Assembly: Arithmetic operations and control flow.

Yipeng Huang

Rutgers University

March 4, 2021
Table of contents

Announcements

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations
  Load effective address

Control flow
Looking ahead

Class plan

1. Thursday, 3/4: Assembly arithmetic operations and control flow.
2. Code review session for PA2 is the week of 3/8 - 3/12. TAs will take attendance to assign participation points.
3. Reading assignment for next three weeks: CS:APP Chapter 3.
Programming Assignment 3: binSub

- PA3 is structured in terms of difficulty similarly to PA1 and PA2.
- The assignment rewards you for starting early.
- Use Piazza; We rely on it to gauge what needs further explanation.
- Later parts (parts 4 and 5) are more open-ended.
Table of contents

Announcements

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations
  Load effective address

Control flow
Sign extension due to unsigned and signed data types

Converting to a data type with more bits

```c
unsigned short uc_to_us ( unsigned char input ) {
    return input;
}
```

```c
signed short sc_to_ss ( signed char input ) {
    return input;
}
```

255 = 1111_1111₂
    = 0000_0000_1111_1111₂
    = 255

127 = 0111_1111₂
    = 0000_0000_0111_1111₂
    = 127

−128 = 1000_0000₂
    = 1111_1111_1000_0000₂
    = −128
Sign extension due to unsigned and signed data types

Converting to a data type with more bits

```c
// unsigned short uc_to_us ( unsigned char input )
unsigned short uc_to_us ( unsigned char input ) {
    return input;
}

// signed short sc_to_ss ( signed char input )
signed short sc_to_ss ( signed char input ) {
    return input;
}
```

<table>
<thead>
<tr>
<th>function signature</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned short uc_to_us ( unsigned char input );</td>
<td>movzbl %dil, %eax</td>
</tr>
<tr>
<td>signed short uc_to_ss ( unsigned char input );</td>
<td>movzbl %dil, %eax</td>
</tr>
<tr>
<td>unsigned short sc_to_us ( signed char input );</td>
<td>movsbw %dil, %ax</td>
</tr>
<tr>
<td>signed short sc_to_ss ( signed char input );</td>
<td>movsbw %dil, %ax</td>
</tr>
</tbody>
</table>

- movz: zero extension in the MSBs
- movs: signed extension in the MSBs
Table of contents

Announcements

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations
  Load effective address

Control flow
Left shift operation

```c
unsigned long sl_ul (unsigned long in0, unsigned long in1) {
    return in0<<in1;
}

signed long sl_sl (signed long in0, signed long in1) {
    return in0<<in1;
}
```

Both C code functions above translate to the assembly on the right.

```
sl_ul:
    movq %rdi, %rax
    movb %sil, %cl
    salq %cl, %rax
    ret

sl_sl:
    movq %rdi, %rax
    movb %sil, %cl
    salq %cl, %rax
    ret
```

Explanation

- **movq**: `in0 → %rdi → %rax`
- **movb**: `in1 → %sil → %cl`
- **salq src,dest**: `(dest << src) → dest`
- Why only use movb for in1?
Right shift operation

Right shift of unsigned types yields logical (zero-filled) right shift

```c
unsigned long sr_ul (unsigned long in0, unsigned long in1) {
    return in0>>in1;
}
```

```
sr_ul:
    movq  %rdi, %rax
    movb  %sil, %cl
    shrq  %cl, %rax
    ret
```

Right shift of signed types yields arithmetic (sign-extended) right shift

```c
signed long sr_sl (signed long in0, signed long in1) {
    return in0>>in1;
}
```

```
sr_sl:
    movq  %rdi, %rax
    movb  %sil, %cl
    sarq  %cl, %rax
    ret
```
### Bitwise operations

<table>
<thead>
<tr>
<th>Assembly instruction</th>
<th>Instruction effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>notq dest</code></td>
<td>( \sim \text{dest} \rightarrow \text{dest} )</td>
</tr>
<tr>
<td><code>andq src,dest</code></td>
<td>( \text{src} &amp; \text{dest} \rightarrow \text{dest} )</td>
</tr>
<tr>
<td><code>orq src,dest</code></td>
<td>( \text{src} \mid \text{dest} \rightarrow \text{dest} )</td>
</tr>
<tr>
<td><code>xorq src,dest</code></td>
<td>( \text{src} \land \text{dest} \rightarrow \text{dest} )</td>
</tr>
</tbody>
</table>
## Integer arithmetic operations

<table>
<thead>
<tr>
<th>Assembly instruction</th>
<th>Instruction effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>incq dest</td>
<td>dest + 1 → dest</td>
</tr>
<tr>
<td>decq dest</td>
<td>dest − 1 → dest</td>
</tr>
<tr>
<td>negq dest</td>
<td>−dest → dest</td>
</tr>
<tr>
<td>addq src, dest</td>
<td>src + dest → dest</td>
</tr>
<tr>
<td>subq src, dest</td>
<td>src − dest → dest</td>
</tr>
<tr>
<td>imulq src, dest</td>
<td>src × dest → dest</td>
</tr>
</tbody>
</table>
Load effective address

Both C code functions above translate to the assembly on the right.

leaq:

mulAdd:

leaq 8(%rdi,%rsi,8), %rax
ret

Explanation

- leaq src,dest takes the effective address of the memory (index, displacement) expression of src and puts it in dest.
- leaq has shorter latency (takes fewer CPU cycles) than imulq, so GCC will use leaq whenever it can to calculate expressions like $y + ax + b$. 
Table of contents

Announcements

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations
  Load effective address

Control flow