Machine-Level Representation of Programs: Instruction set architectures

Yipeng Huang

Rutgers University

October 23, 2025

Table of contents

Reading assignments

CS:APP Chapters 3.1-3.4

Table of contents

Computer organization

Layer cake: remember the first day of class, we discussed what are parts of a computer?

- Society
- Human beings
- Applications
- Algorithms
- High-level programming languages
- Interpreters
- Low-level programming languages
- Compilers
- Architectures
- Microarchitectures
- Sequential/combinational logic
- Transistors
- Semiconductors

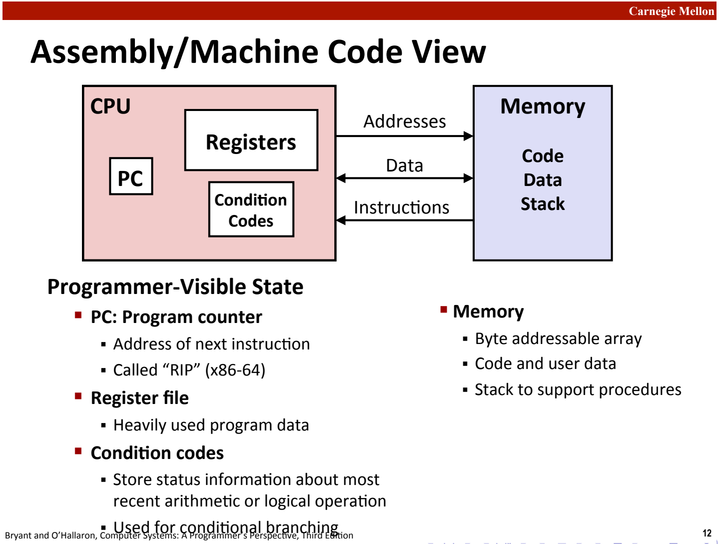
Stored program computer

Stored program:

Instructions reside in memory, loaded as needed.

von Neumann architecture:

Data and instructions share same connection to memory.



Memory hierarchy

	Capacity	Access speed
Internet		
Tape	250Pb	
Hard drives	16TB	2Mb/s
Solid state drives	4TB	2Gb/s
DRAM	8Gb - 1Tb+	8Gb/s
Last-level cache	64Mb	
Level-1 cache	1Mb	
Registers	1Kb	

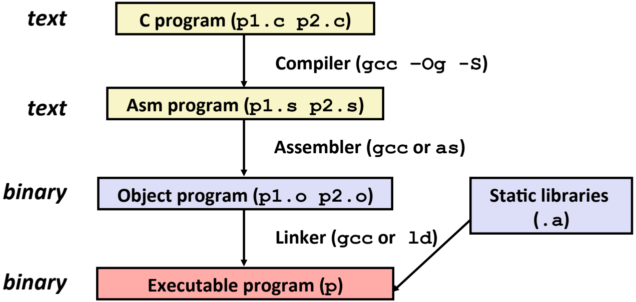
- Registers (.25ns; 4GHz => .25e-9s)

Table of contents

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`



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

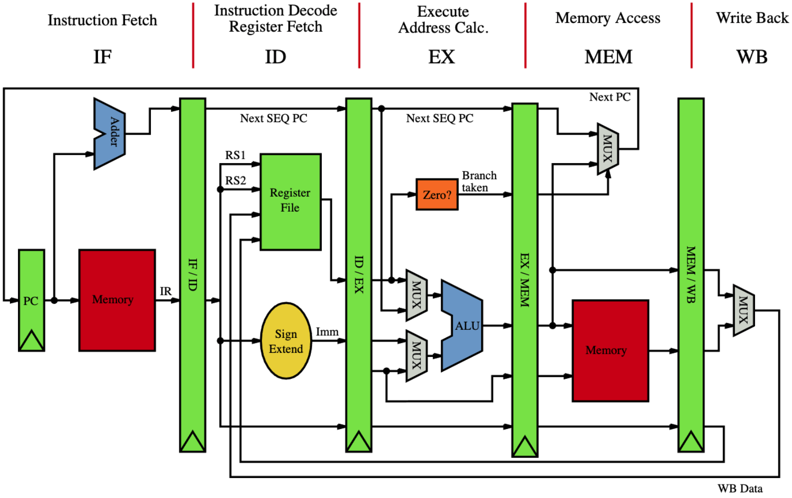


Figure: Stages of compilation. Image credit Wikimedia

Table of contents

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

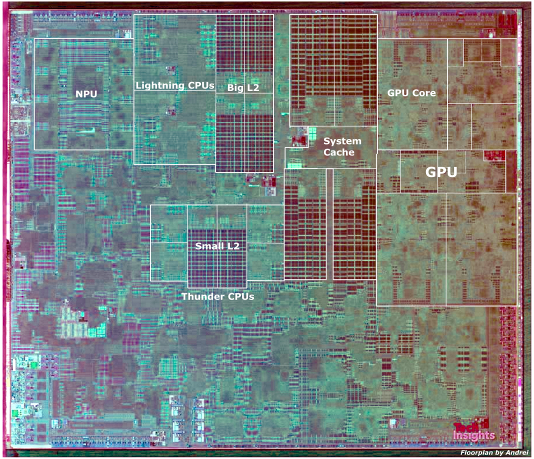
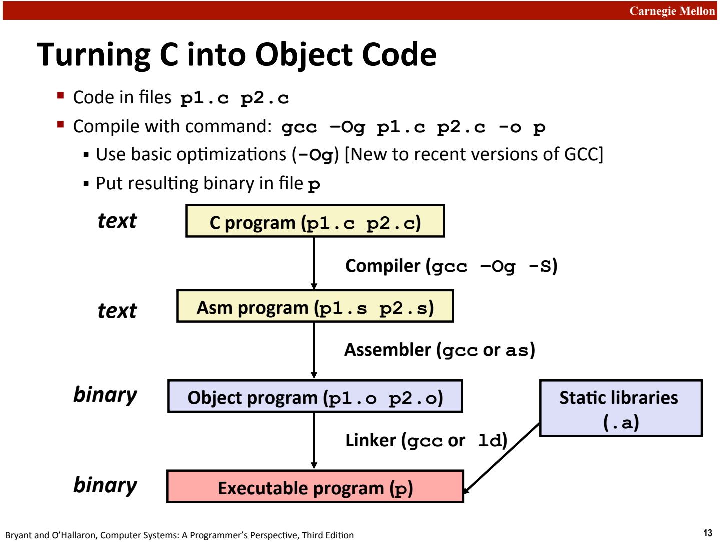


Figure: Apple A13 (2019 Apple iPhone 11 CPU). Image credit AnandTech

Table of contents

Unraveling the compilation chain



- gcc -0g -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

Data size and x86 / x86-64 registers

Assembly syntax

Instruction Source, Dest

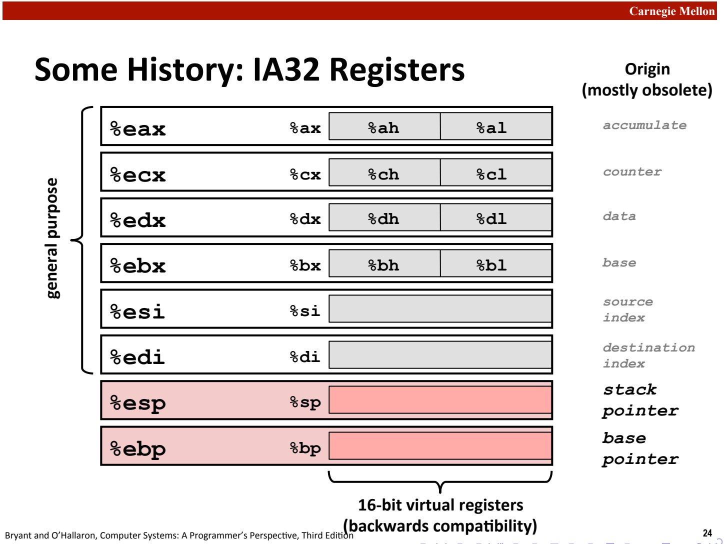
```
swap_l:
 movq (%rsi), %rax
 movq (%rdi), %rdx
 movq %rdx, (%rsi)
 movq %rax, (%rdi)
 ret
```

swap	data type	mov operation	registers
swap_uc	unsigned char	movb (move byte)	%al, %dl
swap_sc	signed char	movb (move byte)	%al, %dl
swap_c	char	movb (move byte)	%al, %dl
swap_s	short	movw (move word)	%ax, %dx
swap_i	int	movl	%eax, %edx
swap_l	long	movq	%rax, %rdx

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

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

Note the backward compatibility.



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

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

Note the backward compatibility.

x86-64 Integer Registers

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

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

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

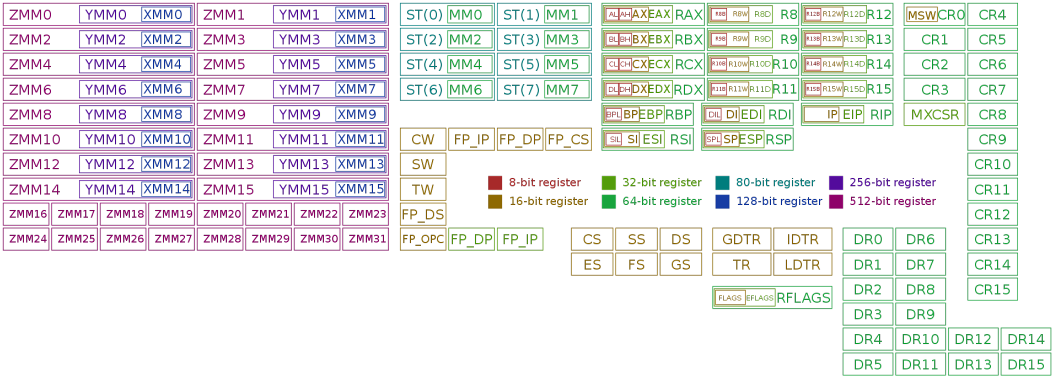


Figure: x86-64 with SIMD extensions registers. Image credit: `https://commons.wikimedia.org/wiki/File:Table_of_x86_Registers_svg.svg`

Table of contents

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;
}
```

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

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

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

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

```c
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

```c
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`	$\sim \text{dest} \to \text{dest}$
`andq src,dest`	$\text{src}\&\text{dest} \to \text{dest}$
`orq src,dest`	$\text{src}\|\text{dest} \to \text{dest}$
`xorq src,dest`	$\text{src} \wedge \text{dest} \to \text{dest}$

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

Table of contents

Immediate, register, and memory

Immediate

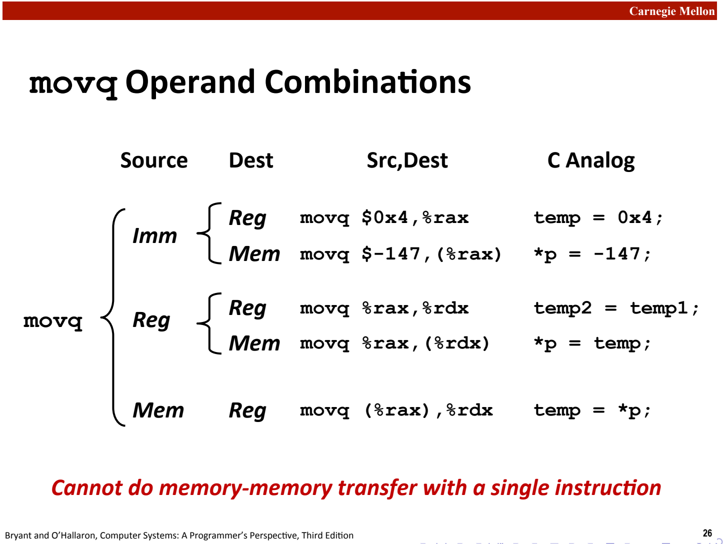
Constant integer values. Example: 2_addressing_modes.c immediate()

Register

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

Memory

Access to memory at calculated



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)   **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

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

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
void displacement_c (char * ptr);	movb $-1, 1(%rdi)
void displacement_s (short * ptr);	movw $-1, 2(%rdi)
void displacement_i (int * ptr);	movl $-1, 4(%rdi)
void displacement_l (long * ptr);	movq $-1, 8(%rdi)

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

C code

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

Assembly code

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

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

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

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

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

C code

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

Assembly code

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

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