Machine-Level Representation of Programs: Moving data, arithmetic and logical operations

Yipeng Huang
Rutgers University
March 7, 2024
Table of contents

Announcements
  Programming assignments
  Reading assignments

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations

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

Programming assignment 3

- Due Friday.
Reading assignments

CS:APP Chapters 3.1-3.4
Table of contents

Announcements
  Programming assignments
  Reading assignments

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations

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

1. **Text**
   - C program `(p1.c p2.c)`
2. **Text**
   - Asm program `(p1.s p2.s)`
3. **Binary**
   - Object program `(p1.o p2.o)`
4. **Binary**
   - Executable program `(p)`

**Compiler** `(gcc -Og -S)`

**Assembler** `(gcc or as)`

**Linker** `(gcc or ld)`

**Static libraries** `.a`

**Figure: Stages of compilation. Image credit CS:APP**
Assembly

Human readable machine code

- Very limited
- Not much control flow
- Any more complex functionality is built up
- for loops, while loops, turn into assembly sequence

Choice of what assembly to experiment with

- MIPS
- ARM
- x86 / x86-64 (not ideal for teaching, but it allows us to experiment on ilab)
Assembly instructions

Instructions for the microarchitecture

- Binary streams that tell an electronic circuit what to do
- Fetch, decode, execute, memory, writeback
A preview of microarchitecture

Figure: Stages of compilation. Image credit Wikimedia
Table of contents

Announcements
  Programming assignments
  Reading assignments

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations

2_addressing_modes.s: Understanding source dest operands and memory addressing modes
Why are instruction set architectures important

Interface between computer science and electrical and computer engineering

- Software is varied, changes
- Hardware is standardized, static

Computer architect Fred Brooks and the IBM 360

- IBM was selling computers with different capacities,
- Compile once, and can run software on all IBM machines.
- Backward compatibility.
- An influential idea.
CISC vs. RISC

Complex instruction set computer

- Intel and AMD
- Have an extensive and complex set of instructions
- For example: x86’s extensions: x87, IA-32, x86-64, MMX, 3DNow!, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.2, SSE5, AES-NI, CLMUL, RDRAND, SHA, MPX, SGX, XOP, F16C, ADX, BMI, FMA, AVX, AVX2, AVX512, VT-x, VT-d, AMD-V, AMD-Vi, TSX, ASF
- Can license Intel’s compilers to extract performance
- Secret: inside the processor, they break it down to more elementary instructions
CISC vs. RISC

Reduced instruction set computer

- MIPS, ARM, RISC-V (can find Patterson and Hennessy Computer Organization and Design textbook in each of these versions), and PowerPC
- Have a relatively simple set of instructions
- For example: ARM’s extensions: SVE, SVE2, TME; All mandatory: Thumb-2, Neon, VFPv4-D16, VFPv4 Obsolete: Jazelle
- ARM: smartphones, Apple ARM M1 Mac
Into the future: Post-ISA world

Post-ISA world
► Increasingly, the CPU is not the only character
► It orchestrates among many pieces of hardware
► Smartphone die shot
► GPU, TPU, FPGA, ASIC

Figure: Apple A13 (2019 Apple iPhone 11 CPU). Image credit AnandTech
Table of contents

Announcements
  Programming assignments
  Reading assignments

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations

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

```
C program (p1.c p2.c)
```

```
Compiler (gcc -Og -S)
```

```
Asm program (p1.s p2.s)
```

```
Assembler (gcc or as)
```

```
Object program (p1.o p2.o)
```

```
Static libraries (.a)
```

```
Linker (gcc or ld)
```

```
Executable program (p)
```

- `gcc -Og -S swap.c`
- `objdump -d 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
  Programming assignments
  Reading assignments

Assembly code: Human readable representation of machine code

Instruction set architectures

*swap.s*: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations

*2_addressing_modes.s*: Understanding source dest operands and memory addressing modes
### Assembly syntax

**Instruction Source, Dest**

#### Data size and x86 / x86-64 registers

**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 IA32, x86, 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

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

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

### x86-64 Integer Registers

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

Note the backward compatibility.

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
# Data size and IA32, x86, 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>
<th>ST(0)</th>
<th>MM0</th>
<th>ST(1)</th>
<th>MM1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZMM2</td>
<td>YMM2</td>
<td>XMM2</td>
<td>ZMM3</td>
<td>YMM3</td>
<td>XMM3</td>
<td>ST(2)</td>
<td>MM2</td>
<td>ST(3)</td>
<td>MM3</td>
</tr>
<tr>
<td>ZMM4</td>
<td>YMM4</td>
<td>XMM4</td>
<td>ZMM5</td>
<td>YMM5</td>
<td>XMM5</td>
<td>ST(4)</td>
<td>MM4</td>
<td>ST(5)</td>
<td>MM5</td>
</tr>
<tr>
<td>ZMM6</td>
<td>YMM6</td>
<td>XMM6</td>
<td>ZMM7</td>
<td>YMM7</td>
<td>XMM7</td>
<td>ST(6)</td>
<td>MM6</td>
<td>ST(7)</td>
<td>MM7</td>
</tr>
<tr>
<td>ZMM8</td>
<td>YMM8</td>
<td>XMM8</td>
<td>ZMM9</td>
<td>YMM9</td>
<td>XMM9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM10</td>
<td>YMM10</td>
<td>XMM10</td>
<td>ZMM11</td>
<td>YMM11</td>
<td>XMM11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM12</td>
<td>YMM12</td>
<td>XMM12</td>
<td>ZMM13</td>
<td>YMM13</td>
<td>XMM13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM14</td>
<td>YMM14</td>
<td>XMM14</td>
<td>ZMM15</td>
<td>YMM15</td>
<td>XMM15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM16</td>
<td>ZMM16</td>
<td>ZMM16</td>
<td>ZMM18</td>
<td>ZMM21</td>
<td>ZMM22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM19</td>
<td>ZMM19</td>
<td>ZMM19</td>
<td>ZMM20</td>
<td>ZMM20</td>
<td>ZMM20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM21</td>
<td>ZMM21</td>
<td>ZMM21</td>
<td>ZMM22</td>
<td>ZMM22</td>
<td>ZMM22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM23</td>
<td>ZMM23</td>
<td>ZMM23</td>
<td>ZMM24</td>
<td>ZMM24</td>
<td>ZMM24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM25</td>
<td>ZMM25</td>
<td>ZMM25</td>
<td>ZMM26</td>
<td>ZMM26</td>
<td>ZMM26</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM27</td>
<td>ZMM27</td>
<td>ZMM27</td>
<td>ZMM28</td>
<td>ZMM28</td>
<td>ZMM28</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM29</td>
<td>ZMM29</td>
<td>ZMM29</td>
<td>ZMM30</td>
<td>ZMM30</td>
<td>ZMM30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM31</td>
<td>ZMM31</td>
<td>ZMM31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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
   Programming assignments
   Reading assignments

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers

MOV instruction sign extension

Arithmetic instructions
   Shift operations
   Bitwise operations
   Integer arithmetic operations

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

\[
255 = 1111_{1111_2} = 0000_0000_{1111_{1111_2}} = 255
\]

\[
127 = 0111_{1111_2} = 0000_0000_{0111_{1111_2}} = 127
\]

\[
-128 = 1000_{0000_2} = 1111_{1111_{1000_0000_2}} = -128
\]
Sign extension due to unsigned and signed data types

Converting to a data type with more bits

```c
1 unsigned short uc_to_us (unsigned char input) {
2     return input;
3 }

1 signed short sc_to_ss (signed char input) {
2     return input;
3 }
```

<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
  Programming assignments
  Reading assignments

Assembly code: Human readable representation of machine code

Instruction set architectures

\texttt{swap.s}: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations

\texttt{2\_addressing\_modes.s}: Understanding source dest operands and memory addressing modes
Left shift operation

```
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
```

**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;
}
```

```assembly
dsrx:
    movq %rdi, %rax
    movb %sil, %cl
    shrxq %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;
}
```

```assembly
dsrl:
    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</code> <code>dest</code></td>
<td><code>~ dest → dest</code></td>
</tr>
<tr>
<td><code>andq</code> <code>src,dest</code></td>
<td><code>src&amp;dest → dest</code></td>
</tr>
<tr>
<td><code>orq</code> <code>src,dest</code></td>
<td>`src</td>
</tr>
<tr>
<td><code>xorq</code> <code>src,dest</code></td>
<td><code>src ∧ dest → dest</code></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>
Table of contents

Announcements
   Programming assignments
   Reading assignments

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers

MOV instruction sign extension

Arithmetic instructions
   Shift operations
   Bitwise operations
   Integer arithmetic operations

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

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Imm</strong></td>
<td>movq $0x4,%rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Mem</strong></td>
<td>movq $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Reg</strong></td>
<td>movq %rax,%rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Mem</strong></td>
<td>movq %rax,(%rdx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td><strong>Reg</strong></td>
<td>movq (%rax),%rdx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction*
Simple Memory Addressing Modes

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

- **Displacement** \(D(R)\) \(\text{Mem}[\text{Reg}[R]+D]\)
  - Register \(R\) specifies start of memory region
  - Constant displacement \(D\) specifies offset
    
    \[
    \text{movq } 8(\%rbp),\%rdx
    \]
Addressing modes

Complete Memory Addressing Modes

■ Most General Form

\[ D(R_{b}, R_{i}, S) \quad \text{Mem}\left[R_{b}\right]+S\ast R_{i}+D \]

- D: Constant “displacement” 1, 2, or 4 bytes
- \( R_{b} \): Base register: Any of 16 integer registers
- \( R_{i} \): Index register: Any, except for \( %r_{sp} \)
- S: Scale: 1, 2, 4, or 8 (why these numbers?)

■ Special Cases

\[
\begin{align*}
(R_{b}, R_{i}) & \quad \text{Mem}[R_{b}]+R_{i} \\
D(R_{b}, R_{i}) & \quad \text{Mem}[R_{b}]+R_{i}+D \\
(R_{b}, R_{i}, S) & \quad \text{Mem}[R_{b}]+S\ast R_{i}
\end{align*}
\]

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>
C code

void immediate ( long * ptr ) {
    *ptr = 0xFFFFFFFFFFFFFFFF;
}

Assembly code

immediate:
    movq $-1, (%rdi)
    ret

$ indicates the immediate value; corresponds to literals in C
(%rdi) indicates memory location at address stored in %rdi register
void displacement_l ( long * ptr ) {
    ptr[1] = 0xFFFFFFFFFFFFFFFF;
}

displacement_l:  
    movq $-1, 8(%rdi)  
    ret

▶ 8(%rdi) indicates memory location at address stored in %rdi register + 8
2_addressing_modes.c: Imm→Mem (with displacement)

<table>
<thead>
<tr>
<th>function signature</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>void displacement_c ( char * ptr );</td>
<td>movb $-1, 1(%rdi)</td>
</tr>
<tr>
<td>void displacement_s ( short * ptr );</td>
<td>movw $-1, 2(%rdi)</td>
</tr>
<tr>
<td>void displacement_i ( int * ptr );</td>
<td>movl $-1, 4(%rdi)</td>
</tr>
<tr>
<td>void displacement_l ( long * ptr );</td>
<td>movq $-1, 8(%rdi)</td>
</tr>
</tbody>
</table>
C code

```c
void index_l ( long * ptr, long index ) {
    ptr[index] = 0xFFFFFFFFFFFFFFFF;
}
```

Assembly code

```assembly
index_l:
    movq $-1, (%rdi,%rsi,8)
    ret
```

▶ (%rdi,%rsi,8) indicates memory location at address stored in %rdi register + 8 × value stored in %rsi register
## 2_addressing_modes.c: Imm→Mem (with index)

<table>
<thead>
<tr>
<th>function signature</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>void index_c ( char * ptr, long index );</td>
<td>movb $-1, (%rdi,%rsi)</td>
</tr>
<tr>
<td>void index_s ( short * ptr, long index );</td>
<td>movw $-1, (%rdi,%rsi,2)</td>
</tr>
<tr>
<td>void index_i ( int * ptr, long index );</td>
<td>movl $-1, (%rdi,%rsi,4)</td>
</tr>
<tr>
<td>void index_l ( long * ptr, long index );</td>
<td>movq $-1, (%rdi,%rsi,8)</td>
</tr>
</tbody>
</table>
2_addressing_modes.c: Imm→Mem (with displacement and index)

C code

```c
void displacement_and_index ( long * ptr, long index ) {
    ptr[index+1] = 0xFFFFFFFFFFFFFFFF;
}
```

Assembly code

```assembly
displacement_and_index:
    movq $-1, 8(%rdi,%rsi,8)
    ret

▶ 8(%rdi,%rsi,8) indicates memory location at address stored in %rdi register + 8 × value stored in %rsi register + 8
```