

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

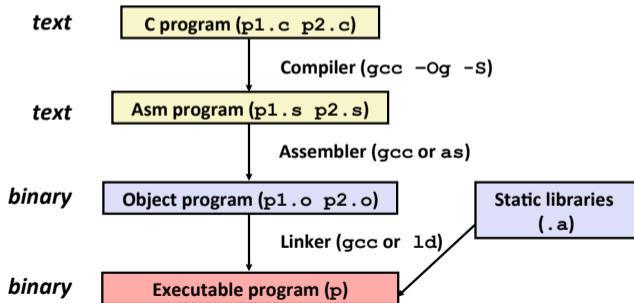
- Bitwise operations

- Integer arithmetic operations

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

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`



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

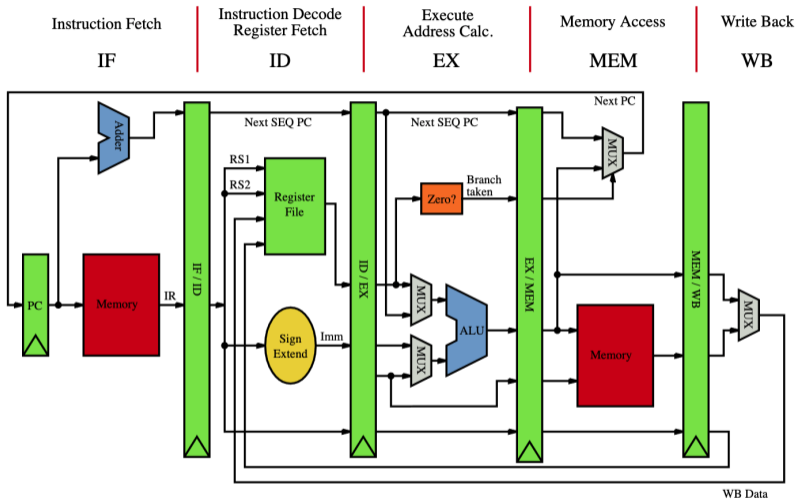


Figure: Stages of compilation. Image credit Wikimedia

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.

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

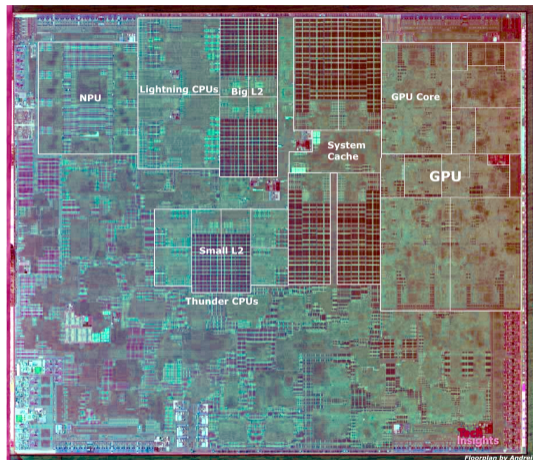


Figure: Apple A13 (2019 Apple iPhone 11 CPU). Image credit AnandTech

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

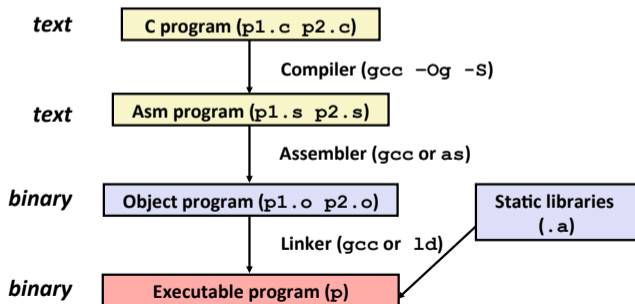
- Integer arithmetic operations

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

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`



▶ `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

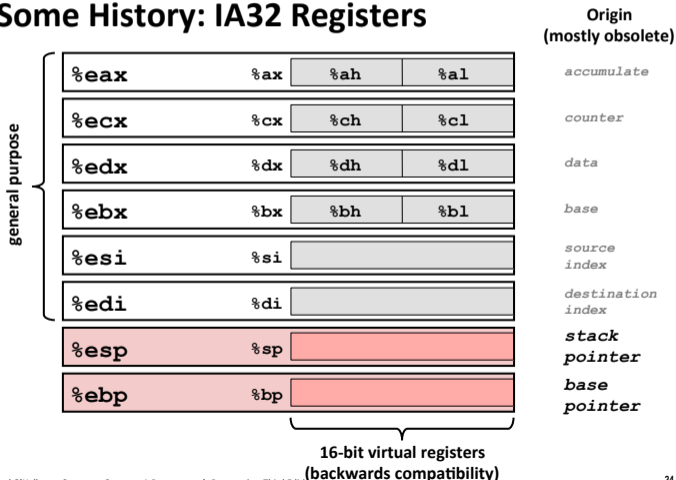
1. `void swap_c (char*a, char*b);`
2. `void swap_s (short*a, short*b);`
3. `void swap_i (int*a, int*b);`
4. `void swap_l (long*a, long*b);`

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

Some History: IA32 Registers

| data type | registers |
|-----------|------------|
| char | %al, %dl |
| short | %ax, %dx |
| int | %eax, %edx |
| long | %rax, %rdx |

Note the backward compatibility.



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

x86-64 Integer Registers

| | | | |
|-------------------|-------------------|-------------------|--------------------|
| <code>%rax</code> | <code>%eax</code> | <code>%r8</code> | <code>%r8d</code> |
| <code>%rbx</code> | <code>%ebx</code> | <code>%r9</code> | <code>%r9d</code> |
| <code>%rcx</code> | <code>%ecx</code> | <code>%r10</code> | <code>%r10d</code> |
| <code>%rdx</code> | <code>%edx</code> | <code>%r11</code> | <code>%r11d</code> |
| <code>%rsi</code> | <code>%esi</code> | <code>%r12</code> | <code>%r12d</code> |
| <code>%rdi</code> | <code>%edi</code> | <code>%r13</code> | <code>%r13d</code> |
| <code>%rsp</code> | <code>%esp</code> | <code>%r14</code> | <code>%r14d</code> |
| <code>%rbp</code> | <code>%ebp</code> | <code>%r15</code> | <code>%r15d</code> |

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

| data type | registers |
|-----------|---------------------------------------|
| char | <code>%al</code> , <code>%dl</code> |
| short | <code>%ax</code> , <code>%dx</code> |
| int | <code>%eax</code> , <code>%edx</code> |
| long | <code>%rax</code> , <code>%rdx</code> |

Note the backward compatibility.

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

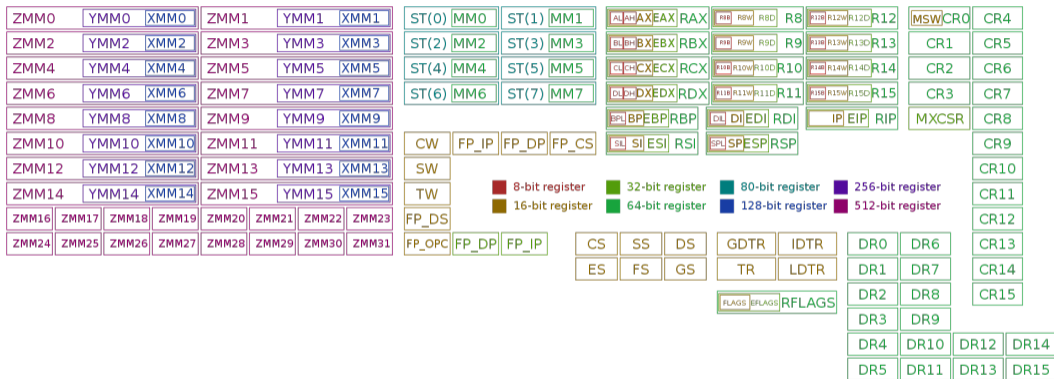


Figure: x86-64 with SIMD extensions registers. Image credit: https://commons.wikimedia.org/wiki/File:Table_of_x86_Registers_svg.svg

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

```
1 unsigned short uc_to_us (  
2     unsigned char input  
3 ) {  
4     return input;  
5 }
```

```
1 signed short sc_to_ss (  
2     signed char input  
3 ) {  
4     return input;  
5 }
```

$$\begin{aligned} 255 &= 1111_1111_2 \\ &= 0000_0000_1111_1111_2 \\ &= 255 \end{aligned}$$

$$\begin{aligned} 127 &= 0111_1111_2 \\ &= 0000_0000_0111_1111_2 \\ &= 127 \end{aligned}$$

$$\begin{aligned} -128 &= 1000_0000_2 \\ &= 1111_1111_1000_0000_2 \\ &= -128 \end{aligned}$$

Sign extension due to unsigned and signed data types

Converting to a data type with more bits

```
1 unsigned short uc_to_us (  
2     unsigned char input  
3 ) {  
4     return input;  
5 }
```

```
1 signed short sc_to_ss (  
2     signed char input  
3 ) {  
4     return input;  
5 }
```

| 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, %eax |
| signed short sc_to_ss (signed char input); | movsbw %dil, %eax |

- ▶ 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

```
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 }
```

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` → `%rdi` → `%rax`
- ▶ `movb: in1` → `%sil` → `%cl`
- ▶ `salq src, dest:`
(`dest << src`) → `dest`
- ▶ Why only use `movb` for `in1`?

Bitwise operations

| Assembly instruction | Instruction effect |
|-----------------------------|---|
| <code>notq dest</code> | $\sim \text{dest} \rightarrow \text{dest}$ |
| <code>andq src, dest</code> | $\text{src} \& \text{dest} \rightarrow \text{dest}$ |
| <code>orq src, dest</code> | $\text{src} \text{dest} \rightarrow \text{dest}$ |
| <code>xorq src, dest</code> | $\text{src} \wedge \text{dest} \rightarrow \text{dest}$ |

Integer arithmetic operations

| Assembly instruction | Instruction effect |
|------------------------------|---|
| <code>incq dest</code> | $\text{dest} + 1 \rightarrow \text{dest}$ |
| <code>decq dest</code> | $\text{dest} - 1 \rightarrow \text{dest}$ |
| <code>negq dest</code> | $-\text{dest} \rightarrow \text{dest}$ |
| <code>addq src, dest</code> | $\text{src} + \text{dest} \rightarrow \text{dest}$ |
| <code>subq src, dest</code> | $\text{src} - \text{dest} \rightarrow \text{dest}$ |
| <code>imulq src, dest</code> | $\text{src} \times \text{dest} \rightarrow \text{dest}$ |

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

| | Source | Dest | Src, Dest | C Analog |
|------|--------|------|----------------------------------|-----------------------------|
| movq | Imm | Reg | <code>movq \$0x4, %rax</code> | <code>temp = 0x4;</code> |
| | | Mem | <code>movq \$-147, (%rax)</code> | <code>*p = -147;</code> |
| | Reg | Reg | <code>movq %rax, %rdx</code> | <code>temp2 = temp1;</code> |
| | | Mem | <code>movq %rax, (%rdx)</code> | <code>*p = temp;</code> |
| | Mem | Reg | <code>movq (%rax), %rdx</code> | <code>temp = *p;</code> |
| | | | | |

Cannot do memory-memory transfer with a single instruction

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()

Complete Memory Addressing Modes

■ Most General Form

$D(Rb, Ri, S)$ $Mem[Reg[Rb]+S*Reg[Ri]+ D]$

- 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) $Mem[Reg[Rb]+Reg[Ri]]$

$D(Rb, Ri)$ $Mem[Reg[Rb]+Reg[Ri]+D]$

(Rb, Ri, S) $Mem[Reg[Rb]+S*Reg[Ri]]$

Indexed

Array access with
variable index.

Example: `2_addressing_modes.c`
`index()`

Address Computation Examples

| | |
|-------------------|---------------------|
| <code>%rdx</code> | <code>0xf000</code> |
| <code>%rcx</code> | <code>0x0100</code> |

| Expression | Address Computation | Address |
|----------------------------|-------------------------------|----------------------|
| <code>0x8(%rdx)</code> | <code>0xf000 + 0x8</code> | <code>0xf008</code> |
| <code>(%rdx,%rcx)</code> | <code>0xf000 + 0x100</code> | <code>0xf100</code> |
| <code>(%rdx,%rcx,4)</code> | <code>0xf000 + 4*0x100</code> | <code>0xf400</code> |
| <code>0x80(,%rdx,2)</code> | <code>2*0xf000 + 0x80</code> | <code>0x1e080</code> |

2_addressing_modes.c: Imm→Mem

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

2_addressing_modes.c: Imm→Mem (with displacement)

C code

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

Assembly code

```
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)

| function signature | assembly code |
|---|---------------------------------|
| <code>void displacement_c (char * ptr);</code> | <code>movb \$-1, 1(%rdi)</code> |
| <code>void displacement_s (short * ptr);</code> | <code>movw \$-1, 2(%rdi)</code> |
| <code>void displacement_i (int * ptr);</code> | <code>movl \$-1, 4(%rdi)</code> |
| <code>void displacement_l (long * ptr);</code> | <code>movq \$-1, 8(%rdi)</code> |

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

C code

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

- ▶ `(%rdi,%rsi,8)` indicates memory location at address stored in `%rdi` register + $8 \times$ value stored in `%rsi` register

Assembly code

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

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

| function signature | assembly code |
|--|---------------------------------------|
| <code>void index_c (char * ptr, long index);</code> | <code>movb \$-1, (%rdi,%rsi)</code> |
| <code>void index_s (short * ptr, long index);</code> | <code>movw \$-1, (%rdi,%rsi,2)</code> |
| <code>void index_i (int * ptr, long index);</code> | <code>movl \$-1, (%rdi,%rsi,4)</code> |
| <code>void index_l (long * ptr, long index);</code> | <code>movq \$-1, (%rdi,%rsi,8)</code> |

2_addressing_modes.c: Imm→Mem (with displacement and index)

C code

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

- ▶ `8(%rdi,%rsi,8)` indicates memory location at address stored in `%rdi` register + $8 \times$ value stored in `%rsi` register + 8

Assembly code

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