Machine-level representation of programs: Moving data, arithmetic

Yipeng Huang

Rutgers University

March 10, 2022
Table of contents

Announcements

1_swap.s: Assembly implementation of function that swaps memory contents
C data type size and register word sizes

2_addressing_modes.s: Understanding source dest operands and memory addressing modes

3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations
Quiz, PA3, PA4

Quiz 5: The limits of floating point numbers
Out now, due Friday night before spring break officially starts.

PA4
PA4 (Bomblab) is an assignment lab offered by the textbook. A CS211 classic. Will be launched before we go on break. Due two weeks after we get back from break.
Table of contents

Announcements

1_swap.s: Assembly implementation of function that swaps memory contents
C data type size and register word sizes

2_addressing_modes.s: Understanding source dest operands and memory addressing modes

3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations
Unraveling the compilation chain

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`) [New to recent versions of GCC]
  - Put resulting binary in file `p`

- `text` C program (`p1.c p2.c`)
- `text` Asm program (`p1.s p2.s`)
- `binary` Object program (`p1.o p2.o`)
- `binary` Executable program (`p`)

- `gcc -Og -S 1_swap.c`
- `objdump -d 1_swap`

Let’s go to CS:APP textbook lecture slides (05-machine-basics.pdf) slide 28
Data movement instructions

Does unsigned / signed matter?

1. void swap_uc ( unsigned char*a, unsigned char*b );
2. void swap_sc ( signed char*a, signed char*b );

Swapping different data sizes

1. void swap_c ( char*a, char*b );
2. void swap_s ( short*a, short*b );
3. void swap_i ( int*a, int*b );
4. void swap_l ( long*a, long*b );
Table of contents

Announcements

1_swap.s: Assembly implementation of function that swaps memory contents
C data type size and register word sizes

2_addressing_modes.s: Understanding source dest operands and memory addressing modes

3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations
C data type size and register word sizes

Assembly syntax

Instruction Source, Dest

```
swap_l:
  movq (%rsi), %rax
  movq (%rdi), %rdx
  movq %rdx, (%rsi)
  movq %rax, (%rdi)
ret
```

<table>
<thead>
<tr>
<th>swap</th>
<th>data type</th>
<th>mov operation</th>
<th>registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>swap_uc</td>
<td>unsigned char</td>
<td>movb (move byte)</td>
<td>%al, %dl</td>
</tr>
<tr>
<td>swap_sc</td>
<td>signed char</td>
<td>movb (move byte)</td>
<td>%al, %dl</td>
</tr>
<tr>
<td>swap_c</td>
<td>char</td>
<td>movb (move byte)</td>
<td>%al, %dl</td>
</tr>
<tr>
<td>swap_s</td>
<td>short</td>
<td>movw (move word)</td>
<td>%ax, %dx</td>
</tr>
<tr>
<td>swap_i</td>
<td>int</td>
<td>movl</td>
<td>%eax, %edx</td>
</tr>
<tr>
<td>swap_l</td>
<td>long</td>
<td>movq</td>
<td>%rax, %rdx</td>
</tr>
</tbody>
</table>
Data size and x86, IA32, and x86-64 registers

<table>
<thead>
<tr>
<th>data type</th>
<th>registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>%al, %dl</td>
</tr>
<tr>
<td>short</td>
<td>%ax, %dx</td>
</tr>
<tr>
<td>int</td>
<td>%eax, %edx</td>
</tr>
<tr>
<td>long</td>
<td>%rax, %rdx</td>
</tr>
</tbody>
</table>

Note the backward compatibility.

Some History: IA32 Registers

- General purpose registers:
  - %eax (EAX), %ax (AX), %ah (AH), %al (AL)
  - %ecx (ECX), %cx (CX), %ch (CH), %cl (CL)
  - %edx (EDX), %dx (DX), %dh (DH), %dl (DL)
  - %ebx (EBX), %bx (BX), %bh (BH), %bl (BL)
  - %esi (ESI), %si (SI)
  - %edi (EDI), %di (DI)
  - %esp (ESP), %sp (SP)
  - %ebp (EBP), %bp (BP)

- Origin (mostly obsolete):
  - accumulate
  - counter
  - data
  - base
  - source
  - index
  - destination
  - index
  - stack
  - pointer
  - base
  - pointer

16-bit virtual registers (backwards compatibility)
Data size and x86, IA32, and x86-64 registers

---

**x86-64 Integer Registers**

<table>
<thead>
<tr>
<th>data type</th>
<th>registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>%al, %dl</td>
</tr>
<tr>
<td>short</td>
<td>%ax, %dx</td>
</tr>
<tr>
<td>int</td>
<td>%eax, %edx</td>
</tr>
<tr>
<td>long</td>
<td>%rax, %rdx</td>
</tr>
</tbody>
</table>

Note the backward compatibility.

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
## Data size and x86, IA32, and x86-64 registers

<table>
<thead>
<tr>
<th>ZMM0</th>
<th>YMM0</th>
<th>XMM0</th>
<th>ZMM1</th>
<th>YMM1</th>
<th>XMM1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZMM2</td>
<td>YMM2</td>
<td>XMM2</td>
<td>ZMM3</td>
<td>YMM3</td>
<td>XMM3</td>
</tr>
<tr>
<td>ZMM4</td>
<td>YMM4</td>
<td>XMM4</td>
<td>ZMM5</td>
<td>YMM5</td>
<td>XMM5</td>
</tr>
<tr>
<td>ZMM6</td>
<td>YMM6</td>
<td>XMM6</td>
<td>ZMM7</td>
<td>YMM7</td>
<td>XMM7</td>
</tr>
<tr>
<td>ZMM8</td>
<td>YMM8</td>
<td>XMM8</td>
<td>ZMM9</td>
<td>YMM9</td>
<td>XMM9</td>
</tr>
<tr>
<td>ZMM10</td>
<td>YMM10</td>
<td>XMM10</td>
<td>ZMM11</td>
<td>YMM11</td>
<td>XMM11</td>
</tr>
<tr>
<td>ZMM12</td>
<td>YMM12</td>
<td>XMM12</td>
<td>ZMM13</td>
<td>YMM13</td>
<td>XMM13</td>
</tr>
<tr>
<td>ZMM14</td>
<td>YMM14</td>
<td>XMM14</td>
<td>ZMM15</td>
<td>YMM15</td>
<td>XMM15</td>
</tr>
<tr>
<td>ZMM16</td>
<td>ZMM17</td>
<td>ZMM18</td>
<td>ZMM19</td>
<td>ZMM20</td>
<td>ZMM21</td>
</tr>
<tr>
<td>ZMM22</td>
<td>ZMM23</td>
<td>ZMM24</td>
<td>ZMM25</td>
<td>ZMM26</td>
<td>ZMM27</td>
</tr>
<tr>
<td>ZMM28</td>
<td>ZMM29</td>
<td>ZMM30</td>
<td>ZMM31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ST(0)</th>
<th>MM0</th>
<th>ST(1)</th>
<th>MM1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(2)</td>
<td>MM2</td>
<td>ST(3)</td>
<td>MM3</td>
</tr>
<tr>
<td>ST(4)</td>
<td>MM4</td>
<td>ST(5)</td>
<td>MM5</td>
</tr>
<tr>
<td>ST(6)</td>
<td>MM6</td>
<td>ST(7)</td>
<td>MM7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AX</th>
<th>BX</th>
<th>CX</th>
<th>DX</th>
<th>RX</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
<th>R12</th>
<th>R13</th>
<th>R14</th>
<th>R15</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R4</td>
<td>R5</td>
<td>R6</td>
<td>R7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MSWCRO</th>
<th>CR0</th>
<th>CR1</th>
<th>CR2</th>
<th>CR3</th>
<th>CR4</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR5</td>
<td>CR6</td>
<td>CR7</td>
<td>CR8</td>
<td>CR9</td>
<td>CR10</td>
</tr>
<tr>
<td>CR11</td>
<td>CR12</td>
<td>CR13</td>
<td>CR14</td>
<td>CR15</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CW</th>
<th>FP_IP</th>
<th>FP_DP</th>
<th>FP_CS</th>
<th>SI</th>
<th>ESI</th>
<th>RSI</th>
<th>ESP</th>
<th>RSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW</td>
<td>TW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CS</th>
<th>SS</th>
<th>DS</th>
<th>GDTR</th>
<th>IDTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES</td>
<td>FS</td>
<td>GS</td>
<td>TR</td>
<td>LDTR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FLAGS</th>
<th>RFLAGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DR0</td>
<td>DR6</td>
</tr>
<tr>
<td>DR1</td>
<td>DR7</td>
</tr>
<tr>
<td>DR2</td>
<td>DR8</td>
</tr>
<tr>
<td>DR3</td>
<td>DR9</td>
</tr>
<tr>
<td>DR4</td>
<td>DR10</td>
</tr>
<tr>
<td>DR5</td>
<td>DR11</td>
</tr>
<tr>
<td>DR12</td>
<td>DR13</td>
</tr>
<tr>
<td>DR14</td>
<td>DR15</td>
</tr>
</tbody>
</table>

**Figure:** x86-64 with SIMD extensions registers. Image credit: [https://commons.wikimedia.org/wiki/File:Table_of_x86_Registers_svg.svg](https://commons.wikimedia.org/wiki/File:Table_of_x86_Registers_svg.svg)
Table of contents

Announcements

1_swap.s: Assembly implementation of function that swaps memory contents
C data type size and register word sizes

2_addressing_modes.s: Understanding source dest operands and memory addressing modes

3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations
Immediate
Constant integer values. Example: 2_addressing_modes.c immediate()

Register
One of the registers of appropriate size for data type.
Example: 1_swap.c

Memory
Access to memory at calculated

```
movq $0x4,%rax  temp = 0x4;
movq $-147,(%rax) *p = -147;
movq %rax,%rdx temp2 = temp1;
movq %rax,(%rdx) *p = temp;
movq (%rax),%rdx temp = *p;
```

```
C Analog
```

```
Cannot do memory-memory transfer with a single instruction
```
Addressing modes

Simple Memory Addressing Modes

- **Normal**  (R)  Mem[Reg[R]]
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C
    
    ```
    movq (%rcx),%rax
    ```

- **Displacement**  D(R)  Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
    
    ```
    movq 8(%rbp),%rdx
    ```

Normal
Simple pointers.
Example: 2_addressing_modes.c
immediate()

Displacement
Array access with constant index.
Example: 2_addressing_modes.c
displacement()
Addressing modes

Complete Memory Addressing Modes

- **Most General Form**
  
  \[ D(Rb,Ri,S) \rightarrow \text{Mem}[\text{Reg}[Rb]+S^*\text{Reg}[Ri]+D] \]
  
  - **D:** Constant “displacement” 1, 2, or 4 bytes
  - **Rb:** Base register: Any of 16 integer registers
  - **Ri:** Index register: Any, except for \%rsp
  - **S:** Scale: 1, 2, 4, or 8 (*why these numbers?*)

- **Special Cases**
  
  \[ (Rb,Ri) \rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \]
  \[ D(Rb,Ri) \rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] \]
  \[ (Rb,Ri,S) \rightarrow \text{Mem}[\text{Reg}[Rb]+S^*\text{Reg}[Ri]] \]

**Indexed**

Array access with variable index.
Example: 2_addressing_modes.c
index()
Addressing modes

Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>
Table of contents

Announcements

1_swap.s: Assembly implementation of function that swaps memory contents
C data type size and register word sizes

2_addressing_modes.s: Understanding source dest operands and memory addressing modes

3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations
3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

Address Computation Instruction

- **leaq** Src, Dst
  - Src is address mode expression
  - Set Dst 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
  long m12(long x)
  {
    return x*12;
  }
  ```

  Converted to ASM by compiler:
  ```asm
  leaq (%rdi,%rdi,2), %rax # t <- x+x*2
  salq $2, %rax # return t<<2
  ```

Example: 3_leaq.c
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

1_swap.s: Assembly implementation of function that swaps memory contents
C data type size and register word sizes

2_addressing_modes.s: Understanding source dest operands and memory addressing modes

3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations
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;
}
```

\[
\begin{align*}
255 &= \text{1111}_2 \\
    &= \text{0000}_2 \text{1111}_2 \\
    &= 255
\end{align*}
\]

\[
\begin{align*}
127 &= \text{0111}_2 \\
    &= \text{0000}_2 \text{0111}_2 \\
    &= 127
\end{align*}
\]

\[
\begin{align*}
-128 &= \text{1000}_2 \\
    &= \text{1111}_2 \text{1000}_2 \\
    &= -128
\end{align*}
\]
Sign extension due to unsigned and signed data types

Converting to a data type with more bits

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

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

Announcements

1_swap.s: Assembly implementation of function that swaps memory contents
C data type size and register word sizes

2_addressing_modes.s: Understanding source dest operands and memory addressing modes

3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

MOV instruction sign extension

Arithmetic instructions
   Shift operations
   Bitwise operations
   Integer arithmetic operations
Left shift operation

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

```
1 unsigned long sl_ul (  
2     unsigned long in0,  
3     unsigned long in1
4 ) {  
5     return in0<<in1;  
6 }

1 signed long sl_sl (  
2     signed long in0,  
3     signed long in1
4 ) {  
5     return in0<<in1;  
6 }
```

```
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
void 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
void 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>notq dest</td>
<td>( \sim dest \rightarrow dest )</td>
</tr>
<tr>
<td>andq src,dest</td>
<td>src &amp; dest \rightarrow dest</td>
</tr>
<tr>
<td>orq src,dest</td>
<td>src</td>
</tr>
<tr>
<td>xorq src,dest</td>
<td>src \land dest \rightarrow 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>