Assembly: Arithmetic operations and control flow.

Yipeng Huang

Rutgers University

March 4, 2021

Table of contents

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.
4. Programming Assignment 3 on bits, bytes, integers, floats out. Due Monday March 22.

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

Sign extension due to unsigned and signed data types

Converting to a data type with more bits

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

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

$$\begin{aligned} 255 &= 1111\_1111_2 \\ &= 0000\_0000\_1111\_1111_2 \\ &= 255 \end{aligned}$$

$$\begin{aligned} 127 &= 0111\_1111_2 \\ &= 0000\_0000\_0111\_1111_2 \\ &= 127 \end{aligned}$$

$$\begin{aligned} -128 &= 1000\_0000_2 \\ &= 1111\_1111\_1000\_0000_2 \\ &= -128 \end{aligned}$$

Sign extension due to unsigned and signed data types

Converting to a data type with more bits

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

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

function signature	assembly code
unsigned short uc_to_us (unsigned char input);	movzbl %dil, %eax
signed short uc_to_ss (unsigned char input);	movzbl %dil, %eax
unsigned short sc_to_us (signed char input);	movsbw %dil, %ax
signed short sc_to_ss (signed char input);	movsbw %dil, %ax

- movz: zero extension in the MSBs
- movs: signed extension in the MSBs

Table of contents

Left shift operation

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

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

Explanation

- `movq`: in0 $\rightarrow$ %rdi $\rightarrow$ %rax
- `movb`: in1 $\rightarrow$ %sil $\rightarrow$ %cl
- `salq src,dest`: (dest << src) $\rightarrow$ dest
- Why only use movb for in1?

Right shift operation

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

```
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

```
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

Assembly instruction	Instruction effect
`notq dest`	$\sim$ dest $\to$ dest
`andq src,dest`	src&dest $\to$ dest
`orq src,dest`	src$\vert$dest $\to$ dest
`xorq src,dest`	src $\wedge$ dest $\to$ dest

Integer arithmetic operations

Assembly instruction	Instruction effect
`incq dest`	dest $+ 1 \rightarrow$ dest
`decq dest`	dest $- 1 \rightarrow$ dest
`negq dest`	$-$dest $\rightarrow$ dest
`addq src,dest`	src $+$ dest $\rightarrow$ dest
`subq src,dest`	src $-$ dest $\rightarrow$ dest
`imulq src,dest`	src $\times$ dest $\rightarrow$ dest

Load effective address

```
long * leaq (
    long * ptr, long index
) {
    return &ptr[index+1];
}
```

```
long mulAdd (
    long base, long index
) {
    return base+index*8+8;
}
```

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