Given the following based on typedef:

```c
src_t v;
dest_t *p;
```

Implement the following operation:

```c
*p = (dest_t) v;
```

Where

- `v` stored in `%eax` (or `%ax` or `%al`)
- Pointer `p` is stored in register `%edx`

<table>
<thead>
<tr>
<th>SRC_T</th>
<th>DEST_T</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td><code>movl %eax, (%edx)</code></td>
</tr>
<tr>
<td>char</td>
<td>int</td>
<td><code>movsbl %al, (%edx)</code></td>
</tr>
<tr>
<td>char</td>
<td>unsigned</td>
<td><code>movsbl %al, (%edx)</code></td>
</tr>
<tr>
<td>unsigned</td>
<td>char</td>
<td><code>movzbl %al, (%edx)</code></td>
</tr>
<tr>
<td>int</td>
<td>char</td>
<td><code>movbl %al, (%edx)</code></td>
</tr>
<tr>
<td>unsigned</td>
<td>unsigned</td>
<td><code>movb %al, (%edx)</code></td>
</tr>
<tr>
<td>unsigned</td>
<td>int</td>
<td><code>movl %eax, (%edx)</code></td>
</tr>
</tbody>
</table>

Function with Prototype:

```c
void decode1(int *xp, int *yp, int*zp);
```

xp=%ebp+8; yp=%ebp+12; zp=%ebp+16;

Assembly code:

```assembly
movl 8(%ebp), %edi        ; get xp
movl 12(%ebp), %edx       ; get yp
movl 16(%ebp), %ecx       ; get zp
movl (%edx), %ebx         ; get y
movl (%ecx), %esi         ; get z
movl (%edi), %eax         ; get x
movl %eax, (%edx)         ; store x at yp
movl %ebx, (%ecx)         ; store y at zp
movl %esi, (%edi)         ; store z at xp
```
## Arithmetic and Logical Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>leal</td>
<td>$S,D$</td>
<td>$&amp;S \rightarrow D$</td>
</tr>
<tr>
<td>INC</td>
<td>$D$</td>
<td>$D+1 \rightarrow D$</td>
</tr>
<tr>
<td>DEC</td>
<td>$D$</td>
<td>$D-1 \rightarrow D$</td>
</tr>
<tr>
<td>NEG</td>
<td>$D$</td>
<td>$-D \rightarrow D$</td>
</tr>
<tr>
<td>NOT</td>
<td>$D$</td>
<td>$\neg D \rightarrow D$</td>
</tr>
<tr>
<td>ADD</td>
<td>$S, D$</td>
<td>$D + S \rightarrow D$</td>
</tr>
<tr>
<td>SUB</td>
<td>$S, D$</td>
<td>$D - S \rightarrow D$</td>
</tr>
<tr>
<td>IMUL</td>
<td>$S, D$</td>
<td>$D \times S \rightarrow D$</td>
</tr>
<tr>
<td>XOR</td>
<td>$S, D$</td>
<td>$D \oplus S \rightarrow D$</td>
</tr>
<tr>
<td>OR</td>
<td>$S, D$</td>
<td>$D \lor S \rightarrow D$</td>
</tr>
<tr>
<td>AND</td>
<td>$S, D$</td>
<td>$D \land S \rightarrow D$</td>
</tr>
<tr>
<td>SAL</td>
<td>$k, D$</td>
<td>$D \ll k \rightarrow D$</td>
</tr>
<tr>
<td>SHL</td>
<td>$k, D$</td>
<td>$D \ll k \rightarrow D$</td>
</tr>
<tr>
<td>SAR</td>
<td>$k, D$</td>
<td>$D \gg k \rightarrow A, D$</td>
</tr>
<tr>
<td>SHR</td>
<td>$k, D$</td>
<td>$D \gg k \rightarrow L, D$</td>
</tr>
</tbody>
</table>

- Watch out for argument order! see SUB
- No distinction between signed and unsigned int
- Notice A/L for arithmetic and logical right shifts
- Operation Groups
  - Variant of the move
  - Unary
  - Binary
  - Shifts

Reminder: Note the difference in instruction between assemble and disassemble – just like the movl vs mov
LEA – load effective address

- Does not reference memory at all
  - You don’t get the value at the address... just the address (&x)
- Copies the effective address to the destination
- Used to generate pointers for later memory references
- Can also be used to compactly describe common arithmetic operations
- The destination operand must be a register

Example: leal 7 (%edx, %edx, 4), %eax
Sets register %eax to 5x+7
%edx + %edx*4 + 7

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>leal 6(%eax), %edx</td>
<td>6 + x</td>
</tr>
<tr>
<td>leal (%eax, %ecx), %edx</td>
<td>x + y</td>
</tr>
<tr>
<td>leal (%eax, %ecx, 4), %edx</td>
<td>x + 4y</td>
</tr>
<tr>
<td>leal 7(%eax, %eax,8), %edx</td>
<td>7 + 9x</td>
</tr>
<tr>
<td>leal 0xA(%ecx,4),%edx</td>
<td>10 + 4y</td>
</tr>
<tr>
<td>leal 9(%eax,%ecx,2), %edx</td>
<td>9 + x + 2y</td>
</tr>
</tbody>
</table>

Assume: %eax = x and %ecx= y
Unary and Binary operations

Unary
- Single operand serves as both source and destination
- Register or memory location
- Similar to C++ and -- operators

Binary
- Second operand is both source and destination
  - Thus cannot be an immediate value
  - Can be memory or register
- First operand can be immediate, memory, or register
- Reminder: both cannot be memory
- Similar to C operations such as x += y

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>VALUE</th>
<th>REGISTER</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
<td>%eax</td>
<td>0x100</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
<td>%ecx</td>
<td>0x1</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
<td>%edx</td>
<td>0x3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>DESTINATION</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl %ecx, (%eax)</td>
<td>0x100</td>
<td>0x100</td>
</tr>
<tr>
<td>subl %edx, 4(%eax)</td>
<td>0x104</td>
<td>0xA8</td>
</tr>
<tr>
<td>imull $16,%eax,%edx,4</td>
<td>0x10C</td>
<td>0x110</td>
</tr>
<tr>
<td>incl 8(%eax)</td>
<td>0x108</td>
<td>0x14</td>
</tr>
<tr>
<td>decl %ecx</td>
<td>%ecx</td>
<td>0x0</td>
</tr>
<tr>
<td>subl %edx, %eax</td>
<td>%eax</td>
<td>0xFD</td>
</tr>
</tbody>
</table>
Shift operations

- Shift amount given in first operand
  - Coded as a single byte
  - Only shift amounts between 0 and 31 possible
    - Only low order 5 bits are considered
  - Immediate value or in the single byte register element %c1 (unusual!)

- Value to shift in second operand

- Arithmetic and logical
  - Left shifts behave the same, though
    - Zero fill
  - Right shifts
    - Sign extend (arithmetic)
    - Zero fill (logical)
Overview

- Instructions work for unsigned or two’s complement arithmetic
  - Except right shift
  - Signed/unsigned division?
  - Unsigned multiplication?
- Makes 2’s comp arithmetic the preferred way to implement signed integer arithmetic
Arithmetic example

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;
}

000000000 <arith>:

movl 8(%ebp), %ecx
movl 12(%ebp), %edx
leal (%edx,%edx,2), %eax
sall $4, %eax
leal 4(%ecx,%eax), %eax
addl %ecx, %edx
addl 16(%ebp), %edx
imull %edx, %eax
push %ebp
mov %esp,%ebp
mov 0x8(%ebp),%ecx
mov 0xc(%ebp),%edx
lea (%edx,%edx,2),%eax
shl $0x4,%eax
lea 0x4(%ecx,%eax,1),%eax
add %ecx,%edx
add 0x10(%ebp),%edx
imul %edx,%eax
pop %ebp
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;
}