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

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

- **%eax** (%ax, %ah, %al)
- **%ecx** (%cx, %ch, %cl)
- **%edx** (%dx, %dh, %dl)
- **%ebx** (%bx, %bh, %bl)
- **%esi** (%si)
- **%edi** (%di)
- **%esp** (%sp)
- **%ebp** (%bp)

**Origin**
- accumulate
- counter
- data
- base
- source
- index
- destination
- index
- stack
- pointer
- base
- pointer

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

- **Moving data**
  
  \[
  \text{movl } \text{source, dest}
  \]

- **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>Imm</td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147,%eax</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl %eax,(%edx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

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

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

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

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

  \[
  \text{movl} \ 8(\%\text{ebp}),\%\text{edx}
  \]
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;
}
```

Register | Value
--- | ---
%edx | xp
%ecx | yp
%ebx | t0
%eax | t1

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>

- `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

```
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>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></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>

### Offset

<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></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
</tbody>
</table>

### Address

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

### Assembly Code

- `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>Func</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>%edx</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>%ecx</td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
<td>%ebp</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></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>%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></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>Register</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>

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

\[
\begin{align*}
\text{movl} & \ 8(\%ebp), \ %edx \quad \# \ edx = xp \\
\text{movl} & \ 12(\%ebp), \ %ecx \quad \# \ ecx = yp \\
\text{movl} & \ (%edx), \ %ebx \quad \# \ ebx = *xp \ (t0) \\
\text{movl} & \ (%ecx), \ %eax \quad \# \ eax = *yp \ (t1) \\
\text{movl} & \ %eax, \ (%edx) \quad \# \ *xp = t1 \\
\text{movl} & \ %ebx, \ (%ecx) \quad \# \ *yp = t0
\end{align*}
\]
## Understanding Swap

<table>
<thead>
<tr>
<th>Register</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>0x11c</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x118</td>
</tr>
<tr>
<td>%esi</td>
<td>0x114</td>
</tr>
<tr>
<td>%edi</td>
<td>0x110</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

### 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
```
## Quiz 1


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

### Which C statements best describe the assembler code?

<table>
<thead>
<tr>
<th>// a</th>
<th>// b</th>
<th>// c</th>
<th>// d</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
<td>int *x;</td>
<td>int **x;</td>
<td>int ***x;</td>
</tr>
<tr>
<td>int y;</td>
<td>int y;</td>
<td>int y;</td>
<td>int y;</td>
</tr>
<tr>
<td>y = x;</td>
<td>y = *x;</td>
<td>y = **x;</td>
<td>y = ***x;</td>
</tr>
</tbody>
</table>
Complete Memory-Addressing Modes

• Most general form
  
  \[ D(Rb,Ri,S) \rightarrow Mem[Reg[Rb]+S*Reg[Ri]+D] \]

  – 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

  \[
  \begin{align*}
  (Rb,Ri) & \rightarrow Mem[Reg[Rb]+Reg[Ri]] \\
  D(Rb,Ri) & \rightarrow Mem[Reg[Rb]+Reg[Ri]+D] \\
  (Rb,Ri,S) & \rightarrow Mem[Reg[Rb]+S*Reg[Ri]] \\
  D & \rightarrow Mem[D]
  \end{align*}
  \]
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 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%edx, %ecx, 4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80 (%edx, 2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</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

1009: 0x09
1008: 0x08
1007: 0x07
1006: 0x06
1005: 0x05
1004: 0x04
1003: 0x03
1002: 0x02
1001: 0x01
1000: 0x00

%eax → 1000: 0x00

Hint:
### x86-64 General-Purpose Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%eax</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%ebx</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%ecx</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%edx</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%esi</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%edi</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r8</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r8d</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r9</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r9d</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r10</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r10d</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r11</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r11d</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r12</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r12d</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r13</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r13d</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r14</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r14d</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r15</code></td>
<td>General purpose register</td>
</tr>
<tr>
<td><code>%r15d</code></td>
<td>General purpose register</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;
}
```

```assembly
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
```
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

---

`swap_1:`

```assembly
movq (%rdi), %rdx
movq (%rsi), %rax
movq %rax, (%rdi)
movq %rdx, (%rsi)
ret
```

- **Set Up**
- **Body**
- **Finish**
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)
```
Example

```c
int sum(int a, int b) {
    return (a+b);
}
```
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>Up to four prefixes of 1-byte each (optional)</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>

```
    7 6 5 3 2 0
   Mod Reg/Opcode R/M
```

```
    7 6 5 3 2 0
   Scale Index Base
```
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></td>
</tr>
<tr>
<td>0x55</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xe5</td>
<td></td>
</tr>
<tr>
<td>0x8b</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x0c</td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x5d</td>
<td></td>
</tr>
<tr>
<td>0xc3</td>
<td></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**
  ```plaintext
gdb <file>  
disable 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>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subl</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td>imull</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>sall</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarl</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shrl</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorl</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andl</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orl</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
  \[\text{incl} \quad \text{Dest} = \text{Dest} + 1\]
  \[\text{decl} \quad \text{Dest} = \text{Dest} - 1\]
  \[\text{negl} \quad \text{Dest} = -\text{Dest}\]
  \[\text{notl} \quad \text{Dest} = \sim\text{Dest}\]

• 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:
```assembly
    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 code:

- `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

```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
```
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;
}
```

$2^{13} = 8192$, $2^{13} - 7 = 8185$

```assembly
xorl %esi, %edi # edi = x^y (t1)
sar $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?

xorl %esi, %esi
incl %esi
sall %esi, %esi
addl %esi, %esi

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