程序的机器级表示：基础知识

Machine-Level Programming : Basics

本章内容

Topic

- Intel处理器体系结构的历史

  History of Intel processors and architectures

- C语言，汇编语言和机器语言

  C, assembly and machine code

Intel处理器体系结构的历史

History of Intel processors and architectures

Intel x86系列处理器
Intel x86 Processors

- 在笔记本、桌面和服务器市场占有统治低位

  Totally dominate laptop/desktop/server market

- 一个不断进化的设计过程

  Evolutionary design

  - 向后兼容至8086处理器（诞生于1978年）

    Backwards compatible up until 8086, introduced in 1978

  - 随着时间增加了许多新的特性

    Added more features as time goes on

- 复杂指令集计算机（CISC）

  Complex instruction set computer (CISC)

  - 指令多、格式复杂

    Many different instructions with many different formats

    - 在Linux程序中只使用其中一个子集

      But, only small subset encountered with Linux programs

  - 理论上CISC的性能很难与精简指令集计算机（RISC）相匹敌

    Hard to match performance of Reduced Instruction Set Computers (RISC)

  - 但是Intel采用了CISC

    But, Intel has done just that!

    - 在低功耗情况下，速度会有影响

      In terms of speed. Less so for low power.

Intel处理器体系结构的历史

History of Intel processors and architectures

Intel x86系列的进化：里程碑
Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
8086	1978	29K	5-10

- 第一个16位Intel处理器，IBM PC计算机， DOS操作系统
  First 16-bit Intel processor. Basis for IBM PC & DOS
- 1MB寻址空间
  1MB address space

386	1985	275K	16-33

- 第一个32位Intel处理器，采用IA32体系结构
  First 32 bit Intel processor , referred to as IA32
- 增加了扁平寻址，可以运行Unix操作系统
  Added “flat addressing”, capable of running Unix

Pentium 4E	2004	125M	2800-3800

- 第一个64位Intel x86处理器，采用x86-64体系架构
  First 64-bit Intel x86 processor, referred to as x86-64

Core 2	2006	291M	1060-3500

- 第一个双核Intel处理器
  First multi-core Intel processor

Core i7	2008	731M	1700-3900
Core i9	2017	7G	3300-4500

Intel处理器体系结构的历史

History of Intel processors and architectures

Intel x86系列处理器
Intel x86 Processors

- 新特性的加入

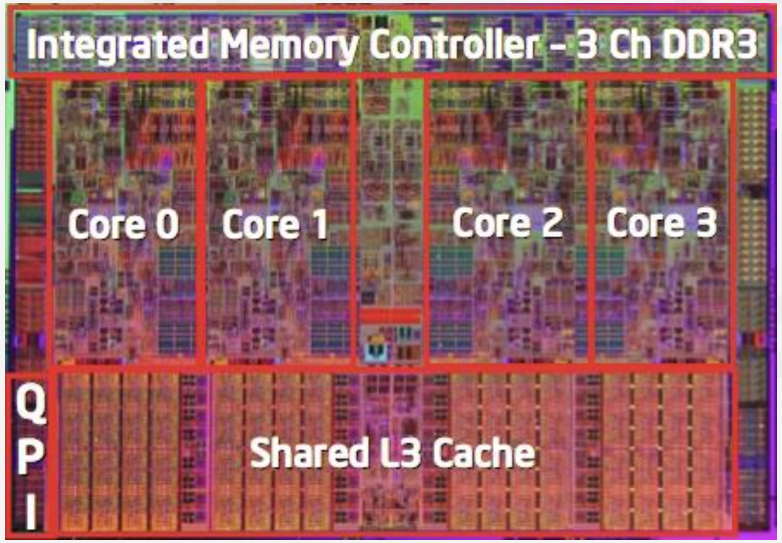
  Added Features
  - 多媒体操作的指令支持

    Instructions to support multimedia operations
  - 提供了更加有效率的条件操作指令

    Instructions to enable more efficient conditional operations
  - 机器字长从32位变为64位

    Transition from 32 bits to 64 bits
  - 多核

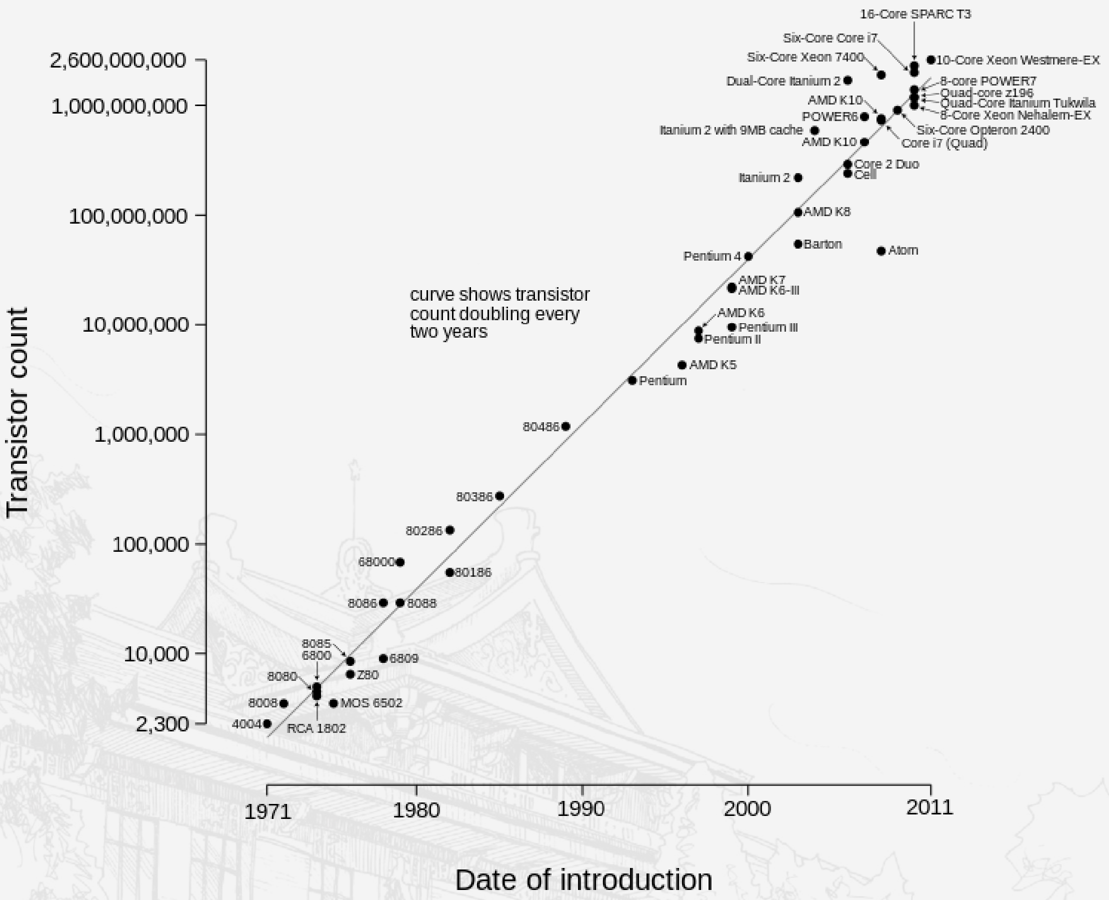
    More cores



Intel处理器体系结构的历史

History of Intel processors and architectures

摩尔定律 Moore’s Law



- 单位面积上可以容纳的晶体管数量几乎每两年增加一倍

The number of transistors in a dense integrated circuit doubles approximately every two years.

Intel处理器体系结构的历史

History of Intel processors and architectures

Core i7 Broadwell 2015

- 桌面版

  Desktop Model

  - 4核

    4 cores

  - 集成图形单元（显卡）

    Integrated graphics

  - 3.3-3.8 GHz
  - 65W

- 服务器版

  Server Model

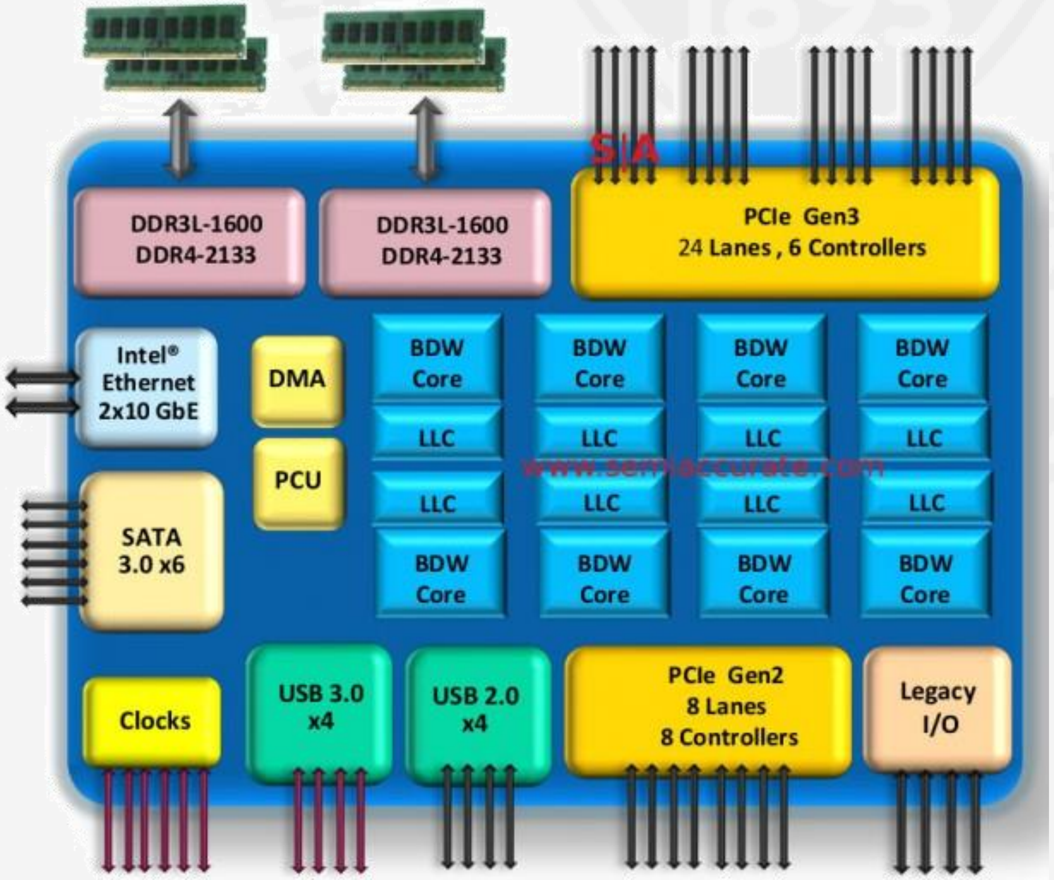
  - 8核

    8 cores

  - 集成 I/O

    Integrated I/O

  - 2-2.6 GHz
  - 45W



X86兼容处理器：AMD
x86 Clones: Advanced Micro Devices (AMD)

- 历史上
  Historically
  - AMD仅仅跟随着Intel
    AMD has followed just behind Intel
  - 性能稍差，价格更便宜
    A little bit slower, a lot cheaper
- 随后
  Then
  - 从DEC和其他业绩下降的公司招聘顶级电路设计师
    Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - 提出了Opteron架构：成为Pentium 4的有力竞争对手
    Built Opteron: tough competitor to Pentium 4
  - 引入了x86-64架构，自主的扩展到64位体系结构
    Developed x86-64, their own extension to 64 bits

- 近年来
  Recent Years
  - Intel重新迎头赶上，重新占据了半导体技术的世界主导地位
    Intel got its act together，leading the world in semiconductor technology
  - AMD稍有落后，依赖外部的半导体代工厂
    AMD has fallen behind, which relies on external semiconductor manufacturer

Intel处理器体系结构的历史

History of Intel processors and architectures

Intel 64位体系结构发展的历史
Intel's 64-Bit History

- 2001年：Intel试图从IA32彻底转变为IA64
  2001: Intel Attempted Radical Shift from IA32 to IA64
  - 完全不同的体系结构（安腾）
    Totally different architecture (Itanium)
  - 以legacy（传统）模式执行IA32的指令
    Executes IA32 code only as legacy
  - 性能令人失望
    Performance disappointing
- 2003年：AMD提出了体系结构进化的解决方案
  2003: AMD Stepped in with Evolutionary Solution
  - x86-64位体系结构（现称为AMD64）
    x86-64 (now called "AMD64")
- Intel觉得有义务专注于IA64
  Intel Felt Obligated to Focus on IA64
  - 难以承认技术路线的错误以及AMD方案更优
    Hard to admit mistake or that AMD is better

- 2004年：Intel提出了EM64T体系结构实现对IA32的64位扩展
  2004: Intel Announces EM64T extension to IA32
  - 扩展实现了64位内存寻址技术
    Extended Memory 64-bit Technology
  - 几乎与x86-64相同
    Almost identical to x86-64!
- 2019年：英特尔宣布放弃IA64架构
  2019: Intel announced abandonment of IA64
- 除低端x86处理器外，其他处理器均支持x86-64
  All but low-end x86 processors support x86-64
  - 但是目前许多程序仍然在32位模式下运行
    But, lots of code still runs in 32-bit mode

本章内容

Topic

- Intel处理器体系结构的历史

  History of Intel processors and architectures

- C语言，汇编语言和机器语言

  C, assembly and machine code

C语言，汇编语言和机器语言

C, assembly and machine code

定义
Definitions

- **体系结构：** （指令集体系结构，ISA）编写汇编代码时需要理解的处理器设计部分。
  Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand to write assembly code.
  - 例如：指令集规范、寄存器组织
    Examples: instruction set specification, registers.
- **微体系结构：** 体系结构的具体实现
  Microarchitecture: Implementation of the architecture.
  - 例如：高速缓存大小、核心频率
    Examples: cache sizes and core frequency.

- 代码格式
  Code Forms
  - **机器语言：** 处理器可以直接执行的字节级的程序
    Machine Code: The byte-level programs that a processor executes
  - **汇编语言：** 文本形式的机器语言
    Assembly Code: A text representation of machine code

- 举例：常见的指令集体系结构
  Example ISAs:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM：几乎所有的移动电话中都使用
    ARM: Used in almost all mobile phones

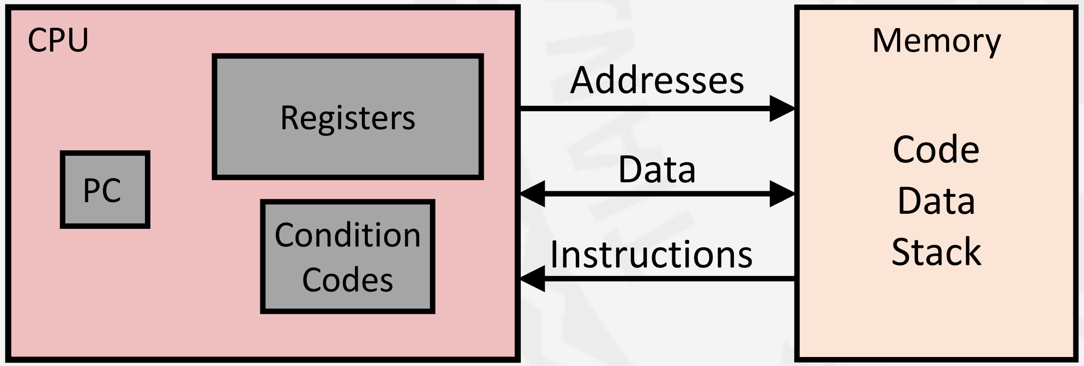
C语言，汇编语言和机器语言

C, assembly and machine code

汇编语言/机器语言视角下的计算机
Assembly/Machine Code View

程序员可见的状态

Programmer-Visible State



- PC：程序计数器
  PC: Program counter
  - 存储下一条要执行指令的地址
    Address of next instruction
  - 在x86-64中的名称为 RIP
    Called “RIP” (x86-64)
- 寄存器文件
  Register file
  - 程序的数据会频繁地使用它来存储
    Heavily used program data

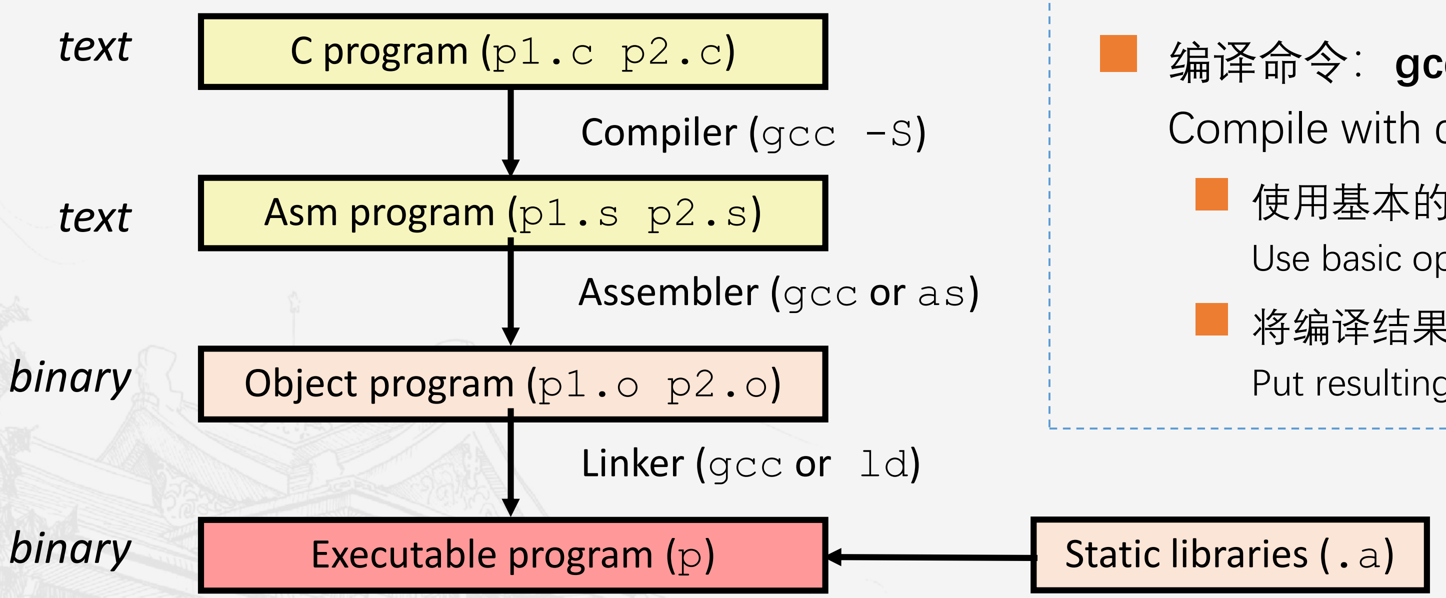
- 条件码
  Condition codes
  - 存储最近一次算术逻辑运算的状态信息
    Store status information about most recent arithmetic or logical operation
  - 用于条件分支
    Used for conditional branching

- 存储器
  Memory
  - 基于字节寻址的阵列
    Byte addressable array
  - 存储程序和用户数据
    Code and user data
  - 存储栈数据（以实现过程的支持）
    Stack to support procedures

C语言，汇编语言和机器语言

C, assembly and machine code

将C语言代码转换为机器语言
Turning C into Object Code



- 代码文件：p1.c p2.c
  Code in files p1.c p2.c
- 编译命令：**gcc –Og p1.c p2.c -o p**
  Compile with command: **gcc –Og p1.c p2.c -o p**
  - 使用基本的编译优化选项 **–Og** **（最新版本GCC支持）**
    Use basic optimizations **-Og** [New to recent versions of GCC]
  - 将编译结果写入文件 **p**
    Put resulting binary in file **p**

C语言，汇编语言和机器语言

C, assembly and machine code

将C语言代码编译为汇编代码
Compiling Into Assembly

C代码

C Code (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y, long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

X86-64汇编

Generated x86-64 Assembly

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

- 使用下面的命令生成
  Obtain with command
  gcc –Og –S sum.c
- 生成文件：**sum.s**
  Produces file **sum.s**

警告：由于编译选项的不同和gcc版本的不同可能会得到不同的编译结果

Warning: Will get very different results on other machines due to different versions of gcc and different compiler settings.

汇编语言的特征：数据类型
Assembly Characteristics: Data Types

- **整数：**1、2、4或8字节的
  “Integer” data of 1, 2, 4, or 8 bytes
  - 数据的值
    Data values
  - 地址 （无类型的指针）
    Addresses (untyped pointers)
- **浮点数：**4、8或10字节
  Floating point data of 4, 8, or 10 bytes
- **代码：**指令的字节序列编码
  Code: Byte sequences encoding series of instructions
- **没有聚合类型**，例如：数组或结构体
  No aggregate types such as arrays or structures
  - 这些在汇编语言中都都表现为在内存中连续分配的字节
    Just contiguously allocated bytes in memory

汇编语言的特征：操作
Assembly Characteristics: Data Types

- 对寄存器或存储器数据执行算术/逻辑运算
  Perform arithmetic function on register or memory data
- 在寄存器和存储器间传输数据
  Transfer data between memory and register
  - 将数据从存储器加载至寄存器
    Load data from memory into register
  - 将寄存器的数据存储至存储器
    Store register data into memory
- 转移控制
  Transfer control
  - 无条件跳转 至/从 过程
    Unconditional jumps to/from procedures
  - 条件分支
    Conditional branches

C语言，汇编语言和机器语言

C, assembly and machine code

目标码
Object Code

Code for `sumstore`

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- 14个字节
  Total of 14 bytes
- 不等长指令，1、3或5个字节
  Each instruction 1, 3, or 5 bytes
- 起始地址 0x0400595
  Starts at address `0x0400595`

- 汇编器
  Assembler
  - 将 .s 翻译为 .o
    Translates .s into .o
  - 对每条指令进行二进制编码
    Binary encoding of each instruction
  - 几乎是完整的可执行代码
    Nearly-complete image of executable code
  - 缺少了不同文件的链接信息
    Missing linkages between code in different files
- 链接器
  Linker
  - 实现了不同文件间的引用
    Resolves references between files
  - 与静态链接库进行了结合
    Combines with static run-time libraries
    - 例如：代码中的 malloc、printf
      E.g., code for malloc, printf
  - 某些库是需要动态链接的
    Some libraries are dynamically linked
    - 链接将出现在程序开始执行时
      Linking occurs when program begins execution

C语言，汇编语言和机器语言

C, assembly and machine code

举例：机器指令
Machine Instruction Example

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e:   48 89 03
```

- C代码
  C Code
  - 将 **t** 的值存储至 **dest** 指向的地址
    Store value **t** where designated by **dest**
- 汇编
  Assembly
  - 将8字节的数据（在x86-64中称为四字）移动至存储器
    Move 8-byte value (Quad words in x86-64 parlance) to memory
  - 操作数
    Operands
    - **t**: Register `%rax`
    - **dest**: Register `%rbx`
    - ***dest**: Memory `M[%rbx]`
- 目标码（机器指令）
  Object Code
  - 3字节指令
    3-byte instruction
  - 存储于地址 0x40059e
    Stored at address 0x40059e

C语言，汇编语言和机器语言

C, assembly and machine code

反汇编目标码
Disassembling Object Code

反汇编

Disassembled

```
0000000000400595 <sumstore>:
  400595:  53                push   %rbx
  400596:  48 89 d3          mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff    callq  400590 <plus>
  40059e:  48 89 03          mov    %rax,(%rbx)
  4005a1:  5b                pop    %rbx
  4005a2:  c3                retq
```

- 反汇编器
  Disassembler **`objdump -d sum`**
  - 将 **t** 的值存储至 **dest** 指向的地址
    Store value **t** where designated by **dest**
  - 探索目标码的一个十分有用的工具
    Useful tool for examining object code
  - 可以分析指令的编码序列
    Analyzes bit pattern of series of instructions
  - 根据目标码重新生成汇编代码
    Produces approximate rendition of assembly code
  - 可以对任何可执行程序文件和.o文件进行反汇编
    Can be run on either a.out (complete executable) or .o file

C语言，汇编语言和机器语言

C, assembly and machine code

另一种反汇编方法
Alternate Disassembly

目标码
Object

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

反汇编
Disassembled

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- 使用gdb调试器
  Within gdb Debugger

  `gdb sum`

  - 反汇编过程（函数）
    Disassemble procedure

  `disassemble sumstore`

  - 反汇编从sumstore开始的的14个字节目标码
    Examine the 14 bytes starting at sumstore

  `x/14xb sumstore`

C语言，汇编语言和机器语言

C, assembly and machine code

什么文件可以被反汇编?
What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:   file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55             push   %ebp
30001001:  8b ec          mov    %esp,%ebp
30001003:  6a ff          push   $0xffffffff
30001005:  68 90 10 00 30 push   $0x30001090
3000100a:  68 91 dc 4c 30 push   $0x304cdc91
```

- 任何可以解释为可执行代码的文件
  Anything that can be interpreted as executable code
- 反汇编程序分析字节并重构为汇编代码
  Disassembler examines bytes and reconstructs assembly source