Most general form

\[ D(Rb,Ri,S) \rightarrow \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]+D] \]

  - **D:** constant “displacement”
  - **Rb:** base register: any of 8 integer registers
  - **Ri:** index register: any, except for %esp
    » unlikely you’d use %ebp either
  - **S:** scale: 1, 2, 4, or 8

• Special cases

\[
\begin{align*}
(Rb,Ri) & \rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \\
D(Rb,Ri) & \rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] \\
(Rb,Ri,S) & \rightarrow \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]] \\
D & \rightarrow \text{Mem}[D]
\end{align*}
\]
# Address-Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8 (%edx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%edx, %ecx)</td>
<td>0xf000 + 0x0100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%edx, %ecx, 4)</td>
<td>0xf000 + 4*0x0100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80 (,%edx, 2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%edx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x0100</td>
</tr>
</tbody>
</table>
Address-Computation Instruction

• `leal src, dest`
  – `src` is address mode expression
  – set `dest` to address denoted by expression

• Uses
  – computing addresses without a memory reference
    » e.g., translation of `p = &x[i];`
  – computing arithmetic expressions of the form `x + k*y`
    » `k = 1, 2, 4, or 8`

• Example

```c
int mul12(int x) {
    return x*12;
}
```

Converted to ASM by compiler:

```
movl 8(%ebp), %eax # get arg
leal (%eax,%eax,2), %eax # t <- x+x*2
sall $2, %eax # return t<<2
```
Quiz 2

What value ends up in %ecx?

movl $1000, %eax
movl $1, %ebx
movl 2(%eax, %ebx, 4), %ecx

a) 0x02030405
b) 0x05040302
c) 0x06070809
d) 0x09080706

Hint:

%eax → 1000: 0x00
1000: 0x00
1001: 0x01
1002: 0x02
1003: 0x03
1004: 0x04
1005: 0x05
1006: 0x06
1007: 0x07
1008: 0x08
1009: 0x09
### x86-64 General-Purpose Registers

<table>
<thead>
<tr>
<th>a1</th>
<th>%rdi</th>
<th>%edi</th>
<th>a5</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td></td>
<td>a6</td>
</tr>
<tr>
<td>a2</td>
<td>%rsi</td>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>a3</td>
<td>%rdx</td>
<td>%edx</td>
<td></td>
</tr>
<tr>
<td>a4</td>
<td>%rcx</td>
<td>%ecx</td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rax</td>
<td>%eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Extend existing registers to 64 bits. Add 8 new ones.
- No special purpose for `%ebp/%rbp`
32-bit Instructions on x86-64

• `addl 4(%rdx), %eax`
  – memory address must be 64 bits
  – operands (in this case) are 32-bit
    » result goes into %eax
    • lower half of %rax
    • upper half is filled with zeroes
32-bit code for swap

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

**swap:**
- **Set Up**
  - `pushl %ebp`
  - `movl %esp,%ebp`
  - `pushl %ebx`

- **Body**
  - `movl 8(%ebp), %edx`
  - `movl 12(%ebp), %ecx`
  - `movl (%edx), %ebx`
  - `movl (%ecx), %eax`
  - `movl %eax, (%edx)`
  - `movl %ebx, (%ecx)`

- **Finish**
  - `popl %ebx`
  - `popl %ebp`
  - `ret`
64-bit code for swap

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

- Arguments passed in registers (why useful?)
  - first (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- No stack operations required
- 32-bit data
  - data held in registers %eax and %edx
  - `movl` operation
64-bit code for long int swap

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

- **64-bit data**
  - data held in registers `%rax` and `%rdx`
  - `movq` operation
    » “q” stands for quad-word
Turning C into Object Code

- Code in files p1.c p2.c
- Compile with command: gcc -O1 p1.c p2.c -o p
  » use basic optimizations (-O1)
  » put resulting binary in file p

```
text
  C program (p1.c p2.c)
  Compiler (gcc -S)

  Asm program (p1.s p2.s)
  Assembler (as)

  Object program (p1.o p2.o)
  Linker (ld)

  Executable program (p)
  Static libraries (.a)
```
Object Code

Code for `sum`

```
0x401040 <sum>:
  0x55
  0x89
  0xe5
  0x8b
  0x45
  0x0c
  0x03
  0x45
  0x08
  0x5d
  0xc3
```

- **Assembler**
  - translates `.s` into `.o`
  - binary encoding of each instruction
  - nearly-complete image of executable code
  - missing linkages between code in different files

- **Linker**
  - resolves references between files
  - combines with static run-time libraries
    - e.g., code for `printf`
  - some libraries are *dynamically linked*
    - linking occurs when program begins execution

- Total of 11 bytes
- Each instruction: 1, 2, or 3 bytes
- Starts at address `0x401040`
Instruction Format

<table>
<thead>
<tr>
<th>Instruction Prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 or 2 byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>Address displacement of 1, 2, or 4 bytes or none</td>
<td>Immediate data of 1, 2, or 4 bytes or none</td>
</tr>
</tbody>
</table>

- **Mod**: 7-6 bits
- **Reg/Opcode**: 5-4 bits
- **R/M**: 3-0 bits
- **Scale**: 7-6 bits
- **Index**: 5-4 bits
- **Base**: 3-0 bits
Disassembling Object Code

Disassembled

```
080483c4  <sum>:
  80483c4:  55  push  %ebp
  80483c5:  89 e5  mov  %esp,%ebp
  80483c7:  8b 45 0c  mov  0xc(%ebp),%eax
  80483ca:  03 45 08  add  0x8(%ebp),%eax
  80483cd:  5d  pop  %ebp
  80483ce:  c3  ret
```

• Disassembler
  
  `objdump -d <file>`
  – useful tool for examining object code
  – analyzes bit pattern of series of instructions
  – produces approximate rendition of assembly code
  – can be run on either executable or object (.o) file
## Alternate Disassembly

### Object

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>0x55</td>
</tr>
<tr>
<td></td>
<td>0x89</td>
</tr>
<tr>
<td></td>
<td>0xe5</td>
</tr>
<tr>
<td></td>
<td>0x8b</td>
</tr>
<tr>
<td></td>
<td>0x45</td>
</tr>
<tr>
<td></td>
<td>0x0c</td>
</tr>
<tr>
<td></td>
<td>0x03</td>
</tr>
<tr>
<td></td>
<td>0x45</td>
</tr>
<tr>
<td></td>
<td>0x08</td>
</tr>
<tr>
<td></td>
<td>0x5d</td>
</tr>
<tr>
<td></td>
<td>0xc3</td>
</tr>
</tbody>
</table>

### Disassembled

Dump of assembler code for function `sum`:

- **0x080483c4** `<sum+0>:` push `%ebp`
- **0x080483c5** `<sum+1>:` mov `%esp,%ebp`
- **0x080483c7** `<sum+3>:` mov `0xc(%ebp),%eax`
- **0x080483ca** `<sum+6>:` add `0x8(%ebp),%eax`
- **0x080483cd** `<sum+9>:` pop `%ebp`
- **0x080483ce** `<sum+10>:` ret

### Within gdb debugger

- `gdb <file>`
  - disassemble `sum`
  - disassemble procedure
- `x/11xb sum`
  - examine the 11 bytes starting at `sum`
How Many Instructions are There?

- We cover ~29
- Implemented by Intel:
  - 80 in original 8086 architecture
  - 7 added with 80186
  - 17 added with 80286
  - 33 added with 386
  - 6 added with 486
  - 6 added with Pentium
  - 1 added with Pentium MMX
  - 4 added with Pentium Pro
  - 8 added with SSE
  - 8 added with SSE2
  - 2 added with SSE3
  - 14 added with x86-64
  - 10 added with VT-x
  - 2 added with SSE4a
- Total: 198
- Doesn’t count:
  - floating-point instructions
  - SIMD instructions
  - AMD-added instructions
  - undocumented instructions
Some Arithmetic Operations

- **Two-operand instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addl</code></td>
<td><code>Dest = Dest + Src</code></td>
</tr>
<tr>
<td><code>subl</code></td>
<td><code>Dest = Dest - Src</code></td>
</tr>
<tr>
<td><code>imull</code></td>
<td><code>Dest = Dest * Src</code></td>
</tr>
<tr>
<td><code>sall</code></td>
<td><code>Dest = Dest &lt;&lt; Src</code></td>
</tr>
<tr>
<td><code>sar1</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
</tr>
<tr>
<td><code>shrl</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
</tr>
<tr>
<td><code>xorl</code></td>
<td><code>Dest = Dest ^ Src</code></td>
</tr>
<tr>
<td><code>andl</code></td>
<td><code>Dest = Dest &amp; Src</code></td>
</tr>
<tr>
<td><code>orl</code></td>
<td>`Dest = Dest</td>
</tr>
</tbody>
</table>

- watch out for argument order!
- no distinction between signed and unsigned int (why?)
Some Arithmetic Operations

• One-operand Instructions

  **incl**  Dest  = Dest + 1
  **decl**  Dest  = Dest - 1
  **negl**  Dest  = - Dest
  **notl**  Dest  = ~Dest

• See book for more instructions
Arithmetic Expression Example

```c
int arith(int x, int y, int z)
{
    int t1 = x + y;
    int t2 = z + t1;
    int t3 = x + 4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

arith:
```
leal (%rdi,%rsi), %eax
addl %edx, %eax
leal (%rsi,%rsi,2), %edx
sall $4, %edx
leal 4(%rdi,%rdx), %ecx
imull %ecx, %eax
ret
```
Understanding arith

```c
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

leal (%rdi,%rsi), %eax
addl %edx, %eax
leal (%rsi,%rsi,2), %edx
sall $4, %edx
leal 4(%rdi,%rdx), %ecx
imull %ecx, %eax
ret
Understanding arith

```c
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```assembly
leal (%rdi,%rsi), %eax          # eax = x+y (t1)
addl %edx, %eax                 # eax = t1+z (t2)
leal (%rsi,%rsi,2), %edx        # edx = 3*y (t4)
sall $4, %edx                   # edx = t4*16 (t4)
leal 4(%rdi,%rdx), %ecx         # ecx = x+4+t4 (t5)
imull %ecx, %eax                # eax *= t5 (rval)
ret
```
## Observations about `arith`

```c
int arith(int x, int y, int z) {
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

- Instructions in different order from C code
- Some expressions might require multiple instructions
- Some instructions might cover multiple expressions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Assembly</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lea (%rdi,%rsi), %eax</code></td>
<td><code>leal (%rdi,%rsi), %eax</code></td>
<td><code>eax = x+y</code> (t1)</td>
</tr>
<tr>
<td><code>addl %edx, %eax</code></td>
<td><code>addl %edx, %eax</code></td>
<td><code>eax = t1+z</code> (t2)</td>
</tr>
<tr>
<td><code>lea (%rsi,%rsi,2), %edx</code></td>
<td><code>leal (%rsi,%rsi,2), %edx</code></td>
<td><code>edx = 3*y</code> (t4)</td>
</tr>
<tr>
<td><code>sal $4, %edx</code></td>
<td><code>sal $4, %edx</code></td>
<td><code>edx = t4*16</code> (t4)</td>
</tr>
<tr>
<td><code>lea 4(%rdi,%rdx), %ecx</code></td>
<td><code>leal 4(%rdi,%rdx), %ecx</code></td>
<td><code>ecx = x+4+t4</code> (t5)</td>
</tr>
<tr>
<td><code>imull %ecx, %eax</code></td>
<td><code>imull %ecx, %eax</code></td>
<td><code>eax *= t5</code> (rval)</td>
</tr>
<tr>
<td><code>ret</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Another Example

```c
int logical(int x, int y) {
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

\[ 2^{13} = 8192, 2^{13} - 7 = 8185 \]

- `xorl %esi, %edi` # edi = x^y (t1)
- `sarl $17, %edi` # edi = t1>>17 (t2)
- `movl %edi, %eax` # eax = edi
- `andl $8185, %eax` # eax = t2 & mask (rval)
Quiz 3

• What is the final value in %esi?

```assembly
xorl %esi, %esi
incl %esi
sall %esi, %esi
addl %esi, %esi
```

a) 2
b) 4
c) 8
d) indeterminate