CS 33

Machine Programming (2)
Jump Instructions

• Unconditional jump
  – just do it

• Conditional jump
  – to jump or not to jump determined by condition-code flags
  – field in the op code indicates how this is computed
  – in assembler language, simply say
    » je
      • jump on equal
    » jne
      • jump on not equal
    » jgt
      • jump on greater than
    » etc.
Addresses

```c
int a, b, c, d;

int main() {
    a = (b + c) * d;
    ...
}
```

```
mov  b, %acc
add  c, %acc
mul  d, %acc
mov  %acc, a
mov  1004, %acc
add  1008, %acc
mul  1012, %acc
mov  %acc, 1000
```

```
1012:
  d
global
1008:
c
1004:
b
1000:
a
```
Addresses

```c
int b;

int func(int c, int d) {
    int a;
    a = (b + c) * d;
    ...
}

mov ?,%acc
add ?,%acc
mul ?,%acc
mov %acc,?
```

- One copy of `b` for duration of program’s execution
  - `b`’s address is the same for each call to `func`
- Different copies of `a`, `c`, and `d` for each call to `func`
  - addresses are different in each call
Relative Addresses

- **Absolute address**
  - actual location in memory

- **Relative address**
  - offset from some other location

- Blob’s absolute address is 10000
- Datum’s relative address (to Blob) is 100
  - its absolute address is 10100

---

Memory

<table>
<thead>
<tr>
<th>2^{64}-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob</td>
</tr>
<tr>
<td>Datum</td>
</tr>
</tbody>
</table>

0 1000 10000 Memory
Base Registers

\[
\begin{align*}
\text{mov } & \$10000, \%\text{base} \\
\text{mov } & \$10, 100(\%\text{base})
\end{align*}
\]
Addresses

```c
long b;

int func(long c, long d) {
    long a;
    a = (b + c) * d;
    ...
}
```

```assembly
mov 1000,%acc
add -8(%base),%acc
mul -12(%base),%acc
mov %acc,-16(%base)
```
Suppose the value in base is 10,000. What is the address of c?

a) 9992  
b) 9996  
c) 10,004  
d) 10,008

mov 1000,%acc
add -8(%base),%acc
mul -12(%base),%acc
mov %acc,-16(%base)
Registers

- Instruction pointer
- Accumulator
- Base register
- More
- Condition codes

Execution engine

Interchangeable
Registers vs. Memory

Execution engine

- Instruction pointer
- Accumulator
- Base register
- More
- Condition codes

Memory (aka RAM)

Instructions and data

Data

A relatively long distance
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

- byte
- short
- long
- quad

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

- **eax**, **ecx**, **edx**, **ebx**, **esi**, **edi**, **esp**, **ebp**:
  - General purpose registers.
  - Used for data manipulation.

- **ax**, **cx**, **dx**, **bx**, **si**, **di**, **bp**:
  - 16-bit virtual registers.
  - Backwards compatibility for compatibility with older systems.

- **ah**, **ch**, **dh**, **bh**, **bl**:
  - Accumulate.
  - Counter.
  - Base.
  - Source.
  - Index.

- **al**, **cl**, **dl**, **bh**, **bl**:
  - Destination.
  - Index.

Origin:
- Accumulate
- Counter
- Data
- Base
- Source
- Index
- Destination
- Stack
- Pointer
- Base
- Pointer

(mostly obsolete)
Moving Data: IA32

- Moving data
  \texttt{movl source, dest}

- Operand types
  - \textit{Immediate}: constant integer data
    - example: $0\times400, -533$
    - like C constant, but prefixed with `$`'
    - encoded with 1, 2, or 4 bytes
  - \textit{Register}: one of 8 integer registers
    - example: \%eax, \%edx
    - but \%esp and \%ebp reserved for special use
    - others have special uses for particular instructions
  - \textit{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>Mem</td>
<td>Reg</td>
<td>movl %eax, %edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Mem</td>
<td>Img</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) \( \text{Mem}[\text{Reg}[R]] \)
  - register R specifies memory address
    
    \[
    \text{movl} \; (\%ecx),\%eax
    \]

- Displacement D(R) \( \text{Mem}[\text{Reg}[R]+D] \)
  - register R specifies start of memory region
  - constant displacement D specifies offset
    
    \[
    \text{movl} \; 8(\%ebp),\%edx
    \]
Using Simple Addressing Modes

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

**swap:**

```assembly
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**
Understanding Swap

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>0x104</td>
<td>0</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
<td>-4</td>
</tr>
<tr>
<td>%eax</td>
<td>0x100</td>
<td>12</td>
</tr>
<tr>
<td>%edx</td>
<td>0x120</td>
<td>8</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x110</td>
<td>4</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td></td>
<td>456</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x120</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x11c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x10c</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x104</td>
<td>0x124</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>0x10c</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

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>Variable</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>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>

#### 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
```

#### Registers

<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>
Understanding Swap

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

<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>0x11c</td>
</tr>
<tr>
<td>0</td>
<td>0x118</td>
</tr>
<tr>
<td>0</td>
<td>0x114</td>
</tr>
<tr>
<td>0</td>
<td>0x110</td>
</tr>
<tr>
<td>0</td>
<td>0x10c</td>
</tr>
<tr>
<td>0</td>
<td>0x108</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>0</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>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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x114</td>
<td>0x108</td>
</tr>
<tr>
<td>0x118</td>
<td>0x104</td>
</tr>
<tr>
<td>0x11c</td>
<td>0x100</td>
</tr>
<tr>
<td>0x120</td>
<td>0x124</td>
</tr>
<tr>
<td>0x124</td>
<td>123</td>
</tr>
<tr>
<td>0x128</td>
<td>456</td>
</tr>
</tbody>
</table>

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

```plaintext
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 2

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

Which C statements best describe the assembler code?

// a  // b  // c  // d
int x;  int *x;  int **x;  int ***x;
int y;  int y;  int y;  int y;
y = x;  y = *x;  y = **x;  y = ***x;
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 \( \%\text{esp} \)
    » unlikely you’d use \( \%\text{ebp} \) either
  - \( S \): scale: 1, 2, 4, or 8

• Special cases

\( (Rb,Ri) \rightarrow Mem[Reg[Rb]+Reg[Ri]] \)
\( D(Rb,Ri) \rightarrow Mem[Reg[Rb]+Reg[Ri]+D] \)
\( (Rb,Ri,S) \rightarrow Mem[Reg[Rb]+S*Reg[Ri]] \)
\( D \rightarrow Mem[D] \)
# 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:

```asm
movl 8(%ebp), %eax  # get arg
leal (%eax,%eax,2), %eax  # t <- x+x*2
sall $2, %eax        # return t<<2
```
Quiz 3

What value ends up in %ecx?

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

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

**Hint:** %eax → 1000: 0x00
### x86-64 General-Purpose Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</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
  – movsbq (%rax, %rsi), %r10
    » 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;
}
```

`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

```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
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)

binary
   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

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

![Instruction Format Diagram](image-url)
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

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 ~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>

- 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;
}
```

```assembly
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
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  # 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)
sar1 $17, %edi    # edi = t1>>17    (t2)
movl %edi, %eax   # eax = edi
andl $8185, %eax  # eax = t2 & mask (rval)
```
Quiz 4

• What is the final value in %ecx?

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