Meet & Greet!

Come hang out with your TAs and Fellow Students (& eat free insomnia cookies)

When: TODAY!! 5-6 pm
Where: 3rd Floor Atrium, CIT
CS 33

Machine Programming (1)
Intel x86

- Intel created the 8008 (in 1972)
- 8008 begat 8080
- 8080 begat 8086
- 8086 begat 8088
- 8086 begat 286
- 286 begat 386
- 386 begat 486
- 486 begat Pentium
- Pentium begat Pentium Pro
- Pentium Pro begat Pentium II
- ad infinitum
2^{64}

• 2^{32} used to be considered a large number
  – one couldn’t afford 2^{32} bytes of memory, so no problem with that as an upper bound

• Intel (and others) saw need for machines with 64-bit addresses
  – devised IA64 architecture with HP
    » became known as Itanium
    » very different from x86

• AMD also saw such a need
  – developed 64-bit extension to x86, called x86-64

• Itanium flopped

• x86-64 dominated

• Intel, reluctantly, adopted x86-64
Data Types on IA32 and x86-64

• “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 \texttt{mov} ...
  - \texttt{movb}
  - \texttt{movs}
  - \texttt{movl}
  - \texttt{movq} (x86-64 only)
### General-Purpose Registers (IA32)

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</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)
- Origin (mostly obsolete): accumulate, counter, data, base, source index, destination index, stack pointer, base pointer.
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”

<table>
<thead>
<tr>
<th>%eax</th>
<th>%ecx</th>
<th>%edx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
</table>
### 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>movl</strong></td>
<td>Reg</td>
<td>Reg</td>
<td>movl $0x4, %eax</td>
</tr>
<tr>
<td></td>
<td><strong>Imm</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Mem</strong></td>
<td>movl $-147, (%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td></td>
<td>Reg</td>
<td>Reg</td>
<td>movl %eax, %edx</td>
</tr>
<tr>
<td></td>
<td><strong>Mem</strong></td>
<td>movl %eax, (%edx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax), %edx</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

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

- Displacement D(R) \( \text{Mem}[\text{Reg}[R]+D] \)
  - 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;
}
```

```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
```
Understanding Swap

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

<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>

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

Address

<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>0x118</td>
</tr>
<tr>
<td>0</td>
<td>0x114</td>
</tr>
<tr>
<td>-4</td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x10c</td>
</tr>
<tr>
<td></td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

%ebp 0x104

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

\begin{itemize}
  \item \%eax
  \item \%edx = 0x124
  \item \%ecx = 0x120
  \item \%ebx
  \item \%esi
  \item \%edi
  \item \%esp
  \item \%ebp = 0x104
\end{itemize}

\begin{itemize}
  \item Offset
  \item Rtn adr
  \item \%ebp \rightarrow 0
  \item \%ebp \rightarrow -4
\end{itemize}

\begin{itemize}
  \item \textbf{movl} 8(\%ebp), \%edx \# edx = xp
  \item \textbf{movl} 12(\%ebp), \%ecx \# ecx = yp
  \item \textbf{movl} (\%edx), \%ebx \# ebx = *xp \text{(t0)}
  \item \textbf{movl} (\%ecx), \%eax \# eax = *yp \text{(t1)}
  \item \textbf{movl} \%eax, (\%edx) \# *xp = t1
  \item \textbf{movl} \%ebx, (\%ecx) \# *yp = t0
\end{itemize}
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>

```
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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>456</td>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>456</td>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0x10c</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>-4</td>
<td>0x100</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>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>yp</td>
<td>123</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>xp</td>
<td>0x124</td>
</tr>
<tr>
<td>0x118</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[ D(R_b,R_i,S) \quad \text{Mem}[\text{Reg}[R_b]+S*\text{Reg}[R_i]+D] \]

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

• Special cases

- \((R_b,R_i)\) \quad \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]]
- \(D(R_b,R_i)\) \quad \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]+D]
- \((R_b,R_i,S)\) \quad \text{Mem}[\text{Reg}[R_b]+S*\text{Reg}[R_i]]
- \(D\) \quad \text{Mem}[D]

---

CS33 Intro to Computer Systems  
XI–21
### 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>
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 \( \text{p} = \&x[i]; \)
  - computing arithmetic expressions of the form \( x + k\times y \)
    » \( k = 1, 2, 4, \text{or} 8 \)

- **Example**

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

Converted to ASM by compiler:

```asm
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

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

<table>
<thead>
<tr>
<th>Column A (a1)</th>
<th>Column B (a2)</th>
<th>Column C (a3)</th>
<th>Column D (a4)</th>
<th>Column E (a5)</th>
<th>Column F (a6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%r8</td>
<td>%r8d</td>
<td>a5</td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
<td></td>
<td>a6</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<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
Bytes

• Each register has a byte version
  – e.g., %r10: %r10b

• Needed for byte instructions
  – movb (%rax, %rsi), %r10b
  – sets *only* the low byte in %r10
    » other seven bytes are unchanged

• Alternatives
  – movzbq (%rax, %rsi), %r10
    » copies byte to low byte of %r10
    » zeroes go to higher bytes
  – movs orbital
    » copies byte to low byte of %r10
    » sign is extended to all higher bits
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
```

- **Set Up**
- **Body**
- **Finish**
64-bit code for swap

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

- Arguments passed in registers
  - 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

```plaintext
movl (%rdi), %edx
movl (%rsi), %eax
movl %eax, (%rdi)
movl %edx, (%rsi)
ret
```
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)
```

```
binary
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
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>

Diagram:

```
+-------------+----------+-------------+----------+-----------------+-------------+
|             |          |             |          |                 |             |
| Instruction | Opcode   | ModR/M      | SIB      | Displacement    | Immediate   |
| Prefixes    |          |             |          |                 |             |
|             |          |             |          |                 |             |
| Up to four  | 1 or 2   | 1 byte      | 1 byte   | Address         | Immediate   |
| prefixes    | byte opcode | (if required) | (if required) | displacement of 1, 2, or 4 bytes or none | data of 1, 2, or 4 bytes or none |
| of 1-byte    |           |             |          |                 |             |
| each (optional) |         |             |          |                 |             |
```

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

### Dump of assembler code for function sum:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x080483c4</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0x080483c5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>0x080483c7</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>0x080483ca</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>0x080483cd</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>0x080483ce</td>
<td>ret</td>
</tr>
</tbody>
</table>

### 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 ~30
- 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
    » ~100
  - SIMD instructions
    » lots
  - 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>

- *Also called shll*
- *Arithmetic*
- *Logical*

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

- **One-operand Instructions**
  - \texttt{incl} \hspace{0.5cm} \texttt{Dest} = \texttt{Dest} + 1
  - \texttt{decl} \hspace{0.5cm} \texttt{Dest} = \texttt{Dest} - 1
  - \texttt{negl} \hspace{0.5cm} \texttt{Dest} = -\texttt{Dest}
  - \texttt{notl} \hspace{0.5cm} \texttt{Dest} = \neg\texttt{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 \texttt{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

```asm
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

```
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 %ecx?

```assembly
xorl %ecx, %ecx
incl %ecx
sall %cl, %ecx  # %cl is the low byte of %ecx
addl %ecx, %ecx
```

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