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

Yipeng Huang

Rutgers University

March 9, 2023
Table of contents

Announcements
  Quizzes and programming assignments
  Reading assignments

Instruction set architectures

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

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
Quizzes and programming assignments

Short quiz 5
- Due Friday. All about floats.

Programming assignment 3
- Has been out, due Friday before spring break.
Reading assignments

CS:APP Chapters 3.1-3.4
Table of contents

Announcements
   Quizzes and programming assignments
   Reading assignments

Instruction set architectures

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

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)  \arrow{right} \text{Compiler (gcc -Og -S)} \rightarrow
Asm program (p1.s p2.s) \arrow{right} \text{Assembler (gcc or as)} \rightarrow
Object program (p1.o p2.o) \arrow{right} \text{Linker (gcc or ld)} \rightarrow
Executable program (p) \arrow{right} \text{Static libraries (.a)}
```

Figure: Stages of compilation. Image credit CS:APP
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
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)
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
   Quizzes and programming assignments
   Reading assignments

Instruction set architectures

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

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
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`

```
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 );
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>
## Some History: IA32 Registers

### General Purpose 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.

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

### 16-bit Virtual Registers (backwards compatibility)

- %esp
- %ebp
Data size and x86, IA32, and x86-64 registers

---

**x86-64 Integer Registers**

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

| ZMM0   | YMM0   | XMM0   | ZMM1   | YMM1   | XMM1   | ST(0) MM0 | ST(1) MM1 | AX | BX | CX | DX | SI | DI | R8 | R9 | R10 | R11 | R12 | R13 | R14 | R15 |
|--------|--------|--------|--------|--------|--------|---------|----------|---|---|---|---|----|----|----|---|---|---|---|----|----|----|----|
| ZMM2   | YMM2   | XMM2   | ZMM3   | YMM3   | XMM3   | ST(2) MM2 | ST(3) MM3 | R0 | R1 | R2 | R3 |    |    |    |   |   |   |   |    |    |    |    |
| ZMM4   | YMM4   | XMM4   | ZMM5   | YMM5   | XMM5   | ST(4) MM4 | ST(5) MM5 | R4 | R5 | R6 | R7 |    |    |    |   |   |   |   |    |    |    |    |
| ZMM6   | YMM6   | XMM6   | ZMM7   | YMM7   | XMM7   | ST(6) MM6 | ST(7) MM7 | R8 | R9 | R10| R11|    |    |    |   |   |   |   |    |    |    |    |
| ZMM8   | YMM8   | XMM8   | ZMM9   | YMM9   | XMM9   |          |          |     |   |   |   |    |    |    |   |   |   |   |    |    |    |    |
| ZMM10  | YMM10  | XMM10  | ZMM11  | YMM11  | XMM11  |          |          |     |   |   |   |    |    |    |   |   |   |   |    |    |    |    |
| ZMM12  | YMM12  | XMM12  | ZMM13  | YMM13  | XMM13  |          |          |     |   |   |   |    |    |    |   |   |   |   |    |    |    |    |
| ZMM14  | YMM14  | XMM14  | ZMM15  | YMM15  | XMM15  |          |          |     |   |   |   |    |    |    |   |   |   |   |    |    |    |    |
| ZMM16  | ZMM17  | ZMM18  | ZMM19  | ZMM20  | ZMM21  | ZMM22  | ZMM23  |    |   |   |   |    |    |    |   |   |   |   |    |    |    |    |
| ZMM24  | ZMM25  | ZMM26  | ZMM27  | ZMM28  | ZMM29  | ZMM30  | ZMM31  |    |   |   |   |    |    |    |   |   |   |   |    |    |    |    |

**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
  Quizzes and programming assignments
  Reading assignments

Instruction set architectures

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

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

```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 }
```

\[
\begin{align*}
255 &= 1111_{1111} \_2 \\
    &= 0000_0000_1111_{1111} \_2 \\
    &= 255 \\
127 &= 0111_{1111} \_2 \\
    &= 0000_0000_0111_{1111} \_2 \\
    &= 127 \\
-128 &= 1000_0000 \_2 \\
    &= 1111_{1111}1000_0000 \_2 \\
    &= -128
\end{align*}
\]
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 }
```

```c
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
  Quizzes and programming assignments
  Reading assignments

Instruction set architectures

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

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
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

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

```
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

<table>
<thead>
<tr>
<th>Assembly instruction</th>
<th>Instruction effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>notq dest</td>
<td>∼ dest → dest</td>
</tr>
<tr>
<td>andq src, dest</td>
<td>src&amp;dest → dest</td>
</tr>
<tr>
<td>orq src, dest</td>
<td>src</td>
</tr>
<tr>
<td>xorq src, dest</td>
<td>src ∧ dest → 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>
Table of contents

Announcements
    Quizzes and programming assignments
    Reading assignments

Instruction set architectures

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

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, register, and memory

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>Imm</td>
<td>Reg</td>
<td>movq $0x4,%rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq %rax,%rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq %rax,(%rdx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</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**  
  \( (R) \rightarrow \text{Mem[Reg[R]]} \)
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

  \[
  \text{movq} \ (%\text{rcx}),\%\text{rax}
  \]

- **Displacement**  
  \( D(R) \rightarrow \text{Mem[Reg[R]+D]} \)
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

  \[
  \text{movq} \ 8(%\text{rbp}),\%\text{rdx}
  \]

**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**
  
  \[ D(Rb, Ri, S) \rightarrow Mem[Reg[Rb]+S*Reg[Ri]+D] \]
  
  - 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 Mem[Reg[Rb]+Reg[Ri]]
  - \(D(Rb, Ri)\) \rightarrow Mem[Reg[Rb]+Reg[Ri]+D]
  - \((Rb, Ri, S)\) \rightarrow Mem[Reg[Rb]+S*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>

%rdx 0xf000
%rcx 0x0100
2_addressing_modes.c: Imm → Mem

C code

```c
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
**C code**

```c
void displacement_l ( long * ptr ) {
    ptr[1] = 0xFFFFFFFFFFFFFFFF;
}
```

**Assembly code**

```assembly
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>
2_addressing_modes.c: Imm→Mem (with index)

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